XA User Guide

PHILIPS NXP XA	**XA-G3, S3, G49**

Ceibo Emulator Supporting XA:	**DS-XA** **http://ceibo.com/eng/products/dsxa.shtml**

Ceibo Low Cost Emulators Supporting XA:	**EB-XAG3** **http://ceibo.com/eng/products/ebxag3.shtml** **EB-XAG49** **http://ceibo.com/eng/products/ebxag49.shtml** **EB-XAS3** **http://ceibo.com/eng/products/ebxas3.shtml**

Ceibo Programmer Supporting XA:	**MP-51** **http://ceibo.com/eng/products/mp51.shtml**

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are de scribed herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Printed in U.S.A.

1 The XA Family - High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers

1.1 Introduction

The role of the microcontroller is becoming increasingly important in the world of electronics as systems which in the past relied on mechanical or simple analog electrical control systems have microcontrollers embedded in them that dramatically improve functionality and reliability, while reducing size and cost. Microcontrollers also provide the general purpose solutions needed so that common software and hardware can be shared among multiple designs to reduce overall design-in time and costs.

The requirements of systems using microcontrollers are also much more demanding now than a few years ago. Whether called by the name “microcontrollers”, “embedded controllers” or “single-chip microcomputers”, the systems that use these devices require a much higher level of performance and on-chip integration.

As microcontrollers begin to enter into more complex control environments, the demand for increased throughput, increased addressing capability, and higher level of on-chip integration has led to the development of 16-bit microcontrollers that are capable of processing much more information than 8-bit microcontrollers. However, simply integrating more bits or more peripheral functions does not solve the demand of the control systems being developed today. New microcontrollers must provide high-level-language support, powerful debugging environments, and advanced methods of real time control in order to meet the more stringent functionality and cost requirements of these systems.

To meet the above goals The XA or “eXtended Architecture” family of general-purpose microcontrollers from Philips is being introduced to provide the highest performance/cost ratio for a variety of high performance embedded-systems-control applications including real-time, multi-tasking environments. The XA family members add to the CPU core a specific complement of on-chip memory, I/Os, and peripherals aimed at meeting the requirements of different application areas. The core-based architecture allows easy expansion of the family according to a wide variety of customer requirements. The powerful instruction set supports faster computing power, faster data transfer, multi-tasking, improved response to external events and efficient high-level language programming.

Upward (assembly-level) code compatibility with the Philips 80C51 family of controllers provides a smooth design transition for system upgrades by providing tremendously enhanced performance.

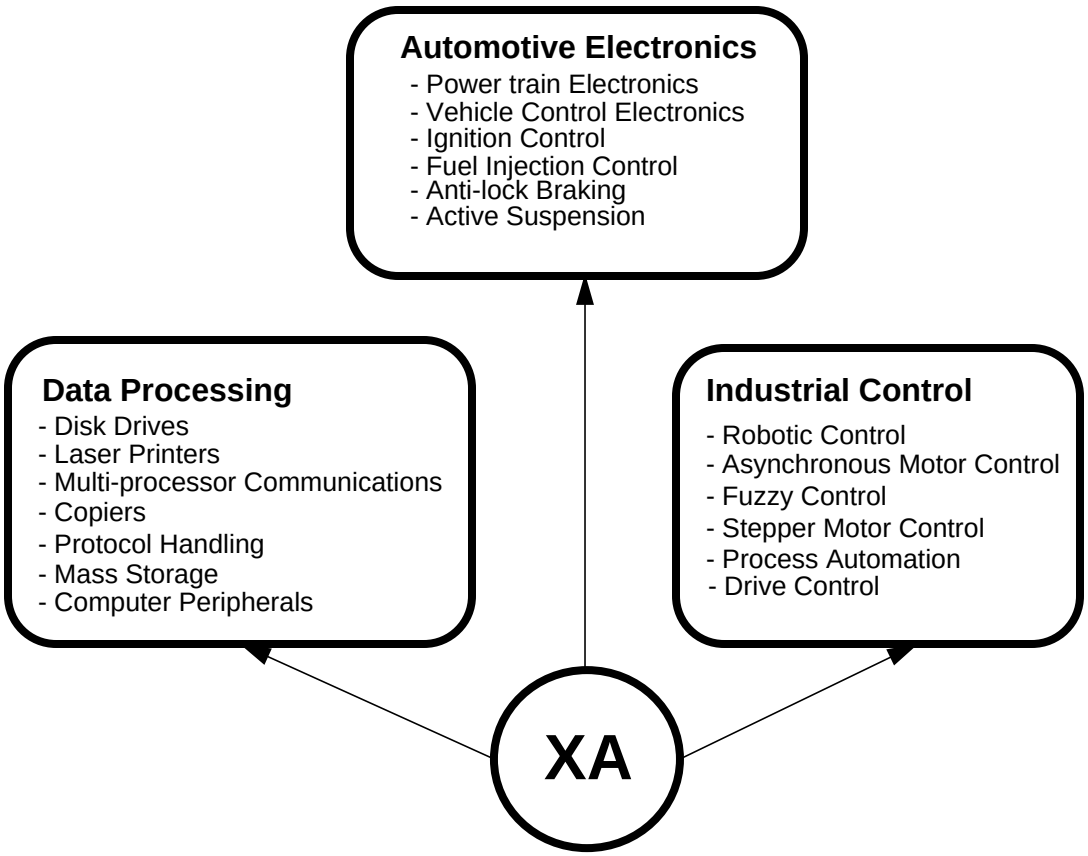


Figure 1. Applications of Philips XA microcontrollers

1.2 Architectural Features of XA

- Upward compatibility with the standard 8XC51 core (assembly source level)
- 24-bit address range (16 Megabytes code and data space)
- 16-bit static CPU
- Enhanced architecture using both 16-bit words and 8-bit bytes
- Enhanced instruction set
- High code efficiency; most of the instructions are 2-4 bytes in length
- Fast 16X16 Multiply and 32x16 Divide Instructions
- 16-bit Stack Pointers and general pointer registers
- Capability to support 32 vectored interrupts - 31 maskable and 1 NMI
- Supports 16 hardware and 16 software traps
- Power Down and Idle power reduction modes
- Hardware support for multi-tasking software

2 Architectural Overview

2.1 Introduction

The Philips XA (eXtended Architecture) has a general purpose register-register architecture to provide the best cost-to-performance trade-off available for a high speed microcontroller using today's technology. Intended as both an upward compatibility path for 80C51 users who need greater performance or more memory, and as a powerful, general-purpose 16-bit controller, the XA also incorporates support for multi-tasking operating systems and high-level languages such as C, while retaining the comprehensive bit-oriented operations that are the hallmark of the 80C51.

This overview introduces the concepts and terminology of the XA architecture in preparation for the detailed descriptions in the following sections of this manual.

2.2 Memory Organization

The XA architecture has several distinct memory spaces. The architecture and the instruction encoding are optimized for register based operations; in addition, arithmetic and logical operations may be done directly on data memory as well. Thus, the XA architecture avoids the bottleneck of having a single accumulator register.

2.2.1 Register File

The register file (Figure 2.1) allows access to 8 words of data at any one time; the eight words are also addressable as 16 bytes. The bottom 4 word registers are "banked". That is, there are four groups of registers, any one of which may occupy the bottom 4 words of the register file at any one time. This feature may be used to minimize the time required for context switching during interrupt service, and to provide more register space for complicated algorithms.

For some instructions –32-bit shifts, multiplies, and divides– adjacent pairs of word registers are referenced as double words.

The upper four words of the register file are not banked. The topmost word register is the stack pointer, while any other word register may be used as a general purpose pointer to data memory.

The entire register file is bit addressable. That is, any bit in the register file (except the 3 unselected banks of the bottom 4 words) may be operated on by bit manipulation instructions.

The XA instruction encoding allows for future expansion of the register file by the addition of 8 word registers. If implemented, these additional registers will be word data registers only and cannot be used as pointers or addressed as bytes.

The overall XA register file structure provides a superset of the 80C51 register structure. For details, refer to the section on 80C51 compatibility.

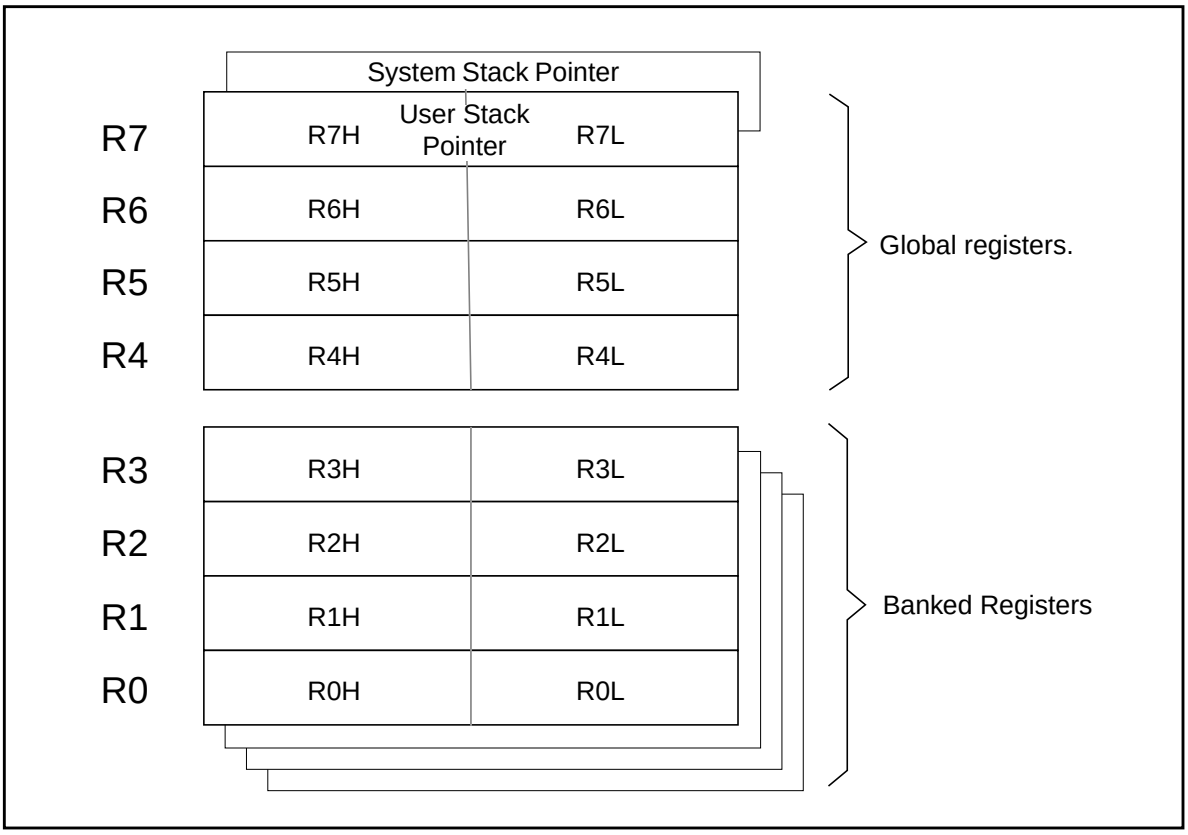


Figure 2.1 XA register file diagram

2.2.2 Data Memory

The XA architecture supports a 16 megabyte data memory space with a full 24-bit address. Some derivative parts may implement fewer address lines for a smaller range. The data space beginning at address 0 is normally on-chip and extends to the limit of the RAM size of a particular XA derivative. For addresses above that on a derivative, the XA will automatically roll over to external data memory.

Data memory in the XA is divided into 64K byte segments (Figure 2.2) to provide an intrinsic protection mechanism for multi-tasking systems and to improve performance. Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require large data memories (Figure 2.3).

The XA provides 2 segment registers used to access data memory, the Data Segment register (DS) and the Extra Segment register (ES). Each pointer register is associated with one of the segment registers via the Segment Select (SSEL) register. Pointer registers retain this association until it is changed under program control.

The XA provides flexible data addressing modes. Most arithmetic, logic, and data movement instructions support the following modes of addressing data memory:

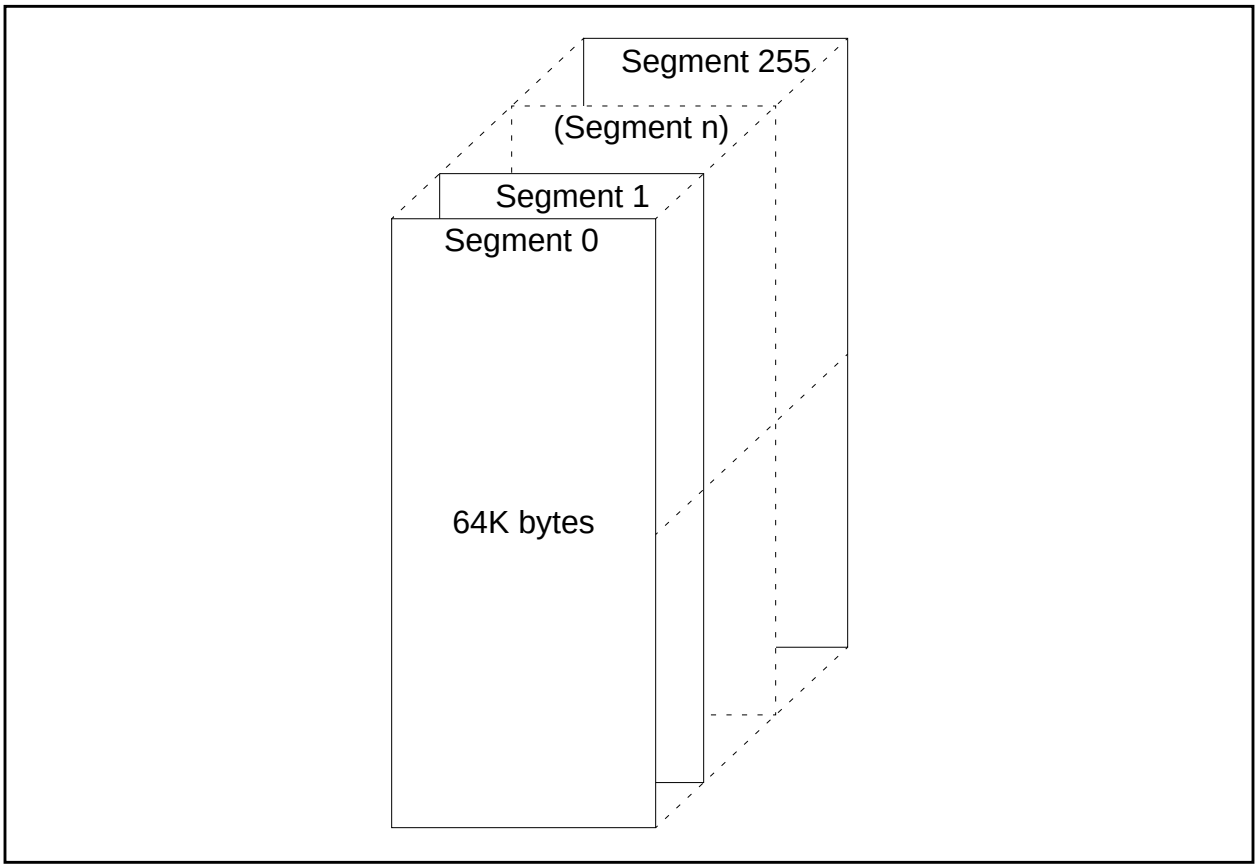


Figure 2.2 XA data memory segments

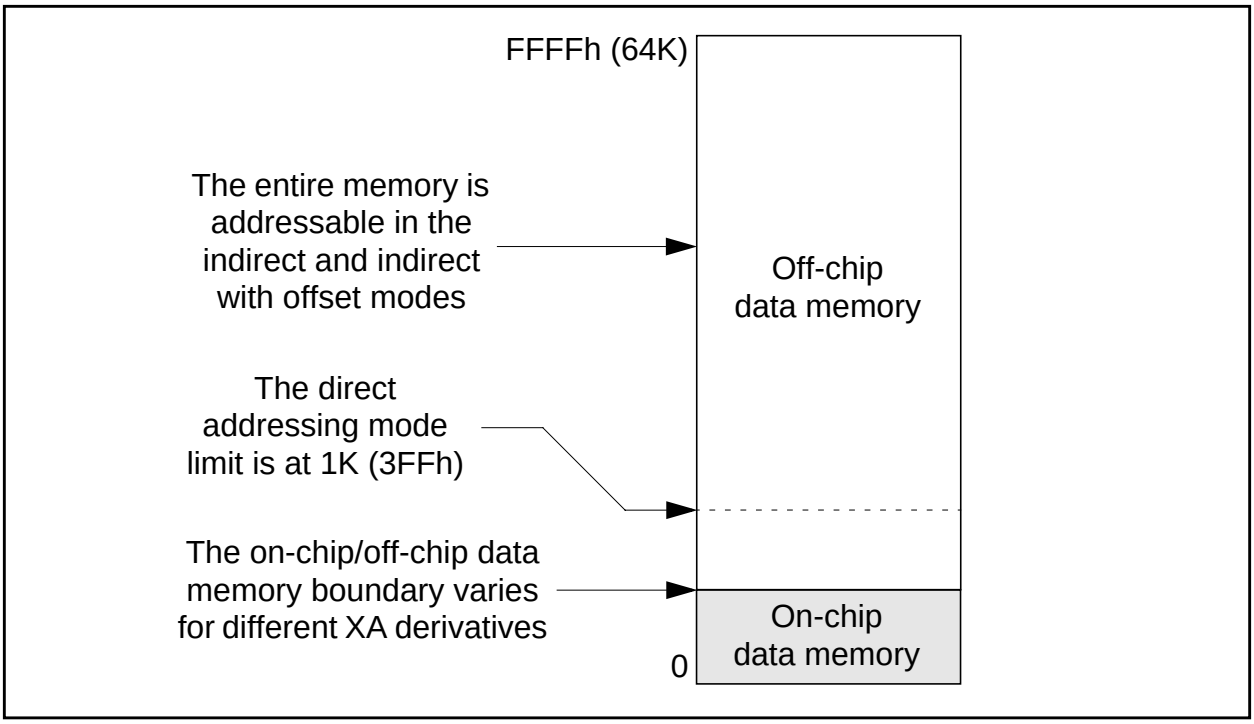


Figure 2.3 Simplified XA data memory diagram

Direct. The first 1K bytes of data on each segment may be accessed by an address contained within the instruction.

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with 16-bits from a pointer register.

Indirect with offset. An 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode allows access into a data structure when a pointer register contains the starting address of the structure. It also allows subroutines to access parameters passed on the stack.

Indirect with auto-increment. The address is formed in the same manner as plain indirect, but the pointer register contents are automatically incremented following the operation.

Data movement instructions and some special purpose instructions also have additional data addressing modes.

The XA data memory addressing scheme provides for upward compatibility with the 80C51. For details, refer to Chapter 9.

2.2.3 Code Memory

The XA is a Harvard architecture device, meaning that the code and data spaces are separate. The XA provides a continuous, unsegmented linear code space that may be as large as 16 megabytes (Figure 2.4). In XA derivatives with on-chip ROM or EPROM code memory, the on-

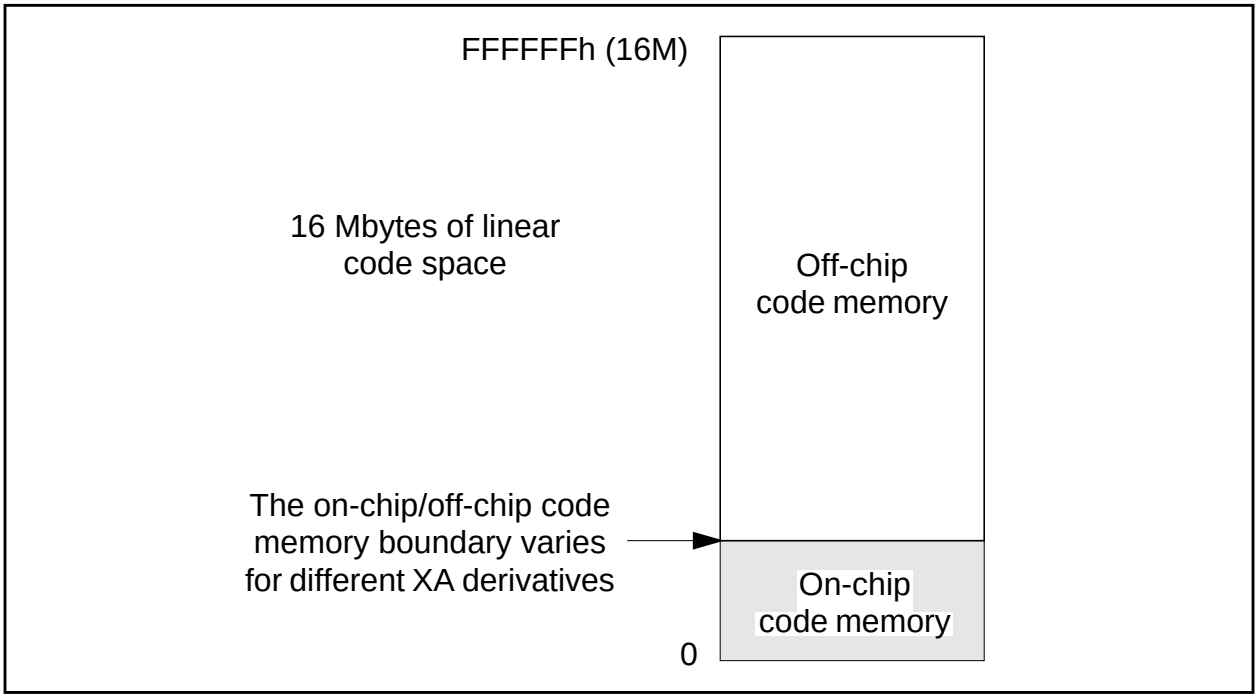


Figure 2.4 XA code memory map

chip space always begins at code address 0 and extends to the limit of the on-chip code memory. Above that, code will be fetched from off-chip. Most XA derivatives will support an external bus for off-chip data and code memory, and may also be used in a ROM-less mode, with no code memory used on-chip.

In some cases, code memory may be addressed as data. Special instructions provide access to the entire code space via pointers. Either a special segment register (CS or Code Segment) or the upper 8-bits of the Program Counter (PC) may be used to identify the portion of code memory referenced by the pointer.

2.2.4 Special Function Registers

Special Function Registers (SFRs) provide a means for the XA to access Core registers, internal control registers, peripheral devices, and I/O ports. Any SFR may be accessed by a program at any time without regard to any pointer or segment. An SFR address is always contained entirely within an instruction. See Figure 2.5.

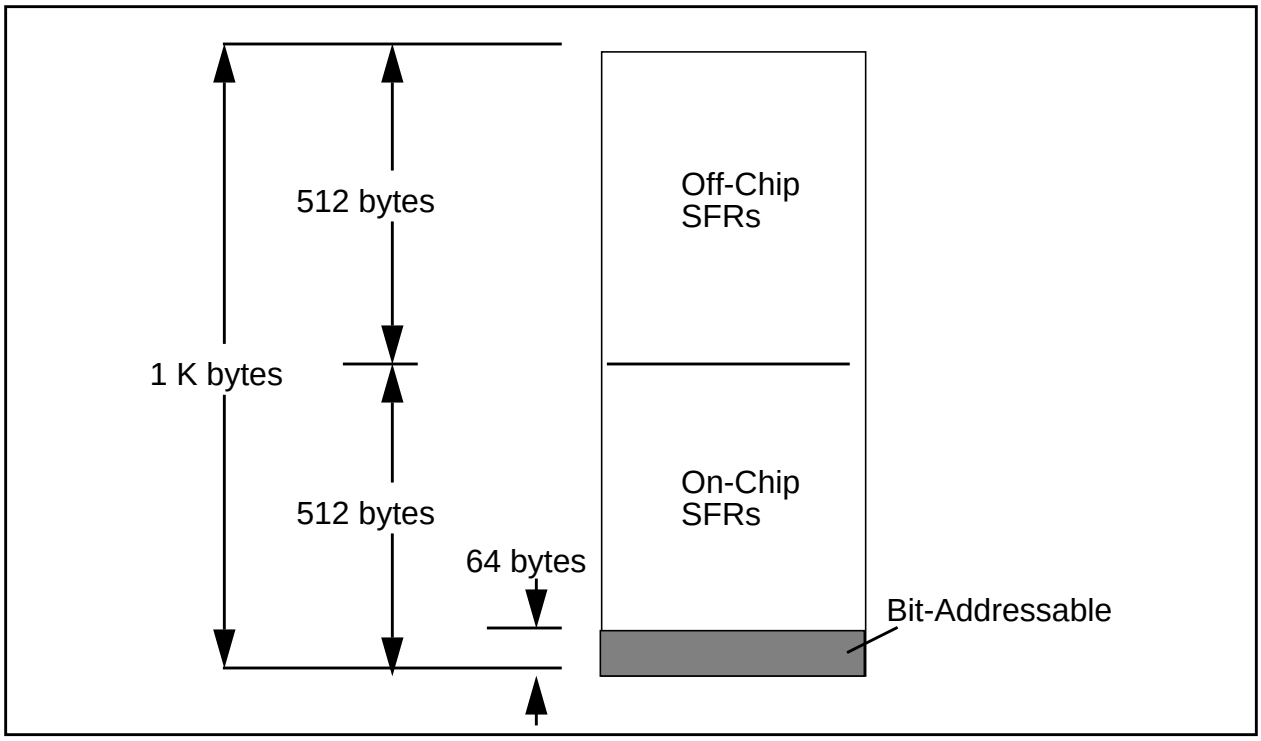


Figure 2.5 SFR Address Space

The total SFR space is 1K bytes in size. This is further divided into two 512 byte regions. The lower half is assigned to on-chip SFRs, while the second half is reserved for off-chip SFRs. This allows provides a means to add off-chip I/O devices mapped into the XA as SFRs. Off-chip SFR access is not implemented on all XA derivatives.

On-chip SFRs are implemented as needed to provide control for peripherals or access to CPU features and functions. Each XA derivative may have a different number of SFRs implemented

because each has a different set of peripheral functions. Many SFR addresses will be unused on any particular XA derivative.

The first 64 bytes of on-chip SFR space are bit-addressable. Any CPU or peripheral register that allows bit access will be allocated an address within that range.

2.3 CPU

Figure 2.6 shows the XA architecture as a whole. Each of the blocks shown are described in this section.

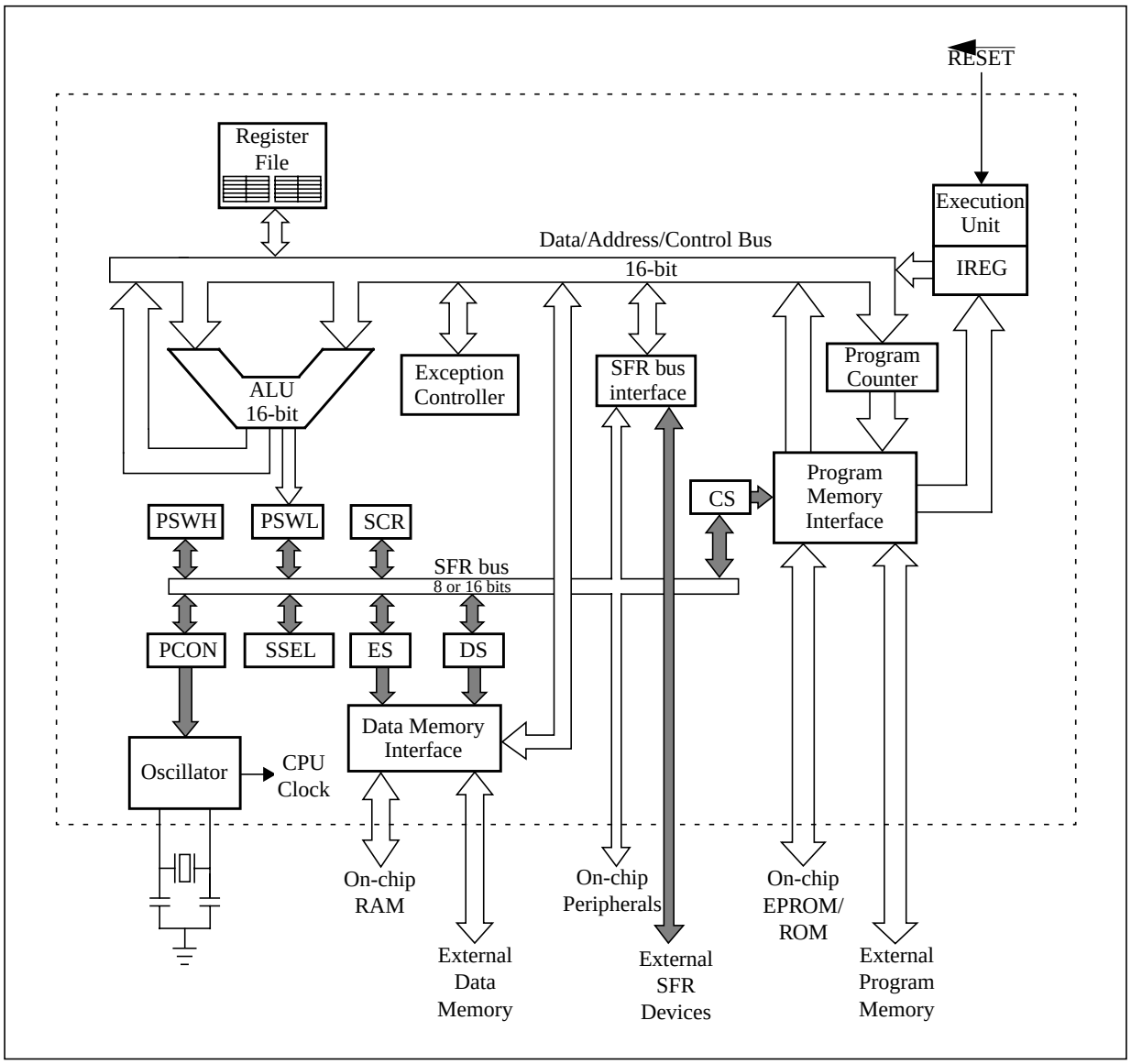


Figure 2.6 The XA Architecture

2.3.1 CPU Blocks

The XA processor is composed of several functional blocks: Instruction fetch and decode; Execution unit; ALU; Exception controller; Interrupt controller; Register File and core registers; Program memory (ROM or EPROM), Data memory (RAM); SFR and external bus interface; Oscillator; and on-chip peripherals and I/O ports.

Certain functional blocks that exist on most XA derivatives are not part of the CPU core and may vary in each derivative. These are: the external bus interface, the Special Function Register bus (SFR bus) interface, specific peripherals, I/O ports, code and data memories, and the interrupt controller.

CPU Performance Features

The XA core is partially pipelined and performs some CPU functions in parallel. For instance, instruction fetch and decode, and in some cases data write-back, are done in parallel with instruction execution. This partial pipelining gives very fast instruction execution at a very low cost. For instance, the instruction execution time for most register-to-register operations on the XA is 3 CPU clocks, or 100 nanoseconds with a 30 MHz oscillator.

ALU

Data operations in the XA core are accomplished with a 16-bit ALU, providing both 8-bit and 16-bit functions. Special circuitry has been included to allow some 32-bit functions, such as shifts, multiply, and divide.

Core Registers

The XA core includes several key Special Function Registers which are accessed by programs.

The System Configuration Register (SCR) sets up the basic operating modes of the XA. The Program Status Word (PSW) contains status flags that show the result of ALU operations, the register select bits for the four register file banks, the interrupt mask bit, and other system flags. The Data Segment (DS), Extra Segment (ES), and Code Segment (CS) registers contain the segment numbers of active data memory segments. The Segment Select register (SSEL), contains bits that determine which segment register is used by each pointer register in the register file. Bits in the Power Control register (PCON) control the reduced power modes of the processor.

Execution and Control

The Execution and Control block fetches instructions from the code memory and decodes the instructions prior to execution. The XA normally attempts to fetch instructions from the code memory ahead of what is immediately needed by the execution unit. These pre-fetched instructions are stored in a 7 byte queue contained in the fetch and decode unit.

If the fetch unit has instructions in the queue, the execution unit will not have to wait for a fetch to occur when it is ready to begin execution of a new instruction. If a program branch is taken, the queue is flushed and instructions are fetched from the new location. This block also decides whether to attempt instruction fetches from on or off-chip code memory.

The instruction at the head of the queue is decoded into separate functional fields that tell the other CPU blocks what to do when the instruction is executed. These fields are stored in staging registers that hold the information until the next instruction begins executing.

Execution Unit

The execution unit controls many of the other CPU blocks during instruction execution. It routes addressing information, sends read and write commands to the register file and memory control blocks, tells the fetch and decode unit when to branch, controls the stack, and ensures that all of these operations are performed in the proper sequence. The execution unit obtains control information for each instruction from a microcode ROM.

Interrupt Controller

The interrupt controller can receive an interrupt request from any of the sources on a particular XA derivative. It prioritizes these based on user programmable registers containing a priority for each interrupt source. It then compares the priority of the highest pending interrupt (if any) to the interrupt mask bits from the PSW. If the interrupt has a higher priority than the currently running code, the interrupt controller issues a request to the execution unit.

The interrupt controller also contains extra registers for processing software interrupts. These are used to run non-critical portions of interrupt service routines at a decreased priority without risking “priority inversion.”

While the interrupt controller is not part of the XA core, it is present in some form on all XA derivatives.

Exception Controller

The exception controller is similar to the interrupt controller except that it processes CPU exceptions rather than hardware and software interrupt requests. Sources of exceptions are: stack overflow; divide by zero; user execution of an RETI instruction; hardware breakpoint; trace mode; and non-maskable interrupt (NMI).

Exceptions are serviced according to a fixed priority ranking. Generally, exceptions must be serviced immediately since each represents some important event or problem that must be dealt with before normal operation can resume.

The Exception Controller is part of the XA core and is always present.

Interrupt and Exception Processing

Interrupt and exception processing both make use of a vector table that resides in the low addresses of the code memory. Each interrupt and exception has an entry in the vector table that includes the starting address of the service routine and a new PSW value to be used at the beginning of the service routine. The starting address of a service routine must be within the first 64K of code memory.

When the XA services an exception or interrupt, it first saves the return address on the stack, followed by the PSW contents. Next, the PC and the PSW are loaded with the starting address of the appropriate service routine and the new PSW contents, respectively, from the vector table.

When the service routine completes, it returns to the interrupted code by executing the RETI (return from interrupt) instruction. This instruction loads first the PSW and then the Program Counter from the stack, resuming operation at the point of interruption. If more than the PC and PSW are used by the service routine, it is up to that routine to save and restore those registers or other portions of the machine state, normally by using the stack, and often by switching register banks.

Reset

Power up reset and any other external reset of the XA is accomplished via an active low reset pin. A simple resistor and capacitor reset circuit is typically used to provide the power-on reset pulse. the reset pin is a Schmitt trigger input, in order to prevent noise on the reset pin from causing spurious or incomplete resets.

The XA may be reset under program control by executing the RESET instruction. This instruction has the effect of resetting the processor as if an external reset occurred, except that some hardware features that are latched following a hardware reset (such as the state of the EA pin and bus width programming) are not re-latched by a software reset. This distinction is necessary because external circuitry driving those inputs cannot determine that a reset is in progress.

Some XA derivatives also have a hardware watchdog timer peripheral that will trigger an equivalent chip reset if it is allowed to time out.

Oscillator and Power Saving Modes

XA derivatives have an on-chip oscillator that may be used with crystals or ceramic resonators to provide a clock source for the processor.

The XA supports two power saving modes of operation: Idle mode and Power Down mode. Either mode is activated by setting a bit in the Power Control (PCON) register. The Idle mode shuts down all processor functions, but leaves most of the on-chip peripherals and the external interrupts functioning. The oscillator continues to run. An interrupt from any operating source will cause the XA to resume operation where it left off.

The Power Down mode goes one step further and shuts down everything, including the on-chip oscillator. This reduces power consumption to a tiny amount of CMOS leakage plus whatever loads are placed on chip pins. Resuming operation from the power down mode requires the oscillator to be restarted, which takes about 10 milliseconds. Power down mode can be terminated either by resetting the XA or by asserting one of the external interrupts, if one was left enabled when power down mode was entered. In Power Down mode, data in on-board RAM is retained. Further power savings may be made by reducing Vdd in Power Down mode; see the device data sheet for details.

