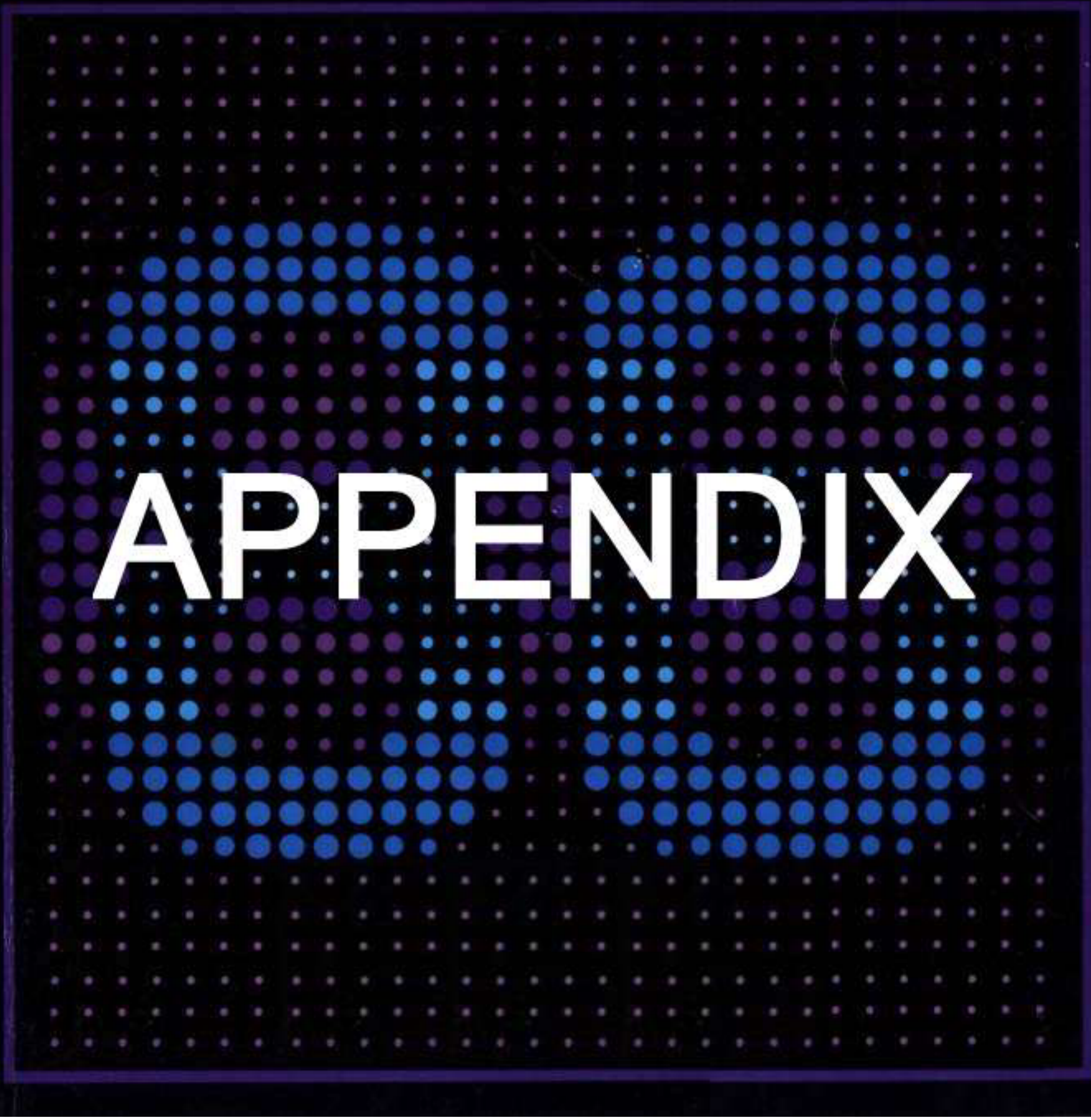
intel

*i*APX 86,88
User's **M**anual



APPENDIX

CEIBO
Development Tools of Choice

MCS-86	**8086 – 8088 – 80C86 – 80C88**

Ceibo In-Circuit Emulator Supporting MCS-86:	**DS-186** **http://ceibo.com/eng/products/ds186.shtml**

www.ceibo.com

APPENDIX A

Application Notes

APPENDIX B

Device Specifications

SUPPLEMENT

iAPX 86/20, 88/20 Numerics Supplement

Table of Contents

SUPPLEMENT

iAPX 86/20, 88/20 Numerics Supplement (Continued)

Tables

Figures

Appendix A
Application Notes

A

APPENDIX A
APPLICATION NOTES

This appendix contains Intel application notes pertinent to the 8086 family microprocessors. The following application notes, in the order listed, have been included within this appendix:

AP-67	8086 System Design
AP-61	Multitasking for the 8086
AP-50	Debugging Strategies and Considerations for 8089 Systems
AP-51	Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter
AP-59	Using the 8259A Programmable Interrupt Controller
AP-28A	Intel® Multibus™ Interfacing
AP-43	Using the iSBC-957™ Execution Vehicle for Executing 8086 Program Code

intel®

APPLICATION
NOTE

AP-67

September 1979

8086 System Design

George Alexy
Microcomputer Applications

8086 System Design

Contents

1. INTRODUCTION

The 8086 family, Intel's new series of microprocessors and system components, offers the designer an advanced system architecture which can be structured to satisfy a broad range of applications. The variety of speed, configuration and component selections available within the family enables optimization of a specific design to both cost and performance objectives. More important however, the 8086 family concept allows the designer to develop a family of systems providing multiple levels of enhancement within a single design and a growth path for future designs.

This application note is directed toward the implementation of the system hardware and will provide an introduction to a representative sample of the systems configurable with the 8086 CPU member of the family. Application techniques and timing analysis will be given to aid the designer in understanding the system requirements, advantages and limitations. Additional Intel publications the reader may wish to reference are the 8086 User's Manual (9800722A), 8086 Assembly Language Reference Guide (9800749A), AP-28A MULTIBUS™ Interfacing (98005876B), INTEL MULTIBUS™ SPECIFICATION (9800683), AP-45 Using the 8202 Dynamic RAM Controller (9800809A), AP-51 Designing 8086, 8088, 8089 Multiprocessor Systems with the 8289 Bus Arbiter and AP-59 Using the 8259A Programmable Interrupt Controller. References to other Intel publications will be made throughout this note.

2. 8086 OVERVIEW AND BASIC SYSTEM CONCEPTS

2A. 8086 Bus Cycle Definition

The 8086 is a true 16-bit microprocessor with 16-bit internal and external data paths, one megabyte of memory address space (2**20) and a separate 64K byte (2**16) I/O address space. The CPU communicates with its external environment via a twenty-bit time multiplexed address, status and data bus and a command bus. To transfer data or fetch instructions, the CPU executes a bus cycle (Fig. 2A1). The minimum bus cycle consists of four CPU clock cycles called T states. During the first T state (T1), the CPU asserts an address on the twenty-bit

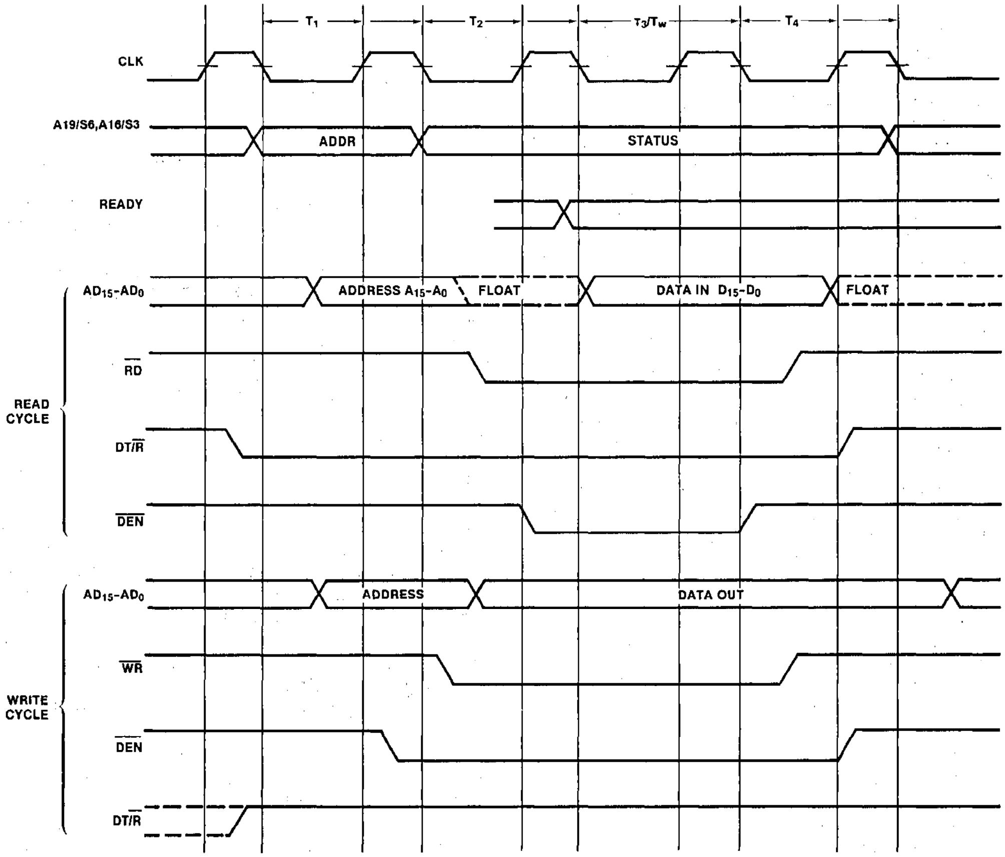


Figure 2A1. Basic 8086 Bus Cycle

multiplexed address/data/status bus. For the second T state (T2), the CPU removes the address from the bus and either three-states its outputs on the lower sixteen bus lines in preparation for a read cycle or asserts write data. Data bus transceivers are enabled in either T1 or T2 depending on the 8086 system configuration and the direction of the transfer (into or out of the CPU). Read, write or interrupt acknowledge commands are always enabled in T2. The maximum mode 8086 configuration (to be discussed later) also provides a write command enabled in T3 to guarantee data setup time prior to command activation.

During T2, the upper four multiplexed bus lines switch from address (A19-A16) to bus cycle status (S6,S5,S4,S3). The status information (Table 2A1) is available primarily for diagnostic monitoring. However, a decode of S3 and S4 could be used to select one of four banks of memory, one assigned to each segment register. This technique allows partitioning the memory by segment to expand the memory addressing beyond one megabyte. It also provides a degree of protection by preventing erroneous write operations to one segment from overlapping into another segment and destroying information in that segment.

The CPU continues to provide status information on the upper four bus lines during T3 and will either continue to assert write data or sample read data on the lower sixteen bus lines. If the selected memory or I/O device is not capable of transferring data at the maximum CPU transfer rate, the device must signal the CPU "not ready" and force the CPU to insert additional clock cycles (Wait states TW) after T3. The 'not ready' indication must be presented to the CPU by the start of T3. Bus activity during TW is the same as T3. When the selected device has had sufficient time to complete the transfer, it asserts "Ready" and allows the CPU to continue from the TW states. The CPU will latch the data on the bus during the last wait state or during T3 if no wait states are requested. The bus cycle is terminated in T4 (command lines are disabled and the selected external device deselects from the bus). The bus cycle appears to devices in the system as an asynchronous event consisting of an address to select the device followed by a read strobe or data and a write strobe. The selected device accepts bus data during a write cycle and drives the desired data onto the bus during a read cycle. On termination of the command, the device latches write data or disables its bus drivers. The only control the device has on the bus cycle is the insertion of wait cycles.

The 8086 CPU only executes a bus cycle when instructions or operands must be transferred to or from memory or I/O devices. When not executing a bus cycle, the bus interface executes idle cycles (TI). During the idle cycles, the CPU continues to drive status information from the previous bus cycle on the upper address lines. If the previous bus cycle was a write, the CPU continues to drive the write data onto the multiplexed bus until the start of the next bus cycle. If the CPU executes idle cycles following a read cycle, the CPU will not drive the lower 16 bus lines until the next bus cycle is required.

Since the CPU prefetches up to six bytes of the instruction stream for storage and execution from an internal instruction queue, the relationship of instruction fetch and associated operand transfers may be skewed in time and separated by additional instruction fetch bus cycles. In general, if an instruction is fetched into the 8086's internal instruction queue, several additional instructions may be fetched before the instruction is removed from the queue and executed. If the instruction being executed from the queue is a jump or other control transfer instruction, any instructions remaining in the queue are not executed and are discarded with no effect on the CPU's operation. The bus activity observed during execution of a specific instruction is dependent on the preceding instructions but is always deterministic within the specific sequence.

Table 2A1

S3	S4	
0	0	Alternate (relative to the ES segment)
1	0	Stack (relative to the SS segment)
0	1	Code/None (relative to the CS segment or a default of zero)
1	1	Data (relative to the DS segment)
S5 = IF (interrupt enable flag)		
S6 = 0 (indicates the 8086 is on the bus)		

2B. 8086 Address and Data Bus Concepts

Since the majority of system memories and peripherals require a stable address for the duration of the bus cycle, the address on the multiplexed address/data bus during T1 should be latched and the latched address used to select the desired peripheral or memory location. Since the 8086 has a 16-bit data bus, the multiplexed bus components of the 8085 family are not applicable to the 8086 (a device on address/data bus lines 8-15 will not be able to receive the byte selection address on lines 0-7). To demultiplex the bus (Fig. 2B1a), the 8086 system provides an Address Latch Enable signal (ALE) to capture the address in either the 8282 or 8283 8-bit bi-stable latches (Diag. 2B1). The latches are either inverting (8283) or non-inverting (8282) and have outputs driven by three-state buffers that supply 32 mA drive capability and can switch a 300 pF capacitive load in 22 ns (inverting) or 30 ns (non-inverting). They propagate the address through to the outputs while ALE is high and latch the address on the falling edge of ALE. This only delays address access and chip select decoding by the propagation delay of the latch. The outputs are enabled through the low active $\overline{OE}$ input. The demultiplexing of the multiplexed address/data bus (latchings of the address from the multiplexed bus), can be done locally at appropriate points in the system or at the CPU with a separate address bus distributing the address throughout the system (Fig. 2B1b). For optimum system performance and compatibility with multiprocessor and MULTIBUS™ configurations, the latter technique is strongly recommended over the first. The remainder of this note will assume the bus is demultiplexed at the CPU.

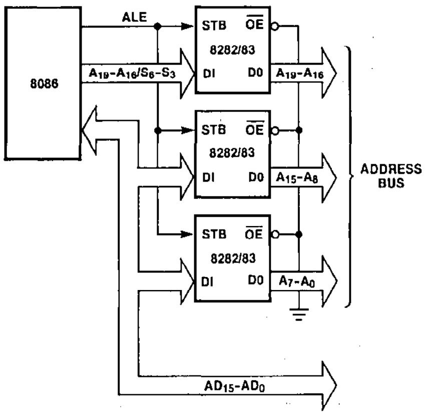
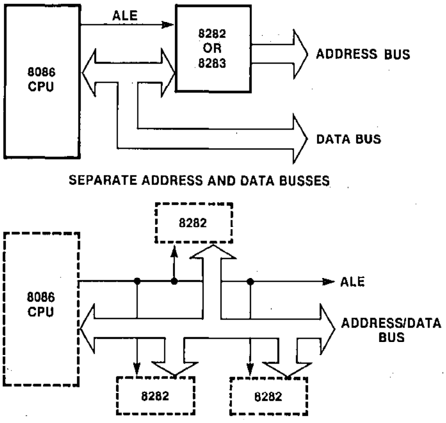


Figure 2B1a. Demultiplexing the 8086 Bus



SEPARATE ADDRESS AND DATA BUSSES

MULTIPLEXED BUS WITH LOCAL ADDRESS DEMULTIPLEXING

Figure 2B1b.

The programmer views the 8086 memory address space as a sequence of one million bytes in which any byte may contain an eight bit data element and any two consecutive bytes may contain a 16-bit data element. There is no constraint on byte or word addresses (boundaries). The address space is physically implemented on a sixteen bit data bus by dividing the address space into two banks of up to 512K bytes (Fig. 2B2). One bank is connected to the lower half of the sixteen-bit data bus (D7-0) and contains even addressed bytes (A0 = 0). The other bank is connected to the upper half of the data bus (D15-8) and contains odd addressed bytes (A0 = 1). A specific byte within each bank is selected by address lines A19-A1. To perform byte transfers to even addresses (Fig. 2B3a), the information is transferred over the lower half of the data bus (D7-0). A0 (active low) is used to enable the bank connected to the lower half of the data bus to participate in the transfer. Another signal provided by the 8086, Bus High Enable ($\overline{BHE}$), is used to disable the bank on the upper half of the data bus from participating in the transfer. This is necessary to prevent a write operation to the lower bank from destroying data in the upper bank. Since $\overline{BHE}$ is a multiplexed signal with timing identical to the A19-A16 address lines, it also should be latched with ALE to provide a stable signal during the bus cycle. During T2 through T4, the $\overline{BHE}$ output is multiplexed with status line S7 which is equal to $\overline{BHE}$. To perform byte transfers to odd addresses (Fig. 2B3b), the information is transferred over the upper half of the data bus (D15-D8) while $\overline{BHE}$ (active low) enables the upper bank and A0 disables the lower bank. Directing the data transfer to the appropriate half of the data bus and activation of $\overline{BHE}$ and A0 is performed by the 8086, transparent to the programmer. As an example, consider loading a byte of data into the CL register (lower half of the CX register) from an odd addressed memory location (referenced over the upper half of the 16-bit data bus). The data is transferred into the 8086 over the upper 8 bits of the data bus, automatically redirected to the lower half of the 8086 internal 16-bit data path and stored into the CL register. This capability also allows byte I/O transfers with the AL register to be directed to I/O devices connected to either the upper or lower half of the 16-bit data bus.

To access even addressed sixteen bit words (two consecutive bytes with the least significant byte at an even

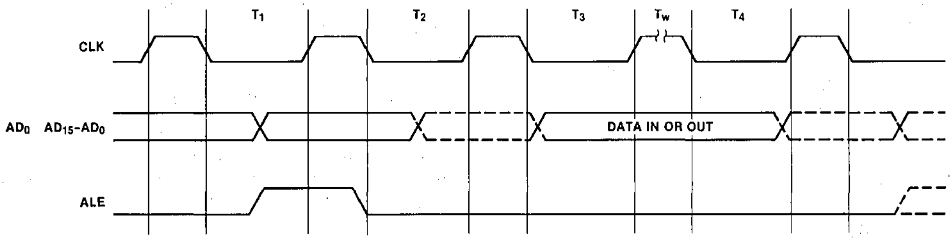


Diagram 2B1. ALE Timing

byte address), A19-A1 select the appropriate byte within each bank and A0 and $\overline{BHE}$ (active low) enable both banks simultaneously (Fig. 2B3c). To access an odd addressed 16-bit word (Fig. 2B3d), the least significant byte (addressed by A19-A1) is first transferred over the upper half of the bus (odd addressed byte, upper bank, $\overline{BHE}$ low active and A0 = 1). The most significant byte is accessed by incrementing the address (A19-A0) which allows A19-A1 to address the next physical word location (remember, A0 was equal to one which indicated a word referenced from an odd byte boundary). A second bus cycle is then executed to perform the transfer of the most significant byte with the lower bank (A0 is now active low and $\overline{BHE}$ is high). The sequence is automatically executed by the 8086 whenever a word transfer is executed to an odd address. Directing the upper and lower bytes of the 8086's internal sixteen-bit registers to the appropriate halves of the data bus is also performed automatically by the 8086 and is transparent to the programmer.

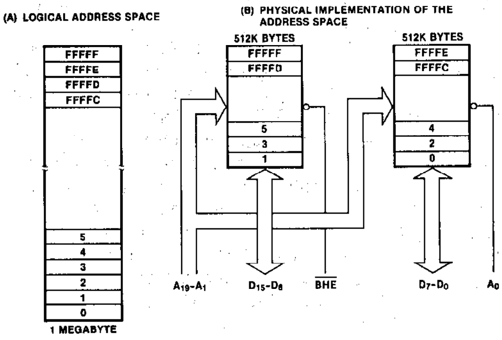


Figure 2B2. 8086 Memory

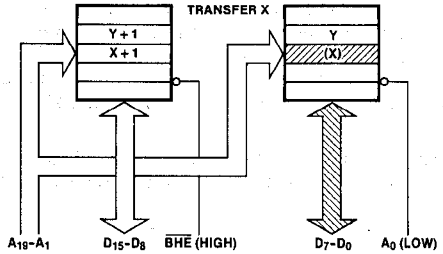


Figure 2B3a. Even Addressed Byte Transfer

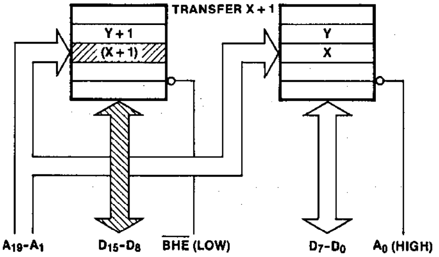


Figure 2B3b. Odd Addressed Byte Transfer

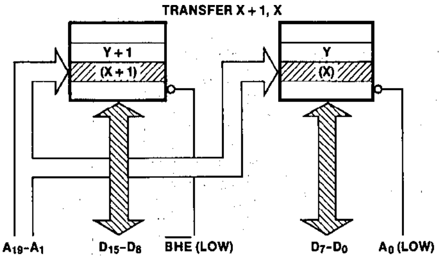


Figure 2B3c. Even Addressed Word Transfer

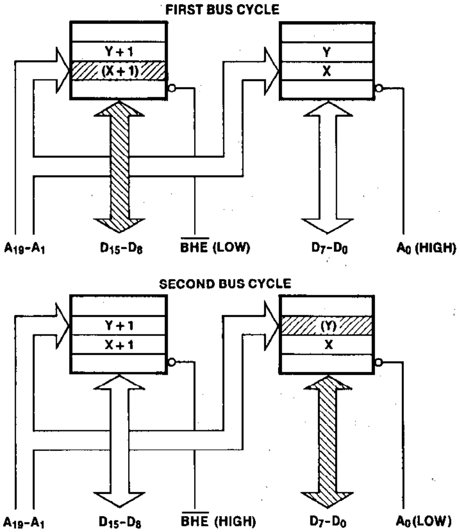


Figure 2B3d. Odd Addressed Word Transfer

During a byte read, the CPU floats the entire sixteen-bit data bus even though data is only expected on the upper or lower half of the data bus. As will be demonstrated later, this action simplifies the chip select decoding requirements for read only devices (ROM, EPROM). During a byte write operation, the 8086 will drive the entire sixteen-bit data bus. The information on the half of the data bus not transferring data is indeterminate. These concepts also apply to the I/O address space. Specific examples of I/O and memory interfacing are considered in the corresponding sections.

2C. System Data Bus Concepts

When referring to the system data bus, two implementation alternatives must be considered; (a) the multiplexed address/data bus (Fig. 2C1a) and a data bus buffered from the multiplexed bus by transceivers (Fig. 2C1b).

If memory or I/O devices are connected directly to the multiplexed bus, the designer must guarantee the devices do not corrupt the address on the bus during T1.

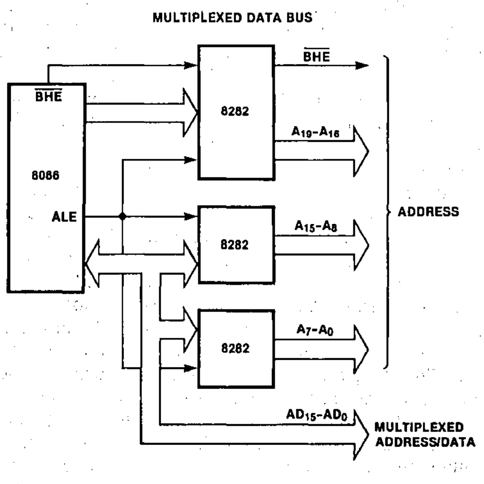


Figure 2C1a. Multiplexed Data Bus

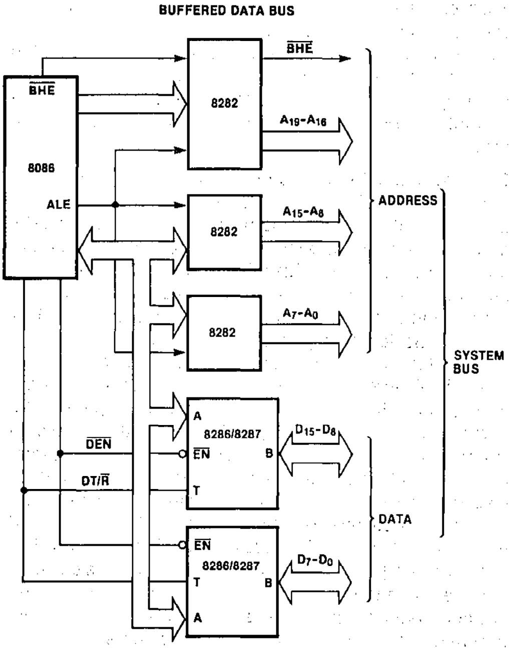


Figure 2C1b. Buffered Data Bus

To avoid this, device output drivers should not be enabled by the device chip select, but should have an output enable controlled by the system read signal (Fig. 2C2). The 8086 timing guarantees that read is not valid until after the address is latched by ALE (Diag. 2C1). All Intel peripherals, EPROM products and RAM's for microprocessors provide output enable or read inputs to allow connection to the multiplexed bus.

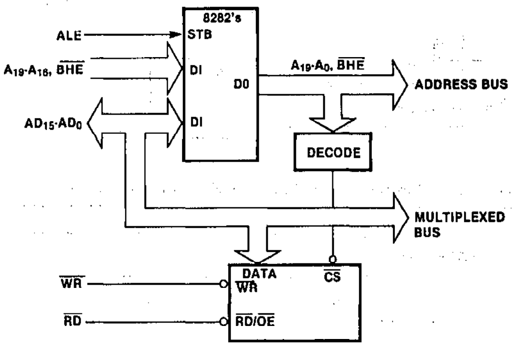


Figure 2C2. Devices with Output Enables on the Multiplexed Bus

Several techniques are available for interfacing devices without output enables to the multiplexed bus but each introduces other restrictions or limitations. Consider Figure 2C3 which has chip select gated with read and write. Two problems exist with this technique. First, the chip select access time is reduced to the read access time, and may require a faster device if maximum system performance (no wait states) is to be achieved (Diag. 2C2). Second, the designer must verify that chip select to write setup and hold times for the device are not violated (Diag. 2C3). Alternate techniques can be extracted from the bus interfacing techniques given later in this section but are subject to the associated restrictions. In general, the best solution is obtained with devices having output enables.

A subsequent limitation on the multiplexed bus is the 8086's drive capability of 2.0 mA and capacitive loading of 100 pF to guarantee the specified A.C. characteristics. Assuming capacitive loads of 20 pF per I/O device, 12 pF per address latch and 5-12 pF per memory device, a system mix of three peripherals and two to four memory devices (per bus line) are close to the loading limit.

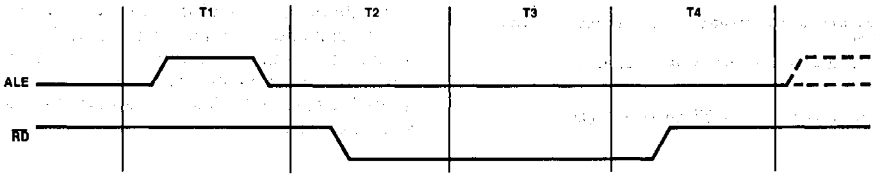


Diagram 2C1. Relationship of ALE to READ

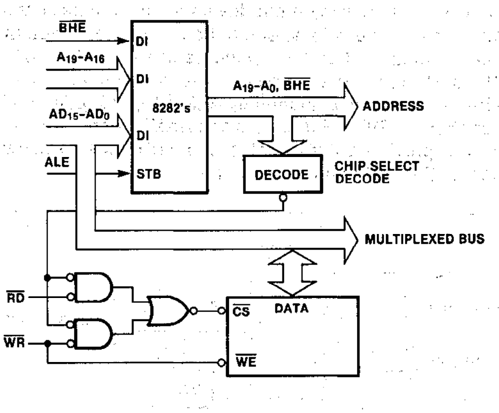
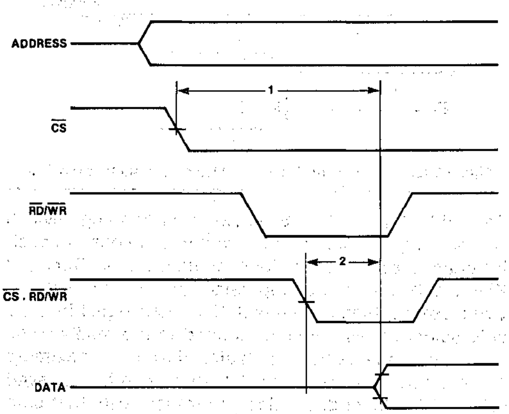


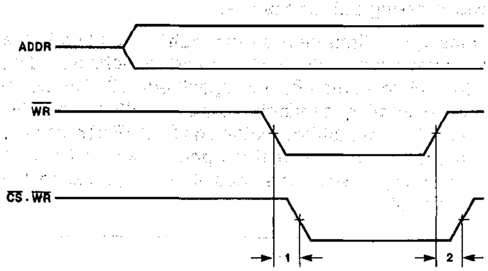
Figure 2C3. Devices without Output Enables on the Multiplexed Bus



1 ACCESS TIME FOR CS GENERATED FROM ADDRESS DECODE.

2 ACCESS TIME IF CS IS GATED WITH RD/WR.

Diagram 2C2. Access Time: CS Gated with $\overline{RD}/\overline{WR}$



1 CS IS NOT VALID PRIOR TO WRITE AND BECOMES ACTIVE ONE OR TWO GATE DELAYS LATER.

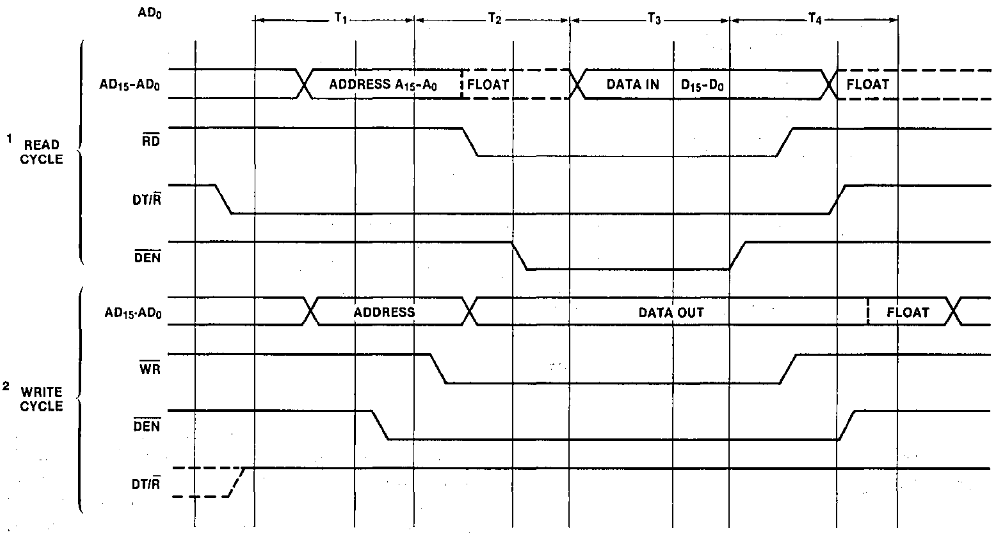
2 CS REMAINS VALID AFTER WRITE ONE OR TWO GATE DELAYS.

Diagram 2C3. CS to $\overline{WR}$ Set-Up and Hold

To satisfy the capacitive loading and drive requirements of larger systems, the data bus must be buffered. The 8286 non-inverting and 8287 inverting octal transceivers are offered as part of the 8086 family to satisfy this requirement. They have three-state output buffers that drive 32 mA on the bus interface and 10 mA on the CPU interface and can switch capacitive loads of 300 pF at the bus interface and 100 pF on the CPU interface in 22 ns (8287) or 30 ns (8286). To enable and control the direction of the transceivers, the 8086 system provides Data ENable (DEN) and Data Transmit/Receive (DT/$\overline{R}$) signals (Fig. 2C1b). These signals provide the appropriate timing to guarantee isolation of the multiplexed bus from the system during T1 and elimination of bus contention with the CPU during read and write (Diag. 2C4). Although the memory and peripheral devices are isolated from the CPU (Fig. 2C4), bus contention may still exist in the system if the devices do not have an output enable control other than chip select. As an example, bus contention will exist during transition from one chip select to another (the newly selected device begins driving the bus before the previous device has disabled its drivers). Another, more severe case exists during a write cycle. From chip select to write active, a device whose outputs are controlled only by chip select, will drive the bus simultaneously with write data being driven through the transceivers by the CPU (Diag. 2C5). The same technique given for circumventing these problems on the multiplexed bus can be applied here with the same limitations.

One last extension to the bus implementation is a second level of buffering to reduce the total load seen by devices on the system bus (Fig. 2C5). This is typically done for multiboard systems and isolation of memory arrays. The concerns with this configuration are the additional delay for access and more important, control of the second transceiver in relationship to the system bus and the device being interfaced to the system bus. Several techniques for controlling the transceiver are given in Figure 2C6. This first technique (Fig. 2C6a) simply distributes DEN and DT/$\overline{R}$ throughout the system. DT/$\overline{R}$ is inverted to provide proper direction control for the second level transceivers. The second example (Fig. 2C6b) provides control for devices with output enables. $\overline{RD}$ is used to normally direct data from the system bus to the peripheral. The buffer is selected whenever a device on the local bus is chip selected. Bus contention is possible on the device's local bus during a read as the read simultaneously enables the device output and changes the transceiver direction. The contention may also occur as the read is terminated.

For devices without output enables, the same technique can be applied (Fig. 2C6c) if the chip select to the device is conditioned by read or write. Controlling the chip select with read/write prevents the device from driving against the transceiver prior to the command being received. The limitations with this technique are access limited to read/write time and limited CS to write setup and hold times.



1 DEN IS ENABLED AFTER THE 8086 HAS FLOATED THE MULTIPLEXED BUS

2 DEN ENABLES THE TRANSCEIVERS EARLY IN THE CYCLE, BUT DT/R GUARANTEES THE TRANSCEIVERS ARE IN TRANSMIT RATHER THAN RECEIVE MODE AND WILL NOT DRIVE AGAINST THE CPU.

Diagram 2C4. Bus Transceiver Control

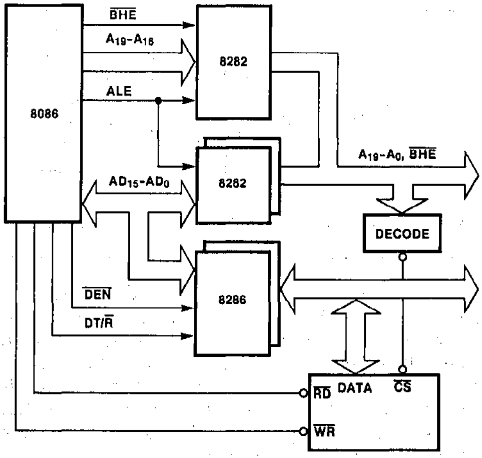


Figure 2C4. Devices with Output Enables on the System Bus

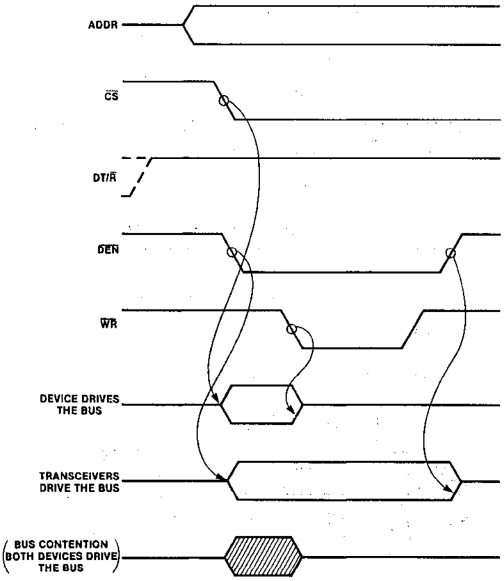

ADDR
CS
DT/R
DEN
WR
DEVICE DRIVES THE BUS
TRANSCEIVERS DRIVE THE BUS
BUS CONTENTION BOTH DEVICES DRIVE THE BUS

Diagram 2C5.

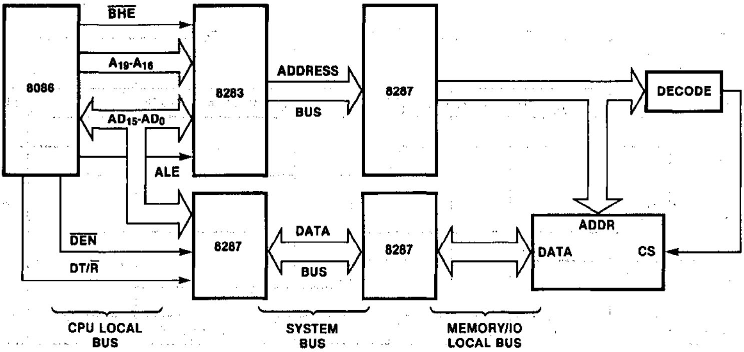


Figure 2C5. Fully Buffered System

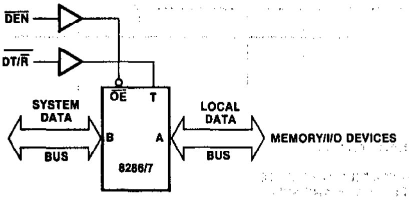


Figure 2C6a. Controlling System Transceivers with DEN and DT/$\overline{\text{R}}$

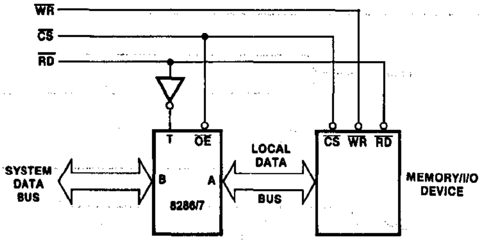


Figure 2C6b. Buffering Devices with $\overline{\text{OE}}/\overline{\text{RD}}$

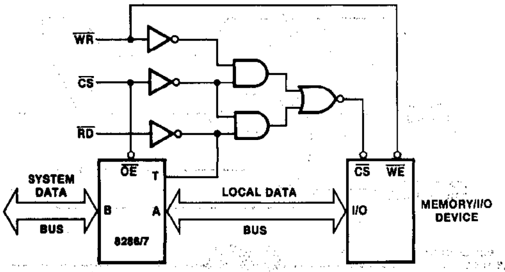


Figure 2C6c. Buffering Devices without $\overline{\text{OE}}/\overline{\text{RD}}$ and with Common or Separate Input/Output

An alternate technique applicable to devices with and without output enables is shown in Figure 2C6d. $\overline{\text{RD}}$ again controls the direction of the transceiver but it is not enabled until a command and chip select are active. The possibility for bus contention still exists but is reduced to variations in output enable vs. direction change time for the transceiver. Full access time from chip select is now available, but data will not be valid prior to write and will only be held valid after write by the delay to disable the transceiver.

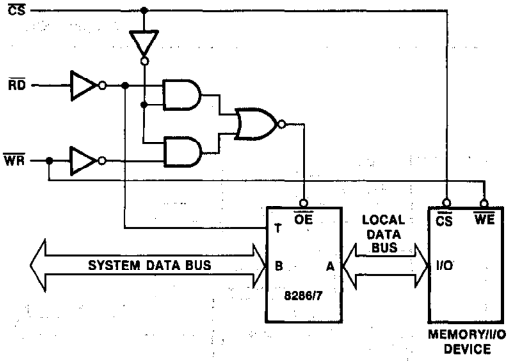


Figure 2C6d. Buffering Devices without $\overline{\text{OE}}/\overline{\text{RD}}$ and with Common or Separate Input/Output

One last technique is given for devices with separate inputs and outputs (Fig. 2C6e). Separate bus receivers and drivers are provided rather than a single transceiver. The receiver is always enabled while the bus driver is controlled by $\overline{\text{RD}}$ and chip select. The only possibility for bus contention in this system occurs as multiple devices on each line of the local read bus are enabled and disabled during chip selection changes.

Throughout this note, the multiplexed bus will be considered the local CPU bus and the demultiplexed address and buffered data bus will be the system bus. For additional information on bus contention and the system problems associated with it, refer to Appendix 1.

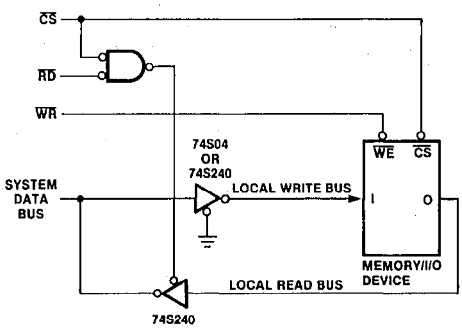


Figure 2C6e. Buffering Devices without $\overline{\text{OE/RD}}$ and with Separate Input/Output

2D. Multiprocessor Environment

The 8086 architecture supports multiprocessor systems based on the concept of a shared system bus (Fig. 2D1). All CPU's in the system communicate with each other and share resources via the system bus. The bus may be either the Intel Multibus™ system bus or an extension of the system bus defined in the previous section. The major addition required to the demultiplexed system bus is arbitration logic to control access to the system bus. As each CPU asynchronously requests access to the shared bus, the arbitration logic resolves priorities and grants bus access to the highest priority CPU. Having gained access to the bus, the CPU completes its transfer and will either relinquish the bus or wait to be forced to relinquish the bus. For a discussion on Multibus™ arbitration techniques, refer to AP-28A, Intel Multibus™ Interfacing.

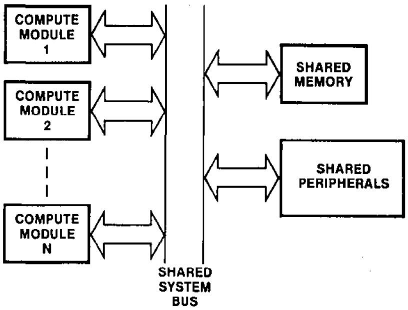


Figure 2D1. 8086 Family Multiprocessor System

To support a multimaster interface to the Multibus system bus for the 8086 family, the 8289 bus arbiter is included as part of the family. The 8289 is compatible with the 8086's local bus and in conjunction with the 8288 bus controller, implements the Multibus protocol for bus arbitration. The 8289 provides a variety of arbitration and prioritization techniques to allow optimization of bus availability, throughput and utilization of shared resources. Additional features (implemented through strapping options) extend the configuration options beyond a pure CPU interface to the multimaster system bus for access to shared resources to include concurrent support of a local CPU bus for private resources. For specific configurations and additional information on the 8289, refer to application note AP-51.

3. 8086 SYSTEM DETAILS

3A. Operating Modes

Possibly the most unique feature of the 8086 is the ability to select the base machine configuration most suited to the application. The MN/$\overline{\text{MX}}$ input to the 8086 is a strapping option which allows the designer to select between two functional definitions of a subset of the 8086 outputs.

MINIMUM MODE

The minimum mode 8086 (Fig. 3A1) is optimized for small to medium (one or two boards), single CPU systems. Its system architecture is directed at satisfying the requirements of the lower to middle segment of high performance 16-bit applications. The CPU maintains the full megabyte memory space, 64K byte I/O space and 16-bit data path. The CPU directly provides all bus control (DT/$\overline{\text{R}}$, $\overline{\text{DEN}}$, ALE, M/$\overline{\text{IO}}$), commands ($\overline{\text{RD}}$,$\overline{\text{WR}}$,$\overline{\text{INTA}}$) and a simple CPU preemption mechanism (HOLD, HLDA) compatible with existing DMA controllers.

MAXIMUM MODE

The maximum mode (Fig. 3A2) extends the system architecture to support multiprocessor configurations, and local instruction set extension processors (coprocessors). Through addition of the 8288 bipolar bus controller, the 8086 outputs assigned to bus control and commands in the minimum mode are redefined to allow these extensions and enhance general system performance. Specifically, (1) two prioritized levels of processor preemption ($\overline{\text{RQ}}$/$\overline{\text{GT0}}$, $\overline{\text{RQ}}$/$\overline{\text{GT1}}$) allow multiple processors to reside on the 8086's local bus and share its interface to the system bus, (2) Queue status (QS0,QS1) is available to allow external devices like ICE™-86 or special instruction set extension co-processors to track the CPU instruction execution, (3) access control to shared resources in multiprocessor systems is supported by a hardware bus lock mechanism and (4) system command and configuration options are expanded via ancillary devices like the 8288 bus controller and 8289 bus arbiter.

The queue status indicates what information is being removed from the internal queue and when the queue is being reset due to a transfer of control (Table 3A1). By monitoring the $\overline{\text{S0}}$,$\overline{\text{S1}}$,$\overline{\text{S2}}$ status lines for instructions entering the 8086 (1,0,0 indicates code access while A0 and $\overline{\text{BHE}}$ indicate word or byte) and QS0, QS1 for instructions leaving the 8086's internal queue, it is possible to track the instruction execution. Since instructions are executed from the 8086's internal queue, the queue status is presented each CPU clock cycle and is not related to the bus cycle activity. This mechanism (1) allows a co-processor to detect execution of an

ESCAPE instruction which directs the co-processor to perform a specific task and (2) allows ICE-86 to trap execution of a specific memory location. An example of a circuit used by ICE is given in Figure 3A3. The first up down counter tracks the depth of the queue while the second captures the queue depth on a match. The second counter decrements on further fetches from the queue until the queue is flushed or the count goes to zero indicating execution of the match address. The first counter decrements on fetch from the queue (QS0 = 1) and increments on code fetches into the queue. Note that a normal code fetch will transfer two bytes into the queue so two clock increments are given to the counter (T201 and T301) unless a single byte is loaded over the upper half of the bus (A0-P is high). Since the execution unit (EU) is not synchronized to the bus interface unit (BIU), a fetch from the queue can occur simultaneously with a transfer into the queue. The exclusive-or gate driving the ENP input of the first counter allows these simultaneous operations to cancel each other and not modify the queue depth.

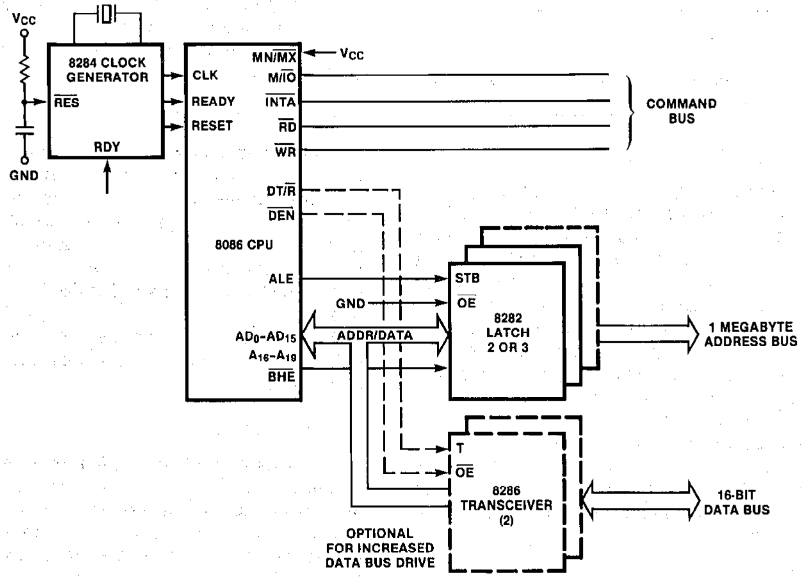


Figure 3A1. Minimum Mode 8086

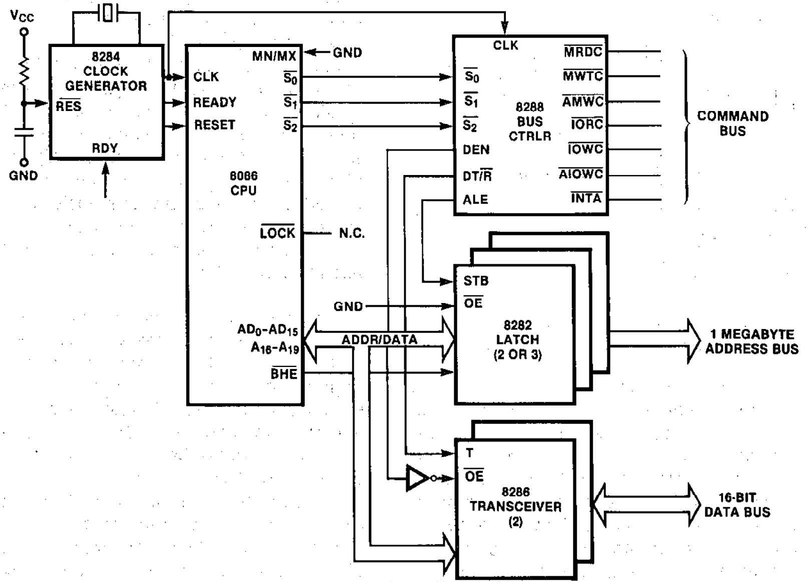


Figure 3A2. Maximum Mode 8086

TABLE 3A1. QUEUE STATUS

QS_1	QS_0	
0 (LOW)	0	No Operation
0	1	First Byte of Op Code from Queue
1 (HIGH)	0	Empty the Queue
1	1	Subsequent Byte from Queue

The queue status is valid during the CLK cycle after which the queue operation is performed.

To address the problem of controlling access to shared resources, the maximum mode 8086 provides a hardware $\overline{\text{LOCK}}$ output. The $\overline{\text{LOCK}}$ output is activated through the instruction stream by execution of the LOCK prefix instruction. The $\overline{\text{LOCK}}$ output goes active in the first CPU clock cycle following execution of the prefix and remains active until the clock following the completion of the instruction following the LOCK prefix. To provide bus access control in multiprocessor systems, the $\overline{\text{LOCK}}$ signal should be incorporated into the system bus arbitration logic resident to the CPU.

During normal multiprocessor system operation, priority of the shared system bus is determined by the arbitration circuitry on a cycle by cycle basis. As each CPU requires a transfer over the system bus, it requests access to the bus via its resident bus arbitration logic. When the CPU gains priority (determined by the system bus arbitration scheme and any associated logic), it takes control of the bus, performs its bus cycle and either maintains bus control, voluntarily releases the bus or is forced off the bus by the loss of priority. The lock mechanism prevents the CPU from losing bus control (either voluntarily or by force) and guarantees a CPU the ability to execute multiple bus cycles (during execution of the locked instruction) without intervention and possible corruption of the data by another CPU. A classic use of the mechanism is the 'TEST and SET semaphore' during which a CPU must read from a shared memory location and return data to the location without allowing another CPU to reference the same location between the TEST operation (read) and the SET operation (write). In the 8086 this is accomplished with a locked exchange instruction.

```
LOCK XCHG reg, MEMORY ; reg is any register
                      ;MEMORY is the address of the
                      ;semaphore
```

The activity of the $\overline{\text{LOCK}}$ output is shown in Diagram 3A1. Another interesting use of the $\overline{\text{LOCK}}$ for multiprocessor systems is a locked block move which allows high speed message transfer from one CPU's message buffer to another.

During the locked instruction, a request for processor preemption ($\overline{\text{RQ}}/\overline{\text{GT}}$) is recorded but not acknowledged until completion of the locked instruction. The $\overline{\text{LOCK}}$ has no direct affect on interrupts. As an example, a locked HALT instruction will cause HOLD (or $\overline{\text{RQ}}/\overline{\text{GT}}$) requests to be ignored but will allow the CPU to exit the HALT state on an interrupt. In general, prefix bytes are considered extensions of the instructions they precede. Therefore, interrupts that occur during execution of a prefix are not acknowledged (assuming interrupts are enabled) until completion of the instruction following the prefixes (except for instructions which allow servicing interrupts during their execution, i.e., HALT, WAIT and repeated string primitives). Note that multiple prefix bytes may precede an instruction. As another example, consider a 'string primitive' preceded by the repetition

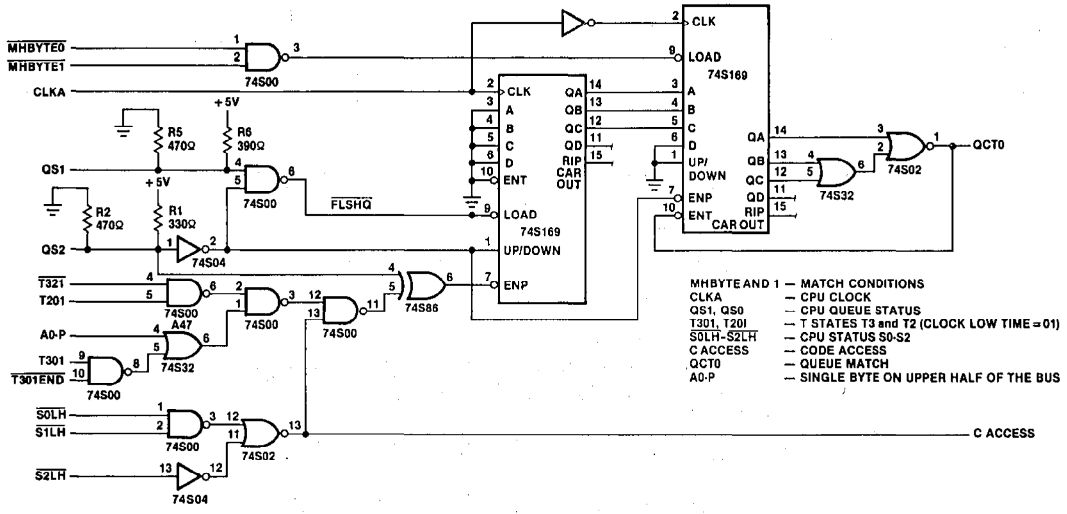


Figure 3A3. Example Circuit to Track the 8086 Queue

prefix (REP) which is interruptible after each execution of the string primitive. This holds even if the REP prefix is combined with the LOCK prefix and prevents interrupts from being locked out during a block move or other repeated string operation. As long as the operation is not interrupted, $\overline{\text{LOCK}}$ remains active. Further information on the operation of an interrupted string operation with multiple prefixes is presented in the section dealing with the 8086 interrupt structure.

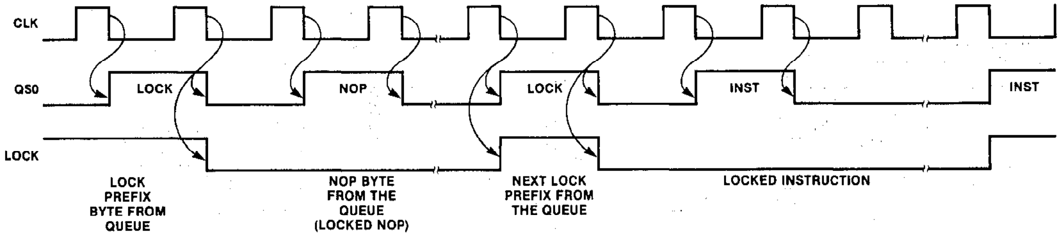
Three additional status lines ($\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$) are defined to provide communications with the 8288 and 8289. The status lines tell the 8288 when to initiate a bus cycle, what type of command to issue and when to terminate the bus cycle. The 8288 samples the status lines at the beginning of each CPU clock (CLK). To initiate a bus cycle, the CPU drives the status lines from the passive state ($\overline{\text{S0}}$, $\overline{\text{S1}}$, $\overline{\text{S2}}$ = 1) to one of seven possible command codes (Table 3A2). This occurs on the rising edge of the clock during T4 of the previous bus cycle or a TI (idle cycle, no current bus activity). The 8288 detects the status change by sampling the status lines on the high to low transition of each clock cycle. The 8288 starts a bus cycle by generating ALE and appropriate buffer direction control in the clock cycle immediately following detection of the status change (T1). The bus transceivers and the selected command are enabled in the next clock cycle (T2) (or T3 for normal write commands). When the status returns to the passive state, the 8288 will terminate the command as shown in Diagram 3A2. Since the CPU will not return the status to the passive state until the 'ready' indication is received, the 8288 will maintain active command and bus control for any number of wait cycles. The status lines may also be used by other processors on the 8086's local bus to monitor bus activity and control the 8288 if they gain control of the local bus.

TABLE 3A2. STATUS LINE DECODES

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O Port
0	1	0	Write I/O Port
0	1	1	Halt
1 (HIGH)	0	0	Code Access
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive

The 8288 provides the bus control (DEN, DT/$\overline{\text{R}}$, ALE) and commands ($\overline{\text{INTA}}$, $\overline{\text{MRDC}}$, $\overline{\text{IORC}}$, $\overline{\text{MWTC}}$, $\overline{\text{AMWC}}$, $\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$) removed from the CPU. The command structure has separate read and write commands for memory and I/O to provide compatibility with the Multibus command structure.

The advanced write commands are enabled one clock period earlier than the normal write to accommodate the wider write pulse widths often required by peripherals and static RAMs. The normal write provides data setup prior to write to accommodate dynamic RAM memories and I/O devices which strobe data on the leading edge of write. The advanced write commands do not guarantee that data is valid prior to the leading edge of the command. The DEN signal in the maximum mode is inverted from the minimum mode to extend transceiver control by allowing logical conjunction of DEN with other signals. While not appearing to be a significant benefit in the basic maximum mode configuration, introduction of interrupt control and various system configurations will demonstrate the usefulness of qualifying DEN. Diagram 3A3 compares the timing of the minimum and maximum mode bus transfer commands. Although the


CLK
QS0
LOCK
LOCK
NOP
LOCK
INST
INST
LOCK
LOCK PREFIX BYTE FROM QUEUE
NOP BYTE FROM THE QUEUE (LOCKED NOP)
NEXT LOCK PREFIX FROM THE QUEUE
LOCKED INSTRUCTION

1 QUEUE STATUS INDICATES FIRST BYTE OF OPCODE FROM THE QUEUE.

2 THE $\overline{\text{LOCK}}$ OUTPUT WILL GO INACTIVE BETWEEN SEPARATE LOCKED INSTRUCTIONS.

3 TWO CLOCKS ARE REQUIRED FOR DECODE OF THE LOCK PREFIX AND ACTIVATION OF THE $\overline{\text{LOCK}}$ SIGNAL.

4 SINCE QUEUE STATUS REFLECTS THE QUEUE OPERATION IN THE PREVIOUS CLOCK CYCLE, THE $\overline{\text{LOCK}}$ OUTPUT ACTUALLY GOES ACTIVE COINCIDENT WITH THE START OF THE NEXT INSTRUCTION AND REMAINS ACTIVE FOR ONE CLOCK CYCLE FOLLOWING THE INSTRUCTION.

5 IF THE INSTRUCTION FOLLOWING THE LOCK PREFIX IS NOT IN THE QUEUE, THE $\overline{\text{LOCK}}$ OUTPUT STILL GOES ACTIVE AS SHOWN WHILE THE INSTRUCTION IS BEING FETCHED.

6 THE BIU WILL STILL PERFORM INSTRUCTION FETCH CYCLES DURING EXECUTION OF A LOCKED INSTRUCTION. THE $\overline{\text{LOCK}}$ MERELY LOCKS THE BUS TO THIS CPU FOR WHATEVER BUS CYCLES THE CPU PERFORMS DURING THE LOCKED INSTRUCTION.

Diagram 3A1. 8086 Lock Activity

maximum mode configuration is designed for multiprocessor environments, large single CPU designs (either Multibus systems or greater than two PC boards) should also use the maximum mode. Since the 8288 is a bipolar dedicated controller device, its output drive for the commands (32 mA) and tolerances on AC characteristics (timing parameters and worse case delays) provide better large system performance than the minimum mode 8086.

In addition to assuming the functions removed from the CPU, the 8288 provides additional strapping options and controls to support multiprocessor configurations and peripheral devices on the CPU local bus. These capabilities allow assigning resources (memory or I/O) as shared (available on the Multibus system bus) or private (accessible only by this CPU) to reduce contention for access to the Multibus system bus and improve multi-CPU system performance. Specific configuration possibilities are discussed in AP-51.

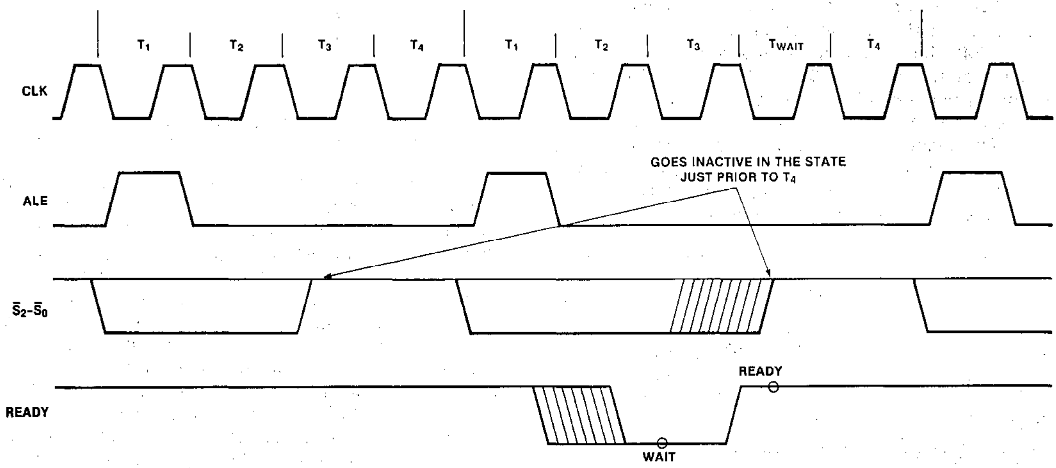


Diagram 3A2. Status Line Activation and Termination

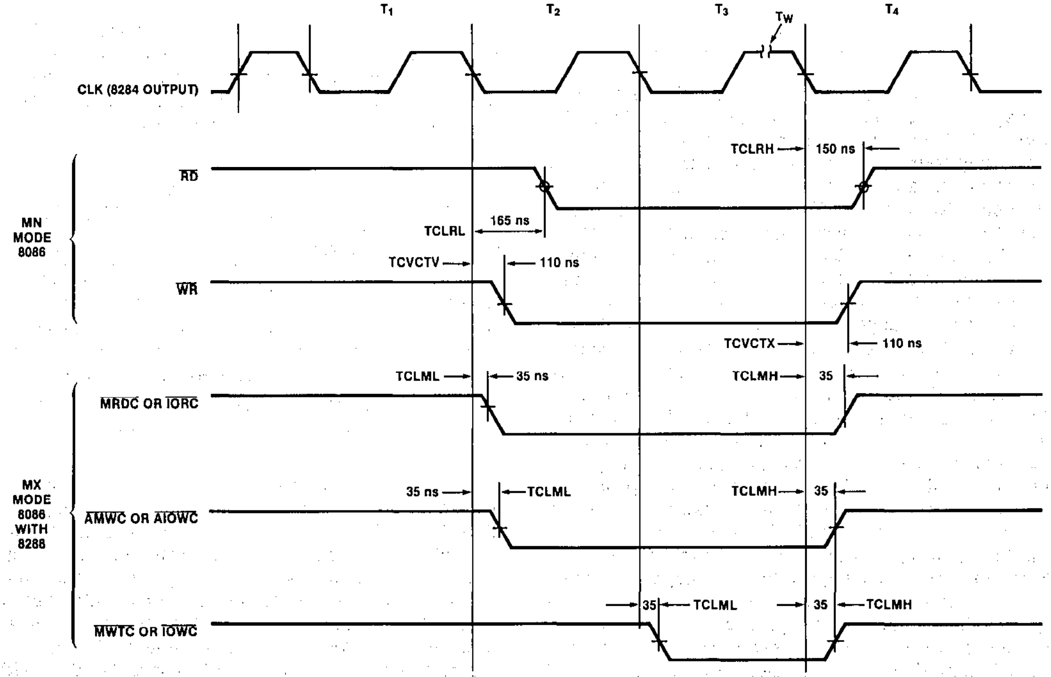


Diagram 3A3. 8086 Minimum and Maximum Mode Command Timing

3B. Clock Generation

The 8086 requires a clock signal with fast rise and fall times (10 ns max) between low and high voltages of −0.5 to +0.6 low and 3.9 to VCC+1.0 high. The maximum clock frequency of the 8086 is 5 MHz and 8 MHz for the 8086-2. Since the design of the 8086 incorporates dynamic cells, a minimum frequency of 2 MHz is required to retain the state of the machine. Due to the minimum frequency requirement, single stepping or cycling of the CPU may not be accomplished by disabling the clock. The timing and voltage requirements of the CPU clock are shown in Figure 3B1. In general, for frequencies below the maximum, the CPU clock need not satisfy the frequency dependent pulse width limitations stated in the 8086 data sheet. The values specified only reflect the minimum values which must be satisfied and are stated in terms of the maximum clock frequency. As the clock frequency approaches the maximum frequency of the CPU, the clock must conform to a 33% duty cycle to satisfy the CPU minimum clock low and high time specifications.

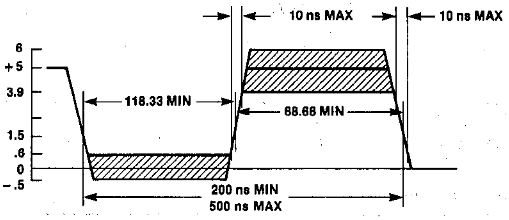


Figure 3B1. 8086 Clock

An optimum 33% duty cycle clock with the required voltage levels and transition times can be obtained with the 8284 clock generator (Fig. 3B2). Either an external frequency source or a series resonant crystal may drive the 8284. The selected source must oscillate at 3X the desired CPU frequency. To select the crystal inputs of the 8284 as the frequency source for clock generation, the F/C̄ input to the 8284 must be strapped to ground. The strapping option allows selecting either the crystal or the external frequency input as the source for clock generation. Although the 8284 provides an input for a tank circuit to accommodate overtone mode crystals, fundamental mode crystals are recommended for more accurate and stable frequency generation. When selecting a crystal for use with the 8284, the series resistance should be as low as possible. Since other circuit components will tend to shift the operating frequency from resonance, the operating impedance will typically be higher than the specified series resistance. If the attenuation of the oscillator's feedback circuit reduces the loop gain to less than one, the oscillator will fail. Since the oscillator delays in the 8284 appear as inductive elements to the crystal, causing it to run at a frequency below that of the pure series resonance, a capacitor should be placed in series with the crystal and the X2 input of the 8284. This capacitor serves to cancel this inductive element. The value of the capacitor (CL) must not cause the impedance of the feedback circuit to reduce the loop gain below one. The impedance of the capacitor is a function of the operating frequency and can be determined from the following equation:

$$XCL = 1/2\pi{*}F{*}CL$$

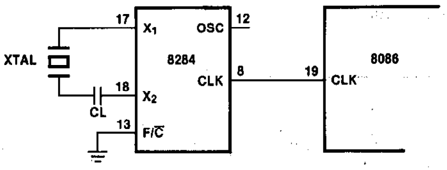


Figure 3B2. 8284 Clock Generator

It is recommended that the crystal series resistance plus XCL be kept less than 1K ohms. This capacitor also serves to debias the crystal and prevent a DC voltage bias from straining and perhaps damaging the crystalline structure. As the crystal frequency increases, the amount of capacitance should be decreased. For example, a 12 MHz crystal may require CL ~ 24 pF while 22 MHz may require CL ~ 8 pF. If very close correlation with the pure series resonance is not necessary, a nominal CL value of 12-15 pF may be used with a 15 MHz crystal (5 MHz 8086 operation). Board layout and component variances will affect the actual amount of inductance and therefore the series capacitance required to cancel it out (this is especially true for wire-wrapped layouts).

Two of the many vendors which supply crystals for Intel microprocessors are listed in Table 3B1 along with a list of crystal part numbers for various frequencies which may be of interest. For additional information on specifying crystals for Intel components refer to application note AP-35.

TABLE 3B1. CRYSTAL VENDORS

f	Parallel/ Series	Crystek[1] Corp.	CTS Knight,[2] Inc.
15.0 MHz	S	CY15A	MP150
18.432	S	CY19B*	MP184*
24.0 MHz	S	CY24A	MP240

*Intel also supplies a crystal numbered 8801 for this application.

Notes: 1. Address: 1000 Crystal Drive, Fort Meyers, Florida 33901
2. Address: 400 Reimann Ave., Sandwich, Illinois

If a high accuracy frequency source, externally variable frequency source or a common source for driving multiple 8284's is desired, the External Frequency Input (EFI) of the 8284 can be selected by strapping the F/C̄ input to 5 volts through ~1K ohms (Fig. 3B3). The external frequency source should be TTL compatible, have a 50% duty cycle and oscillate at three times the desired CPU operating frequency. The maximum EFI frequency the 8284 can accept is slightly above 24 MHz with minimum clock low and high times of 13 ns. Although no minimum EFI frequency is specified, it should not violate the CPU minimum clock rate. If a common frequency source is used to drive multiple 8284's distributed throughout the system, each 8284 should be driven by its own line from the source. To minimize noise in the system, each line should be a twisted pair driven by a buffer like the 74LS04 with the ground of the twisted pair connecting the grounds of the source and receiver. To minimize clock skew, the lines to all 8284's should be of equal length. A simple technique for generating a master frequency source for additional 8284's is shown in Figure 3B4. One 8284 with a crystal is used to generate the desired frequency. The oscillator output of the 8284 (OSC) equals the crystal frequency and is used to drive the external frequency to all other 8284's in the system.

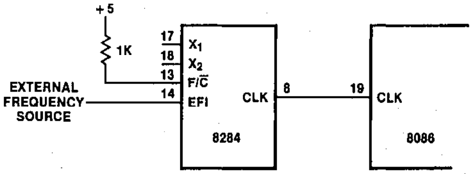


Figure 3B3. 8284 with External Frequency Source

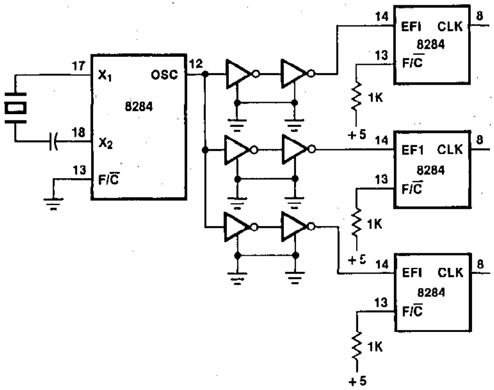


Figure 3B4. External Frequency for Multiple 8284s

The oscillator output is inverted from the oscillator signal used to drive the CPU clock generator circuit. Therefore, the oscillator output of one 8284 should not drive the EFI input of a second 8284 if both are driving clock inputs of separate CPU's that are to be synchronized. The variation on EFI to CLK delay over a range of 8284's may approach 35 to 45 ns. If, however, all 8284's are of the same package type, have the same relative supply voltage and operate in the same temperature environment, the variation will be reduced to between 15 and 25 ns.

There are three frequency outputs from the 8284, the oscillator (OSC) mentioned above, the system clock (CLK) which drives the CPU, and a peripheral clock (PCLK) that runs at one half the CPU clock frequency. The oscillator output is only driven by the crystal and is not affected by the F/C̄ strapping option. If a crystal is not connected to the 8284 when the external frequency input is used, the oscillator output is indeterminate. The CPU clock is derived from the selected frequency source by an internal divide by three counter. The counter generates the 33% duty cycle clock which is optimum for the CPU at maximum frequency. The peripheral clock has a 50% duty cycle and is derived from the CPU clock. Diagram 3B0 shows the relationship of CLK to OSC and PCLK to CLK. The maximum skew is 20 ns between OSC and CLK, and 22 ns between CLK and PCLK.

Since the state of the 8284 divide by three counter is indeterminate at system initialization (power on), an external sync to the counter (CSYNC) is provided to allow synchronization of the CPU clock to an external event. When CSYNC is brought high, the CLK and PCLK outputs are forced high. When CSYNC returns low, the next positive clock from the frequency source starts clock generation. CSYNC must be active for a minimum of two periods of the frequency source. If CSYNC is asynchronous to the frequency source, the circuit in Figure 3B5 should be used for synchronization. The two latches minimize the probability of a meta-stable state in the latch driving CSYNC. The latches are clocked with the inverse of the frequency source to guarantee the 8284 setup and hold time of CSYNC to the frequency source (Diag. 3B1). If a single 8284 is to be synchronized to an external event and an external frequency source is not used, the oscillator output of the 8284 may be used to

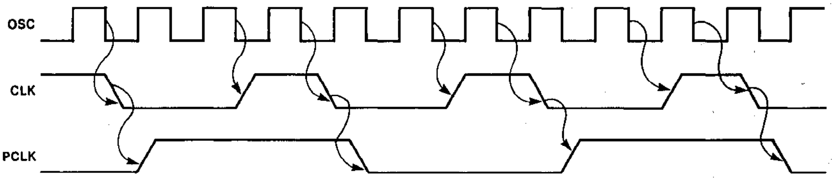


Diagram 3B0. OSC → CLK and CLK → PCLK Relationships synchronize CSYNC (Fig. 3B6). Since the oscillator output is inverted from the internal oscillator signal, the inverter in the previous example is not required. If multiple 8284's are to be synchronized, an external frequency source must drive all 8284's and a single CSYNC synchronization circuit must drive the CSYNC input of all 8284's (Fig. 3B7). Since activation of CSYNC may cause violation of CPU minimum clock low time, it should only be enabled during reset or CPU clock high. CSYNC must also be disabled a minimum of four CPU clocks before the end of reset to guarantee proper CPU reset.

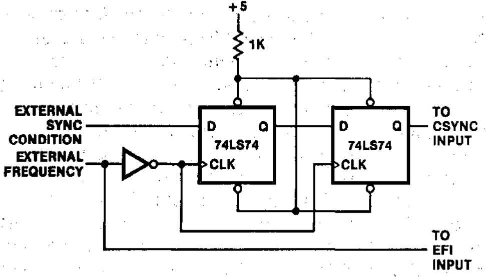
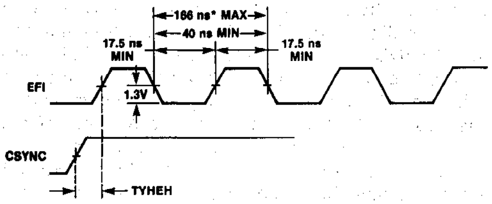


Figure 3B5. Synchronizing CSYNC with EFI



*MAX IS SPEC'ED TO GUARANTEE MAX 8086 CLOCK FREQUENCY

Diagram 3B1. CSYNC Setup and Hold to EFI

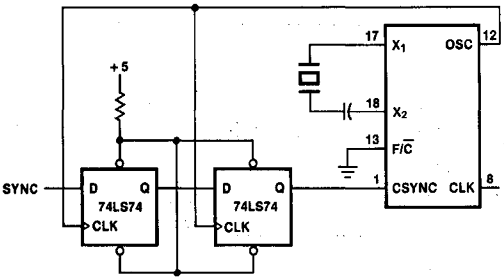


Figure 3B6. EFI from 8284 Oscillator

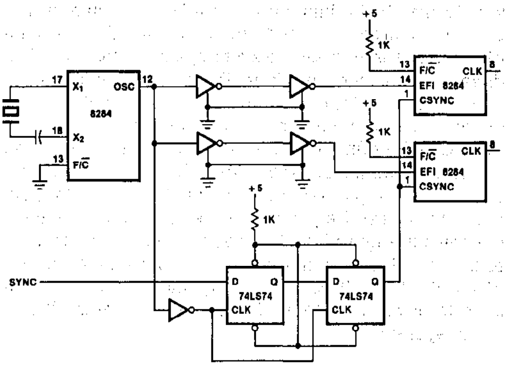


Figure 3B7. Synchronizing Multiple 8284s

Due to the fast transitions and high drive (5 mA) of the 8284 CLK output, it may be necessary to put a 10 to 100 ohm resistor in series with the clock line to eliminate ringing (resistor value depending on the amount of drive required). If multiple sources of CLK are needed with minimum skew, CLK can be buffered by a high drive device (74S241) with outputs tied to 5 volts through 100 ohms to guarantee VOH = 3.9 min (8086 minimum clock input high voltage) (Fig. 3B8). A single 8284 should not be used to generate the CLK for multiple CPU's that do not share a common local (multiplexed) bus since the 8284 synchronizes ready to the CPU and can only accommodate ready for a single CPU. If multiple CPU's share a local bus, they should be driven with the same clock to optimize transfer of bus control. Under these circumstances, only one CPU will be using the bus for a particular bus cycle which allows sharing a common READY signal (Fig. 3B9).

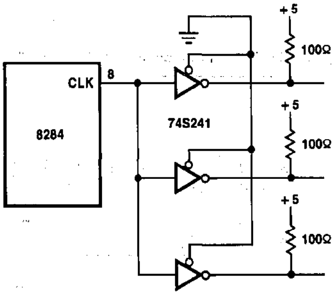


Figure 3B8. Buffering the 8284 CLK Output

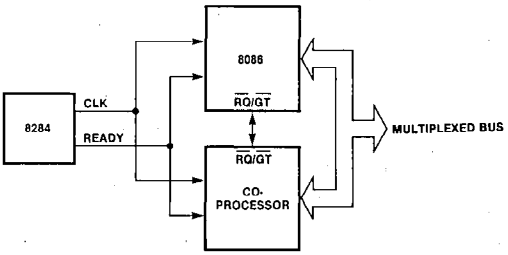


Figure 3B9. 8086 and Co-Processor on the Local Bus Share a Common 8284

3C. Reset

The 8086 requires a high active reset with minimum pulse width of four CPU clocks except after power on which requires a 50 μs reset pulse. Since the CPU internally synchronizes reset with the clock, the reset is internally active for up to one clock period after the external reset. Non-Maskable Interrupts (NMI) or hold requests on $\overline{RQ}/\overline{GT}$ which occur during the internal reset, are not acknowledged. A minimum mode hold request or maximum mode $\overline{RQ}$ pulses active immediately after the internal reset will be honored before the first instruction fetch.

From reset, the 8086 will condition the bus as shown in Table 3C1. The multiplexed bus will three-state upon detection of reset by the CPU. Other signals which three-state will be driven to the inactive state for one clock low interval prior to entering three-state (Fig. 3C1). In the minimum mode, ALE and HLDA are driven inactive and are not three-stated. In the maximum mode, $\overline{RQ}/\overline{GT}$ lines are held inactive and the queue status indicates no activity. The queue status will not indicate a reset of the queue so any user defined external circuits monitoring the queue should also be reset by the system reset. 22K ohm pull-up resistors should be connected to the CPU command and bus control lines to guarantee the inactive state of these lines in systems where leakage currents or bus capacitance may cause the voltage levels to settle below the minimum high voltage of devices in the system. In maximum mode systems, the 8288 contains internal pull-ups on the $\overline{S0}$-$\overline{S2}$ inputs to maintain the inactive state for these lines when the CPU floats the bus. The high state of the status lines during reset causes the 8288 to treat the reset sequence as a passive state. The condition of the 8288 outputs for the passive state are shown in Table 3C2. If the reset occurs during a bus cycle, the return of the status lines to the passive state will terminate the bus cycle and return the command lines to the inactive state. Note that the 8288 does not three-state the command outputs based on the passive state of the status lines. If the designer needs to three-state the CPU off the bus during reset in a single CPU system, the reset signal should also be connected to the 8288's $\overline{AEN}$ input and the output enable of the address latches (Fig. 3C2). This forces the command and address bus interface to three-state while the inactive state of DEN from the 8288 three-states the transceivers on the data bus.

Table 3C1. 8086 Bus During Reset

Signals	Condition
$AD_{15\text{-}0}$	Three-State
$A_{19\text{-}16}/S_{6\text{-}3}$	Three-State
$\overline{BHE}/S_7$	Three-State
$\overline{S2}/(M/\overline{IO})$	Driven to "1" then three-state
$\overline{S1}/(DT/\overline{R})$	Driven to "1" then three-state
$\overline{S0}/\overline{DEN}$	Driven to "1" then three-state
$\overline{LOCK}/\overline{WR}$	Driven to "1" then three-state
$\overline{RD}$	Driven to "1" then three-state
$\overline{INTA}$	Driven to "1" then three-state
ALE	0
HLDA	0
$\overline{RQ}/\overline{GT0}$	1
$\overline{RQ}/\overline{GT1}$	1
QS0	0
QS1	0

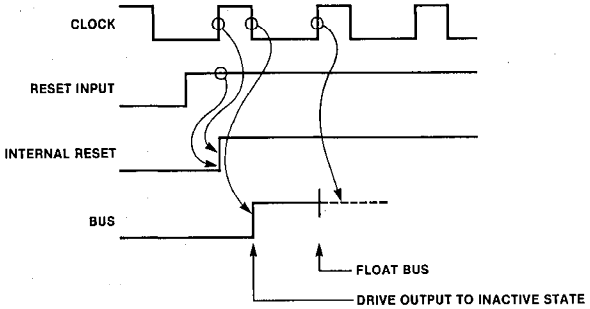


Figure 3C1. 8086 Bus Conditioning on Reset

TABLE 3C2. 8288 OUTPUTS DURING PASSIVE MODE

ALE	0
DEN	0
DT/$\overline{R}$	1
MCE/$\overline{PDEN}$	0/1
COMMANDS	1

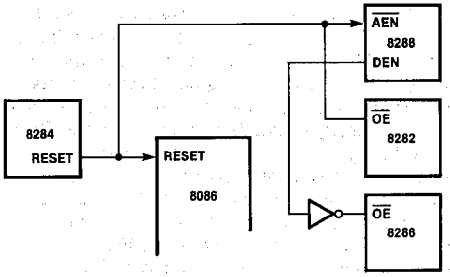


Figure 3C2. Reset Disable for Max Mode 8086 Bus Interface

For multiple processor systems using arbitration of a multimaster bus, the system reset should be connected to the $\overline{INIT}$ input of the 8289 bus arbiter in addition to the 8284 reset input (Fig. 3C3). The low active $\overline{INIT}$ input forces all 8289 outputs to their inactive state. The inactive state of the 8289 $\overline{AEN}$ output will force the 8288 to three-state the command outputs and the address latches to three-state the address bus interface. DEN inactive from the 8288 will three-state the data bus interface. For the multimaster CPU configuration, the reset should be common to all CPU's (8289's and 8284's) and satisfy the maximum of either the CPU reset requirements or 3 TBLBL (3 8289 bus clock times) + 3 TCLCL (3 8086 clock cycle times) to satisfy 8289 reset requirements.

If the 8288 command outputs are three-stated during reset, the command lines should be pulled up to V_{CC} through 2.2K ohm resistors.

The reset signal to the 8086 can be generated by the 8284. The 8284 has a schmitt trigger input ($\overline{RES}$) for generating reset from a low active external reset. The hysteresis specified in the 8284 data sheet implies that at least .25 volts will separate the 0 and 1 switching point of the 8284 reset input. Inputs without hysteresis will switch from low to high and high to low at approximately the same voltage threshold. The inputs are guaranteed to switch at specified low and high voltages (VIL and VIH) but the actual switching point is anywhere in-between. Since VIL min is specified at .8 volts, the hysteresis guarantees that the reset will be active until the input reaches at least 1.05 volts. A reset will not be recognized until the input drops at least .25 volts below the reset inputs VIH of 2.6 volts.

To guarantee reset from power up, the reset input must remain below 1.05 volts for 50 microseconds after V_{CC} has reached the minimum supply voltage of 4.5 volts. The hysteresis allows the reset input to be driven by a simple RC circuit as shown in Figure 3C4. The calculated RC value does not include time for the power supply to reach 4.5 volts or the charge accumulated during this interval. Without the hysteresis, the reset output might oscillate as the input voltage passes through the switching voltage of the input. The calculated RC value provides the minimum required reset period of 50 microseconds for 8284's that switch at the 1.05 volt level and a reset period of approximately 162 microseconds for 8284's that switch at the 2.6 volt level. If tighter tolerance between the minimum and maximum reset times is necessary, the reset circuit shown in Figure 3C5 might be used rather than the simple RC circuit. This circuit provides a constant current source and a linear charge rate on the capacitor rather than the inverse exponential charge rate of the RC circuit. The maximum reset period for this implementation is 124 microseconds.

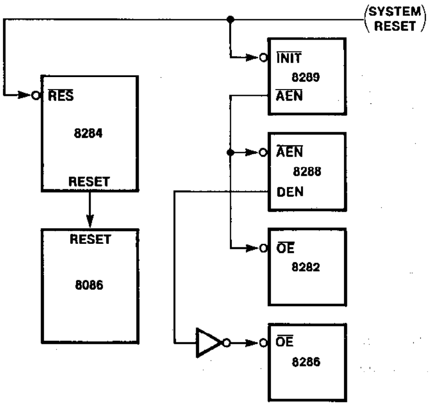


Figure 3C3. Reset Disable of for Max Mode 8086 Bus Interface in Multi CPU System

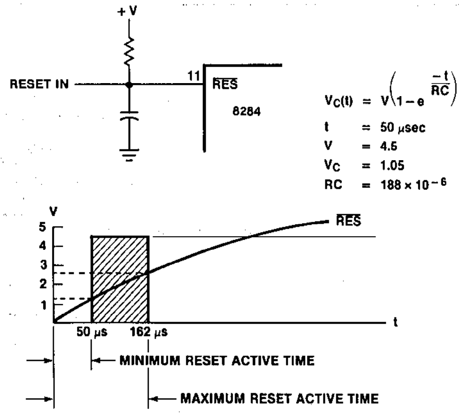

+V
RESET IN
11
$\overline{RES}$
8284
$V_C(t) = V\left(1 - e^{\frac{-t}{RC}}\right)$
$t = 50\ \mu sec$
$V = 4.5$
$V_C = 1.05$
$RC = 188 \times 10^{-6}$
V
5
4
3
2
1
$\overline{RES}$
t
50 μs
162 μs
MINIMUM RESET ACTIVE TIME
MAXIMUM RESET ACTIVE TIME

Figure 3C4. 8284 Reset Circuit

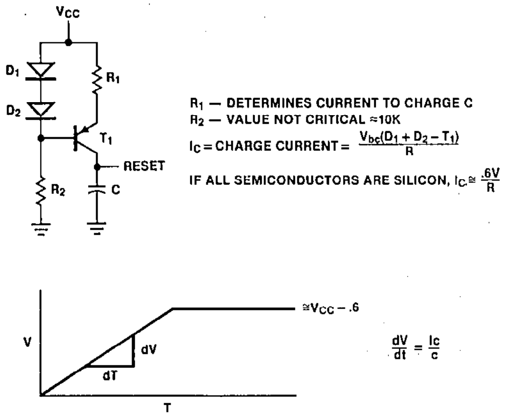

Vcc
D1
D2
R1
T1
RESET
R2
C
R1 — DETERMINES CURRENT TO CHARGE C
R2 — VALUE NOT CRITICAL ≈10K
I_C = CHARGE CURRENT = $\frac{V_{bc}(D_1 + D_2 - T_1)}{R}$
IF ALL SEMICONDUCTORS ARE SILICON, $I_C \cong \frac{.6V}{R}$
≅Vcc − .6
V
dV
dT
T
$\frac{dV}{dt} = \frac{I_C}{C}$

Figure 3C5. Constant Current Power-On Reset Circuit

The 8284 synchronizes the reset input with the CPU clock to generate the RESET signal to the CPU (Fig. 3C6). The output is also available as a general reset to the entire system. The reset has no effect on any clock circuits in the 8284.

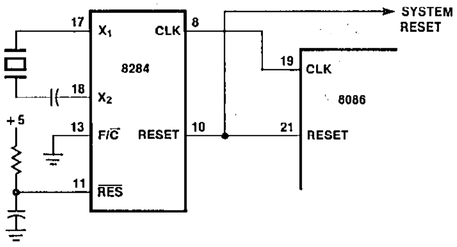


Figure 3C6. 8086 Reset and System Reset

3D. Ready Implementation and Timing

As discussed previously, the ready signal is used in the system to accommodate memory and I/O devices that cannot transfer information at the maximum CPU bus bandwidth. Ready is also used in multiprocessor systems to force the CPU to wait for access to the system bus or Multibus system bus. To insert a wait state in the bus cycle, the READY signal to the CPU must be inactive (low) by the end of T2. To avoid insertion of a wait state, READY must be active (high) within a specified setup time prior to the positive transition during T3. Depending on the size and characteristics of the system, ready implementation may take one of two approaches.

The classical ready implementation is to have the system 'normally not ready.' When the selected device receives the command (R̄D̄/W̄R̄/ĪN̄T̄Ā) and has had sufficient time to complete the command, it activates READY to the CPU, allowing the CPU to terminate the bus cycle. This implementation is characteristic of large multiprocessor, Multibus systems or systems where propagation delays, bus access delays and device characteristics inherently slow down the system. For maximum system performance, devices that can run with no wait states must return 'READY' within the previously described limit. Failure to respond in time will only result in the insertion of one or more wait cycles.

An alternate technique is to have the system 'normally ready.' All devices are assumed to operate at the maximum CPU bus bandwidth. Devices that do not meet the requirement must disable READY by the end of T2 to guarantee the insertion of wait cycles. This implementation is typically applied to small single CPU systems and reduces the logic required to control the ready signal. Since the failure of a device requiring wait states to disable READY by the end of T2 will result in premature termination of the bus cycle, the system timing must be carefully analyzed when using this approach.

The 8086 has two different timing requirements on READY depending on the system implementation. For a 'normally ready' system to insert a wait state, the READY must be disabled within 8 ns (TRYLCL) after the end of T2 (start of T3) (Diag. 3D1). To guarantee proper

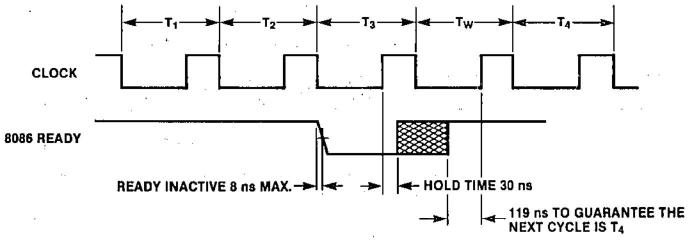


Diagram 3D1. Normally Ready System Inserting a Wait State operation of the 8086, the READY input must not change from ready to not ready during the clock low time of T3. For a 'normally not ready' system to avoid wait states, READY must be active within 119 ns (TRYHCH) of the positive clock transition during T3 (Diag. 3D2). For both cases, READY must satisfy a hold time of 30 ns (TCHRYX) from the T3 or TW positive clock transition.

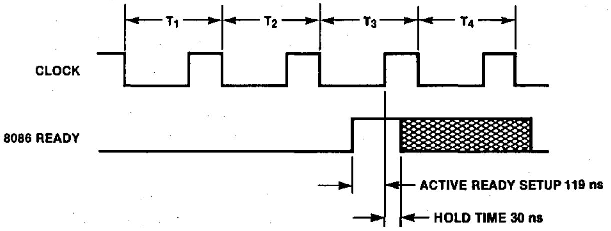


Diagram 3D2. Normally Not Ready System Avoiding a Wait State

To generate a stable READY signal which satisfies the previous setup and hold times, the 8284 provides two separate system ready inputs (RDY1, RDY2) and a single synchronized ready output (READY) for the CPU. The RDY inputs are qualified with separate access enables ($\overline{AEN1}$,$\overline{AEN2}$, low active) to allow selecting one of the two ready signals (Fig. 3D1). The gated signals are logically OR'ed and sampled at the beginning of each CLK cycle to generate READY to the CPU (Diag. 3D3). The sampled READY signal is valid within 8 ns (TRYLCL) after CLK to satisfy the CPU timing requirements on 'not ready' and ready. Since READY cannot change until the next CLK, the hold time requirements are also satisfied. The system ready inputs to the 8284 (RDY1,RDY2) must be valid 35 ns (TRIVCL) before T3 and $\overline{AEN}$ must be valid 60 ns before T3. For a system using only one RDY input, the associated $\overline{AEN}$ is tied to ground while the other $\overline{AEN}$ is connected to 5 volts through ~1K ohms (Fig. 3D2a). If the system generates a low active ready signal, it can be connected to the 8284 $\overline{AEN}$ input if the additional setup time required by the 8284 $\overline{AEN}$ input is satisfied. In this case, the associated RDY input would be tied high (Fig. 3D2b).

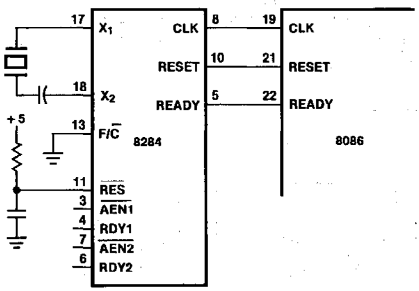
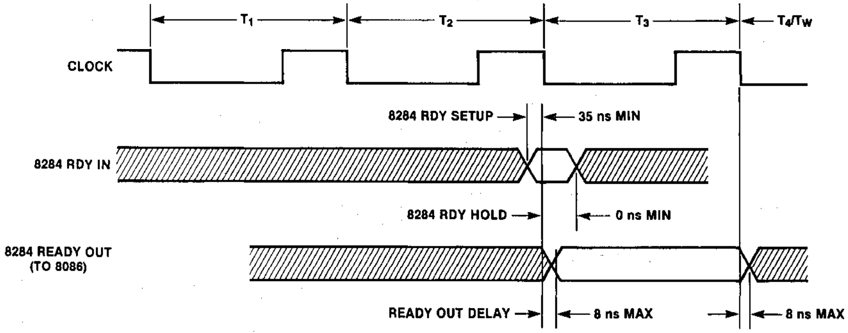


Figure 3D1. Ready Inputs to the 8284 and Output to the 8086



NOTE: THE 8284 DATA SHEET SPECIFIES READY OUT DELAY (TRYLCL) AS −8 ns 'BEFORE' THE END OF T_2 WHICH IMPLIES THE TIMING SHOWN.

Diagram 3D3. 8284 with 8086 Ready Timing

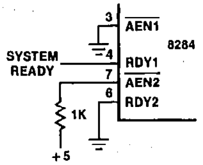


Figure 3D2a. Using RDY1/RDY2 to Generate Ready

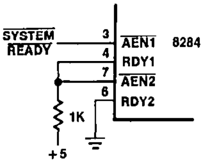


Figure 3D2b. Using $\overline{\text{AEN1}}/\overline{\text{AEN2}}$ to Generate Ready

The majority of memory and peripheral devices which fail to operate at the maximum CPU frequency typically do not require more than one wait state. The circuit given in Figure 3D3 is an example of a simple wait state generator. The system ready line is driven low whenever a device requiring one wait state is selected. The flip flop is cleared by ALE, enabling RDY to the 8284. If no wait states are required, the flip flop does not change. If the system ready is driven low, the flip flop toggles on the low to high clock transition of T2 to force one wait state. The next low to high clock transition toggles the flip flop again to indicate ready and allow completion of the bus cycle. Further changes in the state of the flip flop will not affect the bus cycle. The circuit allows approximately 100 ns for chip select decode and conditioning of the system ready (Diag. 3D4).

If the system is 'normally not ready,' the programmer should not assign executable code to the last six bytes of physical memory. Since the 8086 prefetches instructions, the CPU may attempt to access non-existent memory when executing code at the end of physical memory. If the access to non-existent memory fails to enable READY, the system will be caught in an indefinite wait.

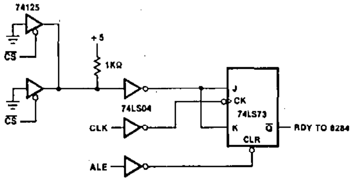


Figure 3D3. Single Wait State Generator

3E. Interrupt Structure

The 8086 interrupt structure is based on a table of interrupt vectors stored in memory locations 0H through 003FFH. Each vector consists of two bytes for the instruction pointer and two bytes for the code segment. These two values combine to form the address of the interrupt service routine. This allows the table to contain up to 256 interrupt vectors which specify the starting address of the service routines anywhere in the one megabyte address space of the 8086. If fewer than 256 different interrupts are defined in the system, the user need only allocate enough memory for the interrupt vector table to provide the vectors for the defined interrupts. During initial system debug, however, it may be desirable to assign all undefined interrupt types to a trap routine to detect erroneous interrupts.

Each vector is associated with an interrupt type number which points to the vector's location in the interrupt vector table. The interrupt type number multiplied by four gives the displacement of the first byte of the associated interrupt vector from the beginning of the table. As an example, interrupt type number 5 points to the sixth entry in the interrupt vector table. The contents of this entry in the table points to the interrupt service routine for type 5 (Fig. 3E1). This structure allows the user to specify the memory address of each service routine by placing the address (instruction pointer and code segment values) in the table location provided for that type interrupt.

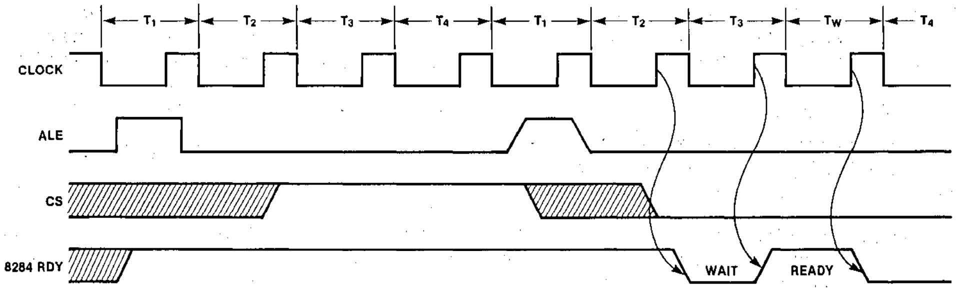


Diagram 3D4.

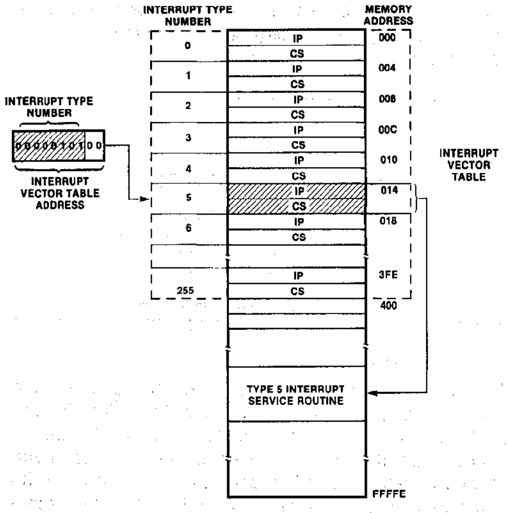


Figure 3E1. Direction to Interrupt Service Routine through the Interrupt Vector Table

All interrupts in the 8086 must be assigned an interrupt type which uniquely identifies each interrupt. There are three classes of interrupt types in the 8086; predefined interrupt types which are issued by specific functions within the 8086 and user defined hardware and software interrupts. Note that any interrupt type including the predefined interrupts can be issued by the user's hardware and/or software.

PREDEFINED INTERRUPTS

The predefined interrupt types in the 8086 are listed below with a brief description of how each is invoked. When invoked, the CPU will transfer control to the memory location specified by the vector associated with the specific type. The user must provide the interrupt service routine and initialize the interrupt vector table with the appropriate service routine address. The user may additionally invoke these interrupts through hardware or software. If the preassigned function is not used in the system, the user may assign some other function to the associated type. However, for compatibility with future Intel hardware and software products for the 8086 family, interrupt types 0-31 should not be assigned as user defined interrupts.

TYPE 0 — DIVIDE ERROR

This interrupt type is invoked whenever a division operation is attempted during which the quotient exceeds the maximum value (ex. division by zero). The interrupt is non-maskable and is entered as part of the execution of the divide instruction. If interrupts are not reenabled by the divide error interrupt service routine, the service routine execution time should be included in the worst case divide instruction execution time (primarily when considering the longest instruction execution time and its effect on latency to servicing hardware interrupts).

TYPE 1 — SINGLE STEP

This interrupt type occurs one instruction after the TF (Trap Flag) is set in the flag register. It is used to allow software single stepping through a sequence of code. Single stepping is initiated by copying the flags onto the stack, setting the TF bit on the stack and popping the flags. The interrupt routine should be the single step routine. The interrupt sequence saves the flags and program counter, then resets the TF flag to allow the single step routine to execute normally. To return to the routine under test, an interrupt return restores the IP, CS and flags with TF set. This allows the execution of the next instruction in the program under test before trapping back to the single step routine. Single Step is not masked by the IF (Interrupt Flag) bit in the flag register.

TYPE 2 — NMI (Non-Maskable Interrupt)

This is the highest priority hardware interrupt and is non-maskable. The input is edge triggered but is synchronized with the CPU clock and must be active for two clock cycles to guarantee recognition. The interrupt signal may be removed prior to entry to the service routine. Since the input must make a low to high transition to generate an interrupt, spurious transitions on the input should be suppressed. If the input is normally high, the NMI low time to guarantee triggering is two CPU clock times. This input is typically reserved for catastrophic failures like power failure or timeout of a system watchdog timer.

TYPE 3 — ONE BYTE INTERRUPT

This is invoked by a special form of the software interrupt instruction which requires a single byte of code space. Its primary use is as a breakpoint interrupt for software debug. With full representation within a single byte, the instruction can map into the smallest instruction for absolute resolution in setting breakpoints. The interrupt is not maskable.

TYPE 4 — INTERRUPT ON OVERFLOW

This interrupt occurs if the overflow flag (OF) is set in the flag register and the INTO instruction is executed. The instruction allows trapping to an overflow error service routine. The interrupt is non-maskable.

Interrupt types 0 and 2 can occur without specific action by the programmer (except for performing a divide for Type 0) while types 1, 3, and 4 require a conscious act by the programmer to generate these interrupt types. All but type 2 are invoked through software activity and are directly associated with a specific instruction.

USER DEFINED SOFTWARE INTERRUPTS

The user can generate an interrupt through the software with a two byte interrupt instruction INT nn. The first byte is the INT opcode while the second byte (nn) contains the type number of the interrupt to be performed. The INT instruction is not maskable by the interrupt enable flag. This instruction can be used to transfer control to routines that are dynamically relocatable and whose location in memory is not known by the calling program. This technique also saves the flags of the calling program on the stack prior to transferring control. The called procedure must return control with an interrupt return (IRET) instruction to remove the flags from the stack and fully restore the state of the calling program.

All interrupts invoked through software (all interrupts discussed thus far with the exception of NMI) are not maskable with the IF flag and initiate the transfer of control at the end of the instruction in which they occur. They do not initiate interrupt acknowledge bus cycles and will disable subsequent maskable interrupts by resetting the IF and TF flags. The interrupt vector for these interrupt types is either implied or specified in the instruction. Since the NMI is an asynchronous event to the CPU, the point of recognition and initiation of the transfer of control is similar to the maskable hardware interrupts.

USER DEFINED HARDWARE INTERRUPTS

The maskable interrupts initiated by the system hardware are activated through the INTR pin of the 8086 and are masked by the IF bit of the status register (interrupt flag). During the last clock cycle of each instruction, the state of the INTR pin is sampled. The 8086 deviates from this rule when the instruction is a MOV or POP to a segment register. For this case, the interrupts are not sampled until completion of the following instruction. This allows a 32-bit pointer to be loaded to the stack pointer registers SS and SP without the danger of an interrupt occurring between the two loads. Another exception is the WAIT instruction which waits for a low active input on the $\overline{\text{TEST}}$ pin. This instruction also continuously samples the interrupt request during its execution and allows servicing interrupts during the wait. When an interrupt is detected, the WAIT instruction is again fetched prior to servicing the interrupt to guarantee the interrupt routine will return to the WAIT instruction.

UNINTERRUPTABLE INSTRUCTION SEQUENCE

```
    MOV SS, NEW$STACK$SEGMENT
    MOV SP, NEW$STACK$POINTER
```

Also, since prefixes are considered part of the instruction they precede, the 8086 will not sample the interrupt line until completion of the instruction the prefix(es) precede(s). An exception to this (other than HALT or WAIT) is the string primatives preceded by the repeat (REP) prefix. The repeated string operations will sample the interrupt line at the completion of each repetition. This includes repeat string operations which include the lock prefix. If multiple prefixes precede a repeated string operation, and the instruction is interrupted, only the prefix immediately preceding the string primative is restored. To allow correct resumption of the operation, the following programming technique may be used:

```
LOCKED$BLOCK$MOVE: LOCK REP MOVS DEST, CS:SOURCE
                   AND CX,     CX
                   JNZ LOCKED$BLOCK$MOVE
```

The code bytes generated by the 8086 assembler for the MOVS instruction are (in descending order): LOCK prefix, REP prefix, Segment Override prefix and MOVS. Upon return from the interrupt, the segment override prefix is restored to guarantee one additional transfer is performed between the correct memory locations. The instructions following the move operation test the repetition count value to determine if the move was completed and return if not.

If the INTR pin is high when sampled and the IF bit is set to enable interrupts, the 8086 executes an interrupt acknowledge sequence. To guarantee the interrupt will be acknowledged, the INTR input must be held active until the interrupt acknowledge is issued by the CPU. If the BIU is running a bus cycle when the interrupt condition is detected (as would occur if the BIU is fetching an instruction when the current instruction completes), the

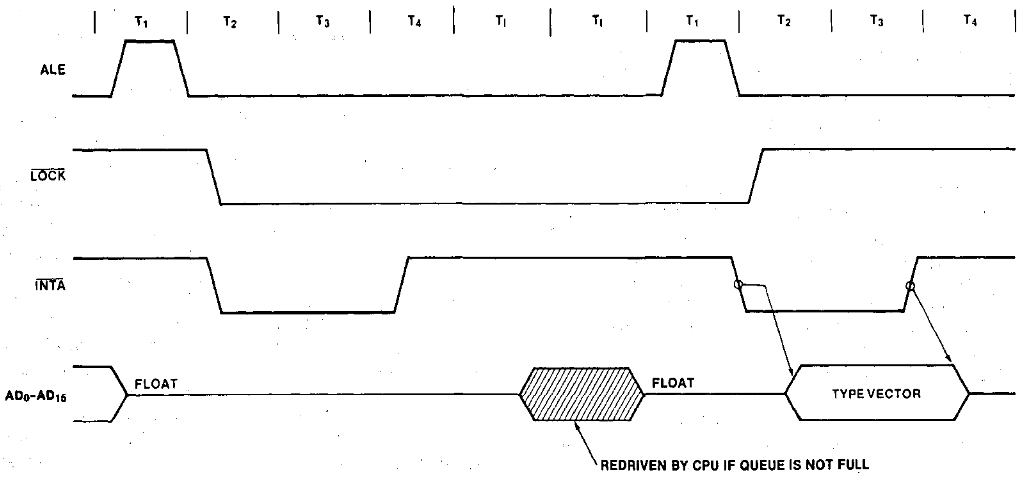


Figure 3E2. Interrupt Acknowledge Sequence

interrupt must be valid at the 8086 2 clock cycles prior to T4 of the bus cycle if the next cycle is to be an interrupt acknowledge cycle. If the 2 clock setup is not satisfied, another pending bus cycle will be executed before the interrupt acknowledge is issued. If a hold request is also pending (this might occur if an interrupt and hold request are made during execution of a locked instruction), the interrupt is serviced after the hold request is serviced.

The interrupt acknowledge sequence is only generated in response to an interrupt on the 8086 INTR input. The associated bus activity is shown in Figure 3E2. The cycle consists of two $\overline{\text{INTA}}$ bus cycles separated by two idle clock cycles. During the bus cycles the $\overline{\text{INTA}}$ command is issued rather than read. No address is provided by the 8086 during either bus cycle ($\overline{\text{BHE}}$ and status are valid), however, ALE is still generated and will load the address latches with indeterminate information. This condition requires that devices in the system do not drive their outputs without being qualified by the Read Command. As will be shown later, the ALE is useful in maximum mode systems with multiple 8259A priority interrupt controllers. During the $\overline{\text{INTA}}$ bus cycles, DT/$\overline{\text{R}}$ and DEN are conditioned to allow the 8086 to receive a one byte interrupt type number from the interrupt system. The first $\overline{\text{INTA}}$ bus cycle signals an interrupt acknowledge cycle is in progress and allows the system to prepare to present the interrupt type number on the next $\overline{\text{INTA}}$ bus cycle. The CPU does not capture information on the bus during the first cycle. The type number must be transferred to the 8086 on the lower half of the 16-bit data bus during the second cycle. This implies that devices which present interrupt type numbers to the 8086 must be located on the lower half of the 16-bit data bus. The timing of the $\overline{\text{INTA}}$ bus cycles (with exception of address timing) is similar to read cycle timing. The 8086 interrupt acknowledge sequence deviates from the form used on 8080 and 8085 in that no instruction is issued as part of the sequence. The 8080 and 8085 required either a restart or call instruction be issued to affect the transfer of control.

In the minimum mode system, the M/$\overline{\text{IO}}$ signal will be low indicating I/O during the $\overline{\text{INTA}}$ bus cycles. The 8086 internal $\overline{\text{LOCK}}$ signal will be active from T2 of the first bus cycle until T2 of the second to prevent the BIU from honoring a hold request between the two $\overline{\text{INTA}}$ cycles.

In the maximum mode, the status lines $\overline{\text{S0}}$-$\overline{\text{S2}}$ will request the 8288 to activate the $\overline{\text{INTA}}$ output for each cycle. The $\overline{\text{LOCK}}$ output of the 8086 will be active from T2 of the first cycle until T2 of the second to prevent the 8086 from honoring a hold request on either $\overline{\text{RQ}}$/$\overline{\text{GT}}$ input and to prevent bus arbitration logic from relinquishing the bus between $\overline{\text{INTA}}$'s in multi-master systems. The consequences of READY are identical to those for READ and WRITE cycles.

Once the 8086 has the interrupt type number (from the bus for hardware interrupts, from the instruction stream for software interrupts or from the predefined condition), the type number is multiplied by four to form the displacement to the corresponding interrupt vector in the interruot vector table. The four bytes of the interrupt vector are: least significant byte of the instruction pointer, most significant byte of the instruction pointer, least significant byte of the code segment register, most significant byte of the code segment register. During the transfer of control, the CPU pushes the flags and current code segment register and instruction pointer onto the stack. The new code segment and instruction pointer values are loaded and the single step and interrupt flags are reset. Resetting the interrupt flag disables response to further hardware interrupts in the service routine unless the flags are specifically re-enabled by the service routine. The CS and IP values are read from the interrupt vector table with data read cycles. No segment registers are used when referencing the vector table during the interrupt context switch. The vector displacement is added to zero to form the 20-bit address and S4, S3 = 10 indicating no segment register selection.

The actual bus activity associated with the hardware interrupt acknowledge sequence is as follows: Two interrupt acknowledge bus cycles, read new IP from the interrupt vector table, read new CS from the interrupt vector table, Push flags, Push old CS, Opcode fetch of the first instruction of the interrupt service routine, and Push old IP. After saving the old IP, the BIU will resume normal operation of prefetching instructions into the queue and servicing EU requests for operands. S5 (interrupt enable flag status) will go inactive in the second clock cycle following reading the new CS.

The number of clock cycles from the end of the instruction during which the interrupt occurred to the start of interrupt routine execution is 61 clock cycles. For software generated interrupts, the sequence of bus cycles is the same except no interrupt acknowledge bus cycles are executed. This reduces the delay to service routine execution to 51 clocks for INT nn and single step, 52 clocks for INT3 and 53 clocks for INTO. The same interrupt setup requirements with respect to the BIU that were stated for the hardware interrupts also apply to the software interrupts. If wait states are inserted by either the memories or the device supplying the interrupt type number, the given clock times will increase accordingly.

When considering the precedence of interrupts for multiple simultaneous interrupts, the following guidelines apply: 1. INTR is the only maskable interrupt and if detected simultaneously with other interrupts, resetting of IF by the other interrupts will mask INTR. This causes INTR to be the lowest priority interrupt serviced after all other interrupts unless the other interrupt service routines reenable interrupts. 2. Of the nonmaskable interrupts (NMI, Single Step and software generated), in general, Single Step has highest priority (will be serviced first) followed by NMI, followed by the software interrupts. This implies that a simultaneous NMI and Single Step trap will cause the NMI service routine to follow single step; a simultaneous software trap and Single Step trap will cause the software interrupt service routine to follow single step and a simultaneous NMI and software trap will cause the NMI service routine to be executed followed by the software interrupt service routine. An exception to this priority structure occurs if all three interrupts are pending. For this case, transfer of control to the software interrupt service routine followed by the NMI trap will cause both the NMI and software interrupt service routines to be executed without single stepping. Single stepping resumes upon execution of the instruction following the instruction causing the software interrupt (the next instruction in the routine being single stepped).

If the user does not wish to single step before INTR service routines, the single step routine need only disable interrupts during execution of the program being single stepped and reenable interrupts on entry to the single step routine. Disabling the interrupts during the program under test prevents entry into the interrupt service routine while single step (TF = 1) is active. To prevent single stepping before NMI service routines, the single step routine must check the return address on the stack for the NMI service routine address and return control to that routine without single step enabled. As examples, consider Figures 3E3a and 3E3b. In 3E3a Single Step and NMI occur simultaneously while in 3E3b, NMI, INTR and a divide error all occur during a divide instruction being single stepped.

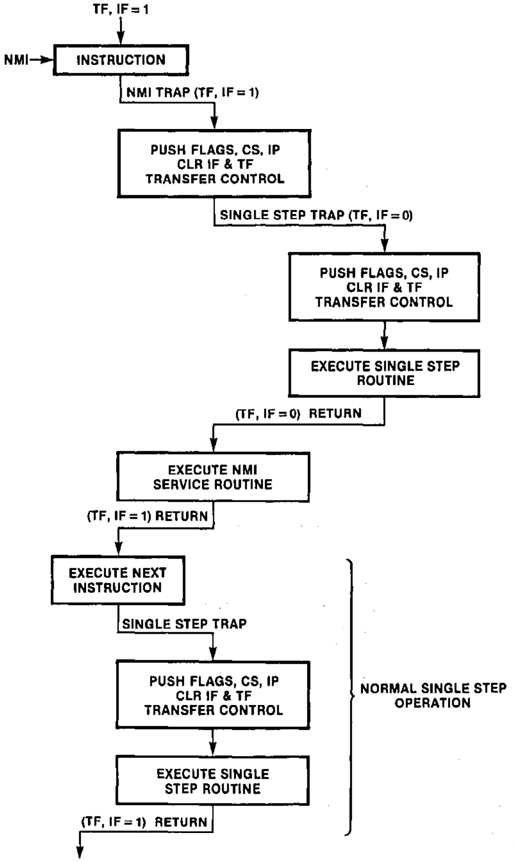


Figure 3E3a. NMI During Single Stepping and Normal Single Step Operation

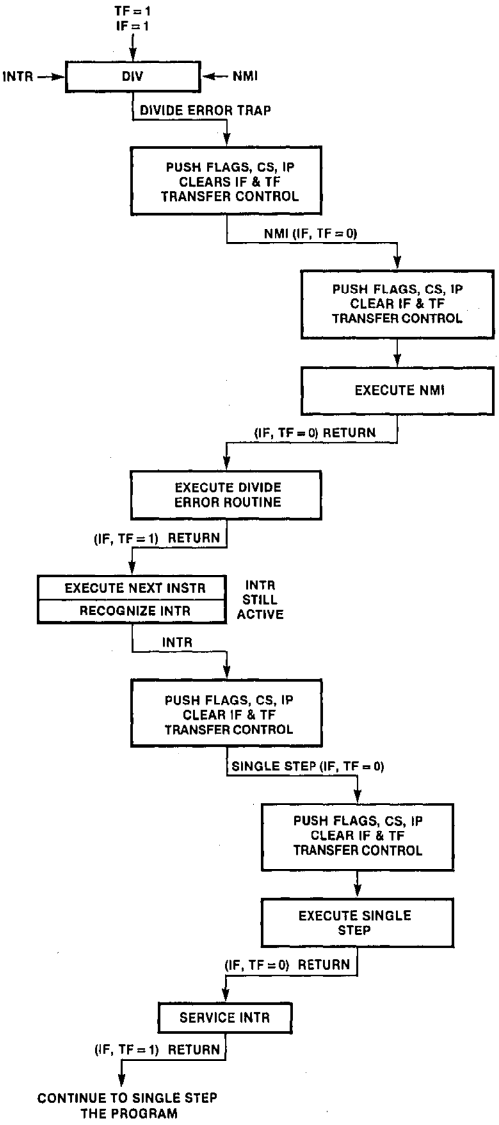
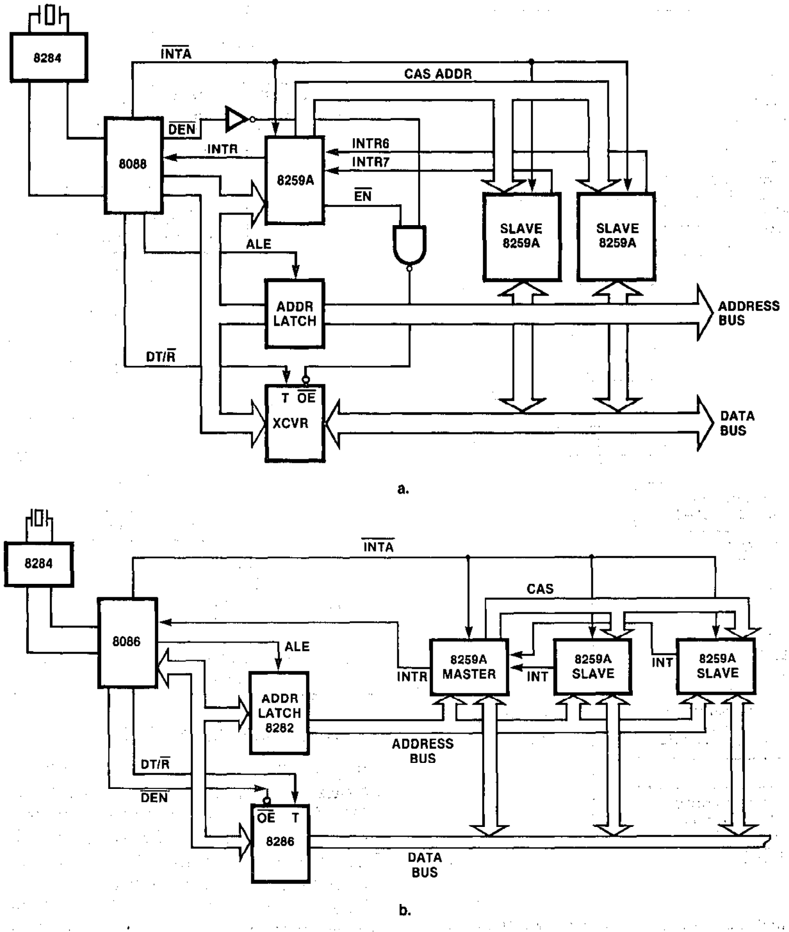


Figure 3E3b. NMI, INTR, Single Step and Divide Error Simultaneous Interrupts

SYSTEM CONFIGURATIONS

To accommodate the $\overline{\text{INTA}}$ protocol of the maskable hardware interrupts, the 8259A is provided as part of the 8086 family. This component is programmable to operate in both 8080/8085 systems and 8086 systems. The devices are cascadable in master/slave arrangements to allow up to 64 interrupts in the system. Figures 3E4 and 3E5 are examples of 8259A's in minimum and maximum mode 8086 systems. The minimum mode configuration (a) shows an 8259A connected to the CPU's multiplexed bus. Configuration (b) illustrates an 8259A connected to a demultiplexed bus system. These interconnects are also applicable to maximum mode systems. The configuration given for a maximum mode system shows a master 8259A on the CPU's multiplexed bus with additional slave 8259A's out on the buffered system bus. This configuration demonstrates several unique features of the maximum mode system interface. If the master 8259A receives interrupts from a mix of slave 8259A's and regular interrupting devices, the slaves must provide the type number for devices connected to them while the master provides the type number for devices directly attached to its interrupt inputs. The master 8259A is programmable to determine if an interrupt is from a direct input or a slave 8259A and will use this information to enable or disable the data bus transceivers (via the 'nand' function of DEN and $\overline{\text{EN}}$). If the master must provide the type number, it will disable the data bus transceivers. If the slave provides the type number, the master will enable the data bus transceivers. The $\overline{\text{EN}}$ output is normally high to allow the 8086/8288 to control the bus transceivers. To select the proper slave when servicing a slave interrupt, the master must provide a cascade address to the slave. If the 8288 is not strapped in the I/O bus mode (the 8288 IOB input connected to ground), the MCE/$\overline{\text{PDEN}}$ output becomes a MCE or Master Cascade Enable output. This signal is only active during $\overline{\text{INTA}}$ cycles as shown in Figure 3E6 and enables the master 8259A's cascade address onto the 8086's local bus during ALE. This allows the address latches to capture the cascade address with ALE and allows use of the system address bus for selecting the proper slave 8259A. The MCE is gated with $\overline{\text{LOCK}}$ to minimize local bus contention between the 8086 three-stating its bus outputs and the cascade address being enabled onto the bus. The first $\overline{\text{INTA}}$ bus cycle allows the master to resolve internal priorities and output a cascade address to be transmitted to the slaves on the subsequent $\overline{\text{INTA}}$ bus cycle. For additional information on the 8259A, reference application note AP-59.



a.

b.

Figure 3E4. Min Mode 8086 with Master 8259A on the Local Bus and Slave 8259As on the System Bus

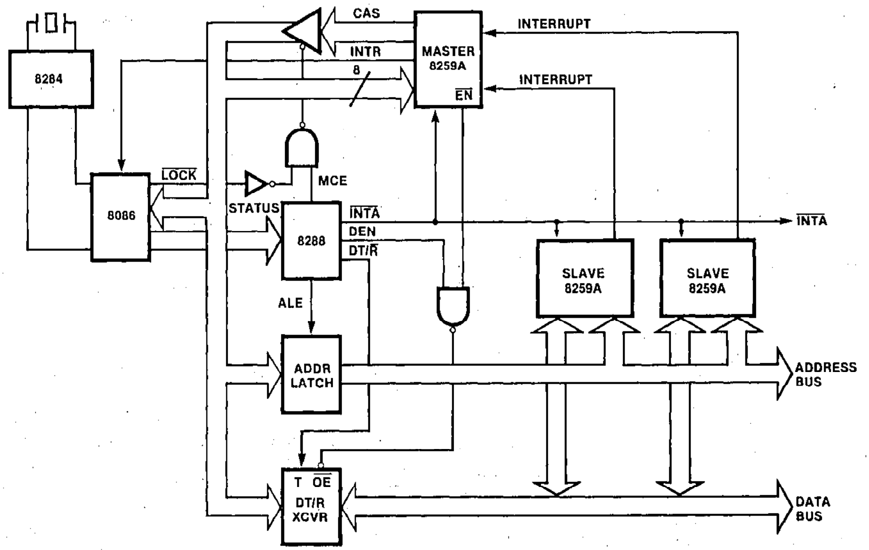


Figure 3E5. Max Mode 8086 with Master 8259A on the Local Bus and Slave 8259As on the System Bus

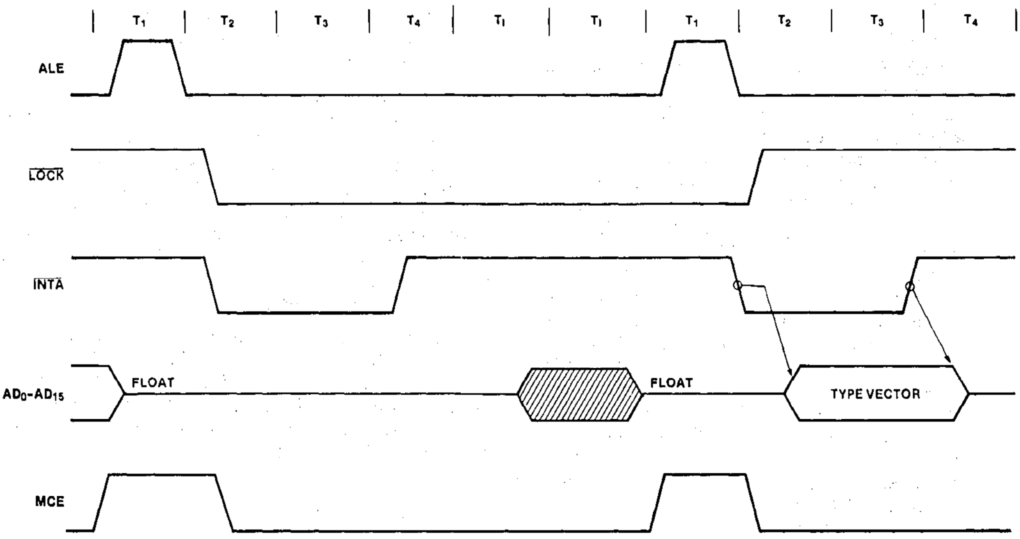


Figure 3E6. MCE Timing to Gate 8259A CAS Address onto the 8086 Local Bus

3F. Interpreting the 8086 Bus Timing Diagrams

At first glance, the 8086 bus timing diagrams (Diag. 3F1 min mode and Diag. 3F2 max mode) appear rather complex. However, with a few words of explanation on how to interpret them, they become a powerful tool in determining system requirements. The timing diagrams for both the minimum and maximum modes may be divided into six sections: (1) address and ALE timing; (2) read cycle timing; (3) write cycle timing; (4) interrupt acknowledge timing; (5) ready timing; and (6) HOLD/HLDA or $\overline{RQ}/\overline{GT}$ timing. Since the A.C. characteristics of the signals are specified relative to the CPU clock, the relationship between the majority of signals can be deduced by simply determining the clock cycles between the clock edges the signals are relative to and adding or subtracting the appropriate minimum or maximum parameter values. One aspect of system timing not compensated for in this approach is the worst case relationship between minimum and maximum parameter values (also known as tracking relationships). As an example, consider a signal which has specified minimum and maximum turn on and turn off delays. Depending on device characteristics, it may not be possible for the component to simultaneously demonstrate a maximum turn-on and minimum turn-off delay even though worst case analysis might imply the possibility. This argument is characteristic of MOS devices and is therefore applicable to the 8086 A.C. characteristics. The message is: worst case analysis mixing minimum and maximum delay parameters will typically exceed the worst case obtainable and therefore should not be subjected to further subjective degradation to obtain worst-worst case values. This section will provide guidelines for specific areas of 8086 timing sensitive to tracking relationships.

A. MINIMUM MODE BUS TIMING

1. ADDRESS and ALE

The address/ALE timing relationship is important to determine the ability to capture a valid address from the multiplexed bus. Since the 8282 and 8283 latches capture the address on the trailing edge of ALE, the critical timing involves the state of the address lines when ALE terminates. If the address valid delay is assumed to be maximum TCLAV and ALE terminates at its earliest point, TCHLLmin (assuming zero minimum delay), the address would be valid only TCLCHmin-TCLAVmax = 8 ns prior to ALE termination. This result is unrealistic in the assumption of maximum TCLAV and minimum TCHLL. To provide an accurate measure of the true worst case, a separate parameter specifies the minimum time for address valid prior to the end of ALE (TAVAL). TAVAL = TCLCH-60 ns overrides the clock related timings and guarantees 58 ns of address setup to ALE termination for a 5 MHz 8086. The address is guaranteed to remain valid beyond the end of ALE by the TLLAX parameter. This specification overrides the relationship between TCHLL and TCLAX which might seem to imply the address may not be valid by the end of the latest possible ALE. TLLAX holds for the entire address bus. The TCLAXmin spec on the address indicates the earliest the bus will go invalid if not restrained by a slow ALE. TLLAX and TCLAX apply to the entire multiplexed bus for both read and write cycles. AD15-0 is three-stated for read cycles and immediately switched to write data during write cycles. AD19-16 immediately switch from address to status for both read and write cycles. The minimum ALE pulse width is guaranteed by TLHLLmin which takes precedence over the value obtained by relating TCLLHmax and TCHLLmin.

To determine the worst case delay to valid address on a demultiplexed address bus, two paths must be considered: (1) delay of valid address and (2) delay to ALE. Since the 8282 and 8283 are flow through latches, a valid address is not transmitted to the address bus until ALE is active. A comparison of address valid delay TCLAVmax with ALE active delay TCLLHmax indicates TCLAVmax is the worst case. Subtracting the latch propagation delay gives the worst case address bus valid delay from the start of the bus cycle.

2. Read Cycle Timing

Read timing consists of conditioning the bus, activating the read command and establishing the data transceiver enable and direction controls. DT/$\overline{R}$ is established early in the bus cycle and requires no further consideration. During read, the $\overline{DEN}$ signal must allow the transceivers to propagate data to the CPU with the appropriate data setup time and continue to do so until the required data hold time. The $\overline{DEN}$ turn on delay allows TCLCL + TCHCLmin − TCVCTVmax − TDVCL = 127 ns transceiver enable time prior to valid data required by the CPU. Since the CPU data hold time TCLDXmin and minimum $\overline{DEN}$ turnoff delay TCVCTXmin are both 10 ns relative to the same clock edge, the hold time is guaranteed. Additionally, $\overline{DEN}$ must disable the transceivers prior to the CPU redriving the bus with the address for the next bus cycle. The maximum $\overline{DEN}$ turn off delay (TCVCTXmax) compared with the minimum delay for addresses out of the 8086 (TCLCL + TCLAVmin) indicates the transceivers are disabled at least 105 ns before the CPU drives the address onto the multiplexed bus.

If memory or I/O devices are connected directly to the multiplexed address and data bus, the TAZRL parameter guarantees the CPU will float the bus before activating read and allowing the selected device to drive the bus. At the end of the bus cycle, the TRHAV parameter specifies the bus float delay the device being deselected must satisfy to avoid contention with the CPU driving the address for the next bus cycle. The next bus cycle may start as soon as the cycle following T4 or any number of clock cycles later.

The minimum delay from read active to valid data at the CPU is 2TCLCL − TCLRLmax − TDVCL = 205 ns. The minimum pulse width is 2TCLCL − 75 ns = 325 ns. This specification (TRLRH) overrides the result which could be derived from clock relative delays (2TCLCL − TCLRLmax + TCLRHmin).

3. Write Cycle Timing

The write cycle involves providing write data to the system, generating the write command and controlling data bus transceivers. The transceiver direction control signal DT/$\overline{R}$ is conditioned to transmit at the end of each read cycle and does not change during a write cycle.

This allows the transceiver enable signal $\overline{DEN}$ to be active early in the cycle (while addresses are valid) without corrupting the address on the multiplexed bus. The write data and write command are both enabled from the leading edge of T2. Comparing minimum $\overline{WR}$ active delay TCVCTVmin with the maximum write data delay TCLDV indicates that write data may be not valid until 100 ns after write is active. The devices in the system should capture data on the trailing edge of the write command rather than the leading edge to guarantee valid data. The data from the 8086 is valid a minimum of 2TCLCL – TCLDVmax + TCVCTXmin = 300 ns before the trailing edge of write. The minimum write pulse width is TWLWH = 2TCLCL – 60 ns = 340 ns. The CPU maintains valid write data TWHDX ns after write. The TWHDZ specification overrides the result derived by relating TCLCHmin and TCHDZmin which implies write data may only be valid 18 ns after $\overline{WR}$. The 8086 floats the bus after write only if being forced off the bus by a HOLD or $\overline{RQ}$ input. Otherwise, the CPU simply switches the output drivers from data to address at the beginning of the next bus cycle. As with the read cycle, the next bus cycle may start in the clock cycle following T4 or any clock cycle later.

$\overline{DEN}$ is disabled a minimum of TCLCHmin + TCVCTXmin – TCVCTXmax = 18 ns after write to guarantee data hold time to the selected device. Since we are again evaluating a minimum TCVCTX with a maximum TCVCTX, the real minimum delay from the end of write to transceiver disable is approximately 60 ns.

4. Interrupt Acknowledge Timing

The interrupt acknowledge sequence consists of two interrupt acknowledge bus cycles as previously described. The detailed timing of each cycle is identical to the read cycle timing with two exceptions: command timing and address/data bus timing.

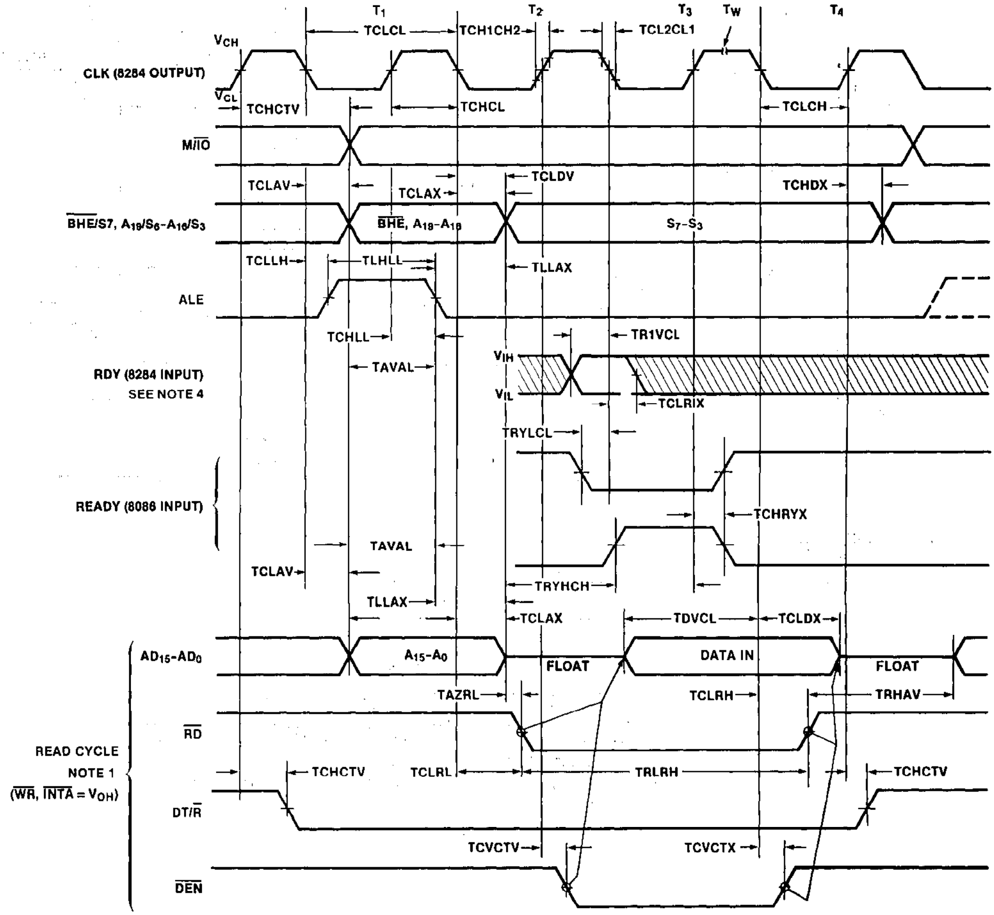
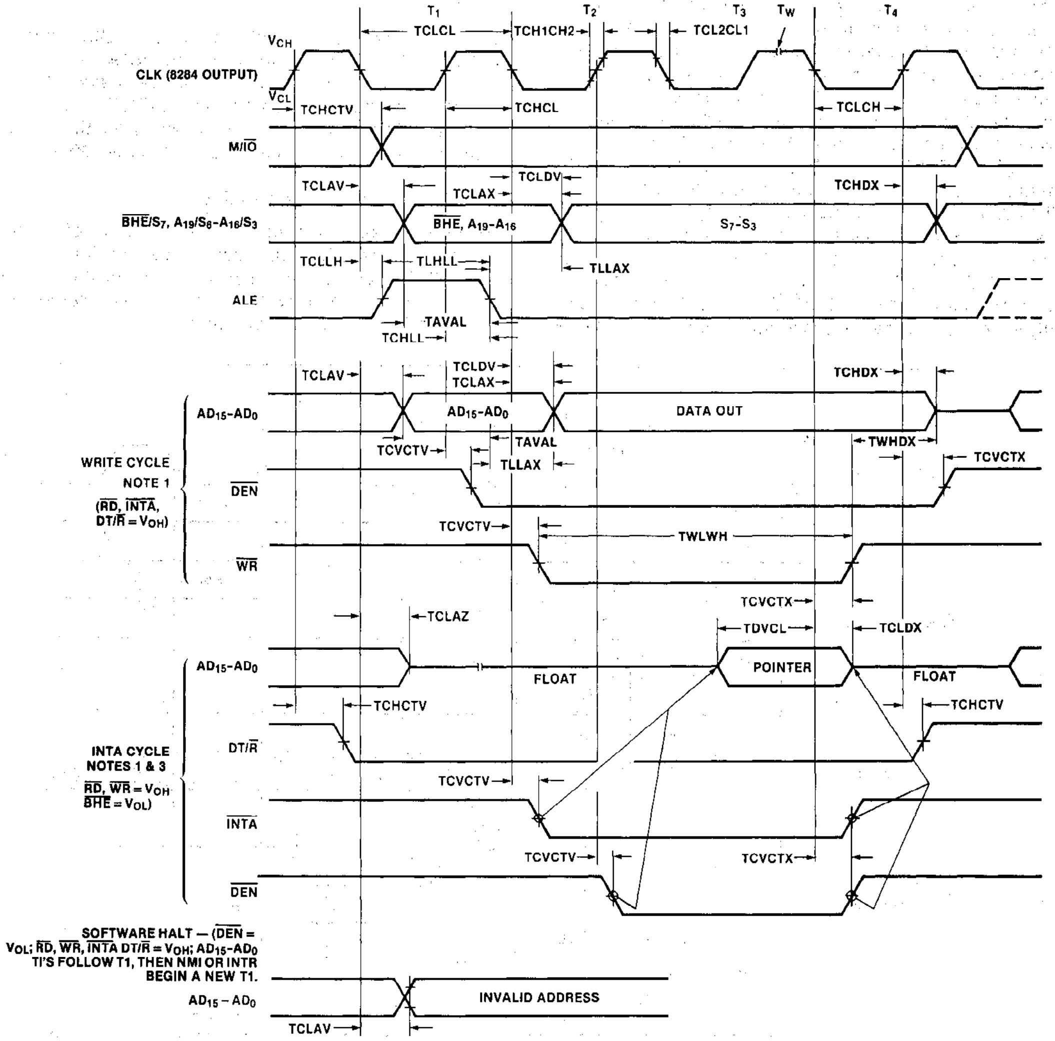


Figure 3F1. 8086 Bus Timing — Minimum Mode System



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OI} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. BOTH INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING THE SECOND INTA CYCLE. CONTROL SIGNALS SHOWN FOR SECOND INTA CYCLE.
4. SIGNALS AT 8284 ARE SHOWN FOR REFERENCE ONLY.
5. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

Figure 3F1. 8086 Bus Timing — Minimum Mode System (Con't)

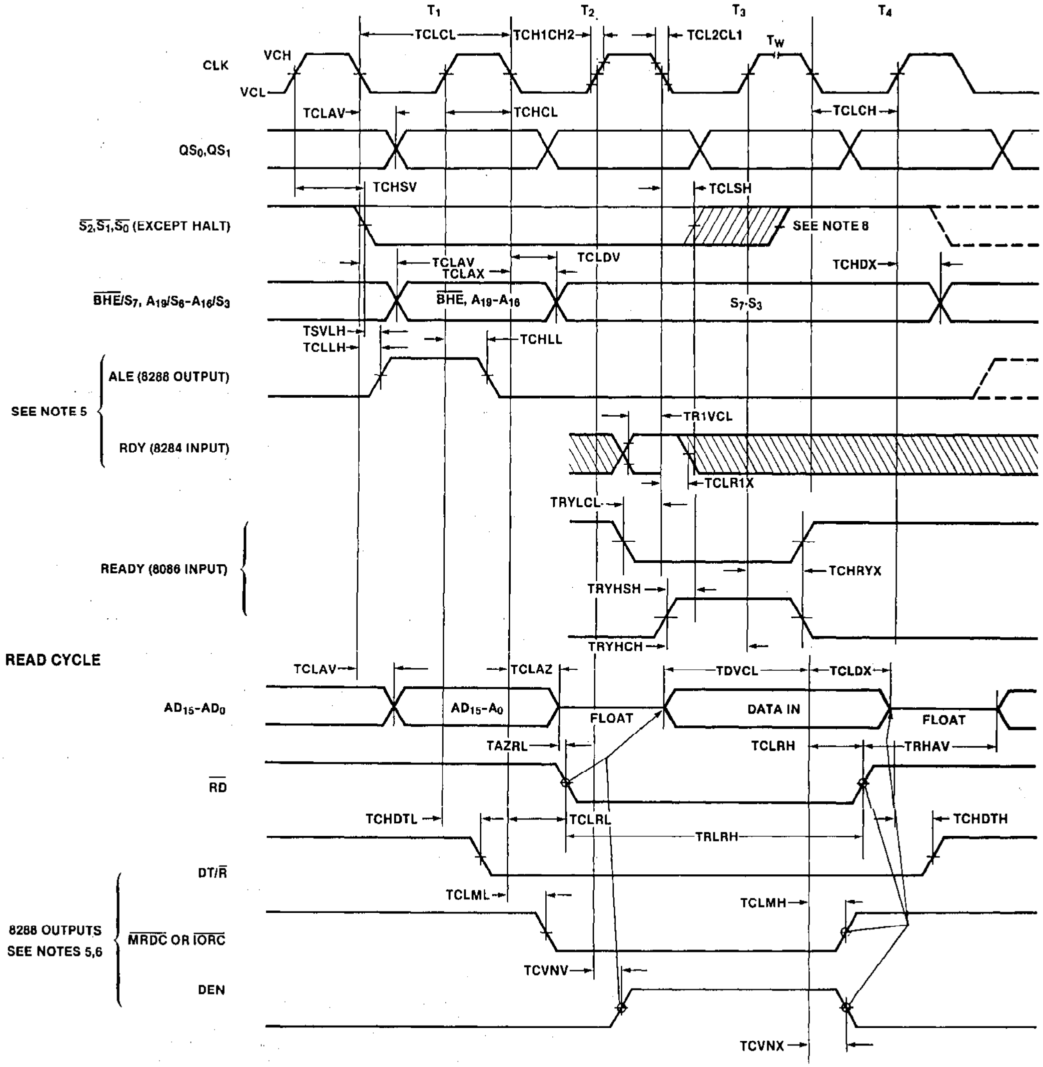
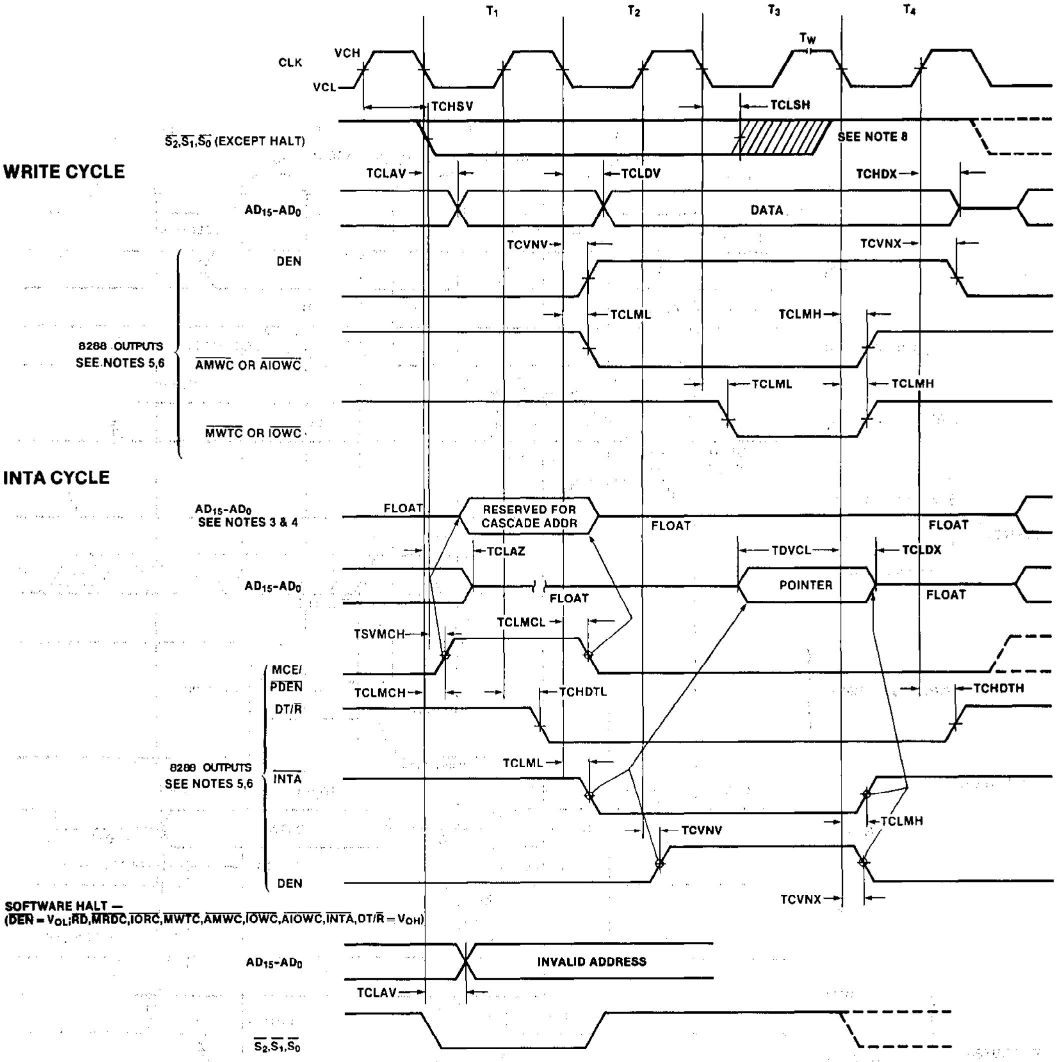


Figure 3F2a. 8086 Bus Timing — Maximum Mode System (Using 8288)



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLES.
4. BOTH INTA CYCLES RUN BACK-TO-BACK. THE 8086 LOCAL ADDR/DATA BUS IS FLOATING DURING THE SECOND INTA CYCLE. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

Figure 3F2b. 8086 Bus Timing — Maximum Mode System (Using 8288) (Con't)

The multiplexed address/data bus floats from the beginning (T1) of the $\overline{\text{INTA}}$ cycle (within TCLAZ ns). The upper four multiplexed address/status lines do not three-state. The address value on A19-A16 is indeterminate but the status information will be valid (S3 = 0, S4 = 0, S5 = IF, S6 = 0, S7 = $\overline{\text{BHE}}$ = 0). The multiplexed address/data lines will remain in three-state until the cycle after T4 of the $\overline{\text{INTA}}$ cycle. This sequence occurs for each of the $\overline{\text{INTA}}$ bus cycles. The interrupt type number read by the 8086 on the second $\overline{\text{INTA}}$ bus cycle must satisfy the same setup and hold times required for data during a read cycle.

The $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ signals are enabled for each $\overline{\text{INTA}}$ cycle and do not remain active between the two cycles. Their timing for each cycle is identical to the read cycle.

The $\overline{\text{INTA}}$ command has the same timing as the write command. It is active within 110 ns of the start of T2 providing 260 ns of access time from command to data valid at the 8086. The command is active a minimum of TCVCTXmin = 10 ns into T4 to satisfy the data hold time of the 8086. This provides minimum $\overline{\text{INTA}}$ pulse width of 300 ns, however taking signal delay tracking into consideration gives a minimum pulse width of 340 ns. Since the maximum inactive delay of $\overline{\text{INTA}}$ is TCVCTXmax = 110 ns and the CPU will not drive the bus until 15 ns (TCLAVmin) into the next clock cycle, 105 ns are available for interrupt devices on the local bus to float their outputs. If the data bus is buffered, $\overline{\text{DEN}}$ provides the same amount of time for local bus transceivers to three-state their outputs.

5. Ready Timing

The detailed timing requirements of the 8086 ready signal and the system ready signal into the 8284 are described in Section 3D. The system ready signal is typically generated from either the address decode of the selected device or the address decode and the command ($\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{INTA}}$). For a system which is normally not ready, the time to generate ready from a valid address and not insert a wait state, is 2TCLCL − TCLAVmax − TR1VCLmax = 255 ns. This time is available for buffer delays and address decoding to determine if the selected device does not require a wait state and drive the RDY line high. If wait cycles are required, the user hardware must provide the appropriate ready delay. Since the address will not change until the next ALE, the RDY will remain valid throughout the cycle. If the system is normally ready, selected devices requiring wait states also have 255 ns to disable the RDY line. The user circuitry must delay re-enabling RDY by the appropriate number of wait states.

If the $\overline{\text{RD}}$ command is used to enable the RDY signal, TCLCL − TCLRLmax − TRIVCLmax = 15 ns are available for external logic. If the $\overline{\text{WR}}$ command is used, TCLCL − TCVCTVmax − TRIVCLmax = 55 ns are available. Comparison of RDY control by address or command indicates that address decoding provides the best timing. If the system is normally not ready, address decode alone could be used to provide RDY for devices not requiring wait states while devices requiring wait states may use a combination of address decode and command to activate a wait state generator. If the system is normally ready, devices not requiring wait states do nothing to RDY while devices needing wait states should disable RDY via the address decode and use a combination of address decode and command to activate a delay to re-enable RDY.

If the system requires no wait states for memory and a fixed number of wait states for $\overline{\text{RD}}$ and $\overline{\text{WR}}$ to all I/O devices, the M/$\overline{\text{IO}}$ signal can be used as an early indication of the need for wait cycles. This allows a common circuit to control ready timing for the entire system without feedback of address decodes.

6. Other Considerations

Detailed HOLD/HLDA timing is covered in the next section and is not examined here. One last signal consideration needs to be mentioned for the minimum mode system. The $\overline{\text{TEST}}$ input is sampled by the 8086 only during execution of the WAIT instruction. The $\overline{\text{TEST}}$ signal should be active for a minimum of 6 clock cycles during the WAIT instruction to guarantee detection.

B. MAXIMUM MODE BUS TIMING

The maximum mode 8086 bus operations are logically equivalent to the minimum mode operation. Detailed timing analysis now involves signals generated by the CPU and the 8288 bus controller. The 8288 also provides additional control and command signals which expand the flexibility of the system.

1. ADDRESS and ALE

In the maximum mode, the address information continues to come from the CPU while the ALE strobe is generated by the 8288. To determine the worst case relationships between ALE and the address, we first must determine 8288 ALE activation relative to the $\overline{\text{S0}}$-$\overline{\text{S2}}$ status from the CPU. The maximum mode timing diagram specifies two possible delay paths to generate ALE. The first is TCHSV + TSVLH measured from the rising edge of the clock cycle preceding T1. The second path is TCLLH measured from the start of T1. Since the 8288 initiates a bus cycle from the status lines leaving the passive state ($\overline{\text{S0}}$-$\overline{\text{S2}}$ = 1), if the 8086 is late in issuing the status (TCHSVmax) while the clock high time is a minimum (TCHCLmin), the status will not have changed by the start of T1 and ALE is issued TSVLH ns after the status changes. If the status changes prior to the beginning of T1, the 8288 will not issue the ALE until TCLLH ns after the start of T1. The resulting worst case delay to enable ALE (relative to the start of T1) is TCHSVmax + TSVLHmax − TCHCLmin = 58 ns. Note, when calculating signal relationships, be sure to use the proper maximum mode values rather than equivalent minimum mode values.

The trailing edge of ALE is triggered in the 8288 by the positive clock edge in T1 regardless of the delay to enable ALE. The resulting minimum ALE pulse width is TCLCHmax − 58 ns = 75 ns assuming TCHLL = 0. TCLCHmax must be used since TCHCLmin was assumed to derive the 58 ns ALE enable delay. The address is guaranteed to be valid TCLCHmin + TCHLLmin − TCLAVmax = 8 ns prior to the trailing edge of ALE to capture the address in the 8282 or 8283 latches. Again we have assumed a very conservative TCHLL = 0. Note, since the address and ALE are driven by separate devices, no tracking of A.C. characteristics can be assumed.

The address hold time to the latches is guaranteed by the address remaining valid until the end of T1 while ALE is disabled a maximum of 15 ns from the positive clock transition in T1 (TCHCLmin – TCHLLmax = 52 ns address hold time). The multiplexed bus transitions from address to status and write data or three-state (for read) are identical to the minimum mode timing. Also, since the address valid delay (TCLAV) remains the critical path in establishing a valid address, the address access times to valid data and ready are the same as the minimum mode system.

2. Read Cycle Timing

The maximum mode system offers read signals generated by both the 8086 and the 8288. The 8086 $\overline{\text{RD}}$ output signal timing is identical to the minimum mode system. Since the A.C. characteristics of the read commands generated by the 8288 are significantly better than the 8086 output, access to devices on the demultiplexed buffered system bus should use the 8288 commands. The 8086 $\overline{\text{RD}}$ signal is available for devices which reside directly on the multiplexed bus. The following evaluations for read, write and interrupt acknowledge only consider the 8288 command timing.

The 8288 provides separate memory and I/O read signals which conform to the same A.C. characteristics. The commands are issued TCLML ns after the start of T2 and terminate TCLMH ns after the start of T4. The minimum command length is 2TCLCL – TCLMLmax + TCLMLmin = 375 ns. The access time to valid data at the CPU is 2TCLCL – TCLMLmax – TDVCLmax = 335 ns. Since the 8288 was designed for systems with buffered data busses, the commands are enabled before the CPU has three-stated the multiplexed bus and should not be used with devices which reside directly on the multiplexed bus (to do so could result in bus contention during 8086 bus float and device turn-on).

The direction control for data bus transceivers is established in T1 while the transceivers are not enabled by DEN until the positive clock transition of T2. This provides TCLCH + TCVNVmin = 123 ns for 8086 bus float delay and TCHCLmin + TCLCL – TCVNVmax – TDVCLmax = 187 ns of transceiver active to data valid at the CPU. Since both DEN and command are valid a minimum of 10 ns into T4, the CPU data hold time TCLDX is guaranteed. A maximum DEN disable of 45 ns (TCVNX max) guarantees the transceivers are disabled by the start of the next 8086 bus cycle (215 ns minimum from the same clock edge). On the positive clock transition of T4, DT/$\overline{\text{R}}$ is returned to transmit in preparation for a possible write operation on the next bus cycle. Since the system memory and I/O devices reside on a buffered system bus, they must three-state their outputs before the device for the next bus cycle is selected (approximately 2TCLCL) or the transceivers drive write data onto the bus (approximately 2TCLCL).

3. Write Cycle Timing

In the maximum mode, the 8288 provides normal and advanced write commands for memory and I/O. The advanced write commands are active a full clock cycle ahead of the normal write commands and have timing identical to the read commands. The advanced write pulse width is 2TCLCL – TCLMLmax + TCLMHmin = 375 ns while the normal write pulse width is TCLCL – TCLMLmax + TCLMHmin = 175 ns. Write data setup time to the selected device is a function of either the data valid delay from the 8086 (TCLDV) or the transceiver enable delay TCVNV. The worst case delay to valid write data is TCLDV = 110 ns minus transceiver propagation delays. This implies the data may not be valid until 100 ns after the advanced write command but will be valid approximately TCLCL – TCLDVmax + TCLMLmin = 100 ns prior to the leading edge of the normal write command. Data will be valid 2TCLCL – TCLDVmax + TCLMHmin = 300 ns before the trailing edge of either write command. The data and command overlap for the advanced command is 300 ns while the overlap with the normal write command is 175 ns. The transceivers are disabled a minimum of TCLCHmin – TCLMHmax + TCVNXmin = 85 ns after the write command while the CPU provides valid data a minimum of TCLCHmin – TCLMHmax + TCHDZmin = 85 ns. This guarantees write data hold of 85 ns after the write command. The transceivers are disabled TCLCL – TCVNXmax + TCHDTLmin = 155 ns (assuming TCHDTL = 0) prior to transceiver direction change for a subsequent read cycle.

4. Interrupt Acknowledge Timing

The maximum mode $\overline{\text{INTA}}$ sequence is logically identical to the minimum mode sequence. The transceiver control (DEN and DT/$\overline{\text{R}}$) and $\overline{\text{INTA}}$ command timing of each interrupt acknowledge cycle is identical to the read cycle. As in the minimum mode system, the multiplexed address/data bus will float from the leading edge of T1 for each $\overline{\text{INTA}}$ bus cycle and not be driven by the CPU until after T4 of each $\overline{\text{INTA}}$ cycle. The setup and hold times on the vector number for the second cycle are the same as data setup and hold for the read. If the device providing the interrupt vector number is connected to the local bus, TCLCL – TCLAZmax + TCLMLmin = 130 ns are available from 8086 bus float to $\overline{\text{INTA}}$ command active. The selected device on the local bus must disable the system data bus transceivers since DEN is still generated by the 8288.

If the 8288 is not in the IOB (I/O Bus) mode, the 8288 MCE/$\overline{\text{PDEN}}$ output becomes the MCE output. This output is active during each $\overline{\text{INTA}}$ cycle and overlaps the ALE signal during T1. The MCE is available for gating cascade addresses from a master 8259A onto three of the upper AD15-AD8 lines and allowing ALE to latch the cascade address into the address latches. The address lines may then be used to provide CAS address selection to slave 8259A's located on the system bus (reference Figure 3E5). MCE is active within 15 ns of status or the start of T1 for each $\overline{\text{INTA}}$ cycle. MCE should not enable the CAS lines onto the multiplexed bus during the first cycle since the CPU does not guarantee to float the bus until 80 ns into the first $\overline{\text{INTA}}$ cycle. The first MCE can be inhibited by gating MCE with $\overline{\text{LOCK}}$. The 8086 $\overline{\text{LOCK}}$ output is activated during T2 of the first cycle and disabled during T2 of the second cycle. The overlap of $\overline{\text{LOCK}}$ with MCE allows the first MCE to be masked and the second MCE to gate the cascade address onto the local bus. Since the 8259A will not provide a cascade address until the second cycle, no information is lost. As with ALE, MCE is guaranteed valid within 58 ns of the start of T1 to allow 75 ns CAS address setup to the trailing edge of ALE. MCE remains active TCHCLmin – TCHLLmax + TCLMCLmin = 52 ns after ALE to provide data hold time to the latches.

If the 8288 is strapped in the IOB mode, the MCE output becomes $\overline{\text{PDEN}}$ and all I/O references are assumed to be devices on the local bus rather than the demultiplexed system bus. Since $\overline{\text{INTA}}$ cycles are considered I/O cycles, all interrupts are assumed to come from the local system and cascade addresses are not gated onto the system address bus. Additionally, the DEN signal is not enabled since no I/O transfers occur on the system bus. If the local I/O bus is also buffered by transceivers, the $\overline{\text{PDEN}}$ signal is used to enable those transceivers. $\overline{\text{PDEN}}$ A.C. characteristics are identical to DEN with $\overline{\text{PDEN}}$ enabled for I/O references and DEN enabled for instruction or memory data references.

5. Ready Timing

Ready timing based on address valid timing is the same for maximum and minimum mode systems. The delay from 8288 command valid to RDY valid at the 8284 is TCLCL – TCLMLmax – TRIVCLmin = 130 ns. This time is available for external circuits to determine the need to insert wait states and disable RDY or enable RDY to avoid wait states. $\overline{\text{INTA}}$, all read commands and advanced write commands provide this timing. The normal write command is not valid until after the RDY signal must be valid. Since both normal and advanced write commands are generated by the 8288 for all write cycles, the advanced write may be used to generate a RDY indication even though the selected device uses the normal write command.

Since separate commands are provided for memory and I/O, no M/$\overline{\text{IO}}$ signal is specifically available as in the minimum mode to allow an early 'wait state required' indication for I/O devices. The $\overline{\text{S2}}$ status line, however is logically equivalent to the M/$\overline{\text{IO}}$ signal and can be used for this purpose.

6. Other Considerations

The $\overline{\text{RQ}}$/$\overline{\text{GT}}$ timing is covered in the next section and will not be duplicated here. The only additional signals to be considered in the maximum mode are the queue status lines QS0, QS1. These signals are changed on the leading edge of each clock cycle (high to low transition) including idle and wait cycles (the queue status is independent of the bus activity). External logic may sample the lines on the low to high transition of each clock cycle. When sampled, the signals indicate the queue activity in the previous clock cycle and therefore lag the CPU's activity by one cycle. The $\overline{\text{TEST}}$ input requirements are identical to those stated for the minimum mode.

To inform the 8288 of HALT status when a HALT instruction is executed, the 8086 will initiate a status transition from passive to HALT status. The status change will cause the 8288 to emit an ALE pulse with an indeterminate address. Since no bus cycle is initiated (no command is issued), the results of this address will not affect CPU operation (i.e., no response such as READY is expected from the system). This allows external hardware to latch and decode all transitions in system status.

3G. Bus Control Transfer (HOLD/HLDA and $\overline{\text{RQ}}$/$\overline{\text{GT}}$)

The 8086 supports protocols for transferring control of the local bus between itself and other devices capable of acting as bus masters. The minimum mode configuration offers a signal level handshake similar to the 8080 and 8085 systems. The maximum mode provides an enhanced pulse sequence protocol designed to optimize utilization of CPU pins while extending the system configurations to two prioritized levels of alternate bus masters. These protocols are simply techniques for arbitration of control of the CPU's local bus and should not be confused with the need for arbitration of a system bus.

1. MINIMUM MODE

The minimum mode 8086 system uses a hold request input (HOLD) to the CPU and a hold acknowledge (HLDA) output from the CPU. To gain control of the bus, a device must assert HOLD to the CPU and wait for the HLDA before driving the bus. When the 8086 can relinquish the bus, it floats the $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{INTA}}$ and M/$\overline{\text{IO}}$ command lines, the $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ bus control lines and the multiplexed address/data/status lines. The ALE signal is not three-stated. The CPU acknowledges the request with HLDA to allow the requestor to take control of the bus. The requestor must maintain the HOLD request active until it no longer requires the bus. The HOLD request to the 8086 directly affects the bus interface unit and only indirectly affects the execution unit. The CPU will continue to execute from its internal queue until either more instructions are needed or an operand transfer is required. This allows a high degree of overlap between CPU and auxiliary bus master operation. When the requestor drops the HOLD signal, the 8086 will respond by dropping HLDA. The CPU will not re-drive the bus, command and control signals from three-state until it needs to perform a bus transfer. Since the 8086 may still be executing from its internal queue when HOLD drops, there may exist a period of time during which no device is driving the bus. To prevent the command lines from drifting below the minimum VIH level during the transition of bus control, 22K ohm pull up resistors should be connected to the bus command lines. The timing diagram in Figure 3G1 shows the handshake sequence and 8086 timing to sample HOLD, float the bus, and enable/disable HLDA relative to the CPU clock.

To guarantee valid system operation, the designer must assure that the requesting device does not assert control of the bus prior to the 8086 relinquishing control and that the device relinquishes control of the bus prior to the 8086 driving the bus. The HOLD request into the 8086 must be stable THVCH ns prior to the CPU's low to high clock transition. Since this input is not synchronized by the CPU, signals driving the HOLD input should be synchronized with the CPU clock to guarantee the setup time is not violated. Either clock edge may be used. The maximum delay between HLDA and the 8086 floating the bus is TCLAZmax − TCLHAVmin = 70 ns. If the system cannot tolerate the 70 ns overlap, HLDA active from the 8086 should be delayed to the device. The minimum delay for the CPU to drive the control bus from HOLD inactive is THVCHmin + 3TCLCL = 635 ns and THVCHmin + 3TCLCL + TCHCL = 701 ns to drive the multiplexed bus. If the device does not satisfy these requirements, HOLD inactive to the 8086 should be delayed. The delay from HLDA inactive to driving the busses is TCLCL + TCLCHmin − TCLHAVmax = 158 ns for the control bus and 2TCLCL − TCLHAVmax = 240 ns for the data bus.

1.1 Latency of HLDA to HOLD

The decision to respond to a HOLD request is made in the bus interface unit. The major factors that influence the decision are the current bus activity, the state of the $\overline{\text{LOCK}}$ signal internal to the CPU (activated by the software LOCK prefix) and interrupts.

If the $\overline{\text{LOCK}}$ is not active, an interrupt acknowledge cycle is not in progress and the BIU (Bus Interface Unit) is executing a T4 or TI when the HOLD request is received, the minimum latency to HLDA is:

35 ns	THVCH min (Hold setup)
65 ns	TCHCL min
200 ns	TCLCL (bus float delay)
10 ns	TCLHAV min (HLDA delay)
310 ns	@ 5 MHz

The maximum delay under these conditions is:

34 ns	(just missed setup time)
200 ns	delay to next sample
82 ns	TCHCL max
200 ns	TCLCL (bus float delay)
160 ns	TCLHAV max (HLDA delay)
677 ns	@ 5 MHz

If the BIU just initiated a bus cycle when the HOLD Request was received, the worst case response time is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
7*200	bus cycle execution
N*200	N wait states/bus cycle
160 ns	TCLHAV max (HLDA delay)
1.676 μs	@ 5 MHz, no wait states

Note, the 200 ns delay for just missing is included in the delay for bus cycle execution. If the operand transfer is a word transfer to an odd byte boundary, two bus cycles are executed to perform the transfer. The BIU will not acknowledge a HOLD request between the two bus cycles. This type of transfer would extend the above maximum latency by four additional clocks plus N additional wait states. With no wait states in the bus cycle, the maximum would be 2.476 microseconds.

Although the minimum mode 8086 does not have a hardware $\overline{\text{LOCK}}$ output, the software LOCK prefix may still be included in the instruction stream. The CPU internally reacts to the LOCK prefix as would the maximum mode 8086. Therefore, the LOCK does not allow a HOLD request to be honored until completion of the instruction following the prefix. This allows an instruction which performs more than one memory reference (ex. ADD [BX], CX; which adds CX to [BX]) to execute without another bus master gaining control of the bus between memory references. Since the LOCK signal is active for one clock longer than the instruction execution, the maximum latency to HLDA is:

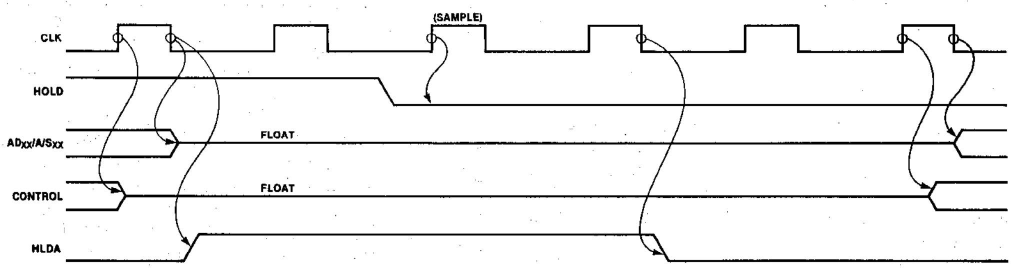


Figure 3G1. HOLD/HLDA Sequence

34 ns	THVCH (just miss)
200 ns	delay to next sample
82 ns	TCHCL max
(M + 1)*200 ns	LOCK instruction execution
200 ns	set up HLDA (internal)
160 ns	TCLHAV max (HLDA delay)
(M*200 ns) + 876 ns	@ 5 MHz

If the HOLD request is made at the beginning of an interrupt acknowledge sequence, the maximum latency to HLDA is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
2600 ns	13 clock cycles for INTA
160 ns	TCLHAV max
2.876 μs	@ 5 MHz

1.2 Minimum Mode DMA Configuration

A typical use of the HOLD/HLDA signals in the minimum mode 8086 system is bus control exchange with DMA devices like the Intel 8257-5 or 8237 DMA controllers. Figure 3G2 gives a general interconnect for this type of configuration using the 8237-2. The DMA controller resides on the upper half of the 8086's local bus and shares the A8-A15 demultiplexing address latch of the 8086. All registers in the 8237-2 must be assigned odd addresses to allow initialization and interrogation by the CPU over the upper half of the data bus. The 8086 RD/WR commands must be demultiplexed to provide separate I/O and memory commands which are compatible with the 8237-2 commands. The AEN control from the 8237-2 must disable the 8086 commands from the command bus, disable the address latches from the lower (A0-A7) and upper (A19-A16) address bus and select the 8237-2 address strobe (ADSTB) to the A8-A15 address latch. If the data bus is buffered, a pull-up resistor on the DEN line will keep the buffers disabled. The DMA controller will only transfer bytes between

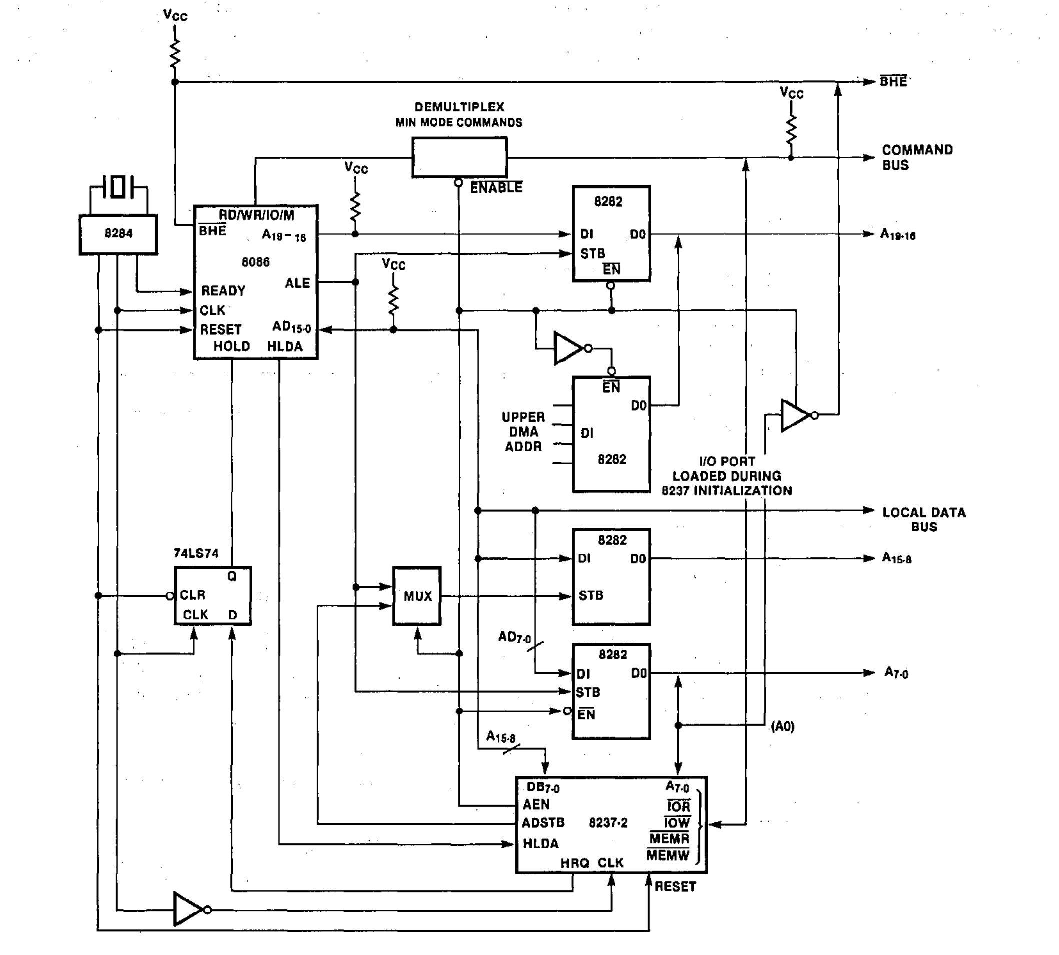


Figure 3G2. DMA Using the 8237-2

memory and I/O and requires the I/O devices to reside on an 8-bit bus derived from the 16-bit to 8-bit bus multiplex circuit given in Section 4. Address lines A7-A0 are driven directly by the 8237 and $\overline{\text{BHE}}$ is generated by inverting A0. If A19-A16 are used, they must be provided by an additional port with either a fixed value or initialized by software and enabled onto the address bus by AEN.

Figure 3G3 gives an interconnection for placing the 8257 on the system bus. By using a separate latch to hold the upper address from the 8257-5 and connecting the outputs to the address bus as shown, 16-bit DMA transfers are provided. In this configuration, AEN simultaneously enables A0 and $\overline{\text{BHE}}$ to allow word transfers. AEN still disables the CPU interface to the command and address busses.

2. MAXIMUM MODE ($\overline{\text{RQ}}/\overline{\text{GT}}$)

The maximum mode 8086 configuration supports a significantly different protocol for transferring bus control. When viewed with respect to the HOLD/HLDA sequence of the minimum mode, the protocol appears difficult to implement externally. However, it is necessary to understand the intent of the protocol and its purpose within the system architecture.

2.1 Shared System Bus ($\overline{\text{RQ}}/\overline{\text{GT}}$ Alternative)

The maximum mode $\overline{\text{RQ}}/\overline{\text{GT}}$ sequence is intended to transfer control of the CPU local bus between the CPU and alternate bus masters which reside totally on the local bus and share the complete CPU interface to the system bus. The complete interface includes the address latches, data transceivers, 8288 bus controller and 8289 multi master bus arbiter. If the alternate bus masters in the system do not reside directly on the 8086 local bus, system bus arbitration is required rather than local CPU bus arbitration. To satisfy the need for multimaster system bus arbitration at each CPU's system interface, the 8289 bus arbiter should be used rather than the CPU $\overline{\text{RQ}}/\overline{\text{GT}}$ logic.

To allow a device with a simple HOLD/HLDA protocol to gain control of a single CPU system bus, the circuit in Figure 3G4 could be used. The design is effectively a simple bus arbiter which isolates the CPU from the system bus when an alternate bus master issues a HOLD request. The output of the circuit, $\overline{\text{AEN}}$ (Address ENable), disables the 8288 and 8284 when the 8086 indicates idle status ($\overline{\text{S0}}$,$\overline{\text{S1}}$,$\overline{\text{S2}}$ = 1), $\overline{\text{LOCK}}$ is not active and a HOLD request is active. With $\overline{\text{AEN}}$ inactive, the 8288 three-states the command outputs and disables DEN

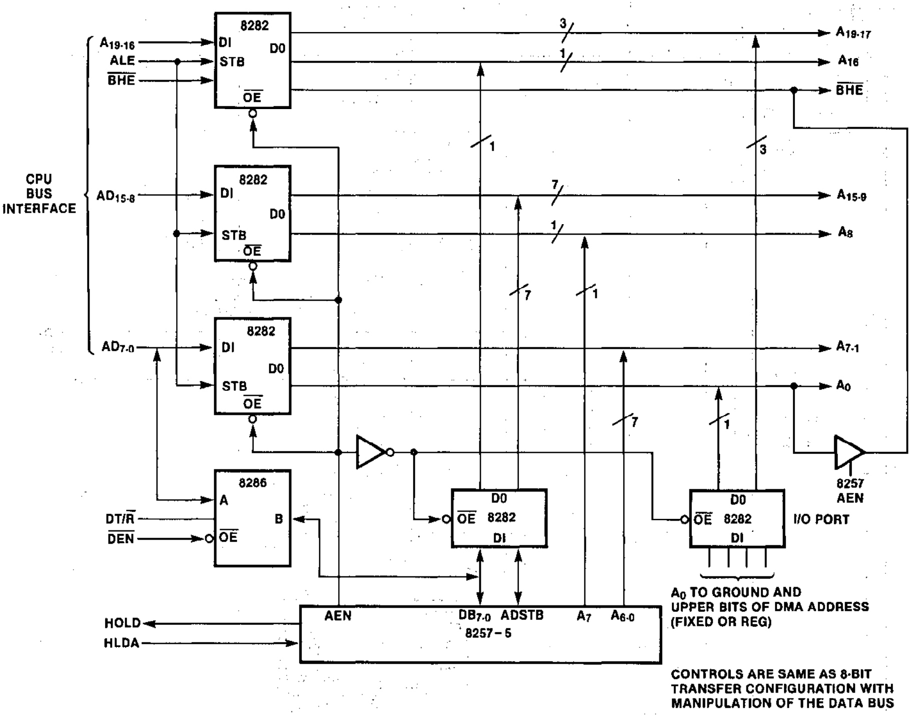


Figure 3G3. 8086 Min System, 8257 on System Bus 16-Bit Transfers which three-states the data bus transceivers. $\overline{\text{AEN}}$ must also three-state the address latch (8282 or 8283) outputs. These actions remove the 8086 from the system bus and allow the requesting device to drive the system bus. The $\overline{\text{AEN}}$ signal to the 8284 disables the ready input and forces a bus cycle initiated by the 8086 to wait until the 8086 regains control of the system bus. The CPU may actively drive its local bus during this interval.

The requesting device will not gain control of the bus during an 8086 initiated bus cycle, a locked instruction or an interrupt acknowledge cycle. The $\overline{\text{LOCK}}$ signal from the 8086 is active between $\overline{\text{INTA}}$ cycles to guarantee the CPU maintains control of the bus. Unlike the minimum mode 8086 HOLD response, this arbitration circuit allows the requestor to gain control of the bus between consecutive bus cycles which transfer a word operand on an odd address boundary and are not locked. Depending on the characteristics of the requesting device, any of the 74LS74 outputs can be used to generate a HLDA to the device.

Upon completion of its bus operations, the alternate bus master must relinquish control of the system bus and drop the HOLD request. After $\overline{\text{AEN}}$ goes inactive, the address latches and data transceivers are enabled but, if a CPU initiated bus cycle is pending, the 8288 will not drive the command bus until a minimum of 105 ns or maximum of 275 ns later. If the system is normally not ready, the 8284 $\overline{\text{AEN}}$ input may immediately be enabled with ready returning to the CPU when the selected device completes the transfer. If the system is normally ready, the 8284 $\overline{\text{AEN}}$ input must be delayed long enough to provide access time equivalent to a normal bus cycle. The 74LS74 latches in the design provide a minimum of TCLCHmin for the alternate device to float the system bus after releasing HOLD. They also provide 2TCLCL ns address access and 2TCLCL − TAEVCHmax ns (8288 command enable delay) command access prior to enabling 8284 ready detection. If HLDA is generated as shown in Figure 3G4, TCLCL ns are available for the 8086 to release the bus prior to issuing HLDA while HLDA is dropped almost immediately upon loss of HOLD.

A circuit configuration for an 8257-5 using this technique to interface with a maximum mode 8086 can be derived from Figure 3G3. The 8257-5 has its own address latch for buffering the address lines A15-A8 and uses its $\overline{\text{AEN}}$ output to enable the latch onto the address bus. The maximum latency from HOLD to HLDA for this circuit is dependent on the state of the system when the HOLD is issued. For an idle system the maximum delay is the propagation delay through the nand gate and R/S flip-flop (TD1) plus 2TCLCL plus TCLCHmax plus propagation delay of the 74LS74 and 74LS02 (TD2). For a locked instruction it becomes: TD1 + TD2 + (M + 2) *TCLCL + TCLCHmax where M is the number of clocks required for execution of the locked instruction. For the interrupt acknowledge cycle the latency is TD1 + TD2 + 9 *TCLCL + TCLCHmax.

2.2 Shared Local Bus ($\overline{\text{RQ}}/\overline{\text{GT}}$ Usage)

The $\overline{\text{RQ}}/\overline{\text{GT}}$ protocol was developed to allow up to two instruction set extension processors (co-processors) or other special function processors (like the 8089 I/O processor in local mode) to reside directly on the 8086 local bus. Each $\overline{\text{RQ}}/\overline{\text{GT}}$ pin of the 8086 supports the full protocol for exchange of bus control (Fig. 3G5). The sequence consists of a request from the alternate bus master to gain control of the system bus, a grant from the CPU to indicate the bus has been relinquished and a release pulse from the alternate master when done. The two $\overline{\text{RQ}}/\overline{\text{GT}}$ pins ($\overline{\text{RQ}}/\overline{\text{GT0}}$ and $\overline{\text{RQ}}/\overline{\text{GT1}}$) are prioritized with $\overline{\text{RQ}}/\overline{\text{GT0}}$ having the highest priority. The prioritization only occurs if requests have been received on both pins before a response has been given to either. For example, if a request is received on $\overline{\text{RQ}}/\overline{\text{GT1}}$ followed by a request on $\overline{\text{RQ}}/\overline{\text{GT0}}$ prior to a grant on $\overline{\text{RQ}}/\overline{\text{GT1}}$, $\overline{\text{RQ}}/\overline{\text{GT0}}$ will gain priority over $\overline{\text{RQ}}/\overline{\text{GT1}}$. However, if $\overline{\text{RQ}}/\overline{\text{GT1}}$ had already received a grant, a request on $\overline{\text{RQ}}/\overline{\text{GT0}}$ must wait until a release pulse is received on $\overline{\text{RQ}}/\overline{\text{GT1}}$.

The request/grant sequence interaction with the bus interface unit is similar to HOLD/HLDA. The CPU continues to execute until a bus transfer for additional instructions or data is required. If the release pulse is

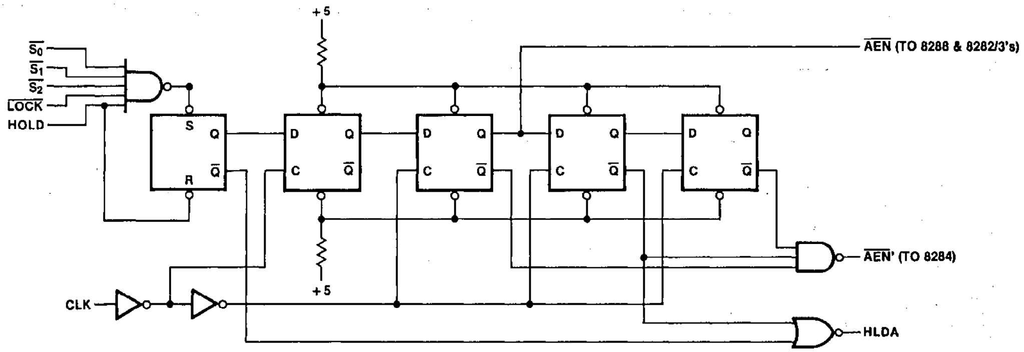


Figure 3G4. Circuit to Translate HOLD into AEN Disable for Max Mode 8086

received before the CPU needs the bus, it will not drive the bus until a transfer is required.

Upon receipt of a request pulse, the 8086 floats the multiplexed address, data and status bus, the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines, the $\overline{LOCK}$ pin and $\overline{RD}$. This action does not disable the 8288 command outputs from driving the command bus and does not disable the address latches from driving the address bus. The 8288 contains internal pull-up resistors on the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines to maintain the passive state while the 8086 outputs are three-state. The passive state prevents the 8288 from initiating any commands or activating DEN to enable the transceivers buffering the data bus. If the device issuing the $\overline{RQ}$ does not use the 8288, it must disable the 8288 command outputs by disabling the 8288 $\overline{AEN}$ input. Also, address latches not used by the requesting device must be disabled.

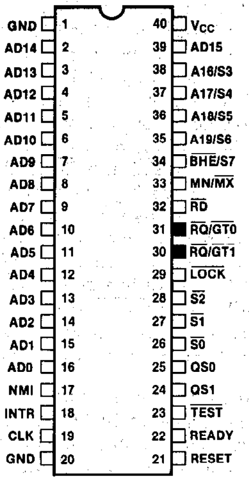


Figure 3G5. 8086 RQ/GT Connections

2.3 $\overline{RQ}/\overline{GT}$ Operation

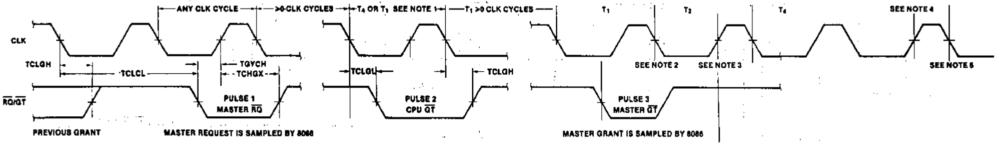
Detailed timing of the $\overline{RQ}/\overline{GT}$ sequence is given in Figure 3G6. To request a transfer of bus control via the $\overline{RQ}/\overline{GT}$ lines, the device must drive the line low for no more than one CPU clock interval to generate a request pulse. The pulse must be synchronized with the CPU clock to guarantee the appropriate setup and hold times to the clock edge which samples the $\overline{RQ}/\overline{GT}$ lines in the CPU. After issuing a request pulse, the device must begin sampling for a grant pulse with the next low to high clock edge. Since the 8086 can respond with a grant pulse in the clock cycle immediately following the request, the $\overline{RQ}/\overline{GT}$ line may not return to the positive level between the request and grant pulses. Therefore edge triggered logic is not valid for capturing a grant pulse. It also implies the circuitry which generates the request pulse must guarantee the request is removed in time to detect a grant from the CPU. After receiving the grant pulse, the requesting device may drive the local bus. Since the 8086 does not float the address and data bus, $\overline{LOCK}$ or $\overline{RD}$ until the high to low clock transition following the low to high clock transition the requestor uses to sample for the grant, the requestor should wait the float delay of the 8086 (TCLAZ) before driving the local bus. This precaution prevents bus contention during the access of bus control by the requestor.

To return control of the bus to the 8086, the alternate bus master relinquishes bus control and issues a release pulse on the same $\overline{RQ}/\overline{GT}$ line. The 8086 may drive the $\overline{S0}$-$\overline{S2}$ status lines, $\overline{RD}$ and $\overline{LOCK}$, three clock cycles after detecting the release pulse and the address/data bus TCHCLmin ns (clock high time) after the status lines. The alternate bus master should be three-stated off the local bus and have other 8086 interface circuits (8288 and address latches) re-enabled within the 8086 delay to regain control of the bus.

2.4 $\overline{RQ}/\overline{GT}$ Latency

The $\overline{RQ}$ to $\overline{GT}$ latency for a single $\overline{RQ}/\overline{GT}$ line is similar to the HOLD to HLDA latency. The cases given for the minimum mode 8086 also apply to the maximum mode. For each case the delay from $\overline{RQ}$ detection by the CPU to $\overline{GT}$ detection by the requestor is:

(HOLD to HLDA delay) − (THVCH + TCHCL + TCLHAV)



NOTES:
1. THE 8086 FLOATS A_XD_X BUS $\overline{RD}$ AND $\overline{LOCK}$ ON THIS EDGE
2. THE OTHER MASTER FLOATS $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ FROM 1.1.1 STATE ON THIS EDGE
3. THE OTHER MASTER FLOATS A_XD_X BUS, $\overline{BHE}$, AND $\overline{LOCK}$ ON THIS EDGE
4. THE 8086 REDRIVES THE CONTROL LINES
5. THE 8086 REDRIVES THE AD_{XX} LINES

Figure 3G6. Request/Grant Sequence

This gives a clock cycle maximum delay for an idle bus interface. All other cases are the minimum mode result minus 476 ns. If the 8086 has previously issued a grant on one of the $\overline{RQ}/\overline{GT}$ lines, a request on the other $\overline{RQ}/\overline{GT}$ line will not receive a grant until the first device releases the interface with a release pulse on its $\overline{RQ}/\overline{GT}$ line. The delay from release on one $\overline{RQ}/\overline{GT}$ line to a grant on the other is typically one clock period as shown in Figure 3G7. Occasionally the delay from a release on $\overline{RQ}/\overline{GT1}$ to a grant on $\overline{RQ}/\overline{GT0}$ will take two clock cycles and is a function of a pending request for transfer of control from the execution unit. The latency from request to grant when the interface is under control of a bus master on the other $\overline{RQ}/\overline{GT}$ line is a function of the other bus master. The protocol embodies no mechanism for the CPU to force an alternate bus master off the bus. A watchdog timer should be used to prevent an errant alternate bus master from 'hanging' the system.

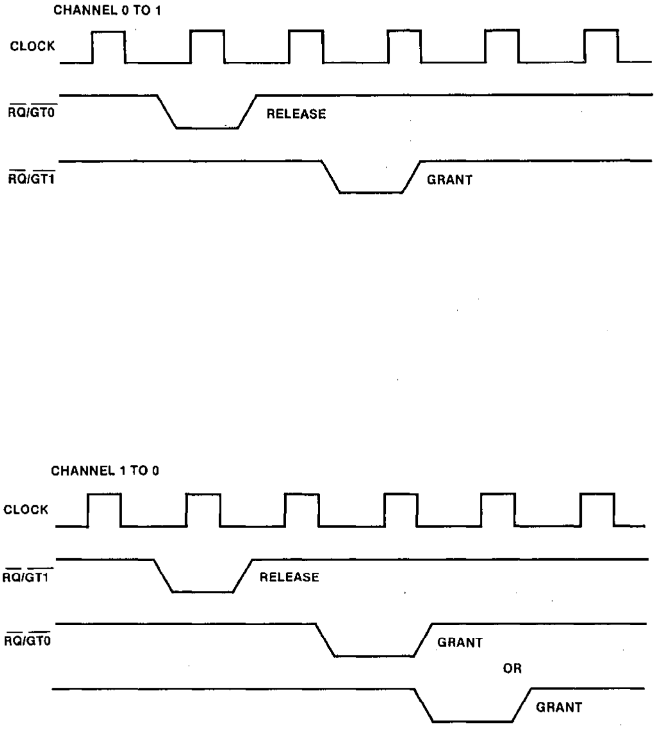


Figure 3G7. Channel Transfer Delay

2.5 RQ/GT to HOLD/HLDA Conversion

A circuit for translating a HOLD/HLDA hand-shake sequence into a $\overline{RQ}/\overline{GT}$ pulse sequence is given in Figure 3G8. After receiving the grant pulse, the HLDA is enabled TCHCLmin ns before the CPU has three-stated the bus. If the requesting circuit drives the bus within 20 ns of HLDA, it may be desirable to delay the acknowledge one clock period. The HLDA is dropped no later than one clock period after HOLD is disabled. The HLDA also drops at the beginning of the release pulse to provide 2TCLCL + TCLCH for the requestor to relinquish control of the status lines and 3TCLCL to float the remaining signals.

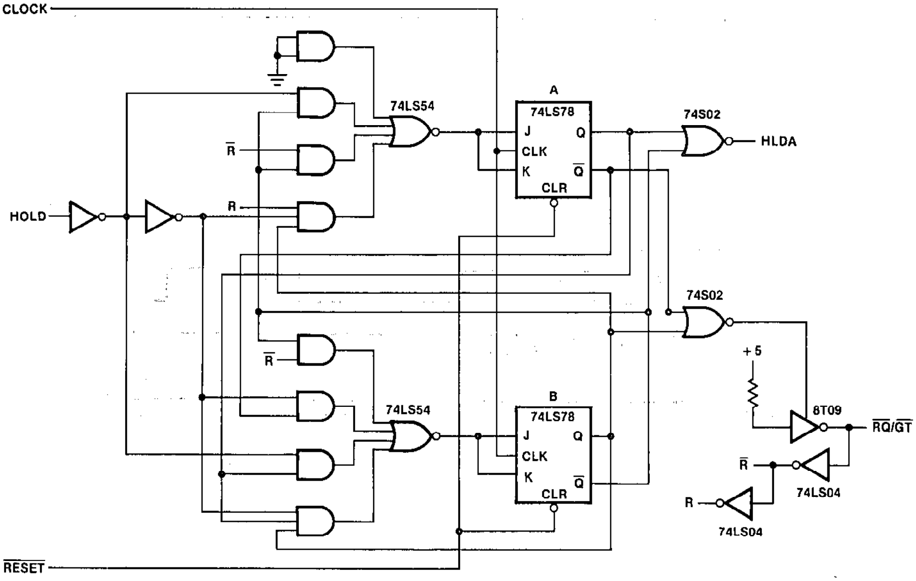


Figure 3G8a. HOLD/HLDA◀▶$\overline{RQ}/\overline{GT}$ Conversion Circuit

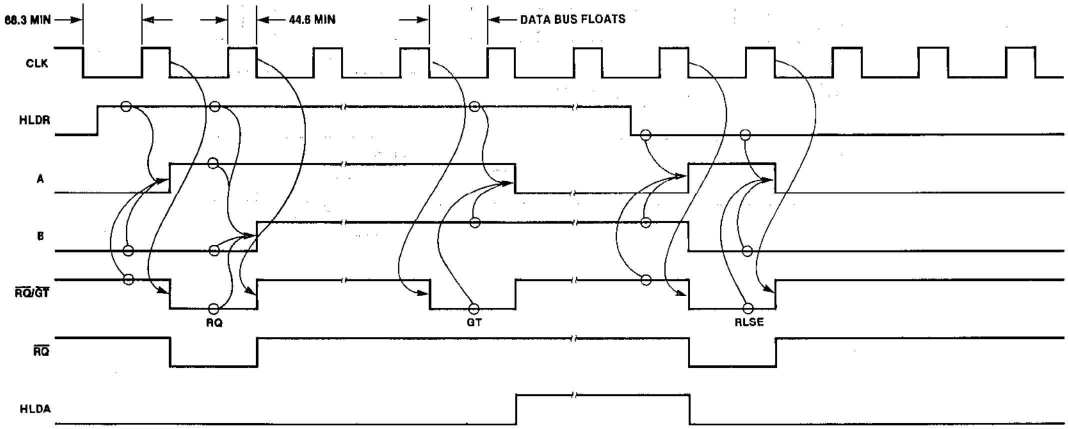


Figure 3G8b. HOLD/HLDA◀▶$\overline{RQ}/\overline{GT}$ Conversion Timing

4. INTERFACING WITH I/O

The 8086 is capable of interfacing with 8- and 16-bit I/O devices using either I/O instructions or memory mapped I/O. The I/O instructions allow the I/O devices to reside in a separate I/O address space while memory mapped I/O allows the full power of the instruction set to be used for I/O operations. Up to 64K bytes of I/O mapped I/O may be defined in an 8086 system. To the programmer, the separate I/O address space is only accessible with INPUT and OUTPUT commands which transfer data between I/O devices and the AX (for 16-bit data transfers) or AL (for 8-bit data transfers) register. The first 256 bytes of the I/O space (0 to 255) are directly addressable by the I/O instructions while the entire 64K is accessible via register indirect addressing through the DX register. The later technique is particularly desirable for service procedures that handle more than one device by allowing the desired device address to be passed to the procedure as a parameter. I/O devices may be connected to the local CPU bus or the buffered system bus.

4A. Eight-Bit I/O

Eight-bit I/O devices may be connected to either the upper or lower half of the data bus. Assigning an equal number of devices to the upper and lower halves of the bus will distribute the bus loading. If a device is connected to the upper half of the data bus, all I/O addresses assigned to the device must be odd (A0 = 1). If the device is on the lower half of the bus, its addresses must be even (A0 = 0). The address assignment directs the eight-bit transfer to the upper (odd byte address) or lower (even byte address) half of the sixteen-bit data bus. Since A0 will always be a one or zero for a specific device, A0 cannot be used as an address input to select registers within a specific device. If a device on the upper half of the bus and one on the lower half are assigned addresses that differ only in A0 (adjacent odd and even addresses), A0 and $\overline{BHE}$ must be conditions of chip select decode to prevent a write to one device from erroneously performing a write to the other. Several techniques for generating I/O device chip selects are given in Figure 4A1.

The first technique (a) uses separate 8205's to generate chip selects for odd and even addressed byte peripherals. If a word transfer is performed to an even addressed device, the adjacent odd addressed I/O device is also selected. This allows accessing the devices individually with byte transfers or simultaneously as a 16-bit device with word transfers. Figure 4A1(b) restricts the chip selects to byte transfers, however a word transfer to an odd address will cause the 8086 to run two byte transfers that the decode technique will not detect. The third technique simply uses a single 8205 to generate odd and even device selects for byte transfers and will only select the even addressed eight-bit device on a word transfer to an even address.

If greater than 256 bytes of the I/O space or memory mapped I/O is used, additional decoding beyond what is shown in the examples may be necessary. This can be done with additional TTL, 8205's or bipolar PROMs (Intel's 3605A). The bipolar PROMs are slightly slower than multiple levels of TTL (50 ns vs 30 to 40 ns for TTL) but provide full decoding in a single package and allow inserting a new PROM to reconfigure the system I/O map without circuit board or wiring modifications (Fig. 4A2).