Stack

The processor stack provides a means to store interrupt and subroutine return addresses, as well as temporary data. The XA includes 2 stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP), which correspond to 2 different stacks: the system stack and the user stack. See Figure 2.7. The system stack always resides in the first data memory segment,

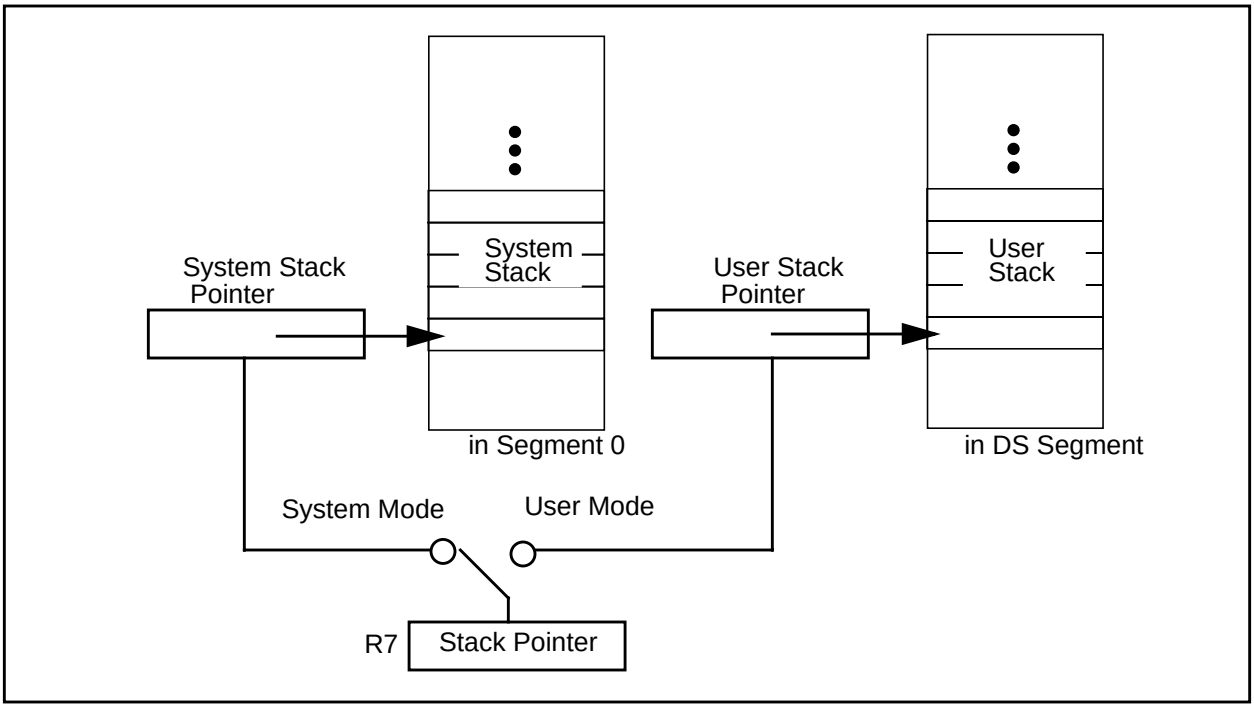


Figure 2.7 XA Stacks

segment 0. The user stack resides in the data memory segment identified by the current value of the data segment (DS) register. Executing code has access to only one of these stacks at a time, via the Stack Pointer, R7. Since each stack resides in a single data memory segment, its maximum size is 64K bytes. The purpose of having two stack pointers will be discussed in more detail in the section on Task Management below.

The XA stack grows downwards, from higher addresses to lower addresses within data memory. The current stack pointer always points to the last item pushed on the stack, unless the stack is empty. Prior to a push operation, the stack pointer is decremented by 2, then data is written to memory. When the stack is popped, the reverse procedure is used. First, data is read from memory, then the stack pointer is incremented by 2. Data on the stack always occupies an even number of bytes and is word aligned in data memory.

Debugging Features

The XA incorporates some special features designed to aid in program and system debugging. There is a software breakpoint instruction that may be inserted in a user's program by a debugger program, causing the user program to break at that point and go to the breakpoint service routine, which can transmit the CPU state so that it can be viewed by the user.

The trace mode is similar to a breakpoint, but is forced by hardware in the XA after the execution of every instruction. The trace service routine can then keep track of every instruction executed by a user program and transmit information about the CPU state to a serial port or other peripheral for display or storage. Trace mode is controlled by a bit in the PSW. The XA is able to alter the trace mode bit whenever an interrupt or exception vector is taken. This gives very flexible use of trace mode, for instance by allowing all interrupts to run at full speed to comply with system hardware requirements, while single stepping through mainline code.

With these two features, a simple monitor debugger routine can allow a user to single step through a program, or to run a program at full speed, stopping only when execution reaches a breakpoint, in either case viewing the CPU state before continuing.

2.4 Task Management

Several features of the XA have been included to facilitate multi-tasking. Multi-tasking can be thought of as running several programs at once on the same processor, with a supervisory program determining when each program, or task, runs, and for how long. Since each task shares the same CPU, the system resources required by each must be kept separate and the CPU state restored when switching execution from one task to another. The problem is much simpler for a microcontroller than it is for a microprocessor, because the code executed by a microcontroller always comes from the same source: the designers of the system it runs on. Thus, this code can be considered to be basically trustworthy and extreme measures to prevent misbehavior are not necessary. The emphasis in the XA design is to protect against simple accidents.

The first step in supporting multi-tasking is to provide two execution contexts, one for the basic tasks –on the XA termed “user mode”– and one for the supervisory program –"system mode.". A program running in system mode has access to all of the processor’s resources and can set up and launch tasks.

Code running in system and user mode use different stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP) respectively. The system stack is always located in the first 64K data memory segment, where it can take advantage of the fast on-chip RAM. The user stack is located within each task’s local data segment, identified by the DS register. The fact that user mode code uses a different stack than system mode code prevents tasks from accidentally destroying data on the system stack and in other task spaces.

Additional protection mechanisms are provided in the form of control bits and registers that are only writable by system mode code. For instance the DS register, that identifies the local data segment for user mode code, is only writable in the system mode. While tasks can still write to the other segment register, the ES register, they cannot write to memory via the ES register unless specifically allowed to do so by the system. The data memory segmentation scheme thus prevents tasks from accessing data memory in unpredictable ways.

Other protected features include enabling of the Trace Mode and alteration of the Interrupt Mask.

The 4 register banks are a feature that can be useful in small multi-tasking systems by using each bank for a different task, including one for system code. This means less CPU state that must be saved during task switching.

2.5 Instruction Set

The XA instruction set is designed to support common control applications. The instruction encoding is optimized for the most commonly used instructions: register to register or register with indirect arithmetic and logic operations; and short conditional and unconditional branches. These instructions are all encoded as 2 bytes. The bulk of XA instructions are encoded as either 2 or 3 bytes, although there are a few 1 byte instructions as well as 4, 5, and 6 byte instructions.

The execution of instructions normally overlaps instruction fetch, and sometimes write-back operations, in order to further speed processing.

2.5.1 Instruction Syntax

The instruction syntax chosen for the XA is similar in many ways to that of the 80C51. A typical XA instruction has a basic mnemonic, such as "ADD", followed by the operands that the operation is to be performed on. The basic syntax is illustrated in Figure 2.8. The direction of operation flow is determined by the order in which operands occur in the source line. For instance, the instruction: "ADD R1, R2" would cause the contents of R1 and R2 to be added together and the result stored in R1. Since R1 and R2 are word registers in the XA, this is a 16-bit operation.

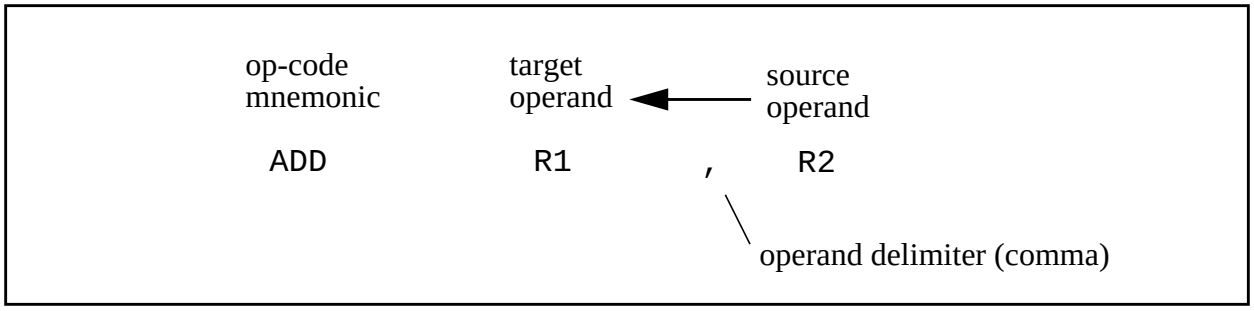


Figure 2.8 Basic Instruction Syntax

An indirect reference (a reference to data memory using the contents of a register as an address) is specified by enclosing the operand in square brackets, as in: "ADD R1, [R2]". See Figure 2.9. This instruction causes the contents of R1 and the data memory location pointed to by R2 (appended to its associated segment register) to be added together and the result stored in R1. Reversing the operand order ("ADD [R2], R1") causes the result to be stored in data memory, as shown in Figure 2.10.

Most instructions support an additional feature called auto-increment that causes the register used to supply the indirect memory address to be automatically incremented after the memory access takes place. The source line for such an operation is written as follows: "ADD R1, [R2+]". As illustrated in Figure 2.11, the auto-increment amount always matches the data size used in the instruction. In the previous example, R2 will have 2 added to it because this was a word operation.

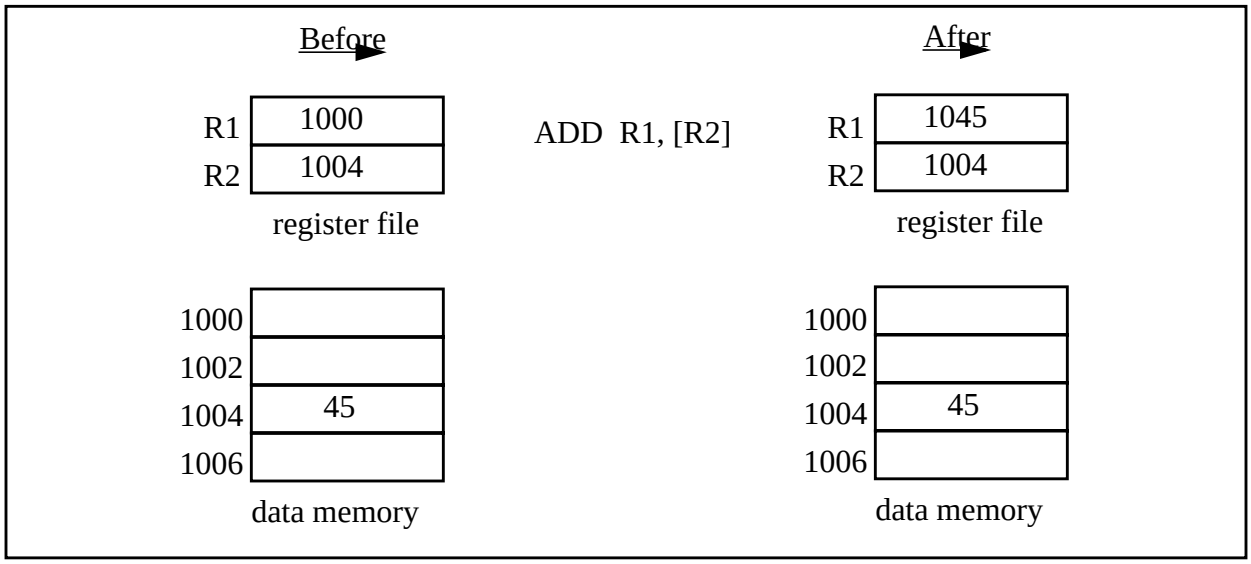


Figure 2.9 Basic Indirect Addressing Syntax, to register

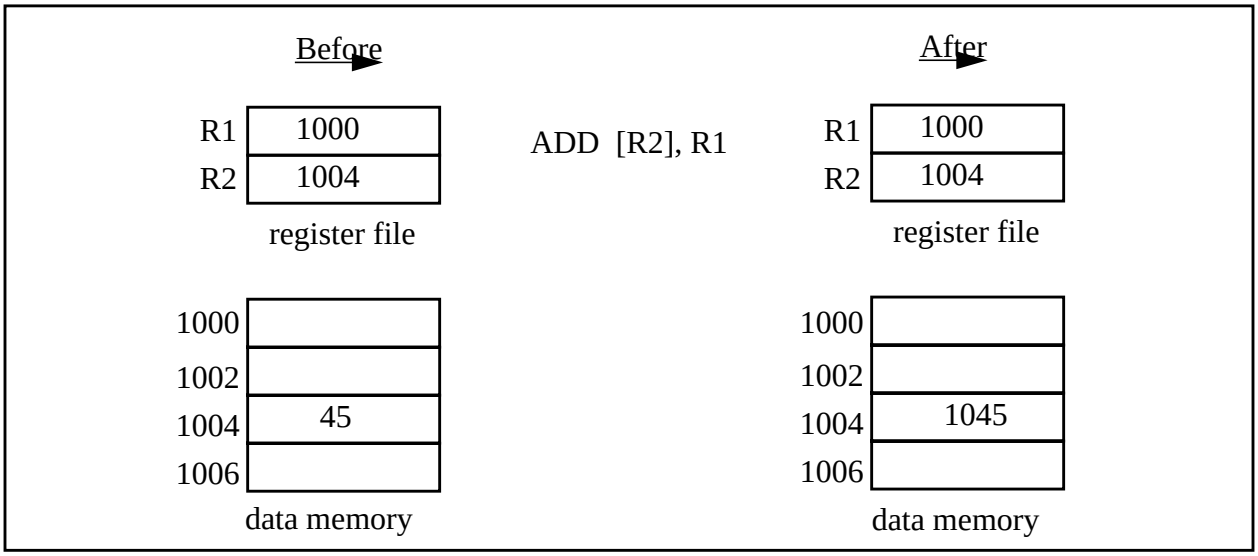


Figure 2.10 Basic Indirect Addressing Syntax, from Register

Another version of indirect addressing is called indirect with offset mode. In this version, an immediate value from the instruction word is added to the contents of the indirect register in order to form the actual address. This result of the add is 16 bits in size, which is then appended to the segment register for that pointer register. If the offset calculation overflows 16 bits, the overflow is ignored, so the indirect reference always remains on the same segment. The immediate data from the instruction is a signed 8-bit or 16-bit offset. Thus, the range is +127 bytes to -128 bytes for an 8-bit offset, and +32,767 to -32,768 bytes for a 16-bit offset. Note that since the address calculation is limited to 16-bits, the 16-bit offset mode allows access to an entire data segment.

When an instruction requires an immediate data value (a value stored within the instruction itself), it is written using the "#" symbol. For example: "ADD R1, #12" says to add the value 12 to register R1.

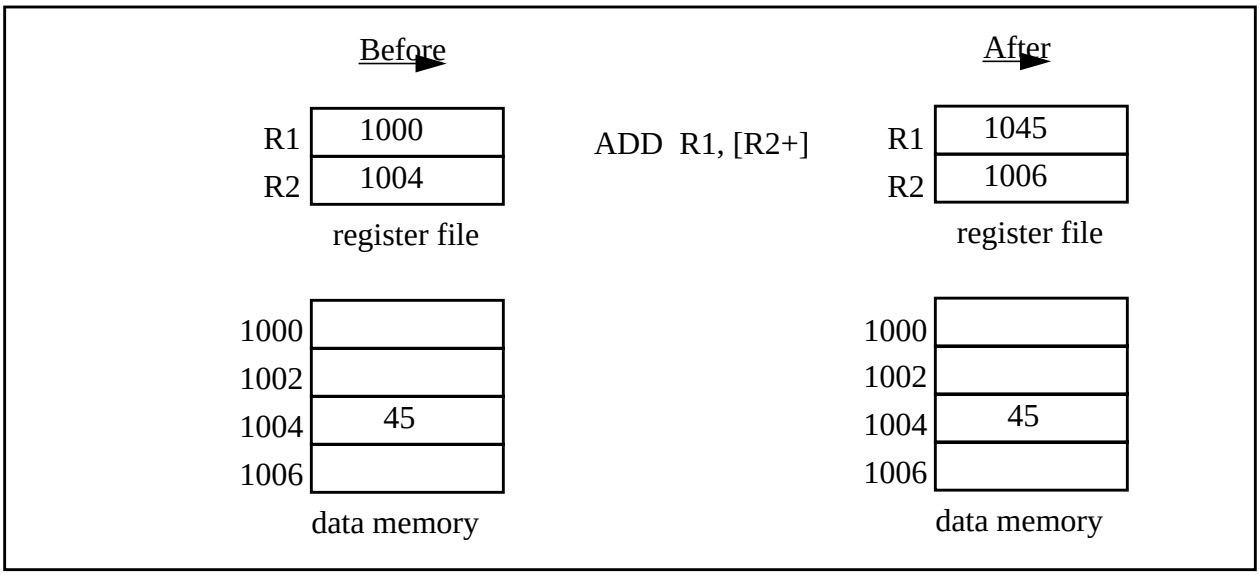


Figure 2.11 Indirect Addressing with Auto-Increment

Since indirect memory references and immediate data values do not implicitly identify the size of the operation to be performed, a few XA instructions must have an operation size explicitly called out. An example would be the instruction: "MOV [R1], #1". The immediate data value does not specify the operation size, and the value stored in memory at the location pointed to by R1 could be either a byte or a word. To clarify the intent of such an instruction, a size identifier is added to the mnemonic as follows: "MOV.b [R1], #1". This tells us that the operation should be performed on a byte. If the line read "MOV.w [R1], #1", it would be a word operation.

If a direct data address is used in an instruction, the address is simply written into the instruction: "ADD 123, R1", meaning to add the contents of register R1 to the data memory value stored at direct address 123. In an actual program, the direct data address could be given a name to make the program more readable, such as "ADD Count, R1".

Operations using Special Function Registers (SFRs) are written in a way similar to direct addresses, except that they are normally called out by their names only: "MOV PSW,#12". Using actual SFR addresses rather than their names in instructions makes the code both harder to read and less transportable between XA derivatives.

Bit addresses within instructions may be specified in one of several ways. A bit may be given a unique name, or it may be referred to by its position within some larger register or entity. An example of a bit name would be one of the status flags in the PSW, for instance the carry ("C") flag. To clear the carry flag, the following instruction could be used: "CLR C". The same bit could be addressed by its position within the PSW as follows: "CLR PSWL.7", where the period (".") character indicates that this is a bit reference. A program may use its own names to identify bits that are defined as part of the application program.

Finally, code addresses are written within instructions either by name or by value. Again, a program is more readable and easier to modify if addresses are called out by name. Examples are: "JMP Loop" and "JMP 124".

2.5.2 Instruction Set Summary

The following pages give a summary of the XA instruction set. For full details, consult Chapter 6.

Basic Arithmetic, Logic, and Data Movement Instructions

The most used operations in most programs are likely to be the basic arithmetic and logic instructions, plus the MOV (move data) instruction. The XA supports the following basic operations:

ADD	Simple addition.
ADDC	Add with carry.
SUB	Subtract.
SUBB	Subtract with borrow.
CMP	Compare.
AND	Logical AND.
OR	Logical OR.
XOR	Exclusive-OR.

These instructions support all of the following standard XA data addressing mode combinations::

Operands	Description
R, R	The source and destination operands are both registers.
R, [R]	The source operand is indirect, the destination operand is a register.
[R], R	The source operand is a register, the destination operand is indirect.
R, [R+]	The source operand is indirect with auto-increment, the destination operand is a register.
[R+], R	The source operand is a register, the destination operand is indirect with auto-increment.
R, [R+offset]	The source operand is indirect with an 8 or 16-bit offset, the destination operand is a register.
[R+offset], R	The source operand is a register, the destination operand is indirect with an 8 or 16-bit offset.
direct, R	The source operand is a register, the destination operand is a direct address.
R, direct	The source operand is a direct address, the destination operand is a register.
R, #data	The source operand is an 8 or 16-bit immediate value, the destination operand is a register.
[R], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect.

Operands	Description
[R+], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with auto-increment.
[R+offset], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with an 8 or 16-bit offset.
direct, #data	The source operand is an 8 or 16 bit immediate value, the destination operand is a direct address.

Other instructions on the XA use different operand combinations. All XA instructions are covered in detail in the Instruction Set section. Following is a summary of other instruction types:Additional arithmetic instructions

Additional arithmetic instructions

ADDS	Add short immediate (4-bit signed value).
NEG	Negate (twos complement).
SEXT	Sign extend.
MUL	Multiply.
DIV	Divide.
DA	Decimal adjust.
ASL	Arithmetic shift left.
ASR	Arithmetic shift right.
LEA	Load effective address.

Additional logic instructions

CPL	Complement (ones complement or logical inverse).
LSR	Logical shift right.
NORM	Normalize.
RL	Rotate left.
RLC	Rotate left through carry.
RR	Rotate right.
RRC	Rotate right through carry.

Other data movement instructions

MOVS	Move short immediate (4-bit signed value).
MOVC	Move to or from code memory.
MOVX	Move to or from external data memory.
PUSH	Push data onto the stack.
POP	Pop data from the stack.
XCH	Exchange data in two locations.

Bit manipulation instructions

SETB	Set (write a 1 to) a bit.
CLR	Clear (write a 0 to) a bit.
MOV	Move a bit to or from the carry flag.
ANL	Logical AND a bit (or its inverse) to the carry flag.
ORL	Logical OR a bit (or its inverse) to the carry flag.

Jump, branch, and call instructions

BR	Branch to code address (plus or minus 256 byte range).
JMP	Jump to code address (range depends on specific JMP variation).
CALL	Call subroutine (range depends on specific CALL variation).
RET	Return from subroutine or interrupt.
Bcc	Conditional branches with 15 possible condition variations.
JB, JNB	Jump if a bit set or not set.
CJNE	Compare two operands and jump if they not equal.
DJNZ	Decrement and jump if the result is not zero.
JZ, JNZ	Jump on zero or not zero (included for 80C51 compatibility).

Other instructions

NOP	No operation (used mainly to align branch targets).
BKPT	Breakpoint (used for debugging).
TRAP	Software trap (used to call system services in a multitasking system).
RESET	Reset the entire chip.

2.6 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices, and initiates code read, data read, or data write strobes. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

As described in section 4.4.4, the initial external bus width is hardware settable, and the XA determines its value (8 or 16 bits) during the reset sequence.

2.6.1 External Bus Signals

The standard XA external bus supports 8 or 16-bit data transfers and up to 24 address lines. The precise number of address lines varies by derivative. The standard control signals and their functions for the external bus are as follows:

Signal name	Function
ALE	Address Latch Enable. This signal directs an external address latch to store a portion of the address for the next bus operation. This may be a data address or a code address.
$\overline{\text{PSEN}}$	Program Store Enable. Indicates that the XA is reading code information over the bus. Typically connected to the Output Enable pin of external EPROMs.
$\overline{\text{RD}}$	Read. The external data read strobe. Typically connected to the $\overline{\text{RD}}$ pin of external peripheral devices.
$\overline{\text{WRL}}$	Write. The low byte write strobe for external data. Typically connected to the $\overline{\text{WR}}$ pin of external peripheral devices. For an 8-bit data bus, this is the only write strobe. For a 16-bit data bus, this strobe applies only to the lower data byte.
$\overline{\text{WRH}}$	Write High. This is the upper byte write strobe for external data when using a 16-bit data bus.
WAIT	Wait. Allows slowing down any type external bus cycle. When asserted during a bus operation, that operation waits for this signal to be de-asserted before it is completed.

2.6.2 Bus Configuration

The standard XA bus is user configurable in several ways. First, the bus size may be configured to either 8 bits or 16 bits. This may be configured by the logic level on a pin at reset, or under firmware control (if code is initially executed from on-chip code memory) prior to any actual external bus operations. As on the 80C51, the $\overline{\text{EA}}$ pin determines whether or not on-chip code memory is used for initial code fetches.

Second, the number of address lines may be configured in order to make optimal use of I/O ports. Since external bus functions are typically shared with I/O ports and/or peripheral I/O functions, it is advantageous to set the number of address lines to only what is needed for a particular application, freeing I/O pins for other uses.

2.6.3 Bus Timing

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, $\overline{\text{PSEN}}$ width, $\overline{\text{RD}}$ and $\overline{\text{WRL}}$/$\overline{\text{WRH}}$ width, and data hold time from $\overline{\text{WRL}}$/$\overline{\text{WRH}}$. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

The following figures show the basic sequence of events and timing of typical XA bus accesses. For more detailed information, consult Section 7 and the device data sheet.

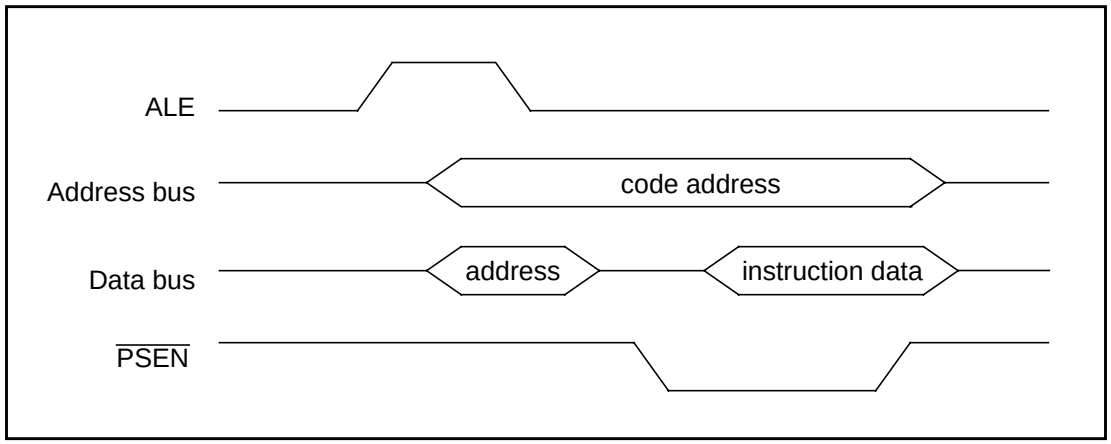


Figure 2.12 Typical External Code Read.

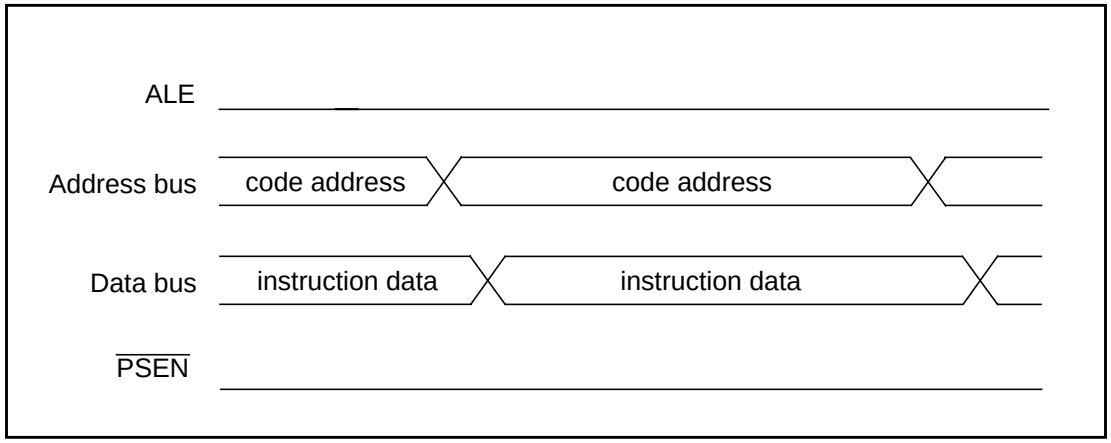


Figure 2.13 Optimized (Sequential Burst) External Code Read.

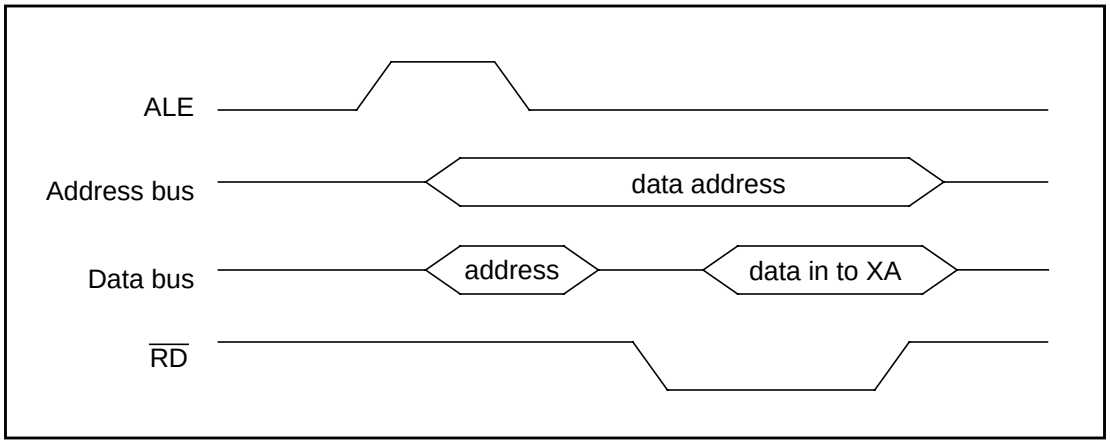


Figure 2.14 Typical External Data Read.

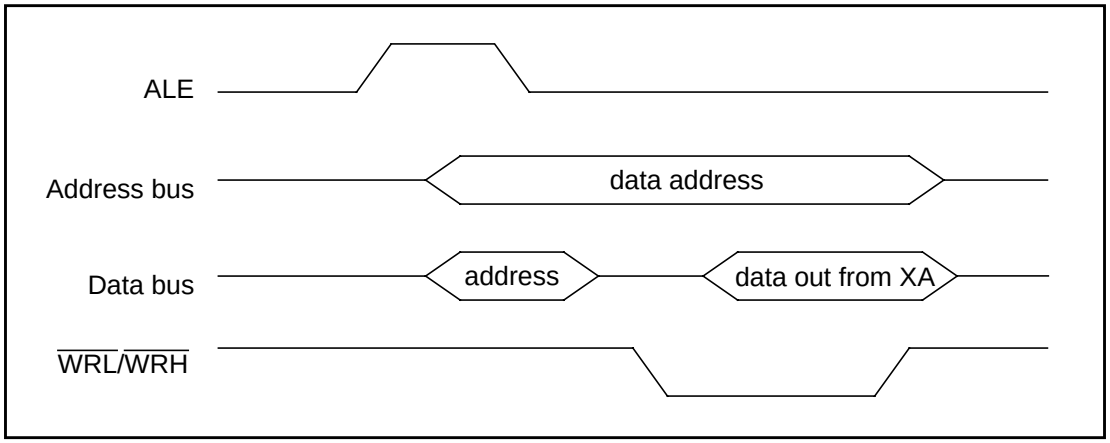


Figure 2.15 Typical External Data Write.

2.7 Ports

Standard I/O ports on the XA have been enhanced to provide better versatility and programmability than was previously available in the 80C51 and most of its derivatives. Access to the I/O ports from a program is through SFR addresses assigned to those ports. Ports may be read and written is the same manner as any other SFR.

The XA provides more flexibility in the use of I/O ports by allowing different output configurations. See Figure 2.16. Port outputs may be programmed to be quasi-bidirectional (80C51 style ports), open drain, push-pull, and high impedance (input only).

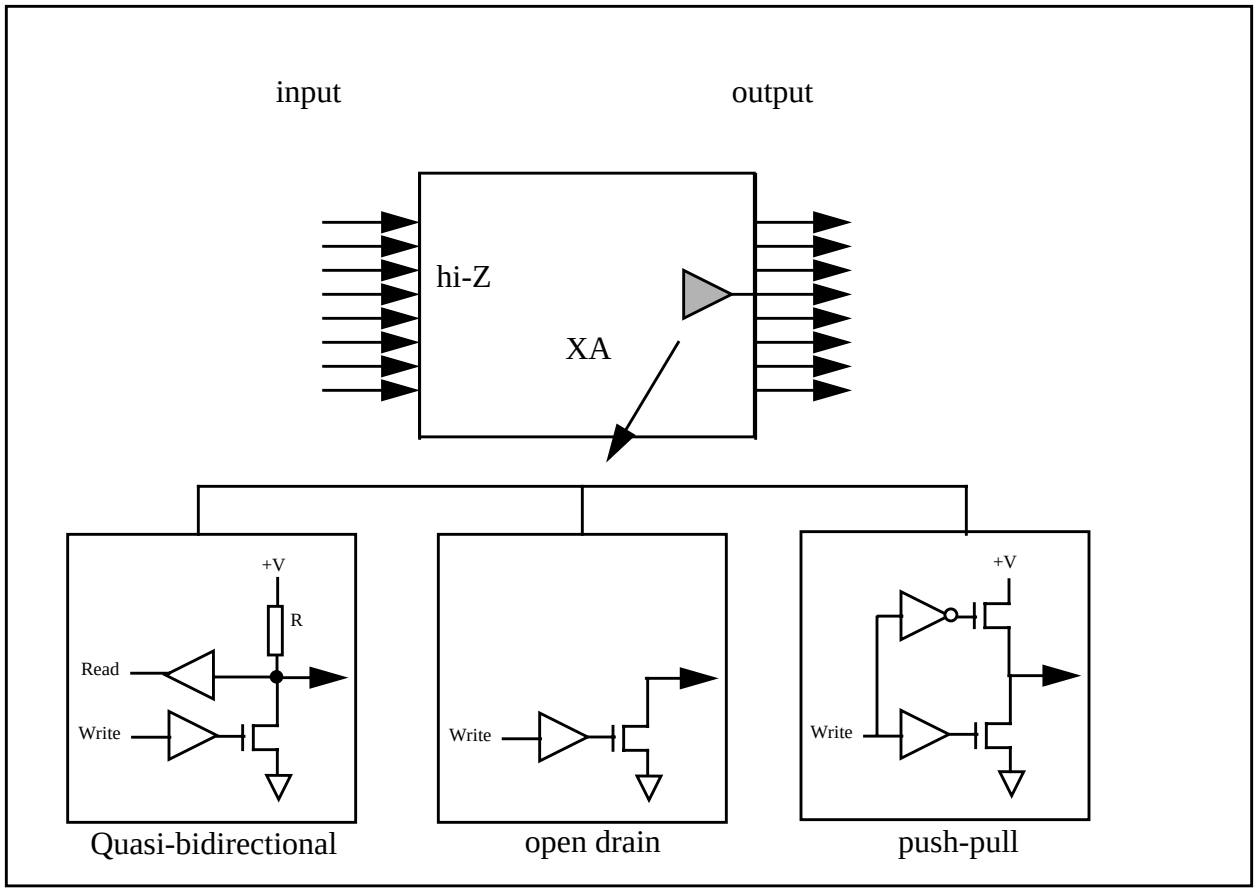


Figure 2.16 XA Port Pins with Driver Option Detail

2.8 Peripherals

The XA CPU core is designed to make derivative design fast and easy. Peripheral devices are not part of the core, but are attached by means of a Special Function Register bus, called the SFR bus, which is distinct from the CPU internal buses. So, a new XA derivative may be made by designing a new SFR bus compatible peripheral function block, if one does not already exist, then attaching it to the XA core.

2.9 80C51 Compatibility

The 80C51 is the most extensively designed-in 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this device family. For customers who use the 80C51 or one of its derivatives, preservation of their investment in code development is an important consideration. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with this higher-performance 16-bit controller. At the same time, the XA hardware was designed with the clear goal of upward compatibility. 80C51 designs may be migrated to the XA with very few changes necessary to software source or hardware.

The XA provides an 80C51 Compatibility Mode, which essentially replicates the 80C51 register architecture for the best possible upward compatibility. In the alternative Native Mode, the XA operates as an optimized 16-bit microcontroller incorporating the best conceptual features of the original 80C51 architecture.

Many trade-offs and considerations were taken into account in the creation of the XA architecture. The most important goal was to make it possible for a software translator to convert 80C51 assembler source code to XA source code on a 1:1 basis, i.e., one XA instruction for one 80C51 instruction.

Some specific compatibility issues are summarized in the following two sections. See Chapter 9 for a complete description of compatibility.

2.9.1 Software Compatibility

Several basic goals were observed in order to design 80C51 software compatibility for the XA, while avoiding over-complicating the XA design. Following are some key issues for XA software:

• Instruction mapping. Each 80C51 instruction translates into one XA instruction. Multi-instruction combinations that could result in problems if split by an interrupt were avoided as much as possible. Only one 80C51 instruction does not have a one-to-one direct replacement with an XA instruction (this instruction, XCHD, is extremely rarely used).

• "As-is" instructions. Most XA instructions are more powerful supersets of 80C51 instructions. Where this was not possible, the original 80C51 instruction is included "as-is" in the XA instruction set.

• Timing. Instruction timing must necessarily change in order to improve performance. The XA does not attempt to retain timing compatibility with the 80C51; rather, the design simply maximizes instruction execution speed. When 80C51 code that is timing critical is translated to the XA, the user must re-analyze the timing and make adjustments.

• SFR Access. Translation of SFR accesses is usually simple, since SFRs are normally referenced by name. Such references are simply retained in the translated XA code. If program source code from a specific 80C51 derivative references an SFR by its address, the translator can directly substitute the appropriate XA SFR, provided both the 80C51 and the XA derivative are correctly identified to the translator.

2.9.2 Hardware Compatibility

The key goal for hardware was to produce a familiar architecture with a good deal of upward compatibility.

• Memory Map. A major consideration in hardware compatibility of the XA with the 80C51 is the memory map. The XA approaches this issue by having each memory area (registers, data memory, code memory, stack, SFRs) be a superset of the corresponding 80C51 area.

• Stack. One area where a functional change could not be avoided is in the use of the processor stack. Due to the fact that the XA supports 16-bit operations in memory, it was necessary to change the direction of stack growth to downward –the standard for 16-bit processors– in order to match stack usage with efficient access of 16-bit variables in memory. This is an important consideration for support of high-level language compilers such as C.

• Pin-for-pin compatibility. XA derivatives are not intended to be exactly pin-compatible with other 80C51 derivatives that have similar features. Many on-chip XA peripherals, for example, have improved capabilities, and maintaining pin-for-pin compatibility would limit access to these capabilities. In general, peripherals have been made upward compatible with the original 80C51 devices, and most enhancements are added transparently. In these cases, 80C51 code will operate correctly on the 80C51 functional subset.

• Bus Interface. The external bus on the XA is an example of a trade-off between 80C51 compatibility and performance. In order to provide more flexibility and maximum performance, the 80C51 bus had to be changed somewhat. The differences are described in detail in the section on the external bus.

3 XA Memory Organization

3.1 Introduction

The memory space of XA is configured in a Harvard architecture which means that code and data memory (including sfrs) are organized in separate address spaces. The XA architecture supports 16 Megabytes (24-bit address) of both code and data space. The size and type of memory are specific to an XA derivative.

The XA supports different types of both code and data memory e.g.,code memory could be Eprom, EEProm, OTP ROM, Flash, and Masked ROM whereas data memory could be RAM, EEProm or Flash.

This chapter describes the XA Memory Organization of register, code, and data spaces; how each of these spaces are accessed, and how the spaces are related.

3.2 The XA Register File

The XA architecture is optimized for arithmetic, logical, and address-computation operations on the contents of one or more registers in the XA Register File.

3.2.1 Register File Overview

The XA architecture defines a total of 16 word registers in the Register File:
In the baseline XA core, only R0 through R7 are implemented. These registers are available for unrestricted use except R7– which is the XA stack pointer, as illustrated in Figure 3.1. In effect, the XA registers provide users with at least 7 distinct “accumulators” which may be used for all operations. As will be seen below, the XA registers are accessible at the bit, byte, word, and doubleword level.

Additional global registers, R8 through R15, are reserved and may be implemented in specific XA derivatives. These registers, when available, are equivalent to R0 through R7 except byte access and use as pointers will not be possible (only word, double-word, and bit-addressable). The Register File is independent of all other XA memory spaces (except in Compatibility Mode; see chapter 9).

Register File Detail

Figure 3.2 describes R0 through R7 in greater detail.

Byte, Word, and Doubleword Registers

All registers are accessible as bits, bytes, words, and –in a few cases– doublewords. Bit access to registers is described in the next section. As for byte and word accesses, R1 –for example– is a word register that can be word referenced simply as “R1”. The more significant byte is labeled as “R1H” and the less significant byte of R1 is referenced as “R1L”. Double-word registers are always formed by adjacent pairs of registers and are used for 32 bit shifts, multiplies, and divides. The pair is referenced by the name of the lower-numbered register (which contains the

less significant word), and this must have an even number. Thus valid double-register pairs are (R0,R1), (R2,R3), (R4,R5) and (R6, R7).

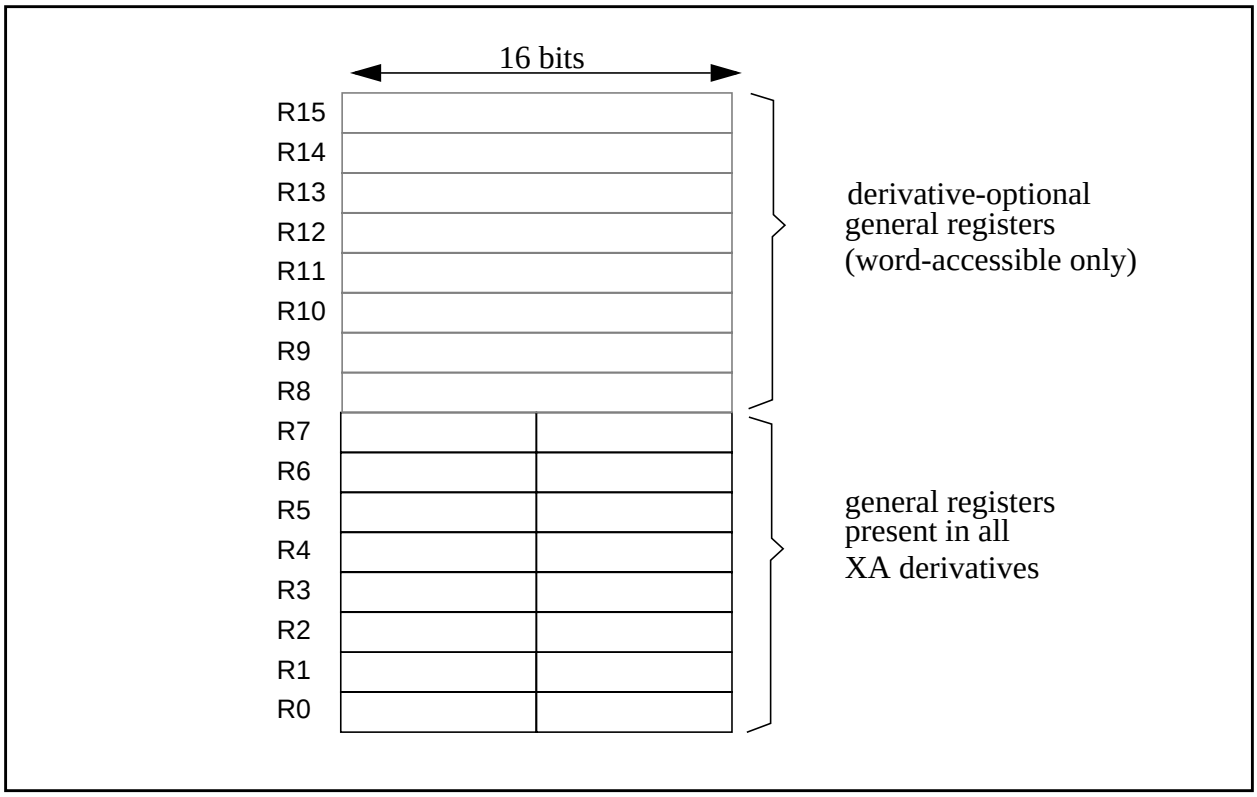


Figure 3.1 XA Register File Overview

As described in section 4.7, there are two stack pointers, one for user mode and another for system mode. At any given instant only one stack pointer is accessible and its value is in R7. When PSW.SM is 0, user mode is active and the USP is accessible via R7. When PSW.SM is 1, the XA is operating in system mode, and SSP is in SP (R7). (Note however, as described in Chapter 4, all interrupts save stack frames on the system stack, using the SSP, regardless of the current operating mode.)

There are four distinct instances of registers R0 through R3. At any given time, only 1 set of the 4 banks is active, referenced as R0 through R3, and the contents of the other banks are inaccessible. This allows high-speed context-switching, for example, for interrupt service routines. **PSW** bits **RS1** and **RS0** select the active register bank:

RS1	RS0	visible register bank
0	0	bank 0
0	1	bank 1
1	0	bank 2
1	1	bank 3

PSW.RSn are writable when the XA is operating in system or user mode, and programs running in either mode may explicitly change these bits to make selected banks visible one at a time. More commonly, the interrupt mechanism, as described in Chapter 4, provides automatic implicit register bank switching so interrupt handlers may immediately begin operating in a reserved register context.

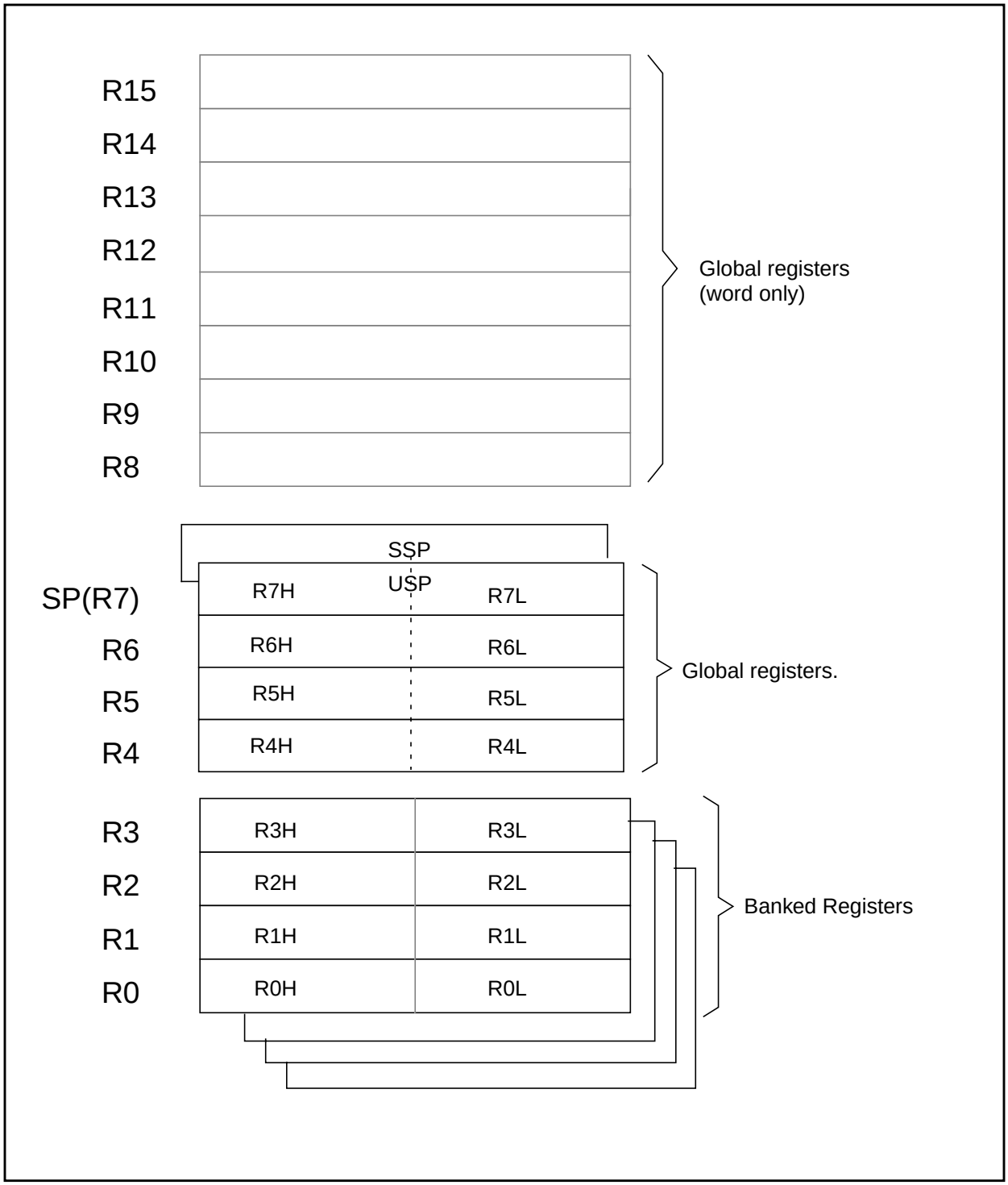


Figure 3.2 XA Register File

Bit Access to Registers

The XA Registers are all bit addressable. Figure 3.3 shows how bit addresses overlie the basic register file map. In general, absolute bit references as given in this map are unnecessary. XA software development tools provide symbolic access to bits in registers. For example, bit 7 may be designated as “R0.7” with no ambiguity

Bit references to banked registers R0 through R3 access the currently accessible register bank, as set by **PSW** bits **RS1**, **RS0** and the currently selected stack pointer USP or SSP. The unselected registers are inaccessible..

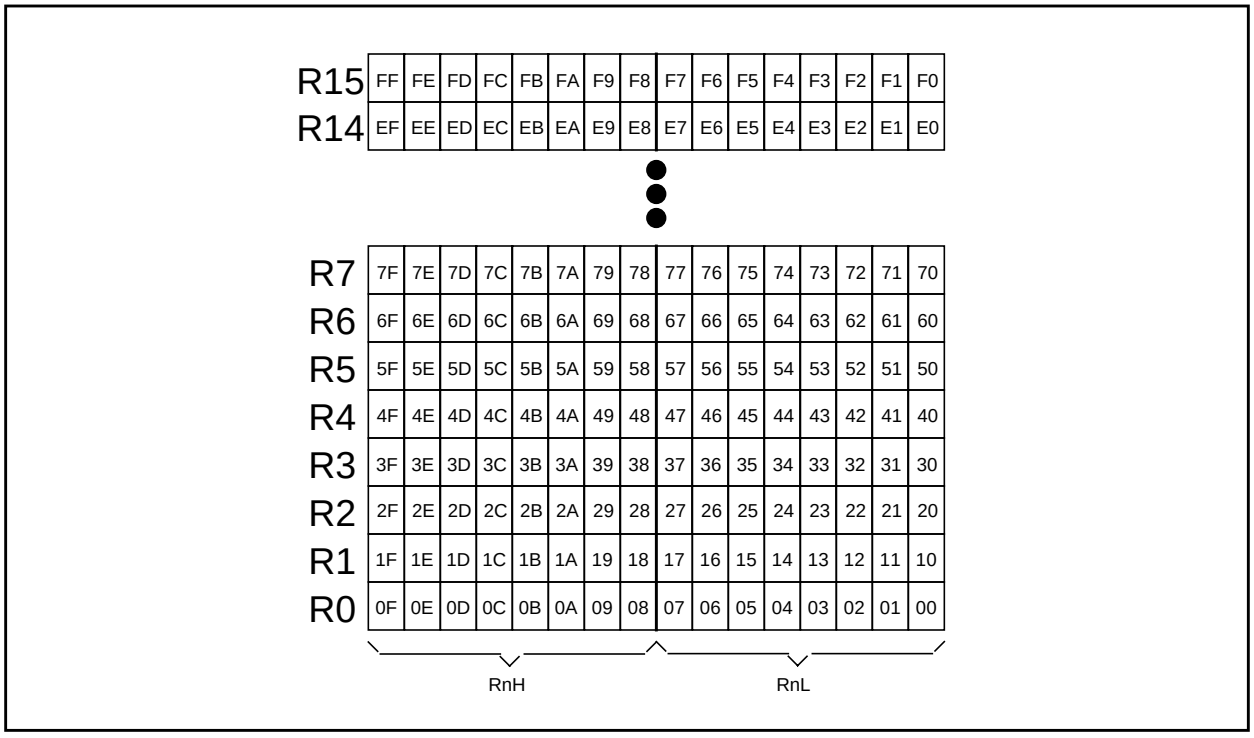


Figure 3.3 Bit Address to Registers

3.3 The XA Memory Spaces

The XA divides physical memory into program and data memory spaces. Twenty-four address bits, corresponding to a 16MB address space, are defined in the XA architecture. In any given XA implementation, fewer than all twenty-four address bits may actually be used, and there is provision for a small-memory mode which uses only 16-bit addresses; see Chapter 4.

Code and data memory may be on-chip or external, depending on the XA variant and the user implementation. Whether a specific region is on-chip or external does not, in general, affect access to the memory.

3.3.1 Bytes, Words, and Alignment

XA memory is addressed in units of *bytes*, where each byte consists of 8 bits. A *word* consists of two bytes, and the word storage order is “Little-Endian”, that is, the less significant byte of word data is located at a lower memory address. See Figure 3.4.

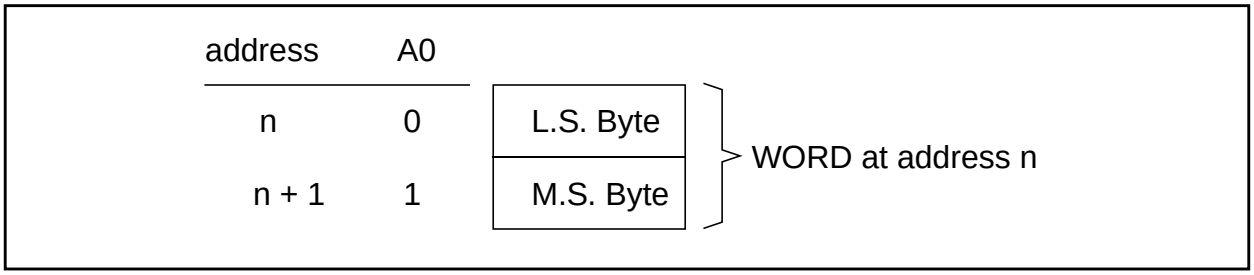


Figure 3.4 Memory byte order

Any word access must be aligned at an even address (Address bit A0=0). If an odd-aligned word access is attempted the word at the next-smallest even address will be accessed, that is, A0 will be set to 0.

The external XA memory spaces may be accessed in byte or word units but the hardware access method does not affect the even alignment restriction on word accesses.

3.4 Data Memory

The data memory space starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. As will be described below, the data memory space is segmented into 256 segments of 64K bytes each. *External Data Memory* starts at the first address following the highest *Internal Data Memory* location. In general, at least 512 bytes of Internal Data Memory, starting at location 0, will be provided in all XA implementations; however, there is no inherent minimum or maximum architectural limitation on Internal Data Memory.

The upper 16 segments of data memory (addresses F0:0000 through FF:FFFF hexadecimal) are reserved for special functions in XA derivatives. A similar range is reserved in the code memory space, see section 3.5.

3.4.1 Alignment in Data Memory

There are no data memory alignment restrictions except that placed on word accesses to all memory: Words must be fetched from even addresses. An attempt to fetch a word at an odd address will fetch a word from the preceding even address.

3.4.2 External and Internal Overlap

If External Data Memory is placed by external logic at addresses that overlaps Internal Data Memory, the Internal Data Memory generally takes precedence. The overlapped portion of the External memory may be accessed only by using a form of the MOVX instruction; see Chapter 6. The use of MOVX always forces external data memory fetch in XA. For non-overlapped portion of external data memory, no MOVX is required.

3.4.3 Use and Read/Write Access

Data memory is defined as read-write, and is intended to contain read/write data. It is logically impossible to execute instructions from XA Data Memory. It is possible, and a common practice, to add logic to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such a modified Harvard architecture, implemented with external logic, it is possible –but not recommended– to write self-modifying XA code. No such overlap is possible for internal data memory.

3.4.4 Data Memory Addressing

XA data memory addressing is optimized for the needs of embedded processing. Data memory in the XA is divided into 64K byte segments. This provides an intrinsic protection mechanism for multitasking applications and improves performance by requiring fewer address bits for localized accesses.

Addressing through Segment Registers

Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require full use of the XA 16 Mbyte address space. Two segment registers are defined in the XA architecture for use in accessing data memory, the Data Segment Register (**DS**), and the Extra Segment Register **(ES**). As user stacks are located in the segment specified by **DS**, it is probably most convenient to address user data structures through **ES**. Each pointer register, namely R0 through R6, is associated with one of the segment registers via the Segment Select (**SSEL**) register as illustrated in Figure 3.5.

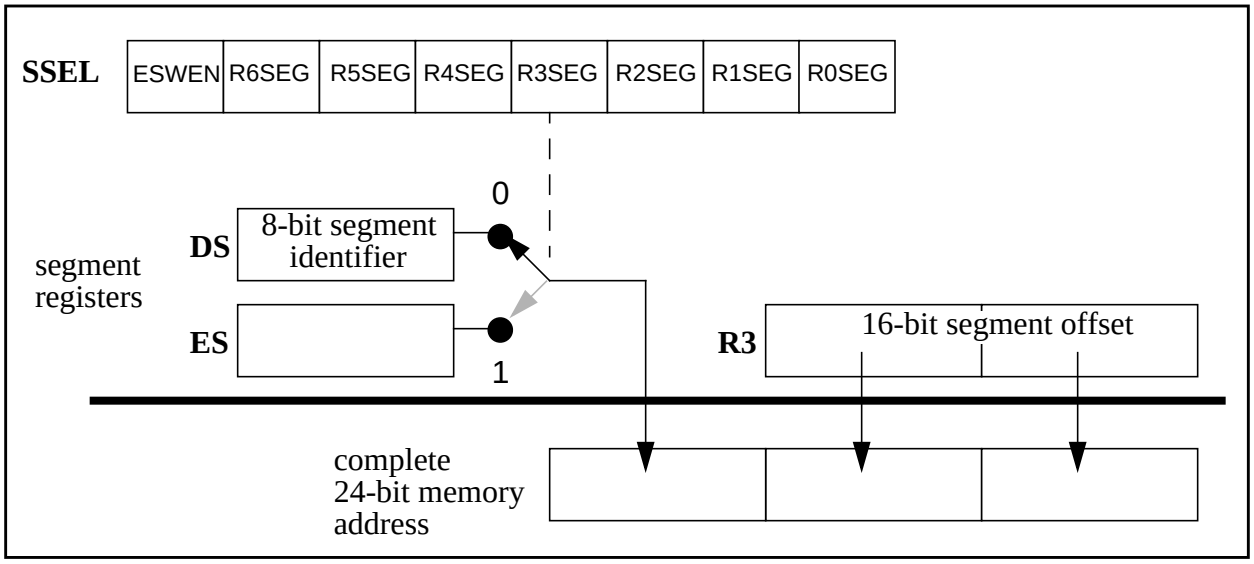


Figure 3.5 Address generation

A 0 in the SSEL bit corresponding to the pointer register selects DS (default on RESET) and 1 selects the ES. For example, when R3 contains a pointer value, the full 24 bit address is formed by concatenating DS or ES, as determined by the state of SSEL bit 3, as the most significant 8 bits. As a consequence of segmented addressing, the XA data memory space may be viewed as 256 segments of 64K bytes each (Figure 3.6).

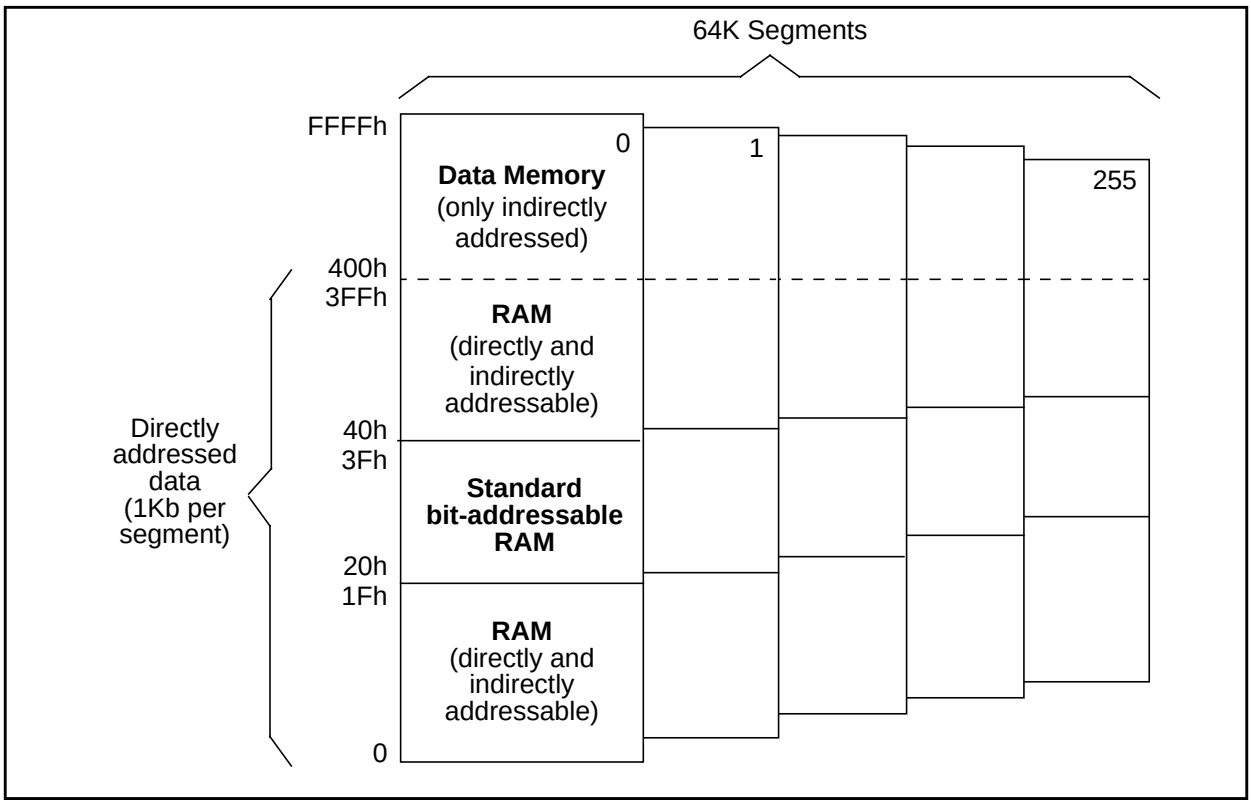


Figure 3.6 Data memory segmentation

If R7 (the stack pointer) is used as a normal indirect pointer, the data segment addressed will always be segment 0 in System Mode and the DS segment in User Mode. More information about the System and User modes may be found in sections 4 and 5.

The ESWEN (bit 7 of SSEL) can be programmed only in the System Mode to enable (1) or disable (0) write privileges to data segment via ES register in the User Mode. This bit defaults to the disabled (0) state after reset.

Addressing Modes

The XA provides flexible data addressing modes. Arithmetic, logic, and data movement instructions generally support the following data memory access:

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with a 16-bit pointer in a register.

Direct. The first 1K bytes of data in each segment may be accessed by an address contained within the instruction. *Indirect with offset.* A signed byte/word offset contained within the instruction is added to the contents of a pointer register, and the result is concatenated with the 8-bit segment register DS to produce a complete 24-bit address.

Indirect with auto-increment. Indirect addresses are formed as above and the pointer register contents are automatically incremented.

Bit-level. Bit-level addresses are absolute references to specific bits.

Data move instructions and some special purpose instructions also have additional data addressing modes as described in Chapter 6.

Indirect Addressing

The entire 16 MByte address space is accessible via register-indirect addressing with a segment register, as illustrated by Figure 3.7 (Note that for simplicity, this figure omits showing how the Extra Segment or Data Segment Register is chosen using **SSEL**.).

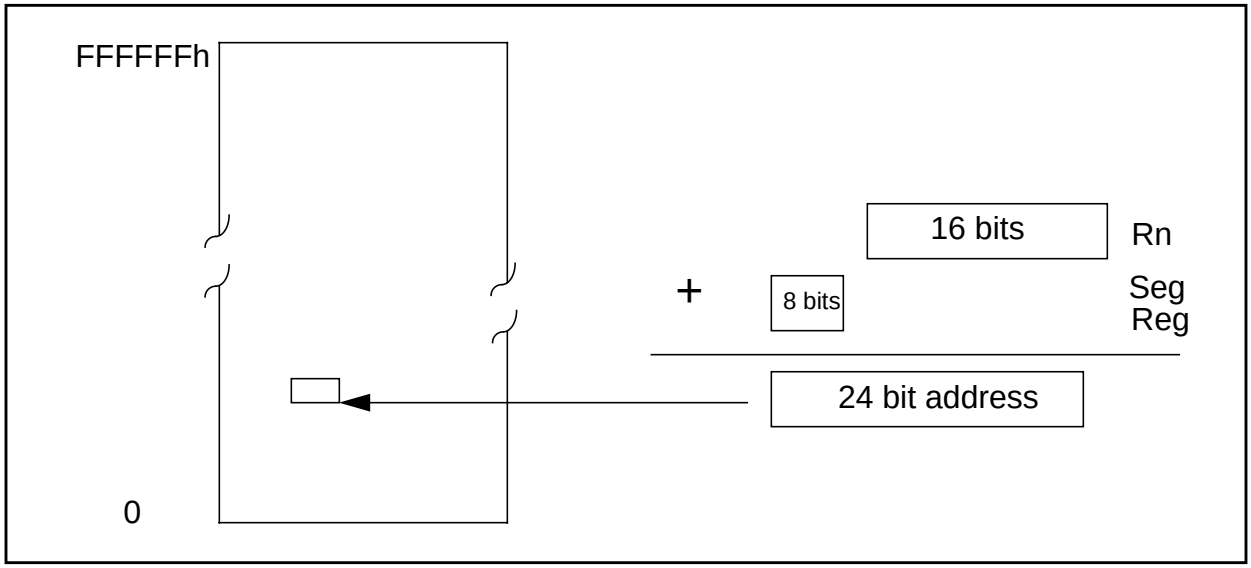


Figure 3.7 Indirect Access to 24 Bit Address Space

Indirect addressing with an offset is a variant of general indirect addressing in which an 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode gives access to data structures when a pointer register contains the starting address of the structure. It also supports stack-based parameter passing.

Indirect addressing with autoincrement is another variant of indirect addressing in which the pointer register contents are automatically incremented following the operation. When the operand is a byte, the increment is one; when the operand is a word, the increment is 2. Using indirect addressing with auto-increment provides a convenient method of traversing data structures smaller than 64K bytes. For data structures exceeding 64K bytes in length, the program code must explicitly adjust the segment register at page boundaries.

Address generation in these two modes of indirect addressing is illustrated inFigures 3.8 and 3.9. When using indirect addressing care is necessary to avoid accessing a word quantity at an odd address. This will result in an access using the next-lower even address, which is generally not desirable. Note that the indirect addressing with an offset will be successful in this case as long as the final, effective address is even. That is, both the base address and the offset may be odd.

Direct Addressing

The first 1K of each segment is directly addressable. Address generation for the direct address mode is summarized in Figure 3.10. Segment register DS is always used.

Direct data-reference instructions encode a maximum of 10 address bits, which are zero extended to sixteen bits and concatenated with DS to form an absolute 24 bit address. In all segments, direct addressing can be used to access any byte in the first 1K bytes of the segment.

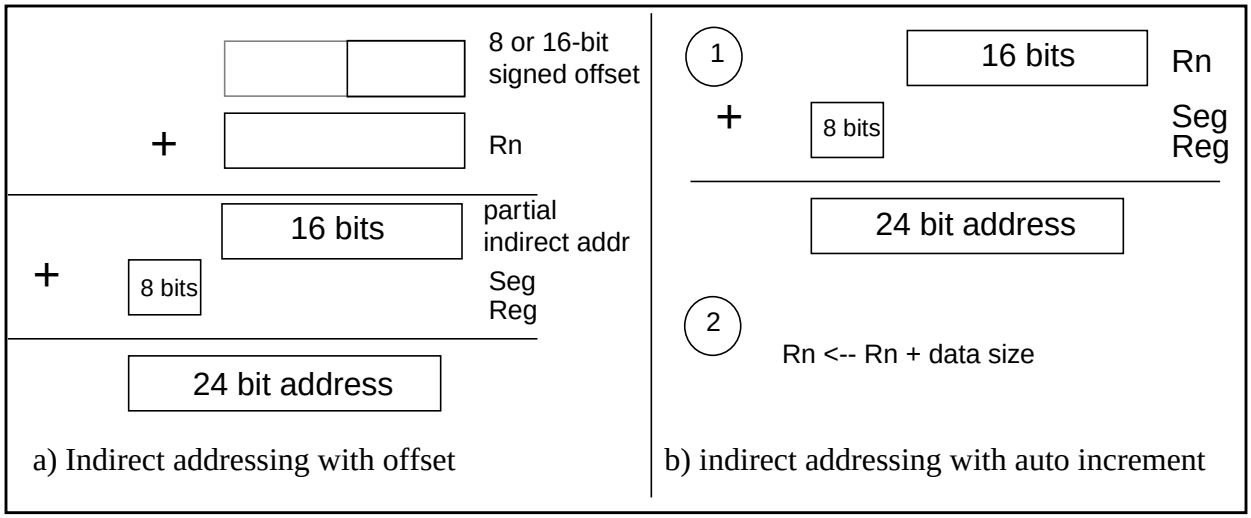


Figure 3.8 Indirect Addressing

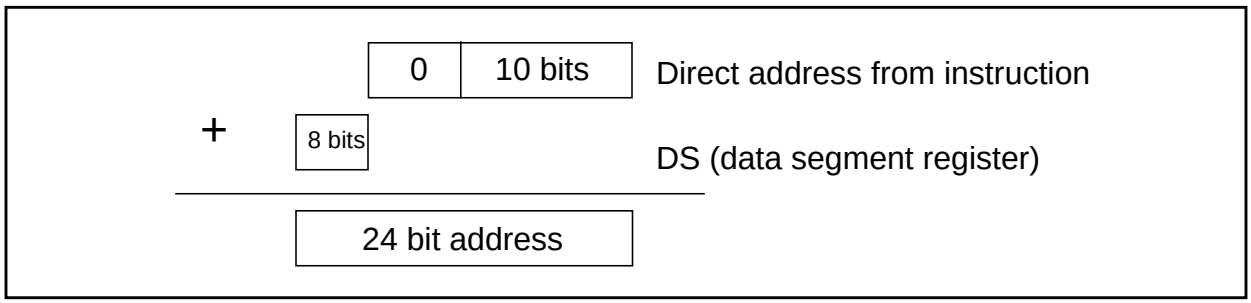


Figure 3.9 Direct address generation

SFR Addressing

A 1K portion of the direct address space, addresses 400h through 7FFh, is reserved for SFR addresses. The SFR address space uses a portion of the direct address space, but represents a completely distinct logical area that is not related to the data memory segmentation scheme. See section 3.6 for a complete description of SFR access.

Bit Addressing

Thirty-two bytes of each segment of data memory are also bit-addressable, starting at offset 20h in the segment addressed by the DS register. Address generation for bit addressing in the data memory space is shown in Figure 3.10. As described in chapter 6, bits are encoded in instructions as 10 bits. Figure 3.11 shows the bit addresses as they appear in memory .

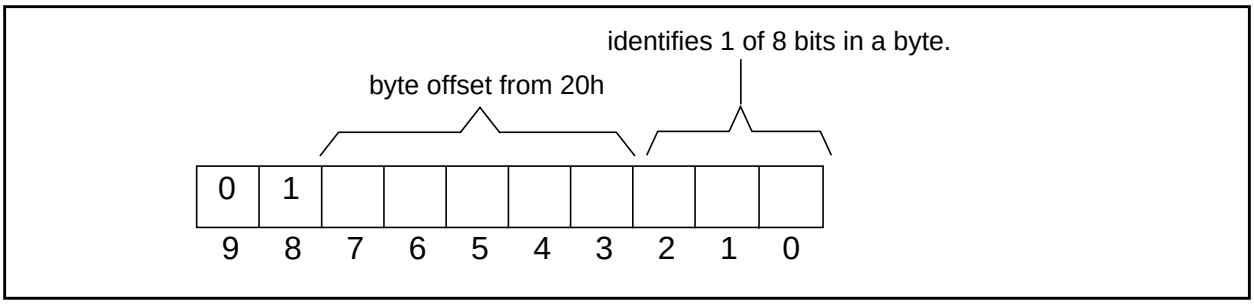


Figure 3.10 Bit address generation in direct memory space

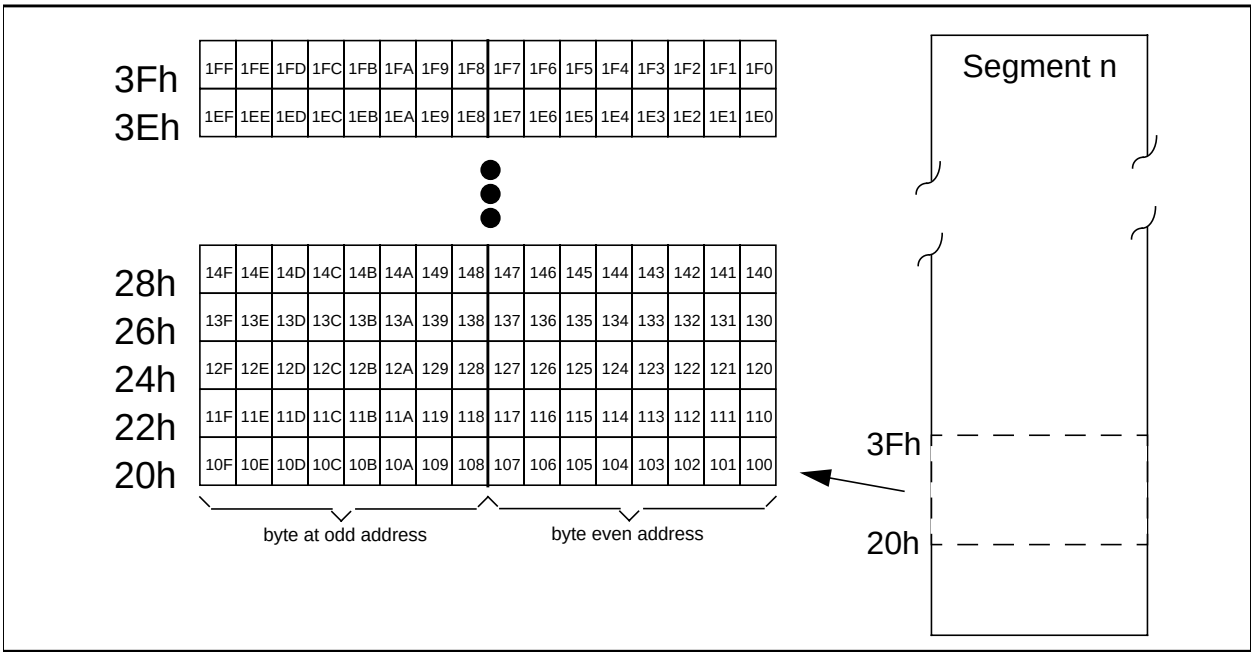


Figure 3.11 Direct memory bit addressing

3.5 Code Memory

Code memory starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. *External Code Memory* (off-chip) starts at the first address following the highest *Internal Code Memory* (on-chip) location, if any. If code memory is present on-chip, it always starts at location 0.