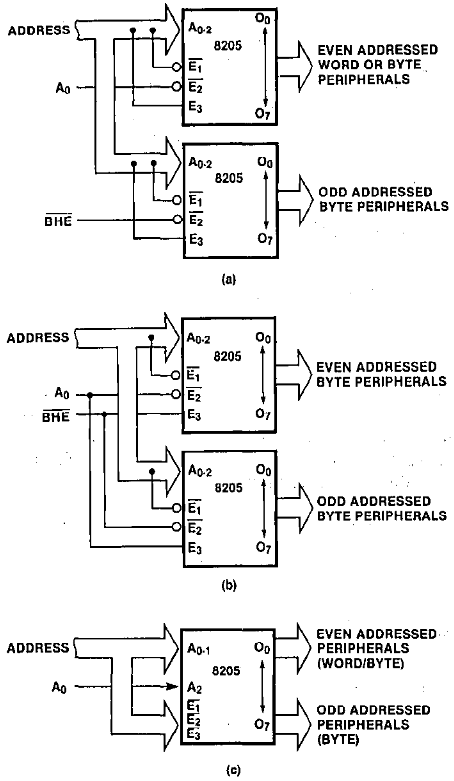

ADDRESS
A0
A0-2
8205
$\overline{E_1}$
$\overline{E_2}$
E3
O0
O7
EVEN ADDRESSED WORD OR BYTE PERIPHERALS
$\overline{BHE}$
ODD ADDRESSED BYTE PERIPHERALS
(a)
ADDRESS
A0
$\overline{BHE}$
A0-2
8205
EVEN ADDRESSED BYTE PERIPHERALS
ODD ADDRESSED BYTE PERIPHERALS
(b)
ADDRESS
A0
A0-1
A2
8205
$\overline{E_1}$
$\overline{E_2}$
E3
O0
O7
EVEN ADDRESSED PERIPHERALS (WORD/BYTE)
ODD ADDRESSED PERIPHERALS (BYTE)
(c)

Figure 4A1. Techniques for I/O Device Chip Selects

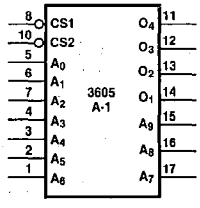


Figure 4A2. Bipolar PROM Decoder

One last technique for interfacing with eight-bit peripherals is considered in Figure 4A3. The sixteen-bit data bus is multiplexed onto an eight-bit bus to accommodate byte oriented DMA or block transfers to memory mapped eight-bit I/O. Devices connected to this interface may be assigned a sequence of odd and even addresses rather than all odd or even.

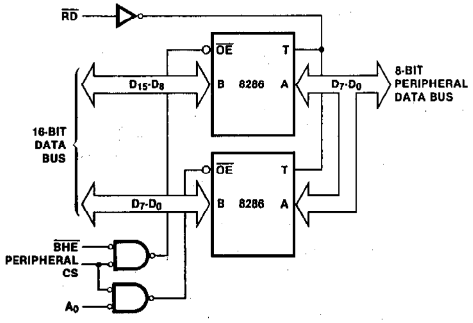


Figure 4A3. 16- to 8-Bit Bus Conversion

4B. Sixteen-Bit I/O

For obvious reasons of efficient bus utilization and simplicity of device selection, sixteen-bit I/O devices should be assigned even addresses. To guarantee the device is selected only for word operations, A0 and $\overline{BHE}$ should be conditions of chip select code (Fig. 4B1).

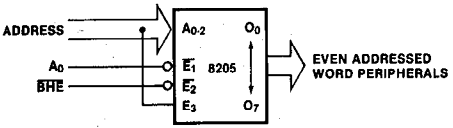
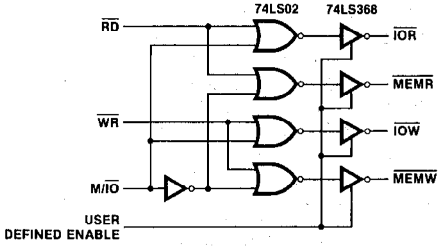


Figure 4B1. Sixteen-Bit I/O Decode

4C. General Design Considerations

MIN/MAX, MEMORY I/O MAPPED AND LINEAR SELECT

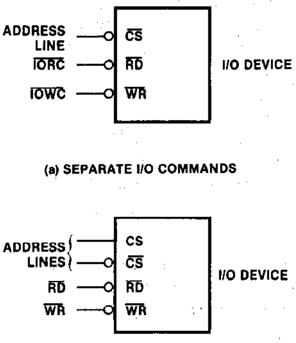
Since the minimum mode 8086 has common read and write commands for memory and I/O, if the memory and I/O address spaces overlap, the chip selects must be qualified by M/$\overline{IO}$ to determine which address space the devices are assigned to. This restriction on chip select decoding can be removed if the I/O and memory addresses in the system do not overlap and are properly decoded; all I/O is memory mapped; or $\overline{RD}$, $\overline{WR}$ and M/$\overline{IO}$ are decoded to provide separate memory and I/O read/write commands (Fig. 4C1). The 8288 bus controller in the maximum mode 8086 system generates separate I/O and memory commands in place of a M/$\overline{IO}$ signal. An I/O device is assigned to the I/O space or memory space (memory mapped I/O) by connection of either I/O or memory command lines to the command inputs of the device. To allow overlap of the memory and I/O address space, the device must not respond to chip select alone but must require a combination of chip select and a read or write command.



NOTE: IF IT IS NOT NECESSARY TO THREE-STATE THE COMMAND LINES, A DECODER (8205 OR 74S138) COULD BE USED. THE 74LS257 IS NOT RECOMMENDED SINCE THE OUTPUTS MAY EXPERIENCE VOLTAGE SPIKES WHEN ENTERING OR LEAVING THREE-STATE.

Figure 4C1. Decoding Memory and I/O $\overline{RD}$ and $\overline{WR}$ Commands for Minimum Mode 8086 Systems

Linear select techniques (Fig. 4C2) for I/O devices can only be used with devices that either reside in the I/O address space or require more than one active chip select (at least one low active and one high active). Devices with a single chip select input cannot use linear select if they are memory mapped. This is due to the assignment of memory address space FFFFF0H-FFFFFFH to reset startup and memory space 00000H-003FFH to interrupt vectors.



(a) SEPARATE I/O COMMANDS

(b) MULTIPLE CHIP SELECTS

Figure 4C2. Linear Select for I/O

4D. Determining I/O Device Compatibility

This section presents a set of A.C. characteristics which represent the timing of the asynchronous bus interface of the 8086. The equations are expressed in terms of the CPU clock (when applicable) and are derived for minimum and maximum modes of the 8086. They represent the bus characteristics at the CPU.

The results can be used to determine I/O device requirements for operation on a single CPU local bus or buffered system bus. These values are not applicable to a Multibus system bus interface. The requirements for a Multibus system bus are available in the Multibus interface specification.

A list of bus parameters, their definition and how they relate to the A.C. characteristics of Intel peripherals are given in Table 4D1. Cycle dependent values of the parameters are given in Table 4D2. For each equation, if more than one signal path is involved, the equation reflects the worst case path.

ex. TAVRL(address valid before read active) =
(1) Address from CPU to $\overline{\text{RD}}$ active
(or)
(2) ALE (to enable the address through the address latches) to $\overline{\text{RD}}$ active

The worst case delay path is (1).

For the maximum mode 8086 configurations, TAVWLA, TWLWHA and TWLCLA are relative to the advanced write signal while TAVWL, TWLWH and TWLCL are relative to the normal write signal.

TABLE 4D1. PARAMETERS FOR PERIPHERAL COMPATIBILITY

TAVRL	— Address stable before RD leading edge	(TAR)
TRHAX	— Address hold after RD trailing edge	(TRA)
TRLRH	— Read pulse width	(TRR)
TRLDV	— Read to data valid delay	(TRD)
TRHDZ	— Read trailing edge to data floating	(TDF)
TAVDV	— Address to valid data delay	(TAD)
TRLRL	— Read cycle time	(TRCYC)
TAVWL	— Address valid before write leading edge	(TAW)
TAVWLA	— Address valid before advanced write	(TAW)
TWHAX	— Address hold after write trailing edge	(TWA)
TWLWH	— Write pulse width	(TWW)
TWLWHA	— Advanced write pulse width	(TWW)
TDVWH	— Data set up to write trailing edge	(TDW)
TWHDX	— Data hold from write trailing edge	(TWD)
TWLCL	— Write recovery time	(TRV)
TWLCLA	— Advanced write recovery time	(TRV)
TSVRL	— Chip select stable before RD leading edge	(TAR)
TRHSX	— Chip select hold after RD trailing edge	(TRA)
TSLDV	— Chip select to data valid delay	(TRD)
TSVWL	— Chip select stable before WR leading edge	(TAW)
TWHSX	— Chip select hold after WR trailing edge	(TWA)
TSVWLA	— Chip select stable before advanced write	(TAW)

Symbols in parentheses are equivalent parameters specified for Intel peripherals.

In the given list of equations, TWHDXB is the data hold time from the trailing edge of write for the minimum mode with a buffered data bus. For this equation, TCVCTX cannot be a minimum for data hold and a maximum for write inactive. The maximum difference is 50 ns giving the result TCLCH-50. If the reader wishes to verify the equations or derive others, refer to Section 3F for assistance with interpreting the 8086 bus timing diagrams.

Figure 4D1 shows four representative configurations and the compatible Intel peripherals (including wait states if required) for each configuration are given in Table 4D3. Configuration 1 and 2 are minimum mode demultiplexed bus 8086 systems without (1) and with (2) data bus transceivers. Configurations 3 and 4 are maximum mode systems with one (3) and two (4) levels of address and data buffering. The last configuration is characteristic of a multi-board system with bus buffers on each board. The 5 MHz parameter values for these configurations are given in Table 4D4 and demonstrate the relaxed device requirements for even a large complex configuration. The analysis assumes all components are exhibiting the specified worst case parameter values and are under the corresponding temperature, voltage and capacitive load conditions. If the capacitive loading on the 8282/83 or 8286/87 is less than the maximum, graphs of delay vs. capacitive loading in the respective data sheets should be used to determine the appropriate delay values.

TABLE 4D2. CYCLE DEPENDENT PARAMETER REQUIREMENTS FOR PERIPHERALS

(a) Minimum Mode

TAVRL = TCLCL + TCLRLmin – TCLAVmax = TCLCL – 100
TRHAX = TCLCL – TCLRHmax + TCLLHmin = TCLCL – 150
TRLRH = 2TCLCL – 60 = 2TCLCL – 60
TRLDV = 2TCLCL – TCLRLmax – TDVCLmin = 2TCLCL – 195
TRHDZ = TRHAVmin = 155 ns
TAVDV = 3TCLCL – TDVCLmin – TCLAVmax = 3TCLCL – 140
TRLRL = 4TCLCL = 4TCLCL
TAVWL = TCLCL + TCVCTVmin – TCLAVmax = TCLCL – 100
TWHAX = TCLCL + TCLLHmin – TCVCTXmax = TCLCL – 110
TWLWH = 2TCLCL – 40 = 2TCLCL – 40
TDVWH = 2TCLCL + TCVCTXmin – TCLDVmax = 2TCLCL – 100
TWHDX = TWHDZmin = 89
TWLCL = 4TCLCL = 4TCLCL
TWHDXB = TCLCHmin + (– TCVCTXmax + TCVCTXmin) =
TCLCHmin – 50

Note: Delays relative to chip select are a function of the chip select decode technique used and are equal to: equivalent delay from address – chip select decode delay.

(b) Maximum Mode

TAVRL = TCLCL + TCLMLmin – TCLAVmax = TCLCL – 100
TRHAX = TCLCL – TCLMHmax + TCLLHmin = TCLCL – 40
TRLRH = 2TCLCL – TCLMLmax + TCLMHmin = 2TCLCL – 25
TRLDV = 2TCLCL – TCLMLmax – TDVCLmin = 2TCLCL – 65
TRHDZ = TRHAVmin = 155
TAVDV = 3TCLCL – TDVCLmin – TCLAVmax = 3TCLCL – 140
TRLRL = 4TCLCL = 4TCLCL
TAVWLA = TAVRL = TCLCL – 100
TAVWL = TAVRL + TCLCL = 2TCLCL – 100
TWHAX = TRHAX = TCLCL – 40
TWLWHA = TRLRH = 2TCLCL – 25
TWLWH = TRLRH – TCLCL = TCLCL – 25
TDVWH = 2TCLCL + TCLMHmin – TCLDVmax = 2TCLCL – 100
TWHDX = TCLCHmin – TCLMHmax + TCHDZmin = TCLCHmin – 30
TWLCL = 3TCLCL = 3TCLCL
TWLCLA = 4TCLCL = 4TCLCL

TABLE 4D3. COMPATIBLE PERIPHERALS (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
8251A	✓	1W	✓	✓
8253-5	✓	1W	✓	✓
8255A-5	✓	1W	✓	✓
8257-5	✓	1W	✓	✓
8259A	✓	✓	✓	✓
8271	✓	1W	✓	✓
8273	✓	1W	✓	✓
8275	✓	1W	✓	✓
8279-5	✓	1W	✓	✓
8041A*	✓	1W	✓	✓
8741A	✓	1W	✓	✓
8291	✓	✓	✓	✓

*Includes other Intel peripherals based on the 8041A (i.e., 8292, 8294, 8295).

✓ implies full operation with no wait states.

W implies the number of wait states required.

TABLE 4D4. PERIPHERAL REQUIREMENTS FOR FULL SPEED OPERATION WITH 5 MHz 8086

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
TAVRL	70	72	70	58
TRHAX	57	27	169	141
TRLRH	340	320	375	347
TRLDV	205	150	305	261
TRHDZ	155	158	382	360
TAVDV	430	400	400	372
TRLRL	800	770	800	772
TAVWL	70	72	270	258
TAVWLA	—	—	70	58
TWHAX	97	67	169	141
TWLWH	360	340	175	147
TWLWHA	—	—	375	347
TDVWH	300	339	270	258
TWHDX	88	15	95	13
TWLCL	800	772	600	572
TWLCLA	—	—	800	772
TSVRL	52	54	52	40
TRHSX	50	50	171	143
TSLDV	412	382	382	354
TSVWL	52	54	252	240
TWHSX	90	90	171	143
TSVWLA	—	—	52	40

— Not applicable.

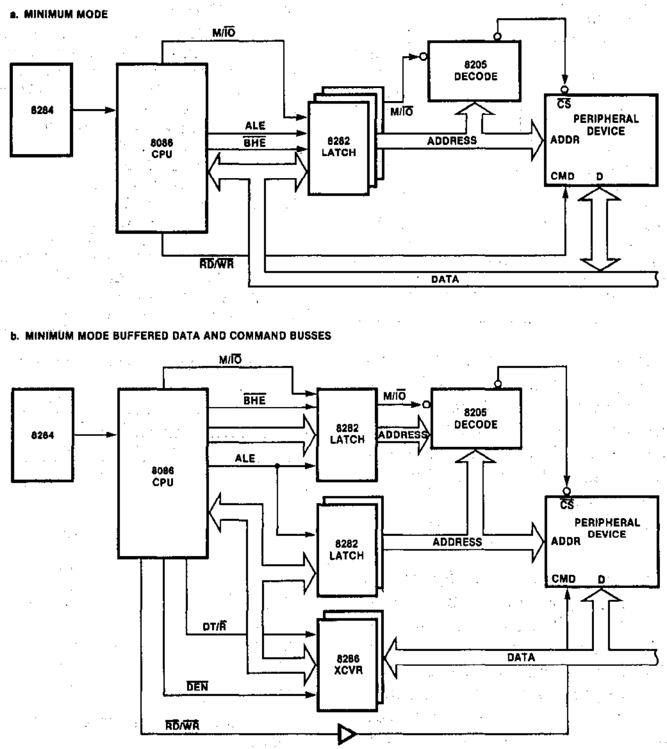
Peripheral compatibility is determined from the equations given for the CPU by modifying them to account for additional delays from address latches and data transceivers in the configuration. Once the system configuration is selected, the system requirements can be determined at the peripheral interface and used to evaluate compatibility of the peripheral to the system. During this process, two areas must be considered. First, can the device operate at maximum bus bandwidth and if not, how many wait states are required. Second, are there any problems that cannot be resolved by wait states.

Examples of the first are TRLRH (read pulse width) and TRLDV (read access or $\overline{RD}$ active to output data valid). Consider address access time (valid address to valid data) for the maximum mode fully buffered configuration.

TAVDV = 3TCYC − 140 ns — address latch delay — address buffer delay — chip select decode delay — 2 transceiver delays

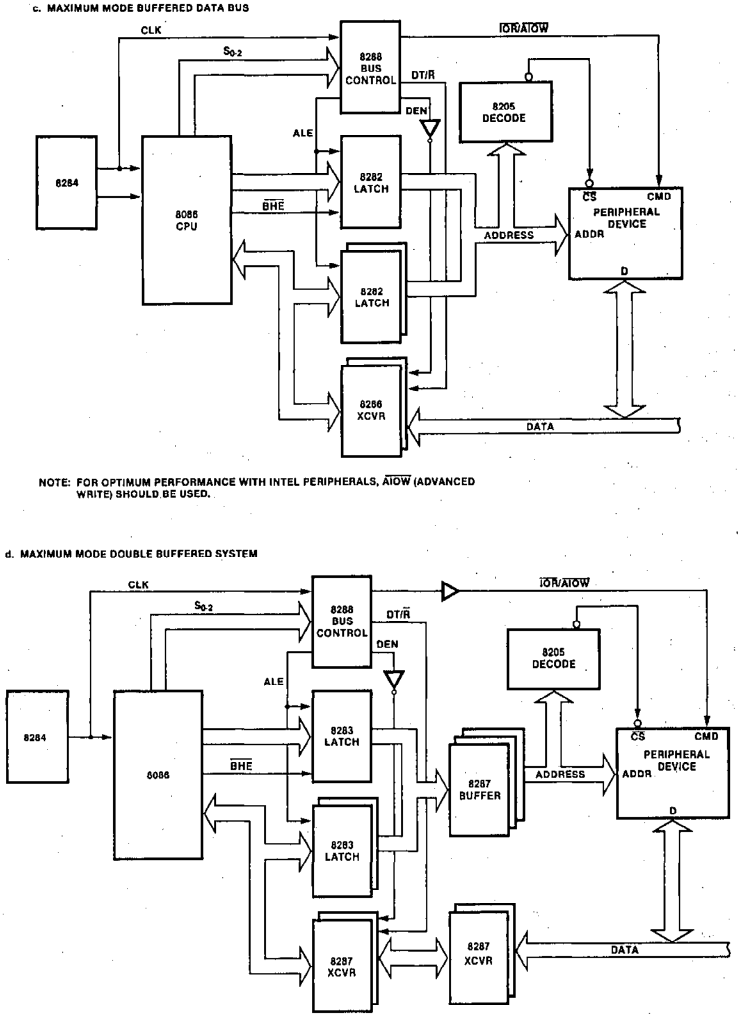
Assuming inverting latches, buffers and transceivers with 22 ns max delays (8283, 8287) and a bipolar PROM decode with 50 ns delay, the result is:

TAVDV = 322 ns @ 5 MHz


a. MINIMUM MODE

b. MINIMUM MODE BUFFERED DATA AND COMMAND BUSSES

Figure 4D1. 8086 System Configurations



c. MAXIMUM MODE BUFFERED DATA BUS

NOTE: FOR OPTIMUM PERFORMANCE WITH INTEL PERIPHERALS, $\overline{AIOW}$ (ADVANCED WRITE) SHOULD BE USED.

d. MAXIMUM MODE DOUBLE BUFFERED SYSTEM

Figure 4D1. 8086 System Configurations (Con't)

The result gives the address to data valid delay required at the peripheral (in this configuration) to satisfy zero wait state CPU access time. If the maximum delay specified for the peripheral is less than the result, this parameter is compatible with zero wait state CPU operation. If not, wait states must be inserted until TAVDV + n * TCYC (n is the number of wait states) is greater than the peripherals maximum delay. If several parameters require wait states, either the largest number required should always be used or different transfer cycles can insert the maximum number required for that cycle.

The second area of concern includes TAVRL (address set up to read) and TWHDX (data hold after write). Incompatibilities in this area cannot be resolved by the insertion of wait states and may require either additional hardware, slowing down the CPU (if the parameter is related to the clock) or not using the device.

As an example consider address valid prior to advanced write low (TAVWLA) for the maximum mode fully buffered system.

TAVWLA = TCYC − 100 ns — address latch delay — address buffer delay — chip select decode delay + write buffer delay (minimum)

Assuming inverting latches and buffers with 22 ns delay (8283, 8287) and an 8205 address decoder with 18 ns delay

TAVWLA = 38 ns which is the time a 5 MHz 8086 system provides

4E. I/O Examples

1. Consider an interrupt driven procedure for handling multiple communication lines. On receiving an interrupt from one of the lines, the invoked procedure polls the lines (reading the status of each) to determine which line to service. The procedure does not enable lines but simply services input and output requests until the associated output buffer is empty (for output requests) or until an input line is terminated (for the example only EOT is considered). On detection of the terminate condition, the routine will disable the line. It is assumed that other routines will fill a lines output buffer and enable the device to request output or empty the input buffer and enable the device to input additional characters.

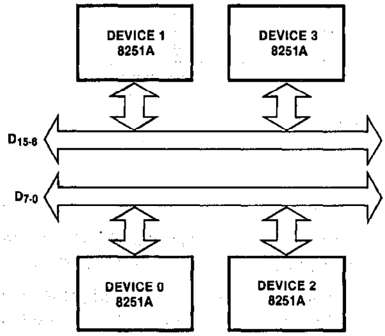
The routine begins operation by loading CX with a count of the number of lines in the system and DX with the I/O address of the first line. The I/O addresses are assigned as shown in Figure 4E1 with 8251A's as the I/O devices. The status of each line is read to determine if it needs service. If yes, the appropriate routine is called to input or output a character. After servicing the line or if no service is needed, CX is decremented and DX is incremented to test the next line. After all lines have been tested and serviced, the routine terminates. If all interrupts from the lines are OR'd together, only one interrupt is used for all lines. If the interrupt is input to the CPU through an 8259A interrupt controller, the 8259A should be programmed in the level triggered mode to guarantee all line interrupts are serviced.

To service either an input or output request, the called routine transfers DX to BX, and shifts BX to form the offset for this device into the table of input or output buffers. The first entry in the buffer is an index to the next character position in the buffer and is loaded into the SI register. By specifying the base address of the table of buffers as a displacement into the data segment, the base + index + displacement addressing mode allows direct access to the appropriate memory location. 8086 code for part of this example is shown in Figure 4E2.

2. As a second example, consider using memory mapped I/O and the 8086 string primative instructions to perform block transfers between memory and I/O. By assigning a block of the memory address space (equivalent in size to the maximum block to be transferred to the I/O device) and decoding this address space to generate the I/O device's chip select, the block transfer capability is easily implemented. Figure 4E3 gives an interconnect for 16-bit I/O devices while Figure 4E4 incorporates the 16-bit bus to 8-bit bus multiplexing scheme to support 8-bit I/O devices. A code example to perform such a transfer is shown in Figure 4E5.

```
; THIS CODE DEMONSTRATES TESTING DEVICE
; STATUS FOR SERVICE, CONSTRUCTING THE
; APPROPRIATE LINE BUFFER ADDRESS FOR INPUT
; AND OUTPUT AND SERVICING AN INPUT
; REQUEST

                 MASK EQU OFFFDH
CHECK_STATUS:    INPUT  AL, DX                  ; GET 8251A STATUS.
                 MOV    AH, AL
                 TEST   AH, READ_OR_WRITE_STATUS
                 JZ     NEXT_IO
                 CALL   ADDRESS
                 TEST   AH, READ STATUS
                 JZ     WRITE_SERVICE
                 CALL   READ
                 TEST   AH, WRITE STATUS
                 JZ     NEXT_IO
WRITE_SERVICE:   CALL   WRITE
NEXT_IO:         DEC    CX                      ; TEST IF DONE.
                 JNC    EXIT                    ; YES, RESTORE & RETURN.
                 AND    DX, MASK                ; REMOVE A1 AND
                 ADD    DX, 3                   ; INCREMENT ADDRESS.
                 OR     DX, 2                   ; SELECT STATUS FOR
                 JMP    CHECK_STATUS            ; NEXT INPUT.

ADDRESS:         AND    DX, MASK                ; SELECT DATA.
                 MOV    BH, DL                  ; CONSTRUCT BUFFER
                 INC    BH                      ; DISPLACEMENT FOR
                 SHR    BH                      ; THIS DEVICE.
                 XOR    BL, BL                  ; BX IS THE DISPLACEMENT.
                 RET

READ:            INPUT AL, DX                         ; READ CHARACTER.
                 MOV SI, READ_BUFFERS [BX]            ; GET CHARACTER POINTER.
                 MOV READ_BUFFERS [BX + SI], AL       ; STORE CHARACTER.
                 INC READ_BUFFERS [BX]                ; INCR CHARACTER POINTER.
                 CMP AL, EOT                          ; END OF TRANSMISSION?
                 JNZ CONT_READ
                 CALL DISABLE READ                    ; YES, DISABLE RECEIVER.
                 CONT_READ:   RET                     ; SEND MESSAGE THAT INPUT
                                                      ; IS READY.
```

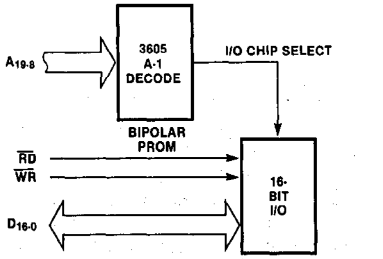
Figure 4E2.



DEVICES ARE CONNECTED TO THE UPPER AND LOWER HALVES OF THE DATA BUS.

ADDRESS		
0	DEVICE 0	DATA
1	DEVICE 1	DATA
2	DEVICE 0	CONTROL/STATUS
3	DEVICE 1	CONTROL/STATUS
4	DEVICE 2	DATA
5	DEVICE 3	DATA
6	DEVICE 2	CONTROL/STATUS
7	DEVICE 3	CONTROL/STATUS
ETC.	"	"

Figure 4E1. Device Assignment



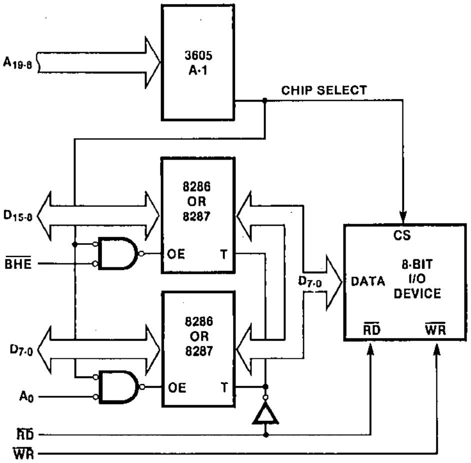
TRANSFER 256 BYTE BLOCKS TO THE I/O DEVICE

THE ADDRESS SPACE ASSIGNED TO THE I/O DEVICE IS

	A19	A8	A7 A0
FROM	BASE	ADDRESS	0's
THRU	BASE	ADDRESS	1's

MEMORY DATA NEED NOT BE ALIGNED TO EVEN ADDRESS BOUNDARIES
I/O TRANSFERS MUST BE WORD TRANSFERS TO EVEN ADDRESS BOUNDARIES

Figure 4E3. Block Transfer to 16-Bit I/O Using 8086 String Primatives



ADDRESS ASSIGNMENT SAME AS PREVIOUS EXAMPLE. 16-BIT BUS IS MULTIPLEXED ONTO AN 8-BIT PERIPHERAL BUS.

Figure 4E4. Block Transfer to 8-Bit I/O Using 8086 String Primatives

```
            ; DEFINE THE I/O ADDRESS SPACE
              I/O SEGMENT
              ORG BLOCK_ADDRESS
I/O_BLOCK:    DW 128 DUP (?)
              I/O ENDS

            ; ASSUME THE DATA IS FROM THE CURRENT
            ; DATA SEGMENT
              CLD                          ; DF = FORWARD
              LES DI, I/O_BLOCK_ADDRESS    ; I/O BLOCK ADDRESS
                                           ; CONTAINS THE ADDRESS
                                           ; OF I/O BLOCK
              MOV CX, BLOCK_LENGTH
              MOV SI, SOURCE_ADDRESS
              MOVS I/O BLOCK               ; PERFORM WORD TRANSFERS

            ; END CODE EXAMPLE
```

NOTE THE CODE IS CAPABLE OF PERFORMING BYTE TRANSFERS BY CHANGING THE I/O BLOCK DEFINITION FROM 128 WORD TO 256 BYTES

Figure 4E5. Code for Block Transfers

5. INTERFACING WITH MEMORIES

Figure 5.1 is a general block diagram of an 8086 memory. The basic characteristics of the diagram are the partitioning of the 16-bit word memory into high and low 8-bit banks on the upper and lower halves of the data bus and inclusion of $\overline{\text{BHE}}$ and A0 in the selection of the banks. Specific implementations depend on the type of memory and the system configuration.

5A. ROM and EPROM

The easiest devices to interface to the system are ROM and EPROM. Their byte format provides a simple bus interface and since they are read only devices, A0 and $\overline{\text{BHE}}$ need not be included in their chip enable/select decoding (chip enable is similar to chip select but additionally determines if the device is in active or standby power mode). The address lines connected to the devices start with A1 and continue up to the maximum number the device can accept, leaving the remaining address lines for chip enable/select decoding. To connect the devices directly to the multiplexed bus, they must have output enables. The output enable is also necessary to avoid bus contention in other configurations. Figure 5A1 shows the bus connections for ROM and EPROM memories. No special decode techniques are required for generating chip enables/selects. Each valid decode selects one device on the upper and lower halves of bus to allow byte and word access. Byte access is achieved by reading the full word onto the bus with the 8086 only accepting the desired byte. For the minimum mode 8086, if $\overline{\text{RD}}$, $\overline{\text{WR}}$ and M/$\overline{\text{IO}}$ are not decoded to form separate commands for memory and I/O, and the I/O space overlaps the memory space assigned to the EPROM/ROM then M/$\overline{\text{IO}}$ (high active) must be a condition of chip enable/select decode. The output enable is controlled by the system memory read signal.

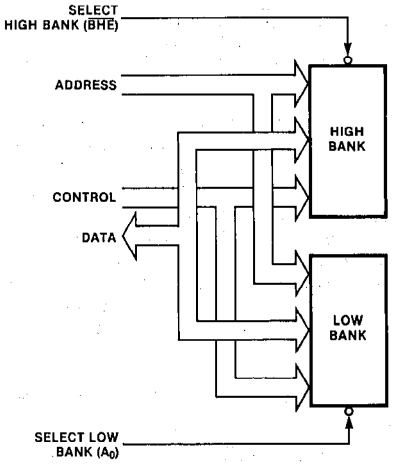
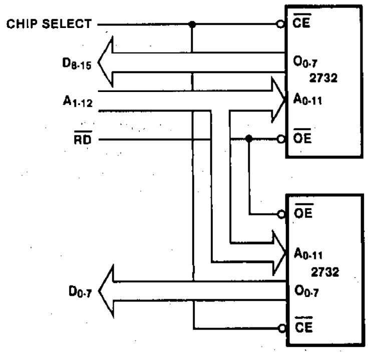


Figure 5.1. 8086 Memory Array



NOTE A_0 AND $\overline{\text{BHE}}$ ARE NOT USED.

Figure 5A1. EPROM/ROM Bus Interface

Static ROM's and EPROM's have only four parameters to evaluate when determining their compatibility to the system. The parameters, equations and evaluation techniques given in the I/O section are also applicable to these devices. The relationship of parameters is given in Table 5A1. TACC and TCE are related to the same equation and differ only by the delay associated with the chip enable/select decoder. As an example, consider a 2716 EPROM memory residing on the multiplexed bus of a minimum mode configuration:

TACC = 3TCLCL − 140 − address buffer delay = 430 ns
(8282 = 30 ns max delay)

TCE = TACC − decoder delay = 412 ns
(8205 decoder delay = 18 ns)

TOE = 2TCLCL − 195 = 205 ns

TDF = = 155 ns

TABLE 5A1. EPROM/ROM PARAMETERS

TOE — Output Enable to Valid Data ≡ TRLDV
TACC — Address to Valid Data ≡ TAVDV
TCE — Chip Enable to Valid Data ≡ TSLDV
TDF — Output Enable High to Output Float ≡ TRHDZ

The results are the times the system configuration requires of the component for full speed compatibility with the system. Comparing these times with 2716 parameter limits indicates the 2716-2 will work with no wait states while the 2716 will require one wait state. Table 5A2 demonstrates EPROM/ROM compatibility for the configurations presented in the I/O section. Before designing a ROM or EPROM memory system, refer to AP-30 for additional information on design techniques that give the system an upgrade path from 16K to 32K and 64K devices.

TABLE 5A2. COMPATIBLE EPROM/ROM (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
2716-1	✓	✓	✓	✓
2716-2	✓	1W	1W	1W
2732	1W	1W	1W	1W
2332	✓	✓	✓	✓
2364	✓	✓	✓	✓

5B. Static RAM

Interfacing static RAM to the system introduces several new requirements to the memory design. A0 and $\overline{\text{BHE}}$ must be included in the chip select/chip enable decoding of the devices and write timing must be considered in the compatibility analysis.

For each device, the data bus connections must be restricted to either the upper or lower half of the data bus. Devices like the 2114 or 2142 must not straddle the upper and lower halves of the data bus (Fig. 5B1). To allow selecting either the upper byte, lower byte or full 16-bit word for a write operation, $\overline{\text{BHE}}$ must be a condition of decode for selecting the upper byte and A0 must be a condition of decode for selecting the lower byte. Figure 5B2 gives several selection techniques for devices with single chip selects and no output enables (2114, 2141, 2147). Figure 5B3 gives selection techniques for devices with chip selects and output enables.

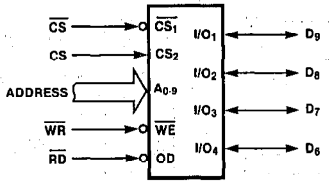


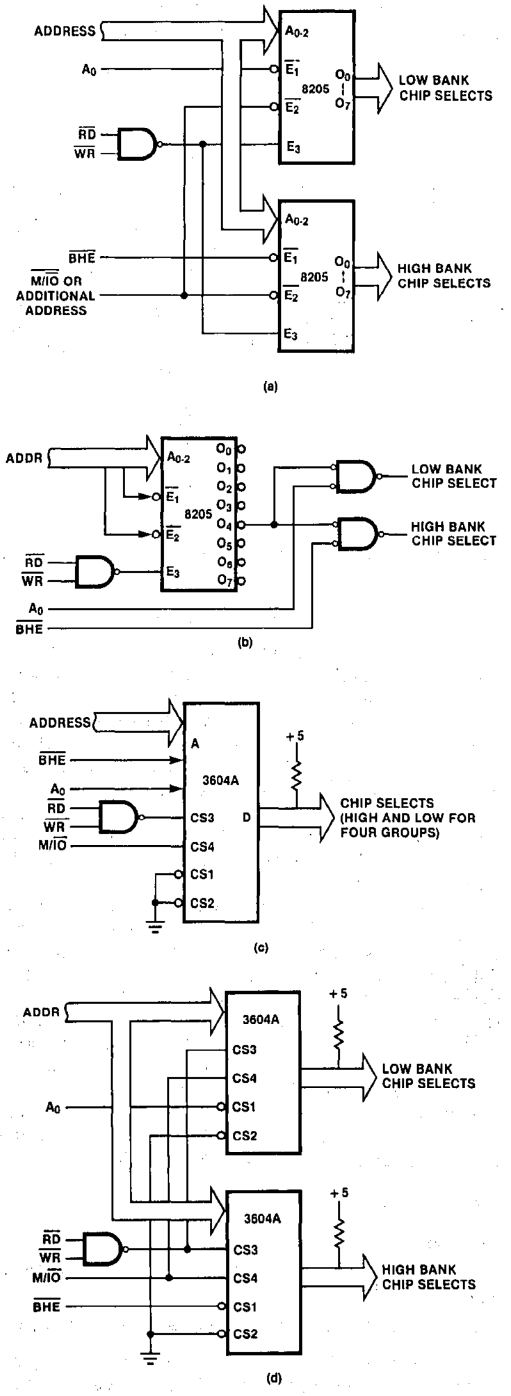
Figure 5B1. Incorrect Connection of 2142 Across Byte Boundaries

The first group requires inclusion of A0 and $\overline{\text{BHE}}$ to decode or enable the chip selects. Since these memories do not have output enables, read and write are used as enables for chip select generation to prevent bus contention. If read and write are not used to enable the chip selects, devices with common input/output pins (like the 2114) will be subjected to severe bus contention between chip select and write active. For devices with separate input/output lines (like 2141, 2147), the outputs can be externally buffered with the buffer enable controlled by read. This solution will only allow bus contention between memory devices in the array during chip select transition periods. These techniques are considered in more detail in Section 2C.

For devices with output enables (2142), write may be gated with $\overline{\text{BHE}}$ and A0 to provide upper and lower bank write strobes. This simplifies chip select decoding by eliminating $\overline{\text{BHE}}$ and A0 as a condition of decode. Although both devices are selected during a byte write operation, only one will receive a write strobe. No bus contention will exist during the write since a read command must be issued to enable the memory output drivers.

If multiple chip selects are available at the device, $\overline{\text{BHE}}$ and A0 may directly control device selection. This allows normal chip select decoding of the address space and direct connection of the read and write commands to the devices. Alternately, the multiple chip select inputs of the device could directly decode the address space (linear select) and be combined with the separate write strobe technique to minimize the control circuitry needed to generate chip selects.

As with the EPROM's and ROM's, if separate commands are not provided for memory and I/O in the minimum mode 8086 and the address spaces overlap, M/$\overline{\text{IO}}$ (high active) must be a condition of chip select decode. Also, the address lines connected to the memory devices must start with A1 rather than A0.



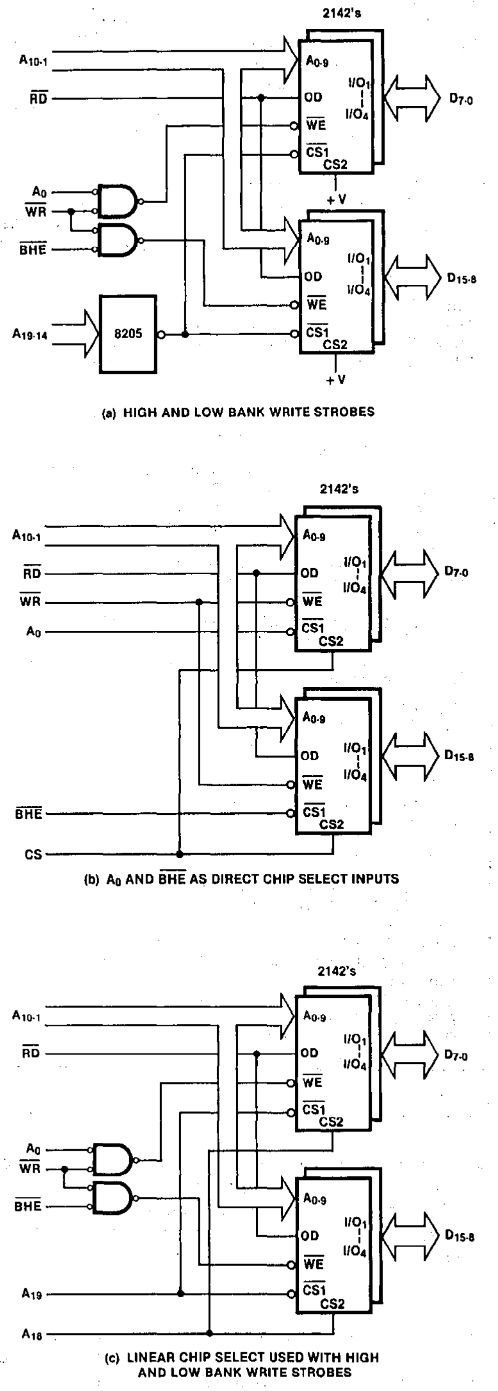
(a)

(b)

(c)

(d)

Figure 5B2. Generating Chip Selects for Devices without Output Enables



(a) HIGH AND LOW BANK WRITE STROBES

(b) A0 AND BHE AS DIRECT CHIP SELECT INPUTS

(c) LINEAR CHIP SELECT USED WITH HIGH AND LOW BANK WRITE STROBES

Figure 5B3. Chip Selection for Devices with Output Enables

For analysis of RAM compatibility, the write timing parameters listed in Table 5B1 may also need to be considered (depending on the RAM device being considered). The CPU clock relative timing is given in Table 5B2. The equations specify the device requirements at the CPU and provide a base for determining device requirements in other configurations. As an example consider the write timing requirements of a 2142 in a maximum mode buffered 8086 system (Figure 5B4). The 2142 write parameters that must be analyzed are TWA advanced write pulse width, TWR write release time, TDWA data to write time overlap and TDH data hold from write time.

TWA = 2TCLCL − TCLMLmax + TCLMHmin = 375 ns.
TWR = 2TCLCL − TCLMHmax + TCLLHmin + TSHOVmin = 170 ns.
TDWA = 2TCLCL − TCLDVmax + TCLMHmin − TIVOVmax = 265 ns.
TDH = TCLCH − TCLMHmax + TCHDXmin + TIVOVmin = 95 ns.

TABLE 5B1. TYPICAL WRITE TIMING PARAMETERS

TW — Write Pulse Width
TWR — Write Release (Address Hold From End of Write)
TDW — Data and Write Pulse Overlap
TDH — Data Hold From End of Write
TAW — Address Valid to End of Write
TCW — Chip Select to End of Write
TASW — Address Valid to Beginning of Write

TABLE 5B2. CYCLE DEPENDENT WRITE PARAMETERS FOR RAM MEMORIES

(a) Minimum Mode

TW = TWLWH = 2TCLCL − 60 = 340 ns
TWR = TCLCL − TCVCTXmax + TCLLHmin = 90 ns
TDW = 2TCLCL − TCLDVmax + TCVCTXmin = 300 ns
TDH = TWHDX = 88 ns
TAW = 3TCLCL − TCLAVmax + TCVCTXmin = 500 ns
TCW = TAW − Chip Select Decode
TASW = TCLCL − TCLAVmax + TCVCTXmin = 100 ns

(b) Maximum Mode

TW = TCLCL − TCLMLmax + TCLMHmin = 175 ns
TWR = TCLCL − TCLMHmax + TCLLHmin = 165 ns
TDW = TW = 175 ns
TDH = TCLCHmin − TCLMHmax + TCHDXmin = 93 ns
TAW = 3TCLCL − TCLAVmax + TCLMHmin = 500 ns
TCW = TAW − Chip Select Decode
TASW = 2TCLCL − TCLAVmax + TCLMLmin = 300 ns
TWA* = TW + TCLCL = 375 ns
TDWA* = 2TCLCL − TCLDVmax + TCLMHmin = 300 ns
TASWA* = TASW − TCLCL = 100 ns

*Relative to Advanced Write.

Comparing these results with the 2142 family indicates the standard 2142 write timing is fully compatible with this 8086 configuration. Read timing analysis is also necessary to completely determine compatibility of the devices.

5C. Dynamic RAM

Dynamic RAM is perhaps the most complex device to design into a system. To relieve the engineer of most of this burden, Intel provides the 8202 dynamic RAM controller as part of the 8086 family of peripheral devices. This section will discuss using the 8202 with the 8086 to build a dynamic memory system for an 8086 system. For additional information on the 8202, refer to the 8202 data sheet (9800873) and application note AP-45 Using the 8202 Dynamic RAM Controller (9800809A).

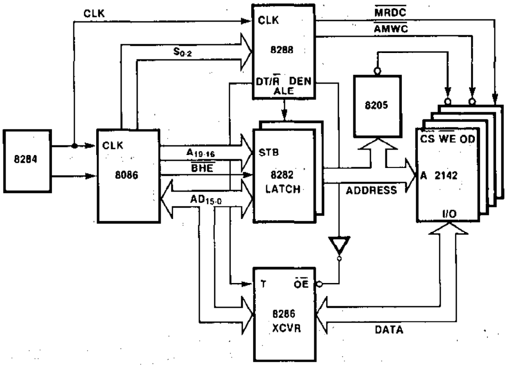


Figure 5B4. Sample Configuration for Compatibility Analysis Example

5.C.1 Standard 8086-8202 Interconnect

Figure 5.C.1.1 shows a standard interconnection for an 8202 into an 8086 system. The configuration accommodates 64K words (128K bytes) of dynamic RAM addressable as words or bytes. To access the RAM, the 8086 initiates a bus cycle with an address that selects the 8202 (via $\overline{PCS}$) and the appropriate transfer command ($\overline{MRDC}$ or $\overline{MWTC}$). If the 8202 is not performing a refresh cycle, the access starts immediately, otherwise, the 8086 must wait for completion of the refresh. $\overline{XACK}$ from the 8202 is connected to the 8284 RDY input to force the CPU to wait until the RAM cycle is completed before the CPU can terminate the bus cycle. This effectively synchronizes the asynchronous events of refresh and CPU bus cycles. The normal write command ($\overline{MWTC}$) is used rather than the advanced command ($\overline{AMWC}$) to guarantee the data is valid at the dynamic RAMS before the write command is issued. The gating of $\overline{WE}$ with A0 and $\overline{BHE}$ provides selective write strobes to the upper and lower banks of memory to allow byte and word write operations. The logic which generates the strobe for the data latches allows read data to propagate to the system as soon as the data is available and latches the data on the trailing edge of $\overline{CAS}$.

DETAILED TIMING

Read Cycle

For no wait state operation, the 8086 requires data to be valid from $\overline{MRDC}$ in:

2TCLCL − TCLML − TDVCL − buffer delays = 291 ns.

Since the 8202 is $\overline{CAS}$ access limited, we need only examine $\overline{CAS}$ access time. The 8202/2118 guarantees data valid from 8202 $\overline{RD}$ low to be:

(tph + 3tp + 100 ns) 8202 TCC delay + TCAC for the 2118

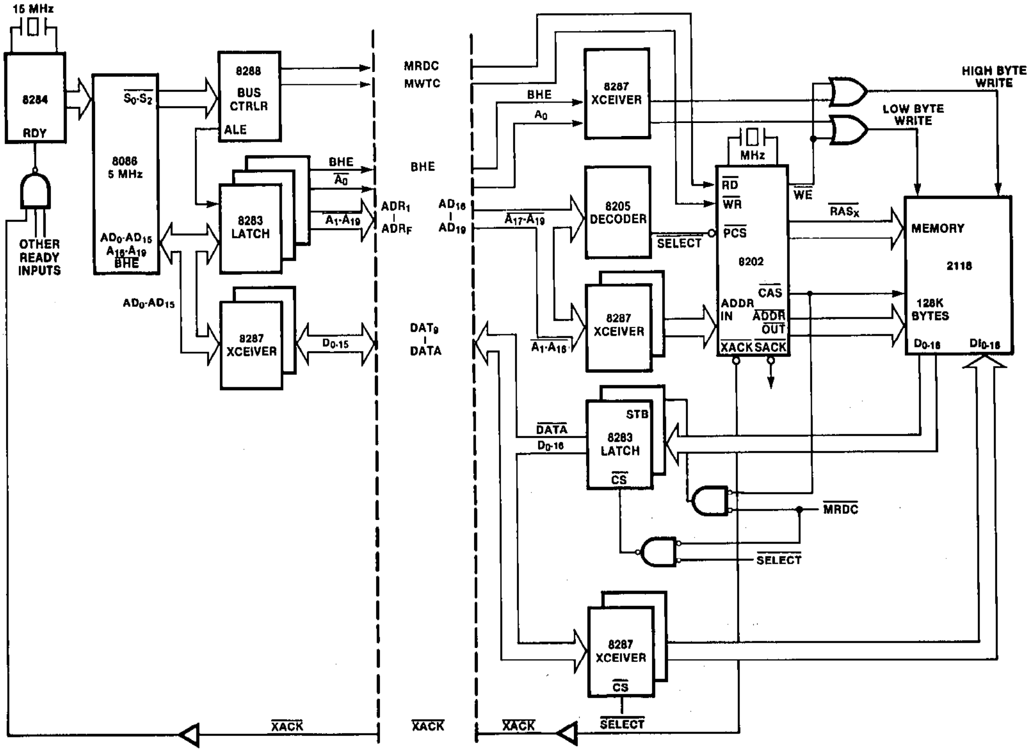


Figure 5C1.1. 5 MHz 8086/8202/128K Byte System — Double Data, Control and Address Buffering (Note: Bus driver on 8202 is not needed if less than 64K bytes are used)

For a 25 MHz 8202 and 2118-3, we get 297 ns which is insufficient for no wait state operation. If only 64K bytes are accessed, the 8202 requires only (tph + 3tp + 85 ns) giving 282 ns access and no wait states required. Refer to Figure 5.C.1.2 and 5C.1.3 for timing information on the 8202 and 2118.

Write Cycle

An important consideration for dynamic RAM write cycles is to guarantee data to the RAM is valid when both $\overline{CAS}$ and $\overline{WE}$ are active. For the 2118, if $\overline{WE}$ is valid prior to $\overline{CAS}$, the data setup is to $\overline{CAS}$ and if $\overline{CAS}$ is valid before $\overline{WE}$ (as would occur during a read modify write cycle) the data setup time is to $\overline{WE}$. For the 8202, the $\overline{WR}$ to $\overline{CAS}$ delay is analyzed to determine the data setup time to $\overline{CAS}$ inherently provided by the 8202 command to $\overline{RAS}/\overline{CAS}$ timing. The minimum delay from $\overline{WR}$ to $\overline{CAS}$ is:

TCCmin = tph + 2tp + 25 = 127 ns @ 25 MHz

Subtracting buffer delays and data setup at the 2118, we have 83 ns to generate valid data after the write command is issued by the CPU (in this case the 8288). Since the 8086 will not guarantee valid data until TCLAVmax − TCLMLmin = 100 ns from the advanced write signal, the normal write signal is used. The normal write $\overline{MWTC}$ guarantees data is valid 100 ns before it is active. The worst case write pulse width is approximately 175 ns which is sufficient for all 2118's.

Synchronization

To force the 8086 to wait during refresh the $\overline{XACK}$ or $\overline{SACK}$ lines must be returned to the 8284 ready input. The maximum delay from $\overline{RD}$ to $\overline{SACK}$ (if the 8202 is not performing refresh) is TAC = tp + 40 = 80 ns. To prevent a wait state at the 8086, RDY must be valid at the 8284 TCLCHmin − TCLMLmax − TR1VCLmax = 48 ns after the command is active. This implies that under worst case conditions, one wait state will be inserted for every read cycle. Since $\overline{MWTC}$ does not occur until one clock later, two wait states may be inserted for writes.

The $\overline{XACK}$ from command delay will assert RDY TCC + TCX = (tph + 3tp + 100) + (5tp + 20) = 460 ns after the command. This will typically insert one or two wait states.

Unless 2118-3's are used in 64K byte or less memories, $\overline{SACK}$ must not be used since it does not guarantee a wait state. From the previous access time analysis we saw that other configurations required a wait state.

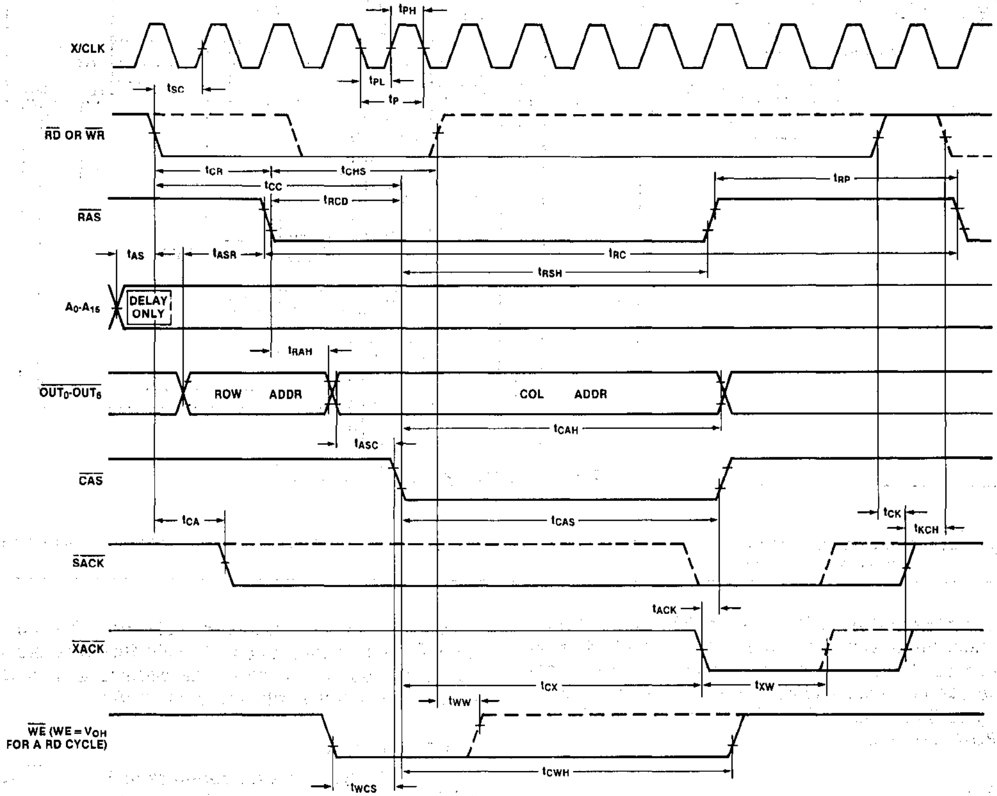


Figure 5C1.2. 8202 Timing Information

A.C. CHARACTERISTICS

$T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

Measurements made with respect to $RAS_1 - RAS_4$, CAS, WE, $OUT_0 - OUT_6$ are at 2.4V and 0.8V. All other pins are measured at 1.5V.

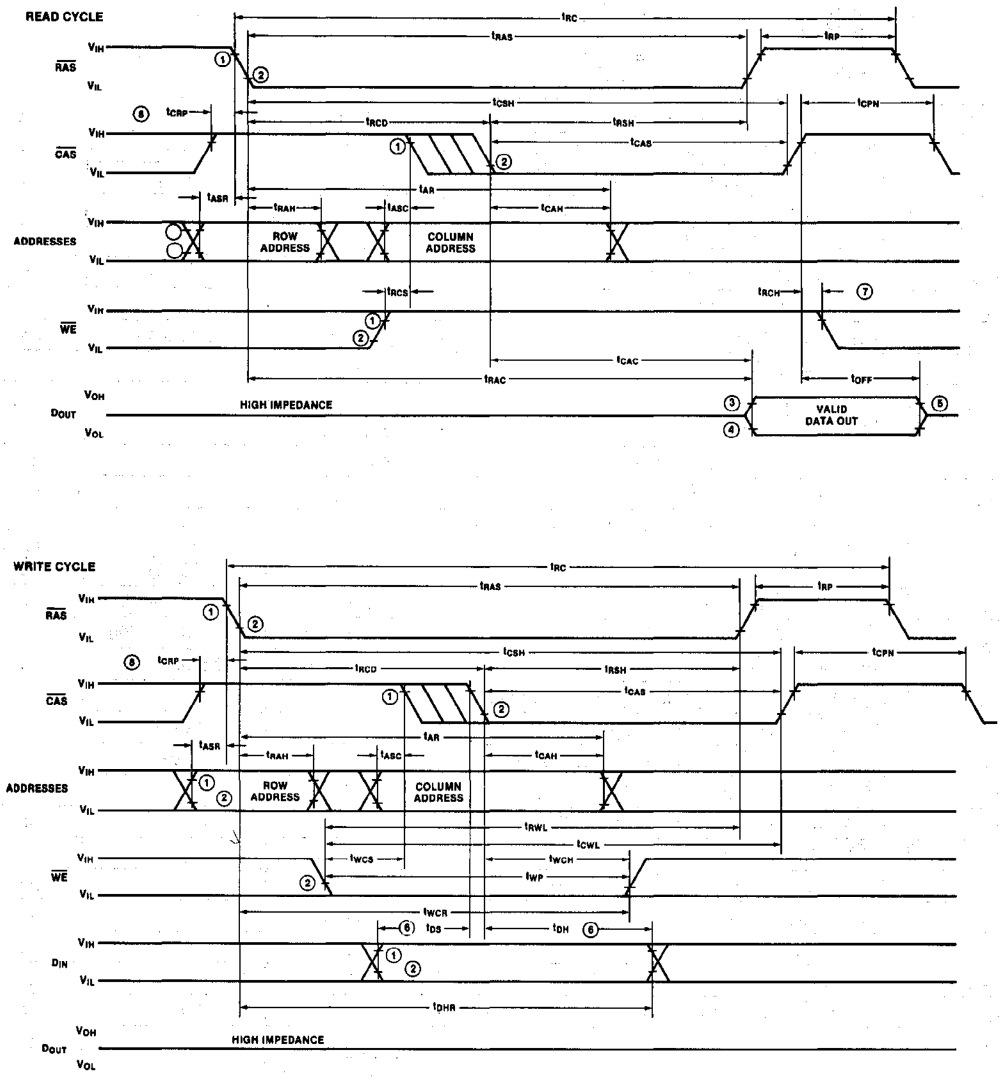
Loading:	$\overline{\text{SACK}}$,$\overline{\text{XACK}}$	CL = 30 pF
64 Devices	$\overline{\text{OUT}_0}$ – $\overline{\text{OUT}_6}$	CL = 320 pF
	$\overline{\text{RAS}_1}$ – $\overline{\text{RAS}_4}$	CL = 230 pF
	$\overline{\text{WE}}$	CL = 450 pF
	$\overline{\text{CAS}}$	CL = 640 pF

Symbol	Parameter	Min	Max	Units
t_P	Clock (Internal/External) Period (See Note 1)	40	54	ns
t_{RC}	Memory Cycle Time	$10\ t_P - 30$	$12\ t_P$	ns
t_{RAH}	Row Address Hold Time	$t_P - 10$		ns
t_{ASR}	Row Address Setup Time	t_{PH}		ns
t_{CAH}	Column Address Hold Time	$5\ t_P$		ns
t_{ASC}	Column Address Setup Time	$t_P - 35$		ns
t_{RCD}	$\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ Delay Time	$2\ t_P - 10$	$2\ t_P + 45$	ns
t_{WCS}	$\overline{\text{WE}}$ Setup to $\overline{\text{CAS}}$	$t_P - 40$		ns
t_{RSH}	$\overline{\text{RAS}}$ Hold Time	$5\ t_P - 30$		ns
t_{CAS}	$\overline{\text{CAS}}$ Pulse Width	$5\ t_P - 30$		ns
t_{RP}	$\overline{\text{RAS}}$ Precharge Time (See Note 2)	$4\ t_P - 30$		ns
t_{WCH}	$\overline{\text{WE}}$ Hold Time to $\overline{\text{CAS}}$	$5\ t_P - 35$		ns
t_{REF}	Internally Generated Refresh to Refresh Time 64 Cycle 128 Cycle	 $548\ t_P$ $264\ t_P$	 $576\ t_P$ $288\ t_P$	 ns ns
t_{CR}	$\overline{\text{RD}}$, $\overline{\text{WR}}$ to $\overline{\text{RAS}}$ Delay	$t_{PH} + 30$	$t_{PH} + t_P + 75$	ns
t_{CC}	$\overline{\text{RD}}$, $\overline{\text{WR}}$ to $\overline{\text{CAS}}$ Delay	$t_{PH} + 2\ t_P + 25$	$t_{PH} + 3\ t_P + 100$	ns
t_{RFR}	REFRQ to $\overline{\text{RAS}}$ Delay	$1.5\ t_P + 30$	$2.5\ t_P + 100$	ns
t_{AS}	A_0—A_{15} to $\overline{\text{RD}}$, $\overline{\text{WR}}$ Setup Time (See Note 4)	0		ns
t_{CA}	$\overline{\text{RD}}$, $\overline{\text{WR}}$ to $\overline{\text{SACK}}$ Leading Edge		$t_P + 40$	ns
t_{CK}	$\overline{\text{RD}}$, $\overline{\text{WR}}$ to $\overline{\text{XACK}}$, $\overline{\text{SACK}}$ Trailing Edge Delay		30	ns
t_{KCH}	$\overline{\text{RD}}$, $\overline{\text{WR}}$ Inactive Hold to $\overline{\text{SACK}}$ Trailing Edge	10		ns
t_{SC}	$\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{PCS}}$ to X/CLK Setup Time (See Note 3)	15		ns
t_{CX}	$\overline{\text{CAS}}$ to $\overline{\text{XACK}}$ Time	$5\ t_P - 40$	$5\ t_P + 20$	ns
t_{ACK}	$\overline{\text{XACK}}$ Leading Edge to $\overline{\text{CAS}}$ Trailing Edge Time	10		ns
t_{XW}	$\overline{\text{XACK}}$ Pulse Width	$2\ t_P - 25$		ns
t_{LL}	REFRQ Pulse Width	20		ns
t_{CHS}	$\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{PCS}}$ Active Hold to $\overline{\text{RAS}}$	0		ns
t_{WW}	$\overline{\text{WR}}$ to $\overline{\text{WE}}$ Propagation Delay	8	50	ns
t_{AL}	S_1 to ALE Setup Time	40		ns
t_{LA}	S_1 to ALE Hold Time	$2\ t_P + 40$		ns
t_{PL}	External Clock Low Time	15		ns
t_{PH}	External Clock High Time	22		ns
t_{PH}	External Clock High Time for $V_{CC} = 5V \pm 5\%$	18		ns

Notes:

1. t_P minimum determines maximum oscillator frequency.
   t_P maximum determines minimum frequency to maintain 2 ms refresh rate and t_{RP} minimum.
2. To achieve the minimum time between the $\overline{\text{RAS}}$ of a memory cycle and the $\overline{\text{RAS}}$ of a refresh cycle, such as a transparent refresh, REFRQ should be pulsed in the previous memory cycle.
3. t_{SC} is not required for proper operation which is in agreement with the other specs, but can be used to synchronize external signals with X/CLK if it is desired.
4. If t_{AS} is less than 0 then the only impact is that t_{ASR} decreases by a corresponding amount.

Figure 5C1.2. 8202 Timing Information (Con't)



NOTES: 1,2. $V_{IH\ MIN}$ AND $V_{IL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF INPUT SIGNALS.

3,4. $V_{OH\ MIN}$ AND $V_{OL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF D_{OUT}.

5. t_{OFF} IS MEASURED TO $I_{OUT} < |I_{OL}|$.

6. t_{DS} AND t_{DH} ARE REFERENCED TO CAS OR WE, WHICHEVER OCCURS LAST.

‡. t_{RCH} IS REFERENCED TO THE TRAILING EDGE OF CAS OR RAS, WHICHEVER OCCURS FIRST.

8. t_{CRP} REQUIREMENT IS ONLY APPLICABLE FOR RAD/CAS CYCLES PRECEDED BY A CAS-ONLY CYCLE (i.e., FOR SYSTEMS WHERE CAS HAS NOT BEEN DECODED WITH RAS).

Figure 5C1.3. 2118 Family Timing

A.C. CHARACTERISTICS[1,2,3]

$T_A = 0°C$ to $70°C$, $V_{DD} = 5V \pm 10\%$, $V_{SS} = 0V$, unless otherwise noted.

READ, WRITE, READ-MODIFY-WRITE AND REFRESH CYCLES

Symbol	Parameter	2118-3		2118-4		2118-7		Unit	Notes
		Min.	Max.	Min.	Max.	Min.	Max.		
t_{RAC}	Access Time From $\overline{\text{RAS}}$		100		120		150	ns	4,5
t_{CAC}	Access Time from $\overline{\text{CAS}}$		55		65		80	ns	4,5,6
t_{REF}	Time Between Refresh		2		2		2	ms	
t_{RP}	$\overline{\text{RAS}}$ Precharge Time	110		120		135		ns	
t_{CPN}	$\overline{\text{CAS}}$ Precharge Time (non-page cycles)	50		55		70		ns	
t_{CRP}	$\overline{\text{CAS}}$ to $\overline{\text{RAS}}$ Precharge Time	0		0		0		ns	
t_{RCD}	$\overline{\text{RAS}}$ to $\overline{\text{CAS}}$ Delay Time	25	45	25	55	25	70	ns	7
t_{RSH}	$\overline{\text{RAS}}$ Hold Time	70		85		105		ns	
t_{CSH}	$\overline{\text{CAS}}$ Hold Time	100		120		165		ns	
t_{ASR}	Row Address Set-Up Time	0		0		0		ns	
t_{RAH}	Row Address Hold Time	15		15		15		ns	
t_{ASC}	Column Address Set-Up Time	0		0		0		ns	
t_{CAH}	Column Address Hold Time	15		15		20		ns	
t_{AR}	Column Address Hold Time to $\overline{\text{RAS}}$	60		70		90		ns	
t_T	Transition Time (Rise and Fall)	3	50	3	50	3	50	ns	8
t_{OFF}	Output Buffer Turn Off Delay	0	45	0	50	0	60	ns	
READ AND REFRESH CYCLES									
T_{RC}	Random Read Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{\text{RAS}}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{\text{CAS}}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{RCS}	Read Command Set-Up Time	0		0		0		ns	
t_{RCH}	Read Command Hold Time	0		0		0		ns	
WRITE CYCLE									
t_{RC}	Random Write Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{\text{RAS}}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{\text{CAS}}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{WCS}	Write Command Set-Up Time	0		0		0		ns	9
t_{WCH}	Write Command Hold Time	25		30		45		ns	
t_{WCR}	Write Command Hold Time, to $\overline{\text{RAS}}$	70		85		115		ns	
t_{WP}	Write Command Pulse Width	25		30		50		ns	
t_{RWL}	Write Command to $\overline{\text{RAS}}$ Lead Time	60		65		110		ns	
t_{CWL}	Write Command to $\overline{\text{CAS}}$ Lead Time	45		50		100		ns	
t_{DS}	Data-In Set-Up Time	0		0		0		ns	
t_{DH}	Data-In Hold Time	25		30		45		ns	
t_{DHR}	Data-In Hold Time, to $\overline{\text{RAS}}$	70		85		115		ns	
READ-MODIFY-WRITE CYCLE									
t_{RWC}	Read-Modify-Write Cycle Time	285		320		410		ns	
t_{RRW}	RMW Cycle $\overline{\text{RAS}}$ Pulse Width	165	10000	190	10000	265	10000	ns	
t_{CRW}	RMW Cycle $\overline{\text{CAS}}$ Pulse Width	105	10000	120	10000	185	10000	ns	
t_{RWD}	$\overline{\text{RAS}}$ to $\overline{\text{WE}}$ Delay	100		120		150		ns	9
t_{CWD}	$\overline{\text{CAS}}$ to $\overline{\text{WE}}$ Delay	55		65		80		ns	9

NOTES:

1. All voltages referenced to V_{SS}.
2. Eight cycles are required after power-up or prolonged periods (greater than 2 ms) of $\overline{\text{RAS}}$ inactivity before proper device operation is achieved. Any 8 cycles which perform refresh are adequate for this purpose.
3. A.C. Characteristics assume $t_T = 5$ ns.
4. Assume that $t_{RCD} \leq t_{RCD}$ (max.). If t_{RCD} is greater than t_{RCD} (max.) then t_{RAC} will increase by the amount that t_{RCD} exceeds t_{RCD} (max.).
5. Load = 2 TTL loads and 100 pF.
6. Assumes $t_{RCD} \geq t_{RCD}$ (max.).
7. t_{RCD} (max.) is specified as a reference point only; if t_{RCD} is less than t_{RCD} (max.) access time is t_{RAC}. If t_{RCD} is greater than t_{RCD} (max.) access time is $t_{RCD} + t_{CAC}$.
8. t_T is measured between V_{IH} (min.) and V_{IL} (max.).
9. t_{WCS}, t_{CWD} and t_{RWD} are specified as reference points only. If $t_{WCS} \geq t_{WCS}$ (min.) the cycle is an early write cycle and the data out pin will remain high impedance throughout the entire cycle. If $t_{CWD} \geq t_{CWD}$ (min.) and $t_{RWD} \geq t_{RWD}$ (min.), the cycle is a read-modify-write cycle and the data out will contain the data read from the selected address. If neither of the above conditions is satisfied, the condition of the data out is indeterminate.

Figure 5C1.3. 2118 Family Timing (Con't)

5.C.2 Enhanced Operation

Two problems are evident from the previous investigation:

1) $\overline{\text{SACK}}$ timing from command will not allow reliable operation while $\overline{\text{XACK}}$ is not active early enough to prevent wait states.

2) The normal write command required to guarantee data setup is not enabled until the CPU has sampled READY thereby forcing multiple wait states during write operations.

The first problem could be resolved if an early command could be generated that would guarantee $\overline{\text{SACK}}$ was valid when READY was sampled and $\overline{\text{SACK}}$ to data valid satisfied the CPU requirements. Figure 5.C.2.1 is a circuit which provides an early read command derived from the maximum mode status. The early command is enabled from the trailing edge of ALE and disabled on the trailing edge of the normal command. The command provides an additional TCHCLmin − TCHLLmax + TCLMLmax − circuit delays = 53 ns of access time and time to generate RDY from the early command. If we go back to our previous equations, early command to valid data at the CPU is now:

TCHCLmin − TCHLLmax + 2TCLCL − TDVCLmax − buffer and circuit delays = 333 ns

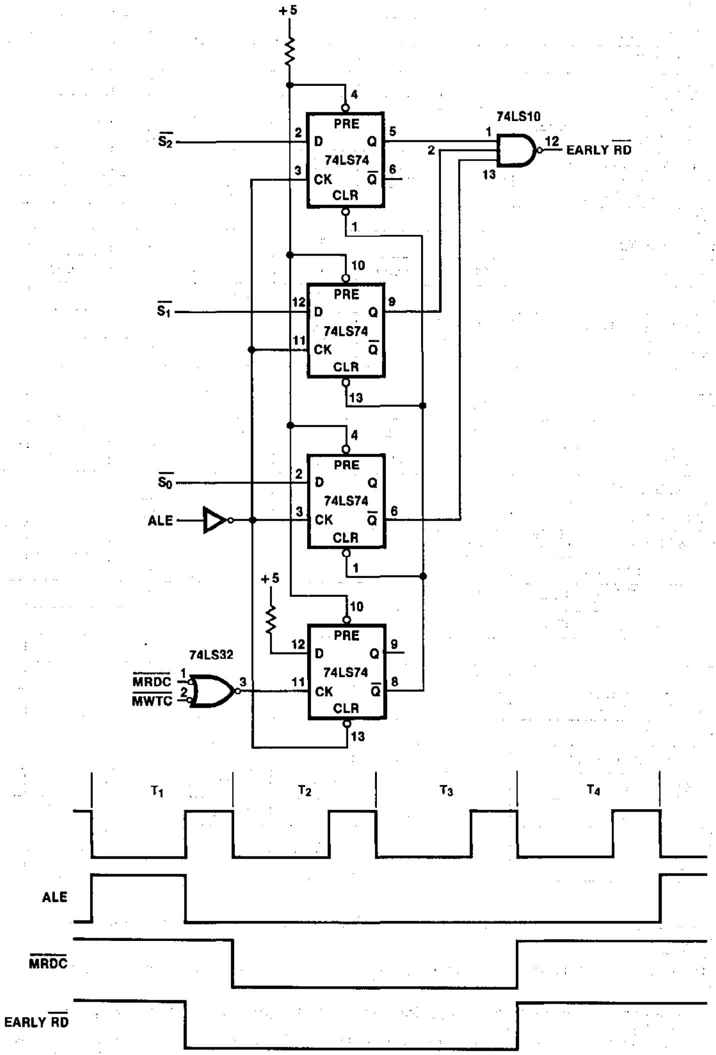


Figure 5C2.1. Early Read and Write Command Generation

We can now use the slowest 2118 which gives 8202 and 2118 access of 320 ns. Early command to RDY timing is TCLCL – TCHLLmax – circuit delays – TR1VCLmax = 115 ns and provides 35 ns of margin beyond the 8202 command to SACK delay.

The write timing of the 8202 and write data valid timing of the 8086 do not allow use of an early write command. However, if the 8202 clock is reduced from 25 MHz to 20 MHz and WE to the RAM's is gated with CAS, the advanced write command (AMWC) may be used. At 20 MHz the minimum command to CAS delay is 148 ns while the maximum data valid delay is 144 ns.

The reduced 8202 clock frequency still satisfies no wait state read operation from early read and will insert no more than one wait state for write (assuming no conflict with refresh). 20 MHz 8202 operation will however require using the 2118-4 to satisfy read access time.

Note that slowing the 8202 to 22.2 MHz guarantees valid data within 10 ns after CAS and allows using the 2118-7. Since this analysis is totally based on worst case minimum and maximum delays, the designer should evaluate the timing requirements of his specific implementation.

It should be noted that the 8202 SACK is equivalent to XACK timing if the cycle being executed was delayed by refresh. Delaying SACK until XACK time causes the CPU to enter wait states until the cycle is completed. If the cycle is a read cycle, the XACK timing guarantees data is valid at the CPU before RDY is issued to the CPU.

The use of the early command signals also solves a problem not mentioned previously. The cycle rate of the 8202 @ 20 MHz requires that commands (from leading edge to leading edge) be separated by a minimum of 695 ns. The maximum mode 8086 however may issue a read command 600 ns after the normal write command. For the early read command and advanced write command, 725 ns are guaranteed between commands.

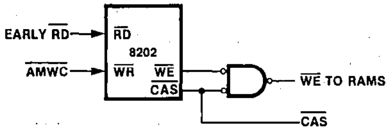


Figure 5C2.2. Delayed Write to Dynamic RAMs

APPENDIX I
BUS CONTENTION AND ITS EFFECT ON SYSTEM INTEGRITY

SYSTEM ARCHITECTURE

As higher performance microprocessors have become available, the architecture of microprocessor systems has been evolving, again placing demands on memory. For many years, system designers have been plagued with the problem of bus contention when connecting multiple memories to a common data bus. There have been various schemes for avoiding the problem, but device manufacturers have been unable to design internal circuits that would guarantee that one memory device would be "off" the bus before another device was selected. With small memories (512x8 and 1Kx8), it has been traditional to connect all the system address lines together and utilize the difference between t_{ACC} and t_{CO} to perform a decode to select the correct device (as shown in Figure 1).

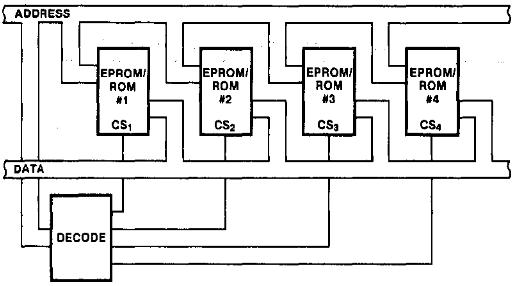


Figure 1. Single Control Line Architecture

With the 1702A, the chip select to output delay was only 100 ns shorter than the address access time; or to state it another way, the t_{ACC} time was 1000 ns while the t_{CO} time was 900 ns. The 1702A t_{ACC} performance of 1000 ns was suitable for the 4004 series microprocessors, but the 8080 processor required that the corresponding numbers be reduced to $t_{ACC} = 450$ ns and $t_{CO} = 120$ ns. This allowed a substantial improvement in performance over the 4004 series of microprocessors, but placed a substantial burden on the memory. The 2708 was developed to be compatible with the 8080 both in access time and power supply requirements. A portion of each 8080 machine cycle time had to be devoted to the architecture of the system decoding scheme used. This devoted portion of the machine cycle included the time required for the system controller (8224) to perform its function before the actual decode process could begin.

Let's pause here and examine the actual decode scheme that was used so we can understand how the control functions that a memory device requires are related to system architecture.

The 2708 can be used to illustrate the problem of having a single control line. The 2708 has only one read control function, chip select ($\overline{CS}$), which is very fast ($t_{CO} = 120$ ns) with respect to the overall access time ($t_{ACC} = 450$ ns) of the 2708. It is this time difference (330 ns) that is used to perform the decode function, as illustrated in Figure 2. The scheme works well and does not limit system performance, but it does lead to the possibility of bus contention.

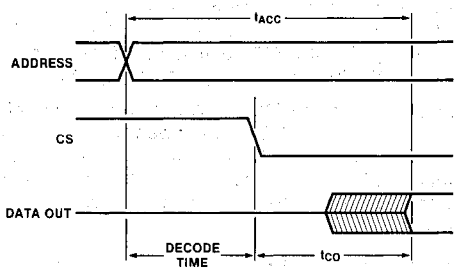


Figure 2. Single Line Control Architecture

BUS CONTENTION

There are actually two problems with the scheme described in the previous section. First, if one device in a multiple memory system has a relatively long deselect time, and a relatively fast decoder is used, it would be possible to have another device selected at the same time. If the two devices thus selected were reading opposite data; that is, device number one reading a HIGH and device number two reading a LOW, the output transistors of the two memory devices would effectively produce a short circuit, as Figure 3 illustrates. In this case, the current path is from V_{CC} on device number one to GND on device number two. This current is limited only by the "on" impedance of the MOS output transistors and can reach levels in excess of 200 mA per device. If the MOS transistors have a lot of "extra" margin, the current is usually not destructive; however, an instantaneous load of 400 mA can produce "glitches" on the V_{CC} supply—glitches large enough to cause standard TTL devices to drop bits or otherwise malfunction, thus causing incorrect address decode or generation.

The second problem with a single control line scheme is more subtle. As previously mentioned, there is only one control function available on the 2708 and any decoding scheme must use it out of necessity. In addition, any inadvertent changes in the state of the high order address lines that are inputs to the decoder will cause a change in the device that is selected. The result is the same as before—bus contention, only from a different source. The deselected device cannot get "off" the bus before the selected one is "on" the bus as the addresses rapidly change state. One approach to solving this problem would be to design (and specify as a maximum) devices with t_{DF} time less than t_{CO} time, thereby assuring that if one device is selected while another is simultaneously being deselected, there would be some small (20 ns) margin. Even with this solution, the user would not be protected from devices which have very fast t_{CO} times (t_{CO} is specified as a maximum).

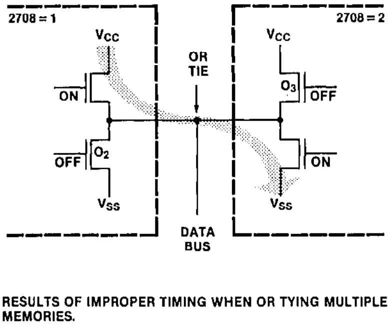


Figure 3. Results of Improper Timing when OR Tying Multiple Memories

The only sure solution appears to be the use of an external bus driver/transceiver that has an independent enable function. Then that function, not the "device selecting function," or addresses, could control the flow of data "on" and "off" the bus, and any contention problems would be confined to a particular card or area of a large card. In fact, many systems are implemented that way—the use of bus drivers is not at all uncommon in large systems where the drive requirements of long, highly capacitive interconnecting lines must be taken into consideration—it also may be the reason why more system designers were not aware of the bus contention problem until they took a previously large (multicard) system and, using an advanced micorprocessor and higher density memory devices, combined them all on one card, thereby eliminating the requirement for the bus drivers, but experiencing the problem of bus contention as described above.

THE MICROPROCESSOR/MEMORY INTERFACE

From the foregoing discussion, it becomes clear that some new concepts, both with regard to architecture and performance are required. A new generation of two control line devices is called for with general requirements as listed below:

1. Capability to control the data "on" and "off" the system bus, independent of the device selecting function identified above.

2. Access time compatible with the high performance microprocessors that are currently available..

Now let's examine the system architecture that is required to implement the two line control and prevent bus contention. This is shown in the form of a timing diagram (Figure 4). As before, addresses are used to generate the unique device selecting function, but a separate and independent Output Enable (OE) control is now used to gate data "on" and "off" the system data bus. With this scheme, bus contention is completely eliminated as the processor determines the time during which data must be present on the bus and then releases the bus by way of the Output Enable line, thus freeing the bus for use by other devices, either memories or peripheral devices. This type of architecture can be easily accomplished if the memory devices have two control functions, and the system is implemented according to the block diagram shown in Figure 5. It differs from the previous block diagram (shown in Figure 1) in that the control bus, which is connected to all memory Output Enable pins, provides separate and independent control over the data bus. In this way, the microprocessor is always in control of the system; while in the previous system, the microprocessor passed control to the particular memory device and then waited for data to become available. Another way to look at it is, with a single control line the sytem is always asynchronous with respect to microprocessor/memory communications. By using two control lines, the memory is synchronized to the processor.

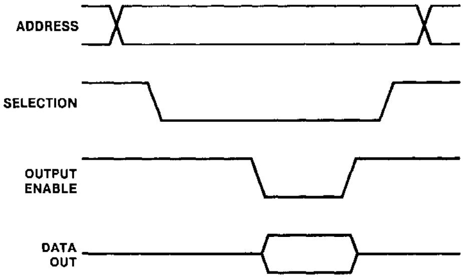


Figure 4. Two Control Line Architecture

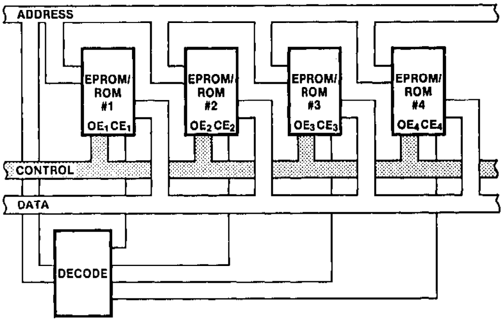


Figure 5. Two Control Line Architecture

APPLICATION
NOTE

AP-61

July 1979

Multitasking for the 8086

Cecil Moore
Applications Engineer
Microprocessor Products

Multitasking For the 8086

Contents

INTRODUCTION

Real-time software systems differ markedly from batch processing systems. An external signal indicating that it is time for an hourly log or an interrupt caused by an emergency condition is an event usually not encountered in batch processing. Because real-time control systems of all types share a number of characteristics, it is possible to develop flexible operating systems which will meet the needs of a great majority of real-time applications. Intel Corporation has developed such a system, the RMX/80™ system, for the iSBC™ line of 8080/85 based single board computers. Thus, the user is released from the chore of designing an operating system and is free to concentrate his efforts on the applications software for the individual tasks and merely integrate them into a pre-existing system.

But what if a user does not need all the capabilities of an RMX/80™ system or wants a different hardware configuration than an iSBC™ computer? This application note contains a set of PL/M-86 procedures designed to be used in medium-complexity 8086 real-time systems.

A normal control system can be broken down into a number of concurrently executable tasks. The CPU can be running only one task at any instant of time but the speed of the processor often makes concurrent tasks appear to be running simultaneously. Breaking the software functions into separate concurrent tasks is the job of the designer/programmer. Once this is done there remains the problem of integrating these tasks with a supervisory program which acts as a traffic cop in the scheduling and execution of the separate tasks. This note discusses a set of PL/M-86 procedures to implement the supervisory program function.

A minimum operating system might (like its batch processing cousin) have only a queue for ready tasks (tasks waiting to be executed). Any task that becomes ready is put on the bottom of the queue and when a running task is finished, the task on the top of the queue is started. Any interrupt causes the state of the system to be saved, an interrupt routine to be executed, the state of the system to be restored, and execution of the interrupted program to continue. The interrupt routine might (or might not) put a new task on the ready queue. This approach has worked well for many simple control systems, especially in the single-chip computer area. But what features are lacking in this approach that are necessary (or at least nice)?

1. A system of priorities is often needed. All waiting ready tasks must be executed sooner or later but some tasks need immediate attention while others can be run when there is nothing else to do. If a midnight monthly report, due for completion by 8 a.m. the next day, is in the process of printing at 1 a.m. and a fire alarm occurs, it is reasonable to assume that the fire alarm has higher priority since the fire could conceivably render the monthly report irrelevant.

There are a number of ways in which to assign priorities. Tasks are usually numbered and may be assigned priorities according to their ascending (or descending) numbers. They could instead be grouped into a number of priority levels, with tasks on the same level having equal priorities. The latter approach is taken in this application note.

Assume that a monthly report is being printed and an alarm occurs in the external world that, because of its importance, must be attended to immediately. The interrupt routine, executed as a result of the alarm input, should not automatically return to the interrupted logging routine but instead should call a preempt routine which checks to see if a higher priority task is ready for execution. The reason for this is that the monthly report routine, if returned to, has no way of "knowing" that a higher priority task is waiting to be executed. The alarm output task has been readied by the interrupt routine and since it is known to be higher priority than the logging task, it is executed first, thereby immediately signaling the system operator that there has been an alarm. It then returns to the logging task provided that there are no further high priority tasks waiting to be executed. The logging printer may not have even paused during the alarm output task. The computer appears to human beings to be executing concurrent tasks simultaneously.

Of course, the alarm output function could be performed inside the interrupt procedure. But sooner or later, the designer will encounter a worst case situation in which there is not enough time to execute all required tasks between interrupts, and the system will fall behind in real-time. It is much cleaner to make the interrupt procedures as short as possible and stack up tasks to be executed than to stack up interrupt procedures.

2. Another feature that might be necessary is a capability to put a task to sleep for a known period of real time. Assume a relay output must remain closed for one second. Most real-time systems cannot tolerate the dedication of the CPU to such a trivial task for that length of time so a system of programmable dynamic delays could be implemented. This application note implements such a system.

Although the PL/M-86 procedures here have been debugged and tested, it is assumed that the user will want to change, add, or delete features as needed. This application note is intended to present ideas for a logical structure of procedures that, because they are written in PL/M-86, can be easily modified to user requirements. Each procedure will be discussed in detail and integration and optional features will be presented.

PL/M-86

PLM-86 is a block structured high level language that allows direct design of software modules. Using PL/M-86, designers can forget their assembly level coding problems and design directly in a subset of the English language. The 8086 architecture was designed to accommodate highly structured languages and the PLM-86 compiler is quite efficient in the generation of machine code.

PLM-86 STRUCTURE

PL/M-86 automatically keeps track of the level of the different software blocks. (See Chapter 10, "PL/M-86 Programming Manual"). There are methods of writing PL/M-86 which contribute to the understandability of the source code without adding to the amount of object code generated. For instance, the following three IF/THEN/ELSE blocks generate identical object code but are compiled from different source statements.

```
Line  Level  Statement
  3     1    IF A = B THEN C = D; ELSE E = F; G = H;
  7     1    IF A = B THEN
  8     1    C = D;
             ELSE
  9     1    E = F;
 10     1    G = H;
 11     1    IF A = B THEN DO;
 13     2    C = D ;
 14     2    END;
 15     1    ELSE DO;
 16     2    E = F;
 17     2    END;
 18     1    G = H;
```

It is not instantly apparent from the code on line 3 or the code starting at line 7 which statements will be executed. However, adding the DO; and END; statements (starting at line 11) remove any doubt. Either the statements starting at line 11 or the statements starting at line 15 will be executed and the statement on line 18 will be executed in either case. Why? Because all these lines are at level 1 in the block structure. The other lines are at level 2 because of the DO;/END; combinations. When one refers to the relatively complex structures of the task multiplexer procedures, the usefulness of such an approach is obvious, as the procedures have been indented according to the level numbers generated by PL/M-86. In particular, if the designer is not careful, nested IF/THEN/ELSE statements can generate improper results. Using a proper number of DO;/END; combinations avoids the possible ambiguity in nested IF/THEN/ELSE statements as can be seen in the ACTIVATE$TASK procedure listed in the PL/M-86 source code later in this note. The DO;/END; construct naturally must be used when multiple statements are required within the IF/THEN/ELSE blocks. Following are examples of the possible primary structures of PL/M-86:

```
DO;
    A = B;
    C = D;
    END;

DO WHILE A = B;
    C = D;
    E = F;
    END;

DO I = 1 TO 5;
    A = I;
    C = D + I;
    END;

DO CASE A;
    A = B;
    A = C;
    A = D;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE DO;
    E = F;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE IF A = C THEN DO;
    D = E;
    END;
ELSE IF A = D THEN DO;
    E = F;
    END;
ELSE DO;
    F = G;
    END;
```

A complete tutorial on structured programming is beyond the scope and intent of this application note and the reader is referred to the appropriate references appearing in the bibliography.

ANATOMY OF THE TASK MULTIPLEXER

Once a decision is made on the details of the kind of data structure that is needed to implement the task multiplexer, the procedures that manipulate the structure are relatively simple to write. The following characteristics are assumed for the task multiplexer appearing in this application note.

There are two levels of priority, high and low. All high priority tasks that are ready to run will be dispatched, executed, and completed, on a FIFO basis, before any low priority task is dispatched.

Any task can be interrupted. No task multiplexer procedure can be interrupted.

If a high priority task is interrupted, it will be completed before any other task is dispatched. If a low priority task is interrupted, all ready high priority tasks will be dispatched, executed, and completed before program control is returned to the low priority task.

There are two ready queues, one for high priority tasks and one for low priority tasks. Each queue has a head (top) pointer and a tail (bottom) pointer and tasks on any queue are link-listed from head to tail. Tasks are "dispatched" (taken off the queue) at the head and "activated" (put on the queue) at the tail on a FIFO basis.

Link-listed queues are chosen for simplicity. All dispatch and activate information is contained in the head and tail pointers. Tasks located in the middle of these link-lists are of no concern for activating and dispatching. This means, of course, that tasks are executed in the order that they appear on the queue, i.e., first-in, first-out.

There is a pointer byte associated with each task. If a task is on either the low priority or high priority ready queue, its associated pointer byte will point to the next task number on the list. These pointer bytes enable the task ready lists to be linked. Note that the pointer byte is 0 for the last task on a list.

There is a status (flag) byte associated with each task. If a task is on a ready list or a delay list, bit 7 will be a "1" indicating that that particular task is busy. If a task is on either high priority or low priority ready queues, bit 6 will be a "1" indicating that the task is on one of the ready queues. If the task is listed on the delay list, (see next item), bit 5 will be a "1" indicating that this particular task has a delay in progress. If a task is unlisted, bits 5-7 will be "0." Bits 0-4 are not used by the task multiplexer procedures and are available to the user, giving 5 user defined flags per task.

There is a delay byte associated with each task. This feature allows tasks to be "put to sleep" for a variable length of time, from 1 to 255 "ticks" of the interrupt clock. If a task does not need an associated delay then this byte is available to the user as a utility byte to be used for any purpose. These delays will be discussed in detail later in the application note.

The following diagram is a representation of the task multiplexer data structure:

TASK NUMBER	POINTER BYTE	STATUS BYTE	DELAY BYTE
0	n	n+1	n+2
1	n+3	n+4	n+5
2	n+6	n+7	n+8
3	n+9	n+10	n+11
4	n+12	n+13	n+14
5	n+15	n+16	n+17
m−1	n+3m−6	n+3m−5	n+3m−4
m	n+3m−2	n+3m−1	n+3m

3m+3 TOTAL RAM BYTES
n = FIRST RAM ADDRESS OF ARRAY

Following is a chart of what a task multiplexer data structure might look like at a given moment in time:

HIGH$PRIORITY$HEAD = 5
HIGH$PRIORITY$TAIL = 3
LOW$PRIORITY$HEAD = 8
LOW$PRIORITY$TAIL = 10
DELAY$HEAD = 4

TASK NUMBER	TASK(n).PNTR	TASK(n).STATUS	TASK(n).DELAY
0	*	*	*
1	3	1100 0000	0
2	0	1010 0000	3
3	0	1100 0000	0
4	7	1010 0000	4
5	1	1100 0000	0
6	0	0000 0000	0
7	2	1010 0000	6
8	10	1100 0000	0
9	0	0000 0000	0
10	0	1100 0000	0

*See text.

What information can one ascertain from observation of the above chart? The ready-to-run high priority tasks, in order, are 5,1,3. This can be seen by following the high priority ready linked list from head to tail. The ready-to-run low priority tasks, in order are 8, 10. The TASK(n).PNTR byte = 0 for the last listed task. Tasks 4, 7, 2 are listed, in order, on the delay list and have associated delays of 4, 10, 13 ticks respectively. Tasks 6 and 9 are not listed and therefore idle. The * for the TASK (0) bytes indicate a special condition. There is no TASK00 allowed and a zero condition is treated as an error condition. TASK(0).PNTR byte is used for the DELAY$HEAD byte to minimize code in the ACTIVATE$DELAY procedure. TASK(0).STATUS and TASK(0).DELAY are unused bytes.

DEFINITIONS

NEW$TASK is the number of the task that will be installed on a ready list or the delay list when ACTIVATE$TASK or ACTIVATE$DELAY is called.

NEW$DELAY is the value of the delay that will be installed on the delay list when ACTIVATE$DELAY is called.

A task is defined as RUNNING if it is in the act of execution or if an interrupt routine is executing which interrupted a RUNNING task.

A task is defined as PREEMPTED if it has been interrupted and a higher priority task is being executed.

A task is defined as READY if it is contained within one of the ready queues.

A task is defined as IDLE if its BUSY$BIT (bit 7) is not set, i.e., it is not listed anywhere else. Note that it is possible to completely disable an IDLE task simply by setting its BUSY$BIT. In that case, it is not and cannot be listed anywhere else. This feature is useful during system integration.

STATE DIAGRAM

The state diagram indicates the relationships among the possible task states and the procedures involved in changing states.

The state diagram looks somewhat complicated and a discussion of the possible change of states is in order. Assuming a certain existing state, future possible states will be discussed including the procedures which can cause the change of state.

From the unlisted (idle) state, the ACTIVATE$TASK procedure will put the NEW$TASK on either the high priority ready queue or the low priority ready queue at the tail end of the queue. The number of the task automatically assigns the priority and therefore the proper queue. All task numbers below FIRST$LOW$PRIORITY$TASK are assumed to be high priority tasks. Also, from the unlisted state the ACTIVATE$DELAY procedure will put the NEW$TASK and NEW$DELAY at the proper position on the delay list.

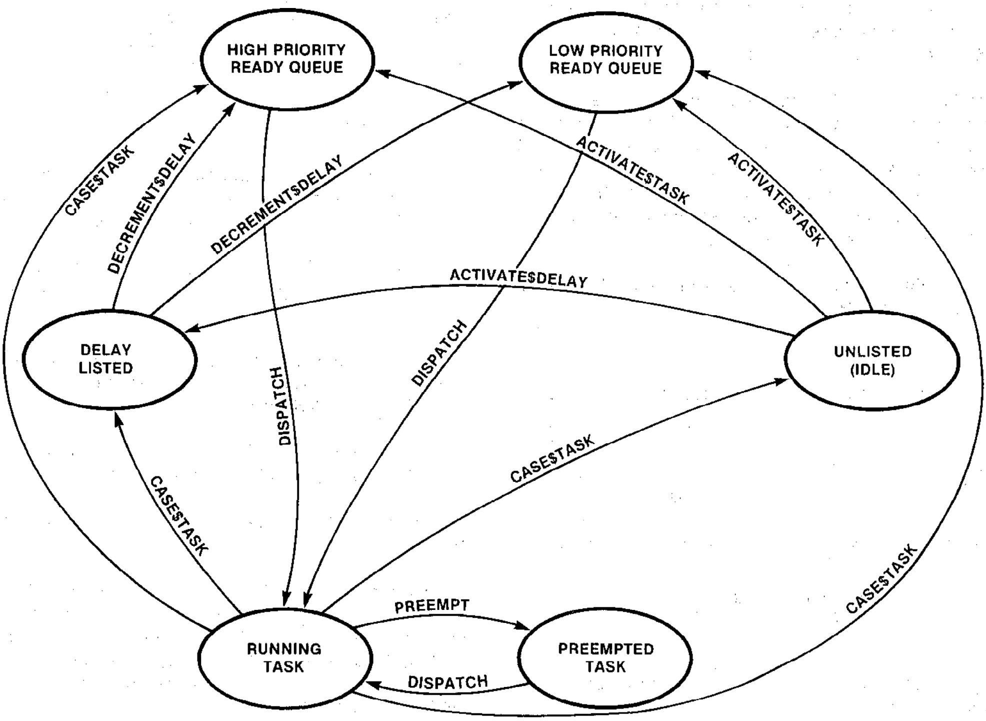
After a task has been put on either high priority ready queue or low priority ready queue it eventually will go to the RUNNING$TASK state. The DISPATCH procedure accomplishes this action.

From the delay list a task can only go to one of the ready queues. When a task's associated delay goes to zero the DECREMENT$DELAY procedure calls the ACTIVATE$TASK procedure and installs the NEW$TASK on the proper ready queue.

From the RUNNING$TASK state a task may use the CASE$TASK procedure to put itself on the ready list tail by setting NEW$TASK = RUNNING$TASK. It may instead put itself on the delay list by setting NEW$TASK = RUNNING$TASK and also setting NEW$DELAY equal to something other than zero. Otherwise, it will progress to the unlisted state upon completion.

The CASE$TASK procedure unlists tasks when they have completed execution. A low priority RUNNING$TASK will go to the preempted state if a high priority task is on the ready list following an interrupt during execution of the low priority task if the PREEMPT procedure is called.

And finally, a PREEMPTED$TASK will return to a RUNNING$TASK state when all high priority ready tasks have completed execution. This is accomplished by the DISPATCH procedure which then returns to the PREEMPT procedure.



STATE DIAGRAM

Some lockouts are necessary to avoid chaos in the task multiplexer. These are as follows:

The BUSY$BIT = 1 in the TASK(n).STATUS byte will abort the ACTIVATE$TASK and the ACTIVATE$DELAY procedures and return an indication of the aborting by setting the STATUS byte equal zero. A task must be unlisted to be able to be installed on a list.

A RUNNING$TASK may put itself on a list after it has executed but it is not allowed to re-list any listed tasks (i.e., no task may ever be listed twice at the same time!). A task that tries to activate another task that is already busy can wait (via the delay feature) for the required task to complete execution, become idle, and therefore be available to be activated. A PREEMPTED$TASK may not be listed. If the ACTIVATE$TASK or ACTIVATE$DELAY procedure is called and NEW$TASK = PREEMPTED$TASK, the procedure will be aborted and return with STATUS = 0. Otherwise, the STATUS byte is returned with the new task status.

Only one task may be preempted as there are only two levels of priority. The user may desire to implement many levels of priority in which case a linked-list of preempted tasks could be declared in a structure which includes the number of the first task in each priority level group of tasks. This obviously complicates the PREEMPT and DISPATCH procedures.

The tasks themselves are made into reentrant procedures because of the necessary forward references of the CASE$TASK procedure.

PL/M-86 allows structures and arrays of structures. The structure needed for the task multiplexer is a link-list pointer byte, a task status byte, and a task delay byte. Each task has an associated pointer byte, status byte, and delay byte. These are combined into an array of up to 255 tasks. For purposes of this discussion, the number of tasks is chosen as an arbitrary 10, leading to the following array declaration.

```
DECLARE TASK(10)STRUCTURE
(PNTR BYTE,STATUS BYTE,DELAY BYTE);
```

Thus the delay byte associated with task number 7 can be accessed by using the variable TASK(7).DELAY and the status of task number 5 can be examined through the use of TASK(5).STATUS. The TASK(n).PNTR byte contains the task number of the next listed task on the same list as TASK(n), i.e., if TASK(n) is on the delay list, then TASK(n).PNTR will contain the number of the next task on the delay list or 0 indicating the end of the list.

TASK(n).STATUS is a byte with the following reserved flags:

```
BIT 7  BUSY$BIT, "1" IF TASK IS BUSY
BIT 6  READY$BIT, "1" IF ON READY LIST
BIT 5  DELAY$BIT, "1" IF ON DELAY LIST
BIT 4 — BIT 0     UNUSED
```

The unused bits in the STATUS byte are available to the user.

The TASK(n).DELAY byte is a number which can put TASK(n) to sleep for up to 255 system clock ticks. The system clock tick is interrupt driven from the user's timer and its period is chosen for the particular application. A one millisecond timer is popular and assuming such a time, delays of up to 255 ms are available in the task multiplexer as it is written. If this delay range is not wide enough, the user may want to define his TASK(n).DELAY as a word instead of a byte in the PL/M-86 declare statement, giving delays of up to 65 seconds from the basic one millisecond clock tick.

LINKED LISTS

Linked lists are useful for a number of reasons. However, a treatise on linked lists would defeat the purpose of this application note and the reader is referred to the references listed in the bibliography.

The linked lists used in this application note have a head byte associated with each list, i.e., the head byte contains the number of the first task on the list. The first task pointer byte points to the second task on the list, etc. The pointer of the last task on the list is set at zero to indicate that it is the last task. Two of the linked lists are ready queues and require a tail byte as well as a head byte. The tail byte points to the last entry on the list. Tasks are put on the bottom, or tail, of the ready lists and are taken off the top, or head, of the ready lists. The delay list has no tail but does have a head, called a DELAY$HEAD. The delay list is not a queue, as delays are installed on the list in order of delay magnitude for reasons to be explained later.

There are two ready lists, one for high priority tasks and one for low priority tasks. The head and tail pointers associated with these two lists are: HIGH$PRIORITY$HEAD, HIGH$PRIORITY$TAIL, LOW$PRIORITY$HEAD, and LOW$PRIORITY$TAIL. Obviously, the structure can be expanded to any number of priority levels by expanding the head and tail pointers and the historical record of the preempted tasks.

DELAY STRUCTURE

A task multiplexer can have a number of simultaneous delays active and it would be efficient if there were a way to keep from decrementing all delays on every clock tick, which is most time consuming. One way to accomplish this feat is to move the problem from the DECREMENT$DELAY routine to the ACTIVATE$DELAY routine. The delays are arranged in a linked-list of ascending sizes such that the value of each delay includes the sum of all previous delays. This allows the decrementing of only one delay during each clock tick interrupt routine. An example will further illuminate this approach. Suppose the following conditions exist:

Task 7 has a 5 millisecond delay

Task 3 has an 8 millisecond delay

Task 9 has a 14 millisecond delay

The delay structure is arranged so that:

DELAY$HEAD = 07
TASK(7).PNTR = 03
TASK(3).PNTR = 09
TASK(9).PNTR = 00
TASK(7).DELAY = 05 (FIRST DELAY = 5)
TASK(3).DELAY = 03 (5 + 3 = 8)
TASK(9).DELAY = 06 (5 + 3 + 6 = 14)

The linked-list is arranged so that the delays are in ascending order and each delay is equal to the sum of all previous delays up through that point. Since this is true, all delays are effectively decremented merely by decrementing the first delay. Of course, something for nothing is impossible and the speed gained by arranging the delays in the above manner is paid for by the complexity of the ACTIVATE$DELAY routine. But since the ACTIVATE$DELAY routine is executed less frequently than the DECREMENT$DELAY routine, the savings in real time is worth the added complexity.

Suppose a new delay is to be activated in the above scheme. Task 5 with a delay of 10 milliseconds is to be added. A before and after chart will indicate what the ACTIVATE$DELAY procedure must accomplish.

BEFORE

TASK NUMBER		07	03	09
POINTER	07	03	09	00
DELAY		05	03	06

AFTER

TASK NUMBER		07	03	05	09
POINTER	07	03	05*	09@	00
DELAY		05	03	02@	04*

FIRST POINTER IS THE DELAY$HEAD
CHANGES ARE MARKED WITH AN *
ADDITIONS ARE MARKED WITH AN @

Note that the pointer before the added task has changed and the delay after the added task has changed. The function of the ACTIVATE$DELAY procedure is to accomplish these changes and additions.

PROCEDURES

The following procedure explanations reference the PL/M-86 source code listing which follows the application note text.

ACTIVATE$TASK Procedure

This procedure is initiated by a call instruction with the byte NEW$TASK containing the number of the task to be put on the proper ready queue.

Interrupts must be disabled whenever the link-lists are being changed. If interrupts are enabled when this procedure is called, they should be re-enabled upon returning.

The assignment of priority is a simple matter. A declare statement, DECLARE FIRST$LOW$PRIORITY$TASK LITERALLY 'N,' (where N is the actual number of the first low priority task) indicates to the procedures that tasks 1 to N are high priority tasks and tasks N or higher are low priority tasks.

This procedure checks the busy bit in the status byte to see if this particular task is already busy and if so, returns a STATUS of zero. Otherwise, it returns the new STATUS of the task. It then checks the priority to see if this particular task is a high or low priority. If it is high priority, then the task pointer pointed to by the HIGH$PRIORITY$TAIL pointer is changed from zero to the number of the NEW$TASK. The HIGH$PRIORITY$TAIL pointer is then changed to the number of the NEW$TASK and the pointer associated with NEW$TASK is made equal to zero. This completes the ACTIVATE$TASK functions. If the new task is a low priority task, then the same functions are performed using the LOW$PRIORITY$TAIL pointer.

ACTIVATE$DELAY Procedure

This procedure is initiated by a call with the byte NEW$TASK containing the number of the task to be put on the delay list and the byte NEW$DELAY containing the value of the associated delay.

Interrupts are disabled and the busy bit of this particular task is checked. If the busy bit is set the STATUS byte is set to zero and the procedure returns without activating the delay. If the busy bit is not set the integer value DIFFERENCE is set equal to the NEW$DELAY value. POINTER$0 is set equal to the DELAY$HEAD. POINTER$1 is set to zero. The DO WHILE loop executes until POINTER$0 equals zero or DIFFERENCE is less than zero. Remember that the proper place to insert the new delay is being searched for, and that will be either at the end of the list (POINTER$0 = 0) or when the sum of the previous delays do not exceed the new delay value. The DO WHILE loop has POINTER$0, POINTER$1, OLD$DIFFERENCE, and DIFFERENCE keeping track of where the procedure is in the loop, while searching for the proper place to insert the new delay. The existing delays are sequentially subtracted from the remains of NEW$DELAY according to the link-listed order until the end of the list or a negative result is encountered indicating that the proper delay insertion point has been reached. At this point POINTER$0 contains the task number to be assigned to TASK(NEW$TASK).PNTR. POINTER$1 contains the task number immediately preceding the NEW$TASK such that TASK(POINTER$1). PNTR = NEW$TASK and our link list is fully updated, with the actual delays yet to go. If POINTER$0 = 0 it means that the new delay is larger than any of the other delays and therefore should go on the end of the list so TASK(NEW$TASK).DELAY is set equal to the DIFFERENCE. If

POINTER$0 is not equal to zero then if POINTER$0 equals POINTER$1 (indicating that there were not any delays previously listed), then TASK(POINTER$1).PNTR is set equal to zero. TASK(NEW$TASK).DELAY is set equal to the OLD$DIFFERENCE and TASK (POINTER$0).DELAY is set equal to the negative of DIFFERENCE which at this point is negative, thereby resulting in a positive unsigned number. The reader is encouraged to implement an example (see Delay Structure section) to prove that the above approach is valid. Particular attention should be paid to the contents of the two pointers, as they are the key to the procedure. The final function of this procedure is to set the BUSY$BIT and DELAY$BIT in the TASK(NEW$ TASK).STATUS byte. The byte named STATUS which is returned by this procedure is set equal to the status of the new task. If it is desired to have interrupts enabled, they must be enabled after the procedure return instruction. The reason for such a complex method of activating a delay will become apparent in the following section.

DECREMENT$DELAY Procedure

The first delay on the linked-list is decremented and, if it is zero, the associated task is put on the appropriate ready queue. The next delay (if any) is checked to see if it is zero and if so, that task is put on the appropriate ready queue, etc. A loop is performed until either no delay or a non-zero delay is found. The procedure then returns.

It is assumed that this procedure is part of an interrupt routine and that the interrupts are disabled during its execution. Interrupts cannot be enabled during changes to any of the linked-lists or else recovery may not be possible.

This procedure begins by checking to see if there are any active delays. If DELAY$HEAD = 0 then this procedure returns immediately. Otherwise it decrements the first delay. If this delay goes to zero then the associated task number is passed to the ACTIVATE$ TASK procedure as the OFF$DELAY byte. A new DELAY$HEAD is chosen from the next link-listed delay and that delay checked for a value of zero which will happen if the first two or more delays are equal. This loop is accomplished by the DO WHILE DELAY$ HEAD <> 0 AND TASK(DELAY$HEAD).DELAY = 0; This procedure is designed to require very little CPU time unless a delay times out. The DO WHILE loop is bypassed if the resulting delay value is not zero. A certain amount of care should be exercised to insure that many delays do not all time out at the same time. One method would be to modify the ACTIVATE$DELAY procedure to insure that there are no zero entries in the delay bytes. The basic procedure, however, assumes that the clock "tick" timing will be chosen to minimize the above potential problem.

CASE$TASK Procedure

This procedure performs the function of calling the task indicated by the contents of the RUNNING$TASK byte. All listed tasks are called in this manner. The CASE$TASK procedure is called by the DISPATCH procedure. When a particular task has completed execution it returns to the CASE$TASK procedure which then resets the BUSY$BIT and the READY$BIT and returns to the DISPATCH procedure after setting RUNNING$TASK equal to zero. This procedure allows a task to relist itself immediately upon returning from execution.

PREEMPT PROCEDURE

The PREEMPT procedure is called whenever it is possible that a high priority task has been put on the ready queue while a low priority task was in the process of execution. An example will illustrate:

Assume that the control system is being interrupted by the 60 Hz line frequency and a register is being incremented each time this 16.67 ms edge occurs. When the register gets to 60 (indicating that one second has passed), the register is zeroed and the high priority timekeeping task is put on the ready queue. Assume also that a low priority data logging task was running when this interrupt occurred. The interrupt routine calls PREEMPT. If a high priority task is running, PREEMPT simply returns. But in our example, a low priority task is running so PREEMPT transfers RUNNING$TASK to PREEMPTED$TASK and calls DISPATCH, which calls CASE$TASK, which calls the time-keeping task. When the time-keeping task has completed, it returns to CASE$TASK which returns to DISPATCH which returns to the PREEMPT procedure which returns to the interrupt routine which returns to the interrupted low priority data logging task if no other high priority tasks are on the ready queue. If the high priority ready queue is not empty, any and all high priority tasks will be completed before the interrupted routine is returned to. PREEMPT refuses to return to the interrupt routine until HIGH$ PRIORITY$HEAD is equal to zero. It is important to note that a low priority task will not be preempted unless the PREEMPT procedure is called. As noted above, it is normally called from the interrupt routine which interrupted the low priority task, but there is nothing to prohibit PREEMPT from being called from inside a low priority task procedure.

DISPATCH PROCEDURE

This procedure calls a high priority task if HIGH$ PRIORITY$HEAD is not equal to zero, restores a preempted task if PREEMPTED$TASK is not equal to zero, calls a low priority task if LOW$PRIORITY$HEAD is not equal to zero, and simply returns if there is nothing to do, all in order of priority. The DISPATCH procedure is called from the main program loop which must enable interrupts as DISPATCH disables interrupts as soon as it is called. It is also called by the PREEMPT procedure. RUNNING$TASK must be 0 when this procedure is called.

PL/M-86 PROCEDURES

Because the block structure and levels are so important to the understanding of the following procedures, they have been indented according to level. This was a simple task accomplished by no indenting for level one, indenting once for level two, etc. The resulting attractive, easy to follow format was worth the effort to increase the initial level of understanding for readers of this application note who are not intimately familiar with PL/M.

Everything except the very simple main program loop has been made into procedures. Interrupt routines and tasks are also procedures. Keeping track of interrupts, calls, and returns is easy for PL/M and a violation of the block structure through such devices as GOTO targets outside the procedure body is the best way the author knows to crash and burn. Honor the power of the structure, accept the limitations involved, and checkout and debugging will be a pleasure.

Since CASE$TASK references the individual tasks, the task procedure structure was included in the PL/M-86 compilation. All the user has to do is insert the particular task code in place of the /*TASKnn CODE*/ comment, define the interrupt procedures and the system should be ready to run. Obviously, the user will desire to change the total number of tasks and the number of the FIRST$LOW$PRIORITY$TASK.

INITIALIZATION AND THE MAIN LOOP

The last entry in the PL/M-86 program is the initialization process which essentially zeros the task multiplexer data and the main loop which loops until TRUE = FALSE, i.e. forever, with interrupts enabled. The STATUS = STATUS instruction simply insures that the loop can be interrupted as the instruction following an ENABLE instruction is not interruptible.

These few instructions are included for information only and will need to be expanded considerably for use in a real-world system. The task multiplexer procedures were checked out on an iSBC 86/12™ computer running under random interrupt control and these instructions were the minimum necessary to cause the system to run. As was stated earlier, the following source code does not include any interrupt procedures and these will have to be generated following the format explained in the PL/M-86 programming manual.

ADDITIONAL IDEAS

Resource allocation is a feature that could be added to the task multiplexer. To keep it simple and yet avoid the deadlock problem (two tasks each grab a resource that the other needs), an extra array can be added to the TASK(n).XXX structure in which each bit in the byte (or word), represents a resource necessary for the execution of a task. A RESOURCES$STATUS byte can then keep the dynamic busy status of the system resources (printers, terminals, floating point math packages, etc.). When the CASE$TASK procedure is called, the resources required by the next RUNNING$ TASK can be compared to the RESOURCES$STATUS byte to see if the required resources are available. If they are, the following PL/M-86 statement will update the new status of the resources:

```
RESOURCES$STATUS = RESOURCES$STATUS OR
        TASK(RUNNING$TASK).RESOURCES;
```

However, if the resources are not available, the CASE$ TASK procedure can return the task to the ready or delay list and try again later. When the task has completed, the following PL/M-86 statement will update the resources status byte:

```
RESOURCES$STATUS = RESOURCES$STATUS AND NOT
        TASK(RUNNING$TASK).RESOURCES;
```

Message passing from task to task may also be necessary. Assuming that a task will have only one message at a time to deliver or receive, another byte could be added to the task structure such that TASK(RUNNING$TASK).MESSAGE could represent a byte containing the number of the task wishing to deliver a message to the RUNNING$TASK. Since a task can call CASE$TASK which in turn will call another task, message block parameters can be passed directly from one task to another. The task that calls CASE$TASK must handle the necessary housekeeping involved in recovering after the message has been passed. Of course, the data structure would have to be expanded to accommodate the message parameters and blocks. For further ideas involving message handling refer to the RMX/80™ user's guide.

Two additional relatively simple procedures could be added to obtain the SUSPEND and RESUME features of the RMX/80™ system. Remember that if the BUSY$BIT is set in a TASK(n).STATUS byte and the task is unlisted, then it cannot be listed. If it is desired to dynamically enable and disable a task, this bit could be set by a SUSPEND procedure and reset by the RESUME procedure.

SOURCE CODE

```
TM86:DO;

DECLARE TOTAL$TASKS LITERALLY '10';
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '0';
DECLARE BUSY$BIT LITERALLY '10000000B';
DECLARE READY$BIT LITERALLY '01000000B';
DECLARE DELAY$BIT LITERALLY '00100000B';
DECLARE FIRST$LOW$PRIORITY$TASK LITERALLY '6';

DECLARE TASK(TOTAL$TASKS) STRUCTURE(PNTR BYTE, STATUS BYTE, DELAY BYTE);
DECLARE HIGH$PRIORITY$HEAD BYTE, HIGH$PRIORITY$TAIL BYTE;
DECLARE LOW$PRIORITY$HEAD BYTE, LOW$PRIORITY$TAIL BYTE;
DECLARE RUNNING$TASK BYTE, PREEMPTED$TASK BYTE;
DECLARE STATUS BYTE, NEW$TASK BYTE, NEW$DELAY BYTE;
DECLARE DELAY$HEAD BYTE AT (@TASK(0).PNTR);

ACTIVATE$TASK: PROCEDURE; /* ASSUMES NEW$TASK<>0 */
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      IF NEW$TASK < FIRST$LOW$PRIORITY$TASK THEN DO;
         IF HIGH$PRIORITY$TAIL<>0 THEN DO;
            TASK(HIGH$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE HIGH$PRIORITY$TAIL=0 THEN */ DO;
            HIGH$PRIORITY$HEAD=NEW$TASK;
            END;
         HIGH$PRIORITY$TAIL=NEW$TASK;
         END;
      ELSE /* SINCE TASK IS LOW PRIORITY THEN */ DO;
         IF LOW$PRIORITY$TAIL<>0 THEN DO;
            TASK(LOW$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE LOW$PRIORITY$TAIL=0 THEN */ DO;
            LOW$PRIORITY$HEAD=NEW$TASK;
            END;
         LOW$PRIORITY$TAIL=NEW$TASK;
         END;
      TASK(NEW$TASK).PNTR=0;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
             BUSY$BIT OR READY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   RETURN;
   END ACTIVATE$TASK;
```

```
ACTIVATE$DELAY: PROCEDURE;/*ASSUMES NEW$TASK, NEW$DELAY<>0*/
   DECLARE POINTER$0 BYTE, POINTER$1 BYTE;
   DECLARE OLD$DIFFERENCE INTEGER, DIFFERENCE INTEGER;
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      DIFFERENCE=INT(NEW$DELAY);
      POINTER$0=DELAY$HEAD;
      POINTER$1=0;
      DO WHILE POINTER$0<>0 AND DIFFERENCE>0;
         OLD$DIFFERENCE=DIFFERENCE;
         DIFFERENCE=DIFFERENCE-INT(TASK(POINTER$0).DELAY);
         IF DIFFERENCE>0 THEN DO;
            POINTER$1=POINTER$0;
            POINTER$0=TASK(POINTER$1).PNTR;
            END;
         END;
      TASK(NEW$TASK).PNTR=POINTER$0;
      TASK(POINTER$1).PNTR=NEW$TASK;
      IF POINTER$0=0 THEN TASK(NEW$TASK).DELAY=LOW(UNSIGN(DIFFERENCE));
      ELSE /* SINCE DIFFERENCE<0 THEN */ DO;
         IF POINTER$0=POINTER$1 THEN TASK(POINTER$1).PNTR=0;
         TASK(NEW$TASK).DELAY=LOW(UNSIGN(OLD$DIFFERENCE));
         TASK(POINTER$0).DELAY=LOW(UNSIGN(-DIFFERENCE));
         END;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
             BUSY$BIT OR DELAY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   NEW$DELAY=0;
   RETURN;
   END ACTIVATE$DELAY;

DECREMENT$DELAY: PROCEDURE;    /* ASSUMES INTERRUPTS DISABLED */
   DECLARE OFF$DELAY BYTE;
   IF DELAY$HEAD<>0 THEN DO;
      TASK(DELAY$HEAD).DELAY=TASK(DELAY$HEAD).DELAY-1;
      DO WHILE DELAY$HEAD<>0 AND TASK(DELAY$HEAD).DELAY=0;
         OFF$DELAY=DELAY$HEAD;
         DELAY$HEAD=TASK(DELAY$HEAD).PNTR;
         TASK(OFF$DELAY).STATUS=TASK(OFF$DELAY).STATUS
             AND NOT(BUSY$BIT OR DELAY$BIT);
         NEW$TASK=OFF$DELAY;
         CALL ACTIVATE$TASK;
         END;
      END;
   RETURN;
   END DECREMENT$DELAY;
```

```
CASE$TASK: PROCEDURE REENTRANT;
   DO CASE RUNNING$TASK;
      CALL TASK00;
      CALL TASK01;
      CALL TASK02;
      CALL TASK03;
      CALL TASK04;
      CALL TASK05;
      CALL TASK06;
      CALL TASK07;
      CALL TASK08;
      CALL TASK09;
      END;
   TASK(RUNNING$TASK).STATUS=TASK(RUNNING$TASK).STATUS AND
            NOT (BUSY$BIT OR READY$BIT);
   TASK(RUNNING$TASK).PNTR=0;
   IF RUNNING$TASK=NEW$TASK THEN DO;
      IF NEW$DELAY<>0 THEN DO;
         CALL ACTIVATE$DELAY;
         END;
      ELSE /* SINCE NEW$DELAY=0 */ DO;
         CALL ACTIVATE$TASK;
         END;
      END;
   RUNNING$TASK=0;
   RETURN;
   END CASE$TASK;


PREEMPT:PROCEDURE REENTRANT; /* ASSUMES INTERRUPTS DISABLED */
   IF PREEMPTED$TASK=0 THEN DO;
      IF (HIGH$PRIORITY$HEAD<>0) AND (RUNNING$TASK>=
            FIRST$LOW$PRIORITY$TASK) THEN DO;
         PREEMPTED$TASK=RUNNING$TASK;
         RUNNING$TASK=0;
         DO WHILE PREEMPTED$TASK<>0;
            CALL DISPATCH;
            END;
         END;
      END;
   RETURN;
   END PREEMPT;
```

```
DISPATCH:PROCEDURE REENTRANT; /* ASSUMES RUNNING$TASK=Ø */
   DISABLE;
   IF HIGH$PRIORITY$HEAD<>Ø THEN DO;
      RUNNING$TASK=HIGH$PRIORITY$HEAD;
      HIGH$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF HIGH$PRIORITY$HEAD = Ø THEN HIGH$PRIORITY$TAIL = Ø;
      CALL CASE$TASK;
      END;
   ELSE IF PREEMPTED$TASK<>Ø THEN DO;
      RUNNING$TASK=PREEMPTED$TASK;
      PREEMPTED$TASK=Ø;
      END;
   ELSE IF LOW$PRIORITY$HEAD<>Ø THEN DO;
      RUNNING$TASK=LOW$PRIORITY$HEAD;
      LOW$PRIORITY$HEAD=TASK(RUNNING$TASK).PNTR;
      IF LOW$PRIORITY$HEAD = Ø THEN LOW$PRIORITY$TAIL = Ø;
      CALL CASE$TASK;
      END;
   ELSE RETURN;
   RETURN;
   END DISPATCH;
```

```
TASK00: PROCEDURE REENTRANT;/*ERROR CODE*/RETURN;END TASK00;

TASK01: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK01 CODE*/
   DISABLE;
   RETURN;
   END TASK01;

TASK02: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK02 CODE*/
   DISABLE;
   RETURN;
   END TASK02;

TASK03: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK03 CODE*/
   DISABLE;
   RETURN;
   END TASK03;

TASK04: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK04 CODE*/
   DISABLE;
   RETURN;
   END TASK04;

TASK05: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK05 CODE*/
   DISABLE;
   RETURN;
   END TASK05;

TASK06: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK06 CODE*/
   DISABLE;
   RETURN;
   END TASK06;

TASK07: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK07 CODE*/
   DISABLE;
   RETURN;
   END TASK07;
```

```
TASK08: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK08 CODE*/
   DISABLE;
   RETURN;
   END TASK08;

TASK09: PROCEDURE REENTRANT;
   ENABLE;
             /*TASK09 CODE*/
   DISABLE;
   RETURN;
   END TASK09;

             /*INITIALIZE*/

DISABLE;
DO STATUS=0 TO 9;
   TASK(STATUS).PNTR=0;
   TASK(STATUS).STATUS=0;
   TASK(STATUS).DELAY=0;
   NEW$TASK,NEW$DELAY=0;
   HIGH$PRIORITY$HEAD,HIGH$PRIORITY$TAIL=0;
   LOW$PRIORITY$HEAD,LOW$PRIORITY$TAIL=0;
   RUNNING$TASK,PREEMPTED$TASK=0;
   END;

             /* MAIN LOOP */

DO WHILE TRUE<>FALSE;
   CALL DISPATCH;
   ENABLE;
   STATUS=STATUS;
   END;


END TM86;
```

REFERENCES

1. Hansen, Brinch, *Operating System Principles,* Prentice-Hall, Englewood, N.J., 1973.
2. Knuth, D. E., *The Art of Computer Programming,* Addison-Wesley, Reading, Mass., 1969.
3. Wirth, Nicklaus, *Algorithms + Data Structures = Programs,* Prentice-Hall, Englewood, N.J., 1976.
4. "PL/M-86 Programming Manual," Intel Corporation, 1978, manual order number 9800466A.
5. "RMX/80 User's Guide," Intel Corporation, 1977, manual order number 9800522B.

intel

APPLICATION NOTE

AP-50

September 1979

Debugging Strategies and Considerations for 8089 Systems

Debugging Stragegies and Considerations for 8089 Systems

Contents

Our thanks to John Atwood and Dave Ferguson, the authors of this note. Both John and Dave are members of Intel's 8089 design engineering group. Please direct any questions you may have to your local Intel FAE (field application engineer) or to MPO Marketing at Intel, Santa Clara.

INTRODUCTION

The Intel 8089 is the first integrated I/O processor available. This I/O processor (IOP) makes available the power of I/O channels, as used in mainframes and minicomputers, in a microcomputer form. Designed as part of the MCS-86™ family, the IOP can be interfaced with the MCS-80™ and MCS-85™ families as well.

An I/O channel is basically a processor remote from the main CPU, which independently runs I/O operations upon command of the CPU. To relate the 8089 to existing LSI components, it is similar to a microprocessor that is time-multiplexed with a DMA controller, but with two channels available. However, since the 8089 processor is optimized for I/O and multiprocessor operations, and the DMA has been made much more flexible than existing DMA controllers, a truly general purpose and powerful I/O control system is available on one chip.

Due to the uniqueness of the 8089, this application note was written to review debugging strategies and point out possible pitfalls when developing an IOP system. Debugging an IOP system is very similar to debugging microprocessor/DMA controller systems, and many of the techniques described here are standard microprocessor techniques. However, several factors are present which can complicate the debugging process:

1. Multiprocessor Operation

Although usable by itself, the IOP is designed to be used with other processors. All factors normally encountered with multiprocessor operation, including bus arbitration, processor communication, critical code sections, etc., must be addressed in the design and debug of an IOP system.

2. DMA Tie-in to IOP Program Execution

The relationship between IOP program execution and DMA transfers and termination is different from earlier DMA controllers and should be fully understood to properly run the system.

3. Dependency of Programs on Real-Time I/O Operations

Requirements by I/O devices for maximum data rates and minimum latency times force the software programmer to be aware of hardware timing constraints and can complicate program debugging.

4. Dual Channel Operation

Related to multiprocessor operation and real-time dependencies, the two independent channels available on the 8089 may have to be coordinated with each other to make the whole system function. Dependence of one channel on the other can also complicate debugging.

Due to the complexities of running in a real-time environment, as many steps as possible should be taken to facilitate debugging. A major help here is to make sure as much of the hardware and software as possible is working before running real-time tasks. This is a good practice anyway, but it should be reemphasized that a complex multichannel system can quickly get out of hand if more than a few things are not right.

An aid to debugging any system is a clean, well organized system design. The 8089 lends itself to structured, modular software interfaces to the host CPU, via the linked-list initialization structure, and parameter communication through the parameter block (PB) area. Some of the aspects of structured programming that aid debugging are:

- *Top Down Programming* — The functions done by low-level routines are well understood, and the number of program fixes, which can cause more errors, is minimized.
- *Program Modularity* — Small, easy to manage subprograms can be debugged independently, increasing the chance that the entire system will work the first time.
- *Modular Remoteness* — By having all program modules communicate only through a well-defined interface, one module's knowledge of the "inner workings" of another is minimized. System software complexity is reduced. Updates to program modules are more reliable, too.

Two major areas of debugging will be outlined here — static (or functional) debugging in which the hardware and software are not tested in a real-time environment, and real-time debugging. Applying a logic analyzer to IOP debugging will also be explained, and a review of IOP operation and potential problems will be done.

STATIC (OR FUNCTIONAL) DEBUGGING

The predominant errors in a system, when first tried out, are either errors in implementation (i.e., wrong hookups or coding errors), or an incorrect implementation (a wrong assumption somewhere). Most of these bugs can be found through static debugging techniques that are usually easier to work with than real-time testing.

Hardware Testing

Static hardware testing is done mainly to see if all individual parts of the system work, so the whole system will "play" when run. The level of testing can run from checking for continuity and shorts (which finds only hookup errors) to trying to move data around and running I/O devices from a monitor or special test programs (which can also find incorrect circuit design). In all but the simplest systems, the latter approach is recommended since it is a step towards software debugging.

Several approaches to hardware testing will be covered. Running diagnostic programs (such as a monitor) out of the IOP's host system, in both the LOCAL and REMOTE modes, will be covered. The case where the host system cannot support diagnostic software and must have an external processor to exercise the IOP and its peripherals will also be explained.

The case where the host system can run diagnostics or test programs that have interactive user I/O, such as a CRT terminal or teletype, provides the most straightforward way to test the IOP. Naturally, before these programs can be run, the basic hardware must be correct enough to run programs. When this point is reached, a monitor program can be used to exercise memory and I/O controllers on the system bus.

It should be mentioned that aids, other than just testing with software, are helpful for hardware debugging. While a necessity for real-time debugging, a logic analyzer is also a definite help for static hardware debugging. Its main use in hardware debugging is showing timing relationships between address or data paths and other signals. It is especially useful for functional software debugging, to be described shortly. The last debugging section outlines the use of an analyzer with the IOP. Of course, an oscilloscope, logic probes and pulsers, etc., can be used to trace out specific logic or timing problems.

LOCAL Mode

When the IOP is running in the LOCAL Mode, all I/O controllers and memory are accessible by the host or controlling CPU. Thus a standard monitor, such as the one supplied with the SDK-86 or available for the iSBC-86/12$^{\text{TM}}$ development kit, can exercise all hardware on the bus.* The breakpoint routines, however, will not work due to the different instruction set. The 8086 or 8088 is best suited for running the IOP in the LOCAL mode due to identical status lines and bus timing, as well as the Request/Grant line, which eliminates bus arbitration hardware. Figure 1 shows the general LOCAL mode configuration.

REMOTE Mode

From a system design standpoint, running the IOP in the REMOTE Mode is advantageous in that it removes the I/O bus cycles from the system bus. Normally, the remote I/O is not accessible to the host CPU. Until the IOP is able to run its own test programs to transfer data from the REMOTE bus to the system bus, I/O controllers and memory on the REMOTE bus will be invisible to the host. To get around this problem during prototyping, either an external processor interface can be used (see next section), or a temporary bypass can be made to access the REMOTE bus from the system bus.

Bypassing the normal REMOTE/SYSTEM interface is a handy technique for doing preliminary debugging on the REMOTE bus. This can be done by memory-mapping the IOP's I/O space into an unused portion of the host CPU's system memory space. When accessing this space, the IOP access to its own I/O space is disabled, and a separate set of address buffers, transceivers and bus control signal buffers are enabled. Reads and writes can then be done to the formerly inaccessible REMOTE bus by the host CPU.

A simple system (Figure 2) implements this bypassing scheme. It was designed for just forcing or examining devices on the REMOTE bus and may not read or write correctly if the IOP is simultaneously trying to do bus cycles. A more sophisticated arbitration system would permit reliable run-time checking also.

*The SDK-86 serial monitor is a good basis for a general 8086 monitor. The IOP cannot be used directly with the SDK-86, since the 8086 is running in the minimum mode. The SDK-86 can be converted to run in the maximum mode, if desired.

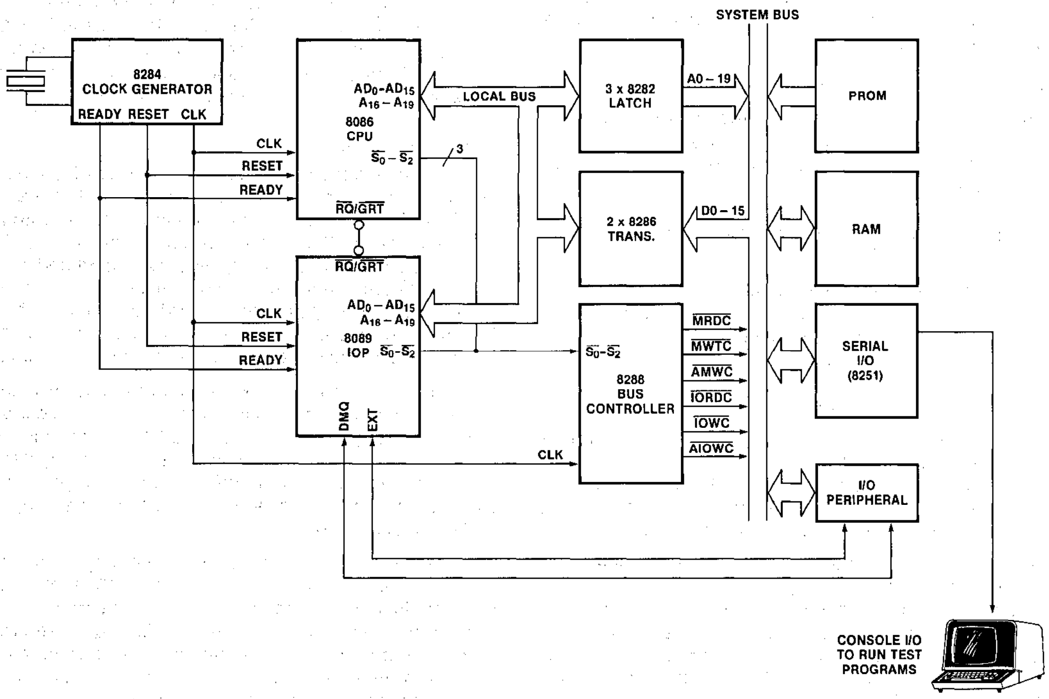


Figure 1. Generalized LOCAL Configuration—8086 in Max Mode

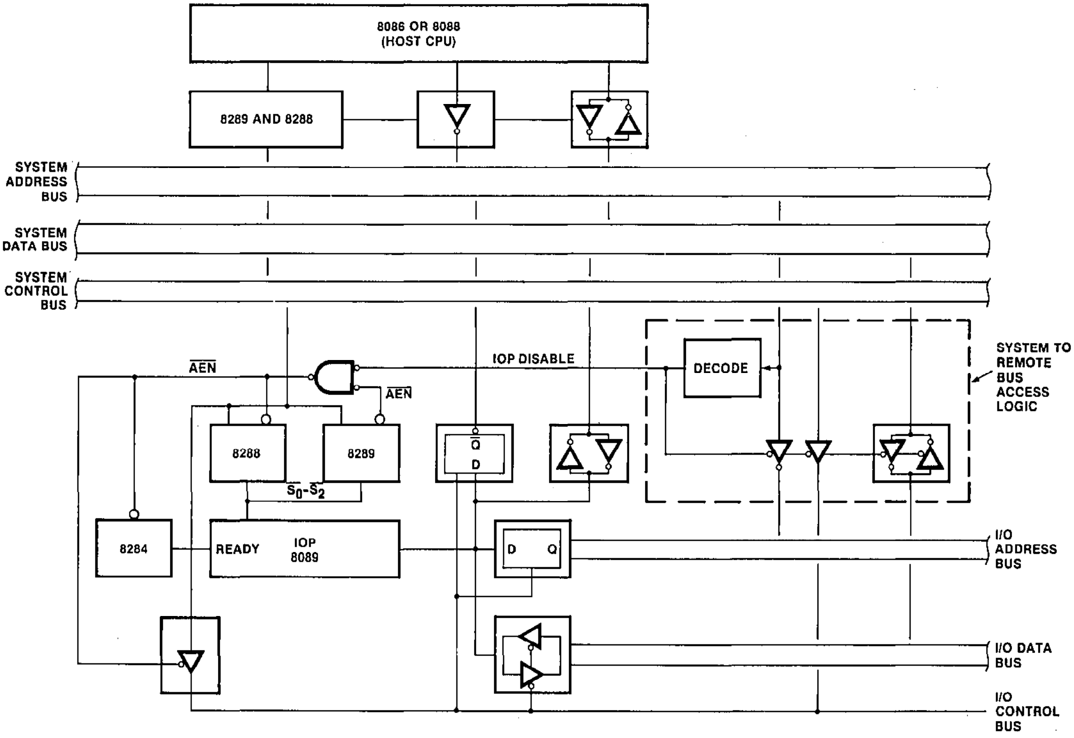


Figure 2. Remote Mode Bypass for Debugging

Running the IOP in the REMOTE mode, particularly if the MULTIBUS™ protocol is adhered to, has the advantage that the IOP can be exercised with any MULTIBUS-compatible processor. If the main processor is not amenable to being used as a debugging tool, another processor could be used to debug the hardware interface. If the microprocessor is of the same type as the intended host processor, software debugging can be done as well. A generalized REMOTE mode configuration using the MULTIBUS is shown in Figure 3.

External Processor Interface

A technique that can be used if the host processor cannot run any debugging or monitor routines is to have an external processor tie into the host processor's bus. This is useful if the main system CPU cannot run an interactive monitor or other debugging programs. If a MULTIBUS interface is being used, an 8289 bus arbiter and a set of address/data/control buffers can be used. A somewhat simpler system, similar to the remote bus access system mentioned above, could be used for static debugging of non-MULTIBUS systems. Again, if true bus arbitration is added (which brings us nearly to a MULTIBUS interface), it could also be used for run-time testing. Intel processors that have the MULTIBUS interface include the iSBC-80/20™, iSBC-86/12™, iSBC-80/10™, iSBC-80/05™, the Intellec® development systems, among others.

In the previously described systems, the external processor would disable the host CPU's access to the bus, either by some form of bus request or by a "brute force" disabling of the CPU's buffers. In the latter case, the external processor could only control the bus during a time that the CPU is halted, without destroying the program flow. Mapping the processor's memory space into the external processor memory space is the simplest method, but can impact programs being run on the external processor. If the processor under test utilizes the MULTIBUS interface (with bus arbitration), then a processor like the iSBC-80/30™ or iSBC-86/12™ could be used as the debug vehicle with no special hardware. A more flexible interface that would have less impact on the system memory space would have the addresses for the system under test generated from latches loaded by the I/O instructions from the external processor. This case must have software routines to interface to the I/O ports and handle the desired debugging routines (see Figure 4).

Software Testing

It is desirable to check as much of the IOP program as possible statically, since various tools and techniques are available which may not be usable during real-time

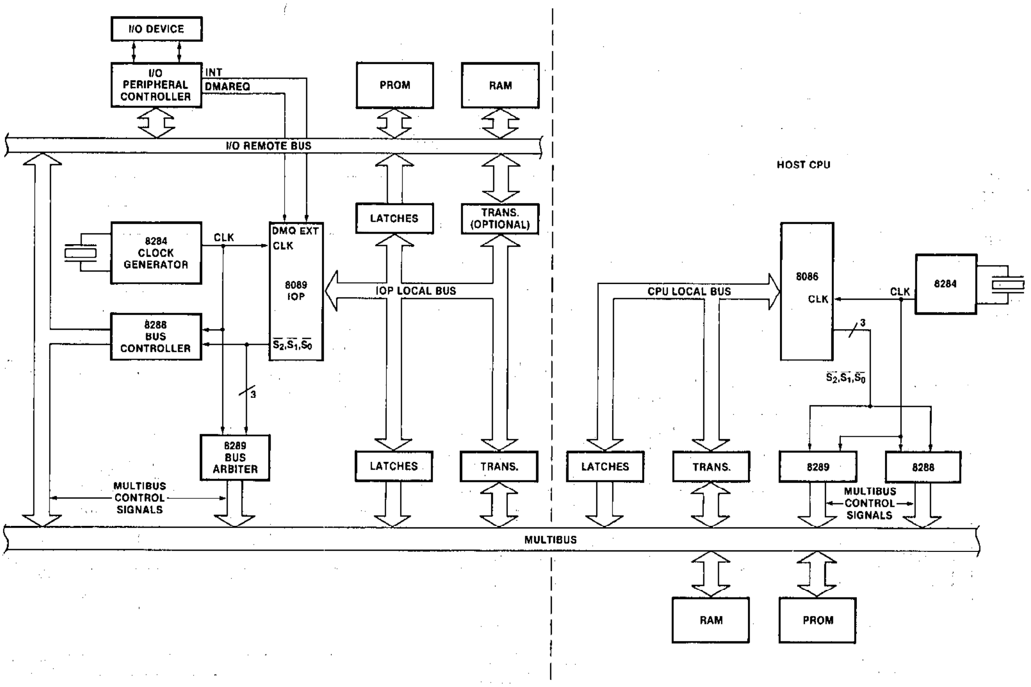
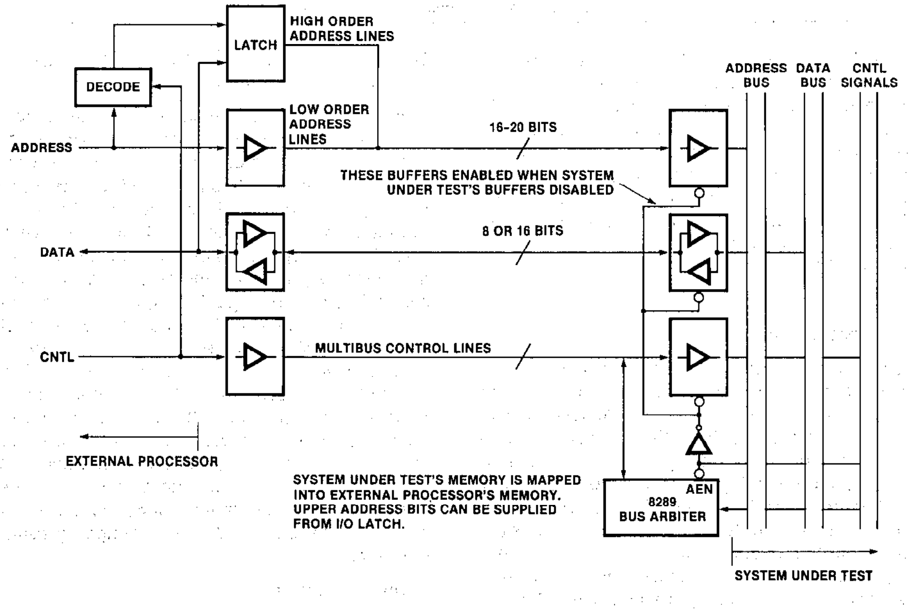


Figure 3. Generalized Remote Bus Using MULTIBUS Interface



SYSTEM UNDER TEST'S MEMORY IS MAPPED INTO EXTERNAL PROCESSOR'S MEMORY. UPPER ADDRESS BITS CAN BE SUPPLIED FROM I/O LATCH.

Figure 4. External Processor Interface

testing. This "static" software testing is not applicable to heavily I/O-dependent or DMA-dependent routines, but is best suited to longer computational or data handling routines. The idea is to test the correctness of algorithms, rather than seeing if the whole system runs.

There are two main approaches to functional software testing. One is to essentially run the program in real time and monitor program flow on a logic analyzer. The difference between this and real-time testing is that program subsections can be tested separately by using different TP (Task Pointer) starting addresses. If it is necessary to set up certain registers or parameters in memory, a small "setup" program can be run after initialization, which can load up registers or memory, then jump to the program section desired.

Another technique is to run the programs with breakpoint routines so that one can step through code segments and follow program execution. Software breakpoints are usually implemented by inserting a jump or restart to a monitor routine at the breakpoint location. This jump or restart is machine language dependent so, unfortunately, the existing breakpoint routines within monitors for the 8080 or 8086 are not applicable.

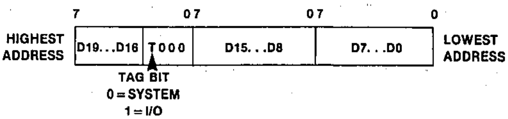
New routines tailored to the 8089 can be used, and, if done properly, can even be used to examine programs running on a REMOTE bus. Using breakpoints is somewhat complicated on the 8089 because the minimum instruction length is two bytes. There is no absolute CALL instruction, only a relative one (which would have to have its displacement recalculated each time it was used). But, with a several-byte absolute jump inserted at each place a breakpoint is desired, full breakpoint capabilities can be obtained.

There are many ways the breakpoints can be implemented. When a breakpoint is reached, the 8089 itself could output the machine state to a console through its own routines. Better suited to debugging, though, is a system that has the 8089 place its machine state in memory, alert the host processor, and then halt. The host then picks up the 8089's state and can treat it in the same way it runs its own breakpoint routines. Since the host processor is more likely to be running a monitor or some other kind of debugging routine (and most likely has at least temporary console I/O), it is the logical system to initiate and examine 8089 breakpoints. If the IOP is running in the REMOTE mode, and the host processor has access to the I/O bus via the scheme mentioned in the hardware debugging section, then IOP programs running on the REMOTE bus can be examined.

The breakpoint itself can consist of an escape sequence that is used to save the TP value and jump to the save routine, or just a jump to the save routine. This routine saves all register contents for the channel the breakpoint is in, signals the host processor, and stops the IOP. All user programmable registers (GA, GB, GC, IX, MC, BC, TP), as well as the pointer tags, are accessible. The PP (Parameter Pointer) and PSW are not normally accessible, but if the generation of the CA is such that the IOP can send itself a CA, then by sending a CA HALT, the PSW will appear at PP+3. Remember that since the IOP doesn't have arithmetic or logical condition codes, the PSW is not as important as in other machines.

The most straightforward way to pass data from the IOP to the host processor is through the PB (Parameter Block) area since the PP will normally remain relatively fixed throughout the IOP program. In order not to infringe on the PB areas used by the programs, an area 18 bytes long should be allocated at the end of the PB block to hold the register contents. Using other areas to store the register data requires saving and reloading a pointer register as part of the breakpoint escape sequence.

The data returned from the breakpoint save routine will appear to the host processor as a sequential block of data in the PB area. Sixteen-bit data can easily be extracted, but 20-bit pointer data will have to be reconstructed from the move pointer (MOVP) format:



Several means are available to signal the host processor that a breakpoint has been reached. A bit could be set in memory or an interrupt sent to the CPU. The best way, though, is to use the BUSY flag (at CP+1 or CP+9). After starting the IOP, the BUSY flag is set to FF. When a breakpoint is reached, the IOP performs its save routine and does either a software or CA HALT. These result in clearing the BUSY flag, which then signals the CPU to obtain valid breakpoint data. The CPU can then restart the IOP by either a CA START or CA CONTINUE.

The breakpoint routine outlined above will work for a "one-shot" test. However, to be more useful as a general purpose debugging tool, some refinements must be added. To keep from destroying the program whenever a breakpoint is placed, the supervisory program running from the host processor must save the IOP code that is occupied by the escape sequence. When the breakpoint is completed and IOP execution is to resume, the host program restores the IOP code, sets the TP in the CB area back to where the breakpoint was placed, and sends a CA START. Since the length of each instruction can be easily found from bits 1-4 of the opcode, a single stepping function can also be done.* By the time this is implemented, the host program is becoming a full-fledged debugging routine. Appendix 3 describes a debugging program that makes use of the ideas presented here.

Breakpoint routines can be quite useful, but some restrictions and limitations should be mentioned. The processor examining the breakpoints must have access to the IOP program memory, either directly, or through IOP programs that simulate direct access. The program memory must be in RAM. The breakpoint must be

*The formula for length of instructions is: length (in bytes) = 2 + 1 (if bits 1,0 = 01) + 1 (if bits 3,2 = 01) + 2 (if bit 3 = 1) + 2 (if LPDI).

placed on an instruction boundary, and multiple breakpoints must not be placed so that they overlap. There may be some impact on the PB area. CA generation may have to be different than usual. But, despite these limitations, the breakpoints offer a useful and more conventional software debugging tool than analyzers.

REAL-TIME TESTING

Running an IOP program in its final environment with real I/O devices is the true test of dynamic operation. The program is no longer in a static, isolated environment. The demands of DMA and multiprocessing may reveal unplanned timing dependencies or critical section problems. There may also be sections of hardware or software, which couldn't be tested statically, that may have bugs. The whole purpose of static or functional testing is to dig these problems out while convenient debugging tools can be used. Since there are no simple techniques for real-time debugging, the use of a logic state analyzer and techniques to fully understand the IOP's real-time operation will be emphasized.

Multiprocessing operations and real-time asynchronous I/O requests can cause the timing complexity of the system as a whole to rise beyond the point of complete comprehension by an individual. It is then essential that techniques to ensure correctness are used. These include good design methods, especially a clean, well-structured design, as well as good testing. A thorough test requires the attitude that the system should be tested for failures, rather than tested for correctness. In other words, one should try to make the system fail, tests should be chosen that will put the worst stress on critical timing areas.

The best way to do this is to write a diagnostic program that puts the CPU, IOP, and I/O devices through the worst conceivable timing and program combinations. Ideally, the program should be self-checking so that it can be run without supervision, printing any data or program errors that occur, much like a memory test.

The two main real-time problem areas are insufficient data rates or latency, and critical section problems. To test for data rate problems, run the system clock at its lowest expected frequency and use memory and I/O with maximum expected wait states. Identify the tightest program timings and try to have these sections coincide with worst case DMA or other heavy bus utilization (see dual channel operation later). Critical section problems can occur when two independent processors communicate with each other with improper "handshaking." This can result in one processor missing another's message, or even having both processors hang up, waiting for each other to go ahead. The 8089 provides aids to these problems, including the TSL instruction (to implement semaphores) and the BUSY flag. However, any interprocessor communication (including one channel of the IOP to the other) should be checked. Beware of cases when one processor is running considerably slower than the other (due to DMA overhead or chained instruction sequences).

The techniques for real-time debugging evolve from functional testing using a logic analyzer. For all but the simplest systems, an analyzer is essential, since it can graphically show program execution and timing relationships during real-time execution. Another aid is a delayed oscilloscope. Triggering the scope from the logic analyzer, the delay can be adjusted so that any signal in the system can be monitored.

To facilitate the use of the logic analyzer, especially if its memory is not very deep or when using it to trigger an oscilloscope, a repetitive system can be used to continually update the display. Using a repetitive reset helps to debug the software-hardware interface, since oscilloscope or logic analyzer probes can be readily moved around the circuit to observe new signals without manually retriggering the display. At its simplest, the reset to the host processor can be strobed, say every 10 ms. The processor will then provide the two channel attentions (CAs) that are needed to initialize the IOP. Where this isn't feasible, the CAs can be externally forced by either a string of one-shots or a simple processor with timing loops (such as a SDK-85 or SDK-86). See Figure 5 for initialization timing.

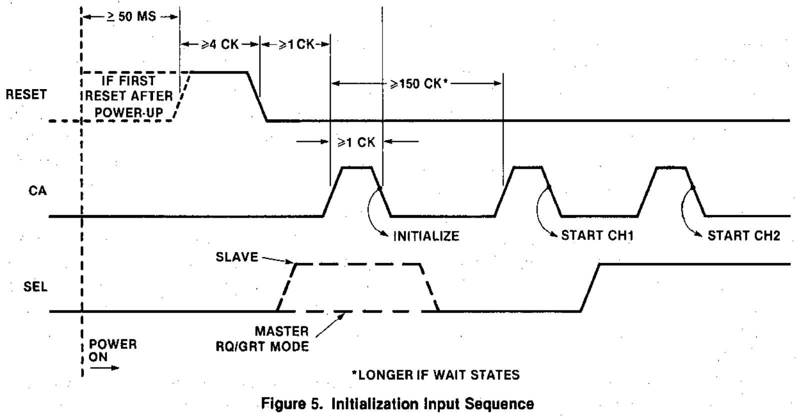


Figure 5. Initialization Input Sequence

Memory protection of the IOP and system programs is helpful when debugging DMA operation. It is quite easy for runaway DMA to wipe out memory. Another precaution to avoid this problem is to set an upper limit on the number of bytes transferred by always specifying a byte count termination.

Logic Analyzer Techniques

In the absence of other powerful debugging systems, the logic analyzer has shown to be an extremely useful tool. Because of its importance in debugging an IOP system, some basic techniques and observations that relate to monitoring IOP operation will be reviewed here. The particular brand or type of analyzer used is not too important, but would be desirable to have the following features:

- At least a 24-bit data width
- Flexible triggering and qualification control
- Display after triggering on a sequence of states
- Capability for hexadecimal data display

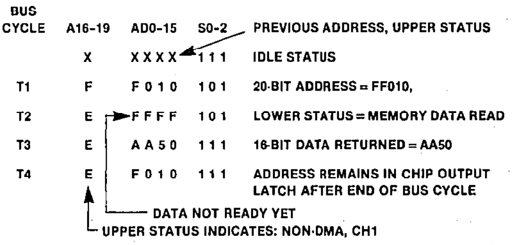
It is best to hook up to the address/data lines at the IOP, as opposed to looking at the separate address and data lines, since 39 lines would be required just to look at address, data and status lines. The three lower status lines should be monitored to show the type of bus cycle being run. Other lines can be connected where needed, at places like the DRQ lines, the EXT lines or other lines related to the system.

For general purpose debugging, triggering the analyzer on the rising edge of the IOP clock shows the most useful data concerning bus cycles. Of course, using the falling edge may be necessary to check certain signals, particularly ones that are active only while the clock is low. The following discussion is based on sampling data on the clock's rising edge.

One should be careful when setting up the triggering for the analyzer that the desired event is what is displayed and not a later event with the same trigger word. This can happen when the logic analyzer is in the repetitive trigger mode. It may retrigger before the system actually resets. A sequence restart feature is helpful.

The basis of following program execution and DMA on a logic analyzer is to follow an 8089 bus cycle, which is identical to a 8086 and 8088 bus cycle. The following diagram shows a typical 8089 bus cycle.

For general purpose debugging, displaying every clock is useful, but for quickly finding one's way around a program, the analyzer can be qualified so that only instruction fetches (status = 100 or 000), with ALE active, are trapped. A much more compact display of execution flow results.



```
 BUS
CYCLE  A16-19  AD0-15   S0-2   PREVIOUS ADDRESS, UPPER STATUS
          X     X X X X  1 1 1  IDLE STATUS
 T1       F     F 0 1 0  1 0 1  20-BIT ADDRESS = FF010,
 T2       E     F F F F  1 0 1  LOWER STATUS = MEMORY DATA READ
 T3       E     A A 5 0  1 1 1  16-BIT DATA RETURNED = AA50
 T4       E     F 0 1 0  1 1 1  ADDRESS REMAINS IN CHIP OUTPUT
                                LATCH AFTER END OF BUS CYCLE
                  DATA NOT READY YET
               UPPER STATUS INDICATES: NON-DMA, CH1
```

As mentioned earlier, on a 16-bit bus, most instructions starting on odd addresses won't show the first fetch, since the internal queue is in use. It is a good idea in that case to use only even instruction boundaries as trigger words. When following dual channel operation, one should keep an eye on the upper status bits (S3-S6), since S3 indicates which channel is running (0 = CH1, 1 = CH2), and S4 indicates DMA/non-DMA transfer (0 = DMA, 1 = non-DMA).

A REVIEW OF IOP OPERATION
(With things to look out for)

When trying to get an unfamiliar system going for the first time, it is too easy to stumble on apparent problems that are really just unexpected operation modes or peculiarities of the machine. For this reason the basic principles of IOP operation will be reviewed here with special emphasis on possible problem areas or pitfalls that a user might encounter when debugging a 8089 system. The topics are covered generally in the order encountered when bringing up a system. For complete details of operation and some design examples, see the 8086 Family User's Manual.

RESET

RESET must be active (HIGH) for at least four clocks in order to fully initialize all internal circuitry. On power up, RESET should be held high for at least 50 microseconds. The chip is only ready to accept a Channel Attention (CA) one clock after RESET goes inactive.

Note that the SEL pin is sampled on the falling edge of the first CA after RESET to tell the 8089 whether it is a master (0) or a slave (1) for its request/grant circuitry. If a master, it will assume it has the bus from the beginning. If a slave, it will strobe the $\overline{RQ}/\overline{GT}$ Line to request the bus back and will not start any bus transfers until it has been granted the bus. If the $\overline{RQ}/\overline{GT}$ line is not being used, make sure the IOP comes up in the master mode.

Initialization

Upon the first CA after reset, a sequence of instructions is executed from an internal ROM. These instructions pick up parameters and load data from the linked list sequence (Figure 6). The instruction sequence is essentially:

MOVB SYSBUS from FFFF6
LPD System Configuration Block (SCB) from FFFF8
MOVB SOC from (SCB)
LPD Control Pointer (CP) from (SCB) + 2
MOVBI "00" to CP + 1 (clears BUSY flag)

Remember that four bytes must be fetched during an LPD. If on a 16-bit bus, with even addressed boundaries, only two fetches are needed. Otherwise (8-bit bus or odd boundaries), four fetches are needed.

Even though no bus cycles are run to fetch these instructions, the CH1 Task Pointer (TP) appears on the address latches during the short internal fetch periods. On power up, this value is meaningless, but if a repetitive RESET is used, the TP remains unchanged from the end of the last program run. See Figure 6 for the start of a typical initialization sequence as viewed on a logic analyzer.

Bit 0 in the SYSBUS field sets the actual (or physical) system bus width that the IOP expects. In the 8-bit mode, only byte accesses are made, and all 8-bit data should appear on the lower eight data lines. In the 16-bit mode, word accesses can be made (if the address is even), all data on even addresses appears on the lower eight data lines, and all data at odd addresses appears on the upper eight.

Bit 0 in the SOC field sets the physical width for the I/O bus. The same rules for the system bus apply here. Note that these bits should reflect the actual hardware implementation and are not to be confused with the DMA logical widths set by the WID instruction.

The R bit (bit 1) in the SOC field is used to change the mode of the $\overline{RQ}/\overline{GT}$ circuitry. When the IOP is on the same bus as an 8086, it is required to have the R bit be 0, with the 8086 as the master and the 8089 as the slave.

CA	A_{19}-A_0	S_3-S_0	T	COMMENTS
1	FFFFF	111		Trigger CLK ↑
1	FFFFF	111		
	FFFFF	111		
	FFFFF	111		
	E0000	111		Bus un-tristated
	E0000	111		
	FFC6D	111		
10				TP to latch
CK	FFC6D	111		
	FFFF6	101	T1	Address loaded to latch
	EFFFF	101	T2	Data not ready yet (nothing on bus)
	EFF01	111	T3	SYSBUS loaded into chip (01)
	EFFFF	111	T4	Nothing on bus
	EFFF6	111		After bus cycle, address remains in latch
	EFFF6	111		
	FFC6D	111		TP is loaded to latch, even though fetches are from internal ROM
14				
CK	FFC6D	111		
	FFFF8	101	T1	Address to latch
	EFFFF	101	T2	
	EFFF0	111	T3	1st 2 bytes of LPD data fetched (FFF0)
	EFFFF	111	T4	
	EFFF8	111		
	EFFF8	111		
	EFFF8	111		
	EFFF8	111		
	FFFFA	101		
	EFFFF	101		
	EFFFA	111		2nd 2 bytes of LPD data fetched (FFFA)
6				
CK	EFFFA	111		
	FFC6D	111		

Figure 6. Start of Initialization Sequence On a 16-Bit Bus

The master (8086 or 8088) can never take the bus away from the slave (8089); only the slave can give back the bus. In other words, during DMA transfers, the 8089 would not have the bus taken away. This is the only mode compatible with the 8086 or 8088.

When two IOPs are being used on the same bus, the $\overline{RQ}/\overline{GT}$ circuity can be put into an equal priority mode by setting the R bit to one. A slave can only be granted the bus if the master is doing unchained instructions or running idle cycles. The master can request the bus back from the slave at any time. The slave grants it if doing unchained instructions or if it is idling. The master and slave are put on essentially the same priority.

At the end of initialization, the "BUSY" flag of CH1 is cleared. For systems where the 8086 is waiting for the initialization sequence to end before giving another CA, it can set the BUSY flag high prior to initialization. The BUSY flag going low is a sign that the IOP is ready for another CA. It is important to remember that the IOP will not respond to, nor latch, a CA during an initialization sequence.

Channel Attentions

The main system processor initiates communications with the IOP through the Channel Attention (CA) line. As mentioned earlier, the first CA after system RESET initializes the IOP. All subsequent CAs cause the IOP to do a two-step process. It first fetches the Channel Control Word (CCW) from the appropriate channel at (PP) for channel 1 or (PP + 8) for channel 2. (SEL at the time of CA falling determines the channel for all following actions.) The lower three bits of the CCW Command Field (CF) are examined and then cause the IOP to execute the desired function.

Command Field (CF)

Control of task block programs is accomplished through the command field. The various CF functions are:

CF

000 — Examine other field only and set BUSY flag

001 — Start task program in I/O space

011 — Start task program in system memory

The start command causes the following instructions to be executed out of the internal ROM:

LDP CP from (CP) + 2 (CH1) or + 10 (CH2)
LDP TP from (PP) (for TP in system) or
MOVB TBP from (PP) (for TBP in I/O)
MOVBI "FF" to (CP) + 1 or + 9 (set BUSY flag)

111 — HALT channel. BUSY flag cleared to "00"

110 — HALT channel. Save state of machine and clear BUSY flag by executing:

MOVP TP to (PP)
MOVB PSW to (PP) + 3
MOVBI "00" to (PP) + 1 or + 9

The channel will HALT and the machine will continue execution on the other channel or go to idle if the other channel is idle.

101 — Continue channel. The channel is revived after a HALT by executing:

MOVP TP from (PP)
MOVB PSW from (PP) + 3
MOVBI "FF" to (CP) + 1 or + 9
(set BUSY flag)

Do not do a CONTINUE after initialization without doing a CA START first since the (PP) register in CH1 is used as a temporary register (to hold SCB) and is only correctly loaded by a CA START.

The upper 5 bits in the CCW will have affect if CF = 000 or upon a CA START. Some things to note about these upper fields are:

- *Priority Bit* — If both channels are doing tasks of the same overall priority, the tasks with the higher priority bit will run. If the priority bits are the same, execution will alternate between the two channels.
- *BLL Bit (Bus Load Limit)* — Keeps nonchained instructions from occurring more often than once every 128 clocks. However, channel attention or termination cycles, even on the other channel, may disrupt the exact time interval to the next instruction.

It should be noted that the setting or clearing of the BUSY flag occurs after the loading or storing of registers, so that in a system where the main CPU uses the BUSY flag as a form of semaphore to tell when the IOP is truly finished, there is no danger that the SCB, CP, PP or TP could be changed before the IOP loads them.

Also since DMA termination cycles and chained instruction execution have a higher priority than CA, it is possible for CA to be "shut-out" by these higher priorities running on the other channel. However, since CA is always latched (except during initialization), it won't be forgotten.

How Can a Channel be Halted?

Sometimes a channel may stop its operation unexpectedly. To see what could cause this, and to show the impact of halting a channel, the various ways of stopping a channel are explained:

HALTED CHANNEL — If the channel has never started after initialization, if it has received a CA HALT command or a software HALT, channel operation is suspended. If the other channel can run, it will, otherwise idle cycles will run. Only a CA START or CONTINUE can resume operation.

WAITING FOR A DMA REQUEST — If the channel is in a source or destination synchronized DMA transfer mode, it will wait until DRQ is active before running its synchronized transfer. To minimize the impact on the overall throughput of the chip, the other channel can run during these DRQ wait periods.

WAITING TO GET THE BUS BY $\overline{RQ}/\overline{GT}$ — If the IOP has given the bus away via $\overline{RQ}/\overline{GT}$, it won't initiate any bus transfers until it has the bus back. The machine will run up to just before T1 of a bus clock cycle and will three-state its address/data and status pins until it has been granted the bus.

WAITING FOR READY — When running bus transfers, READY is sampled at T3 of a busy cycle. If inactive, the whole chip will wait until READY goes active.

The last two cases of waiting (or "wait" states) stop the whole chip and do not permit the other channel to run. However, with READY inactive or with the bus not acquired, there is not much that can be done on the other channel anyway. These two cases only stop the chip when running bus cycles. Any internal operations can proceed without having the bus or with the system not READY.

Note the difference between when the chip is HALTed when using $\overline{RQ}/\overline{GT}$ and an external arbiter (8289) for bus arbitration. Not having the bus due to $\overline{RQ}/\overline{GT}$ will inhibit the bus cycle from even starting. Since the 8289 stops the chip by forcing $\overline{AEN}$ inactive, which goes through the 8284 clock generator to force READY inactive to the IOP (or 8086/8088), a bus cycle has already been started, with ALE asserted, and the address on the address/data lines. When the bus is obtained, operation proceeds at T3 of the bus cycle.

As will be mentioned later, many invalid opcodes will cause the machine to hang up. In these cases the address/data lines will point to where the bad opcode was fetched.

Task Execution

Although optimized for fast and flexible DMA operation, the IOP is also a full-fledged microprocessor. The 8086 Family User's Manual deals with programming strategies and other details. Some of the things to be noted during debugging will be mentioned here.

Instruction Fetching

Unlike the 8085 (but like the 8086), the 8089 labels all fetches from the instruction stream, whether OPCODE, offset, displacement, or literal data, as an instruction fetch on the status lines. In some cases, such as MOV R,I and ADD R,I, the instruction fetch time greatly exceeds execution time because literals are treated as instruction fetches. When following programs on a logic analyzer, triggering on status = 100 or 000 (instruction fetch) and a known program address is the handiest way to trace the flow of the program.

When running programs on a 16-bit bus, a 1-byte queue register comes into play, saving the upper byte fetched from the last instruction fetch, if not used by the previous instruction. This reduces fetch time and bus utilization since the odd byte doesn't need to be fetched again. An internal four-clock cycle fetches data from the queue. Like the internal ROM fetches, the task pointer is put out on the address/data lines, but no bus cycle is run.

The queue can have some possible unexpected affects that have to be taken into account during debugging. These apply only to 16-bit systems and are:

1. Instructions that start on odd boundaries will not likely have bus cycles run to fetch the odd byte unless jumped to, unless preceded by LPDI (which clears the queue), or an instruction that modifies the task pointer is executed. The latter causes the queue to be cleared so that part of an old instruction won't become part of the new one.
2. There is a queue register for each channel so loading or clearing the queue on one channel has no affect on the other channel's queue.
3. The second word of immediate data fetched by a LPDI is done during a pseudo-instruction fetch cycle that cannot make use of the queue or already fetched data. Thus, if on an odd boundary, fetching an LPDI will be byte, word, byte, byte, byte, and the queue will not be loaded.

When Can the Other Channel Interrupt Instruction Execution?

This will be explained more in the "dual channel" operation section, but a few points will be mentioned here. All instructions are made up of internal cycles, with each cycle composed of two to eight clocks. Each bus cycle is one internal cycle, but there can be internal cycles with no comunications to outside the chip. Internal cycles will be extended by the number of wait states in each bus cycle. Between any of these cycles, DMA from the other channel can intervene if the priorities permit it. Instruction fetching and execution can only interrupt instructions on the other channel when the instruction has been completed, not between internal cycles.

Registers

All the registers have some special purpose use in the Instruction Execution or DMA, but all except the CC register can be used as general purpose registers during instruction sequences. A few are loaded specially:

- *CP* — Is only loaded during an initialization sequence. There is one CP register that handles both channels. (All others are duplicated, one set for each channel.)
- *PP* — Is only properly loaded during a CA START command. It holds the SCB value after the initialization sequence.
- *TP* — This is included as part of the registers in the RRR field, but cannot be operated on unless you plan on having your program execution jump around. Everytime this is operated on, the queue is cleared. The TP is loaded from two words (address and displacement) on a CA START, LPD, or LPDI, and loaded from 3-byte MOVP format (see illustration on page 5) on a CA CONTINUE, and can be operated on using any register oriented instructions.

The following registers are loaded during program execution, but can have special effects:

- *CC* — The only thing that affects instructions in the CC register is the chaining bit. If chaining doesn't matter (if only one channel is being used without channel attentions, for example), then the CC register can be general purpose. However, for portability of programs, it is strongly suggested not to use the CC register except for altering DMA parameters and chaining.
- *MC* — Is a general purpose 16-bit register, but is also used to do a masked comparison either for DMA search/match termination or for the JMCE and JMCNE instructions.
- *BC, IX* — Both general purpose 16-bit registers. In instructions that reference memory using the AA field, if AA = 11, the IX register is incremented by the number of bytes fetched or stored.
- *Pointer Registers (GA, GB, GC and TP)* — Are 20-bit registers, but can also be used as 16-bit registers. Adds will carry into the upper 4 bits, but other operations (COMP, OR, AND) are done only on the lower 16 bits. Note that when used as pointers to system memory, it is possible to add a large 16-bit number to the pointer and to put the pointer into another 64K block of memory.

Sign Extension

All program data brought into the chip, either literals or displacements in opcodes, or program data fetched from memory, is sign-extended. Offsets used for calculating addresses are not sign extended. Any 8-bit data brought in has bit 7 sign-extended up to bit 19. Sixteen-bit data is sign-extended from bit 15 to bit 19. It is important to note this, because it can affect logical operations. For example, if one wanted to OR 0084H with 1234H in register GC, you couldn't do ORBI GC, 84H, because bit 7 would sign-extend into the upper byte. Instead, you should code ORI, 0084H to do this properly (note that this has a word for the immediate data). The non-ADD operations will cause the upper four bits of the pointer registers to be invalid since the upper four bits of the ALU come only from the adder.

Tags

It should be noted that the way the IOP knows which bus to access (system or I/O) is via the Tag bit associated with the pointer register used. The TAG can only be set in these ways: loading as a 16-bit register (MOV R,M, MOV R,I) sets TAG to I/O space, loading as a pointer (LPD, LPDI) sets TAG to a system space), or bringing the TAG in from memory by a MOVP instruction.

Effects of Invalid Opcodes

The upper 6 bits of the 2-btye opcode actually determine which opcode will be executed. If these bits are a valid opcode, but lower bits are invalid, the chances are good that the bad bits will be ignored. But if the upper six bits are invalid, there is a very good chance that the chip will hang up and stop execution in that channel. The only way to get out of this mode is to reset the chip. If this hang-up occurs, it can usually be traced because the last address of the instruction fetch will still be on the address/data lines, showing where the program went astray.

Going from Instruction Execution into DMA

The XFER instruction places the current channel into the DMA mode after the next instruction. This permits one last instruction to start up an I/O device (start CRT display on an 8275, for example). However, in order for the IOP to get setup for DMA, the GA, GB, and CC registers should not be altered during this last instruction. Failure to observe this will probably result in an improper first DMA fetch. The WID instruction can be placed after XFER.

DMA Transfers

Incrementing/Non-Incrementing pointers

A memory or I/O pointer can be made to increment for each byte transferred during DMA or it can remain fixed. Incrementing is used primarily for memory block transfers, and non-incrementing is used to access I/O ports.

B/W Mode

Each DMA transfer is composed of separate fetch and store cycles so that 8/16-bit data can be assembled and disassembled, and translation and termination may also be easily handled. There are four possible transfers or B/W modes. They are:

B – B — 1 byte fetched, 1 byte stored
B/B – W — 2 bytes fetched, 1 word stored
W – B/B — 1 word fetched, 2 bytes stored
W – W — 1 word fetched, 1 word stored

The B/W mode used depends on the logical bus width (selected by the WID instruction), address boundary, and incrementing mode.

All systems with 8-bit physical buses will run in the B/B mode. On 16-bit physical buses the other modes are possible, depending on the logical widths selected. Note that the logical bus width can be different than the physical bus width since there are cases where an 8-bit peripheral may be used on a 16-bit bus. The selection of the logical width, and not the physical width, is what determines the B/W mode. Thus it is the responsibility of the programmer not to program an invalid combination (i.e., don't specify a 16-bit logical width on an 8-bit physical bus).

Any transfer on an odd boundary will be B/B but if the pointer is incrementing and on a 16-bit logical bus, after the first transfer, the pointer will be on an even boundary. The IOP will then try to maintain word transfers in order to transfer data as effeciently as possible. See the user's manual for details. The change in B/W mode occurs only after the first transfer or, as explained in the termination section, upon certain byte count terminations.

Synchronization

In the unsynchronzied mode, transfers occur as fast as priorities will allow. This is the IOP's "block-move" mode. Most I/O peripherals only want a DMA transfer on demand; the DRQ lines, along with synchronization specified, will handle this need. Source synchronization is used for I/O reads and destination synchronization is used for I/O writes.

If the IOP is waiting for a DMA request, it will run programs or DMA on the other channel, or execute idle cycles if nothing is pending. If running idle cycles when the DRQ comes, the transfer starts five clocks after DRQ is recognized. If running DMA or instructions on the other channel, the DRQ cannot be serviced until the current internal cycle is done, and may require a maximum of 12 clocks (without bus arbitration or wait states).

Consecutive DRQ-synchronized DMA transfers on the same channel are separated by four idle clocks (assuming no other delays) by an internal sampling mechanism. This happens between the 2-byte fetches on source-synchronized B/B-W cycles, and between the two stores on destination-synchronized W-B/B cycles. This delay between consecutive DMA cycles allows adequate time for proper acknowledgement of the current DMA request before the next request is processed. On destination-synchronized DMA, this isn't a problem, but on source-synchronized DMA, there will be four extra clocks per transfer. Unless one is running right at the speed limit, this won't be a problem. Near the maximum data rate, unsynchronized transfers can be used, with synchronization done by manipulating the READY line.

Translate Mode

When the translate bit is set, the data fetched during DMA will be added to the GC register. This new pointer will in turn be used to fetch, via a seven clock extra fetch cycle, new data, which will then be stored. Translate is only defined for byte transfers. The bytes are added to GC as a positive offset, so a lookup table for translating data can be a maximum of 256 bytes long. Even if the data to be translated falls within a smaller range (such as ASCII code), a full 256-byte lookup table is recommended so that erroneous data can be flagged and controlled.

Translate can be run on any of the B/B transfer modes, so it is useful for doing block translation within program execution as well as translation directly to or from an I/O port.

DMA Termination

One of the powerful features of the IOP is its varied DMA termination conditions and their close tie-in with resuming Instruction Block programs. However, because of the multitude of DMA modes, care must be taken in predicting the exact termination parameters. Various things to be careful about will be outlined here.

Byte Count (BC) Termination

The BC register is decremented for every byte transferred whether or not BC termination is set. If BC termination is set, the last transfer done is the one that results in BC being zero. To avoid the problem of missing BC = 0 on word transfers, if BC is odd between every transfer, the IOP detects when BC is 1, and forces the last transfer to be in the B/B mode. Since both the fetch and store cycles are complete, the source and destination pointers point exactly to the next byte or word that would have been fetched.

Masked Compare (MC) Termination

An MC termination occurs when a pattern matches (or doesn't match, depending on mode selected) the lower half of the MC register (the match pattern) with only the bits that are enabled by the upper half of MC (the mask pattern) contributing to a match. Thus the masked bits can be "don't cares" in both the data byte and the match byte.

The masked comparison is only done on store (deposit) cycles. Any bytes transferred (in B/B or W-B/B mode) will be compared. But, since the MC comparison is done on only one byte, any words stored (W-W or B-B/W) have only their lower byte compared. This may be fine, but if not, make the destination logical width 8 bits.

Just like BC termination, the pointers will point to the next data to be transferred. The BC will also be decremented correctly, except if the termination occurs on the first byte of a W-B/B transfer. In this case the BC will be decremented as if the entire transfer (both bytes) had taken place.

The store cycle that causes an MC termination will be lengthened by two extra clocks (or by one extra clock if there are wait states), to allow time to set up the termination cycle.

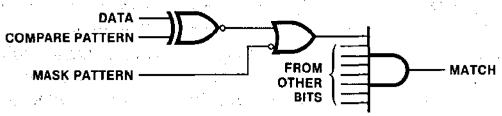


Figure 7. Masked Compare Logic for 1-Bit

External (EXT) Termination

External termination allows the I/O device or controller to use its own conditions to generate a termination. Basically, the IOP will halt DMA as soon as it recognizes an EXT terminate, even if a transfer is only partially complete. There might be concern that multibyte cycles (W-B/B or B/B-W) might have data lost if an EXT terminate stopped the store cycle. In unsynchronized DMA this would happen, but this mode is typically not used with I/O controllers that could generate external terminations. In synchronized DMA modes, it is assumed that the I/O controller will only do a DRQ for valid data transferred, and that it won't give an EXT terminate with its DRQ active. In destination synchronization, the possible problem occurs in the W-B/B mode, where EXT terminate comes after the first store but before the second. This is fine, since even though data was overfetched, the proper amount was actually transferred. In source synchronization, the B/B-W mode raises problems since if an EXT terminate came after the first byte fetched and before the second byte fetched, normally no store cycles would be done at all, thus losing the first byte fetched. In this case (i.e., source synced, DRQ inactive, and 1 byte already fetched), a single byte store cycle is run before the termination cycle, ensuring data integrity.

In order to prevent an invalid signal level from becoming trapped from the asynchronous EXT term lines, two clocks of delay and signal conditioning are done on these lines. In addition, a termination cycle can only be started at certain times during DMA (or TB on the other channel — see dual channel operation section). The EXT terminate lines should be valid eight clocks before the start of the DMA cycle to be stopped.

EXT is sampled even when the IOP is running something on the other channel. Remember though, that despite the high priority of termination, the current instruction on the other channel has to finish before the termination cycle is run. Simultaneous EXTs on both channels result in CH1 termination being done first.

In order to have enough time to process a byte count termination, the BC register is always decremented during DMA fetch cycles. Because of this, external or MC terminations that occur during W-B/B cycles will result in the byte count always being decremented by two, even if only one byte is stored. This also occurs in the block-to-block or block-to-port B/B-W modes. To find the exact number of bytes transferred, the source pointer address can be checked in the block-to-port and block-to-block modes during B/B-W cycles and in the block-to-port W-B/B mode. The destination pointer address can be used to find the number of bytes transferred in the port-to-block and block-to-block modes during W-B/B cycles.

Termination Cycles and Multiple Terminations

Upon termination, the user can run different task block programs, depending on which type of termination has occurred, by specifying an appropriate termination offset. That is, instruction fetching will begin after a termination cycle starting at either the TP value before the DMA started, TP + 4 or TP + 8. These offsets permit long or short jumps to termination routines.

The termination cycle is an add immediate instruction that runs from the internal ROM and adds the proper offset to the TP. It is 15 clocks long for TP + 4 and TP + 8 termination and 12 clocks long for TP + 0 termination.

As mentioned earlier, EXT terminate must come a certain time before the end of a transfer to ensure that the next transfer doesn't start. If it comes in time and MC termination also occurs on the current transfer, then the termination cycle with the largest offset is run. A simultaneous BC terminate cycle will have priority over MC and will result in the running the BC termination program.

Priorities/Dual Channel Operation

The IOP can share its internal and external hardware between two separate channels. The user sees two identical IOP channels with all registers, machine flags, etc., independent of the other channel. The only register in common is the CP register, loaded by the initialization sequence. The mechanism for achieving dual channel operation is time multiplexing between the two channels.

Since interleaving two channels affects their response time to external events and since interfacing to these events is the prime purpose of the IOP, several means of adjusting the priorities of the channels are provided.

Before going into the priority algorithms in detail the four types of cycles that are affected by the priorities will be outlined:

1. *DMA Cycles* — Any type of DMA transfer cycle, including single transfers and translate cycles. DMA can be interrupted after any bus transfer by the other channel.
2. *Instruction Cycles* — Any instructions that have been fetched out of I/O or system memory. Instruction cycles are made up of internal cycles, each two to eight clocks long (assuming no wait states). Some cycles may not run bus transfers. Instructions can be interrupted by DMA after any one of the internal cycles, but can only be interrupted by instructions on the other channel (normal ones or ones from internal ROM) after the current instruction is completed.
3. *Termination Cycle* — Performed when DMA transfers end and instructions resume (except on single transfers).
4. *Channel Attention Cycles* — Performed when channel attention is given, performs actions specified in the CCW field. Both termination and CA cycles can be interrupted by DMA after any internal cycle, but can only be interrupted by instruction cycles after the complete sequence of internal cycles is done.

Termination and channel attention cycles as well as the initialization cycle (which never runs concurrently with other operations) are sequences of instructions fetched from an internal ROM.

Recognizing the higher importance in doing DMA, termination and (to a lesser extent) CA cycles, the following priority scheme is built into the IOP. Any channel that has a higher-priority operation will run continuously until done. If both channels are running the same priority, execution will alternate between them.

Highest Priority

1. DMA transfers, termination, chained instructions
2. Channel attention cycles
3. Instruction cycles
4. Idle cycles

Lowest Priority

Two ways exist to alter the priority scheme. One way is to utilize the priority bits for each channel. If one is greater than the other, that channel will run at the expense of the other if both channels are otherwise running at the same priority. Thus the P bit only has effect on channels running at the same priority level.

If one wants to run instructions along with or in place of DMA on the other channel, the other technique is to set the chaining bit (in the CC register) which brings the instruction priority up to the level of DMA. Care should be taken with this since now CAs are at a lower priority than instructions and will not be serviced unless that channel goes idle. Chaining will also lock out normal instructions on the other channel. Chaining should thus be used with care.

In order to reduce the possibility of shutting out channel attentions, an exception is made to the above priority scheme. After every DMA transfer, whether synchronized or unsynchronized, the IOP will service any pending CA. However, chained task block execution will still shut out CAs on the other channel.

What is the importance of priorities? Well, as an example, let's say that we are running long periods of non-time-critical block moves (via DMA) on one channel and running short bursts of DMA that must be serviced promptly on the other channel. With the default priorities, the short DMA channel bursts would be interleaved with the longer DMA, reducing the maximum transfer rate for both channels. If, however, the priority bit was one on the burst mode DMA and zero on the other, the bursts would be serviced continuously at the fastest possible data rate.

An even more critical case would be the same low priority, long DMA transfers on one channel with DMA on the other channel that must terminate, run a short instruction sequence, and resume DMA again within a short, fixed time. (This might be the case in running a CRT display with linked list processing between lines.) Normally, the low priority, long DMA could indefinitely block the short TB sequence. By setting the high-priority channel's priority bit to one and putting it into the chained instruction mode, the low priority channel would stop its DMA entirely so that the termination/instruction sequence could run.

When establishing the priorities to be run, care should be taken that both channels will run successfully under a worst case combination. This can be tricky when the channels are running asynchronously with fast data rates and/or short latencies, but must be taken into account. Of course, running only one channel on the IOP is an easy solution, but if more than one IOP is being used in the system, the priorities and delays of the bus arbitration used (either $\overline{RQ}/\overline{GT}$ or an 8289 bus arbiter) must be taken into account. It may be found that the on-chip arbitration between the two channels is faster and more powerful than external arbitration.

SUMMARY

It is hoped that the material presented here will aid those who are putting together and debugging an 8089 IOP system, and help them in understanding the operation of the IOP. Many of the debugging techniques should be familiar to those who have worked with micro- and minicomputer systems before. Other debugging techniques not mentioned here, which work well with microprocessor systems, could be just as applicable to the 8089. The unique nature of the IOP among LSI devices warrants special consideration for its I/O functions and multiprocessor capabilities.

Appendix I

CHECKLIST OF POSSIBLE PROBLEMS

HARDWARE PROBLEMS

- Is RESET at least four clocks long?
- Are both V_{SS} lines connected to ground?
- Does the first CA falling edge come at least two clocks after RESET goes away?
- Does the second CA come at least 150 clocks (16-bit system, no wait states) after the first CA?
- Is READY correctly synchronized and gated by local/system bus lines?
- Is SEL correct for first CA so that IOP comes up correctly as master or slave?
- If two IOPs are local to each other, is a 2.7K pull-up resistor used on RQ/GT?

SOFTWARE PROBLEMS

- Are the initialization parameters in the initialization linked-list correct?
- Is BUSY flag being properly tested by host CPU software before modifying PB or providing a new command?
- Has the chaining, translate, or lock bit in the CC register been erroneously set?
- Have DMA termination conditions been met? The IOP could be trying to do endless DMA.

Appendix II

BREAKPOINT ROUTINE AND CONTROL PROGRAM

The debugging program described here is an example of the kind of software development tool that can be developed for the 8089 IOP. It was written to try out various breakpoint schemes, and has been used to debug an engineering application test system. The program is not meant to be the ultimate debugging tool, but is an example of what can be put together to utilize the breakpoint routine described earlier in the application note.

The debugging program was tested on a 8086-based system that emulates the SDK-86 I/O structure, and uses the SDK-86 serial monitor. This enables it to use the SDK-86 Serial Downloader to interface to an Intellec® development system on which the software was created. The 8086 system is interfaced via a MULTIBUS™ interface to an IOP running in the REMOTE mode. The remote bus access technique, mentioned earlier in this note, is implemented on this system, but was not used in the software debugging program.

The breakpoint routine uses a simple jump to a save routine. The PL/M-86 supervisory or control program handles the placement of the jump within the users program. Since it can not normally access the remote bus, all IOP programs to be tested must run out of system memory.

When the control program starts, it assumes the IOP has just been reset. It then prompts the user for the CP and PP values. After this, it sends the first (initialization) channel attention. It then asks the user for the channel to be run, and the starting and stopping addresses. After the stopping address has been entered, a Channel Attention Start is given. If the breakpoint is reached, a HALT is executed, and the control program prints the register contents. If the breakpoint hasn't been reached, the user can type any character, and a Channel Attention Halt will be sent to the IOP. If the IOP responds within 50 ms, the TP where it was halted is printed. Otherwise, the control program issues an error message. If, at any time, the user wants to get out of the program, typing an ESC will pass control back to the SDK-86 monitor. Figure 9 shows the flow of the control program.

Note that, unlike a single CPU debugging routine, having the 8086 supervise the 8089 enables a clean exit from crashed IOP programs. The program code where jumps had been placed are always restored. The control program is a good example of how the power of dual processors can be put to good advantage.

Comments within the control program indicate parameters that need to be changed to run on different systems. It should be noted that channel attentions are invoked by the recommended method of using an I/O write to a port to generate CA and using A0 for SEL.

Source and object files of this program are available through Intel's INSITE™ User's Program Library as program 8089 Break. 89 (number AD6).

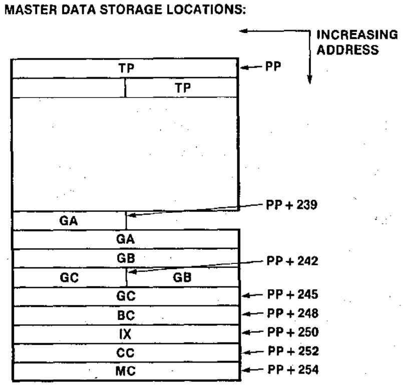


Figure 8. Breakpoint Routine to Run 8089 Program out of System Memory

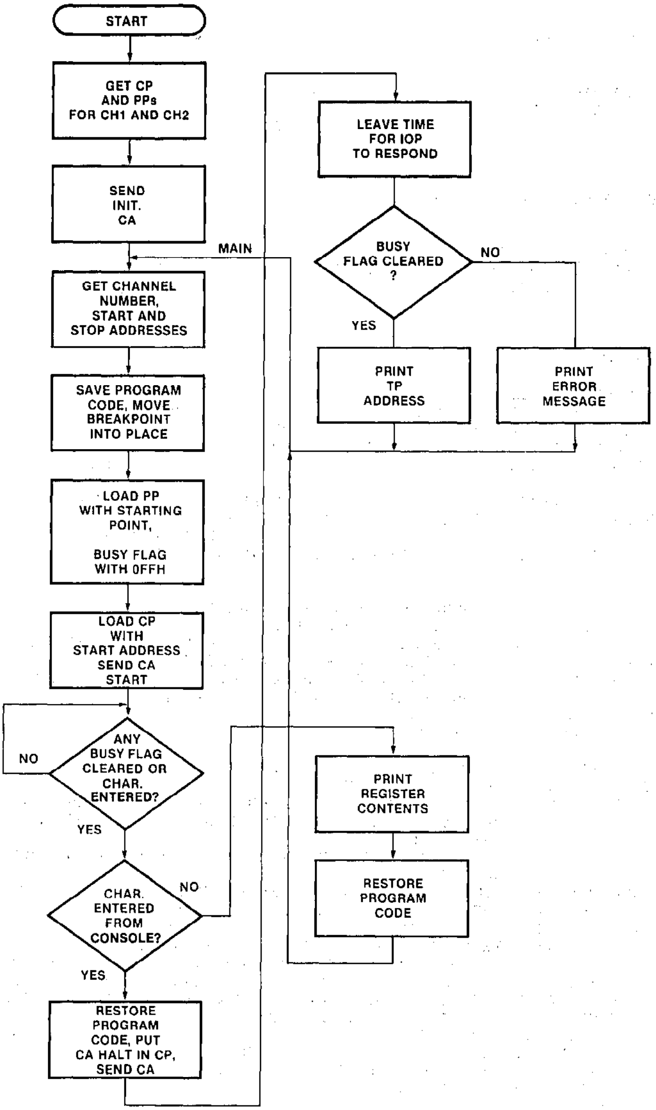
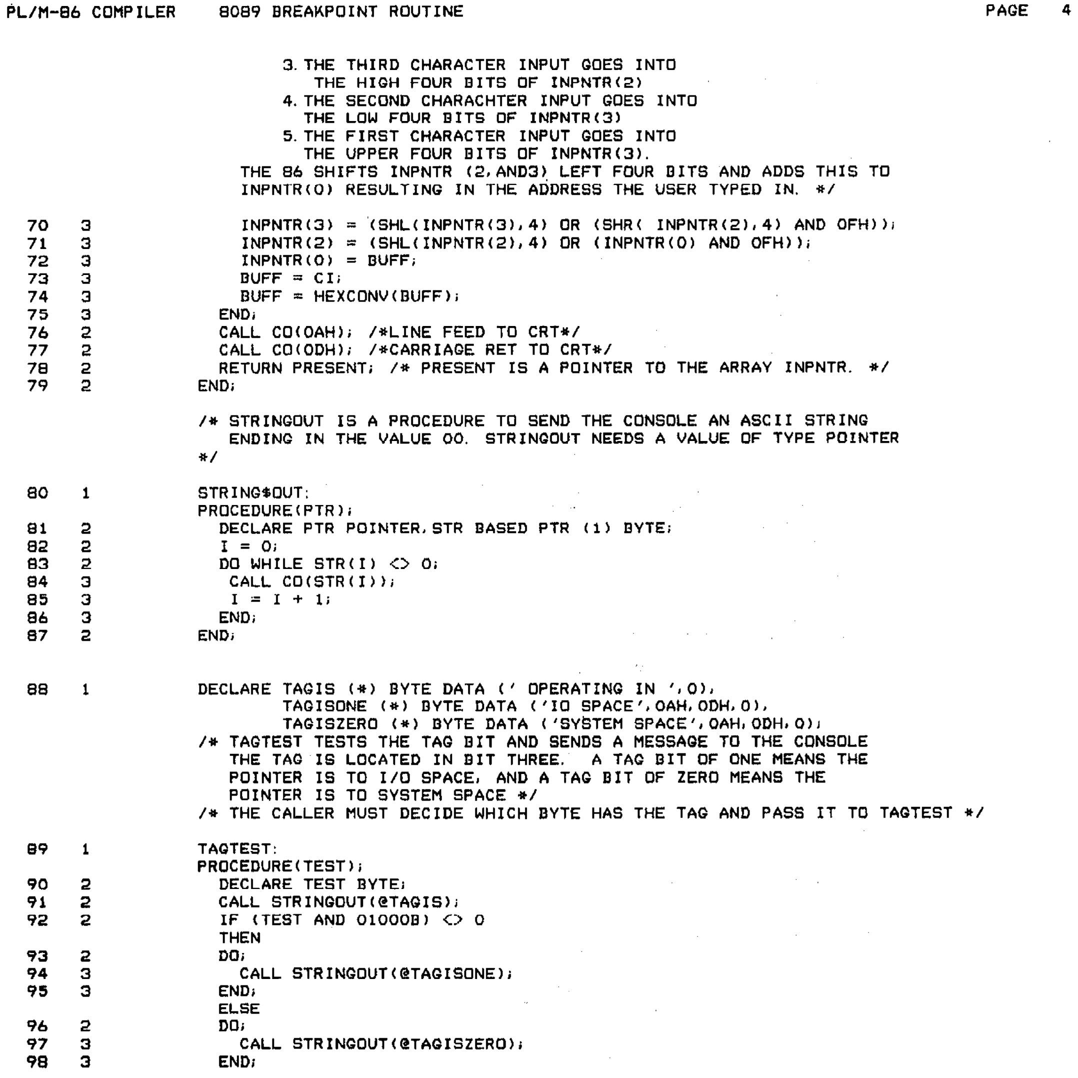

START
GET CP AND PPs FOR CH1 AND CH2
SEND INIT. CA
MAIN
GET CHANNEL NUMBER, START AND STOP ADDRESSES
SAVE PROGRAM CODE, MOVE BREAKPOINT INTO PLACE
LOAD PP WITH STARTING POINT, BUSY FLAG WITH 0FFH
LOAD CP WITH START ADDRESS SEND CA START
ANY BUSY FLAG CLEARED OR CHAR. ENTERED?
NO
YES
CHAR. ENTERED FROM CONSOLE?
NO
YES
RESTORE PROGRAM CODE, PUT CA HALT IN CP, SEND CA
LEAVE TIME FOR IOP TO RESPOND
BUSY FLAG CLEARED ?
NO
YES
PRINT TP ADDRESS
PRINT ERROR MESSAGE
PRINT REGISTER CONTENTS
RESTORE PROGRAM CODE

Figure 9. Breakpoint Routine to Run 8089 Program out of System Memory

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                        PAGE   1


ISIS-II PL/M-86 X103 COMPILATION OF MODULE BREAKPOINT
OBJECT MODULE PLACED IN BREAK.OBJ
COMPILER INVOKED BY:  :F1:PLM86 BREAK.SRC PAGEWIDTH (100)


                 $TITLE ('8089 BREAKPOINT ROUTINE')
                 /*...............................................................................