The upper sixteen 64K byte code pages (addresses F00000 through FFFFFF hexadecimal) are reserved for special functions in XA derivatives. The same address range is reserved in the data memory space, see section 3.4.

3.5.1 Alignment in Code Memory

As instructions are variable in length, from 1 to 6 bytes (see Chapter 6), instructions in code memory can be located at odd addresses. As described in Chapter 6, instruction branch targets, i.e., targets of jumps, calls, branches, traps, and interrupts must be aligned on an even address.

3.5.2 External and Internal Overlap

If External Code Memory is placed by external logic at locations that overlap Internal Code Memory, the Internal Code Memory takes precedence, and the overlapped portion of the External memory will in not be accessed. However, on XA implementations that provide an External Address ($\overline{EA}$) hardware input, setting EA low will cause external program memory to be used.

3.5.3 Access

Code memory is intended to contain executable XA instructions**.** The XA architecture supports storing constant data in Code Memory and provides special access modes for retrieving this information. Constant data is implicitly stored within the instruction of many data manipulation instructions when immediate operands are specified.

It is possible, and a common practice, to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such an architecture, implemented with external logic, code memory is logically read-only memory that is writable when accessed as external data memory. No such overlap is possible for internal code memory.

MOVC addressing in Code Memory

A special instruction, MOVC, is defined in the XA for accessing constant data (e.g lookup tables, string constants etc.) stored in code memory. There is a standard form of MOVC that reflects the native XA architecture, and there are two variations that reflect 80C51 compatibility; see Chapter 9 for details of 80C51 compatibility. The standard form of MOVC uses a 16-bit register value as a pointer, appended to either the top 8 bits of the Program Counter (PC) or the Code Segment register (CS) to form a 24-bit address, as shown in Figure 3.12. The source for the upper 8 address bits is determined by the setting of the segment selection bit (0 = PC and 1= CS) in the SSEL register that corresponds to the operand register.

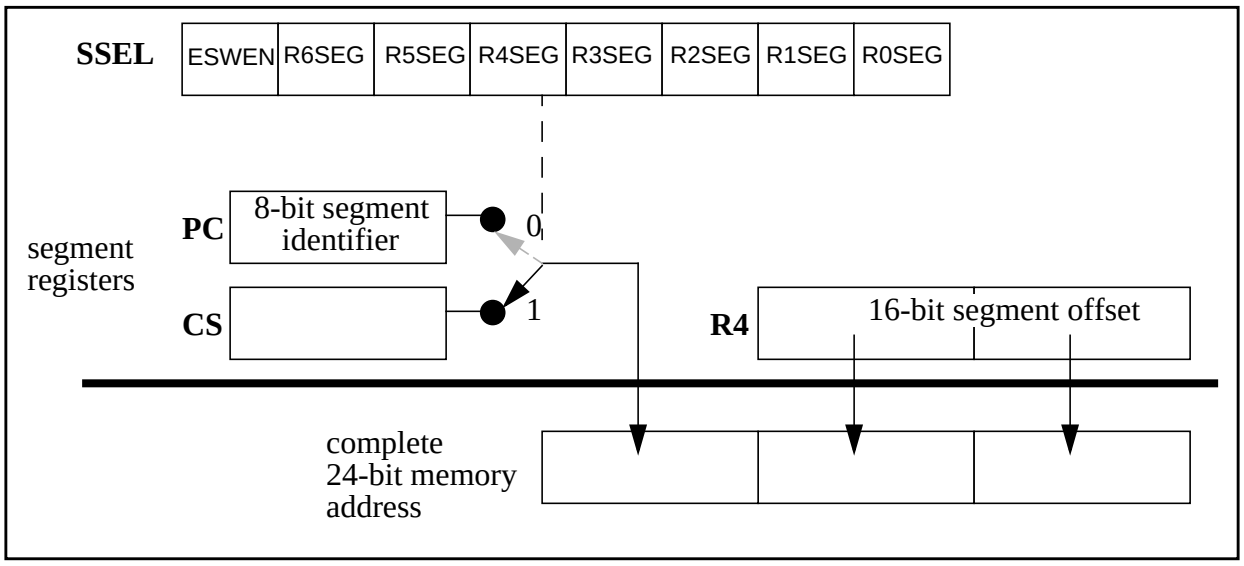


Figure 3.12 MOVC addressing in code memory

3.6 Special Function Registers (SFRs)

Special Function Registers (SFRs) provide a means for programs to access CPU control and status registers, peripheral devices, and I/O ports. The SFR mechanism provides a consistent mechanism for accessing standard portions of the XA core, peripheral functions added to the core within each XA derivative, and external devices as implemented in future derivatives.

Figure 3.13 highlights the core registers that are accessed as SFRs: **PCON**, **SCR**, **SSEL**, **PSWH**, **PSW**L, **CS**, **ES**, **DS**. Communication with these registers as well as on-chip peripheral devices is performed via the dedicated Special Function Register Bus (see section 8).

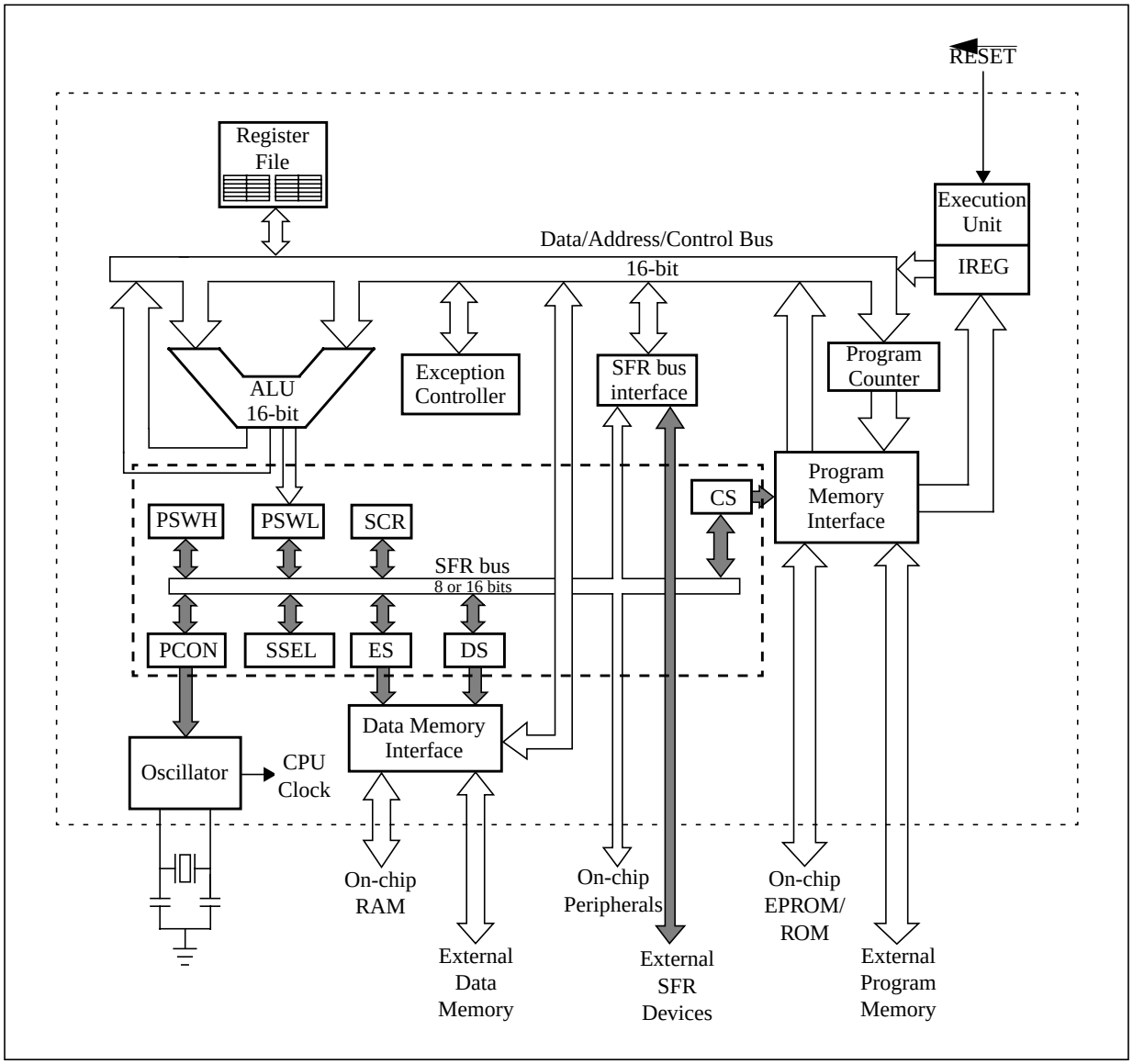


Figure 3.13 XA Core with SFRs highlighted

The SFR address space is 1K bytes (Figure 3.14). The first half of this space (400h through 5FFh) is dedicated to accessing core registers and on-chip peripherals outside the XA core. SFRs

assigned addresses in the range 400h through 43Fh are both byte and bit-addressable. The second half (600h through 7FFh) of the SFR space is reserved for providing access to off-chip SFRs. The off-chip sfr space is provided to allow faster access of off-chip memory mapped I/O devices without having to create a pointer for each access.

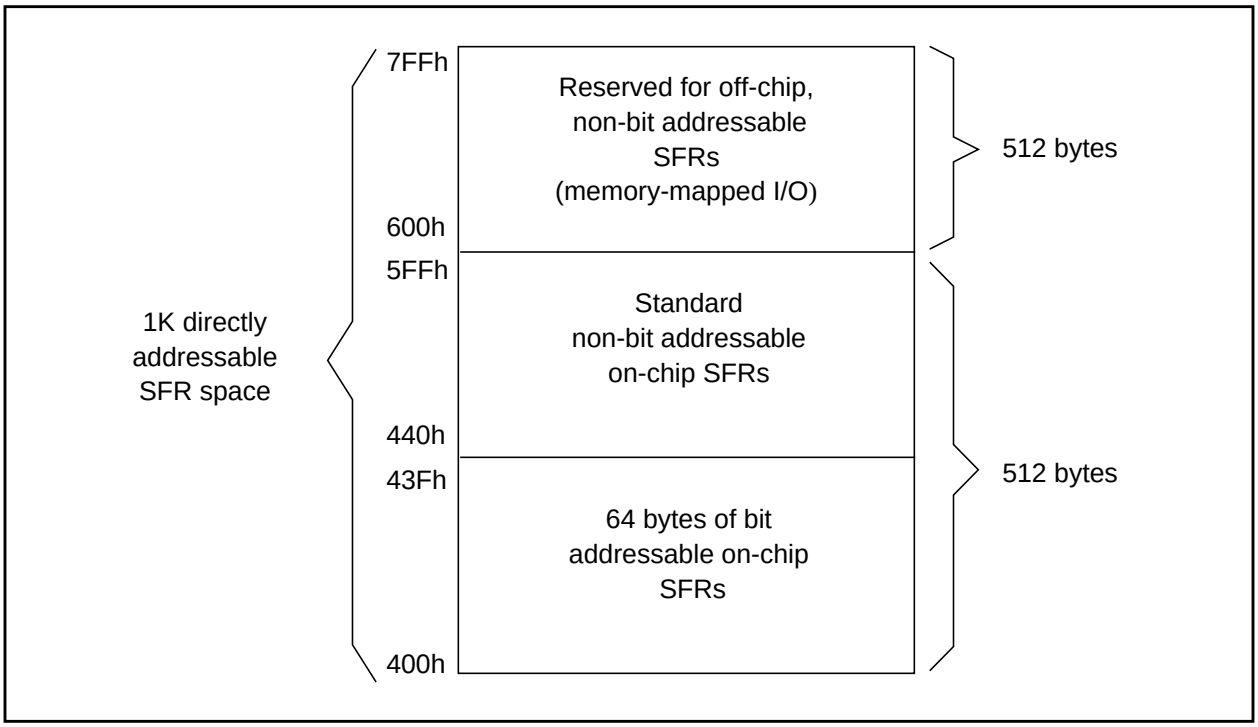


Figure 3.14 SFR address space

Following are some key points to remember when using SFRs:

SFRs should be symbolically addressed. Because SFR assignments may vary from derivative to derivative, it is important to always use symbolic references to SFRs. XA software development tools provide symbolic constants for all SFRs in the form of header/include files and the tools will be updated as new SFRs are added with each added XA derivative.

Verify that your application uses the right header/include files. Although baseline SFRs are likely to retain their addresses in future XA derivatives, this is not guaranteed. SFRs used for optional peripherals may well have different addresses on different derivatives, and the same address on one derivative may access a different peripheral SFR.

Any SFR may be accessed at any time without reference to a pointer or segment. SFR access is independent of any segment register, so SFRs are always accessible with the 10 bit address encoded in instructions accessing SFRs.

SFRs may not be accessed via indirect address. Any time indirection is used, data memory is accessed. If an SFR address is referenced as an indirect address, physical RAM at that address – if it exists– is accessed.

An SFR address is always contained entirely within an instruction. The SFR address is always encoded in the instruction providing the access, and there is no other way of addressing an SFR.

Details of access to external SFRs is determined by derivative implementation. Access to off-chip SFRs is a reserved feature not implemented in the baseline XA. Consult derivative product datasheets for details of external SFR access, e.g., timing.

3.7 Summary of Bit Addressing

Several sections of this chapter have described portions of the XA that are bit-addressable. There are a total of 1024 addressable bits distributed in the XA architecture, chosen to make important data structures immediately accessible via XA bit-processing instructions, specifically, all registers in the register file, R0 through R7 (and R8 through R15 if implemented); directly addressable RAM addresses 20h through 3Fh in the page currently specified by DS, and a portion of the on-chip SFRs. Figure 3.15 summarizes all the bit-addressable portions of the XA.5

bit space			overlaps bytes...		
start		end	type	start	end
0	◄►	0FFh	registers	R0 ◄►	R15
100h	◄►	1FFh	direct RAM	20h ◄►	3Fh
200h	◄►	3FFh	on-chip SFRs	400h ◄►	43Fh

Figure 3.15 Bit addressing summary

4 CPU Organization

This chapter describes the Central Processing Unit (CPU) of the XA Core. The CPU contains all status and control logic for the XA architecture. The XA reset sequence and the system oscillator interface with the CPU, and power control is handled here. The CPU performs interrupt and exception handling. The XA CPU is equipped with special functions to support debugging.

4.1 Introduction

Figure 4.1 is a block diagram of the XA Core.

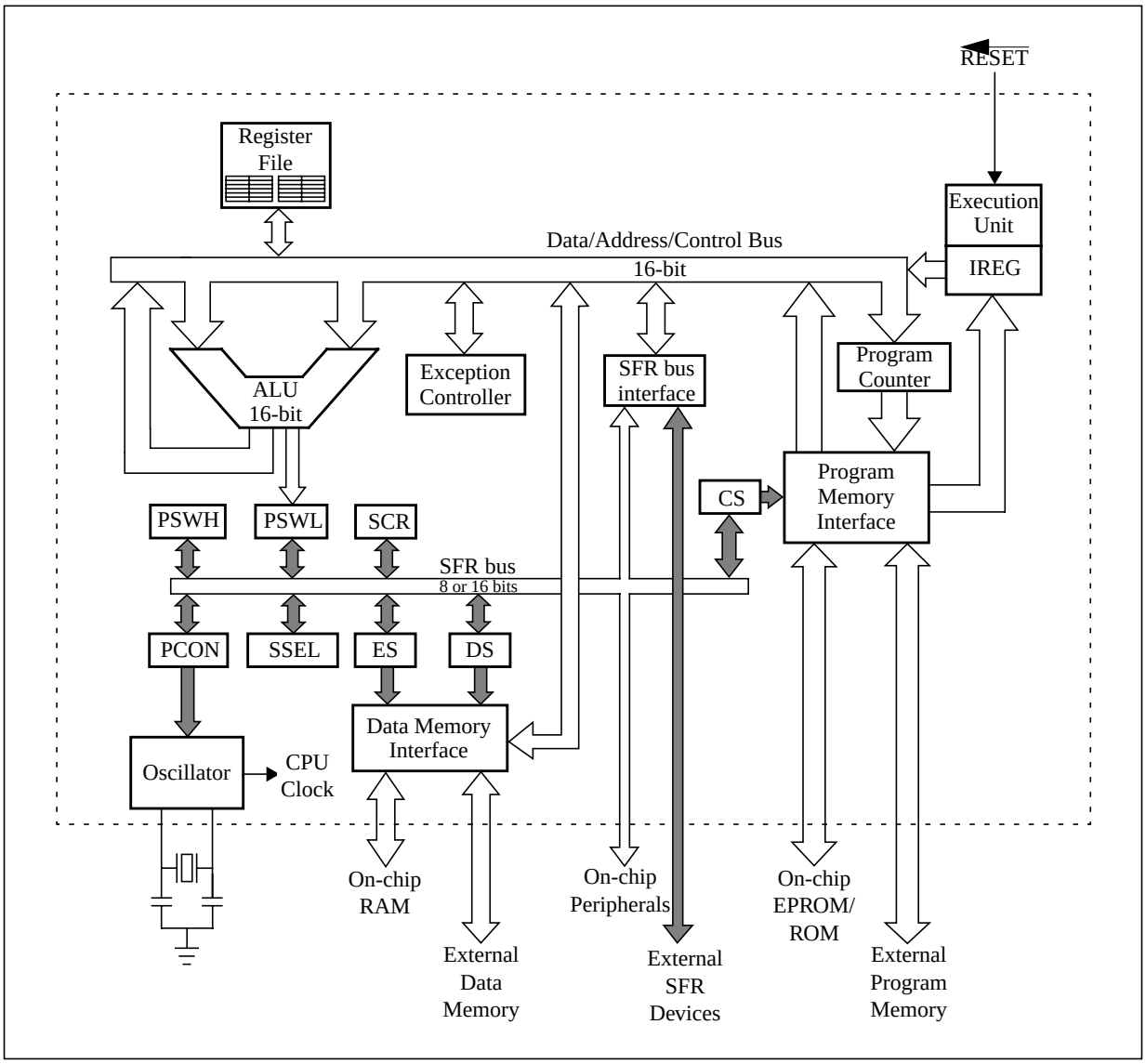


Figure 4.1 The XA Core

Here is an overview of core elements: The XA Core oscillator provides a basic system clock. Timing and control logic are initialized by an external reset signal; once initialized, this logic

provides internal and external timing for program and data memory access. This logic supervises loading the Program Counter and storing instructions fetched by the Program Memory Interface into the Instruction Register. The timing and control logic sequences data transfers to and from the Data Memory Interface. Under the same control, the ALU performs Arithmetic and Logical operations. The ALU stores status information in the low byte of the Program Status Word (**PSWL**). The on-board register file is used for intermediate storage and contains the current value of the Stack Pointer (**SP**). The high byte of the Program Status Word (**PSWH**) chooses between a privileged System Mode and a restricted User Mode; controls a Trace Mode used for single-step debugging, chooses the active register bank, and records the priority of the currently executing process. The System Configuration Register (**SCR**) is initialized to choose native XA mode execution or an 80C51 family compatibility mode. The Segment Selection Register (**SSL**) controls the use of the Code Segment (**CS**), Data Segment (**DS**), and the Extra Segment (**ES)** registers. The XA Core architecture supports interfaces to on- and off-chip RAM, ROM/ EPROM, and Special Function Registers (SFRs).

This chapter describes all these core elements in detail.

4.2 Program Status Word

The Program Status Word (**PSW**) is a two-byte SFR register that is a focal point of XA operations. The least significant byte contains the CPU status flags, which generally reflect the result of each XA instruction execution. This byte is readable and writable by programs running in both User and System modes.

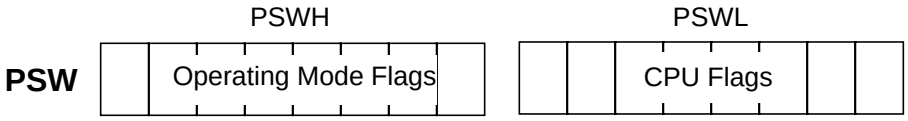


Figure 4.2 XA PSW

The most significant byte of **PSW** is written by programs to set important XA operating modes and parameters: system/user mode, trace mode, register bank select bits, and task execution priority. **PSWH** is readable by any process but only the register select bits may be modified by User mode code. All of the flags may be modified by code running in System Mode.

It should be noted that the XA includes a special SFR that mimics the original 80C51 PSW register. This register, called PSW51, allows complete compatibility with 80C51 code that manipulates bits in the PSW. See Chapter 9 for details of 80C51 compatibility.

4.2.1 CPU Status Flags

The PSW CPU flags (Figure 4.3) signify Carry, Auxiliary Carry, Overflow, Negative, and Zero. Some instructions affect all these flags, others only some of them, and a few XA instructions have no effect on the PSW status flags. In general, these flags are read by programs in order to make logical decisions about program flow. Chapter 6 describes comprehensively how CPU

Status Flags are affected by each instruction type. Consult reference pages in Chapter 6 for details about how individual instructions affect the PSW Status Flags.

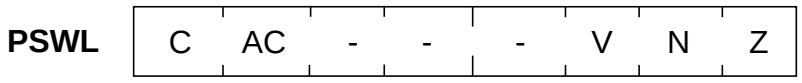


Figure 4.3 PSW CPU status flags

C, the Carry Flag, generally reflects the results of arithmetic and logical operations. It contains the carry out of the most significant bit of an arithmetic operation, if any, for the instructions ADD, ADDC, CMP, CJNE, DA, SUB, and SUBB.The carry flag is also used as an intermediate bit for shift and rotate instructions ASL, ASR, LSR, RLC, and RRC.

The multiply and divide instructions (MUL16, MULU8, MULU16, DIV16, DIV32, DIVU8, DIVU16, and DIVU32) unconditionally clear the carry flag.

AC, the auxiliary carry flag, is updated to reflect the result of arithmetic instructions ADD, ADDC, CMP, SUB, and SUBB with the carry out of the least significant nibble of the ALU. This flag is used primarily to support BCD arithmetic using the decimal adjust instruction (DA).

V is the overflow flag. It is set by an arithmetic overflow condition during signed arithmetic using instructions ADD, ADDC, CMP, NEG, SUB, and SUBB.

V is also set when the result of a divide instruction (DIV16, DIV32, DIVU8, DIVU16, DIVU32) exceeds the size of the specified destination register and when a divide-by-zero has occurred. For multiply instructions (MUL16, MULU8, MULU16) this flag is set when the result of a multiply instruction exceeds the source operand size. In this case “overflow” provides an indication to the program that the result is a larger data type than the source, such as a long integer product resulting from the multiply of two integers).

N reflects the twos complement sign (the high-order or “negative” bit) of the result of arithmetic operations and the value transferred by data moves. This flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Z (“zero”) reflects the value of the result of arithmetic operations and the value transferred by data moves. This flag is set if the result or value is zero, otherwise it is cleared. The flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Other bits (marked with “-” in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.2.2 Operating Mode Flags

The PSW operating mode flags (Figure 4.4) set several aspects of the XA operating mode. All of the flags in the upper byte of the PSW (PSWH) except the bits RS1 and RS0 may be modified only by code running in system mode.

Figure 4.4 PSW operating mode flags

The System Mode bit, **SM**, when asserted, allows the currently running program full System Mode access to all XA registers, instructions, and memories. (For example, most of PSWH can only be modified when **SM** is asserted.) When this bit is cleared, the XA is running in User Mode and some privileges are denied to the currently running program.

The Trace Mode bit, **TM**, when set to 1, enables the built-in XA debugging facilities described in section 4.9. When **TM** is cleared, the XA debugging features are disabled.

The bits **RS1** and **RS0** identify one of the four banks of word registers R0 through R3 as the active register set. The other three banks are not accessible as registers (but also see the Compatibility Mode description in the System Configuration Register section).

The 4 bits **IM3** through **IM0** (Interrupt Mask bits) identify the execution priority of the current executing program. The event interrupt controller compares the setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest priority, or fully interruptible code. The value 15 (or F hexadecimal) indicates the highest priority, not interruptible by event interrupts. Note that priority 15 does not inhibit servicing of exception interrupts or NMI.

The value of the IM bits may be written only by code operating in the system mode. Their value may be read by interrupt handler code to implement software-based interrupt priorities. Note that simply writing a new value to the interrupt mask bits can sometimes cause what is called a priority inversion, that is, the currently executing code may have a lower priority than previously interrupted code. The Software Interrupt mechanism is included on some XA derivatives specifically to avoid priority inversion in complex systems. Refer to the section on Software Interrupts for details.

4.2.3 Program Writes to PSW

The bytes comprising the PSW, namely PSWH and PSWL, are accessible as SFRs, and there is a potential ambiguity when a write to the PSW is performed by an instruction whose execution also modifies one or more PSW bits. The XA resolves this by giving full precedence to explicit writes to the PSW.

For example, executing

```
MOV.b R0L,#81h
```

sets PSW bit **N** to 1, since the byte value transferred is a twos complement negative number. However, executing

```
MOV.b PSWL, #81h
```

will set PSW bits **C** and **Z** and leave bit **N** cleared, since the value explicitly written to PSW takes precedence.

This precedence rule suppresses *all* PSW flag updates. When a value is written to the PSW, for example when executing

```
OR.b PSWH, #30
```

the contents of PSWL are unaffected.

4.2.4 PSW Initialization

As described below, at XA reset, the initial PSW value is loaded from the reset vector located at program memory address 0. Philips recommends that the PSW initialization value in the reset vector sets **IM3** through **IM0** to all 1's so that XA initialization is marked as the highest priority process (and therefore cannot be interrupted except by an exception or NMI). At the conclusion of the initialization code, the execution priority is typically reduced, often to 0, to allow all other tasks to run. It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode.

4.3 System Configuration Register

The System Configuration Register (**SCR**), described in Figure 4.5, sets XA global operating mode. **SCR** is intended to be written once during system start-up and left alone thereafter. Four bits are currently defined:

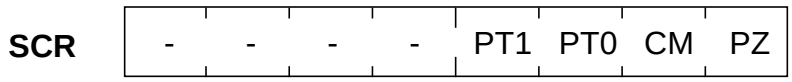


Figure 4.5 System Configuration Register (SCR)

PZ set to 0 (the default) puts the XA in the Large-Memory mode that uses full 24-bit XA addressing. When **PZ** = 1 the XA uses a small-memory "Page 0" mode that uses 16 bit addresses. The intent of Page 0 mode is to save stack space and improve interrupt latency in systems with less than 64K bytes of code and data memory. See the following sections for details.

CM chooses between standard "native" mode XA operation and 80C51 compatibility mode. When 80C51 compatibility mode is enabled, two things happen. First, the bottom 32 bytes of data memory in each data segment are replaced by the four banks of R0 through R3 from the register file. R0L of bank 0 will appear at data address 0, R0H of bank 0 will appear at data address 1, etc. Second, the use of R0 and R1 as indirect pointers is altered. To mimic 80C51 indirect addressing, indirect references to R0 use the byte R0L (zero extended to 16-bits) as the actual pointer value. References to R1 similarly use the byte R0H (zero extended to 16-bits) as the actual pointer value. Note that R0L and R0H on the XA are the same registers as R0 and R1 on the 80C51. No other XA features are altered or affected by compatibility mode. Operation of the XA with compatibility mode off (CM = 0) is reflected in descriptions found in the first 8 chapters of this User Guide. Operation with compatibility mode on (CM = 1) is discussed in Chapter 9.

PT1 and **PT0** select a submultiple of the oscillator clock as a Peripheral Timing clock source, in particular for timers but possibly for other peripherals in XA derivatives. Here are the values for these bits and the resulting clock frequency:

PT1	PT0	Peripheral Clock
0	0	oscillator/4
0	1	oscillator/16
1	0	oscillator/64
1	1	reserved

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.3.1 XA Large-Memory Model Description

When the default XA operation is chosen via the **SCR** (**CM** = 0 and **PZ** = 0), all addresses are maintained by the core as 24 bit values, providing a full 16 MByte address space. On a specific XA derivative, fewer than 24 bits may be available at the external bus interface. All 24 address bits are pushed on the stack during calls and interrupts and 24 bits are popped by RETs and RETIs.

4.3.2 XA Page 0 Memory Model Description

When XA Page 0 mode is chosen, only 16 address bits are maintained by the XA core. This operating mode supports XA applications for which a 64K byte address space is sufficient. The external memory interface port used for the upper 8 address bits, if present, is available for other uses. A single 16-bit word is pushed on the stack during calls and interrupts and 16 bits are, in turn popped by RETs and RETIs. Using Page 0 mode when only a small memory model is needed saves stack space and speeds up address PUSH and POP operations on the stack.

Switching into or out of Page 0 mode after the original initialization is not recommended. First, switching into Page 0 mode can only be done by code running on Page 0, since the code address will be truncated to 16-bits as soon as Page 0 mode takes effect. Instructions already in the XA pre-fetch queue would have been fetched prior to Page 0 mode taking effect. Any addresses that may have been pushed onto the stack previously also become invalid when Page 0 mode is changed. Thus Page 0 mode could not be changed while in an interrupt service routine, or any subroutine.

4.4 Reset

The term “reset” refers specifically to the hardware input required when power is first applied to the XA device, and generally to the sequence of initialization that follows a hardware reset, which may occur at any time. The term also refers to the effect of the RESET instruction (see Chapter 6); in addition, an overflowing Watchdog timer (if this peripheral is present) has an identical effect.

This section describes the XA reset sequence and its implications for user hardware and software.

4.4.1 Reset Sequence Overview

A specific hardware reset sequence must be initiated by external hardware when the XA device is powered-up, before execution of a program may begin. If a proper reset at power up is not done, the XA may fail wholly or in part. The XA reset sequence includes the following sequential components:

- Reset signal generated by external hardware
- Internal Reset Sequence occurs
- $\overline{\text{RST}}$ line goes high
- External bus width and memory configuration determined
- Reset exception interrupt generated
- Startup Code executed

Figure 4.6 illustrates this process.

4.4.2 Power-up Reset

This section describes the reset sequence for powering up an XA device.

The XA $\overline{\text{RST}}$ input must be held low for a minimum reset period after Vdd has been applied to the XA device and has stabilized within specifications. The minimum reset period for a typical system with a reasonably fast power supply ramp-up time is 10 milliseconds. This reset period provides sufficient time for the XA oscillator to start and stabilize and for the CPU to detect the reset condition. At this point, the CPU initiates an internal reset sequence. $\overline{\text{RST}}$ must continue to be low for a sufficient time for the internal reset sequence to complete.

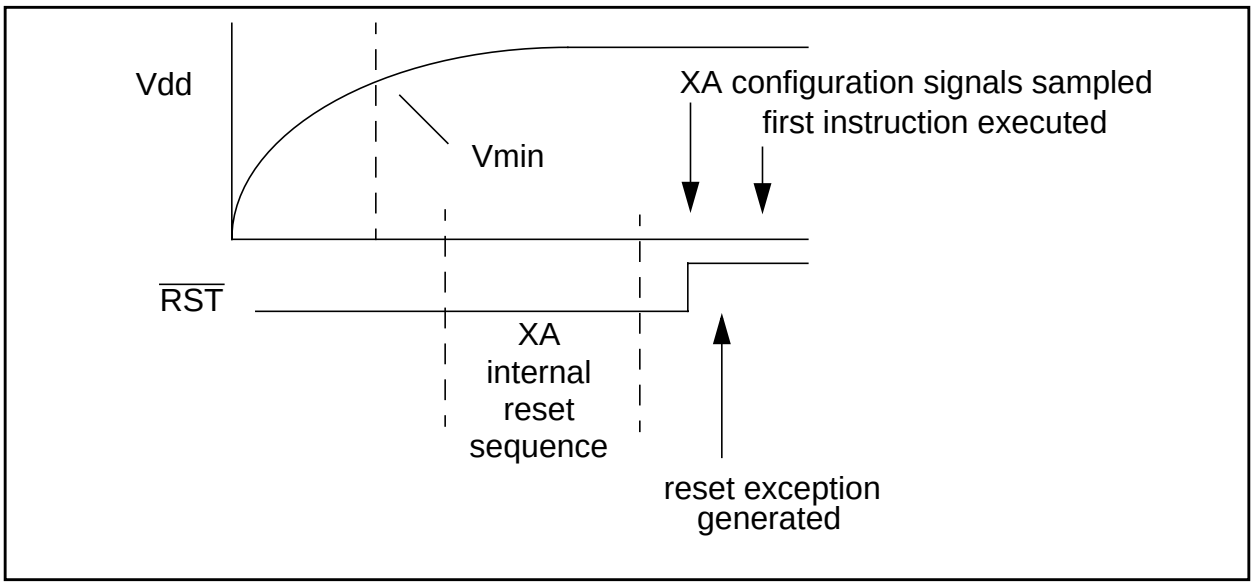


Figure 4.6 XA power-up sequence

4.4.3 Internal Reset Sequence

The XA internal reset sequence occurs after power-up or any time a sufficiently long reset pulse is applied to the $\overline{\text{RST}}$ input while the XA is operating. This sequence requires a minimum of a 10 microseconds (or 10 clocks, whichever is greater) to complete, and $\overline{\text{RST}}$ must remain low for at least this long.

The internal reset sequence does the following:

- Writes a 00 to most core and many peripheral SFRs. Other values are written to some peripheral SFRs. Consult the data sheet of a specific device for details.
- Sets **CS**, **DS**, and **ES** to 0.
- Sets **SSEL** = 0, i.e., sets all accesses through DS.
- Sets all registers in the Register File to 0.
- Sets the user and the system stack pointers (**USP** and **SSP**) to 0100h.
- Clears SCR bit **PZ**, i.e., 24-bit memory addresses will be used by default.
- Clears SCR bit **CM**, i.e., starts execution in XA Native Mode.
- Clears IE bit **EA**, disabling all maskable interrupts.

Note that the internal reset sequence does not initialize internal or external RAM. Note also that the contents of **PSW** at this point is not important, as it will immediately be replaced as described further below.

The effect of the internal reset sequence on components outside the XA core depends on the peripheral complement and configuration of the specific XA derivative. In general, the internal reset sequence has the following effects:

- Sets all port pins to inputs (quasi-bidirectional output configuration with port value = FF hex)
- Clears most SFRs to 0
- Initializes most other SFRs to appropriate non-zero values

Note that serial port buffers, PCA capture registers, and WatchDog feed registers (if present) are unaffected. Consult the XA derivative data sheet for more information.

After the XA internal reset sequence has been completed, the device is quiescent until the $\overline{RST}$ line goes high.

4.4.4 XA Configuration at Reset

As the $\overline{RST}$ line goes high, the value on two input pins is sampled to determine the XA memory and bus configuration. The $\overline{EA}$ and BUSW pins (if present on a specific XA derivative) have special function during the reset sequence, to allow external hardware to determine the use of internal or external program memory, and to select the default external bus width.

Immediately after the $\overline{RST}$ line goes high, the CPU triggers a reset exception interrupt, as described in the next section.

Selecting Internal or External Program Memory

The XA is capable of reading instructions from internal or external memory, both of which may be present. The XA $\overline{EA}$ input pin determines whether internal or external program memory will be used. The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse. If $\overline{EA}$ = 0, the XA will operate out of external program memory, otherwise it will use internal code memory. The selection of external or internal code memory is fixed until the next time $\overline{RST}$ is asserted and released; until then all code fetches will access the selected code memory.

The XA cannot detect inconsistencies between the setting detected on the $\overline{EA}$ input and the hardware memory configuration. For example, setting $\overline{EA}$ = 1 on a ROMless XA variant will cause the XA to attempt to execute internal code memory, which is undefined on a ROMless device, typically resulting in a system failure.

Selecting External Bus Width

The XA is capable of accessing an 8 or 16 bit external data bus. The BUSW pin tells the XA the external data bus configuration. BUSW=0 selects an 8-bit bus and BUSW=1 selects an 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The BUSW pin is sampled on the rising edge of the $\overline{RST}$ pulse. If BUSW is low, the XA operates its external bus interface in 8 bit mode, otherwise, the XA uses 16 bit bus operation. The bus width may also be set under software control on derivatives equipped with the **BCR** (“Bus Configuration Register”) SFR.

After $\overline{RST}$ is released, the BUSW pin may be used an alternate function on some XA derivatives. Consult derivative data sheets for exact pinouts and details of how pins such as these may be shared to keep package size small.

4.4.5 The Reset Exception Interrupt

Immediately after the $\overline{\text{RST}}$ line goes high, the CPU generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the "start-up code") is fetched from the reset vector in code memory at location 0. Here's an example in generalized assembler format of the setup for the Reset Exception:

```
          code_seg            ; establish code segment
          org  0h             ; start at address 0

; reset_vector
          dw   initial_PSW    ; define a word constant
          dw   startup_code   ; define a word constant

          org  120h           ; move to address 120h
                              ; (above vector table)

startup_code:
          ...                 ; put startup code here
```

The initial value of **PSWL** set in the Reset Vector is generally of no special system-wide importance and may be set to zero or some other value to meet special needs of the XA application. The initial **PSWH** value sets the stage for system software initialization and its value requires more attention. Here's an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode    equ  8000h
max_priority   equ  0F00h
initial_PSW    equ  system_mode + max_priority
```

It is generally appropriate to initialize the XA in System Mode so that the start-up code has unrestricted access to the entire architecture. This is done by using a initial value that sets the PSWH bit **SM**.

Philips recommends initializing the execution priority of the start-up code to the highest possible value of 15 (that is, IM0 through IM3 to all ones) so that the start-up code is recognizable as the highest priority process. As described above, the hardware initialization sequence turns off all possible interrupts, so the only potential interrupting process would arise from a non-maskable interrupt (NMI). It is generally a good idea to prevent NMI generation with a hardware lock-out until XA start-up procedures are completed.

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

4.4.6 Startup Code

Philips recommends that the first instruction of start-up code set the value of the System Configuration Register (**SCR**), described in section 4.3, to reflect the system architecture.

The next recommended step is explicitly initializing the stack pointers. The default values (section 4.7) are usually insufficient for application needs.

The start-up code sequence may be concluded by a simple branch or jump to application code. A RETI may not be used at the conclusion of a Reset Exception Interrupt handler (which causes the start-up code to run) because a reset initializes the SP and does not leave an interrupt stack frame.

4.4.7 Reset Interactions with XA Subsystems

The following describes how the reset process interacts with some key subsystems:

- Trace Exception. The trace exception is aborted by an external reset; see section 4.9.
- WatchDog. In XA derivatives equipped with a WatchDog timer feature, an internal reset will be asserted for a derivative-defined number of clocks.
- Resets while in Idle Mode or during normal code execution. Since the XA oscillator is running in Idle Mode, the $\overline{\text{RST}}$ input must be kept low for only 10 microseconds (or 10 clocks, whichever is greater) to achieve a complete reset.
- Resets while in Power-Down Mode. The XA oscillator is stopped in Power-Down mode, so the $\overline{\text{RST}}$ input must be low for at least 10 milliseconds. An exception to this is when an external oscillator is used and the XA is in Power-Down mode. In this case, if the external oscillator is running, a reset during Power-Down mode may be the same as a reset in Idle Mode.

4.4.8 An External Reset Circuit

The $\overline{\text{RST}}$ pin is a high-impedance Schmitt trigger input pin. For applications that have no special start-up requirements, it is practical to generate a reset period known to be much longer than that required by the power supply rise time and by the XA under all foreseeable conditions. One simple way to build a reset circuit is illustrated in Figure 4.7.

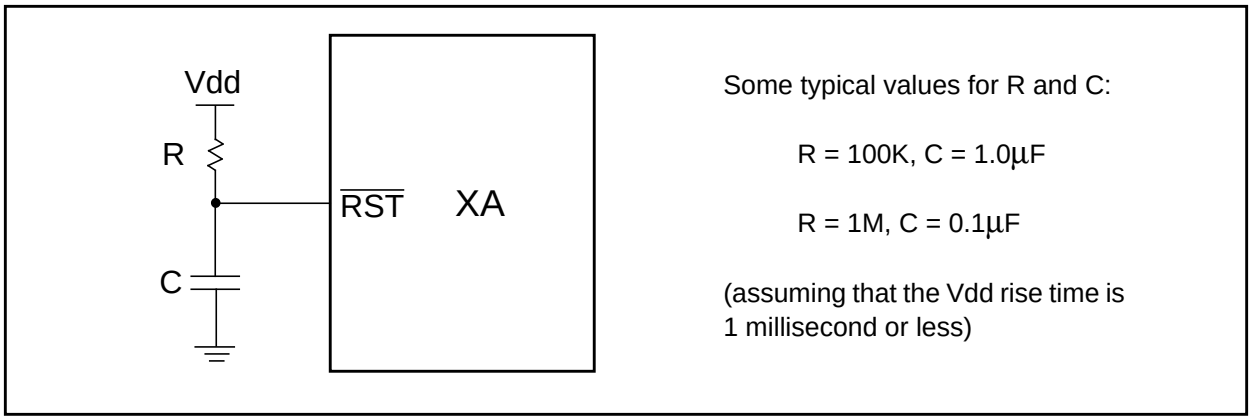


Figure 4.7 An external reset circuit

4.5 Oscillator

The XA contains an on-chip oscillator which may be used as the clock source for the XA CPU, or an external clock source may be used. A quartz crystal or ceramic resonator may be connected as shown in Figure 4.8a to use the internal oscillator. To use an external clock, connect the source to pin XTAL1 and leave pin XTAL2 open, as shown in Figure 4.8b.

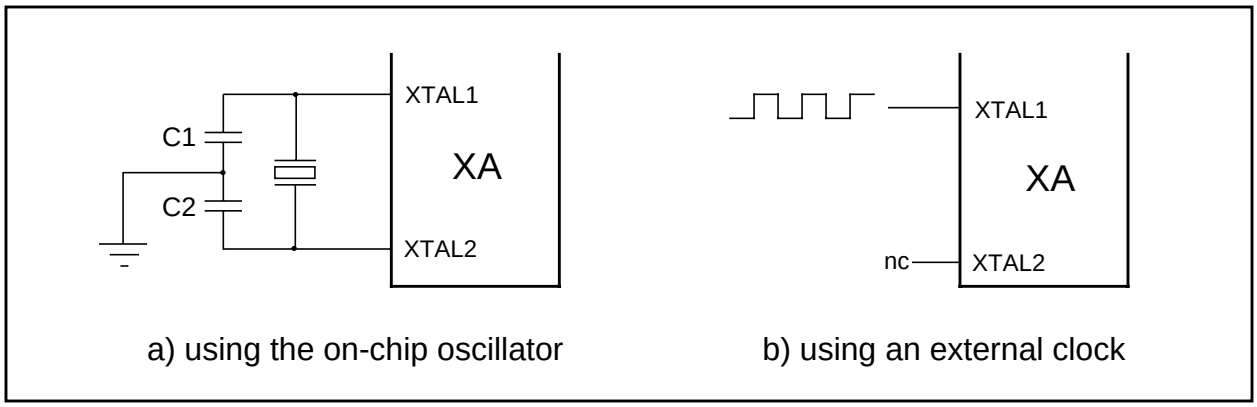


Figure 4.8 XA clock sources

The on-chip oscillator of the XA consists of a single stage linear inverter intended for use as a positive reactance oscillator. In this application, the crystal is operated in its fundamental response mode as an inductive reactance in parallel resonance with capacitance external to the crystal.

A quartz crystal or ceramic resonator is connected between the XTAL1 and XTAL2 pins, capacitors ar connected from both pins to ground. In the case of a quartz crystal, a parallel resonant crystal must be used in order to obtain reliable operation. The capacitor values used in the oscillator circuit should normally be those recommended by the crystal or resonator manufacturer. For crystals, the values may generally be from 18 to 24 pF for frequencies above 25 MHz and 28 to 34 pF for lower frequencies. Too large or too small capacitor values may prevent oscillator start-up or adversely affect oscillator start-up time.

4.6 Power Control

The XA CPU implements two modes of reduced power consumption: Idle mode, for moderate power savings, and Power-Down mode. Power-Down reduces XA consumption to a bare minimum. These modes are initiated by writing SFR **PCON**, as illustrated in Figure 4.9.

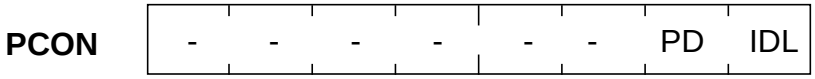


Figure 4.9 PCON

Idle Mode is activated by setting the PCON bit **IDL**. This stops CPU execution while leaving the oscillator and some peripherals running.

Power-Down mode is activated set by setting the PCON bit **PD**. This shuts down the XA entirely, stopping the oscillator.

The reset values of **IDL** and **PD** are 0. If a 1 is written to both bits simultaneously, **PD** takes precedence and the XA goes into Power-Down mode.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.6.1 Idle Mode

Idle mode stops program execution while leaving the oscillator and selected peripherals active. This greatly reduces XA power consumption. Those peripheral functions may cause interrupts (if the interrupt is enabled) that will cause the processor to resume execution where it was stopped.

In the Idle mode, the port pins retains their logical states from their pre-idle mode. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). ALE and $\overline{\text{PSEN}}$ are held in their respective non-asserted states. When Idle is exited normally (via an active interrupt), port values and configurations will remain in their original state.

4.6.2 Power-Down Mode

Power-Down mode stops program execution and shuts down the on-chip oscillator. This stops all XA activity. The contents of internal registers, SFRs and internal RAM are preserved. Further power savings may be gained by reducing XA Vdd to the RAM retention voltage in Power Down mode; see the device data sheet for the applicable Vdd value. The processor may be re-activated by the assertion of $\overline{\text{RST}}$ or by assertion of one of an external interrupt, if enabled. When the processor is re-activated, the oscillator will be restarted and program execution will resume where it left off.

In Power-Down mode, the ALE and $\overline{\text{PSEN}}$ outputs are held in their respective non-asserted states. The port pins output the values held by their respective SFRs. Thus, port pins that are not configured to be part of an external bus retain their state. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). If Power-Down mode is exited via Reset, all port values and configurations will be set to the default Reset state.

In order to use an external interrupt to re-activate the XA while in Power-Down mode, the external interrupt must be enabled and be configured to level sensitive mode. When Power-Down mode is exited via an external interrupt, port values and configurations will remain in their original state. Since the XA oscillator is stopped when the XA leaves Power-Down mode via an interrupt, time must be allowed for the oscillator to re-start. Rather than force the external logic asserting the interrupt to remain active during the oscillator start-up time, the XA implements its own timer to insure proper wake-up. This timer counts 9,892 oscillator clocks before allowing the XA to resume program execution, thus insuring that the oscillator is running and stable at

that time. Once the oscillator counter times out, the XA will execute the interrupt that woke it up, if that interrupt is of a higher priority than the currently executing code.

Note that if an external oscillator is used, power supply current reduction in the Power-Down mode is reduced from what would be obtained when using the XA on-chip oscillator. In this case, full power savings may be gained by turning off the external clock source or stopping it from reaching the XTAL1 pin of the XA. If the clock source may be turned off, it may be advantageous to use Idle mode rather than Power-Down mode, to allow more ways of terminating the power reduction mode and to avoid the 9,892 clock waiting period for exiting Power-Down mode.

4.7 XA Stacks

The XA stacks are word-aligned LIFO data structures that grow downward in data memory, from high to low address. This and some other details of the XA stack implementation differ from 80C51 stack operation. Refer to the chapter on 8051 compatibility for a detailed discussion of this topic.

The XA implements two distinct stacks, one for User Mode and one for System Mode. The User Stack may be placed anywhere in data memory, while the System Stack must be located in the first 64K bytes, i.e., segment 0.

4.7.1 The Stack Pointers

The XA has two stacks, the system stack and the user stack. Each stack has an associated stack pointer, the System Stack Pointer (SSP) and the User Stack Pointer (USP), respectively. Only one of these stacks is active at a given time. The current stack pointer at any instant (which may be the SSP or the USP) appears as word register SP (R7) in the register file; the other stack pointer will not be visible. The value of the PSW bit **SM** determines which stack is active (and whose stack pointer therefore appears as R7). In User Mode (**SM** = 0), SP (R7) contains the User Stack Pointer. In System Mode (**SM** =1), SP (R7) contains the System Stack Pointer. The XA automatically switches SSP and USP values when the operating mode is changed. Note that the terms “USP” and “SSP” are logical terms, denoting the value of SP (R7) in each mode.

Segments and Protection

The User stack is always addressed relative to the current data segment (DS) value. This is consistent with each user task being associated with a specific data segment. Moreover, code running in User Mode cannot modify **DS**, so there is no possibility of changing the segment in which the stack resides within the User context. The System Stack must always be located in segment 0, that is, the first 64K of data memory.

4.7.2 PUSH and POP

The PUSH operation is illustrated by Figure 4.10. The stack pointer always points to an existing data item at the top of the stack, and is decremented by 2 prior to writing data.

The POP operation copies the data at the top of the stack and then adds two to the stack pointer, as follows shown in Figure 4.11.

All stack pushes and pops occur in word multiples. If a byte quantity is pushed on the stack it is stored as the least significant byte of a word and the high byte is left unwritten;
see Figure 4.12. A POP to a byte register removes a word from the stack and the byte register receives the least significant 8 bits of the word, as shown in Figure 4.13.

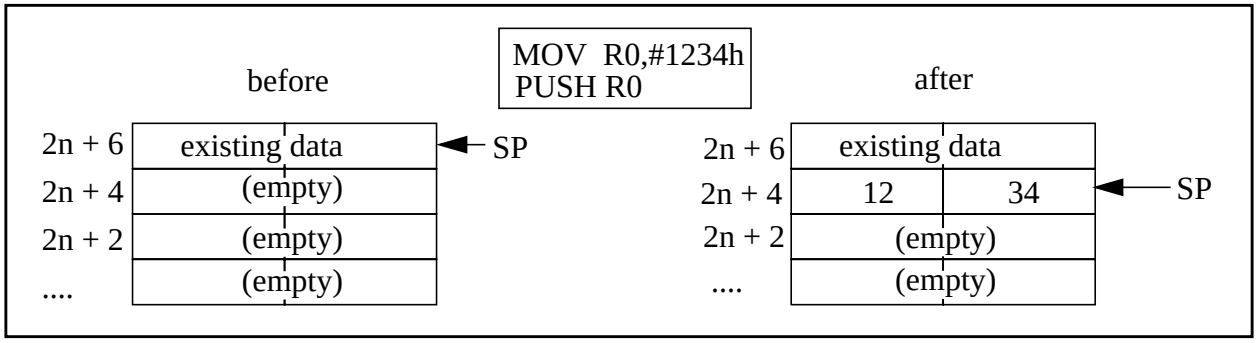


Figure 4.10 PUSH operation

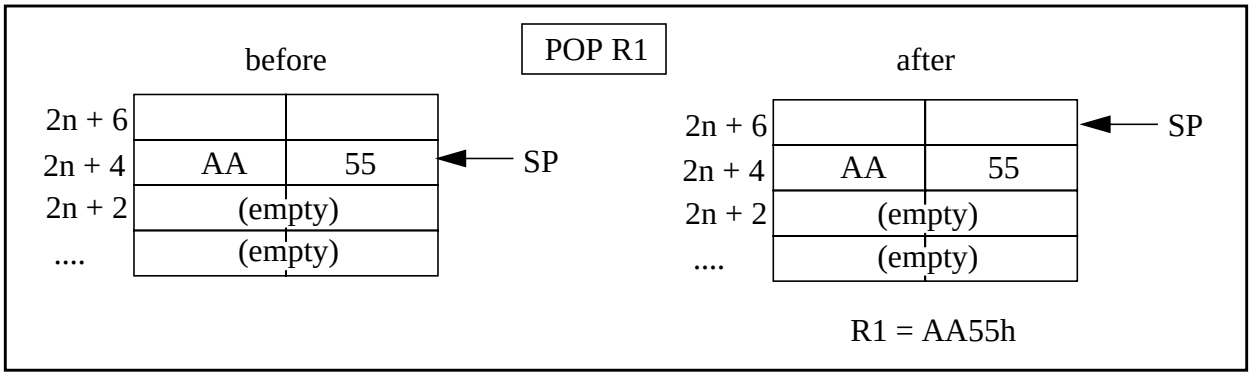


Figure 4.11 POP operation

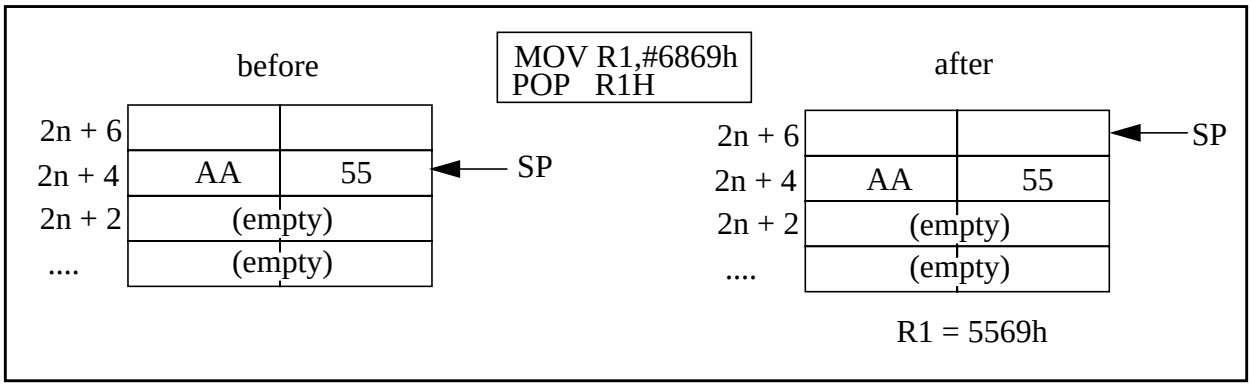


Figure 4.12 POP a byte

The stack should always be word-aligned. If the SP (R7) is modified to an odd value, the offending LSB of the stack pointer is ignored and the word at the next-lower even address is accessed.

Note that neither PUSH or POP operations have any effect on the PSW flags.

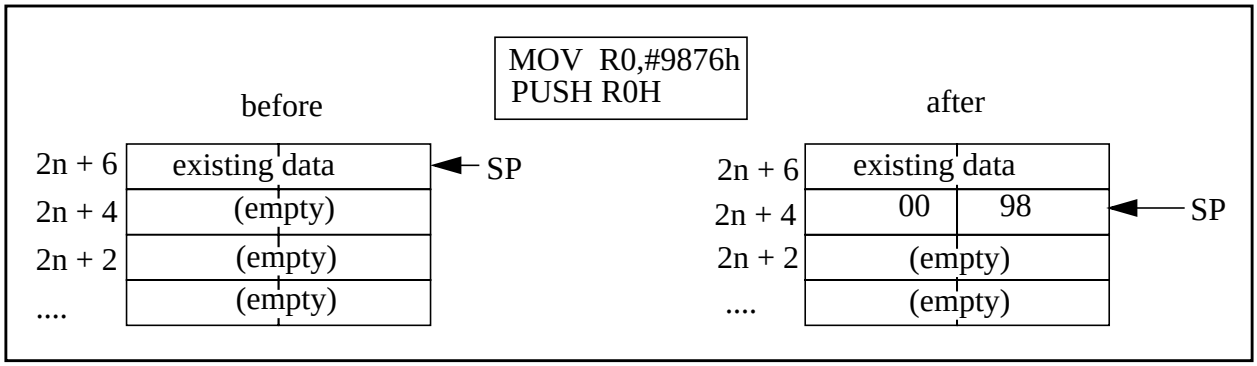


Figure 4.13 PUSH a byte

4.7.3 Stack-Based Addressing

Stack-based data addressing is fully supported by the XA. R0 through R7 may be used in all indexed address modes; the stack pointer in R7 is equally valid as an index.
Figure 4.14 illustrates an example of stack-based addressing. The segment used for stack relative addressing is always the same as for other stack operations (Segment 0 for System mode code and DS for User mode code).

Note that the precautions mentioned in section 3.3.4 apply here: when referencing a word quantity, the final (effective) address must be even, otherwise incorrect data will be accessed. This topic is discussed further in the section Stack Pointer Misalignment.

4.7.4 Stack Errors

Special attention is required to avoid problems due to stack overflow, stack underflow, and stack pointer misalignment

Stack Overflow

Stack overflow occurs when too many items are pushed, either explicitly or as the result of interrupts. As items are pushed on to the stack, it may grow downward past the memory allocated to it. It is not always possible for programs to detect stack overflow, so the XA triggers a Stack Overflow Exception Interrupt whenever the value of the *current* stack pointer (SSP or USP) decrements from 80h to 7Eh (simply setting SP to a value lower than 80h would NOT cause a stack overflow). This value was chosen so that stack space sufficient to handle a stack overflow exception interrupt is always guaranteed, as follows:

The 80h limit leaves 64 bytes available for stack overflow processing. A worst case might be occurs when the Stack Pointer is at 80h and a program executes an 8 word push; this generates a stack overflow. If an NMI occurs at the same time, 3 additional words are pushed. The balance

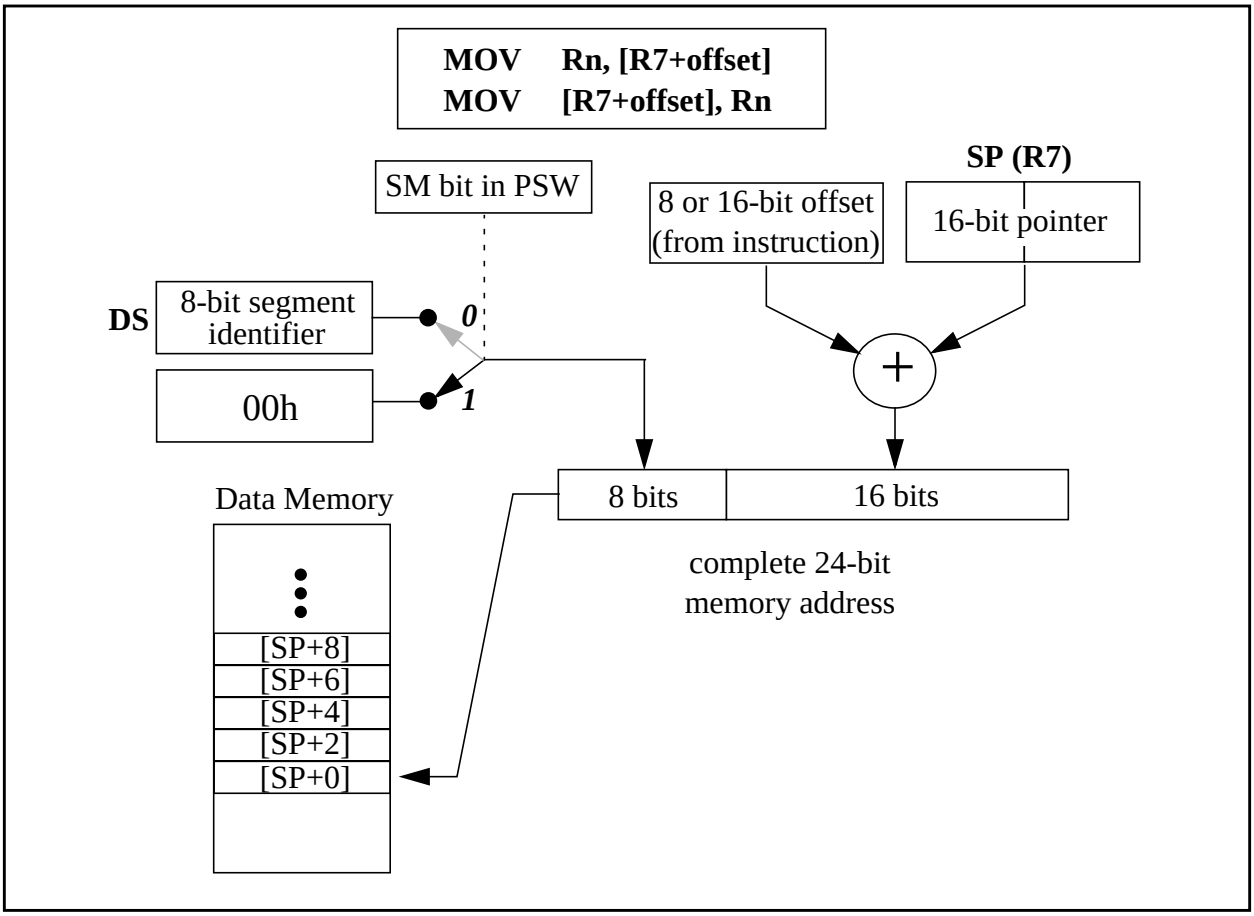


Figure 4.14 Stack-based addressing

of the 64 bytes on the stack is available for handler processing, which should carefully limit further use of the stack.

Stack Underflow

Stack underflow occurs when too many items are popped and the stack pointer value becomes greater than its initial value, i.e., the stack top. The XA does not support stack underflow detection.

Stack Pointer Misalignment

Pointer misalignment occurs when a pointer contains an odd value and is used by an instruction to access a word value in memory. The same situation could occur if some program action forced the stack pointer to an odd value. In these cases, the XA ignores the bottom bit of the pointer and continues with a word memory access.

4.7.5 Stack Initialization

At power-on reset, *both* USP and SSP in all XA derivatives are initialized to 100h. Since SP is pre-decremented, the first PUSH operation will store a word at location FEh and the stack will grow downwards from there.

These default stack pointer start-up values overlap the System and User stacks and are applicable only when one of these stacks will never be used.

Since the System stack is used for all exception and interrupt processing, this may not be appropriate in all XA applications. The startup code should normally set new and different values of both USP and SSP.

4.8 XA Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Event Interrupts
- Software Interrupts
- Trap Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are always processed immediately as they occur, regardless of the priority of currently executing code.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. Event interrupts may be associated with some on-chip device or an external interrupt input. Event interrupts are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are settable by software.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in an SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7.

Trap interrupts are processed as part of the execution of a TRAP instruction. So, the interrupt vector is always taken when the instruction is executed.

All forms of interrupts trigger the same sequence: First, a *stack frame* containing the address of the next instruction and then the current value of the PSW is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, i.e., at the specific interrupt handler.

The new PSW value may include a new setting of PSW bit **SM**, allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3** through **IM0**, reflecting the execution *priority* of the new task. These capabilities are basic to multi-tasking support on the XA. See Chapter 5 for more details.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. Since RETI executed while in User Mode will result in an exception trap, as described further below, interrupt service routines will normally be executed in System Mode.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their execution priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software and trap interrupts occur only when program instructions generating them are executed so there is no need for conflict resolution.

The Non-Maskable Interrupt requires special consideration. It is generated outside the XA core, and in that respect is an event interrupt. However, it shares many characteristics of exception interrupts, since it is not maskable. Note that NMI, while part of the XA CPU core, may not always be connected to a pin or other event source on all XA derivatives.

4.8.1 Interrupt Type Detailed Descriptions

This section describes the four kinds of interrupts in detail.

Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Divide-by-0, Stack overflow, Return from Interrupt (RETI) executed in User Mode, and Trace. Nine additional exception interrupts are reserved. NMI is listed in the table of exception interrupts (Table 4.1) below because NMI is handled by the XA core in same manner as exceptions, and factors into the precedence order of exception processing.

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the priority level of currently executing code, as defined by the IM bits in the PSW. In the unusual case that more than one exception is triggered at the same time, there is a hard-wired *service precedence* ranking. This determines which exception vector is taken first if multiple exceptions occur. In these cases, the exception vector taken *last* may be considered the highest priority, since its code will execute first. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately.

Programmers should be aware of the following when writing exception handlers:

1. Since another exception could interrupt a stack overflow exception handler routine, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the bottom address limit, 80h.

2. The breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Table 4.1: Exception interrupts, vectors, and precedence

Exception Interrupt	Vector Address	Service Precedence
Breakpoint	0004h:0007h	0
Trace	0008h:000Bh	1
Stack Overflow	000Ch:000Fh	2
Divide-by-zero	0010h:0013h	3
User RETI	0014h:0017h	4
<reserved>	0018h - 003Fh	5
NMI	009Ch:009Fh	6
Reset	0000h:0003h	7 (always serviced immediately, aborts other exceptions)

Event Interrupts

Event Interrupts are typically related to on-chip or off-chip peripheral devices and so occur asynchronously with respect to XA core activities. The XA core contains no inherent event interrupt sources, so event interrupts are handled by an interrupt control unit that resides on-chip but outside of the processor core.

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on interrupt input pins. Event interrupts may be globally disabled via the **EA** bit in the Interrupt Enable register (IE) and individually masked by specific bits the IE register or its extension. When an event interrupt for a peripheral device is disabled but the peripheral is not turned off, the peripheral interrupt flag can still be set by the peripheral and an interrupt will occur if the peripheral is re-enabled. An event interrupt that is enabled is serviced when its priority is higher than that of the currently executing code. Each event interrupt is assigned a priority level in the Interrupt Priority register(s). If more than one event interrupt occurs at the same time, the priority setting will determine which one is serviced first. If more than one interrupt is pending at the same level priority, a hardwares precedence scheme is used to choose the first to service. The XA architecture defines 15 interrupt occurrence priorities that may be programmed into the Interrupt Priority registers for Event Interrupts. Note that some XA implementations may not support all 15 levels of occurrence priority. Consult the data sheet for a specific XA derivative for details.

Note that, like all other forms of interrupts, the PSW (including the Interrupt Mask bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the interrupt service routine executes could be different than the priority at which the interrupt occurred (since that was determined not by the PSW image in the vector table, but by the Interrupt Priority register setting for that interrupt). Normally, it is advisable to set the execution priority in the interrupt vector to be the same as the Interrupt Priority register setting that will be used in the program.

Furthermore, the occurrence priority of an interrupt should never be set higher than the execution priority. This could lead to infinite interrupt nesting where the interrupt service routine is re-interrupted immediately upon entry by the same interrupt source.

Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard implementation of the software interrupt mechanism provides 7 interrupts which are associated with 2 Special Function Registers. One SFR, the software interrupt request register (SWR), contains 7 request bits: one for each software interrupt. The second SFR is an enable register (SWE), containing one enable bit matching each software interrupt request bit.

Software interrupts are initiated by setting one of the request bits in the SWR register. If the corresponding enable bit in the SWE register is also set, the software interrupt will occur when it becomes the highest priority pending interrupt and its priority is higher than the current execution level. The software interrupt request bit in SWR must be cleared by software prior to returning from the software interrupt service routine.

Software interrupts have fixed interrupt priorities, one each at priorities 1 through 7. These are shown in Table 4.2 below. Software Interrupts are defined outside the XA core and may not be present on all XA derivatives; consult the specific XA derivative data sheet for details.

Table 4.2: Software interrupts, vectors, and fixed priorities

Software Interrupt	Vector Address	Fixed Priority
SWI1	0100h:0103h	1
SWI2	0104h:0107h	2
SWI3	0108h:010Bh	3
SWI4	010Ch:010Fh	4
SWI5	0110h:0113h	5
SWI6	0114h:0117h	6
SWI7	0118h:011Bh	7

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of event interrupt routines may be executed at a lower priority level than the one

at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or “clean-up” code which does not need to be completed right away. Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called “priority inversion” could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in Figure 4.15.

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR returns, the previously interrupted level 10 ISR is re-activated and eventually competes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

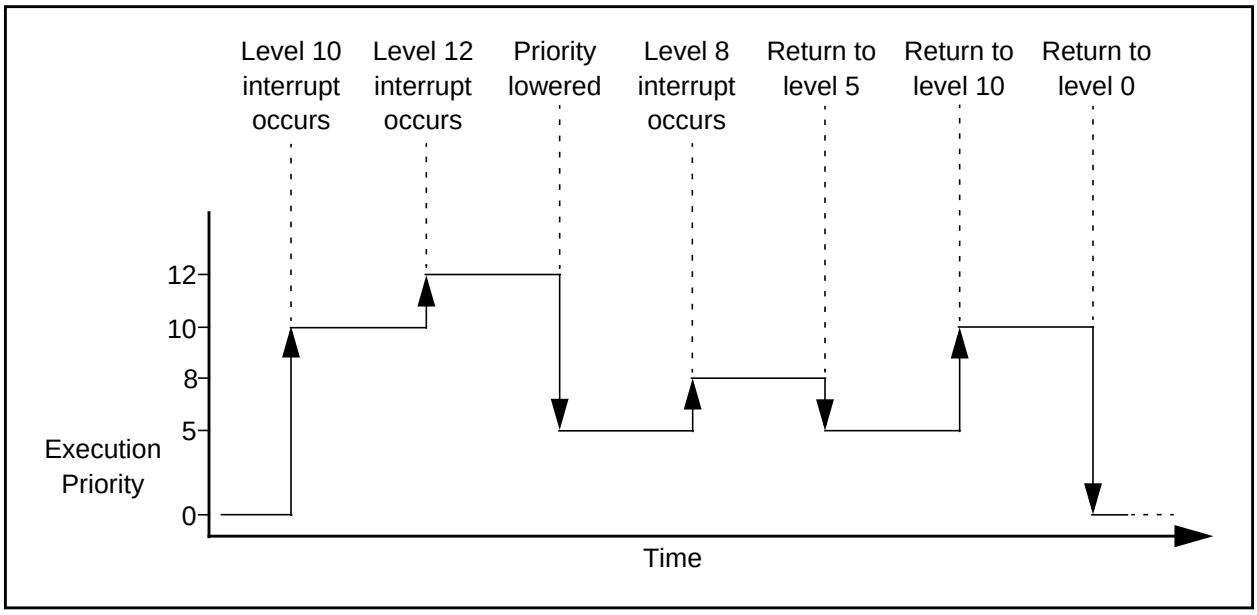


Figure 4.15 Example of priority inversion (see text)

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be

split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for software interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, then returns. the remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in Figure 4.16 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

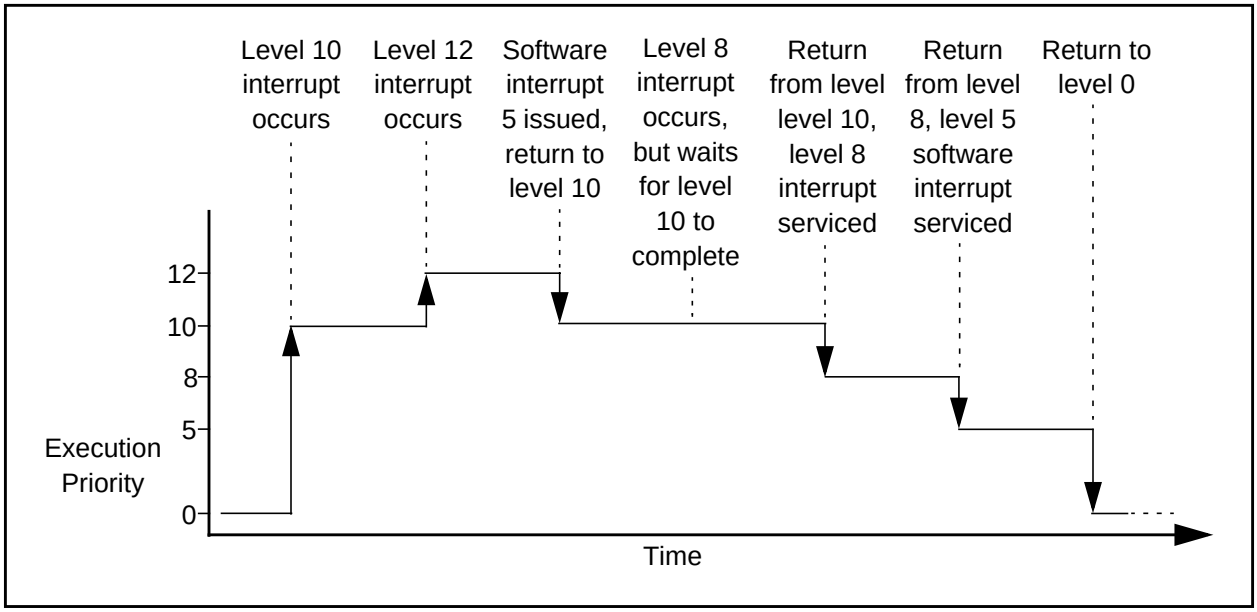


Figure 4.16 Example use of software interrupt (see text)

Trap Interrupts

Trap Interrupts are generated by the TRAP instruction. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap Interrupts are intended to support application-specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between user mode and system mode. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps.