                    8089 BREAK POINT PROCEDURE
                    WRITTEN BY DAVE FERGUSON 2/2/79   REV 2  8/14/79
                    INTEL CORPORATION

                    ........................................................................ */

   1             BREAK$POINT:
                    DO;
   2    1           DECLARE I BYTE;
   3    1           DECLARE SAVECODE (4) WORD; /*BUFFER FOR STORAGE*/
   4    1           DECLARE ONEPP POINTER; /* CHAN ONE PP */
   5    1           DECLARE TWOPP POINTER; /* CHAN TWO PP */
   6    1           DECLARE STARTBYTES (4) BYTE; /* BUFFER FOR START ADDRESS */

   7    1           DECLARE STARTPOINTER POINTER; /* POINTER FOR START ADDR. */
   8    1           DECLARE ENDPOINTER POINTER; /* POINTER FOR END ADDR. */
   9    1           DECLARE PRESENT POINTER AT (@INPNTR); /* POINTER BUFFER */
  10    1           DECLARE TRUE LITERALLY 'OFFH',FALSE LITERALLY 'OOOH';


                    /*  YOU MUST CONFIGURE YOUR I/O STRUCTURE AND
                      SYSTEM TO MATCH THE PROGRAM OR VISA VERSA */
  11    1           DECLARE CRTSTATUS LITERALLY 'OFFF2H', /* 8251 STATUS PORT */
                    CRTDATA LITERALLY 'OFFFOH', /* 8251 DATA PORTS */
                    CHANATTEN LITERALLY 'OFAH', /* CHANNEL ONE CHANNEL ATTENTION PORT */
                     /* CHANNEL TWO CHANNEL ATTENTION PORT = CHANATTEN + 1 */
                    CHANNELONE LITERALLY 'OOH',
                    CHANNELTWO LITERALLY 'O1H',

                    /* ASCII IS A STRING OF HEX CHARACHTERS IN ASCII FORM */
                    ASCII (*) BYTE DATA ('0123456789ABCDEF'),
                    TITLE$STRING (*) BYTE DATA (OAH,ODH,'8089 BREAKPOINT VER 1.0',
                                  OAH,ODH,'TYPE ESCAPE TO RETURN TO MONITOR. ',
                                  OAH,ODH,O),
                    CHANGIVEN (*) BYTE DATA ('CHANNEL ATTENTION GIVEN TYPE ANY KEY TO ABORT. '
                                 ,OAH,ODH,O),
                    BKREACHED (*) BYTE DATA (OAH,ODH,'BREAKPOINT REACHED',OAH,ODH,O),
                    GETCP (*) BYTE DATA ('INPUT CP IN HEX',OAH,ODH,OO),
                    GET$PP (*) BYTE DATA ('INPUT PP IN HEX FOR ',OOH),
                    GETSTART (*) BYTE DATA (OAH,ODH,'INPUT STARTING ADDRESS IN HEX',OAH,ODH,OOH),
                    STOPADDR (*) BYTE DATA ('INPUT END ADDRESS IN HEX',OAH,ODH,OOH),
                    CHANNUMBER (*) BYTE DATA (OAH,ODH,'CHANNEL ONE OR TWO? ',OOH),
                    ABORT (*) BYTE DATA (' FATAL ERROR - IOP DOES NOT RESPOND TO CHANNEL',
                    ' ATTENTION. RE-INITIALIZE SYSTEM ',O),
                    ABORTAT (*) BYTE DATA (' TP WAS ',O),
                    ONE (*) BYTE DATA (' CHANNEL ONE',OAH,ODH,OOH),
                    TWO (*) BYTE DATA (' CHANNEL TWO',OAH,ODH,OOH),
                    GASTRING (*) BYTE DATA  ('GA = ',OOH),
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                      PAGE   2

                    GBSTRING (*) BYTE DATA ('GB = ',OOH),
                    GCSTRING (*) BYTE DATA ('GC = ',OH),
                    BCSTRING (*) BYTE DATA  (OAH,ODH,'BC = ',OOH),
                    IXSTRING (*) BYTE DATA (OAH,ODH,'IX = ',OOH),
                    CCSTRING (*) BYTE DATA (OAH,ODH,'CC = ',OOH),
                    MCSTRING (*) BYTE DATA (OAH,ODH,'MC = ',OOH)

  12   1            DECLARE CHAR BYTE;
  13   1            DECLARE ONETWO BYTE;

                    /* SDKMON IS A PLM TECHNIQUE USED TO FORCE THE CPU INTO AN
                       INTERUPT LEVEL 3.  IN ORDER TO USE THIS THE PROGRAM MUST
                       BE COMPILED (LARGE). */
  14   1            SDKMON:
                    PROCEDURE;
  15   2             DECLARE HERE (*) BYTE DATA (OCCH),
                    /* THIS IS AN INT.  3  */
                       WHERE WORD  DATA(.HERE);
  16   2               CALL WHERE;
  17   2            END;


                    /* CO SENDS A CHAR TO THE CONSOLE WHEN READY */
                    /* THIS ROUTINE IS WRITTEN TO RUN VIA THE SERIAL
                       PORT OF AN SDK86 */
  18   1            CO:
                    PROCEDURE (C);
  19   2             DECLARE C BYTE;
  20   2             DO WHILE (INPUT(CRTSTATUS) AND 01H) = 0; END;
  22   2             OUTPUT (CRTDATA) = C;
  23   2            END;

                    /* CI GETS A CHARACHTER FROM THE USER VIA THE SERIAL PORT */
                    /* CI AUTOMATICALLY ECHOS THE CHARACHTER TO THE USER CONSOLE */
  24   1            DECLARE ESCAPE LITERALLY '1BH';

  25   1            CI: PROCEDURE BYTE;
  26   2              DO WHILE (INPUT(CRT$STATUS) AND 02H) = 0; END;
  28   2              CHAR = INPUT (CRTDATA) AND 07FH;
  29   2              CALL CO(CHAR);
  30   2              IF CHAR = ESCAPE THEN CALL SDKMON; /* GO TO SDK MONITOR */
  32   2              RETURN CHAR;
  33   2            END;

                    /* VALIDHEX CHECKS THE VALIDITY OF A BYTE AS A HEX CHARACHTER*/
                    /* THE PROCEDURE RETURNS TRUE IF VALID FALSE IF NOT */

  34   1            VALIDHEX:
                    PROCEDURE (H) BYTE;
  35   2             DECLARE H BYTE;
  36   2             DO I=0 TO LAST(ASCII);
  37   3              IF H=ASCII(I) THEN RETURN TRUE;
  39   3             END;
  40   2             RETURN FALSE;
  41   2            END;
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                      PAGE   3

                    /* HEXCONV CONVERTS A HEX CHARACTER TO BINARY FOR MACHINE USE.
                       IF THE CHARACTER IS NOT A VALID HEX CHAR, THE PROCEDURE RETURNS
                       THE VALUE OFFH */
  42    1           HEXCONV:
                    PROCEDURE (DAT)  BYTE;
  43    2             DECLARE DAT BYTE;
  44    2             IF VALIDHEX(DAT) <> OFFH THEN RETURN TRUE;
  46    2             DO I=0 TO LAST(ASCII);
  47    3               IF DAT = ASCII(I) THEN RETURN I;
  49    3             END;
  50    2             END;

                    /* HEXOUT WILL CONVERT A VALUE OF TYPE BYTE TO AN ASCII STRING
                       AND SEND IT TO THE CONSOLE */

  51    1           HEXOUT:
                    PROCEDURE(C);
  52    2             DECLARE C BYTE;
  53    2             CALL CO(ASCII(SHR(C,4) AND OFH));
  54    2             CALL CO(ASCII(C AND OFH));
  55    2           END;


                    /* WORDOUT CONVERTS A VALUE OF TYPE WORD TO AN ASCII STRING
                       AND SENDS IT TO THE CONSOLE */
  56    1           WORDOUT:
                    PROCEDURE (W);
  57    2             DECLARE W WORD;
  58    2             CALL HEXOUT(HIGH(W));
  59    2             CALL HEXOUT(LOW(W));
  60    2           END;

                    /* GETADDRESS IS A PROCEDURE TO GET AN ADDRESS FROM THE CONSOLE.
                       THIS PROCEDURE WILL ONLY CONSIDER THE LAST 5 CHARACHTERS ENTERED
                       */
  61    1           DECLARE INPNTR (4) BYTE;

  62    1           GET$ADDRESS:
                    PROCEDURE POINTER;
  63    2             DECLARE BUFF BYTE;
                    /*CLEAR ALL VALUES TO ZERO */
  64    2             INPNTR(0) = 0;
  65    2             INPNTR(1) = 0;
  66    2             INPNTR(2) = 0;
  67    2             INPNTR(3) = 0;

  68    2             BUFF = 0;
  69    2             DO WHILE BUFF <> TRUE;
                    /* THIS SEQUENCE OF SHIFTS ALLOW THE USER TO TYPE IN FIVE
                       OR MORE CHARACHTERS TO BECOME THE ACTUAL POINTER FOR 8089
                       OR 8086. THIS PROCEDURE RETURNS THE LAST FIVE IN PROPER
                       SEQUENCE  STORED IN INPNTR(0-3).  THE STORAGE
                       IS AS FOLLOWS:
                            1. THE LAST CHARACTER INPUT GOES INTO
                              THE LOW FOUR BITS OF INPNTR(0).
                            2. THE NEXT TO LAST CHARACTER GOES INTO
                               THE LOW FOUR BITS OF INPNTR(2).
```



```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                         PAGE   4

                              3. THE THIRD CHARACTER INPUT GOES INTO
                                   THE HIGH FOUR BITS OF INPNTR(2)
                              4. THE SECOND CHARACHTER INPUT GOES INTO
                                 THE LOW FOUR BITS OF INPNTR(3)
                              5. THE FIRST CHARACTER INPUT GOES INTO
                                 THE UPPER FOUR BITS OF INPNTR(3).
                          THE 86 SHIFTS INPNTR (2,AND3) LEFT FOUR BITS AND ADDS THIS TO
                          INPNTR(0) RESULTING IN THE ADDRESS THE USER TYPED IN. */

  70   3                  INPNTR(3) = (SHL(INPNTR(3),4) OR (SHR( INPNTR(2),4) AND OFH));
  71   3                  INPNTR(2) = (SHL(INPNTR(2),4) OR (INPNTR(0) AND OFH));
  72   3                  INPNTR(0) = BUFF;
  73   3                  BUFF = CI;
  74   3                  BUFF = HEXCONV(BUFF);
  75   3                END;
  76   2                CALL CO(OAH);  /*LINE FEED TO CRT*/
  77   2                CALL CO(ODH);  /*CARRIAGE RET TO CRT*/
  78   2                RETURN PRESENT;  /* PRESENT IS A POINTER TO THE ARRAY INPNTR. */
  79   2              END;

              /* STRINGOUT IS A PROCEDURE TO SEND THE CONSOLE AN ASCII STRING
                 ENDING IN THE VALUE 00.  STRINGOUT NEEDS A VALUE OF TYPE POINTER
              */

  80   1              STRING$OUT:
                      PROCEDURE(PTR);
  81   2                DECLARE PTR POINTER,STR BASED PTR (1) BYTE;
  82   2                I = 0;
  83   2                DO WHILE STR(I) <> 0;
  84   3                 CALL CO(STR(I));
  85   3                 I = I + 1;
  86   3                END;
  87   2              END;


  88   1              DECLARE TAGIS (*) BYTE DATA (' OPERATING IN ',0),
                              TAGISONE (*) BYTE DATA ('IO SPACE',OAH,ODH,0),
                              TAGISZERO (*) BYTE DATA ('SYSTEM SPACE',OAH,ODH,0);
              /* TAGTEST TESTS THE TAG BIT AND SENDS A MESSAGE TO THE CONSOLE
                 THE TAG IS LOCATED IN BIT THREE.  A TAG BIT OF ONE MEANS THE
                 POINTER IS TO I/O SPACE, AND A TAG BIT OF ZERO MEANS THE
                 POINTER IS TO SYSTEM SPACE */
              /* THE CALLER MUST DECIDE WHICH BYTE HAS THE TAG AND PASS IT TO TAGTEST */

  89   1              TAGTEST:
                      PROCEDURE(TEST);
  90   2                DECLARE TEST BYTE;
  91   2                CALL STRINGOUT(@TAGIS);
  92   2                IF (TEST AND 01000B) <> 0
                        THEN
  93   2                DO;
  94   3                  CALL STRINGOUT(@TAGISONE);
  95   3                END;
                        ELSE
  96   2                DO;
  97   3                  CALL STRINGOUT(@TAGISZERO);
  98   3                END;
  99   2          END;
 100   1          DECLARE SAVE$ADDR LITERALLY '2000H',
                      SAVE$SEG LITERALLY '00C0H';

 101   1          DECLARE BREAK89 (4) WORD DATA (9B81H,0891H,SAVE$ADDR,SAVE$SEG);
                  /* BREAK89 IS AN 4 WORD ESCAPE SEQUENCE TO ADDRESS 2000H
                     CONSISTING OF AN  LPDI TP,SAVE$ADDR WITH SEGMENT
                         LOCATED AT OC00H.    */

                  /* BRKRTN IS 33 BYTES OF CODE THAT STORES ALL REGISTERS
                     AS FOLLOWS:
                      GA STORED    AT PP + 239
                      GB STORED    AT PP + 242
                      GC STORED    AT PP + 245
                      BC STORED    AT PP + 248
                      IX STORED    AT PP + 250
                      CC STORED    AT PP + 252
                      MC STORED    AT PP + 254
                        */

 102   1          DECLARE BRKRTN (33) BYTE AT (02C00H)
              /* 02C00H IS ACTUALLY (SAVE$ADDR + (SHL(SAVE$SEG),4)), AND SHOULD
                     MATCH ADDRESS AND SEGMENT WHERE BREAK ROUTINE IS WANTED    */
                INITIAL
             (03H,09BH,0EFH,023H,09BH,0F2H,043H,09BH,0F5H,063H,087H,0F8H,0A3H,087H,
             0FAH,0C3H,087H,0FCH,0E3H,087H,0FEH,020H,048H) ;
 103   1          DECLARE PP POINTER;
 104   1          DECLARE PPP BASED PP (1) BYTE;

 105   1          START$PRGM:
                    PROCEDURE(ONE$TWO,PPP);
 106   2            DECLARE ONE$TWO BYTE,PPP POINTER,
                    WHERE BASED PPP (1) BYTE;

 107   2                WHERE(0) = START$BYTES(0);
 108   2            WHERE(1) = 0;
 109   2            WHERE(2) = START$BYTES(2);
 110   2            WHERE(3) = START$BYTES(3);
 111   2            CPDAT((ONE$TWO) * 8) = 3;
                    /* IF ONETWO = 1 THEN OUTPUT TO PORT 0FBH,  IF ONETWO
                       IS 0 THEN OUTPUT TO PORT 0FAH */
 112   2            OUTPUT(CHANATTEN + (ONETWO )) = 0;
 113   2            CALL STRINGOUT(@CHANGIVEN);
 114   2          END;

                  /* THIS PART OF THE PROGRAM ALLOWS THE USER TO DEFINE THE
                    CP,PP OF EACH CHANNEL */
 115   1          DECLARE BREAKOUT BASED ENDPOINTER (1) WORD;

 116   1          DECLARE CP POINTER;
 117   1          DECLARE CPDAT BASED CP (1) BYTE;

 118   1          DECLARE ONEPPDAT BASED ONEPP (1) BYTE;
 119   1          DECLARE TWOPPDAT BASED TWOPP (1) BYTE;


 120   1          CALL STRINGOUT (@TITLESTRING);
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                    PAGE   6


 121    1          CALL STRINGOUT(@GETCP);
 122    1          CP = GETADDRESS;
 123    1          CALL STRINGOUT(@GETPP);
 124    1          CALL STRINGOUT(@ONE);
 125    1          ONEPP = GETADDRESS;
 126    1          CALL STRINGOUT(@GETPP);
 127    1          CALL STRINGOUT(@TWO);
 128    1          TWOPP = GETADDRESS;
 129    1          OUTPUT (CHANATTEN) = 0;  /* INITIALIZATION CA */


 130    1      MAIN:
                   CALL STRINGOUT(@CHANNUMBER);
 131    1          CHAR = CI;  /* GET CHANNEL NUMBER */
 132    1          IF (CHAR AND 01H) <> 0 /* CHECK BIT ZERO TO DEFINE
                                  CHANNEL NUMBER */
                   THEN DO;
 134    2            CALL STRINGOUT(@ONE);
 135    2            ONETWO = CHANNEL$ONE;
 136    2          END;
                   ELSE
 137    1           DO;
 138    2             CALL STRINGOUT(@TWO);
 139    2             ONETWO = CHANNEL$TWO;
 140    2          END;

 141    1          CALL STRINGOUT(@GET$START);  /* GET STARTING ADDRESS
                                    FROM USER */

 142    1          STARTPOINTER = GETADDRESS;
 143    1          DO I = 0 TO 3;  /* MOVE STARTING ADDRESS INTO CP AREA */
 144    2            STARTBYTES(I) = INPNTR(I);
 145    2          END;
 146    1          CALL STRINGOUT(@STOPADDR);  /* GET STOP ADDRESS
                                    FROM USER */

 147    1          ENDPOINTER = GETADDRESS;
 148    1          DO I = 0 TO 3;  /* MOVE CODE TO SAFE AREA   */
 149    2            SAVECODE(I) = BREAKOUT(I);
 150    2          END;
 151    1          DO I = 0 TO 3;
 152    2            BREAKOUT(I) = BREAK89(I);  /* MOVE ESCAPE SEQUENCE INTO PLACE */
 153    2          END;
 154    1          CPDAT(1) = 0FFH;  /* SET  CHANNEL ONE BUSY FLAG */
 155    1          CPDAT(9) = 0FFH;  /* SET CHANNEL TWO BUSY FLAG */
 156    1          DO CASE ONETWO;
 157    2          PP = ONEPP;
 158    2          PP = TWOPP;
 159    2          END;
 160    1          CALL START$PRGM(ONE$TWO,PP);
                   /* WAIT FOR ONE OF THE FOLLOWING
                      1. CPDAT(1) = 0  CH1 NOT BUSY
                      2. CPDAT(9) = 0 CH2 NOT BUSY
                      3. THE 8251 REC. BUFFER IS FULL BECAUSE USER HAS DEPRESSED A KEY
                   */
 161    1          DO WHILE (  (CPDAT(1) AND CPDAT(9)) AND (NOT (INPUT(CRT$STATUS) AND 02H))) = 0FFH;
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                          PAGE   7

 162   2            END;
 163   1          IF (INPUT(CRT$STATUS) AND 02H) <> 0
                     THEN
 164   1             DO;
 165   2               CHAR = CI;
 166   2               DO I = 0 TO 3;
 167   3                 BREAKOUT(I) = SAVECODE(I);
 168   3               END;
                  /* IF ONETWO = 0 THEN PUT CHA HLT IN CPDAT(0)
                     IF ONETWO = 1 THEN PUT CHA HLT IN CPDAT(8)
                  */
 169   2               CPDAT(ONE$TWO *8) = 06H;
                  /* IF ONETWO = 0 THEN OUTPUT TO PORT 0FAH, IF ONETWO
                     IS 1 THEN OUTPUT TO PORT 0FBH.
                  */
 170   2               OUTPUT(CHANATTEN + ONETWO) = 0;
 171   2               DO I = 0 TO 5;
 172   3                CALL TIME(100);
 173   3               END;

                  /* IF BUSY FLAG HAS BEEN CLEARED, THEN A CA HALT&SAVE
                     WAS EXECUTED.  IF SO, PRINT SAVED TP; IF NOT, ABORT  */

 174   2               IF CPDAT(SHL(ONETWO,3) + 1) <> 0   /* CHECK BUSY FLAG */
                       THEN
 175   2               DO;
 176   3                CALL STRINGOUT(@ABORT);
 177   3               END;
                       ELSE
 178   2               DO;
 179   3                CALL STRINGOUT(@ABORTAT);
 180   3                CALL CO(ASCII(SHR(PPP(2),4))); /* UPPER NIBBLE OF ADDR
                                      STORED BY HALT */

 181   3                CALL HEXOUT(PPP(1)); /* MIDDLE BYTE OF ADDR
                                  STORED BY HALT */

 182   3                CALL HEXOUT(PPP(0)); /* LEAST SIG BYTE OF ADDR
                                  STORED BY HALT */

 183   3               END;
 184   2               CPDAT(ONETWO * 8) = 3H; /* CA START IN CPDAT(0) OR CPDAT(8) */
 185   2               GO TO MAIN;
 186   2           END;
 187   1          DO;

 188   2           CALL STRINGOUT(@BKREACHED);

 189   2           CALL STRINGOUT(@GASTRING);
 190   2           CALL CO(ASCII(SHR(PPP(241),4)));
 191   2           CALL HEXOUT(PPP(240));
 192   2           CALL HEXOUT(PPP(239));
 193   2           CALL TAGTEST(PPP(241));

 194   2           CALL STRINGOUT(@GBSTRING);
 195   2           CALL CO(ASCII(SHR(PPP(244),4)));
 196   2           CALL HEXOUT(PPP(243));
```

```
PL/M-86 COMPILER    8089 BREAKPOINT ROUTINE                                  PAGE   8

 197   2           CALL HEXOUT(PPP(242));
 198   2           CALL TAGTEST(PPP(244));

 199   2           CALL STRINGOUT(@GCSTRING);
 200   2           CALL CO(ASCII(SHR(PPP(247),4)));
 201   2           CALL HEXOUT(PPP(246));
 202   2           CALL HEXOUT(PPP(245));
 203   2           CALL TAGTEST(PPP(247));

 204   2           CALL STRINGOUT(@BCSTRING);
 205   2           CALL HEXOUT(PPP(249));
 206   2           CALL HEXOUT(PPP(248));

 207   2           CALL STRINGOUT(@IXSTRING);
 208   2           CALL HEXOUT(PPP(251));
 209   2           CALL HEXOUT(PPP(250));

 210   2           CALL STRINGOUT(@CCSTRING);
 211   2           CALL HEXOUT(PPP(253));
 212   2           CALL HEXOUT(PPP(252));

 213   2           CALL STRINGOUT(@MCSTRING);
 214   2           CALL HEXOUT(PPP(255));
 215   2           CALL HEXOUT(PPP(254));

 216   2          END;
                  /* RESTORE CODE TO ORIGINAL LOCATION   */
 217   1          DO I = 0 TO 3;
 218   2            BREAKOUT(I) = SAVECODE(I);
 219   2          END;

 220   1          GO TO MAIN;

 221   1      END;


MODULE INFORMATION

     CODE AREA SIZE      = 0619H   1561D
     CONSTANT AREA SIZE = 01EFH    495D
     VARIABLE AREA SIZE = 0020H     32D
     MAXIMUM STACK SIZE = 0014H     20D
     427 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
8089 ASSEMBLER


ISIS-II 8089 ASSEMBLER X004 ASSEMBLY OF MODULE AP50_BREAKPOINT_ROUTINE
OBJECT MODULE PLACED IN :F0:BRKASM.OBJ
ASSEMBLER INVOKED BY ASM89.4 BRKASM.SRC

                                   1 NAME    AP50_BREAKPOINT_ROUTINE
0000                               2 BRKPNT  SEGMENT
                                   3 ;***************************************
                                   4 ;   BASIC 8089 BREAKPOINT ROUTINE
                                   5 ;     BY JOHN ATWOOD    REV 3  8/13/79
                                   6 ;     INTEL CORPORATION
                                   7 ;***************************************
                                   8
                                   9 ;   THE FOLLOWING CODE IS CONTAINED IN THE PL/M-86
                                  10 ;   CONTROL PROGRAM(BREAK.89) AND IS ASSEMBLED HERE
                                  11 ;   TO ILLUSTRATE HOW THE ESCAPE SEQUENCE AND SAVE
                                  12 ;   ROUTINE CODE WAS GENERATED.  TO USE THE 8089 BREAK-
                                  13 ;   POINT PROGRAM, THIS ASM89 PROGRAM WOULD NOT BE
                                  14 ;   NEEDED.  SAVE_ADDR IS THE SAME AS SAVE$ADDR IN THE
                                  15 ;   BREAK.89 PROGRAM.
                                  16
 2000                             17 SAVE_ADDR        EQU      2000H    ;SAVE ROUTINE ADDRESS
                                  18
0000    9108 00200000             19         LPDI TP,SAVE_ADDR         ;JUMP TO SAVE ROUTINE
                                  20
                                  21 ;****************************************
                                  22
                                  23 ;   REGISTER SAVE LOCATIONS WITHIN PB:
                                  24
                                  25 REGS    STRUC
0000                              26 PBLOCK: DS      239      ;PARAMETER BLOCK
00EF                              27 GASAV:  DS      3        ;GA AREA
00F2                              28 GBSAV:  DS      3        ;GB AREA
00F5                              29 GCSAV:  DS      3        ;GC AREA
00F8                              30 BCSAV:  DS      2        ;BC AREA
00FA                              31 IXSAV:  DS      2        ;IX AREA
00FC                              32 CCSAV:  DS      2        ;CC AREA
00FE                              33 MCSAV:  DS      2        ;MC AREA
0100                              34 REGS    ENDS
                                  35
                                  36 ;   REGISTER SAVE ROUTINE:
                                  37
                                  38 ORG     SAVE_ADDR
                                  39
2000    039B EF                   40         MOVP [PP].GASAV,GA        ;SAVE GA
2003    239B F2                   41         MOVP [PP].GBSAV,GB        ;SAVE GB
2006    439B F5                   42         MOVP [PP].GCSAV,GC        ;SAVE GC
2009    6387 F8                   43         MOV [PP].BCSAV,BC         ;SAVE BC
200C    A387 FA                   44         MOV [PP].IXSAV,IX         ;SAVE IX
200F    C387 FC                   45         MOV [PP].CCSAV,CC         ;SAVE CC
2012    E387 FE                   46         MOV [PP].MCSAV,MC         ;SAVE MC
                                  47
2015    2048                      48         HLT                       ;STOP THIS CHANNEL,
                                  49                                   ;CLEAR BUSY FLAG.
                                  50 ;****************************************
2017                              51 BRKPNT  ENDS
                                  52
                                  53 END
```

intel®

APPLICATION
NOTE

AP-51

March 1979

Designing 8086, 8088, 8089 Multiprocessing Systems with the 8289 Bus Arbiter

Designing 8086, 8088, 8089 Multiprocessor Systems with the 8289 Bus Arbiter

Contents

Our thanks to Jim Nadir, the author of this application note. Jim is a design engineer in the microprocessors and peripherals operation division. Please direct any technical questions you may have to your local Intel FAE (Field Application Engineer).

INTRODUCTION

Over the past several years, microprocessors have been increasing in popularity. The performance improvements and cost reductions afforded by LSI technology have spurred on the design motivation of using multiple processors to meet system real-time performance requirements. The desire for improved system real-time response, system reliability and modularity has made multiprocessing techniques an increasingly attractive alternative to the system design engineer; techniques that are characterized as having more than one microprocessor share common resources, such as memory and I/O, over a common multiple processor bus.

This type of design concept allows the system designer to partition overall system functions into tasks that each of several processors can handle individually to increase system performance and throughput. But, how should a designer proceed to implement a multiprocessing system? Should he design his own? If so, how are the microprocessors synchronized to avoid contention problems? The designer could put them all in phase using one clock for all the microprocessors. This may work, until the physical dimensions of the system become large. When this occurs, the designer is faced with many problems, like clock skew (resulting in bus spec violations) and duty cycle variations.

A better approach to implementing a multiprocessor system is not to have a common processor clock, but allow each processor to work asynchronously with respect to each other. The microprocessor requests to use the multiple processor bus could then be synchronized to a high frequency external clock which will permit duty cycle and phase shift variations. This type of approach has the benefit of allowing modularity of hardware. When new system functions are desired, more processing power can be added without impacting existing processor task partitioning.

One approach to implement this asynchronous processing structure would be to have all the bus requests enter a priority encoder which samples its inputs as a function of the higher frequency "bus clock". The inputs would arrive asynchronously to the priority encoder and would be resolved by the priority encoder structure as to which microprocessor would be granted the bus. Another approach, that used by Intel, is rather than allowing the requests to arrive asynchronously with respect to one another at the priority encoder, the bus requests are synchronized first to an external high frequency bus clock and then sent to the priority encoder to be resolved. In this way, the resolving circuitry common to all microprocessors is kept at a minimum. Overall system reliability is improved in the sense that should a circuit which serves to synchronize the processor's request (which is now located on the same card as the microprocessor itself) fail, it is only necessary to remove that card from the system and the rest of the system will continue to function. Whereas in the other approach, should the synchronizing mechanism fail, the whole system goes down, as the synchronizing mechanism is located at the shared resource. In addition to the improved system reliability, moving the synchronization mechanism to the processor permits processor control over that mechanism, thereby permitting system flexibility (as will be shown) which could not be reasonably obtained by any other approach.

This synchronizing or arbitrating function was integrated into the 8289, a custom arbitration unit for the 8086, 8088, and 8089 processors. This note basically describes the 8289 arbitration unit, illustrates its different modes of operation and hardware connect in a multiprocessor system. Related and useful documents are: 8086 user's manual, 8289 data sheet, Article Reprint -55: Design Motivations for Multiple Processor Microcomputer Systems (which discusses implementing a semaphore with the MULTIBUS^{TM}) and Application Note 28A, Intel MULTIBUS^{TM} interfacing.

BUS ARBITER OPERATING CHARACTERISTICS

The 8289 Bus Arbiter operates in conjunction with the 8288 Bus Controller to interface an 8086, 8088, or 8089 processor to a multi-master system bus (the 8289 is used as a general bus arbitration unit). The processor is unaware of the arbiter's existence and issues commands as though it has exclusive use of the system bus. If the processor does not have the use of the multi-master system bus, the bus arbiter prevents the bus controller, the data transceivers and the address latches from accessing the system bus (i.e., all bus driver outputs are forced into the high impedance state). Since the command was not issued, a transfer acknowledge (XACK) will not be returned and the processor will enter into wait states. Transfer acknowledges are signals returned from the addressed resource to indicate to the processor that the transfer is complete. This signal is typically used to control the ready inputs of the clock generator. The processor will remain in wait until the bus arbiter acquires the use of the multi-master system bus, whereupon the bus arbiter will allow the bus controller, the data transceivers and the address latches to access the system bus. Once the command has been issued and a data transfer has taken place, a transfer acknowledge (XACK) is returned to the processor. The processor then completes its transfer cycle. Thus, the arbiter serves to multiplex a processor (or bus master) onto a multi-master system bus and avoid contention problems between bus masters.

Since there can be many bus masters on a multi-master system bus, some means of resolving priority between bus masters simultaneously requesting the bus must be provided. The 8289 Bus Arbiter provides for several resolving techniques. All the techniques are based on a priority concept that at a given time one bus master will have priority above all the rest. These techniques include the parallel priority resolving techniques, serial priority resolving and rotating priority techniques.

A parallel priority resolving technique has a separate bus request (BREQ) line for each arbiter on the multi-master bus (see Figure 1). Each BREQ line enters into a priority encoder which generates the binary address of the highest priority BREQ line which is active at the inputs. The output binary address is decoded by a decoder to select the corresponding BPRN (bus priority in) line to be returned to the highest priority requesting arbiter. The arbiter receiving priority (BPRN active low) then allows its associated bus master onto the multi-master system bus as soon as it becomes available (i.e., it is no longer busy). When one bus arbiter gains priority over another arbiter, it cannot immediately seize the bus, it must wait until the present bus occupant completes its transfer cycle. Upon completing its transfer cycle, the present bus occupant recognizes that it no longer has priority and surrenders the bus, releasing BUSY. BUSY is an active low OR-tied signal line which goes to every bus arbiter on the system bus. When BUSY goes high, the arbiter which presently has bus priority (BPRN active low) then seizes the bus and pulls BUSY low to keep other arbiters off the bus. (See waveform timing diagram, Figure 2.) Note that all multi-master system bus transactions are synchronized to the bus clock (BCLK). This allows for the parallel priority resolving circuitry or, any other priority resolving scheme employed, time to settle and make a correct decision.

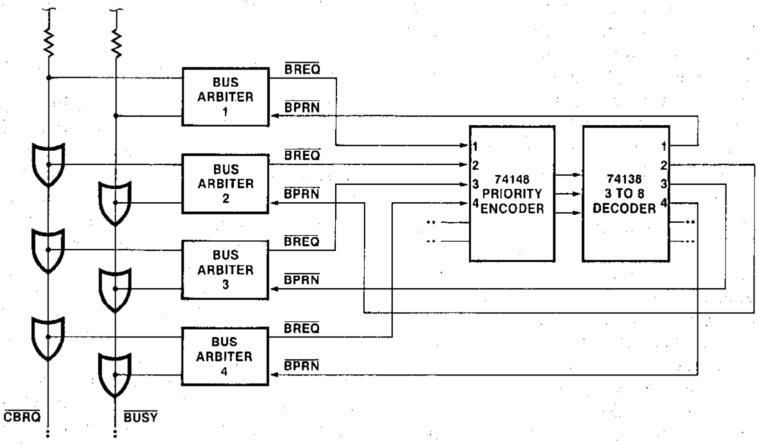
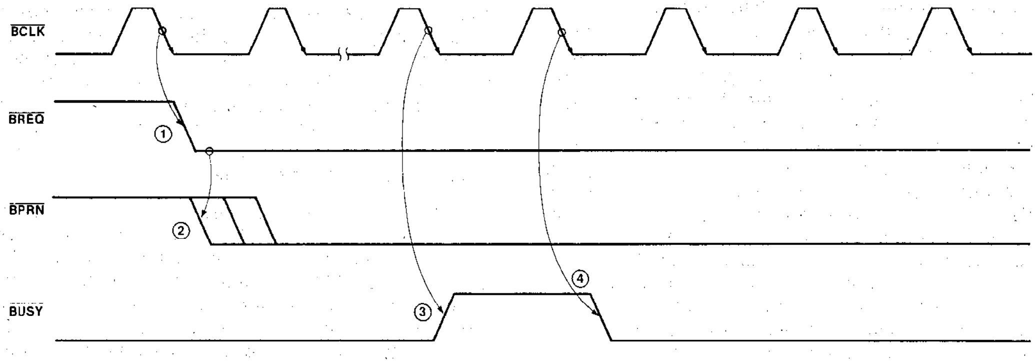


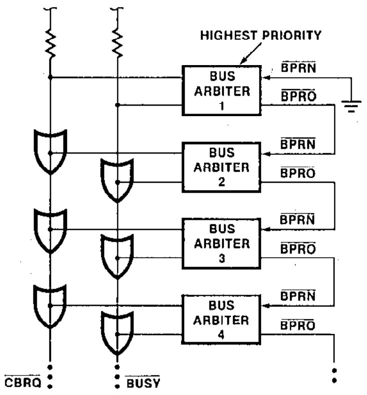
Figure 1. Parallel Priority Resolving Technique



① HIGHER PRIORITY BUS ARBITER REQUESTS THE MULTI-MASTER SYSTEM BUS.
② ATTAINS PRIORITY.
③ LOWER PRIORITY BUS ARBITER RELEASES BUSY.
④ HIGHER PRIORITY BUS ARBITER THEN ACQUIRES THE BUS AND PULLS BUSY DOWN.

Figure 2. Higher Priority Arbiter Obtaining The Bus From A Lower Priority Arbiter

A serial priority resolving technique eliminates the need for the priority encoder-decoder arrangement by daisy-chaining the bus arbiters together. This is accomplished by connecting the higher priority bus arbiter's $\overline{\text{BPRO}}$ (bus priority out) output to the $\overline{\text{BPRN}}$ of the next lower priority (see Figure 3). The highest priority bus arbiter would have its $\overline{\text{BPRN}}$ line grounded, signifying to the arbiter that it always has highest priority when requesting the bus.



THE NUMBER OF ARBITERS THAT MAY BE DAISY-CHAINED TOGETHER IN THE SERIAL PRIORITY RESOLVING TECHNIQUE IS A FUNCTION OF $\overline{\text{BCLK}}$ AND THE PROPAGATION DELAY FROM ARBITER TO ARBITER. NORMALLY, AT 10 MHz ONLY 3 ARBITERS MAY BE DAISY-CHAINED. SEE TEXT.

Figure 3. Serial Priority Resolving

A rotating priority resolving technique arrangement is similar to that of the parallel priority resolving technique except that priority is dynamically reassigned. The priority encoder is replaced by a more complex circuit which rotates priority between requesting arbiters, thus guaranteeing each arbiter equal time on the multi-master system bus.

There are advantages and disadvantages for each of the techniques described above. The rotating priority resolving technique requires an extensive amount of logic to implement, while the serial technique can accommodate only a limited number of bus arbiters before the daisy-chain propagation delay exceeds the multi-master system bus clock ($\overline{\text{BCLK}}$). The parallel priority resolving technique is, in general, the best compromise. It allows for many arbiters to be present on the bus while not requiring much logic to implement.

Whatever resolving technique is chosen, it is the highest priority bus arbiter requesting use of the multi-master system bus which obtains the bus. Exceptions do exist with the 8289 Bus Arbiter where a lower priority arbiter may take away the bus from a higher priority arbiter without the need for any additional external logic. This is accomplished through the use of the $\overline{\text{CBRQ}}$ pin, discussed in a later section.

MULTI-MASTER SYSTEM BUS SURRENDER AND REQUEST

The 8289 Bus Arbiter provides an intelligent interface to allow a processor or bus master of the 8086 family to access a multi-master system bus. The arbiter directs the processor onto the bus and allows both higher and lower priority bus masters to acquire the bus. Higher priority masters obtain the bus when the present bus master utilizing the bus completes its transfer cycle (including hold time). Lower priority bus masters obtain the bus when a higher priority bus master is not accessing the system bus and a lower priority arbiter has pulled $\overline{\text{CBRQ}}$ low. This signifies to the arbiter presently holding the multi-processor bus that a lower priority arbiter would like to acquire the bus when it is not being used. A strapping option (ANYRQST) allows the multi-master system bus to be surrendered to any bus master requesting the bus, regardless of its priority. If there are no other bus masters requesting the bus, the arbiter maintains the bus as long as its associated bus master has not entered the HALT state. *The 8289 Bus Arbiter will not voluntarily surrender the system bus and has to be forced off by another bus master.* An exception to this can be obtained by strapping $\overline{\text{CBRQ}}$ low and ANYRQST high. In this configuration the 8289 will release the bus after each transfer cycle.

How the 8289 Bus Arbiter is configured determines the manner in which the arbiter requests and surrenders the system bus. If the arbiter is configured to operate with a processor which has access to both a multi-master system bus and a resident bus, the arbiter requests the use of the multi-master system bus only for system bus accesses (i.e., it is a function of the SYSB/$\overline{\text{RESB}}$ input pin). While the processor is accessing the resident bus, the arbiter permits a lower priority bus master to seize the system bus via $\overline{\text{CBRQ}}$, since it is not being used. A processor configuration with both an I/O peripheral bus and a system bus behaves similarly. If the processor is accessing the peripheral bus, the arbiter permits the surrendering of the multi-master system bus to a lower priority bus master. To request the use of the multi-master system bus, the processor must perform a system memory access (as opposed to an I/O access).

The arbiter decodes the processor status lines to determine what type of access is being performed and behaves correspondingly. For simpler system configurations, such as a processor which accesses only a multi-master system bus, the arbiter requests the use of the system bus when it detects the status lines initiating a transfer cycle. The decoding of these status lines can be referenced in the 8086, 8088 (non-I/O processor) data sheets or the 8089 (I/O processor) data sheet.

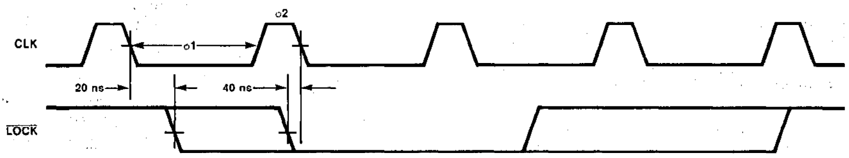
There is one condition common to all system configurations where the multi-master system bus is surrendered to a lower priority bus master requesting the bus by pulling $\overline{\text{CBRQ}}$ low. This is the idle or inactive state (TI) which is unique to the 8086 and 8088 processor family. This TI state comes about due to the processor's ability to fetch instructions in advance and store them internally for quick access. The size of the internal queue was optimized so that the processor would make the most effective use of its resources and be slightly execution bound. Since the processor can fetch code faster than it can execute it, it will fill to capacity its internal storage queue. When this occurs, the processor will enter into idle or inactive states (TI) until the processor has executed some of the code in the storage queue. Once this occurs, the processor will exit the TI state and again start code fetching. Between entering into and exiting from the TI state an indeterminate number of TI states can occur during which the bus arbiter permits the surrendering of the multi-master system bus to a lower priority bus master. As noted earlier and worth repeating here, once the 8289 Bus Arbiter acquires the use of the multi-master system it will not voluntarily surrender the bus and has to be forced off by another bus master. This will be discussed in more detail later.

Two other signals, $\overline{\text{LOCK}}$ and $\overline{\text{CRQLCK}}$ (Figure 4), lend to the flexibility of the 8289 Bus Arbiter within system configurations. $\overline{\text{LOCK}}$ is a signal generated by the processor to prevent the bus arbiter from surrendering the multi-master system bus to any other bus master, either higher or lower priority. $\overline{\text{CRQLCK}}$ (common request lock) serves to prevent the bus arbiter from surrendering the bus to a lower priority bus master when conditions warrant it. $\overline{\text{LOCK}}$ is used for implementing software semaphores for critical code sections and real time critical events (such as refreshing or hard disk transfers).

8289 BUS ARBITER INTERFACING TO THE 8288 BUS CONTROLLER

Once the 8289 Bus Arbiter determines to either allow its associated processor onto the multi-master system bus or to surrender the bus, it must guarantee that command setup and hold times are not violated. This is a two part problem. One, guaranteeing hold time and two, guaranteeing setup time. The 8288 Bus Controller performs the actual task of establishing setup time, while the 8289 Bus Arbiter establishes hold time (see Figure 5).

The 8289 Bus Arbiter communicates with the 8288 Bus Controller via the $\overline{\text{AEN}}$ line. When the arbiter allows its associated processor access to the multi-master system bus, it activates $\overline{\text{AEN}}$. $\overline{\text{AEN}}$ immediately enables the address latches and data transceivers. The bus controller responds to $\overline{\text{AEN}}$ by bringing its command output buffers out of high impedance state but keeping all commands disqualified until command setup time is established. Once established, the appropriate command is then issued. $\overline{\text{AEN}}$ is brought to the false state after the command hold time has been established by the arbiter when surrendering the bus.

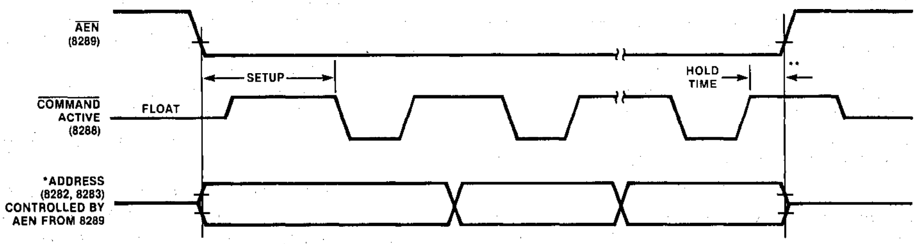


LOCK TIMING

THE ONLY CRITICAL LOCK TIMING IS THAT SHOWN ABOVE. $\overline{\text{LOCK}}$ MUST BE ACTIVATED NO SOONER THAN 20 ns INTO ϕ1 AND NO LATER THAN 40 ns PRIOR TO THE END OF ϕ2. $\overline{\text{LOCK}}$ INACTIVE HAS NO CRITICAL TIMING AND CAN BE ASYNCHRONOUS.

$\overline{\text{CRQLCK}}$ HAS NO CRITICAL TIMING AND IS CONSIDERED AS AN ASYNCHRONOUS INPUT SIGNAL.

Figure 4. Lock Timing


$\overline{\text{AEN}}$ (8289)
SETUP
HOLD TIME
**
$\overline{\text{COMMAND}}$ ACTIVE (8288)
FLOAT
*ADDRESS (8282, 8283) CONTROLLED BY AEN FROM 8289

*ADDRESSES ARE ACTIVATED IMMEDIATELY WHILE COMMAND IS DELAY TO ESTABLISH SETUP TIME REQUIREMENTS.

**THE 8289 ARBITER INTERNALLY TRACKS THE PROCESSOR CYCLE TO ESTABLISH THE PROPER AMOUNT OF HOLD TIME AFTER THE COMMAND HAS GONE INACTIVE.

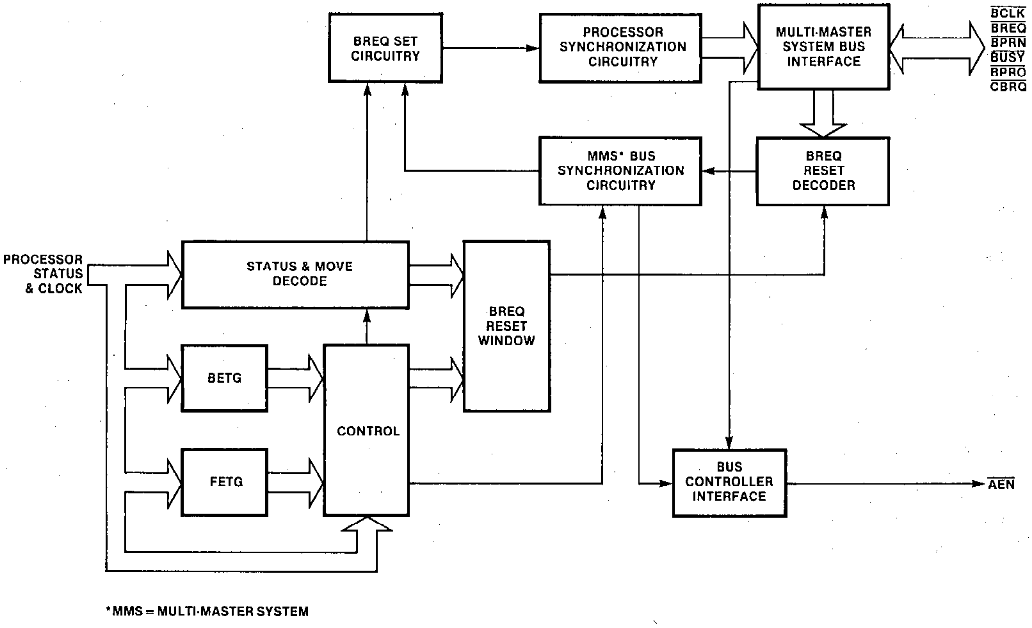
Figure 5. Single Bus Interface Timing

8289 BUS ARBITER INTERNAL ARCHITECTURE

A block diagram of the internal architecture of the 8289 Bus Arbiter is shown in Figure 6. It is useful to understand this block diagram when discussing the different modes of the 8289 and their impact on processor bus operations; however, you may want to skip this section to "8086 family processor types and system configurations" and return to it afterwards, as this section addresses the very involved reader. The front end state generator (FETG) and the back end state generator (BETG) allow the arbiter to track the processor cycle. An examination of an 8086 family processor state timings show that all command and control signals are issued in states T1 and T2 while being terminated in states T3 and T4, with an indeterminate number of wait states (Tw) occurring in between. Note further, that an indeterminate number of idle or inactive states can occur immediately proceeding and following a given transfer cycle. Since an indeterminate number of wait states can occur, two state generators are required; one to generate control signals (the FETG) and one to terminate control signals (the BETG). The FETG is triggered into operation when the processor activates the status lines. The FETG is reset and the BETG is triggered into operation by the status lines going to the passive condition. The BETG is reset when the status lines again go active.

It is necessary for the 8289 Bus Arbiter to track the processor in order that it is properly able to determine where and when to request or surrender the use of the multi-master system bus. In system configurations which access a resident bus, the use of the multi-master system bus is requested later in order to allow time for the SYSB/$\overline{\text{RESB}}$ input to become valid. For systems which access a peripheral bus, the arbiter issues a request for the system bus only for memory transfer cycles which it decodes from the status lines (and time must be allowed for the status lines to become valid and then decoded). In a system which accesses only a multi-master system bus, a request is made as soon as the arbiter detects an active-going transition on the processor's status lines. Thus, when the processor initiates a transfer cycle, the FETG is triggered into operation and, depending upon what mode the arbiter is configured in, the STATUS & MODE DECODE circuitry initiates a request for the system bus at the appropriate time. The request enters the BREQ SET circuitry where it is then synchronized to the multi-master system bus clock ($\overline{\text{BCLK}}$) by the PROCESSOR SYNCHRONIZATION circuitry.* Once synchronized, the multi-master system bus interface circuitry issues a $\overline{\text{BREQ}}$. When the priority resolving circuitry returns a $\overline{\text{BPRN}}$ (bus priority in), the PROCESSOR SYNCHRONIZATION circuitry seizes the bus the next time it becomes available (i.e., $\overline{\text{BUSY}}$ goes high) by pulling $\overline{\text{BUSY}}$ low one $\overline{\text{BCLK}}$ after it goes high and enables $\overline{\text{AEN}}$. (See waveform timing diagram in Figure 2). Once the arbiter acquires the use of the system bus and a data exchange has taken place (a transfer acknowledge, XACK, was returned to the processor), the processor status lines go passive and the

*Due to the asynchronous nature of processor trasnsfer request to the multi-master system bus clock, it is necessary to synchronize the processor's transfer request to BCLK.



*MMS = MULTI-MASTER SYSTEM

Figure 6. 8289 Bus Arbiter Block Diagram

BETG is triggered into operation. The BETG provides the timing for the bus surrender circuitries in the event that conditions warrant the surrender of the multimaster bus, i.e., the bus arbiter lost priority to a higher bus master or the processor has entered into TI states and $\overline{\text{CBRQ}}$ is pulled low, etc. If such is the case, the BREQ RESET DECODER initiates a bus surrender request. The bus surrender request is synchronized by the MMS BUS SYNCHRONIZATION CIRCUITRY to the processor clock. The MMS BUS SYNCHRONIZATION CIRCUITRY instructs the bus controller interface circuitry to make $\overline{\text{AEN}}$ go false and resets the BREQ SET circuitry. Resetting the BREQ SET circuitry will cause its output to go false and be synchronized by the processor synchronization, eventually instructing the MULTI-MASTER SYSTEM BUS INTERFACE circuitry to reset $\overline{\text{BREQ}}$. In the event that a lower priority arbiter has caused the arbiter to surrender the bus, it is necessary that $\overline{\text{BREQ}}$ be reset. Resetting $\overline{\text{BREQ}}$ allows the priority resolving circuitry to generate $\overline{\text{BPRN}}$ to the next highest priority bus master requesting the bus. The BREQ RESET WINDOW circuitry provides a 'window' wherein the arbiter allows the multi-master system bus to be surrendered and serves as part of the MMS bus-processor synchronization circuitry.

8086 FAMILY PROCESSOR TYPES AND SYSTEM CONFIGURATIONS

There are two types of processors in the 8086 family — an I/O processor (the 8089 IOP) and a non-I/O processor (the 8086 and 8088 CPUs). Consequently, there are two basic operating modes in the 8289 Bus Arbiter. One, the IOB (I/O peripheral bus) mode, permits the processor access to both an I/O peripheral bus and a multi-master system bus. The second, the RESB (resident bus) mode, permits the processor to communicate over both a resident bus and a multi-master system bus. Even though it is intended for the arbiter to be configured in the IOB mode when interfacing to an I/O processor and for it to be in the RESB mode when interfacing to a non-I/O processor, it is quite possible for the reverse to be true. That is, it is possible for a non-I/O processor to have access to an I/O peripheral bus or for an I/O processor to have access to a resident bus as well as access to a multimaster system bus. The IOB strapping option configures the 8289 Bus Arbiter into the IOB mode and RESB strapping option configures it into the resident bus mode. If both strapping options are strapped false, a third mode of operation is created, the single bus mode, in which the arbiter interfaces the processor to a multi-master system bus only. With both options strapped true, the arbiter interfaces the processor to a multimaster system bus, a resident bus and an I/O bus.

To better understand the 8289 Bus Arbiter, each of the operating modes, along with their respective timings, are examined by means of examples. The simplest configuration, the Single Bus Configuration, (both IOB and RESB strapped inactive) will be considered first, followed by the I/O bus Configuration and the Resident Bus Configuration. Finally, brief mention is made of a configuration that allows the processor to interface to two multi-master system buses. This particular configuration is briefly mentioned because, as will be seen, it is simply an extension of the resident bus configuration. When discussing the Single Bus Configuration, processor/arbiter, arbiter/system bus and internal arbiter, considerations are made resulting in a table that illustrates overhead in requesting the system bus. As this applies to the other 8289 configurations, only additional considerations will be given. A summary of when to use the different configurations is given at the end.

8289 SINGLE BUS INTERFACE

Figure 7 shows a block diagram of a bus master which has to interface only to a system bus — preferably the MULTIBUS — where there exists more than one bus master. In later configurations, it will be shown how the processor can be made to interface with more than one bus. Since the processor has only to interface with one bus, this configuration is called "single".

Connecting the 8289 Bus Arbiter to the processor is as simple as it was to connect the 8288 Bus Controller. Namely, the three status lines, $\overline{\text{S0}}$, $\overline{\text{S1}}$, and $\overline{\text{S2}}$ are directly connected from the processor to the arbiter. The clock line from the 8284 Clock Generator is brought down and connected. (Note that both the 8288 Bus Controller and the 8289 Bus Arbiter are connected to the same clock, CLK and not the peripheral clock, PCLK as the 8086 processor.) From the arbiter, $\overline{\text{AEN}}$ is connected to the bus controller and to the clock generator. The $\overline{\text{IOB}}$ pin on the arbiter is strapped high and on the controller the IOB pin is strapped low. In addition, the RESB pin on the arbiter is strapped low, finishing the processor interface.

Some flexibility exists with the MULTIBUS or multimaster system bus interface. The system designer must first decide upon the type of priority resolving scheme to be employed, whether it is to be the serial, parallel, or rotating priority scheme. A rotating priority scheme would be employed where the system designer would want to guarantee that every bus master on the bus would be given time on the bus. In the serial and parallel schemes, the possibility exists that the lowest assigned priority bus master may not acquire the bus for long periods of time. This occurs because priority is permanently assigned and if bus demand is high by the higher assigned priorities, then the lower priorities must wait. In most cases, this situation is acceptable because the highest priority is assigned to the bus master that cannot wait. Highest priority is usually assigned to DMA type devices where service requirements occur in real time. CPUs are assigned the lower priorities. For the purpose of this discussion, the parallel priority scheme will be used with brief reference to the serial priority scheme.

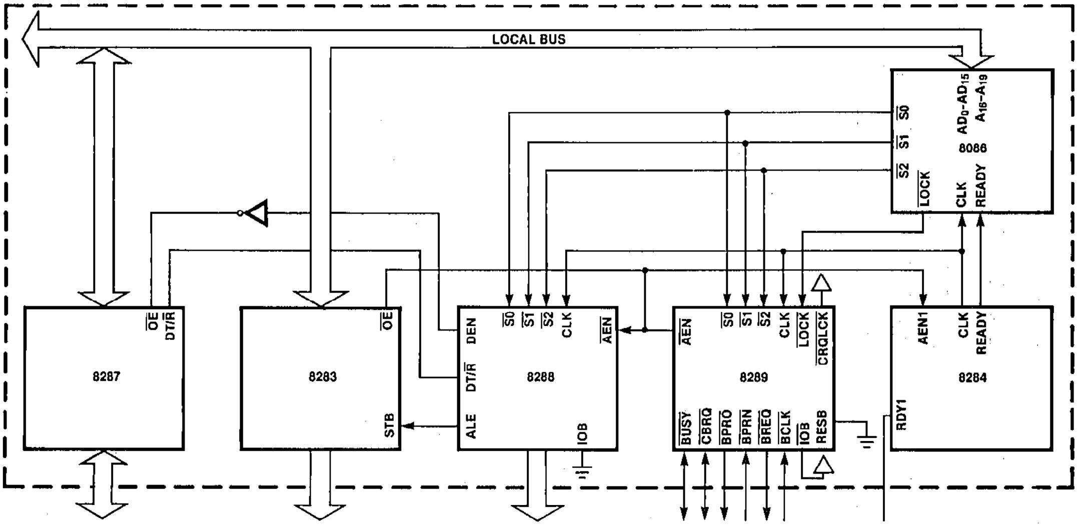


Figure 7. Single Multimaster Bus Interface

Figure 8 shows how a typical multi-processing system might be configured with the 8289 in the Single Bus mode. In the system there are three bus masters, each having the assigned priority as indicated—priority 1 being the highest and priority 3 being the lowest. Priority is established using the parallel priority scheme (ignore the dotted signal interconnect for the moment). Each bus arbiter monitors its associated processor and issues a bus request ($\overline{BREQ}$) whenever its processor wants the bus. A common clocking signal ($\overline{BCLK}$) runs to each of the arbiters in the system. It is from the falling edge of this clock that all bus requests are issued. Since all bus requests are made on the same clock edge, a valid priority can be established by the priority resolving circuitry by the next falling $\overline{BCLK}$ edge. Note that all multi-master system bus (MULTIBUS) input signals are considered to be valid at the falling edge of $\overline{BCLK}$. And that all multi-master system bus output signals are issued from the falling edge of $\overline{BCLK}$. With the parallel resolving module, arbiters 2 and 3 would issue their respective $\overline{BREQ}$s (Figure 9) on the falling edge of $\overline{BCLK\ 1}$, as shown. The outputs ($\overline{BPRN\ 1}$, $\overline{BPRN\ 2}$, and $\overline{BPRN\ 3}$) of the priority encoder-decoder arrangement change to reflect their new input conditions and need to be valid early enough in front of $\overline{BCLK\ 2}$ to guarantee the arbiter's setup time requirements. Since arbiter 2 at the time is the highest priority arbiter requesting the bus, bus priority is given to arbiter 2 ($\overline{BPRN\ 2}$ goes low), and since the bus was not busy ($\overline{BUSY}$ is high) at the time priority was granted to arbiter 2, arbiter 2 pulls $\overline{BUSY}$ inactive on $\overline{BCLK\ 2}$, thereby seizing the bus and excluding all other arbiters access to the bus. Once the bus is seized, arbiter 2 activates its $\overline{AEN}$. $\overline{AEN}$ going low directly enables the 8283 address latches and wakes up the 8288 Bus Controller. The bus controller enables the 8287 transceivers, waits until the address to command setup time has been established, and then enables its command drivers onto the bus.

If the serial priority resolving mode was used instead, much of the events that happened for the parallel priority resolving mode would be the same except, of course, there would be no parallel priority resolving module. Instead, the system would be connected as indicated in Figure 8 by the dotted signal lines connecting the $\overline{BPRO}$ of one arbiter to $\overline{BPRN}$ of the next lower priority arbiter.

The $\overline{BREQ}$ lines would be disconnected and the priority encoder-decoder arrangement removed. This arrangement is simpler than the parallel priority arrangement except that the daisy-chain propagation delay of the highest priority bus arbiter's $\overline{BPRO}$ to the lowest priority bus arbiter's $\overline{BPRN}$, including setup time requirement ($\overline{BPRN}$ to $\overline{BLCK}$), cannot exceed the $\overline{BCLK}$ period. In short, this means there are only so many arbiters that can be daisy-chained for a given $\overline{BCLK}$ frequency. Of course, the lower the $\overline{BCLK}$ frequency, the more arbiters can be daisy-chained. The maximum $\overline{BCLK}$ frequency is specified at 10 MHz, which would allow for three 8289 arbiters to be daisy-chained. In general, the number of arbiters that can be connected in the serial daisy-chain configuration can be determined from the following equation:

$$\overline{BCLK}\text{ period} \geq \text{TBLPOH} + \text{TPNPO}\,(N-1) + \text{TPNBL}$$

where N = # of arbiters in system

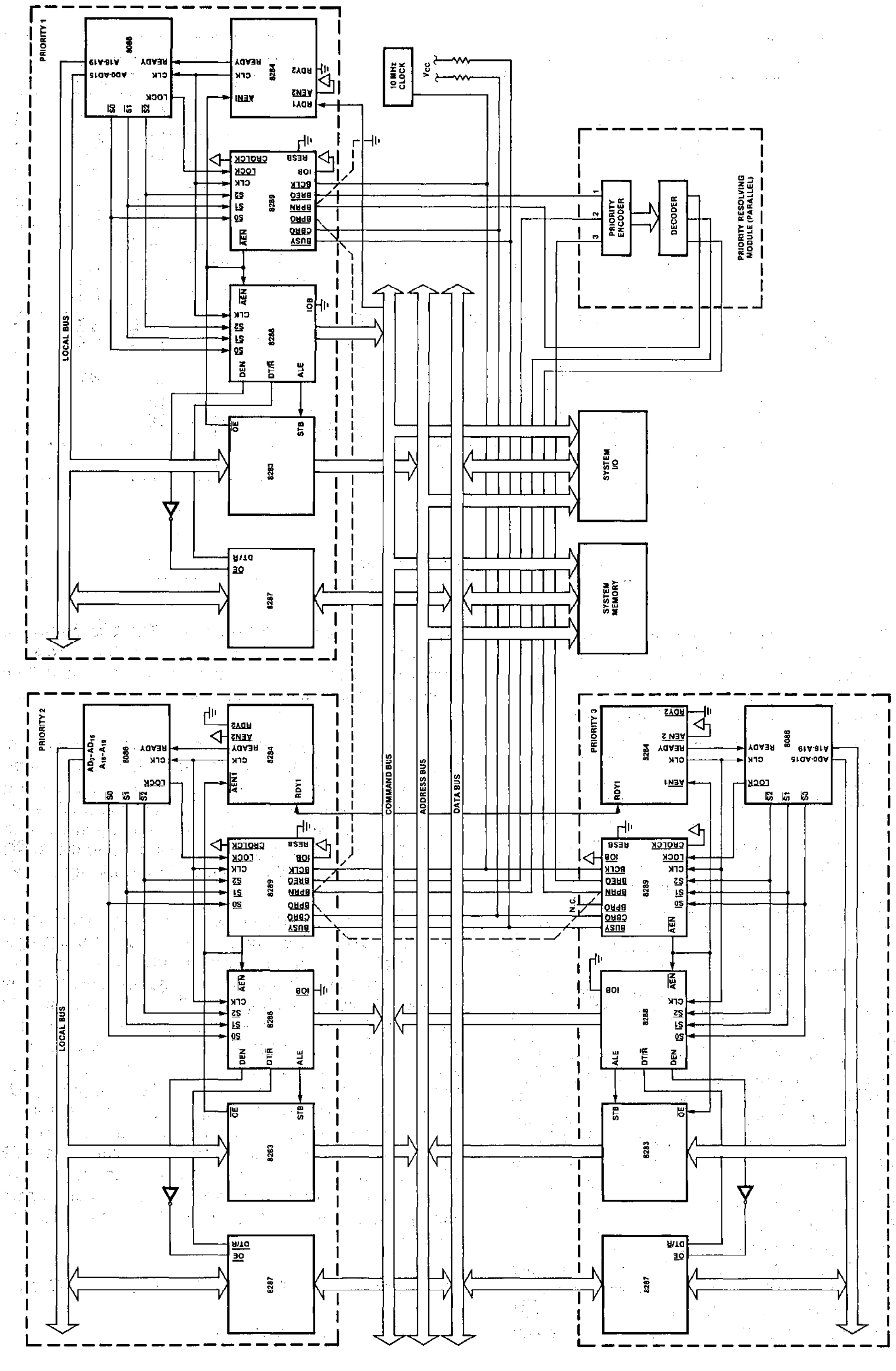
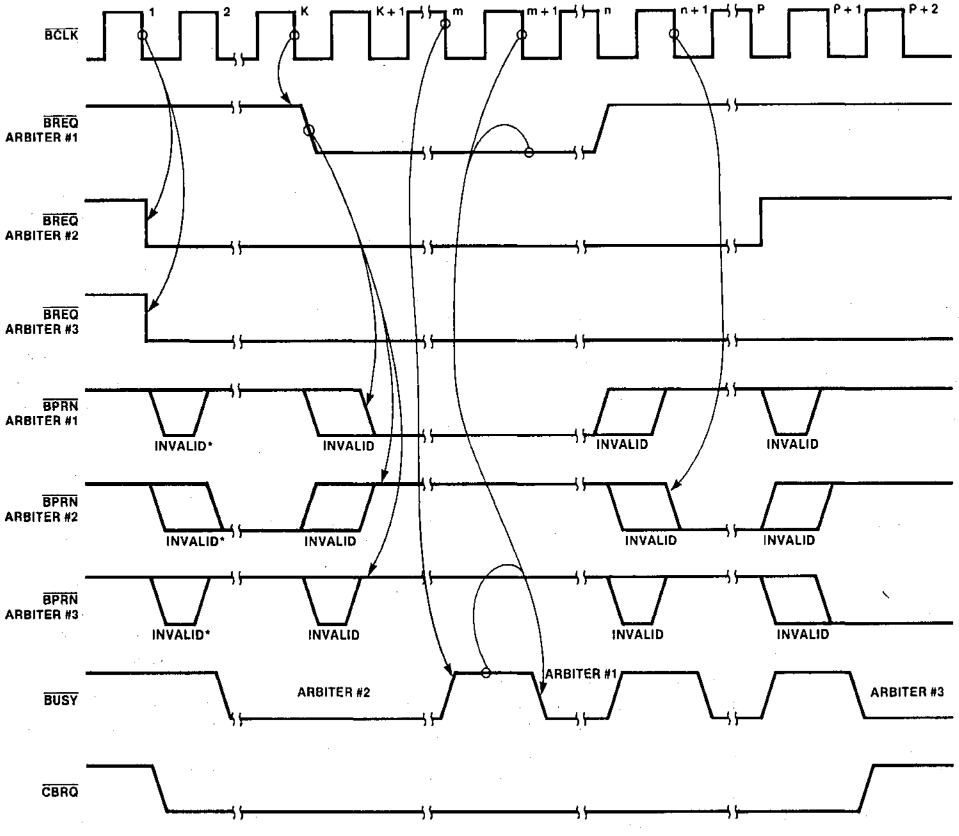


Figure 8. Multiprocessing System With 8289 In Single Bus Mode



*DECODING GLITCHES ARE PERMITTED

Figure 9. Example Timing For Figure 8

Returning to Figure 9, it can be seen that K BCLKs later, arbiter 1 has decided to request the bus and its BREQ, BREQ 1, has gone low. Since arbiter 1 is of higher priority than arbiter 2, which presently has the bus, bus priority is reassigned by the priority module (or the daisy-chain approach in the serial priority) to arbiter 1. BPRN 1 goes low and BPRN 2 now goes high (BPRN 3 remains high, even though decoding can cause it to glitch momentarily). The loss of priority instructs arbiter 2 that a higher priority arbiter wants the bus and that it is to release the bus as soon as its present transfer cycle is done. Since arbiter 2 cannot immediately release the bus, arbiter 1 must wait. In the particular case illustrated in Figure 9, arbiter 2 releases the bus (allows BUSY to go high) on clock edge M, and on clock edge M + 1, arbiter 1 now seizes the bus, pulling BUSY low. Arbiter 1 is the highest priority arbiter in the system and it now has the bus. Arbiters 2 and 3 still want the bus (their BREQs are both low).

How quickly arbiter 1 can acquire the bus is dependent upon the configuration and strapping options of the arbiter it is trying to acquire it from. For example, if the LOCK input to arbiter 2 was active (low) at the time, then arbiter 1, even though it was of higher priority, would not have acquired the bus until after LOCK was released (goes high). Effectively, LOCK locks the arbiter onto the bus once the bus has been acquired. LOCK will not force another arbiter to release the bus any sooner, it just prevents the bus from being given away no matter what the priority of the other arbiter. Another factor to be considered is where in the transfer cycle is the processor when the arbiter is instructed to give up the bus. Obviously, if the cycle had just started, it will take longer for the bus to be released than if the cycle was just ending. Another factor to be included in this consideration is the phase relationship of the processor's clock (CLK) to the bus clock (BCLK). This relationship is examined in more detail later on. Table 1 lists the time requirements for various arbiter actions such as bus acquisition and bus release (under $\overline{\text{LOCK}}$ and other circumstances) taking into account the phase relationships between CLK and $\overline{\text{BCLK}}$.

Bus Request (BREQ↓)	Mode	Delay (Max)	Delay (Min)
Status→BREQ↓	Single	2 $\overline{\text{BCLK}}$s	1 $\overline{\text{BCLK}}$
Status→BREQ↓	IOB	2 $\overline{\text{BCLK}}$s + ~1 CLK*	1 $\overline{\text{BCLK}}$ + ~½ CLK*
Status→BREQ↓	RESB	2 $\overline{\text{BCLK}}$s + ~2 CLKs†	1 $\overline{\text{BCLK}}$ + ~1½ CLKs†
Status→BREQ↓	IOB•RESB	2 $\overline{\text{BCLK}}$s + ~2 CLKs†	1 $\overline{\text{BCLK}}$ + 1½ CLKs†

*Request originates off of $\phi 2$ of T1 and $\overline{\text{BREQ}}$↓ occurs 1 $\overline{\text{BCLK}}$ (min) to 2 $\overline{\text{BCLK}}$s (max) thereafter. Depending upon where status occurs with respect to clock determines how long a time exists between status and $\phi 2$ of T1, and is anywhere from ½ CLK (min) to 1 CLK (max).

†Request originates off of T2•$\phi 1$ and $\overline{\text{BREQ}}$↓ occurs 1 $\overline{\text{BCLK}}$ (min) to 2 $\overline{\text{BCLK}}$s (max) thereafter. The same reasoning as used in the IOB mode is valid here.

Bus Release (BREQ↑)	Mode	Delay (Max)	Delay (Min)
Higher Priority (BPRN↑)	All	2 CLKs + 2 $\overline{\text{BCLK}}$s	1 CLK + 1 $\overline{\text{BCLK}}$
Lower Priority (CBRQ↓)	All	2 CLKs + 2 $\overline{\text{BCLK}}$s	1 CLK + 1 $\overline{\text{BCLK}}$
Surrender occurs once the proper surrender conditions exist.			

Table 1. Surrender and Request Time Delays

One signal which has been basically ignored to this point is $\overline{\text{CBRQ}}$. $\overline{\text{CBRQ}}$, like $\overline{\text{BUSY}}$, is an open-collector signal from the arbiter which is tied to the $\overline{\text{CBRQ}}$ signals of the other arbiters and to a pull-up resistor (see Figure 8). $\overline{\text{CBRQ}}$ is both an input and an output. As an output, $\overline{\text{CBRQ}}$ serves to instruct the arbiter presently on the bus that another arbiter wishes to acquire the bus. As an input, $\overline{\text{CBRQ}}$ serves to instruct the arbiter presently on the bus that another arbiter wants the bus. $\overline{\text{CBRQ}}$ is an input or output, dependent on whether the arbiter is on the bus or not (respectively), and is issued as a function of $\overline{\text{BREQ}}$. Thus, a lower priority arbiter requesting the bus already controlled by a higher priority arbiter will pull $\overline{\text{CBRQ}}$ low, as well as $\overline{\text{BREQ}}$. Even a higher priority arbiter will pull $\overline{\text{CBRQ}}$ low until it acquires the bus. Note, however, that the higher priority arbiter will acquire the bus through the reassignment of priorities — it being given priority and the other arbiter presently on the bus losing it. In effect, $\overline{\text{CBRQ}}$ serves to notify the arbiter that an arbiter of lower priority wants the bus.

If the arbiter presently on the bus is configured to react to $\overline{\text{CBRQ}}$ and the proper surrender conditions exist, the bus is released. When releasing the bus, the arbiter also turns off its $\overline{\text{BREQ}}$ ($\overline{\text{BREQ}}$ goes high) in order to allow priority to be established to the next lower arbiter requesting the bus. Such is the case shown in Figure 9. Whereas it was assumed that the proper surrender conditions did not exist for arbiter 2 when it had the bus, it is assumed that the proper conditions do exist during the time that arbiter 1 has the bus. Arbiter 2 had to give up the bus because an arbiter of higher priority was requesting it. Arbiter 1 surrenders the bus because the proper surrender conditions exist and a lower priority arbiter requested the bus by pulling $\overline{\text{CBRQ}}$ low. This is an assumed condition which is not otherwise shown in Figure 9. This is not an unrealistic condition. Normally, a higher priority arbiter will acquire the bus through the reassignment of priorities, while lower priority arbiters acquire the bus through $\overline{\text{CBRQ}}$.

Digressing for a moment, the 8289 Bus Arbiter will not voluntarily surrender the bus (except when the processor halts execution). As a result, it has to be forced off the bus. The 8289 Bus Arbiter does not generate a $\overline{\text{BREQ}}$ for each cycle. It generates a $\overline{\text{BREQ}}$ once and then hangs onto the bus. To do otherwise would require that $\overline{\text{BREQ}}$ be dropped (go high) after each transfer cycle so that if it did need to do another transfer cycle, another arbiter would automatically be assigned priority. This approach, however, entails certain overhead. Command to address setup and hold time must be prefixed and appended to each transfer cycle. Each transfer cycle would be characterized by first acquiring the bus, then establishing the setup time requirements, finally performing the transfer cycle, establishing the hold time requirements, and then releasing the bus (see Figure 10). If another transfer cycle was to immediately follow and if the arbiter still had priority, then the whole above procedure would be repeated. The end result would be wasted time as hold times following setup times (see Figure 10A). The approach taken by the 8289 Bus Arbiter of having to be forced off the bus, even when it is not using the bus (i.e., forced off by a lower priority arbiter), provides for greater bus efficiency. A lower priority arbiter having to force off another arbiter that is not using the bus but just hanging on to it, may not seem very efficient. In actuality it is a good trade-off. In many multi-master systems some bus masters occasionally demand the bus, while others demand the bus constantly. The bus master which constantly demands the bus may momentarily need not to access the bus. Why should that arbiter surrender the bus when chances are that the other bus masters which occasionally access the bus don't want it at the time? If it doesn't give up the bus, then it can momentarily cease access to the bus and then continue, without any performance penalty of having to reestablish control of the bus. The greater bus efficiency that it affords is well worth the added complexity (Figure 10B).

Returning to Figure 9, the combination of the proper surrender conditions existing and $\overline{\text{CBRQ}}$ being low, forced the higher priority arbiter, arbiter 1, off the bus. Arbiter 2, being of next higher priority and wanting the bus, acquired the bus on clock edge N + 1. If arbiter 1 decides to re-access the bus, it would reacquire the bus through the reassignment of priorities. This is not the case shown in Figure 9. Arbiter 1 has decided that it does not need the bus and does not renew its $\overline{\text{BREQ}}$. Arbiter 2, having acquired the bus through $\overline{\text{CBRQ}}$, is now the highest priority arbiter requesting the bus. As can be seen it is not the only arbiter requesting the bus. Arbiter 3 is still patiently waiting for the bus and $\overline{\text{CBRQ}}$ remains low. The same conditions that forced arbiter 1 off the bus for arbiter 2 now forces arbiter 2 off the bus for arbiter 3. When the proper surrender conditions exist, arbiter 2 releases its $\overline{\text{BREQ}}$ and surrenders the bus to arbiter 3. Arbiter 3 acquires the bus on clock edge P+1 and releases its $\overline{\text{CBRQ}}$. Since no other arbiter wants the bus (i.e., there is no other arbiter holding $\overline{\text{CBRQ}}$ low), $\overline{\text{CBRQ}}$ goes high (inactive). This would have also been true when arbiter 2 acquired the bus and released its $\overline{\text{CBRQ}}$ if arbiter 3 didn't want the bus.

In the Single interface, the arbiter monitors the processor's status lines, which are activated whenever the processor performs a transfer cycle. The arbiter, on detecting the status lines going active, will issue a $\overline{\text{BREQ}}$ if the status is not the HALT status. If the processor issues the HALT status, the arbiter will not request the bus, and if it has the bus, will release it.

This effectively concludes how arbiters interact to one another on the bus. Having examined the processor-to-arbiter interface, and arbiter-to-MULTIBUS (arbiter-to-arbiter) interaction, one interface is left, the internal interface of processor-related signals to that of MULTIBUS-related signals.

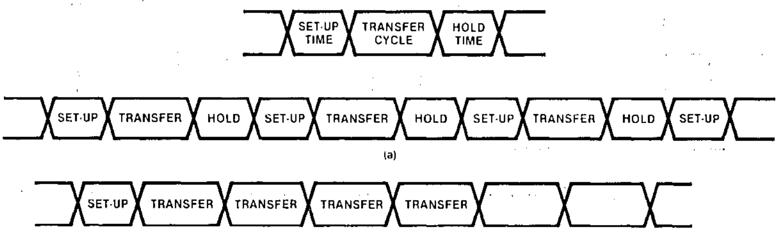
An important point to remember is that the processor has its own clock (CLK) and the multi-master system bus has its own ($\overline{\text{BCLK}}$). These two clocks are usually out of phase and of different frequencies. Thus, the arbiter must synchronize events occurring on one interface to events occurring on another interface. As a result of this back and forth synchronization, ambiguity can arise as to when events actually do take place.

Very simply, the 8289 arbiter operation can be represented as two events, requesting and surrendering. Figure 11 is a representation of the timing relationships involved. The request input is a function of the processor's clock and the surrender input is a function of either the bus clock or the processor's clock. To request the bus, the processor activates its status lines which in turn enables the request input. Depending upon the phase relationship between the occurrence of status (request active) and $\overline{\text{BCLK}}$, $\overline{\text{BREQ}}$ appears one to two $\overline{\text{BCLK}}$s later. As shown in Figure 12, the phase relationship between request and $\overline{\text{BCLK}}$ is such that the BRQ1 flip-flop may or may not catch request on the first $\overline{\text{BCLK}}$.*

If BRQ1 flip-flop does catch the request, then one $\overline{\text{BCLK}}$ later, $\overline{\text{BREQ}}$ goes low and one $\overline{\text{BCLK}}$ after that, $\overline{\text{BUSY}}$ goes low (it is assumed that priority is immediately granted and that the bus is available). If BRQ1 flip-flop does not catch the request, then request is caught on the next $\overline{\text{BCLK}}$ and $\overline{\text{BREQ}}$ goes low one $\overline{\text{BCLK}}$ later, followed by $\overline{\text{BUSY}}$ which also goes low one $\overline{\text{BCLK}}$ later. Note that $\overline{\text{BREQ}}$ and $\overline{\text{BUSY}}$ track, as $\overline{\text{BREQ}}$ is an input term for $\overline{\text{BUSY}}$. During bus acquisition, the surrender flip-flop is false (SURNDR Q=low) and $\overline{\text{AEN}}$ follows $\overline{\text{BUSY}}$.

Once the bus is acquired, the surrender circuitry is enabled so that when a valid surrender condition exists, the bus can be surrendered. The surrender circuitry synchronizes the surrender request to the processor's clock and drives SURNDR low. Like the acquisition circuitry, it takes from one to two processor clocks to generate SURNDR and depends upon the phase relationship between the surrender request and the processor's clock.

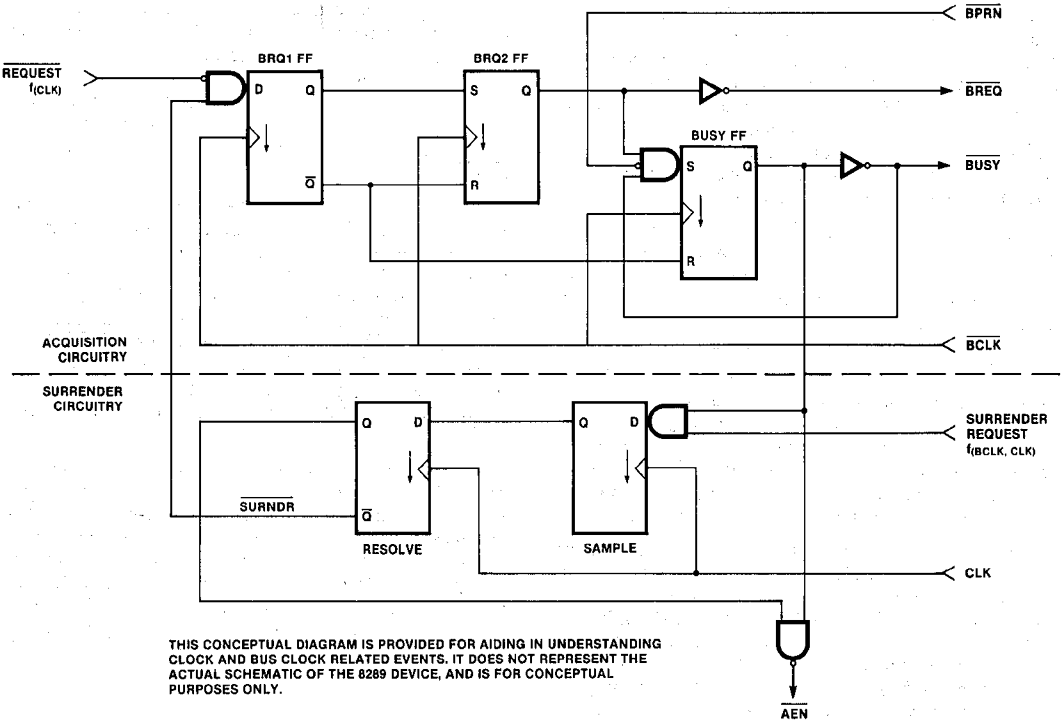
*The two bus request flip-flops, BRQ1 and BRQ2, are edge-triggered, high resolution flip-flops and serve to reduce the probability of walkout down to an acceptable level. Walkout occurs because $\overline{\text{BCLK}}$ is asynchronous with respect to request. If walkout does occur on BRQ1 flip-flop, the probability is high that the BRQ1 flip-flop will resolve itself prior to BRQ2 flip-flop being triggered. Even if BRQ1 flip-flop did not quite resolve itself, the probability of BRQ2 flip-flop walking out to an unacceptable point in time is itself low.



a) BUS UTILIZATION AS A RESULT OF HAVING TO REQUEST AND RELEASE THE BUS FOR EACH TRANSFER CYCLE. THIS PERMITS LOWER PRIORITY ARBITERS EASY ACCESS TO THE BUS SHOULD THE HIGHER PRIORITY ARBITER NO LONGER NEED THE BUS. HOWEVER, BUS EFFICIENCY IS POOR DUE TO THE ARBITER THRASING ON AND OFF OF THE BUS FOR EACH TRANSFER CYCLE.

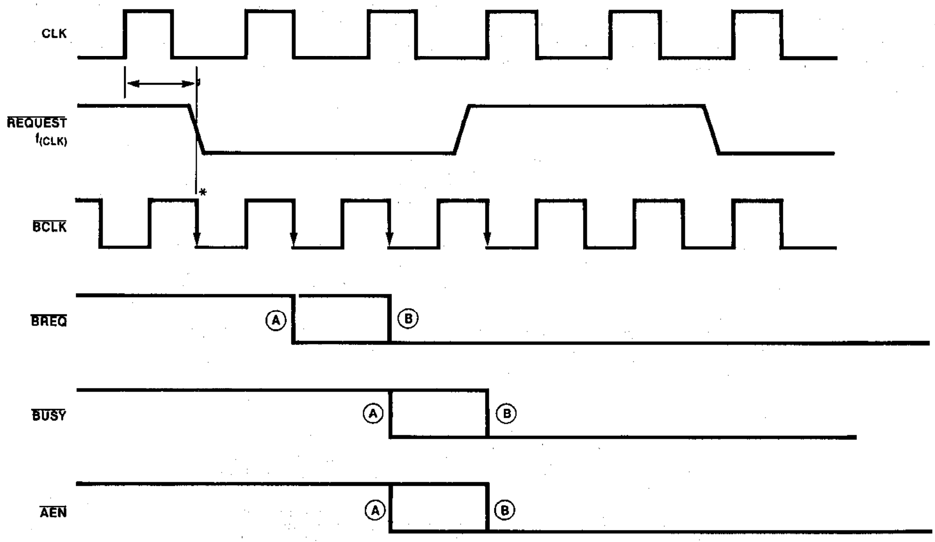
b) 8289 BUS UTILIZATION IS MORE EFFICIENT IN THAT THE ARBITER HAS ONLY TO ACQUIRE THE BUS ONCE. THE 8289 HANGS ONTO THE BUS UNTIL FORCED OFF. THIS APPROACH ADDS A LITTLE MORE COMPLEXITY TO THE SYSTEM INASMUCH AS SOME MEANS MUST BE PROVIDED FOR LOWER PRIORITY ARBITERS TO FORCE THE HIGHER PRIORITY ARBITER OFF OF THE BUS WHEN IT IS NOT USING IT. THE ADDED COMPLEXITY IS WELL WORTH THE BUS EFFICIENCY AND SYSTEM FLEXIBILITY IT AFFORDS. THE 8289 ARBITER CAN BE CONFIGURED TO HAVE THE TRANSFER TIMING AS SHOWN IN (a) (IMITATING THE METHOD 8218 AND 8219 USES, BUS ARBITERS FOR 8080 AND 8085 RESPECTIVELY) BY STRAPPING ANYRQST HIGH AND $\overline{\text{CBREQ}}$ LOW.

Figure 10. Two Techniques For Doing Multibus Transfer Cycles



THIS CONCEPTUAL DIAGRAM IS PROVIDED FOR AIDING IN UNDERSTANDING CLOCK AND BUS CLOCK RELATED EVENTS. IT DOES NOT REPRESENT THE ACTUAL SCHEMATIC OF THE 8289 DEVICE, AND IS FOR CONCEPTUAL PURPOSES ONLY.

Figure 11. Symbolic Representation of Internal 8289 Timing



*WHEN THE REQUEST OCCURS SIMULTANEOUSLY WITH BCLK, BCLK MAY OR MAY NOT CATCH THE REQUEST. IF IT DOES, THE WAVEFORMS FOLLOW THOSE SHOWN DESIGNATED BY Ⓐ . IF NOT, THE REQUEST IS PICKED UP ON THE NEXT EDGE OF BCLK AND THE WAVEFORMS FOLLOW THOSE SHOWN DESIGNATED BY Ⓑ .

Figure 12. Results Of An Asynchronous Event

Having synchronized the surrender request to the processor's clock to generate SURNDR, SURNDR is then synchronized to $\overline{\text{BCLK}}$ to reset the BUSY and BRQ flip-flops. When BUSY-Q goes low, the surrender circuitry is reset which in turn re-enables the request input. The timing in Figure 13 shows the surrender request input going high on the falling edge of the clock. If the Sample flip-flop was able to catch the surrender request on the edge of clock 1, then SURNDR would be generated (go low) on clock edge 2. If not, SURNDR would be generated on clock edge 3. SURNDR going low on clock edge 2 will be, for ease of discussion, referred to as SURNDR a and SURNDR going low on clock edge 3 will be referred to as SURNDR b. As can be seen from Figure 13, SURNDR a just happens to go low on $\overline{\text{BCLK}}$ edge 2. Since SURNDR is used to reset the BRQ flip-flops, which are clocked by the falling edge of $\overline{\text{BCLK}}$, the BRQ1 flip-flop may or may not catch SURNDR a on $\overline{\text{BCLK}}$ edge 2. If it does, then BRQ and $\overline{\text{BUSY}}$ go high on $\overline{\text{BCLK}}$ edge 3 which, for convenience, will be called $\overline{\text{BREQ}}$ a or $\overline{\text{BUSY}}$ a. If not, theh $\overline{\text{BREQ}}$ and $\overline{\text{BUSY}}$ will go high on $\overline{\text{BCLK}}$ edge 4, which will be referred to as $\overline{\text{BREQ}}$ b or $\overline{\text{BUSY}}$ b, respectively. SURNDR b occurs early enough to assure that $\overline{\text{BUSY}}$ and $\overline{\text{BREQ}}$ are reset on $\overline{\text{BCLK}}$ edge 5, which will be referred to as $\overline{\text{BUSY}}$ b1 and $\overline{\text{BREQ}}$ b1. Depending upon when $\overline{\text{BUSY}}$ goes high, determines when the surrender circuitry is reset and how soon the next $\overline{\text{BREQ}}$ can be generated. $\overline{\text{BUSY}}$ a1 causes SURNDR c to occur where shown and SURNDR c in turn would allow the earliest bus request to occur at $\overline{\text{BREQ}}$ c1. At the other extreme, $\overline{\text{BUSY}}$ b1 allows the earliest bus request to occur at $\overline{\text{BREQ}}$ e1.

Table 1 summarizes the maximum and minimum delays for bus request, once the proper request and surrender conditions exist. Table 2 lists the proper surrender conditions.

Mode	Surrender Conditions
Single	HALT state, loss of BPRN, TI•CBREQ
IOB	HALT state, loss of BPRN, TI•CBREQ, I/O Command•CBRQ
RESB	HALT state, loss of BPRN, TI•CBREQ, (SYSB/RESB = 0)•CBRQ
IOB•RESB	HALT state, loss of BPRN, TI•CBREQ, (SYSB/RESB = 0)•CBREQ, I/O Command•CBRQ

Table 2. Surrender Conditions

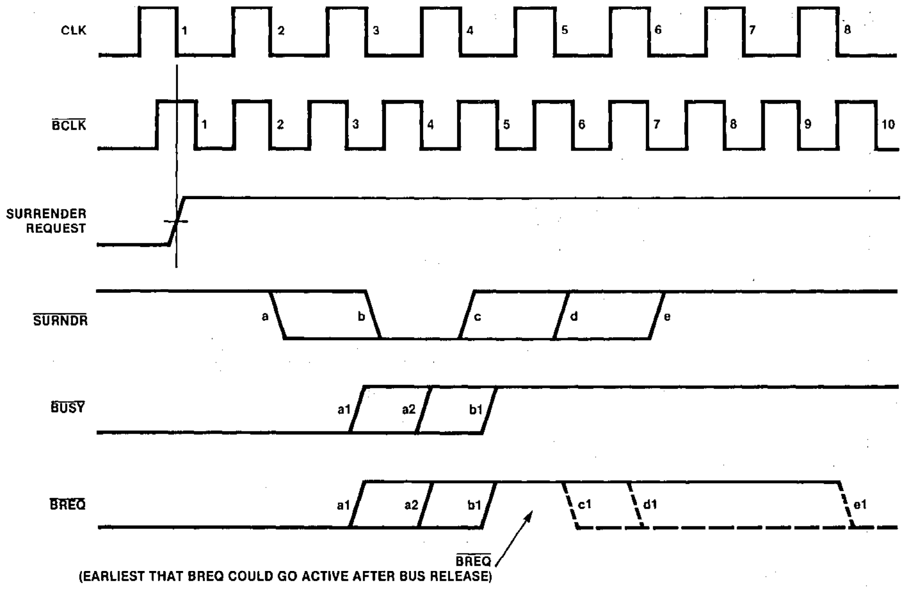


Figure 13. Asynchronous Bus Release

IOB INTERFACE

Now that the processor-arbiter, arbiter-system bus and internal arbiter timings have been discussed, it is appropriate to consider the other interfaces that the 8289 Bus Arbiter provides.

In the IOB mode, the processor communicates and controls a host of peripherals over the peripheral bus. When the I/O processor needs to communicate with system memory, it is done so over the system memory bus. Figure 14 shows a possible I/O processor system configuration, utilizing the 8089 I/O processor in its REMOTE mode. Resident memory exists on the peripheral bus in order that canned I/O routines and buffer storage can be provided. Resident memory is treated as an I/O peripheral. When a peripheral device needs servicing, the I/O processor accesses resident memory for the proper I/O driver routine and services the device, transmitting or storing peripheral data in buffer storage area of resident memory. The resident memory's buffer storage area could then be emptied or replenished from system memory via the system bus. Using the IOB interface allows an I/O processor the capability of executing from local memory (on the peripheral bus) concurrently with the host processor.

Timing in this mode is no different from timing in the SINGLE BUS mode. The only difference lies in the request and surrender conditions. The arbiter extends the single bus mode conditions to qualify when the system bus is requested and adds on additional surrender conditions. The system bus is only requested during system bus commands (the arbiter decodes the processor's status lines) and, in addition to the other surrender terms, the arbiter permits surrender to occur during I/O bus (or local bus) commands, when the I/O processor is using its own local bus.

Like the arbiter, the bus controller must also be informed of the mode it is operating in. In the IOB mode, the 8288 bus controller issues I/O bus commands independently of the state of $\overline{\text{AEN}}$ from the arbiter. It is assumed that all I/O bus commands are intended for the I/O bus and hence there is a separate I/O command bus from the controller. All I/O bus commands are sent directly to the I/O bus and are not influenced by $\overline{\text{AEN}}$. System bus commands are assumed as going to the system bus. Since system bus commands are directed to the system bus, they must still be influenced by $\overline{\text{AEN}}$ and the arbitration mechanism provided by the 8289.

As an example, suppose the processor issues an I/O bus command. The 8288 Bus Controller generates the necessary control signal to latch the I/O address and configure the transceivers in the correct direction. In the IOB mode, the multiplexed MCE/$\overline{\text{PDEN}}$ pin of the 8288 becomes $\overline{\text{PDEN}}$ (peripheral data enable) and serves to enable the I/O bus's data transceivers during I/O bus commands. DEN similarily serves to enable the system bus's data transceivers during memory commands. $\overline{\text{PDEN}}$ and DEN are mutually exclusive, so it is not possible for both sets of transceivers to be on, thereby avoiding contention between the two sets. Since the I/O bus commands are generated independently of $\overline{\text{AEN}}$ in the IOB mode, the I/O bus has no delay effects due to the arbiter. During this time in which the processor is accessing memory the arbiter, if it already has the bus, will permit it to be surrendered to either a higher or lower priority independently of where the processor is in

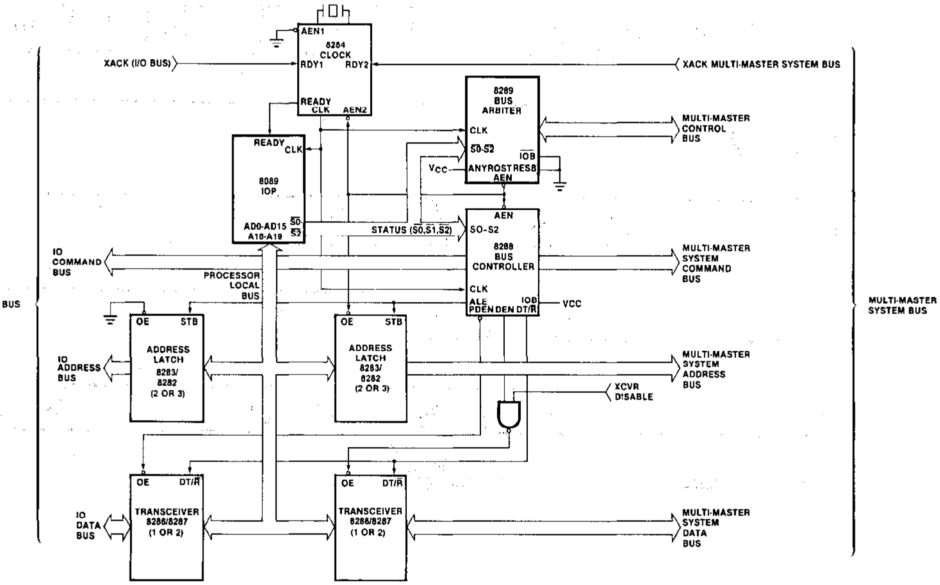


Figure 14. 8289 Configured In I/O Bus Mode With 8089 I/O Processor

its transfer cycle (i.e., independent of the machine state).* If the arbiter does not already have the bus, it will make no effort to acquire the bus.

If the processor issues a memory command instead, the same set of events take place, except that 1) the system bus's data transceivers are enabled instead of the peripherals bus's data transceivers, and 2) when the command is issued depends upon the state of the arbiter. In both cases of I/O bus commands and system bus commands, the address generated for that command is latched into both sets of address latches, the system bus's address latches, and the peripherals bus's address latches. For each command (regardless of command type), an address is put out on the I/O bus and on the system bus if the arbiter has the bus at that particular time. However, the bus controller only issues a command to one of the buses and hence, no ill effects are suffered by addressing both buses.

If the arbiter already has the system bus when a system bus command is issued, no delays due to the arbiter will be noticed by the processor. If the arbiter doesn't have the bus and must acquire it, then the processor will be delayed (via the system bus command being delayed by the bus controller through $\overline{\text{AEN}}$ from the arbiter) until the arbiter has acquired the bus. The arbiter will then permit the bus controller to issue the command and the transfer cycle continues.

RESB INTERFACE

The non-I/O processors in the 8086 family can communicate with both a resident bus and a multi-master system bus. Two bus controllers would be needed in such a configuration as shown in Figure 15. In such a system configuration the processor would have to access to memory and peripherals of both buses. Address mapping techniques can be applied to select which bus is to be accessed. The SYSB/$\overline{\text{RESB}}$ (system bus/resident bus) input on the arbiter serves to instruct the arbiter as to whether or not the system bus is to be accessed. It also enables or disables commands from one of the bus controllers.

In such a system configuration, it is possible to issue both memory and I/O commands to either bus and as a result, two bus controllers are needed, one for each bus. Since the controllers have to issue both memory and I/O commands to their respective buses, the IOB options on the controllers are strapped off (IOB is low). The arbiter, too, has to be informed of the system configuration in order to respond appropriately to system inputs and has its RESB option strapped on (RESB is high). The arbiter's IOB option is strapped inactive ($\overline{\text{IOB}}$ is high). Strapping the arbiter into the resident bus mode enables the arbiter to respond to the state of the SYSB/$\overline{\text{RESB}}$ input. Depending upon the state of this input, the arbiter either requests and acquires the system bus or permits the surrendering of that bus.

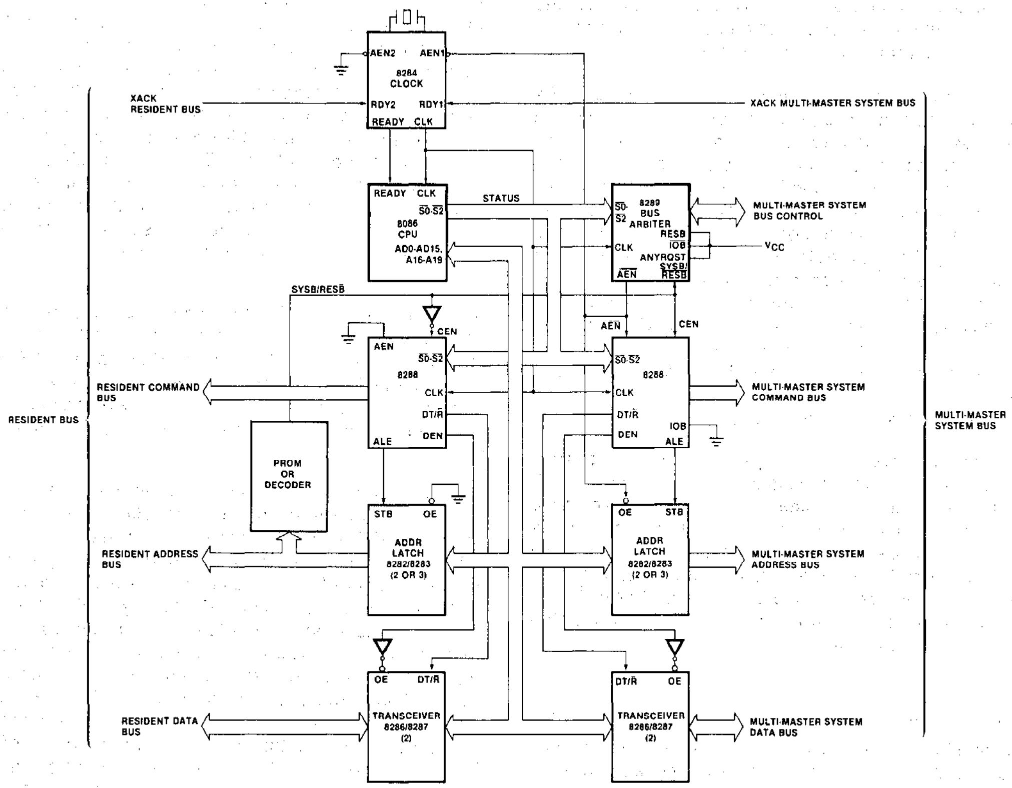
In the system shown in Figure 15, memory mapping techniques are applied on the resident bus side of the system rather than on the multiprocessor or system bus side. As mentioned earlier in the IOB interface, both sets of address latches (the resident bus's address latches and the system bus's address latches) are latched with the same address; in this case, by their respective bus controllers.* The system bus's address latches, however, may or may not be enabled depending upon the state of the arbiter. The resident bus's address latches are always enabled, hence the address mapping technique is applied to the resident bus.

Address mapping techniques can range in complexity from a single bit of the address bus (usually the most significant bit of the address), to a decoder, to a PROM. The more elaborate mapping technique, such as PROM, provides segment mapping, system flexibility, and easy mapping modifications (simply make a new PROM).

In actual operation, both bus controllers respond to the processor's status lines and both will simultaneously issue an address latch strobe (ALE) to their respective address latches. Both bus controllers will issue command and control signals unless inhibited. The purpose of the address mapping circuitry is to inhibit one of the bus controllers before contention or erroneous commands can occur. The transceivers are enabled off the same clock edge the commands are issued, namely $\phi1$ of T2 (Figure 16). The address is strobed into the address latches by ALE. ALE is activated as soon as the processor issues status, and is terminated on $\phi2$ of of T1. From when ALE is issued, plus the propagation delay of the address latches, determines where the address is valid. The time from which the address is valid to where control and commands are issued determines how much settling time is available for the address mapping circuitry. The mapping circuitry must inhibit (via CEN) one of the bus controllers prior to where controls and commands are issued. Part of the settling time (see Figure 16) is consumed as a setup time requirement to the bus controllers. As it turns out, CEN (command enable) can be disqualified as late as on the falling edge of clock (the leading edge of $\phi1$ of T2) without fear of the bus controller issuing any commands or transceiver control signals. In systems (8 MHz) where less time is available for the address mapping circuitry, the address latches can be bypassed, hooking the mapping circuitry straight onto the processor's multiplexed address/data bus (the local bus) and using ALE to strobe the mapping circuitry. This would avoid the propagation delay time of the transceivers. Besides needing to inhibit one of the bus controllers, the arbiter needs to be informed of the address mapping circuitry's decision. Depending upon that decision, the arbiter acquires or permits the release of the system bus.

*Under other circumstances, bus surrendering would only be permitted during the period from where address to command hold time has been established just prior to where the next command would be issued.

*A simpler system with an 8086 or 8088 can exist, if it is desirable to only have PROM, ROM, or a read only peripheral interface on the resident bus. The 8086 and 8088 additionally generate a read signal in conjunction with the 8288 control signals. By using this read signal and memory mapping, the 8086 or 8088 could operate from local program store without having the contention of using the system bus.



*BY ADDING ANOTHER 8289 ARBITER AND CONNECTING ITS AEN TO THE 8288 WHOSE $\overline{\text{AEN}}$ IS PRESENTLY GROUNDED, THE PROCESSOR COULD HAVE ACCESS TO TWO MULTI-MASTER BUSES.

Figure 15. 8289 Configured In Resident Bus Mode

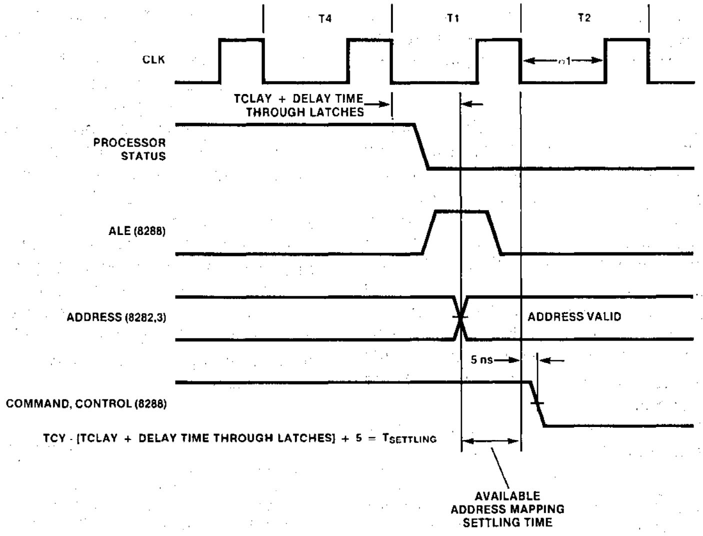


Figure 16. Time Available For Address Mapping Prom

The arbiter is informed of this decision via its SYSB/$\overline{\text{RESB}}$ input. If the memory mapping circuitry selects the resident bus, then SYSB/$\overline{\text{RESB}}$ input to the arbiter and CEN input of the system bus controller are brought low; and the CEN Input of the resident bus controller is brought high. The commands and control signals of the resident bus are now enabled and those of the system bus are disabled. In addition, with the arbiter being informed that the transfer cycle is occurring on the resident bus, the system bus is permitted to be surrendered. Glitching is permitted on the SYSB/$\overline{\text{RESB}}$ input of the arbiter up until $\phi 1$ of T2. Thereafter, only clean transitions can occur on the input.* So, if mapping circuitry can settle prior to $\phi 1$ of T2, there is no need to be concerned over glitching. If the mapping circuitry is unable to settle prior to this time, then the designer must guarantee a clean transition on the SYSB/$\overline{\text{RESB}}$ input.

INTERFACE TO TWO MULTI-MASTER BUSES

The interface of an 8086 family processor to two multisystem buses is simply an extension of the resident bus interface. The only difference is that now two arbiters are needed, one for each multi-master bus, and the address mapping circuitry must acquire its input straight off the processor's multiplexed address/data bus (the local bus), using ALE as an address strobe input. Figure 17 depicts how such a system might be configured.

Figure 17 illustrates the use of the 8289 in a system environment in three of its four modes. The host 8086 CPU (priority 3) is using the 8289 in its single bus multimaster mode, while an 8089 I/O processor is using the 8289 in its IOB mode. A work station based on an 8088 processor uses the 8289 in it system/resident bus mode. This diagram represents a hypothetical system wherein there can exist more than one work station (only one shown). Each work station shares system resources and I/O. The lowest priority processor (8086) would provide supervisory functions and system control, i.e., allow operator intervention into the system resources. A work station would call in assemblers and compilers or application programs as needed. When compiled or assembled, the results are transferred to the I/O station for output, thus freeing up a work station for another user.

If one work station is used, the serial priority resolving technique could be used between the 8289 Bus Arbiters (shown in dotted lines). If more than one work station is desired, it would be necessary to either slow down the system bus clock to accommodate the additional arbiters, or resort to the parallel resolving technique (as shown).

WHEN TO USE THE DIFFERENT MODES

Single Bus Multi-Master Interface

This mode is the simplest and is sufficient for systems where a multiprocessing environment exists and the system bus bandwidth is sufficient to handle the peak concurrent requirements of a multi-master environment. This solution can provide an inexpensive solution for multi-masters to access an expensive I/O device. If, however, the system bus bandwidth is exceeded, the IOB or system/resident modes should be considered.

IOB Mode

The IOB mode is ideal when the bus can be separated into an I/O bus and memory or system bus. This mode is commonly used with the 8089 I/O processor in its REMOTE configuration to separate the I/O space from memory space. With the 8089, all instructions operate on either system or I/O address space. 64K bytes of I/O space can be accessed by the processors in the 8086 family.

The remaining processors in the 8086 family are constrained to using only I/O instructions when referencing I/O space. If this is a limitation, and it is desirable to remove some of the processor functions to its private resources, the resident bus mode should be considered.

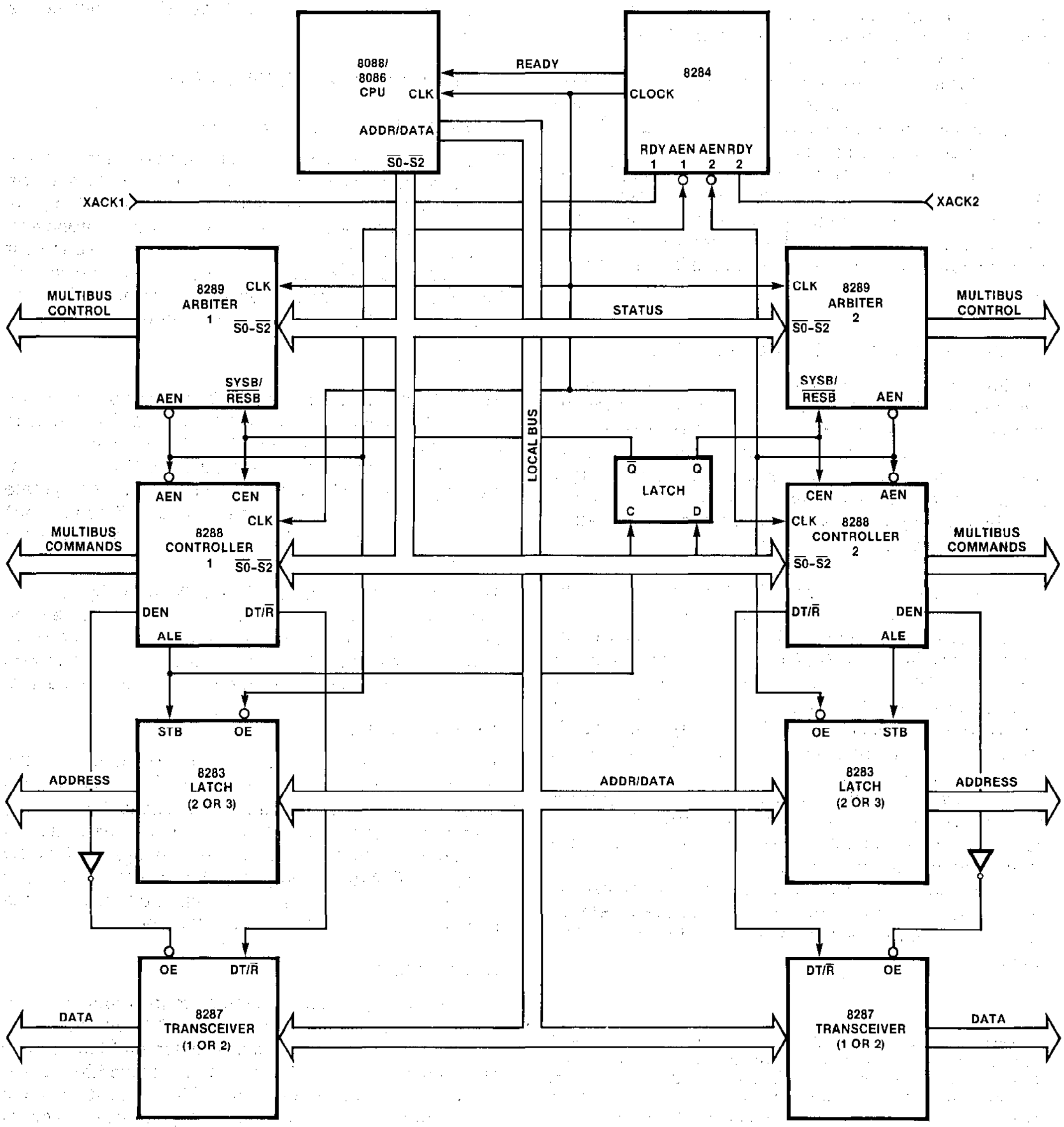
Resident Bus Mode

The resident bus mode allows for maximum flexibility for a CPU device, giving it both access to its own local resources with full instruction set capability, and the system resources. The CPU can work from its own local resources without contention on the system bus. By using a PROM for memory mapping, memory space can be easily altered in this mode. This mode requires the use of a second 8288 bus controller chip.

CONCLUSION

The 8289 brings a new dimension to microcomputer architecture by allowing the advanced 8/16-bit microprocessors to play easily in a multi-master, multiprocessing environment. With the flexible modes of the 8289, a user can define one of several bus architectures to meet his cost/performance needs. Modularity, improved system reliability and increased performance are just a few of the benefits that designing a multiprocessing system provides.

*In certain memory mapping techniques, the $\overline{\text{CEN}}$s of the bus controllers are controlled differently from the SYSB/$\overline{\text{RESB}}$ input of the arbiter. In short, CEN is brought low automatically to both bus controllers, thereby disabling their command and control outputs. This permits a longer settling time for the memory mapping circuitry, since both controllers are disabled. When the mapping circuitry settles, sometime after $\phi 1$ of T2, one of the bus controllers and its associated bus arbiter (if one exists) is enabled. After $\overline{\phi 1}$ of T2, the arbiter can only permit clean transitions on the SYSB/$\overline{\text{RESB}}$ input line.



MEMORY MAPPING DECODING IS SHOWN TAKING PLACE DIRECTLY OFF OF THE PROCESSOR'S LOCAL MULTIPLEXED ADDRESS/DATA BUS.

Figure 17. Using 8289s To Interface To Two Multimaster System Buses.

THIS PAGE LEFT INTENTIONALLY BLANK

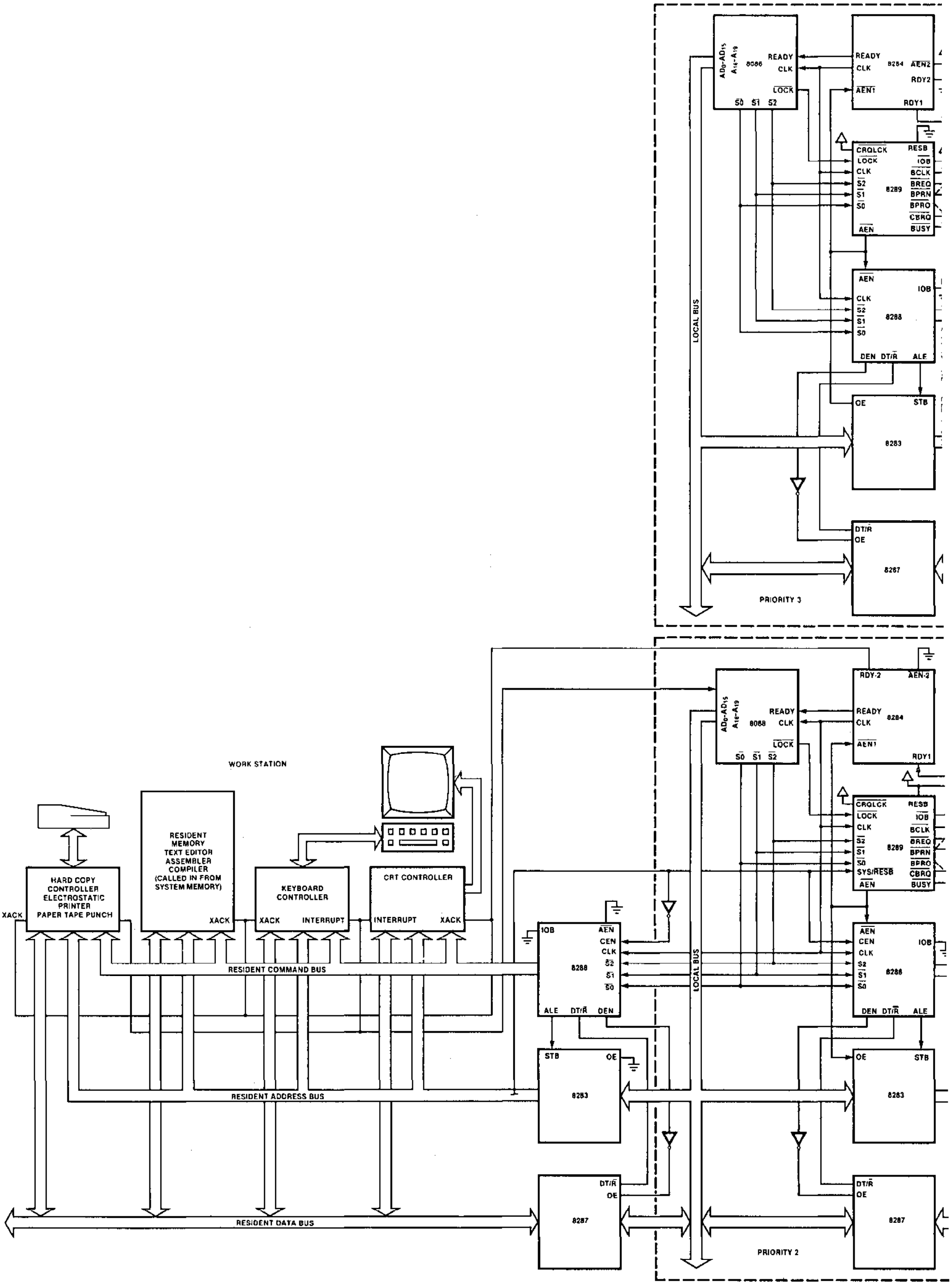


Figure 18. 8289 Used In Each Of 3 Modes, Single Bus, I/O Bus, and Resident Bus Modes Implementing A Hypothetical Multimaster Bus System

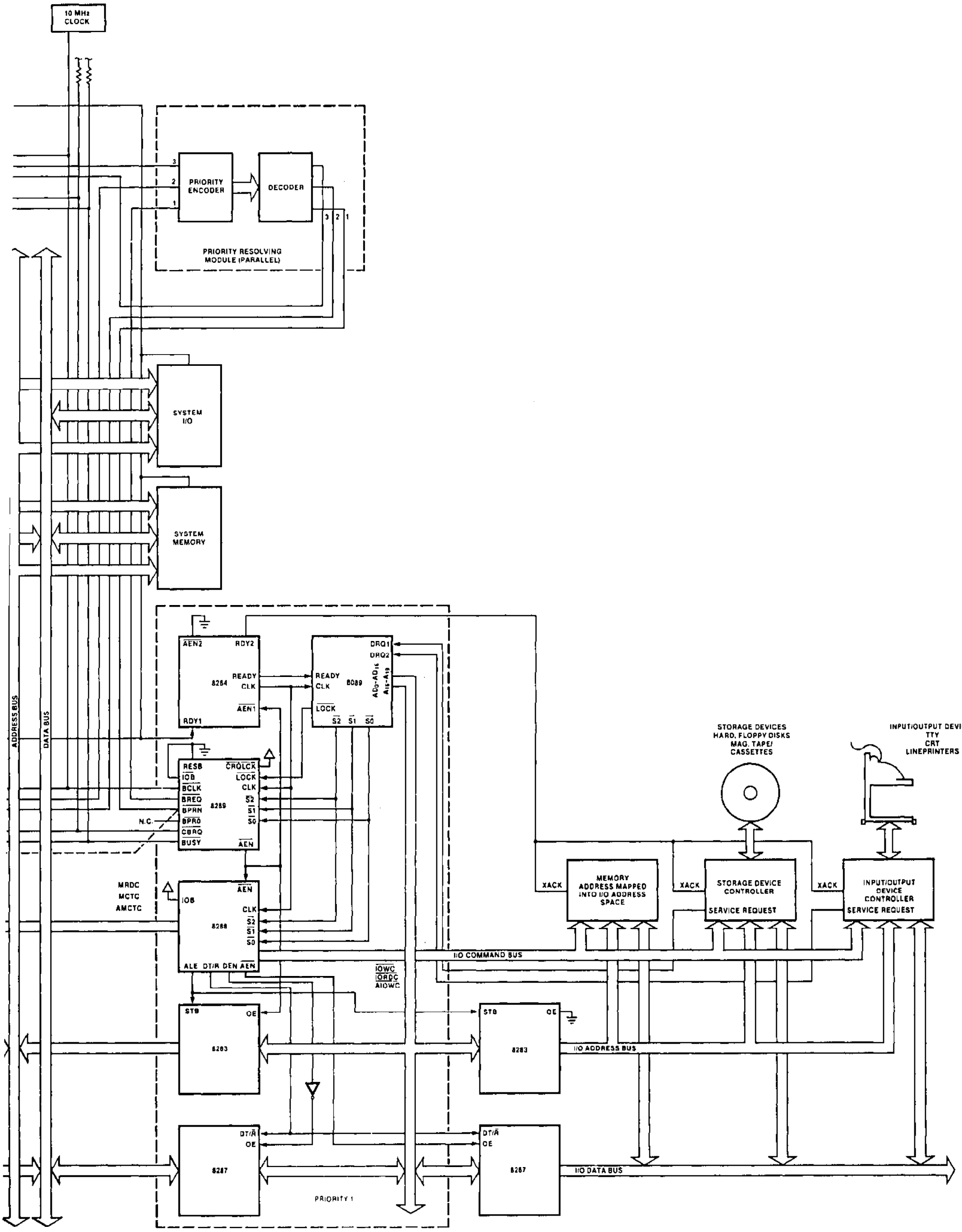


Figure 18. 8289 Used In Each Of 3 Modes, Single Bus, I/O Bus, and Resident Bus Modes Implementing A Hypothetical Multimaster Bus System

intel®

APPLICATION
NOTE

AP-59

September 1979

Using the 8259A Programmable Interrupt Controller

Robin Jigour
Microcomputer Applications

Using the 8259A Programmable Interrupt Controller

Contents

INTRODUCTION

The Intel 8259A is a Programmable Interrupt Controller (PIC) designed for use in real-time interrupt driven microcomputer systems. The 8259A manages eight levels of interrupts and has built-in features for expansion up to 64 levels with additional 8259A's. Its versatile design allows it to be used within MCS-80, MCS-85, MCS-86, and MCS-88 microcomputer systems. Being fully programmable, the 8259A provides a wide variety of modes and commands to tailor 8259A interrupt processing for the specific needs of the user. These modes and commands control a number of interrupt oriented functions such as interrupt priority selection and masking of interrupts. The 8259A programming may be dynamically changed by the software at any time, thus allowing complete interrupt control throughout program execution.

The 8259A is an enhanced, fully compatible revision of its predecessor, the 8259. This means the 8259A can use all hardware and software originally designed for the 8259 without any changes. Furthermore, it provides additional modes that increase its flexibility in MCS-80 and MCS-85 systems and allow it to work in MCS-86 and MCS-88 systems. These modes are:

- MCS-86/88 Mode
- Automatic End of Interrupt Mode
- Level Triggered Mode
- Special Fully Nested Mode
- Buffered Mode

Each of these are covered in depth further in this application note.

This application note was written to explain completely how to use the 8259A within MCS-80, MCS-85, MCS-86, and MCS-88 microcomputer systems. It is divided into five sections. The first section, "Concepts", explains the concepts of interrupts and presents an overview of how the 8259A works with each microcomputer system mentioned above. The second section, "Functional Block Diagram", describes the internal functions of the 8259A in block diagram form and provides a detailed functional description of each device pin. "Operation of the 8259A", the third section, explains in depth the operation and use of each of the 8259A modes and commands. For clarity of explanation, this section doesn't make reference to the actual programming of the 8259A. Instead, all programming is covered in the fourth section, "Programming the 8259A". This section explains how to program the 8259A with the modes and commands mentioned in the previous section. These two sections are referenced in Appendix A. The fifth and final section "Application Examples", shows the 8259A in three typical applications. These applications are fully explained with reference to both hardware and software.

The reader should note that some of the terminology used throughout this application note may differ slightly from existing data sheets. This is done to better clarify and explain the operation and programming of the 8259A.

1. CONCEPTS

In microcomputer systems there is usually a need for the processor to communicate with various Input/Output (I/O) devices such as keyboards, displays, sensors, and other peripherals. From the system viewpoint, the processor should spend as little time as possible servicing the peripherals since the time required for these I/O chores directly affects the amount of time available for other tasks. In other words, the system should be designed so that I/O servicing has little or no effect on the total system throughput. There are two basic methods of handling the I/O chores in a system: status polling and interrupt servicing.

The status poll method of I/O servicing essentially involves having the processor "ask" each peripheral if it needs servicing by testing the peripheral's status line. If the peripheral requires service, the processor branches to the appropriate service routine; if not, the processor continues with the main program. Clearly, there are several problems in implementing such an approach. First, how often a peripheral is polled is an important constraint. Some idea of the "frequency-of-service" required by each peripheral must be known and any software written for the system must accommodate this time dependence by "scheduling" when a device is polled. Second, there will obviously be times when a device is polled that is not ready for service, wasting the processor time that it took to do the poll. And other times, a ready device would have to wait until the processor "makes its rounds" before it could be serviced, slowing down the peripheral.

Other problems arise when certain peripherals are more important than others. The only way to implement the "priority" of devices is to poll the high priority devices more frequently than lower priority ones. It may even be necessary to poll the high priority devices while in a low priority device service routine. It is easy to see that the polled approach can be inefficient both time-wise and software-wise. Overall, the polled method of I/O servicing can have a detrimental effect on system throughput, thus limiting the tasks that can be performed by the processor.

A more desirable approach in most systems would allow the processor to be executing its main program and only stop to service the I/O when told to do so by the I/O itself. This is called the interrupt service method. In effect, the device would asynchronously signal the processor when it required service. The processor would finish its current instruction and then vector to the service routine for the device requesting service. Once the service routine is complete, the processor would resume exactly where it left off. Using the interrupt service method, no processor time is spent testing devices, scheduling is not needed, and priority schemes are readily implemented. It is easy to see that, using the interrupt service approach, system throughput would increase, allowing more tasks to be handled by the processor.

However, to implement the interrupt service method between processor and peripherals, additional hardware is usually required. This is because, after interrupting the processor, the device must supply information for vectoring program execution. Depending on the processor used, this can be accomplished by the device taking control of the data bus and "jamming" an instruction(s) onto it. The instruction(s) then vectors the program to the proper service routine. This of course requires additional control logic for each interrupt requesting device. Yet the implementation so far is only in the most basic form. What if certain peripherals are to be of higher priority than others? What if certain interrupts must be disabled while others are to be enabled? The possible variations go on, but they all add up to one theme; to provide greater flexibility using the interrupt service method, hardware requirements increase.

So, we're caught in the middle. The status poll method is a less desirable way of servicing I/O in terms of throughput, but its hardware requirements are minimal. On the other hand, the interrupt service method is most desirable in terms of flexibility and throughput, but additional hardware is required.

The perfect situation would be to have the flexibility and throughput of the interrupt method in an implementation with minimal hardware requirements. The 8259A Programmable Interrupt Controller (PIC) makes this all possible.

The 8259A Programmable Interrupt Controller (PIC) was designed to function as an overall manager of an interrupt driven system. No additional hardware is required. The 8259A alone can handle eight prioritized interrupt levels, controlling the complete interface between peripherals and processor. Additional 8259A's can be "cascaded" to increase the number of interrupt levels processed. A wide variety of modes and commands for programming the 8259A give it enough flexibility for almost any interrupt controlled structure. Thus, the 8259A is the feasible answer to handling I/O servicing in microcomputer systems.

Now, before explaining exactly how to use the 8259A, let's go over interrupt structures of the MCS-80, MCS-85, MCS-86, and MCS-88 systems, and how they interact with the 8259A. Figure 1 shows a block diagram of the 8259A interfacing with a standard system bus. This may prove useful as reference throughout the rest of the "Concepts" section.

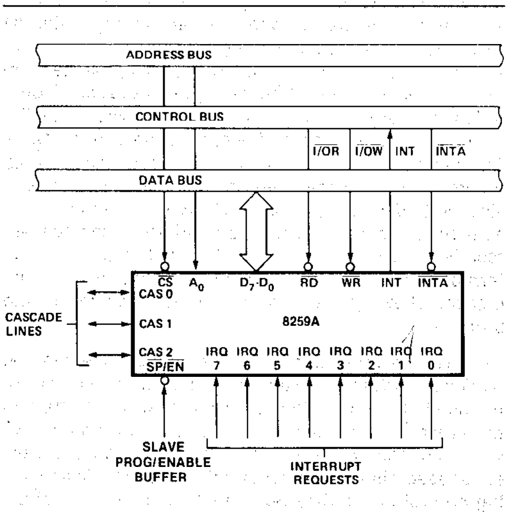


Figure 1. 8259A Interface to Standard System Bus

1.1 MCS-80™—8259A OVERVIEW

In an MCS-80—8259A interrupt configuration, as in Figure 2, a device may cause an interrupt by pulling one of the 8259A's interrupt request pins (IR0-IR7) high. If the 8259A accepts the interrupt request (this depends on its programmed condition), the 8259A's INT (interrupt) pin will go high, driving the 8080A's INT pin high.

The 8080A can receive an interrupt request any time, since its INT input is asynchronous. The 8080A, however, doesn't always have to acknowledge an interrupt request immediately. It can accept or disregard requests under software control using the EI (Enable Interrupt) or DI (Disable Interrupt) instructions. These instructions either set or reset an internal interrupt enable flip-flop. The output of this flip-flop controls the state of the INTE (Interrupt Enabled) pin. Upon reset, the 8080A interrupts are disabled, making INTE low.

At the end of each instruction cycle, the 8080A examines the state of its INT pin. If an interrupt request is present and interrupts are enabled, the 8080A enters an interrupt machine cycle. During the interrupt machine cycle the 8080A resets the internal interrupt enable flip-flop, disabling further interrupts until an EI instruction is executed. Unlike normal machine cycles, the interrupt machine cycle doesn't increment the program counter. This ensures that the 8080A can return to the pre-interrupt program location after the interrupt is completed. The 8080A then issues an $\overline{\text{INTA}}$ (Interrupt Acknowledge) pulse via the 8228 System Controller Bus Driver. This $\overline{\text{INTA}}$ pulse signals the 8259A that the 8080A is honoring the request and is ready to process the interrupt.

The 8259A can now vector program execution to the corresponding service routine. This is done during a sequence of the three $\overline{\text{INTA}}$ pulses from the 8080A via the 8228. Upon receiving the first $\overline{\text{INTA}}$ pulse the 8259A places the opcode for a CALL instruction on the data bus. This causes the contents of the program counter to be pushed onto the stack. In addition, the CALL instruction causes two more $\overline{\text{INTA}}$ pulses to be issued, allowing the 8259A to place onto the data bus the starting address of the corresponding service routine. This address is called the interrupt-vector address. The lower 8 bits (LSB) of the interrupt-vector address are released during the second $\overline{\text{INTA}}$ pulse and the upper 8 bits (MSB) during the third $\overline{\text{INTA}}$ pulse. Once this sequence is completed, program execution then vectors to the service routine at the interrupt-vector address.

If the same registers are used by both the main program and the interrupt service routine, their contents should be saved when entering the service routine. This includes the Program Status Word (PSW) which consists of the accumulator and flags. The best way to do this is to "PUSH" each register used onto the stack. The service routine can then "POP" each register off the stack in the reverse order when it is completed. This prevents any ambiguous operation when returning to the main program.

Once the service routine is completed, the main program may be re-entered by using a normal RET (Return) instruction. This will "POP" the original contents of the program counter back off the stack to resume program execution where it left off. Note, that because interrupts are disabled during the interrupt acknowledge sequence, the EI instruction must be executed either during the service routine or the main program before further interrupts can be processed.

For additional information on the 8080A interrupt structure and operation, refer to the MCS-80 User's Manual.

1.2 MCS-85™—8259A OVERVIEW

An MCS-85—8259A configuration processes interrupts in much the same format as an MCS-80—8259A configuration. When an interrupt occurs, a sequence of three $\overline{\text{INTA}}$ pulses causes the 8259A to release onto the data bus a CALL instruction and an interrupt-vector address for the corresponding service routine. Other events that occur during the 8080A interrupt machine cycle, such as disabling interrupts and not incrementing the program counter, also occur in the 8085A interrupt acknowledge machine cycle. Additionally, the instructions for saving registers, enabling or disabling of interrupts, and returning from service routines are literally the same.

The 8085A, however, has a different interrupt hardware scheme as shown in Figure 3. For one, the 8085A supplies its own $\overline{\text{INTA}}$ output pin rather than using an addi-

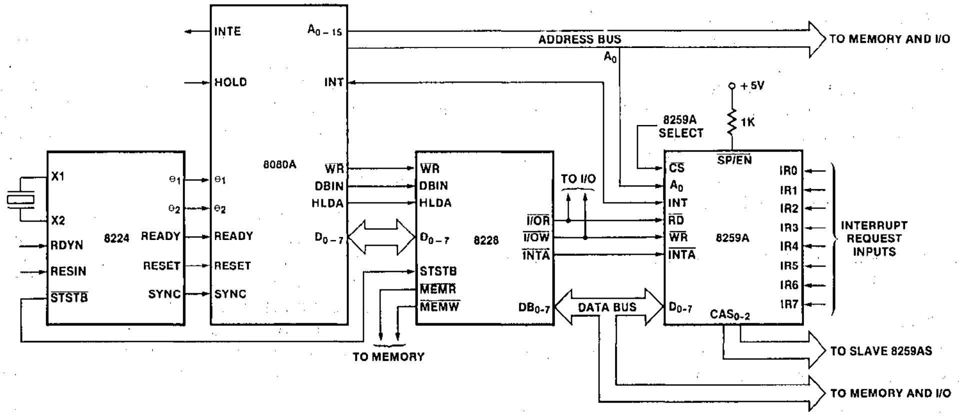


Figure 2. MCS-80 8259A Basic Configuration Example

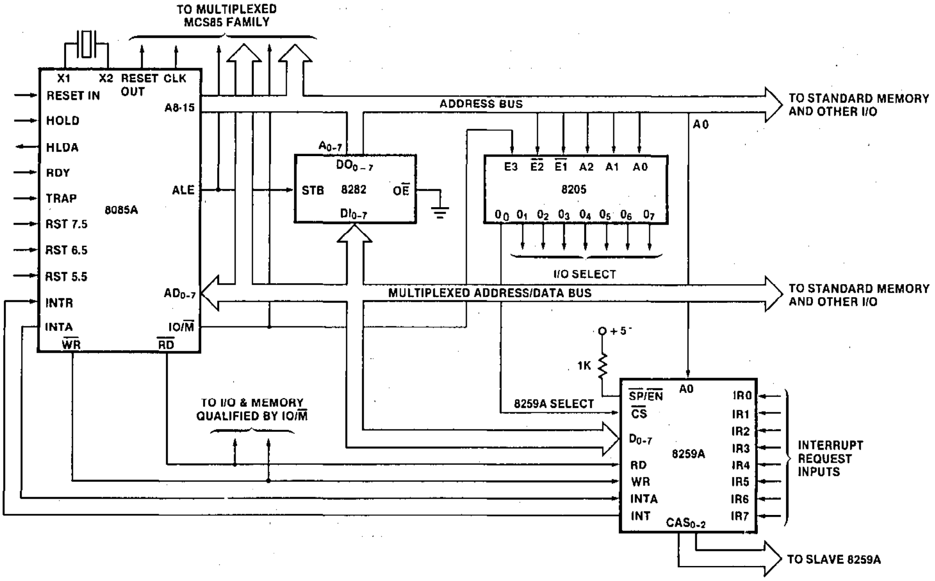


Figure 3. MCS-85™ 8259A Basic Configuration Example

tional chip, as the 8080A uses the 8228 System Controller Bus Driver. Another hardware difference is the 8085A has five hardware interrupt pins: INTR, RST 7.5, RST6.5, RST 5.5, and TRAP. The INTR (Interrupt Request) pin is the equivalent to the 8080A's INT pin. The RST (Restart) pins and TRAP pin are all restart interrupts which vector program execution to an individual dedicated address when asserted. The important factor associating these interrupts is their relative priority, as shown below:

TRAP	Highest Priority
RST 7.5	
RST 6.5	
RST 5.5	
INTR	Lowest Priority

The INTR pin has lowest priority among the other 8085A hardware interrupts. Thus, precautions to prevent interrupting 8259A service routines may be necessary. This, of course, depends on how the 8085A interrupts are being used in a particular application. Such precautions can be implemented, however, by masking the RST pins using the SIM instruction. The TRAP pin on the other hand is non-maskable; all interrupt pins but TRAP can be controlled by the EI (Enable Interrupt) and DI (Disable Interrupt) instructions.

For a complete description of the 8085A interrupt structure, refer to the MCS-85 User's Manual.

1.3 MCS-86/88™—8259A OVERVIEW

Operation of an MCS-86/88—8259A configuration has basic similarities of the MCS-80/85—8259A configurations. That is, a device can cause an interrupt by pulling one of the 8259A's interrupt request pins (IR0–IR7) high. If the 8259A honors the request, its INT pin will go high, driving the 8086/8088's INTR pin high. Like the 8080A and 8085A, the INTR pin of the 8086/8088 is asynchronous, thus it can receive an interrupt any time. The 8086/8088 can also accept or disregard requests on INTR under software control using the STI (Set Interrupt) or CLI (Clear Interrupt) instructions. These instructions set or clear the interrupt-enabled flag IF. Upon 8086/8088 reset the IF flag is cleared, disabling external interrupts on INTR. Beside the INTR pin, the 8086/8088 provides an NMI (Non-Maskable Interrupt) pin. The NMI functions similar to the 8085A's TRAP; it can't be disabled or masked. NMI has higher priority than INTR.

Figure 4 shows an MCS-86 MAX Mode system interfacing with an 8259A on the local bus. This MCS-86—8259A configuration is also representative of an MCS-88—8259A configuration except for the data bus which is 16 bits for 8086 and 8 bits for 8088. In the MCS-86 system the 8259A must be on the lower 8 bits of the data bus. Note that the 8259A could also be interfaced on the system bus.

Although there are some basic similarities, the actual processing of interrupts with an 8086/8088 is different than an 8080A or 8085A. When an interrupt request is present and interrupts are enabled, the 8086/8088 enters its interrupt acknowledge machine cycle. The interrupt acknowledge machine cycle pushes the flag registers onto the stack (as in a PUSHF instruction). It then clears the IF flag which disables interrupts. The contents of

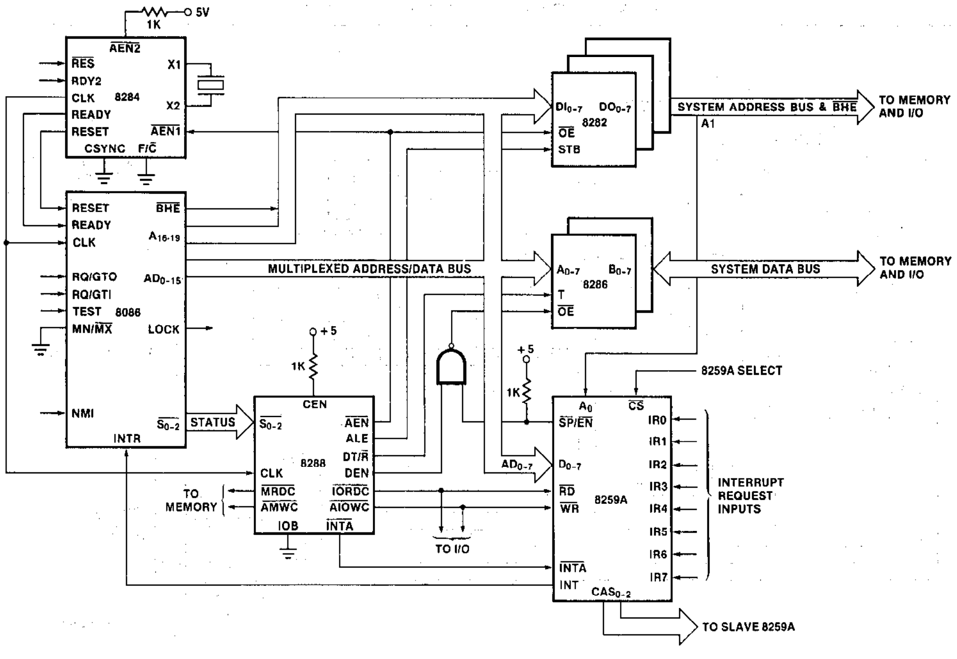


Figure 4. MSC-86™ 8259A Basic Configuration Example (8086 In Max. Mode)

both the code segment and the instruction pointer are then also pushed onto the stack. Thus, the stack retains the pre-interrupt flag status and pre-interrupt program location which are used to return from the service routine. The 8086/8088 then issues the first of two INTA pulses which signal the 8259A that the 8086/8088 has honored its interrupt request. If the 8086/8088 is used in its "MIN Mode" the INTA signal is available from the 8086/8088 on its INTA pin. If the 8086/8088 is used in the "MAX Mode" the INTA signal is available via the 8288 Bus Controller INTA pin. Additionally, in the "MAX Mode" the 8086/8088 LOCK pin goes low during the interrupt acknowledge sequence. The LOCK signal can be used to indicate to other system bus masters not to gain control of the system bus during the interrupt acknowledge sequence. A "HOLD" request won't be honored while LOCK is low.

The 8259A is now ready to vector program execution to the corresponding service routine. This is done during the sequence of the two INTA pulses issued by the 8086/8088. Unlike operation with the 8080A or 8085A, the 8259A doesn't place a CALL instruction and the starting address of the service routine on the data bus. Instead, the first INTA pulse is used only to signal the 8259A of the honored request. The second INTA pulse causes the 8259A to place a single interrupt-vector byte onto the data bus. Not used as a direct address, this interrupt-vector byte pertains to one of 256 interrupt "types" supported by the 8086/8088 memory. Program execution is vectored to the corresponding service routine by the contents of a specified interrupt type.

All 256 interrupt types are located in absolute memory locations 0 through 3FFH which make up the 8086/8088's interrupt-vector table. Each type in the interrupt-vector table requires 4 bytes of memory and stores a code segment address and an instruction pointer address. Figure 5 shows a block diagram of the interrupt-vector table. Locations 0 through 3FFH should be reserved for the interrupt-vector table alone. Furthermore, memory locations 00 through 7FH (types 0-31) are reserved for use by Intel Corporation for Intel hardware and software products. To maintain compatibility with present and future Intel products, these locations should not be used.

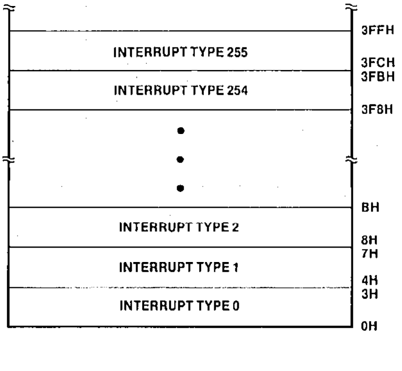


Figure 5. 8086/8088 Interrupt Vector Table

When the 8086/8088 receives an interrupt-vector byte from the 8259A, it multiplies its value by four to acquire the address of the interrupt type. For example, if the interrupt-vector byte specifies type 128 (80H), the vectored address in 8086/8088 memory is 4 x 80H, which equals 200H. Program execution is then vectored to the service routine whose address is specified by the code segment and instruction pointer values within type 128 located at 200H. To show how this is done, let's assume interrupt type 128 is to vector data to 8086/8088 memory location 2FF5FH. Figure 6 shows two possible ways to set values of the code segment and instruction pointer for vectoring to location 2FF5FH. Address generation by the code segment and instruction pointer is accomplished by an offset (they overlap). Of the total 20-bit address capability, the code segment can designate the upper 16 bits, the instruction pointer can designate the lower 16 bits.

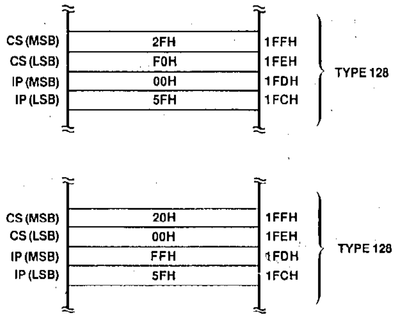


Figure 6. Two Examples of 8086/8088 Interrupt Type 128 Vectoring to Location 2FF5FH

When entering an interrupt service routine, those registers that are mutually used between the main program and service routine should be saved. The best way to do this is to "PUSH" each register used onto the stack immediately. The service routine can then "POP" each register off the stack in the same order when it is completed.

Once the service routine is completed the main program may be re-entered by using a IRET (Interrupt Return) instruction. The IRET instruction will pop the pre-interrupt instruction pointer, code segment and flags off the stack. Thus the main program will resume where it was interrupted with the same flag status regardless of changes in the service routine. Note especially that this includes the state of the IF flag, thus interrupts are re-enabled automatically when returning from the service routine.

Beside external interrupt generation from the INTR pin, the 8086/8088 is also able to invoke interrupts by software. Three interrupt instructions are provided: INT, INT (Type 3), and INTO. INT is a two byte instruction, the second byte selects the interrupt type. INT (Type 3) is a one byte instruction which selects interrupt Type 3. INTO is a conditional one byte interrupt instruction which selects interrupt Type 4 if the OF flag (trap on overflow) is set. All the software interrupts vector program execution as the hardware interrupts do.

For further information on 8086/8088 interrupt operation and internal interrupt structure refer to the MCS-86 User's Manual and the 8086 System Design application note.

2. 8259A FUNCTIONAL BLOCK DIAGRAM

A block diagram of the 8259A is shown in Figure 7. As can be seen from this figure, the 8259A consists of eight major blocks: the Interrupt Request Register (IRR), the In-Service Register (ISR), the Interrupt Mask Register (IMR), the Priority Resolver (PR), the cascade buffer/comparator, the data bus buffer, and logic blocks for control and read/write. We'll first go over the blocks directly related to interrupt handling, the IRR, ISR, IMR, PR, and the control logic. The remaining functional blocks are then discussed.

2.1 INTERRUPT REGISTERS AND CONTROL LOGIC

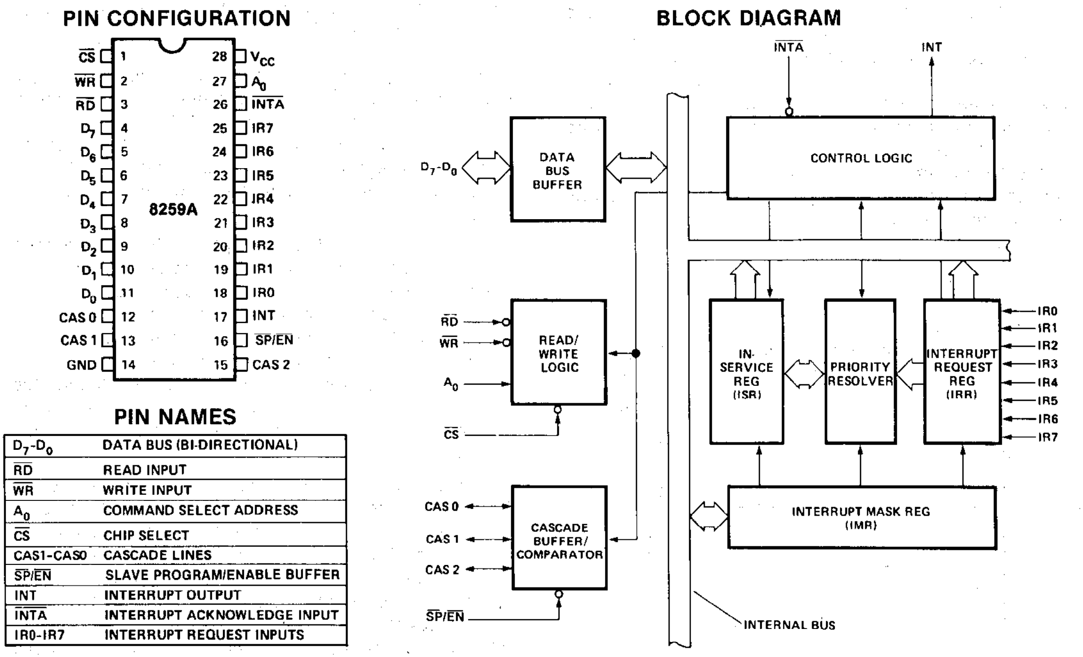
Basically, interrupt requests are handled by three "cascaded" registers: the Interrupt Request Register (IRR) is use to store all the interrupt levels requesting service; the In-Service Register (ISR) stores all the levels which are being serviced; and the Interrupt Mask Register (IMR) stores the bits of the interrupt lines to be masked. The Priority Resolver (PR) looks at the IRR, ISR and IMR, and determines whether an INT should be issued by the the control logic to the processor.

Figure 8 shows conceptually how the Interrupt Request (IR) input handles an interrupt request and how the various interrupt registers interact. The figure represents one of eight "daisy-chained" priority cells, one for each IR input.

The best way to explain the operation of the priority cell is to go through the sequence of internal events that happen when an interrupt request occurs. However, first, notice that the input circuitry of the priority cell allows for both level sensitive and edge sensitive IR inputs. Deciding which method to use is dependent on the particular application and will be discussed in more detail later.

When the IR input is in an inactive state (LOW), the edge sense latch is set. If edge sensitive triggering is selected, the "Q" output of the edge sense latch will arm the input gate to the request latch. This input gate will be disarmed after the IR input goes active (HIGH) and the interrupt request has been acknowledged. This disables the input from generating any further interrupts until it has returned low to re-arm the edge sense latch. If level sensitive triggering is selected, the "Q" output of the edge sense latch is rendered useless. This means the level of the IR input is in complete control of interrupt generation; the input won't be disarmed once acknowledged.

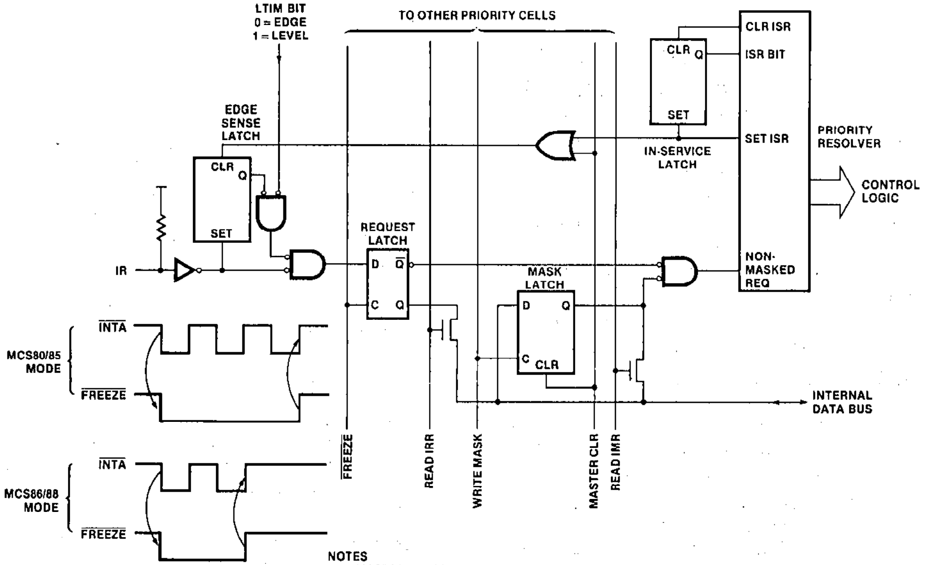
When an interrupt occurs on the IR input, it propagates through the request latch and to the PR (assuming the input isn't masked). The PR looks at the incoming requests and the currently in-service interrupts to ascertain whether an interrupt should be issued to the processor. Let's assume that the request is the only one incoming and no requests are presently in service. The PR then causes the control logic to pull the INT line to the processor high.



PIN NAMES

D_7-D_0	DATA BUS (BI-DIRECTIONAL)
RD	READ INPUT
WR	WRITE INPUT
A_0	COMMAND SELECT ADDRESS
CS	CHIP SELECT
CAS1-CAS0	CASCADE LINES
SP/EN	SLAVE PROGRAM/ENABLE BUFFER
INT	INTERRUPT OUTPUT
INTA	INTERRUPT ACKNOWLEDGE INPUT
IR0-IR7	INTERRUPT REQUEST INPUTS

Figure 7. 8259A Block Diagram and Pin Configuration



NOTES
1. MASTER CLEAR ACTIVE ONLY DURING ICW1
2. FREEZE/ IS ACTIVE DURING INTA/ AND POLL SEQUENCES ONLY
3. TRUTH TABLE FOR D-LATCH

C	D	Q	OPERATION
1	DI	DI	FOLLOW
0	X	Qn−1	HOLD

Figure 8. Priority Cell

When the processor honors the INT pulse, it sends a sequence of INTA pulses to the 8259A (three for 8080A/8085A, two for 8086/8088). During this sequence the state of the request latch is frozen (note the INTA-freeze request timing diagram). Priority is again resolved by the PR to determine the appropriate interrupt vectoring which is conveyed to the processor via the data bus.

Immediately after the interrupt acknowledge sequence, the PR sets the corresponding bit in the ISR which simultaneously clears the edge sense latch. If edge sensitive triggering is used, clearing the edge sense latch also disarms the request latch. This inhibits the possibility of a still active IR input from propagating through the priority cell. The IR input must return to an inactive state, setting the edge sense latch, before another interrupt request can be recognized. If level sensitive triggering is used, however, clearing the edge sense latch has no affect on the request latch. The state of the request latch is entirely dependent upon the IR input level. Another interrupt will be generated immediately if the IR level is left active after its ISR bit has been reset. An ISR bit gets reset with an End-of-Interrupt (EOI) command issued in the service routine. End-of-interrupts will be covered in more detail later.

2.2 OTHER FUNCTIONAL BLOCKS

Data Bus Buffer

This three-state, bidirectional 8-bit buffer is used to interface the 8259A to the processor system data bus (via DB0–DB7). Control words, status information, and interrupt-vector data are transferred through the data bus buffer.

Read/Write Control Logic

The function of this block is to control the programming of the 8259A by accepting OUTput commands from the processor. It also controls the releasing of status onto the data bus by accepting INput commands from the processor. The initialization and operation command word registers which store the various control formats are located in this block. The $\overline{RD}$, $\overline{WR}$, $\overline{A0}$, and $\overline{CS}$ pins are used to control access to this block by the processor.

Cascade Buffer/Comparator

As mentioned earlier, multiple 8259A's can be combined to expand the number of interrupt levels. A master-slave relationship of cascaded 8259A's is used for the expansion. The $\overline{SP}/\overline{EN}$ and the CAS0–2 pins are used for operation of this block. The cascading of 8259A's is covered in depth in the "Operation of the 8259A" section of this application note.

2.3 PIN FUNCTIONS

Name	Pin #	I/O	Function
V_{CC}	28	I	+5V supply
GND	14	I	Ground

Name	Pin #	I/O	Function
$\overline{CS}$	1	I	*Chip Select:* A low on this pin enables $\overline{RD}$ and $\overline{WR}$ communication between the CPU and the 8259A. $\overline{INTA}$ functions are independent of $\overline{CS}$.
$\overline{WR}$	2	I	*Write:* A low on this pin when $\overline{CS}$ is low enables the 8259A to accept command words from the CPU.
$\overline{RD}$	3	I	*Read:* A low on this pin when $\overline{CS}$ is low enables the 8259A to release status onto the data bus for the CPU.
D7-D0	4-11	I/O	*Bidirectional Data Bus:* Control, status and interrupt-vector information is transferred via this bus.
CAS0-CAS2	12,13, 15	I/O	*Cascade Lines:* The CAS lines form a private 8259A bus to control a multiple 8259A structure. These pins are outputs for a master 8259A and inputs for a slave 8259A.
$\overline{SP}/\overline{EN}$	16	I/O	*Slave Program/Enable Buffer:* This is a dual function pin. When in the buffered mode it can be used as an output to control buffer transceivers ($\overline{EN}$). When not in the buffered mode it is used as an input to designate a master ($\overline{SP}=1$) or slave ($\overline{SP}=0$).
INT	17	O	*Interrupt:* This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus it is connected to the CPU's interrupt pin.
IR0-IR7	18-25	I	*Interrupt Requests:* Asynchronous inputs. An interrupt request can be generated by raising an IR input (low to high) and holding it high until it is acknowledged (edge triggered mode), or just by a high level on an IR input (level triggered mode).
$\overline{INTA}$	26	I	*Interrupt Acknowledge:* This pin is used to enable 8259A interrupt-vector data onto the data bus. This is done by a sequence of interrupt acknowledge pulses issued by the CPU.
A0	27	I	*A0 Address Line:* This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 8259A to decipher between various command words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for 8086/8088).

3. OPERATION OF THE 8259A

Interrupt operation of the 8259A falls under five main categories: vectoring, priorities, triggering, status, and cascading. Each of these categories use various modes and commands. This section will explain the operation of these modes and commands. For clarity of explanation, however, the actual programming of the 8259A isn't covered in this section but in "Programming the 8259A". Appendix A is provided as a cross reference between these two sections.

3.1 INTERRUPT VECTORING

Each IR input of the 8259A has an individual interrupt-vector address in memory associated with it. Designation of each address depends upon the initial programming of the 8259A. As stated earlier, the interrupt sequence and addressing of an MCS-80 and MCS-85 system differs from that of an MCS-86 and MCS-88 system. Thus, the 8259A must be initially programmed in either a MCS-80/85 or MCS-86/88 mode of operation to insure the correct interrupt vectoring.

MCS-80/85™ Mode

When programmed in the MCS-80/85 mode, the 8259A should only be used within an 8080A or an 8085A system. In this mode the 8080A/8085A will handle interrupts in the format described in the "MCS-80—8259A or MCS-85—8259A Overviews."

Upon interrupt request in the MCS-80/85 mode, the 8259A will output to the data bus the opcode for a CALL instruction and the address of the desired routine. This is in response to a sequence of three $\overline{INTA}$ pulses issued by the 8080A/8085A after the 8259A has raised INT high.

The first INTA pulse to the 8259A enables the CALL opcode "CD_H" onto the data bus. It also resolves IR priorities and effects operation in the cascade mode, which will be covered later. Contents of the first interrupt-vector byte are shown in Figure 9A.

During the second and third $\overline{INTA}$ pulses, the 8259A conveys a 16-bit interrupt-vector address to the 8080A/8085A. The interrupt-vector addresses for all eight levels are selected when initially programming the 8259A. However, only one address is needed for programming. Interrupt-vector addresses of IR0-IR7 are automatically set at equally spaced intervals based on the one programmed address. Address intervals are user definable to 4 or 8 bytes apart. If the service routine for a device is short it may be possible to fit the entire routine within an 8-byte interval. Usually, though, the service routines require more than 8 bytes. So, a 4-byte interval is used to store a Jump (JMP) instruction which directs the 8080A/8085A to the appropriate routine. The 8-byte interval maintains compatibility with current 8080A/8085A Restart (RST) instruction software, while the 4-byte interval is best for a compact jump table. If the 4-byte interval is selected, then the 8259A will automatically insert bits A0-A4. This leaves A5-A15 to be programmed by the user. If the 8-byte interval is selected, the 8259A will automatically insert bits A0-A5. This leaves only A6-A15 to be programmed by the user.

The LSB of the interrupt-vector address is placed on the data bus during the second $\overline{INTA}$ pulse. Figure 9B shows the contents of the second interrupt-vector byte for both 4 and 8-byte intervals.

The MSB of the interrupt-vector address is placed on the data bus during the third $\overline{INTA}$ pulse. Contents of the third interrupt-vector byte is shown in Figure 9C.

	D7	D6	D5	D4	D3	D2	D1	D0
CALL CODE	1	1	0	0	1	1	0	1

A. FIRST INTERRUPT VECTOR BYTE, MCS80/85 MODE

IR	Interval = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

IR	Interval = 8							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

B. SECOND INTERRUPT VECTOR BYTE, MCS80/85 MODE

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

C. THIRD INTERRUPT VECTOR BYTE, MCS80/85 MODE

Figure 9. 9A-C. Interrupt-Vector Bytes for 8259A, MCS 80/85 Mode

MCS-86/88™ Mode

When programmed in the MCS-86/88 mode, the 8259A should only be used within an MCS-86 or MCS-88 system. In this mode, the 8086/8088 will handle interrupts in the format described earlier in the "8259A—8086/8088 Overview".

Upon interrupt in the MCS-86/88 mode, the 8259A will output a single interrupt-vector byte to the data bus. This is in response to only two INTA pulses issued by the 8086/8088 after the 8259A has raised INT high.

The first INTA pulse is used only for set-up purposes internal to the 8259A. As in the MCS-80/85 mode, this set-up includes priority resolution and cascade mode operations which will be covered later. Unlike the MCS-80/85 mode, no CALL opcode is placed on the data bus.

The second INTA pulse is used to enable the single interrupt-vector byte onto the data bus. The 8086/8088 uses this interrupt-vector byte to select one of 256 interrupt "types" in 8086/8088 memory. Interrupt type selection for all eight IR levels is made when initially programming the 8259A. However, reference to only one interrupt type is needed for programming. The upper 5 bits of the interrupt vector byte are user definable. The lower 3 bits are automatically inserted by the 8259A depending upon the IR level.

Contents of the interrupt-vector byte for 8086/8088 type selection is put on the data bus during the second INTA pulse and is shown in Figure 10.

IR	D7	D6	D5	D4	D3	D2	D1	D0
7	T7	T6	T5	T4	T3	1	1	1
6	T7	T6	T5	T4	T3	1	1	0
5	T7	T6	T5	T4	T3	1	0	1
4	T7	T6	T5	T4	T3	1	0	0
3	T7	T6	T5	T4	T3	0	1	1
2	T7	T6	T5	T4	T3	0	1	0
1	T7	T6	T5	T4	T3	0	0	1
0	T7	T6	T5	T4	T3	0	0	0

Figure 10. Interrupt Vector Byte, MCS 86/88™ Mode

3.2 INTERRUPT PRIORITIES

A variety of modes and commands are available for controlling interrupt priorities of the 8259A. All of them are programmable, that is, they may be changed dynamically under software control. With these modes and commands, many possibilities are conceivable, giving the user enough versatility for almost any interrupt controlled application.

Fully Nested Mode

The fully nested mode of operation is a general purpose priority mode. This mode supports a multilevel-interrupt structure in which priority order of all eight IR inputs are arranged from highest to lowest.

Unless otherwise programmed, the fully nested mode is entered by default upon initialization. At this time, IR0 is assigned the highest priority through IR7 the lowest. The fully nested mode, however, is not confined to this IR structure alone. Once past initialization, other IR inputs can be assigned highest priority also, keeping the multilevel-interrupt structure of the fully nested mode. Figure 11A-C shows some variations of the priority structures in the fully nested mode.

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	7	6	5	4	3	2	1	0

A

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	4	3	2	1	0	7	6	5

B

IR LEVELS	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
PRIORITY	1	0	7	6	5	4	3	2

C

Figure 11. A-C. Some Variations of Priority Structure in the Fully Nested Mode

Further explanation of the fully nested mode, in this section, is linked with information of general 8259A interrupt operations. This is done to ease explanation to the user in both areas.

In general, when an interrupt is acknowledged, the highest priority request is determined from the IRR (Interrupt Request Register). The interrupt vector is then placed on the data bus. In addition, the corresponding bit in the ISR (In-Service Register) is set to designate the routine in service. This ISR bit remains set until an EOI (End-Of-Interrupt) command is issued to the 8259A. EOI's will be explained in greater detail shortly.

In the fully nested mode, while an ISR bit is set, all further requests of the same or lower priority are inhibited from generating an interrupt to the microprocessor. A higher priority request, though, can generate an interrupt, thus vectoring program execution to its service routine. Interrupts are only acknowledged, however, if the microprocessor has previously executed an "Enable Interrupts" instruction. This is because the interrupt request pin on the microprocessor gets disabled automatically after acknowledgement of any interrupt. The assembly language instructions used to enable interrupts are "EI" for 8080A/8085A and "STI" for 8086/8088. Interrupts can be disabled by using the instruction "DI" for 8080A/ 8085A and "CLI" for 8086/8088. When a routine is completed a "return" instruction is executed, "RET" for 8080A/8085A and "IRET" for 8086/8088.

Figure 12 illustrates the correct usage of interrupt related instructions and the interaction of interrupt levels in the fully nested mode.

Assuming the IR priority assignment for the example in Figure 12 is IR0 the highest through IR7 the lowest, the sequence is as follows. During the main program, IR3 makes a request. Since interrupts are enabled, the microprocessor is vectored to the IR3 service routine. During the IR3 routine, IR1 asserts a request. Since IR1 has higher priority than IR3, an interrupt is generated. However, it is not acknowledged because the microprocessor disabled interrupts in response to the IR3 interrupt. The IR1 interrupt is not acknowledged until the "Enable Interrupts" instruction is executed. Thus the IR3 routine has a "protected" section of code over which no interrupts (except non-maskable) are allowed. The IR1 routine has no such "protected" section since an "Enable Interrupts" instruction is the first one in its service routine. Note that in this example the IR1 request must stay high until it is acknowledged. This is covered in more depth in the "Interrupt Triggering" section.

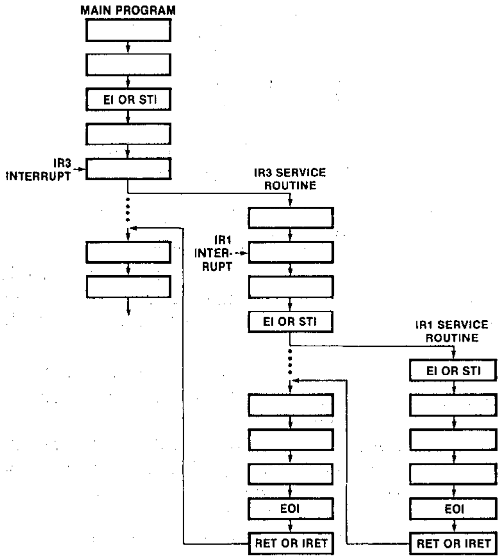


Figure 12. Fully Nested Mode Example (MCS 80/85™ or MCS 86/88™)

What is happening to the ISR register? While in the main program, no ISR bits are set since there aren't any interrupts in service. When the IR3 interrupt is acknowledged, the ISR3 bit is set. When the IR1 interrupt is acknowledged, both the ISR1 and the ISR3 bits are set, indicating that neither routine is complete. At this time, only IR0 could generate an interrupt since it is the only input with a higher priority than those previously in service. To terminate the IR1 routine, the routine must inform the 8259A that it is complete by resetting its ISR bit. It does this by executing an EOI command. A "return" instruction then transfers execution back to the IR3 routine. This allows IR0-IR2 to interrupt the IR3 routine again, since ISR3 is the highest ISR bit set. No further interrupts occur in the example so the EOI command resets ISR3 and the "return" instruction causes the main program to resume at its pre-interrupt location, ending the example.

A single 8259A is essentially always in the fully nested mode unless certain programming conditions disturb it. The following programming conditions can cause the 8259A to go out of the high to low priority structure of the fully nested mode.

- The automatic EOI mode
- The special mask mode
- A slave with a master not in the special fully nested mode

These modes will be covered in more detail later, however, they are mentioned now so the user can be aware of them. As long as these program conditions aren't inacted, the fully nested mode remains undisturbed.

End of Interrupt

Upon completion of an interrupt service routine the 8259A needs to be notified so its ISR can be updated. This is done to keep track of which interrupt levels are in the process of being serviced and their relative priorities. Three different End-Of-Interrupt (EOI) formats are available for the user. These are: the non-specific EOI command, the specific EOI command, and the automatic EOI Mode. Selection of which EOI to use is dependent upon the interrupt operations the user wishes to perform.

Non-Specific EOI Command

A non-specific EOI command sent from the microprocessor lets the 8259A know when a service routine has been completed, without specification of its exact interrupt level. The 8259A automatically determines the interrupt level and resets the correct bit in the ISR.

To take advantage of the non-specific EOI the 8259A must be in a mode of operation in which it can predetermine in-service routine levels. For this reason the non-specific EOI command should only be used when the most recent level acknowledged and serviced is always the highest priority level. When the 8259A receives a non-specific EOI command, it simply resets the highest priority ISR bit, thus confirming to the 8259A that the highest priority routine of the routines in service is finished.

The main advantage of using the non-specific EOI command is that IR level specification isn't necessary as in the "Specific EOI Command", covered shortly. However, special consideration should be taken when deciding to use the non-specific EOI. Here are two program conditions in which it is best not used:

- Using the set priority command within an interrupt service routine.
- Using a special mask mode.

These conditions are covered in more detail in their own sections, but are listed here for the users reference.

Specific EOI Command

A specific EOI command sent from the microprocessor lets the 8259A know when a service routine of a particular interrupt level is completed. Unlike a non-specific EOI command, which automatically resets the highest priority ISR bit, a specific EOI command specifies an exact ISR bit to be reset. One of the eight IR levels of the 8259A can be specified in the command.

The reason the specific EOI command is needed, is to reset the ISR bit of a completed service routine whenever the 8259A isn't able to automatically determine it. An example of this type of situation might be if the priorities of the interrupt levels were changed during an interrupt routine ("Specific Rotation"). In this case, if any other routines were in service at the same time, a non-specific EOI might reset the wrong ISR bit. Thus the specific EOI command is the best bet in this case, or for that matter, any time in which confusion of interrupt priorities may exist. The specific EOI command can be used in all conditions of 8259A operation, including those that prohibit non-specific EOI command usage.

Automatic EOI Mode

When programmed in the automatic EOI mode, the microprocessor no longer needs to issue a command to notify the 8259A it has completed an interrupt routine. The 8259A accomplishes this by performing a non-specific EOI automatically at the trailing edge of the last $\overline{\text{INTA}}$ pulse (third pulse in MCS-80/85, second in MCS-86).

The obvious advantage of the automatic EOI mode over the other EOI command is no command has to be issued. In general, this simplifies programming and lowers code requirements within interrupt routines.

However, special consideration should be taken when deciding to use the automatic EOI mode because it disturbs the fully nested mode. In the automatic EOI mode the ISR bit of a routine in service is reset right after it's acknowledged, thus leaving no designation in the ISR that a sevice routine is being executed. If any interrupt request occurs during this time (and interrupts are enabled) it will get serviced regardless of its priority, low or high. The problem of "over nesting" may also happen in this situation. "Over nesting" is when an IR input keeps interrupting its own routine, resulting in unnecessary stack pushes which could fill the stack in a worst case condition. This is not usually a desired form of operation!

So what good is the automatic EOI mode with problems like those just covered? Well, again, like the other EOIs, selection is dependent upon the application. If interrupts are controlled at a predetermined rate, so as not to cause the problems mentioned above, the automatic EOI mode works perfect just the way it is. However, if interrupts happen sporadically at an indeterminate rate, the automatic EOI mode should only be used under the following guideline:

- When using the automatic EOI mode with an indeterminate interrupt rate, the microprocessor should keep its interrupt request input disabled during execution of service routines.

By doing this, higher priority interrupt levels will be serviced only after the completion of a routine in service. This guideline restores the fully nested structure in regards to the IRR; however, a routine in-service can't be interrupted.

Automatic Rotation — Equal Priority

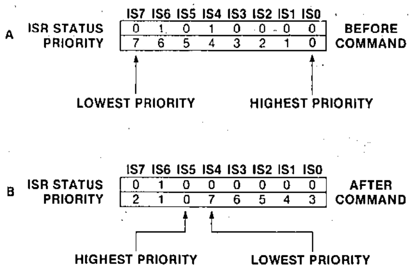
Automatic rotation of priorities serves in applications where the interrupting devices are of equal priority, such as communications channels. The concept is that once a peripheral is serviced, all other equal priority peripherals should be given a chance to be serviced before the original peripheral is serviced again. This is accomplished by automatically assigning a peripheral the lowest priority after being serviced Thus, in worst case, the device would have to wait until all other devices are serviced before being serviced again.

There are two methods of accomplishing automatic rotation. One is used in conjunction with the non-specific EOI, "rotate on non-specific EOI command". The other is used with the automatic EOI mode, "rotate in automatic EOI mode".

Rotate on Non-Specific EOI Command

When the rotate on non-specific EOI command is issued, the highest ISR bit is reset as in a normal non-specific EOI command. After it's reset though, the corresponding IR level is assigned lowest priority. Other IR priorities rotate to conform to the fully nested mode based on the newly assigned low priority

Figures 13A and B show how the rotate on non-specific EOI command effects the interrupt priorities. Let's assume the IR priorities were assigned with IR0 the highest and IR7 the lowest, as in 13A. IR6 and IR4 are already in service but neither is completed. Being the higher priority routine, IR4 is necessarily the routine being executed. During the IR4 routine a rotate on non-specific EOI command is executed. When this happens, bit 4 in the ISR is reset. IR4 then becomes the lowest priority and IR5 becomes the highest as in 13B.



	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
ISR STATUS	0	1	0	1	0	0	0	0
PRIORITY	7	6	5	4	3	2	1	0

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
ISR STATUS	0	1	0	0	0	0	0	0
PRIORITY	2	1	0	7	6	5	4	3

Figure 13. A-B. Rotate on Non-specific EOI Command Example

Rotate in Automatic EOI Mode

The rotate in automatic EOI mode works much like the rotate on non-specific EOI command. The main difference is that priority rotation is done automatically after the last $\overline{\text{INTA}}$ pulse of an interrupt request. To enter or exit this mode a rotate-in-automatic-EOI set command and rotate-in-automatic-EOI clear command is provided. After that, no commands are needed as with the normal automatic EOI mode. However, it must be remembered, when using any form of the automatic EOI mode, special consideration should be taken. Thus, the guideline for the automatic EOI mode also stands for the rotate in automatic EOI mode.

Specific Rotation — Specific Priority

Specific rotation gives the user versatile capabilities in interrupt controlled operations. It serves in those applications in which a specific device's interrupt priority must be altered. As opposed to automatic rotation which automatically sets priorities, specific rotation is completely user controlled. That is, the user selects which interrupt level is to receive lowest or highest priority. This can be done during the main program or within interrupt routines. Two specific rotation commands are available to the user, the "set priority command" and the "rotate on specific EOI command."

Set Priority Command

The set priority command allows the programmer to assign an IR level the lowest priority. All other interrupt levels will conform to the fully nested mode based on the newly assigned low priority.

An example of how the set priority command works is shown in Figures 14A and 14B. These figures show the status of the ISR and the relative priorities of the interrupt levels before and after the set priority command. Two interrupt routines are shown to be in service in Figure 14A. Since IR2 is the highest priority, it is necessarily the routine being executed. During the IR2 routine, priorities are altered so that IR5 is the highest. This is done simply by issuing the set priority command to the 8259A. In this case, the command specifies IR4 as being the lowest priority. The result of this set priority command is shown in Figure 14B. Even though IR7 now has higher priority than IR2, it won't be acknowledged until the IR2 routine is finished (via EOI). This is because priorities are only resolved upon an interrupt request or an interrupt acknowledge sequence. If a higher priority request occurs during the IR2 routine, then priorities are resolved and the highest will be acknowledged.

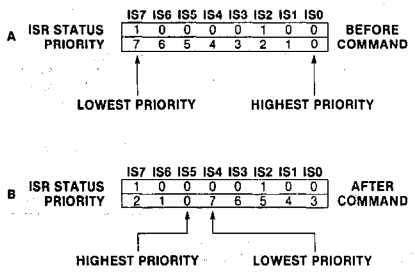


Figure 14. A-B. Set Priority Command Example

When completing a service routine in which the set priority command is used, the correct EOI must be issued. The non-specific EOI command shouldn't be used in the same routine as a set priority command. This is because the non-specific EOI command resets the highest ISR bit, which, when using the set priority command, is not always the most recent routine in service. The automatic EOI mode, on the other hand, can be used with the set priority command. This is because it automatically performs a non-specific EOI before the set priority command can be issued. The specific EOI command is the best bet in most cases when using the set priority command within a routine. By resetting the specific ISR bit of a routine being completed, confusion is eliminated.

Rotate on Specific EOI Command

The rotate on specific EOI command is literally a combination of the set priority command and the specific EOI command. Like the set priority command, a specified IR level is assigned lowest priority. Like the specific EOI command, a specified level will be reset in the ISR. Thus the rotate on specific EOI command accomplishes both tasks in only one command.

If it is not necessary to change IR priorities prior to the end of an interrupt routine, then this command is advantageous. For an EOI command must be executed anyway (unless in the automatic EOI mode), so why not do both at the same time?

Interrupt Masking

Disabling or enabling interrupts can be done by other means than just controlling the microprocessor's interrupt request pin. The 8259A has an IMR (Interrupt Mask Register) which enhances interrupt control capabilities. Rather than all interrupts being disabled or enabled at the same time, the IMR allows individual IR masking. The IMR is an 8-bit register, bits 0-7 directly correspond to IR0-IR7. Any IR input can be masked by writing to the IMR and setting the appropriate bit. Likewise, any IR input can be enabled by clearing the correct IMR bit.

There are various uses for masking off individual IR inputs. One example is when a portion of a main routine wishes only to be interrupted by specific interrupts. Another might be disabling higher priority interrupts for a portion of a lower priority service routine. The possibilities are many.

When an interrupt occurs while its IMR bit is set, it isn't necessarily forgotten. For, as stated earlier, the IMR acts only on the output of the IRR. Even with an IR input masked it is still possible to set the IRR. Thus, when resetting an IMR, if its IRR bit is set it will then generate an interrupt. This is providing, of course, that other priority factors are taken into consideration and the IR request remains active. If the IR request is removed before the IMR is reset, no interrupt will be acknowledged.

Special Mask Mode

In various cases, it may be desirable to enable interrupts of a lower priority than the routine in service. Or, in other words, allow lower priority devices to generate interrupts. However, in the fully nested mode, all IR levels of priority below the routine in service are inhibited. So what can be done to enable them?

Well, one method could be using an EOI command before the actual completion of a routine in service. But beware, doing this may cause an "over nesting" problem, similar to in the automatic EOI mode. In addition, resetting an ISR bit is irreversible by software control, so lower priority IR levels could only be later disabled by setting the IMR.

A much better solution is the special mask mode. Working in conjunction with the IMR, the special mask mode enables interrupts from all levels except the level in service. This is done by masking the level that is in service and then issuing the special mask mode command. Once the special mask mode is set, it remains in effect until reset.

Figure 15 shows how to enable lower priority interrupts by using the Special Mask Mode (SMM). Assume that IR0 has highest priority when the main program is interrupted by IR4. In the IR4 service routine an enable interrupt instruction is executed. This only allows higher priority interrupt requests to interrupt IR4 in the normal fully nested mode. Further in the IR4 routine, bit 4 of the IMR is masked and the special mask mode is entered. Priority operation is no longer in the fully nested mode. All interrupt levels are enabled except for IR4. To leave the special mask mode, the sequence is executed in reverse.

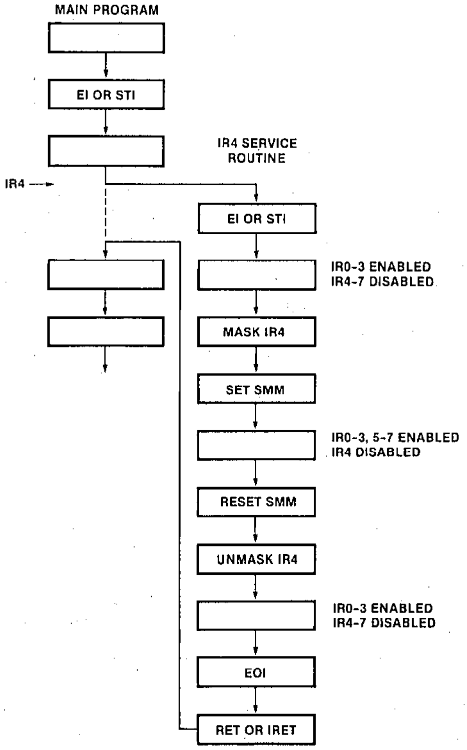


Figure 15. Special Mask Mode Example (MCS 80/85™ or MCS 86/88™)

Precautions must be taken when exiting an interrupt service routine which has used the special mask mode. A non-specific EOI command can't be used when in the special mask mode. This is because a non-specific won't clear an ISR bit of an interrupt which is masked when in the special mask mode. In fact, the bit will appear invisible. If the special mask mode is cleared before an EOI command is issued a non-specific EOI command can be used. This could be the case in the example shown in Figure 15, but, to avoid any confusion it's best to use the specific EOI whenever using the special mask mode.

It must be remembered that the special mask mode applies to all masked levels when set. Take, for instance, IR1 interrupting IR4 in the previous example. If this happened while in the special mask mode, and the IR1 routine masked itself, all interrupts would be enabled except IR1 and IR4 which are masked.

3.3 INTERRUPT TRIGGERING

There are two classical ways of sensing an active interrupt request: a level sensitive input or an edge sensitive input. The 8259A gives the user the capability for either method with the edge triggered mode and the level triggered mode. Selection of one of these interrupt triggering methods is done during the programmed initialization of the 8259A.

Level Triggered Mode

When in the level triggered mode the 8259A will recognize any active (high) level on an IR input as an interrupt request. If the IR input remains active after an EOI command has been issued (resetting its ISR bit), another interrupt will be generated. This is providing of course, the processor INT pin is enabled. Unless repetitious interrupt generation is desired, the IR input must be brought to an inactive state before an EOI command is issued in its service routine. However, it must not go inactive so soon that it disobeys the necessary timing requirements shown in Figure 16. Note that the request on the IR input must remain until after the falling edge of the first $\overline{\text{INTA}}$ pulse. If on any IR input, the request goes inactive before the first $\overline{\text{INTA}}$ pulse, the 8259A will respond as if IR7 was active. In any design in which there's a possibility of this happening, the IR7 default feature can be used as a safeguard. This can be accomplished by using the IR7 routine as a "clean-up routine" which might recheck the 8259A status or merely return program execution to its pre-interrupt location.

Depending upon the particular design and application, the level triggered mode has a number of uses. For one, it provides for repetitious interrupt generation. This is useful in cases when a service routine needs to be continually executed until the interrupt request goes inactive. Another possible advantage of the level triggered mode is it allows for "wire-OR'ed" interrupt requests. That is, a number of interrupt requests using the same IR input. This can't be done in the edge triggered mode, for if a device makes an interrupt request while the IR input is high (from another request), its transition will be "shadowed". Thus the 8259A won't recognize further interrupt requests because its IR input is already high. Note that when a "wire-OR'ed" scheme is used, the ac-

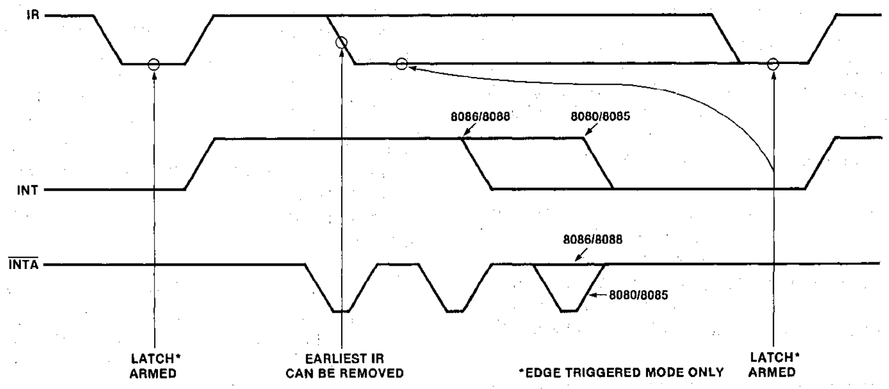


Figure 16. IR Triggering Timing Requirements

tual requesting device has to be determined by the software in the service routine.

Caution should be taken when using the automatic EOI mode and the level triggered mode together. Since in the automatic EOI mode an EOI is automatically performed at the end of the interrupt acknowledge sequence, if the processor enables interrupts while an IR input is still high, an interrupt will occur immediately. To avoid this situation interrupts should be kept disabled until the end of the service routine or until the IR input returns low.

Edge Triggered Mode

When in the edge triggered mode, the 8259A will only recognize interrupts if generated by an inactive (low) to active (high) transition on an IR input. The edge triggered mode incorporates an edge lockout method of operation. This means that after the rising edge of an interrupt request and the acknowledgement of the request, the positive level of the IR input won't generate further interrupts on this level. The user needn't worry about quickly removing the request after acknowledgement in fear of generating further interrupts as might be the case in the level triggered mode. Before another interrupt can be generated the IR input must return to the inactive state.

Referring back to Figure 16, the timing requirements for interrupt triggering is shown. Like the level triggered mode, in the edge triggered mode the request on the IR input must remain active until after the falling edge of the first $\overline{\text{INTA}}$ pulse for that particular interrupt. Unlike the level triggered mode, though, after the interrupt request is acknowledged its IRR latch is disarmed. Only after the IR input goes inactive will the IRR latch again become armed, making it ready to receive another interrupt request (in the level triggered mode, the IRR latch is always armed). Because of the way the edge triggered mode functions, it is best to use a positive level with a negative pulse to trigger the IR requests. With this type of input, the trailing edge of the pulse causes the interrupt and the maintained positive level meets the necessary timing requirements (remaining high until after the interrupt acknowledge occurs). Note that the IR7 default feature mentioned in the "level triggered mode" section also works for the edge triggered mode.

Depending upon the particular design and application, the edge triggered mode has various uses. Because of its edge lockout operation, it is best used in those applications where repetitious interrupt generation isn't desired. It is also very useful in systems where the interrupt request is a pulse (this should be in the form of a negative pulse to the 8259A). Another possible advantage is that it can be used with the automatic EOI mode without the cautions in the level triggered mode. Overall, in most cases, the edge triggered mode simplifies operation for the user, since the duration of the interrupt request at a positive level is not usually a factor.

3.4 INTERRUPT STATUS

By means of software control, the user can interrogate the status of the 8259A. This allows the reading of the internal interrupt registers, which may prove useful for interrupt control during service routines. It also provides for a modified status poll method of device monitoring, by using the poll command. This makes the status of the internal IR inputs available to the user via software control. The poll command offers an alternative to the interrupt vector method, especially for those cases when more than 64 interrupts are needed.

Reading Interrupt Registers

The contents of each 8-bit interrupt register, IRR, ISR, and IMR, can be read to update the user's program on the present status of the 8259A. This can be a versatile tool in the decision making process of a service routine, giving the user more control over interrupt operations. Before delving into the actual process of reading the registers, let's briefly review their general descriptions:

IRR (Interrupt Request Register)	Specifies all interrupt levels requesting service.
ISR (In-Service Register)	Specifies all interrupt levels which are being serviced.
IMR (Interrupt Mask Register)	Specifies all interrupt levels that are masked.

To read the contents of the IRR or ISR, the user must first issue the appropriate read register command (read IRR or read ISR) to the 8259A. Then by applying a $\overline{\text{RD}}$ pulse to the 8259A (an INput instruction), the contents of the desired register can be acquired. There is no need to issue a read register command every time the IRR or ISR is to be read. Once a read register command is received by the 8259A, it "remembers" which register has been selected. Thus, all that is necessary to read the contents of the same register more than once is the $\overline{\text{RD}}$ pulse and the correct addressing (A0 = 0, explained in "Programming the 8259A"). Upon initialization, the selection of registers defaults to the IRR. Some caution should be taken when using the read register command in a system that supports several levels of interrupts. If the higher priority routine causes an interrupt between the read register command and the actual input of the register contents, there's no guarantee that the same register will be selected when it returns. Thus it is best in such cases to disable interrupts during the operation.

Reading the contents of the IMR is different than reading the IRR or ISR. A read register command is not necessary when reading the IMR. This is because the IMR can be addressed directly for both reading and writing. Thus all that the 8259A requires for reading the IMR is a $\overline{\text{RD}}$ pulse and the correct addressing (A0 = 1, explained in "Programming the 8259A").

Poll Command

As mentioned towards the beginning of this application note, there are two methods of servicing peripherals: status polling and interrupt servicing. For most applications the interrupt service method is best. This is because it requires the least amount of CPU time, thus increasing system throughput. However, for certain applications, the status poll method may be desirable.

For this reason, the 8259A supports polling operations with the poll command. As opposed to the conventional method of polling, the poll command offers improved device servicing and increased throughput. Rather than having the processor poll each peripheral in order to find the actual device requiring service, the processor polls the 8259A. This allows the use of all the previously mentioned priority modes and commands. Additionally, both polled and interrupt methods can be used within the same program.

To use the poll command the processor must first have its interrupt request pin disabled. Once the poll command is issued, the 8259A will treat the next ($\overline{\text{CS}}$ qualified) $\overline{\text{RD}}$ pulse issued to it (an INput instruction) as an interrupt acknowledge. It will then set the appropriate bit in the ISR, if there was an interrupt request, and enable a special word onto the data bus. This word shows whether an interrupt request has occurred and the highest priority level requesting service. Figure 17 shows the contents of the "poll word" which is read by the processor. Bits W0-W2 convey the binary code of the highest priority level requesting service. Bit I designates whether or not an interrupt request is present. If an interrupt request is present, bit I will equal 1. If there isn't an interrupt request at all, bit I will equal 0 and bits W0-W2 will be set to ones. Service to the requesting device is achieved by software decoding the poll word and branching to the appropriate service routine. Each time the 8259A is to be polled, the poll command must be written before reading the poll word.

The poll command is useful in various situations. For instance, it's a good alternative when memory is very limited, because an interrupt-vector table isn't needed. Another use for the poll command is when more than 64 interrupt levels are needed (64 is the limit when cascading 8259's). The only limit of interrupts using the poll command is the number of 8259's that can be addressed in a particular system. Still another application of the poll command might be when the INT or $\overline{\text{INTA}}$ signals are not available. This might be the case in a large system where a processor on one card needs to use an 8259A on a different card. In this instance, the poll command is the only way to monitor the interrupt devices and still take advantage of the 8259A's prioritizing features. For those cases when the 8259A is using the poll command only and not the interrupt method, each 8259A must receive an initialization sequence (interrupt vector). This must be done even though the interrupt vector features of the 8259A are not used. In this case, the interrupt vector specified in the initialization sequence could be a "fake".

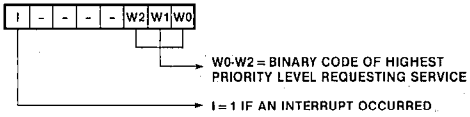

I
-
-
-
-
W2
W1
W0
W0-W2 = BINARY CODE OF HIGHEST PRIORITY LEVEL REQUESTING SERVICE
I = 1 IF AN INTERRUPT OCCURRED

Figure 17. Poll Word

3.5 INTERRUPT CASCADING

As mentioned earlier, more than one 8259A can be used to expand the priority interrupt scheme to up to 64 levels without additional hardware. This method for expanded interrupt capability is called "cascading". The 8259A supports cascading operations with the cascade mode. Additionally, the special fully nested mode and the buffered mode are available for increased flexibility when cascading 8259A's in certain applications.

Cascade Mode

When programmed in the cascade mode, basic operation consists of one 8259A acting as a master to the others which are serving as slaves. Figure 18 shows a system containing a master and two slaves, providing a total of 22 interrupt levels.

A specific hardware set-up is required to establish operation in the cascade mode. With Figure 18 as a reference, note that the master is designated by a high on the $\overline{\text{SP/EN}}$ pin, while the $\overline{\text{SP/EN}}$ pins of the slaves are grounded (this can also be done by software, see buffered mode). Additionally, the INT output pin of each slave is connected to an IR input pin of the master. The CAS0-2 pins for all 8259A's are paralleled. These pins act as outputs when the 8259A is a master and as inputs for the slaves. Serving as a private 8259A bus, they control which slave has control of the system bus for interrupt vectoring operation with the processor. All other pins are connected as in normal operation (each 8259A receives an INTA pulse).

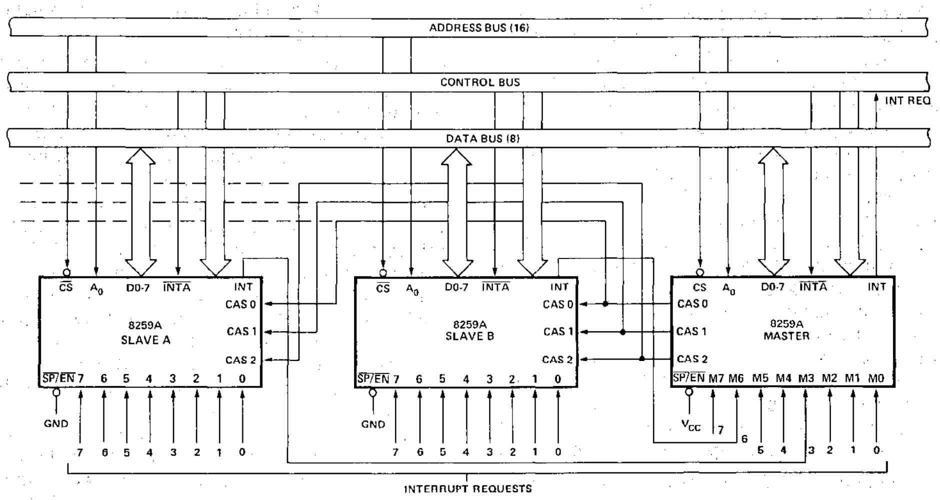


Figure 18. Cascaded 8259A'S 22 Interrupt Levels

Besides hardware set-up requirements, all 8259A's must be software programmed to work in the cascade mode. Programming the cascade mode is done during the initialization of each 8259A. The 8259A that is selected as master must receive specification during its initialization as to which of its IR inputs are connected to a slave's INT pin. Each slave 8259A, on the other hand, must be designated during its initialization with an ID (0 through 7) corresponding to which of the master's IR inputs its INT pin is connected to. This is all necessary so the CAS0–2 pins of the masters will be able to address each individual slave. Note that as in normal operation, each 8259A must also be initialized to give its IR inputs a unique interrupt vector. More detail on the necessary programming of the cascade mode is explained in "Programming the 8259A".

Now, with background information on both hardware and software for the cascade mode, let's go over the sequence of events that occur during a valid interrupt request from a slave. Suppose a slave IR input has received an interrupt request. Assuming this request is higher priority than other requests and in-service levels on the slave, the slave's INT pin is driven high. This signals the master of the request by causing an interrupt request on a designated IR pin of the master. Again, assuming that this request to the master is higher priority than other master requests and in-service levels (possibly from other slaves), the master's INT pin is pulled high, interrupting the processor.

The interrupt acknowledge sequence appears to the processor the same as the non-cascading interrupt acknowledge sequence; however, it's different among the 8259A's. The first $\overline{\text{INTA}}$ pulse is used by all the 8259A's for internal set-up purposes and, if in the 8080/8085 mode, the master will place the CALL opcode on the data bus. The first $\overline{\text{INTA}}$ pulse also signals the master to place the requesting slave's ID code on the CAS lines. This turns control over to the slave for the rest of the interrupt acknowledge sequence, placing the appropriate pre-programmed interrupt vector on the data bus, completing the interrupt request.

During the interrupt acknowledge sequence, the corresponding ISR bit of both the master and the slave get set. This means two EOI commands must be issued (if not in the automatic EOI mode), one for the master and one for the slave.

Special consideration should be taken when mixed interrupt requests are assigned to a master 8259A; that is, when some of the master's IR inputs are used for slave interrupt requests and some are used for individual interrupt requests. In this type of structure, the master's IR0 must not be used for a slave. This is because when an IR input that isn't initialized as a slave receives an interrupt request, the CAS0–2 lines won't be activated, thus staying in the default condition addressing for IR0 (slave IR0). If a slave is connected to the master's IR0 when a non-slave interrupt occurs on another master IR input, erroneous conditions may result. Thus IR0 should be the last choice when assigning slaves to IR inputs.

Special Fully Nested Mode

Depending on the application, changes in the nested structure of the cascade mode may be desired. This is because the nested structure of a slave 8259A differs from that of the normal fully nested mode. In the cascade mode, if a slave receives a higher priority interrupt request than one which is in service (through the same slave), it won't be recognized by the master. This is because the master's ISR bit is set, ignoring all requests of equal or lower priority. Thus, in this case, the higher priority slave interrupt won't be serviced until after the master's ISR bit is reset by an EOI command. This is most likely after the completion of the lower priority routine.

If the user wishes to have a truly fully nested structure within a slave 8259A, the special fully nested mode should be used. The special fully nested mode is programmed in the master only. This is done during the master's initialization. In this mode the master will ignore only those interrupt requests of lower priority than the set ISR bit and will respond to all requests of equal or higher priority. Thus if a slave receives a higher priority request than one in service, it will be recognized. To insure proper interrupt operation when using the special fully nested mode, the software must determine if any other slave interrupts are still in service before issuing an EOI command to the master. This is done by resetting the appropriate slave ISR bit with an EOI and then reading its ISR. If the ISR contains all zeros, there aren't any other interrupts from the slave in service and an EOI command can be sent to the master. If the ISR isn't all zeros, an EOI command shouldn't be sent to the master. Clearing the master's ISR bit with an EOI command while there are still slave interrupts in service would allow lower priority interrupts to be recognized at the master. An example of this process is shown in the second application in the "Applications Examples" section.

Buffered Mode

The buffered mode is useful in large systems where buffering is required on the data bus. Although not limited to only 8259A cascading, it's most pertinent in this use. In the buffered mode, whenever the 8259A's data bus output is enabled, its SP/$\overline{\text{EN}}$ pin will go low. This signal can be used to enable data transfer through a buffer transceiver in the required direction.

Figure 19 shows a conceptual diagram of three 8259A's in cascade, each slave is controlling an individual 8286 8-bit bidirectional bus driver by means of the buffered mode. Note the pull-up on the SP/$\overline{\text{EN}}$. It is used to enable data transfer to the 8259A for its initial programming. When data transfer is to go from the 8259A to the processor, SP/$\overline{\text{EN}}$ will go low; otherwise, it will be high.

A question should arise, however, from the fact that the SP/$\overline{\text{EN}}$ pin is used to designate a master from a slave; how can it be used for both master-slave selection and buffer control? The answer to this is the provision for software programmable master-slave selection when in the buffer mode. The buffered mode is selected during each 8259A's initialization. At the same time, the user can assign each individual 8259A as a master or slave (see "Programming the 8259A").

4. PROGRAMMING THE 8259A

Programming the 8259A is accomplished by using two types of command words: Initialization Command Words (ICWs) and Operational Command Words (OCWs). All the modes and commands explained in the previous section, "Operation of the 8259A", are programmable using the ICWs and OCWs (see Appendix A for cross reference). The ICWs are issued from the processor in a sequential format and are used to set-up the 8259A in an initial state of operation. The OCWs are issued as needed to vary and control 8259A operation.

Both ICWs and OCWs are sent by the processor to the 8259A via the data bus (8259A $\overline{\text{CS}}=0$, $\overline{\text{WR}}=0$). The 8259A distinguishes between the different ICWs and OCWs by the state of its A0 pin (controlled by processor addressing), the sequence they're issued in (ICWs only), and some dedicated bits among the ICWs and OCWs. Those bits which are dedicated are indicated so by fixed values (0 or 1) in the corresponding ICW or OCW programming formats which are covered shortly. Note, when issuing either ICWs or OCWs, the interrupt request pin of the processor should be disabled.

4.1 INITIALIZATION COMMAND WORDS (ICWs)

Before normal operation can begin, each 8259A in a system must be initialized by a sequence of two to four programming bytes called ICWs (Initialization Command Words). The ICWs are used to set-up the necessary conditions and modes for proper 8259A operation.

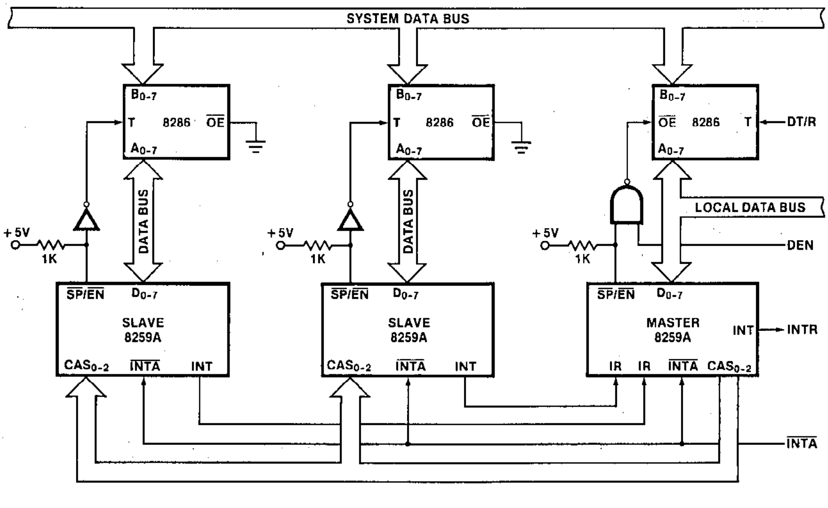


Figure 19. Cascade-Buffered Mode Example

Figure 20 shows the initialization flow of the 8259A. Both ICW1 and ICW2 must be issued for any form of 8259A operation. However, ICW3 and ICW4 are used only if designated so in ICW1. Determining the necessity and use of each ICW is covered shortly in individual groupings. Note that, once intialized, if any programming changes within the ICWs are to be made, the entire ICW sequence must be reprogrammed, not just an individual ICW.

Certain internal set-up conditions occur automatically within the 8259A after the first ICW has been issued. These are:

A. Sequencer logic is set to accept the remain'ng ICWs as designated in ICW1.

B. The ISR (In-Service Register) and IMR (Interrupt Mask Register) are both cleared.

C. The special mask mode is reset.

D. The rotate in automatic EOI mode flip-flop is cleared.

E. The IRR (Interrupt Request Register) is selected for the read register command.

F. If the IC4 bit equals 0 in ICW1, all functions in ICW4 are cleared; 8080/8085 mode is selected by default.

G. The fully nested mode is entered with an initial priority assignment of IR0 highest through IR7 lowest.

H. The edge sense latch of each IR priority cell is cleared, thus requiring a low to high transition to generate an interrupt (edge triggered mode effected only).

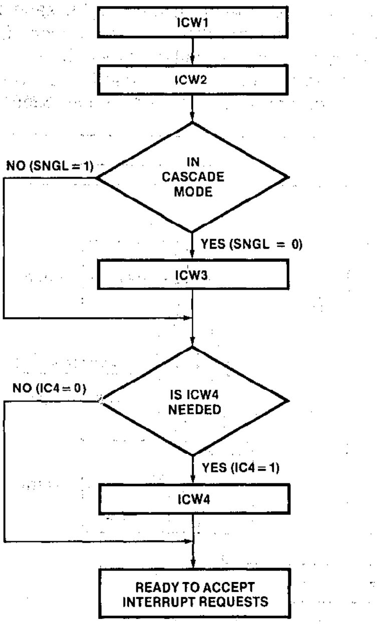
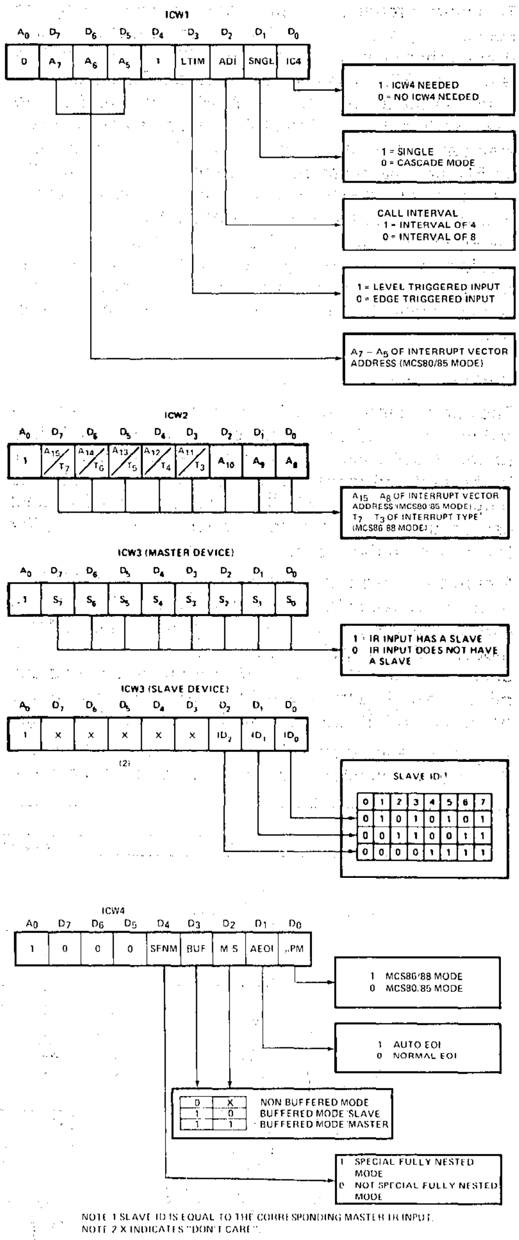


Figure 20. Initialization Flow

The ICW programming format, Figure 21, shows bit designation and a short definition of each ICW. With the ICW format as reference, the functions of each ICW will now be explained individually.



ICW1
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 A7 A6 A5 1 LTIM ADI SNGL IC4
1 = ICW4 NEEDED
0 = NO ICW4 NEEDED
1 = SINGLE
0 = CASCADE MODE
CALL INTERVAL
1 = INTERVAL OF 4
0 = INTERVAL OF 8
1 = LEVEL TRIGGERED INPUT
0 = EDGE TRIGGERED INPUT
A7 – A5 OF INTERRUPT VECTOR ADDRESS (MCS80/85 MODE)
ICW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 A15/T7 A14/T6 A13/T5 A12/T4 A11/T3 A10 A9 A8
A15 A8 OF INTERRUPT VECTOR ADDRESS (MCS80 85 MODE)
T7 T3 OF INTERRUPT TYPE (MCS86 88 MODE)
ICW3 (MASTER DEVICE)
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 S7 S6 S5 S4 S3 S2 S1 S0
1 IR INPUT HAS A SLAVE
0 IR INPUT DOES NOT HAVE A SLAVE
ICW3 (SLAVE DEVICE)
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 X X X X X ID2 ID1 ID0
(2)
SLAVE ID (1)
0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1
ICW4
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 0 0 0 SFNM BUF M S AEOI μPM
1 MCS86/88 MODE
0 MCS80/85 MODE
1 AUTO EOI
0 NORMAL EOI
0 X NON BUFFERED MODE
1 0 BUFFERED MODE SLAVE
1 1 BUFFERED MODE MASTER
1 SPECIAL FULLY NESTED MODE
0 NOT SPECIAL FULLY NESTED MODE

NOTE 1 SLAVE ID IS EQUAL TO THE CORRESPONDING MASTER IR INPUT.
NOTE 2 X INDICATES "DON'T CARE".

SOME OF THE TERMINOLOGY USED MAY DIFFER SLIGHTLY FROM EXISTING 8259A DATA SHEETS. THIS IS DONE TO BETTER CLARIFY AND EXPLAIN THE PROGRAMMING OF THE 8259A, THE OPERATIONAL RESULTS REMAIN THE SAME.

Figure 21. Initialization Command Words (ICWS) Programming Format

ICW1 and ICW2

Issuing ICW1 and ICW2 is the minimum amount of programming needed for any type of 8259A operation. The majority of bits within these two ICWs are used to designate the interrupt vector starting address. The remaining bits serve various purposes. Description of the ICW1 and ICW2 bits is as follows:

IC4: The IC4 bit is used to designate to the 8259A whether or not ICW4 will be issued. If any of the ICW4 operations are to be used, ICW4 must equal 1. If they aren't used, then ICW4 needn't be issued and IC4 can equal 0. Note that if IC4 = 0, the 8259A will assume operation in the MCS-80/85 mode.

SNGL: The SNGL bit is used to designate whether or not the 8259A is to be used alone or in the cascade mode. If the cascade mode is desired, SNGL must equal 0. In doing this, the 8259A will accept ICW3 for further cascade mode programming. If the 8259A is to be used as the single 8259A within a system, the SNGL bit must equal 1; ICW3 won't be accepted.

ADI: The ADI bit is used to specify the address interval for the MCS-80/85 mode. If a 4-byte address interval is to be used, ADI must equal 1. For an 8-byte address interval, ADI must equal 0. The state of ADI is ignored when the 8259A is in the MCS-86/88 mode.

LTIM: The LTIM bit is used to select between the two IR input triggering modes. If LTIM = 1, the level triggered mode is selected. If LTIM = 0, the edge triggered mode is selected.

A5-A15: The A5-A15 bits are used to select the interrupt vector address when in the MCS-80/85 mode. There are two programming formats that can be used to do this. Which one is implemented depends upon the selected address interval (ADI). If ADI is set for the 4-byte interval, then the 8259A will automatically insert A0-A4 (A0, A1 = 0 and A2, A3, A4 = IR0-7). Thus A5-A15 must be user selected by programming the A5-A15 bits with the desired address. If ADI is set for the 8-byte interval, then A0-A5 are automatically inserted (A0, A1, A2 = 0 and A3, A4, A5 = IR0-7). This leaves A6-A15 to be selected by programming the A6-A15 bits with the desired address. The state of bit 5 is ignored in the latter format.

T3-T7: The T3-T7 bits are used to select the interrupt type when the MCS-86/88 mode is used. The programming of T3-T7 selects the upper 5 bits. The lower 3 bits are automatically inserted, corresponding to the IR level causing the interrupt. The state of bits A5-A10 will be ignored when in the MCS-86/88 mode. Establishing the actual memory address of the interrupt is shown in Figure 22.

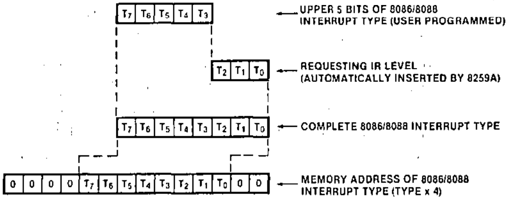

T7 T6 T5 T4 T3
UPPER 5 BITS OF 8086/8088 INTERRUPT TYPE (USER PROGRAMMED)
T2 T1 T0
REQUESTING IR LEVEL (AUTOMATICALLY INSERTED BY 8259A)
T7 T6 T5 T4 T3 T2 T1 T0
COMPLETE 8086/8088 INTERRUPT TYPE
0 0 0 0 T7 T6 T5 T4 T3 T2 T1 T0 0 0
MEMORY ADDRESS OF 8086/8088 INTERRUPT TYPE (TYPE x 4)

Figure 22. Establishing Memory Address of 8086/8088 Interrupt Type

ICW3

The 8259A will only accept ICW3 if programmed in the cascade mode (ICW1, SNGL = 0). ICW3 is used for specific programming within the cascade mode. Bit definition of ICW3 differs depending on whether the 8259A is a master or a slave. Definition of the ICW3 bits is as follows:

S0-7 (Master): If the 8259A is a master (either when the SP/$\overline{EN}$ pin is tied high or in the buffered mode when M/S = 1 in ICW4), ICW3 bit definition is S0-7, corresponding to "slave 0-7". These bits are used to establish which IR inputs have slaves connected to them. A 1 designates a slave, a 0 no slave. For example, if a slave was connected to IR3, the S3 bit should be set to a 1. (S0) should be last choice for slave designation.

ID0-ID2 (Slave): If the 8259A is a slave (either when the SP/$\overline{EN}$ pin is low or in the buffered mode when M/S = 0 in ICW4), ICW3 bit definition is used to establish its individual identity. The ID code of a particular slave must correspond to the number of the masters IR input it is connected to. For example, if a slave was connected to IR6 of the master, the slaves ID0-2 bits should be set to ID0 = 0, ID1 = 1, and ID2 = 1.

ICW4

The 8259A will only accept ICW4 if it was selected in ICW1 (bit IC4 = 1). Various modes are offered by using ICW4. Bit definition of ICW4 is as follows:

μPM: The μPM bit allows for selection of either the MCS-80/85 or MCS-86/88 mode. If set as a 1 the MCS-86/88 mode is selected, if a 0, the MCS-80/85 mode is selected.

AEOI: The AEOI bit is used to select the automatic end of interrupt mode. If AEOI = 1, the automatic end of interrupt mode is selected. If AEOI = 0, it isn't selected; thus an EOI command must be used during a service routine.

M/S: The M/S bit is used in conjunction with the buffered mode. If in the buffered mode, M/S defines whether the 8259A is a master or a slave. When M/S is set to a 1, the 8259A operates as the master; when M/S is 0, it operates as a slave. If not programmed in the buffered mode, the state of the M/S bit is ignored.

BUF: The BUF bit is used to designate operation in the buffered mode, thus controlling the use of the $\overline{SP}/\overline{EN}$ pin. If BUF is set to a 1, the buffered mode is programmed and $\overline{SP}/\overline{EN}$ is used as a transceiver enable output. If BUF is 0, the buffered mode isn't programmed and $\overline{SP}/\overline{EN}$ is used for master/slave selection. Note if ICW4 isn't programmed, $\overline{SP}/\overline{EN}$ is used for master/slave selection.

SFNM: The SFNM bit designates selection of the special fully nested mode which is used in conjunction with the cascade mode. Only the master should be programmed in the special fully nested mode to assure a truly fully nested structure among the slave IR inputs. If SFNM is set to a 1, the special fully nested mode is selected; if SFNM is 0, it is not selected.

4.2 OPERATIONAL COMMAND WORD (OCWs)

Once initialized by the ICWs, the 8259A will most likely be operating in the fully nested mode. At this point, operation can be further controlled or modified by the use of OCWs (Operation Command Words). Three OCWs are available for programming various modes and commands. Unlike the ICWs, the OCWs needn't be in any type of sequential order. Rather, they are issued by the processor as needed within a program.

Figure 23, the OCW programming format, shows the bit designation and short definition of each OCW. With the OCW format as reference, the functions of each OCW will be explained individually.

OCW1

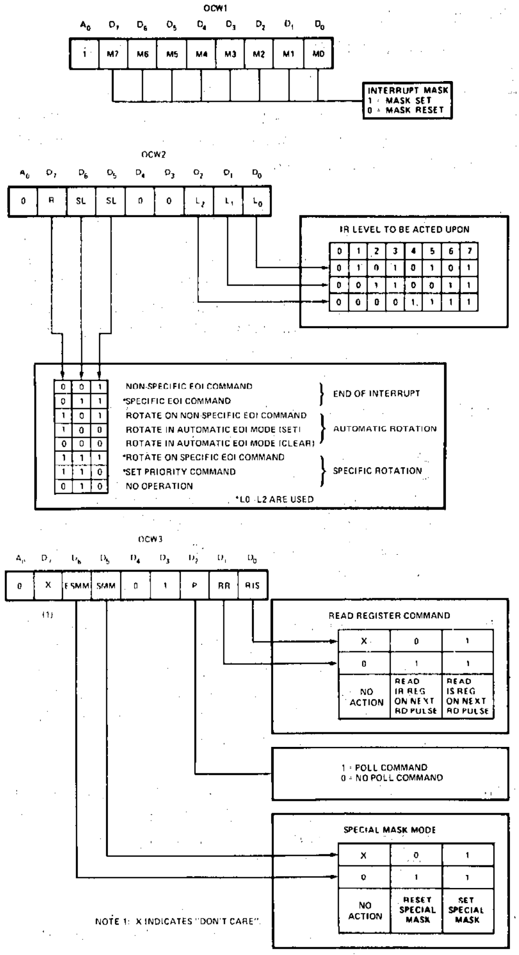
OCW1 is used solely for 8259A masking operations. It provides a direct link to the IMR (Interrupt Mask Register). The processor can write to or read from the IMR via OCW1. The OCW1 bit definition is as follows:

M0–M7: The M0–M7 bits are used to control the masking of IR inputs. If an M bit is set to a 1, it will mask the corresponding IR input. A 0 clears the mask, thus enabling the IR input. These bits convey the same meaning when being read by the processor for status update.

OCW2

OCW2 is used for end of interrupt, automatic rotation, and specific rotation operations. Associated commands and modes of these operations (with the exception of AEOI initialization), are selected using the bits of OCW2 in a combined fashion. Selection of a command or mode should be made with the corresponding table for OCW2 in the OCW programming format (Figure 20), rather than on a bit by bit basis. However, for completeness of explanation, bit definition of OCW2 is as follows:

L0–L2: The L0–L2 bits are used to designate an interrupt level (0–7) to be acted upon for the operation selected by the EOI, SL, and R bits of OCW2. The level designated will either be used to reset a specific ISR bit or to set a specific priority. The L0–L2 bits are enabled or disabled by the SL bit.


OCW1
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 M7 M6 M5 M4 M3 M2 M1 M0
INTERRUPT MASK
1 = MASK SET
0 = MASK RESET
OCW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 R SL EOI 0 0 L2 L1 L0
IR LEVEL TO BE ACTED UPON
0 1 2 3 4 5 6 7
0 1 0 1 0 1 0 1
0 0 1 1 0 0 1 1
0 0 0 0 1 1 1 1
0 0 1 NON-SPECIFIC EOI COMMAND
0 1 1 *SPECIFIC EOI COMMAND
END OF INTERRUPT
1 0 1 ROTATE ON NON-SPECIFIC EOI COMMAND
1 0 0 ROTATE IN AUTOMATIC EOI MODE (SET)
0 0 0 ROTATE IN AUTOMATIC EOI MODE (CLEAR)
AUTOMATIC ROTATION
1 1 1 *ROTATE ON SPECIFIC EOI COMMAND
1 1 0 *SET PRIORITY COMMAND
0 1 0 NO OPERATION
SPECIFIC ROTATION
*L0 - L2 ARE USED
OCW3
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 X ESMM SMM 0 1 P RR RIS
(1)
READ REGISTER COMMAND
X 0 1
0 1 1
NO ACTION | READ IR REG ON NEXT RD PULSE | READ IS REG ON NEXT RD PULSE
1 = POLL COMMAND
0 = NO POLL COMMAND
SPECIAL MASK MODE
X 0 1
0 1 1
NO ACTION | RESET SPECIAL MASK | SET SPECIAL MASK
NOTE 1: X INDICATES "DON'T CARE".

SOME OF THE TERMINOLOGY USED MAY DIFFER SLIGHTLY FROM EXISTING 8259A DATA SHEETS. THIS IS DONE TO BETTER CLARIFY AND EXPLAIN THE PROGRAMMING OF THE 8259A, THE OPERATIONAL RESULTS REMAIN THE SAME.

Figure 23. Operational Command Words (OCWs) Programming Format

EOI: The EOI bit is used for all end of interrupt commands (not automatic end of interrupt mode). If set to a 1, a form of an end of interrupt command will be executed depending on the state of the SL and R bits. If EOI is 0, an end of interrupt command won't be executed.

SL: The SL bit is used to select a specific level for a given operation. If SL is set to a 1, the L0–L2 bits are enabled. The operation selected by the EOI and R bits will be executed on the specified interrupt level. If SL is 0, the L0–L2 bits are disabled.

R: The R bit is used to control all 8259A rotation operations. If the R bit is set to a 1, a form of priority rotation will be executed depending on the state of SL and EOI bits. If R is 0, rotation won't be executed.

OCW3

OCW3 is used to issue various modes and commands to the 8259A. There are two main categories of operation associated with OCW3: interrupt status and interrupt masking. Bit definition of OCW3 is as follows:

RIS: The RIS bit is used to select the ISR or IRR for the read register command. If RIS is set to 1, ISR is selected. If RIS is 0, IRR is selected. The state of the RIS is only honored if the RR bit is a 1.

RR: The RR bit is used to execute the read register command. If RR is set to a 1, the read register command is issued and the state of RIS determines the register to be read. If RR is 0, the read register command isn't issued.

P: The P bit is used to issue the poll command. If P is set to a 1, the poll command is issued. If it is 0, the poll command isn't issued. The poll command will override a read register command if set simultaneously.

SMM: The SMM bit is used to set the special mask mode. If SMM is set to a 1, the special mask mode is selected. If it is 0, it is not selected. The state of the SMM bit is only honored if it is enabled by the ESMM bit.

ESMM: The ESMM bit is used to enable or disable the effect of the SMM bit. If ESMM is set to a 1, SMM is enabled. If ESMM is 0, SMM is disabled. This bit is useful to prevent interference of mode and command selections in OCW3.

5. APPLICATION EXAMPLES

In this section, the 8259A is shown in three different application examples. The first is an actual design implementation supporting an 8080A microprocessor system, "Power Fail/Auto Start with Battery Back-Up RAM". The second is a conceptual example of incorporating more than 64 interrupt levels in an 8080A or 8085A system, "78 Level Interrupt System". The third application is a conceptual design using an 8086 system, "Timer Controlled Interrupts". Although specific microprocessor systems are used in each example, these applications can be applied to either MCS-80, MCS-85, MCS-86, or MCS-88 systems, providing the necessary hardware and software changes are made. Overall, these applications should serve as a useful guide, illustrating the various procedures in using the 8259A.

5.1 POWER FAIL/AUTO-START WITH BATTERY BACK-UP RAM

The first application illustrates the 8259A used in an 8080A system, supporting a battery back-up scheme for the RAM (Random Access Memory) in a microcomputer system. Such a scheme is important in numerical and process control applications. The entire microcomputer system could be supported by a battery back-up scheme, however, due to the large amount of current usually required and the fact that most machinery is not supported by an auxiliary power source, only the state of calculations and variables usually need to be saved. In the event of a loss of power, if these items are not already stored in RAM, they can be transferred there and saved using a simple battery back-up system.

The vehicle used in this application is the Intel® SBC-80/20 Single Board Computer. An 8259A is used in the SBC-80/20 along with control lines helpful in implementing the power-down and automatic restart sequence used in a battery back-up system. The SBC-80/20 also contains user-selectable jumpers which allow the on-board RAM to be powered by a supply separate from the supply used for the non-RAM components. Also, the output of an undedicated latch is available to be connected to the IR inputs of the 8259A (the latch is cleared via an output port). In addition, an undedicated, buffered input line is provided, along with an input to the RAM decoder that will protect memory when asserted.

The additional circuitry to be described was constructed on an SBC-905 prototyping board. An SBC-635 power supply was used to power the non-RAM section of the SBC-80/20 while an external DC supply was used to simulate the back-up battery supplying power to the RAM. The SBC-635 was used since it provides an open collector ACLO output which indicates that the AC input line voltage is below 103/206 VAC (RMS).

The following is an example of a power-down and restart sequence that introduces the various power fail signals.

1. An AC power failure occurs and the ACLO goes high (ACLO is pulled up by the battery supply). This indicates that DC power will be reliable for at most 7.5 ms. The power fail circuitry generates a Power Fail Interrupt ($\overline{\text{PFI}}$) signal. This signal sets the $\overline{\text{PFI}}$ latch, which is connected to the IR0 input of the 8259A, and sets the Power Fail Sense (PFS) latch. The state of this latch will indicate to the processor, upon reset, whether it is coming up from a power failure (warm start) or if it is coming up initially (cold start).
2. The processor is interrupted by the 8259A when the PFI latch is set. This pushes the pre-power-down program counter onto the stack and calls the service routine for the IR0 input. The IR0 service routine saves the processor status and any other needed variables. The routine should end with a HALT instruction to minimize bus transitions.
3. After a predetermined length of time (5 ms in this example) the power fail circuitry generates a Memory Protect ($\overline{\text{MPRO}}$) signal. All processing for the power failure (including the interrupt response delays) must be completed within this 5 ms window. The $\overline{\text{MPRO}}$ signal ensures that spurious transitions on the system control bus caused by power going down do not alter the contents of the RAM.
4. DC power goes down.
5. AC power returns. The power-on reset circuitry on the SBC-80/20 generates a system RESET.
6. The processor reads the state of the $\overline{\text{PFS}}$ line to determine the appropriate start-up sequence. The PFS latch is cleared, the MPRO signal is removed, and the PFI latch driving IR0 is cleared by the Power Fail Sense Reset ($\overline{\text{PFSR}}$) signal. The system then continues from the pre-power-down location for a warm start by restoring the processor status and popping the pre-power-down program counter off the stack.

Figure 24 illustrates this timing.

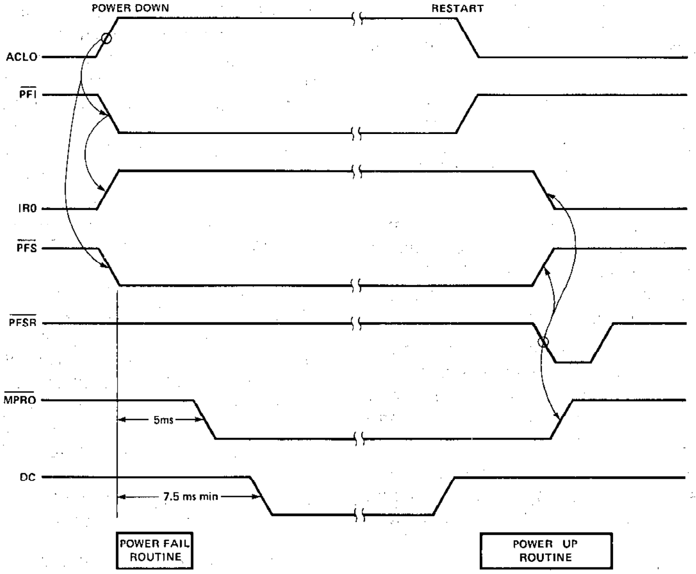


Figure 24. Power Down Restart Timing

Figure 25 shows the block diagram for the system. Notice that the RAM, the RAM decoder, and the power-down circuitry are powered by the battery supply.

The schematic of the power-down circuitry and the SBC-80/20 interface is shown in Figure 26. The design is very straightforward and uses CMOS logic to minimize the battery current requirements. The cold start switch is necessary to ensure that during a cold start, the $\overline{\text{PFS}}$ line is indicating "cold start" sense ($\overline{\text{PFS}}$ high). Thus, for a cold start, the cold start switch is depressed during power on. After that, no further action is needed. Notice that the $\overline{\text{PFI}}$ signal sets the on-board PFI latch. The output of this latch drives the 8259A IR0 input. This latch is cleared during the restart routine by executing an OUTput D4H instruction. The state of the $\overline{\text{PFS}}$ line may be read on the least significant data bus line (DB0) by executing an INput D4H instruction. An 8255 port (8255 #1, port C, bit 0) is used to control the $\overline{\text{PFSR}}$ line.

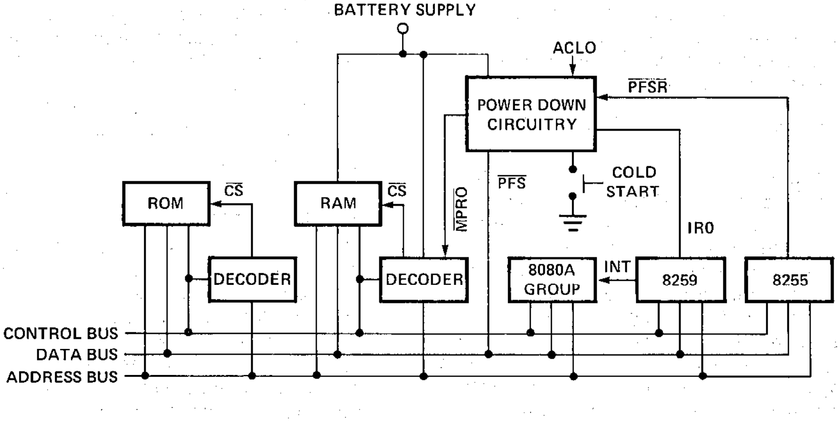


Figure 25. Block Diagram of SBC 80/20 with Power Down Circuit

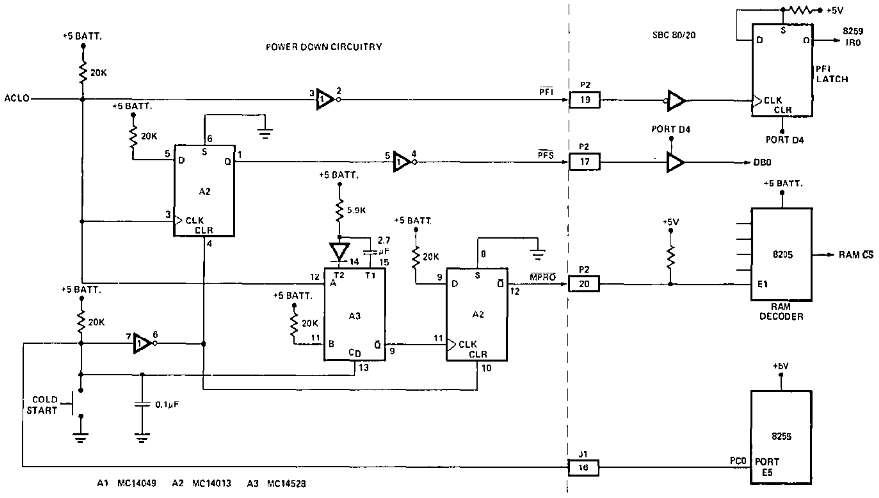


Figure 26. Power Down Circuit - SBC 80/20 Interface

The fully nested mode for the 8259A is used in its initial state to ensure the IR0 always has the highest priority. The remaining IR inputs can be used for any other purpose in the system. The only constraint is that the service routines must enable interrupts as early as possible. Obviously, this is to ensure that the power-down interrupt does not have to wait for service. If a rotating priority scheme is desired, another 8259A could be added as a slave and be programmed to operate in a rotating mode. The master would remain in the initial state of the fully nested mode so that the IR0 still remains the highest priority input.

The software to support the power-down circuitry is shown in Figure 27. The flow for each label will be discussed.

After any system reset, the processor starts execution at location 0000H (START). The $\overline{\text{PFS}}$ status is read and execution is transferred to CSTART if $\overline{\text{PFS}}$ indicates a cold start (i.e., someone is depressing the cold start switch) or WSTART if a warm start is indicated ($\overline{\text{PFS}}$ LOW). CSTART is the start of the user's program. The Stack Pointers (SP) and device initialization were included just to remind the reader that these must occur. The first EI instruction must appear after the 8259A has received its initialization sequence. The 8259A (and other devices) are initialized in the INIT subroutine.

When a power failure occurs, execution is vectored by the 8259A to REGSAV by way of the jump table at JSTART. The pre-power-down program counter is placed on the stack. REGSAV saves the processor registers and flags in the usual manner by pushing them onto the stack. Other items, such as output port status, programmable peripheral states, etc., are pushed onto the stack at this time. The Stack Pointer (SP) could be pushed onto the stack by way of the register pair HL but the top of the stack can exist anywhere in memory and there is no way then of knowing where that is when in the power-up routine. Thus, the SP is saved at a dedicated location in RAM. It isn't really necessary to send an EOI command to the 8259A in REGSAV since power will be removed from the 8259A, but one is included for completeness. The final instruction before actually losing power is a HALT. This minimizes somewhat spurious transitions on the various busses and lets the processor die gracefully.

On reset, when a warm start is detected, execution is transferred to WSTART. WSTART activates $\overline{\text{PFSR}}$ by way of the 8255 (all outputs go low then the 8255 is initialized). In the power-down circuitry, $\overline{\text{PFSR}}$ clears the PFS latch and removes the $\overline{\text{MPRO}}$ signal which then allows access to the RAM. WSTART also clears the PFI latch which arms the 8259A IR0 input. Then the 8259A is re-initialized along with any other devices. The SP is retrieved from RAM and the processor registers and flags are restored by popping them off the stack. Interrupts are then enabled. Now the power-down program counter is on top of the stack, so executing a RETurn instruction transfers the processor to exactly where it left off before the power failure.

Aside from illustrating the usefulness of the 8259A (and the SBC-80/20) in implementing a power failure protected microcomputer system, this application should also point out a way of preserving the processor status when using interrupts.