The effect of a TRAP is immediate, the corresponding TRAP service routine is entered upon completion of the TRAP instruction.

See Chapter 6 for a detailed description of the TRAP instruction.

4.8.2 Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning

of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. With one exception, the stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. (The exception is an Exception Interrupt triggered by a Reset event. Since Reset re-initializes the stack pointers, no stack frame is preserved. See section 4.4 for details.) The stack frame in the native 24-bit XA operating mode is illustrated in Figure 4.17. Three words are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current PC, i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad. In sum, a complete 24-bit address is stored in the stack frame. The third word contains a copy of the PSW at the instant the interrupt was serviced.

When the XA is operating in Page 0 Mode (SCR bit **PZ** = 1) the stack frame is smaller because, in this mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is illustrated in Figure 4.18. Obviously it is very important that stack frames of both sizes not be mixed; this is one reason for the admonition in section 4.3 to set the System Configuration Register once during XA initialization and leave it unchanged thereafter.

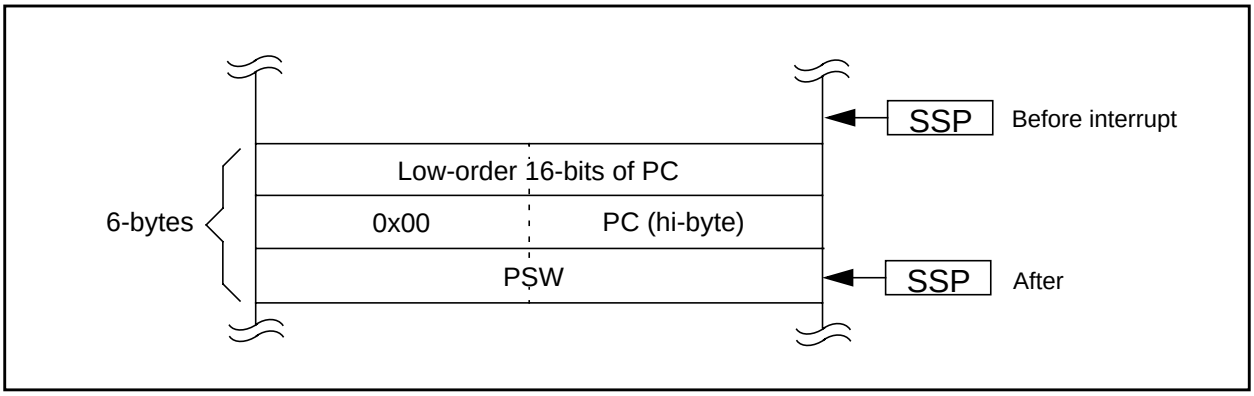


Figure 4.17 Interrupt stack frame (non- page zero mode)

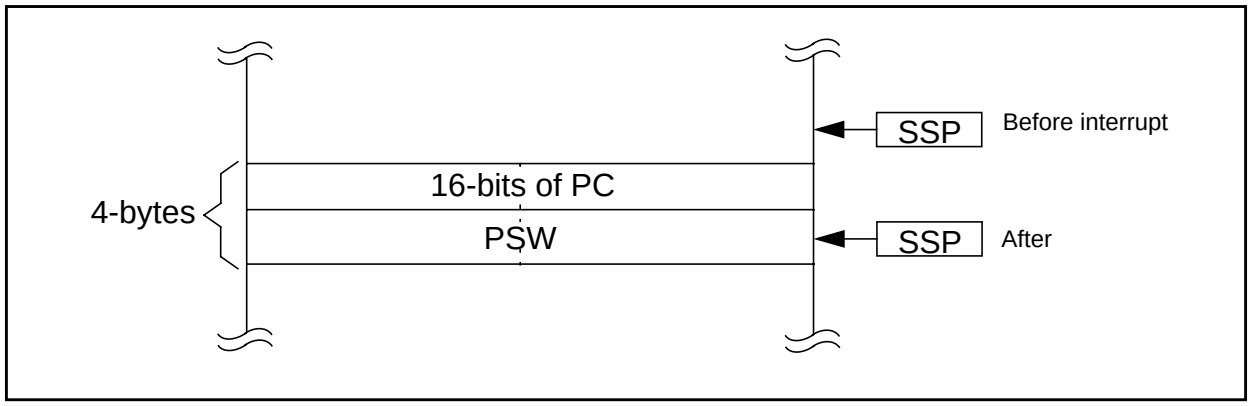


Figure 4.18 Interrupt stack frame (page 0 mode)

Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 through 11B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an interrupt service routine address and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of service routines must be located in the first 64K bytes of XA memory. The first instruction of all service routines must be word-aligned. Key elements of the replacement PSW value are the choice of System or User mode for the service routine, the Register Bank selection, and an Execution Priority setting. For more details on PSW elements, see section 4.2.2.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. Figure 4.19 illustrates the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 6 are assigned to events specific to the XA CPU and 10 are reserved. All 16 Trap Interrupts may be used freely. Assignments of Event Interrupt vectors are derivative-independent and vary with the peripheral device complement and pinout of each XA derivative.

Unused interrupt vectors should normally be set to point to a dummy service routine. The dummy service routine should clear the interrupt flag (if it is not self-clearing) and execute an RETI to return to the user program. This is especially true of the exception interrupts and NMI, since these could conceivably occur in a system where the designer did not expect them. If these vectors are routed to a dummy service routine, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

Note that when using some hardware development tools, it may be preferable not to initialize unused vector locations, allowing the development tool to flag unexpected occurrences of these conditions.

4.9 Trace Mode Debugging

The XA has an optional Trace Mode in which a special trace exception is generated at the conclusion of each instruction. Trace Mode supports user-supplied debugger/monitor programs which can single-step through any code, even code in ROM.

4.9.1 Trace Mode Operation

Trace Mode is initiated by asserting **PSW.TM** in the context of the program to be traced.

Using Trace Mode requires a detailed understanding of the XA instruction execution sequence because when and if a trace exception occurs depends on events within the execution sequence of a single instruction. Figure 4.20 illustrates the XA instruction sequence in overview.

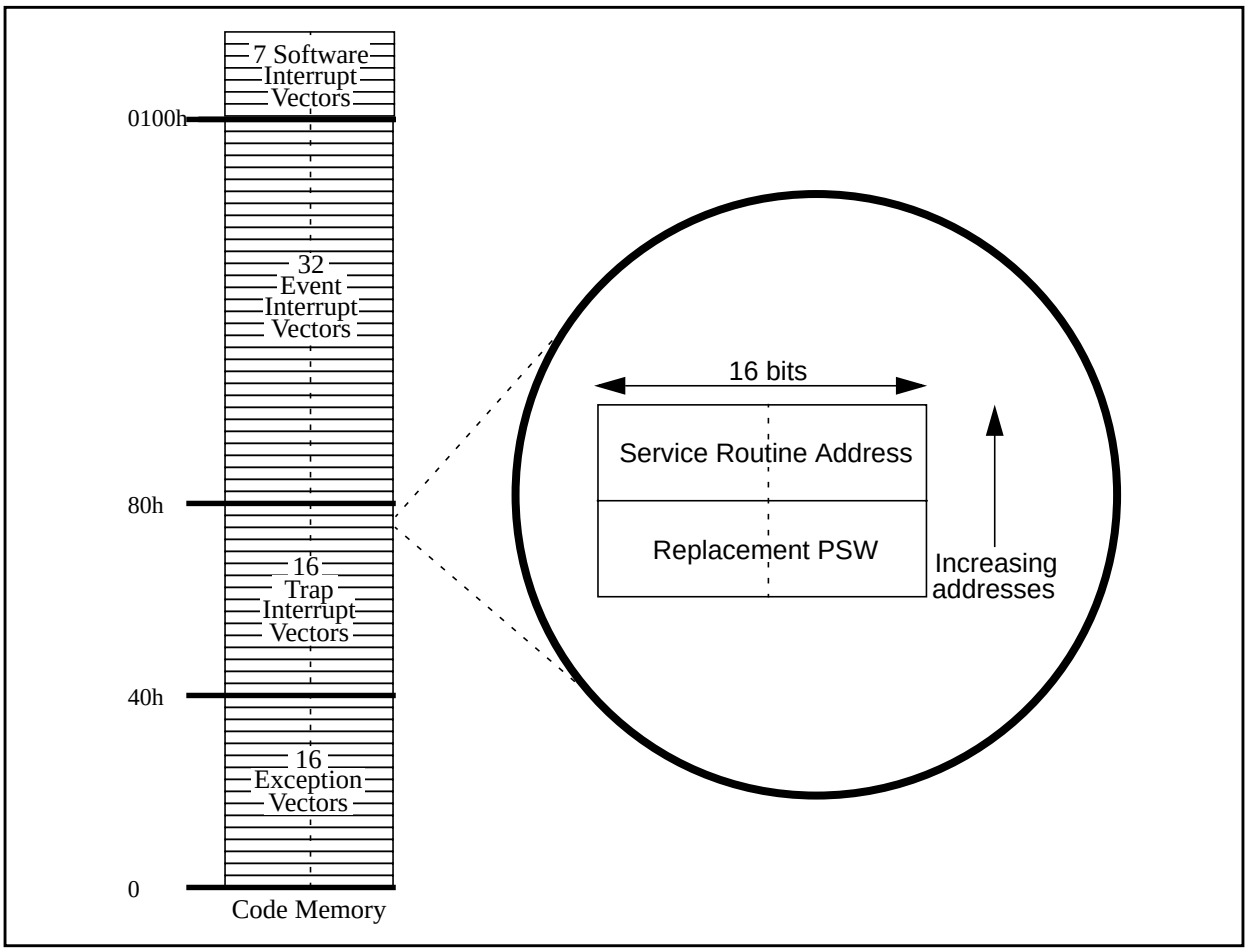


Figure 4.19 Interrupt vectors

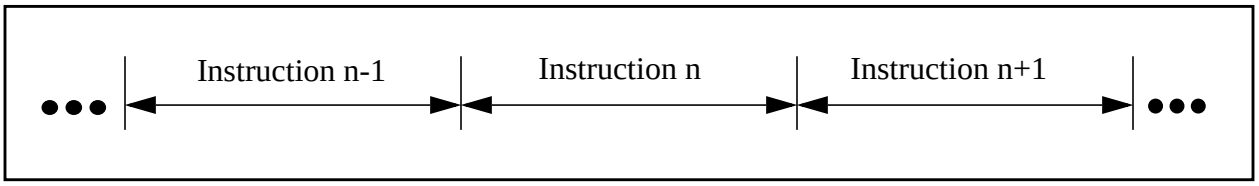


Figure 4.20 XA Instruction Sequence Overview

A detailed model of this sequence is shown in Figure 4.21: First, at the beginning of the instruction cycle, the state of the TM flag is latched. Next, the instruction is checked to see if it is valid; undefined instructions or disallowed operations (like a write through ES in User Mode) are simply not executed, and there is no chance for a trace to occur. The sequence then checks for instructions illegal in the current context (currently only an IRET while in User Mode is detected here) and services an exception if one is found. If, and only if, none of these special conditions occur, the instruction is actually executed. Just after execution, if the Trace Mode bit had been latched TRUE at the beginning of the instruction cycle, the Trace is serviced. Finally, the cycle checks for a pending interrupt and performs interrupt service if one is found
Note that an external reset may occur at any point during the cycle illustrated in Figure 4.21. This will abort processing when it occurs.

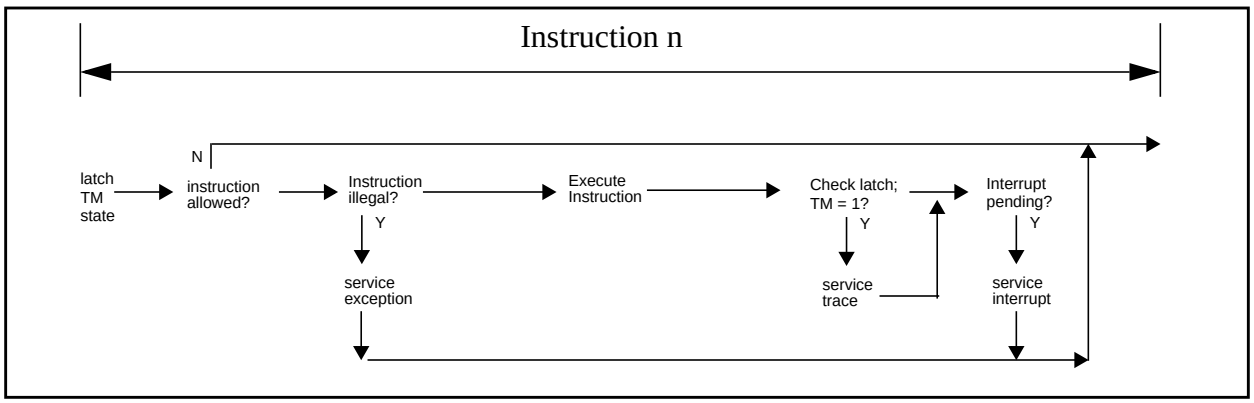
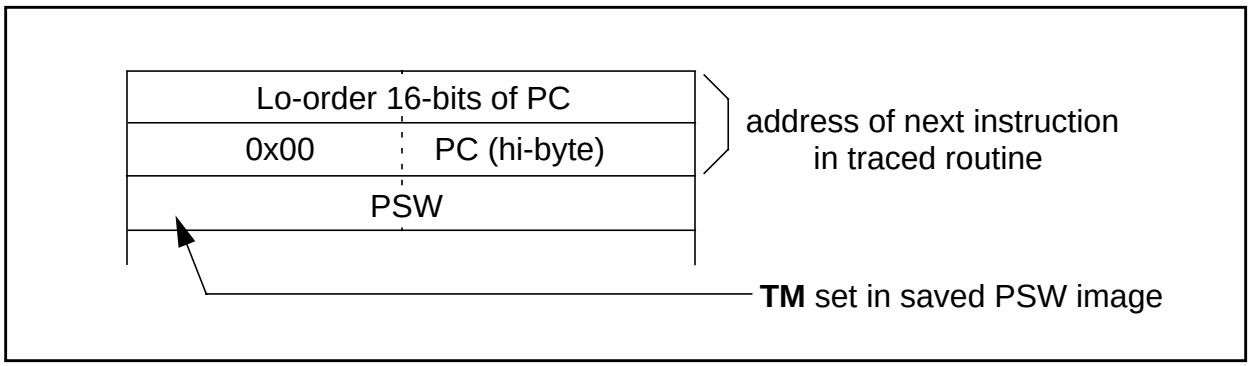


Figure 4.21 Instruction Execution Clock Detail

One consequence of this sequence is that the instruction that sets TM = 1 cannot generate a Trace, since TM is not latched when the instruction is actually executed. Another consequence is that an instruction that generates an exception will never be traced. Finally if an event interrupt occurs during an instruction clock when the instruction being executed is a TRAP, the TRAP will be executed, then the trace service, and finally the interrupt will be serviced.

4.9.2 Trace Mode Initialization and Deactivation

Since **PSW.TM** is in the protected portion of the PSW (i.e., in PSWH), only code executing in System Mode can initiate or turn off Trace Mode. In practice, this may be done by invoking a trap whose replacement PSW clears this bit, or by executing a RETI instruction with a synthetic Exception/Interrupt stack frame explicitly pushed on the top of the System Stack, as follows:



Tracing will continue until the PSW bit **TM** is cleared. This may be done by the trace service routine by examining the stack frame at the top of the system stack and clearing the TM bit prior to returning to the currently traced process. A similar method may be used to initiate trace mode. Note that stack frames generated by exception interrupts are always placed on the System stack. It is probably a good idea for the trace service routine to verify that the item in the stack frame is consistent with the traced process before modifying the TM bit.

5 Real-time Multi-tasking

Multi-tasking as the name suggests, allows tasks, which are pieces of code that do specific duties, to run in an apparently concurrent manner. This means that tasks will seem to all run at the same time, doing many specific jobs simultaneously.

High end applications (like automotive) require instantaneous responses when dealing with high speed events, such as engine management, traction control and adaptive braking system (ABS) and hence there is a trend towards multi-tasking in a wide variety of high performance embedded control applications.

Real-time application programs are often comprised of multiple tasks. Each task manages a specific facet of application program. Building a real-time application from individual tasks allows subdividing a complicated application program into independent and manageable modules. Each task shares the processor with other tasks in the application program according to an assigned priority level.

In real-time multi-tasking, the main concern is the *system overhead*. Switching tasks involve moving lots of data of the terminated and initiated tasks, and extensive book-keeping to be able to restore dormant tasks when required. Thus it is extremely crucial to minimize the system overhead as much as possible. In some cases, some of the tasks may be associated with real-time response, which further complicates the requirements from the system.

The following section analyzes the requirements and the XA suitability to these applications.

5.1 Multi-tasking Support in XA

The XA has numerous provisions to support multi-tasking systems. The architecture provides direct support for the concept of a multi-tasking OS by providing two (System/User) privilege levels for isolation between tasks. High performance, interrupt driven, multi-tasking applications systems requiring protection are feasible with the XA.

The XA architecture offers the following features which will appeal to multi-tasking implementations.

5.1.1 Dual stack approach

The architecture defines a System Stack Pointer (SSP) as well as an User Stack Pointer (USP). The dual stack feature supports fast task switching, and ease the creation of a multi-tasking monitor kernel. The separation of the two offers a reduction is storing and retrieving stack pointers or using a single stack, when switching to the kernel and back to an application. It also serves to speed up interrupt processing in large systems with external data memory. User stacks can be allocated in the slower external memory, while system memory is in internal SRAM, allowing for fast interrupt latency in this environment. The dual stack approach also adds the benefit of a better potential to recover from an ill-behaved task, since the system stack is still intact when an error is sensed.

5.1.2 Register Banks

The XA also supports 4 banks of 8 byte/4 word registers, in addition to 12 shared registers. In some applications, the register banks can be designated statically to tasks, cutting significantly on the overhead for saving and restoring registers on context switching.

5.1.3 Interrupt Latency and Overhead

Interrupt latency is extremely critical in a multitasking environment. For a real-time multitasking environment, a fast interrupt response is crucial for switching between tasks. The XA is designed to provide such fast task switching environment through improved interrupt latency time.

The interrupt service mechanism saves the PC (1 or 2 words, depending on the Page0 mode flag PZ) and the PSW (1 word) on the stack. The interrupt stack normally resides in the internal data memory, and interrupt call including saving of three words takes 23 clocks. Prefetching the service routine takes 3 additional clocks.

When interrupt or an exception/trap occurs, the current instruction in progress always gets executed prior servicing the interrupt. This present an overhead, while increasing the effective interrupt latency, since the event that interrupted the machine cannot be dealt with before the book-keeping is completed. In XA, the longest uninterrupted instruction is the signed 32x16 Divide, which takes 24 clocks.

This puts the worst case interrupt latency at [24 + 23 + 3] = 50 clocks (3.125 microseconds at 16.0 MHz, 2.5 microseconds at 20.0 MHz, and 1.67 microseconds at 30.0 MHz). Saving the state of the lower registers can be done by simply switching the register bank.

In the general case, up to 16 registers would be saved on the stack, which takes 32 clocks. The total latency+overhead at start of an interrupt is a maximum of 68 clocks (4.25 microsecond at 16 MHz, 3.4 at 20 MHz and 2.27 at 30 MHz). This allows for extremely fast context switching for multitasking environments.

5.1.4 Protection

The issue is mentioned here simply to clarify what is and what is not supported by the XA architecture. Dual stack pointer and minor privileges to what looks like a supervisor mode do not mean full protection. It is assumed that code in a microcontroller does not require guarding from intentional system break-in by a lower privilege task. A table of the protected features in XA is given below. Note that features marked “disallowed” are simply not completed if attempted in the User mode. There are no exceptions or flags associated with these occurrences.

Protected Features in the XA

Table 5.1: Segment and Stack Register Protection

Mode	Write to DS	Write through DS	Write to ES	Write through ES	Read through DS	Read through ES	Read through SSP	Write to SSP	Write to SSEL bit 7
System	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
User	Dis-allowed	Allowed	Allowed	Select-able [1]	Allowed	Allowed	Not possible	Not possible	Dis-allowed

Note 1: The MSB of SSEL (bit 7) selects whether write through ES is allowed in User mode. However, this bit is accessible only in System mode.

Table 5.2: PSW bit protection

Mode	Write to SM bit	Write to RS0:1 bits	Write to TM bit	Write to IM0:3 bits
System	Allowed	Allowed	Allowed	Allowed
User	Disallowed	Allowed	Disallowed	Disallowed

In addition to the above, the System Stack is protected from corruption by User Mode execution of the RETI instruction. If User Mode code attempts to execute that instruction, it causes an exception interrupt. If it is necessary to run TRAP routines, for instance, in User Mode, the User RETI exception handler can perform the return for the User Mode code. To accomplish this, the User RETI exception handler may pop the topmost return address from the stack (2 or 3 words, depending on whether the XA is in Page Zero mode) and then execute the RETI.

Protection Via Data Memory Segmentation

In User/Application mode, each task is protected from all others via the separation of data spaces (unless explicit sharing is planned in advance). If the address spaces of two tasks include no shared data, one task cannot affect the data of another, but it can read any data in the full address space. Code sharing is always safe since code memory may never be written[1]. An application mode program is prohibited from writing the segment registers, thus confining the writable area per an ill-behaved task to its dedicated segment. Most applications, which are not expected to utilize multi-tasking or use external memory, do not require any protection. They will remain after reset in system mode, and could access all system resources.

At any given instant, two segments of memory are immediately accessible to an executing XA program. These are the data segment DS, where the stack and local variables reside, and the extra segment ES, which may be used to read remote data structures. Restricting the addressability of task modules helps gain complete control of system resources for efficient, reliable operation in a multi-tasking environment.

---

1. True for non-writable code memory only like EPROM, ROM, OTP. This might change for FLASH parts.

Protection Via Dual Stack Pointers

The XA provides a two-level user/supervisor protection mechanism. These are the *user* or *application* mode and the *system* or *supervisor* mode. In a multitasking environment, tasks in a supervisor level are protected from tasks in the application level.

The XA has two stack pointers (in the register file) called the System Stack Pointer (SSP) and the User Stack Pointer (USP). In multitasking systems one stack pointer is used for the supervisory system and another for the currently active task. This helps in the protection mechanism by providing isolation of system software from user applications. The two stack pointers also help to improve the performance of interrupts. If the stack for a particular application would exceed the space available in the on-chip RAM, or on-chip RAM is needed for other time critical purposes (since on-chip RAM is accessed more quickly than off-chip memory), the main stack can be put off-chip and the interrupt stack (using the System SP) may be put in on-chip RAM.

These features of the XA place it well above the competition in suitability to multi-tasking applications.

6 Instruction Set and Addressing

This section contains information about the addressing modes and data types used in the XA. The intent is to help the user become familiar with the programming capabilities of the processor.

6.1 Addressing Modes

Addressing modes are ways to form effective addresses of the operands. The XA provides seven *basic* powerful addressing modes for access on word, byte, and bit data, or to specify the target address of a branch instruction. These *basic* addressing modes are uniformly available on a large number of instructions. Table 6.1 includes the basic addressing modes in the XA. An instruction could use a combination of these basic addressing modes, e.g., ADD R0, #020 is a combination of Register and Immediate addressing modes.

All modes (non-register) generate ADDR[15:0]. This address is combined with DS/ES[23:16] for data and PC/CS[23:16] for code to form a 24-bit address[1].

An XA instruction can have zero, one, two, or three operands, whose locations are defined by the addressing mode. A *destination* operand is one that is replaced by a result, or is in some way affected by the instruction. The destination operand is listed first in an addressing mode expression. A *source* operand is a value that is moved or manipulated by the instruction, but is not altered. The source is listed second in an addressing mode expression.

Table 6.1 Basic Addressing Modes

MODE	MNEMONIC	OPERANDS
Register	R	operand(s) in register (in Register file)
Indirect	[R]	Byte/Word whose 16-bit address is in R
Indirect-Offset	[R+off 8/16]	Byte or Word data whose address (16-bit) contained in R, is offset by 8/16-bit signed integer “off 8/16’
Direct	mem_addr	Byte/Word at given memory “mem_addr’
SFR [1]	sfr_addr	Byte/Word at “sfr_addr’ address
Immediate	#data 4/5 #data 8/16	Immediate 4/5 and 8/16-bit integer constants “data8/16”
Bit	bit	10-bit address field specifying Register File, Data Memory or SFR bit address space

1. This is a special case of direct addressing mode but separately identified, as SFR space is separate from data memory.

1. Exception is Page 0 mode, where all addresses are 16-bit.

6.2 Description of the Modes

6.2.1 Register Addressing

Instructions using this addressing mode contain a field that addresses the Register File that contains an operand. The Register file is byte[2], word, double-word or bit addressable.

Example: ADD R6, R4

Before: R4 contains 005Ah
R6 contains A5A5h

After: R4 contains 005Ah
R6 contains A5FFh

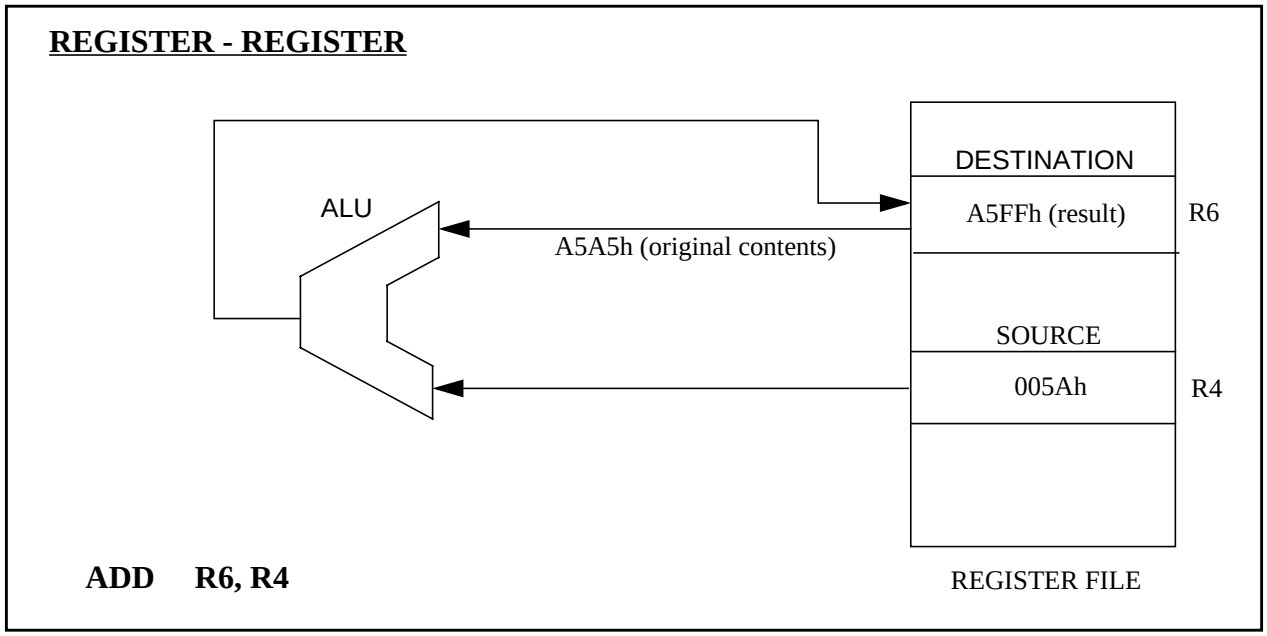


Figure 6.1

2. The unimplemented 8 word registers are not Byte addressable

6.2.2 Indirect Addressing

Instructions using this addressing mode contain a 16-bit address field. This field is contained in 1 out of 8 pointer registers in the Register File (that contain the 16-bit address of the operand in any 64K data segment). For data, the segment is identified by the 8-bit contents of DS or the ES and for code by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL corresponding to the indirect register number. The address of the pointer word for word operands should be even

Example:	ADD R6, [R4]	Before**:**	R6 contains 1005h
	SSEL.4 = 1		R4 contains A000h
	i.e., the operand is in		Word at A000h contains A5A5h
	segment determined		
	by the contents of ES	After:	R4 contains A000h
	So, if ES = 08, the		R6 contains B5AAh
	operand is in		Word at A000h in segment 8
	segment 8 of data memory.		of data memory contains A5A5h

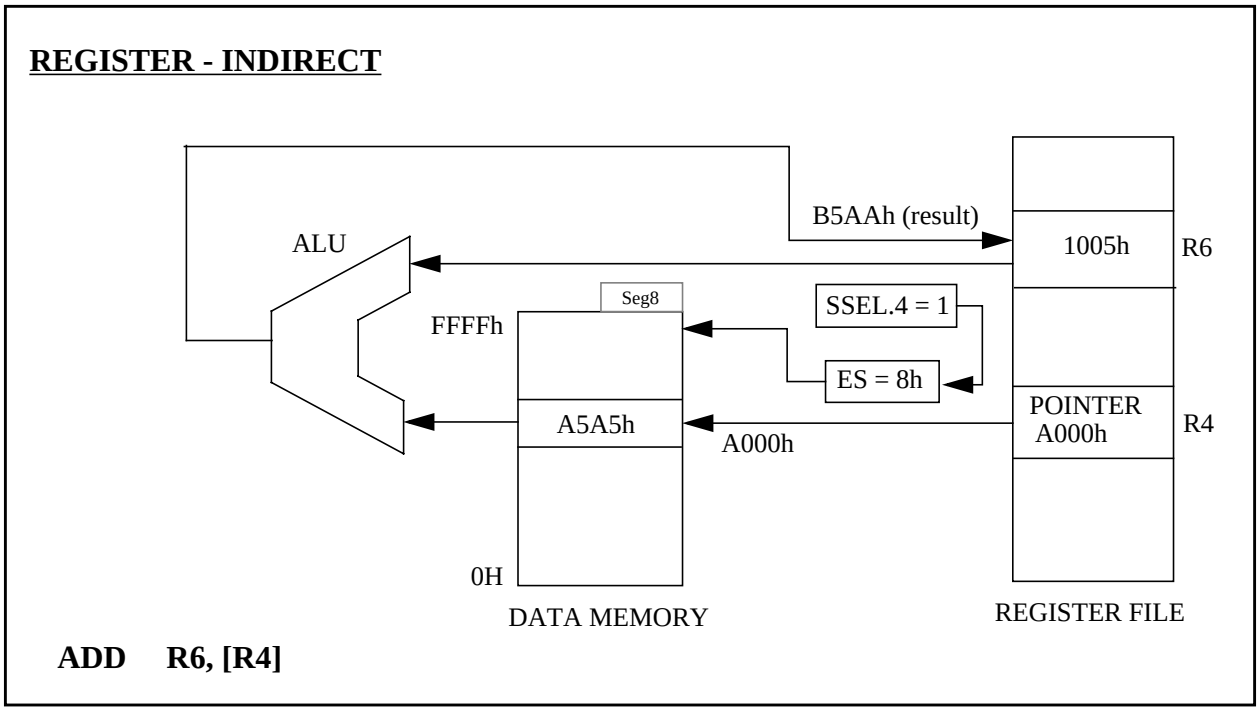


Figure 6.2

6.2.3 Indirect-Offset Addressing

This addressing mode is just like the Register-Indirect addressing mode above except that an additional displacement value is added to obtain the final effective address. Instructions using this addressing mode contain a 16-bit address field and an 8 or 16-bit signed displacement field. This field addresses 1 out of 8 pointer registers in the Register File that contains the 16-bit address of the operand in any 64K data segment. The contents of the pointer register are added to the signed displacement to obtain the effective address[3] (which *must* be even) of the operand. For data the segment is identified by the 8-bit contents of DS or the ES and for code, by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL.

Example:	ADD R5, [R3 +30h]	Before:	R3 contains C000h
	SSEL.3 = 1		R5 contains 0065h
	i.e., the operand is in		Word at C030h = A540h
	segment determined		
	by the contents of ES	After:	R3 contains C000h
	So, if ES = 04, the		R5 contains A5A5h
	operand is in segment		Word at C030h = A540h
	4 of data memory.		

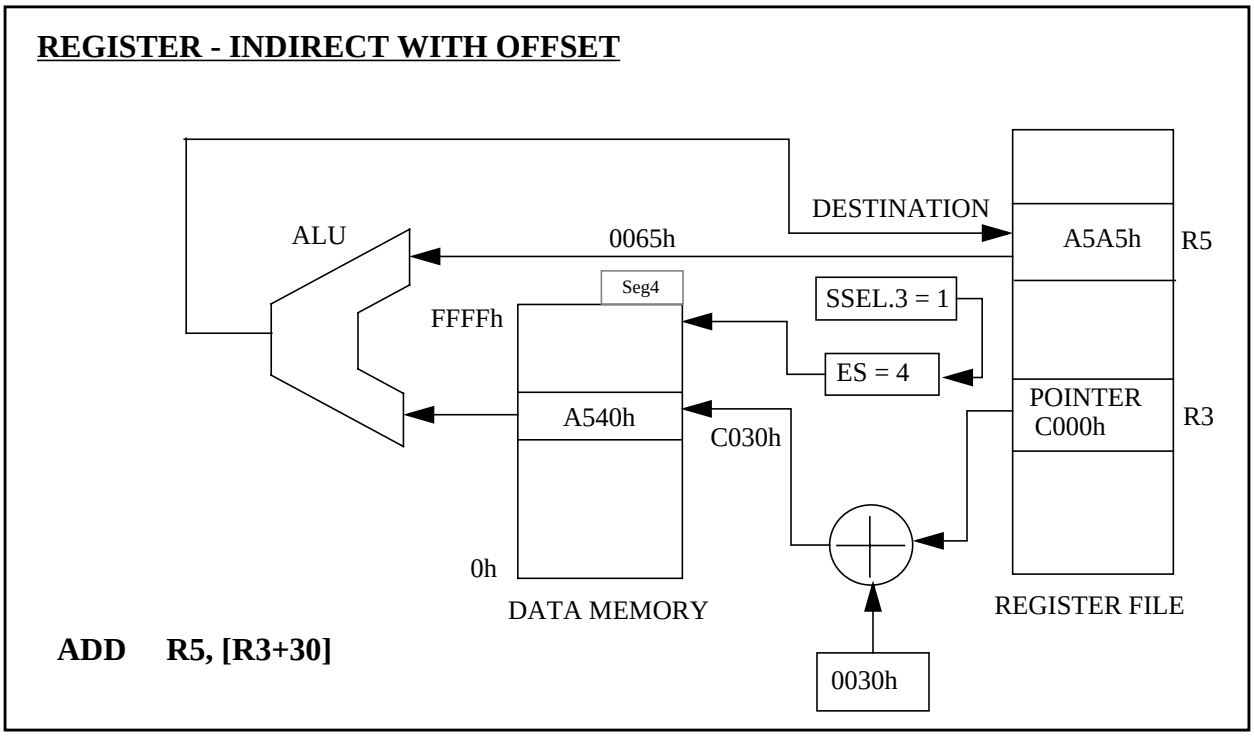


Figure 6.3

3. In case of an odd address, the XA forces the operand fetch from the next lower even boundary (address.bit0 = 0)

6.2.4 Direct Addressing

Instructions using this addressing mode contain an 10-bit address field, which contains the actual address of the operand in any 64K data memory segment or sfr space.The direct address data memory space is always the bottom 1K byte (0:3FFh) of any segment. The associated data segment is always identified by the 8-bit contents of DS.

Example: SUB R0, 200h
If DS = 02, the operand is in segment 2 of data memory.

Before: R0 contains A5FFh
200H contains 5555h

After: R0 contains 50AAh
200h contains 5555h

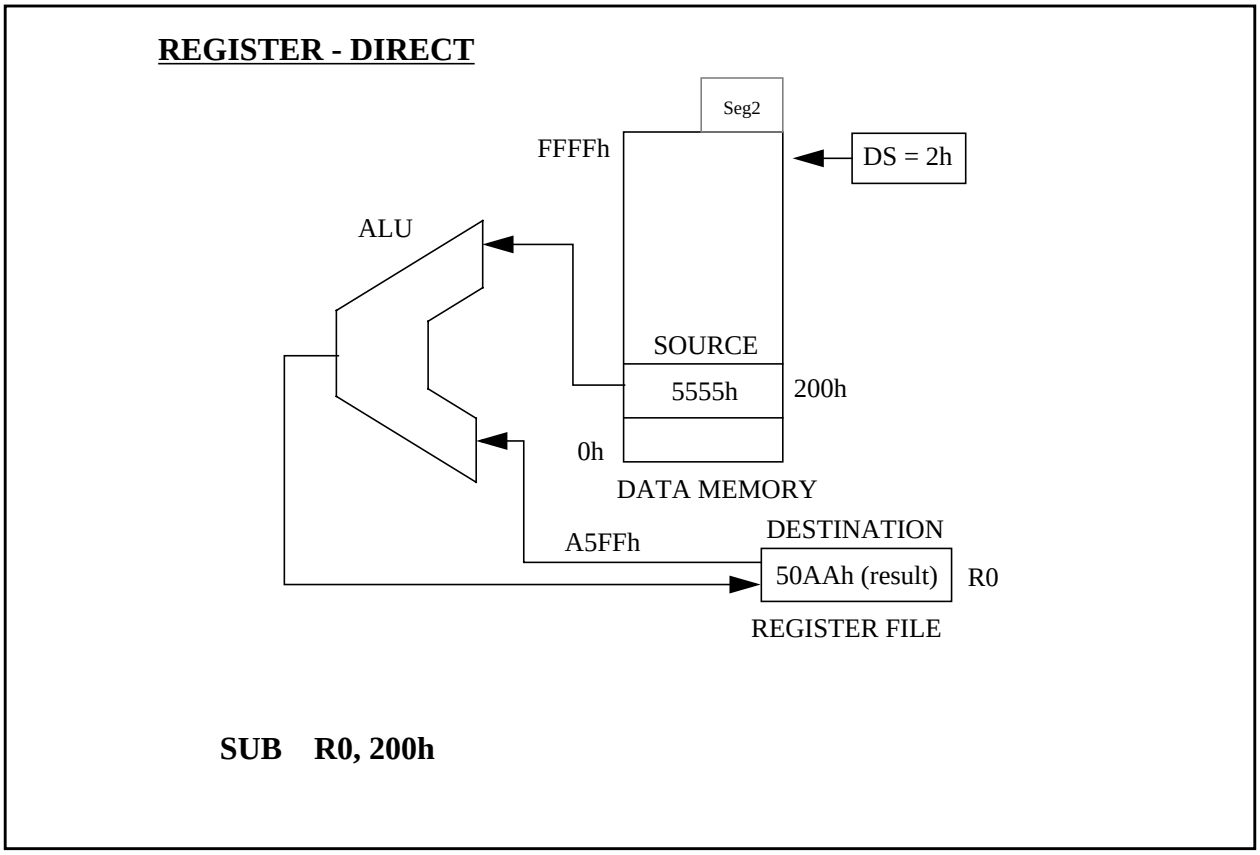


Figure 6.4

6.2.5 SFR Addressing

This is identical to the direct addressing mode described before, except it addresses the 1K SFR space. Although encoded into the same instruction field as the direct addressing described above, this is actually a separate space. Instructions using this addressing mode contain an 10-bit SFR address. The 1K SFR space is always directly addressed (400:7FFh) and is mapped directly above the 1K direct-addressed RAM space.

Example:	MOV R0H, 406h[4]	Before:	R0H contains 05h 406h contains A5h
		After:	R0H contains A5h 406h contains A5h

6.2.6 Immediate Addressing

In immediate addressing, the actual operand is given explicitly in the instruction.The immediate operand is either an 4/5, 8 or 16-bit integer which constitutes the source operand. 4-bit short immediate operands used with instructions ADDS and MOVS are sign extended.

Example:	ADD R0L,#0B9h	Before:	R0 contains 13h
		After:	R0L contains CCh

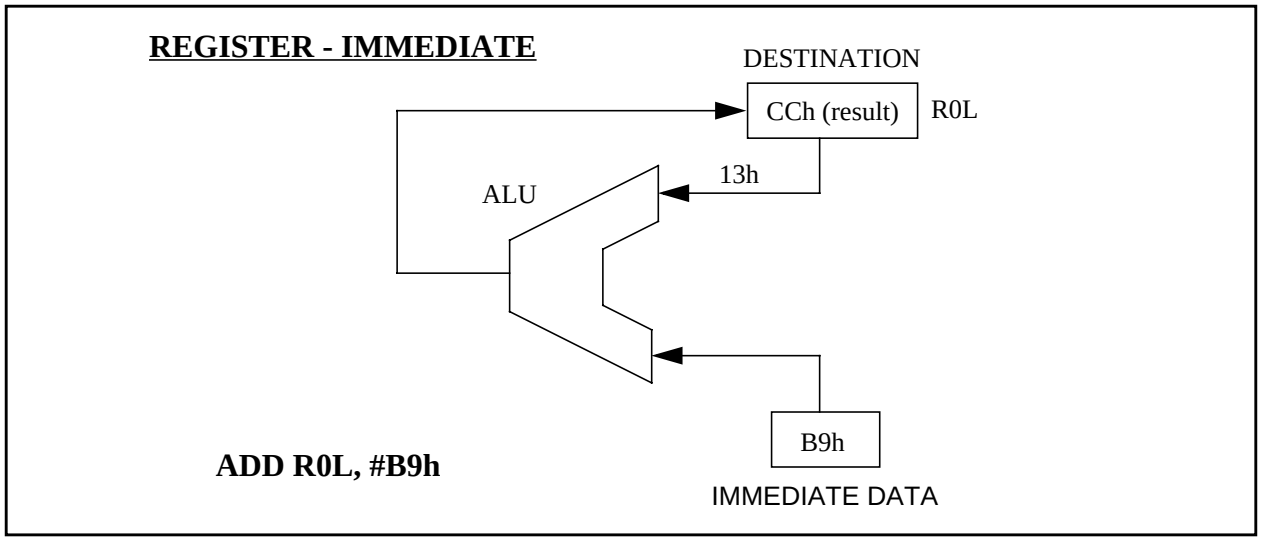


Figure 6.5

4. The syntax always refers to the SFR address starting from the base address of 400H.

6.2.7 Bit Addressing

Instructions using the bit addressing mode contain a 10-bit field containing the address of the bit operand. The XA supports three bit address spaces, which are encoded into the same format. The spaces are: 256 bits in the register file (the entire active register file); 256 bits in the data memory (byte addresses 20 through 3F hex on the current data segment); and 512 bits in the SFR space (byte addresses 400 through 43F hex).

Bit addresses 0 to FF hex map to the register file, bit addresses 100 to 1FF hex map to data memory, and bit addresses 200 to 3FF map to the SFR space.

A separate bit-addressable space (20-3F hex) in the direct-address data memory, exists for each segment. The current working segment for the direct-address space being always identified by the DS register.

The encoding of the 10-bit field for bit addresses is as follows:

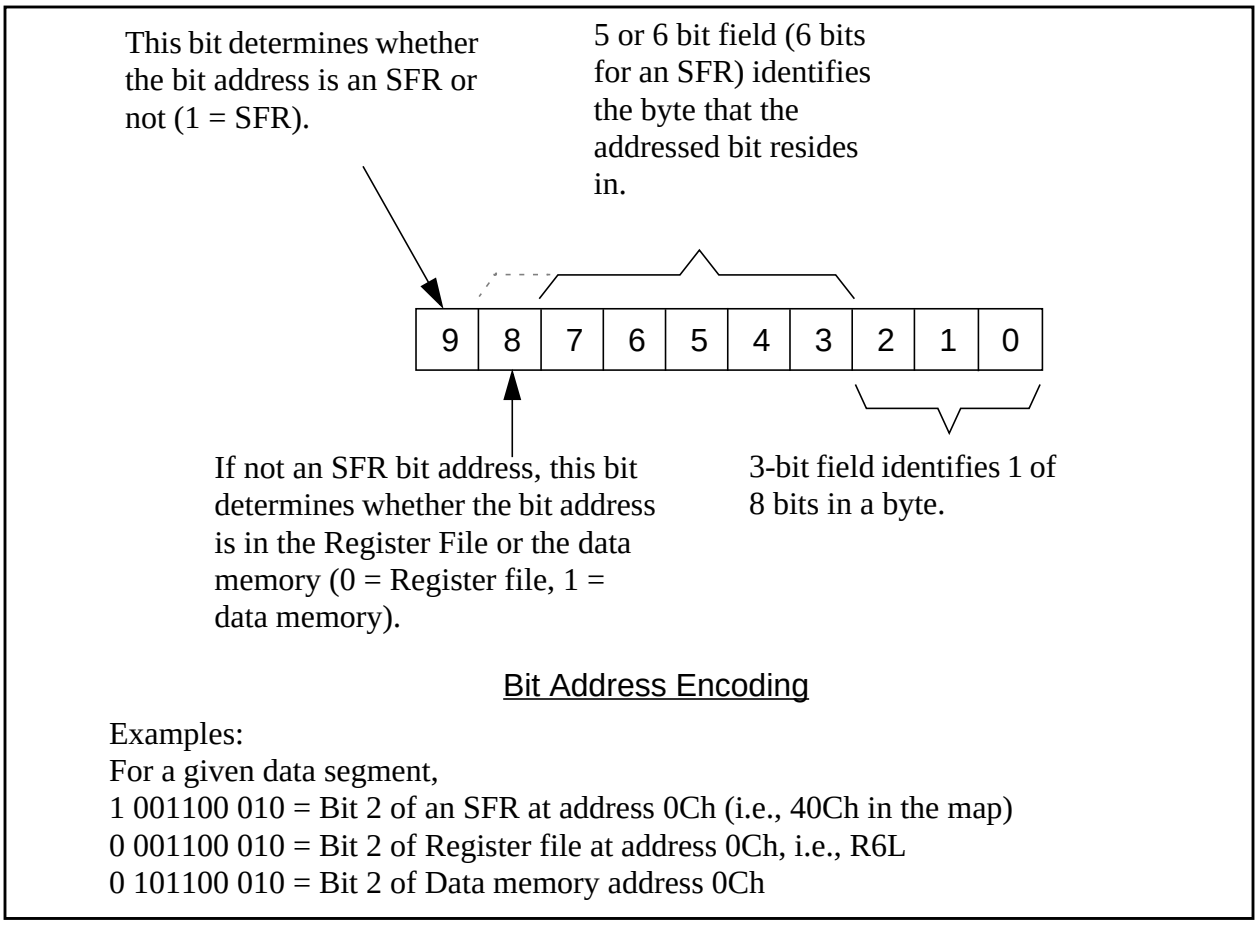


Figure 6.6

6.3 Relative Branching and Jumps

Program memory addresses as referenced by Jumps, Calls, and Branch instructions must be word aligned in XA. For instance, a branch instruction may occur at any code address, but it may only branch to an even address. This forced alignment to even address provides three benefits:

- Branch ranges are doubled without providing an extra bit in the instruction and
- Faster execution as XA always fetches first two byte of an instruction simultaneously.
- Allows translated 8051 code to have branches extended over intervening code that will tend to grow when translated and generally increase the chances of a branch target being in that range.

The *rel8* displacement is a 9-bit two's complement integer which is encoded as 8-bits that represents the relative distance in words from the current PC to the destination PC. Similarly, the *rel16* displacement is a 17-bit twos complement integer which is encoded as 16-bits. The value of the PC used in the target address calculation is the address of the instruction following the Branch, Jump or Call instruction.

The 8-bit signed displacement is between -128 to +127. The branch range for rel8 is (sample calculation shown below) is really +254 bytes to -256 bytes for instructions located at an *even* address, and +253 to -257 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

The 16-bit signed displacement is -32,768 to +32,767. The branch range is therefore +65,534 bytes to -65,536 bytes for instructions located at an *even* address, and +65,533 to -65,537 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

Sample calculation for rel8 range:

Assuming word aligned branch target, forward range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = +127; So, rel8 << 1 is +254