```
LOC  OBJ        SEQ       SOURCE STATEMENT

                  0 ;
                  1 ;
                  2 ;POWER DOWN AND RESTART FOR THE SBC 80/20
                  3 ;
                  4 ;
                  5 ;SYSTEM EQUATES
00DA              6 PT59A   EQU     0DAH           ;8259 PORT WITH A0=0
00DB              7 PT59B   EQU     0DBH           ;8259 PORT WITH A0=1
00E7              8 PPI1CT  EQU     0E7H           ;8255 #1 CONTROL PORT
00E6              9 PPI1C   EQU     0E6H           ;8255 #1 PORT C
3800             10 SPSAVE  EQU     3800H          ;SP STORAGE IN RAM
0001             11 JPT     EQU     01H            ;MSB OF 8259 JUMP TABLE
                 12 ;
                 13 ;
                 14 ;STARTING POINT AFTER SYSTEM RESET
                 15 ;
                 16 ;
0000             17         ORG     0H
0000 DB04        18 START:  IN      0D4H           ;READ PFS/ STATUS
0002 1F          19         RAR                    ;PFS/ ON D0, PUT IN CARRY
0003 DA2001      20         JC      CSTART         ;PFS/=1, THEN COLD START
                 21 ;
                 22 ;
                 23 ;RSTART LOCATION  PFS/=0, THEN WARM START
                 24 ;
                 25 ;
0006 3E80        26 WSTART: MVI     A,80H          ;SET 8255 #1 TO OUTPUT MODE
0008 D3E7        27         OUT     PPI1CT         ;8255 CONTROL PORT, PFSR/ GOES LOW
                 28         ;
                 29         ;OUTPUT COMMAND MAKES PFSR/ GO LOW WHICH REMOVES MPRO/ AND
                 30         ;CLEARS PFS LATCH
                 31         ;
000A 3E01        32         MVI     A,01H          ;RETURN PFSR/ HIGH
000C D3E6        33         OUT     PPI1C          ;8255 #1 PORT C
000E D3D4        34         OUT     0D4H           ;RESET PFI LATCH
0010 CD1D00      35         CALL    INIT           ;GO INITIALIZE EVERYTHING
0013 2A0038      36         LHLD    SPSAVE         ;RETRIVE SP FROM RAM
0016 F9          37         SPHL                   ;PUT BACK INTO SP
0017 C1          38         POP     B              ;RESTORE BC
0018 D1          39         POP     D              ;RESTORE DE
0019 E1          40         POP     H              ;RESTORE HL
001A F1          41         POP     PSW            ;RESTORE A PLUS FLAGS
001B FB          42         EI                     ;ENABLE INTERRUPTS
001C C9          43         RET                    ;PRE-POWER-DOWN PC ON TOP OF STACK
                 44                                ;RETURN TO IT
                 45 ;
                 46 ;
                 47 ;INITIALIZATION ROUTINE.  AT LEAST DO 8259 BUT OTHERS CAN BE ADDED
                 48 ;
                 49 ;
001D 3E16        50 INIT:   MVI     A,16H          ;F=1,S=1,A7-A5=0  ICW1
001F D3DA        51         OUT     PT59A          ;8259 PORT WITH A0=0
0021 3E01        52         MVI     A,JPT          ;MSB OF JUMP TABLE ICW2
0023 D3DB        53         OUT     PT59B          ;8259 PORT WITH A0=1
                 54         ;
                 55         ;ADD ANY OTHER INITIALIZATIONS HERE
                 56         ;
0025 C9          57         RET                    ;RETURN
                 58 ;- - - - - - - - - - - - - - - - - - - - - - - - - - -
                 59 ;
                 60 ;POWER DOWN ROUTINE TO SAVE REGISTERS AND STACK
                 61 ;
                 62 ;
0026 F5          63 REGSAV: PUSH    PSW            ;SAVE A PLUS FLAGS
0027 E5          64         PUSH    H              ;SAVE HL
0028 D5          65         PUSH    D              ;SAVE DE
0029 C5          66         PUSH    B              ;SAVE BC
002A 210000      67         LXI     H,0000H        ;SET HL TO GET SP
002D 39          68         DAD     SP             ;SP NOW IN HL
002E 220038      69         SHLD    SPSAVE         ;SAVE SP IN RAM
                 70         ;
                 71         ;EOI NOT REALLY NEEDED BUT INCLUDED FOR COMPLETENESS
                 72         ;
0031 3E20        73         MVI     A,20H          ;NON-SPECIFIC EOI
0033 D3DA        74         OUT     PT59A          ;8259 PORT WITH A0=0
0035 76          75         HLT                    ;HALT - GO DOWN GRACEFULLY
                 76 ;
                 77 ;
                 78 ;8259 JUMP TABLE  ONLY IR0 IS USED, OTHERS DIRECTED TO RAM
                 79 ;
                 80
0100             81         ORG     100H
0100 C32600      82 JSTART: JMP     REGSAV         ;IR0
0103 00          83         NOP
0104 C31038      84         JMP     3810H          ;IR1
0107 00          85         NOP
0108 C32038      86         JMP     3820H          ;IR2
010B 00          87         NOP
010C C33038      88         JMP     3830H          ;IR3
010F 00          89         NOP
0110 C34038      90         JMP     3840H          ;IR4
0113 00          91         NOP
0114 C35038      92         JMP     3850H          ;IR5
0117 00          93         NOP
0118 C36038      94         JMP     3860H          ;IR6
011B 00          95         NOP
011C C37038      96         JMP     3870H          ;IR7
011F 00          97         NOP
                 98 ;
                 99 ;
                100 ;COLD START LOCATION  USER'S PROGRAM ENTERS HERE
                101 ;
                102 ;
0120 31003F     103 CSTART: LXI     SP,3F00H       ;INITIALIZE SP
0123 CD1D00     104         CALL    INIT           ;INITIALIZE EVERYTHING ELSE
0126 D3D4       105         OUT     0D4H           ;RESET PFI LATCH
0129 FB         106         EI                     ;ENABLE INTERRUPTS
                107         ;
                108         ;USER PROGRAM STARTS HERE
                109         ;
                110         END                    ;DONE
```

Figure 27. Power Down and Restart Software

5.2 78 LEVEL INTERRUPT SYSTEM

The second application illustrates an interrupt structure with greater than 64 levels for an 8080A or 8085A system. In the cascade mode, the 8259A supports up to 64 levels with direct vectoring to the service routine. Extending the structure to greater than 64 levels requires polling, using the poll command. A 78 level interrupt structure is used as an illustration; however, the principles apply to systems with up to 512 levels.

To implement the 78 level structure, 3 tiers of 8259A's are used. Nine 8259A's are cascaded in the master-slave scheme, giving 64 levels at tier 2. Two additional 8259A's are connected, by way of the INT outputs, to two of the 64 inputs. The 16 inputs at tier 3, combined with the 62 remaining tier 2 inputs, give 78 total levels. The fully nested structure is preserved over all levels, although direct vectoring is supplied for only the tier 2 inputs. Software is required to vector any tier 3 requests. Figure 28 shows the tiered structure used in this example. Notice that the tier 3 8259A's are connected to the bottom level slave (SA7). The master-slaves are interconnected as shown in "Interrupt Cascading", while the tier 3 8259A's are connected as "masters"; that is, the SP/EN pins are pulled high and the CAS pins are left unconnected. Since these 8259A's are only going to be used with the poll command, no INTA is required, therefore the INTA pins are pulled high.

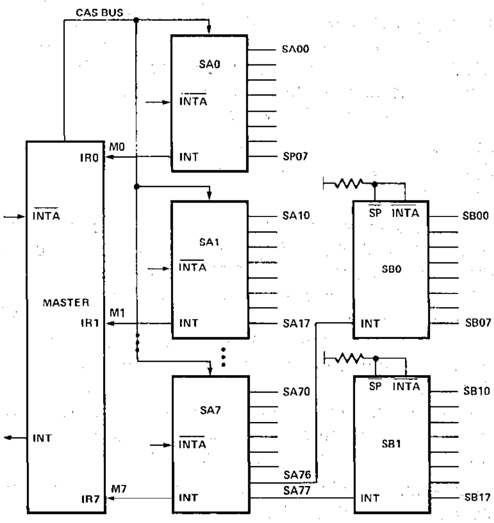


Figure 28. 78 Level Interrupt Structure

The concept used to implement the 78 levels is to directly vector to all tier 2 input service routines. If a tier 2 input contains a tier 3 8259A, the service routine for that input will poll the tier 3 8259A and branch to the tier 3 input service routine based on the poll word read after the poll command. Figure 29 shows how the jump table is organized assuming a starting location of 1000H and contiguous tables for all the tier 2 8259A's. Note that "SA35" denotes the IR5 input of the slave connected to the master IR3 input. Also note that for the normal tier 2 inputs, the jump table vectors the processor directly to the service routine for that input, while for the tier 2 inputs with 8259A's connected to their IR inputs, the processor is vectored to a service routine (i.e., SB0) which will poll to determine the actual tier 3 input requesting service. The polling routine utilizes the jump table starting at 1200H to vector the processor to the correct tier 3 service routine.

Each 8259A must receive an initialization sequence regardless of the mode. Since the tier 1 and 2 8259A's are in cascade and the special fully nested mode is used (covered shortly), all ICWs are required. The tier 3 8259A's don't require ICW3 or ICW4 since only polling will be used on them and they are connected as masters not in the cascade mode. The initialization sequence for each tier is shown in Figure 30. Notice that the master is initialized with a "dummy" jump table starting at 00H since all vectoring is done by the slaves. The tier 3 devices also receive "dummy" tables since only polling is used on tier 3.

As explained in "Interrupt Cascading", to preserve a truly fully nested mode within a slave, the master 8259A should be programmed in the special fully nested mode. This allows the master to acknowledge all interrupts at and above the level in service disregarding only those of lower priority. The special fully nested mode is programmed in the master only, so it only affects the immediate slaves (tier 2 not tier 3). To implement a fully nested structure among tier 3 slaves some special housekeeping software is required in all the tier-2-with-tier-3-slave routines. The software should simply save the state of the tier 2 IMR, mask all the lower tier 2 interrupts, then issue a specific EOI, resetting the ISR of the tier 2 interrupt level. On completion of the routine the IMR is restored.

Figure 31 shows an example flow and program for any tier 2 service routine without a tier 3 8259A. Figure 32 shows an example flow and program for any tier 2 service routine with a tier 3 8259A. Notice the reading of the ISR in both examples; this is done to determine whether or not to issue an EOI command to the master (refer to the section on "Special Fully Nested Mode" for further details).

```
LOCATION    8259    CODE                   COMMENTS
 1000 H     SA0     JMP   SA00       ; SA00 SERVICE ROUTINE
    .
 101C H             JMP   SA07       ; SA07 SERVICE ROUTINE
 1020 H     SA1     JMP   SA10       ; SA10 SERVICE ROUTINE
    .
 103C H             JMP   SA17       ; SA17 SERVICE ROUTINE
    .        .              .
    .        .              .        ; SA20–SA67 SERVICE ROUTINES
    .        .              .
 10E0 H     SA7     JMP   SA70       ; SA70 SERVICE ROUTINE
    .
 10F8 H             JMP   SB0        ; SB0 POLL ROUTINE
 10FC H             JMP   SB1        ; SB1 POLL ROUTINE
 1200 H     SB0     JMP   SB00       ; SB00 SERVICE ROUTINE
    .
 121C H             JMP   SB07       ; SB07 SERVICE ROUTINE
 1220 H     SB1     JMP   SB10       ; SB10 SERVICE ROUTINE
    .
 123C H             JMP   SB17       ; SB17 SERVICE ROUTINE
```

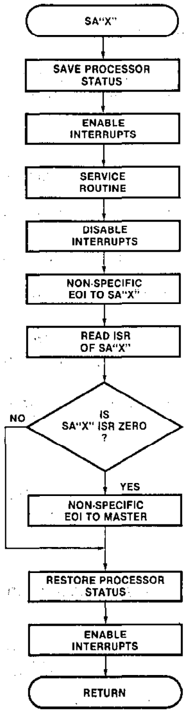
Figure 29. Jump Table Organization

```
; INITIALIZATION SEQUENCE FOR 78 LEVEL INTERRUPT STRUCTURE
;
; INITIALIZE MASTER
;
 MINT:    MVI    A,15H     ; ICWI, LTM = 0, ADI = 1, S = 0, IC4 = 1
          OUT    MPTA      ; MASTER PORT A0 = 0
          MVI    A,00H     ; ICW2, DUMMY ADDRESS
          OUT    MPTB      ; MASTER PORT A0 = 1
          MVI    A,0FFH    ; ICW3, S7-S0 = 1
          OUT    MPTB      ; MASTER PORT A0 = 1
          MVI    A,I0H     ; ICW4, SFNM = 1
          OUT    MPTB      ; MASTER PORT A0 = 1
;
; INITIALIZE SA SLAVES - X DENOTES SLAVE ID (SEE KEY)
;
 SAXINT:  MVI    A,α       ; SEE KEY FOR ICW1, LTM = 0, ADI = 1, S = 0, IC4 = 1
          OUT    SAXPTA    ; SA"X" PORT A0 = 0
          MVI    A, 10H    ; ICW2, ADDRESS MSB
          OUT    SAXPTB    ; SA"X" PORT A0 = 1
          MVI    A0XH      ; ICW3, SA ID
          OUT    SAXPTB    ; SA"X" PORT A0 = 1
          MVI    A10H      ; ICW4, SFNM = 1
          OUT    SAXPTB    ; SA"X" PORT A0 = 1
;
; REPEAT ABOVE FOR EACH SA SLAVE
;
; INITIALIZE SB SLAVES - X DENOTES 0 or 1 (DO SB0, REPEAT FOR SB1)
;
 SBXINT   MVI    A,I6H     ; ICW1, LTM = 0, ADI = 1, S = 1, IC4 = 0
          OUT    SBXPTA    ; SB"X" PORT A0 = 0
          MVI    A,00H     ; ICW2, DUMMY ADDRESS
          OUT    SBXPTB    ; SB"X" PORT A0 = 1
```

SA INITIALIZATION KEY

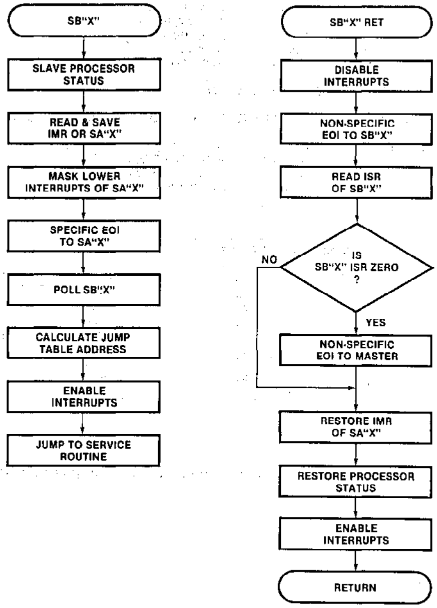
SA"X"	α (ICW1)	JUMP TABLE START (H)
0	15	1000
1	35	1020
2	55	1040
3	75	1060
4	95	1080
5	B5	10A0
5	D5	10C0
7	F5	10E0

Figure 30. Initialization Sequence for 78 Level Interrupt Structure



```
; SA"X" ROUTINE - GENERAL INTERRUPT SERVICE ROUTINE
; FOR TIER 2 INTERRUPTS WITHOUT TIER 3 8259A
;
 SAX:     PUSH D            ; SAVE DE
          PUSH B            ; SAVE BC
          PUSH H            ; SAVE HL
          PUSH PSW          ; SAVE A, FLAGS
          EI                ; ENABLE INTERRUPTS
;
; SERVICE ROUTINE GOES HERE
;
          DI                ; DISABLE INTERRUPTS
          MVI    20H        ; OCW2, NON-SPECIFIC EOI
          OUT    SAXPTA     ; SA"X" PORT A0 = 0
          MUI    A,0BH      ; OCW3, READ REGISTER, ISR
          OUT    SAXPTA     ; SA"X" PORT A0 = 0
          IN     SAXPTA     ; SA"X" PORT A0 = 0, SA"X" ISR
          ANI    0FFH       ; TEST FOR ZERO
          JZN    SAXRSR     ; IF NOT ZERO, RESTORE STATUS
          MVI    A,0BH      ; OCW2, NON-SPECIFIC EOI
          OUT    MASPTA     ; MASTER PORT A0 = 0
 SAXRSR:  POP    PSW        ; RESTORE A, FLAGS
          POP    H          ; RESTORE HL
          POP    B          ; RESTORE BC
          POP    D          ; RESTORE DE
          EI                ; ENABLE INTERRUPTS
          RET               ; RETURN
```

Figure 31. Example Service Routine for Tier 2 Interrupt (SA"X") without Tier 3 8259A (SB"X")



```
; SB"X" ROUTINE - SERVICE ROUTINE FOR TIER 2
; INTERRUPTS WITH TIER 3 8259AS
 SBX:     PUSH D            ; SAVE DE
          PUSH B            ; SAVE BC
          PUSH H            ; SAVE HL
          PUSH PSW          ; SAVE A, FLAGS
          IN     SAXPTB     ; READ SA"X" IMR
          MOV    D,A        ; SAVE
          MVI    A,XXH      ; MASK SA"X" LOWER IR
          OUT    SAXPTB     ; SA"X" PORT A0 = 1
          MVI    A,6XH      ; OCW2 SPECIFIC EOI SA"X"
          OUT    SAXPTA     ; SA"X" PORT A0 = 1
          LXI    H,1200H    ; JUMP TABLE START
          MVI    B,00H      ; CLEAR B
          MVI    A,0CH      ; OCW3, POLL COMMAND
          OUT    SBXPTA     ; SB"X" PORT A0 = 0
          IN     SBXPTA     ; GET POLL WORD
          ANI    07H        ; LIMIT TO 3 BITS
          ADD    A          ; GET TABLE OFFSET
          ADD    A          ;
          MOV    C,A        ; OFFSET TO C
          DAD    B          ; HL HAS TABLE ADDRESS
          EI                ; ENABLE INTERRUPTS
;
; SB"X"RET ROUTINE - FOR EOI AND MASK RESTORE
; AFTER SB"X" ROUTINE
;
 SBXRET   DI                ; DISABLE INTERRUPTS
          MVI    A,20H      ; OCW2, NON SPECIFIC EOI
          OUT    SBXPTA     ; SA"X" PORT A0 = 0
          MVI    A,0BH      ; OCW3, READ REGISTER ISR
          OUT    SAXPTA     ; SA"X" PORT A0 = 0
          IN     SBXPTA     ; SA"X" PORT A0 = 0, ISR
          ANI    0FFH       ; TEST FOR ZERO
          JNZ    SBXRSR     ; IF ≠ 0 RESTORE IMR
          MVI    A,20H      ; OCW2, NON-SPECIFIC EOI
          OUT    MASPTA     ; MASTER PORT A0 = 0
 SBXRSR:  MOV    A,D        ; RESTORE SA"X" IMR
          OUT    SAXPTB     ; SA"X" PORT A0 = 1
          POP    PSW        ; RESTORE A, FLAGS
          POP    H          ; RESTORE HL
          POP    B          ; RESTORE BC
          POP    D          ; RESTORE BC
          EI                ; RESTORE DE
          RET               ; RETURN
```

Figure 32. Example Service Routine for Tier 2 Interrupt (SA"X") with Tier 3 8259A (SB"X")

5.3 TIMER CONTROLLED INTERRUPTS

In a large number of controller type microprocessor designs, certain timing requirements must be implemented throughout program execution. Such time dependent applications include control of keyboards, displays, CRTs, printers, and various facets of industrial control. These examples, however, are just a few of many designs which require device servicing at specific rates or generation of time delays. Trying to maintain these timing requirements by processor control alone can be costly in throughput and software complexity. So, what can be done to alleviate this problem? The answer, use the 8259A Programmable Interrupt Controller and external timing to interrupt the processor for time dependent device servicing.

This application example uses the 8259A for timer controlled interrupts in an 8086 system. External timing is done by two 8253 Programmable Interval Timers. Figure 33 shows a block diagram of the timer controlled interrupt circuitry which was built on the breadboard area of an SDK-86 (system design kit). Besides the 8259A and the 8253's, the necessary I/O decoding is also shown. The timer controlled interrupt circuitry interfaces with the SDK-86 which serves as the vehicle of operation for this design.

A short overview of how this application operates is as follows. The 8253's are programmed to generate interrupt requests at specific rates to a number of the 8259A IR inputs. The 8259A processes these requests by interrupting the 8086 and vectoring program execution to the appropriate service routine. In this example, the routines use the SDK-86 display panel to display the number of the interrupt level being serviced. These routines are merely for demonstration purposes to show the necessary procedures to establish the user's own routines in a timer controlled interrupt scheme.

Let's go over the operation starting with the actual interrupt timing generation which is done by two 8253 Programmable Interval Timers (8253 #1 and 8253 #2). Each 8253 provides three individual 16-bit counters (counters 0-2) which are software programmable by the processor. Each counter has a clock input (CLK), gate input (GATE), and an output (OUT). The output signal is based on divisions of the clock input signal. Just how or when the output occurs is determined by one of the 8253's six programmable modes, a programmable 16-bit count, and the state of the gate input.

Figure 34 shows the 8253 timing configuration used for generating interrupts to the 8259A. The SDK-86's PCLK (peripheral clock) signal provides a 400 ns period clock to CLK0 of 8253 #1. Counter 0 is used in mode 3 (square wave rate generator), and acts as a prescaler to provide the clock inputs of the other counters with a 10 ms period square wave. This 10 ms clock period made it easy to calculate exact timings for the other counters. Counter 2 of the 8253 #1 is used in mode 2 (rate generator), it is programmed to output a 10 ms pulse for every 200 pulses it receives (every 2 sec). The output of counter 2 causes an interrupt on IR1 of the 8259A. All the 8253 #2 counters are used in mode 5 (hardware triggered strobe) in which the gate input initiates counter operations. In this case the output of 8253 #1 counter 2 controls the gate of each 8253 #2 counter. When one of the 8253 #2 counters receive the 8253 #1 counter 2 output pulse on its gate, it will output a pulse (10 ms in duration) after a certain preprogrammed number of clock pulses have occurred. The programmed number of clock pulses for the 8253 #2 counters is as follows: 50 pulses (0.5 sec) for counter 0, 100 pulses (1 sec) for counter 1, and 150 pulses (1.5 sec) for counter 2. The outputs of these counters cause interrupt requests on IR2 through IR4 of the 8259A. Counter 1 of 8253 #1 is used in mode 0 (interrupt on terminal count). Unlike the other modes used which initialize operation automatically or by gate triggering, mode 0 allows software controlled counter initialization. When counter 1 of 8253 #1 is set during program execution, it will count 25 clocks (250 ms) and then pull its output high, causing an interrupt request on IR0 of the 8259A. Figure 35 shows the timing generated by the 8253's which cause interrupt request on the 8259A IR inputs.

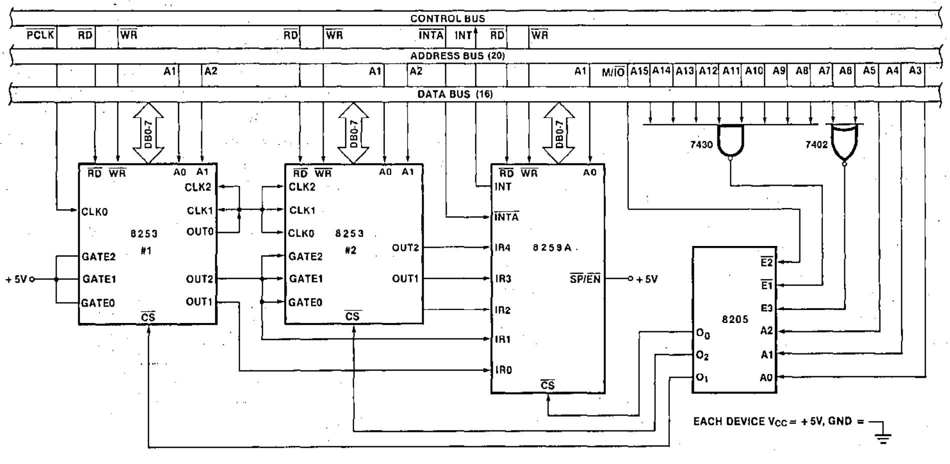


Figure 33. Timer Controlled Interrupt Circuit on SDK 86 Breadboard Area

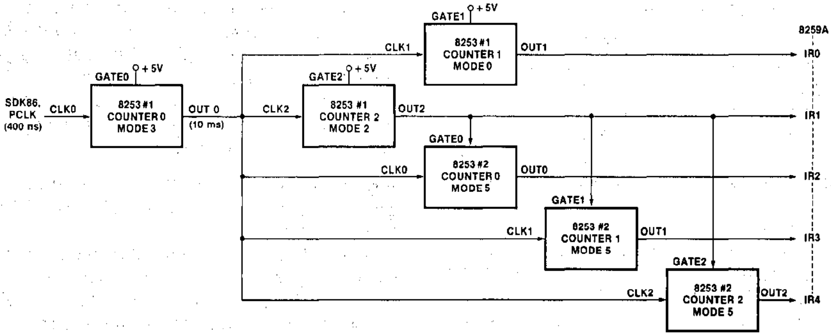


Figure 34. 8253 Timing Configuration for Timer Controlled Interrupts

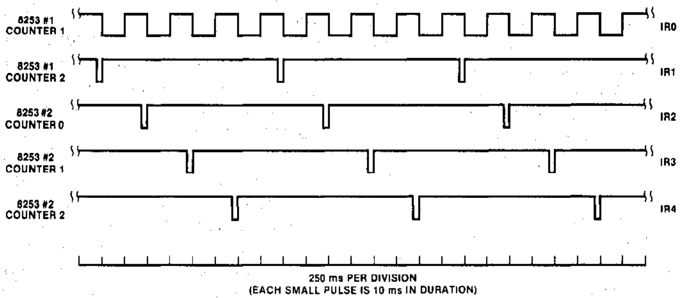
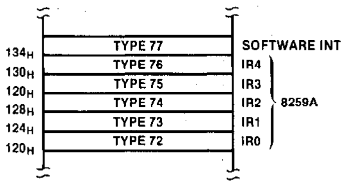


Figure 35. 8259A IR Input Signal From 8253S

There are basically two methods of timing generation that can be used in a timer controlled interrupt structure: dependent timing and independent timing. Dependent timing uses a single timing occurrence as a reference to base other timing occurrences on. On the other hand, independent timing has no mutual reference between occurrences. Industrial controller type applications are more apt to use dependent timing, whereas independent timing is prone to individual device control.

Although this application uses primarily dependent timing, independent timing is also incorporated as an example. The use of dependent timing can be seen back in Figure 34, where timing for IR2 through IR4 uses the IR1 pulse as reference. Each one of the 8253 #2 counters will generate an interrupt request a specific amount of times after the IR1 interrupt request occurs. When using the dependent method, as in this case, the IR2 through IR4 requests must occur before the next IR1 request. Independent timing is used to control the IR0 interrupt request. Note that its timing isn't controlled by any of the other IR requests. In this timer controlled interrupt configuration the dependent timing is initially set to be self running and the independent timing is software initialized. However, both methods can work either way by using the various 8253 modes to generate the same interrupt timing.

The 8259A processes the interrupts generated by the 8253's according to how it is programmed. In this application it is programmed to operate in the edge triggered mode, MCS-86/88 mode, and automatic EOI mode. In the edge triggered mode an interrupt request on an 8259A IR input becomes active on the rising edge. With this in mind, Figure 35 shows that IR0 will generate an interrupt every half second and IR1 through IR4 will each generate an interrupt every 2 seconds spaced apart at half second intervals. Interrupt vectoring in the MCS-86/88 mode is programmed so IR0, when activated, will select interrupt type 72. This means IR1 will select interrupt type 73, IR2 interrupt type 74, and so on through IR4. Since IR5 through IR7 aren't used, they are masked off. This prevents the possibility of any accidental interrupts and rids the necessity to tie the unused IR inputs to a steady level. Figure 36 shows the 8259A IR levels (IR0–IR4) with their corresponding interrupt type in the 8086 interrupt-vector table. Type 77 in the table is selected by a software "INT" instruction during program execution. Each type is programmed with the necessary code segment and instruction pointer values for vectoring to the appropriate service routine. Since the 8259A is programmed in the automatic EOI Mode, it doesn't require an EOI command to designate the completion of the service routine.



134_H	TYPE 77	SOFTWARE INT	
130_H	TYPE 76	IR4	8259A
120_H	TYPE 75	IR3	
128_H	TYPE 74	IR2	
124_H	TYPE 73	IR1	
120_H	TYPE 72	IR0	

Figure 36. Interrupt "Type" Designation

As mentioned earlier, the interrupt service routines in this application are used merely to demonstrate the timer controlled interrupt scheme, not to implement a particular design. Thus a service routine simply displays the number of its interrupting level on the SDK-86 display panel. The display panel is controlled by the 8279 Keyboard and Display Controller. It is initialized to display "Ir" in its two left-most digits during the entire display sequence. When an interrupt from IR1 through IR4 occurs the corresponding routine will display its IR number via the 8279. During each IR1 through IR4 service routine a software "INT77" instruction is executed. This instruction vectors program execution to the service routine designated by type 77, which sets the 8253 counter controlling IR0 so it will cause an interrupt in 250 ms. When the IR0 interrupt occurs its routine will turn off the digit displayed by the IR1 through IR4 routines. Thus each IR level (IR1-IR4) will be displayed for 250 ms followed by a 250 ms off time caused by IR0. Figure 37 shows the entire display sequence of the timer controlled interrupt application.

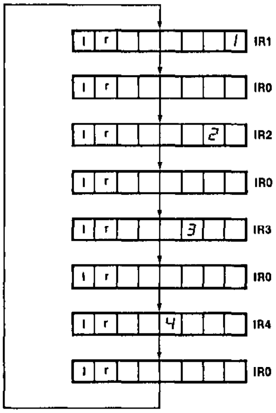


Figure 37. SDK Display Sequence for Timer Controlled Interrupts Program (Each Display Block Shown is 250 msec in Duration)

Now that we've covered the operation, let's move on to the program flow and structure of the timer controlled interrupt program. The program flow is made up of an initialization section and six interrupt service routines. The initialization program flow is shown in Figure 38. It starts by initializing some of the 8086's registers for program operation; this includes the extra segment, data segment, stack segment, and stack pointer. Next, by using the extra segement as reference, interrupt types 72 through 77 are set to vector interrupts to the appropriate routines. This is done by moving the code segment and instruction pointer values of each service routine into the corresponding type location. The 8253 counters are then programmed with the proper mode and count to provide the interrupt timing mentioned earlier. All counters with the exception of the 8253 #1, counter 1 are fully initialized at this point and will start counting. Counter 1 of 8253 #1 starts counting when its counter is loaded during the "INTR77" service routine, which will be covered shortly. Next, the 8259A is issued ICW1, ICW2, ICW4, and OCW1. The ICWs program the 8259A for the edge triggered mode, automatic EOI mode, and the proper interrupt vectoring (IR0, type 72). OCW1 is used to mask off the unused IR inputs (IR5-IR7). The 8279 is then set to display "IR" on its two left-most digits. After that the 8086 enables interrupts and a "dummy" main program is executed to wait for interrupt requests.

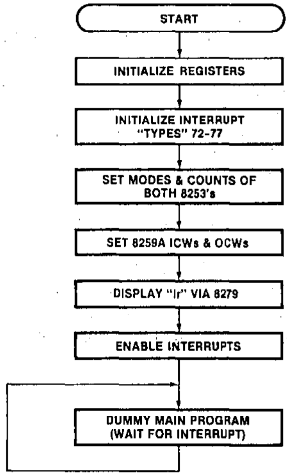
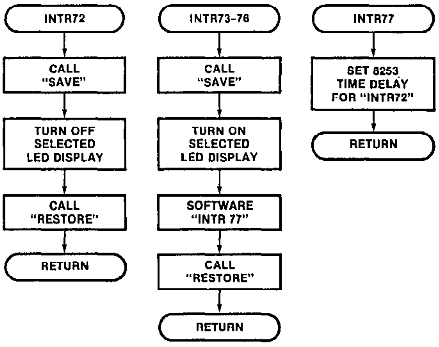

START
INITIALIZE REGISTERS
INITIALIZE INTERRUPT "TYPES" 72-77
SET MODES & COUNTS OF BOTH 8253's
SET 8259A ICWs & OCWs
DISPLAY "Ir" VIA 8279
ENABLE INTERRUPTS
DUMMY MAIN PROGRAM (WAIT FOR INTERRUPT)

Figure 38. Initialization Program Flow for Timer Controlled Interrupts

There are six different interrupt service routines used in the program. Five of these routines, "INTR72" through "INTR76", are vectored to via the 8259A. Figure 39A-C shows the program flow for all six service routines. Note that "INTR73" through "INTR76" (IR1-IR4) basically use the same flow. These four similar routines display the number of its interrupting IR level on the SDK-86 display panel. The "INTR77" routine is vectored to by software during each of the previously mentioned routines and sets up interrupt timing to cause the "INTR72" (IR0) routine to be executed. The "INTR72" routine turns off the number on the SDK-86 display panel.



A. INTERRUPT ON 8259A IR0

B. INTERRUPT ON 8259A IR1-IR4

C. SOFTWARE INVOKED INTERRUPT

Figure 39. A-C. Interrupts Service Routine Flow for Timer Controlled Interrupts.

To best explain how these service routines work, let's assume an interrupt occurred on IR1 of the 8259A. The associated service routine for IR1 is "INTR73". Entering "INTR73", the first thing done is saving the pre-interrupt program status. This isn't really necessary in this program since a "dummy" main program is being executed; however, it is done as an example to show the operation. Rather than having code for saving the registers in each separate routine, a mutual call routine, "SAVE", is used. This routine will save the register status by pushing it on the stack. The next portion of "INTR73" will display the number of its IR level, "1", in the first digit of the SDK-86 display panel. After that, a software INT instruction is executed to vector program execution to the "INTR77" service routine. The "INTR77" service routine simply sets the 8253 #1 counter 1 to cause an interrupt on IR0 in 250 ms and then returns to "INTR73". Once back in "INTR73", the pre-interrupt status is restored by a call routine, "RESTORE". It does the opposite of "SAVE", returning the register status by popping it off the stack. The "INTR73" routine then returns to the "dummy " main program. The flow for the "INTR74" through "INTR76" routines are the same except for the digit location and the IR level displayed.

After 250 ms have elapsed, counter 1 of 8253 #1 makes an interrupt request on IR0 of the 8259A. This causes the "INTR72" service routine to be executed. Since this routine interrupts the main program, it also uses the "SAVE" routine to save pre-interrupt program status. It then turns off the digit displaying the IR level. In the case of the "INTR73" routine, the "1" is blanked out. The pre-interrupt status is then restored using the "RESTORE" routine and program execution returns to the "dummy" main program.

The complete program for the timer controlled interrupts application is shown in Appendix B. The program was executed in SDK-86 RAM starting at location 0500H (code segment = 0050, instruction pointer = 0).

CONCLUSION

This application note has explained the 8259A in detail and gives three applications illustrating the use of some of the numerous programmable features available. It should be evident from these discussions that the 8259A is an extremely flexible and easily programmable member of the Intel® MCS-80, MCS-85, MCS-86, and MCS-88 families.

This table is provided merely for reference information between the "Operation of the 8259A" and "Programming the 8259A" sections of this application note. It shouldn't be used as a programming reference guide (see "Programming the 8259A").

Operational Description	Command Words	Bits
MCS-80/85™ Mode	ICW1, ICW4*	IC4, μPM*
Address Interval for MCS-80/85 Mode	ICW1	ADI
Interrupt Vector Address for MCS-80/85 Mode	ICW1, ICW2	A5–A15
MCS-86/88 Mode	ICW1, ICW4	IC4, μPM
Interrupt Vector Byte for MCS-86/88 Mode	ICW2	T3–T7
Fully Nested Mode	OCW-Default	—
Non-Specific EOI Command	OCW2	EOI
Specific EOI Command	OCW2	SEOI, EOI, LO–L2
Automatic EOI Mode	ICW1, ICW4	IC4, AEOI
Rotate On Non-Specific EOI Command	OCW2	EOI
Rotate In Automatic EOI Mode	OCW2	R, SEOI, EOI
Set Priority Command	OCW2	L0–L2
Rotate on Specific EOI Command	OCW2	R, SEOI, EOI
Interrupt Mask Register	OCW1	M0–M7
Special Mask Mode	OCW3	ESMM-SMM
Level Triggered Mode	ICW1	LTIM
Edge Triggered Mode	ICW1	LTIM
Read Register Command, IRR	OCW3	ERIS, RIS
Read Register Command, ISR	OCW3	ERIS, RIS
Read IMR	OCW1	M0–M7
Poll Command	OCW3	P
Cascade Mode	ICW1, ICW3	SNGL, S0-7, ID0-2
Special Fully Nested Mode	ICW1, ICW4	IC4, SFNM
Buffered Mode	ICW1, ICW4	IC4, BUF, M/S

*Only needed if ICW4 is used for purposes other than μP mode set.

```
MCS-86 ASSEMBLER    TCI59A


ISIS-II MCS-86 ASSEMBLER V1.0 ASSEMBLY OF MODULE TCI59A
OBJECT MODULE PLACED IN :F1:TCI59A.OBJ
ASSEMBLER INVOKED BY: :F1:ASM86 :F1:TCI59A.SRC

LOC  OBJ                LINE   SOURCE

                           1   ;******************* TIMER CONTROLLED INTERRUPTS ******************
                           2   ;
                           3   ;
                           4   ;
                           5   ;               EXTRA SEGMENT DECLARATIONS
                           6   ;
----                       7   EXTRA SEGMENT
                           8   ;
0120                       9           ORG     120H
0120 0401                 10   TP72IP  DW      INTR72          ;TYPE 72 INSTRUCTION POINTER
0122 ????                 11   TP72CS  DW      ?               ;TYPE 72 CODE SEGMENT
0124 1801                 12   TP73IP  DW      INTR73          ;TYPE 73 INSTRUCTION POINTER
0126 ????                 13   TP73CS  DW      ?               ;TYPE 73 CODE SEGMENT
0128 3001                 14   TP74IP  DW      INTR74          ;TYPE 74 INSTRUCTION POINTER
012A ????                 15   TP74CS  DW      ?               ;TYPE 74 CODE SEGMENT
012C 4801                 16   TP75IP  DW      INTR75          ;TYPE 75 INSTRUCTION POINTER
012E ????                 17   TP75CS  DW      ?               ;TYPE 75 CODE SEGMENT
0130 6001                 18   TP76IP  DW      INTR76          ;TYPE 76 INSTRUCTION POINTER
0132 ????                 19   TP76CS  DW      ?               ;TYPE 76 CODE SEGMENT
0134 7801                 20   TP77IP  DW      INTR77          ;TYPE 77 INSTRUCTION POINTER
0136 ????                 21   TP77CS  DW      ?               ;TYPE 77 CODE SEGMENT
                          22   ;
----                      23   EXTRA ENDS
                          24   ;
                          25   ;               DATA SEGMENT DECLARATIONS
                          26   ;
----                      27   DATA    SEGMENT
                          28   ;
0000 ????                 29   STACK1  DW      ?                       ;VARIABLE TO SAVE CALL ADDRESS
0002 ????                 30   AXTEMP  DW      ?                       ;VARIABLE TO SAVE AX REGISTER
0004 ??                   31   DIGIT   DB      ?                       ;VARIABLE TO SAVE SELECTED DIGIT
                          32   ;
----                      33   DATA    ENDS
                          34   ;
                          35   ;               CODE SEGMENT DECLARATION
                          36   ;
----                      37   CODE    SEGMENT
                          38   ;
                          39   ASSUME  ES:EXTRA,DS:DATA,CS:CODE
                          40   ;
                          41   ;               INITIALIZE REGISTERS
                          42   ;
0000 B80000               43   START:  MOV     AX,0H                   ;EXTRA SEGMENT AT 0H
0003 8EC0                 44           MOV     ES,AX
0005 B87000               45           MOV     AX,70H                  ;DATA SEGMENT AT 700H
0008 8ED8                 46           MOV     DS,AX
000A B87800               47           MOV     AX,78H                  ;STACK SEGMENT AT 780H
000D 8ED0                 48           MOV     SS,AX
000F BC8000               49           MOV     SP,80H                  ;STACK POINTER AT 80H (STACK=800H)
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ                LINE  SOURCE

                          50  ;
                          51  ;               LOAD INTERRUPT VECTOR TABLE
                          52  ;
0012 B80401               53  TYPES:  MOV     AX,OFFSET (INTR72)      ;LOAD TYPE 72
0015 26A32001             54          MOV     TP72IP,AX
0019 268C0E2201           55          MOV     TP72CS,CS
001E B81801               56          MOV     AX,OFFSET (INTR73)      ;LOAD TYPE 73
0021 26A32401             57          MOV     TP73IP,AX
0025 268C0E2601           58          MOV     TP73CS,CS
002A B83001               59          MOV     AX,OFFSET (INTR74)      ;LOAD TYPE 74
002D 26A32801             60          MOV     TP74IP,AX
0031 268C0E2A01           61          MOV     TP74CS,CS
0036 B84801               62          MOV     AX,OFFSET (INTR75)      ;LOAD TYPE 75
0039 26A32C01             63          MOV     TP75IP,AX
003D 268C0E2E01           64          MOV     TP75CS,CS
0042 B86001               65          MOV     AX,OFFSET (INTR76)      ;LOAD TYPE 76
0045 26A33001             66          MOV     TP76IP,AX
0049 268C0E3201           67          MOV     TP76CS,CS
004E B87801               68          MOV     AX,OFFSET (INTR77)      ;LOAD TYPE 77
0051 26A33401             69          MOV     TP77IP,AX
0055 268C0E3601           70          MOV     TP77CS,CS
                          71  ;
                          72  ;               8253 INITIALIZATION
                          73  ;
005A BA0EFF               74  SET531: MOV     DX,0FF0EH               ;8253 #1 CONTROL WORD
005D B036                 75          MOV     AL,36H                  ;COUNTER 0, MODE 3, BINARY
005F EE                   76          OUT     DX,AL
0060 B071                 77          MOV     AL,71H                  ;COUNTER 1, MODE 0, BCD
0062 EE                   78          OUT     DX,AL
0063 B0B5                 79          MOV     AL,0B5H                 ;COUNTER 2, MODE 2, BCD
0065 EE                   80          OUT     DX,AL
0066 BA08FF               81          MOV     DX,0FF08H               ;LOAD COUNTER 0 (10MS)
0069 B0A8                 82          MOV     AL,0A8H                 ;LSB
006B EE                   83          OUT     DX,AL
006C B061                 84          MOV     AL,61H                  ;MSB
006E EE                   85          OUT     DX,AL
006F BA0CFF               86          MOV     DX,0FF0CH               ;LOAD COUNTER 2 (2SEC)
0072 B000                 87          MOV     AL,00H                  ;LSB
0074 EE                   88          OUT     DX,AL
0075 B002                 89          MOV     AL,02H                  ;MSB
0077 EE                   90          OUT     DX,AL
0078 BA16FF               91  SET532: MOV     DX,0FF16H               ;8253 #2 CONTROL WORD
007B B03B                 92          MOV     AL,3BH                  ;COUNTER 0, MODE 5,BCD
007D EE                   93          OUT     DX,AL
007E B07B                 94          MOV     AL,7BH                  ;COUNTER 1, MODE 5, BCD
0080 EE                   95          OUT     DX,AL
0081 B0BB                 96          MOV     AL,0BBH                 ;COUNTER 2, MODE 5, BCD
0083 EE                   97          OUT     DX,AL
0084 BA10FF               98          MOV     DX,0FF10H               ;LOAD COUNTER 0 (.5SEC)
0087 B050                 99          MOV     AL,50H                  ;LSB
0089 EE                  100          OUT     DX,AL
008A B000                101          MOV     AL,00H                  ;MSB
008C EE                  102          OUT     DX,AL
008D BA12FF              103          MOV     DX,0FF12H               ;LOAD COUNTER 1 (1SEC)
0090 B000                104          MOV     AL,00H                  ;LSB
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ              LINE   SOURCE

0092 EE                105          OUT     DX,AL
0093 B001              106          MOV     AL,01H                  ;MSB
0095 EE                107          OUT     DX,AL
0096 BA14FF            108          MOV     DX,0FF14H               ;LOAD COUNTER 2 (1.5SEC)
0099 B050              109          MOV     AL,50H                  ;LSB
009B EE                110          OUT     DX,AL
009C B001              111          MOV     AL,01H                  ;MSB
009E EE                112          OUT     DX,AL
                       113   ;
                       114   ;              8259A INITIALIZATION
                       115   ;
009F BA00FF            116   SET59A: MOV    DX,0FF00H               ;8259A A0=0
00A2 B013              117          MOV     AL,13H                  ;ICW1-LTIM=0,S=1,IC4=1
00A4 EE                118          OUT     DX,AL
00A5 BA02FF            119          MOV     DX,0FF02H               ;8259A A0=1
00A8 B048              120          MOV     AL, 48H                 ;ICW2-INTERRUPT TYPE 72 (120H)
00AA EE                121          OUT     DX,AL
00AB B003              122          MOV     AL, 03H                 ;ICW4-SFNM=0,BUF=0,AEOI=1,MPM=1
00AD EE                123          OUT     DX,AL
00AE B0E0              124          MOV     AL,0E0H                 ;OCW1-MASK IR5,6,7 (NOT USED)
00B0 EE                125          OUT     DX,AL
                       126   ;
                       127   ;              8279 INITIALIZATION
                       128   ;
00B1 BAEAFF            129   SET79:  MOV    DX,0FFEAH               ;8279 COMMAND WORDS AND STATUS
00B4 B0D0              130          MOV     AL,0D0H                 ;CLEAR DISPLAY
00B6 EE                131          OUT     DX,AL
00B7 EC                132   WAIT79: IN     AL,DX                   ;READ STATUS
00B8 D0C0              133          ROL     AL,1                    ;"DU" BIT TO CARRY
00BA 72FB              134          JB      WAIT79                  ;JUMP IF DISPLAY IS UNAVAILABLE
00BC B087              135          MOV     AL, 87H                 ;DIGIT 8
00BE EE                136          OUT     DX,AL
00BF BAE8FF            137          MOV     DX,0FFE8H               ;8279 DATA WORD
00C2 B006              138          MOV     AL,06H                  ;CHARACTER "1"
00C4 EE                139          OUT     DX,AL
00C5 BAEAFF            140          MOV     DX,0FFEAH               ;8279 COMMAND WORD
00C8 B086              141          MOV     AL,86H                  ;DIGIT 7
00CA EE                142          OUT     DX,AL
00CB BAE8FF            143          MOV     DX,0FFE8H               ;8279 DATA WORD
00CE B050              144          MOV     AL,50H                  ;CHARACTER "R"
00D0 EE                145          OUT     DX,AL
00D1 FB                146          STI                             ;ENABLE INTERRUPTS
                       147   ;
                       148   ;
                       149   ;              DUMMY PROGRAM
                       150   ;
00D2 EBFE              151   DUMMY:  JMP    DUMMY                   ;WAIT FOR INTERRUPT
                       152   ;
                       153   ;
00D4 A30200            154   SAVE:   MOV    AXTEMP,AX               ;SAVE AX
00D7 58                155          POP     AX                      ;POP CALL RETURN ADDRESS
00D8 A30000            156          MOV     STACK1,AX               ;SAVE CALL RETURN ADDRESS
00DB A10200            157          MOV     AX,AXTEMP               ;RESTORE AX
00DE 50                158          PUSH    AX                      ;SAVE PROCESSOR STATUS
00DF 53                159          PUSH    BX
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ              LINE   SOURCE

00E0 51                160           PUSH   CX
00E1 52                161           PUSH   DX
00E2 55                162           PUSH   BP
00E3 56                163           PUSH   SI
00E4 57                164           PUSH   DI
00E5 1E                165           PUSH   DS
00E6 06                166           PUSH   ES
00E7 A10000            167           MOV    AX,STACK1                ;RESTORE CALL RETURN ADDRESS
00EA 50                168           PUSH   AX                       ;PUSH CALL RETURN ADDRESS
00EB C3                169           RET
                       170    ;
00EC 58                171    RESTOR: POP   AX                       ;POP CALL RETURN ADDRESS
00ED A30000            172           MOV    STACK1,AX                ;SAVE CALL RETURN ADDRESS
00F0 07                173           POP    ES                       ;RESTORE PROCESSOR STATUS
00F1 1F                174           POP    DS
00F2 5F                175           POP    DI
00F3 5E                176           POP    SI
00F4 5D                177           POP    BP
00F5 5A                178           POP    DX
00F6 59                179           POP    CX
00F7 5B                180           POP    BX
00F8 58                181           POP    AX
00F9 A30200            182           MOV    AXTEMP,AX                ;SAVE AX
00FC A10000            183           MOV    AX,STACK1                ;RESTORE CALL RETURN ADDRESS
00FF 50                184           PUSH   AX                       ;PUSH CALL RETURN ADDRESS
0100 A10200            185           MOV    AX,AXTEMP                ;RESTORE AX
0103 C3                186           RET
                       187    ;
                       188    ;
                       189    ;              INTERRUPT 72, CLEAR DISPLAY,IR0 8259A
                       190    ;
0104 E8CDFF            191    INTR72: CALL  SAVE                     ;ROUTINE TO SAVE PROCESSOR STATUS
0107 BAEAFF            192           MOV    DX,0FFEAH                ;8279 COMMAND WORD
010A A00400            193           MOV    AL,DIGIT                 ;SELECTED LED DIGIT
010D EE                194           OUT    DX,AL
010E BAE8FF            195           MOV    DX,0FFE8H                ;8279 DATA
0111 B000              196           MOV    AL,00H                   ;BLANK OUT DIGIT
0113 EE                197           OUT    DX,AL
0114 E8D5FF            198           CALL   RESTOR                   ;ROUTINE TO RESTORE PROCESSOR STATUS
0117 CF                199           IRET                            ;RETURN FROM INTERRUPT
                       200    ;
                       201    ;
                       202    ;              INTERRUPT 73, IR1 8259A
                       203    ;
0118 E8B9FF            204    INTR73: CALL  SAVE                     ;ROUTINE TO SAVE PROCESSOR STATUS
011B BAEAFF            205           MOV    DX,0FFEAH                ;8279 COMMAND WORD
011E B080              206           MOV    AL,80H                   ;LED DISPLAY DIGIT 1
0120 A20400            207           MOV    DIGIT,AL
0123 EE                208           OUT    DX,AL
0124 BAE8FF            209           MOV    DX,0FFE8H                ;8279 DATA
0127 B006              210           MOV    AL,06H                   ;CHARACTER "1"
0129 EE                211           OUT    DX,AL
012A CD4D              212           INT    77                       ;TIMER DELAY FOR LED ON TIME
012C E8BDFF            213           CALL   RESTOR                   ;ROUTINE TO RESTORE PROCESSOR STATUS
012F CF                214           IRET                            ;RETURN FROM INTERRUPT
```

```
MCS-86 ASSEMBLER    TCI59A

LOC  OBJ             LINE  SOURCE

                           ;
                           ;
                           ;                 INTERRUPT 74, IR2 8259A
                           ;
0130 E8A1FF                INTR74: CALL    SAVE                   ;ROUTINE TO SAVE PROCESSOR STATUS
0133 BAEAFF                        MOV     DX,0FFEAH              ;8279 COMMAND WORD
0136 B081                          MOV     AL,81H                 ;LED DISPLAY DIGIT 2
0138 A20400                        MOV     DIGIT,AL
013B EE                            OUT     DX,AL
013C BAE8FF                        MOV     DX,0FFE8H              ;8279 DATA
013F B05B                          MOV     AL,5BH                 ;CHARACTER "2"
0141 EE                            OUT     DX,AL
0142 CD4D                          INT     77                     ;TIMER DELAY FOR LED ON TIME
0144 E8A5FF                        CALL    RESTOR                 ;ROUTINE TO RESTORE PROCESSOR STATUS
0147 CF                            IRET                           ;RETURN FROM INTERRUPT
                           ;
                           ;
                           ;                 INTERRUPT 75, IR3 8259A
                           ;
0148 E889FF                INTR75: CALL    SAVE                   ;ROUTINE TO SAVE PROCESSOR STATUS
014B BAEAFF                        MOV     DX,0FFEAH              ;8279 COMMAND WORD
014E B082                          MOV     AL,82H                 ;LED DISPLAY DIGIT 3
0150 A20400                        MOV     DIGIT,AL
0153 EE                            OUT     DX,AL
0154 BAE8FF                        MOV     DX,0FFE8H              ;8279 DATA
0157 B04F                          MOV     AL,4FH                 ;CHARACTER "3"
0159 EE                            OUT     DX,AL
015A CD4D                          INT     77                     ;TIMER DELAY FOR LED ON TIME
015C E88DFF                        CALL    RESTOR                 ;ROUTINE TO RESTORE PROCESSOR STATUS
015F CF                            IRET                           ;RETURN FROM INTERRUPT
                           ;
                           ;
                           ;                 INTERRUPT 76, IR4 8259A
                           ;
0160 E871FF                INTR76: CALL    SAVE                   ;ROUTINE TO SAVE PROCESSOR STATUS
0163 BAEAFF                        MOV     DX,0FFEAH              ;8279 COMMAND WORD
0166 B083                          MOV     AL,83H                 ;LED DISPLAY DIGIT 4
0168 A20400                        MOV     DIGIT,AL
016B EE                            OUT     DX,AL
016C BAE8FF                        MOV     DX,0FFE8H              ;8279 DATA
016F B066                          MOV     AL,66H                 ;CHARACTER "4"
0171 EE                            OUT     DX,AL
0172 CD4D                          INT     77                     ;TIMER DELAY FOR LED ON TIME
0174 E875FF                        CALL    RESTOR                 ;ROUTINE TO RESTORE PROCESSOR STATUS
0177 CF                            IRET                           ;RETURN FROM INTERRUPT
                           ;
                           ;
                           ;                 INTERRUPT 77, TIMER DELAY, SOFTWARE CONTROLLED
                           ;
0178 BA0AFF                INTR77: MOV     DX,0FF0AH              ;LOAD COUNTER 1 8253 #1 (250 MSEC)
017B B025                          MOV     AL,25H                 ;LSB
017D EE                            OUT     DX,AL
017E B000                          MOV     AL,00H                 ;MSB
0180 EE                            OUT     DX,AL
0181 CF                            IRET                           ;RETURN FROM INTERRUPT
```

```
MCS-86 ASSEMBLER    TC159A



LOC  OBJ                 LINE   SOURCE

                          270   ;
                          271   ;
----                      272   CODE    ENDS;
                          273   ;
                          274   ;
  0000                    275           END     START

SYMBOL TABLE LISTING
------ ----- -------


NAME    TYPE     VALUE  ATTRIBUTES

??SEG   SEGMENT         SIZE=0000H PARA PUBLIC
AXTEMP  V WORD   0002H  DATA
CODE.   SEGMENT         SIZE=0182H PARA
DATA.   SEGMENT         SIZE=0005H PARA
DIGIT   V BYTE   0004H  DATA
DUMMY   L NEAR   00D2H  CODE
EXTRA   SEGMENT         SIZE=0138H PARA
INTR72  L NEAR   0104H  CODE
INTR73  L NEAR   0118H  CODE
INTR74  L NEAR   0130H  CODE
INTR75  L NEAR   0148H  CODE
INTR76  L NEAR   0160H  CODE
INTR77  L NEAR   0178H  CODE
RESTOR. L NEAR   00ECH  CODE
SAVE.   L NEAR   00D4H  CODE
SET531  L NEAR   005AH  CODE
SET532  L NEAR   0078H  CODE
SET59A  L NEAR   009FH  CODE
SET79   L NEAR   00B1H  CODE
STACK1  V WORD   0000H  DATA
START   L NEAR   0000H  CODE
TP72CS  V WORD   0122H  EXTRA
TP72IP  V WORD   0120H  EXTRA
TP73CS  V WORD   0126H  EXTRA
TP73IP  V WORD   0124H  EXTRA
TP74CS  V WORD   012AH  EXTRA
TP74IP  V WORD   0128H  EXTRA
TP75CS  V WORD   012EH  EXTRA
TP75IP  V WORD   012CH  EXTRA
TP76CS  V WORD   0132H  EXTRA
TP76IP  V WORD   0130H  EXTRA
TP77CS  V WORD   0136H  EXTRA
TP77IP  V WORD   0134H  EXTRA
TYPES   L NEAR   0012H  CODE
WAIT79  L NEAR   00B7H  CODE


ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPLICATION NOTE

AP-28A

January 1979

Intel® MULTIBUS Interfacing

Joe Barthmaier
OEM Microcomputer
Systems Applications

Related Intel Publications

MCS-80™ User's Manual, 98-153D

MCS-85™ User's Manual, 98-366C.

MCS-86™ User's Manual, 9800722A.

iSBC 80/20 and iSBC 80/20-4 Single Board Computer Hardware Reference Manual, 98-317C.

iSBC™ 86/12 Single Board Computer Hardware Reference Manual, 9800645A.

Intel® Multibus™ Specification, 9800683.

Intel® MULTIBUS™ Interfacing

Contents

I. INTRODUCTION

A significant measure of the power and flexibility of the Intel OEM Computer Product Line can be attributed to the design of the Intel MULTIBUS system bus. The bus structure provides a common element for communication between a wide variety of system modules which include: Single Board Computers, memory, digital, and analog I/O expansion boards, and peripheral controllers.

The purpose of this application note is to help you develop a working knowledge of the Intel MULTIBUS specification. This knowledge is essential for configuring a system containing multiple modules. Another purpose is to provide you with the information necessary to design a bus interface for a slave module. One of the tools that will be used to achieve this goal is the complete description of a MULTIBUS slave design example. Other portions of this application note provide an in depth examination of the bus signals, operating characteristics, and bus interface circuits.

This application note was originally written in 1977. Since 1977, the MULTIBUS specification has been significantly expanded to cover operation with both 8 and 16-bit system modules and with an auxiliary power bus. This application note now contains information on these new MULTIBUS specification features.

In addition, a detailed MULTIBUS specification has also been published which provides the user with further information concerning MULTIBUS interfacing. The MULTIBUS specification and other useful documents are listed in the overleaf of this note under Related Intel Publications.

II. MULTIBUS™ SYSTEM BUS DESCRIPTION

Overview

The Intel MULTIBUS signal lines can be grouped in the following categories: 20 address lines, 16 bidirectional data lines, 8 multilevel interrupt lines, and several bus control, timing and power supply lines. The address and data lines are driven by three-state devices, while the interrupt and some other control lines are open-collector driven.

Modules that use the MULTIBUS system bus have a master-slave relationship. A bus master module can drive the command and address lines: it can control the bus. A Single Board Computer is an example of a bus master. A bus slave cannot control the bus. Memory and I/O expansion boards are examples of bus slaves. The MULTIBUS architecture provides for both 8 and 16-bit bus masters and slaves.

Notice that a system may have a number of bus masters. Bus arbitration results when more than one master requests control of the bus at the same time. A bus clock is usually provided by one of the bus masters and may be derived independently from the processor clock. The bus clock provides a timing reference for resolving bus contention among multiple requests from bus masters. For example, a processor and a DMA (direct memory access) module may both request control of the bus. This feature allows different speed masters to share resources on the same bus. Actual transfers via the bus, however, proceed asynchronously with respect to the bus clock. Thus, the transfer speed is dependent on the transmitting and receiving devices only. The bus design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. Once a bus request is granted, single or multiple read/write transfers can proceed. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations and high-speed direct-memory-access (DMA) operations. However, the master-slave capabilities of the bus are by no means limited to these two applications.

MULTIBUS™ Signal Descriptions

This section defines the signal lines that comprise the Intel MULTIBUS system bus. These signals are contained on either the P1 or P2 connector of boards compatible with the MULTIBUS specification. The P1 signal lines contain the address, data, bus control, bus exchange, interrupt and power supply lines. The P2 signal lines contain the optional auxiliary signal lines. Most signals on the bus are active-low. For example, a low level on a control signal on the bus indicates active, while a low level on an address or data signal on the bus represents logic "1" value.

NOTE

> In this application note, a signal will be designated active-low by placing a slash (/) after the mnemonic for the signal.

Appendix A contains a pin assignment list of the following signals:

MULTIBUS P1 Signal Lines —

Initialization Signal Line

INIT/

Initialization signal; resets the entire system to a known internal state. INIT/ may be driven by one of the bus masters or by an external source such as a front panel reset switch.

Address and Inhibit Lines

ADR0/ - ADR13/

20 address lines; used to transmit the address of the memory location or I/O port to be accessed. The lines are labeled ADR0/ through ADR9/, ADRA/ through ADRF/ and ADR10/ through ADR13/. ADR13/ is the most significant bit. 8-bit masters use 16 address lines (ADR0/ - ADRF/) for memory addressing and 8 address lines (ADR0/ - ADR7/) for I/O port selection. 16-bit masters use all twenty address lines for memory addressing and 12 address lines (ADR0/ - ADRB/) for I/O port selection. Thus, 8-bit masters may address 64K bytes of memory and 256 I/O devices while 16-bit masters may address 1 megabyte of memory and 4096 I/O devices. (The 8086 CPU actually permits 16 address bits to be used to specify I/O devices, the MULTIBUS specification, however, states that only the low order 12 address bits can be used to specify I/O ports.) In a 16-bit system, the ADR0/ line is used to indicate whether a low (even) byte or a high (odd) byte of memory or I/O space is being accessed in a word oriented memory or I/O device.

BHEN/

Byte High Enable; the address control line which is used to specify that data will be transferred on the high byte (DAT8/ - DATF/) of the MULTIBUS data lines. With current iSBC boards, this signal effectively specifies that a word (two byte) transfer is to be performed. This signal is used only in systems which incorporate sixteen bit memory or I/O modules.

INH1/

Inhibit RAM signal; prevents RAM memory devices from responding to the memory address on the system address bus. INH1/ effectively allows ROM memory devices to override RAM devices when ROM and RAM memory are assigned the same memory addresses. INH1/ may also be used to allow memory mapped I/O devices to override RAM memory.

INH2/

Inhibit ROM signal; prevents ROM memory devices from responding to the memory address on the system address bus. INH2/ effectively allows auxiliary ROM (e.g., a bootstrap program) to override ROM devices when ROM and auxiliary ROM memory are assigned the same memory addresses. INH2/ may also be used to allow memory mapped I/O devices to override ROM memory.

Data Lines

DAT0/ - DATF/

16 bidirectional data lines; used to transmit or receive information to or from a memory location or I/O port. DATF/ being the most significant bit. In 8-bit systems, only lines DAT0/ - DAT7/ are used (DAT7/ being the most significant bit). In 16-bit systems, either 8 or 16 lines may be used for data transmission.

Bus Priority Resolution Lines

BCLK/

Bus clock; the negative edge (high to low) of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35% to 65% duty cycle. BCLK/ may be slowed, stopped, or single stepped for debugging.

CCLK/

Constant clock; a bus signal which provides a clock signal of constant frequency for unspecified general use by modules on the system bus. CCLK/ has a minimum period of 100 ns and a 35% to 65% duty cycle.

BPRN/

Bus priority in signal; indicates to a particular master module that no higher priority module is requesting use of the system bus. BPRN/ is synchronized with BCLK/. This signal is not bused on the backplane.

BPRO/

Bus priority out signal; used with serial (daisy chain) bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower bus priority. BPRO/ is synchronized with BCLK/. This signal is not bused on the backplane.

BUSY/

Bus busy signal; an open collector line driven by the bus master currently in control to indicate that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is synchronized with BCLK/.

BREQ/

Bus request signal; used with a parallel bus priority network to indicate that a particular master module requires use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. This signal is not bused on the backplane.

CBRQ/

Common bus request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another master wishes to use the bus. If CBRQ/ is high, it indicates to the bus master that no other master is requesting the bus, and therefore, the present bus master can retain the bus. This saves the bus exchange overhead for the current master.

Information Transfer Protocol Lines

A bus master provides separate read/write command signals for memory and I/O devices: MRDC/, MWTC/, IORC/ and IOWC/, as explained below. When a read/write command is active, the address signals must be stabilized at all slaves on the bus. For this reason, the protocol requires that a bus master must issue address signals (and data signals for a write operation) at least 50 ns ahead of issuing a read/write command to the bus, initiating the data transfer. The bus master must keep address signals unchanged until at least 50 ns after the read/write command is turned off, terminating the data transfer.

A bus slave must provide an acknowledge signal to the bus master in response to a read or write command signal.

MRDC/

Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents (8 or 16 bits) of the addressed location are to be read and placed on the system data bus. MRDC/ is asynchronous with respect to BCLK/.

MWTC/

Memory write command; indicates that the address of a memory location has been placed on the system address lines and that data (8 or 16 bits) has been placed on the system data bus. MWTC/ specifies that the data is to be written into the addressed memory location. MWTC/ is asynchronous with respect to BCLK/.

IORC/

I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data (8 or 16 bits) at that input port is to be read and placed on the system data bus. IORC/ is asynchronous with respect to BCLK/.

IOWC/

I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents of the system data bus (8 or 16 bits) are to be output to the address port. IOWC/ is asynchronous with respect to BCLK/.

XACK/

Transfer acknowledge signal; the required response of a slave board which indicates that the specified read/write operation has been completed. That is, data has been placed on, or accepted from, the system data bus lines. XACK/ is asynchronous with respect to BCLK/.

Asynchronous Interrupt Lines

INT0/ - INT7/

8 Multi-level, parallel interrupt request lines;

used with a parallel interrupt resolution network. INT0/ has the highest priority, while INT7/ has lowest priority. Interrupt lines should be driven with open collector drivers.

INTA/

Interrupt acknowledge; an interrupt acknowledge line (INTA/), driven by the bus master, requests the transfer of interrupt information onto the bus from slave priority interrupt controllers (8259s or 8259As). The specific information timed onto the bus depends upon the implementation of the interrupt scheme. In general, the leading edge of INTA/ indicates that the address bus is active while the trailing edge indicates that data is present on the data lines.

MULTIBUS P2 Signal Lines — The signals contained on the MULTIBUS P2 auxiliary connector are used primarily by optional power back-up circuitry for memory protection. P2 signals are not bused on the backplane, and therefore, require a separate connector for each board using the P2 signals. Present iSBC boards have a slot in the card edge and should be used with a keyed P2 edge connector. Use of the P2 signal lines is optional.

ACLO

AC Low; this signal generated by the power supply goes high when the AC line voltage drops below a certain voltage (e.g., 103v AC in 115v AC line voltage systems) indicating D.C. power will fail in 3 msec. ACLO goes low when all D.C. voltages return to approximately 95% of the regulated value. This line must be pulled up by the optional standby power source, if one is used.

PFIN/

Power fail interrupt; this signal interrupts the processor when a power failure occurs, it is driven by external power fail circuitry.

PFSN/

Power fail sense; this line is the output of a latch which indicates that a power failure has occurred. It is reset by PFSR/. The power fail sense latch is part of external power fail circuitry and must be powered by the standby power source.

PFSR/

Power fail sense reset; this line is used to reset the power fail sense latch (PFSN/).

MPRO/

Memory protect; prevents memory operation during period of uncertain DC power, by inhibiting memory requests. MPRO/ is driven by external power fail circuitry.

ALE

Address latch enable; generated by the CPU (8085 or 8086) to provide an auxiliary address latch.

HALT/

Halt; indicates that the master CPU is halted.

AUX RESET/

Auxiliary Reset; this externally generated signal initiates a power-up sequence.

WAIT/

Bus master wait state; this signal indicates that the processor is in a wait state.

Reserved — Several P1 and P2 connector bus pins are unused. However, they should be regarded as reserved for dedicated use in future Intel products.

Power Supplies — The power supply bus pins are detailed in Appendix A which contains the pin assignment of signals on the MULTIBUS backplane.

It is the designer's responsibility to provide adequate bulk decoupling on the board to avoid current surges on the power supply lines. It is also recommended that you provide high frequency decoupling for the logic on your board. Values of 22uF for +5v and +12v pins and 10uF for -5v and -12v pins are typical on iSBC boards.

Operating Characteristics

Beyond the definition of the MULTIBUS signals themselves, it is important to examine the operating characteristics of the bus. The AC requirements outline the timing of the bus signals and in particular, define the relationships between the various bus signals. On the other hand, the DC requirements specify the bus driver characteristics, maximum bus loading per board, and the pull-up/down resistors.

The AC requirements are best presented by a discussion of the relevant timing diagrams. Appendix B contains a list of the MULTIBUS timing specifications. The following sections will discuss data transfers, inhibit operations, interrupt operations, MULTIBUS multi-master operation and power fail considerations.

Data Transfers — Data transfers on the MULTIBUS system bus occur with a maximum bandwidth of 5 MHz for single or multiple read/write transfers. Due to bus arbitration and memory access time, a typical maximum transfer rate is often on the order of 2 MHz.

Read Data

Figure 1 shows the read operation AC timing diagram. The address must be stable (t_{AS}) for a minimum of 50 ns before command (IORC/ or MRDC/). This time is typically used by the bus interface to decode the address and thus provide the required device selects. The device selects establish the data paths on the user system in anticipation of the strobe signal (command) which will follow. The minimum command pulse width is 100 ns. The address must remain stable for at least 50 ns following the command (t_{AH}). Valid data should not be driven onto the bus prior to command, and must not be removed until the command is cleared. The XACK/ signal, which is a response indicating the specified read/write operation has been completed, must coincide or follow both the read access and valid data (t_{DXL}). XACK/ must be held until the command is cleared (t_{XAH}).

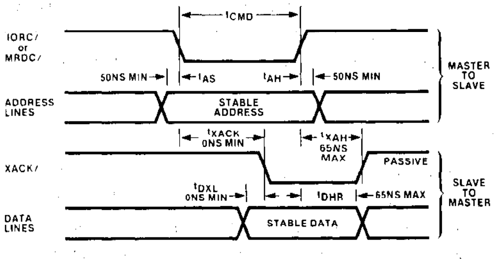


Figure 1. Read AC Timing

Write Data

The write operation AC timing diagram is shown in Figure 2. During a write data transfer, valid data must be presented simultaneously with a stable address. Thus, the write data setup time (t_{DS}) has the same requirement as the address setup time (t_{AS}). The requirement for stable data both before and after command (IOWC/ or MWTC/) enables the bus interface circuitry to latch data on either the leading or trailing edge of command.

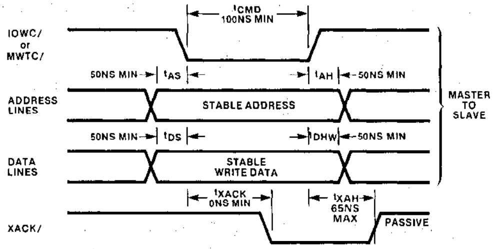


Figure 2. Write AC Timing

Data Byte Swapping in 16-bit Systems

A 16-bit master may transfer data on the MULTIBUS data lines using 8-bit or 16-bit paths depending on whether a byte or word (2 byte) operation has been specified. (A word transfer specified with an odd I/O or memory address will actually be executed as two single byte transfers.) An 8-bit master may only perform byte transfers on the MULTIBUS data lines DAT0/ - DAT7/.

In order to maintain compatibility with older 8-bit masters and slaves, a byte swapping buffer is included in all new 16-bit masters and 16-bit slaves. In the iSBC product line, all byte transfers will take place on the low 8 data lines DAT0/ - DAT7/. Figure 3 contains a example of 8/16-bit data driver logic for 16-bit master and slave systems. In the 8/16-bit system, there are three sets of buffers; the lower byte buffer which accesses DAT0/ - DAT7/, the upper byte buffer which accesses DAT8/ - DATF/, and the swap byte buffer which accesses the MULTIBUS data lines DAT0/ - DAT7/ and transfers the data to/from the on-board data bus lines D8 - DF.

Figure 4 summarizes the 8 and 16-bit data paths used for three types of MULTIBUS transfers. Two signals control the data transfers.

Byte High Enable (BHEN/) active indicates that the bus is operating in sixteen bit mode, and Address Bit 0 (ADR0/) defines an even or odd byte transfer address.

On the first type of transfer, BHEN/ is inactive, and ADR0/ is inactive indicating the transfer of an even eight bit byte. The transfer takes place across data lines DAT0/ - DAT7/.

On the second type of transfer, BHEN/ is inactive, and ADR0/ is active indicating the transfer of a high (odd) byte. On this type of transfer, the odd (high) byte is transferred through the Swap Byte Buffer to DAT0/ - DAT7/. This makes eight bit and sixteen bit systems compatible.

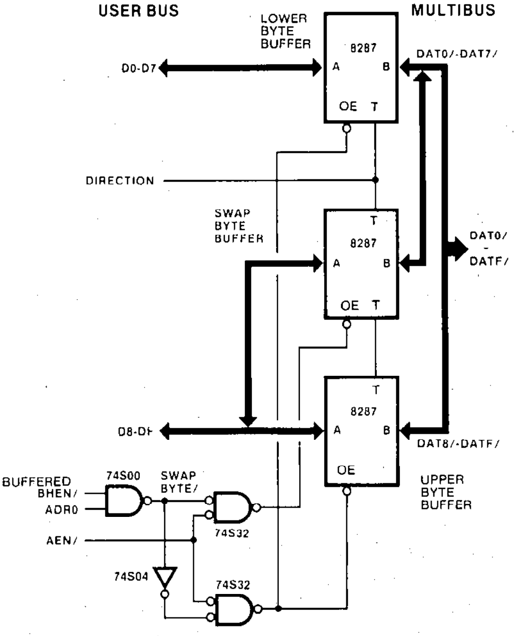


Figure 3. 8/16-Bit Data Drivers

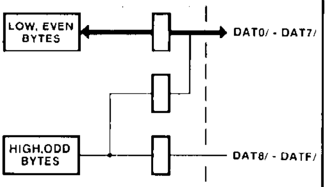
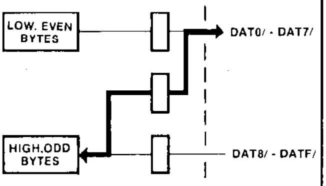
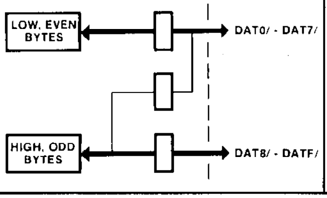
16-BIT DEVICE	MULTIBUS	BHEN/	ADR0/	MULTIBUS TRANSFER DATA PATH	DEVICE BYTE TRANSFERRED
 LOW, EVEN BYTES HIGH,ODD BYTES DAT0/ - DAT7/ DAT8/ - DATF/		H	H	8-BIT, DAT0/ - DAT7/	EVEN
 LOW, EVEN BYTES HIGH,ODD BYTES DAT0/ - DAT7/ DAT8/ - DATF/		H	L	8-BIT, DAT0/ - DAT7/	ODD
 LOW, EVEN BYTES HIGH, ODD BYTES DAT0/ - DAT7/ DAT8/ - DATF/		L	H	16-BIT, DAT0/ - DATF/	EVEN AND ODD

Figure 4. 8/16-Bit Device Transfer Operation

The third type of transfer is a 16 bit (word) transfer. This is indicated by BHEN/ being active, and ADR0/ being inactive. On this type of transfer, the low (even) byte is transferred on DAT0/ - DAT7/ and the high (odd) byte is transferred on DAT8/ - DATF/.

Note that the condition when both BHEN/ and ADR0/ are active is not used with present iSBC boards. This condition could be used to transfer a high odd byte of data on DAT8/ - DATF/, thus eliminating the need for the swap byte buffer. However, this is not a recommended transfer type, because it eliminates the capability of communicating with 8-bit modules.

Inhibit Operations — Bus inhibit operations are required by certain bootstrap and memory mapped I/O configurations. The purpose of the inhibit operation is to allow a combination of RAM, ROM, or memory mapped I/O to occupy the same memory address space. In the case of a bootstrap, it may be desirable to have both ROM and RAM memory occupy the same address space, selecting ROM instead of RAM for low order memory only when the system is reset. A system designed to use memory mapped I/O, which has actual memory occupying the memory mapped I/O address space, may need to inhibit RAM or ROM memory to perform its functions.

There are two essential requirements for a successful inhibit operation. The first is that the inhibit signal must be asserted as soon as possible, within a maximum of 100 ns (t_{CI}), after stable address. The second requirement for a successful inhibit operation is that the acknowledge must be delayed (t_{XACKB}) to allow the inhibited slave to terminate any irreversible timing operations initiated by detection of a valid command prior to its inhibit.

This situation may arise because a command can be asserted within 50 ns after stable address (t_{AS}) and yet inhibit is not required until 100 ns (t_{ID}) after stable address. The acknowledge delay time (t_{XACKB}) is a function of the cycle time of the inhibited slave memory. Inhibiting the iSBC 016 RAM board, for example, requires a minimum of 1.5 usec. Less time is typically needed to inhibit other memory modules. For example, the iSBC 104 board requires 475 ns.

Figure 5 depicts a situation in which both RAM

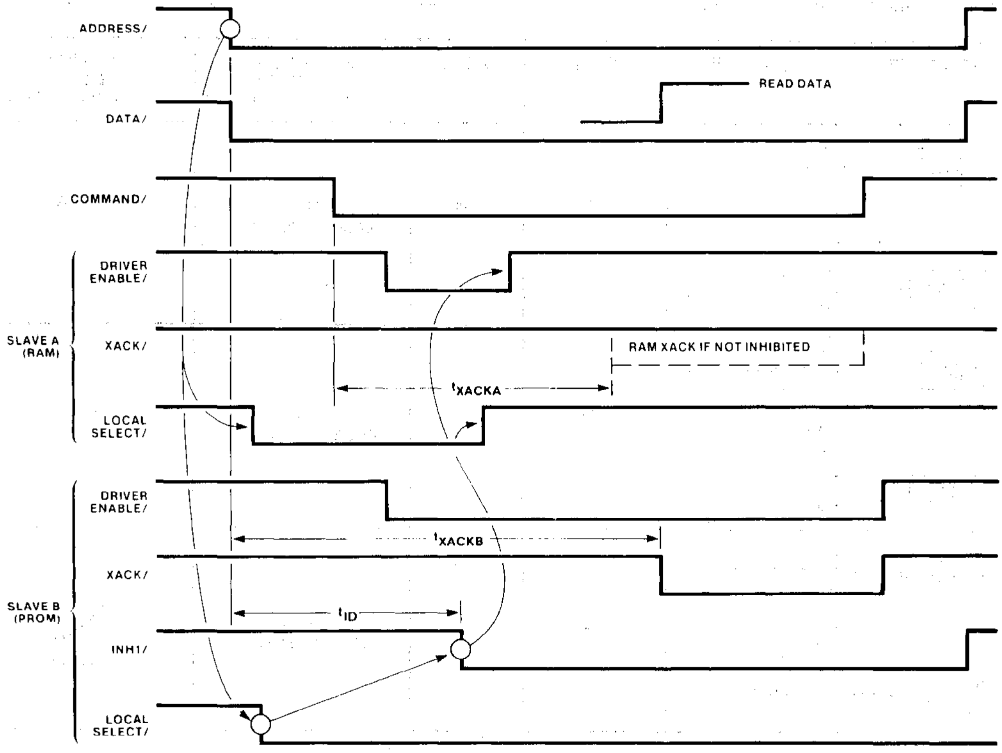


Figure 5. Inhibit Timing

and PROM memory have the same memory addresses. In this case, PROM inhibits RAM, producing the effect of PROM overriding RAM. After address is stable, local selects are generated for both the PROM and the RAM. The PROM local select produces the INH1/ signal which then removes the RAM local select and its driver enable. Because the slave RAM has been inhibited after it had already begun its cycle, the PROM XACK/ must be delayed (t_{XACKB}) until after the latest possible acknowledgement from the RAM (t_{XACKA}).

Interrupt Operations — The MULTIBUS interrupt lines INT0/ - INT7/ are used by a MULTIBUS master to receive interrupts from bus slaves, other bus masters or external logic such as power fail logic. A bus master may also contain internal interrupt sources which do not require the bus interrupt lines to interrupt the master. There are two interrupt implementation schemes used by bus interrupts, Non Bus Vectored Interrupts and Bus Vectored Interrupts. Non Bus Vectored Interrupts do not convey interrupt vector address information on the bus. Bus Vectored Interrupts are interrupts from slave Priority Interrupt Controllers (PICs) which do convey interrupt vector address information on the bus.

Non Bus Vectored Interrupts

Non Bus Vectored Interrupts are those interrupts whose interrupt vector address is generated by the bus master and do not require the MULTIBUS address lines for transfer of the interrupt vector address. The interrupt vector address is generated by the interrupt controller on the master and transferred to the processor over the local bus. The source of the interrupt can be on the master module or on other bus modules, in which case the bus modules use the MULTIBUS interrupt request lines (INT0/ - INT7/) to generate their interrupt requests to the bus master. When an interrupt request line is activated, the bus master performs it own interrupt operation and processes the interrupt. Figure 6 shows an example of Non Bus Vectored Interrupt implementation.

Bus Vectored Interrupts

Bus Vectored Interrupts (Figure 7) are those interrupts which transfer the interrupt vector address along the MULTIBUS address lines from the slave to the bus master using the INTA/ command signal for synchronization.

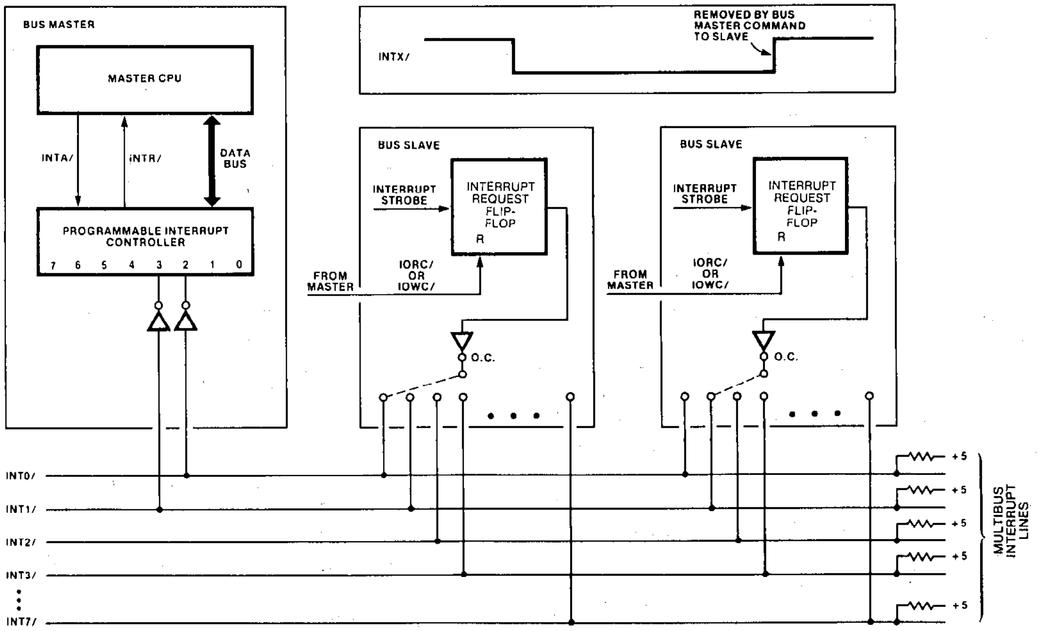


Figure 6. Non Bus Vectored Interrupt Implementation

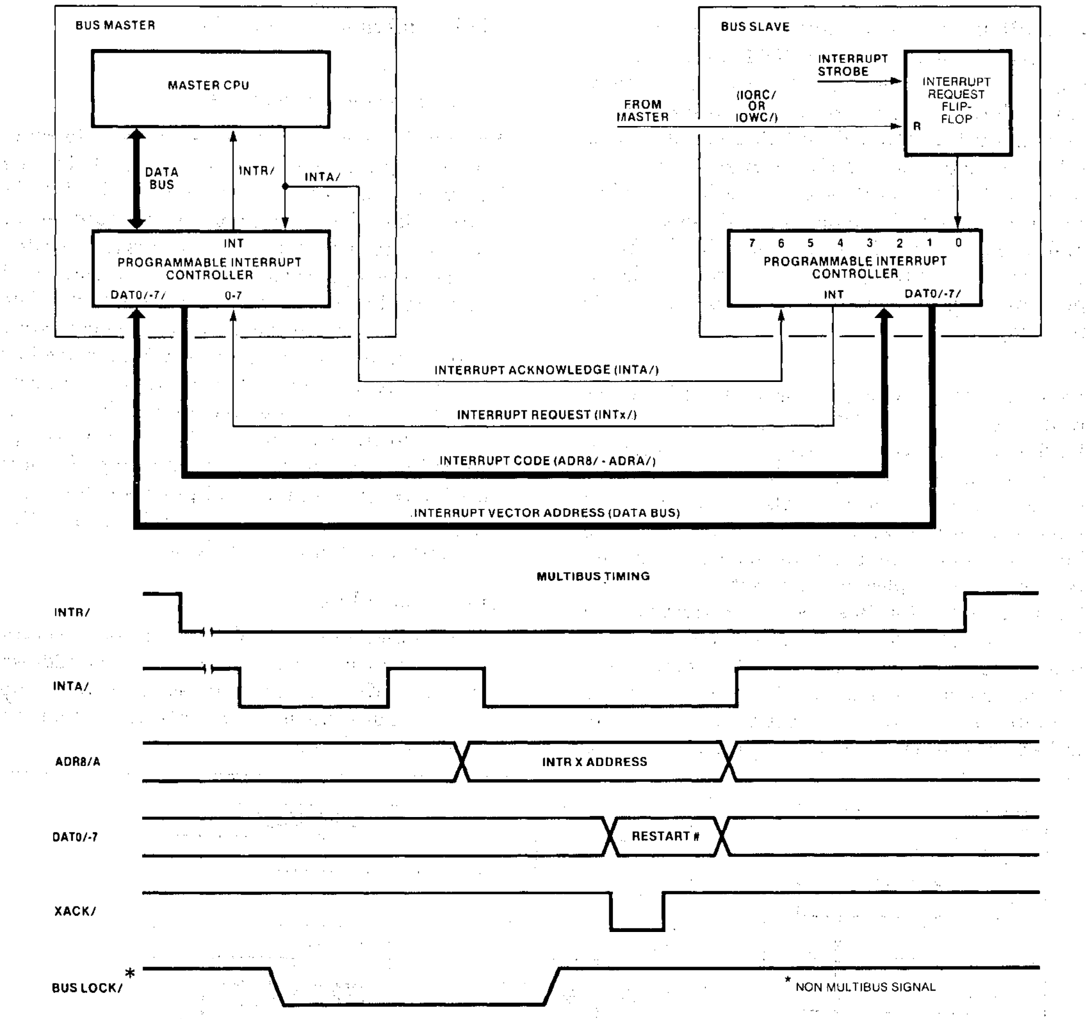


Figure 7. Bus Vectored Interrupt Logic (With 2 INTA/ Timing Diagram)

When an interrupt request from the MULTIBUS interrupt lines INT0/ - INT7/ occurs, the interrupt control logic on the bus master interrupts its processor. The processor on the bus master generates an INTA/ command which freezes the state of the interrupt logic on the MULTIBUS slaves for priority resolution. The bus master also locks (retains the bus between bus cycles) the MULTIBUS control lines to guarantee itself consecutive bus cycles. After the first INTA/ command, the bus master's interrupt control logic puts an interrupt code on to the MULTIBUS address lines ADR8/ - ADRA/. The interrupt code is the address of the highest priority active interrupt request line. At this point in the Bus Vectored Interrupt procedure, two different sequences could take place. The difference occurs, because the MULTIBUS specification can support masters which generate one additional INTA/ (8086 masters) or two additional INTA/s (8080A and 8085 masters).

If the bus master generates one additional INTA/, this second INTA/ causes the bus slave interrupt control logic to transmit an interrupt vector 8-bit pointer on the MULTIBUS data lines. The vector pointer is used by the bus master to determine the memory address of the interrupt service routine.

If the bus master generates two additional INTA/s, these two INTA/ commands allow the bus slave to put a two byte interrupt vector address on to the MULTIBUS data lines (one byte for each INTA/). The interrupt vector address is used by the bus master to service the interrupt.

The MULTIBUS specification provides for only one type of Bus Vectored Interrupt operation in a given system. Slave boards which have an 8259 interrupt controller are only capable of 3 INTA/ operation (2 additional INTA/s after the first INTA/). Slave boards with the 8259A interrupt controller are capable of either 2 INTA/ or 3 INTA/ operation. All slave boards in a given system must operate in the same way (2 INTA/s or 3 INTA/s) if Bus Vectored Interrupts are to be used. However, the MULTIBUS specification does provide for Bus Vectored Interrupts and Non Bus Vectored Interrupts in the same system.

MULTIBUS Multi-Master Operation — The MULTIBUS system bus can accommodate several bus masters on the same system, each one taking control of the bus as it needs to affect data transfers. The bus masters request bus control through a bus exchange sequence.

Two bus exchange priority resolution techniques are discussed, a serial technique and a parallel technique. Figures 8 and 9 illustrate these two techniques. The bus exchange operation discussed later is the same for both techniques.

Serial Priority Technique

Serial priority resolution is accomplished with a daisy chain technique (see Figure 8). The priority input (BPRN/) of the highest priority master is tied to ground. The priority output (BPRO/) of the highest priority master is then connected to the priority input (BPRN/) of the next lower priority master, and so on. Any master generating a bus request will set its BPRO/ signal high to the next lower priority master. Any master seeing a high signal on its BPRN/ line will sets its BPRO/ line high, thus passing down priority information to lower priority masters. In this implementation, the bus request line (BREQ/) is not used outside of the individual masters. A limited number of masters can be accommodated by this technique, due to gate delays through the daisy chain. Using the current Intel MULTIBUS controller chip on the master boards up to 3 masters may be accommodated if a BCLK/ period of 100 ns is used. If more bus masters are required, either BCLK/ must be slowed or a parallel priority technique used.

Parallel Priority Technique

In the parallel priority technique, the priority is resolved in a priority resolution circuit in which the highest priority BREQ/ input is encoded with a priority encoder chip (74148). This coded value is then decoded with a priority decoder chip (74S138) to activate the appropriate BPRN/ line. The BPRO/ lines are not used in the parallel priority scheme. However, since the MULTIBUS backplane contains a trace from the BPRN/ signal of one card slot to the BPRO/ signal of the adjacent lower card slot, the BPRO/ must be disconnected from the bus on the board or the backplane trace must be cut. A practical limit of sixteen masters can be accommodated using the parallel priority technique due to physical bus length limitations. Figure 9 contains the schematic for a typical parallel resolution network. Note that the parallel priority resolution network must be externally supplied.

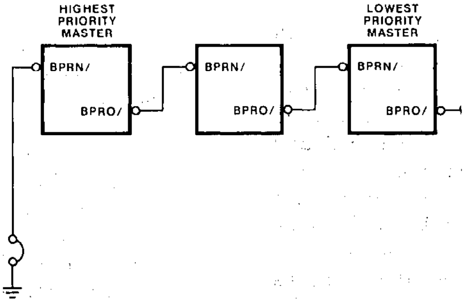


Figure 8. Serial Priority Technique

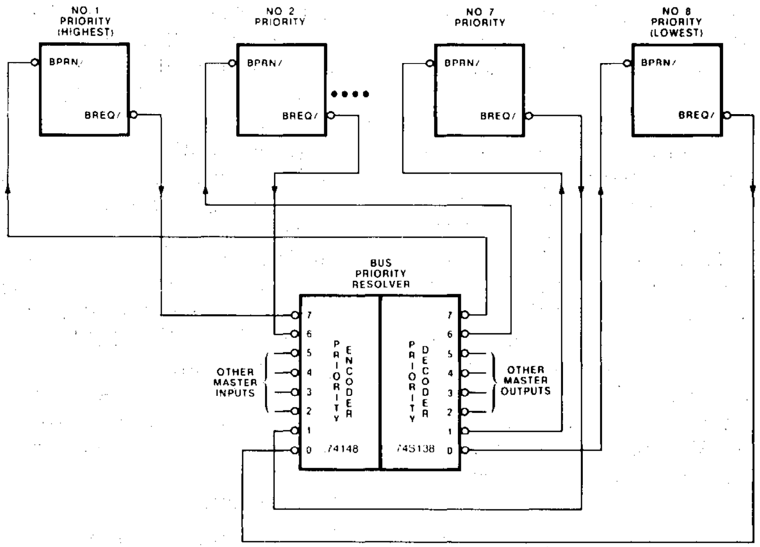


Figure 9. Parallel Priority Technique

MULTIBUS Exchange Operation — A timing diagram for the MULTIBUS exchange operation is shown in Figure 10. This implementation example uses a parallel resolution scheme, however, the timing would be basically the same for the serial resolution scheme.

In this example, master A has been assigned a lower priority than master B. The bus exchange occurs because master B generates a bus request during a time when master A has control of the bus.

The exchange process begins when master B requires the bus to access some resource such as an I/O or memory module while master A controls the bus. This internal request is synchronized with the trailing edge (high to low) of BCLK/ to generate a bus request (BREQ/). The bus priority resolution circuit changes the BPRN/ signal from active (low) to inactive (high) for master A and from inactive to active for master B. Master A must first complete the current bus command if one is in operation. After master A completes the command, it sets BUSY/ inactive on the next trailing edge of BCLK/. This allows the actual bus exchange to occur, because master A has relinquished control of the bus, and master B has been granted its BPRN/. During this time, the drivers for master A are disabled. Master B must take control of the bus with the next trailing edge of BCLK/ to complete the bus exchange. Master B takes control by activating BUSY/ and enabling its drivers.

It is possible for master A to retain control of the bus and prevent master B from getting control. Master A activates the Bus Override (or Bus Lock) signal which keeps BUSY/ active allowing control of the bus to stay with master A. This guarantees a master consecutive bus cycles for software or hardware functions which require exclusive, continuous access to the bus.

Note that in systems with only a single master it is necessary to ground the BPRN/ pin of the master, if slave boards are to be accessed. In single board systems which use a CPU board capable of Bus Vectored Interrupt operation, the BPRN/ pin must also be grounded.

In a single master system bus transfer efficiency may be gained if the BUS OVERRIDE signal is kept active continuously. This permits the master to maintain control of the bus at all times, therefore saving the overhead of the master reacquiring the bus each time it is needed.

The CBRQ/ line may be used by a master in control of the bus to determine if another master

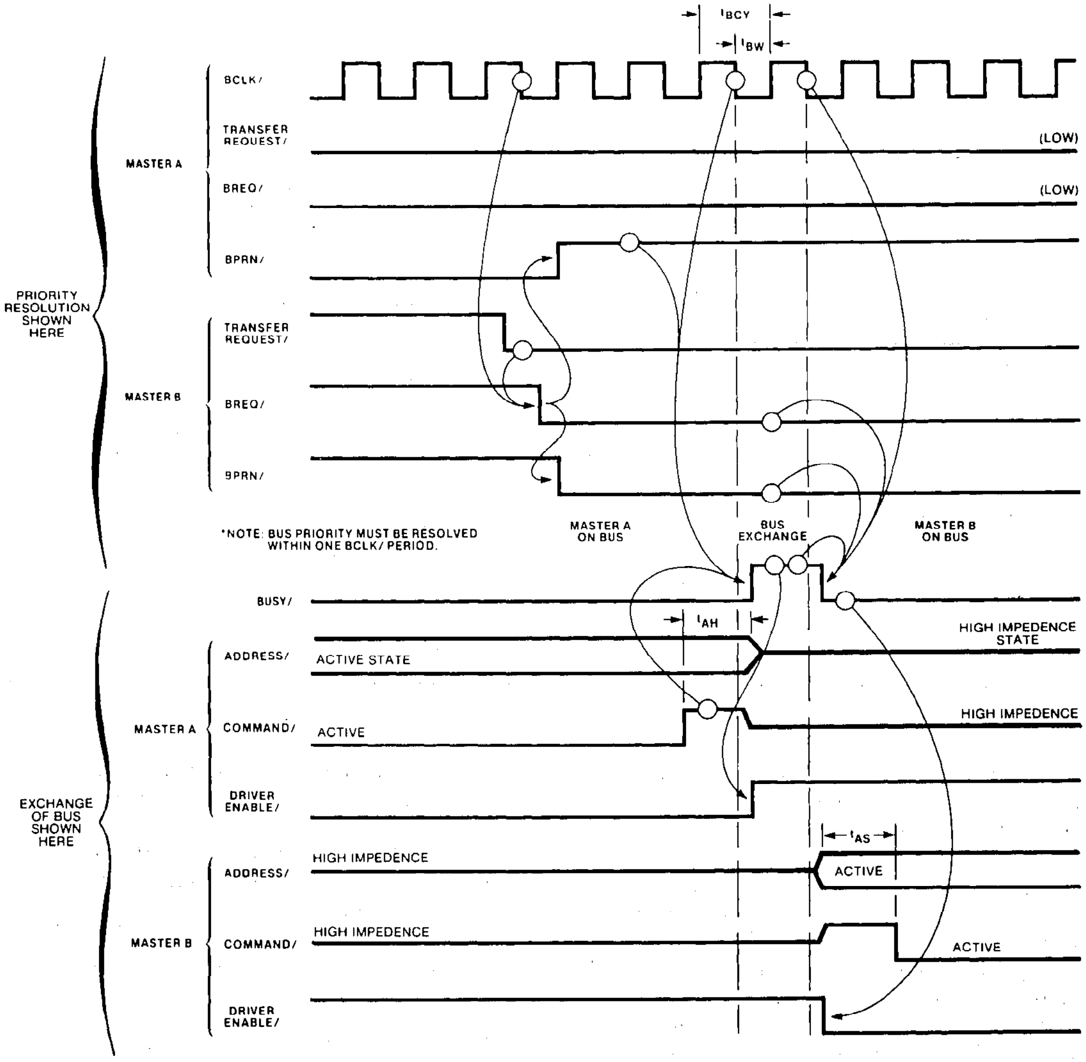


Figure 10. Bus Control Exchange Operation

requires the bus. If a master currently in control of the bus sees the CBRQ/ line inactive, it will maintain control of the bus between adjacent bus accesses. Therefore, when a bus access is required, the master saves the overhead of reacquiring the bus. If a current bus master sees the CBRQ/ line active, it will then relinquish control of the bus after the current bus access and will contend for the bus with the other master(s) requiring the bus. The relative priorities of the masters will determine which master receives the bus.

Note that except for the BUS OVERRIDE state, no single master may keep exclusive control of the bus. This is true because it is impossible for the CPU on a master to require continuous access to the bus. Other lower priority masters will always be able to gain access to the bus between accesses of a higher priority master.

Power Fail Considerations — The MULTIBUS P2 connector signals provide a means of handling power failures. The circuits required for power

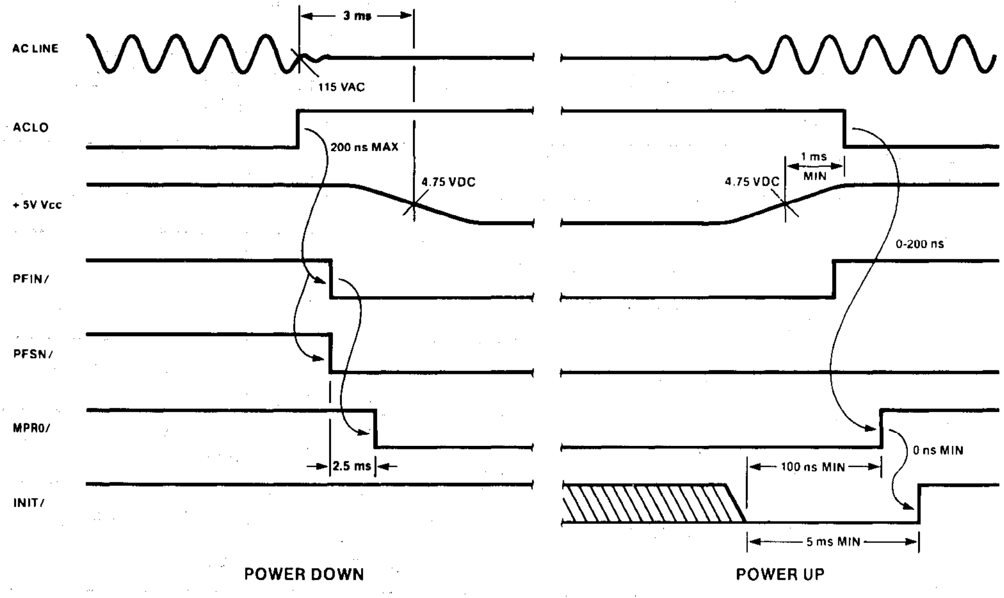


Figure 11. Power Fail Timing Sequence

failure detection and handling are optional and must be supplied by the user. Figure 11 shows the timing of a power fail sequence.

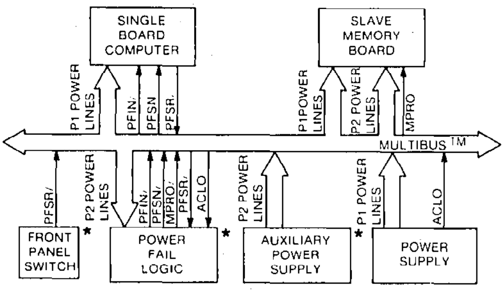
The power supply monitors the AC power level. When power drops below an acceptable value, the power supply raises ACLO which tells the power fail logic that a minimum of three milliseconds will elapse before DC power will fall below regulated voltage levels. The power fail logic sets a sense latch (PFSN/) and generates an interrupt (PFIN/) to the processor so the processor can store its environment. After a 2.5 millisecond timeout, the memory protect signal (MPRO/) is asserted by the power fail logic preventing any memory activity. As power falls, the memory goes on standby power. Note that the power fail logic must be powered from the standby source.

As the AC line revives, the logic voltage level is monitored by the power supply. After power has been at its operating level for one millisecond minimum, the power supply sets the signal ACLO low, beginning the restart sequence. First, the memory protect line (MPRO/) then the initialize line (INIT/) become inactive. The bus master now starts running. The bus master checks the power fail latch (PFSN/) and, if it finds it set, branches to a power up routine which resets the latch (PFSR/), restores the environment, and resumes execution.

Note that INIT/ is activated only after DC power has risen to the regulated voltage levels and must stay low for five milliseconds minimum before the system is allowed to restart. Alternatively, INIT/ may be held low through an open collector device by MPRO/.

How the power failure equipment is configured is left to the system designer. The backup power source may be batteries located on the memory boards or more elaborate facilities located off-board. The location of the power fail logic determines which MULTIBUS power fail lines are used. Pins on the P2 connector have been specified for the power failure functions for use as needed.

To further clarify the location and use of the power fail circuitry, an example of a typical power fail system block diagram is shown in Figure 12. A single board computer and a slave memory board are contained in the system. It is desired to power the memory circuit elements of the memory board from auxiliary power. The single board computer will remain on the main power supply. To accomplish this, user supplied power fail logic and



* USER SUPPLIED

Figure 12. Typical Power Fail System Block Diagram

an auxiliary power supply have been included in the system.

The single board computer is powered from the P1 power lines and accesses the P2 signal lines PFIN/, PFSN/ and PFSR/ (only the P2 signal lines used by a particular functional block are shown on the block diagram). The PFSR/ line is driven from two sources: a front panel switch and the single board computer. The front panel switch is used during normal power-up to reset the power fail sense latch. The single board computer uses the PFSR/ line to reset the latch during a power-up sequence after a power failure. Current single board computers must access the PFSN/ and PFSR/ signals either directly with dedicated circuitry and a P2 pin connection or through the parallel I/O lines with a cable connection from the parallel I/O connector to the P2 connector.

The slave memory board uses both the P1 and P2 power lines, the P2 power lines are used (at all times) to power the memory circuit elements and other support circuits, the P1 power lines power all other circuitry. In addition, the MPRO/ line is input and used to sense when memory contents should be protected.

The power fail logic contains the power fail sense latch, and uses the PFSR/ and ACLO lines for inputs and the PFIN/ PFSN/, and MPRO/ lines for outputs. The power fail logic must be powered by the P2 power lines.

DC Requirements — The drive and load characteristics of the bus signals are listed in Appendix C. The physical locations of the drivers and loads, as well as the terminating resistor value for each bus line, are also specified. Appendix D contains the MULTIBUS power specifications.

MULTIBUS™ Slave Interface Circuit Elements

There are three basic elements of a slave bus interface: address decoders, bus drivers, and control signal logic. This section discusses each of these elements in general terms. A description of a detailed implementation of a slave interface is presented in a later section of this application note.

Address Decoding — This logic decodes the appropriate MULTIBUS address bits into RAM requests, ROM requests, or I/O selects. Care must be taken in the design of the address decode logic to ensure flexibility in the selection of base address assignments. Without this flexibility, restrictions may be placed upon various system configurations. Ideally, switches and jumper connections should be associated with the decode logic to permit field modification of base address assignments.

The initial step in designing the address decode portion of a MULTIBUS interface is to determine the required number of unique address locations. This decision is influenced by the fact that address decoding is usually done in two stages. The first stage decodes the base address, producing an enable for the second stage which generates the actual device selects for the user logic. A convenient implementation of this two stage decoding scheme utilizes a pair of decoders driven by the high order bits of the address for the first stage and a second decoder for the low order bits of the address bus. This technique forces the number of unique address locations to be a power of two, based at the address decoded by the first stage. Consider the scheme illustrated in Figure 13.

As shown in Figure 13, the address bits A_4 - A_B are used to produce switch selected outputs of the first stage of decoding. The 1 out of 8 binary decoders have been used. The top decoder decodes address lines A_4 - A_7, and the bottom decoder decodes address lines A_8 - A_B. If only address lines A_0 - A_7 are being used for device selection, as in the case of I/O port selection in 8-bit systems, the bottom decoder may be disabled by setting switch S2 to the ground position. Address lines A_7 and A_B drive enable inputs E2 or E3 of the decoders. The address lines A_0 - A_3 enter the second stage address decoder to produce 8 user device selects. The second stage decoder must first be enabled by an address that corresponds to the switch-selected base address.

Address decoding must be completed before the arrival of a command. Since the command may become active within 50 ns after stable address, the decode logic should be kept simple with a minimal number of layers of logic. Furthermore, the timing is extremely critical in systems which make use of the inhibit lines.

A linear or unary select scheme in which no binary encoding of device address (e.g., address bit A_0 selects device 0, address bit A_1 selects device 1, etc.) is performed is not recommended because the scheme offers no protection in case multiple devices are simultaneously selected, and because the addressing within such a system is restricted by the extent of the address space occupied by such a scheme.

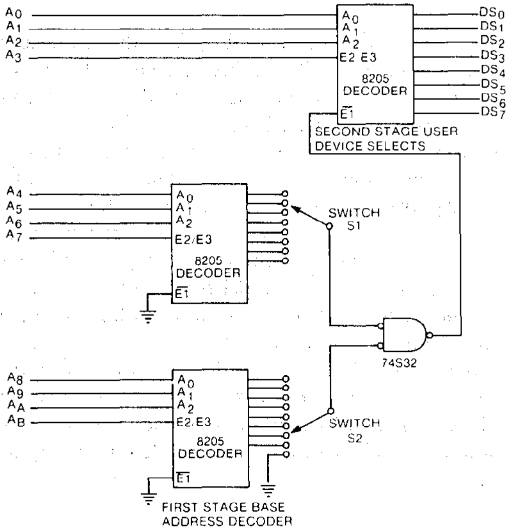


Figure 13. Two Stage Decoding Scheme

Data Bus Drivers — For user designed logic which simply receives data from the MULTIBUS data lines, this portion of the bus interface logic may only consist of buffers. Buffers are required to ensure that maximum allowable bus loading is not exceeded by the user logic.

In systems where the user designed logic must place data onto the MULTIBUS data lines, three-state drivers are required. These drivers should be enabled only when a memory read command (MRDC/) or an I/O read command (IORC/) is present and the module has been addressed.

When both the read and write functions are required, parallel bidirectional bus drivers (e.g., Intel 8226, 8287, etc.) are used. A note of caution must be included for the designer who uses this type of device. A problem may arise if data hold time requirements must be satisfied for user logic following write operations. When bus commands are used to directly produce both the chip select for the bidirectional bus driver and a strobe to a latch in the user logic, removal of that signal may not provide the user's latch with adequate data hold time. Depending on the specifics of the user logic, this problem may be solved by permanently enabling the data buffer's receiver circuits and controlling only the direction of the buffers.

Control Signal Logic — The control signal logic consists of the circuits that forward the I/O and memory read/write commands to their respective destinations, provide the bus with a transfer acknowledge response, and drive the system interrupt lines.

Bus Command Lines

The MULTIBUS information transfer protocol lines (MRDC/, MWTC/, IORD/. and IOWC/) should be buffered by devices with very high speed switching. Because the bus DC requirements specify that each board may load these lines with 2.0 mA, Schottky devices are recommended. LS devices are not recommended due to their poor noise immunity. The commands should be gated with a signal indicating the base address has been decoded to generate read and write strobes for the user logic.

Transfer Acknowledge Generation

The user interface transfer acknowledge generation logic provides a transfer acknowledge response, XACK/, to notify the bus master that write data provided by the bus master has been accepted or that read data it has requested is available on the MULTIBUS data lines. XACK/ allows the bus master to conclude its current instruction.

Since XACK/ timing requirements depend on both the CPU of the bus master and characteristics of the user logic, a circuit is needed which will provide a range of easily modified acknowledge responses.

The transfer acknowledge signals must be driven by three-state drivers which are enabled when the bus interface is addressed and a command is present.

Interrupt Signal Lines

The asynchronous interrupt lines must be driven by open collector devices with a minimum drive of 16 mA.

In a typical Non Bus Vectored Interrupt system, logic must be provided to assert and latch-up an interrupt signal. In addition to driving the MULTIBUS interrupt lines, the latched interrupt signal would be read by an I/O operation such as reading the module's status. The interrupt signal would be cleared by writing to the status register.

III. MULTIBUS™ SLAVE DESIGN EXAMPLE

A MULTIBUS slave design example has been included in this application note to reinforce the theory previously discussed. The design example is of general purpose I/O slave interface. This design example could easily be modified to be used as a slave memory interface by buffering the address signals and using the appropriate MULTIBUS memory commands. In addition, to help the reader better understand an application for an I/O slave interface, two Intel 8255A Parallel Peripheral Interface (PPI) devices are shown connected to the slave interface.

The design example is shown in both 8/16-bit version and an 8-bit version. The 8/16-bit version is an I/O interface which will permit a 16-bit master to perform 8 or 16 bit data transfers. 8-bit masters may also use the 8/16-bit version of the design example to perform 8-bit data transfers.

The 8-bit version of the design example may be used by both 8 or 16-bit masters, but will only perform 8-bit data transfers. It does not contain the circuitry required to perform 16-bit data transfers.

Both the 8/16-bit version and the 8-bit version of the design example were implemented on an iSBC 905 prototype board. The schematics for each of the examples are given in Appendices F and G.

Functional/Programming Characteristics

This section describes the organization of the slave interface from two points of view, the functional point of view and the programming characteristics. First, the principal functions performed by the hardware are identified and the general data flow is illustrated. This point of view is intended as an introduction to the detailed description provided in the next section; Theory of Operation. In the second point of view, the information needed by a programmer to access the slave is summarized.

Functional Description — The function of this I/O slave is to provide the bus interface logic for general purpose I/O functions and for two Intel 8255A Parallel Peripheral Interface (PPI) devices. Eight device selects (port addresses) are available for general purpose I/O functions. One of these device select lines is used to read and reset the state of an interrupt status flip-flop, the other seven device selects are unused in this design. An additional eight I/O device port addresses are used by the two 8255A devices; four I/O port addresses per 8255A (three I/O port address for the three parallel ports A, B, and C and the fourth I/O port address for the device control register).

Figure 14 contains a functional block diagram of the slave design example. This block diagram shows the fundamental circuit elements of a bus slave: bidirectional data bus drivers/receivers, address decoding logic and bus control logic. Also shown is the address decoding logic for the low order four bits, the interrupt logic which is selected by this decoding logic, and the two 8255A devices.

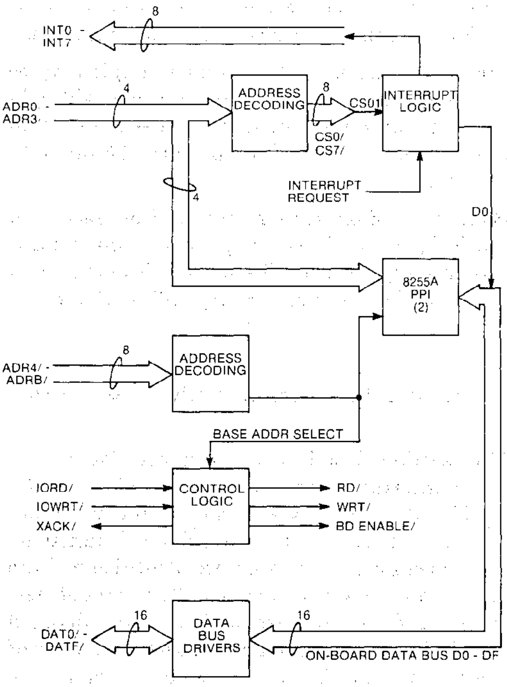


Figure 14. MULTIBUS™ Slave Design Example Functional Block Diagram

Programming Characteristics — The slave design example provides 16 I/O port addresses which may be accessed by user software. The base address of the 16 contiguous port addresses is selected by wire wrap connections on the prototype board. The wire wrap connections specify address bits ADR4/ - ADRB/. They allow the selection of a base address on any 16 byte boundary. Twelve address bits (ADR0/ - ADRB/) are used since 16-bit (8086 based) masters use 12 bits to specity I/O port addresses. If an 8 bit (8080 or 8085 based) master is used with this slave board, the high order address bits (ADR8/ - ADRB/) must not be used by the decoding circuits; a wire wrap jumper position (ground position) is provided for this.

The 16 I/O port addresses are divided into two groups of 8 port addresses by decoding address line ADR3/. Port addresses XX0 - XX7 are used for general I/O functions (XX indicates any hexidecimal digit combination). Port address XX0 is used for accessing the interrupt status flip-flop and port addresses XX1 - XX7 are not used in this example. Port addresses XX8 - XXF are used for accessing the PPIs. If port addresses XX8 - XXF are selected, then ADR0/ is used to specify which of two PPIs are selected. If the address is even (XX8, XXA, XXC, or XXE) then one PPI is selected. If the address is odd (XX9, XXB, XXD, or XXF), then the other PPI is selected. ADR1/ and ADR2/ are connected directly to the PPIs. Table 1 summarizes the I/O port addresses of the slave design example. Note that if a 16-bit master is used, it is possible to access the slave in a byte or word mode. If word access is used with port address XX8, XXA, XXC, or XXE, then 16 bit transfers will occur between the PPIs and the master. These 16 bit transfers occur because an even address has been specified and the MULTIBUS BHEN/ signal indicates that a 16-bit transfer is requested.

Theory of Operation

In the preceding section, each of the slave design example functional blocks was identified and briefly explained. This section explains how these functions are implemented. For detailed circuit information, refer to the schematics in Appendices F and G. The schematic in Appendix F is on a foldout page so that the following text may easily be related to the schematic.

The discussion of the theory of operation is divided into five segments, each of which discusses a different function performed by the MULTIBUS slave design example. The five segments are:

1. Bus address decoding
2. Data buffers
3. Control signals
4. Interrupt logic
5. PPI operation

Each of these topics are discussed with regard to the 8/16-bit version of the design example; followed by a discussion of the circuit elements which are required by the 8-bit version of the interface.

Bus Address Decoding — Bus address decoding is performed by two 8205 1 out of 8 binary decoders. One decoder (A3) decodes address bits ADR8/ - ADRB/ and the second decoder (A2) decodes address bits ADR4/ - ADR7/. The base address

Table 1
SLAVE DESIGN EXAMPLE PORT ADDRESSES

I/O PORT ADDRESS	READ	WRITE
BYTE ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX1 - XX7	Unused	Unused
XX8	Parallel Port A, Even PPI	Parallel Port A, Even PPI
XX9	Parallel Port A, Odd PPI	Parallel Port A, Odd PPI
XXA	Parallel Port B, Even PPI	Parallel Port B, Even PPI
XXB	Parallel Port B, Odd PPI	Parallel Port B, Odd PPI
XXC	Parallel Port C, Even PPI	Parallel Port C, Even PPI
XXD	Parallel Port C, Odd PPI	Parallel Port C, Odd PPI
XXE	Illegal Condition	Control, Even PPI
XXF	Illegal Condition	Control, Odd PPI
WORD ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX2 - XX6	Unused	Unused
XX8	Parallel Port A, Even and Odd PPIs	Parallel Port A, Even and Odd PPIs
XXA	Parallel Port B, Even and Odd PPIs	Parallel Port B, Even and Odd PPIs
XXC	Parallel Port C, Even and Odd PPIs	Parallel Port C, Even and Odd PPIs
XXE	Illegal Condition	Control, Even and Odd PPIs

XX = Any hex digits, assigned by jumpers; XX defines the base address.

selected is determined by the position of wire wrap jumpers. The outputs of the two decoders are ANDed together to form the BASE ADR SELECT/ signal. This signal specifies the base address for a group of 16 I/O ports. Using the wire wrap jumper positions shown in the schematic, a base address of E3 has been selected. Therefore, this MULTIBUS slave board will respond to I/O port addresses in the E30 - E3F range.

If this slave board is to be used with 8-bit MULTIBUS masters, the high order address bits must not be decoded. Therefore, the wire wrap jumper which selects the output of decoder A3 must be placed in the top (ground) position (pin 10 of gate A9 to ground).

The low order 4 address lines (ADR0/ - ADR3/) are buffered and inverted using 74LS04 inverters. These address lines are input to an 8205 for decoding a chip select for the interrupt logic; the address lines are also used directly by the PPIs. LS-Series logic is required for buffering to meet the MULTIBUS specification for I_{IL} (low level input current). S-Series or standard series logic will not meet this specification.

Address decoder A4 is used to decode addresses E30 - E37. The CS0/ output of this decoder is used to select the interrupt logic, thus I/O port address E30 is used to read and reset the interrupt latch. The remaining outputs from decoder A4 (CS1/ - CS7/) are not used in this example. They would normally be used to select other functions in a slave board with more capability. Note that in the schematic shown in Appendix G for the 8-bit version of this slave design example, the high order (ADR8/ - ADRB/) address decoder is not included and the BHEN/ signal is not used.

Data Buffers — Intel 8287 8-bit parallel bidirectional bus drivers are used for the MULTIBUS data lines DAT0/ - DATF/. In the 8/16-bit version of the slave board, three 8287 drivers are used.

When an 8-bit data transfer is requested, either driver A5, which is connected to on-board data lines D0 - D7, or driver A6, which is connected to on-board data lines D8 - DF, is used. If a byte transfer is requested from an even address, driver A5 will be selected. If a byte transfer from an odd address is requested, driver A6 will be selected. All byte transfers take place on MULTIBUS data lines DAT0/ - DAT7/. When a word (16-bit) transfer is requested from an even address, drivers A5 and A7 will be used. Note that if a user program requests a word transfer from an odd address, 16-bit masters in the iSBC product line will actually perform two byte transfer requests.

The logic which determines the chip selection (8287 input signal OE, output enable) signals for the bus drivers uses the low order address bit (ADR0/) and the buffered Byte High Enable signal (BHENBL/). Note that the MULTIBUS signal BHEN/ has been buffered with an 74LS04 inverter. This is done to meet the bus address line loading specification. The SWAP BYTE/ signal which is generated is qualified by the BD ENBL/ signal and used to select the bus drivers.

The steering pin for the 8287 drivers is labelled T (transmit) and is driven by the signal RD. When an input (read) request is active or when neither a read or write command is being serviced, the direction of data transfer of the 8287 will be set for B to A.

The 8287 drivers are set to point IN (direction B to A) when no MULTIBUS I/O transfer command is being serviced for two reasons. First, if the driver were pointed OUT (direction A to B) and a write command occured, it would be necessary to turn the buffers IN and set the OE (output enable) signal active before the data could be transferred to the on-board bus. A possibility of a "buffer-fight" could occur in some designs if the OE signal permitted an 8287 to drive the MULTIBUS data lines momentarily before the steering signal could switch the direction of the 8287. In this case, both the MULTIBUS master and the slave would be driving the data lines; this is not recommended. (In this particular design, the steering signal will always stabilize before the OE signal becomes active.)

The second reason the driver is pointing IN when no command is present is due to the "data valid after WRITE" requirements of the 8255As. The 8255A requires that data remain on its data lines for 30 ns after the WRITE command ($\overline{WR}$ at the 8255A) is removed. This requirement will be met if the direction of the 8287 drivers is not switched when the MULTIBUS IOWC/ signal is removed (WRT/ could have been used to steer the 8287 instead of RD); and if the capacitance of the on-board data bus lines is sufficient to hold the data values on the bus after the 8287 OE signal and the 8255A PPI WRT/ signal go inactive. The on-board data bus may easily be designed such that the capacitance of the lines is sufficient to meet the 30 ns data hold time requirement. In addition, the current leakage of all devices connected to the on-board bus must be kept small to meet the 30 ns data hold time requirement.

The 8-bit version of this design example uses only one 8287 instead of the three required by the 8/16-bit version. The logic required to control the swap byte buffer is also not necessary. The chip select signal used for the 8287 is the BD ENBL/ signal.

Control Signals — The MULTIBUS control signals used by this slave design example are IORC/, IOWC/, and XACK/. IORC/ and IOWC/ are qualified by the BASE ADR SELECT/ signal to form the signals RD and WRT. RD and WRT are used to drive the interrupt logic, the PPI logic and the XACK/ (transfer acknowledge) logic.

For the XACK/ logic RD and WRT are ORed to form the BD ENBL/ signal which is inverted and used to drive the CLEAR pin of a shift register. When the slave board is not being accessed, the CLEAR pin of the shift register will be low (BD ENBL/ is high). This causes the shift register to remain cleared and all outputs of the shift register will be low. When the slave board is accessed, the CLEAR pin will be high, and the A and B inputs (which are high) will be clocked to the output pins by CCLK/. To select a delay for the XACK/ signal, a jumper must be installed from one of the shift register output pins to the 8089 tri-state driver. Each of the shift register output pins select an integer multiple of CCLK/ periods for the signal delay. Since the CCLK/ signal is asynchronous, the actual delay selected may only be specified with a tolerance of one CCLK/ period. In this example a delay of 3 - 4 CCLK/ periods was selected; with a CCLK/ period of 100 ns, the XACK/ delay would occur somewhere within the range of 300 - 400 ns from the time when the CLEAR signal goes high.

The control signal logic used in the 8-bit version of the slave design example is identical to the logic used in the 8/16-bit version.

Interrupt Logic — The interrupt logic uses a 74S74 flip-flop to latch an asynchronous interrupt request from some external logic. The Q output of the INTERRUPT REQUEST LATCH is output through an open collector gate to one of the MULTIBUS interrupt lines. The state of the INTERRUPT REQUEST LATCH is transferred to the INTERRUPT STATUS LATCH when a read command is performed on I/O port BASE ADDRESS+0 (E30 for the jumper configuration shown). The Q output of INTERRUPT STATUS LATCH is used to drive data line D0 of the on-board data bus by using an 8089 tri-state driver. If a user program performs an INPUT from I/O port E30, data bit 0 will be set to 1 if the INTERRUPT REQUEST LATCH is set.

The purpose of INTERRUPT STATUS LATCH is to minimize the possibility of the asynchronous interrupt occuring while the interrupt status is being read by a bus master. If the latch was not included in the design and an asynchronous interrupt did occur while a bus master is reading MULTIBUS data line DAT0/, a data buffer on the master could go into a meta-stable state. By adding the extra latch, which is clocked by the IORD/ command for I/O port E30, the possibility of data line DAT0/ changing during a bus master read operation is eliminated.

The INTERRUPT REQUEST LATCH is cleared when a user program performs an OUTPUT to I/O port E30.

This interrupt structure assumes that several interrupt sources may exist on the same MULTIBUS interrupt line (for example, INT3/). When the MULTIBUS master gets interrupted, it must poll the possible sources of the interrupt received and after determining the source of the interrupt, it must clear the INTERRUPT REQUEST LATCH for that particular interrupt source.

The interrupt logic for the 8-bit version of the design example is identical to the interrupt logic of the 8/16-bit version of the design example.

PPI Operation — Two 8255A Parallel Peripheral Interface (PPI) devices are shown interfaced to the slave design example logic. One PPI is connected to the on-board data bus lines D0 - D7 and is addressed with the even I/O port addresses E38, E3A, E3C, and E3E. The second PPI is connected to data bus lines D8 - DF and is addressed with the odd I/O port addresses E39, E3B, E3D, and E3F. The even or odd I/O port selection is controlled by using the ADR0 address line in the chip select term of the PPIs. In addition, the odd PPI (A11) is selected when the BHENBL term is high. This occurs when the MULTIBUS signal BHEN/ is low indicating that a word (16-bit) I/O instruction is being executed. When a word I/O instruction is executed, both PPIs will perform the I/O operation specified.

The specifications of the 8255A device state that the address lines A0 and A1 and the chip select lines must be stable before the $\overline{RD}$ or $\overline{WR}$ lines are activated. The MULTIBUS specification address set-up time of 50 ns and the short gate propagation delays in this design assure that the address lines are stable before $\overline{RD}$ or $\overline{WR}$ are active.

The data hold requirements of the 8255A were discussed in a previous section. The 8255A specification states that data will be stable on the data bus lines a maximum of 250 ns after a READ command. This specification was used to select the delay for the XACK/ signal.

The PPI operation for the 8-bit version of the design example is slightly different than that used for the 8/16-bit version. The chip select signal for the bottom PPI does not use the BHENBL term since 16-bit data transfers are not possible with an 8-bit I/O slave board. Also, the chip select and address signals have been swapped so the top PPI occupies I/O address range X8 - XB, and the bottom PPI occupies I/O address range XC - XF (X is the base address of the 8-bit version). This swapping of the address lines was not necessary; however, it was thought to be more convenient to access the PPIs in two groups of 4 contiguous I/O port addresses.

IV. SUMMARY

This application note has shown the structure of the Intel MULTIBUS system bus. The structure supports a wide range of system modules from the Intel OEM Microcomputer Systems product line that can be extended with the addition of user designed modules. Because the user designed modules are no doubt unique to particular applications, a goal of this application note has been to describe in detail the singular common element - the bus interface. Material has also been presented to assist the systems designer to understanding the bus functions so that successful systems integration can be achieved.

APPENDIX A
PIN ASSIGNMENT OF BUS SIGNALS ON MULTIBUS BOARD P1 CONNECTOR

		(COMPONENT SIDE)			(CIRCUIT SIDE)	
	PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
POWER SUPPLIES	1	GND	Signal GND	2	GND	Signal GND
	3	+5V	+5Vdc	4	+5V	+5Vdc
	5	+5V	+5Vdc	6	+5V	+5Vdc
	7	+12V	+12Vdc	8	+12V	+12Vdc
	9	−5V	−5Vdc	10	−5V	−5Vdc
	11	GND	Signal GND	12	GND	Signal GND
BUS CONTROLS	13	BCLK/	Bus Clock	14	INIT/	Initialize
	15	BPRN/	Bus Pri. In	16	BPRO/	Bus Pri. Out
	17	BUSY/	Bus Busy	18	BREQ/	Bus Request
	19	MRDC/	Mem Read Cmd	20	MWTC/	Mem Write Cmd
	21	IORC/	I/O Read Cmd	22	IOWC/	I/O Write Cmd
	23	XACK/	XFER Acknowledge	24	INH1/	Inhibit 1 disable RAM
BUS CONTROLS AND ADDRESS	25		Reserved	26	INH2/	Inhibit 2 disable PROM or ROM
	27	BHEN/	Byte High Enable	28	AD10/	
	29	CBRQ/	Common Bus Request	30	AD11/	Address
	31	CCLK/	Constant Clk	32	AD12/	Bus
	33	INTA/	Intr Acknowledge	34	AD13/	
INTERRUPTS	35	INT6/	Parallel	36	INT7/	Parallel
	37	INT4/	Interrupt	38	INT5/	Interrupt
	39	INT2/	Requests	40	INT3/	Requests
	41	INT0/		42	INT1/	
ADDRESS	43	ADRE/		44	ADRF/	
	45	ADRC/		46	ADRD/	
	47	ADRA/	Address	48	ADRB/	Address
	49	ADR8/	Bus	50	ADR9/	Bus
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
DATA	59	DATE/		60	DATF/	
	61	DATC/		62	DATD/	
	63	DATA/	Data	64	DATB/	Data
	65	DAT8/	Bus	66	DAT9/	Bus
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
POWER SUPPLIES	75	GND	Signal GND	76	GND	Signal GND
	77		Reserved	78		Reserved
	79	−12V	−12Vdc	80	−12V	−12Vdc
	81	+5V	+5Vdc	82	+5V	+5Vdc
	83	+5V	+5Vdc	84	+5V	+5Vdc
	85	GND	Signal GND	86	GND	Signal GND

All Mnemonics © Intel Corporation 1978

APPENDIX A (Continued)

P2 CONNECTOR PIN ASSIGNMENT OF OPTIONAL BUS SIGNALS

	(COMPONENT SIDE)			(CIRCUIT SIDE)	
PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
1	GND	Signal GND	2	GND	Signal GND
3	5 VB	+5V Battery	4	5 VB	+5V Battery
5		Reserved	6	VCCPP	+5V Pulsed Power
7	-5 VB	−5V Battery	8	-5 VB	−5V Battery
9		Reserved	10	Reserved	
11	12 VB	+12V Battery	12	12 VB	+12V Battery
13	PFSR/	Power Fail Sense Reset	14	Reserved	
15	-12 VB	−12V Battery	16	-12 VB	−12V Battery
17	PFSN/	Power Fail Sense	18	ACLO	AC Low
19	PFIN/	Power Fail Interrupt	20	MPRO/	Memory Protect
21	GND	Signal GND	22	GND	Signal GND
23	+15V	+15V	24	+15V	+15V
25	−15V	−15V	26	−15V	−15V
27	PAR1/	Parity 1	28	HALT/	Bus Master HALT
29	PAR2/	Parity 2	30	WAIT/	Bus Master WAIT STATE
31	Reserved		32	ALE	Bus Master ALE
33			34	Reserved	
35			36	Reserved	
37			38	AUX RESET/	Reset switch
39			40	Reserved	
40			42		
43			44		
45			46		
47			48		
49			50		
51			52		
53			54		
55			56		
57			58		
59			60		

Notes:

1. PFIN, on slave modules, if possible, should have the option of connecting to INT0/ on P1.
2. All undefined pins are reserved for future use.

APPENDIX B
BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BCY}	Bus Clock Period	100	D.C.	ns
t_{BW}	Bus Clock Width	0.35 t_{BCY}	0.65 t_{BCY}	
t_{SKEW}	BCLK/skew		3	ns
t_{PD}	Standard Bus Propagation Delay		3	
t_{AS}	Address Set-Up Time (at Slave Board)	50		ns
t_{DS}	Write Data Set· Up Time	50		ns
t_{AH}	Address Hold Time	50		ns
t_{DHW}	Write Data Hold Time	50		ns
t_{DXL}	Read Data Set Up Time To XACK	0		ns
t_{DHR}	Read Data Hold Time	0	65	ns
t_{XAH}	Acknowledge Hold Time	0	65	ns
t_{XACK}	Acknowledge Time	0	t_{TOUT}	ns
t_{CMD}	Command Pulse Width	100	t_{TOUT}	ns
t_{ID}	Inhibit Delay	0	100 (Recommend < 100 ns)	ns
t_{XACKA}	Acknowledge Time of of an Inhibited Slave	t_{IAD} + 50 ns	t_{TOUT}	
t_{XACKB}	Acknowledge Time of an Inhibiting Slave	1.5	t_{TOUT}	μs
t_{IAD}	Acknowledge Disable from Inhibit (An internal parameter on an inhibited slave; used to determine t_{XACKA} Min.)	0	100 (arbitrary)	ns
t_{AIZ}	Address to Inhibits High delay		100	ns
t_{INTA}	INTA/ Width	250		ns
t_{CSEP}	Command Separation	100		ns

APPENDIX B (Continued)
BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BREQL}	↓BCLK/ to BREQ/ Low Delay	0	35	ns
t_{BREQH}	↓BCLK/ to BREQ/ High Delay	0	35	ns
t_{BPRNS}	BPRN/ to ↓BCLK/ Setup Time	22		ns
t_{BUSY}	BUSY/ delay from ↓BCLK/	0	70	ns
t_{BUSYS}	BUSY/ to ↓BCLK/ Setup Time	25		ns
t_{BPRO}	↓BCLK/ to BPRO/ (CLK to Priority Out)	0	40	ns
t_{BPRNO}	BPRN/ to BPRO/ (Priority In to Out)	0	30	ns
t_{CBRO}	↓BCLK/ to CBRQ/ (CLKto Common Bus Request)	0	60	ns
t_{CBRQS}	CBRQ/ to ↓BCLK/ Setup Time	35		ns
t_{CPM}	Central Priority Module Resolution Delay (Parallel Priority)	0	$t_{BCY}-t_{BREQ}$ $-2t_{PD}$ $-t_{BPRNS}$ $-t_{SKEW}$	
t_{CCY}	C-clock Period	100	110	ns
t_{CW}	C-clock Width	0.35 t_{CCY}	0.65 t_{CCY}	ns
t_{INIT}	INIT/Width	5		ms
t_{INITS}	INIT/ to MPRO/ Setup Time	100		ns
t_{PBD}	Power Backup Logic Delay	0	200	ns
t_{PFINW}	PFIN/ Width	2.5		ms
t_{MPRO}	MPRO/ Delay	2.0	2.5	ms
t_{ACLOW}	ACLO/ Width	3.0		ms
t_{PFSRW}	PFSR/ Width	100		ns
t_{TOUT}	Timeout Delay	5	∞	ms
t_{DCH}	D.C. Power Supply Hold from ALCO/	3.0		ms
t_{DCS}	D.C. Power Supply Setup to ACLO/	5		ms

APPENDIX C

BUS DRIVERS, RECEIVERS, AND TERMINATIONS

	Driver 1,3					Receiver 2,3				Termination			
Bus Signals	Location	Type	I_{OL} Min$_{ma}$	I_{OH} Min$_{\mu a}$	C_O Max$_{pf}$	Location	I_{IL} Max$_{ma}$	I_{IH} Max$_{\mu a}$	C_I Max$_{pf}$	Location	Type	R	Units
DAT0/→DATF/ (16 lines)	Masters and Slaves	TRI	16	-2000	300	Masters and Slaves	−0.8	125	18	1 place	Pullup	2.2	KΩ
ADR0/-ADRB/, BHEN/ (21 lines)	Masters	TRI	16	-2000	300	Slaves	−0.8	125	18	1 place	Pullup	2.2	KΩ
MRDC/,MWTC/	Masters	TRI	32	-2000	300	Slaves (Memory; memory-mapped I/O)	−2	125	18	1 place	Pullup	1	KΩ
IORC/,IOWC/	Masters	TRI	32	-2000	300	Slaves (I/O)	−2	125	18	1 place	Pullup	1	KΩ
XACK/	Slaves	TRI	32	-2000	300	Masters	−2	125	18	1 place	Pullup	510	Ω
INH1/,INH2/	Inhibiting Slaves	OC	16	—	300	Inhibited Slaves (RAM, PROM, ROM, Memory-Mapped I/O)	−2	50	18	1 place	Pullup	1	KΩ
BCLK/	1 place (Master us)	TTL	48	−3000	300	Master	−2	125	18	Mother-board	To +5V To GND	220 330	Ω Ω
BREQ/	Each Master	TTL	5	−400	60	Central Priority Module	2	50	18	Central Priority Module (not req)	Pullup	1	KΩ
BPRO/	Each Master	TTL	5	−400	60	Next Master in Serial Priority Chain at its BPRN/	−1.6	50	18	(not req)			
BPRN/	Parallel: Central Priority Module Serial:Prev Masters BPRO/	TTL	5	−400	300	Master	−2	50		(not req)			
BUSY/, CBRQ	All Masters	O.C.	32	—	300	All Masters	−2	50	18	1 place	Pullup	1	KΩ
INIT/	Master	O.C.	32	—	300	All	−2	50	18	1 place	Pullup	2.2	KΩ
CCLK/	1 place	TTL	48	−3000	300	Any	−2	125	18	Mother-board	To +5V To GND	220 330	Ω Ω
INTA/	Masters	TRI	32	-2000	300	Slaves (Interrupting I/O)	−2	125	18	1 place	Pullup	1	KΩ
INT0/→INT7/ (8 lines)	Slaves	O.C.	16	—	300	Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFSR/	User's Fron Panel?	TTL	16	−400	300	Slaves, Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFSN/	Power Back Up Unit	TTL	16	−400	300	Masters	−1.6	40	16	1 place	Pullup	1	KΩ
ACLO	Power Supply	O.C.	16	−400	300	Slaves, Masters	−1.6	40	18	1 place	Pullup	1	KΩ
PFIN/	Power Back-Up Unit	O.C.	16	−400	300	Masters	−1.6	40	18	1 place	Pullup	1	KΩ
MPRO/	Power Back-Up Unit	TTL	16	−400	300	Slaves Masters	−1.6	40	18	1 place	Pullup	1	KΩ

APPENDIX C (Continued)

BUS DRIVERS, RECEIVERS, AND TERMINATIONS

	Driver 1,3					Receiver 2,3				Termination		
Bus Signals	Location	Type	I_{OL} Min$_{ma}$	I_{OH} Min$_{\mu a}$	C_O Max$_{pf}$	Location	I_{IL} Max$_{ma}$	I_{IH} Max$_{\mu a}$	C_I Max$_{pf}$	Location	Type	R Units
Aux Reset/	User's Front Panel?	Switch to GND	—	—	—	Masters	−2	50	18	None		

Notes:

1. Driver Requirements

   I_{OH} = High Output Current Drive
   I_{OL} = Low Output Current Drive
   C_O = Capacitance Drive Capability
   TRI = 3-State Drive
   O.C. = Open Collector Driver
   TTL = Totem-pole Driver

2. Receiver Requirements

   I_{IH} = High Input Current Load
   I_{IL} = Low Input Current Load
   C_I = Capacitive Load

3. TTL low state must be ≥ -0.5v but ≤ 0.8v at the receivers
   TTL high state must be ≥ 2.0v but ≤ 5.5v at the receivers

4. For the iSBC 80/10 and the iSBC 80/10A use only a 1K pull-up resistor to +5v for BCLK/ and CCLK/ termination.

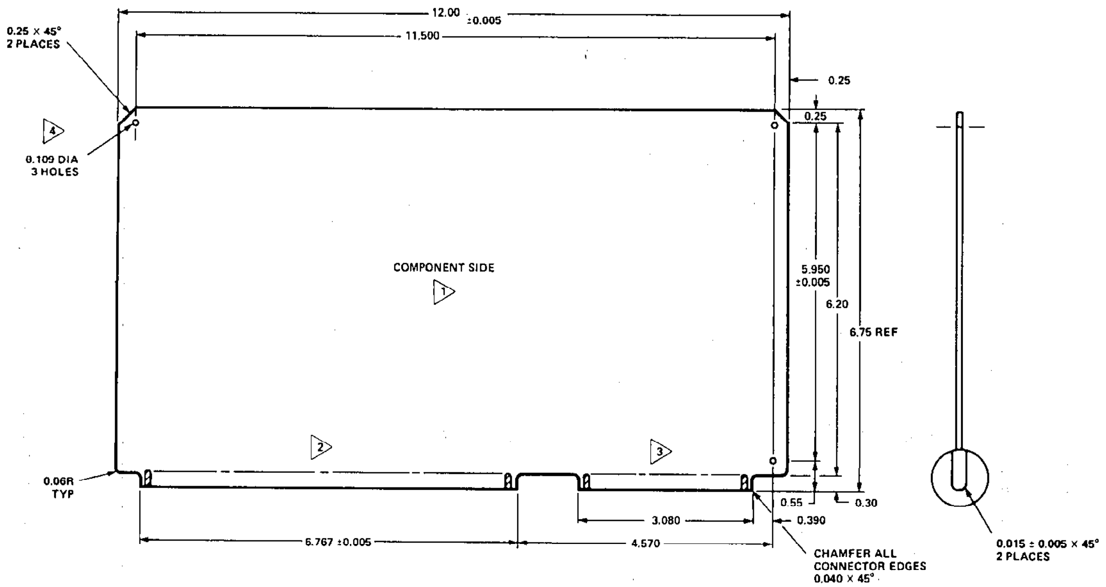
APPENDIX D
BUS POWER SPECIFICATIONS

	Standard (P1)				Optional (P2)					
					Analog Power		Battery Power Backup			
	Ground	**+5**	**+12**	**−12**	**+15**	**−15**	**+5**	**+12**	**−12**	**−5**
Mnemonic	GND	+5V	+12V	−12V	+15V	−15V	+5B	+12B	−12B	−5B
Bus Pins	P1+1,2, 11,12, 75,76 85,86	P1+3,4, 5,6,81, 82,83, 84	P1+7,8	P1+79, 80	P2+23, 24	P2+25, 26	P2+3,4, 5,6	P2+11, 12	P2+15, 16	P2−7,8
Nominal Output	Ref.	+5.0V	+12.0V	−12.0V	+15.0V	−15.0V	+5.0V	+12.0V	−12.0V	−5.0V
Tolerance from Nominal[1]	Ref.	±5%	±5%	±5%	±3%	±3%	±5%	±5%	±5%	±5%
Ripple (Pk-Pk)[2]	Ref.	50 mV	50 mV	50 mV	10 mV	10 mV	50 mV	50 mV	50 mV	50 mV
Transient Response Time[3]		500 μs	500μs	500 μs	100 μs	100 μs	500 μs	500 μs	500 μs	500 μs
Transient Deviation[4]		±10%	±10%	±10%	±10%	±10%	±10%	±10%	±10%	±10%

NOTES:

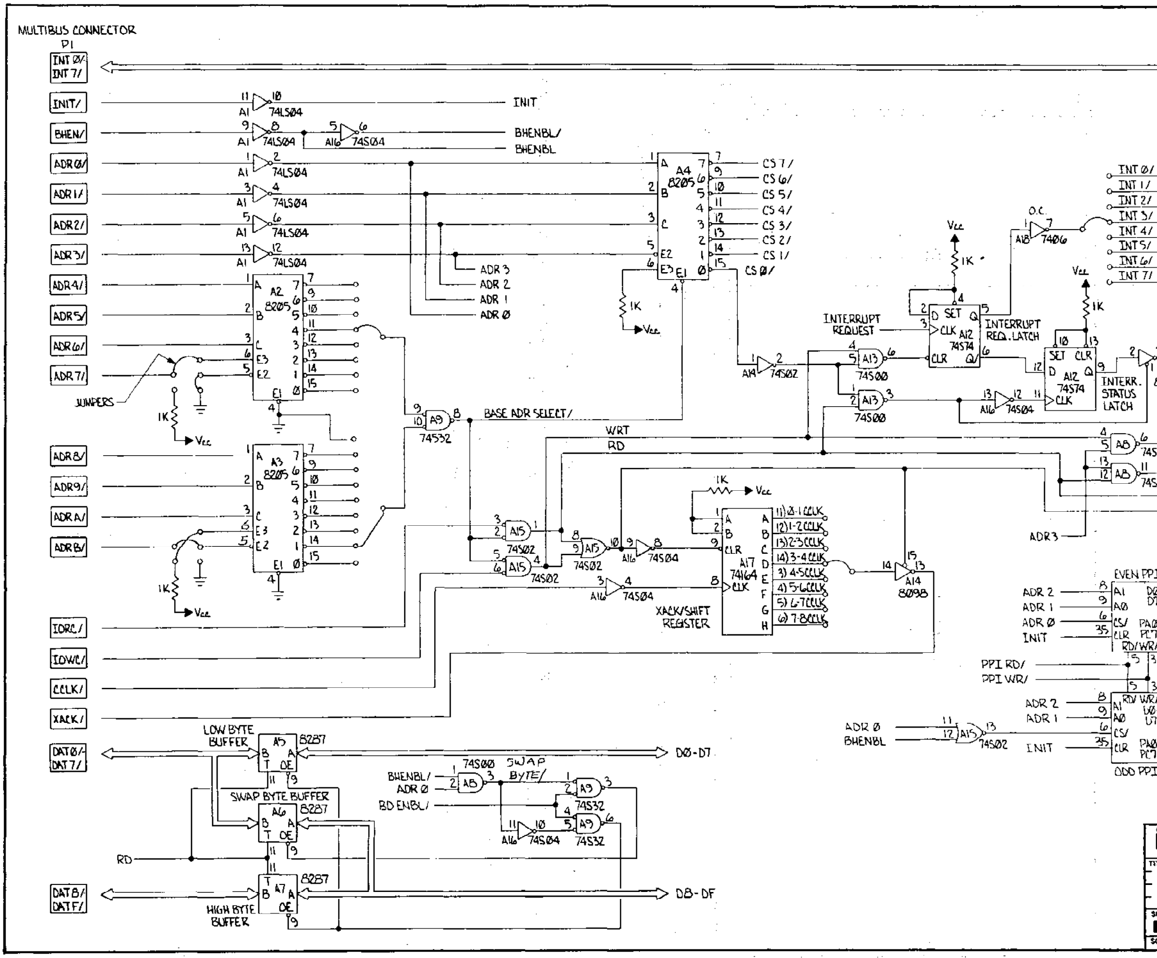
1. Tolerance is worst case, including initial voltage setting line and load effects of power source, temperature drift, and any additional steady state influences.
2. As measured over any bandwidth not to exceed 0 to 500 kHz.
3. As measured from the start of a load change to the time an output recovers within ±0.1% of final voltage.
4. Measured as the peak deviation from the initial voltage.

APPENDIX E
MECHANICAL SPECIFICATIONS

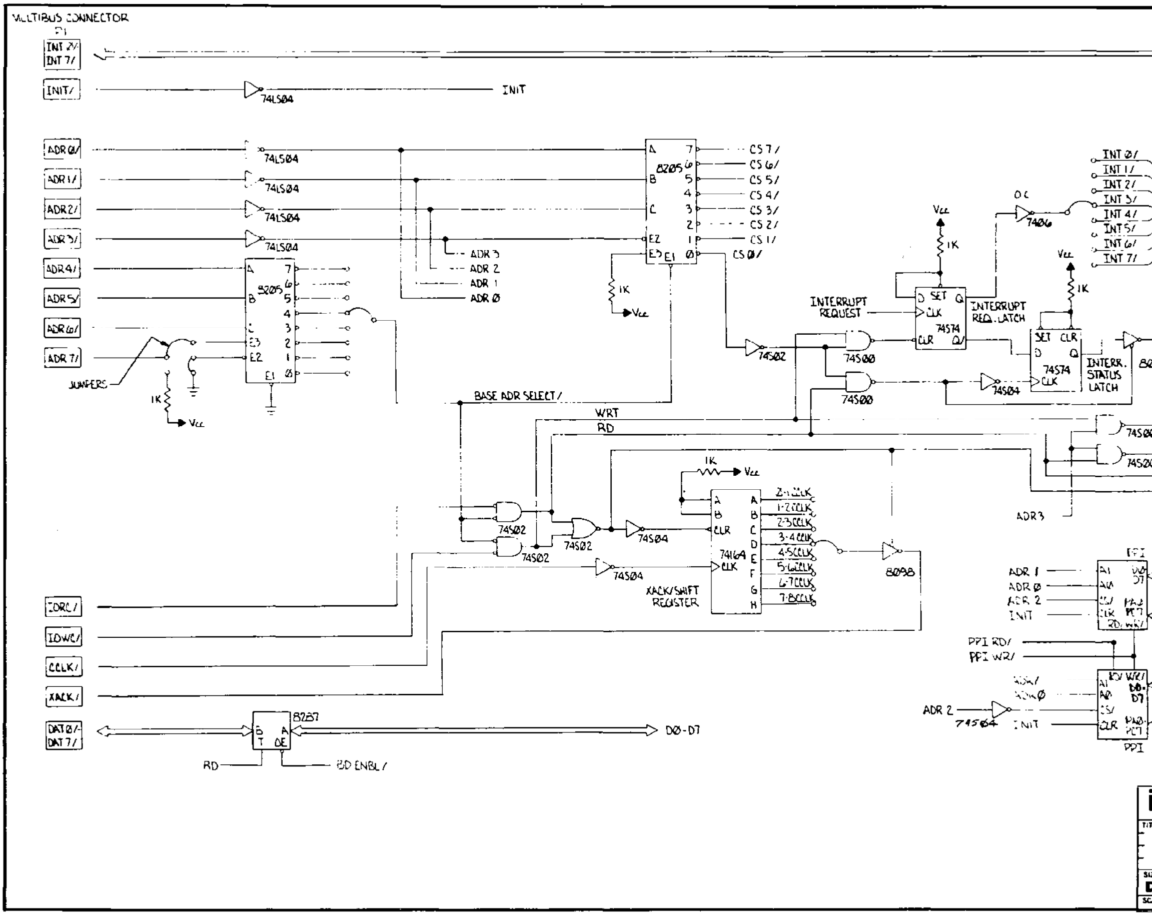


NOTES:

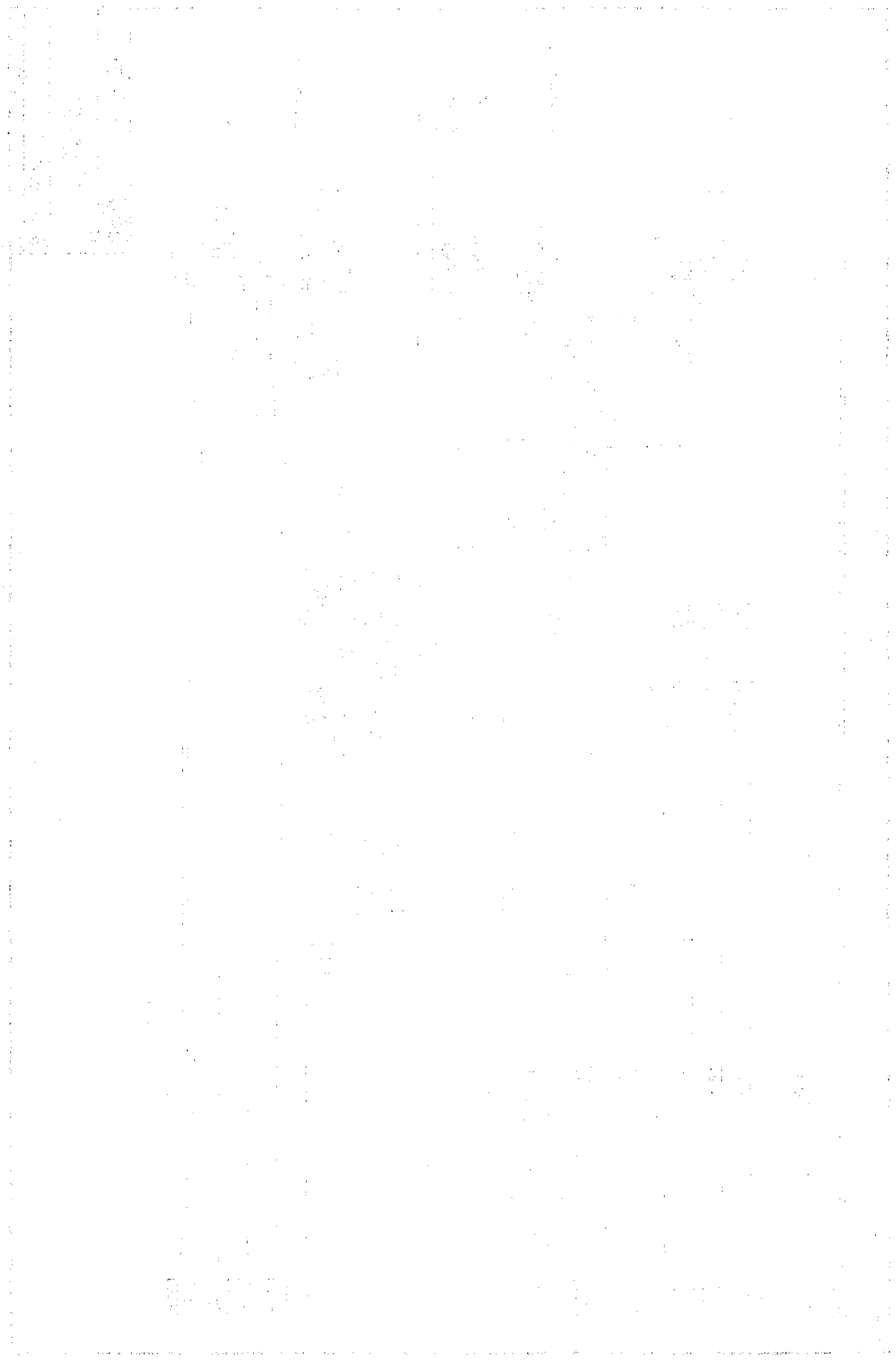
1. BOARD THICKNESS: 0.062
2. MULTIBUS CONNECTOR: 86-PIN, 0.156 SPACING
   CDC VFB01E43D00A1
   VIKING 2VH43/1ANE5
3. AUXILIARY CONNECTOR: 60-PIN, 0.100 SPACING
   CDC VPB01B30D00A1
   TI H311130
   AMP PE5-14559
4. EJECTOR TYPE: SCANBE #S203
5. BUS DRIVERS AND RECEIVERS SHOULD BE LOCATED AS CLOSE AS POSSIBLE TO THEIR RESPECTIVE MULTIBUS PIN CONNECTIONS
6. BOARD SPACING: 0.6
7. COMPONENT HEIGHT: 0.4
8. CLEARANCE ON CONDUCTOR NEAR EDGES: 0.050



MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8/16-BIT VERSION



MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8-BIT VERSION



Appendix B
Device Specifications

B

iAPX 86/10
16-BIT HMOS MICROPROCESSOR

8086/8086-2/8086-1

- **Direct Addressing Capability to 1 MByte of Memory**
- **Architecture Designed for Powerful Assembly Language and Efficient High Level Languages.**
- **14 Word, by 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8 and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **Range of Clock Rates:**
  5 MHz for 8086,
  8 MHz for 8086-2,
  10 MHz for 8086-1
- **MULTIBUS™ System Compatible Interface**

The Intel iAPX 86/10 high performance 16-bit CPU is available in three clock rates: 5, 8 and 10 MHz. The CPU is implemented in N-Channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The iAPX 86/10 operates in both single processor and multiple processor configurations to achieve high performance levels.

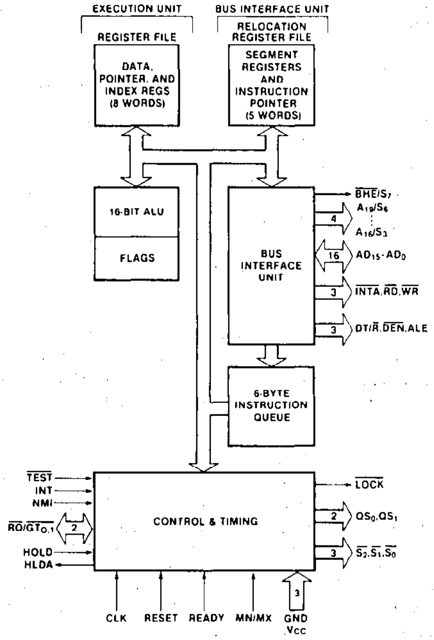


Figure 1. iAPX 86/10 CPU Block Diagram

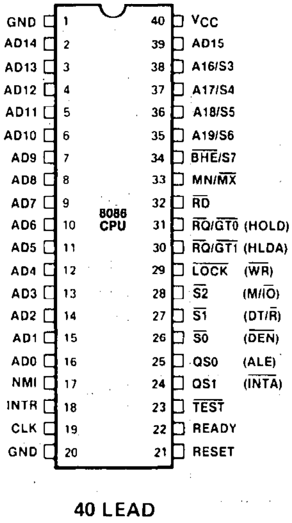


Figure 2. iAPX 86/10 Pin Configuration

Table 1. Pin Description

The following pin function descriptions are for iAPX 86 systems in either minimum or maximum mode. The "Local Bus" in these descriptions is the direct multiplexed bus interface connection to the 8086 (without regard to additional bus buffers).

<table>
<tr><th>Symbol</th><th>Pin No.</th><th>Type</th><th>Name and Function</th></tr>
<tr><td>AD_{15}-AD_0</td><td>2-16, 39</td><td>I/O</td><td><b>Address Data Bus:</b> These lines constitute the time multiplexed memory/IO address (T_1) and data (T_2, T_3, T_W, T_4) bus. A_0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D_7-D_0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. Eight-bit oriented devices tied to the lower half would normally use A_0 to condition chip select functions. (See $\overline{BHE}$.) These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge."</td></tr>
<tr><td>A_{19}/S_6,
A_{18}/S_5,
A_{17}/S_4,
A_{16}/S_3</td><td>35-38</td><td>O</td><td><b>Address/Status:</b> During T_1 these are the four most significant address lines for memory operations. During I/O operations these lines are LOW. During memory and I/O operations, status information is available on these lines during T_2, T_3, T_W, and T_4. The status of the interrupt enable FLAG bit (S_5) is updated at the beginning of each CLK cycle. A_{17}/S_4 and A_{16}/S_3 are encoded as shown.
This information indicates which relocation register is presently being used for data accessing.
These lines float to 3-state OFF during local bus "hold acknowledge."
<table>
<tr><th>A_{17}/S_4</th><th>A_{16}/S_3</th><th>Characteristics</th></tr>
<tr><td>0 (LOW)</td><td>0</td><td>Alternate Data</td></tr>
<tr><td>0</td><td>1</td><td>Stack</td></tr>
<tr><td>1 (HIGH)</td><td>0</td><td>Code or None</td></tr>
<tr><td>1</td><td>1</td><td>Data</td></tr>
<tr><td>S_6 is 0 (LOW)</td><td></td><td></td></tr>
</table></td></tr>
<tr><td>$\overline{BHE}/S_7$</td><td>34</td><td>O</td><td><b>Bus High Enable/Status:</b> During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D_{15}-D_8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read, write, and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus. The S_7 status information is available during T_2, T_3, and T_4. The signal is active LOW, and floats to 3-state OFF in "hold." It is LOW during T_1 for the first interrupt acknowledge cycle.
<table>
<tr><th>$\overline{BHE}$</th><th>A_0</th><th>Characteristics</th></tr>
<tr><td>0</td><td>0</td><td>Whole word</td></tr>
<tr><td>0</td><td>1</td><td>Upper byte from/ to odd address</td></tr>
<tr><td>1</td><td>0</td><td>Lower byte from/ to even address</td></tr>
<tr><td>1</td><td>1</td><td>None</td></tr>
</table></td></tr>
<tr><td>$\overline{RD}$</td><td>32</td><td>O</td><td><b>Read:</b> Read strobe indicates that the processor is performing a memory of I/O read cycle, depending on the state of the S_2 pin. This signal is used to read devices which reside on the 8086 local bus. $\overline{RD}$ is active LOW during T_2, T_3 and T_W of any read cycle, and is guaranteed to remain HIGH in T_2 until the 8086 local bus has floated.
This signal floats to 3-state OFF in "hold acknowledge."</td></tr>
<tr><td>READY</td><td>22</td><td>I</td><td><b>READY:</b> is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The READY signal from memory/IO is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH. The 8086 READY input is not synchronized. Correct operation is not guaranteed if the setup and hold times are not met.</td></tr>
<tr><td>INTR</td><td>18</td><td>I</td><td><b>Interrupt Request:</b> is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.</td></tr>
<tr><td>$\overline{TEST}$</td><td>23</td><td>I</td><td><b>$\overline{TEST}$:</b> input is examined by the "Wait" instruction. If the $\overline{TEST}$ input is LOW execution continues, otherwise the processor waits in an "Idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.</td></tr>
</table>

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
NMI	17	I	**Non-maskable interrupt:** an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from a LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.
RESET	21	I	**Reset:** causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the Instruction Set description, when RESET returns LOW. RESET is internally synchronized.
CLK	19	I	**Clock:** provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}	40		V_{CC}: +5V power supply pin.
GND	1, 20		**Ground**
MN/$\overline{MX}$	33	I	**Minimum/Maximum:** indicates what mode the processor is to operate in. The two modes are discussed in the following sections.

The following pin function descriptions are for the 8086/8288 system in maximum mode (i.e., MN/$\overline{MX}$ = V_{SS}). Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

<table>
<tr><td>$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$</td><td>26-28</td><td>O</td><td><b>Status:</b> active during T_4, T_1, and T_2 and is returned to the passive state (1,1,1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. Any change by $\overline{S_2}$, $\overline{S_1}$, or $\overline{S_0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle.
These signals float to 3-state OFF in "hold acknowledge." These status lines are encoded as shown.
<table>
<tr><th>$\overline{S_2}$</th><th>$\overline{S_1}$</th><th>$\overline{S_0}$</th><th>Characteristics</th></tr>
<tr><td>0 (LOW)</td><td>0</td><td>0</td><td>Interrupt Acknowledge</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>Read I/O Port</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>Write I/O Port</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>Halt</td></tr>
<tr><td>1 (HIGH)</td><td>0</td><td>0</td><td>Code Access</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>Read Memory</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>Write Memory</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>Passive</td></tr>
</table></td></tr>
<tr><td>$\overline{RQ}/\overline{GT_0}$,
$\overline{RQ}/\overline{GT_1}$</td><td>30, 31</td><td>I/O</td><td><b>Request/Grant:</b> pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ has an internal pull-up resistor so may be left unconnected. The request/grant sequence is as follows (see Figure 9):
1. A pulse of 1 CLK wide from another local bus master indicates a local bus request ("hold") to the 8086 (pulse 1).
2. During a T_4 or T_I clock cycle, a pulse 1 CLK wide from the 8086 to the requesting master (pulse 2), indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during "hold acknowledge."
3. A pulse 1 CLK wide from the requesting master indicates to the 8086 (pulse 3) that the "hold" request is about to end and that the 8086 can reclaim the local bus at the next CLK.
Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW.
If the request is made while the CPU is performing a memory cycle, it will release the local bus during T_4 of the cycle when all the following conditions are met:
1. Request occurs on or before T_2.
2. Current cycle is not the low byte of a word (on an odd address).
3. Current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A locked instruction is not currently executing.</td></tr>
</table>

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
			If the local bus is idle when the request is made the two possible events will follow: 1. Local bus will be released during the next clock. 2. A memory cycle will start within 3 clocks. Now the four rules for a currently active memory cycle apply with condition number 1 already satisfied.
$\overline{\text{LOCK}}$	29	O	**$\overline{\text{LOCK}}$:** output indicates that other system bus masters are not to gain control of the system bus while $\overline{\text{LOCK}}$ is active LOW. The $\overline{\text{LOCK}}$ signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state OFF in "hold acknowledge."
QS_1, QS_0	24, 25	O	**Queue Status:** The queue status is valid during the CLK cycle after which the queue operation is performed. QS_1 and QS_0 provide status to allow external tracking of the internal 8086 instruction queue.

The following pin function descriptions are for the 8086 in minimum mode (i.e., MN/$\overline{\text{MX}}$ = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

M/$\overline{\text{IO}}$	28	O	**Status line:** logically equivalent to S_2 in the maximum mode. It is used to distinguish a memory access from an I/O access. M/$\overline{\text{IO}}$ becomes valid in the T_4 preceding a bus cycle and remains valid until the final T_4 of the cycle (M = HIGH, IO = LOW). M/$\overline{\text{IO}}$ floats to 3-state OFF in local bus "hold acknowledge."
$\overline{\text{WR}}$	29	O	**Write:** indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/$\overline{\text{IO}}$ signal. $\overline{\text{WR}}$ is active for T_2, T_3 and T_W of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge."
$\overline{\text{INTA}}$	24	O	**$\overline{\text{INTA}}$** is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T_2, T_3 and T_W of each interrupt acknowledge cycle.
ALE	25	O	**Address Latch Enable:** provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during T_1 of any bus cycle. Note that ALE is never floated.
DT/$\overline{\text{R}}$	27	O	**Data Transmit/Receive:** needed in minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically DT/$\overline{\text{R}}$ is equivalent to $\overline{S_1}$ in the maximum mode, and its timing is the same as for M/$\overline{\text{IO}}$. (T = HIGH, R = LOW.) This signal floats to 3-state OFF in local bus "hold acknowledge."
$\overline{\text{DEN}}$	26	O	**Data Enable:** provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{\text{DEN}}$ is active LOW during each memory and I/O access and for INTA cycles. For a read or $\overline{\text{INTA}}$ cycle it is active from the middle of T_2 until the middle of T_4, while for a write cycle it is active from the beginning of T_2 until the middle of T_4. $\overline{\text{DEN}}$ floats to 3-state OFF in local bus "hold acknowledge."
HOLD, HLDA	31, 30	I/O	**HOLD:** indicates that another master is requesting a local bus "hold." To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement in the middle of a T_4 or T_I clock cycle. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will LOWer HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines. The same rules as for $\overline{\text{RQ}}$/$\overline{\text{GT}}$ apply regarding when the local bus will be released. HOLD is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the setup time.

FUNCTIONAL DESCRIPTION

GENERAL OPERATION

The internal functions of the iAPX 86/10 processor are partitioned logically into two processing units. The first is the Bus Interface Unit (BIU) and the second is the Execution Unit (EU) as shown in the block diagram of Figure 1.

These units can interact directly but for the most part perform as separate asynchronous operational processors. The bus interface unit provides the functions related to instruction fetching and queuing, operand fetch and store, and address relocation. This unit also provides the basic bus control. The overlap of instruction pre-fetching provided by this unit serves to increase processor performance through improved bus bandwidth utilization. Up to 6 bytes of the instruction stream can be queued while waiting for decoding and execution.

The instruction stream queuing mechanism allows the BIU to keep the memory utilized very efficiently. Whenever there is space for at least 2 bytes in the queue, the BIU will attempt a word fetch memory cycle. This greatly reduces "dead time" on the memory bus. The queue acts as a First-In-First-Out (FIFO) buffer, from which the EU extracts instruction bytes as required. If the queue is empty (following a branch instruction, for example), the first byte into the queue immediately becomes available to the EU.

The execution unit receives pre-fetched instructions from the BIU queue and provides un-relocated operand addresses to the BIU. Memory operands are passed through the BIU for processing by the EU, which passes results to the BIU for storage. See the Instruction Set description for further register set and architectural descriptions.

MEMORY ORGANIZATION

The processor provides a 20-bit address to memory which locates the byte being referenced. The memory is organized as a linear array of up to 1 million bytes, addressed as 00000(H) to FFFFF(H). The memory is logically divided into code, data, extra data, and stack segments of up to 64K bytes each, with each segment falling on 16-byte boundaries. (See Figure 3a.)

All memory references are made relative to base addresses contained in high speed segment registers. The segment types were chosen based on the addressing needs of programs. The segment register to be selected is automatically chosen according to the rules of the following table. All information in one segment type share the same logical attributes (e.g. code or data). By structuring memory into relocatable areas of similar characteristics and by automatically selecting segment registers, programs are shorter, faster, and more structured.

Word (16-bit) operands can be located on even or odd address boundaries and are thus not constrained to even boundaries as is the case in many 16-bit computers. For address and data operands, the least significant byte of the word is stored in the lower valued address location and the most significant byte in the next higher address location. The BIU automatically performs the proper number of memory accesses, one if the word operand is on an even byte boundary and two if it is on an odd byte boundary. Except for the performance penalty, this double access is transparent to the software. This performance penalty does not occur for instruction fetches, only word operands.

Physically, the memory is organized as a high bank (D_{15}-D_8) and a low bank (D_7-D_0) of 512K 8-bit bytes addressed in parallel by the processor's address lines

A_{19} - A_1. Byte data with even addresses is transferred on the D_7-D_0 bus lines while odd addressed byte data (A_0 HIGH) is transferred on the D_{15}-D_8 bus lines. The processor provides two enable signals, $\overline{BHE}$ and A_0, to selectively allow reading from or writing into either an odd byte location, even byte location, or both. The instruction stream is fetched from memory as words and is addressed internally by the processor to the byte level as necessary.

Memory Reference Need	Segment Register Used	Segment Selection Rule
Instructions	CODE (CS)	Automatic with all instruction prefetch.
Stack	STACK (SS)	All stack pushes and pops. Memory references relative to BP base register except data references.
Local Data	DATA (DS)	Data references when: relative to stack, destination of string operation, or explicitly overridden.
External (Global) Data	EXTRA (ES)	Destination of string operations: Explicitly selected using a segment override.

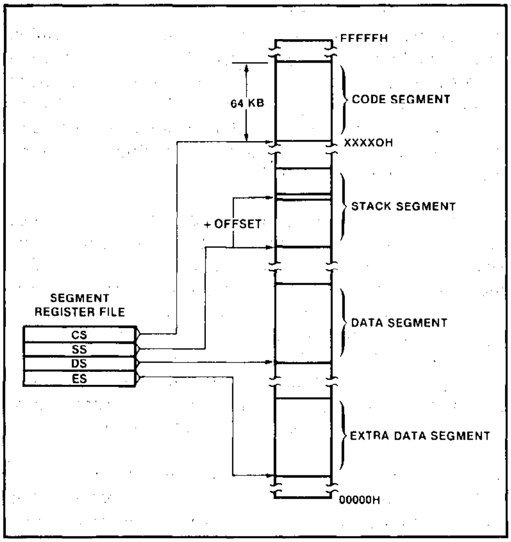


Figure 3a. Memory Organization

In referencing word data the BIU requires one or two memory cycles depending on whether or not the starting byte of the word is on an even or odd address, respectively. Consequently, in referencing word operands performance can be optimized by locating data on even address boundaries. This is an especially useful technique for using the stack, since odd address references to the stack may adversely affect the context switching time for interrupt processing or task multiplexing.

Certain locations in memory are reserved for specific CPU operations (see Figure 3b.) Locations from address FFFF0H through FFFFFH are reserved for operations including a jump to the initial program loading routine. Following RESET, the CPU will always begin execution at location FFFF0H where the jump must be. Locations 00000H through 003FFH are reserved for interrupt operations. Each of the 256 possible interrupt types has its service routine pointed to by a 4-byte pointer element consisting of a 16-bit segment address and a 16-bit offset address. The pointer elements are assumed to have been stored at the respective places in reserved memory prior to occurrence of interrupts.

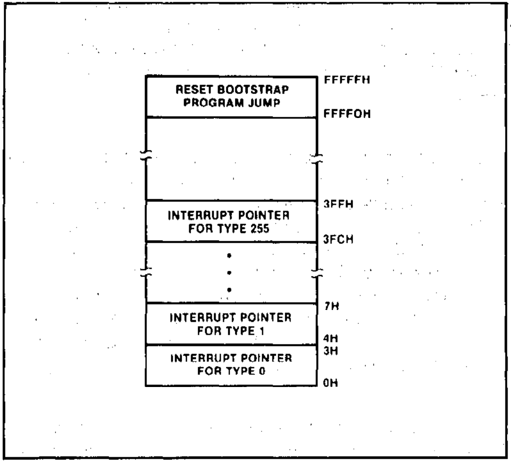

RESET BOOTSTRAP PROGRAM JUMP
FFFFFH
FFFF0H
INTERRUPT POINTER FOR TYPE 255
3FFH
3FCH
INTERRUPT POINTER FOR TYPE 1
7H
4H
3H
INTERRUPT POINTER FOR TYPE 0
0H

Figure 3b. Reserved Memory Locations

MINIMUM AND MAXIMUM MODES

The requirements for supporting minimum and maximum iAPX 86/10 systems are sufficiently different that they cannot be done efficiently with 40 uniquely defined pins. Consequently, the 8086 is equipped with a strap pin (MN/$\overline{MX}$) which defines the system configuration. The definition of a certain subset of the pins changes dependent on the condition of the strap pin. When MN/$\overline{MX}$ pin is strapped to GND, the 8086 treats pins 24 through 31 in maximum mode. An 8288 bus controller interprets status information coded into $\overline{S_0}$,$\overline{S_1}$,$\overline{S_2}$ to generate bus timing and control signals compatible with the MULTIBUS™ architecture. When the MN/$\overline{MX}$ pin is strapped to V_{CC}, the 8086 generates bus control signals itself on pins 24 through 31, as shown in parentheses in Figure 2. Examples of minimum mode and maximum mode systems are shown in Figure 4.

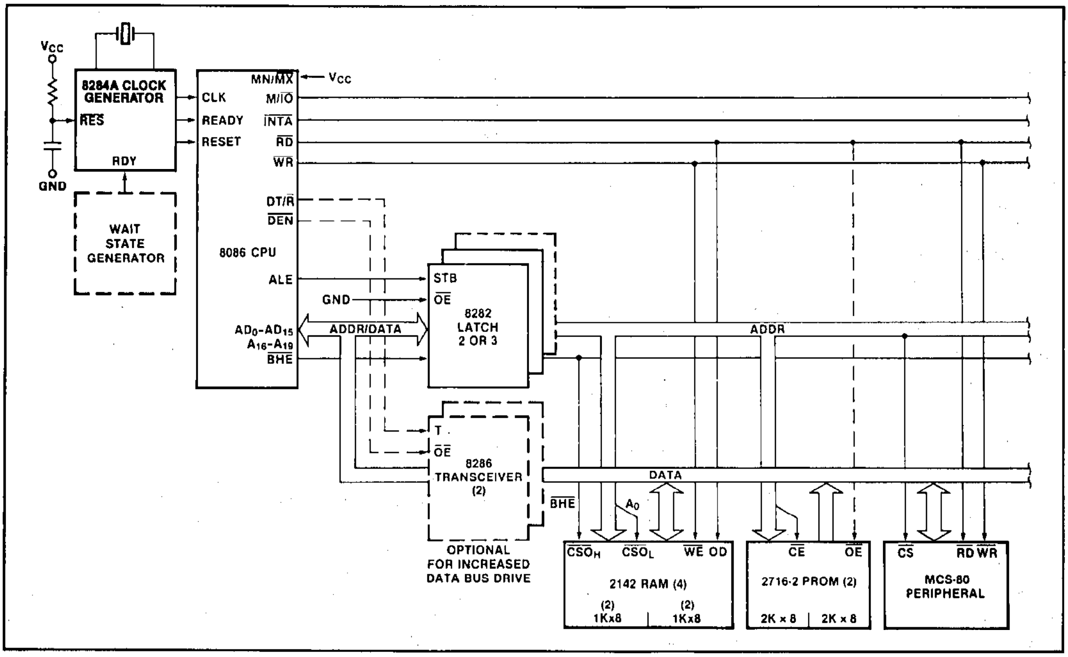


Figure 4a. Minimum Mode iAPX 86/10 Typical Configuration

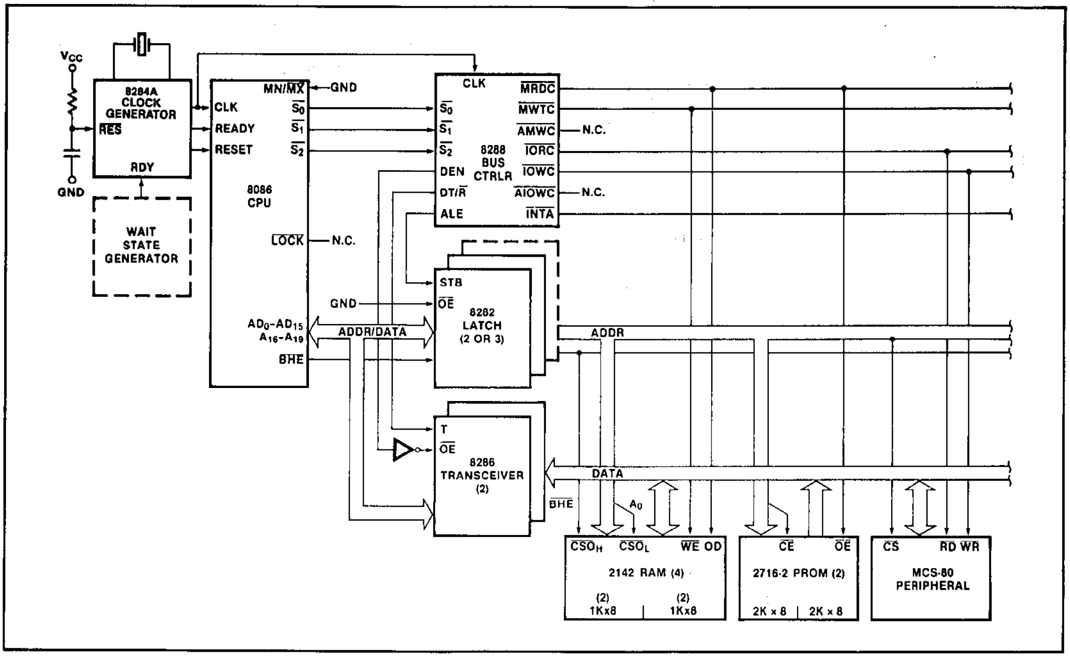


Figure 4b. Maximum Mode iAPX 86/10 Typical Configuration

BUS OPERATION

The 86/10 has a combined address and data bus commonly referred to as a time multiplexed bus. This technique provides the most efficient use of pins on the processor while permitting the use of a standard 40-lead package. This "local bus" can be buffered directly and used throughout the system with address latching provided on memory and I/O modules. In addition, the bus can also be demultiplexed at the processor with a single set of address latches if a standard non-multiplexed bus is desired for the system.

Each processor bus cycle consists of at least four CLK cycles. These are referred to as T_1, T_2, T_3 and T_4 (see Figure 5). The address is emitted from the processor during T_1 and data transfer occurs on the bus during T_3 and T_4. T_2 is used primarily for changing the direction of the bus during read operations. In the event that a "NOT READY" indication is given by the addressed device, "Wait" states (T_W) are inserted between T_3 and T_4. Each inserted "Wait" state is of the same duration as a CLK cycle. Periods can occur between 8086 bus cycles. These are referred to as "Idle" states (T_I) or inactive CLK cycles. The processor uses these cycles for internal housekeeping.

During T_1 of any bus cycle the ALE (Address Latch Enable) signal is emitted (by either the processor or the 8288 bus controller, depending on the MN/$\overline{MX}$ strap). At the trailing edge of this pulse, a valid address and certain status information for the cycle may be latched.

Status bits $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ are used, in maximum mode, by the bus controller to identify the type of bus transaction according to the following table:

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	CHARACTERISTICS
0 (LOW)	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1 (HIGH)	0	0	Instruction Fetch
1	0	1	Read Data from Memory
1	1	0	Write Data to Memory
1	1	1	Passive (no bus cycle)

Status bits S_3 through S_7 are multiplexed with high-order address bits and the $\overline{BHE}$ signal, and are therefore valid during T_2 through T_4. S_3 and S_4 indicate which segment register (see Instruction Set description) was used for this bus cycle in forming the address, according to the following table:

S_4	S_3	CHARACTERISTICS
0 (LOW)	0	Alternate Data (extra segment)
0	1	Stack
1 (HIGH)	0	Code or None
1	1	Data

S_5 is a reflection of the PSW interrupt enable bit. S_6=0 and S_7 is a spare status bit.

I/O ADDRESSING

In the 86/10, I/O operations can address up to a maximum of 64K I/O byte registers or 32K I/O word registers. The I/O address appears in the same format as the memory address on bus lines A_{15}-A_0. The address lines A_{19}-A_{16} are zero in I/O operations. The variable I/O instructions which use register DX as a pointer have full address capability while the direct I/O instructions directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space.

I/O ports are addressed in the same manner as memory locations. Even addressed bytes are transferred on the D_7-D_0 bus lines and odd addressed bytes on D_{15}-D_8. Care must be taken to assure that each register within an 8-bit peripheral located on the lower portion of the bus be addressed as even.

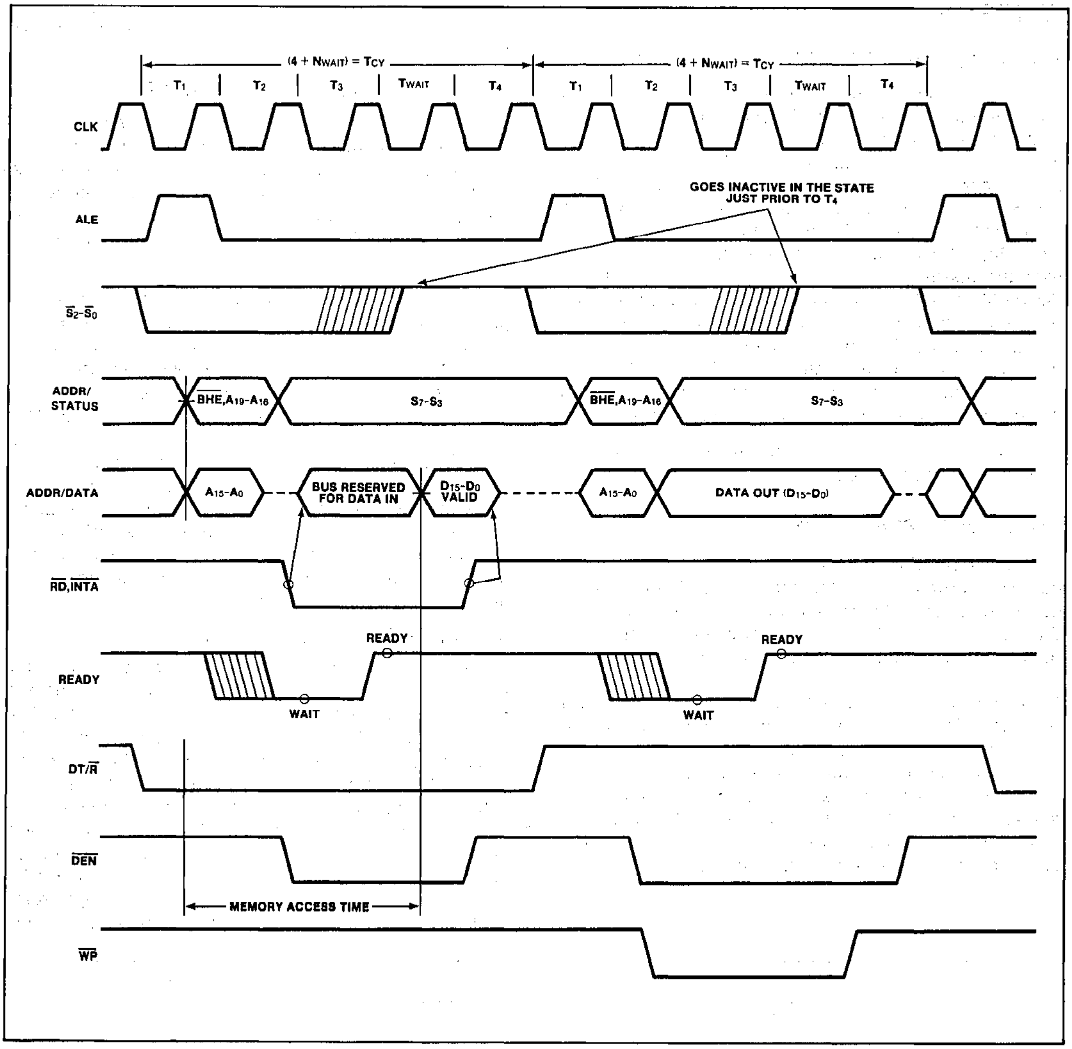


Figure 5. Basic System Timing

EXTERNAL INTERFACE

PROCESSOR RESET AND INITIALIZATION

Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin. The 8086 RESET is required to be HIGH for greater than 4 CLK cycles. The 8086 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH. The low-going transition of RESET triggers an internal reset sequence for approximately 10 CLK cycles. After this interval the 8086 operates normally beginning with the instruction in absolute location FFFF0H (see Figure 3B). The details of this operation are specified in the Instruction Set description of the MCS-86 Family User's Manual. The RESET input is internally synchronized to the processor clock. At initialization the HIGH-to-LOW transition of RESET must occur no sooner than 50 μs after power-up, to allow complete initialization of the 8086.

NMI may not be asserted prior to the 2nd CLK cycle following the end of RESET.

INTERRUPT OPERATIONS

Interrupt operations fall into two classes; software or hardware initiated. The software initiated interrupts and software aspects of hardware interrupts are specified in the Instruction Set description. Hardware interrupts can be classified as non-maskable or maskable.

Interrupts result in a transfer of control to a new program location. A 256-element table containing address pointers to the interrupt service program locations resides in absolute locations 0 through 3FFH (see Figure 3b), which are reserved for this purpose. Each element in the table is 4 bytes in size and corresponds to an interrupt "type". An interrupting device supplies an 8-bit type number, during the interrupt acknowledge sequence, which is used to "vector" through the appropriate element to the new interrupt service program location.

NON-MASKABLE INTERRUPT (NMI)

The processor provides a single non-maskable interrupt pin (NMI) which has higher priority than the maskable interrupt request pin (INTR). A typical use would be to activate a power failure routine. The NMI is edge-triggered on a LOW-to-HIGH transition. The activation of this pin causes a type 2 interrupt. (See Instruction Set description.)

NMI is required to have a duration in the HIGH state of greater than two CLK cycles, but is not required to be synchronized to the clock. Any high-going transition of NMI is latched on-chip and will be serviced at the end of the current instruction or between whole moves of a block-type instruction. Worst case response to NMI would be for multiply, divide, and variable shift instructions. There is no specification on the occurrence of the low-going edge; it may occur before, during, or after the servicing of NMI. Another high-going edge triggers another response if it occurs after the start of the NMI procedure. The signal must be free of logical spikes in general and be free of bounces on the low-going edge to avoid triggering extraneous responses.

MASKABLE INTERRUPT (INTR)

The 86/10 provides a single interrupt request input (INTR) which can be masked internally by software with the resetting of the interrupt enable FLAG status bit. The interrupt request signal is level triggered. It is internally synchronized during each clock cycle on the high-going edge of CLK. To be responded to, INTR must be present (HIGH) during the clock period preceding the end of the current instruction or the end of a whole move for a block-type instruction. During the interrupt response sequence further interrupts are disabled. The enable bit is reset as part of the response to any interrupt (INTR, NMI, software interrupt or single-step), although the

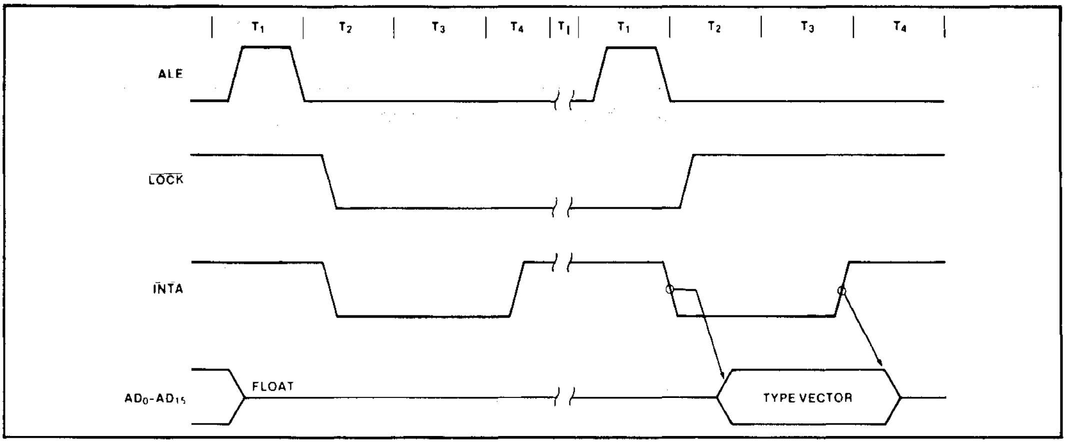


Figure 6. Interrupt Acknowledge Sequence

FLAGS register which is automatically pushed onto the stack reflects the state of the processor prior to the interrupt. Until the old FLAGS register is restored the enable bit will be zero unless specifically set by an instruction.

During the response sequence (figure 6) the processor executes two successive (back-to-back) interrupt acknowledge cycles. The 8086 emits the LOCK signal from T_2 of the first bus cycle until T_2 of the second. A local bus "hold" request will not be honored until the end of the second bus cycle. In the second bus cycle a byte is fetched from the external interrupt system (e.g., 8259A PIC) which identifies the source (type) of the interrupt. This byte is multiplied by four and used as a pointer into the interrupt vector lookup table. An INTR signal left HIGH will be continually responded to within the limitations of the enable bit and sample period. The INTERRUPT RETURN instruction includes a FLAGS pop which returns the status of the original interrupt enable bit when it restores the FLAGS.

HALT

When a software "HALT" instruction is executed the processor indicates that it is entering the "HALT" state in one of two ways depending upon which mode is strapped. In minimum mode, the processor issues one ALE with no qualifying bus control signals. In Maximum Mode, the processor issues appropriate HALT status on $\overline{S_2}\overline{S_1}\overline{S_0}$ and the 8288 bus controller issues one ALE. The 8086 will not leave the "HALT" state when a local bus "hold" is entered while in "HALT". In this case, the processor reissues the HALT indicator. An interrupt request or RESET will force the 8086 out of the "HALT" state.

READ/MODIFY/WRITE (SEMAPHORE) OPERATIONS VIA LOCK

The $\overline{\text{LOCK}}$ status information is provided by the processor when directly consecutive bus cycles are required during the execution of an instruction. This provides the processor with the capability of performing read/modify/write operations on memory (via the Exchange Register With Memory instruction, for example) without the possibility of another system bus master receiving intervening memory cycles. This is useful in multiprocessor system configurations to accomplish "test and set lock" operations. The $\overline{\text{LOCK}}$ signal is activated (forced LOW) in the clock cycle following the one in which the software "LOCK" prefix instruction is decoded by the EU. It is deactivated at the end of the last bus cycle of the instruction following the "LOCK" prefix instruction. While $\overline{\text{LOCK}}$ is active a request on a RQ/GT pin will be recorded and then honored at the end of the LOCK.

EXTERNAL SYNCHRONIZATION VIA TEST

As an alternative to the interrupts and general I/O capabilities, the 8086 provides a single software-testable input known as the $\overline{\text{TEST}}$ signal. At any time the program may execute a WAIT instruction. If at that time the $\overline{\text{TEST}}$ signal is inactive (HIGH), program execution becomes suspended while the processor waits for $\overline{\text{TEST}}$ to become active. It must remain active for at least 5 CLK cycles. The WAIT instruction is re-executed repeatedly until that time. This activity does not consume bus cycles. The processor remains in an idle state while waiting. All 8086 drivers go to 3-state OFF if bus "Hold"is entered. If interrupts are enabled, they may occur while the processor is waiting. When this occurs the processor fetches the WAIT instruction one extra time, processes the interrupt, and then re-fetches and re-executes the WAIT instruction upon returning from the interrupt.

BASIC SYSTEM TIMING

Typical system configurations for the processor operating in minimum mode and in maximum mode are shown in Figures 4a and 4b, respectively. In minimum mode, the MN/$\overline{\text{MX}}$ pin is strapped to V_{CC} and the processor emits bus control signals in a manner similar to the 8085. In maximum mode, the MN/$\overline{\text{MX}}$ pin is strapped to V_{SS} and the processor emits coded status information which the 8288 bus controller uses to generate MULTIBUS compatible bus control signals. Figure 5 illustrates the signal timing relationships.

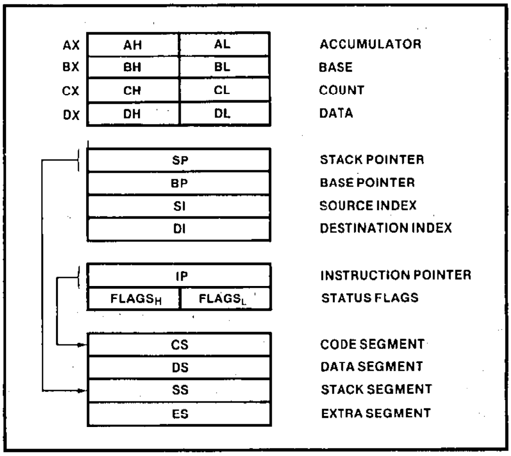


Figure 7. iAPX 86/10 Register Model

SYSTEM TIMING — MINIMUM SYSTEM

The read cycle begins in T_1 with the assertion of the Address Latch Enable (ALE) signal. The trailing (low-going) edge of this signal is used to latch the address information, which is valid on the local bus at this time, into the 8282/8283 latch. The $\overline{\text{BHE}}$ and A_0 signals address the low, high, or both bytes. From T_1 to T_4 the M/$\overline{\text{IO}}$ signal indicates a memory or I/O operation. At T_2 the address is removed from the local bus and the bus goes to a high impedance state. The read control signal is also asserted at T_2. The read ($\overline{\text{RD}}$) signal causes the addressed device to enable its data bus drivers to the local bus. Some time later valid data will be available on the bus and the addressed device will drive the READY line HIGH. When the processor returns the read signal to a HIGH level, the addressed device will again 3-state its bus drivers. If a transceiver (8286/8287) is required to buffer the 8086 local bus, signals DT/$\overline{R}$ and $\overline{DEN}$ are provided by the 8086.

A write cycle also begins with the assertion of ALE and the emission of the address. The M/$\overline{IO}$ signal is again asserted to indicate a memory or I/O write operation. In the T_2 immediately following the address emission the processor emits the data to be written into the addressed location. This data remains valid until the middle of T_4. During T_2, T_3, and T_W the processor asserts the write control signal. The write ($\overline{WR}$) signal becomes active at the beginning of T_2 as opposed to the read which is delayed somewhat into T_2 to provide time for the bus to float.

The $\overline{BHE}$ and A_0 signals are used to select the proper byte(s) of the memory/IO word to be read or written according to the following table:

$\overline{BHE}$	A0	CHARACTERISTICS
0	0	Whole word
0	1	Upper byte from/ to odd address
1	0	Lower byte from/ to even address
1	1	None

I/O ports are addressed in the same manner as memory location. Even addressed bytes are transferred on the D_7-D_0 bus lines and odd addressed bytes on D_{15}-D_8.

The basic difference between the interrupt acknowledge cycle and a read cycle is that the interrupt acknowledge signal ($\overline{INTA}$) is asserted in place of the read ($\overline{RD}$) signal and the address bus is floated. (See Figure 6.) In the second of two successive INTA cycles, a byte of information is read from bus lines D_7-D_0 as supplied by the interrupt system logic (i.e., 8259A Priority Interrupt Controller). This byte identifies the source (type) of the interrupt. It is multiplied by four and used as a pointer into an interrupt vector lookup table, as described earlier.

BUS TIMING—MEDIUM SIZE SYSTEMS

For medium size systems the MN/$\overline{MX}$ pin is connected to V_{SS} and the 8288 Bus Controller is added to the system as well as an 8282/8283 latch for latching the system address, and a 8286/8287 transceiver to allow for bus loading greater than the 8086 is capable of handling. Signals ALE, DEN, and DT/$\overline{R}$ are generated by the 8288 instead of the processor in this configuration although their timing remains relatively the same. The 8086 status outputs ($\overline{S}_2$, $\overline{S}_1$, and $\overline{S}_0$) provide type-of-cycle information and become 8288 inputs. This bus cycle information specifies read (code, data, or I/O), write (data or I/O), interrupt acknowledge, or software halt. The 8288 thus issues control signals specifying memory read or write, I/O read or write, or interrupt acknowledge. The 8288 provides two types of write strobes, normal and advanced, to be applied as required. The normal write strobes have data valid at the leading edge of write. The advanced write strobes have the same timing as read strobes, and hence data isn't valid at the leading edge of write. The 8286/8287 transceiver receives the usual T and OE inputs from the 8288's DT/$\overline{R}$ and DEN.

The pointer into the interrupt vector table, which is passed during the second INTA cycle, can derive from an 8259A located on either the local bus or the system bus. If the master 8259A Priority Interrupt Controller is positioned on the local bus, a TTL gate is required to disable the 8286/8287 transceiver when reading from the master 8259A during the interrupt acknowledge sequence and software "poll".

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias.........0°C to 70°C
Storage Temperature.............−65°C to +150°C
Voltage on Any Pin with
Respect to Ground..................−1.0 to +7V
Power Dissipation.........................2.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS

(8086: T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)
(8086-1: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)
(8086-2: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}+0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL}=2.5 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH}= −400 μA
I_{CC}	Power Supply Current: 8086 8086-1 8086-2		340 360 350	mA	T_A=25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC}+1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except AD_0-AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc=1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0-AD_{15}, $\overline{RQ}/\overline{GT}$)		15	pF	fc=1 MHz

A.C. CHARACTERISTICS (8086: T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)
(8086-1: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)
(8086-2: T_A = 0°C to 70°C, V_{CC} = 5V ± 5%)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL)−15		(⅔ TCLCL)−14		(⅔ TCLCL)−15		ns	
TCHCL	CLK High Time	(⅓ TCLCL)+2		(⅓ TCLCL)+6		(⅓ TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data in Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (See Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (See Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	(⅔ TCLCL)−15		53		(⅔ TCLCL)−15		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		−10		−8		ns	
THVCH	HOLD Setup Time	35		20		20		ns	
TINVCH	INTR, NMI, $\overline{\text{TEST}}$ Setup Time (See Note 2)	30		15		15		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

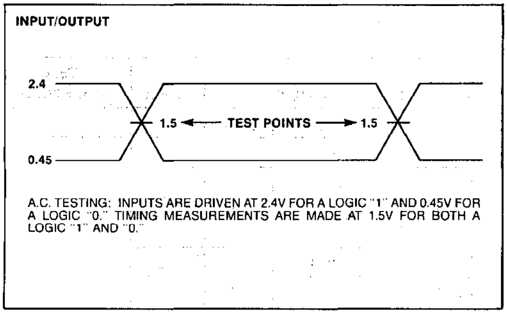
TIMING RESPONSES

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	*C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH−20		TCLCH−10		TCLCH-10		ns	
TCLLH	ALE Active Delay		80		40		50	ns	
TCHLL	ALE Inactive Delay		85		45		55	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL−10		TCHCL−10		TCHCL−10		ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	
TCHDX	Data Hold Time	10		10		10		ns	
TWHDX	Data Hold Time After WR	TCLCH−30		TCLCH−25		TCLCH−30		ns	
TCVCTV	Control Active Delay 1	10	110	10	50	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	45	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	50	10	70	ns	
TAZRL	Address Float to READ Active	0		0		0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	10	70	10	100	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	10	60	10	80	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL−45		TCLCL−35		TCLCL−40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	60	10	100	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL−75		2TCLCL−40		2TCLCL−50		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL−60		2TCLCL−35		2TCLCL−40		ns	
TAVAL	Address Valid to ALE Low	TCLCH−60		TCLCH−35		TCLCH−40		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

NOTES:

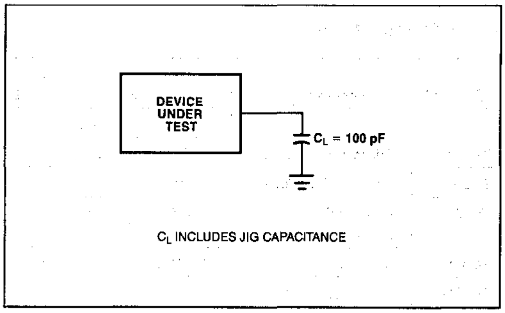
1. Signal at 8284A shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state. (8 ns into T3).

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

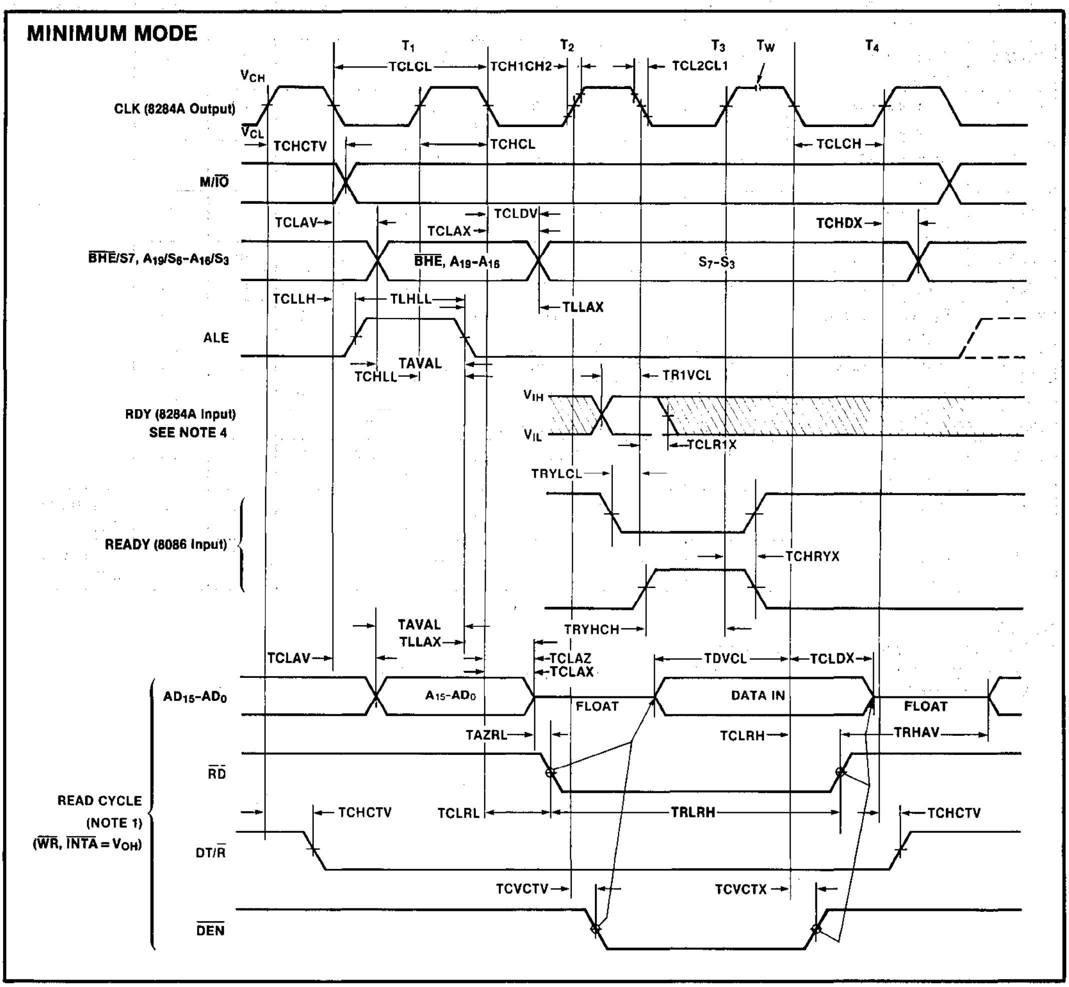
A.C. TESTING LOAD CIRCUIT



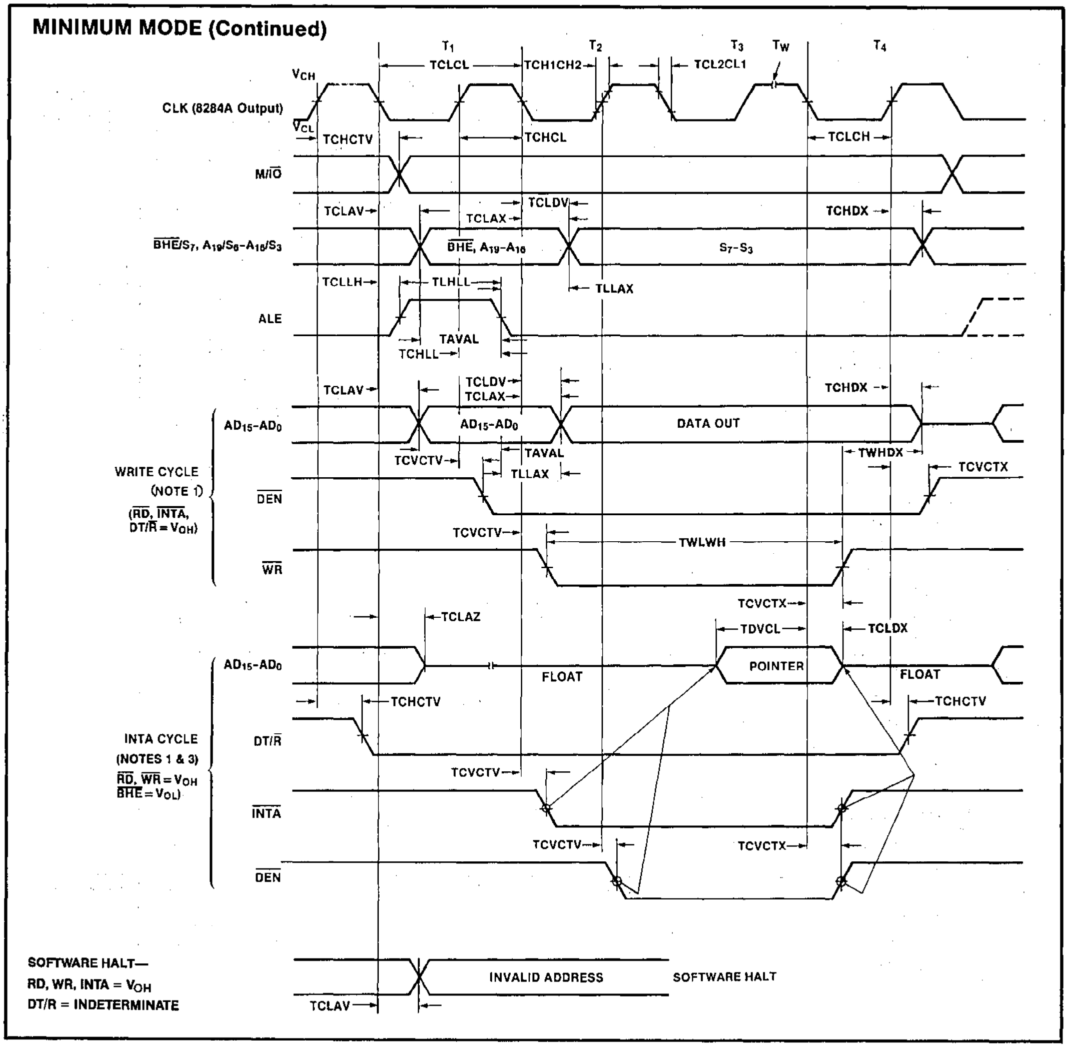
C_L INCLUDES JIG CAPACITANCE

WAVEFORMS

MINIMUM MODE



WAVEFORMS (Continued)



NOTES:

1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control signals shown for second INTA cycle.
4. Signals at 8284A are shown for reference only.
5. All timing measurements are máde at 1.5V unless otherwise noted.

A.C. CHARACTERISTICS

MAX MODE SYSTEM (USING 8288 BUS CONTROLLER) TIMING REQUIREMENTS

Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period	200	500	100	500	125	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL)−15		(⅔ TCLCL)−14		(⅔ TCLCL)−15		ns	
TCHCL	CLK High Time	(⅓ TCLCL)+2		(⅓ TCLCL)+6		(⅓ TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		5		20		ns	
TCLDX	Data In Hold Time	10		10		10		ns	
TR1VCL	RDY Setup Time into 8284A (See Notes 1, 2)	35		35		35		ns	
TCLR1X	RDY Hold Time into 8284A (See Notes 1, 2)	0		0		0		ns	
TRYHCH	READY Setup Time into 8086	(⅔ TCLCL)−15		53		(⅔ TCLCL)−15		ns	
TCHRYX	READY Hold Time into 8086	30		20		20		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	−8		−10		−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, $\overline{TEST}$) (See Note 2)	30		15		15		ns	
TGVCH	$\overline{RQ}$/$\overline{GT}$ Setup Time	30		12		15		ns	
TCHGX	$\overline{RQ}$ Hold Time into 8086	40		20		30		ns	
TILIH	Input Rise Time (Except CLK)		20		20		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12		12		12	ns	From 2.0V to 0.8V

NOTES:

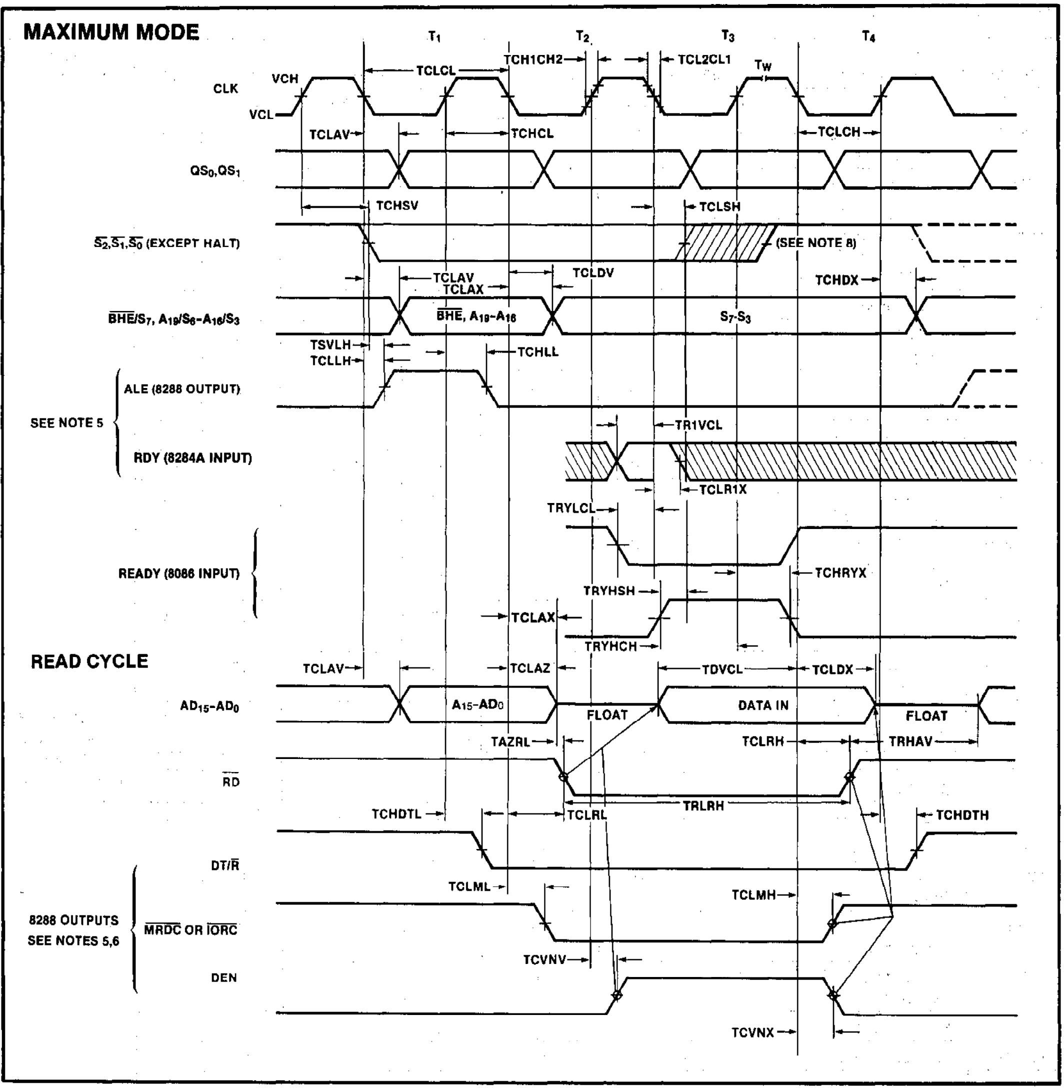
1. Signal at 8284A or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T3 and wait states.
4. Applies only to T2 state (8 ns into T3).

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

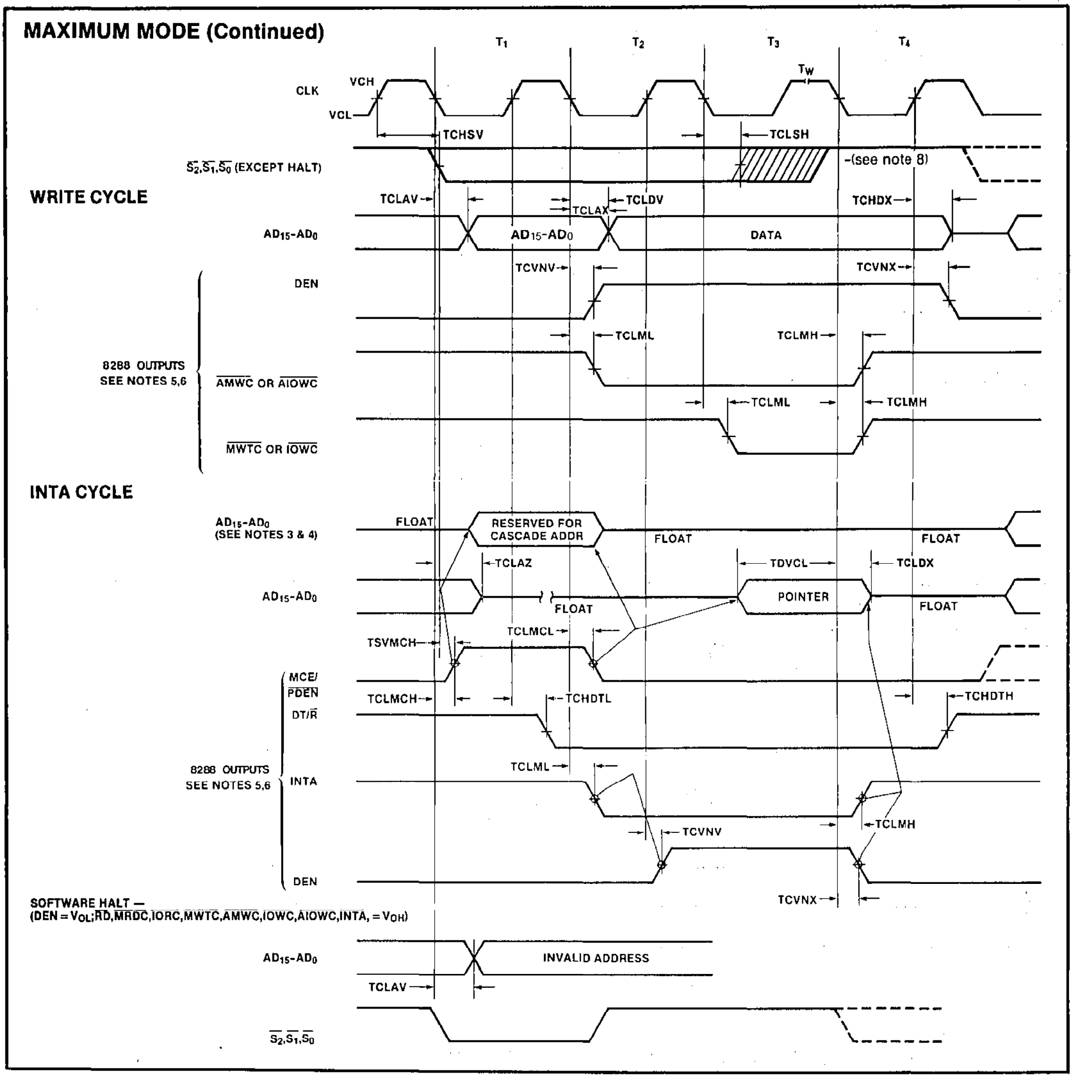
Symbol	Parameter	8086		8086-1 (Preliminary)		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TCLML	Command Active Delay (See Note 1)	10	35	10	35	10	35	ns	C_L = 20-100 pF for all 8086 Outputs (In addition to 8086 self-load)
TCLMH	Command Inactive Delay (See Note 1)	10	35	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110		45		65	ns	
TCHSV	Status Active Delay	10	110	10	45	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	55	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	50	10	60	ns	
TCLAX	Address Hold Time	10		10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	10	40	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15		15		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15		15		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15		15		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15		15		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15		15		15	ns	
TCLMCL	MCE Inactive Delay (See Note 1)		15		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	50	10	60	ns	
TCHDX	Data Hold Time	10		10		10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	5	45	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	10	45	10	45	ns	
TAZRL	Address Float to Read Active	0		0		0		ns	
TCLRL	RD Active Delay	10	165	10	70	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	60	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL−45		TCLCL−35		TCLCL−40		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50		50		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30		30		30	ns	
TCLGL	GT Active Delay	0	85	0	45	0	50	ns	
TCLGH	GT Inactive Delay	0	85	0	45	0	50	ns	
TRLRH	RD Width	2TCLCL−75		2TCLCL−40		2TCLCL−50		ns	
TOLOH	Output Rise Time		20		20		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12		12		12	ns	From 2.0V to 0.8V

WAVEFORMS


MAXIMUM MODE
T1
T2
T3
T4
TW
TCLCL
TCH1CH2
TCL2CL1
CLK
VCH
VCL
TCLAV
TCHCL
TCLCH
QS0,QS1
TCHSV
TCLSH
$\overline{S_2},\overline{S_1},\overline{S_0}$ (EXCEPT HALT)
(SEE NOTE 8)
TCLAV
TCLAX
TCLDV
TCHDX
$\overline{BHE}$/S7, A19/S6-A16/S3
$\overline{BHE}$, A19-A16
S7-S3
TSVLH
TCLLH
TCHLL
SEE NOTE 5
ALE (8288 OUTPUT)
RDY (8284A INPUT)
TR1VCL
TCLR1X
TRYLCL
READY (8086 INPUT)
TCHRYX
TRYHSH
TCLAX
TRYHCH
READ CYCLE
TCLAV
TCLAZ
TDVCL
TCLDX
AD15-AD0
A15-AD0
FLOAT
DATA IN
FLOAT
TAZRL
TCLRH
TRHAV
$\overline{RD}$
TRLRH
TCHDTL
TCLRL
TCHDTH
DT/$\overline{R}$
TCLML
TCLMH
8288 OUTPUTS
SEE NOTES 5,6
$\overline{MRDC}$ OR $\overline{IORC}$
TCVNV
DEN
TCVNX

WAVEFORMS (Continued)

MAXIMUM MODE (Continued)

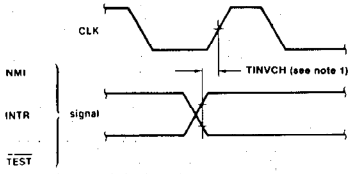


NOTES:

1. All signals switch between V_{OH} and V_{OL} unless otherwise specified.
2. RDY is sampled near the end of T_2, T_3, T_W to determine if T_W machines states are to be inserted.
3. Cascade address is valid between first and second INTA cycle.
4. Two INTA cycles run back-to-back. The 8086 LOCAL ADDR/DATA BUS is floating during both INTA cycles. Control for pointer address is shown for second INTA cycle.
5. Signals at 8284A or 8288 are shown for reference only.
6. The issuance of the 8288 command and control signals ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ and DEN) lags the active high 8288 CEN.
7. All timing measurements are made at 1.5V unless otherwise noted.
8. Status inactive in state just prior to T_4.

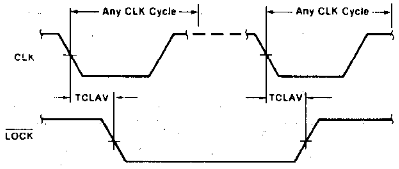
WAVEFORMS (Continued)

ASYNCHRONOUS SIGNAL RECOGNITION

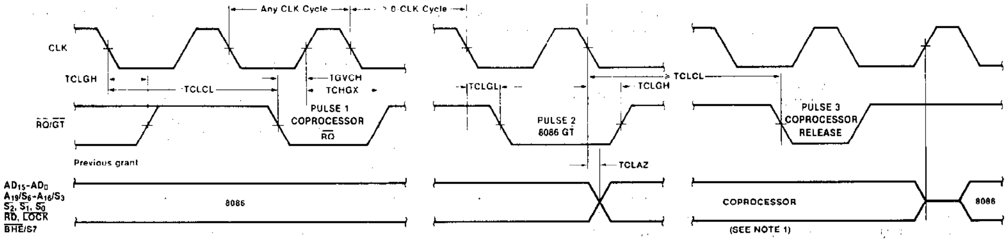


NOTE: 1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK

BUS LOCK SIGNAL TIMING (MAXIMUM MODE ONLY)



REQUEST/GRANT SEQUENCE TIMING (MAXIMUM MODE ONLY)



NOTES: 1. THE COPROCESSOR MAY NOT DRIVE THE BUSES OUTSIDE THE REGION SHOWN WITHOUT RISKING CONTENTION.

HOLD/HOLD ACKNOWLEDGE TIMING (MINIMUM MODE ONLY)

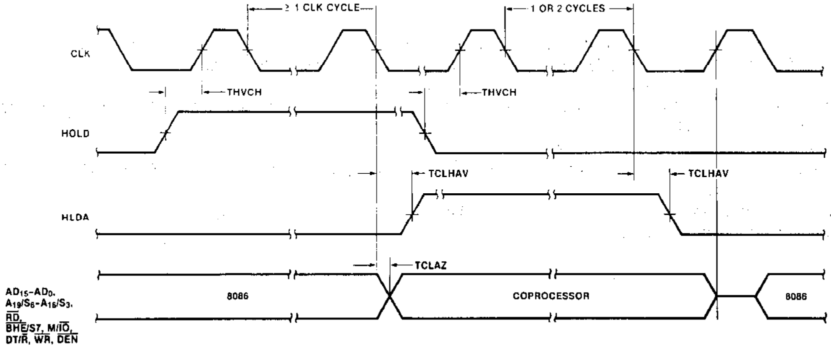


Table 2. Instruction Set Summary

DATA TRANSFER

MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate to register	1 0 1 1 w reg	data	data if w 1	
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with accumulator	1 0 0 1 0 reg			
IN=Input from:				
Fixed port	1 1 1 0 0 1 0 w	port		
Variable port	1 1 1 0 1 1 0 w			
OUT = Output to:				
Fixed port	1 1 1 0 0 1 1 w	port		
Variable port	1 1 1 0 1 1 1 w			
XLAT=Translate byte to AL	1 1 0 1 0 1 1 1			
LEA=Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS=Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES=Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF=Load AH with flags	1 0 0 1 1 1 1 1			
SAHF=Store AH into flags	1 0 0 1 1 1 1 0			
PUSHF=Push flags	1 0 0 1 1 1 0 0			
POPF=Pop flags	1 0 0 1 1 1 0 1			
ARITHMETIC				
ADD = Add:				
Reg./memory with register to either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s w-01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w 1	
ADC = Add with carry:				
Reg./memory with register to either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s w=01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w=1	
INC = Increment:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA=ASCII adjust for add	0 0 1 1 0 1 1 1			
DAA=Decimal adjust for add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./memory and register to either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s:w=01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w=1	
SBB = Subtract with borrow				
Reg./memory and register to either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w=1	

DEC Decrement:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP Compare:				
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w 01
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w 1	
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1			
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1			
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW Convert byte to word	1 0 0 1 1 0 0 0			
CWD Convert word to double word	1 0 0 1 1 0 0 1			
LOGIC				
NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL Shift logical/arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND And:				
Reg./memory and register to either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w 1	
TEST And function to flags, no result:				
Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w 1	
OR Or:				
Reg./memory and register to either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w 1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w 1	
XOR = Exclusive or:				
Reg./memory and register to either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w 1	
STRING MANIPULATION				
REP=Repeat	1 1 1 1 0 0 1 z			
MOVS=Move byte/word	1 0 1 0 0 1 0 w			
CMPS=Compare byte/word	1 0 1 0 0 1 1 w			
SCAS=Scan byte/word	1 0 1 0 1 1 1 w			
LODS=Load byte/wd to AL/AX	1 0 1 0 1 1 0 w			
STOS=Stor byte/wd from AL/A	1 0 1 0 1 0 1 w			

Mnemonics ©Intel, 1978

Table 2. Instruction Set Summary (Continued)

CONTROL TRANSFER			
CALL = Call:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	
JMP = Unconditional Jump:			
Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment-short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	
RET = Return from CALL:			
Within segment	1 1 0 0 0 0 1 1		
Within seg. adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment, adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ=Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE=Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG=Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE=Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA=Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE=Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO=Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS=Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ=Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE=Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG=Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNB/JAE=Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp
JNBE/JA=Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp
JNP/JPO=Jump on not par/par odd	0 1 1 1 1 0 1 1	disp
JNO=Jump on not overflow	0 1 1 1 0 0 0 1	disp
JNS=Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE=Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ=Jump on CX zero	1 1 1 0 0 0 1 1	disp
INT Interrupt		
Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO=Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET=Interrupt return	1 1 0 0 1 1 1 1	
PROCESSOR CONTROL		
CLC Clear carry	1 1 1 1 1 0 0 0	
CMC Complement carry	1 1 1 1 0 1 0 1	
STC Set carry	1 1 1 1 1 0 0 1	
CLD Clear direction	1 1 1 1 1 1 0 0	
STD Set direction	1 1 1 1 1 1 0 1	
CLI Clear interrupt	1 1 1 1 1 0 1 0	
STI Set interrupt	1 1 1 1 1 0 1 1	
HLT Halt	1 1 1 1 0 1 0 0	
WAIT Wait	1 0 0 1 1 0 1 1	
ESC Escape (to external device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK Bus lock prefix	1 1 1 1 0 0 0 0	

Footnotes:

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value.
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

if s:w = 01 then 16 bits of immediate data form the operand.
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand.
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with Z.F FLAG.

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

MILITARY iAPX 86/10
16-BIT HMOS MICROPROCESSOR

(M8086)
MILITARY

- **Direct Addressing Capability to 1 MByte of Memory**
- **Assembly Language Compatible with 8080/8085**
- **14 Word, By 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8-and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **5 MHz Clock Rate**
- **MULTIBUS™ System Compatible Interface**
- **Military Temperature Range: −55°C to +125°C**

The Intel® Military iAPX 86/10 is a new generation, high performance 16-bit microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance.

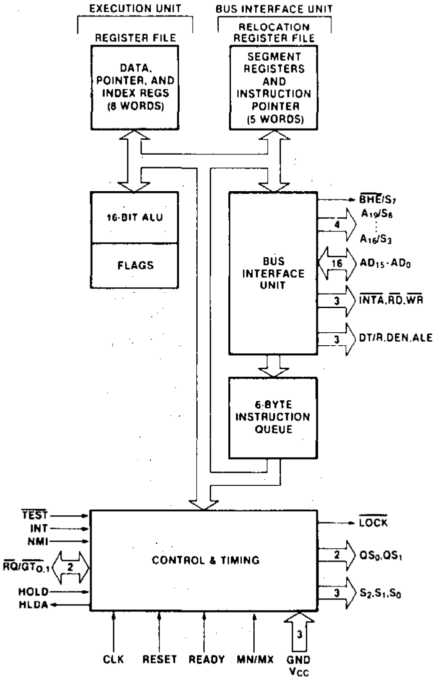


Figure 1. Functional Block Diagram

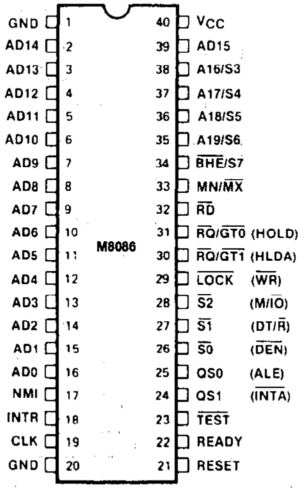


Figure 2. Pin Configuration

Intel Corporation Assumes No Responsibilty for the Use of Any Circuitry Other Than Circuitry Embodied in an Intel Product. No Other Circuit Patent Licenses are Implied.

I8086
16-BIT HMOS MICROPROCESSOR
INDUSTRIAL

- **Industrial Grade Temperature Range: −40°C to +85°C**
- **Direct Addressing Capability to 1 MByte of Memory**
- **Assembly Language Compatible with MCS-80,85®**
- **14 Word, By 16-Bit General Register Set**
- **24 Operand Addressing Modes**
- **Bit, Byte, Word, and Block Operations**
- **8- and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal Including Multiply and Divide**
- **5 MHz Clock Rate**
- **MULTIBUS™ System Compatible Interface**

The Intel® Industrial iAPX 86/10 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It addresses memory as a sequence of 8-bit bytes, but has a 16-bit wide physical path to memory for high performance.

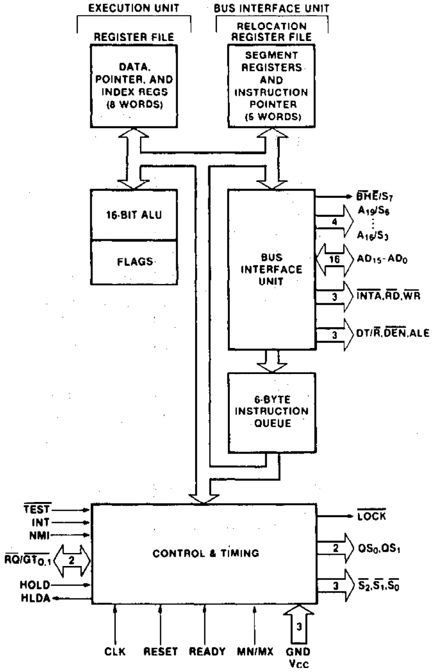


Figure 1. I8086 CPU Functional Block Diagram

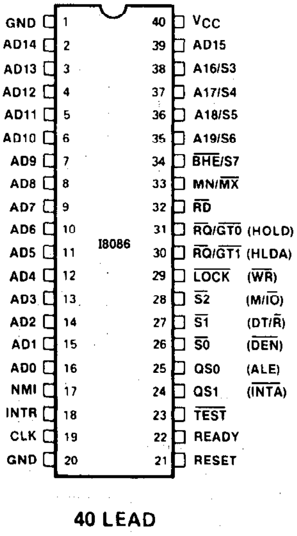


Figure 2. I8086 Pin Diagram

PRELIMINARY

iAPX 88/10
(8088)
8-BIT HMOS MICROPROCESSOR

- **8-Bit Data Bus Interface**
- **16-Bit Internal Architecture**
- **Direct Addressing Capability to 1 Mbyte of Memory**
- **Direct Software Compatibility with iAPX 86/10 (8086 CPU)**
- **14-Word by 16-Bit Register Set with Symmetrical Operations**
- **24 Operand Addressing Modes**
- **Byte, Word, and Block Operations**
- **8-Bit and 16-Bit Signed and Unsigned Arithmetic in Binary or Decimal, Including Multiply and Divide**
- **Compatible with 8155-2, 8755A-2 and 8185-2 Multiplexed Peripherals**

The Intel® iAPX 88/10 is a new generation, high performance microprocessor implemented in N-channel, depletion load, silicon gate technology (HMOS), and packaged in a 40-pin CerDIP package. The processor has attributes of both 8- and 16-bit microprocessors. It is directly compatible with iAPX 86/10 software and 8080/8085 hardware and peripherals.

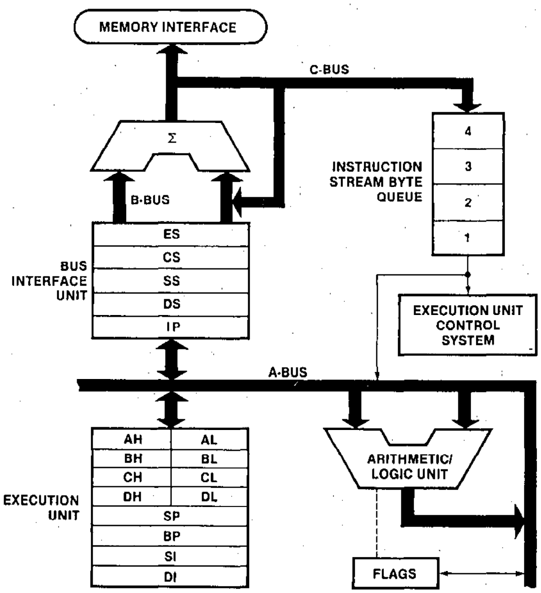


Figure 1. iAPX 88/10 CPU Functional Block Diagram

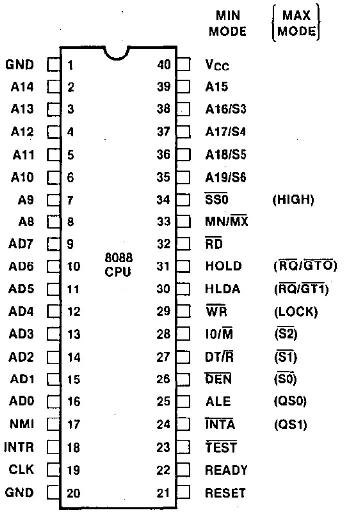


Figure 2. iAPX 88/10 Pin Configuration

Table 1. Pin Description

The following pin function descriptions are for 8088 systems in either minimum or maximum mode. The "local bus" in these descriptions is the direct multiplexed bus interface connection to the 8088 (without regard to additional bus buffers).

<table>
<tr><th>Symbol</th><th>Pin No.</th><th>Type</th><th>Name and Function</th></tr>
<tr><td>AD7–AD0</td><td>9-16</td><td>I/O</td><td><b>Address Data Bus:</b> These lines constitute the time multiplexed memory/IO address (T1) and data (T2, T3, Tw, and T4) bus. These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge".</td></tr>
<tr><td>A15–A8</td><td>2-8, 39</td><td>O</td><td><b>Address Bus:</b> These lines provide address bits 8 through 15 for the entire bus cycle (T1–T4). These lines do not have to be latched by ALE to remain valid. A15–A8 are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus "hold acknowledge".</td></tr>
<tr><td>A19/S6, A18/S5, A17/S4, A16/S3</td><td>34-38</td><td>O</td><td><b>Address/Status:</b> During T1, these are the four most significant address lines for memory operations. During I/O operations, these lines are LOW. During memory and I/O operations, status information is available on these lines during T2, T3, Tw, and T4. S6 is always low. The status of the interrupt enable flag bit (S5) is updated at the beginning of each clock cycle. S4 and S3 are encoded as shown.
<table>
<tr><th>S4</th><th>S3</th><th>CHARACTERISTICS</th></tr>
<tr><td>0 (LOW)</td><td>0</td><td>Alternate Data</td></tr>
<tr><td>0</td><td>1</td><td>Stack</td></tr>
<tr><td>1 (HIGH)</td><td>0</td><td>Code or None</td></tr>
<tr><td>1</td><td>1</td><td>Data</td></tr>
<tr><td>S6 is 0 (LOW)</td><td></td><td></td></tr>
</table>

This information indicates which segment register is presently being used for data accessing.
These lines float to 3-state OFF during local bus "hold acknowledge".</td></tr>
<tr><td>$\overline{RD}$</td><td>32</td><td>O</td><td><b>Read:</b> Read strobe indicates that the processor is performing a memory or I/O read cycle, depending on the state of the IO/$\overline{M}$ pin or S2. This signal is used to read devices which reside on the 8088 local bus. $\overline{RD}$ is active LOW during T2, T3 and Tw of any read cycle, and is guaranteed to remain HIGH in T2 until the 8088 local bus has floated.
This signal floats to 3-state OFF in "hold acknowledge".</td></tr>
<tr><td>READY</td><td>22</td><td>I</td><td><b>READY:</b> is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The RDY signal from memory or I/O is synchronized by the 8284 clock generator to form READY. This signal is active HIGH. The 8088 READY input is not synchronized. Correct operation is not guaranteed if the set up and hold times are not met.</td></tr>
<tr><td>INTR</td><td>18</td><td>I</td><td><b>Interrupt Request:</b> is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.</td></tr>
<tr><td>$\overline{TEST}$</td><td>23</td><td>I</td><td><b>$\overline{TEST}$:</b> input is examined by the "wait for test" instruction. If the $\overline{TEST}$ input is LOW, execution continues, otherwise the processor waits in an "idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.</td></tr>
<tr><td>NMI</td><td>17</td><td>I</td><td><b>Non-Maskable Interrupt:</b> is an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from a LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.</td></tr>
</table>

Table 1. Pin Description (Continued)

Symbol	Pin No.	Type	Name and Function
RESET	21	I	**RESET:** causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the instruction set description, when RESET returns LOW. RESET is internally synchronized.
CLK	19	I	**Clock:** provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}	40		V_{CC}: is the +5V ±10% power supply pin.
GND	1, 20		**GND:** are the ground pins.
MN/$\overline{\text{MX}}$	33	I	**Minimum/Maximum:** indicates what mode the processor is to operate in. The two modes are discussed in the following sections.

The following pin function descriptions are for the 8088 minimum mode (i.e., MN/MX = V_{CC}). Only the pin functions which are unique to minimum mode are described; all other pin functions are as described above.

IO/$\overline{\text{M}}$	28	O	**Status Line:** is an inverted maximum mode $\overline{\text{S2}}$. It is used to distinguish a memory access from an I/O access. IO/$\overline{\text{M}}$ becomes valid in the T4 preceding a bus cycle and remains valid until the final T4 of the cycle (I/O=HIGH, M=LOW). IO/$\overline{\text{M}}$ floats to 3-state OFF in local bus "hold acknowledge".
$\overline{\text{WR}}$	29	O	**Write:** strobe indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the IO/$\overline{\text{M}}$ signal. WR is active for T2, T3, and Tw of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".
$\overline{\text{INTA}}$	24	O	**INTA:** is used as a read strobe for interrupt acknowledge cycles. It is active LOW during T2, T3, and Tw of each interrupt acknowledge cycle.
ALE	25	O	**Address Latch Enable:** is provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during clock low of T1 of any bus cycle. Note that ALE is never floated.
DT/$\overline{\text{R}}$	27	O	**Data Transmit/Receive:** is needed in a minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically, DT/$\overline{\text{R}}$ is equivalent to $\overline{\text{S1}}$ in the maximum mode, and its timing is the same as for IO/$\overline{\text{M}}$ (T=HIGH, R=LOW). This signal floats to 3-state OFF in local "hold acknowledge".
$\overline{\text{DEN}}$	26	O	**Data Enable:** is provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{\text{DEN}}$ is active LOW during each memory and I/O access, and for $\overline{\text{INTA}}$ cycles. For a read or $\overline{\text{INTA}}$ cycle, it is active from the middle of T2 until the middle of T4, while for a write cycle, it is active from the beginning of T2 until the middle of T4. $\overline{\text{DEN}}$ floats to 3-state OFF during local bus "hold acknowledge".
HOLD, HLDA	30,31	I, O	**HOLD:** indicates that another master is requesting a local bus "hold". To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement, in the middle of a T4 or TI clock cycle. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor lowers HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines. Hold is not an asynchronous input. External synchronization should be provided if the system cannot otherwise guarantee the set up time.
$\overline{\text{SSO}}$	34	O	**Status line:** is logically equivalent to $\overline{\text{SO}}$ in the maximum mode. The combination of $\overline{\text{SSO}}$, IO/$\overline{\text{M}}$ and DT/$\overline{\text{R}}$ allows the system to completely decode the current bus cycle status.

IO/$\overline{\text{M}}$	DT/$\overline{\text{R}}$	$\overline{\text{SSO}}$	CHARACTERISTICS
1 (HIGH)	0	0	Interrupt Acknowledge
1	0	1	Read I/O port
1	1	0	Write I/O port
1	1	1	Halt
0 (LOW)	0	0	Code access
0	0	1	Read memory
0	1	0	Write memory
0	1	1	Passive

Table 1. Pin Description (Continued)

The following pin function descriptions are for the 8088, 8228 system in maximum mode (i.e., MN/MX=GND.) Only the pin functions which are unique to maximum mode are described; all other pin functions are as described above.

<table>
<tr><th>Symbol</th><th>Pin No.</th><th>Type</th><th>Name and Function</th></tr>
<tr><td>$\overline{S2}$, $\overline{S1}$, $\overline{S0}$</td><td>26-28</td><td>O</td><td>**Status:** is active during clock high of T4, T1, and T2, and is returned to the passive state (1,1,1) during T3 or during Tw when READY is HIGH. This status is used by the 8288 bus controller to generate all memory and I/O access control signals. Any change by $\overline{S2}$, $\overline{S1}$, or $\overline{S0}$ during T4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T3 or Tw is used to indicate the end of a bus cycle.
<table>
<tr><th>$\overline{S2}$</th><th>$\overline{S1}$</th><th>$\overline{S0}$</th><th>CHARACTERISTICS</th></tr>
<tr><td>0 (LOW)</td><td>0</td><td>0</td><td>Interrupt Acknowledge</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>Read I/O port</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>Write I/O port</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>Halt</td></tr>
<tr><td>1 (HIGH)</td><td>0</td><td>0</td><td>Code access</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>Read memory</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>Write memory</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>Passive</td></tr>
</table>
These signals float to 3-state OFF during "hold acknowledge". During the first clock cycle after RESET becomes active, these signals are active HIGH. After this first clock, they float to 3-state OFF.</td></tr>
<tr><td>$\overline{RQ}/\overline{GT0}$,
$\overline{RQ}/\overline{GT1}$</td><td>30, 31</td><td>I/O</td><td>**Request/Grant:** pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT0}$ having higher priority than $\overline{RQ}/\overline{GT1}$. $\overline{RQ}/\overline{GT}$ has an internal pull-up resistor, so may be left unconnected. The request/grant sequence is as follows (See Figure 8):
1. A pulse of one CLK wide from another local bus master indicates a local bus request ("hold") to the 8088 (pulse 1).
2. During a T4 or TI clock cycle, a pulse one clock wide from the 8088 to the requesting master (pulse 2), indicates that the 8088 has allowed the local bus to float and that it will enter the "hold acknowledge" state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during "hold acknowledge". The same rules as for HOLD/HOLDA apply as for when the bus is released.
3. A pulse one CLK wide from the requesting master indicates to the 8088 (pulse 3) that the "hold" request is about to end and that the 8088 can reclaim the local bus at the next CLK. The CPU then enters T4.
Each master-master exchange of the local bus is a sequence of three pulses. There must be one idle CLK cycle after each bus exchange. Pulses are active LOW.
If the request is made while the CPU is performing a memory cycle, it will release the local bus during T4 of the cycle when all the following conditions are met:
1. Request occurs on or before T2.
2. Current cycle is not the low bit of a word.
3. Current cycle is not the first acknowledge of an interrupt acknowledge sequence.
4. A locked instruction is not currently executing.
If the local bus is idle when the request is made the two possible events will follow:
1. Local bus will be released during the next clock.
2. A memory cycle will start within 3 clocks. Now the four rules for a currently active memory cycle apply with condition number 1 already satisfied.</td></tr>
</table>

Table 1. Pin Description (Continued)

<table>
<tr><th>Symbol</th><th>Pin No.</th><th>Type</th><th>Name and Function</th></tr>
<tr><td>$\overline{\text{LOCK}}$</td><td>29</td><td>O</td><td><b>$\overline{\text{LOCK}}$:</b> indicates that other system bus masters are not to gain control of the system bus while $\overline{\text{LOCK}}$ is active (LOW). The $\overline{\text{LOCK}}$ signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state off in "hold acknowledge".</td></tr>
<tr><td>QS1, QS0</td><td>24, 25</td><td>O</td><td><b>Queue Status:</b> provide status to allow external tracking of the internal 8088 instruction queue.
The queue status is valid during the CLK cycle after which the queue operation is performed.
<table><tr><th>QS1</th><th>QS0</th><th>CHARACTERISTICS</th></tr><tr><td>0 (LOW)</td><td>0</td><td>No operation</td></tr><tr><td>0</td><td>1</td><td>First byte of opcode from queue</td></tr><tr><td>1 (HIGH)</td><td>0</td><td>Empty the queue</td></tr><tr><td>1</td><td>1</td><td>Subsequent byte from queue</td></tr></table></td></tr>
<tr><td>—</td><td>34</td><td>O</td><td>Pin 34 is always high in the maximum mode.</td></tr>
</table>

FUNCTIONAL DESCRIPTION

Memory Organization

The processor provides a 20-bit address to memory which locates the byte being referenced. The memory is organized as a linear array of up to 1 million bytes, addressed as 00000(H) to FFFFF(H). The memory is logically divided into code, data, extra data, and stack segments of up to 64K bytes each, with each segment falling on 16-byte boundaries. (See Figure 3.)

All memory references are made relative to base addresses contained in high speed segment registers. The segment types were chosen based on the addressing needs of programs. The segment register to be selected is automatically chosen according to the rules of the following table. All information in one segment type share the same logical attributes (e.g. code or data). By structuring memory into relocatable areas of similar characteristics and by automatically selecting segment registers, programs are shorter, faster, and more structured.

Word (16-bit) operands can be located on even or odd address boundaries. For address and data operands, the least significant byte of the word is stored in the lower valued address location and the most significant byte in the next higher address location. The BIU will automatically execute two fetch or write cycles for 16-bit operands.

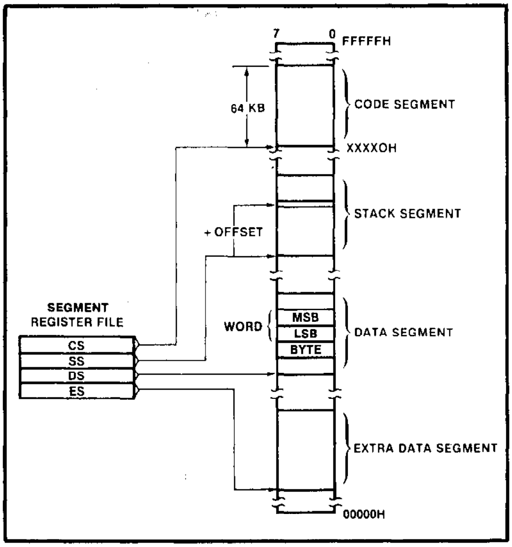


Figure 3. Memory Organization

Certain locations in memory are reserved for specific CPU operations. (See Figure 4.) Locations from addresses FFFF0H through FFFFFH are reserved for operations including a jump to the initial system initialization routine. Following RESET, the CPU will always begin execution at location FFFF0H where the jump must be located. Locations 00000H through 003FFH are reserved for interrupt operations. Four-byte pointers consisting of a 16-bit segment address and a 16-bit offset address direct program flow to one of the 256 possible interrupt service routines. The pointer elements are assumed to have been stored at their respective places in reserved memory prior to the occurrence of interrupts.

Minimum and Maximum Modes

The requirements for supporting minimum and maximum 8088 systems are sufficiently different that they cannot be done efficiently with 40 uniquely defined pins. Consequently, the 8088 is equipped with a strap pin (MN/$\overline{\text{MX}}$) which defines the system configuration. The definition of a certain subset of the pins changes, dependent on the condition of the strap pin. When the MN/$\overline{\text{MX}}$ pin is strapped to GND, the 8088 defines pins 24 through 31 and 34 in maximum mode. When the MN/$\overline{\text{MX}}$ pin is strapped to V_{CC}, the 8088 generates bus control signals itself on pins 24 through 31 and 34.

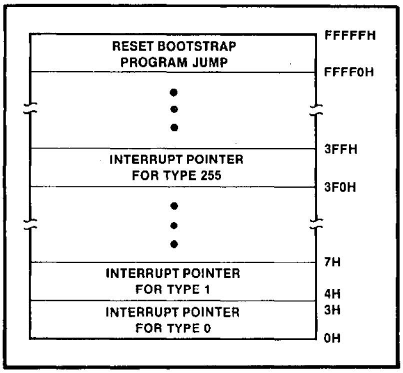

RESET BOOTSTRAP PROGRAM JUMP
FFFFFH
FFFF0H
INTERRUPT POINTER FOR TYPE 255
3FFH
3F0H
INTERRUPT POINTER FOR TYPE 1
7H
4H
INTERRUPT POINTER FOR TYPE 0
3H
0H

Figure 4. Reserved Memory Locations

Memory Reference Need	Segment Register Used	Segment Selection Rule
Instructions	CODE (CS)	Automatic with all instruction prefetch.
Stack	STACK (SS)	All stack pushes and pops. Memory references relative to BP base register except data references.
Local Data	DATA (DS)	Data references when: relative to stack, destination of string operation, or explicitly overridden.
External (Global) Data	EXTRA (ES)	Destination of string operations: Explicitly selected using a segment override.

The minimum mode 8088 can be used with either a multiplexed or demultiplexed bus. The multiplexed bus configuration is compatible with the MCS-85™ multiplexed bus peripherals (8155, 8156, 8355, 8755A, and 8185). This configuration (See Figure 5) provides the user with a minimum chip count system. This architecture provides the 8088 processing power in a highly integrated form.

The demultiplexed mode requires one latch (for 64K addressability) or two latches (for a full megabyte of addressing). A third latch can be used for buffering if the address bus loading requires it. An 8286 or 8287 transceiver can also be used if data bus buffering is required. (See Figure 6.) The 8088 provides $\overline{\text{DEN}}$ and DT/$\overline{\text{R}}$ to control the transceiver, and ALE to latch the addresses. This configuration of the minimum mode provides the standard demultiplexed bus structure with heavy bus buffering and relaxed bus timing requirements.

The maximum mode employs the 8288 bus controller. (See Figure 7.) The 8288 decodes status lines $\overline{\text{S0}}$, $\overline{\text{S1}}$, and $\overline{\text{S2}}$, and provides the system with all bus control signals. Moving the bus control to the 8288 provides better source and sink current capability to the control lines, and frees the 8088 pins for extended large system features. Hardware lock, queue status, and two request/grant interfaces are provided by the 8088 in maximum mode. These features allow co-processors in local bus and remote bus configurations.

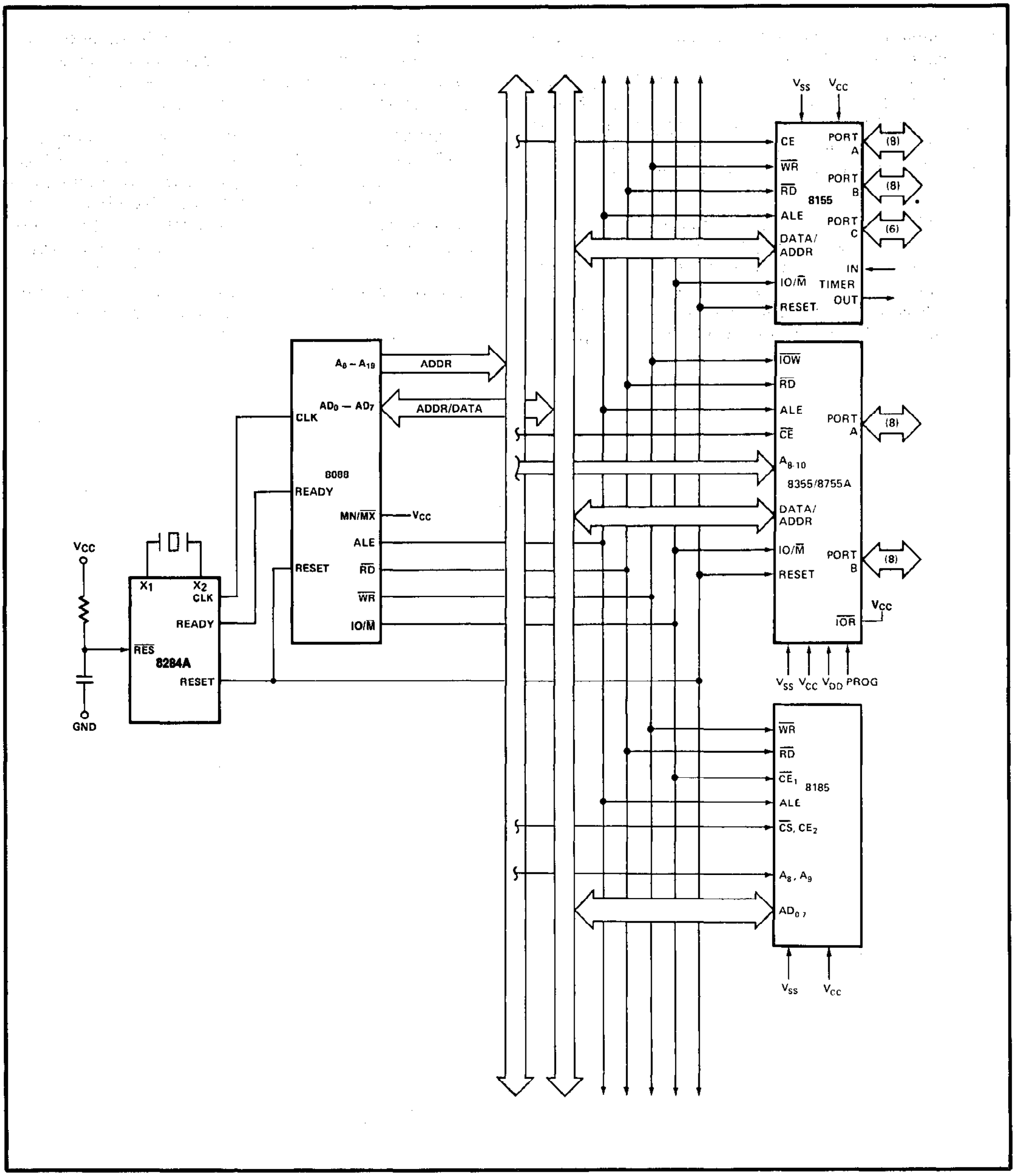


Figure 5. Multiplexed Bus Configuration

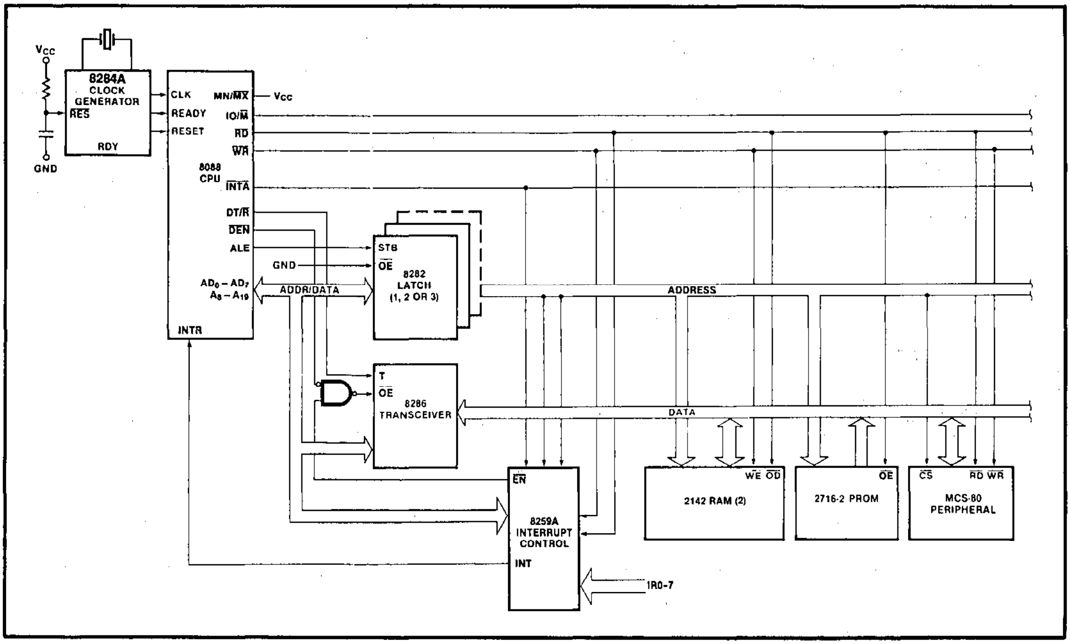


Figure 6. Demultiplexed Bus Configuration

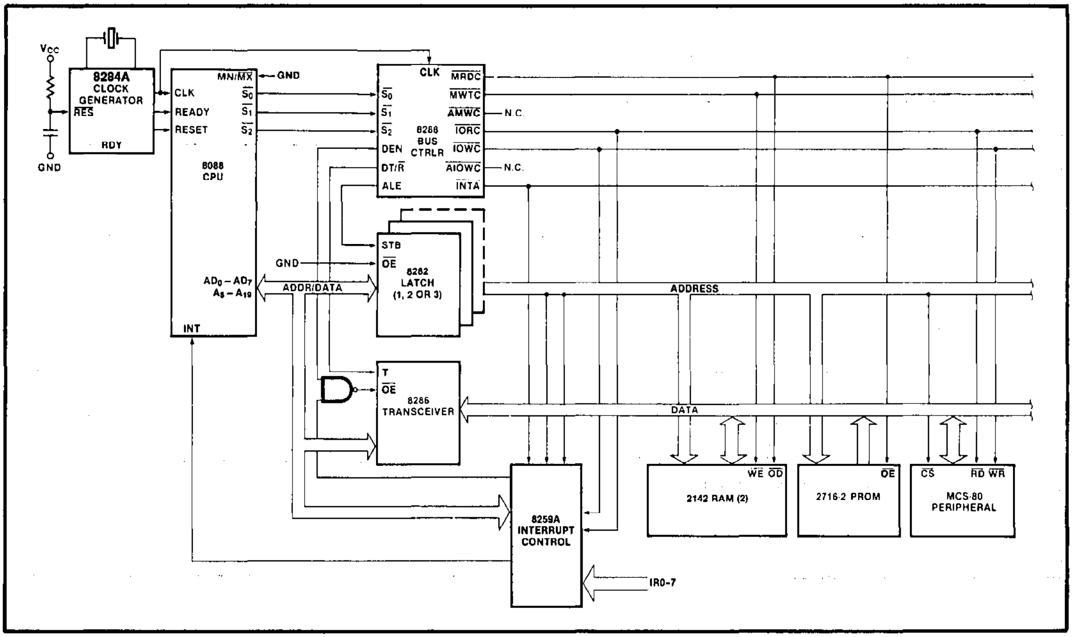


Figure 7. Fully Buffered System Using Bus Controller

Bus Operation

The 8088 address/data bus is broken into three parts — the lower eight address/data bits (AD0-AD7), the middle eight address bits (A8-A15), and the upper four address bits (A16-A19). The address/data bits and the highest four address bits are time multiplexed. This technique provides the most efficient use of pins on the processor, permitting the use of a standard 40 lead package. The middle eight address bits are not multiplexed, i.e. they remain valid throughout each bus cycle. In addition, the bus can be demultiplexed at the processor with a single address latch if a standard, non-multiplexed bus is desired for the system.

Each processor bus cycle consists of at least four CLK cycles. These are referred to as T1, T2, T3, and T4. (See Figure 8). The address is emitted from the processor during T1 and data transfer occurs on the bus during T3 and T4. T2 is used primarily for changing the direction of the bus during read operations. In the event that a "NOT READY" indication is given by the addressed device,

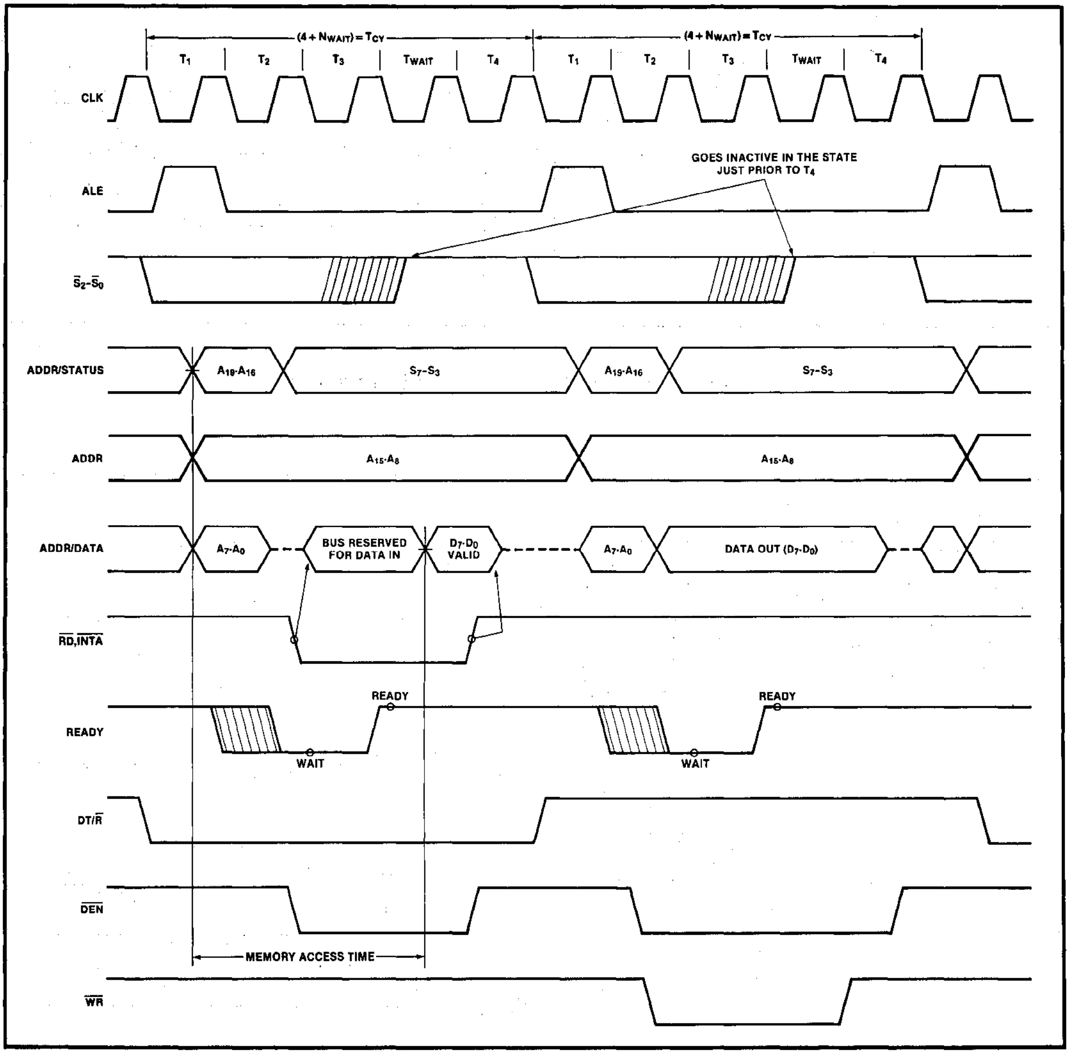


Figure 8. Basic System Timing

"wait" states (Tw) are inserted between T3 and T4. Each inserted "wait" state is of the same duration as a CLK cycle. Periods can occur between 8088 driven bus cycles. These are referred to as "idle" states (Ti), or inactive CLK cycles. The processor uses these cycles for internal housekeeping.

During T1 of any bus cycle, the ALE (address latch enable) signal is emitted (by either the processor or the 8288 bus controller, depending on the MN/$\overline{MX}$ strap). At the trailing edge of this pulse, a valid address and certain status information for the cycle may be latched.

Status bits $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ are used by the bus controller, in maximum mode, to identify the type of bus transaction according to the following table:

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	CHARACTERISTICS
0 (Low)	0	0	Interrupt Acknowledge
0	0	1	Read I/O
0	1	0	Write I/O
0	1	1	Halt
1 (High)	0	0	Instruction fetch
1	0	1	Read data from memory
1	1	0	Write data to memory
1	1	1	Passive (no bus cycle)

Status bits S3 through S6 are multiplexed with high order address bits and are therefore valid during T2 through T4, S3 and S4 indicate which segment register was used for this bus cycle in forming the address according to the following table:

S4	S3	CHARACTERISTICS
0 (Low)	0	Alternate data (Extra Segment)
0	1	Stack
1 (High)	0	Code or none
1	1	Data

S5 is a reflection of the PSW interrupt enable bit. S6 is always equal to 0.

I/O Addressing

In the 8088, I/O operations can address up to a maximum of 64K I/O registers. The I/O address appears in the same format as the memory address on bus lines A15-A0. The address lines A19-A16 are zero in I/O operations. The variable I/O instructions, which use register DX as a pointer, have full address capability, while the direct I/O instructions directly address one or two of the 256 I/O byte locations in page 0 of the I/O address space. I/O ports are addressed in the same manner as memory locations.

Designers familiar with the 8085 or upgrading an 8085 design should note that the 8085 addresses I/O with an 8-bit address on both halves of the 16-bit address bus. The 8088 uses a full 16-bit address on its lower 16 address lines.

EXTERNAL INTERFACE

Processor Reset and Initialization

Processor initialization or start up is accomplished with activation (HIGH) of the RESET pin. The 8088 RESET is required to be HIGH for greater than four clock cycles. The 8088 will terminate operations on the high-going edge of RESET and will remain dormant as long as RESET is HIGH. The low-going transition of RESET triggers an internal reset sequence for approximately 7 clock cycles. After this interval the 8088 operates normally, beginning with the instruction in absolute location FFFF0H. (See Figure 4.) The RESET input is internally synchronized to the processor clock. At initialization, the HIGH to LOW transition of RESET must occur no sooner than 50 μs after power up, to allow complete initialization of the 8088.

If INTR is asserted sooner than nine clock cycles after the end of RESET, the processor may execute one instruction before responding to the interrupt.

All 3-state outputs float to 3-state OFF during RESET. Status is active in the idle state for the first clock after RESET becomes active and then floats to 3-state OFF.

Interrupt Operations

Interrupt operations fall into two classes; software or hardware initiated. The software initiated interrupts and software aspects of hardware interrupts are specified in the instruction set description in the 8086 Family User's Manual. Hardware interrupts can be classified as nonmaskable or maskable.

Interrupts result in a transfer of control to a new program location. A 256 element table containing address pointers to the interrupt service program locations resides in absolute locations 0 through 3FFH (see Figure 4), which are reserved for this purpose. Each element in the table is 4 bytes in size and corresponds to an interrupt "type". An interrupting device supplies an 8-bit type number, during the interrupt acknowledge sequence, which is used to vector through the appropriate element to the new interrupt service program location.

Non-Maskable Interrupt (NMI)

The processor provides a single non-maskable interrupt (NMI) pin which has higher priority than the maskable interrupt request (INTR) pin. A typical use would be to activate a power failure routine. The NMI is edge-triggered on a LOW to HIGH transition. The activation of this pin causes a type 2 interrupt.

NMI is required to have a duration in the HIGH state of greater than two clock cycles, but is not required to be synchronized to the clock. Any higher going transition of NMI is latched on-chip and will be serviced at the end of the current instruction or between whole moves (2 bytes in the case of word moves) of a block type instruction. Worst case response to NMI would be for multiply, divide, and variable shift instructions. There is no specification on the occurrence of the low-going edge; it may occur before, during, or after the servicing of NMI. Another high-going edge triggers another response if it occurs after the start of the NMI procedure. The signal must be free of logical spikes in general and be free of bounces on the low-going edge to avoid triggering extraneous responses.

Maskable Interrupt (INTR)

The 8088 provides a single interrupt request input (INTR) which can be masked internally by software with the resetting of the interrupt enable (IF) flag bit. The interrupt request signal is level triggered. It is internally synchronized during each clock cycle on the high-going edge of CLK. To be responded to, INTR must be present (HIGH) during the clock period preceding the end of the current instruction or the end of a whole move for a block type instruction. During interrupt response sequence, further interrupts are disabled. The enable bit is reset as part of the response to any interrupt (INTR, NMI, software interrupt, or single step), although the FLAGS register which is automatically pushed onto the stack reflects the state of the processor prior to the interrupt. Until the old FLAGS register is restored, the enable bit will be zero unless specifically set by an instruction.

During the response sequence (See Figure 9), the processor executes two successive (back to back) interrupt acknowledge cycles. The 8088 emits the LOCK signal (maximum mode only) from T2 of the first bus cycle until T2 of the second. A local bus "hold" request will not be honored until the end of the second bus cycle. In the second bus cycle, a byte is fetched from the external interrupt system (e.g., 8259A PIC) which identifies the source (type) of the interrupt. This byte is multiplied by four and used as a pointer into the interrupt vector lookup table. An INTR signal left HIGH will be continually responded to within the limitations of the enable bit and sample period. The interrupt return instruction includes a flags pop which returns the status of the original interrupt enable bit when it restores the flags.

HALT

When a software HALT instruction is executed, the processor indicates that it is entering the HALT state in one of two ways, depending upon which mode is strapped. In minimum mode, the processor issues ALE, delayed by one clock cycle, to allow the system to latch the halt status. Halt status is available on IO/$\overline{M}$, DT/$\overline{R}$, and $\overline{SSO}$. In maximum mode, the processor issues appropriate HALT status on $\overline{S2}$, $\overline{S1}$, and $\overline{S0}$, and the 8288 bus controller issues one ALE. The 8088 will not leave the HALT state when a local bus hold is entered while in HALT. In this case, the processor reissues the HALT indicator at the end of the local bus hold. An interrupt request or RESET will force the 8088 out of the HALT state.

Read/Modify/Write (Semaphore) Operations via LOCK

The LOCK status information is provided by the processor when consecutive bus cycles are required during the execution of an instruction. This allows the processor to perform read/modify/write operations on memory (via the "exchange register with memory" instruction), without another system bus master receiving intervening memory cycles. This is useful in multiprocessor system configurations to accomplish "test and set lock" operations. The $\overline{LOCK}$ signal is activated (LOW) in the clock cycle following decoding of the LOCK prefix instruction. It is deactivated at the end of the last bus cycle of the instruction following the LOCK prefix. While $\overline{LOCK}$ is active, a request on a $\overline{RQ}/\overline{GT}$ pin will be recorded, and then honored at the end of the LOCK.

External Synchronization via $\overline{TEST}$

As an alternative to interrupts, the 8088 provides a single software-testable input pin ($\overline{TEST}$). This input is utilized by executing a WAIT instruction. The single

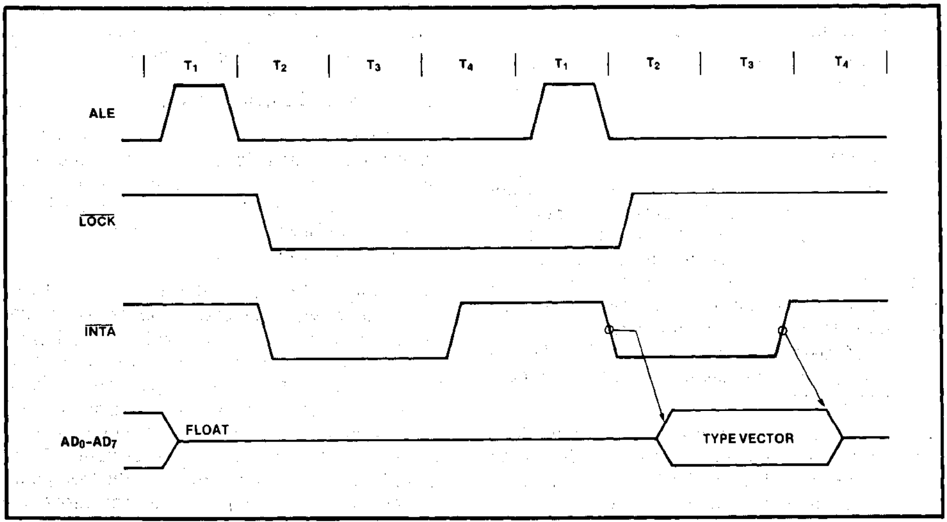


Figure 9. Interrupt Acknowledge Sequence

WAIT instruction is repeatedly executed until the $\overline{\text{TEST}}$ input goes active (LOW). The execution of WAIT does not consume bus cycles once the queue is full.

If a local bus request occurs during WAIT execution, the 8088 3-states all output drivers. If interrupts are enabled, the 8088 will recognize interrupts and process them. The WAIT instruction is then refetched, and reexecuted.

Basic System Timing

In minimum mode, the MN/$\overline{\text{MX}}$ pin is strapped to V_{CC} and the processor emits bus control signals compatible with the 8085 bus structure. In maximum mode, the MN/$\overline{\text{MX}}$ pin is strapped to GND and the processor emits coded status information which the 8288 bus controller uses to generate MULTIBUS compatible bus control signals.

System Timing — Minimum System

(See Figure 8.)

The read cycle begins in T1 with the assertion of the address latch enable (ALE) signal. The trailing (low going) edge of this signal is used to latch the address information, which is valid on the address/data bus (AD0-AD7) at this time, into the 8282/8283 latch. Address lines A8 through A15 do not need to be latched because they remain valid throughout the bus cycle. From T1 to T4 the IO/$\overline{\text{M}}$ signal indicates a memory or I/O operation. At T2 the address is removed from the address/data bus and the bus goes to a high impedance state. The read control signal is also asserted at T2. The read ($\overline{\text{RD}}$) signal causes the addressed device to enable its data bus drivers to the local bus. Some time later, valid data will be available on the bus and the addressed device will drive the READY line HIGH. When the processor returns the read signal to a HIGH level, the addressed device will again 3-state its bus drivers. If a transceiver (8286/8287) is required to buffer the 8088 local bus, signals DT/$\overline{\text{R}}$ and $\overline{\text{DEN}}$ are provided by the 8088.

A write cycle also begins with the assertion of ALE and the emission of the address. The IO/$\overline{\text{M}}$ signal is again asserted to indicate a memory or I/O write operation. In T2, immediately following the address emission, the processor emits the data to be written into the addressed location. This data remains valid until at least the middle of T4. During T2, T3, and T_W, the processor asserts the write control signal. The write ($\overline{\text{WR}}$) signal becomes active at the beginning of T2, as opposed to the read, which is delayed somewhat into T2 to provide time for the bus to float.

The basic difference between the interrupt acknowledge cycle and a read cycle is that the interrupt acknowledge ($\overline{\text{INTA}}$) signal is asserted in place of the read ($\overline{\text{RD}}$) signal and the address bus is floated. (See Figure 9.) In the second of two successive $\overline{\text{INTA}}$ cycles, a byte of information is read from the data bus, as supplied by the interrupt system logic (i.e. 8259A priority interrupt controller). This byte identifies the source (type) of the interrupt. It is multiplied by four and used as a pointer into the interrupt vector lookup table, as described earlier.

Bus Timing — Medium Complexity Systems

(See Figure 10.)

For medium complexity systems, the MN/$\overline{\text{MX}}$ pin is connected to GND and the 8288 bus controller is added to the system, as well as an 8282/8283 latch for latching the system address, and an 8286/8287 transceiver to allow for bus loading greater than the 8088 is capable of handling. Signals ALE, $\overline{\text{DEN}}$, and DT/$\overline{\text{R}}$ are generated by the 8288 instead of the processor in this configuration, although their timing remains relatively the same. The 8088 status outputs ($\overline{\text{S2}}$, $\overline{\text{S1}}$, and $\overline{\text{S0}}$) provide type of cycle information and become 8288 inputs. This bus cycle information specifies read (code, data, or I/O), write (data or I/O), interrupt acknowledge, or software halt. The 8288 thus issues control signals specifying memory read or write, I/O read or write, or interrupt acknowledge. The 8288 provides two types of write strobes, normal and advanced, to be applied as required. The normal write strobes have data valid at the leading edge of write. The advanced write strobes have the same timing as read strobes, and hence, data is not valid at the leading edge of write. The 8286/8287 transceiver receives the usual T and $\overline{\text{OE}}$ inputs from the 8288's DT/$\overline{\text{R}}$ and $\overline{\text{DEN}}$ outputs.

The pointer into the interrupt vector table, which is passed during the second $\overline{\text{INTA}}$ cycle, can derive from an 8259A located on either the local bus or the system bus. If the master 8289A priority interrupt controller is positioned on the local bus, a TTL gate is required to disable the 8286/8287 transceiver when reading from the master 8259A during the interrupt acknowledge sequence and software "poll".

The 8088 Compared to the 8086

The 8088 CPU is an 8-bit processor designed around the 8086 internal structure. Most internal functions of the 8088 are identical to the equivalent 8086 functions. The 8088 handles the external bus the same way the 8086 does with the distinction of handling only 8 bits at a time. Sixteen-bit operands are fetched or written in two consecutive bus cycles. Both processors will appear identical to the software engineer, with the exception of execution time. The internal register structure is identical and all instructions have the same end result. The differences between the 8088 and 8086 are outlined below. The engineer who is unfamiliar with the 8086 is referred to the 8086 Family User's Manual, Chapters 2 and 4, for function description and instruction set information.

Internally, there are three differences between the 8088 and the 8086. All changes are related to the 8-bit bus interface.

- The queue length is 4 bytes in the 8088, whereas the 8086 queue contains 6 bytes, or three words. The queue was shortened to prevent overuse of the bus by the BIU when prefetching instructions. This was required because of the additional time necessary to fetch instructions 8 bits at a time.

- To further optimize the queue, the prefetching algorithm was changed. The 8088 BIU will fetch a new instruction to load into the queue eac. time there is a 1 byte hole (space available) in the queue. The 8086 waits until a 2-byte space is available.
- The internal execution time of the instruction set is affected by the 8-bit interface. All 16-bit fetches and writes from/to memory take an additional four clock cycles. The CPU is also limited by the speed of instruction fetches. This latter problem only occurs when a series of simple operations occur. When the more sophisticated instructions of the 8088 are being used, the queue has time to fill and the execution proceeds as fast as the execution unit will allow.

The 8088 and 8086 are completely software compatible by virture of their identical execution units. Software that is system dependent may not be completely transferable, but software that is not system dependent will operate equally as well on an 8088 or an 8086.

The hardware interface of the 8088 contains the major differences between the two CPUs. The pin assignments are nearly identical, however, with the following functional changes:

- A8–A15 — These pins are only address outputs on the 8088. These address lines are latched internally and remain valid throughout a bus cycle in a manner similar to the 8085 upper address lines.
- $\overline{\text{BHE}}$ has no meaning on the 8088 and has been eliminated.
- $\overline{\text{SSO}}$ provides the $\overline{\text{SO}}$ status information in the minimum mode. This output occurs on pin 34 in minimum mode only. DT/$\overline{\text{R}}$, IO/$\overline{\text{M}}$, and $\overline{\text{SSO}}$ provide the complete bus status in minimum mode.
- IO/$\overline{\text{M}}$ has been inverted to be compatible with the MCS-85 bus structure.
- ALE is delayed by one clock cycle in the minimum mode when entering HALT, to allow the status to be latched with ALE.

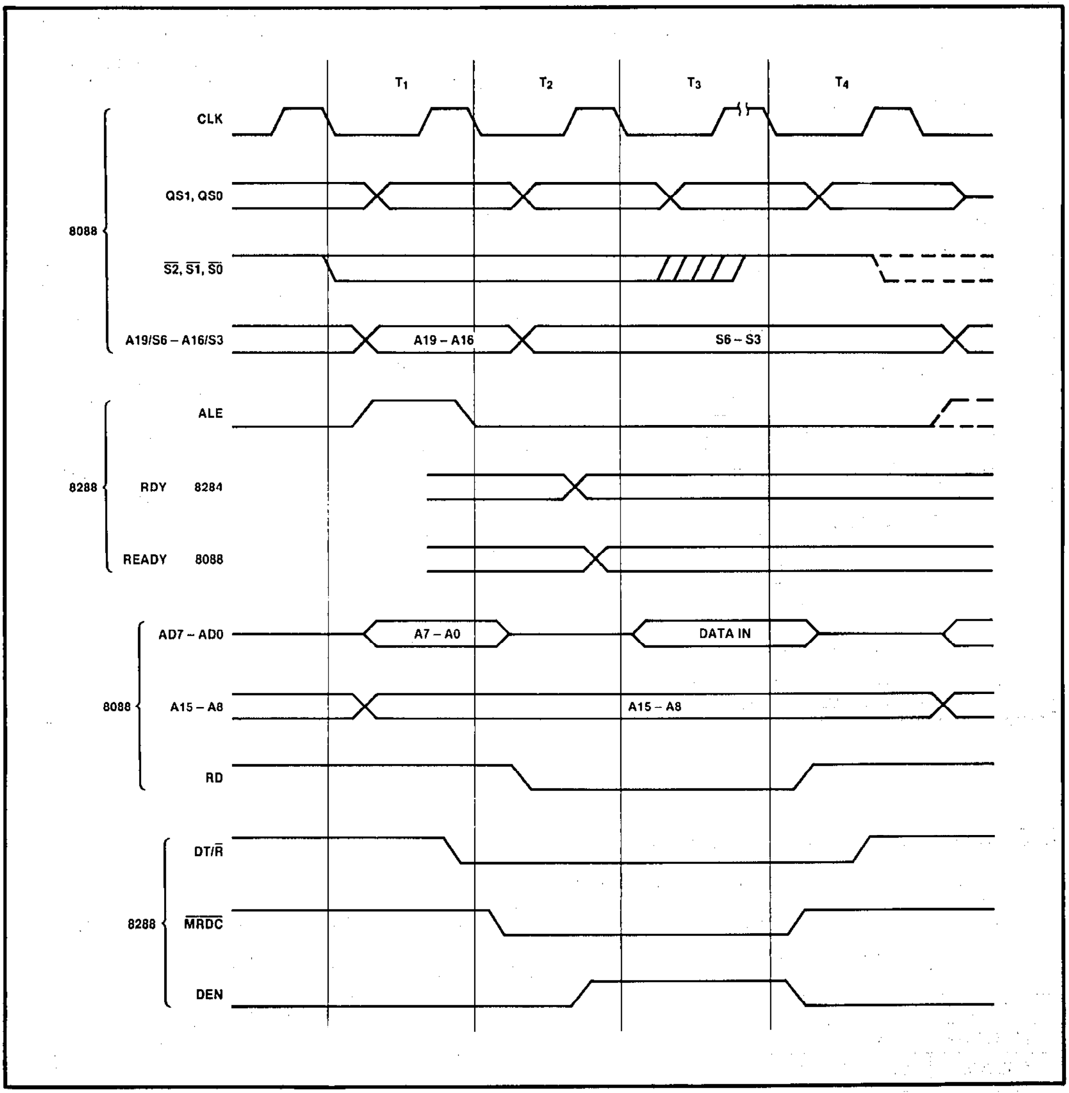


Figure 10. Medium Complexity System Timing

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias.........0°C to 70°C
Storage Temperature............. −65°C to +150°C
Voltage on Any Pin with
Respect to Ground.................. −1.0 to +7V
Power Dissipation .... 2.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ±10%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}+0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = 400 μA
I_{CC}	Power Supply Current		340	mA	T_A = 25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC}+1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except AD_0-AD_7 RQ/GT)		15	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD_0-AD_7 RQ/GT)		15	pF	fc = 1 MHz

A.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ±10%)

MINIMUM COMPLEXITY SYSTEM TIMING REQUIREMENTS

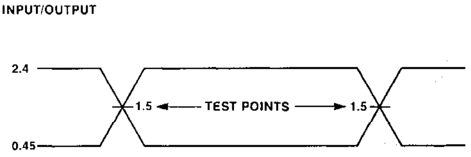
Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL)−15		ns	
TCHCL	CLK High Time	(⅓ TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1,2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8088	(⅔ TCLCL)−15		ns	
TCHRYX	READY Hold Time into 8088	30		ns	
TRYLCL	READY Inactive to CLK(See Note 3)	−8		ns	
THVCH	HOLD Setup Time	35		ns	
TINVCH	INTR, NMI, $\overline{\text{TEST}}$ Setup Time (See Note 2)	30		ns	
TILIH	Input Rise Time (Except CLK)		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

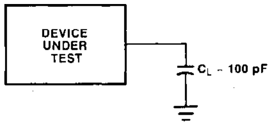
Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLAV	Address Valid Delay	10	110	ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TLHLL	ALE Width	TCLCH−20		ns	
TCLLH	ALE Active Delay		80	ns	
TCHLL	ALE Inactive Delay		85	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL−10		ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TWHDX	Data Hold Time After $\overline{WR}$	TCLCH−30		ns	
TCVCTV	Control Active Delay 1	10	110	ns	
TCHCTV	Control Active Delay 2	10	110	ns	
TCVCTX	Control Inactive Delay	10	110	ns	
TAZRL	Address Float to READ Active	0		ns	
TCLRL	$\overline{RD}$ Active Delay	10	165	ns	
TCLRH	$\overline{RD}$ Inactive Delay	10	150	ns	
TRHAV	$\overline{RD}$ Inactive to Next Address Active	TCLCL−45		ns	
TCLHAV	HLDA Valid Delay	10	160	ns	
TRLRH	$\overline{RD}$ Width	2TCLCL−75		ns	
TWLWH	$\overline{WR}$ Width	2TCLCL−60		ns	
TAVAL	Address Valid to ALE Low	TCLCH−60		ns	
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." THE CLOCK IS DRIVEN AT 4.3V AND 0.25V. TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

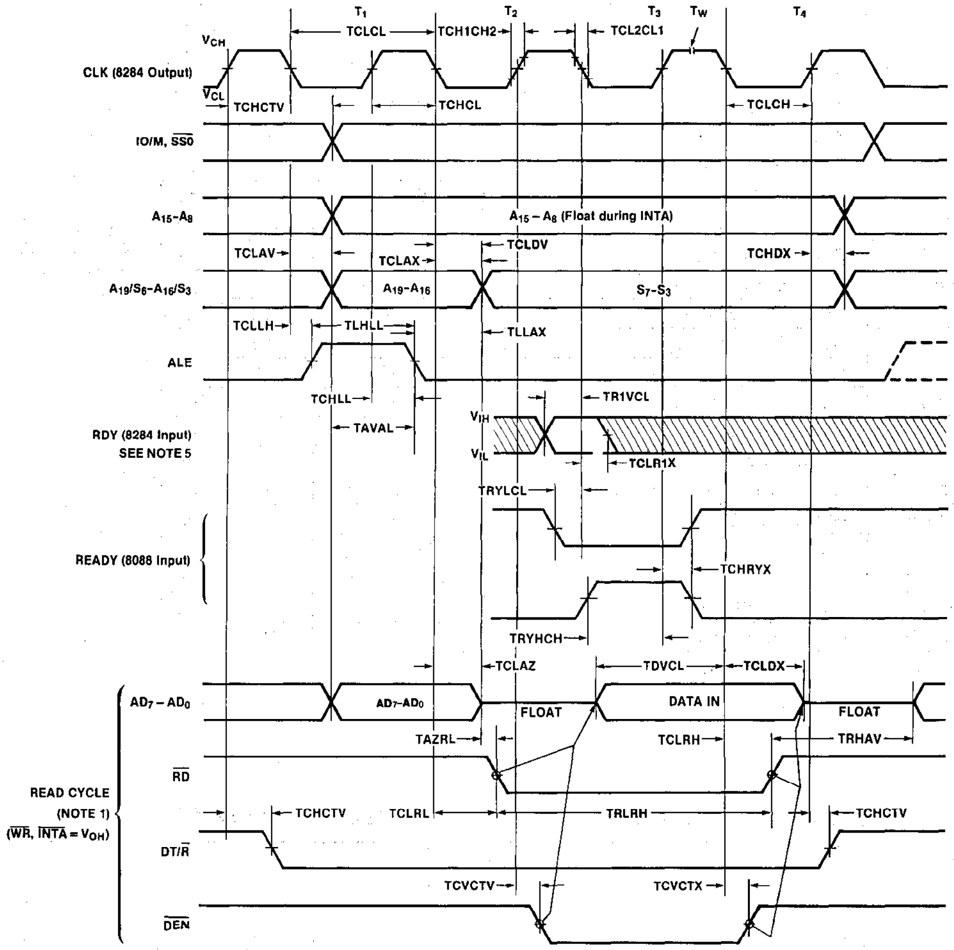
A.C. TESTING LOAD CIRCUIT



C_L INCLUDES JIG CAPACITANCE

WAVEFORMS

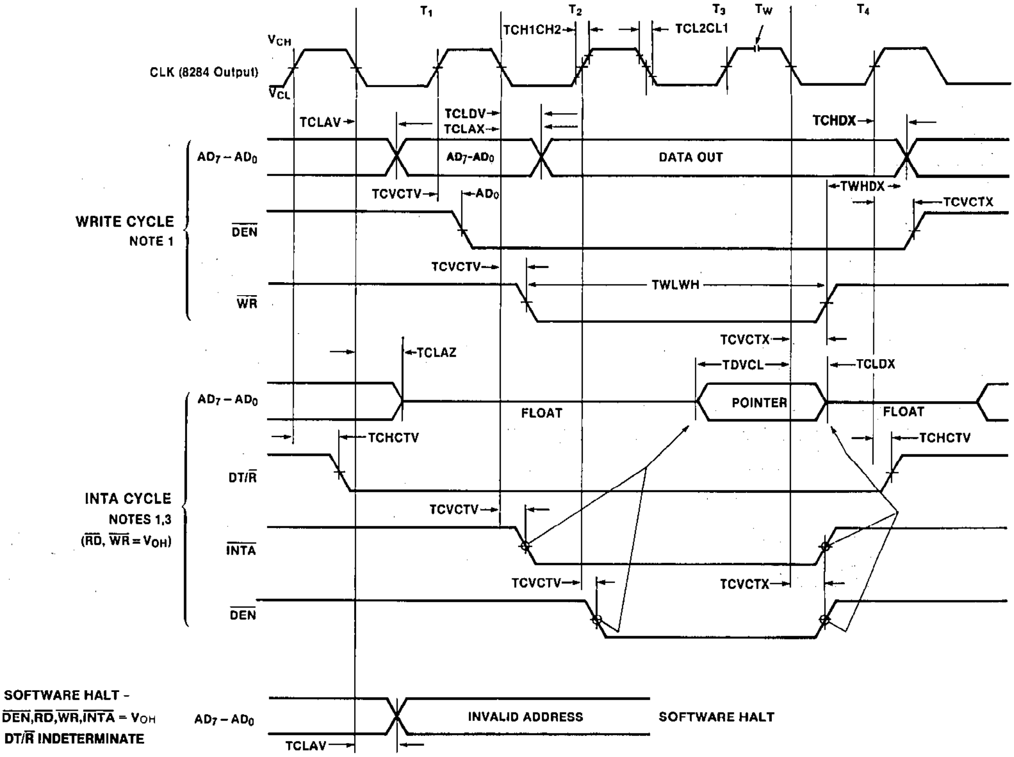
BUS TIMING—MINIMUM MODE SYSTEM



AFN-00826B

WAVEFORMS (Continued)

BUS TIMING—MINIMUM MODE SYSTEM (Continued)



NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL SIGNALS ARE SHOWN FOR THE SECOND INTA CYCLE.
4. SIGNALS AT 8284 ARE SHOWN FOR REFERENCE ONLY.
5. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

A.C. CHARACTERISTICS (Continued)

MAX MODE SYSTEM (USING 8288 BUS CONTROLLER)
TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL)−15		ns	
TCHCL	CLK High Time	(⅓ TCLCL)+2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8088	(⅔ TCLCL)−15		ns	
TCHRYX	READY Hold Time into 8088	30		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, TEST) (See Note 2)	30		ns	
TGVCH	RQ/GT Setup Time	30		ns	
TCHGX	RQ Hold Time into 8086	40		ns	
TILIH	Input Rise Time (Except CLK)		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

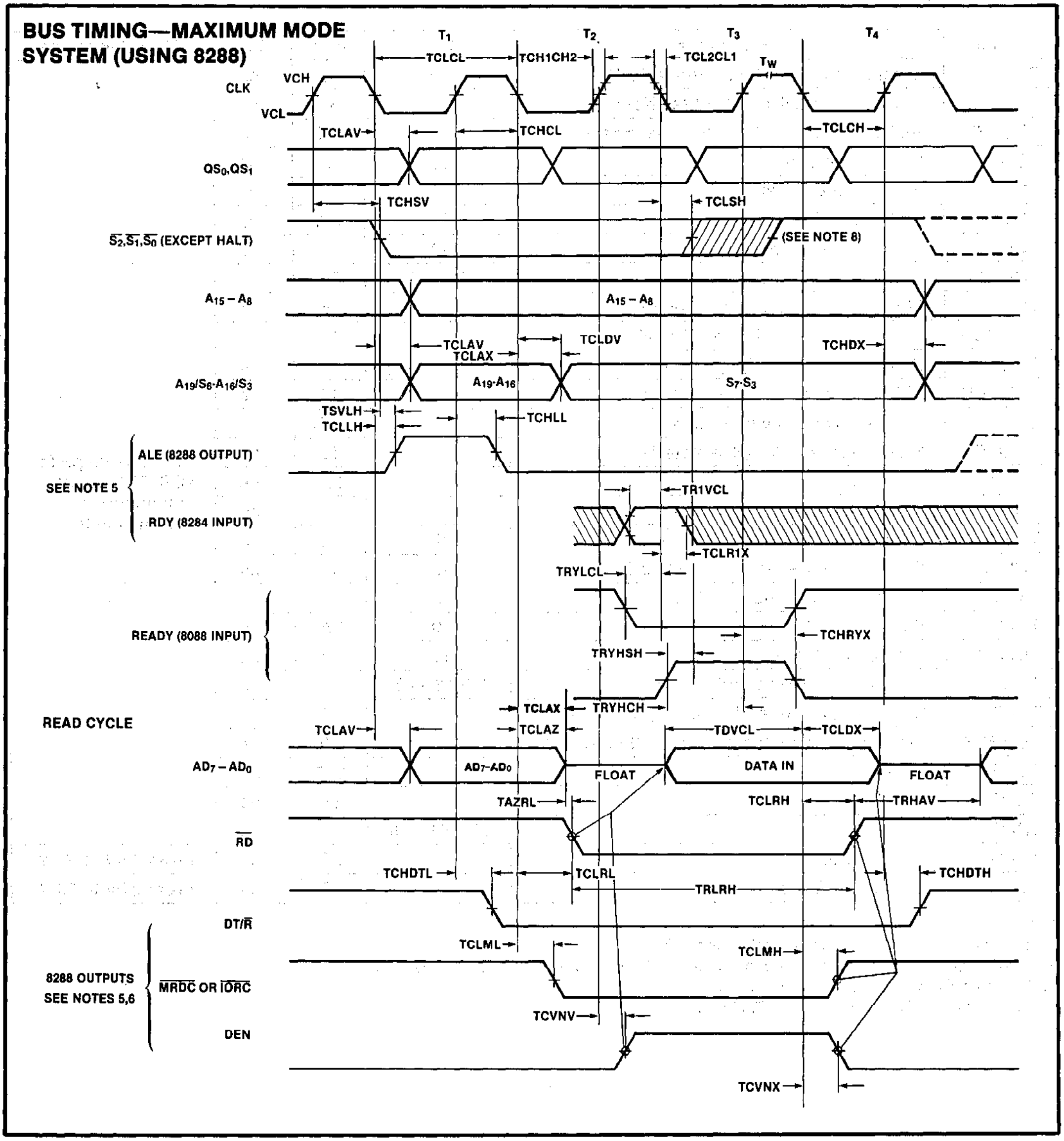
TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	C_L = 20-100 pF for all 8088 Outputs in addition to internal loads
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	
TRYHSH	READY Active to Status Passive (See Note 3)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	10	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TSVMCH	Status Valid to MCE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	
TCLMCH	CLK Low to MCE High (See Note 1)		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	
TCLMCL	MCE Inactive Delay (See Note 1)		15	ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TAZRL	Address Float to Read Active	0		ns	
TCLRL	RD Active Delay	10	165	ns	
TCLRH	RD Inactive Delay	10	150	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL−45		ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	GT Active Delay		110	ns	
TCLGH	GT Inactive Delay		85	ns	
TRLRH	RD Width	2TCLCL−75		ns	
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

NOTES:

1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Applies only to T2 state (8 ns into T3 state).
4. Applies only to T2 state (8 ns into T3 state).

WAVEFORMS (Continued)

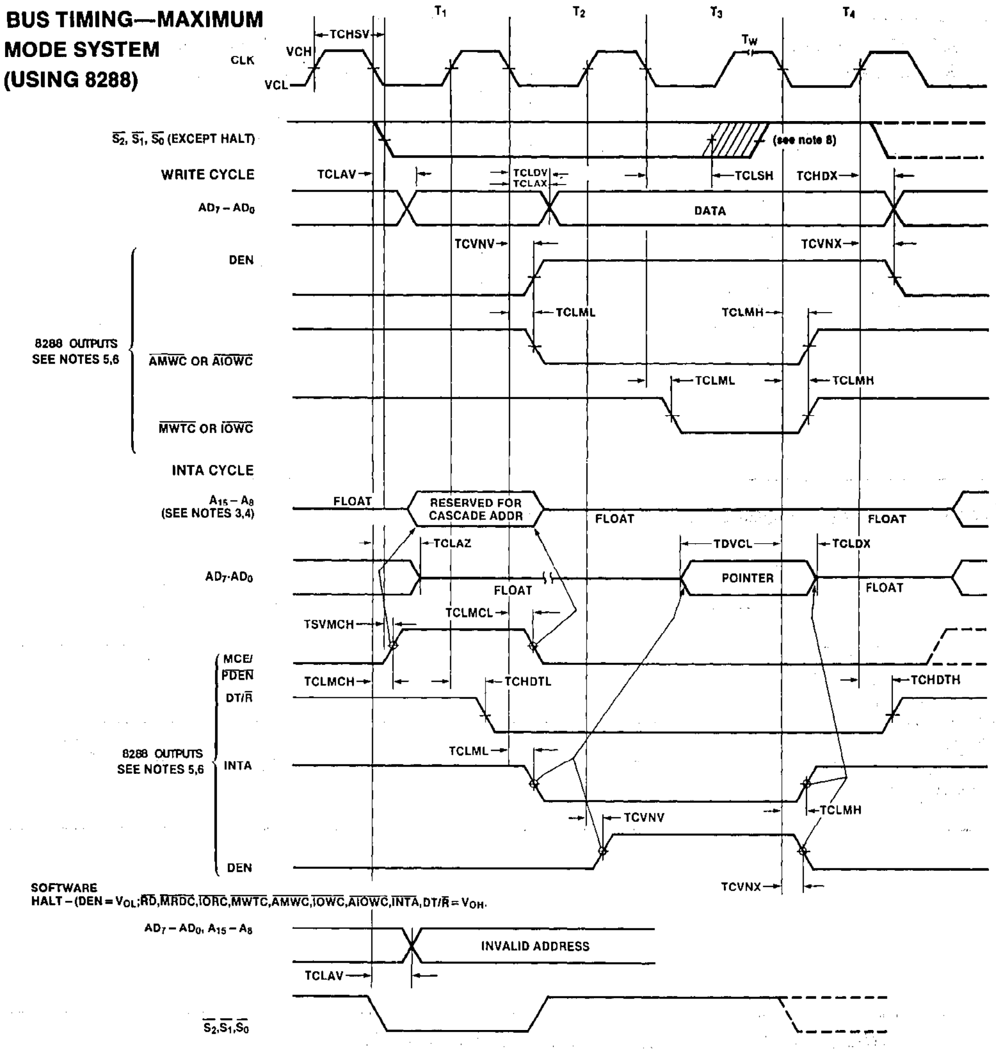


BUS TIMING—MAXIMUM MODE SYSTEM (USING 8288)

AFN-00826B

WAVEFORMS (Continued)

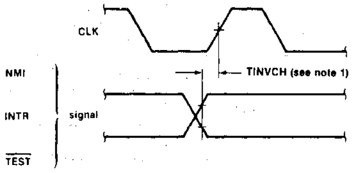
BUS TIMING—MAXIMUM MODE SYSTEM (USING 8288)


T1
T2
T3
T4
TW
TCHSV
CLK
VCH
VCL
$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ (EXCEPT HALT)
(see note 8)
WRITE CYCLE
TCLAV
TCLDV
TCLAX
TCLSH
TCHDX
$AD_7 - AD_0$
DATA
TCVNV
TCVNX
DEN
TCLML
TCLMH
8288 OUTPUTS
SEE NOTES 5,6
$\overline{AMWC}$ OR $\overline{AIOWC}$
TCLML
TCLMH
$\overline{MWTC}$ OR $\overline{IOWC}$
INTA CYCLE
$A_{15} - A_8$
(SEE NOTES 3,4)
FLOAT
RESERVED FOR CASCADE ADDR
FLOAT
FLOAT
TCLAZ
TDVCL
TCLDX
AD_7-AD_0
FLOAT
POINTER
FLOAT
TCLMCL
TSVMCH
MCE/$\overline{PDEN}$
TCLMCH
TCHDTL
TCHDTH
DT/$\overline{R}$
8288 OUTPUTS
SEE NOTES 5,6
TCLML
INTA
TCVNV
TCLMH
DEN
TCVNX
SOFTWARE
HALT – (DEN = V_{OL}; $\overline{RD}$, $\overline{MRDC}$, $\overline{IORC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$, DT/$\overline{R}$ = V_{OH}.
$AD_7 - AD_0$, $A_{15} - A_8$
INVALID ADDRESS
TCLAV
$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$

NOTES:
1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2, T_3, T_W TO DETERMINE IF T_W MACHINES STATES ARE TO BE INSERTED.
3. CASCADE ADDRESS IS VALID BETWEEN FIRST AND SECOND INTA CYCLES.
4. TWO INTA CYCLES RUN BACK-TO-BACK. THE 8088 LOCAL ADDR/DATA BUS IS FLOATING DURING BOTH INTA CYCLES. CONTROL FOR POINTER ADDRESS IS SHOWN FOR SECOND INTA CYCLE.
5. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
6. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
7. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
8. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.

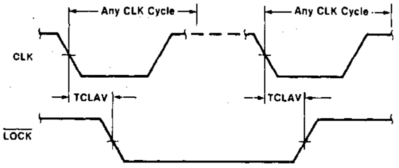
WAVEFORMS (Continued)

ASYNCHRONOUS SIGNAL RECOGNITION

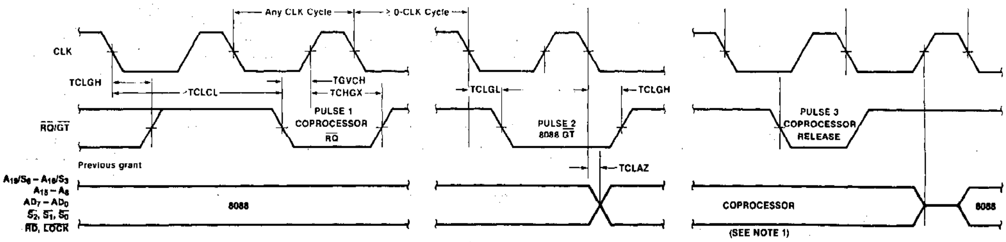


NOTE: 1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK

BUS LOCK SIGNAL TIMING (MAXIMUM MODE ONLY)

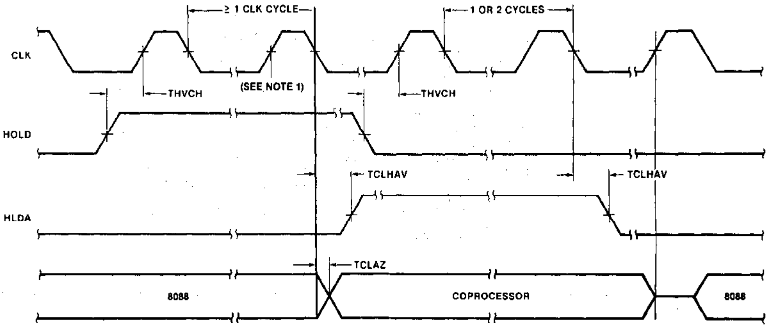


REQUEST/GRANT SEQUENCE TIMING (MAXIMUM MODE ONLY)



NOTE: 1. THE COPROCESSOR MAY NOT DRIVE THE BUSSES OUTSIDE THE REGION SHOWN WITHOUT RISKING CONTENTION.

HOLD/HOLD ACKNOWLEDGE TIMING (MINIMUM MODE ONLY)



iAPX 86/10, 88/10 INSTRUCTION SET SUMMARY

DATA TRANSFER

MOV = Move:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory to/from register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate to register	1 0 1 1 w reg	data	data if w 1	
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/memory with register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with accumulator	1 0 0 1 0 reg			
IN=Input from:				
Fixed port	1 1 1 0 0 1 0 w	port		
Variable port	1 1 1 0 1 1 0 w			
OUT = Output to:				
Fixed port	1 1 1 0 0 1 1 w	port		
Variable port	1 1 1 0 1 1 1 w			
XLAT=Translate byte to AL	1 1 0 1 0 1 1 1			
LEA=Load EA to register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS=Load pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES=Load pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF=Load AH with flags	1 0 0 1 1 1 1 1			
SAHF=Store AH into flags	1 0 0 1 1 1 1 0			
PUSHF=Push flags	1 0 0 1 1 1 0 0			
POPF=Pop flags	1 0 0 1 1 1 0 1			

ARITHMETIC

ADD = Add:				
Reg./memory with register to either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s w 01
Immediate to accumulator	0 0 0 0 0 1 0 w	data	data if w 1	
ADC = Add with carry:				
Reg./memory with register to either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s w 01
Immediate to accumulator	0 0 0 1 0 1 0 w	data	data if w 1	
INC = Increment:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA=ASCII adjust for add	0 0 1 1 0 1 1 1			
DAA=Decimal adjust for add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./memory and register to either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w 01
Immediate from accumulator	0 0 1 0 1 1 0 w	data	data if w 1	
SBB = Subtract with borrow				
Reg./memory and register to either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from register/memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w 01
Immediate from accumulator	0 0 0 1 1 1 0 w	data	data if w 1	

DEC Decrement:	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP Compare:				
Register/memory and register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with register/memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w 01
Immediate with accumulator	0 0 1 1 1 1 0 w	data	data if w 1	
AAS ASCII adjust for subtract	0 0 1 1 1 1 1 1			
DAS Decimal adjust for subtract	0 0 1 0 1 1 1 1			
MUL Multiply (unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL Integer multiply (signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM ASCII adjust for multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV Divide (unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV Integer divide (signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD ASCII adjust for divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW Convert byte to word	1 0 0 1 1 0 0 0			
CWD Convert word to double word	1 0 0 1 1 0 0 1			

LOGIC

NOT Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL Shift logical arithmetic left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR Shift logical right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR Shift arithmetic right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL Rotate left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR Rotate right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL Rotate through carry flag left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR Rotate through carry right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND And:				
Reg./memory and register to either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 0 0 1 0 w	data	data if w 1	
TEST And function to flags, no result:				
Register/memory and register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate data and register/memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w 1
Immediate data and accumulator	1 0 1 0 1 0 0 w	data	data if w 1	
OR Or:				
Reg./memory and register to either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w 1
Immediate to accumulator	0 0 0 0 1 1 0 w	data	data if w 1	
XOR Exclusive or:				
Reg./memory and register to either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to register/memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w 1
Immediate to accumulator	0 0 1 1 0 1 0 w	data	data if w 1	

STRING MANIPULATION

REP=Repeat	1 1 1 1 0 0 1 z
MOVS=Move byte/word	1 0 1 0 0 1 0 w
CMPS=Compare byte/word	1 0 1 0 0 1 1 w
SCAS=Scan byte/word	1 0 1 0 1 1 1 w
LODS=Load byte/wd to AL/AX	1 0 1 0 1 1 0 w
STOS=Stor byte/wd from AL/A	1 0 1 0 1 0 1 w

INSTRUCTION SET SUMMARY (Continued)

CONTROL TRANSFER

CALL = Call:

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Direct within segment	1 1 1 0 1 0 0 0	disp-low	disp-high
Indirect within segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m	
Direct intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m	

JMP = Unconditional Jump:

Direct within segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within segment-short	1 1 1 0 1 0 1 1	disp	
Indirect within segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	

RET = Return from CALL:

Within segment	1 1 0 0 0 0 1 1		
Within seg. adding immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment, adding immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ=Jump on equal/zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE=Jump on less/not greater or equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG=Jump on less or equal/not greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE=Jump on below/not above or equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA=Jump on below or equal/not above	0 1 1 1 0 1 1 0	disp	
JP/JPE=Jump on parity/parity even	0 1 1 1 1 0 1 0	disp	
JO=Jump on overflow	0 1 1 1 0 0 0 0	disp	
JS=Jump on sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ=Jump on not equal/not zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE=Jump on not less/greater or equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG=Jump on not less or equal/greater	0 1 1 1 1 1 1 1	disp	

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
JNB/JAE=Jump on not below/above or equal	0 1 1 1 0 0 1 1	disp
JNBE/JA=Jump on not below or equal/above	0 1 1 1 0 1 1 1	disp
JNP/JPO=Jump on not par/par odd	0 1 1 1 1 0 1 1	disp
JNO=Jump on not overflow	0 1 1 1 0 0 0 1	disp
JNS Jump on not sign	0 1 1 1 1 0 0 1	disp
LOOP Loop CX times	1 1 1 0 0 0 1 0	disp
LOOPZ/LOOPE Loop while zero/equal	1 1 1 0 0 0 0 1	disp
LOOPNZ/LOOPNE Loop while not zero/equal	1 1 1 0 0 0 0 0	disp
JCXZ Jump on CX zero	1 1 1 0 0 0 1 1	disp

INT Interrupt

Type specified	1 1 0 0 1 1 0 1	type
Type 3	1 1 0 0 1 1 0 0	
INTO Interrupt on overflow	1 1 0 0 1 1 1 0	
IRET Interrupt return	1 1 0 0 1 1 1 1	

PROCESSOR CONTROL

CLC Clear carry	1 1 1 1 1 0 0 0	
CMC Complement carry	1 1 1 1 0 1 0 1	
STC Set carry	1 1 1 1 1 0 0 1	
CLD Clear direction	1 1 1 1 1 1 0 0	
STD Set direction	1 1 1 1 1 1 0 1	
CLI Clear interrupt	1 1 1 1 1 0 1 0	
STI Set interrupt	1 1 1 1 1 0 1 1	
HLT Halt	1 1 1 1 0 1 0 0	
WAIT Wait	1 0 0 1 1 0 1 1	
ESC Escape (to external device)	1 1 0 1 1 x x x	mod x x x r/m
LOCK Bus lock prefix	1 1 1 1 0 0 0 0	

Footnotes:

AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value.
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction

if s:w = 01 then 16 bits of immediate data form the operand
if s:w = 11 then an immediate data byte is sign extended to form the 16-bit operand.
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with Z.F FLAG.

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high: disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)

*except if mod = 00 and r/m = 110 then EA = disp-high: disp-low.

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

PRELIMINARY

8089
8 & 16-BIT HMOS I/O PROCESSOR

- **High Speed DMA Capabilities Including I/O to Memory, Memory to I/O, Memory to Memory, and I/O to I/O**
- **iAPX 86, 88 Compatible: Removes I/O Overhead from CPU in iAPX 86/11 or 88/11 Configuration**
- **Allows Mixed Interface of 8- & 16-Bit Peripherals, to 8- & 16-Bit Processor Busses**
- **1 Mbyte Addressability**
- **Memory Based Communication with CPU**
- **Supports LOCAL or REMOTE I/O Processing**
- **Flexible, Intelligent DMA Functions Including Translation, Search, Word Assembly/Disassembly**
- **MULTIBUS™ Compatible System Interface**

The Intel® 8089 is a revolutionary concept in microprocessor input/output processing. Packaged in a 40-pin DIP package, the 8089 is a high performance processor implemented in N-channel, depletion load silicon gate technology (HMOS). The 8089's instruction set and capabilities are optimized for high speed, flexible and efficient I/O handling. It allows easy interface of Intel's 16-bit iAPX 86 and 8-bit iAPX 88 microprocessors with 8- and 16-bit peripherals. In the REMOTE configuration, the 8089 bus is user definable allowing it to be compatible with any 8/16-bit Intel microprocessor, interfacing easily to the Intel multiprocessor system bus standard MULTIBUS™.

The 8089 performs the function of an intelligent DMA controller for the Intel iAPX 86, 88 family and with its processing power, can remove I/O overhead from the iAPX 86 or iAPX 88. It may operate completely in parallel with a CPU, giving dramatically improved performance in I/O intensive applications. The 8089 provides two I/O channels, each supporting a transfer rate up to 1.25 mbyte/sec at the standard clock frequency of 5 MHz. Memory based communication between the IOP and CPU enhances system flexibility and encourages software modularity, yielding more reliable, easier to develop systems.

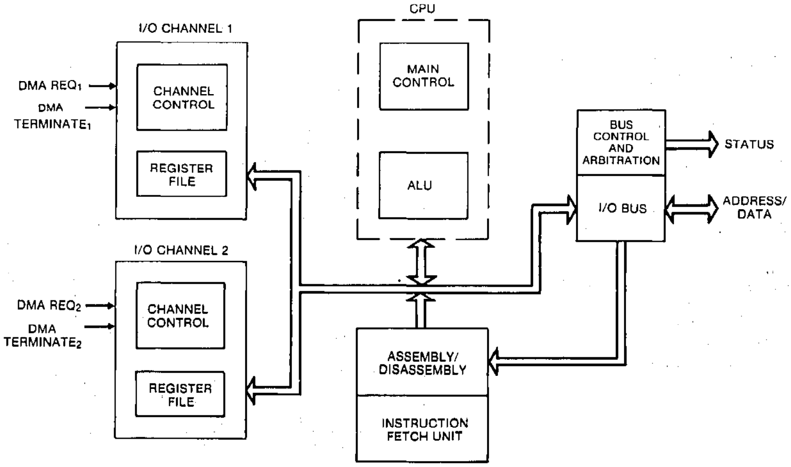


Figure 1. 8089 I/O Processor Block Diagram

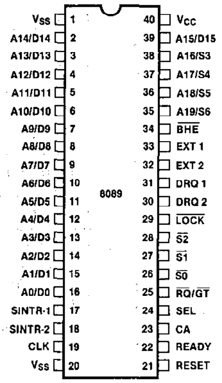


Figure 2. 8089 Pin Configuration

Table 1. Pin Description

Symbol	Type	Name and Function
A0–A15/ D0–D15	I/O	**Multiplexed Address and Data Bus:** The function of these lines are defined by the state of $\overline{S0}$, $\overline{S1}$ and $\overline{S2}$ lines. The pins are floated after reset and when the bus is not acquired. A8–A15 are stable on transfers to a physical 8-bit data bus (same bus as 8088), and are multiplexed with data on transfers to a 16-bit physical bus.
A16–A19/ S3–S6	O	**Address and Status:** Multiplexed most significant address lines and status information. The address lines are active only when addressing memory. Otherwise, the status lines are active and are encoded as shown below. The pins are floated after reset and when the bus is not acquired. **S6 S5 S4 S3** 1 1 0 0 DMA cycle on CH1 1 1 0 1 DMA cycle on CH2 1 1 1 0 Non-DMA cycle on CH1 1 1 1 1 Non-DMA cycle on CH2
$\overline{BHE}$	O	**Bus High Enable:** The Bus High Enable is used to enable data operations on the most significant half of the data bus (D8–D15). The signal is active low when a byte is to be transferred on the upper half of the data bus. The pin is floated after reset and when the bus is not acquired. $\overline{BHE}$ does not have to be latched.
$\overline{S0}$, $\overline{S1}$, $\overline{S2}$	O	**Status:** These are the status pins that define the IOP activity during any given cycle. They are encoded as shown below: **$\overline{S2}$ $\overline{S1}$ $\overline{S0}$** 0 0 0 Instruction fetch; I/O space 0 0 1 Data fetch; I/O space 0 1 0 Data store; I/O space 0 1 1 Not used 1 0 0 Instruction fetch; System Memory 1 0 1 Data fetch; System Memory 1 1 0 Data store; System Memroy 1 1 1 Passive The status lines are utilized by the bus controller and bus arbiter to generate all memory and I/O control signals. The signals change during T4 if a new cycle is to be entered while the return to passive state in T3 or T_W indicates the end of a cycle. The pins are floated after system reset and when the bus is not acquired.
READY	I	**Ready:** The ready signal received from the addressed device indicates that the device is ready for data transfer. The signal is active high and is synchronized by the 8284 clock generator.
$\overline{LOCK}$	O	**Lock:** The lock output signal indicates to the bus controller that the bus is needed for more than one contiguous cycle. It is set via the channel control register, and during the TSL instruction. The pin floats after reset and when the bus is not acquired. This output is active low.
RESET	I	**Reset:** The receipt of a reset signal causes the IOP to suspend all its activities and enter an idle state until a channel attention is received. The signal must be active for at least four clock cycles.
CLK	I	**Clock:** Clock provides all timing needed for internal IOP operation.
CA	I	**Channel Attention:** Gets the attention of the IOP. Upon the falling edge of this signal, the SEL input pin is examined to determine Master/Slave or CH1/CH2 information. This input is active high.
SEL	I	**Select:** The first CA received after system reset informs the IOP via the SEL line, whether it is a Master or Slave (0/1 for Master/Slave respectively) and starts the initialization sequence. During any other CA the SEL line signifies the selection of CH1/CH2. (0/1 respectively.)
DRQ1–2	I	**Data Request:** DMA request inputs which signal the IOP that a peripheral is ready to transfer/receive data using channels 1 or 2 respectively. The signals must be held active high until the appropriate fetch/stroke is initiated.
$\overline{RQ}/\overline{GT}$	I/O	**Request Grant:** Request Grant implements the communication dialogue required to arbitrate the use of the system bus (between IOP and CPU, LOCAL mode) or I/O bus when two IOPs share the same bus (REMOTE mode). The $\overline{RQ}/\overline{GT}$ signal is active low. An internal pull-up permits $\overline{RQ}/\overline{GT}$ to be left floating if not used.
SINTR1–2	O	**Signal Interrupt:** Signal Interrupt outputs from channels 1 and 2 respectively. The interrupts may be sent directly to the CPU or through the 8295A interrupt controller. They are used to indicate to the system the occurrence of user defined events.
EXT1–2	I	**External Terminate:** External terminate inputs for channels 1 and 2 respectively. The EXT signals will cause the termination of the current DMA transfer operation if the channel is so programmed by the channel control register. The signal must be held active high until termination is complete.
V_{CC}		**Voltage:** +5 volt power input.
V_{SS}		**Ground.**

FUNCTIONAL DESCRIPTION

The 8089 IOP has been designed to remove I/O processing, control and high speed transfers from the central processing unit. Its major capabilities include that of initializing and maintaining peripheral components and supporting versatile DMA. This DMA function boasts flexible termination conditions (such as external terminate, mask compare, single transfer and byte count expired). The DMA function of the 8089 IOP uses a two cycle approach where the information actually flows through the 8089 IOP. This approach to DMA vastly simplifies the bus timings and enhances compatibility with memory and peripherals, in addition to allowing operations to be performed on the data as it is transferred. Operations can include such constructs as translate, where the 8089 automatically vectors through a lookup table and mask compare, both on the "fly".

The 8089 is functionally compatible with Intel's iAPX 86, 88 family. It supports any combination of 8/16-bit busses. In the REMOTE mode it can be used to complement other Intel processor families. Hardware and communication architecture are designed to provide simple mechanisms for system upgrade.

The only direct communication between the IOP and CPU is handled by the Channel Attention and Interrupt lines. Status information, parameters and task programs are passed via blocks of shared memory, simplifying hardware interface and encouraging structured programming.

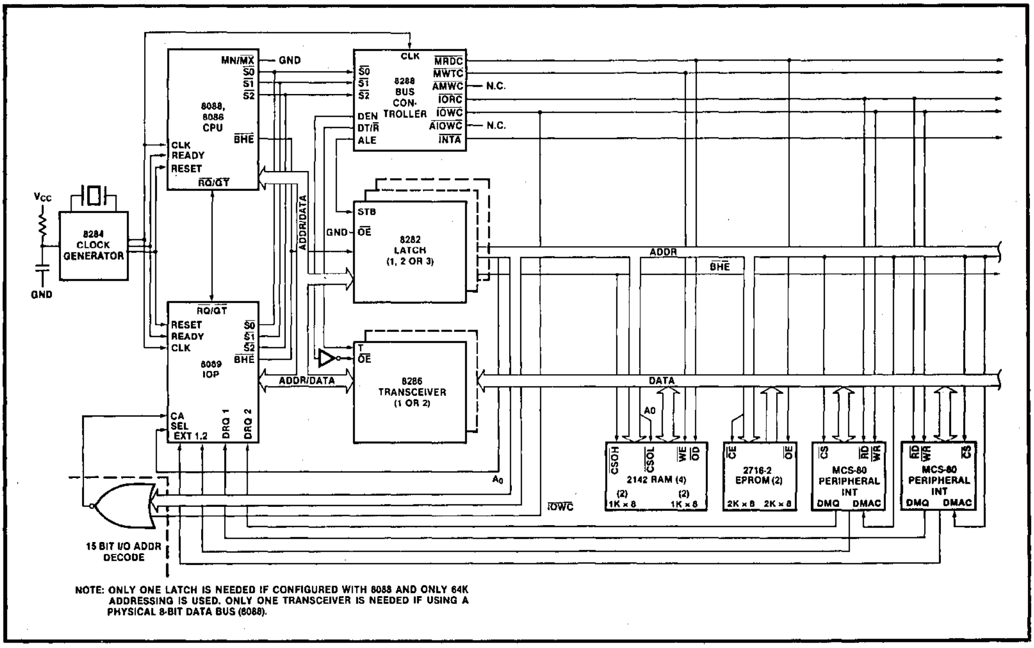
The 8089 can be used in applications such as file and buffer management in hard disk or floppy disk control. It can also provide for soft error recovery routines and scan control. CRT control, such as cursor control and auto scrolling, is simplified with the 8089. Keyboard control, communication control and general I/O are just a few of the typical applications for the 8089.

Remote and Local Modes

Shown in Figure 3 is the 8089 in a LOCAL configuration. The iAPX 86 (or iAPX 88) is used in its maximum mode. The 8089 and iAPX 86 reside on the same local bus, sharing the same set of system buffers. Peripherals located on the system bus can be addressed by either the iAPX 86 or the 8089. The 8089 requests the use of the LOCAL bus by means of the RQ/GT line. This performs a similar function to that of HOLD and HLDA on the Intel 8085A, 8080A and iAPX 86 minimum mode, but is implemented on one physical line. When the iAPX 86 relinquishes the system bus, the 8089 uses the same bus control, latches and transceiver components to generate the system address, control and data lines. This mode allows a more economical system configuration at the expense of reduced CPU thruput due to IOP bus utilization.

A typical REMOTE configuration is shown in Figure 4. In this mode, the IOP's bus is physically separated from the system bus by means of system buffers/latches. The IOP maintains its own local bus and can operate out of local or system memory. The system bus interface contains the following components:

- Up to three 8282 buffer/latches to latch the address to the system bus.
- Up to two 8286 devices bidirectionally buffer the system data bus.



NOTE: ONLY ONE LATCH IS NEEDED IF CONFIGURED WITH 8088 AND ONLY 64K ADDRESSING IS USED. ONLY ONE TRANSCEIVER IS NEEDED IF USING A PHYSICAL 8-BIT DATA BUS (8088).

Figure 3. Typical iAPX 86/11, 88/11 Configuration with 8089 in LOCAL Mode, 8088, 8086 in MAX Mode

- An 8288 bus controller supplies the control signals necessary for buffer operation as well as MRDC (Memory Read) and MWTC (Memory Write) signals.
- An 8289 bus arbiter performs all the functions necessary to arbitrate the use of the system bus. This is used in place of the $\overline{RQ}/\overline{GT}$ logic in the LOCAL mode. This arbiter decodes type of cycle information from the 8089 status lines to determine if the IOP desires to perform a transfer over the "common" or system bus.

The peripheral devices PER1 and PER2 are supported on their own data and address bus. the 8089 communicates with the peripherals without affecting system bus operation. Optional buffers may be used on the local bus when capacitive loading conditions so dictate. I/O programs and RAM buffers may also reside on the local bus to further reduce system bus utilization.

COMMUNICATION MECHANISM

Fundamentally, communication between the CPU and IOP is performed through messages prepared in shared memory. The CPU can cause the 8089 to execute a program by placing it in the 8089's memory space and/or directing the 8089's attention to it by asserting a hardware Channel Attention (CA) signal to the IOP, activating the proper I/O channel. The SEL Pin indicates to the IOP which channel is being addressed. Communication from the IOP to the processor can be performed in a similar manner via a system interrupt (SINTR 1,2), if the CPU has enabled interrupts for this purpose. Additionally, the 8089 can store messages in memory regarding its status and the status of any peripherals. This communication mechanism is supported by a hierarchial data structure to provide a maximum amount of flexibility of memory use with the added capability of handling multiple IOP's.

Illustrated in Figure 5 is an overview of the communication data structure hierarchy that exists for the 8089 I/O processor. Upon the first CA from RESET, if the IOP is initialized as the BUS MASTER, 5 bytes of information are read into the 8089 starting at location FFFF6 (FFFF6, FFFF8-FFFFB) where the type of system bus (16-bit or 8-bit) and pointers to the system configuration block are obtained. This is the only fixed location the 8089 accesses. The remaining addresses are obtained via the data structure hierarchy. The 8089 determines addresses in the same manner as does the iAPX 86; i.e., a 16-bit relocation pointer is offset left 4 bits and added to the 16-bit address offset, obtaining a 20-bit address. Once these 20-bit addresses are formed, they are stored as such, as all the 8089 address registers are 20 bits long. After the system configuration pointer address is formed, the 8089 IOP accesses the system configuration block.

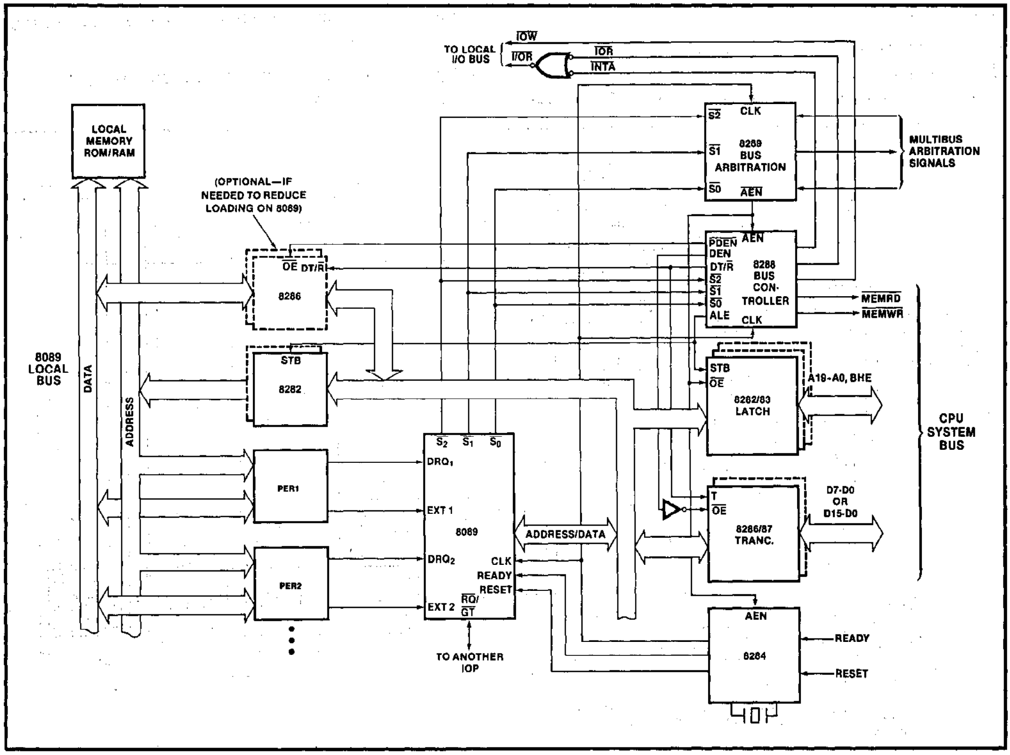


Figure 4. Typical REMOTE Configuration

AFN-00840C

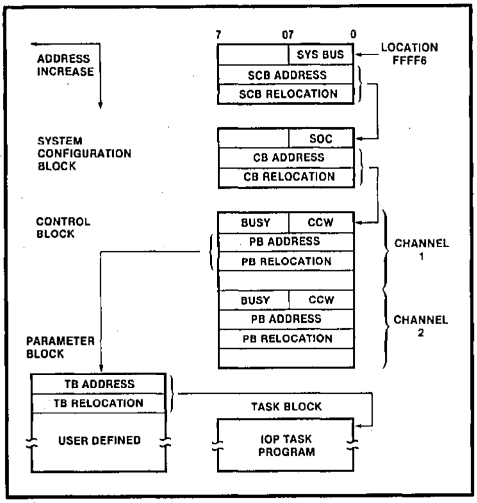


Figure 5. Communication Data Structure Hierarchy

The System Configuration Block (SCB), used only during startup, points to the Control Block (CB) and provides IOP system configuration data via the SOC byte. The SOC byte initializes IOP I/O bus width to 8/16, and defines one of two IOP $\overline{RQ}/\overline{GT}$ operating modes. For $\overline{RQ}/\overline{GT}$ mode 0, the IOP is typically initialized as SLAVE and has its $\overline{RQ}/\overline{GT}$ line tied to a MASTER CPU (typical LOCAL configuration). In this mode, the CPU normally has control of the bus, grants control to the IOP as needed, and has the bus restored to it upon IOP task completion (IOP request—CPU grant—IOP done). For $\overline{RQ}/\overline{GT}$ mode 1, useful only in remote mode between two IOPs, MASTER/SLAVE designation is used only to initialize bus control: from then on, each IOP requests and grants as the bus is needed (IOP1 request—IOP2 grant—IOP2 request—IOP1 grant). Thus, each IOP retains bus control until the other requests it. The completion of initialization is signalled by the IOP clearing the BUSY flag in the CB. This type of startup allows the user to have the startup pointers in ROM with the SCB in RAM. Allowing the SCB to be in RAM gives the user the flexibility of being able to initialize multiple IOPs.

The Control Block furnishes bus control Initialization for the IOP operation (CCW or Channel Control Word) and provides pointers to the Parameter Block or "data" memory for both channels 1 and 2. The CCW is retrieved and analyzed upon all CA's other than the first after a reset. The CCW byte is decoded to determine channel operation.

The Parameter Block contains the address of the Task Block and acts as a messge center between the IOP and CPU. Parameters or variable information is passed from the CPU to its IOP in this block to customize the software interface to the peripheral device. It is also used for transferring data and status information between the IOP and CPU.

The Task Block contains the instructions for the respective channel. This block can reside on the local bus of the IOP, allowing the IOP to operate concurrently with the CPU, or reside in system memory.

The advantage of this type of communication between the processor, IOP and peripheral, is that it allows for a very clean method for the operating system to handle I/O routines. Canned programs or "Task Blocks" allow for execution of general purpose I/O routines with the status and peripheral command information being passed via the Parameter Block ("data" memory). Task Blocks (or "program" memory) can be terminated or restarted by the CPU, if need be. Clearly, the flexibility of this communication lends itself to modularity and applicability to a large number of peripheral devices and upward compatibility to future end user systems and microprocessor families.

Register Set

The 8089 maintains separate registers for its two I/O channels as well as some common registers (see Figure 6). There are sufficient registers for each channel to sustain its own DMA transfers, and process its own instruction stream. The basic DMA pointer registers (GA, GB — 20 bits each), can point to either the system bus or local bus, DMA source or destination, and can be autoincremented. A third register set (GC) can be used to allow translation during the DMA process through a lookup table it points to. Additionally, registers are provided for a masked compare during the data transfer and can be set up to act as one of the termination conditions. Other registers are also provided. Many of these registers can be used as general purpose registers during program execution, when the IOP is not performing DMA cycles.

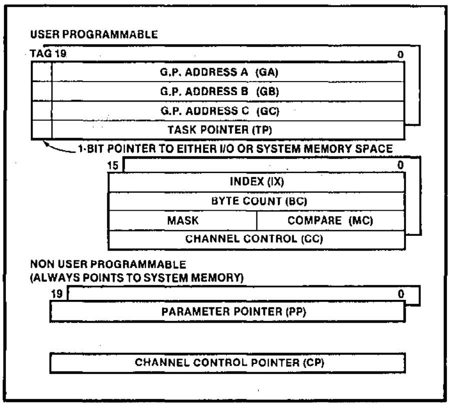


Figure 6. Register Model

Bus Operation

The 8089 utilizes the same bus structure as the iAPX 86, 88 in their maximum mode configurations (see Figure 7). The address is time multiplexed with the data on the first 16/8 lines. A16 through A19 are time multiplexed with four status lines S3-S6. For 8089 cycles, S4 and S3 determine what type of cycle (DMA versus non-DMA) is being performed on channels 1 or 2. S5 and S6 are a unique code assigned to the 8089 IOP, enabling the user to detect which processor is performing a bus cycle in a multiprocessing environment.

The first three status lines, S0-S2, are used with an 8288 bus controller to determine if an instruction fetch or data transfer is being performed in I/O or system memory space.

DMA transfers require at least two bus cycles with each bus cycle requiring a minimum of four clock cycles. Additional clock cycles are added if wait states are required. This two cycle approach simplifies considerably the bus timings in burst DMA. The 8089 optimizes the transfer between two different bus widths by using three bus cycles versus four to transfer 1 word. More than one read (write) is performed when mapping an 8-bit bus onto a 16-bit bus (vice versa). For example, a data transfer from an 8-bit peripheral to a 16-bit physical location in memory is performed by first doing two reads, with word assembly within the IOP assembly register file and then one write.

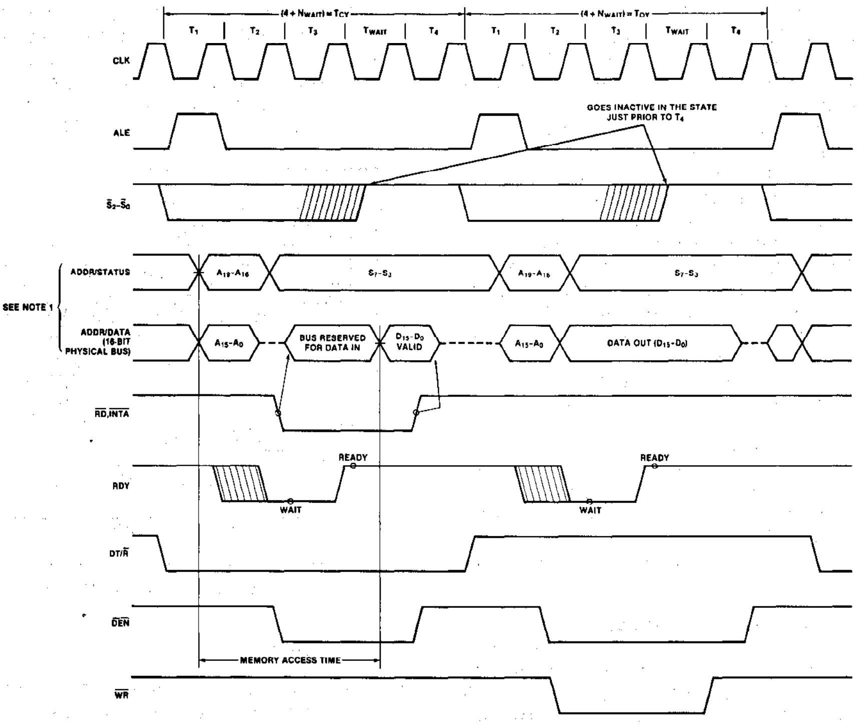
As can be expected, the data bandwidth of the IOP is a function of the physical bus width of the system and I/O busses. Table 2 gives the bandwidth, latency and bus utilization of the 8089. The system bus is assumed to be 16-bits wide with either an 8-bit peripheral (under byte column) or 16-bit peripheral (word column) being shown.

The latency refers to the worst case response time bv the IOP to a DMA request, without the bus arbitration times. Notice that the word transfer allows 50% more bandwidth. This occurs since three bus cycles are required to map 8-bit data into a 16-bit location, versus two for a 16-bit to 16-bit transfer. Note that it is possible to fully saturate the system bus in the LOCAL mode whereas in the REMOTE mode this is reduced to a maximum of 50%.

Table 2. Achievable 5 MHz 8089 Operations

	Local		Remote	
	Byte	Word	Byte	Word
Bandwidth	830 KB/S	1250 KB/S	830 KB/S	1250 KB/S
Latency	1.0/2.4 μsec*	1.0/2.4 μsec*	1.0/2.4 μsec*	1.0/2.4 μsec*
System Bus Utilization	2.4 μsec PER TRANSFER	1.6 μsec PER TRANSFER	0.8 μsec PER TRANSFER	0.8 μsec PER TRANSFER

*2.4 μsec if interleaving with other channel and no wait states. 1μsec if channel is waiting for request.



NOTE 1: $\overline{BHE}$ IS STABLE (i.e., NON MULTIPLEXED) THROUGHOUT EACH TRANSFER CYCLE. A_8-A_{15} ARE ALSO STABLE ON TRANSFERS TO A PHYSICAL 8-BIT BUS.

Figure 7. 8089 Bus Operation

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ....0°C to 70°C
Storage Temperature .... −65°C to +150°C
Voltage on Any Pin with
Respect to Ground .... −1.0 to +7V
Power Dissipation .... 2.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ±10%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} + 1.0	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −400 μA
I_{CC}	Power Supply Current		350	mA	T_A = 25°C
I_{LI}	Input Leakage Current[1]		± 10	μA	0V < VIN < V_{CC}
I_{LO}	Output Leakage Current		± 10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Input Buffer (All input except $AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		15	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer ($AD_0 - AD_{15}$, $\overline{RQ}/\overline{GT}$)		15	pF	fc = 1 MHz

A.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ±10%)

8089/8086 MAX MODE SYSTEM (USING 8288 BUS CONTROLLER) TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(⅔TCLCL) − 15		ns	
TCHCL	CLK High Time	(⅓TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TR1VCL	RDY Setup Time into 8284 (See Notes 1, 2)	35		ns	
TCLR1X	RDY Hold Time into 8284 (See Notes 1, 2)	0		ns	
TRYHCH	READY Setup Time into 8089	(⅔TCLCL) − 15		ns	
TCHRYX	READY Hold Time into 8089	30		ns	
TRYLCL	READY Inactive to CLK (See Note 4)	−8		ns	
TINVCH	Setup Time Recognition (DRQ 1,2 RESET, Ext 1,2) (See Note 2)	30		ns	
TGVCH	$\overline{RQ}/\overline{GT}$ Setup Time	30		ns	
TCAHCAL	CA Width	95		ns	
TSLVCAL	SEL Setup Time	75		ns	
TCALSLX	SEL Hold Time	0		ns	
TCHGX	GT Hold Time into 8089	40		ns	
TILIH	Input Rise Time (Except CLK)		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12	ns	From 2.0V to 0.8V

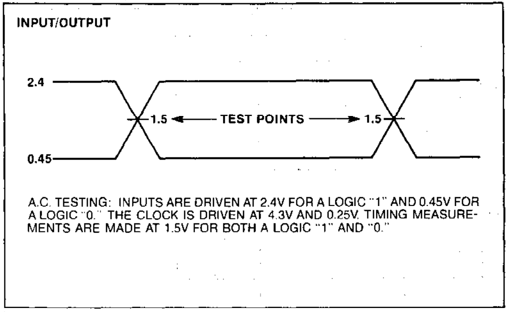
A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	$C_L = 80$ pF
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	$C_L = 150$ pF
TRYHSH	READY Active to Status Passive (See Note 3)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	10	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	$\overline{RQ}$ Active Delay	0	85	ns	$C_L = 100$ pF
TCLGH	$\overline{RQ}$ Inactive Delay		85	ns	Note 5: $C_L = 30$ pF
TCLSRV	SINTR Valid Delay		150	ns	$C_L = 100$ pF
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

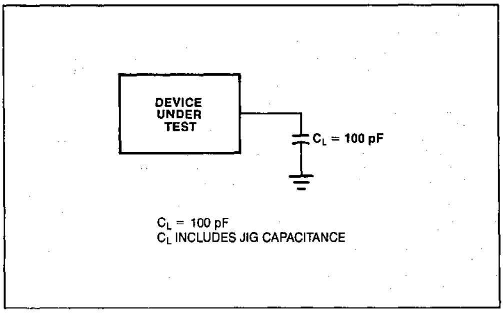
NOTES: 1. Signal at 8284 or 8288 shown for reference only.
2. Setup requirement for asynchronous signal only to guarantee recognition at next CLK.
3. Aplies only to T3 and TW states.
4. Applies only to T2 state.
5. Applies only if RQ/GT Mode 1 C_L=30pf, 2.7 KΩ pull up to Vcc.

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." THE CLOCK IS DRIVEN AT 4.3V AND 0.25V. TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

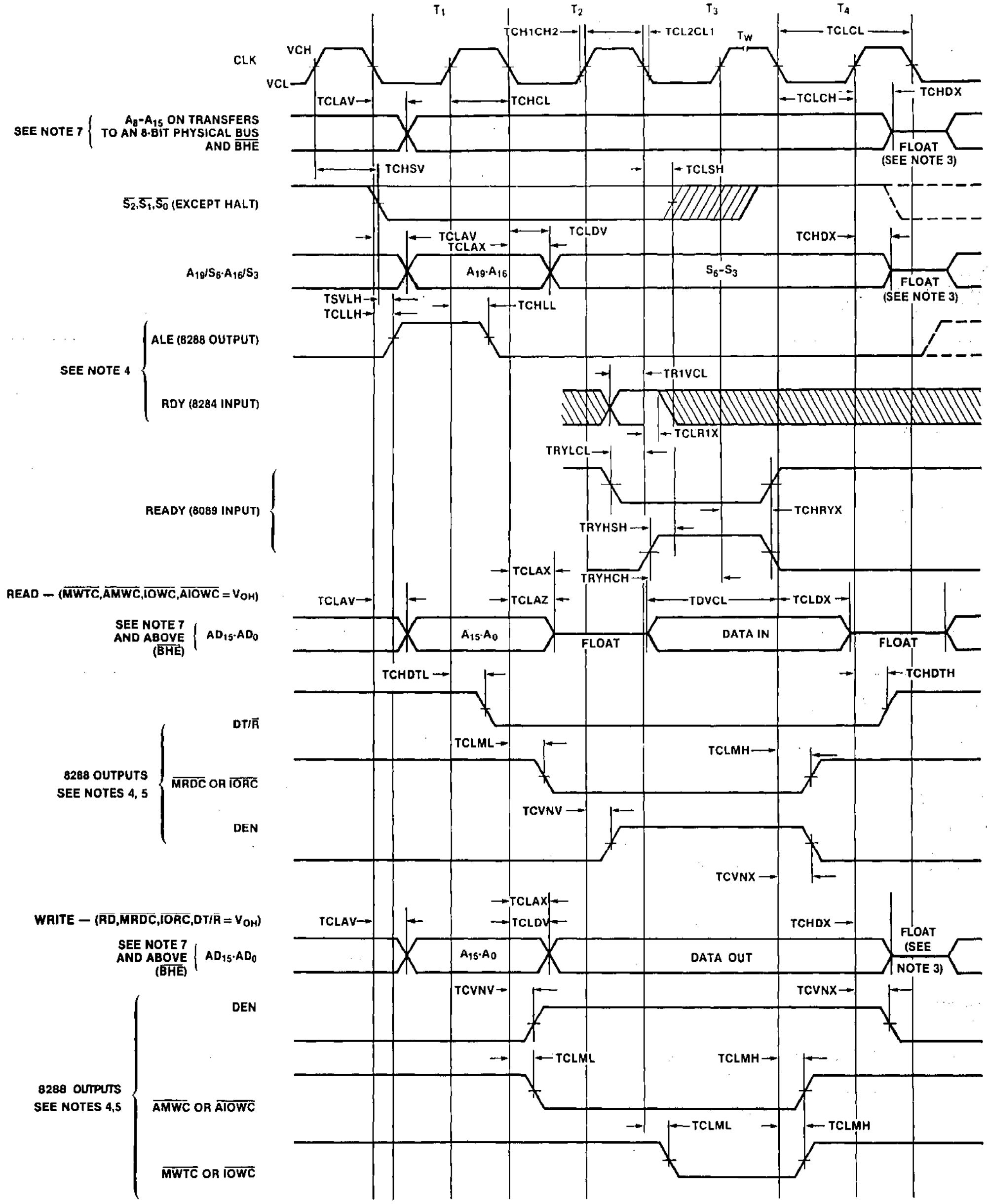
A.C. TESTING LOAD CIRCUIT



$C_L = 100$ pF
C_L INCLUDES JIG CAPACITANCE

WAVEFORMS

8089 BUS TIMING USING 8288

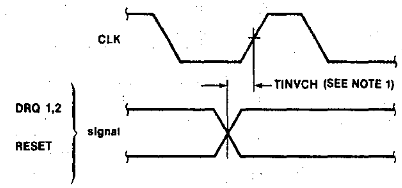

T_1
T_2
T_3
T_4
TCH1CH2
TCL2CL1
T_W
TCLCL
VCH
VCL
CLK
TCLAV
TCHCL
TCLCH
TCHDX
SEE NOTE 7
A_8-A_{15} ON TRANSFERS TO AN 8-BIT PHYSICAL BUS AND $\overline{BHE}$
FLOAT (SEE NOTE 3)
TCHSV
TCLSH
$\overline{S_2}$,$\overline{S_1}$,$\overline{S_0}$ (EXCEPT HALT)
TCLAV
TCLAX
TCLDV
TCHDX
A_{19}/S_6-A_{16}/S_3
A_{19}-A_{16}
S_6-S_3
FLOAT (SEE NOTE 3)
TSVLH
TCLLH
TCHLL
SEE NOTE 4
ALE (8288 OUTPUT)
RDY (8284 INPUT)
TR1VCL
TCLR1X
TRYLCL
READY (8089 INPUT)
TCHRYX
TRYHSH
TRYHCH
TCLAX
READ — ($\overline{MWTC}$,$\overline{AMWC}$,$\overline{IOWC}$,$\overline{AIOWC}$ = V_{OH})
TCLAV
TCLAZ
TDVCL
TCLDX
SEE NOTE 7 AND ABOVE ($\overline{BHE}$)
AD_{15}-AD_0
A_{15}-A_0
FLOAT
DATA IN
FLOAT
TCHDTL
TCHDTH
DT/$\overline{R}$
TCLML
TCLMH
8288 OUTPUTS SEE NOTES 4, 5
$\overline{MRDC}$ OR $\overline{IORC}$
TCVNV
DEN
TCVNX
WRITE — ($\overline{RD}$,$\overline{MRDC}$,$\overline{IORC}$,DT/$\overline{R}$ = V_{OH})
TCLAX
TCLAV
TCLDV
TCHDX
FLOAT (SEE NOTE 3)
SEE NOTE 7 AND ABOVE ($\overline{BHE}$)
AD_{15}-AD_0
A_{15}-A_0
DATA OUT
TCVNV
TCVNX
DEN
TCLML
TCLMH
8288 OUTPUTS SEE NOTES 4,5
$\overline{AMWC}$ OR $\overline{AIOWC}$
TCLML
TCLMH
$\overline{MWTC}$ OR $\overline{IOWC}$

NOTES:

1. ALL SIGNALS SWITCH BETWEEN V_{OH} AND V_{OL} UNLESS OTHERWISE SPECIFIED.
2. RDY IS SAMPLED NEAR THE END OF T_2,T_3,T_W TO DETERMINE IF T_W MACHINE STATES ARE TO BE INSERTED.
3. FOLLOWING A WRITE CYCLE DATA REMAINS VALID ON THE 8089 LOCAL BUS UNTIL A LOCAL BUS MASTER DECIDES TO RUN ANOTHER BUS CYCLE. THE LOCAL BUS IS FLOATED BY THE 8089 WHEN THE 8089 ENTERS A REQUEST BUS ACKNOWLEDGE STATE.
4. SIGNALS AT 8284 OR 8288 ARE SHOWN FOR REFERENCE ONLY.
5. THE ISSUANCE OF THE 8288 COMMAND AND CONTROL SIGNALS ($\overline{MRDC}$, $\overline{MWTC}$, $\overline{AMWC}$, $\overline{IORC}$, $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$, AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
6. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.
7. A_8-A_{15} ARE STABLE ON TRANSFERS TO AN 8-BIT PHYSICAL DATA BUS i.e., A_8-A_{15} DON'T FLOAT ON A READ FROM AN 8-BIT PHYSICAL BUS OR MULTIPLEX WITH DATA ON A WRITE TO AN 8 BIT PHYSICAL BUS $\overline{BHE}$ IS STABLE (NON MULTIPLEXED) FOR ALL TRANSFERS

WAVEFORMS (Continued)

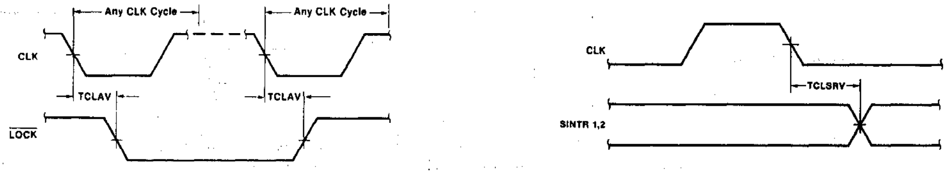
ASYNCHRONOUS SIGNAL RECOGNITION



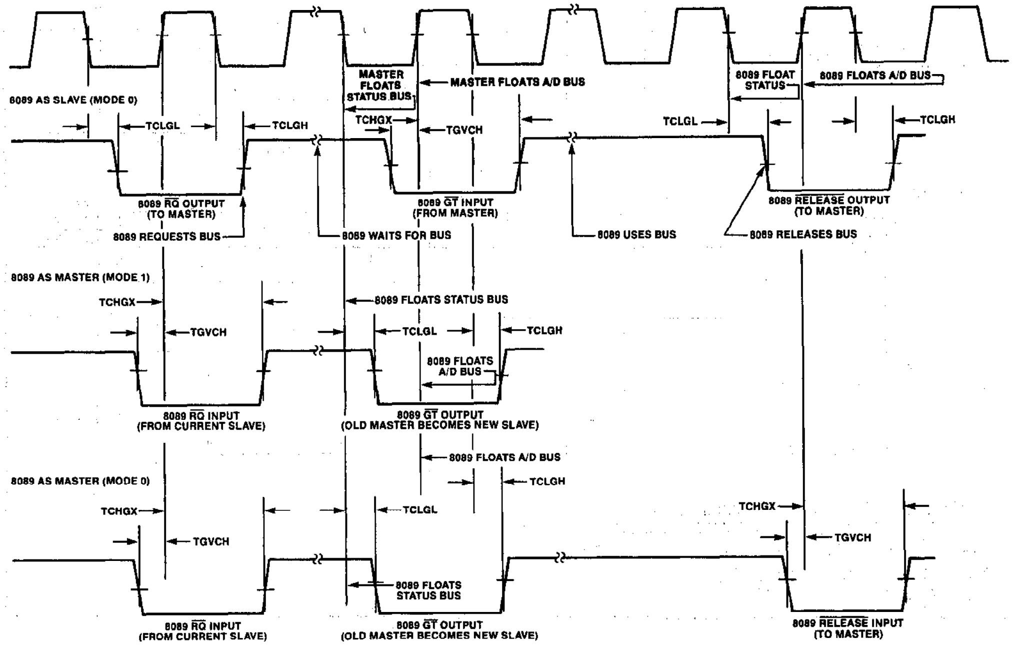
NOTES:

1. SETUP REQUIREMENTS FOR ASYNCHRONOUS SIGNALS ONLY TO GUARANTEE RECOGNITION AT NEXT CLK.
2. ALL INPUTS EXCEPT CA ARE LATCHED ON A CLK EDGE. THE CA INPUT IS NEGATIVE EDGE TRIGGERED.
3. DRQ BECOMING ACTIVE GREATER THAN 30 ns AFTER THE RISING EDGE OF CLK WILL GUARANTEE NON-RECOGNITION UNTIL THE NEXT RISING CLOCK EDGE.

BUS LOCK SIGNAL TIMING AND SINTR

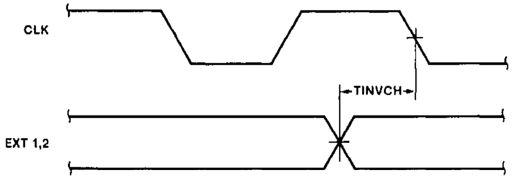


REQUEST/GRANT SEQUENCE

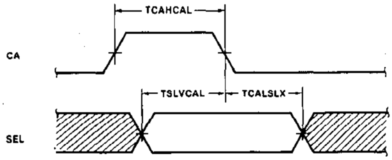


WAVEFORMS (Continued)

EXTERNAL TERMINATE SETUP



SEL SETUP AND TIMING



8089 INSTRUCTION SET SUMMARY

Data Transfers

POINTER INSTRUCTIONS			OPCODE	
			7 0	7 0
LPD	P,M	Load Pointer PPP from Addressed Location	PPP0 0AA1	1000 10MM
LPDI	P,I	Load Pointer PPP Immediate 4 Bytes	PPP1 0001	0000 1000
MOVP	M,P	Store Contents of Pointer PPP in Addressed Location	PPP0 0AA1	1001 10MM
MOVP	P,M	Restore Pointer	PPP0 0AA1	1000 11MM

MOVE DATA			OPCODE	
MOV	M,M	Move from Source to Destination Source—	0000 0AAW	1001 00MM
		Destination—	0000 0AAW	1100 11MM
MOV	R,M	Load Register RRR from Addressed Location	RRR0 0AAW	1000 00MM
MOV	M,R	Store Contents of Register RRR in Addressed Location	RRR0 0AAW	1000 01MM
MOVI	R	Load Register RRR Immediate (Byte) Sign Extend	RRR wb 00W	0011 0000
MOVI	M	Move Immediate to Addressed Location	000 wb AAW	0100 11MM

Control Transfer

CALLS			OPCODE	
			7 0	7 0
*CALL		Call Unconditional	100 dd AAW	1001 11MM

JUMP			OPCODE	
JMP		Unconditional	100 dd 00W	0010 0000
JZ	M	Jump on Zero Memory	000 dd AAW	1110 01MM
JZ	R	Jump on Zero Register	RRR dd 000	0100 0100
JNZ	M	Jump on Non-Zero Memory	000 dd AAW	1110 00MM
JNZ	R	Jump on Non-Zero Register	RRR dd 000	0100 0000
JBT		Test Bit and Jump if True	BBB dd AA0	1011 11MM
JNBT		Test Bit and Jump if Not True	BBB dd AA0	1011 10MM
JMCE		Mask/Compare and Jump on Equal	000 dd AA0	1011 00MM
JMCNE		Mask/Compare and Jump on Non-Equal	000 dd AA0	1011 01MM

Arithmetic and Logic Instructions

INCREMENT, DECREMENT			OPCODE	
			7 0	7 0
*ADDI	M,I	ADD Immediate to Memory	0000 0AAW	1110 10MM
*ADDI	R,I	ADD Immediate to Register	RRR0 0000	0011 1000
†ADD	M,R	ADD Register to Memory	0000 0AAW	1110 11MM
†ADD	R,M	ADD Memory to Register	RRR0 0000	0011 1100

Arithmetic and Logic Instructions

		ADD	OPCODE	
			7 0	7 0
ADDI	M,I	ADD Immediate to Memory	0 0 0 wb A A W	1 1 0 0 0 0 M M
ADDI	R,I	ADD Immediate to Register	R R R wb 0 0 W	0 0 1 0 0 0 0 0
ADD	M,R	ADD Register to Memory	R R R 0 0 A A W	1 1 0 1 0 0 M M
ADD	R,M	ADD Memory to Register	R R R 0 0 A A W	1 0 1 0 0 0 M M

		AND	OPCODE	
ANDI	M,I	AND Memory with Immediate	0 0 0 wb A A W	1 1 0 0 1 0 M M
ANDI	R,I	AND Register with Immediate	R R R wb 0 0 W	0 0 1 0 1 0 0 0
AND	M,R	AND Memory with Register	R R R 0 0 A A W	1 1 0 1 1 0 M M
AND	R,M	AND Register with Memory	R R R 0 0 A A W	1 0 1 0 1 0 M M

		OR	OPCODE	
ORI	M,I	OR Memory with Immediate	0 0 0 wb A A W	1 1 0 0 0 1 M M
ORI	R,I	OR Register with Immediate	R R R wb A A W	0 0 1 0 0 1 0 0
OR	M,R	OR Memory with Register	R R R 0 0 A A W	1 1 0 1 0 1 M M
OR	R,M	OR Register with Memory	R R R 0 0 A A W	1 0 1 0 0 1 M M

		NOT	OPCODE	
NOT	R	Complement Register	R R R 0 0 0 0 0	0 0 1 0 1 1 0 0
NOT	M	Complement Memory	0 0 0 0 0 A A W	1 1 0 1 1 1 M M
NOT	R,M	Complement Memory, Place in Register	R R R 0 0 A A W	1 0 1 0 1 1 M M

Bit Manipulation and Test Instructions

	BIT MANIPULATION	OPCODE	
		7 0	7 0
SET	Set the Selected Bit	B B B 0 0 A A 0	1 1 1 1 0 1 M M
CLR	Clear the Selected Bit	B B B 0 0 A A 0	1 1 1 1 1 0 M M

	TEST	OPCODE	
TSL	Test and Set Lock	0 0 0 1 1 A A 0	1 0 0 1 0 1 M M

Control

	Control	OPCODE	
		7 0	7 0
HLT	Halt Channel Execution	0 0 1 0 0 0 0 0	0 1 0 0 1 0 0 0
SINTR	Set Interrupt Service Flip Flop	0 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0
NOP	No Operation	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
XFER	Enter DMA Transfer	0 1 1 0 0 0 0 0	0 0 0 0 0 0 0 0
WID	Set Source, Destination Bus Width; S,D 0 = 8, 1 = 16	1 S D 0 0 0 0 0	0 0 0 0 0 0 0 0

*II field in call instruction can be 00, 01, 10 only.
**OPCODE is second byte fetched.

All instructions consist of at least 2 bytes, while some instructions may use up to 3 additional bytes to specify literals and displacement data. The definition of the various fields within each instruction is given below:

7			0	7		0
R R R	w b	A A	W	OPCODE	M M	
PPP BBB						

MM	Base Pointer Select
00	GA
01	GB
10	GC
11	PP

RRR Register Field

The RRR field specifies a 16-bit register to be used in the instruction. If GA, GB, GC or TP, are referenced by the RRR field, the upper 4 bits of the registers are loaded with the sign bit (Bit 15). PPP registers are used as 20-bit address pointers.

RRR			
000	r0	GA	
001	r1	GB	
010	r2	GC	
011	r3	BC	; byte count
100	r4	TP	; task block
101	r5	IX	; index register
110	r6	CC	; channel control (mode)
111	r7	MC	; mask/compare

PPP			
000	p0	GA	;
001	p1	GB	;
010	p2	GC	:
100	p4	TP	; task block pointer

NOTES:

BBB Bit Select Field

The bit select field replaces the RRR field in bit manipulation instructions and is used to select a bit to be operated on by those instructions. Bit 0 is the least significant bit.

wb

01	1 byte literal
10	2 byte (word) literal

dd

01	1 byte displacement
10	2 byte (word) displacement.

AA Field

00 The selected pointer contains the operand address.

01 The operand address is formed by adding an 8-bit, unsigned, offset contained in the instruction to the selected pointer. The contents of the pointer are unchanged.

10 The operand address is formed by adding the contents of the Index register to the selected pointer. Both registers remain unchanged.

11 Same as 10 except the Index register is post autoincremented (by 1 for 8-bit transfer, by 2 for 16-bit transfer).

W Width Field

0 The selected operand is 1 byte long.

1 The selected operand is 2 bytes long.

Additional Bytes

OFFSET :	8-bit unsigned offset.
SDISP :	8/16-bit signed displacement.
LITERAL :	8/16-bit literal. (32 bits for LDPI).

The order in which the above optional bytes appear in IOP instructions is given below:

OFFSET	LITERAL	SDISP

Offsets are treated as unsigned numbers. Literals and displacements are sign extended (2's complement).

8259A/8259A-2/8259A-8
PROGRAMMABLE INTERRUPT CONTROLLER

- **iAPX 86, iAPX 88 Compatible**
- **MCS-80®, MCS-85® Compatible**
- **Eight-Level Priority Controller**
- **Expandable to 64 Levels**
- **Programmable Interrupt Modes**
- **Individual Request Mask Capability**
- **Single +5V Supply (No Clocks)**
- **28-Pin Dual-In-Line Package**

The Intel® 8259A Programmable Interrupt Controller handles up to eight vectored priority interrupts for the CPU. It is cascadable for up to 64 vectored priority interrupts without additional circuitry. It is packaged in a 28-pin DIP, uses NMOS technology and requires a single +5V supply. Circuitry is static, requiring no clock input.

The 8259A is designed to minimize the software and real time overhead in handling multi-level priority interrupts. It has several modes, permitting optimization for a variety of system requirements.

The 8259A is fully upward compatible with the Intel® 8259. Software originally written for the 8259 will operate the 8259A in all 8259 equivalent modes (MCS-80/85, Non-Buffered, Edge Triggered).

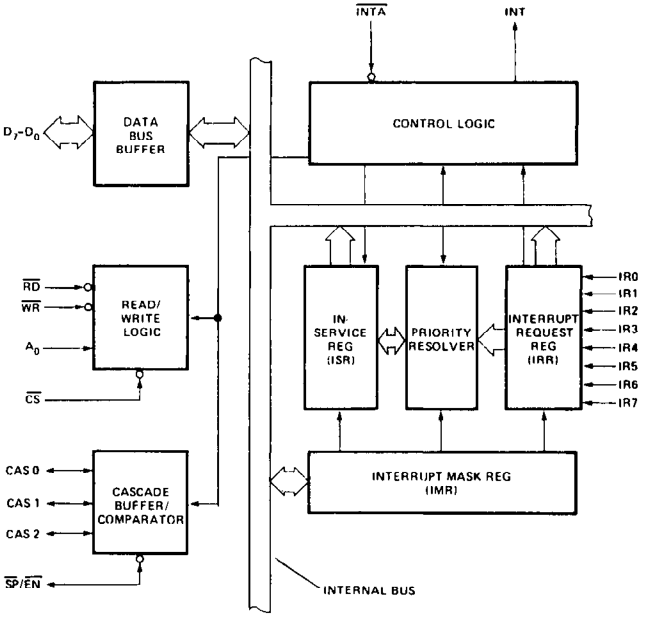


Figure 1. Block Diagram

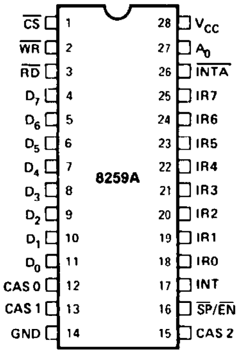


Figure 2. Pin Configuration

Table 1. Pin Description

Symbol	Pin No.	Type	Name and Function
V_{CC}	28	I	**Supply:** +5V Supply.
GND	14	I	**Ground.**
$\overline{CS}$	1	I	**Chip Select:** A low on this pin enables $\overline{RD}$ and $\overline{WR}$ communication between the CPU and the 8259A. INTA functions are independent of CS.
$\overline{WR}$	2	O	**Write:** A low on this pin when CS is low enables the 8259A to accept command words from the CPU.
$\overline{RD}$	3	I	**Read:** A low on this pin when CS is low enables the 8259A to release status onto the data bus for the CPU.
D_7–D_0	4–11	I/O	**Bidirectional Data Bus:** Control, status and interrupt-vector information is transferred via this bus.
CAS_0–CAS_2	12, 13, 15	I/O	**Cascade Lines:** The CAS lines form a private 8259A bus to control a multiple 8259A structure. These pins are outputs for a master 8259A and inputs for a slave 8259A.
$\overline{SP}/\overline{EN}$	16	I/O	**Slave Program/Enable Buffer:** This is a dual function pin. When in the Buffered Mode it can be used as an output to control buffer transceivers (EN). When not in the buffered mode it is used as an input to designate a master (SP = 1) or slave (SP = 0).
INT	17	O	**Interrupt:** This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus it is connected to the CPU's interrupt pin.
IR_0–IR_7	18–25	I	**Interrupt Requests:** Asynchronous inputs. An interrupt request is executed by raising an IR input (low to high), and holding it high until it is acknowledged (Edge Triggered Mode), or just by a high level on an IR input (Level Triggered Mode).
$\overline{INTA}$	26	I	**Interrupt Acknowledge:** This pin is used to enable 8259A interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the CPU.
A_0	27	I	**AO Address Line:** This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 8259A to decipher various Command Words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for iAPX 86, 88).

FUNCTIONAL DESCRIPTION

Interrupts in Microcomputer Systems

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors and other components receive servicing in an efficient manner so that large amounts of the total system tasks can be assumed by the microcomputer with little or no effect on throughput.

The most common method of servicing such devices is the *Polled* approach. This is where the processor must test each device in sequence and in effect "ask" each one if it needs servicing. It is easy to see that a large portion of the main program is looping through this continuous polling cycle and that such a method would have a serious, detrimental effect on system throughput, thus limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices.

A more desirable method would be one that would allow the microprocessor to be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete, however, the processor would resume exactly where it left off.

This method is called *Interrupt*. It is easy to see that system throughput would drastically increase, and thus more tasks could be assumed by the microcomputer to further enhance its cost effectiveness.

The Programmable Interrupt Controller (PIC) functions as an overall manager in an Interrupt-Driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

Each peripheral device or structure usually has a special program or "routine" that is associated with its specific functional or operational requirements; this is referred to as a "service routine". The PIC, after issuing an Interrupt to the CPU, must somehow input information into the CPU that can "point" the Program Counter to the service routine associated with the requesting device. This "pointer" is an address in a vectoring table and will often be referred to, in this document, as vectoring data.

The 8259A

The 8259A is a device specifically designed for use in real time, interrupt driven microcomputer systems. It manages eight levels or requests and has built-in features for expandability to other 8259A's (up to 64 levels). It is programmed by the system's software as an I/O peripheral. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the 8259A can be configured to match his system requirements. The priority modes can be changed or reconfigured dynamically at any time during the main program. This means that the complete interrupt structure can be defined as required, based on the total system environment.