If Current PC = ODD, then
Range = 254 - 1 = +253 as PC is forced to an even location, else
If current PC = EVEN, then
Range = +254

Similarly, reverse range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = -128; So, rel8 << 1 is -256

If Current PC = ODD, then
Range = -257
Else if current PC = EVEN, then
Range = -256

6.4 Data Types in XA

The XA uses the following types of data:

- Bits
- 4/5-bit signed integers
- 8-bit (byte) signed and unsigned integers
- 8-bit, two digit BCD numbers
- 16-bit (word) signed and unsigned integers
- 10-bit address for bit-addressing in data memory and SFR space
- 24-bit effective address comprising of 16-bit address and 8-bit segment select. See addressing modes for more information.

A byte consists of 8-bits. A word is a 16-bit value consisting of two contiguous bytes. A double word consists of two 16-bit words packed in two contiguous words in memory.

Negative integers are represented in twos complement form. 4-bit signed integers (sign extended to byte/word) are used as immediate operands in MOVS and ADDS instructions.

Binary coded decimal numbers are packed, 2 digits per byte. BCD operations use byte operands.

6.5 Instruction Set Overview

The XA uses a powerful and efficient instruction set, offering several different types of addressing modes. A versatile set of “branch” and “jump” instructions are available for controlling program flow based on register or memory contents. Special emphasis has been placed on the instruction support of structured high-level languages and real-time multi-tasking operating systems.

This section discusses the set of instructions provided in the XA microcontroller, and also shows how to use them. It includes descriptions of the instruction format and the operands used by the instructions. After a summary of the instructions by category, the section provides a detailed description of the operation of each instruction, in alphabetical order.

Five summary tables are provided that describes the available instructions. The first table is a summary of instructions available in the XA along with their common usage. The second and third table are tables of addressing modes and operands, and the instruction type they pertain to. A fourth table that lists the summary of status flags update by different instructions. A fifth table lists the available instructions with their different addressing modes and briefly describes what each instruction does along with the number of bytes, and number of clocks required for each instruction.

The formats have been chosen to optimize the length and execution speed of those instructions that would be used the most often in critical code. Only the first and sometimes the second byte of an instruction are used for operation encoding. The length of the instruction and the first execution cycle activity are determined from the first byte. Instruction bytes following the first two bytes (if any) are always immediate operands, such as addresses, relative displacements, offsets, bit addresses, and immediate data.

Glossary of mnemonics, notations used

General:

offset8	An 8-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
offset16	A 16-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
direct	An 11-bit immediate address contained in the instruction.
#data4	4 bits of immediate data contained in the instruction. (range +7 to -8 for signed immediate data and 0-15 for shifts)
#data5	5 bits of immediate data contained in the instruction. (0-31 for shifts)
#data8	8 bits of immediate data contained in the instruction. (+127 to -128)
#data16	16 bits of immediate data contained in the instruction. (+32,767 to -32,768)
bit	The 10-bit address of an addressable bit.
rel8	An 8-bit relative displacement for branches. (+254 to -256)
rel16	An 16-bit relative displacement for branches.(+65,534 to -65,536)
addr16	A 16-bit absolute branch address within a 64K code page.
addr24	A 24-bit absolute branch address, able to access the entire XA address space.
SP	The current Stack Pointer (User or System) depending on the operation mode.
USP	The User Stack Pointer.
SSP	The System Stack Pointer
C	Carry flag from the PSW.
AC	Auxiliary Carry flag from the PSW.
V	Overflow flag from the PSW.
N	Negative flag from the PSW.
Z	Zero flag from the PSW.
DS	Data segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for local data segments.
ES	Extra segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for addressing remote data structures.

direct	Uses the current DS for data memory for the upper byte of the 24-bit address or none (uses only the low 16-bit address) for accessing the special functions register (SFR) space. The interpretation should be as below:
	if (data range)
	then (direct = (DS:direct)
	if (sfr range)
	then (direct) = (sfr)

Operation encoding fields:

SZ	Data Size. This field encodes whether the operation is byte, word or double-word.
IND	This field flags indirect operation in some instructions.
H/L	This field selects whether PUSH and POP Rlist use the upper or lower half of the available registers.
dddd	Destination register field, specifies one of 16 registers in the register file.
ddd	Destination register field for indirect references, specifies one of 8 pointer registers in the register file.
ssss	Source register field, specifies one of 16 registers in the register file.
sss	Source register field for indirect references, specifies one of 8 pointer registers in the register file.

Mnemonic text:

Rs	Source register.
Rd	Destination register.
[ ]	In the instruction mnemonic, indicates an indirect reference (e.g.: [R4] refers to the memory address pointed to by the contents of register 4).
[R+]	Used to indicate an automatic increment of the pointer register in some indirect addressing modes.
[WS:R]	Indicates that the pointer register (R) is extended to a 24-bit pointer by the selected segment register (either DS or ES for all instructions except MOVC, which uses either PC_{23-16} or CS).
Rlist	A bitmap that represents each register in the register file during a PUSH or POP operation. These registers are R0-R7 for word or R0L-R7H for byte.

Pseudocode:

( )	Used to indicate "contents of" in the instruction operation pseudocode (e.g.: (R4) refers to the contents of register 4).
<---	Pseudocode assignment operator. Occasionally used as <--> to indicate assignment in both directions (interchange of data).
((SP))	Data memory contents at the location pointed to by the current stack pointer. In system mode, the current SP is the SSP, and the segment used is always segment 0. In user mode, the current SP is the USP, and the segment used is the Data Segment (DS). This segment apply to the uses of the SP, not just PUSH and POP. In a few cases, “((SSP))” or “((USP))” indicate that a specific SP is used, regardless of the operating mode.
Rn.x	Indicates bit x of register n.
Rn.x-y	Indicates a range of bits from bit x to bit y of register n.

Note: all indirect addressing is accomplished using the contents of the data segment register as the upper 8 address bits unless otherwise specified. Example: [ES:Rs] indicates that the extra segment register generates the upper 8 bits of the address in this case.

Execution time:

PZ	- In Page 0
nt	- Not Taken
t	- Taken

Syntax For Operand size:
.w = For word operands
.b = byte operands
.d = double-word operands

Default operand size is dependant on the operands used e.g MOV R0,R1 is always word-size whereas MOV R0L, R0H is always byte etc. For INDIRECT_IMMEDIATE, DIRECT_IMMEDIATE, DIRECT_DIRECT, etc., user must specify operand size.

Others

0x = prefix for Hex values
[] = For indirect addressing
[[]] = For Double-indirect addressing
dest = destination
src = source

Table 6.2 Instruction Set in XA

Mnemonic	Usage
MOV, MOVC, MOVS, MOVX, LEA, XCH, PUSH, POP, PUSHU, POPU	Data Movement
ADD, ADDS, ADDC, SUB, SUBB	Add and Subtract
MULU.b, MULU.w, MUL.w DIVU.b, DIVU.w, DIVU.d, DIV.w, DIV.d	Multiply and Divide
RR, RRC, RL, RLC, LSR, ASR, ASL, NORM	Shifts and Rotates
CLR, SETB, MOV, ANL, ORL	Bit Operations
JB, JBC, JNB, JNZ, JZ, DJNZ, CJNE,	Conditional Jumps/Calls
BOV, BNV, BPL, BCC, BCS, BEQ, BNE, BG, BGE, BGT, BL, BLE, BLT, BMI	Conditional Branches
AND, OR, XOR	Boolean Functions
JMP, FJMP, CALL, FCALL, BR	Unconditional Jumps/Calls/Branches
RET, RETI	Return from subroutines, interrupts
SEXT, NEG, CPL, DA	Sign Extend, Negate, Complement, Decimal Adjust
BKPT, TRAP#, RESET	Exceptions
NOP	No Operation

Table 6.3 shows a summary of the basic addressing modes available for data transfer and calculation related instructions.

Table 6.3 Instruction Addressing Modes

Modes/ Operands	MOVX	MOV	CMP	ADD ADDC	SUB SUBB	AND OR XOR	ADDS MOVS	MUL DIV	Shift	XCH	bytes
R, R		•	•	•	•	•		•	•	•	2
R, [R]	•	•	•	•	•	•				•	2
[R], R	•	•	•	•	•	•					2
R, [R+off8]		•	•	•	•	•					3
[R+off8], R		•	•	•	•	•					3
R, [R+off16]		•	•	•	•	•					4
[R+off16], R		•	•	•	•	•					4
R, [R+]		•	•	•	•	•					2
[R+], R		•	•	•	•	•					2
[R+], [R+]		•									2
dir, R		•	•	•	•	•					3
R, dir		•	•	•	•	•				•	3
dir, [R]		•									3
[R], dir		•									3
R, #data		•	•	•	•	•	•	•	•		2*/3/4
[R], #data		•	•	•	•	•	•				2*/3/4
[R+], #data		•	•	•	•	•	•				2*/3/4
[R+off8], #data		•	•	•	•	•	•				3*/4/5
[R+off16], #data		•	•	•	•	•	•				4*/5/6
dir, #data		•	•	•	•	•	•				3*/4/5
dir, dir		•									4
R, USP		•									2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Modes/ Operands	**MOVC**	PUSH POP	DA, SEXT CPL, NEG	JUMP CALL	DJNZ	CJNE	BIT OPS	MISC	bytes
R, [R+]	•								2
[R+], R	•								2
A, [A+DPTR]	•								2
A, [A+PC]	•								2
direct		•							3
Rlist		•							2
R			•						2
addr24				•					4
[R]				•					2
[A+DPTR]				JMP					2
R, rel					•				3
direct, rel					•				4
R, direct, rel						•			4
R, #data, rel						•			4/5
[R], #data, rel						•			4/5
bit							•		3
bit, C; C, bit							•		3
C, /bit							•		3
rel				•				Cond. Branch	2
bit, rel								Cond. Branch	4
#data4								TRAP	2
R, R+off8								LEA	3
r, R+off16								LEA	4
<none> **								•	1/2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Table 6.4 summarizes the status flag updates for the various XA instruction types.

Table 6.4 Status Flag Updates

Instruction Type	Flags Updated				
	C	AC	V	N	Z
ADD, ADDC, CMP, SUB, SUBB	X	X	X	X	X
ADDS, MOVS	-	-	-	X	X
AND, OR, XOR	-	-	-	X	X
ASR, LSR	*	-	-	X	X
branches, all bit operations, NOP	-	-	-	-	-
Calls, Jumps, and Returns	-	-	-	-	-
CJNE	X	-	-	X	X
CPL	-	-	-	X	X
DA	*	-	-	X	X
DIV, MUL	*	-	*	X	X
DJNZ	-	-	-	X	X
LEA	-	-	-	-	-
MOV, MOVC, MOVX	-	-	-	X	X
NEG	-	-	X	X	X
NORM	-	-	-	X	X
PUSH, POP	-	-	-	-	-
RESET	*	*	*	*	*
RL, RR	-	-	-	X	X
RLC, RRC	*	-	-	X	X
SEXT	-	-	-	-	-
TRAP, BKPT	-	-	-	-	-
XCH	-	-	-	-	-
ASL	*	-	X	X	X

Notes:
-: flag not updated.
X: flag updated according to the standard definition.
*: flag update is non-standard, refer to the individual instruction description.
Note: Explicit writes to PSW flags takes precedence over flag updates.

Instruction Set Summary

Table 6.5 lists the entire XA instruction set by instruction type. This can be used as a quick reference to find specific instructions that may be looked up in the detailed alphabetical description section.

Instruction timing data given in this table and in the following detailed instruction description section are based on code execution from internal code memory and data accesses to internal RAM and registers only. Due to the highly programmable timing of accesses to external code and data memory on the XA and the interaction of pipelined functions, detailed timing for all conditions cannot be documented in a concise fashion. The instruction timing data given here also assumes that the CPU does not need to stall while the instruction is read into the pre-fetch queue.

In the case of branches, one on-chip code fetch (16 bits) is built into the timing numbers. The time given will be valid if the instruction that is branched to is not longer than two bytes. For longer instructions, the CPU will wait until the entire instruction is contained in the pre-fetch queue before resuming execution. This may take one or two additional fetches since the XA has instructions up to six bytes in length.

Following is a summary of events or conditions that may cause timing differences from the given data. These are generally stalls that occur when the CPU must wait for some information to become available.

— Instruction fetch. Execution stalls if the pre-fetch queue does not contains a complete instruction when it is needed. Except following branches, the state of the queue depends upon the history of instructions that have previously executed.

— Instruction sequence dependencies. This typically occurs when an instruction that reads data from a resource such as the SFR bus or the external bus follows an instruction that caused a write to the same resource. The CPU must stall while the write completes (which otherwise requires no CPU time) before the read can begin. Execution cannot resume until the read is complete.

— Internal data memory versus SFR accesses. SFR reads require an additional 2 clocks to complete. Because XA peripherals run from the CPU clock divided by 2, there may be one clock used to synchronize the CPU and the SFR bus.

— Program flow changes. When any change occurs in the program flow, the XA must flush the pre-fetch queue and begin loading it from the new execution address. The published timing values include one internal code fetch for all branches, jumps, calls, etc. If the instruction at the new address is longer than two bytes, additional fetch cycles must occur to obtain a complete instruction in the queue. In the case of a return from subroutine or interrupt, the first code fetch may only obtain one byte of the next instruction since returns may resume execution at odd code addresses.

— Internal versus external code execution. Programmable bus timing and other bus considerations result in a different timing for internal and external code accesses. Use of the 8-bit bus width for external code access has a substantial effect on overall performance. Possible use of the WAIT signal adds an additional variable to this effect. The external bus requirement for an ALE cycle at 16-byte address boundaries, during program flow changes, and after external bus data accesses also adds to the variability.

— Internal versus external data access. Programmable bus timing again causes different timing for internal and external data accesses. The 8-bit data bus setting contributes to the differences. Use of the WAIT signal may vary the timing still further.

— Collision of external code fetch and external data access. When an externally executing program accesses data on the external bus, the pre-fetch queue tends to starve more often that for internal execution.

Table 6.5

Mnemonic		Description	Bytes	Clocks
Arithmetic Operations				
ADD	Rd, Rs	Add registers direct	2	3
ADD	Rd, [Rs]	Add register-indirect to register	2	4
ADD	[Rd], Rs	Add register to register-indirect	2	4
ADD	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register	3	6
ADD	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset	3	6
ADD	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register	4	6
ADD	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset	4	6
ADD	Rd, [Rs+]	Add register-indirect with auto increment to register	2	5
ADD	[Rd+], Rs	Add register-indirect with auto increment to register	2	5
ADD	direct, Rs	Add register to memory	3	4
ADD	Rd, direct	Add memory to register	3	4
ADD	Rd, #data8	Add 8-bit immediate data to register	3	3
ADD	Rd, #data16	Add 16-bit immediate data to register	4	3
ADD	[Rd], #data8	Add 8-bit immediate data to register-indirect	3	4
ADD	[Rd], #data16	Add 16-bit immediate data to register-indirect	4	4
ADD	[Rd+], #data8	Add 8-bit immediate data to register-indirect with auto-increment	3	5
ADD	[Rd+], #data16	Add 16-bit immediate data to register-indirect with auto-increment	4	5
ADD	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset	4	6
ADD	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset	5	6
ADD	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADD	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset	6	6
ADD	direct, #data8	Add 8-bit immediate data to memory	4	4
ADD	direct, #data16	Add 16-bit immediate data to memory	5	4
ADDC	Rd, Rs	Add registers direct with carry	2	3
ADDC	Rd, [Rs]	Add register-indirect to register with carry	2	4
ADDC	[Rd], Rs	Add register to register-indirect with carry	2	4
ADDC	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register with carry	3	6
ADDC	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset with carry	3	6
ADDC	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register with carry	4	6
ADDC	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset with carry	4	6
ADDC	Rd, [Rs+]	Add register-indirect with auto increment to register with carry	2	5
ADDC	[Rd+], Rs	Add register-indirect with auto increment to register with carry	2	5
ADDC	direct, Rs	Add register to memory with carry	3	4
ADDC	Rd, direct	Add memory to register with carry	3	4
ADDC	Rd, #data8	Add 8-bit immediate data to register with carry	3	3
ADDC	Rd, #data16	Add 16-bit immediate data to register with carry	4	3
ADDC	[Rd], #data8	Add 16-bit immediate data to register-indirect with carry	3	4
ADDC	[Rd], #data16	Add 16-bit immediate data to register-indirect with carry	4	4
ADDC	[Rd+], #data8	Add 8-bit immediate data to register-indirect and auto-increment with carry	3	5
ADDC	[Rd+], #data16	Add 16-bit immediate data to register-indirect and auto-increment with carry	4	5
ADDC	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset and carry	4	6
ADDC	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset and carry	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADDC	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset and carry	5	6
ADDC	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset and carry	6	6
ADDC	direct, #data8	Add 8-bit immediate data to memory with carry	4	4
ADDC	direct, #data16	Add 16-bit immediate data to memory with carry	5	4
ADDS	Rd, #data4	Add 4-bit signed immediate data to register	2	3
ADDS	[Rd], #data4	Add 4-bit signed immediate data to register-indirect	2	4
ADDS	[Rd+], #data4	Add 4-bit signed immediate data to register-indirect with auto-increment	2	5
ADDS	[Rd+offset8], #data4	Add register-indirect with 8-bit offset to 4-bit signed immediate data	3	6
ADDS	[Rd+offset16], #data4	Add register-indirect with 16-bit offset to 4-bit signed immediate data	4	6
ADDS	direct, #data4	Add 4-bit signed immediate data to memory	3	4
ASL	Rd, Rs	Logical left shift destination register by the value in the source register	2	See Note1
ASL	Rd, #data4	Logical left shift register by the 4-bit immediate value	2	See Note1
ASR	Rd, Rs	Arithmetic shift right destination register by the count in the source	2	See Note1
ASR	Rd, #data4	Arithmetic shift right register by the 4-bit immediate count	2	See Note1
CMP	Rd, Rs	Compare destination and source registers	2	3
CMP	[Rd], Rs	Compare register with register-indirect	2	4
CMP	Rd, [Rs]	Compare register-indirect with register	2	4
CMP	[Rd+offset8], Rs	Compare register with register-indirect with 8-bit offset	3	6
CMP	[Rd+offset16], Rs	Compare register with register-indirect with 16-bit offset	4	6
CMP	Rd, [Rs+offset8]	Compare register-indirect with 8-bit offset with register	3	6
CMP	Rd,[Rs+offset16]	Compare register-indirect with 16-bit offset with register	4	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
CMP	Rd, [Rs+]	Compare auto-increment register-indirect with register	2	5
CMP	[Rd+], Rs	Compare register with auto-increment register-indirect	2	5
CMP	direct, Rs	Compare register with memory	3	4
CMP	Rd, direct	Compare memory with register	3	4
CMP	Rd, #data8	Compare 8-bit immediate data to register	3	3
CMP	Rd, #data16	Compare 16-bit immediate data to register	4	3
CMP	[Rd], #data8	Compare 8 -bit immediate data to register-indirect	3	4
CMP	[Rd], #data16	Compare 16-bit immediate data to register-indirect	4	4
CMP	[Rd+], #data8	Compare 8-bit immediate data to register-indirect with auto-increment	3	5
CMP	[Rd+], #data16	Compare 16-bit immediate data to register-indirect with auto-increment	4	5
CMP	[Rd+offset8], #data8	Compare 8-bit immediate data to register-indirect with 8-bit offset	4	6
CMP	[Rd+offset8], #data16	Compare 16-bit immediate data to register-indirect with 8-bit offset	5	6
CMP	[Rd+offset16], #data8	Compare 8-bit immediate data to register-indirect with 16-bit offset	5	6
CMP	[Rd+offset16], #data16	Compare 16-bit immediate data to register-indirect with 16-bit offset	6	6
CMP	direct, #data8	Compare 8-bit immediate data to memory	4	4
CMP	direct, #data16	Compare 16-bit immediate data to memory	5	4
DA	Rd	Decimal Adjust byte register	2	4
DIV.w	Rd, Rs	16x8 signed register divide	2	14
DIV.w	Rd, #data8	16x8 signed divide register with immediate word	3	14
DIV.d	Rd, Rs	32x16 signed double register divide	2	24
DIV.d	Rd, #data16	32x16 signed double register divide with immediate word	4	24
DIVU.b	Rd, Rs	8x8 unsigned register divide	2	12
DIVU.b	Rd, #data8	8X8 unsigned register divide with immediate byte	3	12

Table 6.5

Mnemonic		Description	Bytes	Clocks
DIVU.w	Rd, Rs	16X8 unsigned register divide	2	12
DIVU.w	Rd, #data8	16X8 unsigned register divide with immediate byte	3	12
DIVU.d	Rd, Rs	32X16 unsigned double register divide	2	22
DIVU.d	Rd, #data16	32X16 unsigned double register divide with immediate word	4	22
LEA	Rd, Rs+offset8	Load 16-bit effective address with 8-bit offset to register	3	3
LEA	Rd, Rs+offset16	Load 16-bit effective address with 16-bit offset to register	4	3
MUL.w	Rd, Rs	16X16 signed multiply of register contents	2	12
MUL.w	Rd, #data16	16X16 signed multiply 16-bit immediate data with register	4	12
MULU.b	Rd, Rs	8X8 unsigned multiply of register contents	2	12
MULU.b	Rd, #data8	8X8 unsigned multiply of 8-bit immediate data with register	3	12
MULU.w	Rd, Rs	16X16 unsigned register multiply	2	12
MULU.w	Rd, #data16	16X16 unsigned multiply 16-bit immediate data with register	4	12
NEG	Rd	Negate (twos complement) register	2	3
SEXT	Rd	Sign extend last operation to register	2	3
SUB	Rd, Rs	Subtract registers direct	2	3
SUB	Rd, [Rs]	Subtract register-indirect to register	2	4
SUB	[Rd], Rs	Subtract register to register-indirect	2	4
SUB	Rd, [Rs+offset8]	Subtract register-indirect with 8-bit offset to register	3	6
SUB	[Rd+offset8], Rs	Subtract register to register-indirect with 8-bit offset	3	6
SUB	Rd, [Rs+offset16]	Subtract register-indirect with 16-bit offset to register	4	6
SUB	[Rd+offset16], Rs	Subtract register to register-indirect with 16-bit offset	4	6
SUB	Rd, [Rs+]	Subtract register-indirect with auto increment to register	2	5
SUB	[Rd+], Rs	Subtract register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUB	direct, Rs	Subtract register to memory	3	4
SUB	Rd, direct	Subtract memory to register	3	4
SUB	Rd, #data8	Subtract 8-bit immediate data to register	3	3
SUB	Rd, #data16	Subtract 16-bit immediate data to register	4	3
SUB	[Rd], #data8	Subtract 8-bit immediate data to register-indirect	3	4
SUB	[Rd], #data16	Subtract 16-bit immediate data to register-indirect	4	4
SUB	[Rd+], #data8	Subtract 8-bit immediate data to register-indirect with auto-increment	3	5
SUB	[Rd+], #data16	Subtract 16-bit immediate data to register-indirect with auto-increment	4	5
SUB	[Rd+offset8], #data8	Subtract 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUB	[Rd+offset8], #data16	Subtract 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUB	[Rd+offset16], #data8	Subtract 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUB	[Rd+offset16], #data16	Subtract 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUB	direct, #data8	Subtract 8-bit immediate data to memory	4	4
SUB	direct, #data16	Subtract 16-bit immediate data to memory	5	4
SUBB	Rd, Rs	Subtract with borrow registers direct	2	3
SUBB	Rd, [Rs]	Subtract with borrow register-indirect to register	2	4
SUBB	[Rd], Rs	Subtract with borrow register to register-indirect	2	4
SUBB	Rd, [Rs+offset8]	Subtract with borrow register-indirect with 8-bit offset to register	3	6
SUBB	[Rd+offset8], Rs	Subtract with borrow register to register-indirect with 8-bit offset	3	6
SUBB	Rd, [Rs+offset16]	Subtract with borrow register-indirect with 16-bit offset to register	4	6
SUBB	[Rd+offset16], Rs	Subtract with borrow register to register-indirect with 16-bit offset	4	6
SUBB	Rd, [Rs+]	Subtract with borrow register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUBB	[Rd+], Rs	Subtract with borrow register-indirect with auto increment to register	2	5
SUBB	direct, Rs	Subtract with borrow register to memory	3	4
SUBB	Rd, direct	Subtract with borrow memory to register	3	4
SUBB	Rd, #data8	Subtract with borrow 8-bit immediate data to register	3	3
SUBB	Rd, #data16	Subtract with borrow 16-bit immediate data to register	4	3
SUBB	[Rd], #data8	Subtract with borrow 8-bit immediate data to register-indirect	3	4