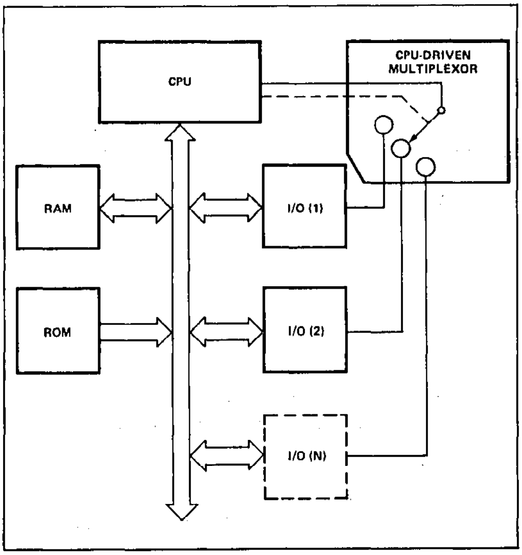


Figure 3a. Polled Method

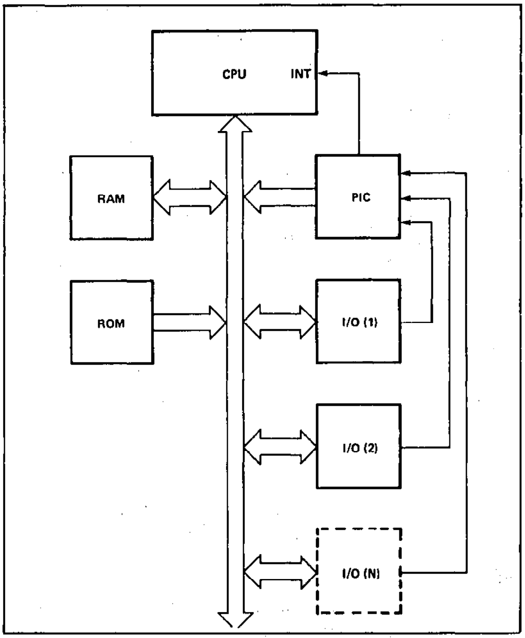


Figure 3b. Interrupt Method

INTERRUPT REQUEST REGISTER (IRR) AND IN-SERVICE REGISTER (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to store all the interrupt levels which are requesting service; and the ISR is used to store all the interrupt levels which are being serviced.

PRIORITY RESOLVER

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during $\overline{\text{INTA}}$ pulse.

INTERRUPT MASK REGISTER (IMR)

The IMR stores the bits which mask the interrupt lines to be masked. The IMR operates on the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

INT (INTERRUPT)

This output goes directly to the CPU interrupt input. The V_{OH} level on this line is designed to be fully compatible with the 8080A, 8085A and 8086 input levels.

$\overline{\text{INTA}}$ (INTERRUPT ACKNOWLEDGE)

$\overline{\text{INTA}}$ pulses will cause the 8259A to release vectoring information onto the data bus. The format of this data depends on the system mode (μPM) of the 8259A.

DATA BUS BUFFER

This 3-state, bidirectional 8-bit buffer is used to interface the 8259A to the system Data Bus. Control words and status information are transferred through the Data Bus Buffer.

READ/WRITE CONTROL LOGIC

The function of this block is to accept OUTput commands from the CPU. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the 8259A to be transferred onto the Data Bus.

$\overline{\text{CS}}$ (CHIP SELECT)

A LOW on this input enables the 8259A. No reading or writing of the chip will occur unless the device is selected.

$\overline{\text{WR}}$ (WRITE)

A LOW on this input enables the CPU to write control words (ICWs and OCWs) to the 8259A.

$\overline{\text{RD}}$ (READ)

A LOW on this input enables the 8259A to send the status of the Interrupt Request Register (IRR), In Service Register (ISR), the Interrupt Mask Register (IMR), or the Interrupt level onto the Data Bus.

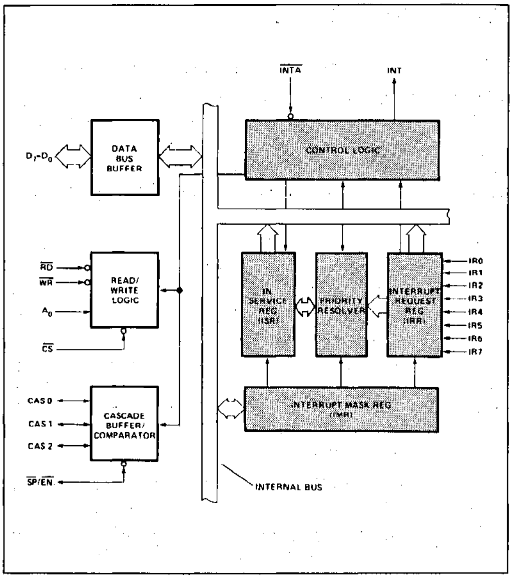


Figure 4a. 8259A Block Diagram

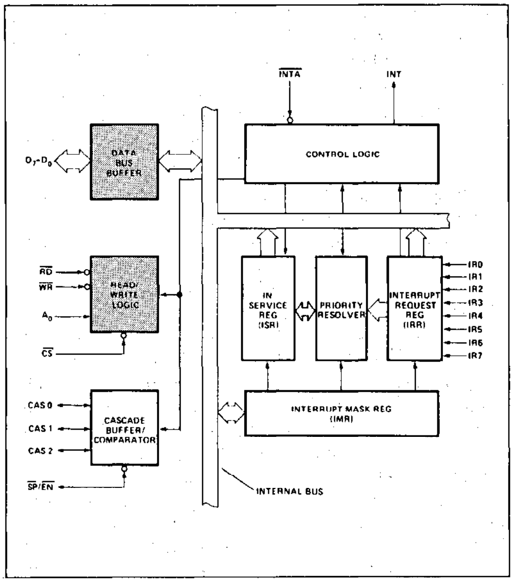


Figure 4b. 8259A Block Diagram

A_0

This input signal is used in conjunction with $\overline{\text{WR}}$ and $\overline{\text{RD}}$ signals to write commands into the various command registers, as well as reading the various status registers of the chip. This line can be tied directly to one of the address lines.

THE CASCADE BUFFER/COMPARATOR

This function block stores and compares the IDs of all 8259A's used in the system. The associated three I/O pins (CAS0-2) are outputs when the 8259A is used as a master and are inputs when the 8259A is used as a slave. As a master, the 8259A sends the ID of the interrupting slave device onto the CAS0-2 lines. The slave thus selected will send its preprogrammed subroutine address onto the Data Bus during the next one or two consecutive $\overline{\text{INTA}}$ pulses. (See section "Cascading the 8259A".)

INTERRUPT SEQUENCE

The powerful features of the 8259A in a microcomputer system are its programmability and the interrupt routine addressing capability. The latter allows direct or indirect jumping to the specific interrupt routine requested without any polling of the interrupting devices. The normal sequence of events during an interrupt depends on the type of CPU being used.

The events occur as follows in an MCS-80/85 system:

1. One or more of the INTERRUPT REQUEST lines (IR7-0) are raised high, setting the corresponding IRR bit(s).
2. The 8259A evaluates these requests, and sends an INT to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an $\overline{\text{INTA}}$ pulse.
4. Upon receiving an $\overline{\text{INTA}}$ from the CPU group, the highest priority ISR bit is set, and the corresponding IRR bit is reset. The 8259A will also release a CALL instruction code (11001101) onto the 8-bit Data Bus through its D7-0 pins.
5. This CALL instruction will initiate two more $\overline{\text{INTA}}$ pulses to be sent to the 8259A from the CPU group.
6. These two $\overline{\text{INTA}}$ pulses allow the 8259A to release its preprogrammed subroutine address onto the Data Bus. The lower 8-bit address is released at the first $\overline{\text{INTA}}$ pulse and and the higher 8-bit address is released at the second $\overline{\text{INTA}}$ pulse.
7. This completes the 3-byte CALL instruction released by the 8259A. In the AEOI mode the ISR bit is reset at the end of the third $\overline{\text{INTA}}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt sequence.

The events occurring in an iAPX 86 system are the same until step 4.

4. Upon receiving an $\overline{\text{INTA}}$ from the CPU group, the highest priority ISR bit is set and the corresponding IRR bit is reset. The 8259A does not drive the Data Bus during this cycle.
5. The iAPX 86/10 will initiate a second $\overline{\text{INTA}}$ pulse. During this pulse, the 8259A releases an 8-bit pointer onto the Data Bus where it is read by the CPU.
6. This completes the interrupt cycle. In the AEOI mode the ISR bit is reset at the end of the second $\overline{\text{INTA}}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt subroutine.

If no interrupt request is present at step 4 of either sequence (i.e., the request was too short in duration) the 8259A will issue an interrupt level 7. Both the vectoring bytes and the CAS lines will look like an interrupt level 7 was requested.

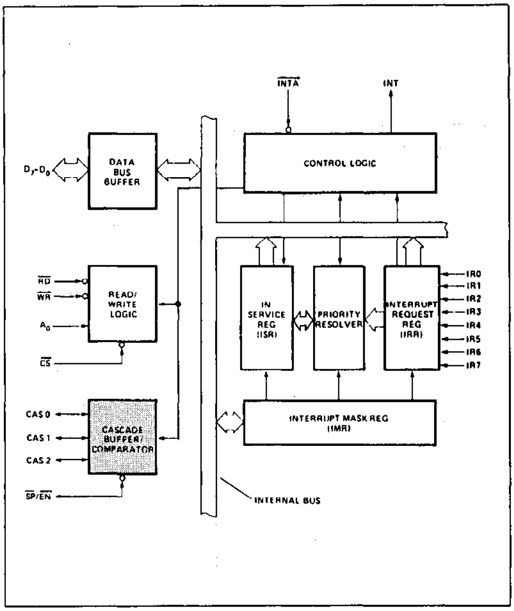


Figure 4c. 8259A Block Diagram

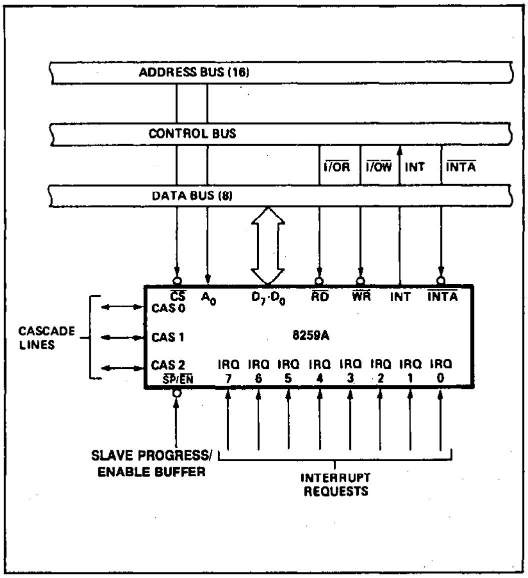


Figure 5. 8259A Interface to Standard System Bus

INTERRUPT SEQUENCE OUTPUTS

MCS-80®, MCS-85®

This sequence is timed by three $\overline{\text{INTA}}$ pulses. During the first $\overline{\text{INTA}}$ pulse the CALL opcode is enabled onto the data bus.

Content of First Interrupt Vector Byte

	D7	D6	D5	D4	D3	D2	D1	D0
CALL CODE	1	1	0	0	1	1	0	1

During the second $\overline{\text{INTA}}$ pulse the lower address of the appropriate service routine is enabled onto the data bus. When Interval = 4 bits A_5-A_7 are programmed, while A_0-A_4 are automatically inserted by the 8259A. When Interval = 8 only A_6 and A_7 are programmed, while A_0-A_5 are automatically inserted.

Content of Second Interrupt Vector Byte

IR	Interval = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

IR	Interval = 8							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

During the third INTA pulse the higher address of the appropriate service routine, which was programmed as byte 2 of the initialization sequence ($A_8 - A_{15}$), is enabled onto the bus.

Content of Third Interrupt Vector Byte

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

iAPX 86, iAPX 88

iAPX 86 mode is similar to MCS-80 mode except that only two Interrupt Acknowledge cycles are issued by the processor and no CALL opcode is sent to the processor. The first interrupt acknowledge cycle is similar to that of MCS-80, 85 systems in that the 8259A uses it to internally freeze the state of the interrupts for priority resolution and as a master it issues the interrupt code on the cascade lines at the end of the INTA pulse. On this first cycle it does not issue any data to the processor and leaves its data bus buffers disabled. On the second interrupt acknowledge cycle in iAPX 86 mode the master (or slave if so programmed) will send a byte of data to the processor with the acknowledged interrupt code composed as follows (note the state of the ADI mode control is ignored and A_5–A_{11} are unused in iAPX 86 mode):

Content of Interrupt Vector Byte for iAPX 86 System Mode

	D7	D6	D5	D4	D3	D2	D1	D0
IR7	T7	T6	T5	T4	T3	1	1	1
IR6	T7	T6	T5	T4	T3	1	1	0
IR5	T7	T6	T5	T4	T3	1	0	1
IR4	T7	T6	T5	T4	T3	1	0	0
IR3	T7	T6	T5	T4	T3	0	1	1
IR2	T7	T6	T5	T4	T3	0	1	0
IR1	T7	T6	T5	T4	T3	0	0	1
IR0	T7	T6	T5	T4	T3	0	0	0

PROGRAMMING THE 8259A

The 8259A accepts two types of command words generated by the CPU:

1. *Initialization Command Words (ICWs):* Before normal operation can begin, each 8259A in the system must be brought to a starting point — by a sequence of 2 to 4 bytes timed by $\overline{\text{WR}}$ pulses.
2. *Operation Command Words (OCWs):* These are the command words which command the 8259A to operate in various interrupt modes. These modes are:
   a. Fully nested mode
   b. Rotating priority mode
   c. Special mask mode
   d. Polled mode

The OCWs can be written into the 8259A anytime after initialization.

INITIALIZATION COMMAND WORDS (ICWS)

GENERAL

Whenever a command is issued with A0 = 0 and D4 = 1, this is interpreted as Initialization Command Word 1 (ICW1). ICW1 starts the initialization sequence during which the following automatically occur.

a. The edge sense circuit is reset, which means that following initialization, an interrupt request (IR) input must make a low-to-high transition to generate an interrupt.

b. The Interrupt Mask Register is cleared.

c. IR7 input is assigned priority 7.

d. The slave mode address is set to 7.

e. Special Mask Mode is cleared and Status Read is set to IRR.

f. If IC4=0, then all functions selected in ICW4 are set to zero. (Non-Buffered mode*, no Auto-EOI, MCS-80, 85 system).

*Note: Master/Slave in ICW4 is only used in the buffered mode.

INITIALIZATION COMMAND WORDS 1 AND 2 (ICW1, ICW2)

A_5-A_{15}: *Page starting address of service routines.* In an MCS 80/85 system, the 8 request levels will generate CALLs to 8 locations equally spaced in memory. These can be programmed to be spaced at intervals of 4 or 8 memory locations, thus the 8 routines will occupy a page of 32 or 64 bytes, respectively.

The address format is 2 bytes long (A_0-A_{15}). When the routine interval is 4, A_0-A_4 are automatically inserted by the 8259A, while A_5-A_{15} are programmed externally. When the routine interval is 8, A_0-A_5 are automatically inserted by the 8259A, while A_6-A_{15} are programmed externally.

The 8-byte interval will maintain compatibility with current software, while the 4-byte interval is best for a compact jump table.

In an iAPX 86 system A_{15}-A_{11} are inserted in the five most significant bits of the vectoring byte and the 8259A sets the three least significant bits according to the interrupt level. A_{10}-A_5 are ignored and ADI (Address interval) has no effect.

LTIM: If LTIM = 1, then the 8259A will operate in the level interrupt mode. Edge detect logic on the interrupt inputs will be disabled.

ADI: CALL address interval. ADI = 1 then interval = 4; ADI = 0 then interval = 8.

SNGL: Single. Means that this is the only 8259A in the system. If SNGL = 1 no ICW3 will be issued.

IC4: If this bit is set — ICW4 has to be read. If ICW4 is not needed, set IC4 = 0.

INITIALIZATION COMMAND WORD 3 (ICW3)

This word is read only when there is more than one 8259A in the system and cascading is used, in which case SNGL = 0. It will load the 8-bit slave register. The functions of this register are:

a. In the master mode (either when SP = 1, or in buffered mode when M/S = 1 in ICW4) a "1" is set for each slave in the system. The master then will release byte 1 of the call sequence (for MCS-80/85 system) and will enable the corresponding slave to release bytes 2 and 3 (for iAPX 86 only byte 2) through the cascade lines.

b. In the slave mode (either when $\overline{SP}$ = 0, or if BUF = 1 and M/S = 0 in ICW4) bits 2-0 identify the slave. The slave compares its cascade input with these bits and, if they are equal, bytes 2 and 3 of the call sequence (or just byte 2 for iAPX 86 are released by it on the Data Bus.

INITIALIZATION COMMAND WORD 4 (ICW4)

SFNM: If SFNM = 1 the special fully nested mode is programmed.

BUF: If BUF = 1 the buffered mode is programmed. In buffered mode $\overline{SP}/\overline{EN}$ becomes an enable output and the master/slave determination is by M/S.

M/S: If buffered mode is selected: M/S = 1 means the 8259A is programmed to be a master, M/S = 0 means the 8259A is programmed to be a slave. If BUF = 0, M/S has no function.

AEOI: If AEOI = 1 the automatic end of interrupt mode is programmed.

μPM: Microprocessor mode: μPM = 0 sets the 8259A for MCS-80, 85 system operation, μPM = 1 sets the 8259A for iAPX 86 system operation.

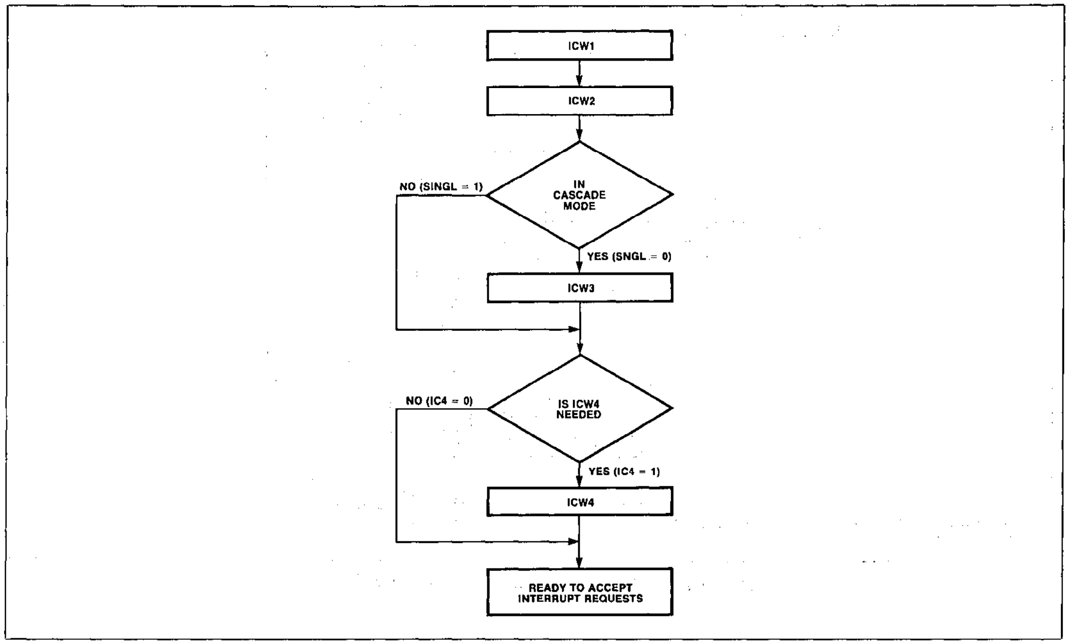
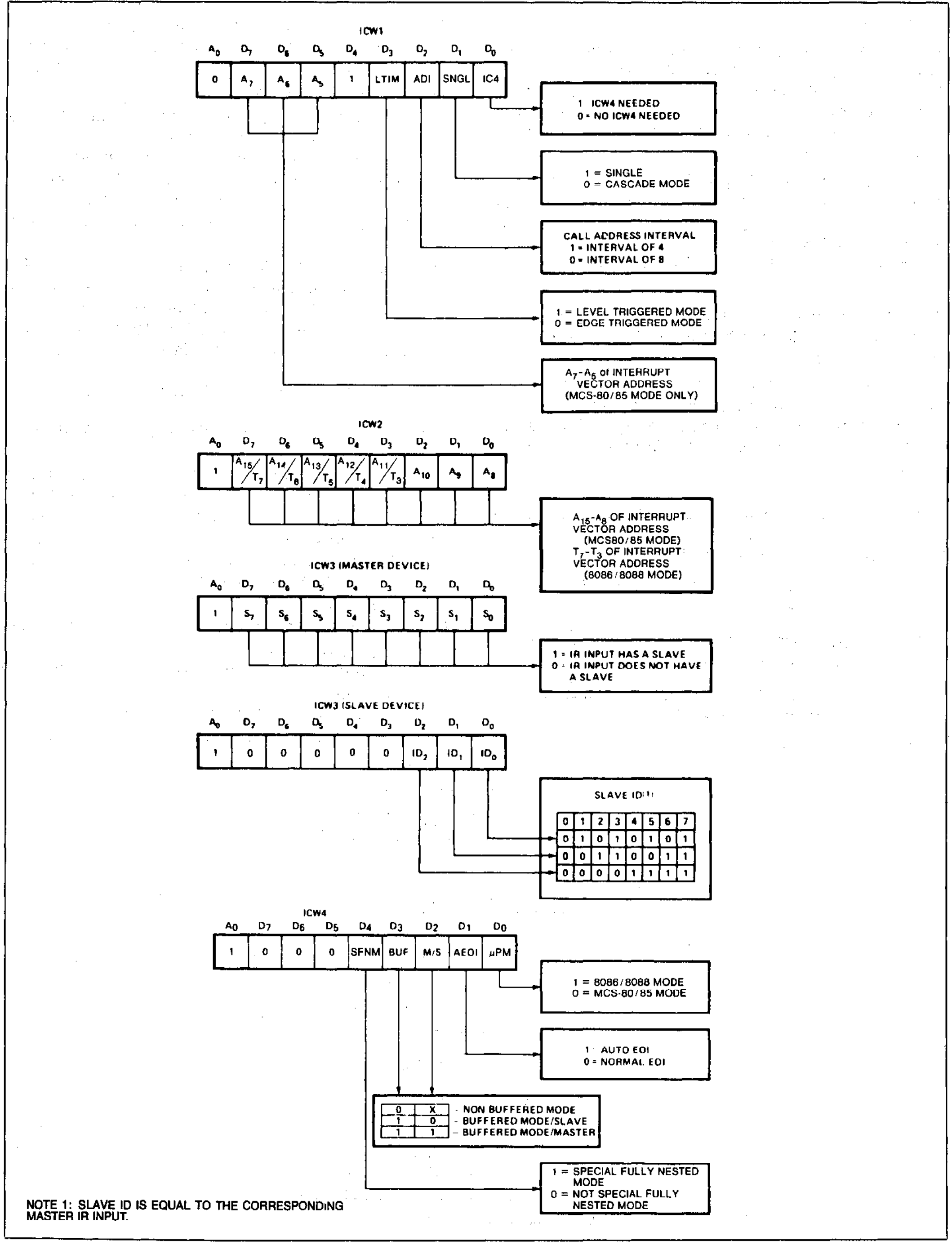


Figure 6. Initialization Sequence


ICW1
A0 D7 D6 D5 D4 D3 D2 D1 D0
0 A7 A6 A5 1 LTIM ADI SNGL IC4
1 ICW4 NEEDED
0 = NO ICW4 NEEDED
1 = SINGLE
0 = CASCADE MODE
CALL ADDRESS INTERVAL
1 = INTERVAL OF 4
0 = INTERVAL OF 8
1 = LEVEL TRIGGERED MODE
0 = EDGE TRIGGERED MODE
A7-A5 of INTERRUPT VECTOR ADDRESS (MCS-80/85 MODE ONLY)
ICW2
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 A15/T7 A14/T6 A13/T5 A12/T4 A11/T3 A10 A9 A8
A15-A8 OF INTERRUPT VECTOR ADDRESS (MCS80/85 MODE)
T7-T3 OF INTERRUPT VECTOR ADDRESS (8086/8088 MODE)
ICW3 (MASTER DEVICE)
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 S7 S6 S5 S4 S3 S2 S1 S0
1 = IR INPUT HAS A SLAVE
0 = IR INPUT DOES NOT HAVE A SLAVE
ICW3 (SLAVE DEVICE)
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 0 0 0 0 0 ID2 ID1 ID0
SLAVE ID(1)
ICW4
A0 D7 D6 D5 D4 D3 D2 D1 D0
1 0 0 0 SFNM BUF M/S AEOI µPM
1 = 8086/8088 MODE
0 = MCS-80/85 MODE
1 AUTO EOI
0 = NORMAL EOI
1 = SPECIAL FULLY NESTED MODE
0 = NOT SPECIAL FULLY NESTED MODE

SLAVE ID(1)

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

0	X	- NON BUFFERED MODE
1	0	- BUFFERED MODE/SLAVE
1	1	- BUFFERED MODE/MASTER

NOTE 1: SLAVE ID IS EQUAL TO THE CORRESPONDING MASTER IR INPUT.

Figure 7. Initialization Command Word Format

OPERATION COMMAND WORDS (OCWs)

After the Initialization Command Words (ICWs) are programmed into the 8259A, the chip is ready to accept interrupt requests at its input lines. However, during the 8259A operation, a selection of algorithms can command the 8259A to operate in various modes through the Operation Command Words (OCWs).

OPERATION CONTROL WORDS (OCWs)

OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

OCW2

0	R	SL	EOI	0	0	L2	L1	L0

OCW3

0	0	ESMM	SMM	0	1	P	RR	RIS

OPERATION CONTROL WORD 1 (OCW1)

OCW1 sets and clears the mask bits in the interrupt Mask Register (IMR). $M_7 - M_0$ represent the eight mask bits. M = 1 indicates the channel is masked (inhibited), M = 0 indicates the channel is enabled.

OPERATION CONTROL WORD 2 (OCW2)

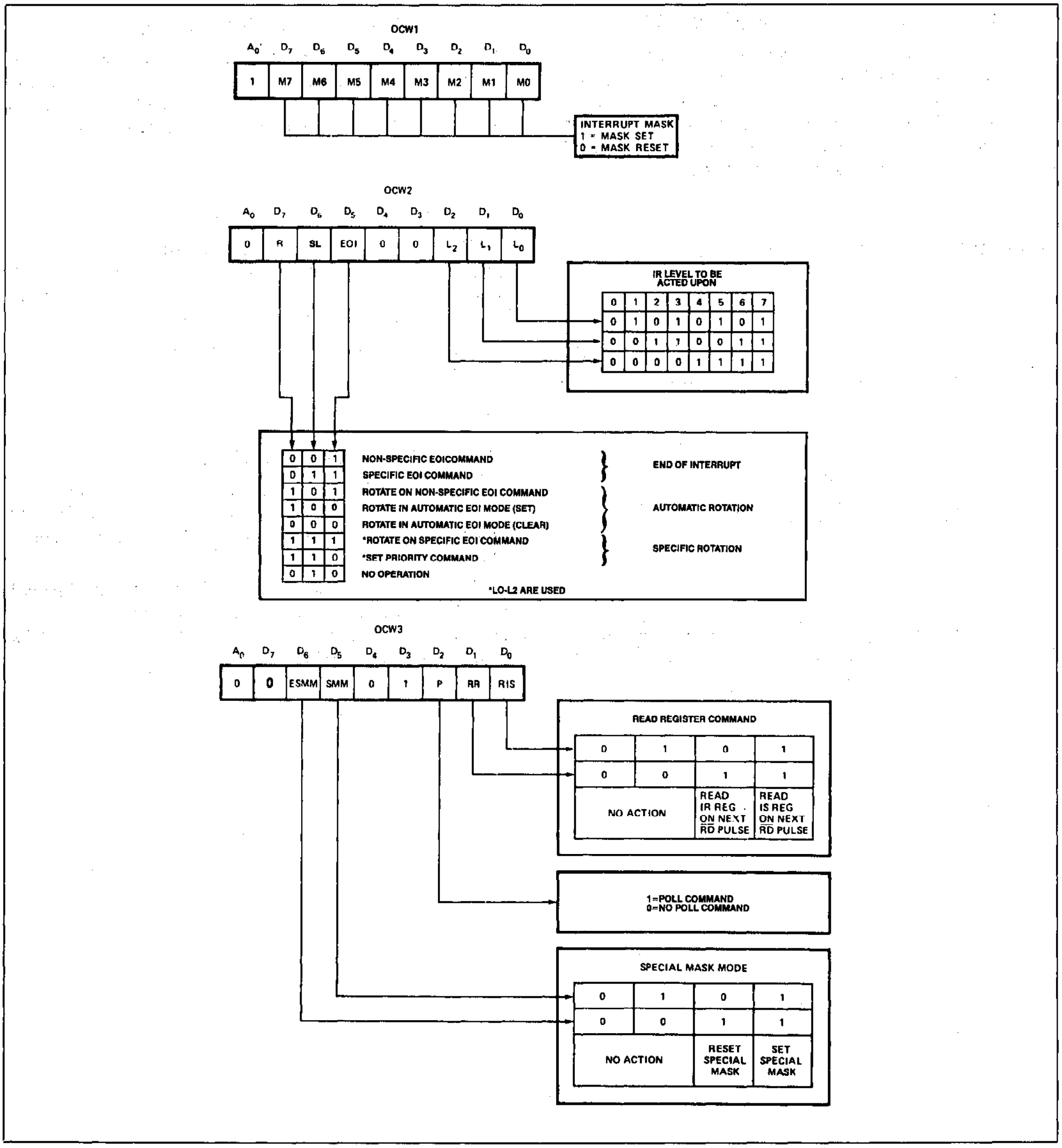
R, SL, EOI — These three bits control the Rotate and End of Interrupt modes and combinations of the two. A chart of these combinations can be found on the Operation Command Word Format.

L_2, L_1, L_0—These bits determine the interrupt level acted upon when the SL bit is active.

OPERATION CONTROL WORD 3 (OCW3)

ESMM — Enable Special Mask Mode. When this bit is set to 1 it enables the SMM bit to set or reset the Special Mask Mode. When ESMM = 0 the SMM bit becomes a "don't care".

SMM — Special Mask Mode. If ESMM = 1 and SMM = 1 the 8259A will enter Special Mask Mode. If ESMM = 1 and SMM = 0 the 8259A will revert to normal mask mode. When ESMM = 0, SMM has no effect.



OCW1

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

INTERRUPT MASK
1 = MASK SET
0 = MASK RESET

OCW2

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	R	SL	EOI	0	0	L2	L1	L0

IR LEVEL TO BE ACTED UPON

	0	1	2	3	4	5	6	7
L0	0	1	0	1	0	1	0	1
L1	0	0	1	1	0	0	1	1
L2	0	0	0	0	1	1	1	1

R	SL	EOI		
0	0	1	NON-SPECIFIC EOICOMMAND	END OF INTERRUPT
0	1	1	SPECIFIC EOI COMMAND	
1	0	1	ROTATE ON NON-SPECIFIC EOI COMMAND	AUTOMATIC ROTATION
1	0	0	ROTATE IN AUTOMATIC EOI MODE (SET)	
0	0	0	ROTATE IN AUTOMATIC EOI MODE (CLEAR)	
1	1	1	*ROTATE ON SPECIFIC EOI COMMAND	SPECIFIC ROTATION
1	1	0	*SET PRIORITY COMMAND	
0	1	0	NO OPERATION	

*LO-L2 ARE USED

OCW3

A0	D7	D6	D5	D4	D3	D2	D1	D0
0	0	ESMM	SMM	0	1	P	RR	RIS

READ REGISTER COMMAND

0	1	0	1
0	0	1	1
NO ACTION		READ IR REG ON NEXT $\overline{RD}$ PULSE	READ IS REG ON NEXT $\overline{RD}$ PULSE

1=POLL COMMAND
0=NO POLL COMMAND

SPECIAL MASK MODE

0	1	0	1
0	0	1	1
NO ACTION		RESET SPECIAL MASK	SET SPECIAL MASK

Figure 8. Operation Command Word Format

FULLY NESTED MODE

This mode is entered after initialization unless another mode is programmed. The interrupt requests are ordered in priority form 0 through 7 (0 highest). When an interrupt is acknowledged the highest priority request is determined and its vector placed on the bus. Additionally, a bit of the Interrupt Service register (ISO-7) is set. This bit remains set until the microprocessor issues an End of Interrupt (EOI) command immediately before returning from the service routine, or if AEOI (Automatic End of Interrupt) bit is set, until the trailing edge of the last INTA. While the IS bit is set, all further interrupts of the same or lower priority are inhibited, while higher levels will generate an interrupt (which will be acknowledged only if the microprocessor internal Interrupt enable flip-flop has been re-enabled through software).

After the initialization sequence, IR0 has the highest priority and IR7 the lowest. Priorities can be changed, as will be explained, in the rotating priority mode.

END OF INTERRUPT (EOI)

The In Service (IS) bit can be reset either automatically following the trailing edge of the last in sequence $\overline{\text{INTA}}$ pulse (when AEOI bit in ICW1 is set) or by a command word that must be issued to the 8259A before returning from a service routine (EOI command). An EOI command must be issued twice if in the Cascade mode, once for the master and once for the corresponding slave.

There are two forms of EOI command: Specific and Non-Specific. When the 8259A is operated in modes which preserve the fully nested structure, it can determine which IS bit to reset on EOI. When a Non-Specific EOI command is issued the 8259A will automatically reset the highest IS bit of those that are set, since in the fully nested mode the highest IS level was necessarily the last level acknowledged and serviced. A non-specific EOI can be issued with OCW2 (EOI = 1, SL = 0, R = 0).

When a mode is used which may disturb the fully nested structure, the 8259A may no longer be able to determine the last level acknowledged. In this case a Specific End of Interrupt must be issued which includes as part of the command the IS level to be reset. A specific EOI can be issued with OCW2 (EOI = 1, SL = 1, R = 0, and LO-L2 is the binary level of the IS bit to be reset).

It should be noted that an IS bit that is masked by an IMR bit will not be cleared by a non-specific EOI if the 8259A is in the Special Mask Mode.

AUTOMATIC END OF INTERRUPT (AEOI) MODE

If AEOI = 1 in ICW4, then the 8259A will operate in AEOI mode continuously until reprogrammed by ICW4. In this mode the 8259A will automatically perform a non-specific EOI operation at the trailing edge of the last interrupt acknowledge pulse (third pulse in MCS-80/85, second in iAPX 86). Note that from a system standpoint, this mode should be used only when a nested multilevel interrupt structure is not required within a single 8259A.

The AEOI mode can only be used in a master 8259A and not a slave.

AUTOMATIC ROTATION (Equal Priority Devices)

In some applications there are a number of interrupting devices of equal priority. In this mode a device, after being serviced, receives the lowest priority, so a device requesting an interrupt will have to wait, in the worst case until each of 7 other devices are serviced at most *once*. For example, if the priority and "in service" status is:

Before Rotate (IR4 the highest priority requiring service)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	1	0	0	0	0

Lowest Priority Highest Priority

Priority Status	7	6	5	4	3	2	1	0

After Rotate (IR4 was serviced, all other priorities rotated correspondingly)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	0	0	0	0	0

Highest Priority Lowest Priority

Priority Status	2	1	0	7	6	5	4	3

There are two ways to accomplish Automatic Rotation using OCW2, the Rotation on Non-Specific EOI Command (R = 1, SL = 0, EOI = 1) and the Rotate in Automatic EOI Mode which is set by (R = 1, SL = 0, EOI = 0) and cleared by (R = 0, SL = 0, EOI = 0).

SPECIFIC ROTATION (Specific Priority)

The programmer can change priorities by programming the bottom priority and thus fixing all other priorities; i.e., if IR5 is programmed as the bottom priority device, then IR6 will have the highest one.

The Set Priority command is issued in OCW2 where: R = 1, SL = 1; LO–L2 is the binary priority level code of the bottom priority device.

Observe that in this mode internal status is updated by software control during OCW2. However, it is independent of the End of Interrupt (EOI) command (also executed by OCW2). Priority changes can be executed during an EOI command by using the Rotate on Specific EOI command in OCW2 (R = 1, SL = 1, EOI = 1 and LO–L2 = IR level to receive bottom priority).

INTERRUPT MASKS

Each Interrupt Request input can be masked individually by the Interrupt Mask Register (IMR) programmed through OCW1. Each bit in the IMR masks one interrupt channel if it is set (1). Bit 0 masks IR0, Bit 1 masks IR1 and so forth. Masking an IR channel does not affect the other channels operation.

SPECIAL MASK MODE

Some applications may require an interrupt service routine to dynamically alter the system priority structure during its execution under software control. For example, the routine may wish to inhibit lower priority requests for a portion of its execution but enable some of them for another portion.

The difficulty here is that if an Interrupt Request is acknowledged and an End of Interrupt command did not reset its IS bit (i.e., while executing a service routine), the 8259A would have inhibited all lower priority requests with no easy way for the routine to enable them

That is where the Special Mask Mode comes in. In the special Mask Mode, when a mask bit is set in OCW1, it inhibits further interrupts at that level *and enables* interrupts from *all other* levels (lower as well as higher) that are not masked.

Thus, any interrupts may be selectively enabled by loading the mask register.

The special Mask Mode is set by OCW3 where: SSMM = 1, SMM = 1, and cleared where SSMM = 1, SMM = 0.

POLL COMMAND

In this mode the INT output is not used or the microprocessor internal Interrupt Enable flip-flop is reset, disabling its interrupt input. Service to devices is achieved by software using a Poll command.

The Poll command is issued by setting P = "1" in OCW3. The 8259A treats the next $\overline{RD}$ pulse to the 8259A (i.e., $\overline{RD}$ = 0, $\overline{CS}$ = 0) as an interrupt acknowledge, sets the appropriate IS bit if there is a request, and reads the priority level. Interrupt is frozen from $\overline{WR}$ to $\overline{RD}$.

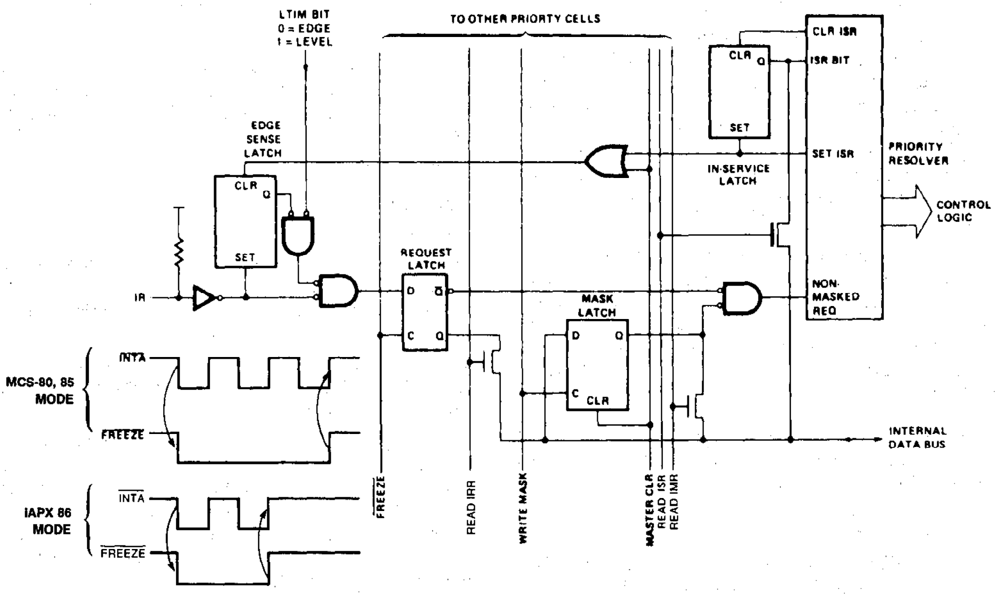
The word enabled onto the data bus during $\overline{RD}$ is:

D7	D6	D5	D4	D3	D2	D1	D0
I	—	—	—	—	W2	W1	W0

W0-W2: Binary code of the highest priority level requesting service.

I: Equal to a "1" if there is an interrupt.

This mode is useful if there is a routine command common to several levels so that the $\overline{INTA}$ sequence is not needed (saves ROM space). Another application is to use the poll mode to expand the number of priority levels to more than 64.



NOTES

1. MASTER CLEAR ACTIVE ONLY DURING ICW1
2. FREEZE/ IS ACTIVE DURING $\overline{INTA}$/ AND POLL SEQUENCES ONLY
3. TRUTH TABLE FOR D-LATCH

C	D	Q	OPERATION
1	Di	Di	FOLLOW
0	X	Qn-1	HOLD

Figure 9. Priority Cell—Simplified Logic Diagram

AFN-00221C

READING THE 8259A STATUS

The input status of several internal registers can be read to update the user information on the system. The following registers can be read via OCW3 (IRR and ISR or OCW1 [IMR]).

Interrupt Request Register (IRR): 8-bit register which contains the levels requesting an interrupt to be acknowledged. The highest request level is reset from the IRR when an interrupt is acknowledged. (Not affected by IMR.)

In-Service Register (ISR): 8-bit register which contains the priority levels that are being serviced. The ISR is updated when an End of Interrupt Command is issued.

Interrupt Mask Register: 8-bit register which contains the interrupt request lines which are masked.

The IRR can be read when, prior to the RD pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 0.)

The ISR can be read when, prior to the RD pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 1).

There is no need to write an OCW3 before every status read operation, as long as the status read corresponds with the previous one; i.e., the 8259A "remembers" whether the IRR or ISR has been previously selected by the OCW3. This is not true when poll is used.

After initialization the 8259A is set to IRR.

For reading the IMR, no OCW3 is needed. The output data bus will contain the IMR whenever $\overline{RD}$ is active and AO =1 (OCW1).

Polling overrides status read when P = 1, RR = 1 in OCW3.

EDGE AND LEVEL TRIGGERED MODES

This mode is programmed using bit 3 in ICW1.

If LTIM = '0', an interrupt request will be recognized by a low to high transition on an IR input. The IR input can remain high without generating another interrupt.

If LTIM = '1', an interrupt request will be recognized by a 'high' level on IR Input, and there is no need for an edge detection. The interrupt request must be removed before the EOI command is issued or the CPU interrupt is enabled to prevent a second interrupt from occurring.

The priority cell diagram shows a conceptual circuit of the level sensitive and edge sensitive input circuitry of the 8259A. Be sure to note that the request latch is a transparent D type latch.

In both the edge and level triggered modes the IR inputs must remain high until after the falling edge of the first INTA. If the IR input goes low before this time a DEFAULT IR7 will occur when the CPU acknowledges the interrupt. This can be a useful safeguard for detecting interrupts caused by spurious noise glitches on the IR inputs. To implement this feature the IR7 routine is used for "clean up" simply executing a return instruction, thus ignoring the interrupt. If IR7 is needed for other purposes a default IR7 can still be detected by reading the ISR. A normal IR7 interrupt will set the corresponding ISR bit, a default IR7 won't. If a default IR7 routine occurs during a normal IR7 routine, however, the ISR will remain set. In this case it is necessary to keep track of whether or not the IR7 routine was previously entered. If another IR7 occurs it is a default.

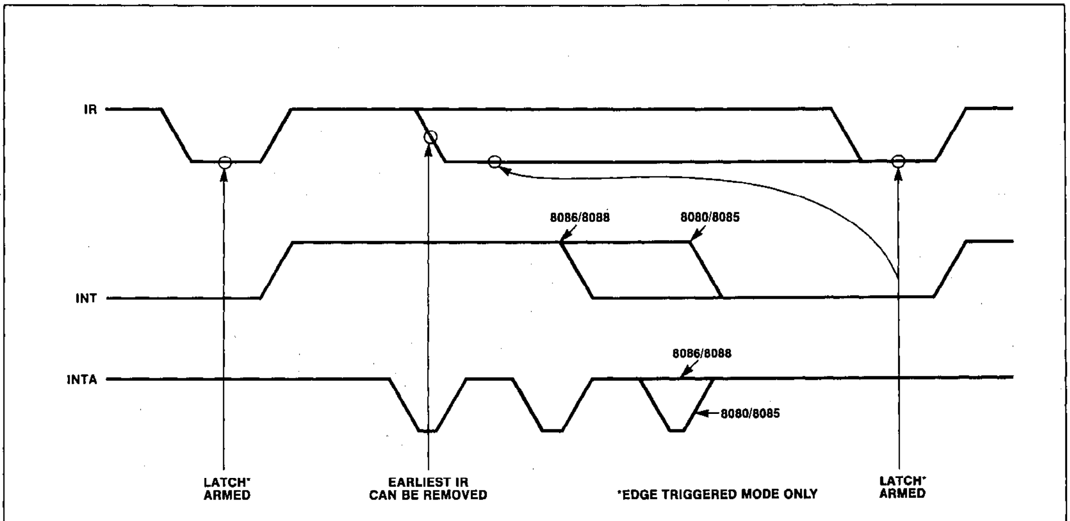


Figure 10. IR Triggering Timing Requirements

THE SPECIAL FULLY NESTED MODE

This mode will be used in the case of a big system where cascading is used, and the priority has to be conserved within each slave. In this case the fully nested mode will be programmed to the master (using ICW4). This mode is similar to the normal nested mode with the following exceptions:

a. When an interrupt request from a certain slave is in service this slave is not locked out from the master's priority logic and further interrupt requests from higher priority IR's within the slave will be recognized by the master and will initiate interrupts to the processor. (In the normal nested mode a slave is masked out when its request is in service and no higher requests from the same slave can be serviced.)

b. When exiting the Interrupt Service routine the software has to check whether the interrupt serviced was the only one from that slave. This is done by sending a non-specific End of Interrupt (EOI) command to the slave and then reading its In-Service register and checking for zero. If it is empty, a non-specific EOI can be sent to the master too. If not, no EOI should be sent.

BUFFERED MODE

When the 8259A is used in a large system where bus driving buffers are required on the data bus and the cascading mode is used, there exists the problem of enabling buffers.

The buffered mode will structure the 8259A to send an enable signal on $\overline{SP}/\overline{EN}$ to enable the buffers. In this mode, whenever the 8259A's data bus outputs are enabled, the $\overline{SP}/\overline{EN}$ output becomes active.

This modification forces the use of software programming to determine whether the 8259A is a master or a slave. Bit 3 in ICW4 programs the buffered mode, and bit 2 in ICW4 determines whether it is a master or a slave.

CASCADE MODE

The 8259A can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels.

The master controls the slaves through the 3 line cascade bus. The cascade bus acts like chip selects to the slaves during the $\overline{INTA}$ sequence.

In a cascade configuration, the slave interrupt outputs are connected to the master interrupt request inputs. When a slave request line is activated and afterwards acknowledged, the master will enable the corresponding slave to release the device routine address during bytes 2 and 3 of INTA. (Byte 2 only for 8086/8088).

The cascade bus lines are normally low and will contain the slave address code from the trailing edge of the first INTA pulse to the trailing edge of the third pulse. Each 8259A in the system must follow a separate initialization sequence and can be programmed to work in a different mode. An EOI command must be issued twice: once for the master and once for the corresponding slave. An address decoder is required to activate the Chip Select (CS) input of each 8259A.

The cascade lines of the Master 8259A are activated only for slave inputs, non slave inputs leave the cascade line inactive (low).

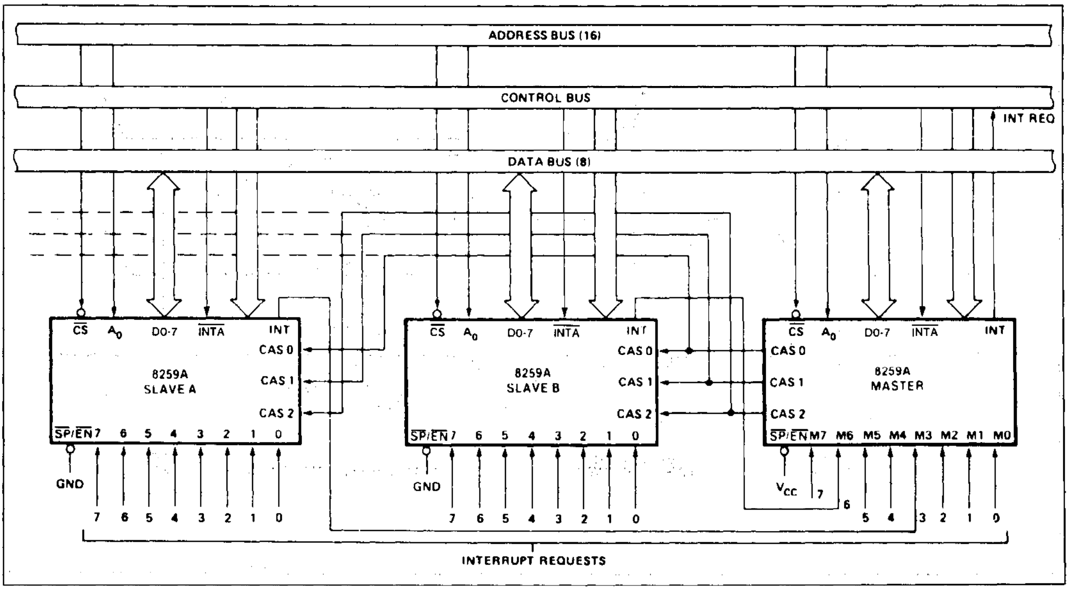


Figure 11. Cascading the 8259A

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ...... −40°C to 85°C
Storage Temperature .... −65°C to +150°C
Voltage on Any Pin
with Respect to Ground .... −0.5V to +7V
Power Dissipation .... 1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.*

D.C. CHARACTERISTICS [T_A = 0°C to 70°C, V_{CC} = 5V ±10% (8259-A), V_{CC} = 5V ±10% (8259A)]

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} +0.5V	V	
V_{OL}	Output High Voltage		0.45	V	I_{OL} = 2.2mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −400μA
$V_{OH(INT)}$	Interrupt Output High Voltage	3.5		V	I_{OH} = −100μA
		2.4		V	I_{OH} = −400μA
I_{LI}	Input Load Current		10	μA	0V ≤V_{IN} ≤V_{CC}
I_{LOL}	Output Leakage Current		−10	μA	0.45V ≤V_{OUT} ≤V_{CC}
I_{CC}	V_{CC} Supply Current		85	mA	
I_{LIR}	IR Input Load Current		−300	μA	V_{IN} = 0
			10	μA	V_{IN} = V_{CC}

CAPACITANCE (T_A = 25°C; V_{CC} = GND = 0V)

Symbol	Parameter	Min.	Typ.	Max.	Unit	Test Conditions
C_{IN}	Input Capacitance			10	pF	fc = 1 MHz
$C_{I/O}$	I/O Capacitance			20	pF	Unmeasured pins returned to V_{SS}

AC CHARACTERISTICS [T_A = 0°C to 70°C, V_{CC} = 5V ±5% (8259A-8), V_{CC} = 5V ±10% (8259A)]

TIMING REQUIREMENTS

Symbol	Parameter	8259A-8		8259A		8259A-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TAHRL	A0/$\overline{CS}$ Setup to $\overline{RD}$/$\overline{INTA}$↓	50		0		0		ns	
TRHAX	AO/$\overline{CS}$ Hold after $\overline{RD}$/$\overline{INTA}$↑	5		0		0		ns	
TRLRH	$\overline{RD}$ Pulse Width	420		235		160		ns	
TAHWL	AO/$\overline{CS}$ Setup to $\overline{WR}$↓	50		0		0		ns	
TWHAX	AO/$\overline{CS}$ Hold after $\overline{WR}$↑	20		0		0		ns	
TWLWH	$\overline{WR}$ Pulse Width	400		290		190		ns	
TDVWH	Data Setup to $\overline{WR}$↑	300		240		160		ns	
TWHDX	Data Hold after $\overline{WR}$↑	40		0		0		ns	
TJLJH	Interrupt Request Width (Low)	100		100		100		ns	See Note 1
TCVIAL	Cascade Setup to Second or Third $\overline{INTA}$↓ (Slave Only)	55		55		40		ns	
TRHRL	End of $\overline{RD}$ to Next Command	160		160		160		ns	
TWHRL	End of $\overline{WR}$ to Next Command	190		190		190		ns	

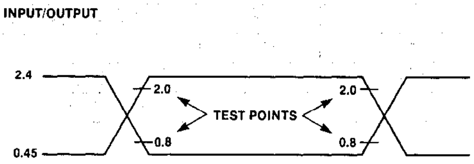
Note: This is the low time required to clear the input latch in the edge triggered mode.

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

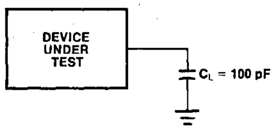
Symbol	Parameter	8259A-8		8259A		8259A-2		Units	Test Conditions
		Min.	Max.	Min.	Max.	Min.	Max.		
TRLDV	Data Valid from $\overline{\text{RD}}/\overline{\text{INTA}}\downarrow$		300		200		120	ns	C of Data Bus = 100 pF
TRHDZ	Data Float after $\overline{\text{RD}}/\overline{\text{INTA}}\uparrow$	10	200		100		85	ns	C of Data Bus Max text C = 100 pF Min. test C = 15 pF
TJHIH	Interrupt Output Delay		400		350		300	ns	
TIALCV	Cascade Valid from First $\overline{\text{INTA}}\downarrow$ (Master Only)		565		565		360	ns	C_{INT} = 100 pF
TRLEL	Enable Active from $\overline{\text{RD}}\downarrow$ or $\overline{\text{INTA}}\downarrow$		160		125		100	ns	$C_{CASCADE}$ = 100 pF
TRHEH	Enable Inactive from $\overline{\text{RD}}\uparrow$ or $\overline{\text{INTA}}\uparrow$		325		150		150	ns	
TAHDV	Data Valid from Stable Address		350		200		200	ns	
TCVDV	Cascade Valid to Valid Data		300		300		200	ns	

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." TIMING MEASUREMENTS ARE MADE AT 2.0V FOR A LOGIC "1" AND 0.8V FOR A LOGIC "0."

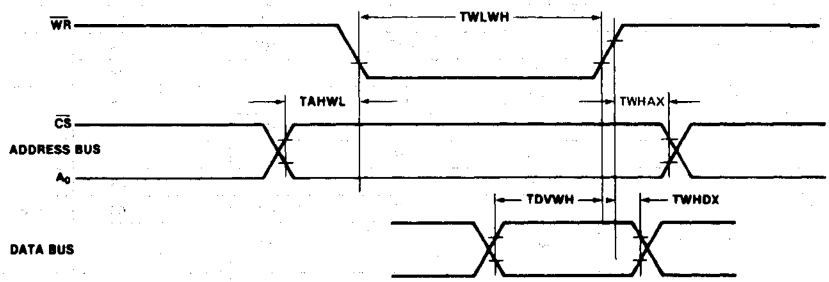
A.C. TESTING LOAD CIRCUIT



C_L = 100 pF
C_L INCLUDES JIG CAPACITANCE

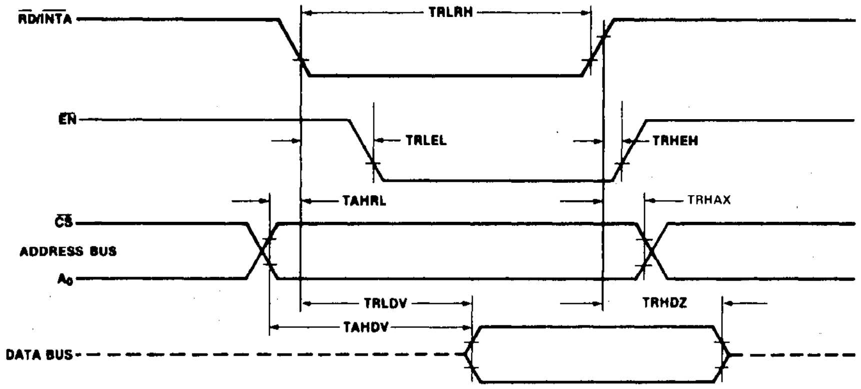
WAVEFORMS

WRITE

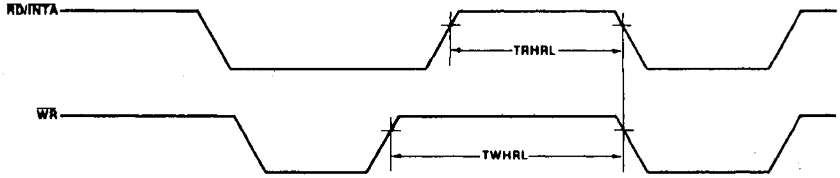


WAVEFORMS (Continued)

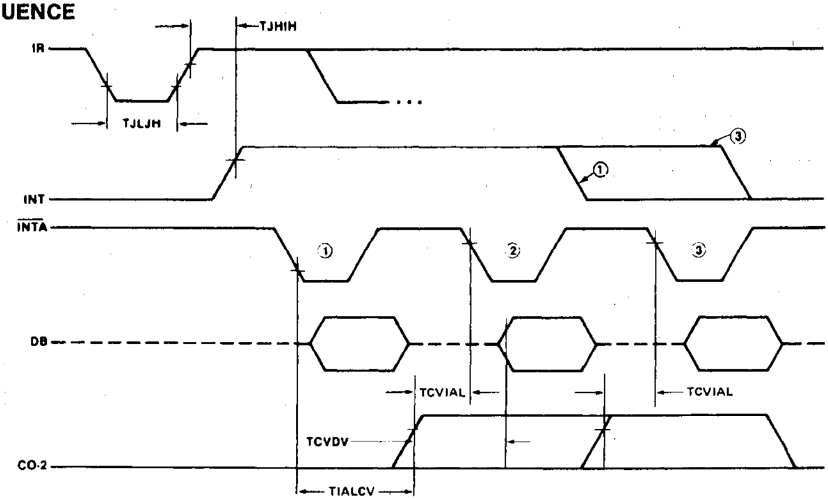
READ/INTA



OTHER TIMING



INTA SEQUENCE



NOTES: Interrupt output must remain HIGH at least until leading edge of first INTA.
1. Cycle 1 in iAPX 86, iAPX 88 systems, the Data Bus is not active.

8282/8283
OCTAL LATCH

- Address Latch for iAPX 86, 88, MCS-80®, MCS-85®, MCS-48® Families
- High Output Drive Capability for Driving System Data Bus
- Fully Parallel 8-Bit Data Register and Buffer
- Transparent during Active Strobe
- 3-State Outputs
- 20-Pin Package with 0.3" Center
- No Output Low Noise when Entering or Leaving High Impedance State

The 8282 and 8283 are 8-bit bipolar latches with 3-state output buffers. They can be used to implement latches, buffers or multiplexers. The 8283 inverts the input data at its outputs while the 8282 does not. Thus, all of the principal peripheral and input/output functions of a microcomputer system can be implemented with these devices.

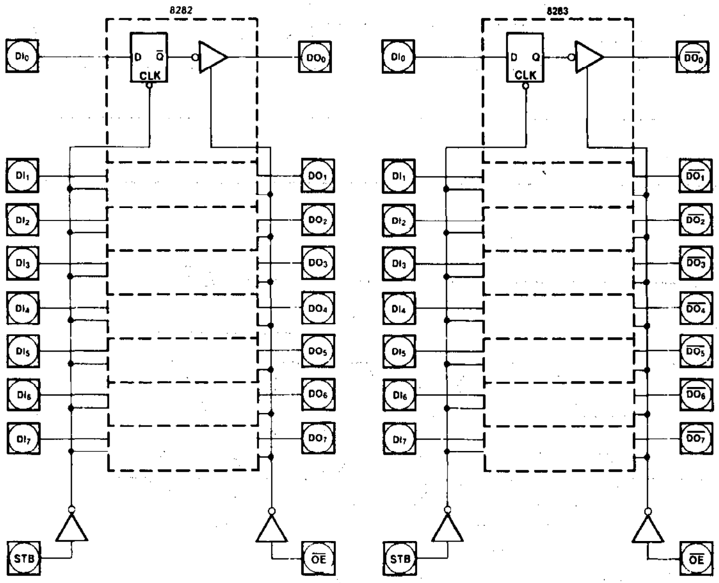


Figure 1. Logic Diagrams

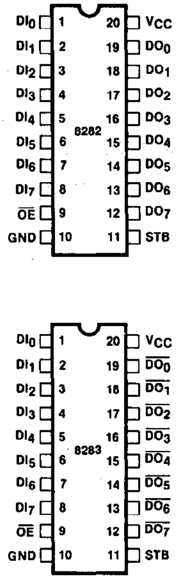


Figure 2. Pin Configurations

Table 1. Pin Description

Pin	Description
STB	STROBE (Input). STB is an input control pulse used to strobe data at the data input pins (A_0-A_7) into the data latches. This signal is active HIGH to admit input data. The data is latched at the HIGH to LOW transition of STB.
$\overline{OE}$	OUTPUT ENABLE (Input). $\overline{OE}$ is an input control signal which when active LOW enables the contents of the data latches onto the data output pin (B_0-B_7). OE being inactive HIGH forces the output buffers to their high impedance state.
DI_0-DI_7	DATA INPUT PINS (Input). Data presented at these pins satisfying setup time requirements when STB is strobed and latched into the data input latches.
DO_0-DO_7 (8282) $\overline{DO_0}$-$\overline{DO_7}$ (8283)	DATA OUTPUT PINS (Output). When $\overline{OE}$ is true, the data in the data latches is presented as inverted (8283) or non-inverted (8282) data onto the data output pins.

FUNCTIONAL DESCRIPTION

The 8282 and 8283 octal latches are 8-bit latches with 3-state output buffers. Data having satisfied the setup time requirements is latched into the data latches by strobing the STB line HIGH to LOW. Holding the STB line in its active HIGH state makes the latches appear transparent. Data is presented to the data output pins by activating the $\overline{OE}$ input line. When $\overline{OE}$ is inactive HIGH the output buffers are in their high impedance state. Enabling or disabling the output buffers will not cause negative-going transients to appear on the data output bus.

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias ....0°C to 70°C
Storage Temperature .... −65°C to +150°C
All Output and Supply Voltages .... −0.5V to +7V
All Input Voltages .... −1.0V to +5.5V
Power Dissipation .... 1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (V_{CC} = 5V ±10%, T_A = 0°C to 70°C)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_C	Input Clamp Voltage		−1	V	I_C = −5 mA
I_{CC}	Power Supply Current		160	mA	
I_F	Forward Input Current		−0.2	mA	V_F = 0.45V
I_R	Reverse Input Current		50	μA	V_R = 5.25V
V_{OL}	Output Low Voltage		.45	V	I_{OL} = 32 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −5 mA
I_{OFF}	Output Off Current		± 50	μA	V_{OFF} = 0.45 to 5.25V
V_{IL}	Input Low Voltage		0.8	V	V_{CC} = 5.0V See Note 1
V_{IH}	Input High Voltage	2.0		V	V_{CC} = 5.0V See Note 1
C_{IN}	Input Capacitance		12	pF	F = 1 MHz V_{BIAS} = 2.5V, V_{CC} = 5V T_A = 25°C

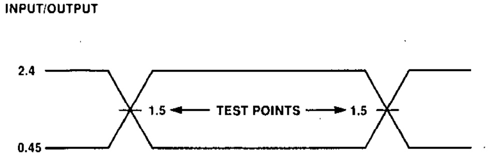
NOTE:
1. Output Loading I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF.

A.C. CHARACTERISTICS (V_{CC} = 5V ±10%, T_A = 0°C to 70°C Loading: Outputs — I_{OL} = 32 mA, I_{OH} = −5 mA, C_L = 300 pF)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TIVOV	Input to Output Delay —Inverting —Non-Inverting	 5 5	 22 30	 ns ns	(See Note 1)
TSHOV	STB to Output Delay —Inverting —Non-Inverting	 10 10	 40 45	 ns ns	
TEHOZ	Output Disable Time	5	18	ns	
TELOV	Output Enable Time	10	30	ns	
TIVSL	Input to STB Setup Time	0		ns	
TSLIX	Input to STB Hold Time	25		ns	
TSHSL	STB High Time	15		ns	
TILIH, TOLOH	Input, Output Rise Time		20	ns	From 0.8V to 2.0V
TIHIL, TOHOL	Input, Output Fall Time		12	ns	From 2.0V to 0.8V

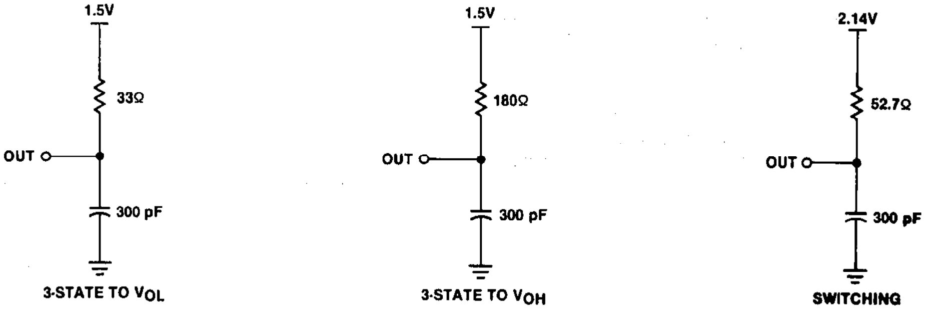
NOTE:
1. See waveforms and test load circuit on following page.

A.C. TESTING INPUT, OUTPUT WAVEFORM

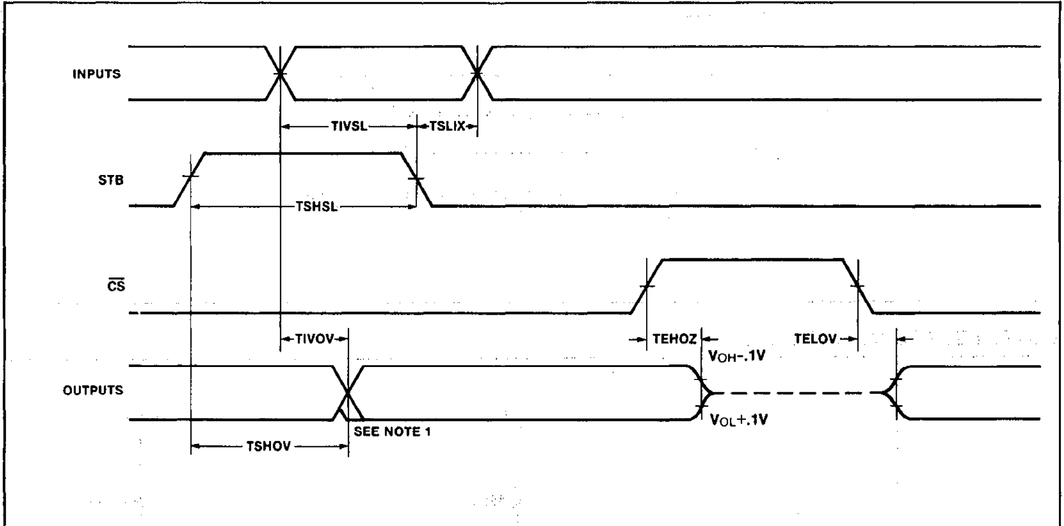


A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

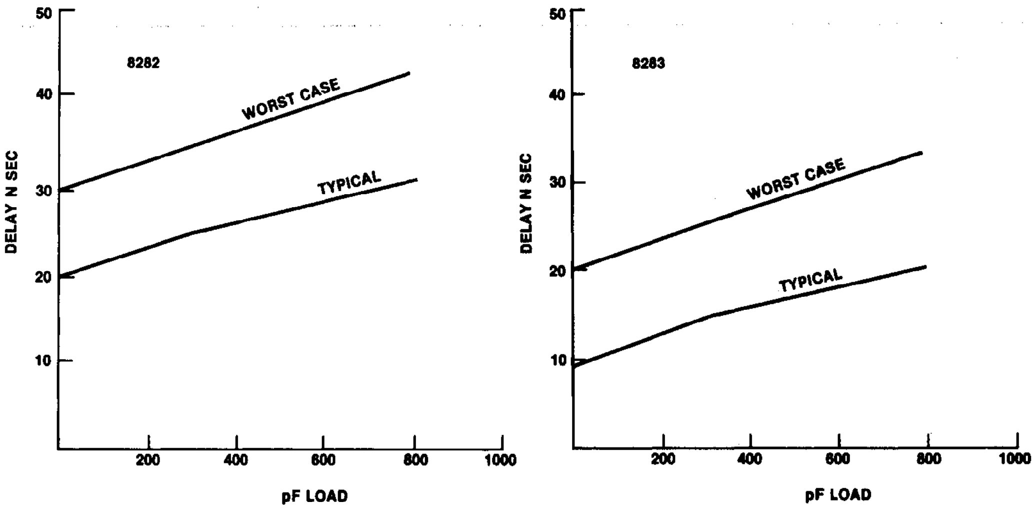
OUTPUT TEST LOAD CIRCUITS



WAVEFORMS



NOTE: 1. 8283 ONLY — OUTPUT MAY BE MOMENTARILY INVALID FOLLOWING THE HIGH GOING STB TRANSITION.
2. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.



Output Delay vs. Capacitance

I8282/8283
OCTAL LATCH
INDUSTRIAL

- **Fully Parallel 8-Bit Data Register and Buffer**
- **Transparent during Active Strobe**
- **Address Latch for iAPX 86, 88, MCS-80®, MCS-85®, MCS-48® Families**
- **High Output Drive Capability for Driving System Data Bus**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**
- **Industrial Temperature Range: −40° to +85°C**

The I8282 and I8283 are 8-bit bipolar latches with 3-state output buffers. They can be used to implement latches, buffers, or multiplexers. The I8283 inverts the input data at its outputs while the I8282 does not. Thus, all of the principal peripheral and input/output functions of a microcomputer system can be implemented with these devices.

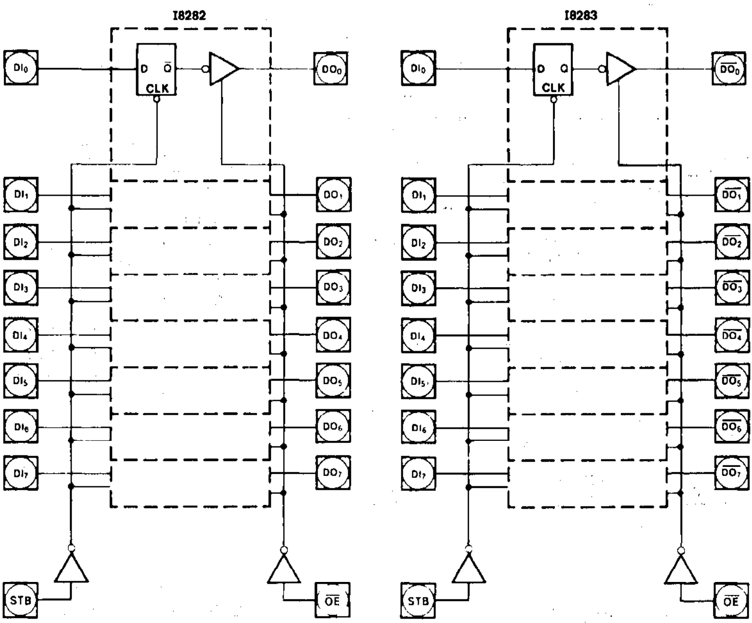


Figure 1. Logic Diagrams

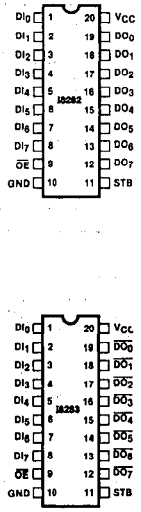


Figure 2. Pin Configurations

8284A
CLOCK GENERATOR AND DRIVER FOR iAPX 86, 88 PROCESSORS

- Generates the System clock for the iAPX 86, 88 Processors
- Uses a Crystal or a TTL Signal for Frequency Source
- Provides Local READY and Multibus™ READY Synchronization
- 18-Pin Package
- Single +5V Power Supply
- Generates System Reset Output from Schmitt Trigger Input
- Capable of Clock Synchronization with Other 8284As

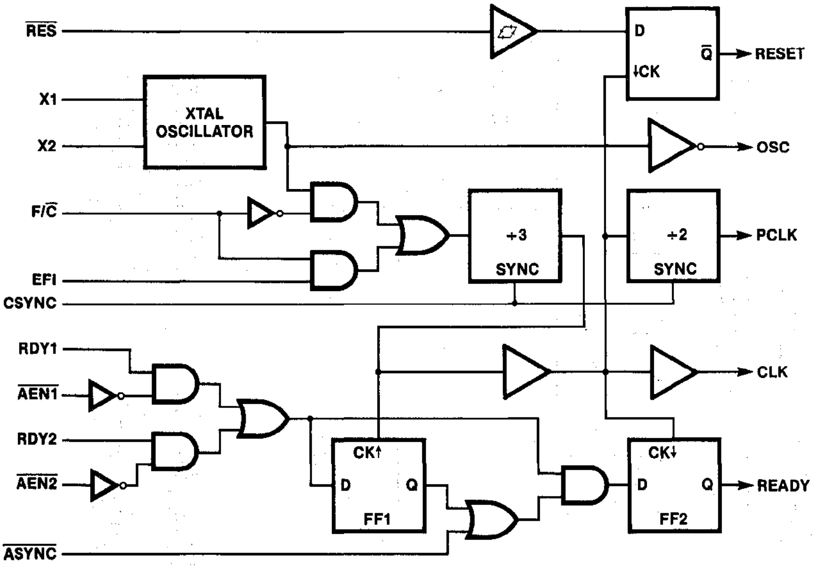


Figure 1. 8284A Block Diagram

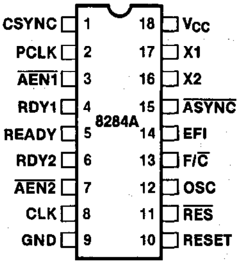


Figure 2. 8284A Pin Configuration

Table 1. Pin Description

Symbol	Type	Name and Function
$\overline{\text{AEN1}}$, $\overline{\text{AEN2}}$	I	**Address Enable:** $\overline{\text{AEN}}$ is an active LOW signal. AEN serves to qualify its respective Bus Ready Signal (RDY1 or RDY2). $\overline{\text{AEN1}}$ validates RDY1 while $\overline{\text{AEN2}}$ validates RDY2. Two AEN signal inputs are useful in system configurations which permit the processor to access two Multi-Master System Busses. In non Multi-Master configurations the $\overline{\text{AEN}}$ signal inputs are tied true (LOW).
RDY1, RDY2	I	**Bus Ready:** (Transfer Complete). RDY is an active HIGH signal which is an indication from a device located on the system data bus that data has been received, or is available. RDY1 is qualified by $\overline{\text{AEN1}}$ while RDY2 is qualified by $\overline{\text{AEN2}}$.
$\overline{\text{ASYNC}}$	I	**Ready Synchronization Select:** $\overline{\text{ASYNC}}$ is an input which defines the synchronization mode of the READY logic. When $\overline{\text{ASYNC}}$ is low, two stages of READY synchronization are provided. When $\overline{\text{ASYNC}}$ is left open or HIGH a single stage of READY synchronization is provided.
READY	O	**Ready:** READY is an active HIGH signal which is the synchronized RDY signal input. READY is cleared after the guaranteed hold time to the processor has been met.
X1, X2	I	**Crystal In:** X1 and X2 are the pins to which a crystal is attached. The crystal frequency is 3 times the desired processor clock frequency.
F/$\overline{\text{C}}$	I	**Frequency/Crystal Select:** F/$\overline{\text{C}}$ is a strapping option. When strapped LOW, F/$\overline{\text{C}}$ permits the processor's clock to be generated by the crystal. When F/$\overline{\text{C}}$ is strapped HIGH, CLK is generated from the EFI input.
EFI	I	**External Frequency:** When F/$\overline{\text{C}}$ is strapped HIGH, CLK is generated from the input frequency appearing on this pin. The input signal is a square wave 3 times the frequency of the desired CLK output.

Symbol	Type	Name and Function
CLK	O	**Processor Clock:** CLK is the clock output used by the processor and all devices which directly connect to the processor's local bus (i.e., the bipolar support chips and other MOS devices). CLK has an output frequency which is ⅓ of the crystal or EFI input frequency and a ⅓ duty cycle. An output HIGH of 4.5 volts (V_{CC}= 5V) is provided on this pin to drive MOS devices.
PCLK	O	**Peripheral Clock:** PCLK is a TTL level peripheral clock signal whose output frequency is ½ that of CLK and has a 50% duty cycle.
OSC	O	**Oscillator Output:** OSC is the TTL level output of the internal oscillator circuitry. Its frequency is equal to that of the crystal.
$\overline{\text{RES}}$	I	**Reset In:** $\overline{\text{RES}}$ is an active LOW signal which is used to generate RESET. The 8284A provides a Schmitt trigger input so that an RC connection can be used to establish the power-up reset of proper duration.
RESET	O	**Reset:** RESET is an active HIGH signal which is used to reset the 8086 family processors. Its timing characteristics are determined by $\overline{\text{RES}}$.
CSYNC	I	**Clock Synchronization:** CSYNC is an active HIGH signal which allows multiple 8284As to be synchronized to provide clocks that are in phase. When CSYNC is HIGH the internal counters are reset. When CSYNC goes LOW the internal counters are allowed to resume counting. CSYNC needs to be externally synchronized to EFI. When using the internal oscillator CSYNC should be hardwired to ground.
GND		**Ground.**
V_{CC}		**Power:** +5V supply.

FUNCTIONAL DESCRIPTION

General

The 8284A is a single chip clock generator/driver for the iAPX 86, 88 processors. The chip contains a crystal-controlled oscillator, a divide-by-three counter, complete MULTIBUS™ "Ready" synchronization and reset logic. Refer to Figure 1 for Block Diagram and Figure 2 for Pin Configuration.

Oscillator

The oscillator circuit of the 8284A is designed primarily for use with an external series resonant, fundamental mode, crystal from which the basic operating frequency is derived.

The crystal frequency should be selected at three times the required CPU clock. X1 and X2 are the two crystal input crystal connections. For the most stable operation of the oscillator (OSC) output circuit, two series resistors (R_1 = R_2 = 510 Ω) as shown in the waveform figures are recommended. The output of the oscillator is buffered and brought out on OSC so that other system timing signals can be derived from this stable, crystal-controlled source.

For systems which have a V_{CC} ramp time ⩾ 1V/ms and/or have inherent board capacitance between X1 or X2, exceeding 10pF (not including 8284A pin capacitance), the configuration in Figures 4 and 6 is recommended. This circuit provides optimum stability for the oscillator in such extreme conditions. It is advisable to limit stray capacitances to less than 10pF on X1 and X2 to minimize deviation from operating at the fundamental frequency.

Clock Generator

The clock generator consists of a synchronous divide-by-three counter with a special clear input that inhibits the counting. This clear input (CSYNC) allows the output clock to be synchronized with an external event (such as another 8284A clock). It is necessary to synchronize the CSYNC input to the EFI clock external to the 8284A. This is accomplished with two Schottky flip-flops. The counter output is a 33% duty cycle clock at one-third the input frequency.

The F/$\overline{C}$ input is a strapping pin that selects either the crystal oscillator or the EFI input as the clock for the ÷3 counter. If the EFI input is selected as the clock source, the oscillator section can be used independently for another clock source. Output is taken from OSC.

Clock Outputs

The CLK output is a 33% duty cycle MOS clock driver designed to drive the iAPX 86, 88 processors directly. PCLK is a TTL level peripheral clock signal whose output frequency is ½ that of CLK. PCLK has a 50% duty cycle.

Reset Logic

The reset logic provides a Schmitt trigger input ($\overline{RES}$) and a synchronizing flip-flop to generate the reset timing. The reset signal is synchronized to the falling edge of CLK. A simple RC network can be used to provide power-on reset by utilizing this function of the 8284A.

READY Synchronization

Two READY inputs (RDY1, RDY2) are provided to accommodate two Multi-Master system busses. Each input has a qualifier ($\overline{AEN1}$ and $\overline{AEN2}$, respectively). The $\overline{AEN}$ signals validate their respective RDY signals. If a Multi-Master system is not being used the $\overline{AEN}$ pin should be tied LOW.

Synchronization is required for all asynchronous active-going edges of either RDY input to guarantee that the RDY setup and hold times are met. Inactive-going edges of RDY in normally ready systems do not require synchronization but must satisfy RDY setup and hold as a matter of proper system design.

The $\overline{ASYNC}$ input defines two modes of READY synchronization operation.

When $\overline{ASYNC}$ is LOW, two stages of synchronization are provided for active READY input signals. Positive-going asynchronous READY inputs will first be synchronized to flip-flop one at the rising edge of CLK and then synchronized to flip-flop two at the next falling edge of CLK, after which time the READY output will go active (HIGH). Negative-going asynchronous READY inputs will be synchronized directly to flip-flop two at the falling edge of CLK, after which time the READY output will go inactive. This mode of operation is intended for use by asynchronous (normally not ready) devices in the system which cannot be guaranteed by design to meet the required RDY setup timing, T_{R1VCL}, on each bus cycle.

When $\overline{ASYNC}$ is high or left open, the first READY flip-flop is bypassed in the READY synchronization logic. READY inputs are synchronized by flip-flop two on the falling edge of CLK before they are presented to the processor. This mode is available for synchronous devices that can be guaranteed to meet the required RDY setup time.

$\overline{ASYNC}$ can be changed on every bus cycle to select the appropriate mode of synchronization for each device in the system.

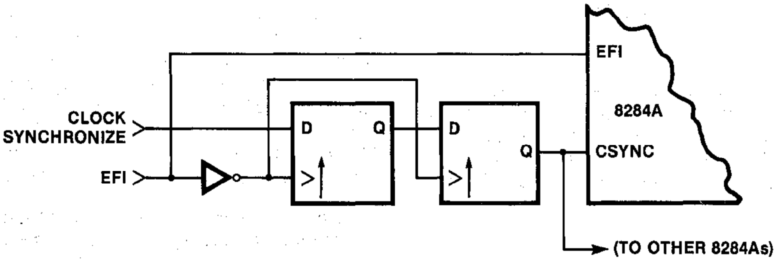


Figure 3. CSYNC Synchronization

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias	0°C to 70°C
Storage Temperature	−65°C to +150°C
All Output and Supply Voltages	−0.5V to +7V
All Input Voltages	−1.0V to +5.5V
Power Dissipation	1 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
I_F	Forward Input Current ($\overline{\text{ASYNC}}$) Other Inputs		−1.3 −0.5	mA mA	$V_F = 0.45V$ $V_F = 0.45V$
I_R	Reverse Input Current ($\overline{\text{ASYNC}}$) Other Inputs		50 50	μA μA	$V_R = V_{CC}$ $V_R = 5.25V$
V_C	Input Forward Clamp Voltage		−1.0	V	$I_C = -5mA$
I_{CC}	Power Supply Current		162	mA	
V_{IL}	Input LOW Voltage		0.8	V	
V_{IH}	Input HIGH Voltage	2.0		V	
V_{IHR}	Reset Input HIGH Voltage	2.6		V	
V_{OL}	Output LOW Voltage		0.45	V	5mA
V_{OH}	Output HIGH Voltage CLK Other Outputs	4 2.4		V V	−1mA −1mA
$V_{IHR} - V_{ILR}$	$\overline{\text{RES}}$ Input Hysteresis	0.25		V	

A.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = 5V ± 10%)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
t_{EHEL}	External Frequency HIGH Time	13		ns	90%-90% V_{IN}
t_{ELEH}	External Frequency LOW Time	13		ns	10%-10% V_{IN}
t_{ELEL}	EFI Period	$t_{EHEL} + t_{ELEH} + \delta$		ns	(Note 1)
	XTAL Frequency	12	30	MHz	
t_{R1VCL}	RDY1, RDY2 Active Setup to CLK	35		ns	$\overline{\text{ASYNC}}$ = HIGH
t_{R1VCH}	RDY1, RDY2 Active Setup to CLK	35		ns	$\overline{\text{ASYNC}}$ = LOW
t_{R1VCL}	RDY1, RDY2 Inactive Setup to CLK	35		ns	
t_{CLR1X}	RDY1, RDY2 Hold to CLK	0		ns	
t_{AYVCL}	$\overline{\text{ASYNC}}$ Setup to CLK	50		ns	
t_{CLAYX}	$\overline{\text{ASYNC}}$ Hold to CLK	0		ns	
t_{A1VR1V}	$\overline{\text{AEN1}}$, $\overline{\text{AEN2}}$ Setup to RDY1, RDY2	15		ns	
t_{CLA1X}	$\overline{\text{AEN1}}$, $\overline{\text{AEN2}}$ Hold to CLK	0		ns	
t_{YHEH}	CSYNC Setup to EFI	20		ns	
t_{EHYL}	CSYNC Hold to EFI	20		ns	
t_{YHYL}	CSYNC Width	$2 \cdot t_{ELEL}$		ns	
t_{I1HCL}	$\overline{\text{RES}}$ Setup to CLK	65		ns	(Note 2)
t_{CLI1H}	$\overline{\text{RES}}$ Hold to CLK	20		ns	(Note 2)
t_{ILIH}	Input Rise Time		20	ns	From 0.8V to 2.0V
t_{IHIL}	Input Fall Time		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

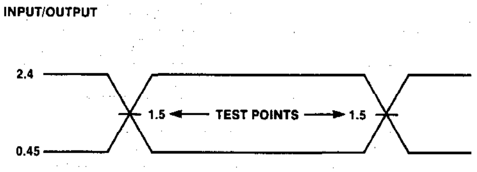
TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
t_{CLCL}	CLK Cycle Period	100		ns	
t_{CHCL}	CLK HIGH Time	(⅓ t_{CLCL})+2 for CLK Freq. ≤ 8 MHz (⅓ t_{CLCL})+6 for CLK Freq.=10 MHz		ns	Fig. 7 & Fig. 8
t_{CLCH}	CLK LOW Time	(⅔ t_{CLCL})−15 for CLK Freq.≤8 MHz (⅔ t_{CLCL})−14 for CLK Freq.=10 MHz		ns	Fig. 7 & Fig. 8
t_{CH1CH2} t_{CL2CL1}	CLK Rise or Fall Time		10	ns	1.0V to 3.5V
t_{PHPL}	PCLK HIGH Time	t_{CLCL}−20		ns	
t_{PLPH}	PCLK LOW Time	t_{CLCL}−20		ns	
t_{RYLCL}	Ready Inactive to CLK (See Note 4)	−8		ns	Fig. 9 & Fig. 10
t_{RYHCH}	Ready Active to CLK (See Note 3)	(⅔ t_{CLCL})−15 for CLK Freq.≤8 MHz (⅔ t_{CLCL})−14 for CLK Freq.=10 MHz		ns	Fig. 9 & Fig. 10
t_{CLIL}	CLK to Reset Delay		40	ns	
t_{CLPH}	CLK to PCLK HIGH DELAY		22	ns	
t_{CLPL}	CLK to PCLK LOW Delay		22	ns	
t_{OLCH}	OSC to CLK HIGH Delay	−5	22	ns	
t_{OLCL}	OSC to CLK LOW Delay	2	35	ns	
t_{OLOH}	Output Rise Time (except CLK)		20	ns	From 0.8V to 2.0V
t_{OHOL}	Output Fall Time (except CLK)		12	ns	From 2.0V to 0.8V

NOTES:

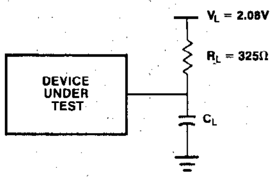
1. δ = EFI rise (5 ns max) + EFI fall (5 ns max).
2. Setup and hold necessary only to guarantee recognition at next clock.
3. Applies only to T3 and TW states.
4. Applies only to T2 states.

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

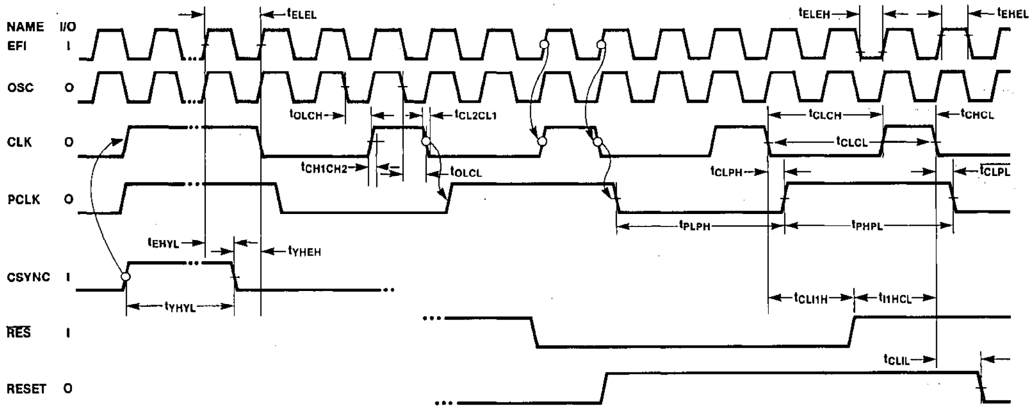
A.C. TESTING LOAD CIRCUIT



C_L = 100pF FOR CLK
C_L = 30pF FOR READY

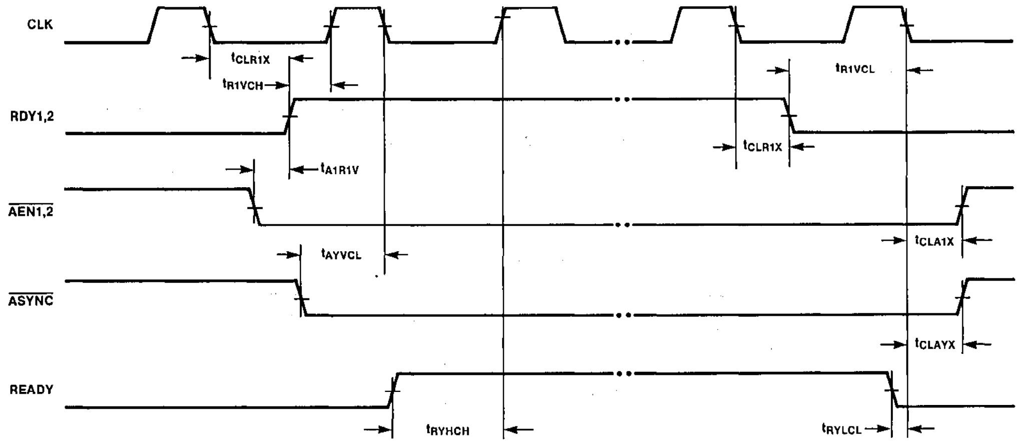
WAVEFORMS

CLOCKS AND RESET SIGNALS



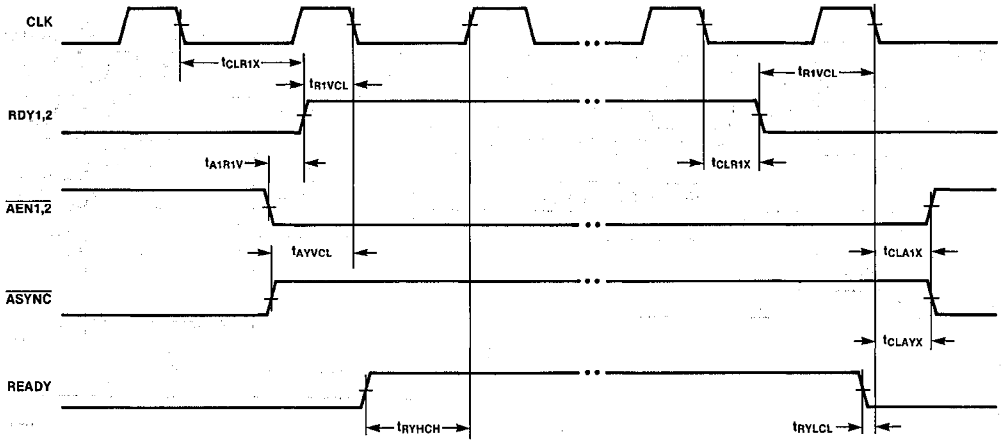
NOTE: ALL TIMING MEASUREMENTS ARE MADE AT 1.5 VOLTS, UNLESS OTHERWISE NOTED.

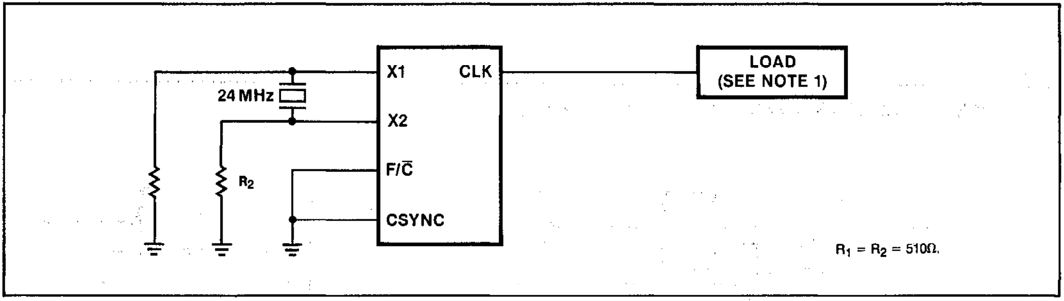
READY SIGNALS (FOR ASYNCHRONOUS DEVICES)



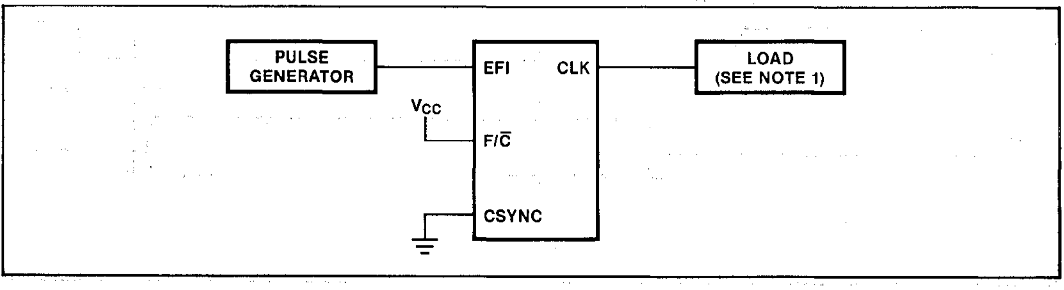
WAVEFORMS (Continued)

READY SIGNALS (FOR SYNCHRONOUS DEVICES)

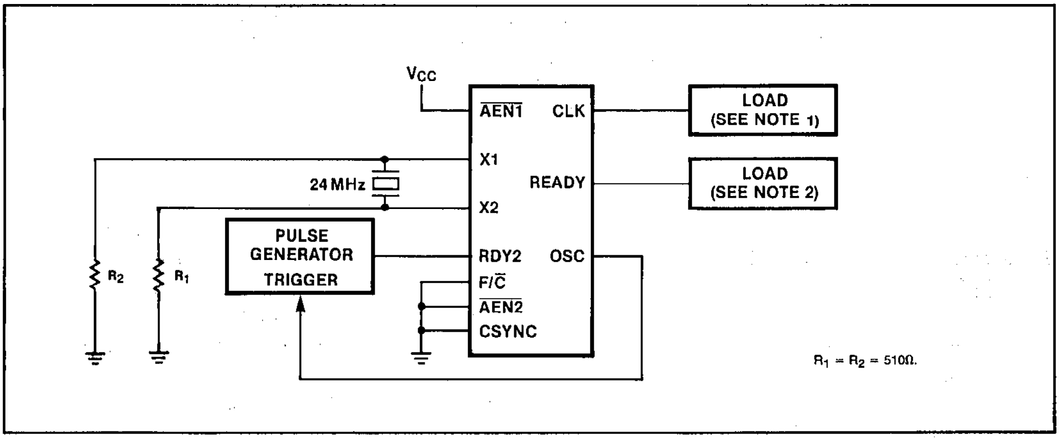




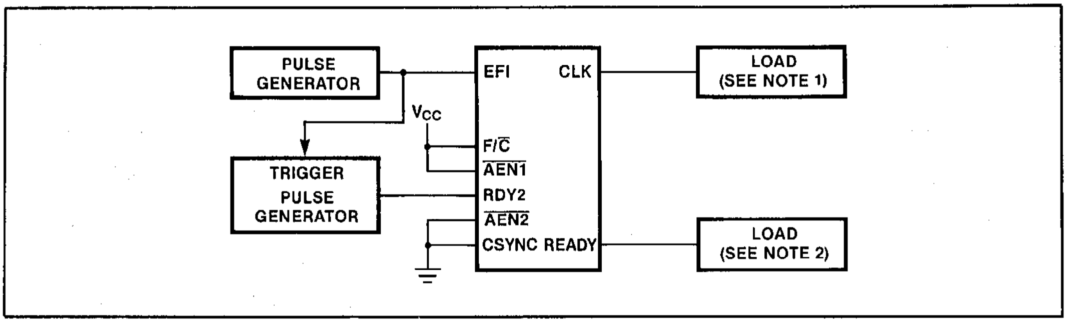
Clock High and Low Time (Using X1, X2)



Clock High and Low Time (Using EFI)



Ready to Clock (Using X1, X2)



Ready to Clock (Using EFI)

NOTES:
1. C_L = 100 pF
2. C_L = 30 pF

M8284
CLOCK GENERATOR AND DRIVER FOR MILITARY iAPX 86

MILITARY

- **Military Temperature Range: −55°C to +125°C**
- **Generates the System Clock for the M8086**
- **Uses a Crystal or TTL Signal for Frequency Source**
- **Single +5V Power Supply**
- **18-Pin Package**
- **Generates System Reset Output from Schmitt Trigger Input**
- **Provides Local Ready and MULTIBUS™ Ready Synchronization**
- **Capable of Clock Synchronization with other M8284's**

The M8284 is a bipolar clock generator/driver designed to provide clock signals for the Military iAPX 86 and peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

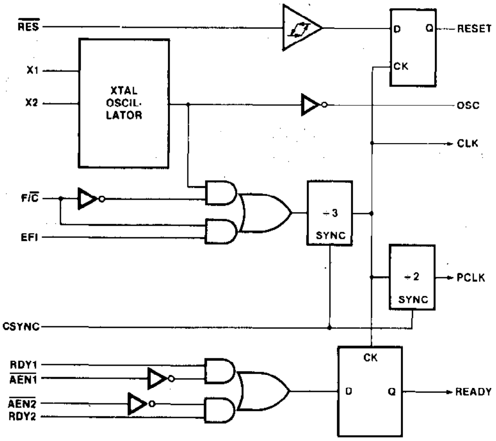


Figure 1. Block Diagram

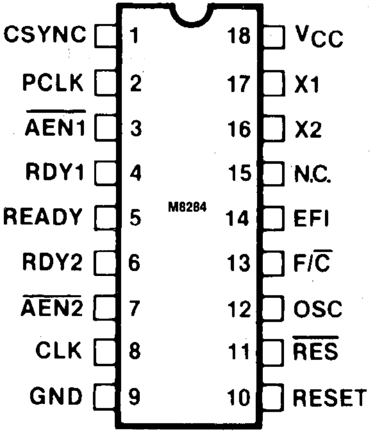


Figure 2. Pin Configuration

X1 X2	CONNECTIONS FOR CRYSTAL	$\overline{RES}$	RESET INPUT
F/$\overline{C}$	CLOCK SOURCE SELECT	RESET	SYNCHRONIZED RESET OUTPUT
EFI	EXTERNAL CLOCK INPUT	OSC	OSCILLATOR OUTPUT
CSYNC	CLOCK SYNCHRONIZATION INPUT	CLK	MOS CLOCK ID8086
RDY1 RDY2	READY SIGNAL FROM TWO MULTIBUS™ SYSTEMS	PCLK	TTL CLOCK FOR PERIPHERALS
		READY	SYNCHRONIZED READY OUTPUT
$\overline{AEN1}$ $\overline{AEN2}$	ADDRESS ENABLED QUALIFIERS FOR RDY1,2	V_{CC}	+5 VOLTS
		GND	0 VOLTS

M8284 Pin Names

I8284
CLOCK GENERATOR AND DRIVER FOR iAPX 86, 88 PROCESSORS

INDUSTRIAL

- **Industrial Temperature Range: −40°C to +85°C**
- **Generates the System Clock for the Industrial iAPX 86/10**
- **Uses a Crystal or a TTL Signal for Frequency Source**
- **Single +5V Power Supply**
- **18-Pin Package**
- **Generates System Reset Output from Schmitt Trigger Input**
- **Provides Local Ready and MULTIBUS™ Ready Synchronization**
- **Capable of Clock Synchronization with other 8284's**

The I8284 is a bipolar clock generator/driver designed to provide clock signals for the iAPX 86, 88 and peripherals. It also contains READY logic for operation with two MULTIBUS™ systems and provides the processors required READY synchronization and timing. Reset logic with hysteresis and synchronization is also provided.

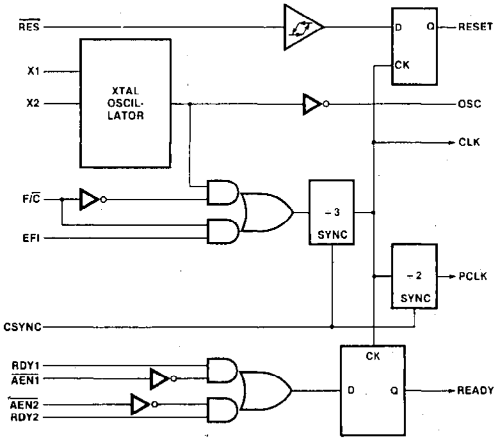


Figure 1. Block Diagram

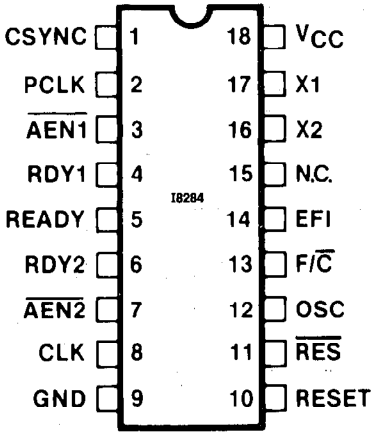


Figure 2. Pin Configuration

X1 X2	CONNECTIONS FOR CRYSTAL	$\overline{RES}$	RESET INPUT
F/$\overline{C}$	CLOCK SOURCE SELECT	RESET	SYNCHRONIZED RESET OUTPUT
EFI	EXTERNAL CLOCK INPUT	OSC	OSCILLATOR OUTPUT
CSYNC	CLOCK SYNCHRONIZATION INPUT	CLK	MOS CLOCK ID8086
RDY1 RDY2	READY SIGNAL FROM TWO MULTIBUS™ SYSTEMS	PCLK	TTL CLOCK FOR PERIPHERALS
$\overline{AEN1}$ $\overline{AEN2}$	ADDRESS ENABLED QUALIFIERS FOR RDY1,2	READY	SYNCHRONIZED READY OUTPUT
		V_{CC}	+5 VOLTS
		GND	0 VOLTS

I8284 Pin Names

I8286/8287
OCTAL BUS TRANSCEIVER

INDUSTRIAL

- **Data Bus Buffer Driver for iAPX 86,88, MCS-80®, MCS-85®, and MCS-48® Families**
- **High Output Drive Capability for Driving System Data Bus**
- **Fully Parallel 8-Bit Transceivers**
- **3-State Outputs**
- **20-Pin Package with 0.3" Center**
- **No Output Low Noise when Entering or Leaving High Impedance State**
- **Industrial Temperature Range: −40°C to +85°C**

The I8286 and I8287 are 8-bit bipolar transceivers with 3-state outputs. The I8287 inverts the input data at its outputs while the I8286 does not. Thus, a wide variety of applications for buffering in microcomputer systems can be met.

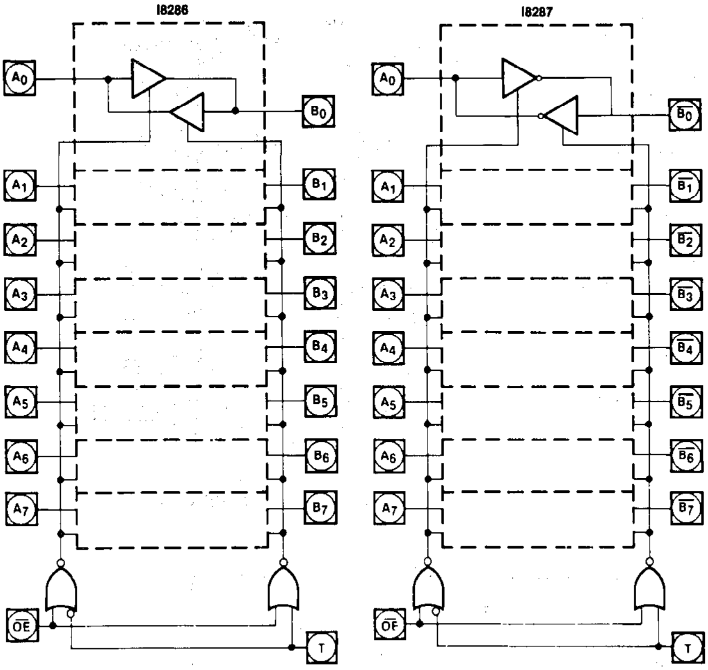


Figure 1. Logic Diagrams

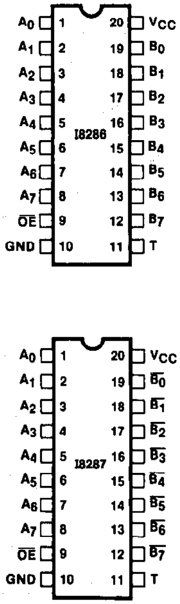


Figure 2. Pin Configuration

8288
BUS CONTROLLER
FOR iAPX 86, 88 PROCESSORS

- **Bipolar Drive Capability**
- **Provides Advanced Commands**
- **Provides Wide Flexibility in System Configurations**
- **3-State Command Output Drivers**
- **Configurable for Use with an I/O Bus**
- **Facilitates Interface to One or Two Multi-Master Busses**

The Intel® 8288 Bus Controller is a 20-pin bipolar component for use with medium-to-large iAPX 86, 88 processing systems. The bus controller provides command and control timing generation as well as bipolar bus drive capability while optimizing system performance.

A strapping option on the bus controller configures it for use with a multi-master system bus and separate I/O bus.

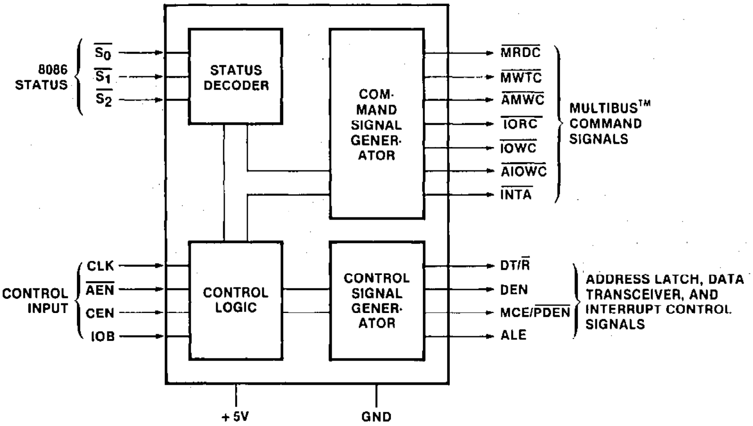
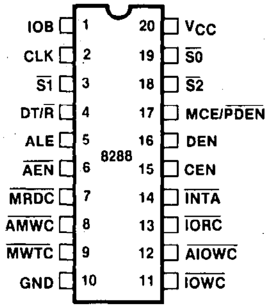


Figure 1. Block Diagram



IOB 1
CLK 2
$\overline{\text{S1}}$ 3
DT/$\overline{\text{R}}$ 4
ALE 5
$\overline{\text{AEN}}$ 6
$\overline{\text{MRDC}}$ 7
$\overline{\text{AMWC}}$ 8
$\overline{\text{MWTC}}$ 9
GND 10
8288
20 V_{CC}
19 $\overline{\text{S0}}$
18 $\overline{\text{S2}}$
17 MCE/$\overline{\text{PDEN}}$
16 DEN
15 CEN
14 $\overline{\text{INTA}}$
13 $\overline{\text{IORC}}$
12 $\overline{\text{AIOWC}}$
11 $\overline{\text{IOWC}}$

Figure 2. Pin Configuration

Table 1. Pin Description

Symbol	Type	Name and Function
V_{CC}		**Power:** +5V supply.
GND		**Ground.**
$\overline{S_0}, \overline{S_1}, \overline{S_2}$	I	**Status Input Pins:** These pins are the status input pins from the 8086, 8088 or 8089 processors. The 8288 decodes these inputs to generate command and control signals at the appropriate time. When these pins are not in use (passive) they are all HIGH. (See chart under Command and Control Logic.)
CLK	I	**Clock:** This is a clock signal from the 8284 clock generator and serves to establish when command and control signals are generated.
ALE	O	**Address Latch Enable:** This signal serves to strobe an address into the address latches. This signal is active HIGH and latching occurs on the falling (HIGH to LOW) transition. ALE is intended for use with transparent D type latches.
DEN	O	**Data Enable:** This signal serves to enable data transceivers onto either the local or system data bus. This signal is active HIGH.
DT/$\overline{\text{R}}$	O	**Data Transmit/Receive:** This signal establishes the direction of data flow through the transceivers. A HIGH on this line indicates Transmit (write to I/O or memory) and a LOW indicates Receive (Read).
$\overline{\text{AEN}}$	I	**Address Enable:** $\overline{\text{AEN}}$ enables command outputs of the 8288 Bus Controller at least 115 ns after it becomes active (LOW). $\overline{\text{AEN}}$ going inactive immediately 3-states the command output drivers. $\overline{\text{AEN}}$ does not affect the I/O command lines if the 8288 is in the I/O Bus mode (IOB tied HIGH).
CEN	I	**Command Enable:** When this signal is LOW all 8288 command outputs and the DEN and $\overline{\text{PDEN}}$ control outputs are forced to their inactive state. When this signal is HIGH, these same outputs are enabled.
IOB	I	**Input/Output Bus Mode:** When the IOB is strapped HIGH the 8288 functions in the I/O Bus mode. When it is strapped LOW, the 8288 functions in the System Bus mode. (See sections on I/O Bus and System Bus modes).
$\overline{\text{AIOWC}}$	O	**Advanced I/O Write Command:** The $\overline{\text{AIOWC}}$ issues an I/O Write Command earlier in the machine cycle to give I/O devices an early indication of a write instruction. Its timing is the same as a read command signal. $\overline{\text{AIOWC}}$ is active LOW.
$\overline{\text{IOWC}}$	O	**I/O Write Command:** This command line instructs an I/O device to read the data on the data bus. This signal is active LOW.
$\overline{\text{IORC}}$	O	**I/O Read Command:** This command line instructs an I/O device to drive its data onto the data bus. This signal is active LOW.
$\overline{\text{AMWC}}$	O	**Advanced Memory Write Command:** The $\overline{\text{AMWC}}$ issues a memory write command earlier in the machine cycle to give memory devices an early indication of a write instruction. Its timing is the same as a read command signal. $\overline{\text{AMWC}}$ is active LOW.
$\overline{\text{MWTC}}$	O	**Memory Write Command:** This command line instructs the memory to record the data present on the data bus. This signal is active LOW.
$\overline{\text{MRDC}}$	O	**Memory Read Command:** This command line instructs the memory to drive its data onto the data bus. This signal is active LOW.
$\overline{\text{INTA}}$	O	**Interrupt Acknowledge:** This command line tells an interrupting device that its interrupt has been acknowledged and that it should drive vectoring information onto the data bus. This signal is active LOW.
MCE/$\overline{\text{PDEN}}$	O	This is a dual function pin. **MCE (IOB is tied LOW):** Master Cascade Enable occurs during an interrupt sequence and serves to read a Cascade Address from a master PIC (Priority Interrupt Controller) onto the data bus. The MCE signal is active HIGH. **$\overline{\text{PDEN}}$ (IOB is tied HIGH):** Peripheral Data Enable enables the data bus transceiver for the I/O bus that DEN performs for the system bus. $\overline{\text{PDEN}}$ is active LOW.

FUNCTIONAL DESCRIPTION

Command and Control Logic

The command logic decodes the three 8086, 8088 or 8089 CPU status lines ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) to determine what command is to be issued.

This chart shows the meaning of each status "word".

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Processor State	8288 Command
0	0	0	Interrupt Acknowledge	$\overline{\text{INTA}}$
0	0	1	Read I/O Port	$\overline{\text{IORC}}$
0	1	0	Write I/O Port	$\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$
0	1	1	Halt	None
1	0	0	Code Access	$\overline{\text{MRDC}}$
1	0	1	Read Memory	$\overline{\text{MRDC}}$
1	1	0	Write Memory	$\overline{\text{MWTC}}$, $\overline{\text{AMWC}}$
1	1	1	Passive	None

The command is issued in one of two ways dependent on the mode of the 8288 Bus Controller.

I/O Bus Mode — The 8288 is in the I/O Bus mode if the IOB pin is strapped HIGH. In the I/O Bus mode all I/O command lines ($\overline{\text{IORC}}$, $\overline{\text{IOWC}}$, $\overline{\text{AIOWC}}$, $\overline{\text{INTA}}$) are always enabled (i.e., not dependent on $\overline{\text{AEN}}$). When an I/O command is initiated by the processor, the 8288 immediately activates the command lines using PDEN and DT/$\overline{\text{R}}$ to control the I/O bus transceiver. The I/O command lines should not be used to control the system bus in this configuration because no arbitration is present. This mode allows one 8288 Bus Controller to handle two external busses. No waiting is involved when the CPU wants to gain access to the I/O bus. Normal memory access requires a "Bus Ready" signal ($\overline{\text{AEN}}$ LOW) before it will proceed. It is advantageous to use the IOB mode if I/O or peripherals dedicated to one processor exist in a multi-processor system.

System Bus Mode — The 8288 is in the System Bus mode if the IOB pin is strapped LOW. In this mode no command is issued until 115 ns after the $\overline{\text{AEN}}$ Line is activated (LOW). This mode assumes bus arbitration logic will inform the bus controller (on the $\overline{\text{AEN}}$ line) when the bus is free for use. Both memory and I/O commands wait for bus arbitration. This mode is used when only one bus exists. Here, both I/O and memory are shared by more than one processor.

COMMAND OUTPUTS

The advanced write commands are made available to initiate write procedures early in the machine cycle. This signal can be used to prevent the processor from entering an unnecessary wait state.

The command outputs are:

$\overline{\text{MRDC}}$ — Memory Read Command
$\overline{\text{MWTC}}$ — Memory Write Command
$\overline{\text{IORC}}$ — I/O Read Command
$\overline{\text{IOWC}}$ — I/O Write Command
$\overline{\text{AMWC}}$ — Advanced Memory Write Command
$\overline{\text{AIOWC}}$ — Advanced I/O Write Command
$\overline{\text{INTA}}$ — Interrupt Acknowledge

$\overline{\text{INTA}}$ (Interrupt Acknowledge) acts as an I/O read during an interrupt cycle. Its purpose is to inform an interrupting device that its interrupt is being acknowledged and that it should place vectoring information onto the data bus.

CONTROL OUTPUTS

The control outputs of the 8288 are Data Enable (DEN), Data Transmit/Receive (DT/$\overline{\text{R}}$) and Master Cascade Enable/Peripheral Data Enable (MCE/$\overline{\text{PDEN}}$). The DEN signal determines when the external bus should be enabled onto the local bus and the DT/$\overline{\text{R}}$ determines the direction of data transfer. These two signals usually go to the chip select and direction pins of a transceiver.

The MCE/$\overline{\text{PDEN}}$ pin changes function with the two modes of the 8288. When the 8288 is in the IOB mode (IOB HIGH) the $\overline{\text{PDEN}}$ signal serves as a dedicated data enable signal for the I/O or Peripheral System bus.

INTERRUPT ACKNOWLEDGE AND MCE

The MCE signal is used during an interrupt acknowledge cycle if the 8288 is in the System Bus mode (IOB LOW). During any interrupt sequence there are two interrupt acknowledge cycles that occur back to back. During the first interrupt cycle no data or address transfers take place. Logic should be provided to mask off MCE during this cycle. Just before the second cycle begins the MCE signal gates a master Priority Interrupt Controller's (PIC) cascade address onto the processor's local bus where ALE (Address Latch Enable) strobes it into the address latches. On the leading edge of the second interrupt cycle the addressed slave PIC gates an interrupt vector onto the system data bus where it is read by the processor.

If the system contains only one PIC, the MCE signal is not used. In this case the second Interrupt Acknowledge signal gates the interrupt vector onto the processor bus.

ADDRESS LATCH ENABLE AND HALT

Address Latch Enable (ALE) occurs during each machine cycle and serves to strobe the current address into the address latches. ALE also serves to strobe the status ($\overline{S_0}$, $\overline{S_1}$, $\overline{S_2}$) into a latch for halt state decoding.

COMMAND ENABLE

The Command Enable (CEN) input acts as a command qualifier for the 8288. If the CEN pin is high the 8288 functions normally. If the CEN pin is pulled LOW, all command lines are held in their inactive state (not 3-state). This feature can be used to implement memory partitioning and to eliminate address conflicts between system bus devices and resident bus devices.

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias	0°C to 70°C
Storage Temperature	−65°C to +150°C
All Output and Supply Voltages	−0.5V to +7V
All Input Voltages	−1.0V to +5.5V
Power Dissipation	1.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70°C)

Symbol	Parameter	Min.	Max.	Unit	Test Conditions
V_C	Input Clamp Voltage		−1	V	$I_C = -5$ mA
I_{CC}	Power Supply Current		230	mA	
I_F	Forward Input Current		−0.7	mA	$V_F = 0.45V$
I_R	Reverse Input Current		50	μA	$V_R = V_{CC}$
V_{OL}	Output Low Voltage				
	Command Outputs		0.5	V	$I_{OL} = 32$ mA
	Control Outputs		0.5	V	$I_{OL} = 16$ mA
V_{OH}	Output High Voltage				
	Command Outputs	2.4		V	$I_{OH} = -5$ mA
	Control Outputs	2.4		V	$I_{OH} = -1$ mA
V_{IL}	Input Low Voltage		0.8	V	
V_{IH}	Input High Voltage	2.0		V	
I_{OFF}	Output Off Current		100	μA	V_{OFF} = 0.4 to 5.25V

A.C. CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70°C)

TIMING REQUIREMENTS

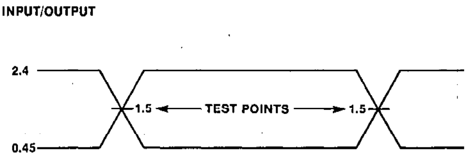
Symbol	Parameter	Min.	Max.	Unit	Test Conditions
TCLCL	CLK Cycle Period	100		ns	
TCLCH	CLK Low Time	50		ns	
TCHCL	CLK High Time	30		ns	
TSVCH	Status Active Setup Time	35		ns	
TCHSV	Status Active Hold Time	10		ns	
TSHCL	Status Inactive Setup Time	35		ns	
TCLSH	Status Inactive Hold Time	10		ns	
TILIH	Input, Rise Time		20	ns	From 0.8V to 2.0V
TIHIL	Input, Fall Time		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

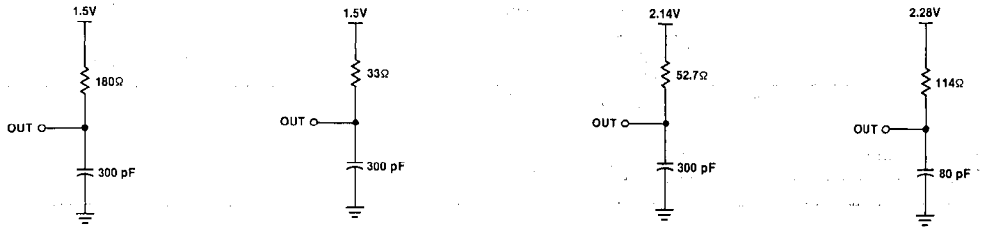
Symbol	Parameter	Min.	Max.	Unit	Test Conditions	
TCVNV	Control Active Delay	5	45	ns		
TCVNX	Control Inactive Delay	10	45	ns		
TCLLH, TCLMCH	ALE MCE Active Delay (from CLK)		20	ns		
TSVLH, TSVMCH	ALE MCE Active Delay (from Status)		20	ns		
TCHLL	ALE Inactive Delay	4	15	ns	$\overline{\text{MRDC}}$	
TCLML	Command Active Delay	10	35	ns	$\overline{\text{IORC}}$	
TCLMH	Command Inactive Delay	10	35	ns	$\overline{\text{MWTC}}$	$I_{OL} = 32$ mA
TCHDTL	Direction Control Active Delay		50	ns	$\overline{\text{IOWC}}$	$I_{OH} = -5$ mA
TCHDTH	Direction Control Inactive Delay		30	ns	$\overline{\text{INTA}}$	$C_L = 300$ pF
TAELCH	Command Enable Time		40	ns	$\overline{\text{AMWC}}$	
TAEHCZ	Command Disable Time		40	ns	$\overline{\text{AIOWC}}$	
TAELCV	Enable Delay Time	115	200	ns		$I_{OL} = 16$ mA
TAEVNV	AEN to DEN		20	ns	Other	$I_{OH} = -1$ mA
TCEVNV	CEN to DEN, PDEN		25	ns		$C_L = 80$ pF
TCELRH	CEN to Command		TCLML	ns		
TOLOH	Output, Rise Time		20	ns	From 0.8V to 2.0V	
TOHOL	Output, Fall Time		12	ns	From 2.0V to 0.8V	

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." THE CLOCK IS DRIVEN AT 4.3V AND 0.25V. TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

TEST LOAD CIRCUITS—3-STATE COMMAND OUTPUT TEST LOAD



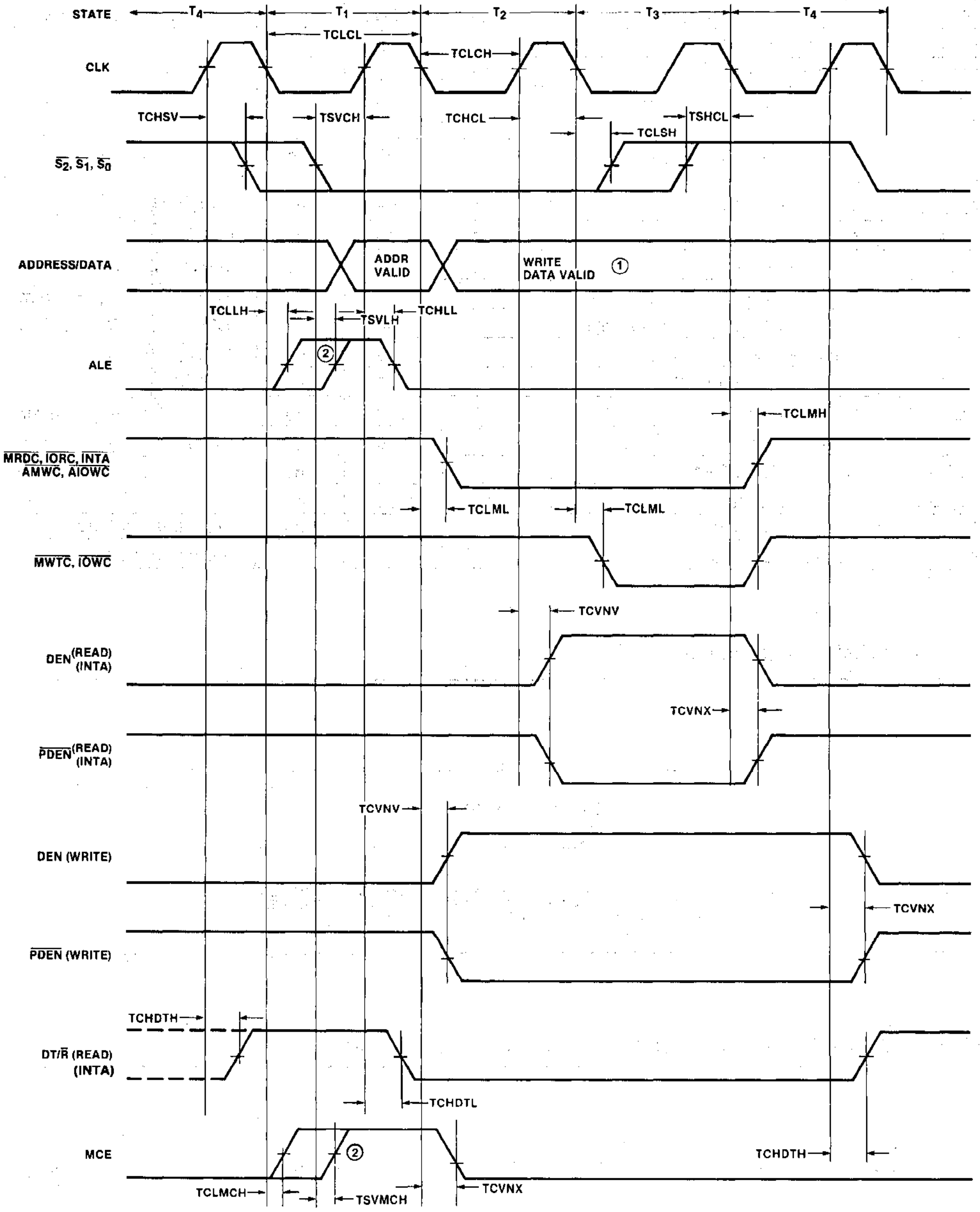
3-STATE TO HIGH

3-STATE TO LOW

COMMAND OUTPUT TEST LOAD

CONTROL OUTPUT TEST LOAD

WAVEFORMS

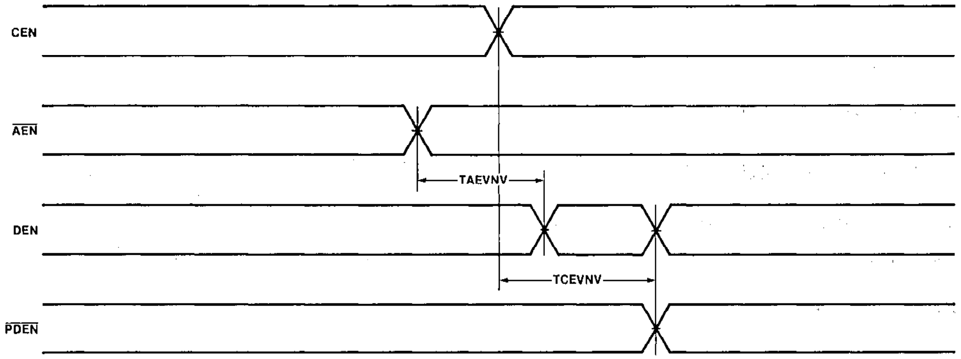

STATE
T4
T1
T2
T3
T4
TCLCL
CLK
TCLCH
TCHSV
TSVCH
TCHCL
TSHCL
TCLSH
$\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$
ADDRESS/DATA
ADDR VALID
WRITE DATA VALID
①
TCLLH
TSVLH
TCHLL
ALE
②
TCLMH
$\overline{MRDC}$, $\overline{IORC}$, $\overline{INTA}$ $\overline{AMWC}$, $\overline{AIOWC}$
TCLML
TCLML
$\overline{MWTC}$, $\overline{IOWC}$
TCVNV
DEN (READ) (INTA)
TCVNX
$\overline{PDEN}$ (READ) (INTA)
TCVNV
DEN (WRITE)
TCVNX
$\overline{PDEN}$ (WRITE)
TCHDTH
DT/$\overline{R}$ (READ) (INTA)
TCHDTL
MCE
②
TCHDTH
TCLMCH
TSVMCH
TCVNX

NOTES:

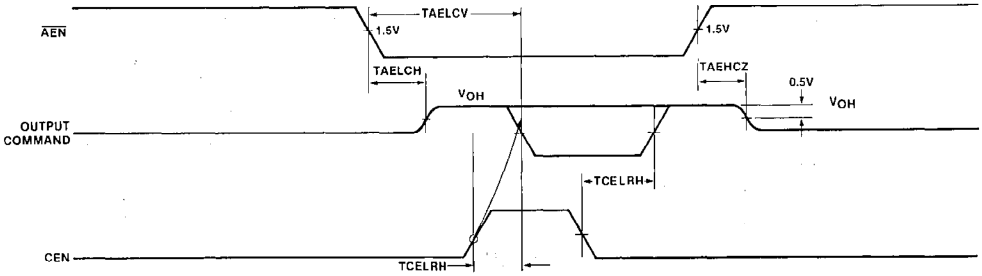
1. ADDRESS/DATA BUS IS SHOWN ONLY FOR REFERENCE PURPOSES.
2. LEADING EDGE OF ALE AND MCE IS DETERMINED BY THE FALLING EDGE OF CLK OR STATUS GOING ACTIVE, WHICHEVER OCCURS LAST.
3. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS SPECIFIED OTHERWISE.

WAVEFORMS (Continued)

DEN, PDEN QUALIFICATION TIMING



ADDRESS ENABLE (AEN) TIMING (3-STATE ENABLE/DISABLE)



NOTE: CEN must be low or valid prior to T2 to prevent the command from being generated.

I8288
BUS CONTROLLER
FOR iAPX 86, 88 PROCESSORS

INDUSTRIAL

- **Bipolar Drive Capability**
- **Provides Advanced Commands**
- **Provides Wide Flexibility in System Configurations**
- **3-State Command Output Drivers**
- **Configurable for Use with an I/O Bus**
- **Facilitates Interface to One or Two Multi-Master Busses**
- **Industrial Temperature Range: −40°C to 85°C**

The Intel® I8288 Bus Controller is a 20-pin bipolar component for use with medium-to-large iAPX 86 processing systems. The bus controller provides command and control timing generation as well as bipolar bus drive capability while optimizing system performance.

A strapping option on the bus controller configures it for use with a multi-master system bus and separate I/O bus.

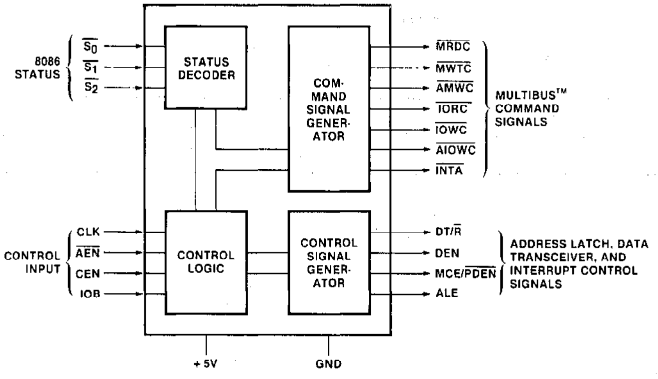


Figure 1. Block Diagram

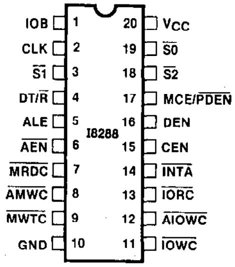


Figure 2. Pin Configuration

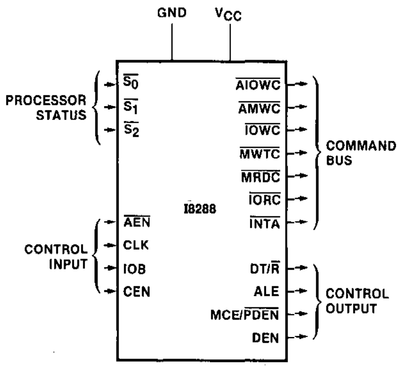


Figure 3. Functional Pin-Out

PRELIMINARY

8289 BUS ARBITER

- Provides Multi-Master System Bus Protocol
- Synchronizes iAPX 86, 88 Processors with Multi-Master Bus
- Provides Simple Interface with 8288 Bus Controller
- Four Operating Modes for Flexible System Configuration
- Compatible with Intel Bus Standard MULTIBUS™
- Provides System Bus Arbitration for 8089 IOP in Remote Mode

The Intel 8289 Bus Arbiter is a 20-pin, 5-volt-only bipolar component for use with medium to large iAPX 86, 88 multi-master/multiprocessing systems. The 8289 provides system bus arbitration for systems with multiple bus masters, such as an 8086 CPU with 8089 IOP in its REMOTE mode, while providing bipolar buffering and drive capability.

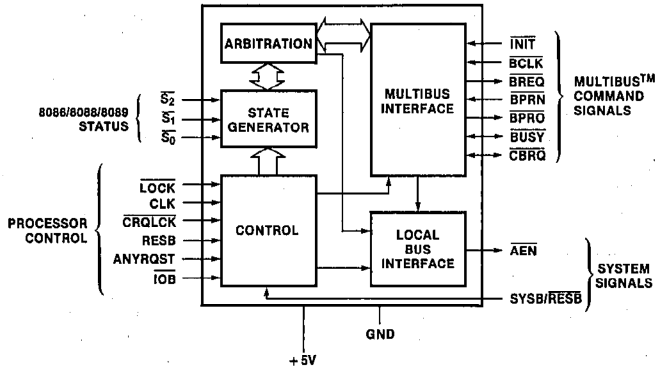


Figure 1. Block Diagram

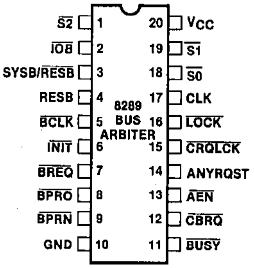


Figure 2. Pin Diagram

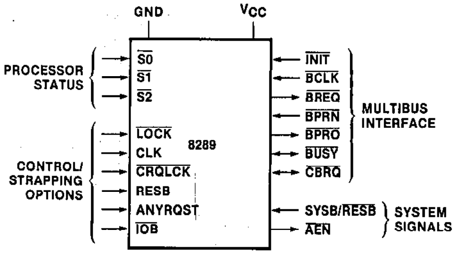


Figure 3. Functional Pinout

Table 1. Pin Description

Symbol	Type	Name and Function
V_{CC}		**Power:** +5V supply ±10%.
GND		**Ground.**
$\overline{S0}$,$\overline{S1}$,$\overline{S2}$	I	**Status Input Pins:** The status input pins from an 8086, 8088 or 8089 processor. The 8289 decodes these pins to initiate bus request and surrender actions. (See Table 2.)
CLK	I	**Clock:** From the 8284 clock chip and serves to establish when bus arbiter actions are initiated.
$\overline{LOCK}$	I	**Lock:** A processor generated signal which when activated (low) prevents the arbiter from surrendering the multi-master system bus to any other bus artiter, regardless of its priority.
$\overline{CRQLCK}$	I	**Common Request Lock:** An active low signal which prevents the arbiter from surrendering the multi-master system bus to any other bus arbiter requesting the bus through the $\overline{CBRQ}$ input pin.
RESB	I	**Resident Bus:** A strapping option to configure the arbiter to operate in systems having both a multi-master system bus and a Resident Bus. Strapped high, the multi-master system bus is requested or surrendered as a function of the SYSB/$\overline{RESB}$ input pin. Strapped low, the SYSB/$\overline{RESB}$ input is ignored.
ANYRQST	I	**Any Request:** A strapping option which permits the multi-master system bus to be surrendered to a lower priority arbiter as if it were an arbiter of higher priority (i.e., when a lower priority arbiter requests the use of the multi-master system bus, the bus is surrendered as soon as it is possible). When ANYRQST is strapped low, the bus is surrendered according to Table 2. If ANYRQST is strapped high and $\overline{CBRQ}$ is activated, the bus is surrendered at the end of the present bus cycle. Strapping $\overline{CBRQ}$ low and ANYRQST high forces the 8289 arbiter to surrender the multi-master system bus after each transfer cycle. Note that when surrender occurs BREQ is driven false (high).
$\overline{IOB}$	I	**IO Bus:** A strapping option which configures the 8289 Arbiter to operate in systems having both an IO Bus (Peripheral Bus) and a multi-master system bus. The arbiter requests and surrenders the use of the multi-master system bus as a function of the status line, $\overline{S2}$. The multi-master system bus is permitted to be surrendered while the processor is performing IO commands and is requested whenever the processor performs a memory command. Interrupt cycles are assumed as coming from the peripheral bus and are treated as an IO command.
$\overline{AEN}$	O	**Address Enable:** The output of the 8289 Arbiter to the processor's address latches, to the 8288 Bus Controller and 8284A Clock Generator. $\overline{AEN}$ serves to instruct the Bus Controller and address latches when to tri-state their output drivers.
SYSB/ $\overline{RESB}$	I	**System Bus/Resident Bus:** An input signal when the arbiter is configured in the S.R. Mode (RESB is strapped high) which determines when the multi-master system bus is requested and multi-master system bus surrendering is permitted. The signal is intended to originate from a form of address-mapping circuitry, as a decoder or PROM attached to the resident address bus. Signal transitions and glitches are permitted on this pin from $\phi 1$ of T4 to $\phi 1$ of T2 of the processor cycle. During the period from $\phi 1$ of T2 to $\phi 1$ of T4, only clean transitions are permitted on this pin (no glitches). If a glitch occurs, the arbiter may capture or miss it, and the multi-master system bus may be requested or surrendered, depending upon the state of the glitch. The arbiter requests the multi-master system bus in the S.R. Mode when the state of the SYSB/$\overline{RESB}$ pin is high and permits the bus to be surrendered when this pin is low.
$\overline{CBRQ}$	I/O	**Common Bus Request:** An input signal which instructs the arbiter if there are any other arbiters of lower priority requesting the use of the multi-master system bus. The $\overline{CBRQ}$ pins (open-collector output) of all the 8289 Bus Arbiters which surrender to the multi-master system bus upon request are connected together. The Bus Arbiter running the current transfer cycle will not itself pull the $\overline{CBRQ}$ line low. Any other arbiter connected to the $\overline{CBRQ}$ line can request the multi-master system bus. The arbiter presently running the current transfer cycle drops its $\overline{BREQ}$ signal and surrenders the bus whenever the proper surrender conditions exist. Strapping $\overline{CBRQ}$ low and ANYRQST high allows the multi-master system bus to be surrendered after each transfer cycle. See the pin definition of ANYRQST.
$\overline{INIT}$	I	**Initialize:** An active low multi-master system bus input signal used to reset all the bus arbiters on the multi-master system bus. After initialization, no arbiters have the use of the multi-master system bus.

Table 1. Pin Descriptions (Continued)

Symbol	Type	Name and Function
$\overline{\text{BCLK}}$	I	**Bus Clock:** The multi-master system bus clock to which all multi-master system bus interface signals are synchronized.
$\overline{\text{BREQ}}$	O	**Bus Request:** An active low output signal in the parallel Priority Resolving Scheme which the arbiter activates to request the use of the multi-master system bus.
$\overline{\text{BPRN}}$	I	**Bus Priority In:** The active low signal returned to the arbiter to instruct it that it may acquire the multi-master system bus on the next falling edge of BCLK. $\overline{\text{BPRN}}$ indicates to the arbiter that it is the highest priority requesting arbiter presently on the bus. The loss of $\overline{\text{BPRN}}$ instructs the arbiter that it has lost priority to a higher priority arbiter.
$\overline{\text{BPRO}}$	O	**Bus Priority Out:** An active low output signal used in the serial priority resolving scheme where $\overline{\text{BPRO}}$ is daisy-chained to $\overline{\text{BPRN}}$ of the next lower priority arbiter.
$\overline{\text{BUSY}}$	I/O	**Busy:** An active low open collector multi-master system bus interface signal used to instruct all the arbiters on the bus when the multi-master system bus is available. When the multi-master system bus is available the highest requesting arbiter (determined by $\overline{\text{BPRN}}$) seizes the bus and pulls $\overline{\text{BUSY}}$ low to keep other arbiters off of the bus. When the arbiter is done with the bus, it releases the $\overline{\text{BUSY}}$ signal, permitting it to go high and thereby allowing another arbiter to acquire the multi-master system bus.

FUNCTIONAL DESCRIPTION

The 8289 Bus Arbiter operates in conjunction with the 8288 Bus Controller to interface iAPX 86, 88 processors to a multi-master system bus (both the iAPX 86 and iAPX 88 are configured in their max mode). The processor is unaware of the arbiter's existence and issues commands as though it has exclusive use of the system bus. If the processor does not have the use of the multi-master system bus, the arbiter prevents the Bus Controller (8288), the data transceivers and the address latches from accessing the system bus (e.g. all bus driver outputs are forced into the high impedance state). Since the command sequence was not issued by the 8288, the system bus will appear as "Not Ready" and the processor will enter wait states. The processor will remain in Wait until the Bus Arbiter acquires the use of the multi-master system bus whereupon the arbiter will allow the bus controller, the data transceivers, and the address latches to access the system. Typically, once the command has been issued and a data transfer has taken place, a transfer acknowledge (XACK) is returned to the processor to indicate "READY" from the accessed slave device. The processor then completes its transfer cycle. Thus the arbiter serves to multiplex a processor (or bus master) onto a multi-master system bus and avoid contention problems between bus masters.

Arbitration Between Bus Masters

In general, higher priority masters obtain the bus when a lower priority master completes its present transfer cycle. Lower priority bus masters obtain the bus when a higher priority master is not accessing the system bus. A strapping option (ANYRQST) is provided to allow the arbiter to surrender the bus to a lower priority master as though it were a master of higher priority. If there are no other bus masters requesting the bus, the arbiter maintains the bus so long as its processor has not entered the HALT State. The arbiter will not voluntarily surrender the system bus and has to be forced off by another master's bus request, the HALT State being the only exception. Additional strapping options permit other modes of operation wherein the multi-master system bus is surrendered or requested under different sets of conditions.

Priority Resolving Techniques

Since there can be many bus masters on a multi-master system bus, some means of resolving priority between bus masters simultaneously requesting the bus must be provided. The 8289 Bus Arbiter provides several resolving techniques. All the techniques are based on a priority concept that at a given time one bus master will have priority above all the rest. There are provisions for using parallel priority resolving techniques, serial priority resolving techniques, and rotating priority techniques.

PARALLEL PRIORITY RESOLVING

The parallel priority resolving technique uses a separate bus request line ($\overline{\text{BREQ}}$) for each arbiter on the multi-master system bus, see Figure 4. Each $\overline{\text{BREQ}}$ line enters into a priority encoder which generates the binary address of the highest priority $\overline{\text{BREQ}}$ line which is active. The binary address is decoded by a decoder to select the corresponding $\overline{\text{BPRN}}$ (Bus Priority In) line to be returned to the highest priority requesting arbiter. The arbiter receiving priority ($\overline{\text{BPRN}}$ true) then allows its associated bus master onto the multi-master system bus as soon as it becomes available (i.e., the bus is no longer busy). When one bus arbiter gains priority over another arbiter it cannot immediately seize the bus, it must wait until the present bus transaction is complete.

Upon completing its transaction the present bus occupant recognizes that it no longer has priority and surrenders the bus by releasing BUSY. BUSY is an active low "OR" tied signal line which goes to every bus arbiter on the system bus. When BUSY goes inactive (high), the arbiter which presently has bus priority (BPRN true) then seizes the bus and pulls BUSY low to keep other arbiters off of the bus. See waveform timing diagram, Figure 5. Note that all multi-master system bus transactions are synchronized to the bus clock (BCLK). This allows the parallel priority resolving circuitry or any other priority resolving scheme employed to settle.

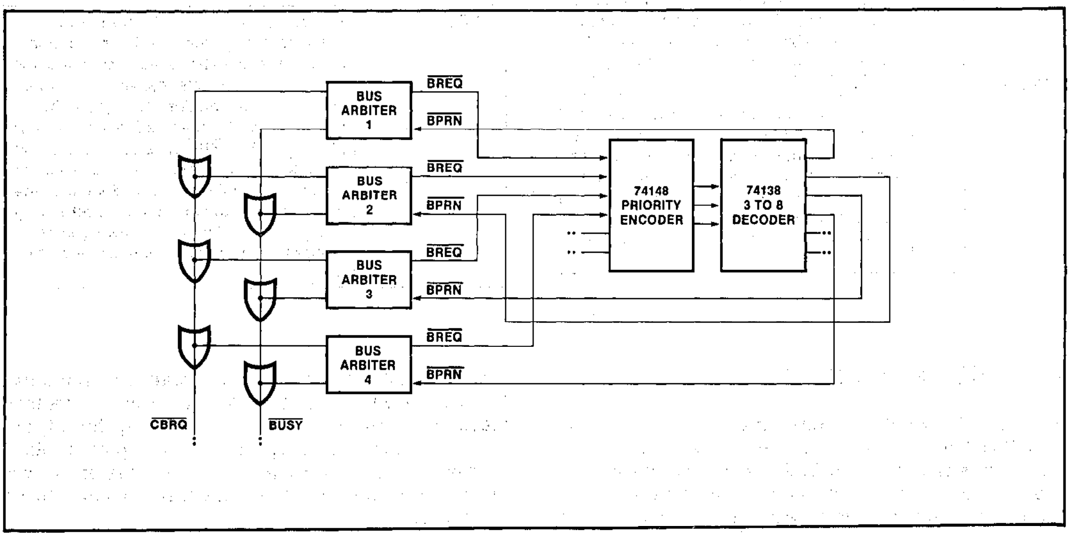
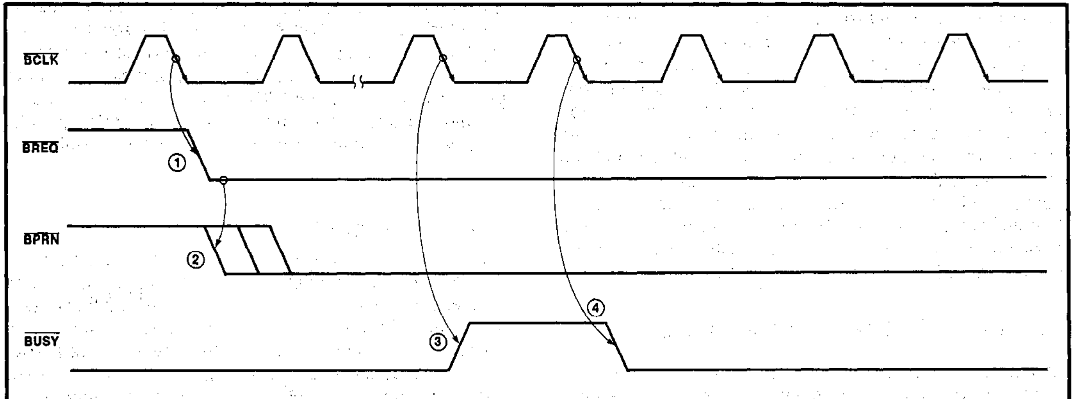


Figure 4. Parallel Priority Resolving Technique

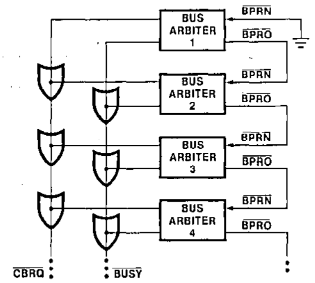


① HIGHER PRIORITY BUS ARBITER REQUESTS THE MULTI-MASTER SYSTEM BUS.
② ATTAINS PRIORITY.
③ LOWER PRIORITY BUS ARBITER RELEASES BUSY.
④ HIGHER PRIORITY BUS ARBITER THEN ACQUIRES THE BUS AND PULLS BUSY DOWN.

Figure 5. Higher Priority Arbiter obtaining the Bus from a Lower Priority Arbiter

SERIAL PRIORITY RESOLVING

The serial priority resolving technique eliminates the need for the priority encoder-decoder arrangement by daisy-chaining the bus arbiters together, connecting the higher priority bus arbiter's $\overline{\text{BPRO}}$ (Bus Priority Out) output to the $\overline{\text{BPRN}}$ of the next lower priority. See Figure 6.



THE NUMBER OF ARBITERS THAT MAY BE DAISY-CHAINED TOGETHER IN THE SERIAL PRIORITY RESOLVING SCHEME IS A FUNCTION OF BCLK AND THE PROPAGATION DELAY FROM ARBITER TO ARBITER. NORMALLY, AT 10 MHz ONLY 3 ARBITER MAY BE DAISY-CHAINED.

Figure 6. Serial Priority Resolving

ROTATING PRIORITY RESOLVING

The rotating priority resolving technique is similar to that of the parallel priority resolving technique except that priority is dynamically re-assigned. The priority encoder is replaced by a more complex circuit which rotates priority between requesting arbiters thus allowing each arbiter an equal chance to use the multi-master system bus, over time.

Which Priority Resolving Technique To Use

There are advantages and disadvantages for each of the techniques described above. The rotating priority resolving technique requires substantial external logic to implement while the serial technique uses no external logic but can accommodate only a limited number of bus arbiters before the daisy-chain propagation delay exceeds the multi-master's system bus clock ($\overline{\text{BCLK}}$). The parallel priority resolving technique is in general a good compromise between the other two techniques. It allows for many arbiters to be present on the bus while not requiring too much logic to implement.

8289 MODES OF OPERATION

There are two types of processors in the iAPX 86 family. An Input/Output processor (the 8089 IOP) and the iAPX 86/10, 88/10 CPUs. Consequently, there are two basic operating modes in the 8289 bus arbiter. One, the IOB (I/O Peripheral Bus) mode, permits the processor access to both an I/O Peripheral Bus and a multi-master system bus. The second, the RESB (Resident Bus mode), permits the processor to communicate over both a Resident Bus and a multi-master system bus. An I/O Peripheral Bus is a bus where all devices on that bus, including memory, are treated as I/O devices and are addressed by I/O commands. All memory commands are directed to another bus, the multi-master system bus. A Resident Bus can issue both memory and I/O commands, but it is a distinct and separate bus from the multi-master system bus. The distinction is that the Resident Bus has only one master, providing full availability and being dedicated to that one master.

The $\overline{\text{IOB}}$ strapping option configures the 8289 Bus Arbiter into the $\overline{\text{IOB}}$ mode and the strapping option RESB configures it into the RESB mode. It might be noted at this point that if both strapping options are strapped false, the arbiter interfaces the processor to a multi-master system bus only (see Figure 7). With both options strapped true, the arbiter interfaces the processor to a multi-master system bus, a Resident Bus, and an I/O Bus.

In the $\overline{\text{IOB}}$ mode, the processor communicates and controls a host of peripherals over the Peripheral Bus. When the I/O Processor needs to communicate with system memory, it does so over the system memory bus. Figure 8 shows a possible I/O Processor system configuration.

The iAPX 86 and iAPX 88 processors can communicate with a Resident Bus and a multi-master system bus. Two bus controllers and only one Bus Arbiter would be needed in such a configuration as shown in Figure 9. In such a system configuration the processor would have access to memory and peripherals of both busses. Memory mapping techniques are applied to select which bus is to be accessed. The SYSB/RESB input on the arbiter serves to instruct the arbiter as to whether or not the system bus is to be accessed. The signal connected to SYSB/RESB also enables or disables commands from one of the bus controllers.

A summary of the modes that the 8289 has, along with its response to its status lines inputs, is summarized in Table 2.

*In some system configurations it is possible for a non-I/O Processor to have access to more than one Multi-Master System Bus, see 8289 Application Note.

Table 2. Summary of 8289 Modes, Requesting and Relinquishing the Multi-Master System Bus

Status Lines From 8086 or 8088 or 8089				IOB Mode Only	RESB (Mode) Only IOB = High RESB = High		IOB Mode RESB Mode IOB = Low RESB = High		Single Bus Mode IOB = High RESB = Low
	$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	$\overline{IOB}$ = Low	SYSB/RESB = High	SYSB/$\overline{RESB}$ = Low	SYSB/$\overline{RESB}$ = High	SYSB/$\overline{RESB}$ = Low	
I/O COMMANDS	0	0	0	x		x	x	x	
	0	0	1	x		x	x	x	
	0	1	0	x		x	x	x	
HALT	0	1	1	x	x	x	x	x	x
MEM COMMANDS	1	0	0			x		x	
	1	0	1			x		x	
	1	1	0			x		x	
IDLE	1	1	1	x	x	x	x	x	x

NOTES:
1. X = Multi-Master System Bus is allowed to be Surrendered.
2. ✓ = Multi-Master System Bus is Requested.

Mode	Pin Strapping	Multi-Master System Bus	
		Requested**	Surrendered*
Single Bus Multi-Master Mode	$\overline{IOB}$ = High RESB = Low	Whenever the processor's status lines go active	HLT + TI • CBRQ + HPBRQ†
RESB Mode Only	$\overline{IOB}$ = High RESB = High	SYSB/$\overline{RESB}$ = High • ACTIVE STATUS	(SYSB/$\overline{RESB}$ = Low + TI) • CBRQ + HLT + HPBRQ
IOB Mode Only	$\overline{IOB}$ = Low RESB = Low	Memory Commands	(I/O Status + TI) • CBRQ + HLT + HPBRQ
IOB Mode·RESB Mode	$\overline{IOB}$ = Low RESB = High	(Memory Command) • (SYSB/$\overline{RESB}$ = High)	((I/O Status Commands) + SYSB/$\overline{RESB}$ = LOW)) • CBRQ + HPBRQ† + HLT

NOTES:
*$\overline{LOCK}$ prevents surrender of Bus to any other arbiter, $\overline{CRQLCK}$ prevents surrender of Bus to any lower priority arbiter.
**Except for HALT and Passive or IDLE Status.
†HPBRQ, Higher priority Bus request or $\overline{BPRN}$ = 1.
1. $\overline{IOB}$ Active Low.
2. RESB Active High.
3. + is read as "OR" and • as "AND."
4. TI = Processor Idle Status $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ = 111
5. HLT = Processor Halt Status $\overline{S2}$, $\overline{S1}$, $\overline{S0}$ = 011

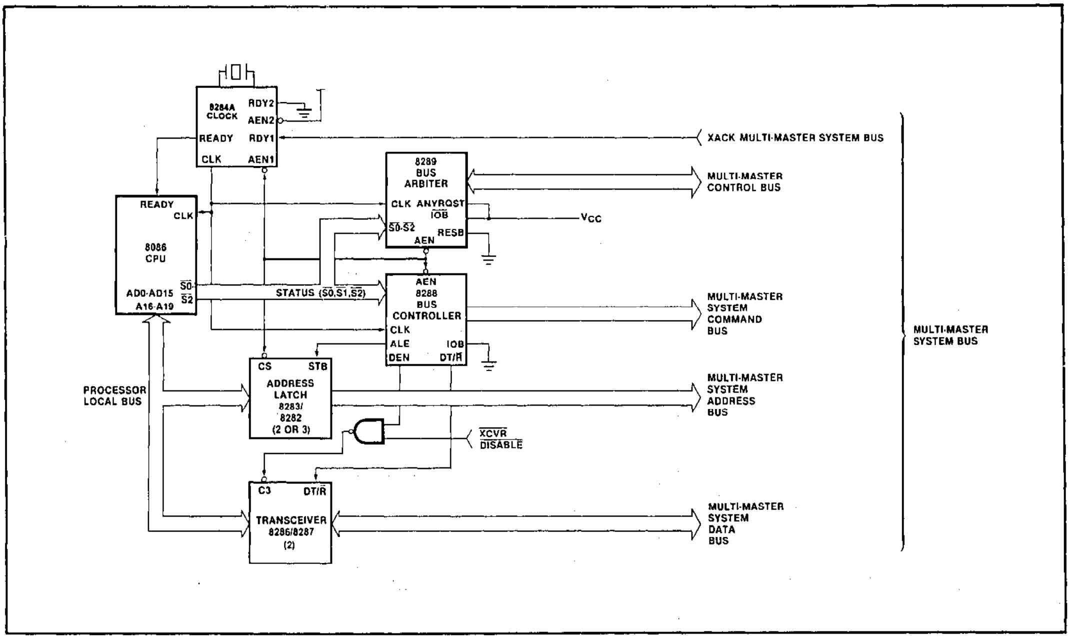


Figure 7. Typical Medium Complexity CPU System

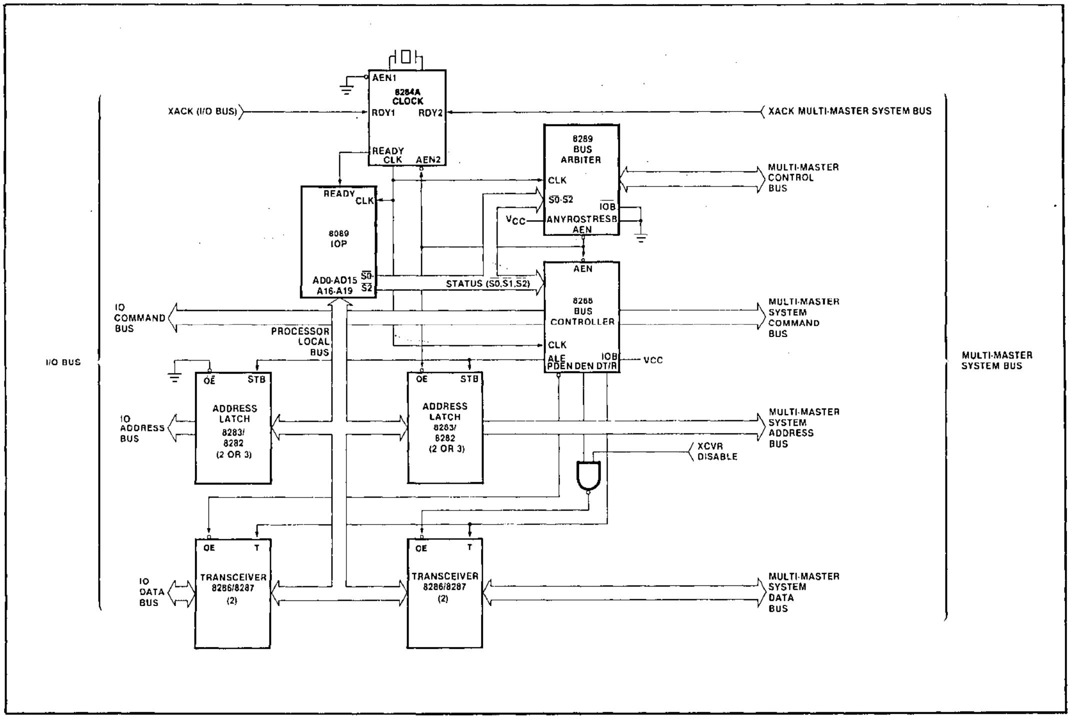
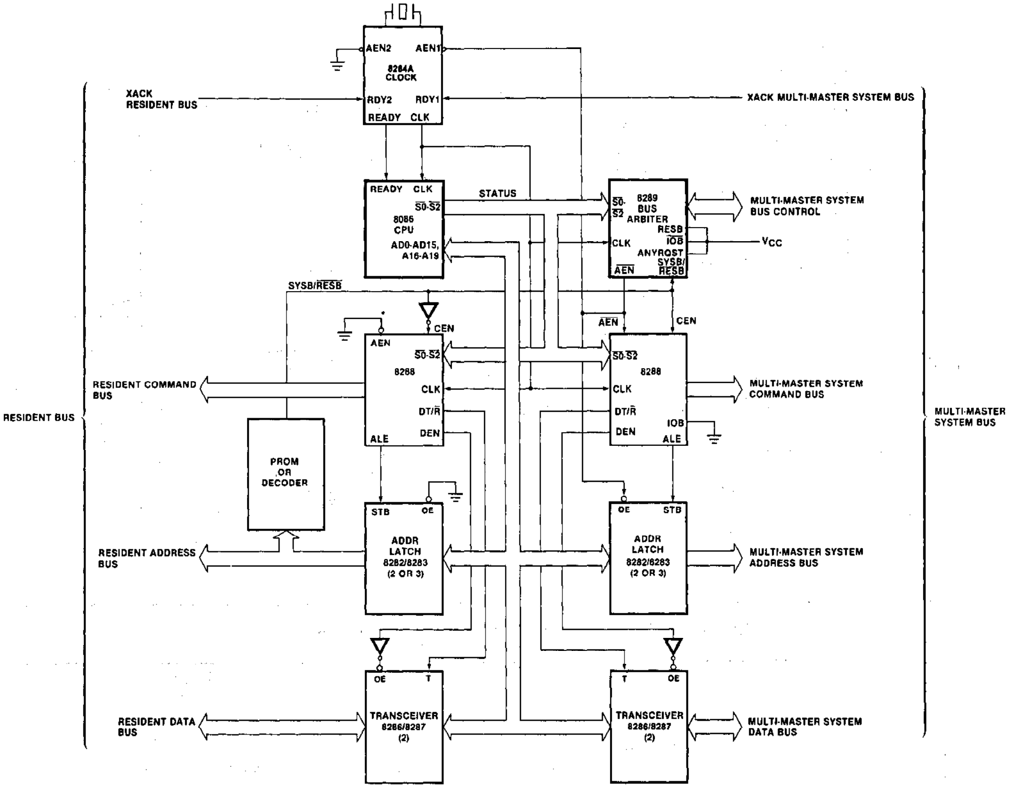


Figure 8. Typical Medium Complexity IOB System



*BY ADDING ANOTHER 8289 ARBITER AND CONNECTING ITS AEN TO THE 8288 WHOSE AEN IS PRESENTLY GROUNDED, THE PROCESSOR COULD HAVE ACCESS TO TWO MULTI-MASTER BUSES.

Figure 9. 8289 Bus Arbiter Shown in System-Resident Bus Configuration

ABSOLUTE MAXIMUM RATINGS*

Temperature Under Bias ....0°C to 70°C
Storage Temperature..............–65°C to +150°C
All Output and Supply Voltages........–0.5V to +7V
All Input Voltages...................–1.0V to +5.5V
Power Dissipation..........................1.5 Watt

**NOTICE: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = +5V ±10%)

Symbol	Parameter	Min.	Max.	Units	Test Condition
V_C	Input Clamp Voltage		–1.0	V	V_{CC} = 4.50V, I_C = –5 mA
I_F	Input Forward Current		–0.5	mA	V_{CC} = 5.50V, V_F = 0.45V
I_R	Reverse Input Leakage Current		60	μA	V_{CC} = 5.50, V_R = 5.50
V_{OL}	Output Low Voltage $\overline{BUSY}$, $\overline{CBRQ}$ $\overline{AEN}$ $\overline{BPRO}$, $\overline{BREQ}$		 0.45 0.45 0.45	 V V V	 I_{OL} = 20 mA I_{OL} = 16 mA I_{OL} = 10 mA
V_{OH}	Output High Voltage $\overline{BUSY}$, $\overline{CBRQ}$	Open Collector			
	All Other Outputs	2.4		V	I_{OH} = 400 μA
I_{CC}	Power Supply Current		165	mA	
V_{IL}	Input Low Voltage		.8	V	
V_{IH}	Input High Voltage	2.0		V	
Cin Status	Input Capacitance		25	pF	
Cin (Others)	Input Capacitance		12	pF	

A.C. CHARACTERISTICS (V_{CC} = +5V ±10%, T_A = 0°C to 70°C)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Unit	Test Condition
TCLCL	CLK Cycle Period	125		ns	
TCLCH	CLK Low Time	65		ns	
TCHCL	CLK High Time	35		ns	
TSVCH	Status Active Setup	65	TCLCL-10	ns	
TSHCL	Status Inactive Setup	50	TCLCL-10	ns	
THVCH	Status Active Hold	10		ns	
THVCL	Status Inactive Hold	10		ns	
TBYSBL	BUSY↑↓Setup to BCLK↓	20		ns	
TCBSBL	CBRQ↑↓Setup to BCLK↓	20		ns	
TBLBL	BCLK Cycle Time	100		ns	
TBHCL	BCLK High Time	30	.65[TBLBL]	ns	
TCLLL1	LOCK Inactive Hold	10		ns	
TCLLL2	LOCK Active Setup	40		ns	
TPNBL	BPRN↓↑to BCLK Setup Time	15		ns	
TCLSR1	SYSB/RESB Setup	0		ns	
TCLSR2	SYSB/RESB Hold	20		ns	
TIVIH	Initialization Pulse Width	3 TBLBL+ 3 TCLCL		ns	
TILIH	Input Rise Time		20	ns	From 0.8 to 2.0V
TIHIL	Input Fall Time		12	ns	From 2.0V to 0.8V

A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

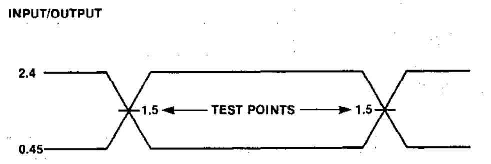
Symbol	Parameter	Min.	Max.	Unit	Test Condition
TBLBRL	BCLK to BREQ Delay↓↑		35	ns	
TBLPOH	BCLK to BPRO↓↑ (See Note 1)		40	ns	
TPNPO	BPRN↓↑to BPRO↓↑Delay (See Note 1)		25	ns	
TBLBYL	BCLK to BUSY Low		60	ns	
TBLBYH	BCLK to BUSY Float (See Note 2)		35	ns	
TCLAEH	CLK to AEN High		65	ns	
TBLAEL	BCLK to AEN Low		40	ns	
TBLCBL	BCLK to CBRQ Low		60	ns	
TRLCRH	BCLK to CBRQ Float (See Note 2)		35	ns	
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

↓↑ Denotes that spec applies to both transitions of the signal.

NOTES:

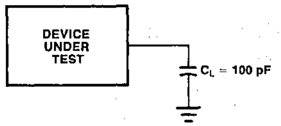
1. BCLK generates the first BPRO wherein subsequent BPRO changes lower in the chain are generated through BPRON.
2. Measured at .5V above GND.

A.C. TESTING INPUT, OUTPUT WAVEFORM



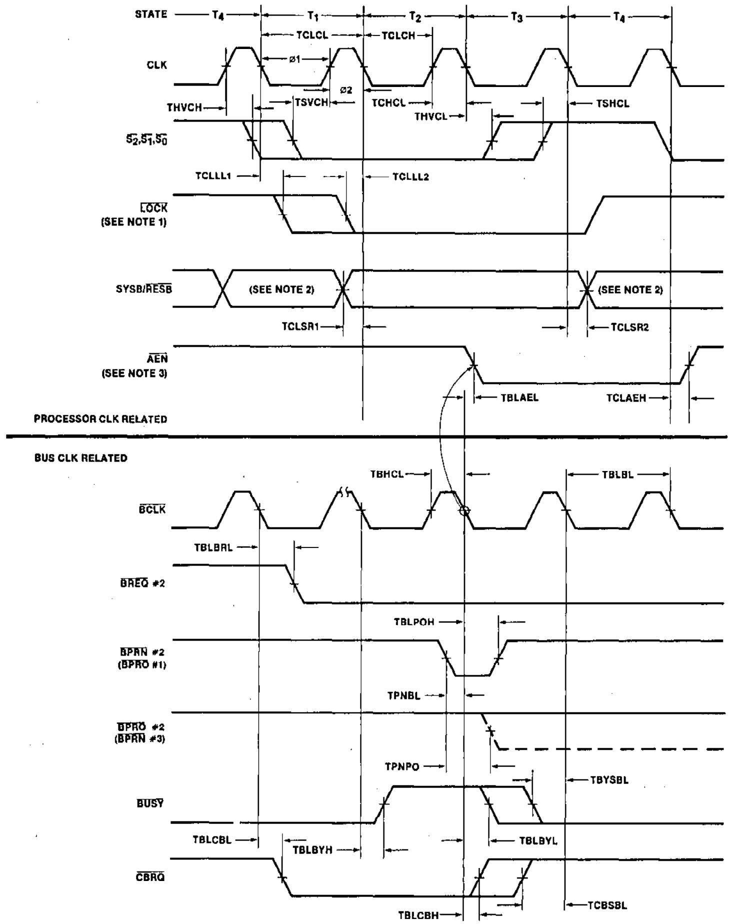
A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." THE CLOCK IS DRIVEN AT 4.3V AND 0.25V. TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

A.C. TESTING LOAD CIRCUIT



C_L = 100 pF
C_L INCLUDES JIG CAPACITANCE

WAVEFORMS



NOTES:

1. LOCK ACTIVE CAN OCCUR DURING ANY STATE, AS LONG AS THE RELATIONSHIPS SHOWN ABOVE WITH RESPECT TO THE CLK ARE MAINTAINED. LOCK INACTIVE HAS NO CRITICAL TIME AND CAN BE ASYNCHRONOUS. -CRQLCK HAS NO CRITICAL TIMING AND IS CONSIDERED AN ASYNCHRONOUS INPUT SIGNAL
2. GLITCHING OF SYSB/RESB PIN IS PERMITTED DURING THIS TIME. AFTER ϕ2 OF T1, AND BEFORE ϕ1 OF T4, SYSB/RESB SHOULD BE STABLE.
3. AEN LEADING EDGE IS RELATED TO BCLK, TRAILING EDGE TO CLK. THE TRAILING EDGE OF AEN OCCURS AFTER BUS PRIORITY IS LOST.

ADDITIONAL NOTES:

The signals related to CLK are typical processor signals, and do not relate to the depicted sequence of events of the signals referenced to BCLK. The signals shown related to the BCLK represent a hypothetical sequence of events for illustration. Assume 3 bus arbiters of priorities 1, 2 and 3 configured in serial priority resolving scheme as shown in Figure 6. Assume arbiter 1 has the bus and is holding busy low. Arbiter #2 detects its processor wants the bus and pulls low BREQ#2. If BPRN#2 is high (as shown), arbiter #2 will pull low CBRQ line. CBRQ signals to the higher priority arbiter #1 that a lower priority arbiter wants the bus. [A higher priority arbiter would be granted BPRN when it makes the bus request rather than having to wait for another arbiter to release the bus through CBRQ].** Arbiter #1 will relinquish the multi-master system bus when it enters a state not requiring it (see Table 1), by lowering its BPRO#1 (tied to BPRN#2) and releasing BUSY. Arbiter #2 now sees that it has priority from BPRN#2 being low and releases CBRQ. As soon as BUSY signifies the bus is available (high), arbiter #2 pulls BUSY low on next falling edge of BCLK. Note that if arbiter #2 didn't want the bus at the time it received priority, it would pass priority to the next lower priority arbiter by lowering its BPRO #2 [TPNPO].

**Note that even a higher priority arbiter which is acquiring the bus through BPRN will momentarily drop CBRQ until it has acquired the bus.

MODEL 230 INTELLEC SERIES II MICROCOMPUTER DEVELOPMENT SYSTEM

Complete microcomputer development center for Intel MCS-86, MCS-80, MCS-85 and MCS-48 microprocessor families

LSI electronics board with CPU, RAM, ROM, I/O, and interrupt circuitry

64K bytes RAM memory

Self-test diagnostic capability

Eight-level nested, maskable priority interrupt system

Built-in interfaces for high speed paper tape reader/punch, printer, and universal PROM programmer

Integral CRT with detachable upper/lower case typewriter-style full ASCII keyboard

Powerful ISIS-II Diskette Operating System software with relocating macroassembler, linker, and locater

1 million bytes (expandable to 2.5M bytes) of diskette storage

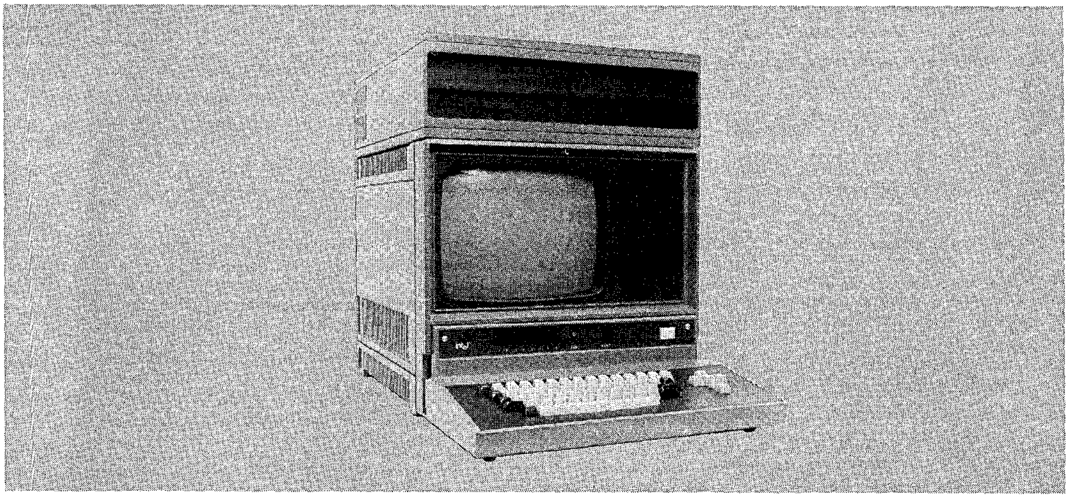
Supports PL/M and FORTRAN high level languages

Standard MULTIBUS with multiprocessor and DMA capability

Compatible with standard Intellec/iSBC expansion modules

Software compatible with previous Intellec systems

The Model 230 Intellec Series II Microcomputer Development System is a complete center for the development of microcomputer-based products. It includes a CPU, 64K bytes of RAM, 4K bytes of ROM memory, a 2000-character CRT, a detachable full ASCII keyboard, and dual double density diskette drives providing over 1 million bytes of on-line data storage. Powerful ISIS-II Diskette Operating System software allows the Model 230 to be used quickly and efficiently for assembling and/or compiling and debugging programs for Intel's MCS-86, MCS-80, MCS-85, or MCS-48 microprocessor families without the need for handling paper tape. ISIS-II performs all file handling operations, leaving the user free to concentrate on the details of his own application. When used in conjunction with an optional in-circuit emulator (ICE) module, the Model 230 provides all the hardware and software development tools necessary for the rapid development of a microcomputer-based product.



FUNCTIONAL DESCRIPTION

Hardware Components

The Intellec Series II Model 230 is a packaged, highly integrated microcomputer development system consisting of a CRT chassis with a 6-slot cardcage, power supply, fans, cables, and five printed circuit cards. A separate, full ASCII keyboard is connected with a cable. A second chassis contains two floppy disk drives capable of double-density operation along with a separate power supply, fans, and cables for connection to the main chassis. A block diagram of the Model 230 is shown in Figure 1.

CPU Cards — The master CPU card contains its own microprocessor, memory, I/O, interrupt and bus interface circuitry fashioned from Intel's high technology LSI components. Known as the integrated processor board (IPB), it occupies the first slot in the cardcage. A second slave CPU card is responsible for all remaining I/O control including the CRT and keyboard interface. This card, mounted on the rear panel, also contains its own microprocessor, RAM and ROM memory, and I/O interface logic, thus, in effect, creating a dual processor environment. Known as the I/O controller (IOC), the slave CPU card communicates with the IPB over an 8-bit bidirectional data bus.

Memory and Control Cards — In addition, 32K bytes of RAM (bringing the total to 64K bytes) is located on a separate card in the main cardcage. Fabricated from Intel's 16K RAMs, the board also contains all necessary address decoding and refresh logic. Two additional boards in the cardcage are used to control the two double-density floppy disk drives.

Expansion — Two remaining slots in the cardcage are available for system expansion. Additional expansion of 4 slots can be achieved through the addition of an Intellec Series II expansion chassis.

System Components

The heart of the IPB is an Intel NMOS 8-bit microprocessor, the 8080A-2, running at 2.6 MHz. 32K bytes of RAM memory are provided on the board using Intel 16K RAMs. 4K of ROM is provided, preprogrammed with system bootstrap "self-test" diagnostics and the Intellec Series II System Monitor. The eight-level vectored priority interrupt system allows interrupts to be individually masked. Using Intel's versatile 8259A interrupt controller, the interrupt system may be user programmed to respond to individual needs.

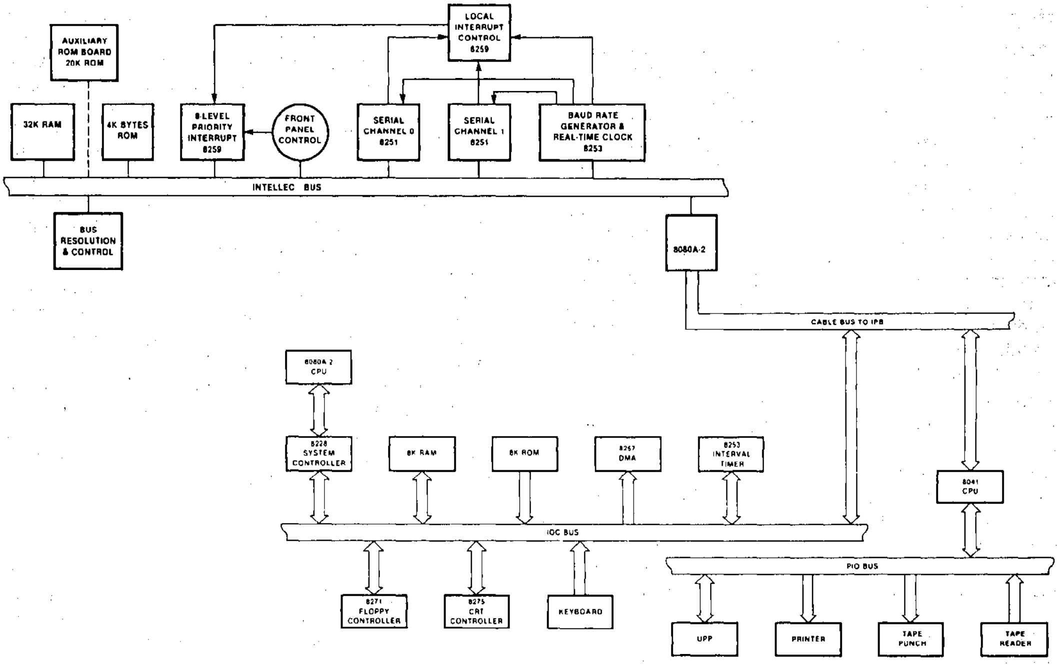


Figure 1. Intellec Series II Model 230 Microcomputer Development System Block Diagram

Input/Output

IPB Serial Channels — The I/O subsystem in the Model 230 consists of two parts: the IOC card and two serial channels on the IPB itself. Each serial channel is RS232 compatible and is capable of running asynchronously from 110 to 9600 baud or synchronously from 150 to 56K baud. Both may be connected to a user defined data set or terminal. One channel contains current loop adapters. Both channels are implemented using Intel's 8251A USART. They can be programmatically selected to perform a variety of I/O functions. Baud rate selection is accomplished programmatically through an Intel 8253 interval timer. The 8253 also serves as a real-time clock for the entire system. I/O activity through both serial channels is signaled to the system through a second 8259 interrupt controller, operating in a polled mode nested to the primary 8259.

IOC Interface — The remainder of system I/O activity takes place in the IOC. The IOC provides interface for the CRT, keyboard, and standard Intellec peripherals including printer, high speed paper tape reader/punch, and universal PROM programmer. The IOC contains its own independent microprocessor, also an 8080A-2. The CPU controls all I/O operations as well as supervising communications with the IPB. 8K bytes of ROM contain all I/O control firmware. 8K bytes of RAM are used for CRT screen refresh storage. These do not occupy space in Intellec Series II main memory since the IOC is a totally independent microcomputer subsystem.

Integral CRT

Display — The CRT is a 12-inch raster scan type monitor with a 50/60 Hz vertical scan rate and 15.5 kHz horizontal scan rate. Controls are provided for brightness and contrast adjustments. The interface to the CRT is provided through an Intel 8275 single chip programmable CRT controller. The master processor on the IPB transfers a character for display to the IOC, where it is stored in RAM. The CRT controller reads a line at a time into its line buffer through an Intel 8257 DMA controller and then feeds one character at a time to the character generator to produce the video signal. Timing for the CRT control is provided by an Intel 8253 interval timer. The screen display is formatted as 25 rows of 80 characters. The full set of ASCII characters are displayed, including lower case alphas.

Keyboard — The keyboard interfaces directly to the IOC processor via an 8-bit data bus. The keyboard contains an Intel UPI-41 Universal Peripheral Interface, which scans the keyboard, encodes the characters, and buffers the characters to provide N-key rollover. The keyboard itself is a high quality typewriter style keyboard containing the full ASCII character set. An upper/lower case switch allows the system to be used for document preparation. Cursor control keys are also provided.

Peripheral Interface

A UPI-41 Universal Peripheral Interface on the IOC board performs similar functions to the UPI-41 on the PIO board in the Model 210. It provides interface for other standard Intellec peripherals including a printer, high speed paper tape reader, high speed paper tape punch, and universal PROM programmer. Communication between the IPB and IOC is maintained over a separate 8-bit bidirectional data bus. Connectors for the four devices named above, as well as the two serial channels, are mounted directly on the IOC itself.

Control

User control is maintained through a front panel, consisting of a power switch and indicator, reset/boot switch, run/halt light, and eight interrupt switches and indicators. The front panel circuit board is attached directly to the IPB, allowing the eight interrupt switches to connect to the primary 8259A, as well as to the Intellec Series II bus.

Diskette System

The Intellec Series II double density diskette system provides direct access bulk storage, intelligent controller, and two diskette drives. Each drive provides ½ million bytes of storage with a data transfer rate of 500,000 bits/second. The controller is implemented with Intel's powerful Series 3000 Bipolar Microcomputer Set. The controller provides an interface to the Intellec Series II system bus, as well as supporting up to four diskette drives. The diskette system records all data in soft sector format. The diskette system is capable of performing seven different operations: recalibrate, seek, format track, write data, write deleted data, read data, and verify CRC.

Diskette Controller Boards — The diskette controller consists of two boards, the channel board and the interface board. These two PC boards reside in the Intellec Series II system chassis and constitute the diskette controller. The channel board receives, decodes and responds to channel commands from the 8080A-2 CPU in the Model 230. The interface board provides the diskette controller with a means of communication with the diskette drives and with the Intellec system bus. The interface board validates data during reads using a cyclic redundancy check (CRC) polynomial and generates CRC data during write operations. When the diskette controller requires access to Intellec system memory, the interface board requests and maintains DMA master control of the system bus, and generates the appropriate memory command. The interface board also acknowledges I/O commands as required by the Intellec bus. In addition to supporting a second set of double density drives, the diskette controller may co-reside with the Intel single density controller to allow up to 2.5 million bytes of on-line storage.

MULTIBUS Capability

All Intellec Series II models implement the industry standard MULTIBUS. MULTIBUS enables several bus masters, such as CPU and DMA devices, to share the bus and memory by operating at different priority levels. Resolution of bus exchanges is synchronized by a bus clock signal derived independently from processor clocks. Read/write transfers may take place at rates up to 5 MHz. The bus structure is suitable for use with any Intel microcomputer family.

SPECIFICATIONS

Host Processor (IPB)

RAM — 64K (system monitor occupies 62K through 64K)

ROM — 4K (2K in monitor, 2K in boot/diagnostic)

Diskette System Capacity (Basic Two Drives)

Unformatted
Per Disk: 6.2 megabits
Per Track: 82.0 kilobits

Formatted
Per Disk: 4.1 megabits
Per Track: 53.2 kilobits

Diskette Performance

Diskette System Transfer Rate — 500 kilobits/sec

Diskette System Access Time
Track-to-Track: 10 ms
Head Settling Time: 10 ms

Average Random Positioning Time — 260 ms

Rotational Speed — 360 rpm
Average Rotational Latency — 83 ms
Recording Mode — M^2FM

Physical Characteristics

Width — 17.37 in. (44.12 cm)
Height — 15.81 in. (40.16 cm)
Depth — 19.13 in. (48.59 cm)
Weight — 73 lb (33 kg)

Keyboard
Width — 17.37 in. (44.12 cm)
Height — 3.0 in. (7.62 cm)
Depth — 9.0 in. (22.86 cm)
Weight — 6 lb (3 kg)

Dual Drive Chassis
Width — 16.88 in. (42.88 cm)
Height — 12.08 in. (30.68 cm)
Depth — 19.0 in. (48.26 cm)
Weight — 64 lb (29 kg)

Electrical Characteristics

DC Power Supply

Volts Supplied	Amps Supplied	Typical System Requirements
+ 5 ± 5%	30	14.25
+ 12 ± 5%	2.5	0.2
− 12 ± 5%	0.3	0.05
− 10 ± 5%	1.5	15
* + 15 ± 5%	1.5	1.3
* + 24 ± 5%	1.7	

*Not available on bus.

AC Requirements — 50/60 Hz, 115/230V AC

Environmental Characteristics

Operating Temperature — 0° to 35°C (95°F)

Equipment Supplied

Model 230 chassis
Integrated processor board (IPB)
I/O controller board (IOC)
32K RAM board
CRT and keyboard
Double density floppy disk controller (2 boards)
Dual drive floppy disk chassis and cables
2 floppy disk drives (512K byte capacity each)
ROM-resident system monitor
ISIS-II system diskette with MCS-80/MCS-85 macroassembler

Reference Manuals

9800558 — A Guide to Microcomputer Development Systems (SUPPLIED)

9800550 — Intellec Series II Installation and Service Guide (SUPPLIED)

9800306 — ISIS-II System User's Guide (SUPPLIED)

9800556 — Intellec Series II Hardware Reference Manual (SUPPLIED)

9800301 — 8080/8085 Assembly Language Programming Manual (SUPPLIED)

9800292 — ISIS-II 8080/8085 Assembler Operator's Manual (SUPPLIED)

9800605 — Intellec Series II Systems Monitor Source Listing (SUPPLIED)

9800554 — Intellec Series II Schematic Drawings (SUPPLIED)

Reference manuals are shipped with each product only if designated SUPPLIED (see above). Manuals may be ordered from any Intel sales representative, distributor office or from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

ORDERING INFORMATION

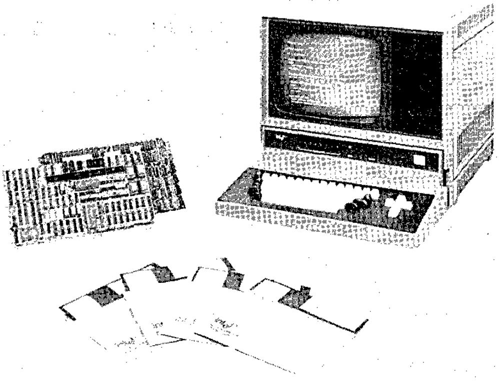
Part Number	Description
MDS-230	Intellec Series II Model 230 microcomputer development system (110V/60 Hz)
MDS-231	Intellec Series II Model 230 microcomputer development system (220V/50 Hz)

MODEL 286
INTELLEC® SERIES III
MICROCOMPUTER DEVELOPMENT SYSTEM

- Supports Intellec 432/100 Evaluation and Educational System
- Compatible with iSBC-090 Series 90 Memory System Upgrade: 512K Byte to 1M Byte
- Complete 16-bit High Performance, Microcomputer Development Solution for Intel iAPX 86,88 Applications. Also Supports MCS-85™, MCS-80 and MCS-48 Families
- Supports Full Range of iAPX 86,88-Resident, High-Level Languages: PL/M 86/88, PASCAL 86/88, and FORTRAN 86/88
- 2 Host CPUs—iAPX 86 and 8085A—for Enhanced System Performance and Two Native Execution Environments
- 96K Bytes of User Program RAM Memory Available for iAPX 86,88 Programs
- Series II/80 and Series II/85 Upgradeable to 8085/iAPX 86 Series III Functionality
- Intellec Model 800 Upgradeable to 8080/iAPX 86 Series III Functionality
- Compatible with Intellec Distributed Development Systems
- Compatible with Previous Intellec Systems
- Software Applications Debugger for User iAPX 86,88 Programs
- Upgradeable to a Complete Ethernet* Communications Development System Environment, Using the Model 677 Upgrade

The Intellec Series-III Microcomputer Development System is a high-performance system solution designed specifically for iAPX 86,88 microprocessor development. It contains two host CPUs, an iAPX 86 and an 8085, that provide two native execution environments for optimum performance and compatibility with the Intellec software packages for both CPUs. The basic system includes 96K bytes of iAPX 86,88 user RAM memory and a 250K byte floppy disk drive. The powerful Disk Operating System maximizes system processing by utilizing the power of both host processors. Standard software includes a full range of iAPX 86,88 resident software. The high-level languages PL/M 86/88, PASCAL 86/88, and FORTRAN 86/88 are also available. A ROM resident software debugger not only provides self-test diagnostic capability, but also gives the user a powerful iAPX 86,88 applications debugger.

*Ethernet is a trademark of Xerox Corporation.



FUNCTIONAL DESCRIPTION

Hardware Components

The Intellec Series III is contained in a single package consisting of a CRT chassis with a 6-slot card cage, power supply, fans, cables, single floppy disk drive, detachable upper/lower case full ASCII keyboard, and four printed circuit cards. A block diagram of the system is shown in Figure 1.

System Components

Two CPU cards reside on the Intellec MULTIBUS bus, each containing its own microprocessor, memory, I/O, interrupt and bus interface circuitry implemented with Intel's high technology LSI components. The integrated processor card (IPC-85), occupies the first slot in the cardcage. A second CPU card, the resident processor board (RPB-86) contains Intel's 16-bit HMOS microprocessor. These CPUs provide the dual processor environment.

A third CPU card performs all remaining I/O including interface to the CRT, integral floppy disk, and keyboard. This card, mounted on the rear panel, contains its own microprocessors, RAM and ROM memory, and I/O interface logic. Known as the I/O controller (IOC), this slave CPU card communicates with the IPC-85 over an 8-bit bidirectional data bus. A 64K byte RAM expansion memory board is also included.

Expansion

Two additional slots in the system cardcage are available for system expansion. The Intellec expansion chassis Model 201 is available to provide 4 additional expansion slots for either memory or I/O expansion.

THE INTELLEC DEVELOPMENT SYSTEM FOR ETHERNET (DS/E)

The Intellec Series III can be expanded to provide the user with the tools necessary to develop and test

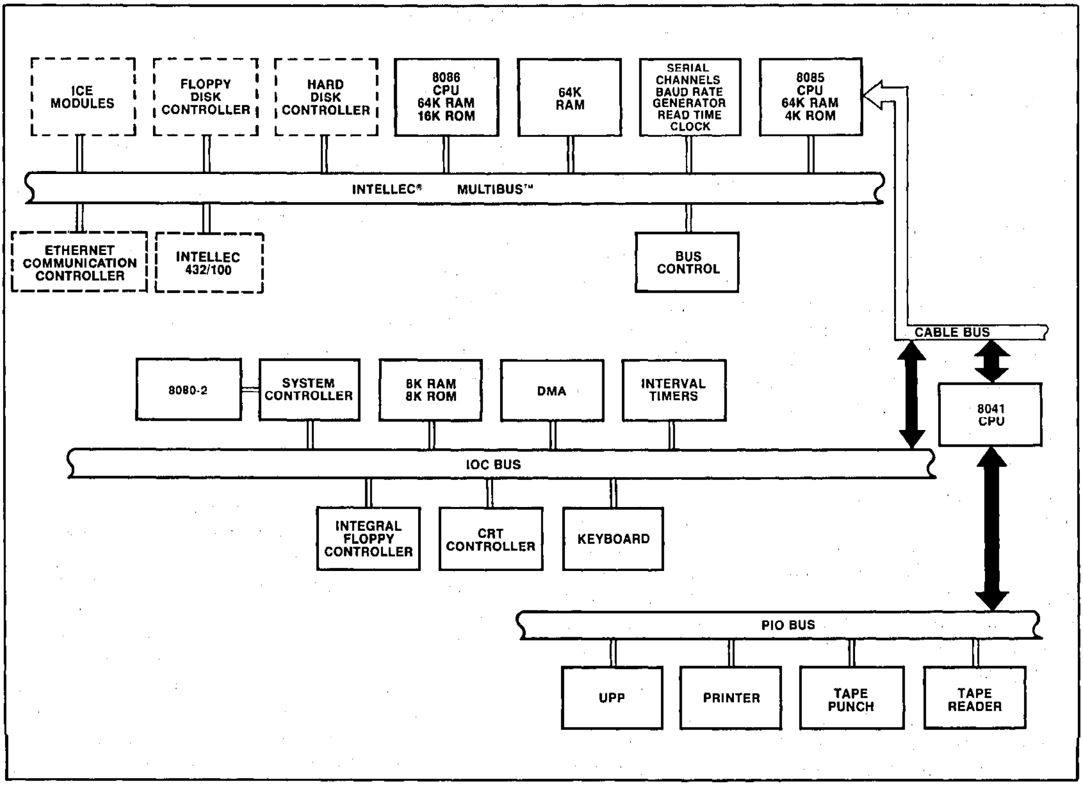


Figure 1. INTELLEC Series III Block Diagram

AFN-01588B

communications software and applications that will use Ethernet as a communications subsystem. The power of the Intellec Series III combined with Model 677 allows the user to develop either 8- or 16-bit Ethernet-based applications.

THE INTELLEC 432/100 EVALUATION AND EDUCATIONAL SYSTEM

The Intellec Series III provides a complete system environment necessary for evaluation of the Intel iAPX 432 32-bit micromainframe. The iSBC 432/100 board plugs into a Multibus slot in the Intellec Series III, sharing system memory and resources. A comprehensive set of documentation, system software and hardware provides the evaluation and educational environment for the powerful iAPX 432.

iAPX 286 Evaluation System

The Intellec Series III provides a complete system environment for evaluation of the iAPX 286 microprocessor's architecture and its instruction set, segmentation timing, memory mapping and protection features. A user can begin the development of complex iAPX 286 programs, systems and operating system nuclei with the Intellec Series III and iAPX 286 evaluation package.

CPU Cards

IPC-85

The heart of the IPC-85 is an Intel NMOS 8-bit microprocessor, the 8085A-2, running at 4.0 MHz. 64K bytes of RAM memory are provided on the board using 16K dynamic RAMs. 4K of ROM is provided, preprogrammed with system bootstrap "self-test" diagnostics and the Intellec System Monitor. The eight-level vectored priority interrupt system allows interrupts to be individually masked. Using Intel's versatile 8259A interrupt controller, the interrupt system may be user programmed to respond to individual needs.

RPB-86

The heart of the RPB-86 is an Intel HMOS 16-bit microprocessor, the iAPX 86 (8086), running at 5.0 MHz. 64K bytes of RAM memory are provided on the board. 16K of ROM is provided on board, preprogrammed with an iAPX 88/86 applications debugger which provides features necessary to debug and execute application software for the iAPX 88/86 microprocessors.

The 8085A-2 and iAPX 86 access two independent memory spaces. This allows the two processors to execute concurrently when an iAPX 88/86 program is run. In this mode, the IPC-85 becomes an intellegent I/O processor board to the RPB-86.

Input/Output

IPC-85 SERIAL CHANNELS

The I/O subsystem in the Series III consists of two parts: the IOC card and two serial channels on the IPC-85 itself. Each serial channel is independently configurable. Both are RS232-compatible and is capable of running asynchronously from 110 to 9600 baud or synchronously from 150 to 56K baud. Both may be connected to a user defined data set or terminal. One channel contains current loop adapters. Both channels are implemented using Intel's 8251A USART. They can be programmed to perform a variety of I/O functions. Baud rate selection is accomplished through an Intel 8253 interval timer. The 8253 also serves as a real-time clock for the entire system. I/O activity through each serial channel is independently signaled to the system through a second 8259A (slave) interrupt controller, operating in a polled mode nested to the master 8259A.

IOC INTERFACE

The remainder of the system I/O activity is handled by the IOC. The IOC provides the interface and control for the keyboard, CRT, integral floppy disk drive, and standard Intellec-compatable peripherals including printer, high speed paper tape reader/punch, and universal PROM programmer. The IOC contains its own independent microprocessor, an 8080A-2. This CPU issues commands, receives status, and controls all I/O operations as well as supervising communications with the IPC-85. The IOC contains interval timers, its own IOC bus system controller, and 8K bytes of ROM for all I/O control firmware. The 8K bytes of RAM are used for CRT screen refresh storage. Neither the ROM nor the RAM occupy space in the Intellec Series III main memory address range because the IOC is a totally independent microcomputer subsystem.

Integral CRT

DISPLAY

The CRT is a 12-inch raster scan type monitor with a 50/60 Hz vertical scan rate and 15.5 kHz horizontal scan rate. Controls are provided for brightness and contrast adjustments. The interface to the CRT is provided through an Intel 8275 single chip programmable CRT controller. The master processor on the IPC-85 transfers a character for display to the IOC, where it is stored in RAM. The CRT controller reads a line at a time into its line buffer through an Intel 8257 DMA Controller. It then feeds one character at a time to the character generator to produce the video signal. Timing for the CRT control is provided by an Intel 8253 programmable interval timer. The screen display is formatted as 25 rows of 80 characters. The full set of ASCII characters are displayed, including lower case alphas.

KEYBOARD

The keyboard interfaces directly to the IOC processor via an 8-bit data bus. The keyboard contains an Intel UPI-41A Universal Peripheral Interface, which scans the keyboard and encodes the characters to provide N-key roll over. The keyboard itself is a typewriter style keyboard containing the full ASCII character set. An upper/lower case switch allows the system to be used for document preparation. Cursor control keys are also provided.

Peripheral Interface

A UPI-41A Universal Peripheral Interface on the IOC board provides built-in interface for standard Intellec-compatable peripherals including a printer, high speed paper tape reader, high speed paper tape punch, and universal PROM programmer. Communication between the IPC-85 and IOC is maintained over a separate 8-bit bidirectional data bus. Connectors for the four devices named above, as well as the two serial channels, are mounted directly on the IOC itself.

Control

User control is maintained through a front panel, consisting of a power switch and indicator, reset/boot switch, run/halt light and eight interrupt switches and LED indicators. The front panel circuit board is attached directly to the IPC-85, allowing the eight interrupt switches to connect the master 8259A, as well as to the Intellec Series III bus.

User program control in the iAPX 88/86 environment of the Intellec Series III is also directed through keyboard control sequences to transfer control to the iAPX 88/86 applications debugger, abort a user program or translator and returning control to the IPC-85.

DISK SYSTEM

Integral Floppy Disk Drive

The integral floppy disk is controlled by an Intel 8271 single chip, programmable floppy disk controller. The disk provides capacity of 250K bytes. It transfers data via an Intel 8257 DMA Controller between an IOC RAM buffer and the diskette. The 8271 handles reading and writing of data, formatting diskettes, and reading status, all upon appropriate commands from the IOC microprocessor.

Dual Drive Floppy Disk System (Option)

The Intellec Series III Double Density Diskette System provides direct access bulk storage, intelligent controller and two diskette drives. Each drive provides 1/2 million bytes of storage with a data transfer of 500,000 bits/second. The controller is implemented with Intel's powerful Series 3000 Bipolar Microcomputer Set and supports up to four diskette drives to allow more than 2 million bytes of on-line storage.

The diskette controller consists of two boards, the channel board and the interface board. These two PC boards reside in the Intellec Series III system chassis. The channel board receives, decodes and responds to channel commands from the 8085A-2 CPU on the IPC-85. The interface board provides the diskette controller with a means of communication with the disk drives and with the Intellec system bus. The interface board also validates data during disk transactions.

An additional cable and connectors are also supplied to optionally convert the integral floppy disk from single density to double density.

Hard Disk System (Option)

The Intellec Series III Hard Disk System provides direct access bulk storage, intelligent controller and a disk drive containing one fixed platter and one removable cartridge. Each provides approximately 3.65 million bytes of storage with a data transfer rate of 2.5 Mbits/second. The controller is implemented with Intel's Series 3000 Bipolar Microcomputer Set. The controller provides an interface to the Intellec Series III system bus, as well as supporting up to 2 disk drives. The disk system records all data in Double Frequency (FM) on 2 surfaces per platter. Each platter can be write protected by a front panel switch.

HARD DISK CONTROLLER BOARDS

The disk controller consists of two boards which reside in the Intellec Series III system chassis. The disk system is capable of performing six operations: recalibrate, seek, format track, write data, read data, and verify CRC. In addition to supporting a second drive, the disk controller may co-exist with the double-density diskette controller to allow up to 17 million bytes of on-line storage.

MULTIBUS Interface Capability

All models of the Intellec Series III implement the industry standard MULTIBUS protocol. The MULTIBUS architecture allows several bus masters, such as CPU and DMA devices, to share the bus and memory by operating at different priority levels. Resolution of bus exchanges is synchronized by a bus clock signal derived independently from processor clocks. Read/write transfers may take place at rates up to 5 MHz. The bus structure is suitable for use with any Intel microcomputer family.

System Software Features

The Model 286 offers many key advantages for iAPX 86,88 applications and Intellec Development Systems: enhanced system performance through a dual host CPU environment, a full spectrum of iAPX 86,88-resident high-level languages, expanded user program space for iAPX 86,88 programs, and a powerful high-level software applications debugger for iAPX 86,88 microprocessor software.

Dual Host CPU—The addition of a 16-bit 8086 to the existing 8-bit host CPU increases iAPX 86,88 compilation speeds and provides for iAPX 86,88 code execution. When the 8086 is executing a program, the 8-bit CPU off-loads all I/O activity and operates as an intelligent I/O controller to double buffer data to and from the 8086. The 8086 also provides an execution vehicle for 8086 and 8088 object code. An added benefit of two host microprocessors is that 8-bit translations and applications are handled by the 8-bit CPU, and 16-bit translations and applications are handled by the 8086. This feature provides complete compatibility for current systems and means that software running on current Intellec Development Systems will run on the new system.

High-Level Languages for iAPX 86,88—The Model 286 allows the current Intellec system user to take advantage of a breadth of new resident iAPX 86,88 high-level languages: PL/M 86/88, PASCAL 86/88, and FORTRAN 86/88. The iAPX 86,88 Resident Macro Assembler and these high-level language compilers execute on the 8086 host CPU, thereby increasing system performance.

Expanded Program Memory—By adding a Model 286 to an existing Intellec Development System, 96K bytes of user program RAM memory are made available for iAPX 86,88 programs. System memory is expandable by adding additional RAM memory modules. This, combined with the two host CPU system architecture, dramatically increases the processing power of the system.

Software Applications Debugger—The RPB-86 contains the applications debugger which allows iAPX 86,88 programs to be developed, tested, and debugged within the Intellec system. The debugger provides a subset of In-Circuit Emulator commands such as symbolic debugging, control structures and compound commands specifically oriented toward software debug needs.

SPECIFICATIONS

Host Processor Boards

INTEGRATED PROCESSOR CARD

—(IPC-85) 8085A-2 based, operating at 4 MHz

—64K RAM, 4K ROM (2K in monitor and 2K in boot/diagnostic)

RESIDENT PROCESSOR BOARD

—(RPB-86) 8086 based, operating at 5 MHz, 64K RAM, 16K ROM (applications debugger)

BUS

—MULTIBUS bus, maximum transfer rate of 5 MHz

DIRECT MEMORY ACCESS

—(DMA) Standard capability on the MULTIBUS bus; implemented for user selected DMA devices through optional DMA module

—Maximum transfer rate of 2 MHz

Integral Floppy Disk

Capacity—250K bytes (formatted)
Transfer Rate—160K bits/sec
Access Time—
Track to Track: 10 ms max.
Average Random Positioning: 260 ns
Rotational Speed: 360 rpm
Average Rotational Latency: 83 ms
Recording Mode: FM

Dual Floppy Disk Option

Capacity—
Per Disk: 4.1 megabits (formatted)
Per Track: 53.2 kilobits (formatted)
Transfer Rate—500 kilobits/sec
Access Time—
Track to Track: 10 ms
Head Setting Time: 10 ms
Average Random Positioning Time—260 ms

Rotational Speed—360 rpm
Average Rotational Latency: 83 ms
Recording Mode: M² FM

Hard Disk Drive Option

Type—5440 top loading cartridge and one fixed platter
Tracks per Inch—200
Mechanical Sectors per Track—12
Recording Technique—double frequency (FM)
Tracks per Surface—400
Density—2,200 bits/inch
Bits per Track—62,500
Recording Surfaces per Platter—2
Capacity—
Per Surface—15M bits
Per Platter—29M bits
Per Drive—59M bits
Per Drive—7.3M bytes (formatted)
Transfer Rate—2.5M bits/sec
Access Time—
Track to Track: 13 ms max
Full Stroke: 100 ms
Rotational Speed: 2,400 rpm

Physical Characteristics

Width—17.37 in. (44.12 cm)
Height—15.81 in. (40.16 cm)
Depth—19.13 in. (48.59 cm)
Weight—81 lb. (37 kg)

KEYBOARD
Width—17.37 in. (44.12 cm)
Height—3.0 in. (7.6 cm)
Depth—9.0 in. (22.86 cm)
Weight—6 lb. (3 kg)

DUAL FLOPPY DRIVE SYSTEM (OPTION)
Width—16.88 in. (42.88 cm)
Height—12.08 in. (30.68 cm)
Depth—1.0 in. (48.26 cm)
Weight—64 lb. (29 kg)

HARD DISK DRIVE SYSTEM (OPTION)
Width—18.5 in. (47.0 cm)
Height—34.0 in. (86.4 cm)
Depth—29.75 in. (75.6 cm)
Weight—202 lb. (92 kg)

ELECTRICAL CHARACTERISTICS

DC Power Supply

Volts Supplied	Amps Supplied	Typical System Requirements
+ 5 ± 5%	30.0	17.0
+12 ± 5%	2.5	1.1
−12 ± 5%	0.3	0.1
−10 ± 5%	1.0	0.08
+15 ± 5%*	1.5	1.5
+24 ± 5%*	1.7	1.7

*Not available on bus

AC Requirements for Mainframe

110V, 60 Hz—5.9 Amp
220V, 50 Hz—3.0 Amp

ENVIRONMENAL CHARACTERISTICS

System Operating Temperature—0° to 35°C (32°F to 95°F)

Humidity—20% to 80%

DOCUMENTATION SUPPLIED

Intellec Series III Microcomputer Development System Product Overview, 121575

A Guide to Intellec Series III Microcomputer Development Systems, 121632-001

Intellec Series III Microcomputer Development System Console Operating Instructions, 121609

Intellec Series III Microcomputer Development System Pocket Reference, 121610

Intellec Series III Microcomputer Development System Programmer's Reference, 121618

iAPX 88/86 Family Utilities User's Guide for 8086-Based Development Systems, 121616

8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems, 121627

8086/8087/8088 Macro Assembly Language Pocket Reference, 9800749

8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems, 121628

Intellec Series III Microcomputer Development System Installation and Checkout Manual, 121612

Intellec Series III Microcomputer Development System Schematic Drawings, 121642

ISIS-II CREDIT (CRT-Based Text Editor) User's Guide, 9800902

ISIS-II CREDIT (CRT-Based Text Editor) Pocket Reference, 9800903

The 8086 Family User's Manual, 9800722

The 8086 Family User's Manual, Numeric Supplement, 121586

For Series III Plus Hard Disk Systems Only:

Model 740 Hard Disk Subsystem Operation and Checkout, 9800943

ORDERING INFORMATION

Part Number	Description
DS286 KIT	Intellec Series III Model 286 Microcomputer Development System (110V/60Hz)
DS287 KIT	Intellec Series III Model 287 Microcomputer Development System (220V/50Hz)
DS286FD KIT	Intellec Series III Model 286 Microcomputer Development System with Dual Double Density Flexible Disk System (110V/60Hz)
DS287FD KIT	Intellec Series III Model 287 Microcomputer Development System with Dual Double Density Flexible Disk System (220V/50Hz)
DS286HD KIT	Intellec Series III Model 286 Microcomputer Development System with Pedestal Mounted Hard Disk. (110V/60Hz)
DS287HD KIT	Intellec Series III Model 287 Microcomputer Development System with Pedestal Mounted Hard Disk. (220V/50Hz)

Requires Software License

PL/M 86/88 SOFTWARE PACKAGE

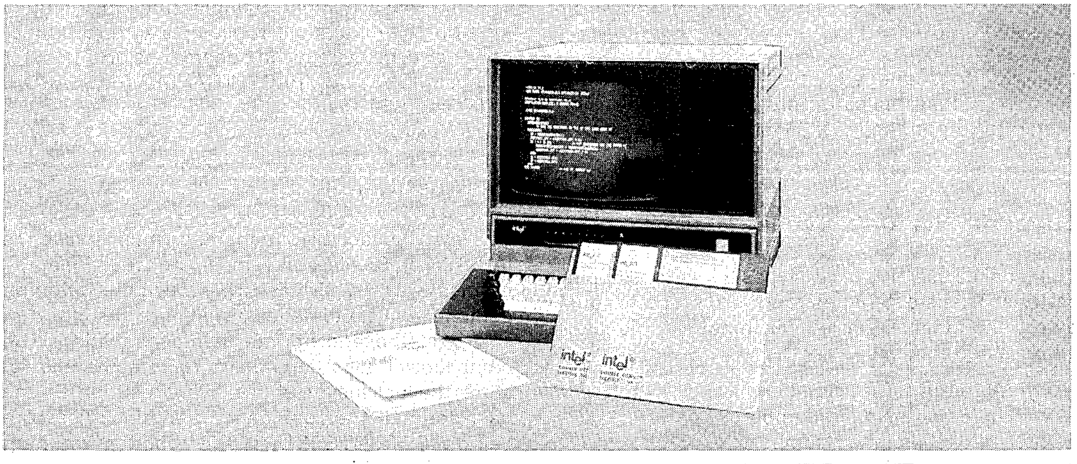
- Executes on Series III iAPX 86 Processor for Fastest Compilations
- Language Is Upward Compatible from PL/M 80, Assuring MCS-80/85 Design Portability
- Supports 16-Bit Signed Integer and 32-Bit Floating Point Arithmetic in Accordance with IEEE Proposed Standard
- Easy-To-Learn Block-Structured Language Encourages Program Modularity
- Improved Compiler Performance Now Supports More User Symbols and Faster Compilation Speeds
- Produces Relocatable Object Code Which Is Linkable to All Other 8086 Object Modules
- Code Optimization Assures Efficient Code Generation and Minimum Application Memory Utilization
- Built-In Syntax Checker Doubles Performance for Compiling Programs Containing Errors

Like its counterpart for MCS-80/85 program development, PL/M 86/88 is an advanced, structured high-level programming language. The PL/M 86/88 compiler was created specifically for performing software development for the Intel 8086 and 8088 Microprocessors.

PL/M is a powerful, structured, high-level system implementation language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the logic of the program without concern for burdensome details of machine or assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M 86/88 compiler efficiently converts free-form PL/M language statements into equivalent 8088/8086 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

The use of PL/M high-level language for system programming, instead of assembly language, results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.



NOTE: The Intellec® Microcomputer Development System pictured here is not included with the PL/M 86/88 Software Package but merely depicts a language in its operating environment.

FEATURES

Major features of the Intel PL/M 86/88 compiler and programming language include:

Block Structure

PL/M source code is developed in a series of modules, procedures, and blocks. Encouraging program modularity in this manner makes programs more readable, and easier to maintain and debug. The language becomes more flexible, by clearly defining the scope of user variables (local to a private procedure, global to a public procedure, for example).

The use of procedures to break down a large problem is paramount to productive software development. The PL/M 86/88 implementation of a block structure allows the use of REENTRANT (recursive) procedures, which are especially useful in system design.

Language Compatibility

PL/M 86/88 object modules are compatible with object modules generated by all other 86/88 translators. This means that PL/M programs may be linked to programs written in any other 86/88 language.

Object modules are compatible with ICE-88 and ICE-86 units; DEBUG compiler control provides the In-Circuit Emulators with symbolic debugging capabilities.

PL/M 86/88 Language is upward-compatible with PL/M 80, so that application programs may be easily ported to run on the iAPX 86 or 88.

Supports Five Data Types

PL/M makes use of five data types for various applications. These data types range from one to four bytes, and facilitate various arithmetic, logic, and addressing functions:

— Byte: 8-bit unsigned number
— Word: 16-bit unsigned number
— Integer: 16-bit signed number
— Real: 32-bit floating point number
— Pointer: 16-bit or 32-bit memory address indicator

Another powerful facility allows the use of BASED variables that map more than one variable to the same memory location. This is especially useful for passing parameters, relative and absolute addressing, and memory allocation.

Two Data Structuring Facilities

In addition to the five data types and based variables, PL/M supports two data structuring facilities. These add flexibility to the referencing of data stored in large groups.

— Array: Indexed list of same type data elements
— Structure: Named collection of same or different type data elements
— Combinations of Each: Arrays of structures or structures of arrays

8087 Numerics Support

PL/M programs that use 32-bit REAL data may be executed using the Numeric Data Processor for improved performance. All floating-point operations supported by PL/M may be executed on the iAPX 86/20 or 88/20 NDP, or the 8087 Emulator (a software module) provided with the package. Determination of use of the chip or Emulator takes place at link-time, allowing compilations to be run-time independent.

Built-In String Handling Facilities

The PL/M 86/88 language contains built-in functions for string manipulation. These byte and word functions perform the following operations on character strings: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET.

Interrupt Handling

PL/M has the facility for generating interrupts to the iAPX 86 or 88 via software. A procedure may be defined with the INTERRUPT attribute, and the compiler will automatically initialize an interrupt vector at the appropriate memory location. The compiler will also generate code to same and restore the processor status, for execution of the user-defined interrupt handler routine. The procedure SET$INTERRUPT, the function retuning an INTERRUPT$PTR, and the PL/M statement CAUSE$INTERRUPT all add flexibility to user programs involving interrupt and handling.

Compiler Controls

Including several that have been mentioned, the PL/M 86/88 compiler offers more than 25 controls that facilitate such features as:

— Conditional compilation
— Including additional PL/M source files from disk
— Intra- and Inter-module cross reference
— Corresponding assembly language code in the listing file
— Setting overflow conditions for run-time handling

Segmentation Control

The PL/M 86/88 compiler takes full advantage of program addressing with the SMALL, COMPACT, MEDIUM, and LARGE segmentation controls. Programs with less than 64KB total code space can exploit the most efficient memory addressing schemes, which lowers total memory requirements. Larger programs can exploit the flexibility of extended one-megabyte addressing.

Code Optimization

The PL/M 86/88 compiler offers four levels of optimization for significantly reducing overall program size.

— Combination or "folding" of constant expressions; and short-circuit evaluation of Boolean expressions.
— "Strength reductions" (such as a shift left rather than multiply by 2); and elimination of common sub-expressions within the same block.
— Machine code optimizations; elimination of superfluous branches; re-use of duplicate code; removal of unreadable code.
— Byte comparisons (rather than 20-bit address calculations) for pointer variables; optimization of based-variable operations.

Error Checking

The PL/M 86/88 compiler has a very powerful feature to speed up compilations. If a syntax or program error is detected, the compiler will skip the code generation and optimization passes. This usually yields a 2X performance increase for compilation of programs with errors.

A fully detailed set of programming and compilation errors is provided by the compiler.

Compiler Performance

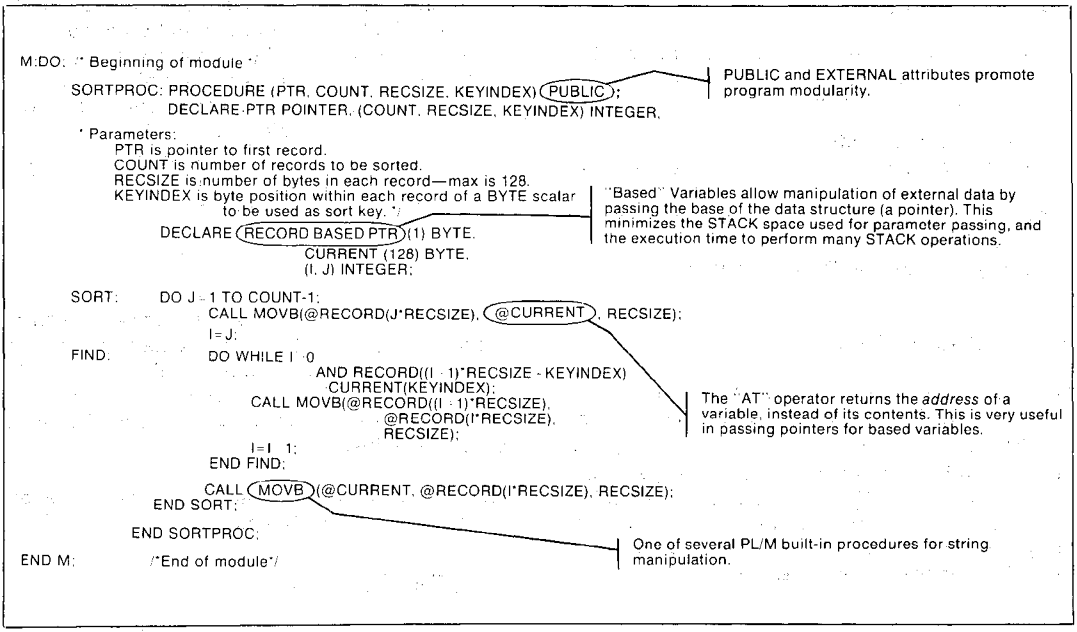
Performance benchmarks may provide valuable information in estimating compile times for various programs. It is extremely important to understand, however, the effect of varying conditions on compiler performance. Storage media, coding style, program length, and the use of INCLUDE files significantly change the compiler's overall performance. We tested typical PL/M programs of varying lengths. The results are listed in Table 1.

Table 1. PL/M Program Compile Times

Program Size	Compile Time(Sec)	Lines/Minute
SMALL (71)	20	213
MEDIUM (610)	54	678
LARGE (1710)	128	802
LARGE (1403) (with very dense code, plus include file)	129	653

NOTE: These programs were run on a Series III with ISIS 4.1 and a hard disk. The lines per minute figures reflect fifteen percent blank lines and comments.

The compiler allows approximately 1000 ten-character user symbols.



```
M:DO: /* Beginning of module */
       SORTPROC: PROCEDURE (PTR, COUNT, RECSIZE, KEYINDEX) PUBLIC;
                 DECLARE PTR POINTER, (COUNT, RECSIZE, KEYINDEX) INTEGER,
        /* Parameters:
           PTR is pointer to first record.
           COUNT is number of records to be sorted.
           RECSIZE is number of bytes in each record—max is 128.
           KEYINDEX is byte position within each record of a BYTE scalar
                    to be used as sort key. */
              DECLARE RECORD BASED PTR (1) BYTE,
                                CURRENT (128) BYTE,
                                (I, J) INTEGER;

       SORT:   DO J = 1 TO COUNT-1;
                   CALL MOVB(@RECORD(J*RECSIZE), @CURRENT, RECSIZE);
                   I=J;
       FIND:       DO WHILE I > 0
                             AND RECORD((I - 1)*RECSIZE - KEYINDEX)
                              >CURRENT(KEYINDEX);
                       CALL MOVB(@RECORD((I - 1)*RECSIZE),
                                 @RECORD(I*RECSIZE),
                                 RECSIZE);
                       I=I - 1;
                   END FIND;
                   CALL MOVB(@CURRENT, @RECORD(I*RECSIZE), RECSIZE);
              END SORT;
           END SORTPROC;
END M;         /*End of module*/
```

PUBLIC and EXTERNAL attributes promote program modularity.

"Based" Variables allow manipulation of external data by passing the base of the data structure (a pointer). This minimizes the STACK space used for parameter passing, and the execution time to perform many STACK operations.

The "AT" operator returns the *address* of a variable, instead of its contents. This is very useful in passing pointers for based variables.

One of several PL/M built-in procedures for string manipulation.

Figure 1. Sample PL/M 86/88 Program

BENEFITS

PL/M 86/88 is designed to be an efficient, cost-effective solution to the special requirements of iAPX 86 or 88 Microsystem Software Development, as illustrated by the following benefits of PL/M use:

Low Learning Effort

PL/M 86/88 is easy to learn and to use, even for the novice programmer.

Earlier Project Completion

Critical projects are completed much earlier than otherwise possible because PL/M 86/88, a structured high-level language, increases programmer productivity.

Lower Development Cost

Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.

Increased Reliability

PL/M 86/88 is designed to aid in the development of reliable software (PL/M 86/88 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.

Easier Enhancements and Maintenance

Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:
Intellec® Microcomputer Development System
— Series III or equivalent
Dual Diskette Drives
— Single- or Double-Density
System Console
— CRT or Hardcopy Interactive Device

OPTIONAL HARDWARE:
Universal PROM Programmer
Line Printer
ICE-86™

REQUIRED SOFTWARE:
ISIS-II Diskette Operating System, V4.1 or later
Series III Operating System

Documentation Package

PL/M-86 User's Guide for 8086-based Development Systems (121636)

ORDERING INFORMATION

Part Number	Description
MDS-313*	PL/M 86/88 Software Package

Requires Software License

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

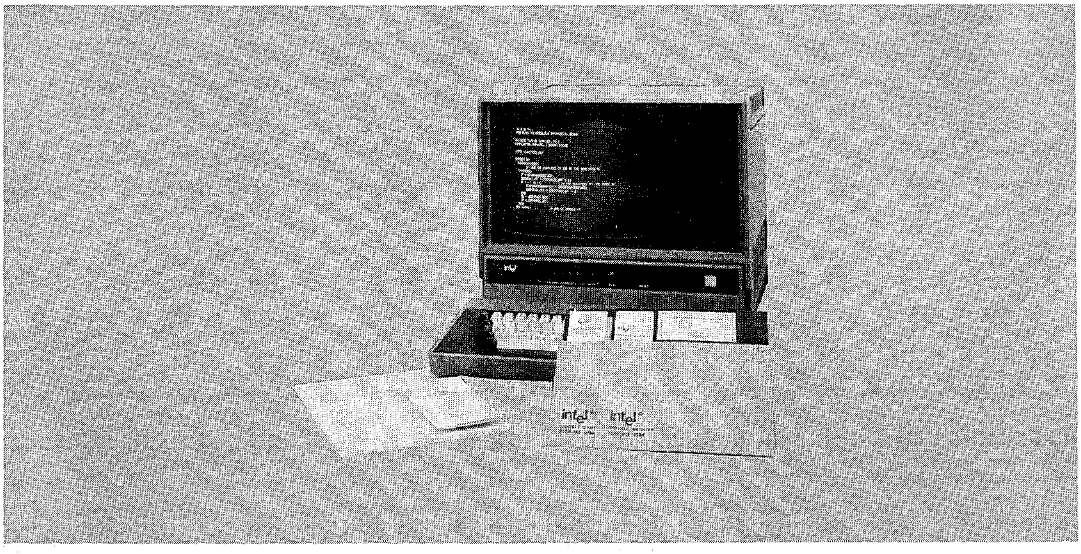
FORTRAN 86/88 SOFTWARE PACKAGE

- **Features high-level language support for floating-point calculations, transcendentals, interrupt procedures, and run-time exception handling**
- **Meets ANS FORTRAN 77 Subset Language Specifications**
- **Supports iAPX 86/20, 88/20 Numeric Data Processor for fast and efficient execution of numeric instructions**
- **Uses REALMATH Floating-Point Standard for consistent and reliable results**
- **Offers powerful extensions tailored to microprocessor applications**
- **Offers upward compatibility with FORTRAN 80**
- **Provides FORTRAN run-time support for iAPX 86,88-based design**
- **Provides users ability to do formatted and unformatted I/O with sequential or direct access methods**

FORTRAN 86/88 meets the ANS FORTRAN 77 Language Subset Specification and includes many features of the full standard. Therefore, the user is assured of portability of most existing ANS FORTRAN programs and of full portability from other computer systems with an ANS FORTRAN 77 Compiler.

FORTRAN 86/88 programs developed and debugged on the iAPX 86 Resident Intellec Series III Microcomputer Development System may be: tested with the prototype using ICE symbolic debugging, and executed on an RMX-86 operating system, or on a user's iAPX 86,88-based operating system.

FORTRAN 86/88 is one of a complete family of compatible programming languages for iAPX 86,88 development: PL/M, Pascal, FORTRAN, and Assembler. Therefore, users may choose the language best suited for a specific problem solution.



NOTE: The Intellec® Microcomputer Development System shown here merely shows the operating environment and is not included with the software package.

FEATURES

Extensive High-Level Language Numeric Processing Support

Single (32-bit), double (64-bit), and double extended precision (80-bit) floating-point data types

REALMATH Proposed IEEE Floating-Point Standard) for consistent and reliable results

Full support for all other data types: integer, logical, character

Ability to use hardware (iAPX 86/20, 88/20 Numeric Data Processor) or software (simulator) floating-point support chosen at link time

ANS FORTRAN 77 Standard

Intel® Microprocessor Support

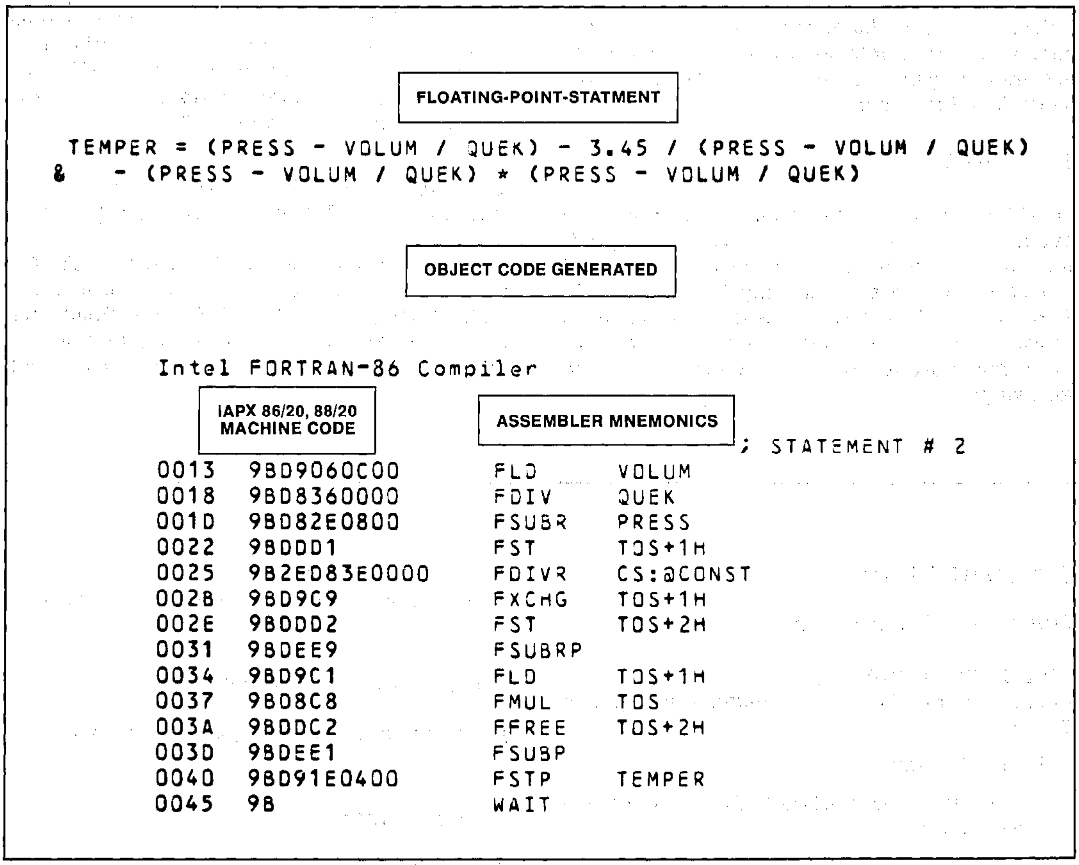
FORTRAN 86/88 language features support of iAPX 86/20, 88/20 Numeric Data Processor

Compiler generates in-line iAPX 86/20, 88/20 Numeric Data Processor object code for floating-point arithmetic (See Figure 1)

Intrinsics allow user to control iAPX 86/20, 88/20 Numeric Data Processor

iAPX 86,88 architectural advantages used for indexing and character-string handling

Symbolic debugging of application using ICE-86 and ICE-88 emulators


FLOATING-POINT-STATMENT
OBJECT CODE GENERATED
IAPX 86/20, 88/20 MACHINE CODE
ASSEMBLER MNEMONICS

```
 TEMPER = (PRESS - VOLUM / QUEK) - 3.45 / (PRESS - VOLUM / QUEK)
&    - (PRESS - VOLUM / QUEK) * (PRESS - VOLUM / QUEK)
```

```
Intel FORTRAN-86 Compiler
                                          ; STATEMENT # 2
0013  9BD9060C00      FLD      VOLUM
0018  9BD8360000      FDIV     QUEK
001D  9BD82E0800      FSUBR    PRESS
0022  9BDDD1          FST      TOS+1H
0025  9B2ED83E0000    FDIVR    CS:@CONST
002B  9BD9C9          FXCHG    TOS+1H
002E  9BDDD2          FST      TOS+2H
0031  9BDEE9          FSUBRP
0034  9BD9C1          FLD      TOS+1H
0037  9BD8C8          FMUL     TOS
003A  9BDDC2          FFREE    TOS+2H
003D  9BDEE1          FSUBP
0040  9BD91E0400      FSTP     TEMPER
0045  9B              WAIT
```

Figure 1. Object Code Generated by FORTRAN 86/88 for a Floating-Point Calculation Using iAPX 86/20, 88/20 Numeric Processor

Microprocessor Application Support

—Direct byte- or word-oriented port I/O

—Reentrant procedures

—Interrupt procedures

Flexible Run-Time Support

Application object code may be executed in iAPX 86, 88-based environment of user's choice:

—a Series III Intellec Development System with Series III Operating System

—an iAPX 86,88-based system with iRMX-86 Operating System

—an iAPX 86,88-based system with user-designed Operating System

Run-time exception handling for fixed-point numerics, floating-point numerics, and I/O errors

Relocatable object libraries for complete run-time support of I/O and arithmetic functions. In-line code execution is generated for iAPX 86/20, 88/20 Numeric Data Processor

BENEFITS

FORTRAN 86/88 provides a means of developing application software for the Intel iAPX 86,88 products lines in a familiar, widely accepted, and industry-standard programming language. FORTRAN 86/88 will greatly enhance the user's ability to provide cost-effective software development for Intel microprocessors as illustrated by the following:

Early Project Completion

FORTRAN is an industry-standard, high-level numerics processing language. FORTRAN programmers can use FORTRAN 86/88 on microprocessor projects with little retraining. Existing FORTRAN software can be compiled with FORTRAN 86/88 and programs developed in FORTRAN 86/88 can run on other computers with ANS FORTRAN 77 with little or no change. Libraries of mathematical programs using ANS 77 standards may be compiled with FORTRAN 86/88.

Application Object Code Portability for a Processor Family

FORTRAN 86/88 modules "talk" to the resident Intellec development operating system using Intel's standard interface for all development-system software. This allows an application developed on the Series III operating system to execute on iRMX/86, or a user-supplied operating system by linking in the iRMX/86 or other appropriate interface library. A standard logical-record interface enables communication with non-standard I/O devices.

Comprehensive, Reliable and Efficient Numeric Processing

The unique combination of FORTRAN 86/88, iAPX 86/20, 88/20 Numeric Data Processor, and REALMATH (Proposed IEEE Floating-Point Standard) provide universal consistency in results of numeric computations and efficient object code generation.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intellec® Series III Microcomputer Development System

—System Console

—Double-Density Dual-Diskette Drive. A Hard Disk is recommended

—Hard Disk*

REQUIRED SOFTWARE

ISIS-II Diskette Operating System V4.1 or later

Documentation Package

FORTRAN 86/88 User's Guide (121539-001)

Shipping Media

Flexible Diskettes

—Single- and Double-Density

*Recommended.

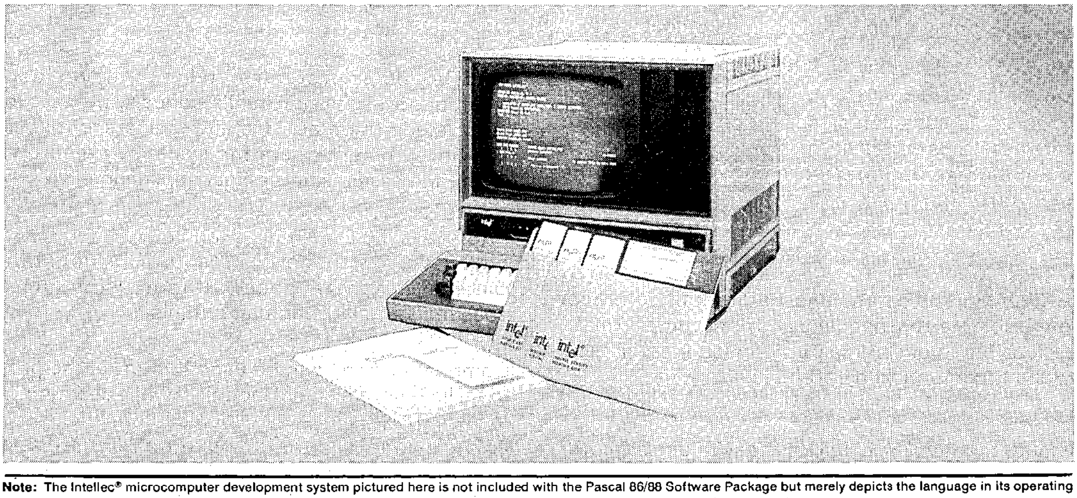
PASCAL 86/88
SOFTWARE PACKAGE

- **Resident on iAPX 86 Based Intellec® Series III Microcomputer Development System for Optimal Performance**
- **Object Compatible and Linkable with PL/M 86/88, ASM 86/88 and FORTRAN 86/88**
- **ICE™ Symbolic Debugging Fully Supported**
- **Implements REALMATH for Consistent and Reliable Results**
- **Supports iAPX86/20, 88/20 Numeric Data Processors**
- **Strict Implementation of ISO Standard Pascal**
- **Useful Extensions Essential for Microcomputer Applications**
- **Separate Compilation with Type-Checking Enforced Between Pascal Modules**
- **Compiler Option to Support Full Run-Time Range-Checking**

PASCAL 86/88 conforms to and implements the ISO Draft Proposed Pascal standard. The language is enhanced to support microcomputer applications with special features, such as separate compilation, interrupt handling and direct port I/O. To assist the development of portable software, the compiler can be directed to flag all non-standard features.

The PASCAL 86/88 compiler runs on the iAPX 86 Resident Intellec® Series III Microcomputer Development System. A well-defined I/O interface is provided for run-time support. This allows a user-written operating system to support application programs as an alternate to the development system environment. Program modules compiled under PASCAL 86/88 are compatible and linkable with modules written in PL/M 86/88, ASM 86/88 or FORTRAN 86/88. With a complete family of compatible programming languages for the iAPX 86, 88 one can implement each module in the language most appropriate to the task at hand.

PASCAL 86/88 object modules contain symbol and type information for program debugging using ICE-86™ emulator. For final production version, the compiler can remove this extra information and code.



Note: The Intellec® microcomputer development system pictured here is not included with the Pascal 86/88 Software Package but merely depicts the language in its operating environment.

FEATURES

Includes all the language features of Jensen & Wirth Pascal as defined in the ISO Draft Proposed Pascal Standard.

Supports required extensions for microcomputer applications.

—Interrupt handling

—Direct port I/O

Separate compilation extensions allow:

—Modular decomposition of large programs

—Linkage with other Pascal modules as well as PL/M 86/88, ASM 86/88 and FORTRAN 86/88.

—Enforcement of type-checking at LINK-time

Supports numerous compiler options to control the compilation process, to INCLUDE files, flag non-standard Pascal statements and others to control program listings and object modules.

Utilizes the IEEE standard for Floating-Point Arithmetic (the Intel REALMATH standard) for arithmetic operations.

Well-defined and documented run-time operating system interfaces allow the user to execute the applications under user-designed operating systems.

BENEFITS

Provides a standard Pascal for iAPX 86, 88 based applications.

—Pascal has gained wide acceptance as the portable application language for microcomputer applications

—It is being taught in many colleges and universities around the world

—It is easy to learn, originally intended as a vehicle for teaching computer programming

—Improves maintainability: Type mechanism is both strictly enforced and user extendable

—Few machine specific language constructs

Strict implementation of the proposed ISO standard for Pascal aids portability of application programs. A compile time option checks conformance to the standard making it easy to write conforming programs.

PASCAL 86/88 extensions via predefined procedures for interrupt handling and direct port I/O make it possible to code an entire application in Pascal without compromising portability.

Standard Intel REALMATH is easy to use and provides reliable results, consistent with other Intel languages and other implementations of the IEEE proposed Floating-Point standard.

Provides run-time support for co-processors. All real-type arithmetic is performed on the 86/20 numeric data processor unit or software emulator. Run-time library routines, common between Pascal and other Intel languages (such as FORTRAN), permit efficient and consistently accurate results.

Extended relocation and linkage support allows the user to link Pascal program modules with routines written in other languages for certain parts of the program. For example, real-time or hardware dependent routines written in ASM 86/88 or PL/M 86/88 can be linked to Pascal routines, further extending the user's ability to write structured and modular programs.

PASCAL 86/88 programs "talk" to the resident operating system using Intel's standard interface for translated programs. This allows users to replace the development operating system by their own operating systems in the final application.

PASCAL 86/88 takes full advantage of iAPX 86, 88 high level language architecture to generate efficient machine code without using time-consuming optimization algorithms.

Compiler options can be used to control the program listings and object modules. While debugging, the user may generate additional information such as the symbol record information required and useful for debugging using ICE emulation. After debugging, the production version may be streamlined by removing this additional information.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intellec® Series III Microcomputer Development System
—System Console
—Double Density Dual Diskette Drive OR Hard Disk

REQUIRED SOFTWARE

ISIS-II Diskette Operating System V4.1 or later

Documentation Package

PASCAL 86 User's Guide (121539-001)

Shipping Media

Flexible Diskettes
—Single and Double Density

ORDERING INFORMATION

Part Number	Description
MDS*-314	PASCAL 86/88 Software Package

Requires software license.

* MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Science.

intel®

8086/8088 SOFTWARE DEVELOPMENT PACKAGE

PL/M-86 high level programming language

ASM86 macro assembler for 8086/8088 assembly language programming

LINK86 and LOC86 linkage and relocation utilities

CONV86 converter for conversion of 8080/8085 assembly language source code to 8086/8088 assembly language· source code

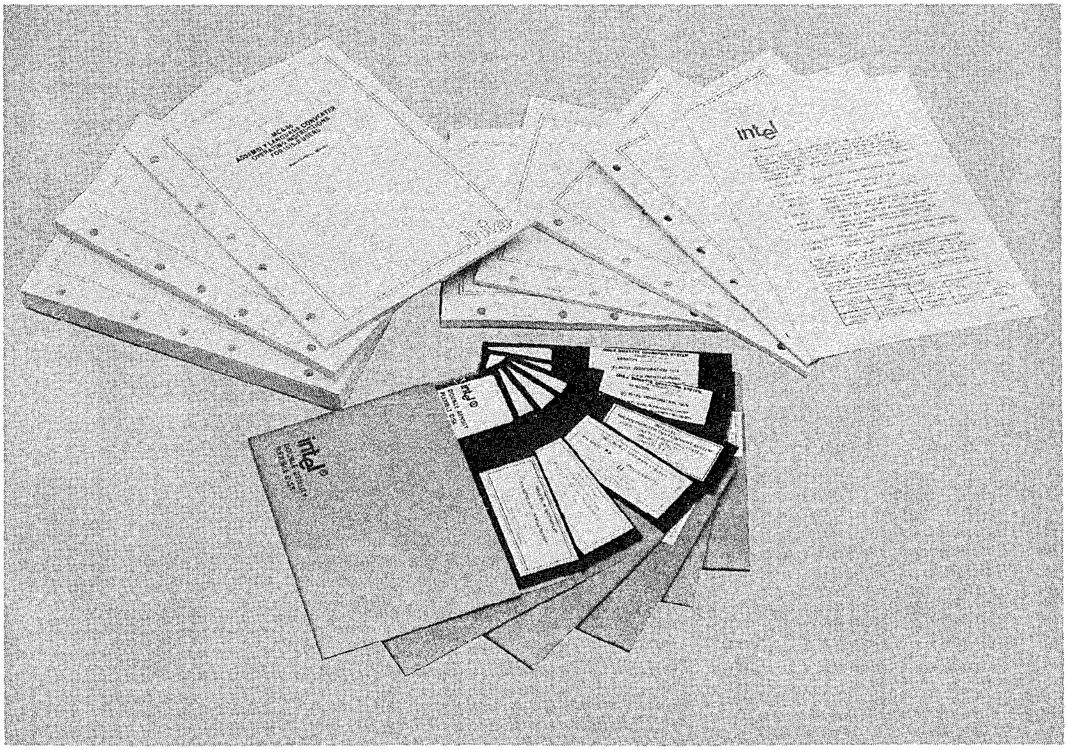
OH86 object-to-hexadecimal converter

LIB86 library manager

The 8086/8088 software development package provides a set of software development tools for the 8086 and the 8088 microprocessors and iSBC 86/12 single board computer. The package operates under the ISIS-II operating system on Intellec Microcomputer Development Systems—Model 800 or Series II—thus minimizing requirements for additional hardware or training for Intel Microcomputer Development System users.

The package permits 8080/8085 users to efficiently convert existing programs into 8086/8088 object code from either 8080/8085 assembly language source code or PL/M-80 source code.

For the new Intel Microcomputer Development System user, the package operating on an Intellec Model 230 Microcomputer Development System provides total 8086/8088 software development capability.



PL/M-86 HIGH LEVEL PROGRAMMING LANGUAGE

Sophisticated new compiler design allows user to achieve maximum benefits of 8086/8088 capabilities

Language is upward compatible from PL/M-80, assuring MCS-80/85 design portability

Supports 16-bit signed integer and 32-bit floating point arithmetic

Produces relocatable and linkable object code

Supports full extended addressing features of the 8086 and the 8088 microprocessors

Code optimization assures efficient code generation and minimum application memory utilization

Like its counterpart for MCS-80/85 program development, PL/M-86 is an advanced structured high level programming language. PL/M-86 is a new compiler created specifically for performing software development for the Intel 8086 and 8088 Microprocessors.

PL/M-86 has significant new capabilities over PL/M-80 that take advantage of the new facilities provided by the 8086 and the 8088 microprocessors, yet the PL/M-86 language remains upward compatible from PL/M-80.

With the exception of interrupts, hardware flags, and time-critical code sequences, PL/M-80 programs may be recompiled under PLM-86 with little or no conversion required. PL/M-86, like PL/M-80, is easy to learn, facilitates rapid program development, and reduces program maintenance costs.

PL/M is a powerful, structured high level algorithmic language in which program statements can naturally express the program algorithm. This frees the programmer to concentrate on the system implementation without concern for burdensome details of assembly language programming (such as register allocation, meanings of assembler mnemonics, etc.).

The PL/M-86 compiler efficiently converts free-form PL/M language statements into equivalent 8086/8088 machine instructions. Substantially fewer PL/M statements are necessary for a given application than if it were programmed at the assembly language or machine code level.

Since PL/M programs are implementation problem oriented and more compact, use of PL/M results in a high degree of engineering productivity during project development. This translates into significant reductions in initial software development and follow-on maintenance costs for the user.

FEATURES

Major features of the Intel PL/M-86 compiler and programming language include:

- **Supports Five Data Types**
  - Byte: 8-bit unsigned number
  - Word: 16-bit unsigned number
  - Integer: 16-bit signed number
  - Real: 32-bit floating point number
  - Pointer: 16-bit or 32-bit memory address indicator
- **Block Structured Language**
  - Permits use of structured programming techniques
- **Two Data Structuring Facilities**
  - Array: Indexed list of same type data elements
  - Structure: Named collection of same or different type data elements
  - Combinations of Each: Arrays of structures or structures of arrays
- **Relocatable and Linkable Object Code**
  - Permits PL/M-86 programs to be developed and debugged in small modules. These modules can be easily linked with other PL/M-86 or ASM86 object modules and/or library routines to form a complete application system.
- **Built-In String Handling Facilities**
  - Operates on byte strings or word strings
  - Six Functions: MOVE, COMPARE, TRANSLATE, SEARCH, SKIP, and SET
- **Automatic Support for 8086 Extended Addressing**
  - Three compiler options offer a separate model of computation for programs up to 1-Megabyte in size
  - Language transparency for extended addressing
- **Support for ICE-86 Emulator and Symbolic Debugging**
  - Debug option for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging

- **Numerous Compiler Options**
  - — A host of 26 compiler options including:
    - Conditional compilation
    - Included file or copy facility
    - Two levels of optimization
    - Intra-module and inter-module cross reference
    - Arbitrary placement of compiler and user files on any available combination of disk drives
- **Reentrant and Interrupt Procedures**
  - — May be specified as user options

BENEFITS

PL/M-86 is designed to be an efficient, cost-effective solution to the special requirements of 8086/8088 Microcomputer Software Development, as illustrated by the following benefits of PL/M-86 use:

- **Reduced Learning Effort** — PL/M-86 is easy to learn and to use, even for the novice programmer.
- **Earlier Project Completion** — Critical projects are completed much earlier than otherwise possible because PL/M-86, a structured high-level language, increases programmer productivity.
- **Lower Development Cost** — Increases in programmer productivity translate immediately into lower software development costs because less programming resources are required for a given programmed function.
- **Increased Reliability** — PL/M-86 is designed to aid in the development of reliable software (PL/M-86 programs are simple statements of the program algorithm). This substantially reduces the risk of costly correction of errors in systems that have already reached full production status, as the more simply stated the program is, the more likely it is to perform its intended function.
- **Easier Enhancements and Maintenance** — Programs written in PL/M tend to be self-documenting, thus easier to read and understand. This means it is easier to enhance and maintain PL/M programs as the system capabilities expand and future products are developed.
- **Simpler Project Development** — The Intellec Development Systems offer a cost-effective hardware base for the development of 8086 and 8088 designs. PL/M-86 and other elements of ISIS-II and the 8086/8088 Software Development Package are all that is needed for development of software for the 8086 and the 8088 microcomputers and iSBC 86/12 single board computer. This further reduces development time and costs because expensive (and remote) time sharing of large computers is not required. Present users of Intel Intellec Development Systems can begin to develop 8086 and 8088 designs without expensive hardware reinvestment or costly retraining.

SAMPLE PROGRAM

```
STATISTICS: DO;

/*The procedure in this module computes the mean and
variance of an array of data, X, of length N + 1, according
to the method of Kahan and Parlett (University of Cali-
fornia, Berkeley, Memo no. UCB/ERL M77/21.*/

STAT: PROCEDURE(X$PTR,N,MEAN$PTR,
      VARIANCE$PTR) PUBLIC;

DECLARE
      (X$PTR,MEAN$PTR,VARIANCE$PTR)
      POINTER,X BASED X$PTR (1) REAL,
      N INTEGER,
      MEAN BASED MEAN$PTR REAL,
      VARIANCE BASED VARIANCE$PTR REAL,
      (M,Q,DIFF) REAL,
      I INTEGER;

M = X(0);
M = 0.0;

DO I = 1 TO N;
    DIFF = X(I) - M;
    M = M + DIFF/FLOAT(I + 1);
    Q = Q + DIFF*DIFF*FLOAT(I)/FLOAT(I + 1);
    END;

    MEAN = M;
    VARIANCE = Q/FLOAT(N);

    END STAT;

    END STATISTICS;
```

ASM86 MACRO ASSEMBLER

Powerful and flexible text macro facility with three macro listing options to aid debugging

Highly mnemonic and compact language, most mnemonics represent several distinct machine instructions

"Strongly typed" assembler helps detect errors at assembly time

High-level data structuring facilities such as "STRUCTUREs" and "RECORDs"

Over 120 detailed and fully documented error messages

Produces relocatable and linkable object code

ASM86 is the "high-level" macro assembler for the 8086/8088 assembly language. ASM86 translates symbolic 8086/8088 assembly language mnemonics into 8086/8088 machine code.

ASM86 should be used where maximum code efficiency and hardware control is needed. The 8086/8088 assembly language includes approximately 100 instruction mnemonics. From these few mnemonics the assembler can generate over 3,800 distinct machine instructions. Therefore, the software development task is simplified, as the programmer need know only 100 mnemonics to generate all possible 8086/8088 machine instructions. ASM86 will generate the shortest machine instruction possible given no forward referencing or given explicit information as to the characteristics of forward referenced symbols.

ASM86 offers many features normally found only in high-level languages. The 8086/8088 assembly language is strongly typed. The assembler performs extensive checks on the usage of variables and labels. The assembler uses the attributes which are derived explicitly when a variable or label is first defined, then makes sure that each use of the symbol in later instructions conforms to the usage defined for that symbol. This means that many programming errors will be detected when the program is assembled, long before it is being debugged on hardware.

FEATURES

Major features of the Intel 8086/8088 assembler and assembly language include:

- **Powerful and Flexible Text Macro Facility**
  - — Macro calls may appear anywhere
  - — Allows user to define the syntax of each macro
  - — Built-in functions
    - conditional assembly (IF-THEN-ELSE, WHILE)
    - repetition (REPEAT)
    - string processing functions (MATCH)
    - support of assembly time I/O to console (IN, OUT)
  - — Three Macro Listing Options include a GEN mode which provides a complete trace of all macro calls and expansions
- **High-Level Data Structuring Capability**
  - — STRUCTURES: Defined to be a template and then used to allocate storage. The familiar dot notation may be used to form instruction addresses with structure fields.
  - — ARRAYS: Indexed list of same type data elements.
  - — RECORDS: Allows bit-templates to be defined and used as instruction operands and/or to allocate storage.
- **Fully Supports 8086/8088 Addressing Modes**
  - — Provides for complex address expressions involving base and indexing registers and (structure) field offsets.
  - — Powerful EQU facility allows complicated expressions to be named and the name can be used as a synonym for the expression throughout the module.
- **Powerful STRING MANIPULATION INSTRUCTIONS**
  - — Permit direct transfers to or from memory or the accumulator.
  - — Can be prefixed with a repeat operator for repetitive execution with a count-down and a condition test.
- **Over 120 Detailed Error Messages**
  - — Appear both in regular list file and error print file.
  - — User documentation fully explains the occurrence of each error and suggests a method to correct it.

- **Generates Relocatable and Linkable Object Code—Fully Compatible with LINK86, LOC86 and LIB86**
  — Permits ASM86 programs to be developed and debugged in small modules. These modules can be easily linked with other ASM86 or PL/M-86 object modules and/or library routines to form a complete application system.

- **Support for ICE-86 Emulation and Symbolic Debugging**
  — Debug options for inclusion of symbol table in object modules for In-Circuit Emulation with symbolic debugging.

BENEFITS

The 8086/8088 macro assembler allows the extensive capabilities of the 8086/8088 to be fully exploited. In any application, time and space critical routines can be effectively written in ASM86. The 8086/8088 assembler outputs relocatable and linkable object modules. These object modules may be easily combined with object modules written in PL/M-86—Intel's structured, high-level programming language. ASM86 compliments PLM-86 as the programmer may choose to write each module in the language most appropriate to the task and then combine the modules into the complete applications program using the 8086/8088 relocation and linkage utilities.

CONV86

MCS-80/85 to MCS-86 ASSEMBLY LANGUAGE CONVERTER UTILITY PROGRAM

Translates 8080/8085 Assembly Language Source Code to 8086/8088 Assembly Language Source Code

Provides a fast and accurate means to convert 8080/8085 programs to the 8086 and the 8088, facilitating program portability

Automatically generates proper ASM-86 directives to set up a "virtual 8080" environment that is compatible with PLM-86

In support of Intel's commitment to software portability, CONV86 is offered as a tool to move 8080/8085 programs to the 8086 and the 8088. A comprehensive manual, "MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users" (9800642), covers the entire conversion process. Detailed methodology of the conversion process is fully described therein.

CONV86 will accept as input an error-free 8080/8085 assembly-language source file and optional controls, and produce as output, optional PRINT and OUTPUT files.

The PRINT file is a formatted copy of the 8080/8085 source and the 8086/8088 source file with embedded caution messages.

The OUTPUT file is an 8086/8088 source file.

CONV86 issues a caution message when it detects a potential problem in the converted 8086/8088 code.

A transliteration of the 8080/8085 programs occurs, with each 8080/8085 construct mapped to its exact 8086/8088 counterpart:

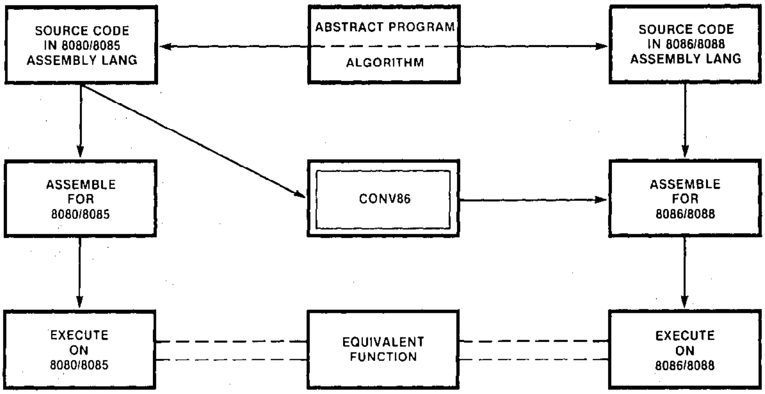
—Registers
—Condition flags
—Instructions
—Operands
—Assembler directives
—Assembler control lines
—Macros

Because CONV86 is a transliteration process, there is the possibility of as much as a 15%-20% code expansion over the 8080/8085 code. For compactness and efficiency it is recommended that critical portions of programs be re-coded in 8086/8088 assembly language.

Also, as a consequence of the transliteration, some manual editing may be required for converting instruction sequences dependent on:

-instruction length, timing, or encoding
-interrupt processing mechanical editing procedures
-PL/M parameter passing conventions for these are suggested in the converter manual.

The accompanying diagram illustrates the flow of the conversion process. Initially, the abstract program may be represented in 8080/8085 or 8086/8088 assembly language to execute on that respective target machine. The conversion process is porting a source destined for the 8080/8085 to the 8086 or the 8088 via CONV86.



PORTING 8080/8085 SOURCE CODE TO THE 8086/8088

LINK86

Automatic combination of separately compiled or assembled 8086/8088 programs into a relocatable module

Automatic selection of required modules from specified libraries to satisfy symbolic references

Extensive debug symbol manipulation, allowing line numbers, local symbols, and public symbols to be purged and listed selectively

Automatic generation of a summary map giving results of the LINK86 process

Abbreviated control syntax

Relocatable modules may be merged into a single module suitable for inclusion in a library

Supports "incremental" linking

Supports type checking of public and external symbols

LINK86 combines object modules specified in the LINK86 input list into a single output module. LINK86 combines segments from the input modules according to the order in which the modules are listed.

Support for incremental linking is provided since an output module produced by LINK86 can be an input to another link. At each stage in the incremental linking process, unneeded public symbols may be purged.

LINK86 supports type checking of public and external symbols reporting an error if their types are not consistent.

LINK86 will link any valid set of input modules without any controls. However, controls are available to control the output of diagnostic information in the LINK86 process and to control the content of the output module.

LINK86 allows the user to create a large program as the combination of several smaller, separately compiled modules. After development and debugging of these component modules the user can link them together, locate them using LOC86, and enter final testing with much of the work accomplished.

LOC86

Automatic and independent relocation of segments. Segments may be relocated to best match users memory configuration

Extensive debug symbol manipulation, allowing line numbers, local symbols, and public symbols to be purged and listed selectively

Automatic generation of a summary map giving starting address, segment addresses and lengths, and debug symbols and their addresses

Extensive capability to manipulate the order and placement of segments in 8086/8088 memory

Abbreviated control syntax

Relocatability allows the programmer to code programs or sections of programs without having to know the final arrangement of the object code in memory.

LOC86 converts relative addresses in an input module to absolute addresses. LOC86 orders the segments in the input module and assigns absolute addresses to the segments. The sequence in which the segments in the input module are assigned absolute addresses is determined by their order in the input module and the controls supplied with the command.

LOC86 will relocate any valid input module without any controls. However, controls are available to control the output of diagnostic information in the LOC86 process, to control the content of the output module, or both.

The program you are developing will almost certainly use some mix of random access memory (RAM), read-only memory (ROM), and/or programmable read-only memory (PROM). Therefore, the location of your program affects both cost and performance in your application. The relocation feature allows you to develop your program on the Intellec development system and then simply relocate the object code to suit your application.

OH86

Converts an 8086/8088 absolute object module to symbolic hexadecimal format

Facilitates preparing a file for later loading by a symbolic hexadecimal loader, such as the iSBC Monitor or Universal PROM Mapper

Converts an absolute module to a more readable format that can be displayed on a CRT or printed for debugging

The OH86 command converts an 8086/8088 absolute object module to the hexadecimal format. This conversion may be necessary to format a module for later loading by a hexadecimal loader such as the iSBC 86/12 monitor or Universal Prom Mapper. The conversion may also be made to put the module in a more readable format that can be displayed or printed.

The module to be converted must be in absolute format; the output from LOC86 is in absolute format.

LIB86

LIB86 is a library manager program which allows you to:

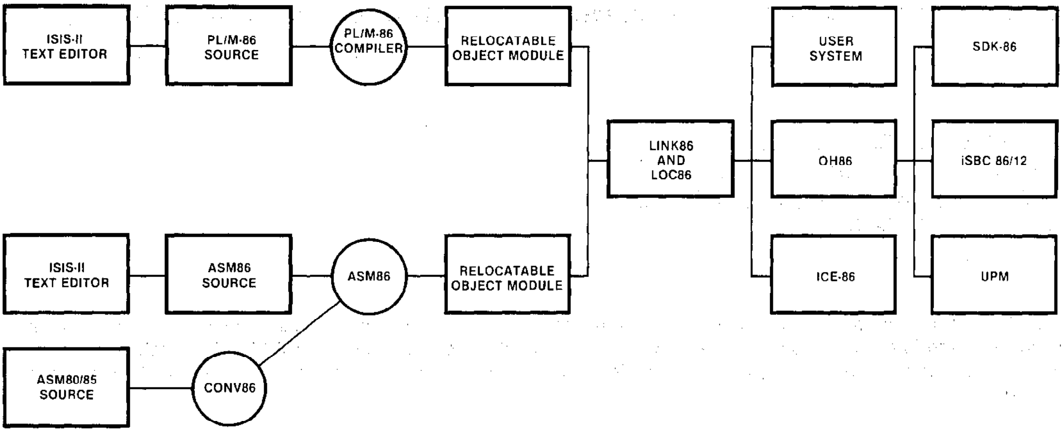
- **Create specially formatted files to contain libraries of object modules**
- **Maintain these libraries by adding or deleting modules**
- **Print a listing of the modules and public symbols in a library file**

Libraries can be used as input to LINK86 which will automatically link modules from the library that satisfy external references in the modules being linked

Abbreviated control syntax

Libraries aid in the job of building programs. The library manager program, LIB86, creates and maintains files containing object modules. The operation of LIB86 is controlled by commands to indicate which operation LIB86 is to perform. The commands are:

CREATE — creates an empty library file
ADD — adds object modules to a library file
DELETE — deletes modules from a library file
LIST — lists the module directory of library files
EXIT — terminates the LIB86 program and returns control to ISIS-II


ISIS-II TEXT EDITOR
PL/M-86 SOURCE
PL/M-86 COMPILER
RELOCATABLE OBJECT MODULE
ISIS-II TEXT EDITOR
ASM86 SOURCE
ASM86
RELOCATABLE OBJECT MODULE
ASM80/85 SOURCE
CONV86
LINK86 AND LOC86
USER SYSTEM
OH86
ICE-86
SDK-86
iSBC 86/12
UPM

SPECIFICATIONS

Operating Environment

Required Hardware

Intellec Microcomputer Development System

- MDS-800, MDS-888
- Series II MDS-220 or MDS-230

64K Bytes of RAM Memory

Dual Diskette Drives

- Single or Double* Density

System Console

- CRT or Hardcopy Interactive Device

Optional Hardware

Universal PROM Programmer
Line Printer*
ICE-86™*

Required Software

ISIS-II Diskette Operating System

- Single or Double* Density

Documentation Package

PL/M-86 Programming Manual (9800466)
ISIS-II PL/M-86 Compiler Operator's Manual (9800478)
MCS-86 User's Manual (9800722)
MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users (9800639)
MCS-86 Macro Assembly Language Reference Manual (9800640)
MCS-86 Macro Assembler Operating Instructions for ISIS-II Users (9800641)
MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users (9800642)
Universal PROM Programmer User's Manual (9800819A)

Flexible Diskettes

- Single and Double* Density

*Recommended

ORDERING INFORMATION

Part Number	Description
MDS-311	8086/8088 Software Development Package

Also available in the following development support packages:

Part Number	Description
SP86A-KIT	SP86A Support Package (for Intellec Model 800) Includes ICE-86 In-Circuit Emulator (MDS-86-ICE) and 8086/8088 Software Development Package (MDS-311)
SP86B-KIT	SP86B Support Package (for Series II) Includes ICE-86 In-Circuit Emulator (MDS-86-ICE), 8086/8088 Software Development Package (MDS-311), and Series II Expansion Chassis (MDS-201)

8087
SOFTWARE SUPPORT PACKAGE

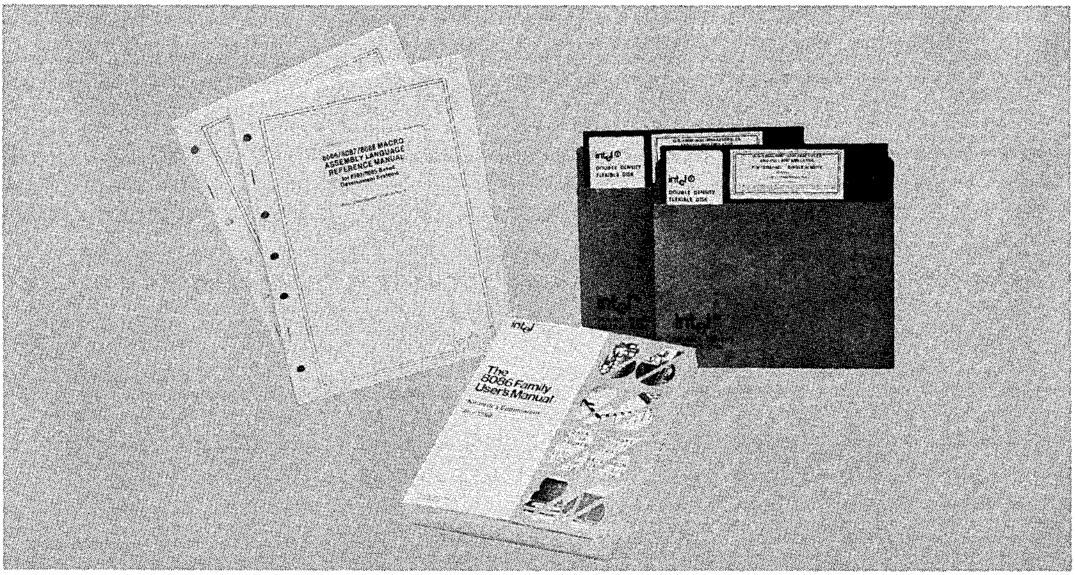
- **Program Generation for the 8087 Numeric Data Processor on the Intellec® Microcomputer Development System**
- **Consists of: 8086/8087/8088 Macro Assembler, 8087 Software Emulator**
- **Macro Assembler Generates Code for 8087 Processor or Emulator, While Also Supporting the 8086/8088 Instruction Set**
- **8087 Emulator Duplicates Each 8087 Floating-Point Instruction in Software, for Evaluation of Prototyping, or for Use in an End Product**
- **Macro Assembler and 8087 Emulator are Fully Compatible with Other 8086/8088 Development Software**
- **Implementation of the IEEE Proposed Floating-Point Standard (the Intel® Realmath Standard)**

The 8087 Software Support Package is an optional extention of Intel's 8086/8088 Software Development Package that runs under ISIS-II on an Intellec or Series II Microcomputer Development System.

The 8087 Software Support Package consists of the 8086/8087/8088 Macro Assembler, and the Full 8087 Emulator. The assembler is a functional superset of the 8086/8088 Macro Assembler, and includes instructions for over sixty new floating-point operations, plus new data types supported by the 8087.

The 8087 Emulator is an 8086/8088 object module that simulates the environment of the 8087, and executes each floating-point operation using software algorithms. This emulator functionally duplicates the operation of the 8087 Numeric Data Processor.

Also included in this package are interface libraries to link with 8086/8087/8088 object modules, which are used for specifying whether the 8087 Processor or the 8087 Emulator is to be used. This enables the run-time environment to be invisible to the programmer at assembly time.


8086/8087/8088 MACRO ASSEMBLY LANGUAGE REFERENCE MANUAL
intel
The 8086 Family User's Manual

FUNCTIONAL DESCRIPTION

8086/8087/8088 Macro Assembler

The 8086/8087/8088 Macro Assembler translates symbolic macro assembly language instructions into appropriate machine instructions. It is an extended version of the 8086/8088 Macro Assembler, and therefore supports all of the same features and functions, such as limited type checking, conditional assembly, data structures, macros, etc. The extensions are the new instructions and data types to support floating-point operations. Realmath floating-point instructions (see Table 1) generate code capable of being converted to either 8087 instructions or interrupts for the 8087 Emulator. The Processor/Emulator selection is made via interface libraries at LINK-time. In addition to the new floating-point instructions, the macro assembler also introduces two new 8087 data types: QWORD (8 bytes) and TBYTE (ten bytes). These support the highest precision of data processed by the 8087.

Full 8087 Emulator

The Full 8087 Emulator is a 16-kilobyte object module that is linked to the application program for floating-point operations. Its functionality is identical to the 8087 chip, and is ideal for prototyping and debugging floating-point applications. The Emulator is an alternative to the use of the 8087 chip, although the latter executes floating-point applications up to 100 times faster than an 8086 with the 8087 Emulator. Furthermore, since the 8087 is a "co-processor," use of the chip will allow many operations to be performed in parallel with the 8086.

Table 1. 8087 Instructions

Arithmetic Instructions

Addition	
FADD FADDP FIADD	Add real Add real and pop Integer add
Subtraction	
FSUB FSUBP FISUB FSUBR FSUBRP FISUBR	Subtract real Subtract real and pop Integer subtract Subtract real reversed Subtract real reversed and pop Integer subtract reversed
Multiplication	
FMUL FMULP FIMUL	Multiply real Multiply real and pop Integer multiply
Division	
FDIV FDIVP FIDIV FDIVR FDIVRP FIDIVR	Divide real Divide real and pop Integer divide Divide real reversed Divide real reversed and pop Integer divide reversed
Other Operations	
FSQRT FSCALE FPREM FRNDINT FXTRACT FABS FCHS	Square root Scale Partial remainder Round to integer Extract exponent and significand Absolute value Change sign

Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Table 1. 8087 Instructions (cont'd)

Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^x-1
FYL2X	$Y \bullet \log_2 X$
FYL2XP1	$Y \bullet \log_2(X+1)$

Constant Instructions

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE

Intellec® Microcomputer Development System
—Model 800
—Series II (Models 220, 225 or equivalent)

64K Bytes of RAM Memory

Minimum One Diskette Drive
—Single or Double* Density

System Console
—CRT or Hardcopy Interactive Device

OPTIONAL HARDWARE

Universal PROM Programmer*
Line Printer*

*Recommended

REQUIRED SOFTWARE

ISIS-II Diskette Operating System
—Single or Double Density

8086/8088 Software Development Package

Documentation Package

8086/8087/8088 Macro Assembly Language Reference Manual for 8080/8085-Based Development Systems (121623-001)

8086/8087/8088 Macro Assembler Operating Instructions for 8080/8085-Based Development Systems (121624-001)

The 8086 Family Users Manual Supplement for the 8087 Numeric Data Processor (121586-001)

Shipping Media

1 Single and 1 Double Density Diskette

ORDERING INFORMATION

Part Number	Description
MDS*-387	8087 Software Support Package

Requires Software License

***MDS** is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

8089 ASSEMBLER SUPPORT PACKAGE

8089 I/O processor program generation on the Intellec Microcomputer Development System.

Relocatable object module compatible with the 8086 and 8088 Microprocessors.

Supports 8089-based addressing modes with a structure facility that enables easy access to based data.

Fully detailed set of error messages.

Includes software development utilities to facilitate 8089 design.

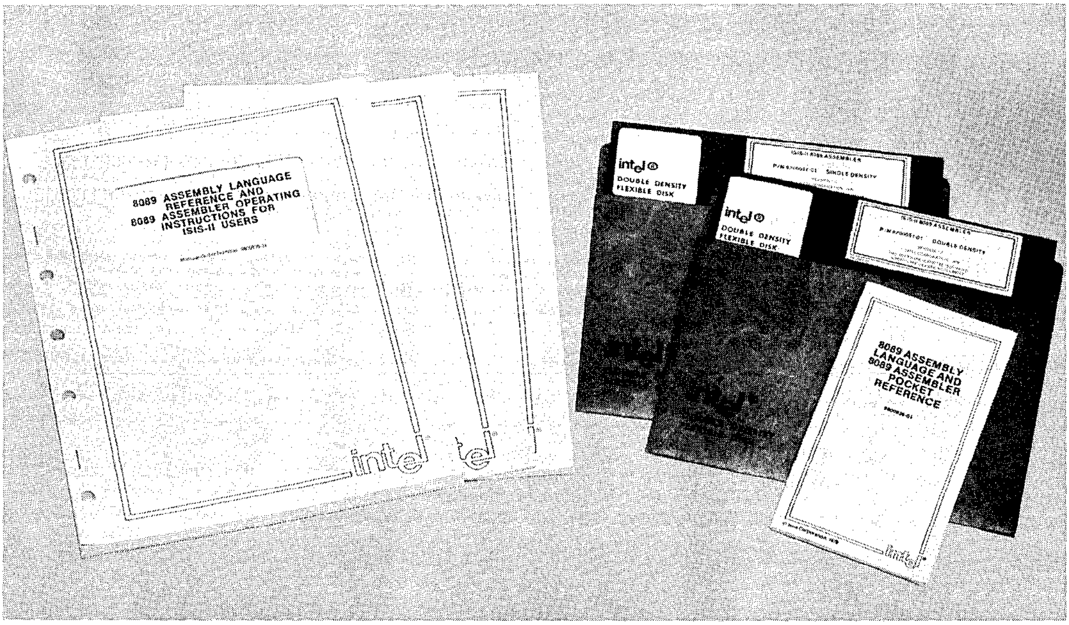
—LINK86: Combines 8086 or 8088 object modules with 8089 object modules and resolves external references.

—LOC86: Assigns absolute memory addresses to 8089 object modules.

—OH86: Converts 8086/8088/8089 object code to symbolic hexadecimal format.

—UPM86: A PROM programming aid which has been updated to support PROM programming for 8086, 8088 and 8089 applications.

The 8089 Assembler Support Package extends Intellec microcomputer development system support to the 8089 I/O Processor. The assembler translates 8089 assembly language source instructions into appropriate machine operation codes. The 8089 Assembler Support Package allows the programmer to fully utilize the capabilities of the 8089 I/O Processor.


8089 ASSEMBLY LANGUAGE REFERENCE
8089 ASSEMBLER OPERATING INSTRUCTIONS FOR ISIS-II USERS
intel
intel DOUBLE DENSITY FLEXIBLE DISK
intel DOUBLE DENSITY FLEXIBLE DISK
SINGLE DENSITY
DOUBLE DENSITY
8089 ASSEMBLY LANGUAGE AND 8089 ASSEMBLER POCKET REFERENCE
intel

8089 ASSEMBLER SUPPORT PACKAGE

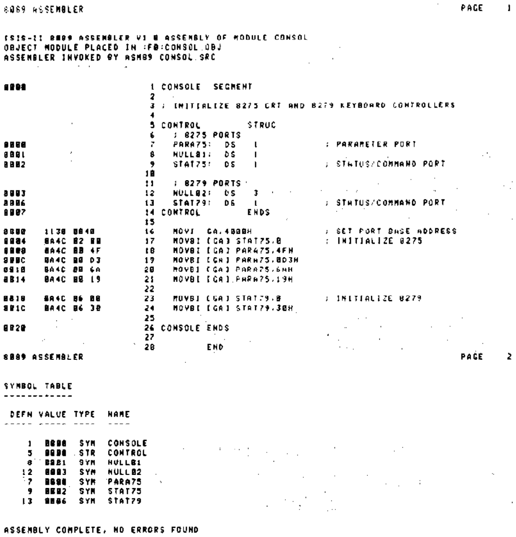
FUNCTIONAL DESCRIPTION

The 8089 Assembler Support Package contains the 8089 assembler (ASM89) as well as LINK86 and LOC86—relocation and linkage utilities, OH86—8086/8088/8089 object code to hexadecimal converter, and UPM86—PROM programming software updated to program object code in the 8086 formats. ASM89 translates symbolic 8089 assembly language instructions into the appropriate machine operation codes. The ability to refer to program addresses with symbolic names eliminates the errors of hand translation and makes it easier to modify programs when adding or deleting instructions.

ASM89 provides relocatable object module compatibility with the 8086 and 8088 microprocessors. This object module compatibility, along with the 8086/8088 relocation and linkage utilities, facilitates the designing of the 8089 into an 8086 or 8088 system.

ASM89 fully supports the based addressing modes of the 8089. A structure facility in the assembler provides easy access to based data. The structure facility allows the user to define a template that enables accessing of based data symbolically.

A sample assembly listing is shown in table 1.



```
8089 ASSEMBLER                                                      PAGE   1

ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE CONSOL
OBJECT MODULE PLACED IN :F0:CONSOL.OBJ
ASSEMBLER INVOKED BY ASM89 CONSOL.SRC

0000                     1 CONSOLE  SEGMENT
                         2
                         3 ; INITIALIZE 8275 CRT AND 8279 KEYBOARD CONTROLLERS
                         4
                         5 CONTROL        STRUC
                         6   ; 8275 PORTS
0000                     7   PARA75:  DS   1             ; PARAMETER PORT
0001                     8   NULL01:  DS   1
0002                     9   STAT75:  DS   1             ; STATUS/COMMAND PORT
                        10
                        11   ; 8279 PORTS
0003                    12   NULL02:  DS   3
0006                    13   STAT79:  DS   1             ; STATUS/COMMAND PORT
0007                    14 CONTROL        ENDS
                        15
0000   1130 0040        16   MOVI  GA,4000H            ; SET PORT BASE ADDRESS
0004   0A4C 02 00       17   MOVBI [GA] STAT75,0       ; INITIALIZE 8275
0008   0A4C 00 4F       18   MOVBI [GA] PARA75,4FH
000C   0A4C 00 D3       19   MOVBI [GA] PARA75,0D3H
0010   0A4C 00 6A       20   MOVBI [GA] PARA75,6AH
0014   0A4C 00 19       21   MOVBI [GA] PARA75,19H
                        22
0018   0A4C 06 00       23   MOVBI [GA] STAT79,0       ; INITIALIZE 8279
001C   0A4C 06 30       24   MOVBI [GA] STAT79,30H
                        25
0020                    26 CONSOLE ENDS
                        27
                        28        END
8089 ASSEMBLER                                                      PAGE   2

SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
 ---- ----- ----  ----

    1  0000  SYM  CONSOLE
    5  0000  STR  CONTROL
    8  0001  SYM  NULL01
   12  0003  SYM  NULL02
    7  0000  SYM  PARA75
    9  0002  SYM  STAT75
   13  0006  SYM  STAT79

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Table 1. Sample 8089 Assembly Listing

SPECIFICATIONS

Operating Environment

Required Hardware

Intellec Microcomputer Development System

—MDS-800, MDS-888

—Series II Models 220 or 230

64K Bytes of RAM Memory

Minimum One Diskette Drive

—Single or Double* Density

System Console

—CRT or Hardcopy Interactive Device

Optional Hardware

Universal PROM Programmer*
Line Printer*

Required Software

ISIS-II Diskette Operating System

—Single or Double* Density

Documentation Package

8089 Assembler User's Guide (9800938)

8089 Assembler Pocket Reference (9800936)

MCS-86 Software Development Utilities
Operating Instructions for ISIS-II User's (9800639)

MCS-86 Absolute Object File Formats (9800821)

Universal PROM Programmer User's Manual (9800819)

Flexible Diskettes

—Single and Double* Density

*Recommended

ORDERING INFORMATION:

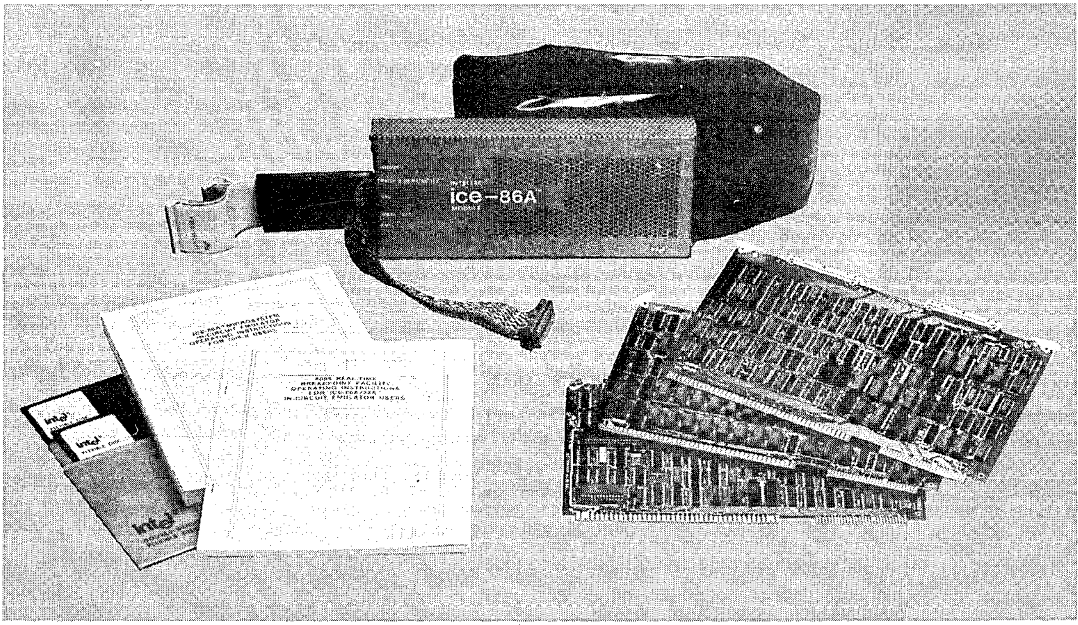
Part Number	Description
MDS-312	8089 Assembler Support Package

ICE-86A™
iAPX 86 IN-CIRCUIT EMULATOR

- **Real-Time In-Circuit Emulation of iAPX 86 Microsystems**
- **Emulate Both Minimum and Maximum Modes of 8086 CPU**
- **Full Symbolic Debugging**
- **Breakpoints to Halt Emulation on a Wide Variety of Conditions**
- **Comprehensive Trace of Program Execution**
- **Disassembly of Trace or Program Memory from Object Code into Assembler Mnemonics**
- **Software Debugging With or Without User System**
- **Handles Full 1 Megabyte Addressability of iAPX 86**
- **Enhance Existing ICE-86™ Emulators to ICE-86A™ Capabilities with ICE-86U™ Upgrade Package**

The Intel® ICE-86A In-Circuit Emulator provides sophisticated hardware and software debugging capabilities for iAPX 86 microsystems and iAPX 86 Single-Board Computers. These capabilities include In-Circuit Emulation for the 8086 Central Processing Unit plus extensions to debug systems including the 8089 I/O Processor and 8087 Numeric Processor Extension. The emulator includes three circuit boards which reside in any Intellec® Microcomputer Development System. A cable and buffer box connect the Intellec system to the user system by replacing the user's 8086, thus extending powerful Intellec system debugging functions into the user system. Using the ICE-86A module, the designer can execute prototype 8086 or 8089 software in continuous or single-step modes and can substitute blocks of Intellec system memory for user equivalents. Breakpoints allow the user to stop emulation on user-specified conditions of the iAPX 86 system, and the trace capability gives a detailed history of the program execution prior to the break. All user access to the prototype system software may be done symbolically by referring to the source program variables and labels.

The ICE-86U In-Circuit Emulator upgrade package converts any existing ICE-86 module (non-A version) to the capabilities of an ICE-86A module.



INTEGRATED HARDWARE/SOFTWARE DEVELOPMENT

The ICE-86A emulator allows hardware and software development to proceed interactively. This is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-86A module, prototype hardware can be added to the system as it is designed. Software and hardware testing occurs while the product is being developed.

Conceptually, the ICE-86A emulator assists three stages of development:

1. It can be operated without being connected to the user's system, so the ICE-86A module's debugging capabilities can be used to facilitate program development before any of the user's hardware is available.

2. Integration of software and hardware can begin when any functional element of the user system hardware is connected to the 8086 socket. Through ICE-86A emulator mapping capabilities, Intellec memory, ICE module memory, or diskette memory can be substituted for missing prototype memory. Time-critical program modules are debugged before hardware implementation by using the 2K-bytes of high-speed ICE-resident memory. As each section of the user's hardware is completed, it is added to the prototype. Thus each section of the hardware and software is "system" tested as it becomes available.

3. When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-86A module is then used for real-time emulation of the 8086 to debug the system as a completed unit.

Thus the ICE-86A module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

SYMBOLIC DEBUGGING

Symbols and PL/M statement numbers may be substituted for numeric values in any of the ICE-86A emulator commands. This allows the user to make symbolic references to I/O ports, memory addresses, and data in the user program. Thus the user need not remember the addresses of variables or program subroutines.

Symbols can be used to reference variables, procedures, program labels, and source statements. A variable can be displayed or changed by referring to it by name rather than by its absolute location in memory. Using symbols for statement labels, program labels, and procedure names simplifies both tracing and breakpoint setting. Disassembly of a section of code from either trace or program memory into its assembly mnemonics is readily accomplished.

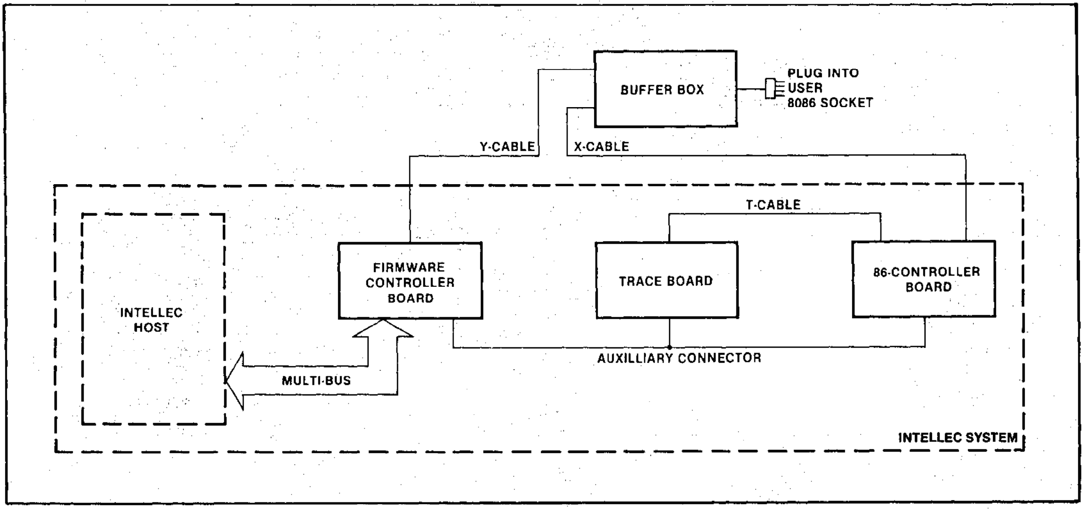
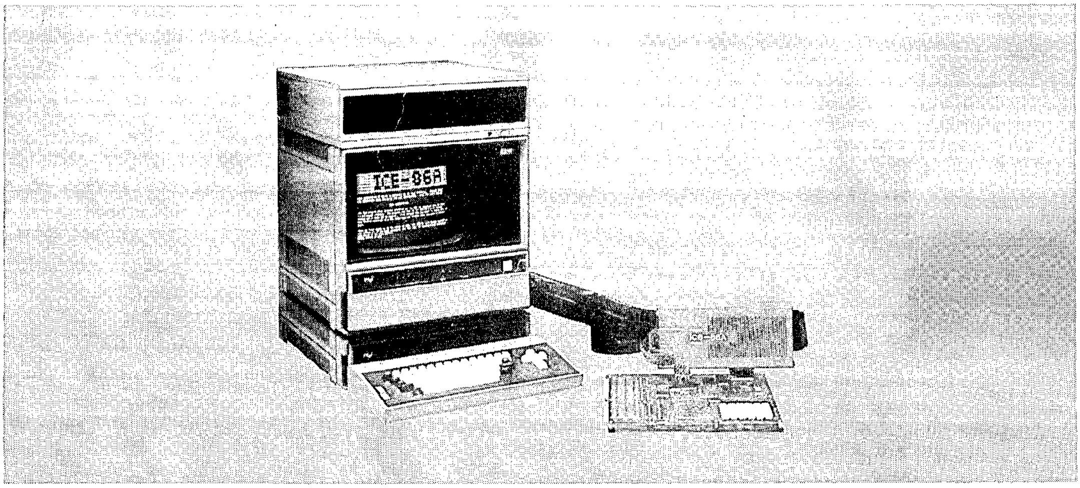


Figure 1. ICE-86A™ Emulator Block Diagram



A typical iAPX 86 development configuration. It is based on Intellec® Series III Development System, which hosts the ICE-86A™ emulator. The ICE-86A™ module is shown connected to a user prototype system, in this case, an SDK-86.

Furthermore, each symbol may have associated with it one of the data types BYTE, WORD, INTEGER, SINTEGER (for short, 8-bit integer), POINTER, REAL, DREAL, or TREAL. Thus the user need not remember the type of a source program variable when examining or modifying it. For example, the command "!VAR" displays the value in memory of variable VAR in a format appropriate to its type, while the command "!VAR = !VAR + 1" increments the value of the variable.

The user symbol table generated along with the object file during a PL/M-86, PASCAL-86 or FORTRAN-86 compilation or an ASM-86 assembly is loaded into memory along with the user program which is to be emulated. The user can utilize the available symbol table space more efficiently by using the SELECT option to choose which program modules will have symbols loaded in the symbol table. The user may also add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found useful during system debugging.

The ICE-86A module provides access through symbolic definition to all of the 8086 registers and flags. The READY, NMI, $\overline{\text{TEST}}$, HOLD, RESET, INTR, MN/$\overline{\text{MX}}$, and $\overline{\text{RQ}}/\overline{\text{GT}}$ pins of the 8086 can also be read. Symbolic references to key ICE-86A emulation information are also provided.

MACROS AND COMPOUND COMMANDS

The ICE-86A module provides a programmable diagnostic facility which allows the user to tailor its operation using macro commands and compound commands.

A macro is a set of ICE-86A commands which is given a single name. Thus, a sequence of commands which is executed frequently may be invoked simply by typing in a single command. The user first defines the macro by entering the entire sequence of commands which he wants to execute. He then names the macro and stores it for future use. He executes the macro by typing its name and passing up to ten parameters to the commands in the macro. Macros may be saved on a disk file for use in subsequent debugging sessions.

Compound commands provide conditional execution of commands (IF), and execution of commands until a condition is met or until they have been executed a specified number of times (COUNT, REPEAT).

Compound commands and macros may be nested any number of times.

MEMORY MAPPING

Memory for the user system can be resident in the user system or "borrowed" from the Intellec System through the ICE-86A emulator's mapping capability. The speed of run emulation by the ICE-86A module depends on which mapping options are being used.

The ICE-86A emulator allows the memory which is addressed by the 8086 to be mapped in 1K-byte blocks to:

1. Physical memory in the user's system, which provides 100 percent real-time emulation at the user-system clock rate (up to 5 MHz) with no wait states.
2. Either of two 1K-byte blocks of ICE-86A module high-speed memory, which allow nearly full-speed emulation (with two additional wait states per 8086-controlled bus cycle).
3. Intellec System memory, which provides emulation at approximately 0.02 percent of real-time with a 5 MHz clock.
4. A random-access diskette file, with emulation speed comparable to Intellec System memory, except the emulation must wait when a new page is accessed on the diskette.

The user can also designate a block of memory as non-existent. The ICE-86A module issues an error message when any such "guarded" memory is addressed by the user program.

As the user prototype progresses to include memory, emulation becomes real time.

OPERATION MODES

The ICE-86A software is a RAM-based program that provides the user with easy-to-use commands for initiating emulation, defining breakpoints, controlling trace data collection, and displaying and controlling system parameters. ICE-86A commands are configured with a broad range of modifiers which provide the user with maximum flexibility in describing the operation to be performed.

Emulation

Emulation commands to the ICE-86A emulator control the process of setting up, running and halting an emulation of the user's iAPX 86 System. Breakpoints and tracepoints enable the ICE-86A module to halt emulation and provide a detailed trace of execution in any part of the user's program. A summary of the emulation commands is shown in Table 1.

Table 1. Summary of ICE-86A™ Emulation Commands

Command	Description
GO	Initializes emulation and allows the user to specify the starting point and breakpoints. Example: GO FROM .START TILL .DELAY EXECUTED where START and DELAY are statement labels.
STEP	Allows the user to single-step through the program.

Breakpoints: The ICE-86A module has two breakpoint registers that allow the user to halt emulation when a specified condition is met. The breakpoint registers may be set up for execution or non-execution breaking. An execution breakpoint consists of a single address which causes a break whenever the 8086 executes from its queue an instruction byte which was obtained from the address. A non-execution breakpoint causes an emulation break when a specified condition other than an instruction execution occurs. A non-execution breakpoint condition, using one or both breakpoint registers, may be specified by any one of or a combination of:

1. *A set of address values.* Break on a set of address values has three valuable features:
   a. Break on a single address.
   b. The ability to set any number of breakpoints within a limited range (1024 bytes maximum) of memory.
   c. The ability to break in an unlimited range. Execution is halted on any memory access to an address greater than (or less than) any 20-bit breakpoint address.
2. *A particular status of the 8086 bus* (one or more of: memory or I/O read or write, instruction fetch, halt, or interrupt acknowledge).
3. *A set of data values* (features comparable to break on a set of address values, explained in point one).
4. *A segment register* (break occurs when the register is used in an effective address calculation).

Emulation break can also be set to occur on an external signal condition. An external breakpoint match output and emulation status lines are provided on the buffer box. These allow synchronization of other test equipment when a break occurs or when emulation is begun.

Tracepoints: The ICE-86A module has two tracepoint registers which establish match conditions to conditionally start and stop trace collection. The trace information is gathered at least twice per bus cycle, first when the address signals are valid and second when the data signals are valid. If the 8086 execution queue is otherwise active, additional frames of trace are collected.

Each trace frame contains the 20 address/data lines and detailed information on the status of the 8086. The trace memory can store 1,023 frames, or an average of about 300 bus cycles, providing ample data for detemining how the 8086 was reacting prior to emulation break. The trace memory contains the last 1,023 frames of trace data collected, even if this spans several separate emulations. The user has the option of displaying each frame of the trace data or displaying by instruction in actual ASM-86 Assembler mnemonics. Unless the user chooses to disable trace, the trace information is always available after an emulation.

Interrogation and Utility

Interrogation and utility commands give the user convenient access to detailed information about the user program and the state of the 8086 that is useful in debugging hardware and software. Changes can be made in both memory and the 8086 registers, flags, input pins, and I/O ports. Commands are also provided for various utility operations such as loading and saving program files, defining symbols and macros, displaying trace data, setting up the memory map, and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in Table 2.

Table 2. Selected ICE-86A^{TM} Module Interrogation and Utility Commands

Memory/Register Commands

Display or change the contents of:

- Memory
- 8086 Registers
- 8086 Status flags
- 8086 Input pins
- 8086 I/O ports
- ICE-86A Pseudo-Registers (e.g. emulation timer)

Memory Mapping Commands

Display, declare, set, or reset the ICE-86A memory mapping.

Symbol Manipulation Commands

Display any or all symbols, program modules, and program line numbers and their associated values (locations in memory).

Set the domain (choose the particular program module) for the line numbers.

Define new symbols as they are needed in debugging.

Remove any or all symbols, modules, and program statements.

Change the value of any symbol.

Select program modules whose symbols will be used in debugging.

TYPE

Assign or change the type of any symbol in the symbol table.

DASM

Disassemble user program memory into ASM-86 Assembler mnemonics.

RQ/GT

Set or display the status of the Request/Grant facility which enables the ICE-86A module to share the system bus with coprocessors.

BUS

Display which device in the user's iAPX 86 system is currently master of the system bus.

CAUSE

Display the cause of the most recent emulation break.

PRINT

Display the specified portion of the trace memory.

LOAD

Fetch user symbol table and object code from the input file.

EVALUATE

Display the value of an expression in binary, octal, decimal, hexadecimal, and ASCII.

CLOCK

Select the internal (ICE-86A module provided, for stand-alone mode only) or an external (user-provided) system clock.

RWTIMEOUT

Allows the user to time out READ/WRITE command signals based on the time taken by the 8086 to access Intellec memory or diskette memory.

ENABLE/DISABLE RDY

Enable or disable logical AND of ICE-86A emulator Ready with the user Ready signal for accessing Intellec memory, ICE memory, or diskette memory.

iAPX 86/20 DEBUGGING

The ICE-86A module has the extended capabilities to debug iAPX 86/20 microsystems which contain both the 8086 microprocessor and the 8087 Numeric Processor Extension (NPX). An iAPX 86/20 system is configured in the 8086's "maximum" mode and communication between the processors is accomplished through the $\overline{\text{RQ}}/\overline{\text{GT}}$ signals. Debugging can be done either using the 8087 chip itself (in which case the 8086 ESCAPE instruction is interpreted as a floating point instruction) or using the 8087 software emulator E8087 (where the 8086 INTERRUPT instruction is interpreted as a floating point instruction). Three new data types are defined to use the NPX:

REAL (4 byte Short Real)
DREAL (8 byte Long Real)
TREAL (10 byte Temporary Real)

While the 8087 NPX is not a programmable part, it does interact closely with the 8086 and can execute instructions in parallel with it. The ICE-86A module provides information about the relative timing of instruction execution in each processor so that the complete system can be debugged. Other debugging capabilities available through the ICE-86A module are: symbolically disassemble NPX call instructions from memory or trace history; display or change the control, status and flag values of the NPX; display the NPX stack either in hexadecimal or disassembled form; and display the last instruction address, last operand, and last operand address.

iAPX 86/11 DEBUGGING

The 8089 Real-Time Breakpoint Facility (RBF-89) is an extension of the ICE-86A emulator that aids in testing and trouble-shooting iAPX 86/11 systems designed around a combination of the 8086 CPU and the 8089 Input/Output Processor (IOP). RBF-89 interrogates 8089 registers, sets breakpoints in 8089 programs, and performs its other functions by preparing special control blocks in application system memory. It then issues input/output channel-attention commands to the 8089 in the user's system to perform these functions. While using the RBF-89 extension, the user can also enter and execute the other standard ICE-86A emulator commands.

RBF-89 allows the user to load his application (channel) program from diskette into 8089 IOP memory and execute it in real time. The program can reside in either local (system) RAM (accessible by both the 8086 and 8089 microprocessors), or remote RAM (accessible by the 8089 IOP only). The user may request execution to begin at any location and continue until normal termination, a specified breakpoint is reached, or the program is otherwise aborted. If a program is modified during a debugging session, RBF-89 can save the latest version by copying it from application system memory to a diskette file.

Breakpoints

RBF-89 supports setting up to twelve breakpoints (six per 8089 channel) in the user program. RBF-89 implements each breakpoint by inserting a HALT instruction at the breakpoint location, while saving the overwritten instruction in temporary storage. When a breakpoint is reached during program execution the program halts. At this point the user can examine 8089 registers, flags, and memory, and optionally resume program execution. The invoked breakpoint address is recorded in one of two breakpoint registers—one register for each 8089 channel. Through simple RBF-89 commands the user can display or change the contents of these registers.

Symbolic Debugging

As in the ICE-86A emulator, the RBF-89 extension accepts symbolic references for variables and labels, including symbols in the symbol table generated by the ASM-89 assembler.

Through RBF-89, the user can display and change the contents of :

— memory, which can be displayed as either numeric data or disassembled (8089 assembly-language mnemonic) code.

— all 8089 registers except the channel control pointer (CCP) and status flags.

Multiprocessor Operation

The ICE-86A emulator and RBF-89 support 8089 configurations in both local and remote modes. The ICE-86A emulator may be operating either in minimum or maximum mode. In maximum mode, the 8086 $\overline{\text{RQ}}/\overline{\text{GT}}$ lines are employed. This is required for the 8089 local mode configuration to provide local bus arbitration between the two processors. Using RBF-89, the user can:

Set $\overline{RQ}/\overline{GT}$ to operate for a local or remote configuration.

Display status to determine which processor controls the system bus.

Start and halt 8089 channel programs.

RBF-89 permits the 8089 and emulated 8086 to run simultaneously as well as sequentially. The user can specify breakpoints and begin program execution in three operating sequences:

Set breakpoints, start the 8089, and return control to the console until a breakpoint is reached or the program runs to completion or is aborted. Use this sequence when the 8086 and 8089 programs do not need to be executed simultaneously.

Set breakpoints, start the 8089, return control to the console, and start the 8086. This sequence lets both microprocessors run simultaneously.

Set breakpoints, start the 8086, and allow that program to drive the 8089 program in a master/slave relationship. This sequence would be used, for instance, to verify the 8086 communication driver program.

RBF-89 System Components

RBF-89 is furnished as a superset of the ICE-86A emulator software. Its main components are:

A HOST PROGRAM that resides in Intellec development system RAM, where it serves as an extension of the ICE-86A emulator's software driver. This program, executed by the development system, translates the user's keyboard input into low-level directives that can be processed by the RBF-89 control program (described below), and converts information supplied by the control program into easily understood display output.

A CONTROL PROGRAM that resides in ICE-86A emulator memory. Running on the emulator's 8086 microprocessor, the control program monitors such operations as preparing program control blocks for communication with the 8089 microprocessor; issuing commands to the 8089 to start, terminate, and continue the 8089 task program; and directing the 8089 to start execution of the RBF-89 utility program (described below).

A UTILITY PROGRAM that resides in the 8089 RAM in the user's prototype application system. This program, running on the 8089, reads and writes data to and from 8089 memory and registers, and sets and removes breakpoints in the user's task program.

The 200 bytes of RAM required by the utility program must be accessible to both the ICE-86A emulator and the 8089.

DC CHARACTERISTICS OF THE ICE-86A™ MODULE USER CABLE

1. Output Low Voltages [V_{OL}**(Max)=0.4V**]

	I_{OL} (Min)
AD0-AD15	12 mA (24 mA @ 0.5V)
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, $\overline{S0}$, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/$\overline{IO}$, DT/$\overline{R}$, $\overline{DEN}$, ALE, $\overline{INTA}$	8 mA (16 mA @ 0.5V)
HLDA	7 mA
$\overline{RQ}/\overline{GT}$	16 mA

2. Output High Voltages [V_{OH} **(Min)=2.4V**]

	I_{OH} (Min)
AD0-AD15	−3 mA
A16/S3-A19/S7, $\overline{BHE}$/S7, $\overline{RD}$, $\overline{LOCK}$, QS0, QS1, S0, $\overline{S1}$, $\overline{S2}$, $\overline{WR}$, M/IO, DT/R, $\overline{DEN}$, ALE, $\overline{INTA}$, HLDA	−2.6 mA
$\overline{RQ}/\overline{GT}$	250 mA

3. Input Low Voltages [V_{IL} **(Max)=0.8V**]

	I_{IL} (Max)
AD0-AD15	−0.2 mA
NMI, CLK	−0.4 mA
READY	−0.8 mA
INTR, HOLD, $\overline{TEST}$, RESET	−1.4 mA
MN/$\overline{MX}$ (0.1 μf to GND)	−3.3 mA

4. Input High Voltages [V_{IH} **(Min)=2.0V**]

	I_{IH} (Max)
AD0-AD15	80 μA
NMI, CLK	20 μA
READY	40 μA
INTR, HOLD, $\overline{TEST}$, RESET	−0.4 mA
MN/$\overline{MX}$ (0.1 μF to GND)	−1.1 mA

5. No current is taken from the user circuit at V_{CC} pin.

SPECIFICATIONS

ICE-86A Operating Environment

REQUIRED HARDWARE

Intellec microcomputer development system with:

1. Three adjacent slots for the ICE-86A module.
2. 64K bytes of Intellec memory. If user prototype program memory is desired, additional memory above the basic 64K is required.

System console
Intellec diskette operating system
ICE-86A module

REQUIRED SOFTWARE

System Monitor
ISIS-II, version 3.4 or subsequent
ICE-86A software

Equipment Supplied

Printed circuit boards (3)
Interface cable and emulation buffer module
Operator's manual
ICE-86A software, diskette-based

Emulation Clock

User system clock up to 5 MHz or 2 MHz ICE-86A internal clock in stand-alone mode

Physical Characteristics

PRINTED CIRCUIT BOARDS

Width: 12.00 in (30.48 cm)
Height: 6.75 in (17.15 cm)
Depth: 0.50 in (1.27 cm)
Packaged Weight: 9.00 lb (4.10 kg)

Electrical Characteristics

DC POWER

V_{CC} = +5V +5% −1%
I_{CC} = 17A maximum; 11A typical
V_{DD} = +12V ±5%
I_{DD} = 120 mA maximum; 80 mA typical
V_{BB} = −10V ±5% or −12V ±5% (optional)
I_{BB} = 25 mA maximum; 12 mA typical

Environmental Characteristics

OPERATING TEMPERATURE

0° to 40°C

OPERATING HUMIDITY

Up to 95% relative humidity without condensation.

ORDERING INFORMATION

Part Number	Description
MDS*-86A-ICE	iAPX 86 microsystem in-circuit emulator, cable assembly, and interactive software
MDS*-86U-ICE	Upgrade kit to convert ICE-86 emulators to ICE-86A emulator capabilities.

*MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

PRELIMINARY

ICE 86A™, ICE 88A™
iAPX 86, 88 IN-CIRCUIT EMULATOR

- **iAPX 86, 88 in-circuit emulation**
- **Upgradable from ICE-86/88**
- **Full symbolic debugging support for all languages**
- **Supports iAPX 86/21, 88/21 configurations**
- **Breakpoints to halt emulation**
- **Comprehensive trace of program execution, both conditional and unconditional**
- **Disassembly of trace or memory from object code into assembler mnemonics**
- **2K bytes of high-speed memory**
- **Software debugging with or without user system**
- **Handles full 1 megabyte addressability of iAPX 86, 88**

The ICE-86A(88A) module provides in-circuit emulation for the 8086(88) microprocessor and the iSBC 86/12A single board computer. It includes three circuit boards which reside in Intellec Series II or Series III Microcomputer Development System. A cable and buffer box connect the Intellec system to the user system by replacing the user's 8086(88). Powerful Intellec debug functions are thus extended into the user system. Using the ICE-86A(88)A module, the designer can execute prototype software in continuous or single-step mode and can substitute blocks of Intellec system memory for user equivalents. Breakpoints allow the user to stop emulation on user-specified conditions, and the trace capability gives a detailed history of the program execution prior to the break. All user access to the prototype system software may be done symbolically by referring to the source program variables and labels for all languages.

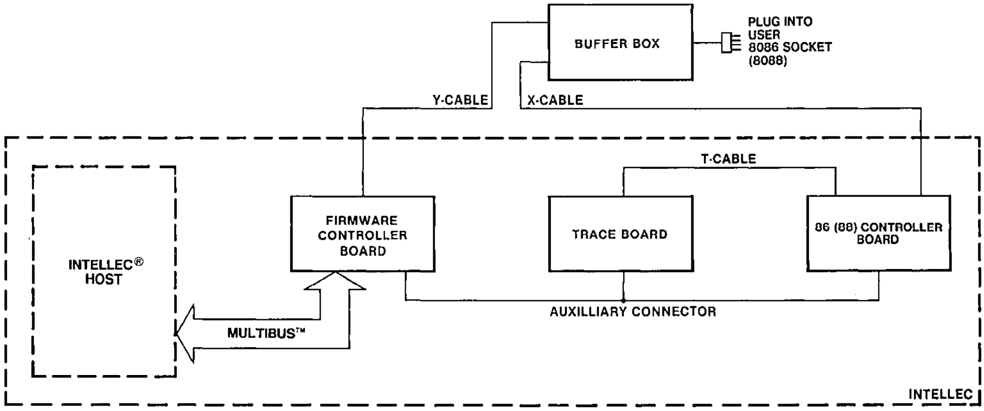


Figure 1. ICE-86A/88A™ Block Diagram

INTEGRATED HARDWARE/SOFTWARE DEVELOPMENT

The ICE-86A(88A) emulator allows hardware and software development to proceed interactively. This is more effective than the traditional method of independent hardware and software development followed by system integration. With the ICE-86A(88A) module, prototype hardware can be added to the system as it is designed. Software and hardware testing occurs while the product is being developed.

Conceptually, the ICE-86A(88A) emulator assists three stages of development:

1. It can be operated without being connected to the user's system, so ICE-86A(88A) debugging capabilities can be used to facilitate program development before any of the user's hardware is available.

2. Integration of software and hardware can begin when any functional element of the user system hardware is connected to the 8086(88) socket. Through ICE-86A(88A) mapping capabilities, Intellec memory, ICE memory, or diskette memory can be substituted for missing prototype memory. Time-critical program modules are debugged before hardware implementation by using the 2K-bytes of high-speed ICE-resident memory. As each section of the user's hardware is completed, it is added to the prototype. Thus each section of the hardware and software is "system" tested as it becomes available.

3. When the user's prototype is complete, it is tested with the final version of the user system software. The ICE-86A(88A) module is then used for real-time emulation of the 8086(88) to debug the system as a completed unit.

Thus the ICE-86A(88A) module provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

iAPX 86/20, 88/20 Numerics Supplement

Table of Contents

Tables

Illustrations

THE 8087 NUMERIC DATA PROCESSOR

This supplement describes the 8087 Numeric Data Processor (NDP). Its organization is similar to chapters 2 and 3 of *The 8086 Family User's Manual*:

1. Processor Overview
2. Processor Architecture
3. Computation Fundamentals
4. Memory
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Programming Facilities
9. Special Features
10. Programming Examples

Section 1 covers both hardware and software topics at a general level. Sections 2 and 4 through 6 are largely hardware-oriented, while sections 3 and 7 through 10 are of greatest interest to programmers. Section 9 describes features of the NDP that will be of interest to specialized groups of users; it is not necessary to understand this section to successfully use the 8087 in most applications. Hardware coverage in this supplement is limited to discussing processor facilities in functional terms. Timing, electrical characteristics, and other physical interface data may be found in Appendix B, as well as in Chapter 4 of *The 8086 Family User's Manual*.

Note that throughout this supplement the term "CPU" refers to either an 8086 or 8088 configured in maximum mode. To make best use of the material in this publication, readers should have a good understanding of the operation of the 8086/8088 CPUs.

S.1 Processor Overview

The 8087 Numeric Data Processor is a coprocessor that performs arithmetic and comparison operations on a variety of numeric data types; it also executes numerous built-in transcendental functions (e.g., tangent and log functions). As a coprocessor to a maximum mode 8086 or 8088, the NDP effectively extends the register and instruction sets of the host CPU and adds several new data types as well. The programmer generally does not perceive the 8087 as a separate device; instead, the computational capabilities of the CPU appear greatly expanded.

The 8087 is the only chip required to add extensive high-speed numeric processing capabilities to an 8086- or 8088-based system. It is specifically designed to deliver stable, correct results when used in a straightforward fashion by programmers who are not expert in numerical analysis. Its applicability to accounting and financial environments, in addition to scientific and engineering settings, further distinguishes the 8087 from the "floating point accelerators" employed in many computer systems, including minicomputers and mainframes. The NDP is housed in a standard 40-pin dual in-line package (figure S-1) and requires a single +5V power source.

The description of the 8087 in this section deliberately omits some operating details in order to provide a coherent overall view of the processor's capabilities. Subsequent sections of the supplement describe these capabilities, and others, in more detail.

Evolution

The performance of first- and second-generation microprocessor-based systems was limited in three principal areas: storage capacity, input/output speed, and numeric computation. The 8086 and 8088 CPUs broke the 64k memory barrier, allowing larger and more time-critical applications to be undertaken. The 8089 Input/Output Processor eliminated many of the I/O bottlenecks and permitted microprocessors to be employed effectively in I/O-intensive designs. The 8087 Numeric Data Processor clears the third roadblock by enabling applications with significant computational requirements to be implemented with microprocessor technology.

Figure S-2 illustrates the progression of Intel numeric products and events that have led to the development of the 8087. In the mid-1970's, Intel

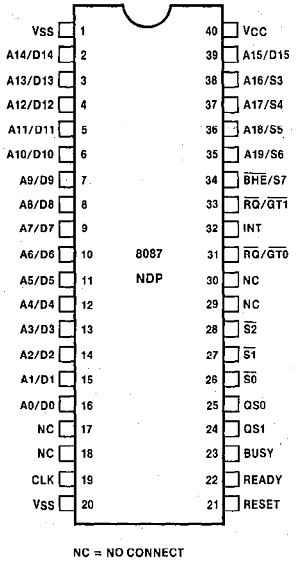


Figure S-1. 8087 Numeric Data Processor Pin Diagram

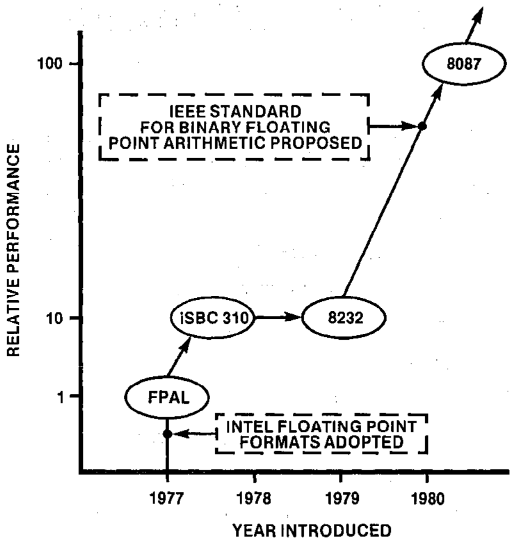


Figure S-2. 8087 Evolution and Relative Performance

made the commitment to expand the computational capabilities of microprocessors from addition and subtraction of integers to an array of widely useful operations on real numbers. (Real numbers encompass integers, fractions, and irrational numbers such as π and √2.) In 1977, the corporation adopted a standard for representing real numbers in a "floating point" format. Intel's Floating Point Arithmetic Library (FPAL) was the first product to utilize this standard format. FPAL is a set of subroutines for the 8080/8085 microprocessors. These routines perform arithmetic and limited standard functions on single precision (32-bit) real numbers; an FPAL multiply executes in about 1.5 ms (1.6 MHz 8080A CPU). The next product, the iSBC 310™ High Speed Math Unit, essentially implements FPAL in a single iSBC™ card, reducing a single-precision multiply to about 100 μs. The Intel® 8232 is a single-chip arithmetic processor for the 8080/8085 family. The 8232 accepts double precision (64-bit) operands as well as single precision numbers. It performs a single precision multiply in about 100 μs and multiplies double precision numbers in about 875 μs (2 MHz version).

In 1979, a working committee of the Institute for Electrical and Electronic Engineers (IEEE) proposed an industry standard for minicomputer and microcomputer floating point arithmetic*. The intent of the standard is to promote portability of numeric programs between computers and to provide a uniform programming environment that encourages the development of accurate, reliable software. The proposed standard specifies requirements and options for number formats as well as the results of computations on these numbers. The floating point number formats are identical to those previously adopted by Intel and used in the products described in this section.

The 8087 Numeric Data Processor is the most advanced development in Intel's continuing effort to provide improved tools for numerically-oriented microprocessor applications. It is a single-chip hardware implementation of the proposed IEEE standard, including all its options for single and double precision numbers. As such, it is compatible with previous Intel numerics products; programs written for the 8087 will be transportable to future products that conform to

* J. Coonen, W. Kahan, J. Palmer, T. Pittman, D. Stevenson, "A Proposed Standard for Binary Floating Point Arithmetic," *ACM SIGNUM Newsletter*, October 1979.

the proposed IEEE standard. The NDP also provides many additional functions that are extensions to the proposed standard.

Performance

As figure S-2 indicates, the 8087 provides about 10 times the instruction speed of the 8232 and a 100-fold improvement over FPAL. The 8087 multiplies 32-bit and 64-bit real numbers in about 19 μs and 27 μs, respectively. Of course, the actual performance of the NDP in a given system depends on numerous application-specific factors.

Table S-1 compares the execution times of several 8087 instructions with the equivalent operations executed in software on a 5 MHz 8086. The software equivalents are highly optimized assembly language procedures from the 8087 emulator, an NDP development tool discussed later in this section.

The performance figures quoted in this section are for operations on real (floating point) numbers. The 8087 also has instructions that enable it to utilize fixed point binary and decimal integers of up to 64 bits and 18 digits, respectively. Using an 8087, rather than multiple precision software algorithms for integer operations, can provide speed improvements of 10-100 times.

The 8087's unique coprocessor interface to the CPU can yield an additional performance increment beyond that of simple instruction speed. No overhead is incurred in setting up the device for a computation; the 8087 decodes its own instructions automatically in parallel with the CPU. Moreover, built-in coordination facilities allow the CPU to proceed with other instructions while the 8087 is simultaneously executing its numeric instruction. Programs can exploit this processor parallelism to increase total system throughput.

Usability

Viewed strictly from the standpoint of raw speed, the 8087 enables serious computation-intensive tasks to be performed by microprocessors for the first time. The 8087 offers more than just high performance, however. By synthesizing advances made by numerical analysts in the past several years, the NDP provides a level of usability that surpasses existing minicomputer and mainframe arithmetic units. In fact, the charter of the 8087 design team was first to achieve exceptional functionality and then to obtain high performance.

The 8087 is explicitly designed to deliver stable, accurate results when programmed using straightforward "pencil and paper" algorithms. While this statement may seem trivial, experienced users of "floating point processors" will

Table S-1. 8087 Emulator Speed Comparison

Instruction	Approximate Execution Time (μs) (5 MHz Clock)	
	8087	8086 Emulation
Multiply (single precision)	19	1,600
Multiply (double precision)	27	2,100
Add	17	1,600
Divide (single precision)	39	3,200
Compare	9	1,300
Load (single precision)	9	1,700
Store (single precision)	18	1,200
Square root	36	19,600
Tangent	90	13,000
Exponentiation	100	17,100

recognize its fundamental importance. For example, most computers can overflow when two single precision floating point numbers are multiplied together and then divided by a third, even if the final result is a perfectly valid 32-bit number. The 8087 delivers the correctly rounded result. Other typical examples of undesirable machine behavior in straightforward calculations occur when solving for the roots of a quadratic equation:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

or computing financial rate of return, which involves the expression: $(1+i)^n$. Straightforward algorithms will not deliver consistently correct results (and will not indicate when they are incorrect) on most machines. To obtain correct results on traditional machines under all conditions usually requires sophisticated numerical techniques that are foreign to most programmers. General application programmers using straightforward algorithms will produce much more reliable programs on the 8087. This simple fact greatly reduces the software investment required to develop safe, accurate computation-based products.

Beyond traditional numerics support for "scientific" applications, the 8087 has built-in facilities for "commerical" computing. It can process decimal numbers of up to 18 digits without round-off errors, and it performs *exact arithmetic* on integers as large as 2^{64}. Exact arithmetic is vital in accounting applications where rounding errors may introduce money losses that cannot be reconciled.

The NDP contains a number of facilities that can optionally be invoked by sophisticated users. Examples of these advanced features include two models of infinity, directed rounding, gradual underflow, and traps to user-written exception handling software.

Applications

The NDP's versatility and performance make it appropriate to a broad array of numerically-oriented applications. In general, applications that exhibit any of the following characteristics can benefit by implementing numeric processing on the 8087:

- Numeric data vary over a wide range of values, or include non-integral values;
- Algorithms produce very large or very small intermediate results;
- Computations must be very precise, i.e., a large number of significant digits must be maintained;
- Performance requirements exceed the capacity of traditional microprocessors;
- Consistently safe, reliable results must be delivered using a programming staff that is not expert in numerical techniques.

Note also that the 8087 can reduce software development costs and improve the performance of systems that do not utilize real numbers but operate on multi-precision binary or decimal integer values.

A few examples, which show how the 8087 might be utilized in specific numerics applications, are described below. In many cases, these types of systems have been implemented in the past with minicomputers. The advent of the 8087 brings the size and cost savings of microprocessor technology to these applications for the first time.

- Business data processing—The NDP's ability to accept decimal operands and produce exact decimal results of up to 18 digits greatly simplifies accounting programming. Financial calculations which use power functions can take advantage of the 8087's exponentiation and logarithmic instructions.
- Process control—The 8087 solves dynamic range problems automatically and its extended precision allows control functions to be fine-tuned for more accurate and efficient performance. Control algorithms implemented with the NDP also contribute to improved reliability and safety, while the 8087's speed can be exploited in real-time operations.
- Numerical control—The 8087 can move and position machine tool heads with extreme accuracy. Axis positioning also benefits from the hardware trigonometric support provided by the 8087.

- Robotics—Coupling small size and modest power requirements with powerful computational abilities, the NDP is ideal for on-board six-axis positioning.
- Navigation—Very small, light weight, and accurate inertial guidance systems can be implemented with the 8087. Its built-in trigonometric functions can speed and simplify the calculation of position from bearing data.
- Graphics terminals—The 8087 can be used in graphics terminals to locally perform many functions which normally demand the attention of a main computer; these include rotation, scaling, and interpolation. By also including an 8089 Input/Output Processor to perform high speed data transfers, very powerful and highly self-sufficient terminals can be built from a relatively small number of 8086 family parts.
- Data acquisition—The 8087 can be used to scan, scale, and reduce large quantities of data as it is collected, thereby lowering storage requirements as well as the time required to process the data for analysis.

The preceding examples are oriented toward "traditional" numerics applications. There are, in addition, many other types of systems that do not appear to the end user as "computational," but can employ the 8087 to advantage. Indeed, the 8087 presents the imaginative system designer with an opportunity similar to that created by the introduction of the microprocessor itself. Many applications can be viewed as numerically-based if sufficient computational power is available to support this view. This is analogous to the thousands of successful products that have been built around "buried" microprocessors, even though the products themselves bear little resemblance to computers.

Programming Interface

The combination of an 8086 or 8088 CPU and an 8087 generally appears to the programmer as a single machine. The 8087, in effect, adds new data types, registers, and instructions to the CPU. The programming languages and the coprocessor architecture take care of most interprocessor coordination automatically.

Table S-2 lists the seven 8087 data types. Internally, the 8087 holds all numbers in the temporary real format; the extended range and precision of this format are key contributors to the NDP's ability to consistently deliver stable, expected results. The 8087's load and store instructions convert operands between the other formats and temporary real. The fact that these conversions are made, and that calculations may be performed on converted numbers, is transparent to the programmer. Integer operands, whether binary or decimal, yield correct integer results, just as real operands yield correct real results. Moreover, a rounding error does not occur when a number in an external format is converted to temporary real.

Computations in the 8087 center on the processor's register stack. These eight 80-bit registers provide the equivalent capacity of 40 of the 16-bit registers found in typical CPUs. This generous register space allows more constants and intermediate results to be held in registers during calculations, reducing memory access and consequently improving execution speed as well as bus availability. The 8087 register set is unique in that it can be accessed both as a stack, with instructions operating implicitly on the top one or two stack elements, and as a fixed register set, with instructions operating on explicitly designated registers.

Table S-3 lists the 8087's major instructions by class. Assembly language programs are written in ASM-86, the 8086/8088/8087 common assembly language. ASM-86 provides directives for defining all 8087 data types and mnemonics for all instructions. The fact that some instructions in a program are executed by the 8087 and others by the CPU is usually of no concern to the programmer. All 8086/8088 addressing modes may be used to access memory-based 8087 operands, enabling convenient processing of numeric arrays, structures, based variables, etc.

NDP routines may also be written in PL/M-86, Intel's high-level language for the 8086 and 8088 CPUs. PL/M-86 provides the programmer with access to many 8087 facilities while reducing the programmer's need to understand the architecture of the chip.

Two features of the 8087 hardware further simplify numeric application programming. First, the 8087 is invoked directly by the programmer's instructions. There is no need to write instructions

Table S-2. Data Types

Data Type	Bits	Significant Digits (Decimal)	Approximate Range (Decimal)
Word integer	16	4	$-32{,}768 \leqslant X \leqslant +32{,}767$
Short integer	32	9	$-2x10^9 \leqslant X \leqslant +2x10^9$
Long integer	64	18	$-9x10^{18} \leqslant X \leqslant +9x10^{18}$
Packed decimal	80	18	$-99...99 \leqslant X \leqslant +99...99$ (18 digits)
Short real*	32	6-7	$8.43x10^{-37} \leqslant \lvert X \rvert \leqslant 3.37x10^{38}$
Long real*	64	15-16	$4.19x10^{-307} \leqslant \lvert X \rvert \leqslant 1.67x10^{308}$
Temporary real	80	19	$3.4x10^{-4932} \leqslant \lvert X \rvert \leqslant 1.2x10^{4932}$

*The short and long real data types correspond to the single and double precision data types defined in other Intel numerics products.

Table S-3. Principal Instructions

Class	Instructions
Data Transfer	Load (all data types), Store (all data types), Exchange
Arithmetic	Add, Subtract, Multiply, Divide, Subtract Reversed, Divide Reversed, Square Root, Scale, Remainder, Integer Part, Change Sign, Absolute Value, Extract
Comparison	Compare, Examine, Test
Transcendental	Tangent, Arctangent, $2^X - 1$, Y•Log$_2$(X+1), Y•Log$_2$(X)
Constants	0, 1, π, Log$_{10}$2, Log$_e$2, Log$_2$10, Log$_2$e
Processor Control	Load Control Word, Store Control Word, Store Status Word, Load Environment, Store Environment, Save, Restore, Enable Interrupts, Disable Interrupts, Clear Exceptions, Initialize

that "address" the NDP as an "I/O device", or to incur the overhead of setting up a DMA operation to perform data transfers. Second, the NDP automatically detects exception conditions that can potentially damage a calculation at run-time. On-chip exception handlers are automatically invoked by default to field these exceptions so that a reasonable result is produced and execution may proceed without program intervention. Alternatively, the 8087 can interrupt the CPU and thus trap to a user procedure when an exception is detected.

Besides the assembler and compiler, Intel provides a software emulator for the 8087. The 8087 emulator (E8087) is a software package that provides the functional equivalent of an 8087; it executes entirely on an 8086 or 8088 CPU. The emulator allows 8087 routines to be developed and checked out on an 8086/8088 execution vehicle before prototype 8087 hardware is operational. At the source code level, there is no difference between a routine that will ultimately run on an 8087 or on a CPU emulation of an 8087. At link time, the decision is made whether to use the NDP or the software emulator; no recompilation or re-assembly is necessary. Source programs are independent of the numeric execution vehicle: except for timing, the operation of the emulated NDP is the same as for "real hardware". The emulator also makes it simple for a product to offer the NDP as a "plug-in" performance option without the necessity of maintaining two sets of source code.

Hardware Interface

As a coprocessor to an 8086 or 8088, the 8087 is wired directly to the CPU as shown in figure S-3. The CPU's queue status lines (QS0 and QS1) enable the NDP to obtain and decode instructions in synchronization with the CPU. The NDP's BUSY signal informs the CPU that the NDP is executing; the CPU WAIT instruction tests this signal to ensure that the NDP is ready to execute a subsequent instruction. The NDP can interrupt the CPU when it detects an exception. The NDP's interrupt request line is typically routed to the CPU through an 8259A Programmable Interrupt Controller.

The NDP uses one of its host CPU's request/grant lines to obtain control of the local bus for data transfers (loads and stores). The other CPU request/grant line is available for general system use, for example, by a local 8089 Input/Output Processor. A local 8089 may also be connected to the 8087's $\overline{\text{RQ}}/\overline{\text{GT1}}$ line. In this configuration, the 8087 passes the request/grant handshake signals between the CPU and the IOP when the 8087 is not in control of the local bus. When it is in control of the bus, the 8087 relinquishes the bus (at the end of the current bus cycle) upon a request from the connected IOP, giving the IOP higher priority than itself. In this way, two local 8089's can be configured in a module that also includes a CPU and an 8087.

All processors utilize the same clock generator and system bus interface components (bus controller, latches, transceivers, and bus arbiter). Thus, no additional hardware beyond the 8087 is required to add powerful computational capabilities to an 8086- or 8088-based system.

S.2 Processor Architecture

As shown in figure S-4, the NDP is internally divided into two processing elements, the control unit (CU) and the numeric execution unit (NEU). In essence, the NEU executes all numeric instructions, while the CU fetches instructions, reads and writes memory operands, and executes the processor control class of instructions. The two

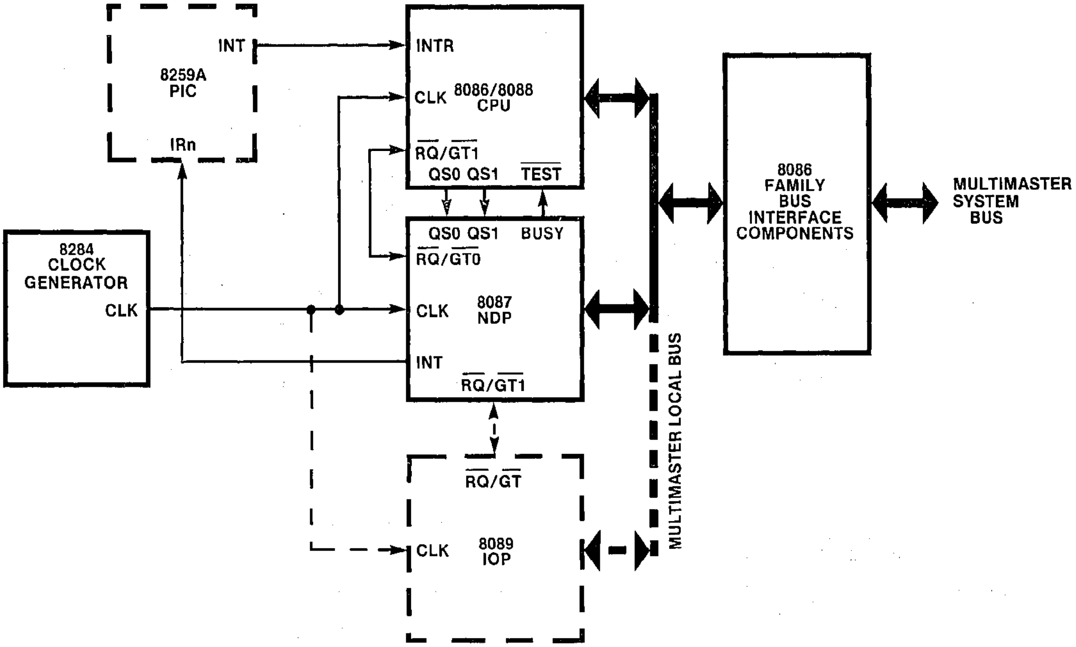


Figure S-3. NDP Interconnect

elements are able to operate independently of one another, allowing the CU to maintain synchronization with the CPU while the NEU executes numeric instructions.

Control Unit

The CU keeps the 8087 operating in synchronization with its host CPU. 8087 instructions are intermixed with CPU instructions in a single instruction stream fetched by the CPU. By monitoring the status signals emitted by the CPU, the NDP control unit can determine when an instruction is being fetched. When the instruction byte or word becomes available on the local bus, the CU taps the bus in parallel with the CPU and obtains that portion of the instruction.

The CU maintains an instruction queue that is identical to the queue in the host CPU. By monitoring the CPU's queue status lines, the CU is able to obtain and decode instructions from the queue in synchronization with the CPU. In effect, both processors fetch and decode the instruction stream in parallel.

The two processors execute the instruction stream differently, however. The first five bits of all 8087 machine instructions are identical; these bits designate the coprocessor escape (ESC) class of instructions. The control unit ignores all instructions that do not match these bits, since these instructions are directed to the CPU only. When the CU decodes an instruction containing the escape code, it either executes the instruction itself, or passes it to the NEU, depending on the type of instruction.

The CPU distinguishes between ESC instructions that reference memory and those that do not. If the instruction refers to a memory operand, the CPU calculates the operand's address and then performs a "dummy read" of the word at that location. This is a normal read cycle, except that the CPU ignores the data it receives. If the ESC instruction does not contain a memory reference, the CPU simply proceeds to the next instruction.

A given 8087 instruction (an ESC to the CPU) will either require loading an operand from memory into the 8087, or will require storing an operand from the 8087 into memory, or will not reference

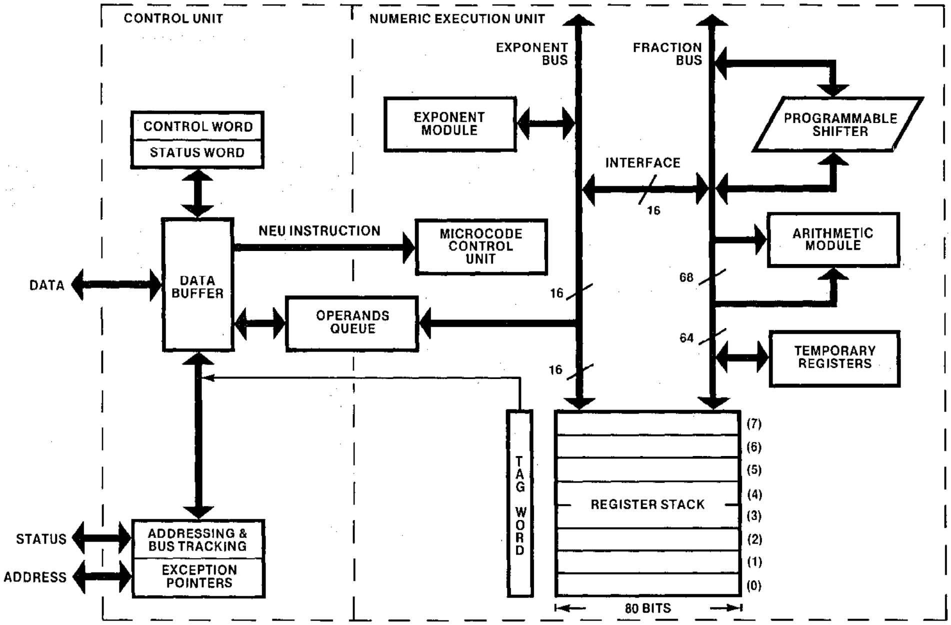


Figure S-4. 8087 Block Diagram

memory at all. In the first two cases, the CU makes use of the "dummy read" cycle initiated by the CPU. The CU captures and saves the operand address that the CPU places on the bus early in the "dummy read". If the instruction is an 8087 load, the CU additionally captures the first (and possibly only) word of the operand when it becomes available on the bus. If the operand to be loaded is longer than one word, the CU immediately obtains the bus from the CPU and reads the rest of the operand in consecutive bus cycles. In a store operation, the CU captures and saves the operand address as in a load, and ignores the data word that follows in the "dummy read" cycle. When the 8087 is ready to perform the store, the CU obtains the bus from the CPU and writes the operand at the saved address using as many consecutive bus cycles as are necessary to store the operand.

Numeric Execution Unit

The NEU executes all instructions that involve the register stack; these include arithmetic, comparison, transcendental, constant, and data transfer instructions. The data path in the NEU is 68 bits wide and allows internal operand transfers to be performed at very high speeds.

Register Stack

Each of the eight registers in the 8087's register stack is 80 bits wide, and each is divided into the "fields" shown in figure S-5. This format corresponds to the NDP's temporary real data type that is used for all calculations. Section S.3 describes in detail how numbers are represented in the temporary real format.

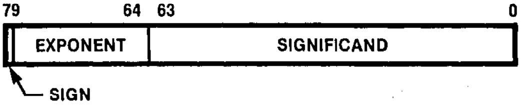


Figure S-5. Register Structure

At a given point in time, the ST field in the status word (described shortly) identifies the current top-of-stack register. A load ("push") operation decrements ST by 1 and loads a value into the new top register. A store-and-pop operation stores the value from the current top register and then increments ST by 1. Thus, like 8086/8088 stacks in memory, the 8087 register stack grows "down" toward lower-addressed registers.

Instructions may address registers either implicitly or explicitly. Many instructions operate on the register at the top of the stack. These instructions implicitly address the register pointed to by ST. For example, the ASM-86 instruction FSQRT replaces the number at the top of the stack with its square root; this instruction takes no operands because the top-of-stack register is implied as the operand. Other instructions allow the programmer to explicitly specify the register that is to be used. Explicit register addressing is "top-relative" where the ASM-86 expression ST denotes the current stack top and ST(*i*) refers to the *i*th register from ST in the stack ($0 \leqslant i \leqslant 7$). For example, if ST contains 011B (register 3 is the top of the stack), the following instruction would add registers 3 and 5:

```
FADD  ST, ST(2)
```

In typical use, the programmer may conceptually "divide" the registers into a fixed group and an adjustable group. The fixed registers are used like the conventional registers in a CPU, to hold constants, accumulations, etc. The adjustable group is used like a stack, with operands pushed on and results popped off. After loading, the registers in the fixed group are addressed explicitly, while those in the adjustable group are addressed implicitly. Of course, all registers may be addressed using either mode, and the "definition" of the fixed versus the adjustable areas may be altered at any time. Section S.8 contains a programming example that illustrates typical register stack use.

The stack organization and top-relative addressing of the registers simplify subroutine programming. Passing subroutine parameters on the register stack eliminates the need for the subroutine to "know" which registers actually contain the parameters and allows different routines to call the same subroutine without having to observe a convention for passing parameters in dedicated registers. So long as the stack is not full, each routine simply loads the parameters on the stack and calls the subroutine. The subroutine addresses the parameters as ST, ST(1), etc., even though ST may, for example, refer to register 3 in one invocation and register 5 in another.

Status Word

The status word reflects the overall condition of the 8087; it may be examined by storing it into memory with an NDP instruction and then inspecting it with CPU code. The status word is divided into the fields shown in figure S-6. The busy field (bit 15) indicates whether the NDP is executing an instruction (B=1) or is idle (B=0).

Several 8087 instructions (for example, the comparison instructions) post their results to the condition code (bits 14 and 10-8 of the status word). The principal use of the condition code is for conditional branching. This may be accomplished by executing an instruction that sets the condition code, storing the status word in memory and then examining the condition code with CPU instructions.

Bits 13-11 of the status word point to the 8087 register that is the current stack top (ST). Note that if ST=000B, a "push" operation, which decrements ST, produces ST=111B; similarly, popping the stack with ST=111B yields ST=000B.

Bit 7 is the interrupt request field. The NDP sets this field to record a pending interrupt to the CPU.

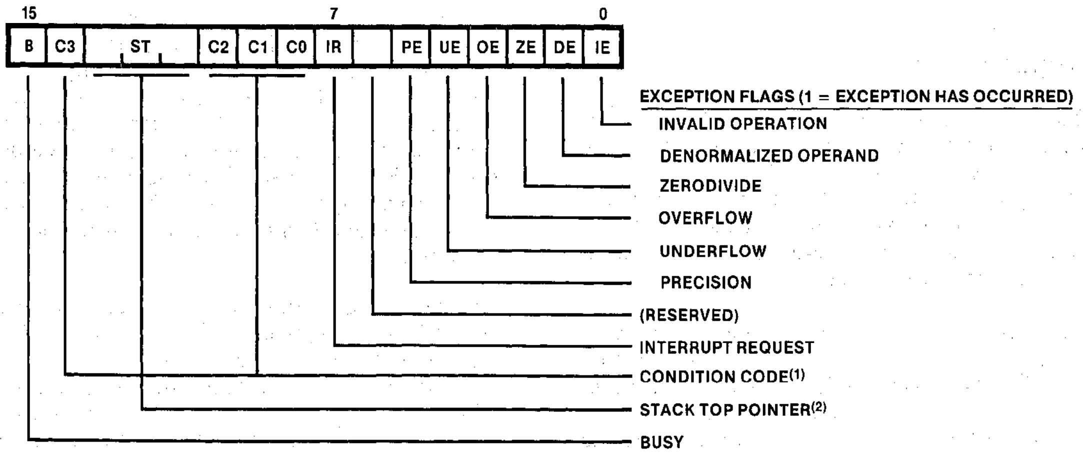
Bits 5-0 are set to indicate that the NEU has detected an exception while executing an instruction. Section S.3 explains these exceptions.

Control Word

To satisfy a broad range of application requirements, the NDP provides several processing options which are selected by loading a word from memory into the control word. Figure S-7 shows the format and encoding of the fields in the control word; it is provided here for reference. Section S.3 explains the use of each of these 8087 facilities except the interrupt-enable control field, which is covered in section S.6.

Tag Word

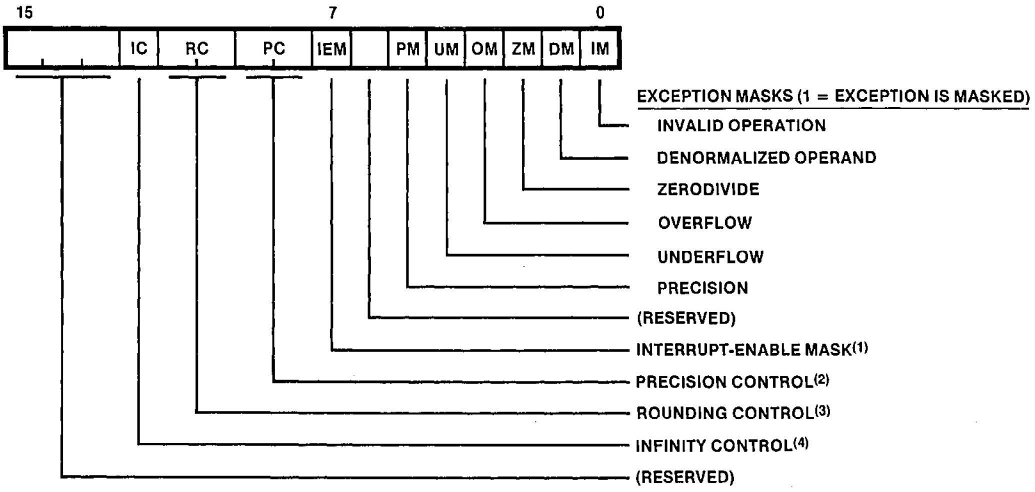
The tag word marks the content of each register as shown in figure S-8. The principal function



(1) See descriptions of compare, test, examine and remainder instructions in section S.7 for condition code interpretation.

(2) ST values:
000 = register 0 is stack top
001 = register 1 is stack top
•
•
•
111 = register 7 is stack top

Figure S-6. Status Word Format



[1] Interrupt-Enable Mask:
0 = Interrupts Enabled
1 = Interrupts Disabled (Masked)

[2] Precision Control:
00 = 24 bits
01 = (reserved)
10 = 53 bits
11 = 64 bits

[3] Rounding Control:
00 = Round to Nearest or Even
01 = Round Down (toward $-\infty$)
10 = Round Up (toward $+\infty$)
11 = Chop (Truncate Toward Zero)

[4] Infinity Control:
0 = Projective
1 = Affine

Figure S-7. Control Word Format

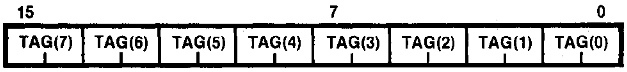
of the tag word is to optimize the NDP's performance under certain circumstances and programmers ordinarily need not be concerned with it.

Exception Pointers

The exception pointers (see figure S-9) are provided for user-written exception handlers. Whenever the 8087 executes an instruction, the CU saves the instruction address and the instruction opcode in the exception pointers. In addition, if the instruction references a memory operand, the address of the operand is retained also. An exception handler can store these pointers in memory and thus obtain information concerning the instruction that caused the exception.

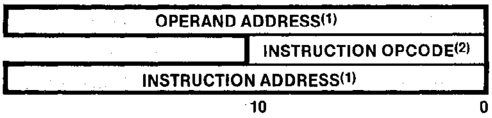
S.3 Computation Fundamentals

This section covers 8087 programming concepts that are common to all applications. It describes the 8087's internal number system and the various types of numbers that can be employed in NDP programs. The most commonly used options for rounding, precision and infinity (selected by fields in the control word) are described, with exhaustive coverage of less frequently used facilities deferred to section S.9. Exception conditions which may arise during execution of NDP instructions are also described along with the options that are available for responding to these exceptions.



Tag values:
00 = Valid (Normal or Unnormal)
01 = Zero (True)
10 = Special (Not-A-Number, ∞, or Denormal)
11 = Empty

Figure S-8. Tag Word Format



(1) 20-bit physical address

(2) 11 least significant bits of opcode; 5 most significant bits are always 8087 hook (11011B)

Figure S-9. Exception Pointers Format

Number System

The system of real numbers that people use for pencil and paper calculations is conceptually infinite and continuous. There is no upper or lower limit to the magnitude of the numbers one can employ in a calculation, or to the precision (number of significant digits) that the numbers can represent. When considering any real number, there are always an infinity of numbers both larger and smaller. There is also an infinity of numbers between (i.e., with more significant digits than) any two real numbers. For example, between 2.5 and 2.6 are 2.51, 2.5897, 2.500001, etc.

While ideally it would be desirable for a computer to be able to operate on the entire real number system, in practice this is not possible. Computers, no matter how large, ultimately have fixed-size registers and memories that limit the system of numbers that can be accommodated. These limitations proscribe both the range and the precision of numbers. The result is a set of numbers that is finite and discrete, rather than infinite and continuous. This sequence is a subset of the real numbers which is designed to form a useful *approximation* of the real number system.

Figure S-10 superimposes the basic 8087 real number system on a real number line (decimal numbers are shown for clarity, although the 8087 actually represents numbers in binary). The dots indicate the subset of real numbers the 8087 can represent as data and final results of calculations. The 8087's range is approximately ±4.19x10^{-307} to ±1.67x10^{308}. Applications that are required to deal with data and final results outside this range are rare. By comparison, the range of the IBM 370 is about ±0.54x10^{-78} to ±0.72x10^{76}.

The finite spacing in figure S-10 illustrates that the NDP can represent a great many, but not all, of the real numbers in its range. There is always a "gap" between two "adjacent" 8087 numbers, and it is possible for the result of a calculation to fall in this space. When this occurs, the NDP rounds the true result to a number that it can represent. Thus, a real number that requires more digits than the 8087 can accommodate (e.g., a 20 digit number) is represented with some loss of accuracy. Notice also that the 8087's representable numbers are not distributed evenly along the real number line. There are, in fact, an equal number of representable numbers between successive powers of 2 (i.e., there are as many representable numbers between 2 and 4 as between 65,536 and 131,072). Therefore, the "gaps" between representable numbers are

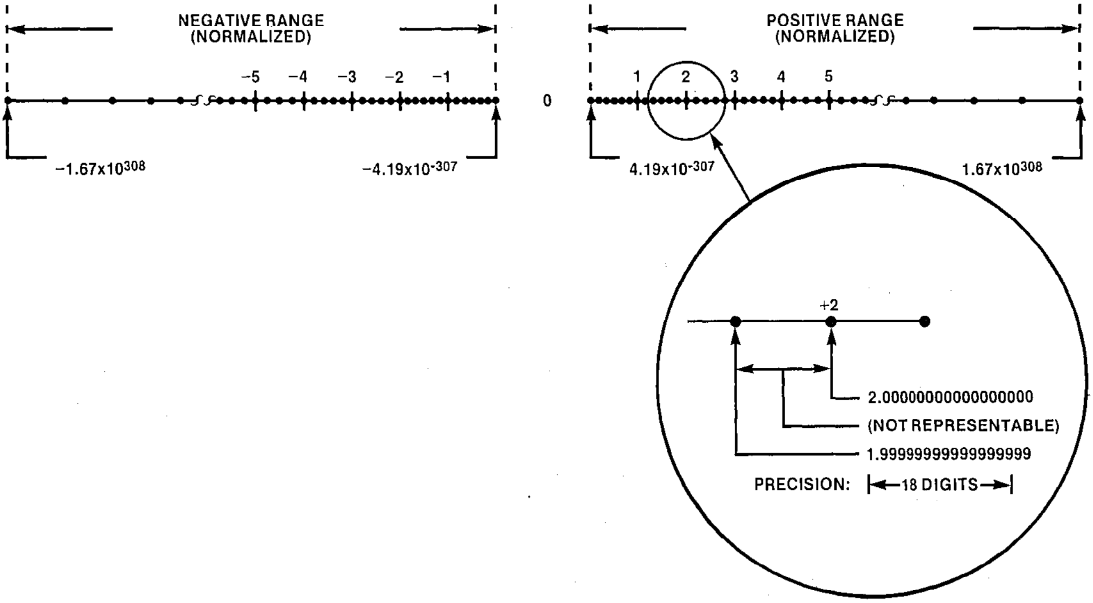


Figure S-10. 8087 Number System

"larger" as the numbers increase in magnitude. All integers in the range $\pm2^{64}$, however, are exactly representable.

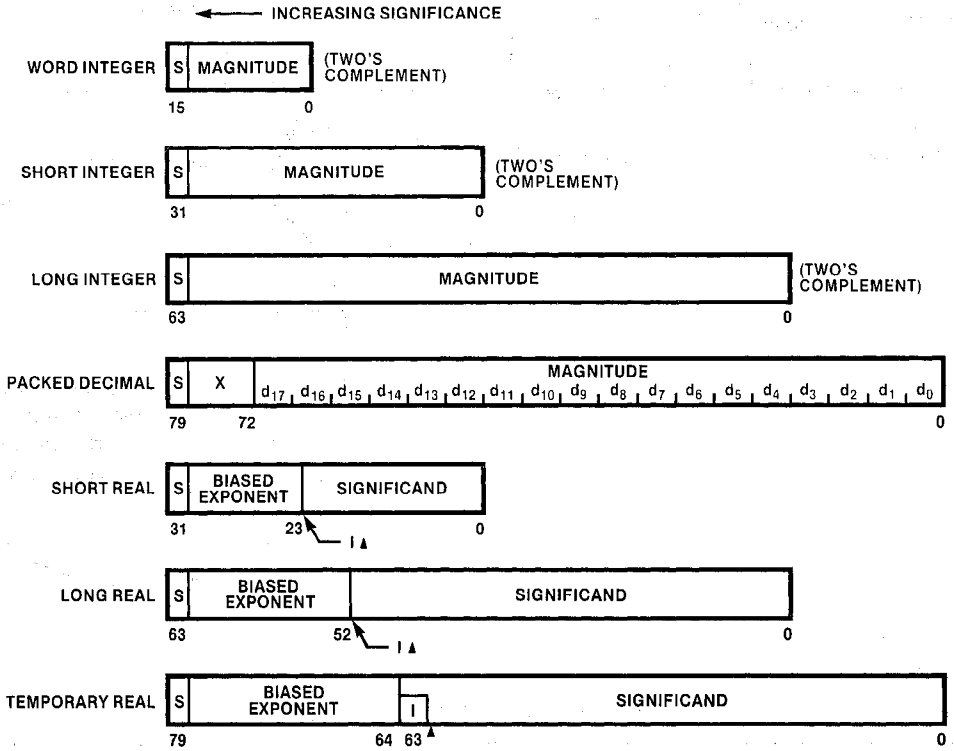
In its internal operations, the 8087 actually employs a number system that is a substantial superset of that shown in figure S-10. The internal format (called temporary real) extends the 8087's range to about $\pm3.4\times10^{-4932}$ to $\pm1.2\times10^{4932}$, and its precision to about 19 (equivalent decimal) digits. This format is designed to provide extra range and precision for constants and intermediate results, and is not normally intended for data or final results.

From a practical standpoint, the 8087's set of real numbers is sufficiently "large" and "dense" so as not to limit the vast majority of microprocessor applications. Compared to most computers, including mainframes, the NDP provides a very good approximation of the real number system. It is important to remember, however, that it is not an exact representation, and that arithmetic on real numbers is inherently approximate.

Conversely, and equally important, the 8087 does perform exact arithmetic on its integer subset of the reals. That is, an operation on two integers returns an exact integral result, provided that the true result is an integer and is in range. For example, 4÷2 yields an exact integer, 1÷3 does not, and $2^{40} \times 2^{30} + 1$ does not, because the result requires greater than 64 bits of precision.

Data Types and Formats

The 8087 recognizes seven numeric data types, divided into three classes: binary integers, packed decimal integers, and binary reals. Section S.4 describes how these formats are stored in memory (the sign is always located in the highest- addressed byte). Figure S-11 summarizes the format of each data type. In the figure, the most significant digits of all numbers (and fields within numbers) are the leftmost digits. Table S-2 provides the range and number of significant (decimal) digits that each format can accommodate.



NOTES:
S = Sign bit (0 = positive, 1 = negative)
d_n = Decimal digit (two per byte)
X = Bits have no significance; 8087 ignores when loading, zeros when storing.
▲ = Position of implicit binary point
I = Integer bit of significand; stored in temporary real, implicit in short and long real
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

Figure S-11. Data Formats

Binary Integers

The three binary integer formats are identical except for length, which governs the range that can be accommodated in each format. The leftmost bit is interpreted as the number's sign: 0=positive and 1=negative. Negative numbers are represented in standard two's complement notation (the binary integers are the only 8087 format to use two's complement). The quantity zero is represented with a positive sign (all bits are 0). The 8087 word integer format is identical to the 16-bit signed integer data type of the 8086 and 8088.

Decimal Integers

Decimal integers are stored in packed decimal notation, with two decimal digits "packed" into each byte, except the leftmost byte, which carries the sign bit (0 = positive, 1 = negative). Negative numbers are not stored in two's complement form and are distinguished from positive numbers only by the sign bit. The most significant digit of the number is the leftmost digit. All digits must be in the range 0H-9H.

Real Numbers

The 8087 stores real numbers in a three-field binary format that resembles scientific, or exponential, notation. The number's significant digits are held in the *significand* field, the *exponent* field locates the binary point within the significant digits (and therefore determines the number's magnitude), and the *sign* field indicates whether the number is positive or negative. (The exponent and significand are analogous to the terms "characteristic" and "mantissa" used to describe floating point numbers on some computers.) Negative numbers differ from positive numbers only in their sign bits.

Table S-4 shows how the real number 178.125 (decimal) is stored in the 8087 short real format. The table lists a progression of equivalent notations that express the same value to show how a number can be converted from one form to another. The ASM-86 and PL/M-86 language translators perform a similar process when they encounter programmer-defined real number constants. Note that not every decimal fraction has an exact binary equivalent. The decimal number 1/10, for example, cannot be expressed exactly in binary (just as the number 1/3 cannot be expressed exactly in decimal). When a translator encounters such a value, it produces a rounded binary approximation of the decimal value.

The NDP usually carries the digits of the significand in normalized form. This means that, except for the value zero, the significand is an *integer* and a *fraction* as follows:

1_Δfff...ff

where Δ indicates an assumed binary point. The number of fraction bits varies according to the real format: 23 for short, 52 for long and 63 for temporary real. By normalizing real numbers so that their integer bit is always a 1, the 8087 eliminates leading zeros in small values ($|x| < 1$). This technique maximizes the number of significant digits that can be accommodated in a significand of a given width. Note that in the short and long real formats the integer bit is *implicit* and is not actually stored; the integer bit is physically present in the temporary real format only.

If one were to examine only the significand with its assumed binary point, all normalized real numbers would have values between 1 and 2. The exponent field locates the *actual* binary point in the significant digits. Just as in decimal scientific notation, a positive exponent has the effect of moving the binary point to the right and a negative exponent effectively moves the binary point to the left, inserting leading zeros as necessary. An unbiased exponent of zero

Table S-4. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	1_Δ78125E2		
Scientific Binary	1_Δ0110010001E111		
Scientific Binary (Biased Exponent)	1_Δ 0110010001E10000110		
8087 Short Real (Normalized)	Sign	Biased Exponent	Significand
	0	10000110	01100100010000000000000 ↑ 1_Δ (implicit)

indicates that the position of the assumed binary point is also the position of the actual binary point. The exponent field, then, determines a real number's magnitude.

In order to simplify comparing real numbers (e.g., for sorting), the 8087 stores exponents in a biased form. This means that a constant is added to the *true exponent* described above. The value of this bias is different for each real format (see figure S-11). It has been chosen so as to force the *biased exponent* to be a positive value. This allows two real numbers (of the same format and sign) to be compared as if they are unsigned binary integers. That is, when comparing them bitwise from left to right (beginning with the leftmost exponent bit), the first bit position that differs orders the numbers; there is no need to proceed further with the comparison. A number's true exponent can be determined simply by subtracting the bias value of its format.

The short and long real formats exist in memory only. If a number in one of these formats is loaded into a register, it is automatically converted to temporary real, the format used for all internal operations. Likewise, data in registers can be converted to short or long real for storage in memory. The temporary real format may be used in memory also, typically to store intermediate results that cannot be held in registers.

Most applications should use the long real form to store real number data and results; it provides sufficient range and precision to return correct results with a minimum of programmer attention. The short real format is appropriate for applications that are constrained by memory, but it should be recognized that this format provides a smaller margin of safety. It is also useful for debugging algorithms because roundoff problems will manifest themselves more quickly in this format. The temporary real format should normally be reserved for holding intermediate results, loop accumulations, and constants. Its extra length is designed to shield final results from the effects of rounding and overflow/underflow in intermediate calculations. When the temporary real format is used to hold data or to deliver final results, the safety features built into the 8087 are compromised. Furthermore, the range and precision of the long real form are adequate for most microcomputer applications.

Special Values

Besides being able to represent positive and negative numbers, the 8087 data formats may be used to describe other entities. These special values provide extra flexibility but most users do not need to understand them in detail to use the 8087 successfully. Accordingly, they are discussed here only briefly; expanded coverage, including the bit encoding of each value, is provided in section S.9.

The value zero may be signed positive or negative in the real and decimal integer formats; the sign of a binary integer zero is always positive. The fact that zero may be signed, however, is transparent to the programmer.

The real number formats allow for the representation of the special values $+\infty$ and $-\infty$. The 8087 may generate these values as its built-in response to exceptions such as division by zero, or the attempt to store a result that exceeds the upper range limit of the destination format. Infinities may participate in arithmetic and comparison operations, and in fact the processor provides two different conceptual models for handling these special values.

If a programmer attempts an operation for which the 8087 cannot deliver a reasonable result, it will, at the programmer's discretion, either request an interrupt, or return the special value *indefinite*. Taking the square root of a negative number is an example of this type of invalid operation. The recommended action in this situation is to stop the computation by trapping to a user-written exception handler. If, however, the programmer elects to continue the computation, the specially coded *indefinite* value will propagate through the calculation and thus flag the erroneous computation when it is eventually delivered as the result. Each format has an encoding that represents the special value *indefinite*.

In the real formats, a whole range of special values, both positive and negative, is designated to represent a class of values called NAN (Not-A-Number). The special value *indefinite* is a reserved NAN encoding, but all other encodings are made available to be defined in any way by application software. Using a NAN as an operand raises the invalid operation exception, and can trap to a user-written routine to process the NAN. Alternatively, the 8087's built-in exception

Table S-5. Rounding Modes

RC Field	Rounding Mode	Rounding Action
00	Round to nearest	Closer to b of a or c; if equally close, select even number (the one whose least significant bit is zero).
01	Round down (toward $-\infty$)	a
10	Round up (toward $+\infty$)	c
11	Chop (toward 0)	Smaller in magnitude of a or c

Note: $a < b < c$; a and c are representable, b is not.

handler will simply return the NAN itself as the result of the operation; in this way NANs, including *indefinite*, may be propagated through a calculation and delivered as a final, special-valued, result. One use for NANs is to detect uninitialized variables.

As mentioned earlier, the 8087 stores non-zero real numbers in "normalized floating point" form. It also provides for storing and operating on reals that are not normalized, i.e., whose significands contain one or more leading zeros. Nonnormals arise when the result of a calculation yields a value that is too small to be represented in normal form. The leading zeros of nonnormals permit smaller numbers to be represented, at the cost of some lost precision (the number of significant digits is reduced by the leading zeros). In typical algorithms, extremely small values are most likely to be generated as intermediate, rather than final results. By using the NDP's temporary real format for holding intermediates, values as small as $\pm 3.4 \text{x} 10^{-4932}$ can be represented; this makes the occurrence of nonnormal numbers a rare phenomenon in 8087 applications. Nevertheless, the NDP can load, store and operate on nonnormalized real numbers.

Rounding Control

Internally, the 8087 employs three extra bits (guard, round and sticky bits) which enable it to represent the infinitely precise true result of a computation; these bits are not accessible to programmers. Whenever the destination can represent the infinitely precise true result, the 8087 delivers it. Rounding occurs in arithmetic and store operations when the format of the destination cannot exactly represent the infinitely precise true result. For example, a real number may be rounded if it is stored in a shorter real format, or in an integer format. Or, the infinitely precise true result may be rounded when it is returned to a register.

The NDP has four rounding modes, selectable by the RC field in the control word (see figure S-7). Given a true result b that cannot be represented by the target data type, the 8087 determines the two representable numbers a and c that most closely bracket b in value ($a < b < c$). The processor then rounds (changes) b to a or to c according to the mode selected by the RC field as shown in table S-5. Rounding introduces an error in a result that is less than one unit in the last place to which the result is rounded. "Round to nearest" is the default mode and is suitable for most applications; it provides the most accurate and statistically unbiased estimate of the true result. The "chop" mode is provided for integer arithmetic applications.

"Round up" and "round down" are termed directed rounding and can be used to implement interval arithmetic. Interval arithmetic generates a certifiable result independent of the occurrence of rounding and other errors. The upper and lower bounds of an interval may be computed by executing an algorithm twice, rounding up in one pass and down in the other.

Precision Control

The 8087 allows results to be calculated with 64, 53, or 24 bits of precision as selected by the PC field of the control word. The default setting, and the one that is best-suited for most applications, is the full 64 bits. The other settings are required by the proposed IEEE standard, and are provided to obtain compatibility with the specifications of certain existing programming languages. Specifying less precision nullifies the advantages of the temporary real format's extended fraction length, and does not improve execution speed. When reduced precision is specified, the rounding of the fraction zeros the unused bits on the right.

Infinity Control

The 8087's system of real numbers may be closed by either of two models of infinity. These two means of closing the number system, projective and affine closure, are illustrated schematically in figure S-12. The setting of the IC field in the control word selects one model or the other. The default means of closure is projective, and this is recommended for most computations. When projective closure is selected, the NDP treats the special values +∞ and −∞ as a single unsigned infinity (similar to its treatment of signed zeros). In the affine mode the NDP respects the signs of +∞ and −∞.

While affine mode may provide more information than projective, there are occasions when the sign may in fact represent misinformation. For example, consider an algorithm that yields an intermediate result x of +0 and −0 (the same numeric value) in different executions. If 1/x were then computed in affine mode, two entirely different values (+∞ and −∞) would result from numerically identical values of x. Projective mode, on the other hand, provides less information but never returns misinformation. In general, then, projective mode should be used globally, with affine mode reserved for local computations where the programmer can take advantage of the sign and knows for certain that the nature of the computation will not produce a misleading result.

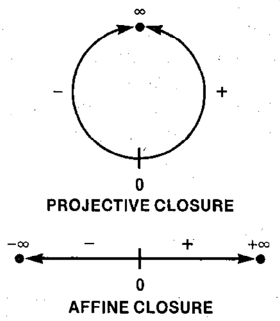


Figure S-12. Projective Versus Affine Closure

Exceptions

During the execution of most instructions, the 8087 checks for six classes of exception conditions.

The 8087 reports *invalid operation* if any of the following occurs:

- An attempt to load a register that is not empty, (e.g., stack overflow),
- An attempt to pop an operand from an empty register (e.g., stack underflow),
- An operand is a NAN,
- The operands cause the operation to be indeterminate (0/0, square root of a negative number, etc.).

An invalid operation generally indicates a program error.

If the exponent of the true result is too large for the destination real format, the 8087 signals *overflow*. Conversely, a true exponent that is too small to be represented results in the *underflow* exception. If either of these occur, the result of the operation is outside the range of the destination real format.

Typical algorithms are most likely to produce extremely large and small numbers in the calculation of intermediate, rather than final, results. Because of the great range of the temporary real format (recommended as the destination format for intermediates), overflow and underflow are relatively rare events in most 8087 applications.

If division of a finite non-zero operand by zero is attempted, the 8087 reports the *zerodivide* exception.

If an instruction attempts to operate on a denormal, the NDP reports the *denormalized* exception. This exception is provided for users who wish to implement, in software, an option of the proposed IEEE standard which specifies that operands must be prenormalized before they are used.

If the result of an operation is not exactly representable in the destination format, the 8087 rounds the number and reports the *precision* exception. This exception occurs frequently and indicates that some (generally acceptable) accuracy has been lost; it is provided for applications that need to perform exact arithmetic only.

Invalid operation, zerodivide, and denormalized exceptions are detected before an operation begins, while overflow, underflow, and precision exceptions are not raised until a true result has been computed. When a "before" exception is detected, the register stack and memory have not yet been updated, and appear as if the offending instruction has not been executed. When an "after" exception is detected, the register stack and memory appear as if the instruction has run to completion, i.e., they may be updated. (However, in a store or store and pop operation, unmasked over/underflow is handled like a "before" exception; memory is not updated and the stack is not popped.) In cases where multiple exceptions arise simultaneously, one exception is signalled according to the following precedence sequence:

- Denormalized (if unmasked),
- Invalid operation,
- Zerodivide,
- Denormalized (if masked),
- Over/underflow,
- Precision.

(The terms "masked" and "unmasked" are explained shortly.) This means, for example, that zero divided by zero will result in an invalid operation and not a zerodivide exception.

The 8087 reports an exception by setting the corresponding flag in the status word to 1. It then checks the corresponding exception mask in the control word to determine if it should "field" the exception (mask=1), or if it should issue an interrupt request to invoke a user-written exception handler (mask=0). In the first case, the exception is said to be *masked* (from user software) and the NDP executes its on-chip *masked response* for that exception. In the second case, the exception is *unmasked*, and the processor performs its *unmasked response*. The masked response always produces a standard result and then proceeds with the instruction. The unmasked response always traps to user software by interrupting the CPU (assuming the interrupt path is clear). These responses are summarized in table S-6. Section S.9 contains a complete description of all exception conditions and the NDP's masked responses.

Note that when exceptions are masked, the NDP may detect multiple exceptions in a single instruction, since it continues executing the instruction after performing its masked response. For example, the 8087 could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

By writing different values into the exception masks of the control word, the user can accept responsibility for handling exceptions, or delegate this to the NDP. Exception handling software is often difficult to write, and the 8087's masked responses have been tailored to deliver the most "reasonable" result for each condition. The majority of applications will find that masking all exceptions other than invalid operation will yield satisfactory results with the least programming investment. An invalid operation exception normally indicates a fatal error in a program that must be corrected; this exception should not normally be masked.

The exception flags are "sticky" and can be cleared only by executing the FCLEX (clear exceptions) instruction, by reinitializing the processor, or by overwriting the flags with an FRSTOR or FLDENV instruction. This means that the flags can provide a cumulative record of the exceptions encountered in a long calculation. A program can therefore mask all exceptions (except, typically, invalid operation), run the calculation and then inspect the status word to see if any exceptions were detected at any point in the calculation. Note that the 8087 has another set of internal exception flags that it clears before each instruction. It is these flags and not those in the status word that actually trigger the 8087's exception response. The flags in the status word provide a cumulative record of exceptions for the programmer only.

If the NDP executes an unmasked response to an exception, it is assumed that a user exception handler will be invoked via an interrupt from the 8087. The 8087 sets the IR (interrupt request) bit in the status word, but this, in itself, does not guarantee an immediate CPU interrupt. The interrupt request may be blocked by the IEM (interrupt-enable mask) in the 8087 control word, Table S-6. Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN, return it; if both are NANs, return NAN with larger absolute value; if neither is NAN, return *indefinite*.	Request interrupt.
Zerodivide	Return ∞ signed with "exclusive or" of operand signs.	Request interrupt.
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then re-evaluate for exceptions.	Request interrupt.
Overflow	Return properly signed ∞.	Register destination: adjust exponent*, store result, request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent*, store result, request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt.

*On overflow, 24,576 decimal is *subtracted* from the true result's exponent; this forces the exponent back into range and permits a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is *added* to the true result's exponent.

by the 8259A Programmable Interrupt Controller, or by the CPU itself. *If any exception flag is unmasked, it is imperative that the interrupt path to the CPU is eventually cleared so that the user's software can field the exception and the offending task can resume execution.* Interrupts are covered in detail in section S.6.

A user-written exception handler takes the form of an 8086/8088 interrupt procedure. Although exception handlers will vary widely from one application to the next, most will include these basic steps:

- Store the 8087 environment (control, status and tag words, operand and instruction pointers) as it existed at the time of the exception;
- Clear the exception bits in the status word;
- Enable interrupts on the CPU;
- Identify the exception by examining the status and control words in the saved environment;
- Take application-dependent action;
- Return to the point of interruption, resuming normal execution.

Possible "application-dependent actions" include:

- Incrementing an exception counter for later display or printing;
- Printing or displaying diagnostic information (e.g., the 8087 environment and registers);
- Aborting further execution of the calculation causing the exception;
- Aborting all further execution;
- Using the exception pointers to build an instruction that will run without exception and executing it.
- Storing a diagnostic value (a NAN) in the result and continuing with the computation.

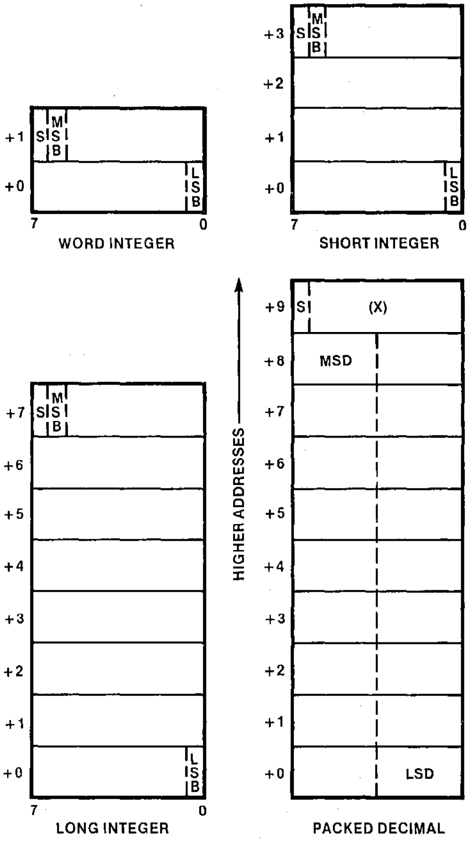
Notice that an exception may or may not constitute an error depending on the application. For example, an invalid operation caused by a stack overflow could signal an ambitious exception handler to extend the register stack to memory and continue running.

S.4 Memory

The 8087 can access any location in its host CPU's megabyte memory space. Because it relies on the CPU to generate the addresses of memory operands, the NDP can take advantage of the CPU's memory addressing modes and its ability to relocate code and data during execution.

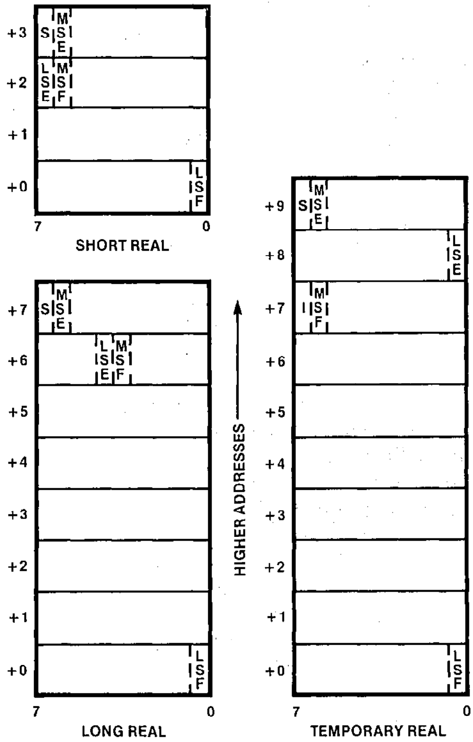
Data Storage

Figures S-13 and S-14 show how the 8087 data types are stored in memory. The sign bit is always located in the highest-addressed byte. The least significant binary or decimal digits in a number



S: Sign bit
MSB/LSB: Most/least significant bit
MSD/LSD: Most/least significant deciṁal digit
(X): Bits have no significance

Figure S-13. Storage of Integer Data Types



S: Sign bit
MSE/LSE: Most/least significant exponent bit
MSF/LSF: Most/least significant fraction bit
I: Integer bit of significand

Figure S-14. Storage of Real Data Types

(or in a field in the case of reals) are those with the lowest addresses. The word integer format is stored exactly like an 8086/8088 16-bit signed integer, and is directly usable by instructions executed on either the CPU or the NDP.

A few special instructions access memory to load or store formatted processor control and state data. The formats of these memory operands are provided with the discussions of the instructions in section S.7.

Storage Access

The host CPU always generates the address of the first (lowest-addressed) byte of a. memory operand. The CPU interprets an 8087 instruction that references memory as an ESC (escape), and generates the operand's effective and physical addresses normally as discussed in section 2.3. Any 8086/8088 memory addressing mode—direct, register indirect, based, indexed or based indexed—can be used to access an 8087 operand in memory. This makes the NDP easy to use with data structures such as arrays, structures, and lists.

When the CPU emits the 20-bit physical address of the memory operand, the 8087 captures the address and saves it. If the instruction loads information into the NDP, the 8087 captures the lowest-addressed word when it becomes available on the bus as a result of the CPU's "dummy read." (The "dummy read" may require either one or two bus cycles depending on the CPU type and the alignment of the operand.) If the operand is longer than one word (all 8087 operands are an integral number of words), the 8087 immediately requests use of the local bus by activating its CPU request/grant ($\overline{RQ}/\overline{GT0}$) line, as described in section S.6. When the NDP obtains the bus; it runs consecutive bus cycles incrementing the saved address until the rest of the operand has been obtained, returns the local bus to the CPU, and then executes the instruction.

If an operation stores data from the NDP to memory, the NDP and the CPU both ignore the data placed on the bus by the CPU's "dummy read." The NDP does not request the bus from the CPU until it is ready to write the result of the instruction to memory. When it obtains the bus, the NDP writes the operand in successive bus cycles, incrementing the saved address as in a load.

As described in section S.6, the 8087 automatically determines the identity of its host CPU. When the NDP is wired to an 8088, it transfers one byte per bus cycle in the same manner as the CPU. When used with an 8086, the NDP again operates like the CPU, accessing odd-addressed words in two bus cycles and even-addressed words in one bus cycle. If the 8087 is reading or writing more than one word of an odd-addressed operand in 8086 memory, it optimizes the transfer by accessing a byte on the first transfer, forcing the address to even, and then transferring words up to the last byte of the operand.

To minimize operand transfer time and 8087 use of the system bus, it is advantageous to align 8087 memory operands on even addresses when the CPU is an 8086. Following the same practice for 8088-based systems will ensure top performance without reprogramming if the application is transferred to an 8086. The ASM-86 EVEN directive can be used to force word alignment.

Dynamic Relocation

Since the host CPU takes care of both instruction fetching and memory operand addressing, the NDP may be utilized in systems that alter program addresses during execution. The only restriction on the CPU is that it should not change the address of an 8087 operand while the 8087 is executing an instruction which stores a result to that address. If this is done, the 8087 will store to the operand's old address (the one it picked up during the "dummy read").

Dedicated and Reserved Memory Locations

The 8087 does not require any addresses in memory to be set aside for special purposes. Care should be taken, however, to respect the dedicated and reserved areas associated with the CPU and the IOP (see sections 2.3 and 3.3). Using any of these areas may inhibit compatibility with current or future Intel hardware and software products.

S.5 Multiprocessing Features

As a coprocessor to an 8086 or 8088 CPU, the NDP is by definition always used in a multiprocessing environment. This section describes the facilities built into the 8087 that simplify the coordinaton of multiple processor systems. Included are descriptions of instruction synchronization, local and system bus arbitration, and shared resource access control.

Instruction Synchronization

In the execution of a typical NDP instruction, the CPU will complete the ESC long before the 8087 finishes its interpretation of the same machine instruction. For example, the NDP performs a square root in about 180 clocks, while the CPU will execute its interpretation of this same instruction in 2 clocks. Upon completion of the ESC, the CPU will decode and execute the next instruction, and the NDP's CU, tracking the CPU, will do the same. (The NDP "executes" a CPU instruction by ignoring it). If the CPU has work to do that does not affect the NDP, it can proceed with a series of instructions while the NDP is executing in parallel; the NDP's CU will ignore these CPU-only instructions as they do not contain the 8087 escape code. This asynchronous execution of the processors can substantially improve the performance of systems that can be designed to exploit it.

There are two cases, however, when it is necessary to synchronize the execution of the CPU to the NDP:

1. An NDP instruction that is executed by the NEU must not be started if the NEU is still busy executing a previous instruction.
2. The CPU should not execute an instruction that accesses a memory operand being referenced by the NDP until the NDP has actually accessed the location.

The 8086/8088 WAIT instruction allows software to synchronize the CPU to the NDP so that the CPU will not execute the following instruction until the NDP is finished with its current (if any) instruction.

Whenever the 8087 is executing an instruction, it activates its BUSY line. This signal is wired to the CPU's $\overline{\text{TEST}}$ input as shown in figure S-3. The NDP ignores the WAIT instruction, and the CPU executes it. The CPU interprets the WAIT instruction as "wait while $\overline{\text{TEST}}$ is active." The CPU examines the $\overline{\text{TEST}}$ pin every 5 clocks; if $\overline{\text{TEST}}$ is inactive, execution proceeds with the instruction following the WAIT. If $\overline{\text{TEST}}$ is active, the CPU examines the pin again. Thus, the effective execution time of a WAIT can stretch from 3 clocks (3 clocks are required for decoding and setup) to infinity, as long as $\overline{\text{TEST}}$ remains active. The WAIT instruction, then, prevents the CPU from decoding the next instruction until the 8087 is not busy. The instruction following a WAIT is decoded simultaneously by both processors.

To satisfy the first case mentioned above, every 8087 instruction that affects the NEU should be preceded by a WAIT to ensure that the NEU is ready. All instructions except the processor control class affect the NEU. To simplify programming, the 8086 family language translators provide the WAIT automatically. When an assembly language programmer codes:

```
FMUL    ;(multiply)
FDIV    ;(divide)
```

the assembler produces *four* machine instructions, as if the programmer had written:

```
WAIT
FMUL
WAIT
FDIV
```

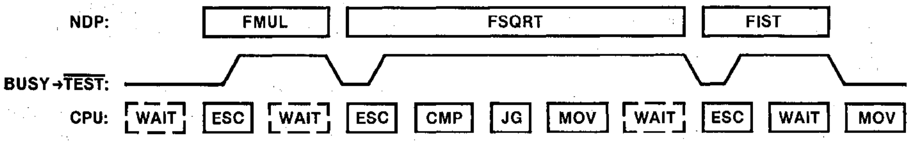
This ensures that the multiply runs to completion before the CPU and the 8087 CU decode the divide.

To satisfy the second case, the programmer should explicitly code the FWAIT instruction immediately before a CPU instruction that accesses a memory operand read or written by a previous 8087 instruction. This will ensure that the 8087 has read or written the memory operand before the CPU attempts to use it. (The FWAIT mnemonic causes the assembler to create a CPU WAIT instruction that can be eliminated at link time if the program is to run on an 8087 emulator. See section S.8 for details.)

Figure S-15 is a hypothetical sequence of instructions that illustrates the effect of the WAIT instruction and parallel execution of the NDP with a CPU.

The first two instructions in the sequence (FMUL and FSQRT) are 8087 instructions that illustrate the ASM-86 assembler's automatic generation of

```
          ;ASSUME 8087 REGISTER STACK IS LOADED WITH OPERANDS,
          ;      NEU IS NOT BUSY,
          ;      AND THAT 'ALPHA' AND 'BETA' ARE WORD
          ;      INTEGERS.
          ;
          FMUL                    ;MULTIPLY TOP STACK
                                  ;ELEMENTS
          FSQRT                   ;SQUARE ROOT OF PRODUCT
          CMP    ALPHA,100        ;ALPHA > 100?
          JG     CONTINUE         ;YES, LEAVE UNALTERED
          MOV    ALPHA,100        ;NO, SET TO 100
CONTINUE: FIST   BETA             ;STORE ROOT AS INTEGER WORD
          FWAIT                   ;WAIT FOR 8087 TO COMPLETE
                                  ;STORE OF BETA
          MOV    AX,BETA          ;PROCEED TO PROCESS BETA
```



NOTES:

- WAIT = Assembler-generated instruction.
- Instruction execution times are not drawn to scale.

Figure S-15. Synchronizing Execution With WAIT

a preceding WAIT, and the effect of the WAIT when the NDP is, and is not, busy. Since the NDP is not busy when the first WAIT is encountered, the CPU executes it and immediately proceeds to the next instruction; the NDP ignores the WAIT. The next instruction is decoded simultaneously by both processors. The NDP starts the multiplication and raises its BUSY line. The CPU executes the ESC and then the second WAIT. Since $\overline{\text{TEST}}$ is active (it is tied to BUSY), the CPU effectively stretches execution of this WAIT until the NDP signals completion of the multiply by lowering BUSY. The next instruction is interpreted as a square root by the NDP and another escape by the CPU. The CPU finishes the ESC well before the NDP completes the FSQRT. This time, instead of waiting, the CPU executes three instructions (compare, jump if greater, and move) while the 8087 is working on the FSQRT. The 8087 ignores these CPU-only instructions. The CPU then encounters the third WAIT, generated by the assembler immediately preceding the FIST (store stack top into integer word). When the NDP finishes the FSQRT, both processors proceed to the next instruction, FIST to the NDP and ESC to the CPU. The CPU completes the escape quickly and then executes an explicit programmer-coded FWAIT to ensure that the 8087 has updated BETA before it moves BETA's new value to register AX.

The 8087 CU can execute most processor control instructions by itself regardless of what the NEU is doing: thus the 8087 can, in these cases, potentially execute two instructions at once. The ASM-86 assembler provides separate "wait" and "no wait" mnemonics for these instructions. For example, the instruction that sets the 8087 interrupt enable mask, and thus disables interrupts, can be coded as FDISI or FNDISI. The assembler does *not* generate a preceding WAIT if the second form is coded, so that interrupts can be disabled while the NEU is busy executing a previous instruction. The no-wait forms are principally used in exception handlers and operating systems.

Local Bus Arbitration

Whenever an NDP instruction writes data to memory, or reads more than one word from memory, the NDP forces the CPU to relinquish the local bus. It does this by means of the request/grant facility built into all 8086 family processors. For memory reads, the NDP requests the bus immediately upon the CPU's completion of its "dummy read" cycle; it follows from this that the CPU may "immediately" update a variable read by the NDP in the previous instruction with the assurance that the NDP will have obtained the old value before the CPU has altered it. For memory writes, the NDP performs as much processing as possible before requesting the bus. In all cases, the 8087 transfers the data in back-to-back bus cycles and then immediately releases the bus.

The 8087's $\overline{RQ}/\overline{GT0}$ line is wired to one of the CPU's request/grant lines. Connecting it to $\overline{RQ}/\overline{GT1}$ on the CPU (see figure S-3) leaves the higher priority $\overline{RQ}/\overline{GT0}$ open for possible attachment of a local 8089 to the CPU. Note that an 8089 on $\overline{RQ}/\overline{GT0}$ will obtain the bus if it requests it simultaneously with an 8087 attached to $\overline{RQ}/\overline{GT1}$; it cannot, however, preempt the 8087 if the 8087 has the bus. The NDP requests the local bus by pulsing its $\overline{RQ}/\overline{GT0}$ line. If the CPU has the bus, it will grant it to the NDP by pulsing the same request/grant line. The CPU grants the bus immediately unless it is running a bus cycle, in which case the grant is delayed until the bus cycle is completed. The NDP releases the bus back to the CPU by sending a final pulse on $\overline{RQ}/\overline{GT0}$ when it has completed the transfer.

The 8087 provides a second request/grant line, $\overline{RQ}/\overline{GT1}$, that may be used to service local bus requests from an 8089 Input/Output Processor (see figure S-3). By using this line, a CPU, two IOPs (one is attached directly to the CPU) and an NDP can all reside on the same local bus, sharing a single set of system bus interface components.

When the 8087 detects a bus request pulse on $\overline{RQ}/\overline{GT1}$, its response depends on whether it is idle, executing, or running a bus cycle. If it is idle or executing, the 8087 passes the bus request through to the CPU via $\overline{RQ}/\overline{GT0}$. The subsequent grant and release pulses are also passed between the CPU and the requesting device. If the 8087 is running a bus cycle (or a series of bus cycles), it has already obtained the bus from the CPU so it grants the bus directly at the end of the current bus cycle rather than passing the request on to the CPU. When the 8089 releases the bus, the 8087 resumes the series of bus cycles it was running before it granted the bus to the 8089. Thus, to an 8089 attached to the 8087's $\overline{RQ}/\overline{GT1}$ line, the NDP appears to be a CPU. An IOP attached to an NDP also effectively has higher local bus priority than the NDP, since it can force the NDP to relinquish the bus even in the midst of a multi-cycle transfer. This satisfies the typical system requirement for I/O transfers to be serviced as soon as possible.

System Bus Arbitration

A single 8288 Bus Controller (plus latches and tranceivers as required) links both the host CPU and the NDP to the system bus. The 8087 performs system bus transfers exactly the same as its CPU; status, address, and data signals and timing are identical.

In systems that allow multiple processing modules on separate local buses common access to a public system bus, the 8087 also shares its host CPU's 8289 Bus Arbiter. The 8289 operates identically regardless of whether the system bus request is initiated by the CPU or the NDP. Since only one of the processors in the module will have control of the local bus at the time of a request to access the system bus, the transfer will be between the controlling processor and the system bus. If the 8289 does not obtain the system bus immediately, it causes the bus to appear "not ready" (as if a slow memory were being accessed), and the 8087 will stretch the bus cycle by adding the wait states.

Because it presents the same system bus interface as a maximum mode 8086 family CPU, the NDP is also electrically compatible with Intel's Multibus™ shared system bus architecture. This means that the 8087 can be utilized in systems that are based on the broad line of iSBC™ single board computers, controllers, and memories.

Controlled Variable Access

If an 8087 and a processor other than its host CPU can both update a variable, access to that variable should be controlled so that one processor at a time has exclusive rights to it. This may be implemented by a semaphore convention as described in section 2.5. However, since the 8087 has no facility for locking the system bus during an instruction, the host CPU should obtain exclusive rights to the variable before the 8087 accesses it. This can be done using an XCHG instruction prefixed by LOCK as discussed in section 2.5. When the NDP no longer needs the controlled variable the CPU should clear the semaphore to signal other processors that the variable is again available for use.

S.6 Processor Control and Monitoring

Initialization

The NDP may be initialized by hardware or software. Hardware initialization occurs in response to a pulse on the 8087's RESET line. When the processor detects RESET going active, it suspends all activities. When RESET subsequently goes inactive, the NDP initializes itself. The state of the NDP following initialization is shown in table S-7. Hardware initialization also causes the 8087 to identify its host CPU and begin to track its instruction fetches and execution. Initialization does not affect the content of the registers or of the exception pointers (these have indeterminate values immediately following power up). However, since the stack is effectively emptied by initialization (ST = 0, all registers tagged empty), the contents of the registers should normally be considered "destroyed" by initialization.

The FINIT (intialize) and FSAVE (save state) instructions also initialize the processor. Unlike a RESET pulse, software initialization does not affect the 8087's tracking of the CPU.

CPU Identification

The 8087's bidirectional $\overline{\text{BHE}}$ (bus high enable) line is tied to pin 34 of the CPU ($\overline{\text{BHE}}$ on the 8086, SSO on the 8088). The 8088 always holds SSO = 1 . The 8086 emits a 0 on BHE whenever it is accessing an even-addressed word or an odd-addressed byte.

Following RESET, the CPU always performs a word fetch of its first instruction from the dedicated memory location: FFFF0H. The 8087 identifies its host CPU by monitoring $\overline{\text{BHE}}$ during the CPU's first fetch following RESET. If $\overline{\text{BHE}}$ =1, the CPU is an 8088; if $\overline{\text{BHE}}$ =0, the CPU is an 8086 (because the first fetch is an even-addressed word). Note that to ensure proper operation, the same pulse must reset both the 8087 and its host CPU.

Table S-7. Processor State Following Initialization

Field	Value	Interpretation
Control Word		
Infinity Control	0	Projective
Rounding Control	00	Round to nearest
Precision Control	11	64 bits
Interrupt-enable Mask	1	Interrupts disabled
Exception Masks	111111	All exceptions masked
Status Word		
Busy	0	Not busy
Condition Code	????	(Indeterminate)
Stack Top	000	Empty stack
Interrupt Request	0	No interrupt
Exception Flags	000000	No exceptions
Tag Word		
Tags	11	Empty
Registers	N.C.	Not changed
Exception Pointers		
Instruction Code	N.C.	Not changed
Instruction Address	N.C.	Not changed
Operand Address	N.C.	Not changed

Interrupt Requests

The 8087 can request an interrupt of its host CPU via the 8087 INT (interrupt request) pin. This signal is normally routed to the CPU's INTR input via an 8259A Programmable Interrupt Controller (PIC). The 8087 should not be tied to the CPU's NMI (non-maskable interrupt) line.

All 8087 interrupt requests originate in the detection of an exception. The interrupt request logic is illustrated in figure S-16. The interrupt request is made if the exception is unmasked *and* 8087 interrupts are enabled, i.e., both the relevant exception mask and the interrupt-enable mask are clear (0). If the exception is masked, the processor executes its masked response and does not set the interrupt request bit.

If the exception is unmasked but interrupts are disabled (IEM = 1), the 8087's action depends on whether the CPU is waiting (the 8087 "knows" if the CPU is waiting because it decodes the WAIT instruction in parallel with the CPU). If the CPU is *not* waiting, the 8087 assumes that the CPU does not want to be interrupted at present and that it will enable interrupts on the 8087 when it does. The 8087 sets the interrupt request bit and holds its BUSY line active. The 8087 CU continues to track the CPU, and if an 8087 instruction (without a preceding WAIT) comes along, it will be executed. Normally in this situation the instruction would be FNENI (enable interrupts without waiting). This will clear the interrupt-enable mask and the 8087 will then activate INT. However, any instruction will be executed, and it is therefore conceivably possible to abort the interrupt request before it is ever handled. Aborting an interrupt request in this manner, however, would normally be considered a program error.

If the CPU is waiting, then the processors are in danger of entering an endless wait condition (discussed shortly). To prevent this condition, the 8087 *ignores* the fact that interrupts are disabled and activates INT even though the interrupt-enable mask is set.

The interrupt request bit remains set until it is explicitly cleared (if INT is not disabled by IEM, it will remain active also). This can be done by the FNCLEX, FNSAVE, or FNINIT instructions. The interrupt procedure that fields the 8087's interrupt request, i.e., the exception handler, must clear the interrupt request bit before returning to normal execution on the 8087. If it does not, the interrupt will immediately be generated again and the program will enter an endless loop.

Interrupt Priority

Most systems can be viewed as consisting of two distinct classes of software: interrupt handlers and application tasks. Interrupt handlers execute in response to external events; in the 8086 family they are implemented as interrupt service procedures. (Of course, the CPU interrupt instructions allow interrupt handlers to respond to internal "events" also.) A hardware interrupt controller, such as the 8259A, usually monitors the external events and invokes the appropriate interrupt handler by activating the CPU INTR line, and passing a code to the CPU that identifies the interrupt handler that is to service the event. Since the 8259A typically monitors several events, a priority-resolving technique is used to select one

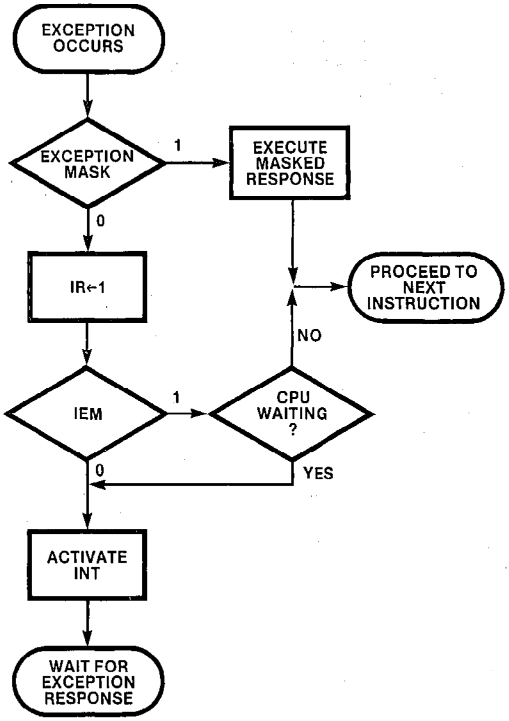


Figure S-16. Interrupt Request Logic

event when several occur simultaneously. Many systems allow higher-priority interrupts to preempt lower-priority interrupt handlers. The 8259A supports several priority-resolving techniques; a system will normally select one of these by programming the 8259A at initialization time.

Application tasks execute only when no external event needs service, i.e., when no interrupt handler is running. Application tasks are invoked by software, rather than hardware; typically a scheduling or dispatching algorithm is used to select one task for execution. In effect, any interrupt handler has higher priority than any application task, since the recognition of an interrupt will invoke the interrupt handler, preempting the application task that was running.

There are two important questions to consider when assigning a priority to the 8087's interrupt request:

- Who can cause 8087 exceptions—only application tasks, or interrupt handlers as well?
- Who should be preempted by NDP exceptions—only applications tasks, or interrupt handlers as well?

Given these considerations, the 8087 should normally be assigned the lowest priority of any interrupting device in the system. This allows the interrupt handler (i.e., the NDP exception handler) to preempt any application task that generates an 8087 exception, and at the same time prevents the exception NDP handler from interfering with other interrupt handlers.

If an *interrupt handler* uses the 8087 and requires the service of the exception handler, it can effectively "raise" the priority of the exception handler by disabling all interrupts lower than itself and higher than the 8087. Then, any unmasked exception caused by the interrupt handler will be fielded without interference from lower-priority interrupts.

If, for some reason, the 8087 must be given higher priority than another interrupt source, the interrupt handler that services the lower-priority device may want to prevent interrupts from the 8087 (which may originate in a long instruction still running on the 8087 when the interrupt handler is invoked) from preempting it. This should be done by executing the FNSTCW and FNDISI instructions before enabling CPU interrupts. Before returning, the interrupt handler should restore the original control word in the 8087 by executing FLDCW.

Users should consult "*Using the 8259A Programmable Interrupt Controller*", Intel Application Note No. AP-59, for a description of the 8259A's various modes of operation.

Endless Wait

The 8087 and its host CPU can enter an endless wait condition when the CPU is executing a WAIT instruction and a pending interrupt request from the 8087 is prevented from being recognized by the CPU. Thus, the CPU will wait for the 8087 to lower its BUSY line, while the NDP will wait for the CPU to invoke the exception handler interrupt procedure, and the task which has generated the exception will be blocked from further execution.

Figure S-17 shows the typical path of an interrupt request from the 8087 to the interrupt procedure which is designated to field NDP exceptions. The interrupt request can be potentially blocked at three points along the path, creating an endless wait if the CPU is executing a WAIT instruction. The first block can occur at the 8087's interrupt-enable mask (IEM). If this mask is set, the interrupt request is blocked except that the 8087 will override the mask if the CPU is waiting (the 8087 decodes the WAIT instruction simultaneously with the CPU). Thus, the 8087 detects and prevents one of the endless wait conditions.

A given interrupt request, IRn, can be masked on the 8259A by setting the corresponding bit in the PIC's interrupt mask register (IMR). This will prevent a request from the 8087 from being passed to the CPU. (The 8259A's normal priority-resolving activity can also block an interrupt request.) Finally, the CPU can exclude all interrupts tied to INTR by clearing its interrupt-enable flag (IF). In these two cases, the CPU can "escape" the endless wait only if another interrupt is recognized (if IF is cleared, the interrupt must arrive on NMI, the CPU's non-maskable interrupt line). Following execution of the interrupt procedure and resumption of the WAIT, the endless wait will be entered again, unless, as part of its response to the interrupt it recognizes, the CPU clears the interrupt path from the 8087.

A user-written exception handler can itself cause an unending wait. When the exception handler starts to run, the 8087 is suspended with its BUSY line active, waiting for the exception to be cleared, and interrupts on the CPU are disabled. If, in this condition, the exception handler issues any 8087 instruction, other than a no-wait form, the result will be an unending wait. To prevent this, the exception handler should clear the exception on the 8087 and enable interrupts on the CPU before executing any instruction that is preceded by a WAIT.

More generally, an instruction that is preceded by a WAIT (or an FWAIT instruction) should never be executed when CPU interrupts are disabled and there is any possibility that the 8087's BUSY line is active.

Status Lines

When the 8087 has control of the local bus, it emits signals on status lines S2-S0 to identify the type of bus cycle it is running. The 8087 generates the restricted (compared to a CPU) set of encodings shown in table S-8. These lines correspond exactly to the signals output by the 8086 and 8088 CPU's, and are normally decoded by an 8288 Bus Controller.

Table S-8. Bus Cycle Status Signals

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Type of Bus Cycle
1	0	1	Read Memory
1	1	0	Write Memory
1	1	1	Passive; no bus cycle

Status line S7 is currently identical to BHE of the same bus cycle, while S4 and S3 are both currently 1; however, these signals are reserved by Intel for possible future use. Status line S6 emits 1 and S5 emits 0.

S.7 Instruction Set

This section describes the operation of each of the 8087's 69 instructions. The first part of the section describes the function of each instruction in detail. For this discussion, the instructions are divided into six functional groups: data transfer, arithmetic, comparison, transcendental, constant, and processor control. The second part provides instruction attributes such as execution speed, bus transfers, and exceptions, as well as a coding example for each combination of operands accepted by the instruction. This information is concentrated in a table, organized alphabetically by instruction mnemonic, for easy reference.

Throughout this section, the instruction set is described as it appears to the ASM-86 programmer who is coding a program. Appendix A covers the actual machine instruction encodings, which are principally of use to those reading unformatted memory dumps, monitoring instruction fetches on the bus, or writing exception handlers.

The instruction descriptions in this section concentrate on describing the normal function of each operation. Table S-19 lists the exceptions that can occur for each instruction and table S-32 details the causes of exceptions as well as the 8087's masked responses.

The typical NDP instruction accepts one or twc operands as "inputs", operates on these, anc produces a result as an "output". Operands are

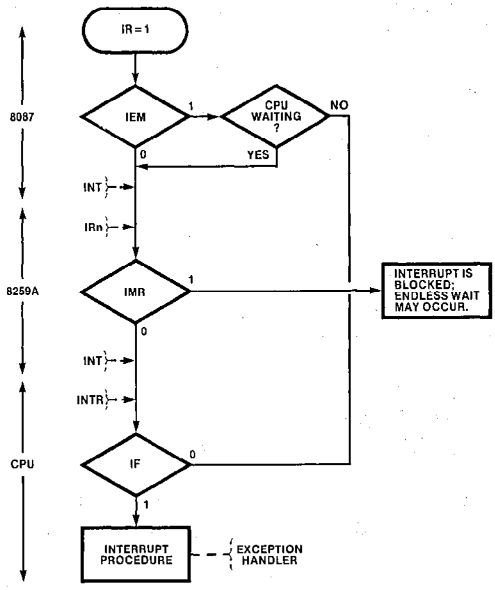


Figure S-17. Interrupt Request Path

most often (the contents of) register or memory locations. The operands of some instructions are predefined; for example, FSQRT always takes the square root of the number in the top stack element. Others allow, or require, the programmer to explicitly code the operand(s) along with the instruction mnemonic. Still others accept one explicit operand and one implicit operand, which is usually the top stack element.

Whether supplied by the programmer or utilized automatically, there are two basic types of operands, *sources* and *destinations*. A source operand simply supplies one of the "inputs" to an instruction; it is not altered by the instruction. Even when an instruction converts the source operand from one format to another (e.g., real to integer), the conversion is actually performed in an internal work area to avoid altering the source operand. A destination operand may also provide an "input" to an instruction. It is distinguished from a source operand, however, because its content may be altered when it receives the result produced by the operation; that is, the destination is replaced by the result.

Many instructions allow their operands to be coded in more than one way. For example, FADD (add real) may be written without operands, with only a source or with a destination and a source. The instruction descriptions in this section employ the simple convention of separating alternative operand forms with slashes; the slashes, however, are not coded. Consecutive slashes indicate an option of no explicit operands. The operands for FADD are thus described as:

//source/destination, source

This means that FADD may be written in any of three ways:

FADD
FADD *source*
FADD *destination, source*

When reading this section, it is important to bear in mind that memory operands may be coded with any of the CPU's memory addressing modes. To review these modes—direct, register indirect, based, indexed, based indexed—refer to sections 2.8 and 2.9. Table S-22 in this chapter also provides several addressing mode examples.

Data Transfer Instructions

These instructions (summarized in table S-9) move operands among elements of the register stack, and between the stack top and memory. Any of the seven data types can be converted to temporary real and loaded (pushed) onto the stack in a single operation; they can be stored to memory in the same manner. The data transfer instructions automatically update the 8087 tag word to reflect the register contents following the instruction.

FLD *source*

FLD (load real) loads (pushes) the source operand onto the top of the register stack. This is done by decrementing the stack pointer by one and then copying the content of the source to the new stack top. The source may be a register on the stack (ST(i)) or any of the real data types in memory. Short and long real source operands are converted to temporary real automatically. Coding FLD ST(0) duplicates the stack top.

Table S-9. Data Transfer Instructions

Real Transfers	
FLD	Load real
FST	Store real
FSTP	Store real and pop
FXCH	Exchange registers
Integer Transfers	
FILD	Integer load
FIST	Integer store
FISTP	Integer store and pop
Packed Decimal Transfers	
FBLD	Packed decimal (BCD) load
FBSTP	Packed decimal (BCD) store and pop

FST *destination*

FST (store real) transfers the stack top to the destination, which may be another register on the stack or a short or long real memory operand. If the destination is short or long real, the significand is rounded to the width of the destination according to the RC field of the control word, and the exponent is converted to the width and bias of the destination format.

If, however, the stack top is tagged special (it contains ∞, a NAN, or a denormal) then the stack top's significand is not rounded but is chopped (on the right) to fit the destination. Neither is the exponent converted, but it also is chopped on the right and transferred "as is". This preserves the value's identification as ∞ or a NAN (exponent all ones) or a denormal (exponent all zeros) so that it can be properly loaded and tagged later in the program if desired.

FSTP *destination*

FSTP (store real and pop) operates identically to FST except that the stack is popped following the transfer. This is done by tagging the top stack element empty and then incrementing ST. FSTP permits storing to a temporary real memory variable while FST does not. Coding FSTP ST(0) is equivalent to popping the stack with no data transfer.

FXCH *//destination*

FXCH (exchange registers) swaps the contents of the destination and the stack top registers. If the destination is not coded explicitly, ST(1) is used. Many 8087 instructions operate only on the stack top; FXCH provides a simple means of effectively using these instructions on lower stack elements. For example, the following sequence takes the square root of the third register from the top:

```
FXCH  ST(3)
FSQRT
FXCH  ST(3)
```

FILD *source*

FILD (integer load) converts the source memory operand from its binary integer format (word, short, or long) to temporary real and loads (pushes) the result onto the stack. The (new) stack top is tagged zero if all bits in the source were zero, and is tagged valid otherwise.

FIST *destination*

FIST (integer store) rounds the content of the stack top to an integer according to the RC field of the control word and transfers the result to the destination. The destination may define a word or short integer variable. Negative zero is stored in the same encoding as postive zero: 0000...00.

FISTP *destination*

FISTP (integer store and pop) operates like FIST and also pops the stack following the transfer. The destination may be any of the binary integer data types.

FBLD *source*

FBLD (packed decimal (BCD) load) converts the content of the source operand from packed decimal to temporary real and loads (pushes) the result onto the stack. The sign of the source is preserved, including the case where the value is negative zero. FBLD is an exact operation; the source is loaded with no rounding error.

The packed decimal digits of the source are assumed to be in the range 0-9H. The instruction does not check for invalid digits (A-FH) and the result of attempting to load an invalid encoding is undefined.

FBSTP *destination*

FBSTP (packed decimal (BCD) store and pop) converts the content of the stack top to a packed decimal integer, stores the result at the destination in memory, and pops the stack. FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then chopping. Users who are concerned about rounding may precede FBSTP with FRNDINT.

Arithmetic Instructions

The 8087's arithmetic instruction set (table S-10) provides a wealth of variations on the basic add, subtract, multiply, and divide operations, and a number of other useful functions. These range from a simple absolute value to a square root instruction that executes faster than ordinary divi-

Table S-10. Arithmetic Instructions

Addition	
FADD	Add real
FADDP	Add real and pop
FIADD	Integer add
Subtraction	
FSUB	Subtract real
FSUBP	Subtract real and pop
FISUB	Integer subtract
FSUBR	Subtract real reversed
FSUBRP	Subtract real reversed and pop
FISUBR	Integer subtract reversed
Multiplication	
FMUL	Multiply real
FMULP	Multiply real and pop
FIMUL	Integer multiply
Division	
FDIV	Divide real
FDIVP	Divide real and pop
FIDIV	Integer divide
FDIVR	Divide real reversed
FDIVRP	Divide real reversed and pop
FIDIVR	Integer divide reversed
Other Operations	
FSQRT	Square root
FSCALE	Scale
FPREM	Partial remainder
FRNDINT	Round to integer
FXTRACT	Extract exponent and significand
FABS	Absolute value
FCHS	Change sign

sion; 8087 programmers no longer need to spend valuable time eliminating square roots from algorithms because they run too slowly. Other arithmetic instructions perform exact modulo division, round real numbers to integers, and scale values by powers of two.

The 8087's basic arithmetic instructions (addition, subtraction, multiplication, and division) are designed to encourage the development of very efficient algorithms. In particular, they allow the programmer to minimize memory references and to make optimum use of the NDP register stack.

Table S-11 summarizes the available operation/operand forms that are provided for basic arithmetic. In addition to the four normal operations, two "reversed" instructions make subtraction and division "symmetrical" like addition and multiplication. The variety of instruction and operand forms give the programmer unusual flexibility:

- operands may be located in registers or memory;
- results may be deposited in a choice of registers;
- operands may be a variety of NDP data types: temporary real, long real, short real, short integer or word integer, with automatic conversion to temporary real performed by the 8087.

Five basic instruction forms may be used across all six operations, as shown in table S-11. The classical stack form may be used to make the 8087 operate like a classical stack machine. No operands are coded in this form, only the instruction mnemonic. The NDP picks the source operand from the stack top and the destination from the next stack element. It then pops the stack, performs the operation, and returns the result to the new stack top, effectively replacing the operands by the result.

The register form is a generalization of the classical stack form; the programmer specifies the stack top as one operand and any register on the stack as the other operand. Coding the stack top as the destination provides a convenient way to access a constant, held elsewhere in the stack, from the stack top. The converse coding (ST is the source operand) allows, for example, adding the top into a register used as an accumulator.

Often the operand in the stack top is needed for one operation but then is of no further use in the computation. The register pop form can be used to pick up the stack top as the source operand, and then discard it by popping the stack. Coding operands of ST(1),ST with a register pop mnemonic is equivalent to a classical stack operation: the top is popped and the result is left at the new top.

Table S-11. Basic Arithmetic Instructions and Operands

Instruction Form	Mnemonic Form	Operand Forms destination, source	ASM-86 Example	
Classical stack	F*op*	{ST(1),ST}	FADD	
Register	F*op*	ST(i),ST or ST,ST(i)	FSUB	ST,ST(3)
Register pop	F*op*P	ST(i),ST	FMULP	ST(2),ST
Real memory	F*op*	{ST,} short-real/long-real	FDIV	AZIMUTH
Integer memory	FI*op*	{ST,} word-integer/short-integer	FIDIV	N_PULSES

NOTES: Braces { } surround *implicit* operands; these are not coded, and are shown here for information only.

op =		
	ADD	destination ← destination + source
	SUB	destination ← destination − source
	SUBR	destination ← source − destination
	MUL	destination ← destination • source
	DIV	destination ← destination ÷ source
	DIVR	destination ← source ÷ destination

The two memory forms increase the flexibility of the 8087's arithmetic instructions. They permit a real number or a binary integer in memory to be used directly as a source operand. This is a very useful facility in situations where operands are not used frequently enough to justify holding them in registers. Note that any memory addressing mode may be used to define these operands, so they may be elements in arrays, structures or other data organizations, as well as simple scalars.

The six basic operations are discussed further in the next paragraphs, and descriptions of the remaining seven arithmetic operations follow.

Addition

FADD *//source/destination,source*
FADDP *destination,source*
FIADD *source*

The addition instructions (add real, add real and pop, integer add) add the source and destination operands and return the sum to the destination. The operand at the stack top may be doubled by coding:

```
FADD   ST,ST(0)
```

Normal Subtraction

FSUB *//source/destination,source*
FSUBP *destination,source*
FISUB *source*

The normal subtraction instructions (subtract real, subtract real and pop, integer subtract) subtract the source operand from the destination and return the difference to the destination.

Reversed Subtraction

FSUBR *//source/destination,source*
FSUBRP *destination,source*
FISUBR *source*

The reversed subtraction instructions (subtract real reversed, subtract real reversed and pop, integer subtract reversed) subtract the destination from the source and return the difference to the destination.

Multiplication

FMUL *//source/destination,source*
FMULP *destination,source*
FIMUL *source*

The multiplication instructions (multiply real, multiply real and pop, integer multiply) multiply the source and destination operands and return the product to the destination. Coding FMUL ST,ST(0) squares the content of the stack top.

Normal Division

FDIV *//source/destination,source*
FDIVP *destination,source*
FIDIV *source*

The normal division instructions (divide real, divide real and pop, integer divide) divide the destination by the source and return the quotient to the destination.

Reversed Division

FDIVR *//source/destination,source*
FDIVRP *destination,source*
FIDIVR *source*

The reversed division instructions (divide real reversed, divide real reversed and pop, integer divide reversed) divide the source operand by the destination and return the quotient to the destination.

FSQRT

FSQRT (square root) replaces the content of the top stack element with its square root. (Note: the square root of −0 is defined to be −0.)

FSCALE

FSCALE (scale) interprets the value contained in ST(1) as an integer, and adds this value to the exponent of the number in ST. This is equivalent to:

$$ST \leftarrow ST \bullet 2^{ST(1)}$$

thus, FSCALE provides rapid multiplication or division by integral powers of 2. It is particularly useful for scaling the elements of a vector.

Note that FSCALE assumes the scale factor in ST(1) is an integral value in the range $-2^{15} \leqslant X < 2^{15}$. If the value is not integral, but is in-range and is greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude, i.e., it chops the value toward 0. If the value is out of range, or $0 < | X | < 1$, the instruction will produce an undefined result and will not signal an exception. The recommended practice is to load the scale factor from a word integer to ensure correct operation.

FPREM

FPREM (partial remainder) performs modulo division of the top stack element by the next stack element, i.e., ST(1) is the modulus. FPREM produces an *exact* result; the precision exception does not occur. The sign of the remainder is the same as the sign of the original dividend.

FPREM operates by performing successive scaled subtractions; obtaining the exact remainder when the operands differ greatly in magnitude can consume large amounts of execution time. Since the 8087 can only be preempted between instructions, the remainder function could seriously increase interrupt latency in these cases. Accordingly, the instruction is designed to be executed iteratively in a software-controlled loop.

FPREM can reduce a magnitude difference of up to 2^{64} in one execution. If FPREM produces a remainder that is less than the modulus, the function is complete and bit C2 of the status word condition code is cleared. If the function is incomplete, C2 is set to 1; the result in ST is then called the partial remainder. Software can inspect C2 by storing the status word following execution of FPREM and re-execute the instruction (using the partial remainder in ST as the dividend), until C2 is cleared. Alternatively, a program can determine when the function is complete by comparing ST to ST(1). If ST>ST(1) then FPREM must be executed again; if ST=ST(1) then the remainder is 0; if ST<ST(1) then the remainder is ST. A higher priority interrupting routine which needs the 8087 can force a context switch between the instructions in the remainder loop.

An important use for FPREM is to reduce arguments (operands) of periodic transcendental functions to the range permitted by these instructions. For example, the FPTAN (tangent) instruction requires its argument to be less than $\pi/4$. Using $\pi/4$ as a modulus, FPREM will reduce an argument so that it is in range of FPTAN. Because FPREM produces an exact result, the argument reduction does *not* introduce roundoff error into the calculation, even if several iterations are required to bring the argument into range. (The rounding of π does not create the effect of a rounded argument, but of a rounded period.)

FPREM also provides the least-significant three bits of the quotient generated by FPREM (in C_3, C_1, C_0). This is also important for transcendental argument reduction since it locates the original angle in the correct one of eight π/4 segments of the unit circle.

FRNDINT

FRNDINT (round to integer) rounds the top stack element to an integer. For example, assume that ST contains the 8087 real number encoding of the decimal value 155.625. FRNDINT will change the value to 155 if the RC field of the control word is set to down or chop, or to 156 if it is set to up or nearest.

FXTRACT

FXTRACT (extract exponent and significand) "decomposes" the number in the stack top into two numbers that represent the actual value of the operand's exponent and significand fields. The "exponent" replaces the original operand on the stack and the "significand" is pushed onto the stack. Following execution of FXTRACT, ST (the new stack top) contains the value of the original significand expressed as a real number: its sign is the same as the operand's, its exponent is 0 true (16,383 or 3FFFH biased), and its significand is identical to the original operand's. ST(1) contains the value of the original operand's true (unbiased) exponent expressed as a real number. If the original operand is zero, FXTRACT produces zeros in ST and ST(1) and *both* are signed as the original operand.

To clarify the operation of FXTRACT, assume ST contains a number whose true exponent is +4 (i.e., its exponent field contains 4003H). After executing FXTRACT, ST(1) will contain the real number +4.0; its sign will be positive, its exponent field will contain 4001H (+2 true) and its significand field will contain $1_\Delta00...00$B. In other words, the value in ST(1) will be $1.0 \times 2^2 = 4$. If ST contains an operand whose true exponent is −7 (i.e., its exponent field contains 3FF8H), then FXTRACT will return an "exponent" of −7.0; after the instruction executes, ST(1)'s sign and exponent fields will contain C001H (negative sign, true exponent of 2) and its significand will be $1_\Delta1100...00$B. In other words the value in ST(1) will be $-1.11 \times 2^2 = -7.0$. In both cases, following FXTRACT, ST's sign and significand fields will be the same as the original operand's, and its exponent field will contain 3FFFH, (0 true).

FXTRACT is useful in conjunction with FBSTP for converting numbers in 8087 temporary real format to decimal representations (e.g., for printing or displaying). It can also be useful for debugging since it allows the exponent and significand parts of a real number to be examined separately.

FABS

FABS (absolute value) changes the top stack element to its absolute value by making its sign positive.

FCHS

FCHS (change sign) complements (reverses) the sign of the top stack element.

Comparison Instructions

Each of these instructions (table S-12) analyzes the top stack element, often in relationship to another operand, and reports the result in the status word condition code. The basic operations are compare, test (compare with zero), and examine (report tag, sign, and normalization). Special forms of the compare operation are provided to optimize algorithms by allowing direct comparisons with binary integers and real numbers in memory, as well as popping the stack after a comparison.

The FSTSW (store status word) instruction may be used following a comparison to transfer the condition code to memory for inspection. Section S.10 contains an example of using this technique to implement conditional branching.

Note that instructions other than those in the comparison group may update the condition code. To insure that the status word is not altered inadvertently, store it immediately following a comparison operation.

FCOM *//source*

FCOM (compare real) compares the stack top to the source operand. The source operand may be a register on the stack, or a short or long real memory operand. If an operand is not coded, ST is compared to ST(1). Positive and negative forms of zero compare identically as if they were unsigned. Following the instruction, the condition codes reflect the order of the operands as follows:

C3	C0	Order
0	0	ST > source
0	1	ST < source
1	0	ST = source
1	1	ST ? source

NANs and ∞ (projective) cannot be compared and return C3=C0=1 as shown above.

Table S-12. Comparison Instructions

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

FCOMP *//source*

FCOMP (compare real and pop) operates like FCOM, and in addition pops the stack.

FCOMPP

FCOMPP (compare real and pop twice) operates like FCOM and additionally pops the stack twice, discarding both operands. The comparison is of the stack top to ST(1); no operands may be explicitly coded.

FICOM *source*

FICOM (integer compare) converts the source operand, which may reference a word or short binary integer variable, to temporary real and compares the stack top to it.

FICOMP *source*

FICOMP (integer compare and pop) operates identically to FICOM and additionally discards the value in ST by popping the stack.

FTST

FTST (test) tests the top stack element by comparing it to zero. The result is posted to the condition codes as follows:

C3	C0	Result
0	0	ST is positive and nonzero
0	1	ST is negative and nonzero
1	0	ST is zero (+ or −)
1	1	ST is not comparable (i.e., it is a NAN or projective ∞)

FXAM

FXAM (examine) reports the content of the top stack element as positive/negative and NAN/unnormal/denormal/normal/zero, or empty. Table S-13 lists and interprets all the condition code values that FXAM generates. Although four different encodings may be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all encodings. Bits C2 and C1 should be ignored when examining for empty.

Transcendental Instructions

The instructions in this group (table S-14) perform the time-consuming *core calculations* for all common trigonometric, inverse trigonometric, hyperbolic, inverse hyperbolic, logarithmic and exponential functions. Prologue and epilogue software may be used to reduce arguments to the range accepted by the instructions and to adjust the result to correspond to the original arguments if necessary. The transcendentals operate on the top one or two stack elements and they return their results to the stack also.

Table S-13. FXAM Condition Code Settings

Condition Code				Interpretation
C3	C2	C1	C0	
0	0	0	0	+ Unnormal
0	0	0	1	+ NAN
0	0	1	0	− Unnormal
0	0	1	1	− NAN
0	1	0	0	+ Normal
0	1	0	1	+ ∞
0	1	1	0	− Normal
0	1	1	1	− ∞
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	− 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty
1	1	1	0	− Denormal
1	1	1	1	Empty

Table S-14. Transcendental Instructions

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	2^X-1
FYL2X	$Y \bullet \log_2 X$
FYL2XP1	$Y \bullet \log_2(X+1)$

The transcendental instructions assume that their operands are *valid and in-range*. The instruction descriptions in this section provide the range of each operation. To be considered valid, an operand to a transcendental must be normalized; denormals, unnormals, infinities and NANs are considered invalid. (Zero operands are accepted by some functions and are considered out-of-range by others.) If a transcendental operand is invalid or out-of-range, the instruction will produce an undefined result without signalling an exception. It is the programmer's responsibility to ensure that operands are valid and in-range before executing a transcendental. For periodic functions, FPREM may be used to bring a valid operand into range.

FPTAN

FPTAN (partial tangent) computes the function $Y/X = \text{TAN}(\Theta)$. Θ is taken from the top stack element; it must lie in the range $0 < \Theta < \pi/4$. The result of the operation is a ratio; Y replaces Θ in the stack and X is pushed, becoming the new stack top.

The ratio result of FPTAN and the ratio argument of FPATAN are designed to optimize the calculation of the other trigonometric functions, including SIN, COS, ARCSIN and ARCCOS. These can be derived from TAN and ARCTAN via standard trigonometric identities.

FPATAN

FPATAN (partial arctangent) computes the function $\Theta = \text{ARCTAN}(Y/X)$. X is taken from the top stack element and Y from ST(1). Y and X must observe the inequality $0 < Y < X < \infty$. The instruction pops the stack and returns Θ to the (new) stack top, overwriting the Y operand.

F2XM1

F2XM1 (2 to the X minus 1) calculates the function $Y = 2^X - 1$. X is taken from the stack top and must be in the range $0 \leqslant X \leqslant 0.5$. The result Y replaces X at the stack top.

This instruction is designed to produce a very accurate result even when X is close to zero. To obtain $Y=2^X$, add 1 to the result delivered by F2XM1.

The following formulas show how values other than 2 may be raised to a power of X:

$$10^x = 2^{x \bullet \text{LOG}_2 10}$$

$$e^x = 2^{x \bullet \text{LOG}_2 e}$$

$$y^x = 2^{x \bullet \text{LOG}_2 y}$$

As shown in the next section, the 8087 has built-in instructions for loading the constants $LOG_2 10$ and $LOG_2 e$, and the FYL2X instruction may be used to calculate $X \bullet LOG_2 Y$.

FYL2X

FYL2X (Y log base 2 of X) calculates the function $Z = Y \bullet LOG_2 X$. X is taken from the stack top and Y from ST(1). The operands must be in the ranges $0 < X < \infty$ and $-\infty < Y < +\infty$. The instruction pops the stack and returns Z at the (new) stack top, replacing the Y operand.

This function optimizes the calculation of log to any base other than two since a multiplication is always required:

$$LOG_n 2 \bullet LOG_2 X$$

FYL2XP1

FYL2XP1 (Y log base 2 of (X + 1)) calculates the function $Z = Y \bullet LOG_2\ (X+1)$. X is taken from the stack top and must be in the range $0 < |X| < (1 -(\sqrt{2}/2))$. Y is taken from ST(1) and must be in the range $-\infty < Y < \infty$. FYL2XP1 pops the stack and returns Z at the (new) stack top, replacing Y.

This instruction provides improved accuracy over FYL2X when computing the log of a number very close to 1, for example $1 + \varepsilon$ where $\varepsilon << 1$. Providing ε rather than $1 + \varepsilon$ as the input to the function allows more significant digits to be retained.

Constant Instructions

Each of these instructions (table S-15) loads (pushes) a commonly-used constant onto the stack. The values have full temporary real precision (64 bits) and are accurate to approximately 19 decimal digits. Since a temporary real constant occupies 10 memory bytes, the constant instructions, which are only two bytes long, save storage and improve execution speed, in addition to simplifying programming.

Table S-15. Constant Instructions

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

FLDZ

FLDZ (load zero) loads (pushes) +0.0 onto the stack.

FLD1

FLD1 (load one) loads (pushes) +1.0 onto the stack.

FLDPI

FLDPI (load π) loads (pushes) π onto the stack.

FLDL2T

FLDL2T (load log base 2 of 10) loads (pushes) the value $LOG_2 10$ onto the stack.

FLDL2E

FLDL2E (load log base 2 of e) loads (pushes) the value $LOG_2 e$ onto the stack.

FLDLG2

FLDLG2 (load log base 10 of 2) loads (pushes) the value $LOG_{10} 2$ onto the stack.

FLDLN2

FLDLN2 (load log base e of 2) loads (pushes) the value $LOG_e 2$ onto the stack.

Processor Control Instructions

Most of these instructions (table S-16) are not used in computations; they are provided principally for system-level activities. These include initialization, exception handling and task switching.

As shown in table S-16, an alternate mnemonic is available for many of the processor control instructions. This mnemonic, distinguished by a second character of "N", instructs the assembler to *not* prefix the instruction with a CPU WAIT instruction (instead, a CPU NOP precedes the instruction). This "no-wait" form is intended for use in critical code regions where a WAIT instruction might precipitate an endless wait. Thus, when CPU interrupts are disabled, and the NDP can potentially generate an interrupt, the no-wait form should be used. When CPU interrupts are enabled, as will normally be the case when an application task is running, the "wait" forms of these instructions should be used.

Except for FNSTENV and FNSAVE, all instructions which provide a no-wait mnemonic are self-synchronizing and can be executed back-to-back in any combination without intervening FWAITs. These instructions can be executed by the 8087 CU while the NEU is busy with a previously decoded instruction. To insure that the processor control instruction executes after completion of any operation in progress in the NEU, the "wait" form of that instruction should be used.

FINIT/FNINIT

FINIT/FNINIT (initialize processor) performs the functional equivalent of a hardware RESET (see section S.6), except that it does not affect the instruction fetch synchronization of the 8087 and its CPU.

For compatibility with the 8087 emulator, a system should call the INIT87 procedure in lieu of executing FINIT/FNINIT when the processor is first initialized (see section S.8 for details). Note that if FNINIT is executed while a previous 8087 memory referencing instruction is running, 8087 bus cycles in progress will be aborted.

FDISI/FNDISI

FDISI/FNDISI (disable interrupts) sets the interrupt enable mask in the control word and prevents the NDP from issuing an interrupt request.

Table S-16. Processor Control Instructions

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environment
FLDENV	Load environment
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack pointer
FDECSTP	Decrement stack pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

FENI/FNENI

FENI/FNENI (enable interrupts) clears the interrupt enable mask in the control word, allowing the 8087 to generate interrupt requests.

FLDCW *source*

FLDCW (load control word) replaces the current processor control word with the word defined by the source operand. This instruction is typically used to establish, or change, the 8087's mode of operation. Note that if an exception bit in the status word is set, loading a new control word that unmasks that exception and clears the interrupt enable mask will generate an immediate interrupt request before the next instruction is executed. When changing modes, the recommended procedure is to first clear any exceptions and then load the new control word.

FSTCW/FNSTCW *destination*

FSTCW/FNSTCW (store control word) writes the current processor control word to the memory location defined by the destination.

FSTSW/FNSTSW *destination*

FSTSW/FNSTCW (store status word) writes the current value of the 8087 status word to the destination operand in memory. The instruction has many uses:

- to implement conditional branching following a comparison or FPREM instruction (FSTSW);
- to poll the 8087 to determine if it is busy (FNSTSW);
- to invoke exception handlers in environments that do not use interrupts (FSTSW).

FCLEX/FNCLEX

FCLEX/FNCLEX (clear exceptions) clears all exception flags, the interrupt request flag and the busy flag in the status word. As a consequence, the 8087's INT and BUSY lines go inactive. An exception handler must issue this instruction before returning to the interrupted computation, or another interrupt request will be generated immediately, and an endless loop may result.

FSAVE/FNSAVE *destination*

FSAVE/FNSAVE (save state) writes the full 8087 state—environment plus register stack—to the memory location defined by the destination operand. Figure S-18 shows the layout of the 94-byte save area; typically the instruction will be coded to save this image on the CPU stack. If an instruction is executing in the 8087 NEU when FNSAVE is decoded, the CPU queues the FNSAVE and delays its execution until the running instruction completes normally or encounters an unmasked exception. Thus, the save image reflects the state of the NDP following the completion of any running instruction. After writing the state image to memory, FSAVE/FNSAVE initializes the 8087 as if FINIT/FNINIT had been executed.

FSAVE/FNSAVE is useful whenever a program wants to save the current state of the NDP and initialize it for a new routine. Three examples are:

- an operating system needs to perform a context switch (suspend the task that had been running and give control to a new task);
- an interrupt handler needs to use the 8087;
- an application task wants to pass a "clean" 8087 to a subroutine.

FNSAVE must be "protected" by executing it in a critical region, i.e., with CPU interrupts disabled. This prevents an interrupt handler from executing a second FNSAVE (or other "no-wait" processor control instruction that references memory) which could destroy the first FNSAVE if it is queued in the 8087. An FWAIT should be executed before CPU interrupts are enabled or any subsequent 8087 instruction is executed. (Because the FNSAVE initializes the NDP, there is no danger of the FWAIT causing an endless wait.) Other CPU instructions may be executed between the FNSAVE and the FWAIT; this parallel execution will reduce interrupt latency if the FNSAVE is queued in the 8087.

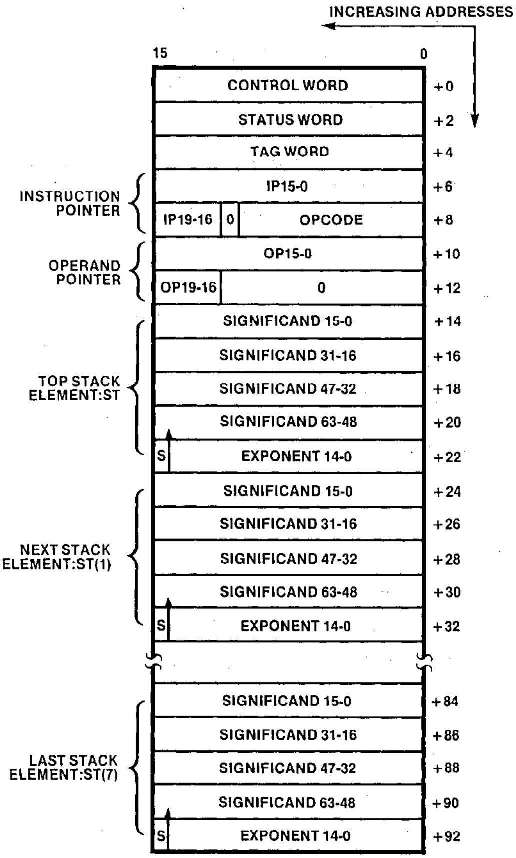
FRSTOR *source*

FRSTOR (restore state) reloads the 8087 from the 94-byte memory area defined by the source operand. This information should have been written by a previous FSAVE/FNSAVE instruction and not altered by any other instruction. CPU instructions (that do not reference the save image) may immediately follow FRSTOR, but no NDP instruction should be without an intervening FWAIT or an assembler-generated WAIT.

Note that the 8087 "reacts" to its new state at the conclusion of the FRSTOR; it will for example, generate an immediate interrupt request if the exception and mask bits in the memory image so indicate.

FSTENV/FNSTENV *destination*

FSTENV/FNSTENV (store environment) writes the 8087's basic status—control, status and tag words, and exception pointers—to the memory location defined by the destination operand. Typically the environment is saved on the CPU stack. FSTENV/FNSTENV is often used by exception handlers because it provides access to the exception pointers which identify the offending instruction and operand. After saving the environment, FSTENV/FNSTENV sets all exception masks in the processor; it does not affect the interrupt-enable mask. Figure S-19 shows the format of the environment data in memory. If FNSTENV is decoded while another instruction is executing concurrently in the NEU, the 8087 queues the FNSTENV and does not store the environment until the other instruction has completed. Thus, the data saved by the instruction reflects the 8087 after any previously decoded instruction has been executed.



NOTES:
S = Sign
Bit 0 of each field is rightmost, least significant bit of corresponding register field.
Bit 63 of significand is integer bit (assumed binary point is immediately to the right).

Figure S-18. FSAVE/FRSTOR Memory Layout

FSTENV/FNSTENV must be allowed to complete before any other 8087 instruction is decoded. When FSTENV is coded, an explicit FWAIT, or assembler-generated WAIT, should precede any subsequent 8087 instruction. An FNSTENV must be executed in a critical region that is protected from interruption, in the same manner as FNSAVE. (There is no risk of the following FWAIT causing an endless wait, because FNSTENV masks all exceptions, thereby preventing an interrupt request from the 8087.)

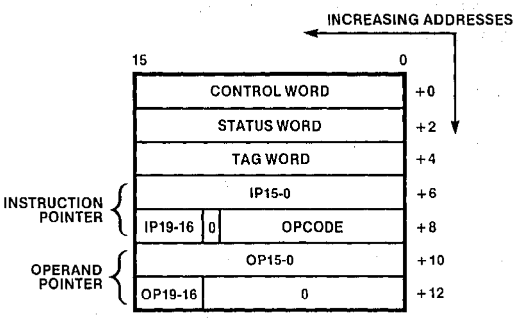


Figure S-19. FSTENV/FLDENV Memory Layout

FLDENV *source*

FLDENV (load environment) reloads the 8087 environment from the memory area defined by the source operand. This data should have been written by a previous FSTENV/FNSTENV instruction. CPU instructions (that do not reference the environment image) may immediately follow FLDENV, but no subsequent NDP instruction should be executed without an intervening FWAIT or assembler-generated WAIT.

Note that loading an environment image that contains an unmasked exception will cause an immediate interrupt request from the 8087 (assuming IEM=0 in the environment image).

FINCSTP

FINCSTP (increment stack pointer) adds 1 to the stack top pointer (ST) in the status word. It does not alter tags or register contents, nor does it transfer data. It is not equivalent to popping the stack since it does not set the tag of the previous stack top to empty. Incrementing the stack pointer when ST=7 produces ST=0.

FDECSTP

FDECSTP (decrement stack pointer) subtracts 1 from ST, the stack top pointer in the status word. No tags or registers are altered, nor is any data transferred. Executing FDECSTP when ST=0 produces ST=7.

FFREE *destination*

FFREE (free register) changes the destination register's tag to empty; the content of the register is unaffected.

FNOP

FNOP (no operation) stores the stack top to the stack top (FST ST,ST(0)) and thus effectively performs no operation.

FWAIT (CPU instruction)

FWAIT is not actually an 8087 instruction, but an alternate mnemonic for the CPU WAIT instruction described in section 2.8. The FWAIT mnemonic should be coded whenever the programmer wants to synchronize the CPU to the NDP, that is, to suspend further instruction decoding until the NDP has completed the current instruction. *A CPU instruction should not attempt to access a memory operand that has been read or written by a previous 8087 instruction until the 8087 instruction has completed*. The following coding shows how FWAIT can be used to force the CPU instruction to wait for the 8087:

```
FNSTSW   STATUS
FWAIT    ;Wait for FNSTSW
MOV      AX,STATUS
```

Programmers should not code WAIT to synchronize the CPU and the NDP. The routines that alter an object program for 8087 emulation eliminate FWAITs (and assembler-generated WAITs) but do not change any explicitly coded WAITs. The program will wait forever if a WAIT is encountered in emulated execution, since there is no 8087 to drive the CPU's $\overline{\text{TEST}}$ pin active.

Instruction Set Reference Information

Table S-19 lists the operating characteristics of all the 8087 instructions. There is one table entry for each instruction mnemonic; the entries are in alphabetical order for quick lookup. Each entry provides the general operand forms accepted by the instruction as well as a list of all exceptions that may be detected during the operation.

There is one entry for each combination of operand types that can be coded with the mnemonic. Table S-17 explains the operand identifiers allowed in table S-19. Following this entry are columns that provide execution time in clocks, the number of bus transfers run during the operation, the length of the instruction in bytes, and an ASM-86 coding sample.

Table S-17. Key to Operand Types

Identifier	Explanation
ST	Stack top; the register currently at the top of the stack.
ST(i)	A register in the stack i (0≤i≤7) stack elements from the top. ST(1) is the next-on-stack register, ST(2) is below ST(1), etc.
Short-real	A short real (32 bits) number in memory.
Long-real	A long real (64 bits) number in memory.
Temp-real	A temporary real (80 bits) number in memory.
Packed-decimal	A packed decimal integer (18 digits, 10 bytes) in memory.
Word-integer	A word binary integer (16 bits) in memory.
Short-integer	A short binary integer (32 bits) in memory.
Long-integer	A long binary integer (64 bits) in memory.
nn-bytes	A memory area nn bytes long.

Execution Time

The execution of an 8087 instruction involves three principal activities, each of which may contribute to the total duration (execution time) of the operation:

- Instruction fetch
- Instruction execution
- Operand transfer

The CPU and NDP simultaneously prefetch and queue their common instruction stream from memory. This activity is performed during spare bus cycles and proceeds in parallel with the execution of instructions from the queue. Because of their complexity, 8087 instructions typically take much longer to execute than to fetch. This means that in a typical sequence of 8087 instructions the processors have a relatively large amount of time available to maintain full instruction queues. Instruction fetching is therefore fully overlapped with execution and does not contribute to the overall duration of a series of instructions. Fetch time does become apparent when a CPU jump or call instruction alters the normal sequential execution. This empties the queues and delays execution of the target instruction until it is fetched from memory. The time required to fetch the instruction depends on its length, the type of CPU, and, if the CPU is an 8086, whether the instruction is located at an even or odd address. (Slow memories, which force the insertion of wait states in bus cycles, and the bus activities of other processors in the system, may also lengthen fetch time.) Section 2.7 covers this topic in more detail.

Table S-19 quotes a typical execution time and a range for each instruction. Dividing the figures in the table by 5 (assuming a 5 MHz clock) produces execution time in microseconds. The typical case is an estimate for operand values that normally characterize most applications. The range encompasses best- and worst-case operand values that may be found in extreme circumstances. Where applicable, the figures *include* all overhead incurred by the CPU's execution of the ESC instruction, local bus arbitration (request/grant time), and the average overhead imposed by a preceding WAIT instruction (half of the 5-clock cycle that it uses to examine the $\overline{\text{TEST}}$ pin).

The execution times assume that no exceptions are detected. Invalid operation, denormalized (unmasked), and zerodivide exceptions usually decrease execution time from the typical figure, but it will still fall within the quoted range. The precision exception has no effect on execution time. Unmasked overflow and underflow, and masked denormalized exceptions, impose the penalties shown in table S-18. Absolute worst-case execution time is therefore the high range figure plus the largest penalty that may be encountered.

For instructions that transfer operands to or from memory, the execution times in table S-19 show that the time required for the CPU to calculate the operand's effective address (EA) should be added. Effective address calculation time varies according to addressing mode; table 2-20 supplies the figures.

Table S-18. Execution Penalties

Exception	Additional Clocks
Overflow (unmasked)	14
Underflow (unmasked)	16
Denormalized (masked)	33

Bus Transfers

Instructions that reference memory execute bus cycles to transfer operands. Each transfer requires one bus cycle. The number of transfers depends on the length of the operand, the type of CPU, and the alignment of the operand if the CPU is an 8086. The figures in table S-19 *include* the "dummy read" transfer(s) performed by the CPU in its execution of the escape instruction that corresponds to the 8087 instruction. The first 8086 figure is for even-addressed operands, and the second is for odd-addressed operands.

A bus cycle (transfer) consumes four clocks if the bus is immediately available and if the memory is running at processor speed, without wait states. Additional time is required if slow memories are employed, because these insert wait states into the bus cycle. In multiprocessor environments, the bus may not be available immediately if a higher priority processor is using it; this also can increase effective transfer time.

Instruction Length

Instructions that do not reference memory are two bytes long. Memory reference instructions vary between two and four bytes. The third and fourth bytes are used for 8- or 16-bit displacement values; the assembler generates the short displacement whenever possible. No displacements are required in memory references that use only CPU register contents to calculate an operand's effective address.

Note that the lengths quoted in table S-19 do not include the one byte CPU WAIT instruction that the assembler automatically inserts in front of all NDP instructions (except those coded with a "no-wait" mnemonic).

Table S-19. Instruction Set Reference Data

FABS

FABS (no operands)
Absolute value

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	14	10-17	0	0	2	FABS

FADD

FADD //source/destination,source
Add real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	85	70-100	0	0	2	FADD ST,ST(4)
short-real	105+EA	90-120+EA	2/4	4	2-4	FADD AIR_TEMP [SI]
long-real	110+EA	95-125+EA	4/6	8	2-4	FADD [BX].MEAN

FADDP

FADDP destination,source
Add real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FADDP ST(2),ST

FBLD

FBLD source
Packed decimal (BCD) load

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
packed-decimal	300+EA	290-310+EA	5/7	10	2-4	FBLD YTD_SALES

Table S-19. Instruction Set Reference Data (Cont'd.)

FBSTP

FBSTP destination
Packed decimal (BCD) store and pop

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
packed-decimal	530+EA	520-540+EA	6/8	12	2-4	FBSTP [BX].FORECAST

FCHS

FCHS (no operands)
Change sign

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	15	10-17	0	0	2	FCHS

FCLEX/FNCLEX

FCLEX (no operands)
Clear exceptions

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FNCLEX

FCOM

FCOM //source
Compare real

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i)	45	40-50	0	0	2	FCOM ST(1)
short-real	65+EA	60-70+EA	2/4	4	2-4	FCOM [BP].UPPER_LIMIT
long-real	70+EA	65-75+EA	4/6	8	2-4	FCOM WAVELENGTH

FCOMP

FCOMP //source
Compare real and pop

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i)	47	42-52	0	0	2	FCOMP ST(2)
short-real	68+EA	63-73+EA	2/4	4	2-4	FCOMP [BP+2].N_READINGS
long-real	72+EA	67-77+EA	4/6	8	2-4	FCOMP DENSITY

Table S-19. Instruction Set Reference Data (Cont'd.)

FCOMPP

FCOMPP (no operands)
Compare real and pop twice

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	50	45-55	0	0	2	FCOMPP

FDECSTP

FDECSTP (no operands)
Decrement stack pointer

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	9	6-12	0	0	2	FDECSTP

FDISI/FNDISI

FDISI (no operands)
Disable interrupts

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FDISI

FDIV

FDIV //source/destination,source
Divide real

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i),ST	198	193-203	0	0	2	FDIV
short-real	220+EA	215-225+EA	2/4	4	2-4	FDIV DISTANCE
long-real	225+EA	220-230+EA	4/6	8	2-4	FDIV ARC [DI]

FDIVP

FDIVP destination,source
Divide real and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	202	197-207	0	0	2	FDIVP ST(4),ST

Table S-19. Instruction Set Reference Data (Cont'd.)

FDIVR

FDIVR //source/destination,source
Divide real reversed

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	199	194-204	0	0	2	FDIVR ST(2),ST
short-real	221+EA	216-226+EA	2/4	6	2-4	FDIVR [BX].PULSE_RATE
long-real	226+EA	221-231+EA	4/6	8	2-4	FDIVR RECORDER.FREQUENCY

FDIVRP

FDIVRP destination,source
Divide real reversed and pop

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	203	198-208	0	0	2	FDIVRP ST(1),ST

FENI/FNENI

FENI (no operands)
Enable interrupts

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FNENI

FFREE

FFREE destination
Free register

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	11	9-16	0	0	2	FFREE ST(1)

FIADD

FIADD source
Integer add

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	102-137+EA	1/2	2	2-4	FIADD DISTANCE_TRAVELLED
short-integer	125+EA	108-143+EA	2/4	4	2-4	FIADD PULSE_COUNT [SI]

Table S-19. Instruction Set Reference Data (Cont'd.)

FICOM

FICOM source
Integer compare

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	80+EA	72-86+EA	1/2	2	2-4	FICOM TOOL.N_PASSES
short-integer	85+EA	78-91+EA	2/4	4	2-4	FICOM [BP+4].PARM_COUNT

FICOMP

FICOMP source
Integer compare and pop

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	82+EA	74-88+EA	1/2	2	2-4	FICOMP [BP].LIMIT [SI]
short-integer	87+EA	80-93+EA	2/4	4	2-4	FICOMP N_SAMPLES

FIDIV

FIDIV source
Integer divide

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	230+EA	224-238+EA	1/2	2	2-4	FIDIV SURVEY.OBSERVATIONS
short-integer	236+EA	230-243+EA	2/4	4	2-4	FIDIV RELATIVE_ANGLE [DI]

FIDIVR

FIDIVR source
Integer divide reversed

Exceptions: I, D, Z, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	230+EA	225-239+EA	1/2	2	2-4	FIDIVR [BP].X_COORD
short-integer	237+EA	231-245+EA	2/4	4	2-4	FIDIVR FREQUENCY

FILD

FILD source
Integer load

Exception: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	50+EA	46-54+EA	1/2	2	2-4	FILD [BX].SEQUENCE
short-integer	56+EA	52-60+EA	2/4	4	2-4	FILD STANDOFF [DI]
long-integer	64+EA	60-68+EA	4/6	8	2-4	FILD RESPONSE.COUNT

Table S-19. Instruction Set Reference Data (Cont'd.)

FIMUL

FIMUL source
Integer multiply

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	130+EA	124-138+EA	1/2	2	2-4	FIMUL BEARING
short-integer	136+EA	130-144+EA	2/4	4	2-4	FIMUL POSITION.Z__AXIS

FINCSTP

FINCSTP (no operands)
Increment stack pointer

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	9	6-12	0	0	2	FINCSTP

FINIT/FNINIT

FINIT (no operands)
Initialize processor

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	5	2-8	0	0	2	FINIT

FIST

FIST destination
Integer store

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	86+EA	80-90+EA	2/4	4	2-4	FIST OBS.COUNT [SI]
short-integer	88+EA	82-92+EA	3/5	6	2-4	FIST [BP].FACTORED__PULSES

FISTP

FISTP destination
Integer store and pop

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	88+EA	82-92+EA	2/4	4	2-4	FISTP [BX].ALPHA__COUNT [SI]
short-integer	90+EA	84-94+EA	3/5	6	2-4	FISTP CORRECTED__TIME
long-integer	100+EA	94-105+EA	5/7	10	2-4	FISTP PANEL.N__READINGS

Table S-19. Instruction Set Reference Data (Cont'd.)

FISUB

FISUB source
Integer subtract

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	102-137+EA	1/2	2	2-4	FISUB BASE_FREQUENCY
short-integer	125+EA	108-143+EA	2/4	4	2-4	FISUB TRAIN_SIZE [DI]

FISUBR

FISUBR source
Integer subtract reversed

Exceptions: I, D, O, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
word-integer	120+EA	103-139+EA	1/2	2	2-4	FISUBR FLOOR [BX] [SI]
short-integer	125+EA	109-144+EA	2/4	4	2-4	FISUBR BALANCE

FLD

FLD source
Load real

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	20	17-22	0	0	2	FLD ST(0)
short-real	43+EA	38-56+EA	2/4	4	2-4	FLD READING [SI].PRESSURE
long-real	46+EA	40-60+EA	4/6	8	2-4	FLD [BP].TEMPERATURE
temp-real	57+EA	53-65+EA	5/7	10	2-4	FLD SAVEREADING

FLDCW

FLDCW source
Load control word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	10+EA	7-14+EA	1/2	2	2-4	FLDCW CONTROL_WORD

FLDENV

FLDENV source
Load environment

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
14-bytes	40+EA	35-45+EA	7/9	14	2-4	FLDENV [BP+6]

Mnemonics © Intel 1980

Table S-19. Instruction Set Reference Data (Cont'd.)

FLDLG2

FLDLG2 (no operands)
Load $\log_{10} 2$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	21	18-24	0	0	2	FLDLG2

FLDLN2

FLDLN2 (no operands)
Load $\log_e 2$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	20	17-23	0	0	2	FLDLN2

FLDL2E

FLDL2E (no operands)
Load $\log_2 e$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	18	15-21	0	0	2	FLDL2E

FLDL2T

FLDL2T (no operands)
Load $\log_2 10$

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	19	16-22	0	0	2	FLDL2T

FLDPI

FLDPI (no operands)
Load π

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	19	16-22	0	0	2	FLDPI

Table S-19. Instruction Set Reference Data (Cont'd.)

FLDZ

FLDZ (no operands)
Load +0.0

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	14	11-17	0	0	2	FLDZ

FLD1

FLD1 (no operands)
Load +1.0

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	18	15-21	0	0	2	FLD1

FMUL

FMUL //source/destination,source
Multiply real

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(i),ST/ST,ST(i)[1]	97	90-105	0	0	2	FMUL ST,ST(3)
//ST(i),ST/ST,ST(i)	138	130-145	0	0	2	FMUL ST,ST(3)
short-real	118+EA	110-125+EA	2/4	4	2-4	FMUL SPEED_FACTOR
long-real[1]	120+EA	112-126+EA	4/6	8	2-4	FMUL [BP].HEIGHT
long-real	161+EA	154-168+EA	4/6	8	2-4	FMUL [BP].HEIGHT

[1] occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (e.g., it was loaded from a short-real memory operand).

FMULP

FMULP destination,source
Multiply real and pop

Exceptions: I, D, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST[1]	100	94-108	0	0	2	FMULP ST(1),ST
ST(i),ST	142	134-148	0	0	2	FMULP ST(1),ST

[1] occurs when one or both operands is "short"—it has 40 trailing zeros in its fraction (e.g., it was loaded from a short-real memory operand).

Table S-19. Instruction Set Reference Data (Cont'd.)

FNOP

FNOP (no operands)
No operation

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	13	10-16	0	0	2	FNOP

FPATAN

FPATAN (no operands)
Partial arctangent

Exceptions: U, P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	650	250-800	0	0	2	FPATAN

FPREM

FPREM (no operands)
Partial remainder

Exceptions: I, D, U

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	125	15-190	0	0	2	FPREM

FPTAN

FPTAN (no operands)
Partial tangent

Exceptions: I, P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	450	30-540	0	0	2	FPTAN

FRNDINT

FRNDINT (no operands)
Round to integer

Exceptions: I, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	45	16-50	0	0	2	FRNDINT

Table S-19. Instruction Set Reference Data (Cont'd.)

FRSTOR

FRSTOR source
Restore saved state

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
94-bytes	210+EA	205-215+EA	47/49	96	2-4	FRSTOR [BP]

FSAVE/FNSAVE

FSAVE destination
Save state

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
94-bytes	210+EA	205-215+EA	48/50	94	2-4	FSAVE [BP]

FSCALE

FSCALE (no operands)
Scale

Exceptions: I, O, U

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	35	32-38	0	0	2	FSCALE

FSQRT

FSQRT (no operands)
Square root

Exceptions: I, D, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	183	180-186	0	0	2	FSQRT

FST

FST destination
Store real

Exceptions: I, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	18	15-22	0	0	2	FST ST(3)
short-real	87+EA	84-90+EA	3/5	6	2-4	FST CORRELATION [DI]
long-real	100+EA	96-104+EA	5/7	10	2-4	FST MEAN_READING

Table S-19. Instruction Set Reference Data (Cont'd.)

FSTCW/FNSTCW

FSTCW destination
Store control word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	15+EA	12-18+EA	2/4	4	2-4	FSTCW SAVE_CONTROL

FSTENV/FNSTENV

FSTENV destination
Store environment

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
14-bytes	45+EA	40-50+EA	8/10	16	2-4	FSTENV [BP]

FSTP

FSTP destination
Store real and pop

Exceptions: I, O, U, P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i)	20	17-24	0	0	2	FSTP ST(2)
short-real	89+EA	86-92+EA	3/5	6	2-4	FSTP [BX].ADJUSTED_RPM
long-real	102+EA	98-106+EA	5/7	10	2-4	FSTP TOTAL_DOSAGE
temp-real	55+EA	52-58+EA	6/8	12	2-4	FSTP REG_SAVE [SI]

FSTSW/FNSTSW

FSTSW destination
Store status word

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
2-bytes	15+EA	12-18+EA	2/4	4	2-4	FSTSW SAVE_STATUS

FSUB

FSUB //source/destination,source
Subtract real

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	85	70-100	0	0	2	FSUB ST,ST(2)
short-real	105+EA	90-120+EA	2/4	4	2-4	FSUB BASE_VALUE
long-real	110+EA	95-125+EA	4/6	8	2-4	FSUB COORDINATE.X

Table S-19. Instruction Set Reference Data (Cont'd.)

FSUBP

FSUBP destination,source
Subtract real and pop

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FSUBP ST(2),ST

FSUBR

FSUBR //source/destination,source
Subtract real reversed

Exceptions: I,D,O,U,P

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST,ST(i)/ST(i),ST	87	70-100	0	0	2	FSUBR ST,ST(1)
short-real	105+EA	90-120+EA	2/4	4	2-4	FSUBR VECTOR[SI]
long-real	110+EA	95-125+EA	4/6	8	2-4	FSUBR [BX].INDEX

FSUBRP

FSUBRP destination,source
Subtract real reversed and pop

Exceptions: I,D,O,U,P

Operands	Executon Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
ST(i),ST	90	75-105	0	0	2	FSUBRP ST(1),ST

FTST

FTST (no operands)
Test stack top against +0.0

Exceptions: I, D

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	42	38-48	0	0	2	FTST

FWAIT

FWAIT (no operands)
(CPU) Wait while 8087 is busy

Exceptions: None (CPU instruction)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	3+5n*	3+5n*	0	0	1	FWAIT

*n = number of times CPU examines $\overline{\text{TEST}}$ line before 8087 lowers BUSY.

Table S-19. Instruction Set Reference Data (Cont'd.)

FXAM

FXAM (no operands)
Examine stack top

Exceptions: None

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	17	12-23	0	0	2	FXAM

FXCH

FXCH //destination
Exchange registers

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
//ST(I)	12	10-15	0	0	2	FXCH ST(2)

FXTRACT

FXTRACT (no operands)
Extract exponent and significand

Exceptions: I

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	50	27-55	0	0	2	FXTRACT

FYL2X

FYL2X (no operands)
$Y \bullet \text{Log}_2 X$

Exceptions: P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	950	900-1100	0	0	2	FYL2X

FYL2XP1

FYL2XP1 (no operands)
$Y \bullet \log_2 (X+1)$

Exceptions: P (operands not checked)

Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	850	700-1000	0	0	2	FYL2XP1

Table S-19. Instruction Set Reference Data (Cont'd.)

F2XM1	F2XM1 (no operands) 2^X-1					Exceptions: U, P (operands not checked)
Operands	Execution Clocks		Transfers		Bytes	Coding Example
	Typical	Range	8086	8088		
(no operands)	500	310-630	0	0	2	F2XM1

Mnemonics © Intel, 1980

S.8 Programming Facilities

Writing programs for the 8087 is a natural extension of the process described in section 2.9, just as the NDP itself is an extension to the CPU. This section describes how PL/M-86 and ASM-86 programmers work with the 8087 in these languages. It also covers the 8087 software emulators provided for both translators.

The level of detail in this section is intended to give programmers a basic understanding of the software tools that can be used with the 8087, but this information is not sufficient to document the full capabilities of these facilities. The definitive description of ASM-86 and the full 8087 emulator is provided in *MCS-86 Assembly Language Reference Manual*, Order No. 9800640, and *MCS-86 Assembler Operating Instructions for ISIS-II Users*, Order No. 9800641. PL/M-86 and the partial emulator are documented in *PL/M-86 Programming Manual*, Order No. 9800466 and *ISIS-II PL/M-86 Compiler Operator's Manual*, Order No. 9800478. These publications may be ordered from Intel's Literature Department.

Readers should be familiar with section 2.9 of the *8086 Family User's Manual* in order to benefit from the material in this section.

PL/M-86

High level language programmers can access a useful subset of the 8087's (real or emulated) capabilities. The PL/M-86 REAL data type corresponds to the NDP's short real (32-bit) format. This data type provides a range of about $8.43*10^{-37} \leqslant |X| \leqslant 3.38*10^{38}$, with about seven significant decimal digits. This representation is adequate for the data manipulated by many microcomputer applications.

The utility of the REAL data type is extended by the PL/M-86 compiler's practice of holding intermediate results in the 8087's temporary real format. This means that the full range and precision of the processor may be utilized for intermediate results. Underflow, overflow, and rounding errors are most likely to occur during intermediate computations rather than during calculation of an expression's final result. Holding intermediate results in temporary real format greatly reduces the likelihood of overflow and underflow and eliminates roundoff as a serious source of error until the final assignment of the result is performed.

The compiler generates 8087 code to evaluate expressions that contain REAL data types, whether variables or constants or both. This means that addition, subtraction, multiplication, division, comparison, and assignment of REALs will be performed by the NDP. INTEGER expressions, on the other hand, are evaluated on the CPU.

Five built-in procedures (table S-20) give the PL/M-86 programmer access to 8087 functions manipulated by the processor control instructions. Prior to any arithmetic operations, a typical PL/M-86 program will setup the NDP after power up using the INIT$REAL$MATH $UNIT procedure and then issue SET$REAL$MODE to configure the NDP. SET$REAL$MODE loads the 8087 control word, and its 16-bit parameter has the format shown in figure S-7. The recommended value of this parameter is 033EH (projective closure, round to nearest, 64-bit precision, interrupts enabled, all exceptions masked except invalid operation). Other settings may be used at the programmer's discretion.

Table S-20. PL/M-86 Built-In Procedures

Procedure	8087 Instruction	Description
INIT$REAL$MATH$UNIT[1]	FINIT	Initialize processor.
SET$REAL$MODE	FLDCW	Set exception masks, rounding precision, and infinity controls.
GET$REAL$ERROR[2]	FNSTSW & FNCLEX	Store, then clear, exception flags.
SAVE$REAL$STATUS	FNSAVE	Save processor state.
RESTORE$REAL$STATUS	FRSTOR	Restore processor state.

[1] Also initializes interrupt pointers for emulation.

[2] Returns low-order byte of status word.

If any exceptions are unmasked, an exception handler must be provided in the form of an interrupt procedure that is designated to be invoked by CPU interrupt pointer (vector) number 16. The exception handler can use the GET$REAL $ERROR procedure to obtain the low-order byte of the 8087 status word and to then clear the exception flags. The byte returned by GET$REAL$ERROR contains the exception flags; these can be examined to determine the source of the exception.

The SAVE$REAL$STATUS and RESTORE $REAL$STATUS procedures are provided for multi-tasking environments where a running task that uses the 8087 may be preempted by another task that also uses the 8087. It is the responsibility of the preempting task to issue SAVE$REAL$STATUS before it executes any statements that affect the 8087; these include the INIT$REAL$MATH$UNIT and SET$REAL $MODE procedures as well as arithmetic expressions. SAVE$REAL$STATUS saves the 8087 state (registers, status, and control words, etc.) on the CPU's stack. RESTORE$REAL$STATUS reloads the state information; the preempting task must invoke this procedure before terminating in order to restore the 8087 to its state at the time the running task was preempted. This enables the preempted task to resume execution from the point of its preemption.

Note that the PL/M-86 compiler prefixes *every* 8087 instruction with a CPU WAIT. Therefore, programmers should not code PL/M-86 statements that generate 8087 instructions if the NDP can request an interrupt and that interrupt is blocked (this may result in the endless wait condition described in section S.6.)

ASM-86

The ASM-86 assembly language provides a single uniform set of facilities for all combinations of the 8086/8088/8087 processors. Assembly language programs can be written to be completely independent of the processor set on which they are destined to execute. This means that a program written originally for an 8088 alone will execute on an 8086/8087 combination without re-assembling. The programmer's view of the hardware is a single machine with these resources:

- 160 instructions
- 12 data types
- 8 general registers
- 4 segment registers
- 8 floating-point registers, organized as a stack

The combination of the assembly language and the 8087 emulator decouple the source code from the execution vehicle. For example, the assembler automatically inserts CPU WAIT instructions in front of those 8087 instructions that require them. If the program actually runs with the emulator rather than the 8087, the WAITs are automatically removed at link time (since there is no NDP for which to wait).

Defining Data

The ASM-86 directives shown in table S-21 allocate storage for 8087 variables and constants. As with other storage allocation directives, the assembler associates a type with any variable defined with these directives. The type value is equal to the length of the storage unit in bytes (10 for DT, 8 for DQ, etc.). The assembler checks the type of any variable coded in an instruction to be certain that it is compatible with the instruction. For example, the coding FIADD ALPHA will be flagged as an error if ALPHA's type is not 2 or 4, because integer addition is only available for word and short integer data types. The operand's type also tells the assembler which machine instruction to produce; although to the programmer there is only an FIADD instruction, a different machine instruction is required for each operand type.

On occasion it is desirable to use an instruction with an operand that has no declared type. For example, if register BX points to a short integer variable, a programmer may want to code FIADD [BX]. This can be done by informing the assembler of the operand's type in the instruction, coding FIADD DWORD PTR [BX]. The corresponding overrides for the other storage allocations are WORD PTR, QWORD PTR, and TBYTE PTR.

The assembler does not, however, check the types of operands used in processor control instructions. Coding FRSTOR [BP] implies that the programmer has set up register BP to point to the stack location where the processor's 94-byte state record has been previously saved.

The initial values for 8087 constants may be coded in several different ways. Binary integer constants may be specified as bit strings, decimal integers, octal integers, or hexadecimal strings. Packed decimal values are normally written as decimal integers, although the assembler will accept and convert other representations of integers. Real values may be written as ordinary decimal real numbers (decimal point required), as decimal numbers in scientific notation, or as hexadecimal strings. Using hexadecimal strings is primarily intended for defining special values such as infinities, NANs, and nonnormalized numbers. Most programmers will find that ordinary decimal and scientific decimal provide the simplest way to initialize 8087 constants. Figure S-20 compares several ways of setting the various 8087 data types to the same initial value.

Note that preceding 8087 variables and constants with the ASM-86 EVEN directive ensures that the operands will be word-aligned in memory. This will produce the best performance in 8086-based systems, and is good practice even for 8088 software, in the event that the programs are transferred to an 8086. All 8087 data types occupy integral numbers of words so that no storage is "wasted" if blocks of variables are defined together and preceded by a single EVEN declarative.

Records and Structures

The ASM-86 RECORD and STRUC (structure) declaratives can be very useful in NDP programming. The record facility can be used to define the bit fields of the control, status, and tag words. Figure S-21 shows one definition of the status word and how it might be used in a routine that polls the 8087 until it has completed an instruction.

Because structures allow different but related data types to be grouped together, they often provide a natural way to represent "real world" data organizations. The fact that the structure template may be "moved" about in memory adds to its flexibility. Figure S-22 shows a simple struc-

Table S-21. 8087 Storage Allocation Directives

Directive	Interpretation	8087 Data Types
DW	Define Word	Word integer
DD	Define Doubleword	Short integer, short real
DQ	Define Quadword	Long integer, long real
DT	Define Tenbyte	Packed decimal, temporary real

```
; THE FOLLOWING ALL ALLOCATE THE CONSTANT: -126
; NOTE TWO'S COMPLEMENT STORAGE OF NEGATIVE BINARY INTEGERS.
;
; EVEN                                  ; FORCE WORD ALIGNMENT
WORD_INTEGER   DW  1111111110000010B ; BIT STRING
SHORT_INTEGER  DD  0FFFFFF82H         ; HEX STRING MUST START WITH DIGIT
LONG_INTEGER   DQ  -126               ; ORDINARY DECIMAL
SHORT_REAL     DD  -126.0             ; NOTE PRESENCE OF '.'
LONG_REAL      DD  -1.26E2            ; ''SCIENTIFIC''
PACKED_DECIMAL DT  -126               ; ORDINARY DECIMAL INTEGER
; IN THE FOLLOWING, SIGN AND EXPONENT IS 'C005',
;    SIGNIFICAND IS '7E00...00', 'R' INFORMS ASSEMBLER THAT
;    THE STRING REPRESENTS A REAL DATA TYPE.
;
TEMP_REAL      DT  0C0057E00000000000000R  ; HEX STRING
```

Figure S-20. Sample 8087 Constants

```
; RESERVE SPACE FOR STATUS WORD
STATUS_WORD                    DW ?
; LAY OUT STATUS WORD FIELDS
STATUS RECORD
&     BUSY:                   1,
&     COND_CODE3:             1,
&     STACK_TOP:              3,
&     COND_CODE2:             1,
&     COND_CODE1:             1,
&     COND_CODE0:             1,
&     INT_REQ:                1,
&     RESERVED:               1,
&     P_FLAG:                 1,
&     U_FLAG:                 1,
&     O_FLAG:                 1,
&     Z_FLAG:                 1,
&     D_FLAG:                 1,
&     I_FLAG:                 1
; POLL STATUS WORD UNTIL 8087 IS NOT BUSY
POLL:   FNSTSW  STATUS_WORD
        TEST    STATUS_WORD, MASK BUSY
        JNZ     POLL
```

Figure S-21. Status Word RECORD Definition

```
SAMPLE      STRUC

   N_OBS          DD    ?   ;SHORT INTEGER
   MEAN           DQ    ?   ;LONG REAL
   MODE           DW    ?   ;WORD INTEGER
   STD_DEV        DQ    ?   ;LONG REAL
   ;ARRAY OF OBSERVATIONS -- WORD INTEGER
   TEST_SCORES    DW    1000 DUP (?)
SAMPLE ENDS
```

Figure S-22. Structure Definition

ture that might be used to represent data consisting of a series of test score samples. A structure could also be used to define the organization of the information stored and loaded by the FSTENV and FLDENV instructions.

Addressing Modes

8087 memory data can be accessed with any of the CPU's five memory addressing modes. This means that 8087 data types can be incorporated in data aggregates ranging from simple to complex according to the needs of the application. The addressing modes, and the ASM-86 notation used to specify them in instructions, make the accessing of structures, arrays, arrays of structures, and other organizations direct and straightforward. Table S-22 gives several examples of 8087 instructions coded with operands that illustrate different addressing modes.

8087 Emulators

Intel offers two software products that provide the functional equivalent of an 8087, implemented in 8086/8088 software. The full emulator (E8087) emulates all 8087 instructions. The partial emulator (PE8087) is a smaller version that implements only the instructions needed to support PL/M-86 programs. The full emulator adds about 16k bytes to a program, while the partial emulator executes in about 8k. Any emulated program will deliver the same results (except for timing) if it is executed on 8087 hardware.

The emulators may be viewed as consisting of emulated hardware and emulated instructions. The emulators establish in CPU memory the equivalent of the 8087 register stack, control, and status words and all other programmer-accessible elements of the NDP architecture. The emulator instructions utilize the same algorithms as their hardware counterparts. Emulator instructions are actually implemented as CPU interrupt procedures. During relocation and linkage the 8087 machine instructions generated by the ASM-86 and PL/M-86 translators are changed to software interrupt (INT) instructions which invoke these procedures as the CPU processes its instruction stream.

Table S-22. Addressing Mode Examples

Coding		Interpretation
FIADD	ALPHA	ALPHA is a simple scalar (mode is direct).
FDIVR	ALPHA.BETA	BETA is a field in a structure that is "overlaid" on ALPHA (mode is direct).
FMUL	QWORD PTR [BX]	BX contains the address of a long real variable (mode is register indirect).
FSUB	ALPHA [SI]	ALPHA is an array and SI contains the offset of an array element from the start of the array (mode is indexed).
FILD	[BP].BETA	BP contains the address of a structure on the CPU stack and BETA is a field in the structure (mode is based).
FBLD	TBYTE PTR [BX] [DI]	BX contains the address of a packed decimal array and DI contains the offset of an array element (mode is based indexed).

Since the decision to produce real or emulated 8087 instructions is made at link time, a program may be switched from one mode to the other without retranslating the source code. When the PL/M-86 compiler or ASM-86 assembler places an 8087 machine instruction into an object module, it also inserts a special external reference. This reference is satisfied by linking the object module to one of two Intel-supplied libraries: the real library, or the emulator library. If the real library is specified, LINK-86 simply deletes the external references, leaving the original 8087 machine instructions.

To run on an emulated 8087, the object program is linked to the emulator library and to a file containing the code of either the full or the partial emulator. LINK-86 then adds the emulator code to the program and changes the 8087 machine instructions (and their preceding WAITs) to CPU software interrupt instructions. Any FWAIT instructions are also changed to CPU NOPs.

Note that an explicitly-coded CPU WAIT instruction will *not* be changed; if it is executed under emulation, the CPU will wait forever. This is why the FWAIT mnemonic should always be used when the external processor that the CPU is to wait for is an 8087.

In order to be compatible with E8087, ASM-86 programs should observe the following conventions:

- Their stack segment and class should be named STACK.
- Interrupt pointer (vector) 16 should be designated for the user's exception handler interrupt procedure.
- The external procedure INIT87 should be called in the program's initialization (power-up) sequence. If the emulator is being used, this procedure will initialize CPU interrupt pointers 20-31 to the addresses of emulator procedures and will execute an (emulated) FINIT instruction. If the program is not being emulated, INIT87 simply executes the FINIT instruction.

PL/M-86 automatically observes corresponding conventions.

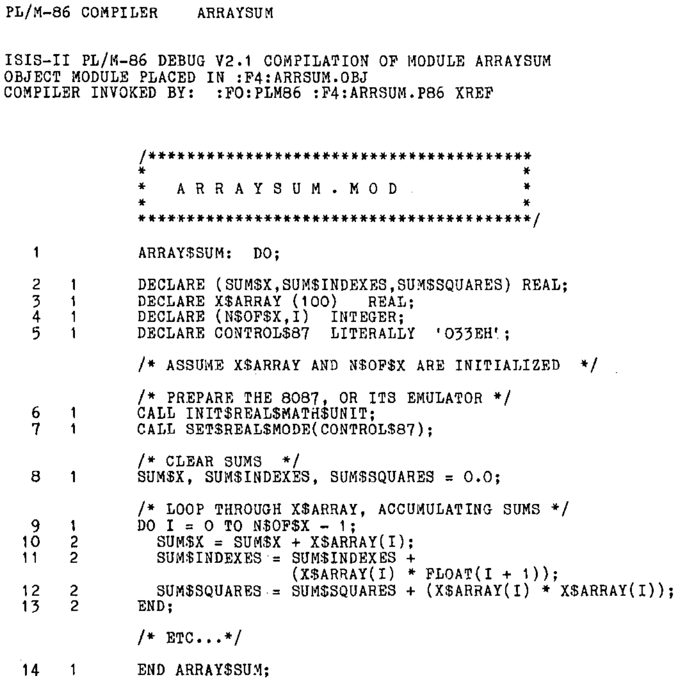
Programming Example

Figures S-23 and S-24 show the PL/M-86 and ASM-86 code for a simple 8087 program, called ARRSUM. The program references an array (X$ARRAY), which contains 0-100 short real values; the integer variable N$OF$X indicates the number of array elements the program is to consider. ARRSUM steps through X$ARRAY accumulating three sums:

- SUM$X, the sum of the array values;
- SUM$INDEXES, the sum of each array value times its index, where the index of the first element is 1, the second is 2, etc.;
- SUM$SQUARES, the sum of each array element squared.

(A true program, of course, would go beyond these steps to store and use the results of these calculations.) The control word is set with the recommended values: projective closure, round to nearest, 64-bit precision, interrupts enabled, and all exceptions masked except invalid operation. It is assumed that an exception handler has been written to field the invalid operation, if it occurs, and that it is invoked by interrupt pointer 16. Either version of the program will run on an actual or an emulated 8087 without altering the code shown.

The PL/M-86 version of ARRSUM (figure S-23) is very straightforward and illustrates how easily the 8087 can be used in this language. After declaring variables the program calls built-in procedures to initialize the processor (or its emulator) and to load the control word. The program clears the sum variables and then steps through X$ARRAY with a DO-loop. The loop control takes into account PL/M-86's practice of considering the index of the first element of an array to be 0. In the computation of SUM$INDEXES, the built-in procedure FLOAT converts I+1 from integer to real because the language does not support "mixed mode" arithmetic. One of the strengths of the NDP, of



```
PL/M-86 COMPILER    ARRAYSUM


ISIS-II PL/M-86 DEBUG V2.1 COMPILATION OF MODULE ARRAYSUM
OBJECT MODULE PLACED IN :F4:ARRSUM.OBJ
COMPILER INVOKED BY:  :FO:PLM86 :F4:ARRSUM.P86 XREF


              /****************************************
              *                                       *
              *   A R R A Y S U M . M O D             *
              *                                       *
              ****************************************/

  1           ARRAY$SUM:  DO;

  2    1      DECLARE (SUM$X,SUM$INDEXES,SUM$SQUARES) REAL;
  3    1      DECLARE X$ARRAY (100)   REAL;
  4    1      DECLARE (N$OF$X,I)  INTEGER;
  5    1      DECLARE CONTROL$87  LITERALLY  '033EH';

              /* ASSUME X$ARRAY AND N$OF$X ARE INITIALIZED  */

              /* PREPARE THE 8087, OR ITS EMULATOR */
  6    1      CALL INIT$REAL$MATH$UNIT;
  7    1      CALL SET$REAL$MODE(CONTROL$87);

              /* CLEAR SUMS  */
  8    1      SUM$X, SUM$INDEXES, SUM$SQUARES = 0.0;

              /* LOOP THROUGH X$ARRAY, ACCUMULATING SUMS */
  9    1      DO I = 0 TO N$OF$X - 1;
 10    2        SUM$X = SUM$X + X$ARRAY(I);
 11    2        SUM$INDEXES = SUM$INDEXES +
                              (X$ARRAY(I) * FLOAT(I + 1));
 12    2        SUM$SQUARES = SUM$SQUARES + (X$ARRAY(I) * X$ARRAY(I));
 13    2      END;

              /* ETC...*/

 14    1      END ARRAY$SUM;
```

Figure S-23. Sample PL/M-86 Program

```
PL/M-86 COMPILER    ARRAYSUM

CROSS-REFERENCE LISTING
-----------------------

   DEFN  ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
  ----- ------ -----  --------------------------------


      1  0002H   151  ARRAYSUM . . . . .     PROCEDURE STACK=0002H
      5               CONTROL87. . . . .     LITERALLY          7
                      FLOAT. . . . . . .     BUILTIN           11
      4  019EH     2  I. . . . . . . . .     INTEGER            9    10    11    12
                      INITREALMATHUNIT .     BUILTIN            6
      4  019CH     2  NOFX . . . . . . .     INTEGER            9
                      SETREALMODE. . . .     BUILTIN            7
      2  0004H     4  SUMINDEXES . . . .     REAL          8   11
      2  0008H     4  SUMSQUARES . . . .     REAL          8   12
      2  0000H     4  SUMX . . . . . . .     REAL          8   10
      3  000CH   400  XARRAY . . . . . .     REAL ARRAY(100)         10    11    12


MODULE INFORMATION:

     CODE AREA SIZE     = 0099H    153D
     CONSTANT AREA SIZE = 0004H      4D
     VARIABLE AREA SIZE = 01A0H    416D
     MAXIMUM STACK SIZE = 0002H      2D
     33 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Figure S-23. Sample PL/M-86 Program (Cont'd.)

course, is that it *does* support arithmetic on mixed data types, and assembly language programmers can take advantage of this facility.

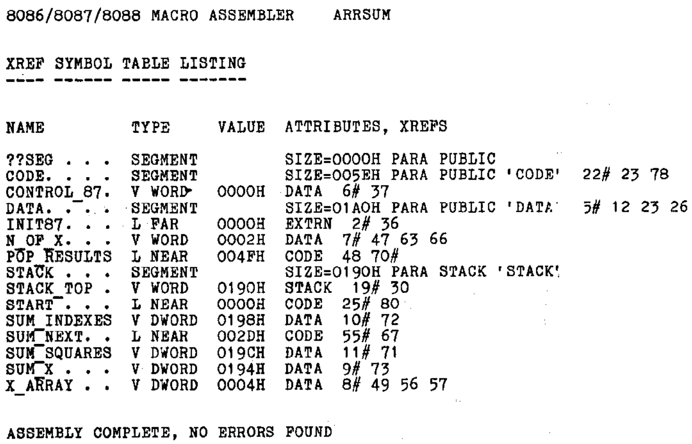
The ASM-86 version (figure S-24) defines the external procedure INIT87, which makes the different initialization requirements of the processor and its emulator transparent to the source code. After defining the data, and setting up the segment registers and stack pointer, the program calls INIT87 and loads the control word. The computation begins with the next three instructions, which clear three registers by loading (pushing) zeros onto the stack. As shown in figure S-25, these registers remain at the bottom of the stack throughout the computation while temporary values are pushed on and popped off the stack above them.

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

ISIS-II 8086/8087/8088 MACRO ASSEMBLER V3.0 ASSEMBLY OF MODULE ARRSUM
OBJECT MODULE PLACED IN :F1:ARRSUM.OBJ
ASSEMBLER INVOKED BY:  :F0:ASM86 :F1:ARRSUM.A86 XREF


LOC  OBJ                   LINE      SOURCE

                              1      ;DEFINE INITIALIZATION ROUTINE
                              2              EXTRN   INIT87:FAR
                              3
                              4      ;ALLOCATE SPACE FOR DATA
----                          5              DATA            SEGMENT PUBLIC 'DATA'
0000 3E03                     6              CONTROL_87      DW      033EH
0002 ????                     7              N_OF_X          DW      ?
0004 (100                     8              X_ARRAY         DD      100 DUP (?)
     ????????
     )
0194 ????????                 9              SUM_X           DD      ?
0198 ????????                10              SUM_INDEXES     DD      ?
019C ????????                11              SUM_SQUARES     DD      ?
----                         12              DATA            ENDS
```

Figure S-24. Sample ASM-86 Program

```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

LOC  OBJ                  LINE     SOURCE
                            13
                            14     ;ALLOCATE CPU STACK SPACE
----                        15             STACK           SEGMENT STACK 'STACK'
0000 (200                   16                             DW      200 DUP (?)
      ????
      )
                            17
                            18     ;LABEL INITIAL TOP OF STACK
0190                        19             STACK_TOP       LABEL   WORD
----                        20             STACK           ENDS
                            21
----                        22             CODE    SEGMENT PUBLIC 'CODE'
                            23             ASSUME  CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
                            24
0000                        25     START:
0000 B8----         R       26             MOV     AX,DATA
0003 8ED8                   27             MOV     DS,AX
0005 B8----         R       28             MOV     AX,STACK
0008 8ED0                   29             MOV     SS,AX
000A BC9001         R       30             MOV     SP,OFFSET STACK_TOP
                            31
                            32     ;ASSUME X_ARRAY & N_OF_X ARE INITIALIZED.
                            33     ;       NOTE: PROGRAM ZEROS N_OF_X
                            34     ;PREPARE THE 8087 OR ITS EMULATOR.
                            35
000D 9A0000----     E       36             CALL    INIT87
0012 9BD92E0000     R       37             FLDCW   CONTROL_87
                            38
                            39     ;CLEAR 3 REGISTERS TO HOLD RUNNING SUMS.
                            40
0017 9BD9EE                 41             FLDZ
001A 9BD9EE                 42             FLDZ
001D 9BD9EE                 43             FLDZ
                            44
                            45     ;SETUP CX AS LOOP COUNTER & SI AS INDEX TO X_ARRAY.
                            46
0020 8B0E0200       R       47             MOV     CX,N_OF_X
0024 E329                   48             JCXZ    POP_RESULTS     ;EXIT EARLY IF X_ARRAY EMPTY
0026 B80400                 49             MOV     AX,TYPE X_ARRAY
0029 F7E9                   50             IMUL    CX
002B 8BF0                   51             MOV     SI,AX
                            52
                            53     ;SI NOW CONTAINS INDEX OF LAST ELEMENT + 1.
                            54     ;LOOP THRU X_ARRAY ACCUMULATING SUMS.
002D                        55     SUM_NEXT:
002D 83EE04                 56             SUB     SI,TYPE X_ARRAY      ;BACKUP ONE ELEMENT
0030 9BD9840400     R       57             FLD     X_ARRAY[SI]          ;PUSH IT ONTO STACK
0035 9BDCC3                 58             FADD    ST(3),ST             ;ADD INTO SUM OF X
0038 9BD9C0                 59             FLD     ST                   ;DUPLICATE X ON TOP
003B 9BDCC8                 60             FMUL    ST,ST                ;SQUARE IT
003E 9BDEC2                 61             FADDP   ST(2),ST             ;ADD INTO SUM OF SQUARES
                            62                                          ;  AND DISCARD
0041 9BDE0E0200     R       63             FIMUL   N_OF_X               ;GET X TIMES ITS INDEX
0046 9BDEC2                 64             FADDP   ST(2),ST             ;ADD INTO SUM OF (INDEX * X)
                            65                                          ;  AND DISCARD
0049 FF0E0200       R       66             DEC     N_OF_X               ;REDUCE INDEX FOR NEXT ITERATION
004D E2DE                   67             LOOP    SUM_NEXT             ;CONTINUE
                            68
                            69     ;POP RUNNING SUMS INTO MEMORY
004F                        70     POP_RESULTS:
004F 9BD91E9C01     R       71             FSTP    SUM_SQUARES
0054 9BD91E9801     R       72             FSTP    SUM_INDEXES
0059 9BD91E9401     R       73             FSTP    SUM_X
                            74
                            75     ;
                            76     ;ETC...
                            77     ;
----                        78     CODE    ENDS
                            79
  0000                      80             END     START
```

Figure S-24. Sample ASM-86 Program (Cont'd.)



```
8086/8087/8088 MACRO ASSEMBLER    ARRSUM

XREF SYMBOL TABLE LISTING
---- ------ ----- -------

NAME          TYPE      VALUE  ATTRIBUTES, XREFS

??SEG . . .   SEGMENT          SIZE=0000H PARA PUBLIC
CODE. . . .   SEGMENT          SIZE=005EH PARA PUBLIC 'CODE'   22# 23 78
CONTROL_87.   V WORD    0000H  DATA  6# 37
DATA. . . .   SEGMENT          SIZE=01A0H PARA PUBLIC 'DATA'   5# 12 23 26
INIT87. . .   L FAR     0000H  EXTRN  2# 36
N_OF_X. . .   V WORD    0002H  DATA  7# 47 63 66
POP_RESULTS   L NEAR    004FH  CODE  48 70#
STACK . . .   SEGMENT          SIZE=0190H PARA STACK 'STACK'
STACK_TOP .   V WORD    0190H  STACK  19# 30
START . . .   L NEAR    0000H  CODE  25# 80
SUM_INDEXES   V DWORD   0198H  DATA  10# 72
SUM_NEXT. .   L NEAR    002DH  CODE  55# 67
SUM_SQUARES   V DWORD   019CH  DATA  11# 71
SUM_X . . .   V DWORD   0194H  DATA  9# 73
X_ARRAY . .   V DWORD   0004H  DATA  8# 49 56 57

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure S-24. Sample ASM-86 Program (Cont'd.)

The program uses the CPU LOOP instruction to control its iteration through X_ARRAY; register CX, which LOOP automatically decrements, is loaded with N_OF_X, the number of array elements to be summed. Register SI is used to select (index) the array elements. The program steps through X_ARRAY from "back to front", so SI is initialized to point at the element just beyond the first element to be processed. The ASM-86 TYPE operator is used to determine the number of bytes in each array element. This permits changing X_ARRAY to a long real array by simply changing its definition (DD to DQ) and re-assembling.

Figure S-25 shows the effect of the instructions in the program loop on the NDP register stack. The figure assumes that the program is in its first iteration, that N_OF_X is 20, and that X_ARRAY(19) (the 20th element) contains the value 2.5. When the loop terminates, the three sums are left as the top stack elements so that the program ends by simply popping them into memory variables.

S.9 Special Topics

This section describes features of the 8087 which will be of interest to groups of users who have special requirements. Most users will not need to understand this material in detail in order to utilize the NDP successfully. Most readers, then, can either browse this section, or skip it altogether in favor of the programming examples in section S.10.

The first four topics in this section cover the 8087's generation and handling of nonnormalized real values, zeros, infinities and NANs. In the great majority of applications, these special values will either not appear at all, or in the case of zeros, will function according to the normal rules of arithmetic. Next the bit encodings of each data type are summarized in table form, including special values. This information may be of use to programmers who are sorting these data types or are decoding unformatted memory dumps or data monitored from the bus. At the end of the section is a table that lists all 8087 exception conditions by class, and the processor's masked response to each exception. This information will principally be of use to writers of exception handlers and to anyone else interested in ascertaining the exact conditions under which the NDP signals a given type of exception.

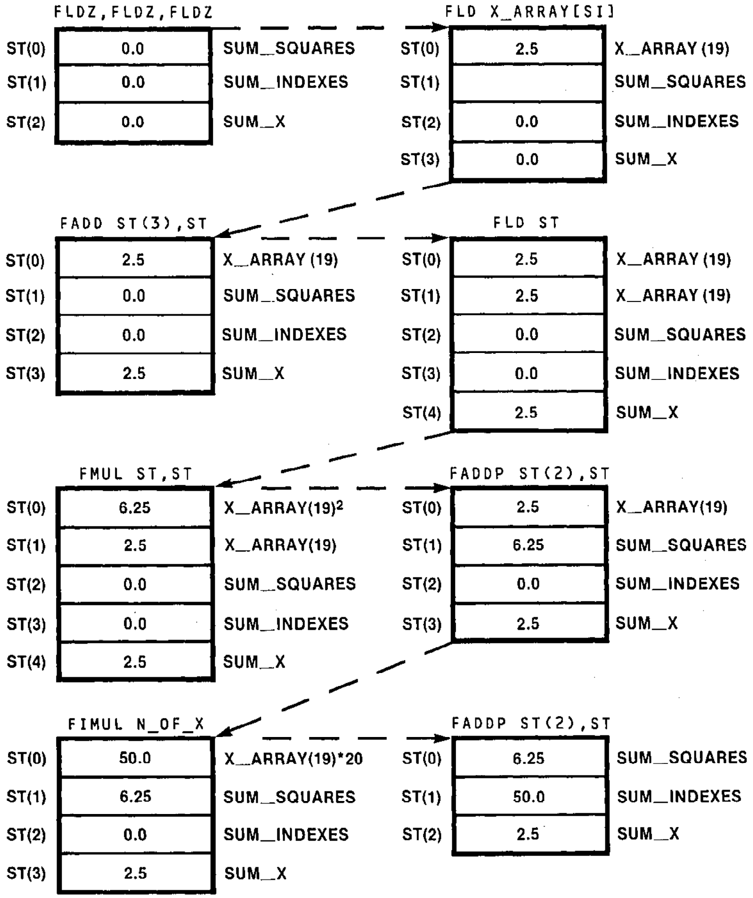


Figure S-25. Instructions and Register Stack

Nonnormal Real Numbers

As discussed in section S.3, the 8087 generally stores nonzero real numbers in normalized floating point form; that is, the integer (leading) bit of the significand is always a 1. This bit is explicitly stored in the temporary real format, and is implicit in the short and long real forms. Normalized storage allows the maximum number of significant digits to be held in a significand of a given width, because leading zeros are eliminated.

Denormals

A denormal is the result of the NDP's masked response to an underflow exception. Underflow occurs when the exponent of a true result is too small to be represented in the destination format. For example, a true exponent of −130 will cause underflow if the destination is short real, because −126 is the smallest exponent this format can accommodate. (No underflow would occur if the destination were long or temporary real since these can handle exponents down to −1023 and −16,383, respectively.)

The NDP's unmasked response to underflow is to stop and request an interrupt if the destination is a memory operand. If the destination is a register, the processor adds the constant 24,576 (decimal) to the true result's exponent, returns the result, and then requests an interrupt. The constant forces the exponent into the range of the temporary real format, and an exception handler can subtract out the constant to ascertain the true exponent. Thus, execution always stops when there is an unmasked underflow.

The intent of the masked response to underflow is to allow computation to continue without program intervention, while introducing an error that carries about the same risk of contaminating the final result as roundoff error. Roundoff (precision) errors occur frequently in real number calculations; sometimes they spoil the result of computation, but often they do not. Recognizing that roundoff errors are often non-fatal, computation usually proceeds and the programmer inspects the final result to see if these errors have had a significant effect. The 8087's masked underflow response allows programmers to treat underflows in a similar manner; the computation continues and the programmer can examine the final result to determine if an underflow has had important consequences. (If the underflow has had a significant effect, an invalid operation will probably be signalled later in the computation.)

Most computers underflow "abruptly"; they simply return a zero result, which is likely to produce an unacceptable final result if computation continues. The 8087, on the other hand, underflows "gradually" when the underflow exception is masked. Gradual underflow is accomplished by denormalizing the result until it is just within the exponent range of the destination. Denormalizing means incrementing the true result's exponent and inserting a corresponding leading zero in the significand, shifting the rest of the significand one place to the right. Table S-23 illustrates how a result might be denormalized to fit a short real destination.

Denormalization produces a denormal or a zero. Denormals are readily identified by their exponents, which are always the minimum for their formats; in biased form, this is always the bit string: 00...00. This same exponent value is also assigned to the zeros, but a denormal has a nonzero significand. A denormal in a register is tagged special.

The denormalization process may cause the loss of low-order significand bits as they are shifted off the right. In a severe case, *all* the significand bits of the true result are shifted out and replaced by the leading zeros. In this case, the result of denormalization is a true zero, and if the value is in a register, it is tagged as such. However, this is a comparatively rare occurrence, and in any case is no worse than "abrupt" underflow.

Denormals are rarely encountered in most applications. Typical debugged algorithms generate extremely small results during the evaluation of intermediate subexpressions; the final result is usually of an appropriate magnitude for its short or long real destination. If intermediate results are held in temporary real, as is recommended, the great range of this format

Table S-23. Denormalization Process

Operation	Sign	Exponent[1]	Significand
True Result	0	−129	$1_{\Delta}01011100...00$
Denormalize	0	−128	$0_{\Delta}101011100...00$
Denormalize	0	−127	$0_{\Delta}0101011100...00$
Denormalize	0	−126	$0_{\Delta}00101011100...00$
Denormal Result[2]	0	−126	$0_{\Delta}00101011100...00$

Notes:

[1] expressed as unbiased, decimal number

[2] Before storing, significand is rounded to 24 bits, integer bit is dropped, and exponent is biased by adding 126.

makes underflow very unlikely. Denormals are likely to arise only when an application generates a great many intermediates, so many that they cannot be held on the register stack or in temporary real memory variables. If storage limitations force the use of short or long reals for intermediates, and small values are produced, underflow may occur, and if masked, may generate denormals.

Accessing a denormal may produce an exception as shown in table S-24. (The denormalized exception signals that a denormal has been fetched.) Denormals may have reduced significance due to lost low-order bits, and an option of the proposed IEEE standard precludes operations on non-normalized operands. This option may be implemented in the form of an exception handler that responds to unmasked denormalized exceptions. Most users will mask this exception so that computation may proceed; any loss of accuracy will be analyzed by the user when the final result is delivered.

As table S-24 shows, the division and remainder operations do not accept denormal divisors and raise the invalid operation exception. Recall, also, that the transcendental instructions require normalized operands and do *not* check for exceptions. In all other cases, the NDP converts denormals to unnormals, and the unnormal arithmetic rules then apply.

Unnormals

An unnormal is the "descendent" of a denormal and therefore of a masked underflow response. An unnormal may exist only in the temporary real format; it may have any exponent that a normal may have, but it is distinguished from a normal by the integer bit of its significand, which is always 0. An unnormal in a register is tagged valid.

Unnormals allow arithmetic to continue following an underflow while still retaining their identity as numbers which may have reduced significance. That is, unnormal operands generate unnormal results, so long as their unnormality has a significant effect on the result. Unnormals are thus prevented from "masquerading" as normals, numbers which have full significance. On the other hand, if an unnormal has an insignificant effect on a calculation with a normal, the result will be normal. For example, adding a small unnormal to a large normal yields a normal result. The converse situation yields an unnormal.

Table S-25 shows how the instruction set deals with unnormal operands. Note that the unnormal may be the original operand or a temporary created by the 8087 from a denormal.

Table S-24. Exceptions Due to Denormal Operands

Operation	Exception	Masked Response
FLD (short/long real)	D	Load as equivalent unnormal
arithmetic (except following)	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Compare and test	D	Convert (in a work area) denormal to equivalent unnormal and proceed
Division or FPREM with denormal divisor	I	Return real *indefinite*

Table S-25. Unnormal Operands and Results

Operation	Result
Addition/subtraction	Normalization of operand with larger absolute value determines normalization of result.
Multiplication	If either operand is unnormal, result is unnormal.
Division (unnormal dividend only)	Result is unnormal.
FPREM (unnormal dividend only)	Result is normalized.
Division/FPREM (unnormal divisor)	Signal invalid operation.
Compare/FTST	Normalize as much as possible before making comparison.
FRNDINT	Normalize as much as possible before rounding.
FSQRT	Signal invalid operation.
FST, FSTP (short/long real destination)	If value is above destination's underflow boundary, then signal invalid operation; else signal underflow.
FSTP (temporary real destination)	Store as usual.
FIST, FISTP, FBSTP	Signal invalid operation.
FLD	Load as usual.
FXCH	Exchange as usual.
Transcendental instructions	Undefined; operands must be normal and are not checked.

Zeros and Pseudo-Zeros

As discussed in section S.3, the real and packed decimal data types support signed zeros, while the binary integers represent a single zero, signed positive. The signed zeros behave, however, as though they are a single unsigned quantity. If necessary, the FXAM instruction may be used to determine a zero's sign.

The zeros discussed above are called true zeros; if one of them is loaded or generated in a register, the register is tagged zero. Table S-26 lists the results of instructions executed with zero operands and also shows how a true zero may be created from nonzero operands. (Nonzero operands are denoted "X" or "Y" in the table.)

Only the temporary real format may contain a special class of values called pseudo-zeros. A pseudo-zero is an unnormal whose significand is all zeros, but whose (biased) exponent is nonzero (true zeros have a zero exponent). Neither is a pseudo-zero's exponent all ones, since this encoding is reserved for infinities and NANs. A pseudo-zero result will be produced if two unnormals, containing a total of more than 64 leading zero bits in their significands, are multiplied together. This is a remote possibility in most applications, but it can happen.

Table S-26. Zero Operands and Results

Operation/Operands	Result	Operation/Operands	Result
FLD, FBLD [1]		Division	
+0	+0	±0 ÷ ±0	Invalid operation
−0	−0	±X ÷ ±0	Zerodivide
FILD [2]		+0 ÷ +X, −0 ÷ −X	+0
+0	+0	+0 ÷ −X, −0 ÷ +X	−0
FST, FSTP		−X ÷ −Y, +X ÷ +Y	+0, underflow [8]
+0	+0	−X ÷ +Y, +X ÷ −Y	−0, underflow [8]
−0	−0		
+X [3]	+0	FPREM	
−X [3]	−0	±0 rem ±0	Invalid operation
FBSTP		±X rem ±0	Invalid operation
+0	+0	+0 rem +X, +0 rem −X	+0
−0	−0	−0 rem +X, −0 rem −X	−0
FIST, FISTP		+X rem +Y, +X rem −Y	+0 [9]
+0	+0	−X rem −Y, −X rem +Y	−0 [9]
−0	+0		
+X [4]	+0	FSQRT	
−X [4]	+0	−0	−0
		+0	+0
Addition			
+0 plus +0	+0	Compare	
−0 plus −0	−0	±0 : +X	A < B
+0 plus −0, −0 plus +0	*0 [5]	±0 : ±0	A = B
−X plus +X, +X plus −X	*0 [5]	±0 : −X	A > B
±0 plus ±X, ±X plus ±0	†X [6]		
		FTST	
Subtraction		±0	Zero
+0 minus −0	+0	FCHS	
−0 minus +0	−0	+0	−0
+0 minus +0, −0 minus −0	*0 [5]	−0	+0
+X minus +X, −X minus −X	*0 [5]	FABS	
±0 minus ±X, ±X minus ±0	†X [6]	±0	+0
		F2XM1	
Multiplication		+0	+0
+0 • +0, −0 • −0	+0	−0	−0
+0 • −0, −0 • +0	−0	FRNDINT	
+0 • +X, +X • +0	+0	+0	+0
+0 • −X, −X • +0	−0	−0	−0
−0 • +X, +X • −0	−0	FXTRACT	
−0 • −X, −X • −0	+0	+0	Both +0
+X • +Y, −X • −Y	+0, underflow [7]	−0	Both −0
+X • −Y, −X • +Y	−0, underflow [7]		

Notes:

[1] Arithmetic and compare operations with real memory operands interpret the memory operand signs in the same way.

[2] Arithmetic and compare operations with binary integers interpret the integer sign in the same manner.

[3] Severe underflows in storing to short or long real may generate zeros.

[4] Small values (|X| < 1) stored into integers may round to zero.

[5] Sign is determined by rounding mode:
* = + for nearest, up or chop
* = − for down

[6] † = sign of X.

(7) Very small values of X and Y may yield zeros, after rounding of true result. NDP signals underflow to warn that zero has been yielded by nonzero operands.

(8) Very small X and very large Y may yield zero, after rounding of true result. NDP signals underflow to warn that zero has been yielded from nonzero operands.

(9) When Y divides into X exactly.

Pseudo-zero operands behave like unnormals, except in the following cases where they produce the same results as true zeros:

- compare and test instructions
- FRNDINT (round to integer)
- division, where the dividend is either a true zero or a pseudo-zero (the divisor is a pseudo-zero).

In addition and subtraction of a pseudo-zero and a true zero or another pseudo-zero, the pseudo-zero(s) behave like unnormals, except for the determination of the result's sign. The sign is determined as shown in table S-26 for two true zero operands.

Infinities

The real formats support signed representations of infinities. These values are encoded with a biased exponent of all ones and a significand of $1_{\Delta}00...00$; if the infinity is in a register, it is tagged special. The significand distinguishes infinities from NANs, including real *indefinite*.

A programmer may code an infinity, or it may be created by the NDP as its masked response to an overflow or a zerodivide exception. Note that when rounding is up or down, the masked response may create the largest valid value representable in the destination rather than infinity. See table S-33 for details. As operands, infinities behave somewhat differently depending on how the infinity control field in the control word is set (see table S-27). When the projective model of infinity is selected, the infinities behave as a single unsigned representation; because of this, infinity cannot be compared with any value except infinity. In affine mode, the signs of the infinities are observed, and comparisons are possible.

Table S-27. Infinity Operands and Results

Operation	Projective Result	Affine Result
Addition		
+∞ plus +∞	Invalid operation	+∞
−∞ plus −∞	Invalid operation	−∞
+∞ plus −∞	Invalid operation	Invalid operation
−∞ plus +∞	Invalid operation	Invalid operation
±∞ plus ±X	*∞	*∞
±X plus ±∞	*∞	*∞
Subtraction		
+∞ minus −∞	Invalid operation	+∞
−∞ minus +∞	Invalid operation	−∞
+∞ minus +∞	Invalid operation	Invalid operation
−∞ minus −∞	Invalid operation	Invalid operation
±∞ minus ±X	*∞	*∞
±X minus ±∞	†∞	†∞
Multiplication		
±∞ • ±∞	⊕∞	⊕∞
±∞ • ±Y	⊕∞	⊕∞
±0 • ±∞, ±∞ * ±0	Invalid operation	Invalid operation

Table S-27. Infinity Operands and Results (Cont'd.)

Operation	Projective Result	Affine Result
Division		
±∞ ÷ ±∞	Invalid operation	Invalid operation
±∞ ÷ ±X	⊕∞	⊕∞
±X ÷ ±∞	⊕0	⊕0
FSQRT		
−∞	Invalid operation	Invalid operation
+∞	Invalid operaton	+∞
FPREM		
±∞ rem ±∞	Invalid operation	Invalid operation
±∞rem ±X	Invalid operation	Invalid operation
±Y rem ±∞	*Y	*Y
±0 rem ±∞	*0	*0
FRNDINT		
±∞	*∞	*∞
FSCALE		
±∞ scaled by ±∞	Invalid operation	Invalid operation
±∞ scaled by ±X	*∞	*∞
±0 scaled by ±∞	*0	*0
±Y scaled by ±∞	Invalid operation	Invalid operation
FXTRACT		
±∞	Invalid operation	Invalid operation
Compare		
±∞: ±∞	A = B	−∞ < +∞
±∞ : ±Y	A ? B (and) invalid operation	−∞ < Y < +∞
±∞ : ±0	A ? B (and) invalid operation	−∞ < 0 < +∞
FTST		
±∞	A ? B (and) invalid operation	*∞

Notes: X = zero or nonzero operand
Y = nonzero operand
* = sign of original operand
† = sign is complement of original operand's sign
⊕= sign is "exclusive or" original operand signs (+ if operands had same sign, − if operands had different signs)

NANs

A NAN (Not-A-Number) is a member of a class of special values that exist in the real formats only. A NAN has an exponent of 11...11B, may have either sign, and may have any significand except $1_{\Delta}00...00$B, which is assigned to the infinities. A NAN in a register is tagged special.

The 8087 will generate the special NAN, real *indefinite*, as its masked response to an invalid operation exception. This NAN is signed negative; its significand is encoded $1_{\Delta}100...00$. All other NANs represent programmer-created values.

Whenever the NDP uses an operand that is a NAN, it signals invalid operation. Its masked response to this exception is to return the NAN as the operation's result. If both operands of an instruction are NANs, the result is the NAN with the larger absolute value. In this way, a NAN that enters a computation propagates through the computation and will eventually be delivered as the final result. Note, however, that the transcendental instructions do not check their operands, and a NAN will produce an undefined result.

By unmasking the invalid operation exception, the programmer can use NANs to trap to the exception handler. The generality of this approach and the large number of NAN values that are available, provide the sophisticated programmer with a tool that can be applied to a variety of special situations.

For example, a compiler could use NANs to references to uninitialized (real) array elements. The compiler could pre-initialize each array element with a NAN whose significand contained the index (relative position) of the element. If an application program attempted to access an element that it had not initialized, it would use the NAN placed there by the compiler. If the invalid operation exception were unmasked, an interrupt would occur, and the exception handler would be invoked. The exception handler could determine which element had been accessed, since the operand address field of the exception pointers would point to the NAN, and the NAN would contain the index number of the array element.

NANs could also be used to speed up debugging. In its early testing phase a program often contains multiple errors. An exception handler could be written to save diagnostic information in memory whenever it was invoked. After storing the diagnsotic data, it could supply a NAN as the result of the erroneous instruction, and that NAN could point to its associated diagnostic area in memory. The program would then continue, creating a different NAN for each error. When the program ended, the NAN results could be used to access the diagnostic data saved at the time the errors occurred. Many errors could thus be diagnosed and corrected in one test run.

Data Type Encodings

Tables S-28 through S-31 summarize how various types of values are encoded in the seven NDP data types. In all tables, the less significant bits are to the right and are stored in the lowest memory addresses. The sign bit is always the left-most bit of the highest-addressed byte.

Notice that in every format one encoding is interpreted as representing the special value *indefinite*. The 8087 produces this encoding as its response to a masked invalid operation exception. In the case of the reals, *indefinite* can be loaded and stored like any NAN and it always retains its special identity; programmers are advised not to use this encoding for any other purpose. Packed decimal *indefinite* may be stored by the NDP in a FBSTP instruction; attempting to use this encoding in a FBLD instruction, however, will have an undefined result. In the binary integers, the same encoding may represent either *indefinite* or the largest negative number supported by the format (-2^{15}, -2^{31} or -2^{63}). The 8087 will store this encoding as its masked response to an invalid operation, or when the value in a source register represents, or rounds to, the largest negative integer representable by the destination. In situations where its origin may be ambiguous, the invalid operation exception flag can be examined to see if the value was produced by an exception response. When this encoding is loaded, or used by an integer arithmetic or compare operation, it is always interpreted as a negative number; thus *indefinite* cannot be loaded from a packed decimal or binary integer.

Table S-28. Binary Integer Encodings

	Class	Sign	Magnitude
Positives	(Largest)	0	11...11
		•	•
		•	•
		•	•
	(Smallest)	0	00...01
Zero		0	00...00
Negatives	(Smallest)	1	11...11
		•	•
		•	•
		•	•
	(Largest/*Indefinite**)	1	00...00

Word: |←15 bits→|
Short: |←31 bits→|
Long: |←63 bits→|

* If this encoding is used as a source operand (as in an integer load or integer arithmetic instruction), the 8087 interprets it as the largest negative number representable in the format: -2^{15}, -2^{31}, or -2^{63}. The 8087 will deliver this encoding to an integer destination in two cases:

1) if the result is the largest negative number,

2) as the response to a masked invalid operation exception, in which case it represents the special value *integer indefinite*.

Exception Handling Details

Table S-32 lists every exception condition that the NDP detects and describes the processor's response when the relevant exception mask is set. The unmasked responses are described in table S-6. Note that if an unmasked overflow or underflow occurs in an FST or FSTP instruction, no result if stored, and the stack and memory are left as they existed *before* the instruction was executed. This gives an exception handler the opportunity to examine the offending operand on the stack top.

When rounding is directed (the RC field of the control word is set to "up" or "down"), the 8087 handles a masked overflow differently than it does for the "nearest" or "chop" rounding modes. Table S-33 shows the NDP's masked response when the true result is too large to be represented in it's destination real format. For a normalized result, the essence of this response is to deliver ∞ or the largest valid number representable in the destination format, as dictated by the rounding mode and the sign of the true result. Thus, when RC=down, a positive overflow is rounded down to the largest positive number. Conversely, when RC=up, a negative overflow is rounded up to the largest negative number. A properly signed ∞ is returned for a positive overflow with RC=up, or a negative overflow with RC=down. For an unnormalized result, the action is similar except that the the unnormal character of the result is preserved if the sign and rounding mode do not indicate that ∞ should be delivered.

In all masked overflow responses for directed rounding, the overflow flag is *not* set, but the precision exception *is* raised to signal that the exact true result has not been returned.

Table S-29. Packed Decimal Encodings

Class		Sign		Magnitude					
				digit	digit	digit	digit	. . .	digit
Positives	(Largest)	0	0000000	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	. . .	1 0 0 1
		• • •	• • •				• • •		
	(Smallest)	0	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 1
	Zero	0	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 0
Negatives	Zero	1	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 0
	(Smallest)	1	0000000	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	. . .	0 0 0 1
		• • •	• • •				• • •		
	(Largest)	1	0000000	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	. . .	1 0 0 1
*Indefinite**		1	1111111	1 1 1 1	1 1 1 1	U U U U	U U U U	. . .	U U U U

|←1 byte→|←9 bytes→|

* The *packed decimal indefinite* encoding is stored by FBSTP in response to a masked invalid operation exception. Attempting to load this value via FBLD produces an undefined result. Note: "UUUU" means bit values are undefined and may contain any value.

Table S-30. Real and Long Real Encodings

Class			Sign	Biased Exponent	Significand* $_{\Delta}$ff...ff
Positives		NANs	0 • • • 0	11...11 • • • 11...11	11...11 • • • 00...01
		∞	0	11...11	00...00
	Reals	Normals	0 • • • 0	11...10 • • • 00...01	11...11 • • • 00...00
		Denormals	0 • • • 0	00...00 • • • 00...00	11...11 • • • 00...01
		Zero	0	00...00	00...00

Table S-30. Real and Long Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand* $_\Delta$ff...ff
Negatives	Reals	Zero	1	00...00	00...00
		Denormals	1	00...00	00...01
			o	o	o
			o	o	o
			o	o	o
			1	00...00	11...11
		Normals	1	00...01	00...00
			o	o	o
			o	o	o
			o	o	o
			1	11...10	11...11
	∞		1	11...11	00...00
	NANs		1	11...11	00...01
			o	o	o
			o	o	o
			o	o	o
		Indefinite	1	11...11	10...00
			o	o	o
			o	o	o
			o	o	o
			1	11...11	11...11

Short: |◁— 8 bits —▷|◁—23 bits —▷|
Long: |◁—11 bits —▷|◁—52 bits —▷|

* Integer bit is implied and not stored.

Table S-31. Temporary Real Encodings

Class		Sign	Biased Exponent	Significand 1$_\Delta$ff..ff
Positives	NANs	0	11...11	111...11
		o	o	o
		o	o	o
		o	o	o
		0	11...11	100...01
	∞	0	11...11	100...00

Table S-31. Temporary Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand I_{Δ}ff...ff
Positives	Reals		0	11...10	Normals
			•	•	111...11
			•	•	•
			•	•	•
			•	•	•
			•	•	•
			•	•	100...00
			•	•	
			•	•	Unnormals
			•	•	
			•	•	011...11
			•	•	•
			•	•	•
			•	•	•
			0	00...01	000...00
					Denormals
			0	00...00	011...11
			•	•	•
			•	•	•
			•	•	•
			0	00...00	000...01
		Zero	0	00...00	000...00
Negatives		Zero	1	00...00	000...00
					Denormals
			1	00...00	000...01
			•	•	•
			•	•	•
			•	•	•
			1	00...00	011...11
			1	00...01	Unnormals
			•	•	000...00
			•	•	•
			•	•	•
			•	•	•
			•	•	011...11
			•	•	
			•	•	Normals
			•	•	
			•	•	100...00
			•	•	•
			•	•	•
			•	•	•
			1	11...10	111...11
	∞		1	11...11	100...00

Table S-31. Temporary Real Encodings (Cont'd.)

Class			Sign	Biased Exponent	Significand I_{Δ}ff...ff
Negatives	NANs		1	11...11	100...00
			•	•	•
			•	•	•
			•	•	•
		Indefinite	1	11...11	110...00
			•	•	•
			•	•	•
			•	•	•
			1	11...11	111...11
				←15 bits→	←64 bits→

Table S-32. Exception Conditions and Masked Responses

Condition	Masked Response
Invalid Operation	
Source register is tagged empty (usually due to stack underflow).	Return real *indefinite*.
Destination register is not tagged empty (usually due to stack overflow).	Return real *indefinite* (overwrite destination value).
One or both operands is a NAN.	Return NAN with larger absolute value (ignore signs).
(Compare and test operations only): one or both operands is a NAN.	Set condition codes "not comparable".
(Addition operations only): closure is affine and operands are opposite-signed infinities; or closure is projective and both operands are ∞ (signs immaterial).	Return real *indefinite*
(Subtraction operations only): closure is affine and operands are like-signed infinities; or closure is projective and both operands are ∞ (signs immaterial).	Return real *indefinite*.
(Multiplication operations only): ∞ * 0; or 0 * ∞.	Return real *indefinite*.
(Division operations only): ∞ ÷ ∞; or 0 ÷ 0; or 0 ÷ pseudo-zero; or divisor is denormal or unnormal.	Return real *indefinite*.
(FPREM instruction only): modulus (divisor) is unnormal or denormal; or dividend is ∞.	Return real *indefinite*, set condition code = "complete remainder".
(FSQRT instruction only): operand is nonzero and negative; or operand is denormal or unnormal; or closure is affine and operand is −∞; or closure is projective and operand is ∞.	Return real *indefinite*.

Exception Conditions and Masked Responses (Cont'd.)

Invalid Operation	
(Compare operations only): closure is projective and ∞ is being compared with 0 or a normal, or ∞.	Set condition code = "not comparable"
(FTST instruction only): closure is projective and operand is ∞.	Set condition code = "not comparable".
(FIST, FISTP instructions only): source register is empty, or a NAN, or denormal, or unnormal, or ∞, or exceeds representable range of destination.	Store integer *indefinite*.
(FBSTP instruction only): source register is empty, or a NAN, or denormal, or unnormal, or ∞, or exceeds 18 decimal digits.	Store packed decimal *indefinite*.
(FST, FSTP instructions only): destination is short or long real and source register is an unnormal with exponent in range.	Store real *indefinite*.
(FXCH instruction only): one or both registers is tagged empty.	Change empty register(s) to real *indefinite* and then perform exchange.
Denormalized Operand	
(FLD instruction only): source operand is denormal.	No special action; load as usual.
(Arithmetic operations only): one or both operands is denormal.	Convert (in a work area) the operand to the equivalent unnormal and proceed.
(Compare and test operations only): one or both operands is denormal *or unnormal* (other than pseudo-zero).	Convert (in a work area) any denormal to the equivalent unnormal; normalize as much as possible, and proceed with operation.
Zerodivide	
(Division operations only): divisor = 0.	Return ∞ signed with "exclusive or" of operand signs.
Overflow	
(Arithmetic operations only): rounding is nearest or chop, and exponent of true result > 16,383.	Return properly signed ∞and signal precision exception.
(FST, FSTP instructions only): rounding is nearest or chop, and exponent of true result > +127 (short real destination) or > +1023 (long real destination).	Return properly signed ∞ and signal precision exception.

Exception Conditions and Masked Responses (Cont'd.)

Underflow	
(Arithmetic operations only): exponent of true result <−16,382 (true).	Denormalize until exponent rises to −16,382 (true), round significand to 64 bits. If denormalized rounded significand = 0, then return true 0; else, return denormal (tag = special, biased exponent =0).
(FST, FSTP instructions only): destination is short real and exponent of true result <− 126 (true).	Denormalize until exponent rises to −126 (true), round significand to 24 bits, store true 0 if denormalized rounded significand = 0; else, store denormal (biased exponent = 0).
(FST, FSTP instructions only): destination is long real and exponent of true result <−1022 (true).	Denormalize until exponent rises to −1022 (true), round significand to 53 bits, store true 0 if rounded denormalized significand = 0; else, store denormal (biased exponent = 0).
Precision	
True rounding error occurs.	No special action.
Masked response to overflow exception earlier in instruction.	No special action.

Table S-33. Masked Overflow Response for Directed Rounding

True Result		Rounding Mode	Result Delivered
Normalization	Sign		
Normal	+	Up	+∞
Normal	+	Down	Largest finite positive number[1]
Normal	−	Up	Largest finite negative number[1]
Normal	−	Down	−∞
Unnormal	+	Up	+∞
Unnormal	−	Down	Largest exponent, result's significand[2]
Unnormal	+	Up	Largest exponent, result's significand[2]
Unnormal	−	Down	−∞

[1] The largest valid representable reals are encoded:
exponent: 11...10B
significand: $(1)_{\Delta}$11...10B

[2] The significand retains its identity as an unnormal; the true result is rounded as usual (effectively chopped toward 0 in this case). The exponent is encoded 11...10B.

S.10 Programming Examples

Conditional Branching

As discussed in section S.7, the comparison instructions post their results to the condition code bits of the 8087 status word. Although there are many ways to implement conditional branching following a comparison, the basic approach is as follows:

- execute the comparison,
- store the status word,
- inspect the condition code bits,
- jump on the result.

Figure S-26 is a code fragment that illustrates how two memory-resident long real numbers might be compared (similar code could be used with the FTST instruction). The numbers are called A and B, and the comparison is A to B. The comparison itself simply requires loading A onto the top of the 8087 register stack and then comparing it to B and popping the stack in the same instruction. The status word is written to memory and the code waits for completion of the store before attempting to use the result.

There are four possible orderings of A and B, and bits C3 and C0 of the condition code indicate which ordering holds. These bits are positioned in the upper byte of the status word so as to correspond to the CPU's zero and carry flags (ZF and CF), if the byte is written into the flags (see figures 2-32 and S-6). The code fragment, then, sets ZF and CF to the values of C3 and C0 and then uses the CPU conditional jumps to test the flags. Table 2-15 shows how each conditional jump instruction tests the CPU flags.

The FXAM instruction updates all four condition code bits. Figure S-27 shows how a jump table can be used to determine the characteristics of the value examined. The jump table (FXAM_TBL) is initialized to contain the 16-bit displacement of 16 labels, one for each possible condition code setting. Note that four of the table entries contain the same value, since there are four condition code settings that correspond to "empty."

The program fragment performs the FXAM and stores the status word. It then manipulates the condition code bits to finally produce a number in register BX that equals the condition code times 2. This involves zeroing the unused bits in the byte that contains the code, shifting C3 to the right so that it is adjacent to C2, and then shifting the code to multiply it by 2. The resulting value is used as an index which selects one of the displacements from FXAM_TBL (the multiplication of the condition code is required because of the 2-byte length of each value in FXAM_TBL). The unconditional JMP instruction effectively vectors through the jump table to the labelled routine that contains code (not shown in the example) to process each possible result of the FXAM instruction.

```
        .
        .
        .
A       DQ     ?
B       DQ     ?
STAT_87 DW     ?
        .
        .
        .
        FLD    A       ;LOAD A ONTO TOP OF 87 STACK
        FCOMP  B       ;COMPARE A:B, POP A
        FSTSW  STAT_87 ;STORE RESULT
        FWAIT          ;WAIT FOR STORE
```

Figure S-26. Conditional Branching for Compares

```
    ;
    ;LOAD CPU REGISTER AH WITH BYTE OF
    ;   STATUS WORD CONTAINING CONDITION CODE
    MOV    AH, BYTE PTR STAT_87+1
    ;
    ;LOAD CONDITION CODES INTO CPU FLAGS
    SAHF
    ;
    ;USE CONDITIONAL JUMPS TO DETERMINE
    ;   ORDERING OF A AND B
    JB     A_LESS_OR_UNORDERED
    ;CF (C0) = 0
    JNE    A_GREATER
A_EQUAL:
    ;CF (C0) = 0, ZF (C3) = 1
    .
    .
    .
A_GREATER:
    ;CF (C0) = 0, ZF (C3) = 0
    .
    .
    .
A_LESS_OR_UNORDERED:
    ;CF (C0) = 1, TEST ZF (C3)
    JNE    A_LESS
A_B_UNORDERED:
    ;CF (C0) = 1, ZF (C3) = 1
    .
    .
    .
A_LESS:
    ;CF (C0) = 1, ZF (C3) = 0
    .
    .
    .
```

Figure S-26. Conditional Branching for Compares (Cont'd.)

```
        .
        .
        .
FXAM_TBL            DW POS_UNNORM, POS_NAN, NEG_UNNORM,
&                       NEG_NAN, POS_NORM, POS_INFINITY,
&                       NEG_NORM, NEG_INFINITY, POS_ZERO,
&                       EMPTY, NEG_ZERO, EMPTY, POS_DENORM,
&                       EMPTY, NEG_DENORM, EMPTY
STAT_87             DW ?
```

Figure S-27. Conditional Branching for FXAM

```
        .
        .
        .
    ;EXAMINE ST, STORE RESULT, WAIT FOR COMPLETION
        FXAM
        FSTSW      STAT_87
        FWAIT
    ;CLEAR UPPER HALF OF BX, LOAD CONDITION CODE
    ;   IN LOWER HALF
        MOV        BH,0
        MOV        BL, BYTE PTR STAT_87+1
    ;COPY ORIGINAL IMAGE
        MOV        AL,BL
    ;CLEAR ALL BITS EXCEPT C2-C0
        AND        BL,00000111B
    ;CLEAR ALL BITS EXCEPT C3
        AND        AL,01000000B
    ;SHIFT C3 TWO PLACES RIGHT
        SHR        AL,1
        SHR        AL,1
    ;SHIFT C2-C0 ONE PLACE LEFT (MULTIPLY BY 2)
        SAL        BX,1
    ;DROP C3 BACK IN ADJACENT TO C2 (000XXXX0)
        OR         BL,AL
    ;JUMP TO THE ROUTINE ''ADDRESSED'' BY CONDITION CODE
        JMP        FXAM_TBL[BX]
    ;
    ;HERE ARE THE JUMP TARGETS, ONE TO HANDLE
    ;   EACH POSSIBLE RESULT OF FXAM
POS_UNNORM:
        .
        .
POS_NAN:
        .
        .
NEG_UNNORM:
        .
        .
NEG_NAN:
        .
        .
POS_NORM:
        .
        .
POS_INFINITY:
        .
        .
NEG_NORM:
        .
        .
NEG_INFINITY:
        .
```

Figure S-27. Conditional Branching for FXAM (Cont'd.)

```
        .
POS_ZERO:
        .
        .
EMPTY:
        .
        .
NEG_ZERO:
        .
        .
POS_DENORM:
        .
        .
NEG_DENORM:
        .
        .
        .
```

Figure S-27. Conditional Branching for FXAM (Cont'd.)

Exception Handlers

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler interrupt procedure as consisting of "prologue," "body" and "epilogue" sections of code. (For compatibility with the 8087 emulators, this procedure should be invoked by interrupt pointer (vector) number 16.)

At the beginning of the prologue, CPU interrupts have been disabled by the CPU's normal interrupt response mechanism. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically this will involve saving CPU registers and transferring diagnostic information from the 8087 to memory. When the critical processing has been completed, the prologue may enable CPU interrupts to allow higher-priority interrupt handlers to preempt the exception handler.

The exception handler body examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution.

The epilogue essentially reverses the actions of the prologue, restoring the CPU and the NDP so that normal execution can be resumed. The epilogue must *not* load an unmasked exception flag into the 8087 or another interrupt will be requested immediately (assuming 8087 interrupts are also loaded as unmasked).

Figures S-28 through S-30 show the ASM-86 coding of three skeleton exception handlers. They show how prologues and epilogues can be written for various situations, but only provide comments indicating where the application-dependent exception handling body should be placed.

Figures S-28 and S-29 are very similar; their only substantial difference is their choice of instructions to save and restore the 8087. The tradeoff here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. For applications that are sensitive to interrupt latency, or do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the CPU will not recognize another interrupt request (unless it is a non-maskable interrupt).

After the exception handler body, the epilogues prepare the CPU and the NDP to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the 8087 are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared). The prologue also provides for indicating to the interrupt controller hardware (e.g., 8259A) that the interrupt has been processed. The actual processing done here is application-dependent, but might typically involve writing an "end of interrupt" command to the interrupt controller.

The examples in figures S-28 and S-29 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in figure S-30 can be employed. The basic technique is to save the full 8087 state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

```
SAVE_ALL          PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE
;FOR 8087 STATE IMAGE
    PUSH      BP
       .
       .
       .
    MOV       BP,SP
    SUB       SP,94
;SAVE FULL 8087 STATE, WAIT FOR COMPLETION,
;ENABLE CPU INTERRUPTS
    FNSAVE    [BP-94]
    FWAIT
    STI
;
;APPLICATION-DEPENDENT EXCEPTION HANDLING
;CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED STATE
;IMAGE
    MOV       BYTE PTR [BP-92], 0H
    FRSTOR    [BP-94]
;WAIT FOR RESTORE TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV       SP,BP
       .
       .
       .
    POP       BP
;
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
;
;RETURN TO INTERRUPTED CALCULATION
    IRET
SAVE_ALL          ENDP
```

Figure S-28. Full State Exception Handler

```
SAVE_ENVIRONMENT PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE
;FOR 8087 ENVIRONMENT
    PUSH     BP
        .
        .
        .
    MOV      BP,SP
    SUB      SP,14
;SAVE ENVIRONMENT, WAIT FOR COMPLETION,
;ENABLE CPU INTERRUPTS
    FNSTENV  [BP-14]
    FWAIT
    STI
;
;APPLICATION EXCEPTION-HANDLING CODE GOES HERE
;
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED
;ENVIRONMENT IMAGE
    MOV      BYTE PTR [BP-12], 0H
    FLDENV   [BP-14]
;WAIT FOR LOAD TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV      SP,BP
        .
        .
        .
    POP      BP
;
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
;
;RETURN TO INTERRUPTED CALCULATION
    IRET
SAVE_ENVIRONMENT ENDP
```

Figure S-29. Reduced Latency Exception Handler

```
          .
          .
          .
      LOCAL_CONTROL  DW  ?  ;ASSUME INITIALIZED
          .
          .
          .
REENTRANT             PROC
;
;SAVE CPU REGISTERS, ALLOCATE STACK SPACE FOR
;8087 STATE IMAGE
    PUSH     BP
       .
       .
       .
    MOV      BP,SP
    SUB      SP,94
;SAVE STATE, LOAD NEW CONTROL WORD, WAIT
;FOR COMPLETION, ENABLE CPU INTERRUPTS
    FNSAVE   [BP-94]
    FLDCW    LOCAL_CONTROL
    FWAIT
    STI
;CODE TO SEND ''END OF INTERRUPT'' COMMAND TO
;8259A GOES HERE
       .
       .
       .
;APPLICATION EXCEPTION HANDLING CODE GOES HERE.
;AN UNMASKED EXCEPTION GENERATED HERE WILL
;CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
;IF LOCAL STORAGE IS NEEDED, IT MUST BE
;ALLOCATED ON THE CPU STACK.
       .
       .
       .
;CLEAR EXCEPTION FLAGS IN STATUS WORD
;RESTORE MODIFIED STATE IMAGE
    MOV      BYTE PTR [BP-92], 0H
    FRSTOR   [BP-94]
;WAIT FOR RESTORE TO FINISH BEFORE RELEASING MEMORY
    FWAIT
;DE-ALLOCATE STACK SPACE, RESTORE CPU REGISTERS
    MOV      SP,BP
       .
       .
       .
    POP      BP
;RETURN TO POINT OF INTERRUPTION
    IRET
REENTRANT             ENDP
```

Figure S-30. Reentrant Exception Handler

PRELIMINARY

iAPX 86/20
iAPX 88/20
NUMERIC DATA PROCESSOR

- **High Performance 2-Chip Numeric Data Processor**
- **Standard iAPX 86/10, 88/10 Instruction Set Plus Arithmetic, Trigonometric, Exponential, and Logarithmic Instructions For All Data Types**
- **All 24 iAPX 86/10, 88/10 Addressing Modes Available**
- **Conforms To Proposed IEEE Floating Point Standard**
- **Support 8 Data Types: 8-, 16-, 32-, 64-Bit Integers, 32-, 64-, 80-Bit Floating Point, and 18-Digit BCD Operands**
- **8x80-Bit Individually Addressable Register Stack plus 14 General Purpose Registers**
- **7 Built-in Exception Handling Functions**
- **MULTIBUS System Compatible Interface**

The Intel iAPX 86/20 and iAPX 88/20 are two-chip numeric data processors (NDP's). They provide the instructions and data types needed for high-performance numeric applications. The NDP provides 100 times the performance of an iAPX 86/10, 88/10 CPU alone for numeric processing. The iAPX 86/20 consists of an iAPX 86/10 (16-bit 8086 CPU) and a numeric processor extension (NPX), the 8087. The iAPX 88/20 consists of the NPX in conjunction with the iAPX 88/10 (8-bit 8088 CPU). The NDP conforms to the proposed IEEE Floating Point Standard.

Both components of the iAPX 86/20 and iAPX 88/20 are implemented in N-channel, depletion load, silicon gate technology (HMOS), housed in two 40-pin packages. The iAPX 86/20, 88/20 adds 68 numeric processing instructions to the iAPX 86/10, 88/10 instruction set and eight 80-bit registers to the register set.

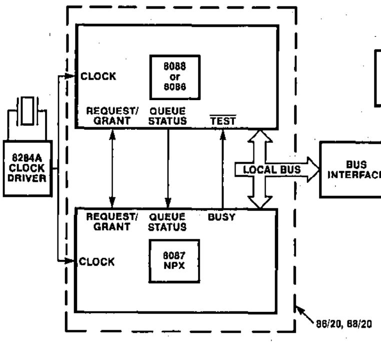


Figure 1. iAPX 86/20, 88/20 Block Diagram

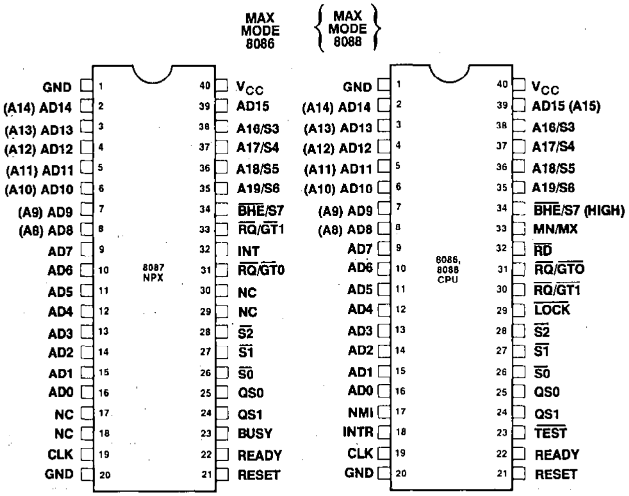


Figure 2. iAPX 86/20, 88/20 Pin Configuration

Intel Corporation Assumes No Responsibilty for the Use of Any Circuitry Other Than Circuitry Embodied in an Intel Product. No Other Circuit Patent Licenses are Implied.

JULY 1981

Table 1. 8087 Pin Description

Symbol	Type	Name and Function
AD15–AD0	I/O	**Address Data:** These lines constitute the time multiplexed memory address (T_1) and data (T_2, T_3, T_W, T_4) bus. A0 is analogous to $\overline{BHE}$ for the lower byte of the data bus, pins D7–D0. It is LOW during T_1 when a byte is to be transferred on the lower portion of the bus in memory operations. Eight-bit oriented devices tied to the lower half of the bus would normally use A0 to condition chip select functions. These lines are active HIGH. They are input/output lines for 8087 driven bus cycles and are inputs which the 8087 monitors when the 8086/8088 is in control of the bus. A15-A8 do not require an address latch in an iAPX 88/20. The 8087 will supply an address for the T_1-T_4 period.
A19/S6, A18/S5, A17/S4, A16/S3	I/O	**Address Memory:** During T_1 these are the four most significant address lines for memory operations. During memory operations, status information is available on these lines during T_2, T_3, T_W, and T_4. For 8087 controlled bus cycles, S6, S4, and S3 are reserved and currently one (HIGH), while S5 is always LOW. These lines are inputs which the 8087 monitors when the 8086/8088 is in control of the bus.
BHE/S7	I/O	**Bus High Enable:** During T_1 the bus high enable signal ($\overline{BHE}$) should be used to enable data onto the most significant half of the data bus, pins D15–D8. Eight-bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during T_1 for read and write cycles when a byte is to be transferred on the high portion of the bus. The S7 status information is available during T_2, T_3, T_W, and T_4. The signal is active LOW. S7 is an input which the 8087 monitors during 8086/8088 controlled bus cycles.
$\overline{S2}$, $\overline{S1}$, $\overline{S0}$	I/O	**Status:** For 8087 driven bus cycles, these status lines are encoded as follows: **$\overline{S2}$ $\overline{S1}$ $\overline{S0}$** 0 (LOW) X X Unused 1 (HIGH) 0 0 Unused 1 0 1 Read Memory 1 1 0 Write Memory 1 1 1 Passive Status is driven active during T_4, remains valid during T_1 and T_2, and is returned to the passive state (1, 1, 1) during T_3 or during T_W when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory access control signals. Any change in $\overline{S2}$, $\overline{S1}$, or $\overline{S0}$ during T_4 is used to indicate the beginning of a bus cycle, and the return to the passive state in T_3 or T_W is used to indicate the end of a bus cycle. These signals are monitored by the 8087 when the 8086/8088 is in control of the bus.
$\overline{RQ}/\overline{GT0}$	I/O	**Request/Grant:** This request/grant pin is used by the NPX to gain control of the local bus from the CPU for operand transfers or on behalf of another bus master. It must be connected to one of the two processor request/grant pins. The request grant sequence on this pin is as follows: 1. A pulse one clock wide is passed to the CPU to indicate a local bus request by either the 8087 or the master connected to the 8087 $\overline{RQ}/\overline{GT1}$ pin. 2. The 8087 waits for the grant pulse and when it is received will either initiate bus transfer activity in the clock cycle following the grant or pass the grant out on the $\overline{RQ}/\overline{GT1}$ pin in this clock if the initial request was for another bus master. 3. The 8087 will generate a release pulse to the CPU one clock cycle after the completion of the last 8087 bus cycle or on receipt of the release pulse from the bus master on $\overline{RQ}/\overline{GT1}$.

Table 1. 8087 Pin Description (Continued)

Symbol	Type	Name and Function
$\overline{RQ}/\overline{GT}1$	I/O	**Request/Grant:** This request/grant pin is used by another local bus master to force the 8087 to request the local bus. If the 8087 is not in control of the bus when the request is made the request/grant sequence is passed through the 8087 on the $\overline{RQ}/\overline{GT0}$ pin one cycle later. Subsequent grant and release pulses are also passed through the 8087 with a two and one clock delay, respectively, for resynchronization. $\overline{RQ}/\overline{GT}1$ has has an internal pullup resistor, and so may be left unconnected. If the 8087 has control of the bus the request/grant sequence is as follows: 1. A pulse 1 CLK wide from another local bus master indicates a local bus request to the 8087 (pulse 1). 2. During the 8087's next T_4 or T_1 a pulse 1 CLK wide from the 8087 to the requesting master (pulse 2) indicates that the 8087 has allowed the local bus to float and that it will enter the "RQ/GT acknowledge" state at the next CLK. The 8087's control unit is disconnected logically from the local bus during "RQ/GT acknowledge." 3. A pulse 1 CLK wide from the requesting master indicates to the 8087 (pulse 3) that the "RQ/GT" request is about to end and that the 8087 can reclaim the local bus at the next CLK. Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW.
QS1, QS0	I	**QS1, QS0:** QS1 and QS0 provide the 8087 with status to allow tracking of the CPU instruction queue. **QS1 QS0** 0 (LOW) 0 No Operation 0 1 First Byte of Op Code from Queue 1 (HIGH) 0 Empty the Queue 1 1 Subsequent Byte from Queue
INT	O	**Interrupt:** This line is used to indicate that an unmasked exception has occurred during numeric instruction execution when 8087 interrupts are enabled. This signal is typically routed to an 8259A. INT is active HIGH.
BUSY	O	**Busy:** This signal indicates that the 8087 NEU is executing a numeric instruction. It is connected to the CPU's $\overline{TEST}$ pin to provide synchronization. In the case of an unmasked exception BUSY remains active until the exception is cleared. BUSY is active HIGH.
READY	I	**Ready:** READY is the acknowledgment from the addressed memory device that it will complete the data transfer. The RDY signal from memory is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH.
RESET	I	**Reset:** RESET causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. RESET is internally synchronized.
CLK	I	**Clock:** The clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.
V_{CC}		**Power:** V_{CC} is the +5V power supply pin.
GND		**Ground:** GND are the ground pins.

NOTE:
For the pin descriptions of the 8086 and 8088 CPU's reference those respective data sheets (iAPX 86/10, iAPX 88/10).

APPLICATION AREAS

The iAPX 86/20 and iAPX 88/20 provide functions meant specifically for high performance numeric processing requirements. Trigonometric, logarithmic, and exponential functions are built into the processor hardware. These functions are essential in scientific, engineering, navigational, or military applications.

The NDP also has capabilities meant for business or commercial computing. An iAPX 86/20, 88/20 can process Binary Coded Decimal (BCD) numbers up to 18 digits without roundoff errors. It can also perform arithmetic on integers as large as 64 bits ($\pm 10^{18}$).

PROGRAMMING LANGUAGE SUPPORT

Programs for the iAPX 86/20 and iAPX 88/20 can be written in ASM-86, the iAPX 86,88 assembly language, PL/M-86, FORTRAN-86, and PASCAL-86, Intel's high-level languages for iAPX 86, 88 systems.

Details

The remainder of the data sheet will concentrate on the numeric processor extension (refered to as NPX or 8087). For iAPX 86/10 or iAPX 88/10 CPU details refer to those respective data sheets.

FUNCTIONAL DESCRIPTION

The iAPX 86/20, 88/20 Numeric Data Processor's architecture is designed for high performance numeric computing in conjunction with general purpose processing.

The 8087 is a numeric processor extension that provides arithmetic and logical instruction support for a variety of numeric data types in iAPX 86/20, 88/20 systems. It also executes numerous built-in transcendental functions (e.g., tangent and log functions). The 8087 executes instructions as a coprocessor to a maximum mode 8086 or 8088. It effectively extends the register and instruction set of an iAPX 86/10 or 88/10 based system and adds several new data types as well. Figure 3 presents the registers of the iAPX 86/20. Table 2 shows the range of data types supported by the NDP. The 8087 is treated as an extension to the iAPX 86/10 or 88/10, providing register, data types, control, and instruction capabilities at the hardware level. At the programmers level the iAPX 86/20, 88/20 is viewed as a single unified processor.

iAPX 86/20, 88/20 System Configuration

As a coprocessor to an 8086 or 8088, the 8087 is wired in parallel with the CPU as shown in Figure 4. The CPU's status ($\overline{\text{S0}}$-$\overline{\text{S2}}$) and queue status lines (QS0-QS1) enable the 8087 to monitor and decode

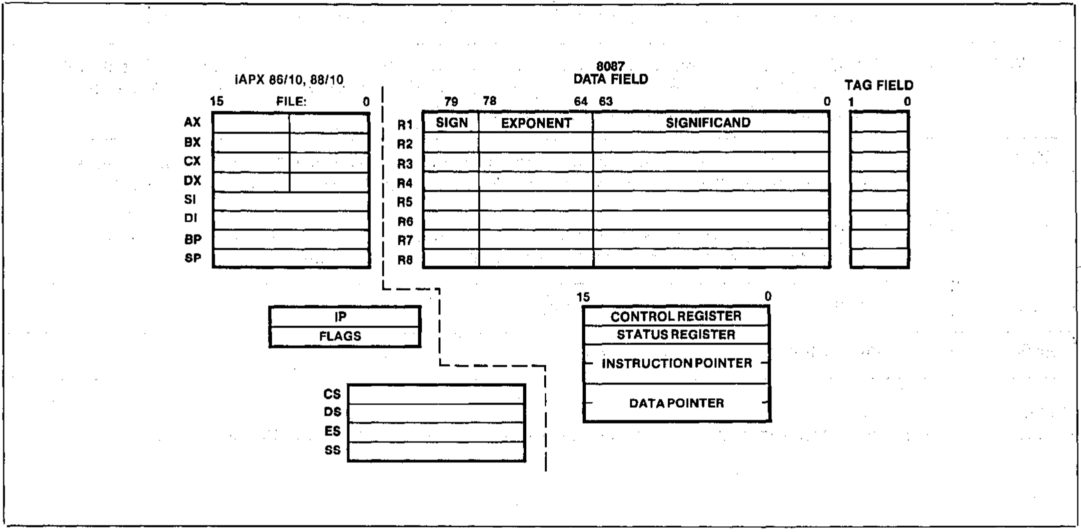


Figure 3. iAPX 86/20 Architecture

Table 2. iAPX 86/20, 88/20 Data Types

Data Formats	Range	Precision	Most Significant Byte 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0 7 0
Byte Integer	10^2	8 Bits	I_7 I_0 Two's Complement
Word Integer	10^4	16 Bits	I_{15} I_0 Two's Complement
Short Integer	10^9	32 Bits	I_{31} I_0 Two's Complement
Long Integer	10^{18}	64 Bits	I_{63} I_0 Two's Complement
Packed BCD	10^{18}	18 Digits	S — $D_{17}D_{16}$ D_1 D_0
Short Real	$10^{\pm38}$	24 Bits	S E_7 E_0 F_1 F_{23} F_0 Implicit
Long Real	$10^{\pm308}$	53 Bits	S E_{10} E_0 F_1 F_{52} F_0 Implicit
Temporary Real	$10^{\pm4932}$	64 Bits	S E_{14} E_0 F_0 F_{63}

Integer: I

Packed BCD: $(-1)^S(D_{17}\ldots D_0)$

Real: $(-1)^S(2^{E-BIAS})(F_0 \cdot F_1 \ldots)$

Bias=127 for Short Real
1023 for Long Real
16383 for Temp Real

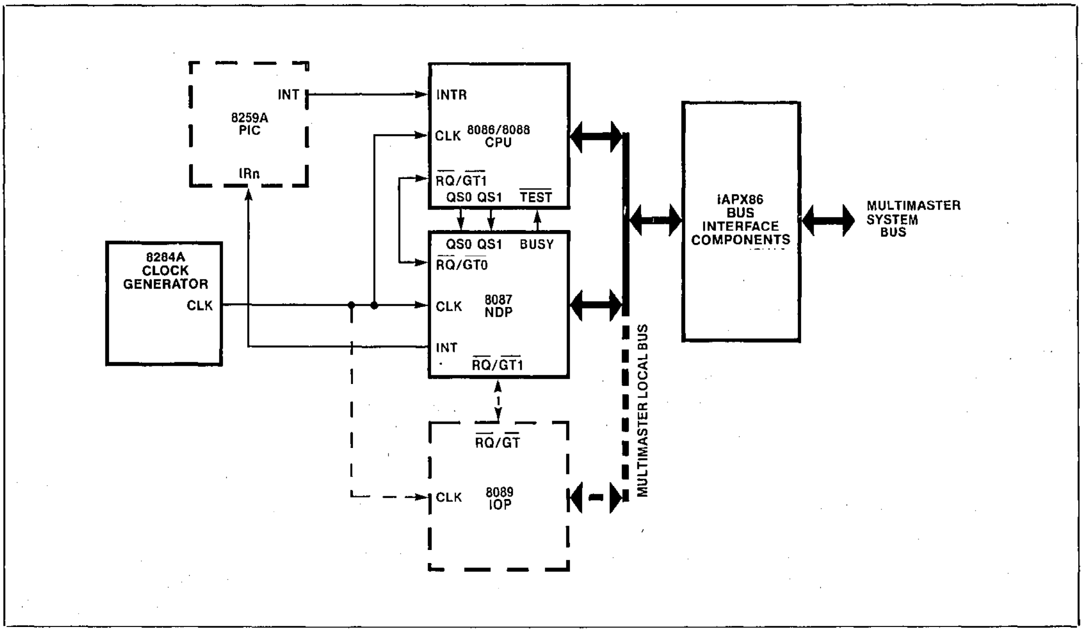


Figure 4. NDP System Configuration

instructions in synchronization with the CPU and without any CPU overhead. Once started the 8087 can process in parallel with and independent of the host CPU. For resynchronization, the NPX's BUSY signal informs the CPU that the NPX is executing an instruction and the CPU WAIT instruction tests this signal to insure that the NPX is ready to execute subsequent instructions. The NPX can interrupt the CPU when it detects an error or exception. The 8087's interrupt request line is typically routed to the CPU through an 8259A Programmable Interrupt Controller. (See Figure 2 for 8087 pinout information.)

The 8087 uses one of the request/grant lines of the iAPX 86, 88 architecture (typically $\overline{\text{RQ/GT1}}$) to obtain control of the local bus for data transfers. The other request/grant line is available for general system use (for instance by an I/O processor in LOCAL mode). A bus master can also be connected to the 8087's $\overline{\text{RQ/GT1}}$ line. In this configuration the 8087 will pass the request/grant handshake signals between the CPU and the attached master when the 8087 is not in control of the bus and will relinquish the bus to the master directly when the 8087 is in control. In this way two additional masters can be configured in an iAPX 86/20, 88/20 system; one will share the 8086 bus with the 8087 on a first come first served basis, and the second will be guaranteed to be higher in priority than the 8087.

As Figure 4 shows, all processors utilize the same clock generator and system bus interface components (bus controller, latches, transceivers and bus arbiter).

Bus Operation

The 8087 bus structure, operation and timing are identical to all other processors in the iAPX 86, 88 series (maximum mode configuration). The address is time multiplexed with the data on the first 16/8 lines of the address/data bus. A16 through A19 are time multiplexed with four status lines S3–S6. S3, S4 and S6 are always one (high) for 8087 driven bus cycles while S5 is always zero (low). When the 8087 is monitoring CPU bus cycles (passive mode) S6 is also monitored by the 8087 to differentiate 8086/8088 activity from that of a local I/O processor or any other local bus master. (The 8086/8088 must be the only processor on the local bus to drive S6 low.) S7 is multiplexed with and has the same value as $\overline{\text{BHE}}$ for all 8087 bus cycles.

The first three status lines, $\overline{\text{S0}}$-$\overline{\text{S2}}$, are used with an 8288 bus controller to determine the type of bus cycle being run:

$\overline{\text{S2}}$	$\overline{\text{S1}}$	$\overline{\text{S0}}$	
0	X	X	Unused
1	0	0	Unused
1	0	1	Memory Data Read
1	1	0	Memory Data Write
1	1	1	Passive (no bus cycle)

Programming Interface

The NDP includes the standard iAPX 86/10, 88/10 instruction set for general data manipulation and program control. It also includes 68 numeric instructions for extended precision integer, floating point, trigonometric, logarithmic, and exponential functions. Sample execution times for several NDP functions are shown in Figure 4. Overall iAPX 86/20 system performance is 100 times that of an iAPX 86/10 class processor for numeric instructions.

Any instruction executed by the NDP is the combined result of the CPU and NPX activity. The CPU and the NPX have specialized functions and registers providing fast concurrent operation. The CPU controls overall program execution while the NPX uses the coprocessor interface to recognize and perform numeric operations.

Table 2 lists the eight data types the iAPX 86/20, 88/20 supports and presents the format for each type. Internally, the NPX holds all numbers in the temporary real format. Load and store instructions automatically convert operands represented in memory as 16-, 32-, or 64-bit integers, 32- or 64-bit floating point numbers or 18-digit packed BCD numbers into temporary real format and vice versa. The NDP also provides the capability to control round off, underflow, and overflow errors in each calculation.

Computations in the NPX use the processor's register stack. These eight 80-bit registers provide the equivalent capacity of 20 32-bit registers. The NPX register set can be accessed as a stack, with instructions operating on the top one or two stack elements, or as a fixed register set, with instructions operating on explicitly designated registers.

Table 5 lists the 8087's instructions by class. All appear as ESCAPE instructions to the host. Assembly language programs are written in ASM-86, the iAPX 86, 88 assembly language. Table 3 gives the execution times of some typical numeric instructions.

Table 3. Execution Times for Selected iAPX 86/20 Numeric Instructions and Corresponding iAPX 86/10 Emulation

Floating Point Instruction	Approximate Execution Time (μs)	
	iAPX 86/20 (5 MHz Clock)	iAPX 86/10 Emulation
Add/Subtract	17	1,600
Multiply (single precision)	19	1,600
Multiply (extended precision)	27	2,100
Divide	39	3,200
Compare	9	1,300
Load (double precision)	10	1,700
Store (double precision)	21	1,200
Square Root	36	19,600
Tangent	90	13,000
Exponentiation	100	17,100

NUMERIC PROCESSOR EXTENSION ARCHITECTURE

As shown in Figure 5, the 8087 is internally divided into two processing elements, the control unit (CU) and the numeric execution unit (NEU). The NEU executes all numeric instructions, while the CU receives and decodes instructions, reads and writes memory operands and executes NPX control instructions. The two elements are able to operate independently of one another, allowing the CU to maintain synchronization with the CPU while the NEU is busy processing a numeric instruction.

Control Unit

The CU keeps the 8087 operating in synchronization with its host CPU. 8087 instructions are intermixed with CPU instructions in a single instruction stream. The CPU fetches all instructions from memory; by monitoring the status signals ($\overline{S0}$-$\overline{S2}$, S6) emitted by the CPU, the NPX control unit determines when an

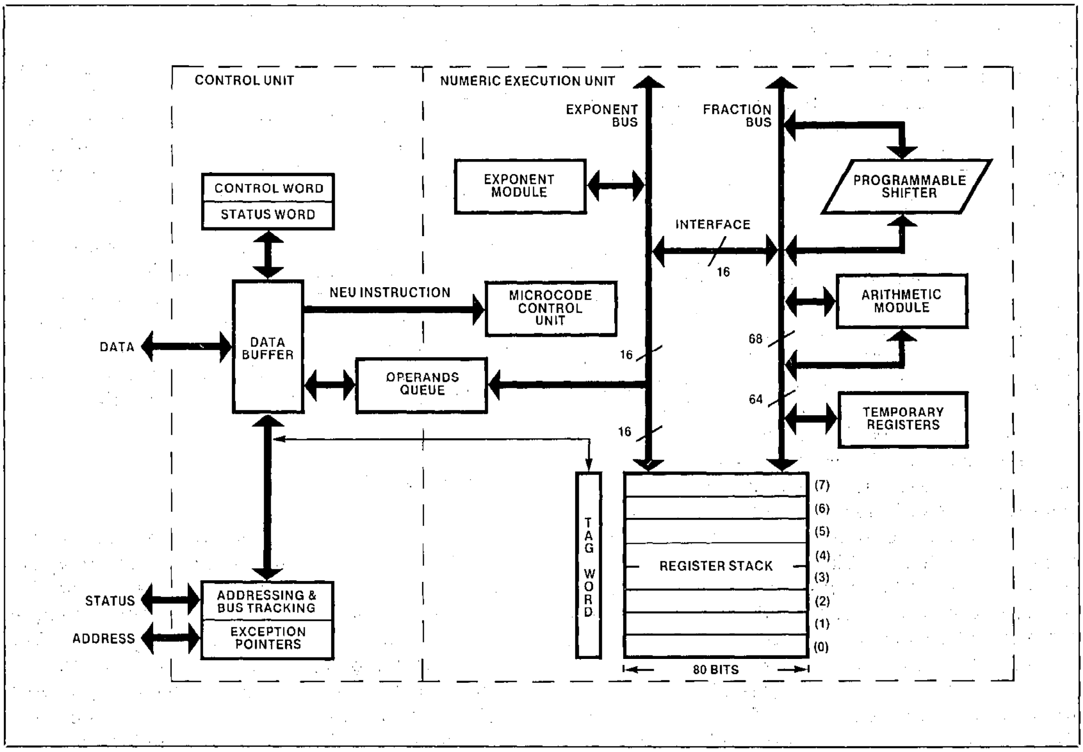


Figure 5. 8087 Block Diagram

8086 instruction is being fetched. The CU monitors the Data bus in parallel with the CPU to obtain instructions that pertain to the 8087.

The CU maintains an instruction queue that is identical to the queue in the host CPU. The CU automatically determines if the CPU is an 8086 or an 8088 immediately after reset (by monitoring the $\overline{\text{BHE}}$/ S7 line) and matches its queue length accordingly. By monitoring the CPU's queue status lines (QS0, QS1), the CU obtains and decodes instructions from the queue in synchronization with the CPU.

A numeric instruction appears as an ESCAPE instruction to the 8086 or 8088 CPU. Both the CPU and NPX decode and execute the ESCAPE instruction together. The 8087 only recognizes the numeric instructions shown in Table 5. The start of a numeric operation is accomplished when the CPU executes the ESCAPE instruction. The instruction may or may not identify a memory operand.

The CPU does, however, distinguish between ESC instructions that reference memory and those that do not. If the instruction refers to a memory operand, the CPU calculates the operand's address using any one of its available addressing modes, and then performs a "dummy read" of the word at that location. (Any location within the 1M byte address space is allowed.) This is a normal read cycle except that the CPU ignores the data it receives. If the ESC instruction does not contain a memory reference (e.g. an 8087 stack operation), the CPU simply proceeds to the next instruction.

An 8087 Instruction can have one of three memory reference options; (1) not reference memory; (2) load an operand word from memory into the 8087; or (3) store an operand word from the 8087 into memory. If no memory reference is required, the 8087 simply executes its instruction. If a memory reference is required, the CU uses a "dummy read" cycle initiated by the CPU to capture and save the address that the CPU places on the bus. If the instruction is a load, the CU additionally captures the data word when it becomes available on the local data bus. If data required is longer than one word, the CU immediately obtains the bus from the CPU using the request/grant protocol and reads the rest of the information in consecutive bus cycles. In a store operation, the CU captures and saves the store address as in a load, and ignores the data word that follows in the "dummy read" cycle. When the 8087 is ready to perform the store, the CU obtains the bus from the CPU and writes the operand starting at the specified address.

Numeric Execution Unit

The NEU executes all instructions that involve the register stack; these include arithmetic, logical, transcendental, constant and data transfer instructions. The data path in the NEU is 84 bits wide (68 fraction bits, 15 exponent bits and a sign bit) which allows internal operand transfers to be performed at very high speeds.

When the NEU begins executing an instruction, it activates the 8087 BUSY signal. This signal can be used in conjunction with the CPU WAIT instruction to resynchronize both processors when the NEU has completed its current instruction.

Register Set

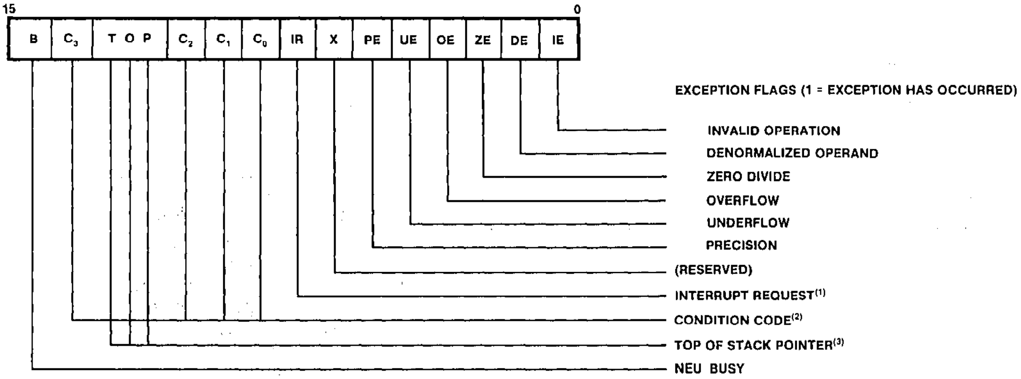
The iAPX 86/20 register set is shown in Figure 3. Each of the eight data registers in the 8087's register stack is 80 bits wide and is divided into "fields" corresponding to the NDP's temporary real data type.

At a given point in time the TOP field in the control word identifies the current top-of-stack register. A "push" operation decrements TOP by 1 and loads a value into the new top register. A "pop" operation stores the value from the current top register and then increments TOP by 1. Like iAPX 86/10, 88/10 stacks in memory, the 8087 register stack grows "down" toward lower-addressed registers.

Instructions may address the data registers either implicitly or explicitly. Many instructions operate on the register at the top of the stack. These instructions implicitly address the register pointed to by the TOP. Other instructions allow the programmer to explicitly specify the register which is to be used. Explicit register addressing is "top-relative."

Status Word

The status word shown in Figure 6 reflects the overall state of the 8087; it may be stored in memory and then inspected by CPU code. The status word is a 16-bit register divided into fields as shown in Figure 6. The busy bit (bit 15) indicates whether the *NEU* is either executing an instruction or has an interrupt request pending (B = 1), or is idle (B = 0). Several instructions which store and manipulate the status word are executed exclusively by the CU, and these do not set the busy bit themselves.



[1] IR is set if any unmasked exception bit is set, cleared otherwise.
[2] See Table 3 for condition code interpretation.
[3] Top Values:
000 = Register 0 is Top of Stack.
001 = Register 1 is Top of Stack.
•
•
•
111 = Register 7 is Top of Stack.

Figure 6. 8087 Status Word

The four numeric condition code bits (C_0–C_3) are similar to the flags in a CPU: various instructions update these bits to reflect the outcome of NDP operations. The effect of these instructions on the condition code bits is summarized in Table 4.

Bits 14–12 of the status word point to the 8087 register that is the current top-of-stack (TOP) as described above.

Bit 7 is the interrupt request bit. This bit is set if any unmasked exception bit is set and cleared otherwise.

Bits 5–0 are set to indicate that the NEU has detected an exception while executing an instruction.

Tag Word

The tag word marks the content of each register as shown in Figure 7. The principal function of the tag word is to optimize the NDP's performance. The tag word can be used, however, to interpret the contents of 8087 registers.

Instruction and Data Pointers

The instruction and data pointers (see Figure 8) are provided for user-written error handlers. Whenever the 8087 executes an NEU instruction, the CU saves the instruction address, the operand address (if present) and the instruction opcode. 8087 instructions can store this data into memory.

Table 4. Condition Code Interpretation

Instruction	C_3	C_2	C_1	C_0	Interpretation
Compare, Test	0	X	X	0	A > B
	0	X	X	1	A < B
	1	X	X	0	A = B
	1	X	X	1	A ? B (not comparable)
Remainder	Q_1	0	Q_0	Q_2	Complete reduction
	Q_1	1	Q_0	Q_2	Incomplete reduction
Examine	0	0	0	0	Valid, positive, unnormalized
	0	0	0	1	Invalid, positive, exponent ≠ 0
	0	0	1	0	Valid, negative, unnormalized
	0	0	1	1	Invalid, negative, exponent ≠ 0
	0	1	0	0	Valid, positive, normalized
	0	1	0	1	Infinity, positive
	0	1	1	0	Valid, negative, normalized
	0	1	1	1	Infinity, negative
	1	0	0	0	Zero, positive
	1	0	0	1	Empty
	1	0	1	0	Zero, negative
	1	0	1	1	Empty
	1	1	0	0	Invalid, positive, exponent = 0
	1	1	0	1	Empty
	1	1	1	0	Invalid, negative, exponent = 0
	1	1	1	1	Empty

X = value is not affected by instruction.
Q = C_0, C_3, C_1 hold 3 LSBs of the quotient generated during a remainder operation.

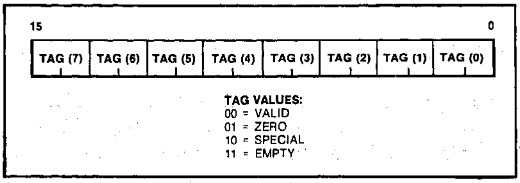


Figure 7. 8087 Tag Word

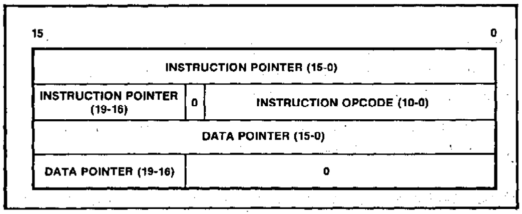

15
0
INSTRUCTION POINTER (15-0)
INSTRUCTION POINTER (19-16)
0
INSTRUCTION OPCODE (10-0)
DATA POINTER (15-0)
DATA POINTER (19-16)
0

Figure 8. 8087 Instruction and Data Pointers

Control Word

The 8087 provides several processing options which are selected by loading a word from memory into the control word. Figure 9 shows the format and encoding of the fields in the control word.

The low order byte of this control word configures 8087 interrupts and exception masking. Bits 5–0 of the control word contain individual masks for each of the six exceptions that the 8087 recognizes and bit 7 contains a general mask bit for all 8087 interrupts. The high order byte of the control word configures the 8087 operating mode including precision, rounding, and infinity controls. The precision control bits (bits 9–8) can be used to set the 8087 internal operating precision at less than the default of temporary real precision. This can be useful in providing compatibility with earlier generation arithmetic processors of smaller precision than the 8087. The rounding control bits (bits 11–10) provide for directed rounding and true chop as well as the unbiased round to nearest mode specified in the proposed IEEE standard. Control over closure of the number space at infinity is also provided (either affine closure, ±∞, or projective closure, ∞, is treated as unsigned, may be specified).

Exception Handling

The 8087 detects six different exception conditions that can occur during instruction execution. Any or all exceptions will cause an interrupt if unmasked and interrupts are enabled.

If interrupts are disabled the 8087 will simply continue execution regardless of whether the host clears the exception. If a specific exception class is masked and that exception occurs, however, the 8087 will post the exception in the status register and perform an on-chip default exception handling procedure, thereby allowing processing to continue. The exceptions that the 8087 detects are the following:

1. INVALID OPERATION: Stack overflow, stack underflow, indeterminate form (0/0, ∞– ∞, etc.) or the use of a Non-Number (NAN) as an operand. An exponent value is reserved and any bit pattern with this value in the exponent field is termed a Non-Number and causes this exception. If this exception is masked, the 8087's default response is to generate a specific NAN called INDEFINITE, or to propagate already existing NANs as the calculation result.

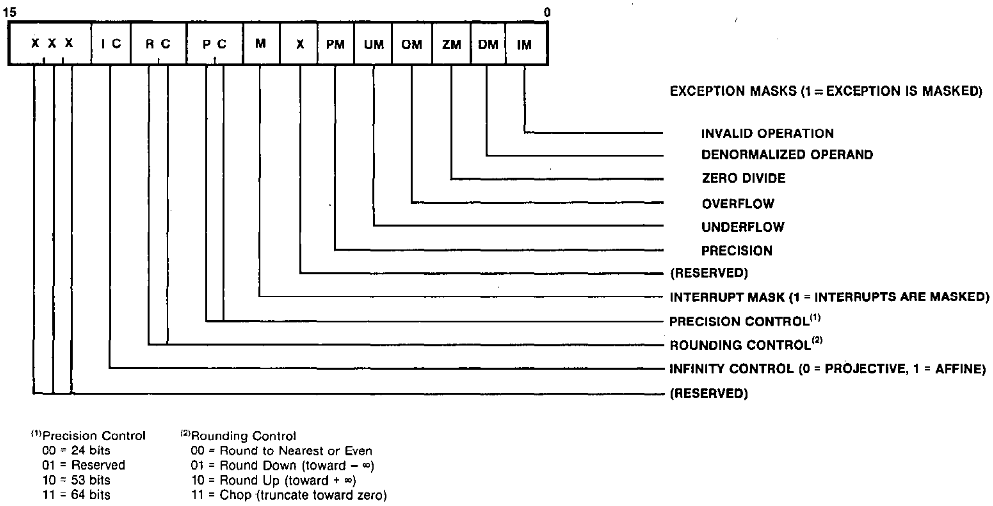


Figure 9. 8087 Control Word

2. OVERFLOW: The result is too large in magnitude to fit the specified format. The 8087 will generate an encoding for infinity if this exception is masked.

3. ZERO DIVISOR: The divisor is zero while the dividend is a non-infinite, non-zero number. Again, the 8087 will generate an encoding for infinity if this exception is masked.

4. UNDERFLOW: The result is non-zero but too small in magnitude to fit in the specified format. If this exception is masked the 8087 will denormalize (shift right) the fraction until the exponent is in range. This process is called gradual underflow.

5. DENORMALIZED OPERAND: At least one of the operands or the result is denormalized; it has the smallest exponent but a non-zero significand. Normal processing continues if this exception is masked off.

6. INEXACT RESULT: If the true result is not exactly representable in the specified format, the result is rounded according to the rounding mode, and this flag is set. If this exception is masked, processing will simply continue.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias ....0°C to 70°C
Storage Temperature ....−65°C to +150°C
Voltage on Any Pin with
Respect to Ground ....−1.0V to +7V
Power Dissipation ....3.0 Watt

**NOTICE: Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = +5V ±10%)

Symbol	Parameter	Min.	Max.	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	+0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} +0.5	V	
V_{OL}	Output Low Voltage		0.45	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −400 μA
I_{CC}	Power Supply Current		475	mA	T_A = 25°C
I_{LI}	Input Leakage Current		±10	μA	0V ≤ V_{IN} ≤ V_{CC}
I_{LO}	Output Leakage Current		±10	μA	0.45V ≤ V_{OUT} ≤ V_{CC}
V_{CL}	Clock Input Low Voltage	−0.5	+0.6	V	
V_{CH}	Clock Input High Voltage	3.9	V_{CC} + 1.0	V	
C_{IN}	Capacitance of Inputs		10	pF	fc = 1 MHz
C_{IO}	Capacitance of I/O Buffer (AD0-15, A_{16}–A_{19}, BHE, S2-S0, RQ/GT) and CLK		15	pF	fc = 1 MHz
C_{OUT}	Capacitance of Outputs BUSY, INT		10	pF	fc = 1 MHz

A.C. CHARACTERISTICS (T_A = 0°C to 70°C, V_{CC} = +5V ±10%)

TIMING REQUIREMENTS

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLCL	CLK Cycle Period	200	500	ns	
TCLCH	CLK Low Time	(⅔ TCLCL) − 15		ns	
TCHCL	CLK High Time	(⅓ TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		ns	
TCLDX	Data In Hold Time	10		ns	
TRYHCH	READY Setup Time	(⅔ TCLCL) − 15		ns	
TCHRYX	READY Hold Time	30		ns	
TRYLCL	READY Inactive to CLK (See Note 3)	−8		ns	
TGVCH	RQ/GT Setup Time	30		ns	
TCHGX	RQ/GT Hold Time	40		ns	
TQVCL	QS0-1 Setup Time	30		ns	
TCLQX	QS0-1 Hold Time	10		ns	
TSACH	Status Active Setup Time	30		ns	
TSNCL	Status Inactive Setup Time	30		ns	
TILIH	Input Rise Time (Except CLK)		20	ns	From 0.8V to 2.0V
TIHIL	Input Fall Time (Except CLK)		12	ns	From 2.0V to 0.8V

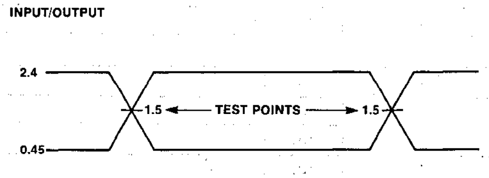
A.C. CHARACTERISTICS (Continued)

TIMING RESPONSES

Symbol	Parameter	Min.	Max.	Units	Test Conditions
TCLML	Command Active Delay (See Note 1)	10	35	ns	C_L = 20–100 pF for all 8087 Outputs (in addition to 8087 self-load)
TCLMH	Command Inactive Delay (See Note 1)	10	35	ns	
TRYHSH	Ready Active to Status Passive (See Note 2)		110	ns	
TCHSV	Status Active Delay	10	110	ns	
TCLSH	Status Inactive Delay	10	130	ns	
TCLAV	Address Valid Delay	10	110	ns	
TCLAX	Address Hold Time	10		ns	
TCLAZ	Address Float Delay	TCLAX	80	ns	
TSVLH	Status Valid to ALE High (See Note 1)		15	ns	
TCLLH	CLK Low to ALE Valid (See Note 1)		15	ns	
TCHLL	ALE Inactive Delay (See Note 1)		15	ns	
TCLDV	Data Valid Delay	10	110	ns	
TCHDX	Data Hold Time	10		ns	
TCVNV	Control Active Delay (See Note 1)	5	45	ns	
TCVNX	Control Inactive Delay (See Note 1)	10	45	ns	
TCHBV	BUSY and INT Valid Delay	10	150	ns	
TCHDTL	Direction Control Active Delay (See Note 1)		50	ns	
TCHDTH	Direction Control Inactive Delay (See Note 1)		30	ns	
TCLGL	RQ/GT Active Delay	0	85	ns	C_L = 40 pF (in addition to 8087 self-load)
TCLGH	RQ/GT Inactive Delay	0	85	ns	
TOLOH	Output Rise Time		20	ns	From 0.8V to 2.0V
TOHOL	Output Fall Time		12	ns	From 2.0V to 0.8V

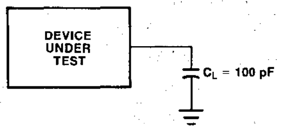
NOTES:
1. Signal at 8284A or 8288 shown for reference only.
2. Applies only to T_3 and wait states.
3. Applies only to T_2 state (8 ns into T_3).

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. TESTING: INPUTS ARE DRIVEN AT 2.4V FOR A LOGIC "1" AND 0.45V FOR A LOGIC "0." THE CLOCK IS DRIVEN AT 4.3V AND 0.25V. TIMING MEASUREMENTS ARE MADE AT 1.5V FOR BOTH A LOGIC "1" AND "0."

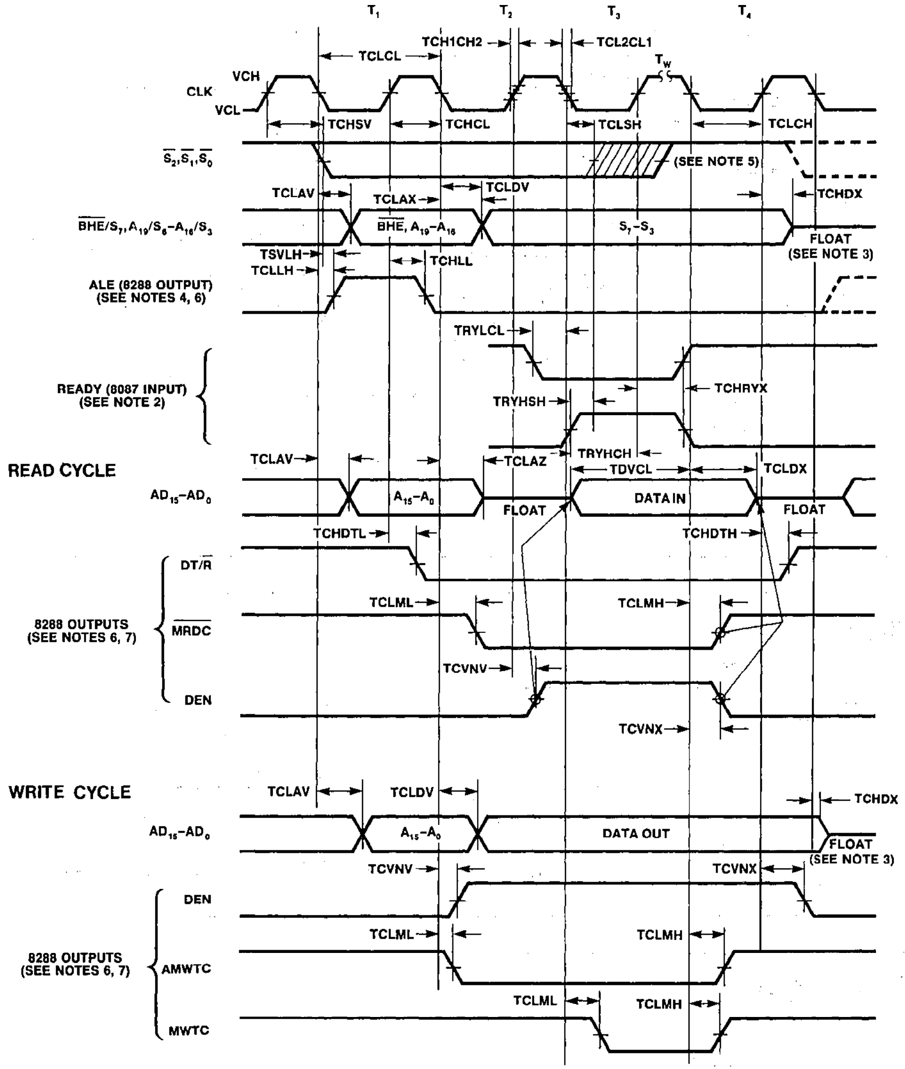
A.C. TESTING LOAD CIRCUIT



C_L INCLUDES JIG CAPACITANCE

WAVEFORMS

MASTER MODE

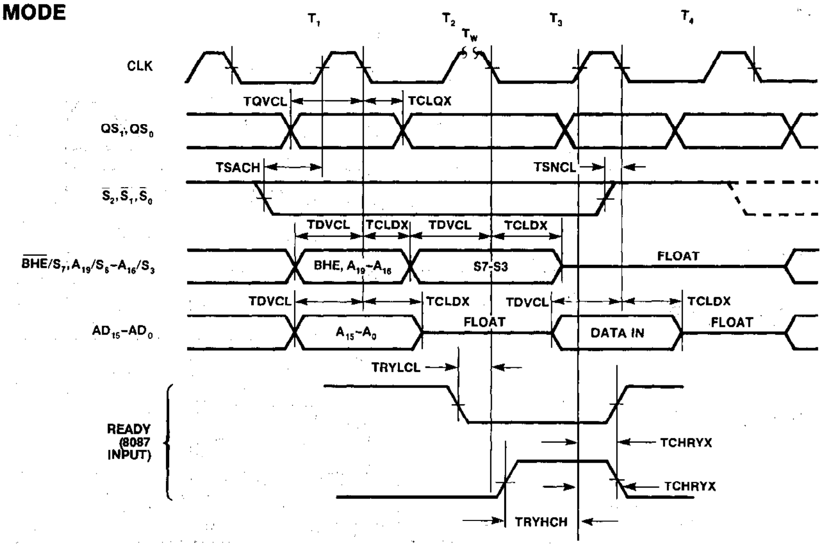


NOTES:

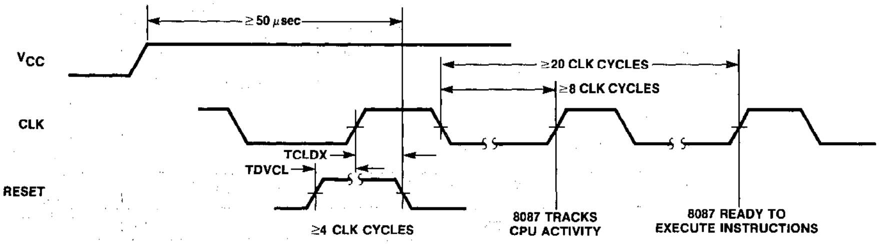
1. ALL SIGNALS SWITCH BETWEEN V_{OL} AND V_{OH} UNLESS OTHERWISE SPECIFIED.
2. READY IS SAMPLED NEAR THE END OF T_2, T_3 AND T_W TO DETERMINE IF T_W MACHINE STATES ARE TO BE INSERTED.
3. THE LOCAL BUS FLOATS ONLY IF THE 8087 IS RETURNING CONTROL TO THE 8086/8088.
4. ALE RISES AT LATER OF (TSVLH, TCLLH).
5. STATUS INACTIVE IN STATE JUST PRIOR TO T_4.
6. SIGNALS AT 8284A OR 8288 ARE SHOWN FOR REFERENCE ONLY.
7. THE ISSUANCE OF 8288 COMMAND AND CONTROL SIGNALS (MRDC, MWTC, AMWC AND DEN) LAGS THE ACTIVE HIGH 8288 CEN.
8. ALL TIMING MEASUREMENTS ARE MADE AT 1.5V UNLESS OTHERWISE NOTED.

WAVEFORMS (Continued)

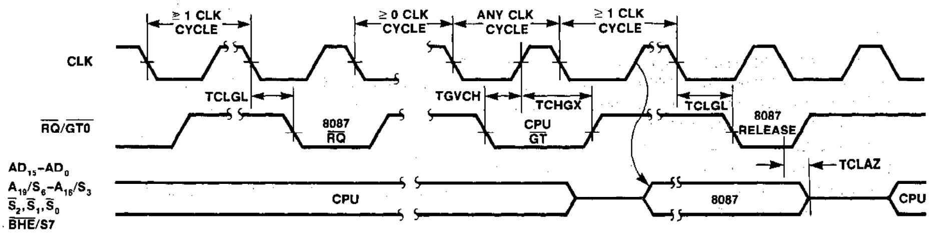
PASSIVE MODE



RESET TIMING



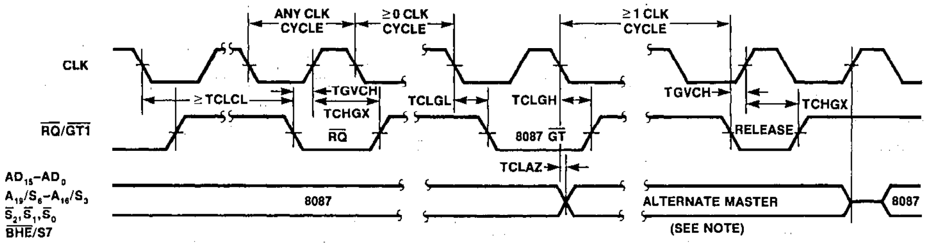
REQUEST/GRANT$_0$ TIMING



NOTE: THE CPU PROVIDES ACTIVE PULLUP OF $\overline{RQ}/\overline{GT0}$, SEE TCLGH SPEC.

WAVEFORMS (Continued)

REQUEST/GRANT$_1$ TIMING



NOTE: ALTERNATE MASTER MAY NOT DRIVE THE BUSES OUTSIDE OF THE REGION SHOWN WITHOUT RISKING BUS CONTENTION.

BUSY AND INTERRUPT TIMING

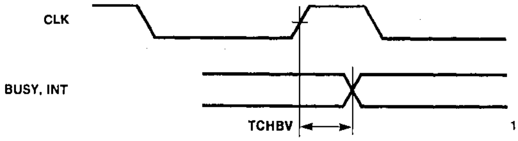


Table 5. 8087 Extensions to the 8086/8088 Instruction Set

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Data Transfer				
FLD = LOAD				
Integer/Real Memory to ST(0)	ESCAPE MF 1	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
Long Integer Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
Temporary Real Memory to ST(0)	ESCAPE 0 1 1	MOD 1 0 1 R/M	(DISP-LO)	(DISP-HI)
BCD Memory to ST(0)	ESCAPE 1 1 1	MOD 1 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 1	1 1 0 0 0 ST(i)		
FST = STORE				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 0 ST(i)		
FSTP = STORE AND POP				
ST(0) to Integer/Real Memory	ESCAPE MF 1	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Long Integer Memory	ESCAPE 1 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to Temporary Real Memory	ESCAPE 0 1 1	MOD 1 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(0) to BCD Memory	ESCAPE 1 1 1	MOD 1 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(0) to ST(i)	ESCAPE 1 0 1	1 1 0 1 1 ST(i)		
FXCH = Exchange ST(i) and ST(0)	ESCAPE 0 0 1	1 1 0 0 1 ST(i)		
Comparison				
FCOM = Compare				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 0 ST(i)		
FCOMP = Compare and Pop				
Integer/Real Memory to ST(0)	ESCAPE MF 0	MOD 0 1 1 R/M	(DISP-LO)	(DISP-HI)
ST(i) to ST(0)	ESCAPE 0 0 0	1 1 0 1 1 ST(i)		
FCOMPP = Compare ST(1) to ST(0) and Pop Twice	ESCAPE 1 1 0	1 1 0 1 1 0 0 1		
FTST = Test ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 0		
FXAM = Examine ST(0)	ESCAPE 0 0 1	1 1 1 0 0 1 0 1		

Table 5. 8087 Extensions to the 8086/8088 Instruction Set (Continued)

Arithmetic	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FADD = Addition				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 0 0 0 R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 0 0 0 ST(i)		
FSUB = Subtraction				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 1 0 R R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 1 0 R R/M		
FMUL = Multiplication				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 0 0 1 R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 0 0 1 R/M		
FDIV = Division				
Integer/Real Memory with ST(0)	ESCAPE MF 0	MOD 1 1 R R/M	(DISP-LO)	(DISP-HI)
ST(i) and ST(0)	ESCAPE d P 0	1 1 1 1 R R/M		
FSQRT = Square Root of ST(0)	ESCAPE 0 0 1	1 1 1 1 1 0 1 0		
FSCALE = Scale ST(0) by ST(1)	ESCAPE 0 0 1	1 1 1 1 1 1 0 1		
FPREM = Partial Remainder of ST(0) ÷ ST(1)	ESCAPE 0 0 1	1 1 1 1 1 0 0 0		
FRNDINT = Round ST(0) to Integer	ESCAPE 0 0 1	1 1 1 1 1 1 0 0		
FXTRACT = Extract Components of ST(0)	ESCAPE 0 0 1	1 1 1 1 0 1 0 0		
FABS = Absolute Value of ST(0)	ESCAPE 0 0 1	1 1 1 0 0 0 0 1		
FCHS = Change Sign of ST(0)	ESCAPE 0 0 1	1 1 1 0 0 0 0 0		
Transcendental				
FPTAN = Partial Tangent of ST(0)	ESCAPE 0 0 1	1 1 1 1 0 0 1 0		
FPATAN = Partial Arctangent of ST(0) ÷ ST(1)	ESCAPE 0 0 1	1 1 1 1 0 0 1 1		
F2XM1 = $2^{ST(0)}-1$	ESCAPE 0 0 1	1 1 1 1 0 0 0 0		
FYL2X = ST(1) · Log_2 [ST(0)]	ESCAPE 0 0 1	1 1 1 1 0 0 0 1		
FYL2XP1 = ST(1) · Log_2 [ST(0) + 1]	ESCAPE 0 0 1	1 1 1 1 1 0 0 1		
Constants				
FLDZ = LOAD + 0.0 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 1 0		
FLD1 = LOAD + 1.0 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 0 0		
FLDPI = LOAD π into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 1 1		
FLDL2T = LOAD $\log_2$ 10 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 0 1		
FLDL2E = LOAD $\log_2$ e into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 0 1 0		
FLDLG2 = LOAD $\log_{10}$ 2 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 0 0		
FLDLN2 = LOAD $\log_e$ 2 into ST(0)	ESCAPE 0 0 1	1 1 1 0 1 1 0 1		

Table 5. 8087 Extensions to the 8086/8088 Instruction Set (Continued)

Processor Control	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
FINIT = Initialize NDP	ESCAPE					0	1	1	1	1	1	0	0	0	1	1		
FENI = Enable Interrupts	ESCAPE					0	1	1	1	1	1	0	0	0	0	0		
FDISI = Disable Interrupts	ESCAPE					0	1	1	1	1	1	0	0	0	0	1		
FLDCW = Load Control Word	ESCAPE					0	0	1	MOD		1	0	1	R/M			(DISP-LO)	(DISP-HI)
FSTCW = Store Control Word	ESCAPE					0	0	1	MOD		1	1	1	R/M			(DISP-LO)	(DISP-HI)
FSTSW = Store Status Word	ESCAPE					1	0	1	MOD		1	1	1	R/M			(DISP-LO)	(DISP-HI)
FCLEX = Clear Exceptions	ESCAPE					0	1	1	1	1	1	0	0	0	1	0		
FSTENV = Store Environment	ESCAPE					0	0	1	MOD		1	1	0	R/M			(DISP-LO)	(DISP-HI)
FLDENV = Load Environment	ESCAPE					0	0	1	MOD		1	0	0	R/M			(DISP-LO)	(DISP-HI)
FSAVE = Save State	ESCAPE					1	0	1	MOD		1	1	0	R/M			(DISP-LO)	(DISP-HI)
FRSTOR = Restore State	ESCAPE					1	0	1	MOD		1	0	0	R/M			(DISP-LO)	(DISP-HI)
FINCSTP = Increment Stack Pointer	ESCAPE					0	0	1	1	1	1	1	0	1	1	1		
FDECSTP = Decrement Stack Pointer	ESCAPE					0	0	1	1	1	1	1	0	1	1	0		
FFREE = Free ST(i)	ESCAPE					1	0	1	1	1	0	0	0	ST(i)				
FNOP = No Operation	ESCAPE					0	0	1	1	1	0	1	0	0	0	0		
FWAIT = CPU Wait for NDP	1	0	0	1	1	0	1	1										

FOOTNOTES:

if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if mod = 11 then r/m is treated as an ST(i) field

if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP

*except if mod = 000 and r/m = 110 then EA = disp-high: disp-low.

MF = Memory Format
00 — 32-bit Real
01 — 32-bit Integer
10 — 64-bit Real
11 — 16-bit Integer

ST(0) = Current stack top
ST(i) = i^{th} register below stack top

d = Destination
0 — Destination is ST(0)
1 — Destination is ST(i)

P = Pop
0 — No pop
1 — Pop ST(0)

R = Reverse: When d = 1 reverse the sense of R.
0 — Destination (op) Source
1 — Source (op) Destination

For **FSQRT:**	$-0 \leq ST(0) \leq +\infty$
For **FSCALE:**	$-2^{15} \leq ST(1) < +2^{15}$ and ST(1) integer
For **F2XM1:**	$0 \leq ST(0) \leq 2^{-1}$
For **FYL2X:**	$0 < ST(0) < \infty$ $-\infty < ST(1) < +\infty$
For **FYL2XP1:**	$0 \leq \lvert ST(0) \rvert < (2 - \sqrt{2})/2$ $-\infty < ST(1) < \infty$
For **FPTAN:**	$0 \leq ST(0) < \pi/4$
For **FPATAN:**	$0 \leq ST(0) < ST(1) < +\infty$

8087 Instructions, Encoding and Decoding

APPENDIX A
MACHINE INSTRUCTION ENCODING AND DECODING

8087 machine instructions assume one of five different forms as shown in table A-1. In all cases, the instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the escape class of instructions. Instructions which reference memory operands are encoded much like similar CPU instructions, since the CPU must calculate the operands effective address from the information contained in the instruction. Section 4.2 discusses this encoding scheme in more detail, and in particular, shows how each memory addressing mode is encoded.

Note that all instructions (except those coded with a "no-wait" mnemonic) are preceded by an assembler-generated CPU WAIT instruction (encoding: 10011011B). Segment override prefixes may also precede 8087 instructions in the instruction stream.

Table A-1. Instruction Encoding

	Lower-addressed Byte								Higher-addressed Byte								0, 1, or 2 bytes
(1)	1	1	0	1	1	OP-A		1	MOD		1	OP-B		R/M			DISPLACEMENT
(2)	1	1	0	1	1	FORMAT		OP-A	MOD		OP-B			R/M			DISPLACEMENT
(3)	1	1	0	1	1	R	P	OP-A	1	1	OP-B			REG			
(4)	1	1	0	1	1	0	0	1	1	1	1	OP					
(5)	1	1	0	1	1	0	1	1	1	1	1	OP					
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	

[1]Memory transfers, including applicable processor control instructions; 0, 1, or 2 displacement bytes may follow.

[2]Memory arithmetic and comparison instructions; 0, 1, or 2 displacement bytes may follow.

[3]Stack arithmetic and comparison instructions.

[4]Constant, transcendental, some arithmetic instructions.

[5]Processor control instructions that do not reference memory.

OP, OP-A, OP-B: Instruction opcode, possibly split into two fields.

MOD: Same as CPU mode field; see table 4-8.

R/M: Sames as CPU register/memory field; see table 4-10.

Table A-1. Instruction Encoding (Cont'd.)

FORMAT: Defines memory operand
00 = short real
01 = short integer
10 = long real
11 = word integer

R: 0 = return result to stack top
1 = return result to other register

P: 0 = do not pop stack
1 = pop stack after operation

REG: register stack element
000 = stack top
001 = next on stack
010 = third stack element, etc.

Table A-2 lists all 8087 machine instructions in binary sequence. This table may be used to "disassemble" instructions in unformatted memory dumps or instructions monitored from the data bus. Users writing exception handlers may also find this information useful to identify the offending instruction.

Table A-2. Machine Instruction Decoding Guide

1st Byte		2nd Byte	Bytes 3, 4	ASM-86 Instruction Format	
Hex	Binary				
D8	1101 1000	MOD00 0R/M	(disp-lo),(disp-hi)	FADD	short-real
D8	1101 1000	MOD00 1R/M	(disp-lo),(disp-hi)	FMUL	short-real
D8	1101 1000	MOD01 0R/M	(disp-lo),(disp-hi)	FCOM	short-real
D8	1101 1000	MOD01 1R/M	(disp-lo),(disp-hi)	FCOMP	short-real
D8	1101 1000	MOD10 0R/M	(disp-lo),(disp-hi)	FSUB	short-real
D8	1101 1000	MOD10 1R/M	(disp-lo),(disp-hi)	FSUBR	short-real
D8	1101 1000	MOD11 0R/M	(disp-lo),(disp-hi)	FDIV	short-real
D8	1101 1000	MOD11 1R/M	(disp-lo),(disp-hi)	FDIVR	short-real
D8	1101 1000	1100 0REG		FADD	ST,ST(i)
D8	1101 1000	1100 1REG		FMUL	ST,ST(i)
D8	1101 1000	1101 0REG		FCOM	ST(i)
D8	1101 1000	1101 1REG		FCOMP	ST(i)
D8	1101 1000	1110 0REG		FSUB	ST,ST(i)
D8	1101 1000	1110 1REG		FSUBR	ST,ST(i)
D8	1101 1000	1111 0REG		FDIV	ST,ST(i)
D8	1101 1000	1111 1REG		FDIVR	ST,ST(i)
D9	1101 1001	MOD00 0R/M	(disp-lo),(disp-hi)	FLD	short-real
D9	1101 1001	MOD00 1R/M		reserved	
D9	1101 1001	MOD01 0R/M	(disp-lo),(disp-hi)	FST	short-real

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte	Bytes 3,4	ASM-86 Instruction Format	
Hex	Binary				
D9	1101 1001	MOD01 1R/M	(disp-lo),(disp-hi)	FSTP	short-real
D9	1101 1001	MOD10 0R/M	(disp-lo),(disp-hi)	FLDENV	14-bytes
D9	1101 1001	MOD10 1R/M	(disp-lo),(disp-hi)	FLDCW	2-bytes
D9	1101 1001	MOD11 0R/M	(disp-lo),(disp-hi)	FSTENV	14-bytes
D9	1101 1001	MOD11 1R/M	(disp-lo),(disp-hi)	FSTCW	2-bytes
D9	1101 1001	1100 0REG		FLD	ST(i)
D9	1101 1001	1100 1REG		FXCH	ST(i)
D9	1101 1001	1101 0000		FNOP	
D9	1101 1001	1101 0001		reserved	
D9	1101 1001	1101 001-		reserved	
D9	1101 1001	1101 01--		reserved	
D9	1101 1001	1101 1REG		*(1)	
D9	1101 1001	1110 0000		FCHS	
D9	1101 1001	1110 0001		FABS	
D9	1101 1001	1110 001-		reserved	
D9	1101 1001	1110 0100		FTST	
D9	1101 1001	1110 0101		FXAM	
D9	1101 1001	1110 011-		reserved	
D9	1101 1001	1110 1000		FLD1	
D9	1101 1001	1110 1001		FLDL2T	
D9	1101 1001	1110 1010		FLDL2E	
D9	1101 1001	1110 1011		FLDPI	
D9	1101 1001	1110 1100		FLDLG2	
D9	1101 1001	1110 1101		FLDLN2	
D9	1101 1001	1110 1110		FLDZ	
D9	1101 1001	1110 1111		reserved	
D9	1101 1001	1111 0000		F2XM1	
D9	1101 1001	1111 0001		FYL2X	
D9	1101 1001	1111 0010		FPTAN	
D9	1101 1001	1111 0011		FPATAN	
D9	1101 1001	1111 0100		FXTRACT	
D9	1101 1001	1111 0101		reserved	
D9	1101 1001	1111 0110		FDECSTP	
D9	1101 1001	1111 0111		FINCSTP	
D9	1101 1001	1111 1000		FPREM	
D9	1101 1001	1111 1001		FYL2XP1	
D9	1101 1001	1111 1010		FSQRT	
D9	1101 1001	1111 1011		reserved	
D9	1101 1001	1111 1100		FRNDINT	
D9	1101 1001	1111 1101		FSCALE	
D9	1101 1001	1111 111-		reserved	
DA	1101 1010	MOD00 0R/M	(disp-lo),(disp-hi)	FIADD	short-integer
DA	1101 1010	MOD00 1R/M	(disp-lo),(disp-hi)	FIMUL	short-integer
DA	1101 1010	MOD01 0R/M	(disp-lo),(disp-hi)	FICOM	short-integer
DA	1101 1010	MOD01 1R/M	(disp-lo),(disp-hi)	FICOMP	short-integer
DA	1101 1010	MOD10 0R/M	(disp-lo),(disp-hi)	FISUB	short-integer
DA	1101 1010	MOD10 1R/M	(disp-lo),(disp-hi)	FISUBR	short-integer

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte	Bytes 3,4	ASM-86 Instruction Format
Hex	Binary			
DA	1101 1010	MOD11 0R/M	(disp-lo),(disp-hi)	FIDIV short-integer
DA	1101 1010	MOD11 1R/M	(disp-lo),(disp-hi)	FIDIVR short-integer
DA	1101 1010	11-- -----		reserved
DB	1101 1011	MOD00 0R/M	(disp-lo),(disp-hi)	FILD short-integer
DB	1101 1011	MOD00 1R/M	(disp-lo),(disp-hi)	reserved
DB	1101 1011	MOD01 0R/M	(disp-lo),(disp-hi)	FIST short-integer
DB	1101 1011	MOD01 1R/M	(disp-lo),(disp-hi)	FISTP short-integer
DB	1101 1011	MOD10 0R/M	(disp-lo),(disp-hi)	reserved
DB	1101 1011	MOD10 1R/M	(disp-lo),(disp-hi)	FLD temp-real
DB	1101 1011	MOD11 0R/M	(disp-lo),(disp-hi)	reserved
DB	1101 1011	MOD11 1R/M	(disp-lo),(disp-hi)	FSTP temp-real
DB	1101 1011	110- ----		reserved
DB	1101 1011	1110 0000		FENI
DB	1101 1011	1110 0001		FDISI
DB	1101 1011	1110 0010		FCLEX
DB	1101 1011	1110 0011		FINIT
DB	1101 1011	1110 01--		reserved
DB	1101 1011	1110 1---		reserved
DB	1101 1011	1111 ----		reserved
DC	1101 1100	MOD00 0R/M	(disp-lo),(disp-hi)	FADD long-real
DC	1101 1100	MOD00 1R/M	(disp-lo),(disp-hi)	FMUL long-real
DC	1101 1100	MOD01 0R/M	(disp-lo),(disp-hi)	FCOM long-real
DC	1101 1100	MOD01 1R/M	(disp-lo),(disp-hi)	FCOMP long-real
DC	1101 1100	MOD10 0R/M	(disp-lo),(disp-hi)	FSUB long-real
DC	1101 1100	MOD10 1R/M	(disp-lo),(disp-hi)	FSUBR long-real
DC	1101 1100	MOD11 0R/M	(disp-lo),(disp-hi)	FDIV long-real
DC	1101 1100	MOD11 1R/M	(disp-lo),(disp-hi)	FDIVR long-real
DC	1101 1100	1100 0REG		FADD ST(i),ST
DC	1101 1100	1100 1REG		FMUL ST(i),ST
DC	1101 1100	1101 0REG		*(2)
DC	1101 1100	1101 1REG		*(3)
DC	1101 1100	1110 0REG		FSUB ST(i),ST
DC	1101 1100	1110 1REG		FSUBR ST(i),ST
DC	1101 1100	1111 0REG		FDIV ST(i),ST
DC	1101 1100	1111 1REG		FDIVR ST(i),ST
DD	1101 1101	MOD00 0R/M	(disp-lo),(disp-hi)	FLD long-real
DD	1101 1101	MOD00 1R/M		reserved
DD	1101 1101	MOD01 0R/M	(disp-lo),(disp-hi)	FST long-real
DD	1101 1101	MOD01 1R/M	(disp-lo),(disp-hi)	FSTP long-real
DD	1101 1101	MOD10 0R/M	(disp-lo),(disp-hi)	FRSTOR 94-bytes
DD	1101 1101	MOD10 1R/M	(disp-lo),(disp-hi)	reserved
DD	1101 1101	MOD11 0R/M	(disp-lo),(disp-hi)	FSAVE 94-bytes
DD	1101 1101	MOD11 1R/M	(disp-lo),(disp-hi)	FSTSW 2-bytes
DD	1101 1101	1100 0REG		FFREE ST(i)
DD	1101 1101	1100 1REG		*(4)
DD	1101 1101	1101 0REG		FST ST(i)
DD	1101 1101	1101 1REG		FSTP ST(i)

Table A-2. Machine Instruction Decoding Guide (Cont'd.)

1st Byte		2nd Byte	Bytes 3,4	ASM-86 Instruction Format
Hex	Binary			
DD	1101 1101	111- ----		reserved
DE	1101 1110	MOD00 0R/M	(disp-lo),(disp-hi)	FIADD word-integer
DE	1101 1110	MOD00 1R/M	(disp-lo),(disp-hi)	FIMUL word-integer
DE	1101 1110	MOD01 0R/M	(disp-lo),(disp-hi)	FICOM word-integer
DE	1101 1110	MOD01 1R/M	(disp-lo),(disp-hi)	FICOMP word-integer
DE	1101 1110	MOD10 0R/M	(disp-lo),(disp-hi)	FISUB word-integer
DE	1101 1110	MOD10 1R/M	(disp-lo),(disp-hi)	FISUBR word-integer
DE	1101 1110	MOD11 0R/M	(disp-lo),(disp-hi)	FIDIV word-integer
DE	1101 1110	MOD11 1R/M	(disp-lo),(disp-hi)	FIDIVR word-integer
DE	1101 1110	1100 0REG		FADDP ST(i),ST
DE	1101 1110	1100 1REG		FMULP ST(i),ST
DE	1101 1110	1101 0---		*(5)
DE	1101 1110	1101 1000		reserved
DE	1101 1110	1101 1001		FCOMPP
DE	1101 1110	1101 101-		reserved
DE	1101 1110	1101 11--		reserved
DE	1101 1110	1110 0REG		FSUBP ST(i),ST
DE	1101 1110	1110 1REG		FSUBRP ST(i),ST
DE	1101 1110	1111 0REG		FDIVP ST(i),ST
DE	1101 1110	1111 1REG		FDIVRP ST(i),ST
DF	1101 1111	MOD00 0R/M	(disp-lo),(disp-hi)	FILD word-integer
DF	1101 1111	MOD00 1R/M	(disp-lo),(disp-hi)	reserved
DF	1101 1111	MOD01 0R/M	(disp-lo),(disp-hi)	FIST word-integer
DF	1101 1111	MOD01 1R/M	(disp-lo),(disp-hi)	FISTP word-integer
DF	1101 1111	MOD10 0R/M	(disp-lo),(disp-hi)	FBLD packed-decimal
DF	1101 1111	MOD10 1R/M	(disp-lo),(disp-hi)	FILD long-integer
DF	1101 1111	MOD11 0R/M	(disp-lo),(disp-hi)	FBSTP packed-decimal
DF	1101 1111	MOD11 1R/M	(disp-lo),(disp-hi)	FISTP long-integer
DF	1101 1111	1100 0REG		*(6)
DF	1101 1111	1100 1REG		*(7)
DF	1101 1111	1101 0REG		*(8)
DF	1101 1111	1101 1REG		*(9)
DF	1101 1111	111- ----		reserved

* The marked encodings are *not* generated by the language translators. If, however, the 8087 encounters one one these encodings in the instruction stream, it will execute it as follows:

(1) FSTP ST(i)

(2) FCOM ST(i)

(3) FCOMP ST(i)

(4) FXCH ST(i)

(5) FCOMP ST(i)

(6) FFREE ST(i) and pop stack

(7) FXCH ST(i)

(8) FSTP ST(i)

(9) FSTP ST(i)