SUBB	[Rd], #data16	Subtract with borrow 16-bit immediate data to register-indirect	4	4
SUBB	[Rd+], #data8	Subtract with borrow 8-bit immediate data to register-indirect with auto-increment	3	5
SUBB	[Rd+], #data16	Subtract with borrow 16-bit immediate data to register-indirect with auto-increment	4	5
SUBB	[Rd+offset8], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUBB	[Rd+offset8], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUBB	[Rd+offset16], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUBB	[Rd+offset16], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUBB	direct, #data8	Subtract with borrow 8-bit immediate data to memory	4	4
SUBB	direct, #data16	Subtract with borrow 16-bit immediate data to memory	5	4
Logical Operations				
AND	Rd, Rs	Logical AND registers direct	2	3
AND	Rd, [Rs]	Logical AND register-indirect to register	2	4
AND	[Rd], Rs	Logical AND register to register-indirect	2	4
AND	Rd, [Rs+offset8]	Logical AND register-indirect with 8-bit offset to register	3	6
AND	[Rd+offset8], Rs	Logical AND register to register-indirect with 8-bit offset	3	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
AND	Rd, [Rs+offset16]	Logical AND register-indirect with 16-bit offset to register	4	6
AND	[Rd+offset16], Rs	Logical AND register to register-indirect with 16-bit offset	4	6
AND	Rd, [Rs+]	Logical AND register-indirect with auto increment to register	2	5
AND	[Rd+], Rs	Logical AND register-indirect with auto increment to register	2	5
AND	direct, Rs	Logical AND register to memory	3	4
AND	Rd, direct	Logical AND memory to register	3	4
AND	Rd, #data8	Logical AND 8-bit immediate data to register	3	3
AND	Rd, #data16	Logical AND 16-bit immediate data to register	4	3
AND	[Rd], #data8	Logical AND 8-bit immediate data to register-indirect	3	4
AND	[Rd], #data16	Logical AND16-bit immediate data to register-indirect	4	4
AND	[Rd+], #data8	Logical AND 8-bit immediate data to register-indirect and auto-increment	3	5
AND	[Rd+], #data16	Logical AND16-bit immediate data to register-indirect and auto-increment	4	5
AND	[Rd+offset8], #data8	Logical AND8-bit immediate data to register-indirect with 8-bit offset	4	6
AND	[Rd+offset8], #data16	Logical AND16-bit immediate data to register-indirect with 8-bit offset	5	6
AND	[Rd+offset16], #data8	Logical AND8-bit immediate data to register-indirect with 16-bit offset	5	6
AND	[Rd+offset16], #data16	Logical AND16-bit immediate data to register-indirect with 16-bit offset	6	6
AND	direct, #data8	Logical AND 8-bit immediate data to memory	4	4
AND	direct, #data16	Logical AND16-bit immediate data to memory	5	4
CPL	Rd	Complement (ones complement) register	2	3
LSR	Rd, Rs	Logical right shift destination register by the value in the source register	2	See Note 1
LSR	Rd, #data4	Logical right shift register by the 4-bit immediate value	2	See Note 1
NORM	Rd, Rs	Logical shift left destination register by the value in the source register until MSB set	2	See Note 1

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	Rd, Rs	Logical OR registers	2	3
OR	Rd, [Rs]	Logical OR register-indirect to register	2	4
OR	[Rd], Rs	Logical OR register to register-indirect	2	4
OR	Rd, [Rs+offset8]	Logical OR register-indirect with 8-bit offset to register	3	6
OR	[Rd+offset8], Rs	Logical OR register to register-indirect with 8-bit offset	3	6
OR	Rd, [Rs+offset16]	Logical OR register-indirect with 16-bit offset to register	4	6
OR	[Rd+offset16], Rs	Logical OR register to register-indirect with 16-bit offset	4	6
OR	Rd, [Rs+]	Logical OR register-indirect with auto increment to register	2	5
OR	[Rd+], Rs	Logical OR register-indirect with auto increment to register	2	5
OR	direct, Rs	Logical OR register to memory	3	4
OR	Rd, direct	Logical OR memory to register	3	4
OR	Rd, #data8	Logical OR 8-bit immediate data to register	3	3
OR	Rd, #data16	Logical OR 16-bit immediate data to register	4	3
OR	[Rd], #data8	Logical OR 8-bit immediate data to register-indirect	3	4
OR	[Rd], #data16	Logical OR 16-bit immediate data to register-indirect	4	4
OR	[Rd+], #data8	Logical OR 8-bit immediate data to register-indirect with auto-increment	3	5
OR	[Rd+], #data16	Logical OR 16-bit immediate data to register-indirect with auto-increment	4	5
OR	[Rd+offset8], #data8	Logical OR 8-bit immediate data to register-indirect with 8-bit offset	4	6
OR	[Rd+offset8], #data16	Logical OR 16-bit immediate data to register-indirect with 8-bit offset	5	6
OR	[Rd+offset16], #data8	Logical OR 8-bit immediate data to register-indirect with 16-bit offset	5	6
OR	[Rd+offset16], #data16	Logical OR 16-bit immediate data to register-indirect with 16-bit offset	6	6
OR	direct, #data8	Logical OR 8-bit immediate data to memory	4	4

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	direct, #data16	Logical OR16-bit immediate data to memory	5	4
RL	Rd, #data4	Rotate left register by the 4-bit immediate value	2	See Note 1
RLC	Rd, #data4	Rotate left register though carry by the 4-bit immediate value	2	See Note 1
RR	Rd, #data4	Rotate right register by the 4-bit immediate value	2	See Note 1
RRC	Rd, #data4	Rotate right register though carry by the 4-bit immediate value	2	See Note 1
XOR	Rd, Rs	Logical XOR registers	2	3
XOR	Rd, [Rs]	Logical XOR register-indirect to register	2	4
XOR	[Rd], Rs	Logical XOR register to register-indirect	2	4
XOR	Rd, [Rs+offset8]	Logical XOR register-indirect with 8-bit offset to register	3	6
XOR	[Rd+offset8], Rs	Logical XOR register to register-indirect with 8-bit offset	3	6
XOR	Rd, [Rs+offset16]	Logical XOR register-indirect with 16-bit offset to register	4	6
XOR	[Rd+offset16], Rs	Logical XOR register to register-indirect with 16-bit offset	4	6
XOR	Rd, [Rs+]	Logical XOR register-indirect with auto increment to register	2	5
XOR	[Rd+], Rs	Logical XOR register-indirect with auto increment to register	2	5
XOR	direct, Rs	Logical XOR register to memory	3	4
XOR	Rd, direct	Logical XOR memory to register	3	4
XOR	Rd, #data8	Logical XOR 8-bit immediate data to register	3	3
XOR	Rd, #data16	Logical XOR 16-bit immediate data to register	4	3
XOR	[Rd], #data8	Logical XOR 8-bit immediate data to register-indirect	3	4
XOR	[Rd], #data16	Logical XOR 16-bit immediate data to register-indirect	4	4
XOR	[Rd+], #data8	Logical XOR 8-bit immediate data to register-indirect with auto-increment	3	5
XOR	[Rd+], #data16	Logical XOR 16-bit immediate data to register-indirect with auto-increment	4	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
XOR	[Rd+offset8], #data8	Logical XOR 8-bit immediate data to register-indirect with 8-bit offset	4	6
XOR	[Rd+offset8], #data16	Logical XOR 16-bit immediate data to register-indirect with 8-bit offset	5	6
XOR	[Rd+offset16], #data8	Logical XOR 8-bit immediate data to register-indirect with 16-bit offset	5	6
XOR	[Rd+offset16], #data16	Logical XOR 16-bit immediate data to register-indirect with 16-bit offset	6	6
XOR	direct, #data8	Logical XOR 8-bit immediate data to memory	4	4
XOR	direct, #data16	Logical XOR16-bit immediate data to memory	5	4
Data transfer				
MOV	Rd, Rs	Move register to register	2	3
MOV	Rd, [Rs]	Move register-indirect to register	2	3
MOV	[Rd], Rs	Move register to register-indirect	2	3
MOV	Rd, [Rs+offset8]	Move register-indirect with 8-bit offset to register	3	5
MOV	[Rd+offset8], Rs	Move register to register-indirect with 8-bit offset	3	5
MOV	Rd, [Rs+offset16]	Move register-indirect with 16-bit offset to register	4	5
MOV	[Rd+offset16], Rs	Move register to register-indirect with 16-bit offset	4	5
MOV	Rd, [Rs+]	Move register-indirect with auto increment to register	2	4
MOV	[Rd+], Rs	Move register-indirect with auto increment to register	2	4
MOV	direct, Rs	Move register to memory	3	4
MOV	Rd, direct	Move memory to register	3	4
MOV	[Rd+], [Rs+]	Move register-indirect to register-indirect, both pointers auto-incremented	2	6
MOV	direct, [Rs]	Move register-indirect to memory	3	4
MOV	[Rd], direct	Move memory to register-indirect	3	4
MOV	Rd, #data8	Move 8-bit immediate data to register	3	3
MOV	Rd, #data16	Move 16-bit immediate data to register	4	3

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOV	[Rd], #data8	Move 16-bit immediate data to register-indirect	3	3
MOV	[Rd], #data16	Move 16-bit immediate data to register-indirect	4	3
MOV	[Rd+], #data8	Move 8-bit immediate data to register-indirect with auto-increment	3	4
MOV	[Rd+], #data16	Move 16-bit immediate data to register-indirect with auto-increment	4	4
MOV	[Rd+offset8], #data8	Move 8-bit immediate data to register-indirect with 8-bit offset	4	5
MOV	[Rd+offset8], #data16	Move 16-bit immediate data to register-indirect with 8-bit offset	5	5
MOV	[Rd+offset16], #data8	Move 8-bit immediate data to register-indirect with 16-bit offset	5	5
MOV	[Rd+offset16], #data16	Move 16-bit immediate data to register-indirect with 16-bit offset	6	5
MOV	direct, #data8	Move 8-bit immediate data to memory	4	3
MOV	direct, #data16	Move 16-bit immediate data to memory	5	3
MOV	direct, direct	Move memory to memory	4	4
MOV	Rd, USP	Move User Stack Pointer to register (system mode only)	2	3
MOV	USP, Rs	Move register to User Stack Pointer (system mode only)	2	3
MOVC	Rd, [Rs+]	Move data from WS:Rs address of code memory to register with auto-increment	2	4
MOVC	A, [A+DPTR]	Move data from code memory to the accumulator indirect with DPTR	2	6
MOVC	A, [A+PC]	Move data from code memory to the accumulator indirect with PC	2	6
MOVS	Rd, #data4	Move 4-bit sign-extended immediate data to register	2	3
MOVS	[Rd], #data4	Move 4-bit sign-extended immediate data to register-indirect	2	3
MOVS	[Rd+], #data4	Move 4-bit sign-extended immediate data to register-indirect with auto-increment	2	4
MOVS	[Rd+offset8], #data4	Move register-indirect with 8-bit offset to 4-bit sign-extended immediate data	3	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOVS	[Rd+offset16], #data4	Move register-indirect with 16-bit offset to 4-bit sign-extended immediate data	4	5
MOVS	direct, #data4	Move 4-bit sign-extended immediate data to memory	3	3
MOVX	Rd, [Rs]	Move external data from memory to register	2	6
MOVX	[Rd], Rs	Move external data from register to memory	2	6
PUSH	direct	Push the memory content (byte/word) onto the current stack	3	5
PUSHU	direct	Push the memory content (byte/word) onto the user stack	3	5
PUSH	Rlist	Push multiple registers (byte/word) onto the current stack	2	See Note 2
PUSHU	Rlist	Push multiple registers (byte/word)from the user stack	2	See Note 2
POP	direct	Pop the memory content (byte/word) from the current stack	3	5
POPU	direct	Pop the memory content (byte/word) from the user stack	3	5
POP	Rlist	Pop multiple registers (byte/word) from the current stack	2	See Note 3
POPU	Rlist	Pop multiple registers (byte/word) from the user stack	2	See Note 3
XCH	Rd, Rs	Exchange contents of two registers	2	5
XCH	Rd, [Rs]	Exchange contents of a register-indirect address with a register	2	6
XCH	Rd, direct	Exchange contents of memory with a register	3	6
Program Branching				
BCC	rel8	Branch if the carry flag is clear	2	6t/3nt
BCS	rel8	Branch if the carry flag is set	2	6t/3nt
BEQ	rel8	Branch if the zero flag is set	2	6t/3nt
BNE	rel8	Branch if the zero flag is not set	2	6t/3nt
BG	rel8	Branch if greater than (unsigned)	2	6t/3nt
BGE	rel8	Branch if greater than or equal to (signed)	2	6t/3nt
BGT	rel8	Branch if greater than (signed)	2	6t/3nt

Table 6.5

Mnemonic		Description	Bytes	Clocks
BL	rel8	Branch if less than or equal to (unsigned)	2	6t/3nt
BLE	rel8	Branch if less than or equal to (signed)	2	6t/3nt
BLT	rel8	Branch if less than (signed)	2	6t/3nt
BMI	rel8	Branch if the negative flag is set	2	6t/3nt
BPL	rel8	Branch if the negative flag is clear	2	6t/3nt
BNV	rel8	Branch if overflow flag is clear	2	6t/3nt
BOV	rel8	Branch if overflow flag is set	2	6t/3nt
BR	rel8	Short unconditional branch	2	6
CALL	[Rs]	Subroutine call indirect with a register	2	8/5(PZ)
CALL	rel16	Relative call (+/- 64K)	3	7/4(PZ)
CJNE	Rd,direct,rel8	Compare direct byte to register and jump if not equal	4	10t/7nt
CJNE	Rd,#data8,rel8	Compare immediate byte to register and jump if not equal	4	9t/6nt
CJNE	Rd,#data16,rel8	Compare immediate word to register and jump if not equal	5	9t/6nt
CJNE	[Rd],#data8,rel8	Compare immediate word to register-indirect and jump if not equal	4	10t/7nt
CJNE	[Rd],#data16,rel8	Compare immediate word to register-indirect and jump if not equal	5	10t/7nt
DJNZ	Rd,rel8	Decrement register and jump if not zero	3	8t/5nt
DJNZ	direct,rel8	Decrement memory and jump if not zero	4	9t/5nt
FCALL	addr24	Far call (anywhere in the 24-bit address space)	4	12/8 (PZ)
FJMP	addr24	Far jump (anywhere in the 24-bit address space)	4	6
JB	bit,rel8	Jump if bit set	4	10t/6nt
JBC	bit,rel8	Jump if bit set and then clear the bit	4	11t/7nt
JMP	rel16	Long unconditional branch	3	6
JMP	[Rs]	Jump indirect to the address in the register (64K)	2	7
JMP	[A+DPTR]	Jump indirect relative to the DPTR	2	5
JMP	[[Rs+]]	Jump double-indirect to the address (pointer to a pointer)	2	8

Table 6.5

Mnemonic		Description	Bytes	Clocks
JNB	bit,rel8	Jump if bit not set	4	10t/6nt
JNZ	rel8	Jump if accumulator not equal zero	2	6t/3nt
JZ	rel8	Jump if accumulator equals zero	2	6t/3nt
NOP		No operation	1	3
RET		Return from subroutine	2	8/6(PZ)
RETI		Return from interrupt	2	10/ 8(PZ)
Bit Manipulation				
ANL	C, bit	Logical AND bit to carry	3	4
ANL	C, /bit	Logical AND complement of a bit to carry	3	4
CLR	bit	Clear bit	3	4
MOV	C, bit	Move bit to the carry flag	3	4
MOV	bit, C	Move carry to bit	3	4
ORL	C, bit	Logical OR a bit to carry	3	4
ORL	C, /bit	Logical OR complement of a bit to carry	3	4
SETB	bit	Sets the bit specified	3	4
Exception / Trap				
BKPT		Cause the breakpoint trap to be executed.	1	23/ 19(PZ)
RESET		Causes a hardware Reset, identical to an external Reset	2	18
TRAP	#data4	Causes 1 of 16 hardware traps to be executed	2	23/ 19(PZ)

Note 1: For 8 and 16 bit shifts, it is 4+1 per additional two bits. For 32-bit shifts, it is 6+1 per additional two bits.
Note 2: 3 clocks per register pushed.
Note 3: 4 clocks for the first register and two clocks for each additional register.

ADD **Integer Addition**

Syntax: ADD dest, source

Operation: dest <- src + dest

Description: Performs a twos complement binary addition of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

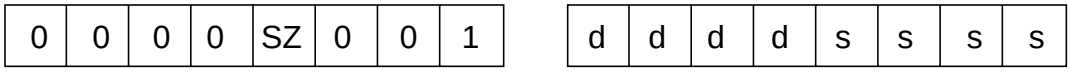
Note: If used with write to PSWL, takes precedence to flag updates

Sizes: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

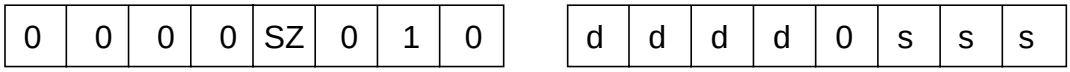
ADD Rd, Rs

Bytes**:** 2
Clocks**:** 3
Operation**:** (Rd) <-- (Rd) + (Rs)
Encoding:



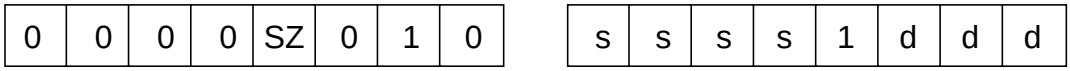
ADD Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



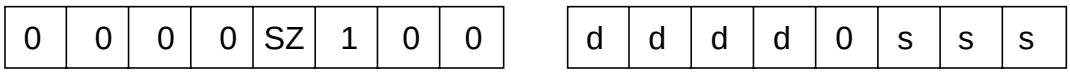
ADD [Rd], Rs

Bytes: 2
Clocks: 4
Operation**:** (WS:Rd) <-- (WS:Rd) + (Rs)
Encoding:



ADD Rd, [Rs+offset8]

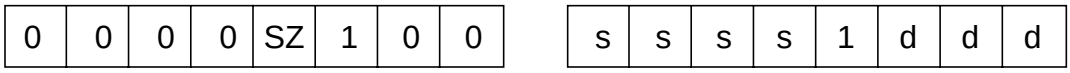
Bytes: 3
Clocks: 6
Operation**:** (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

ADD [Rd+offset8], Rs

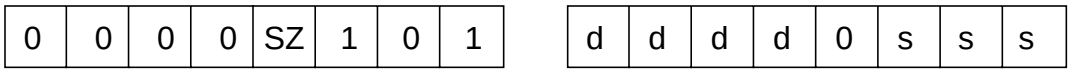
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

ADD Rd, [Rs+offset16]

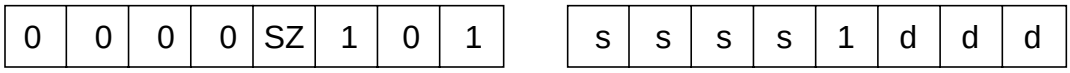
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADD [Rd+offset16], Rs

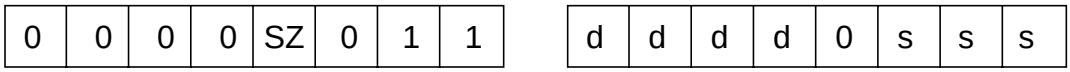
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

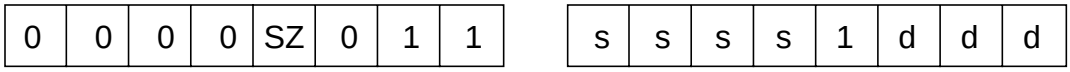
ADD Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



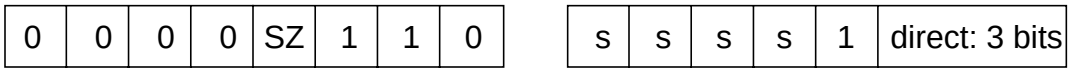
ADD [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



ADD direct, Rs

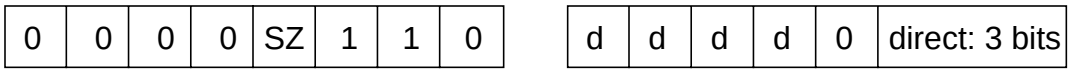
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, direct

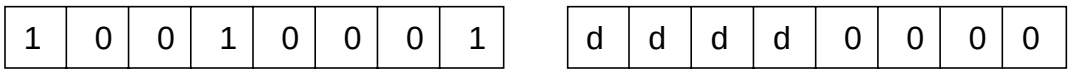
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, #data8

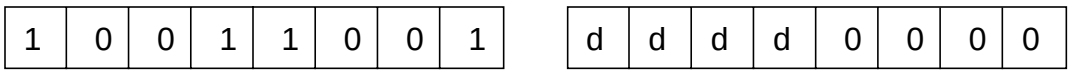
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

ADD Rd, #data16

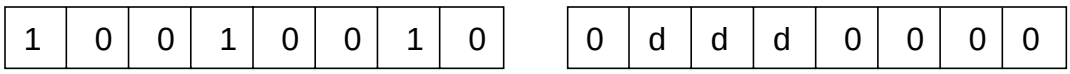
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd], #data8

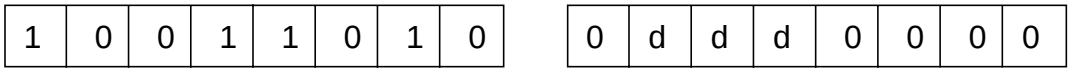
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

ADD [Rd], #data16

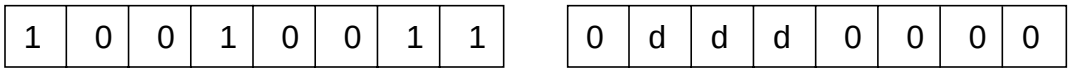
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+], #data8

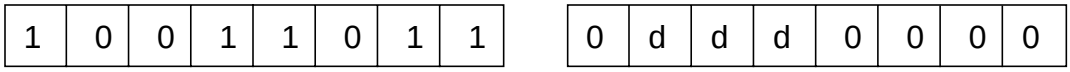
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

ADD [Rd+], #data16

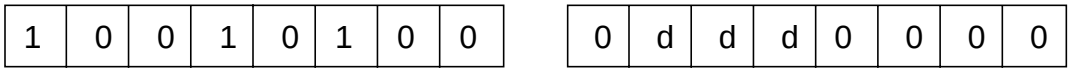
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+offset8], #data8

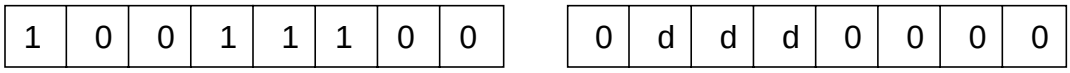
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

ADD [Rd+offset8], #data16

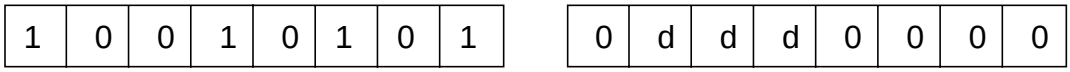
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADD [Rd+offset16], #data8

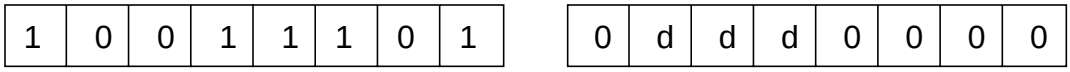
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

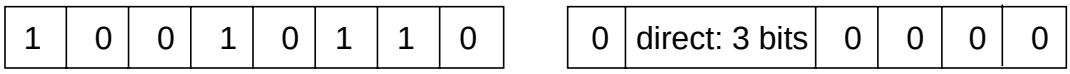
ADD [Rd+offset16], #data16

Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

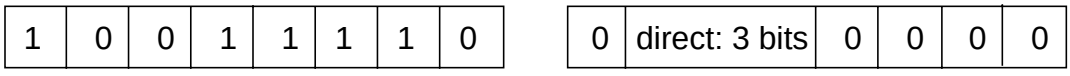
ADD direct, #data8
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADD direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ADDC dest**,** source

Operation: dest <- dest + src + C

Description: Performs a two's complement binary addition of the source operand and the previously generated carry bit with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is one (C=1), the result is greater than the sum of the operands; if it is zero (C=0), the result is the exact sum.

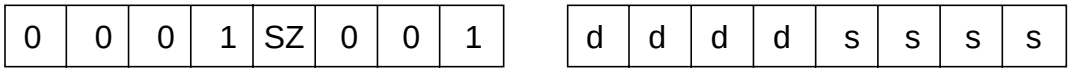
This form of addition is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then ADDC is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

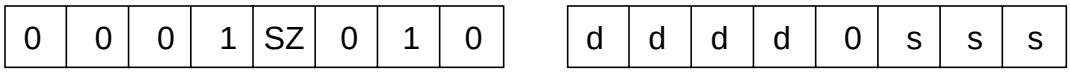
ADDC Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs) + (C)
Encoding:



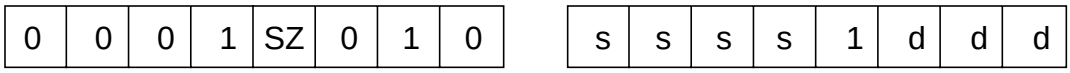
ADDC Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
Encoding:



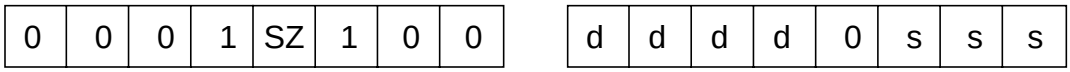
ADDC [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
Encoding:



ADDC Rd, [Rs+offset8]

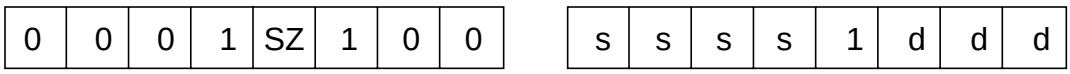
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8) + (C)
Encoding:



byte 3: offset8

ADDC [Rd+offset8], Rs

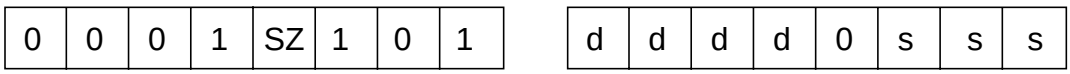
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs) + (C)
Encoding:



byte 3: offset8

ADDC Rd, [Rs+offset16]

Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16) + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC [Rd+offset16], Rs

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs) + (C)
Encoding:

0	0	0	1	SZ	1	0	1	s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	SZ	0	1	1	d	d	d	d	0	s	s	s

ADDC [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	SZ	0	1	1	s	s	s	s	1	d	d	d

ADDC direct, Rs

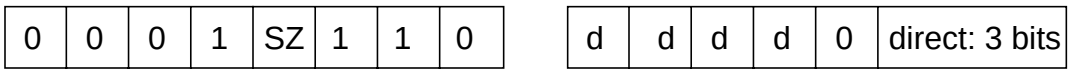
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs) + (C)
Encoding:

0	0	0	1	SZ	1	1	0	s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

ADDC Rd, direct

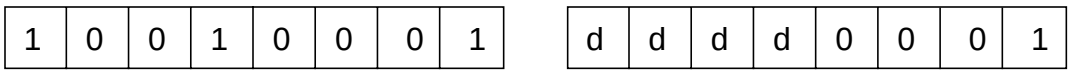
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct) + (C)
Encoding:



byte 3: lower 8 bits of direct

ADDC Rd, #data8

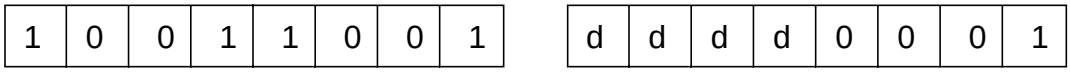
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8 + (C)
Encoding:



byte 3: #data8

ADDC Rd, #data16

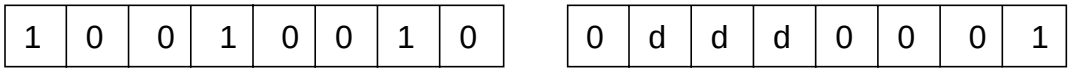
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd], #data8

Bytes: 3
Clocks : 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
Encoding:

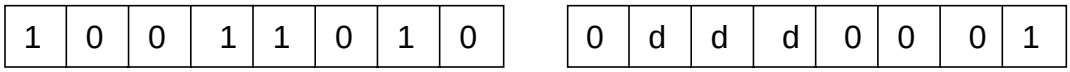


byte 3: #data8

ADDC [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)

Encoding:

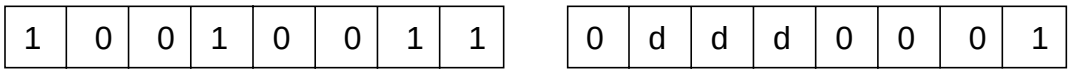


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
(Rd) <-- (Rd) + 1

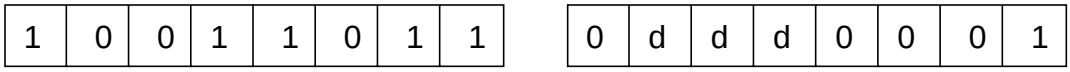
Encoding:



byte 3: #data8

ADDC [Rd+], #data16

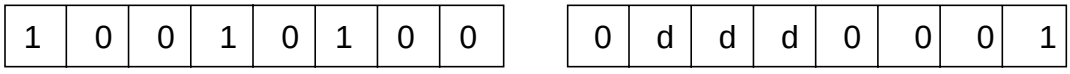
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset8], #data8

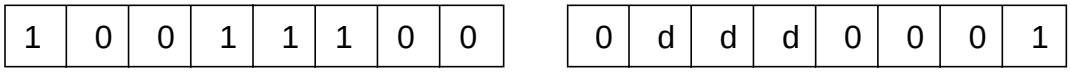
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8 + (C)
Encoding:



byte 3: offset8
byte 4: #data8

ADDC [Rd+offset8], #data16

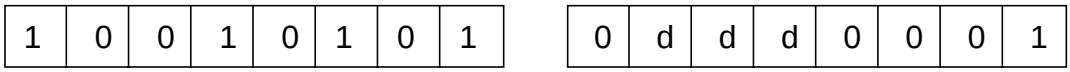
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16 + (C)
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset16], #data8

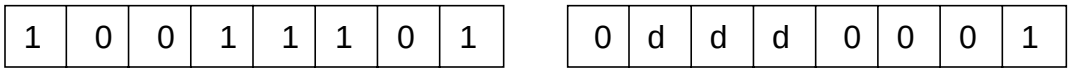
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

ADDC [Rd+offset16], #data16

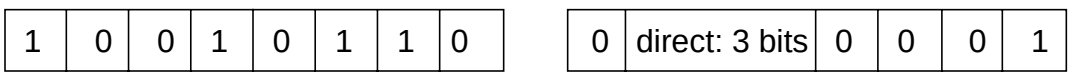
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

ADDC direct, #data8

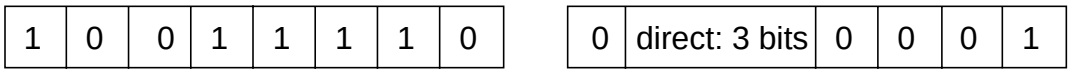
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADDC direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADDS Add Short

Syntax: ADDS dest, #value

Operation: dest <- dest + #data4

Description: Four bits of signed immediate data are added to the destination. The immediate data is sign-extended to the proper size, then added to the variable specified by the destination operand, which may be either a byte or a word. The immediate data range is +7 to -8. This instruction is used primarily to increment or decrement pointers and counters.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

(Note: the C and AC flags must not be updated by ADDS since this instruction is used to replace the 80C51 INC and DEC instructions, which do not update the flags.)

ADDS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + #data4
Encoding:

1	0	1	0	SZ	0	0	1	d	d	d	d	#data4			

ADDS [Rd], #data4

Bytes: 2
Clocks: 4
Operation:((WS:Rd)) <-- ((WS:Rd)) + #data4
Encoding:

1	0	1	0	SZ	0	1	0	0	d	d	d	#data4			

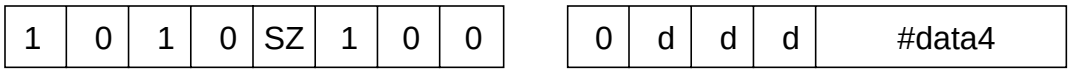
ADDS [Rd+], #data4

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

1	0	1	0	SZ	0	1	1	0	d	d	d	#data4			

ADDS [Rd+offset8], #data4

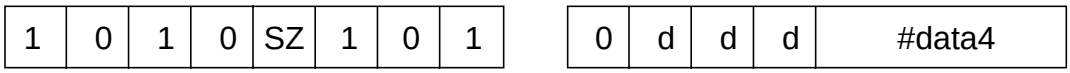
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data4
Encoding:



byte 3: offset8

ADDS [Rd+offset16], #data4

Bytes: 4
Clocks: 6
Operation:((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data4
Encoding:

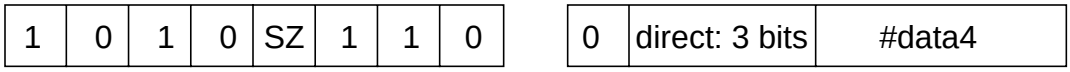


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDS direct, #data4

Bytes: 3
Clocks: 4
Operation:(direct) <-- (direct) + #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: AND dest, src

Operation: dest <- dest AND src

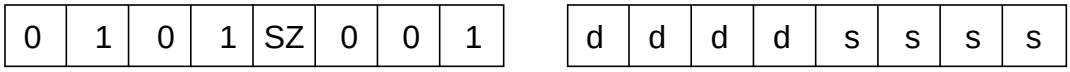
Description: Bitwise logical AND the contents of the source to the destination. The byte or word specified by the source operand is logically ANDed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

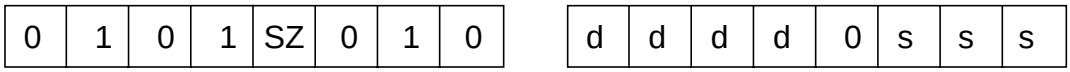
AND Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) • (Rs)
Encoding:



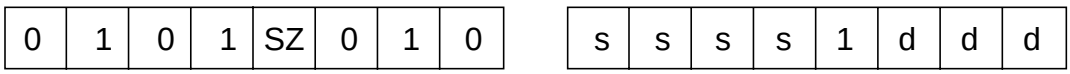
AND Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) • ((WS:Rs))
Encoding:



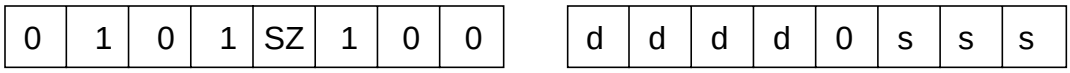
AND [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
Encoding:



AND Rd, [Rs+offset8]

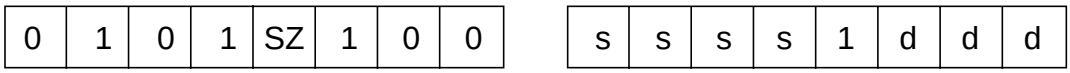
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

AND [Rd+offset8], Rs

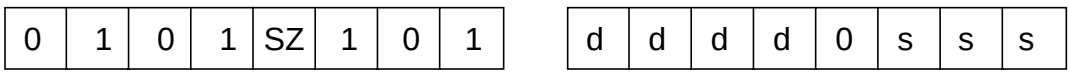
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • (Rs)
Encoding:



byte 3: offset8

AND Rd, [Rs+offset16]

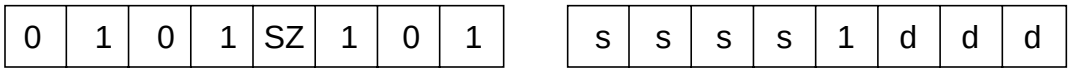
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND [Rd+offset16], Rs

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • (Rs)
Encoding:

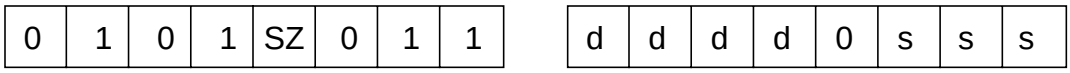


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND Rd, [Rs+]

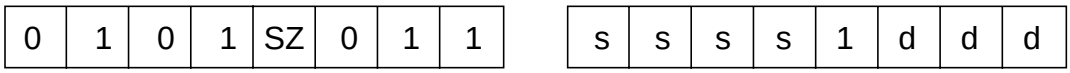
Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) • ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:



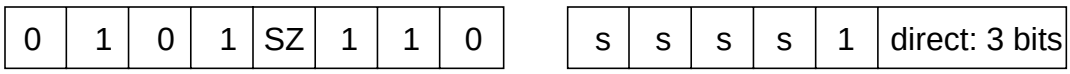
AND [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



AND direct, Rs

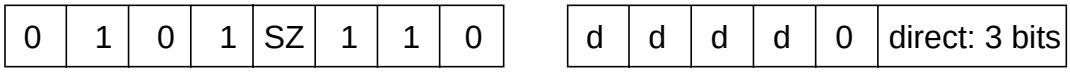
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) • (Rs)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, direct

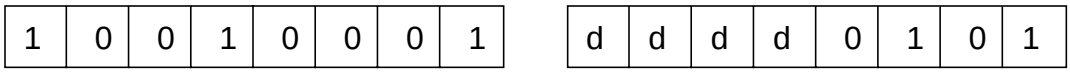
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) • (direct)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, #data8

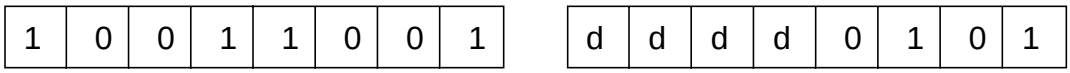
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) • #data8
Encoding:



byte 3: #data8

AND Rd, #data16

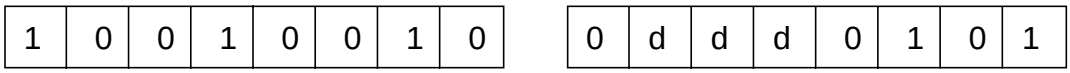
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd], #data8

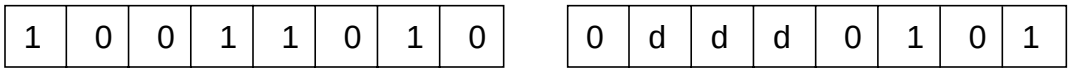
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
Encoding:



byte 3: #data8

AND [Rd], #data16

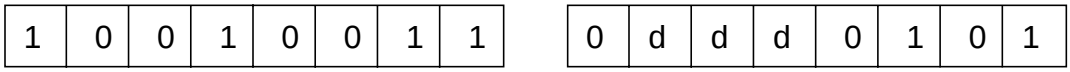
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+], #data8

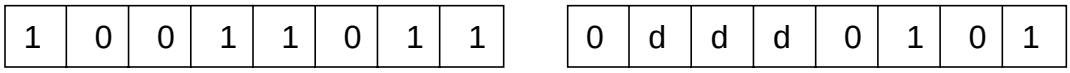
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

AND [Rd+], #data16

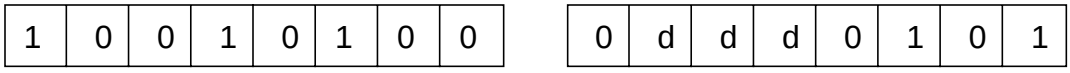
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+offset8], #data8

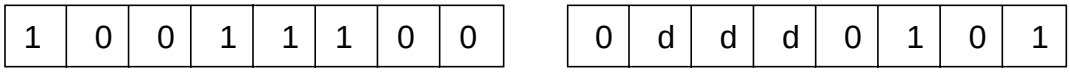
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data8
Encoding:



byte 3: offset8
byte 4: #data8

AND [Rd+offset8], #data16

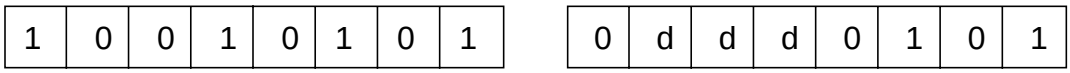
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

AND [Rd+offset16], #data8

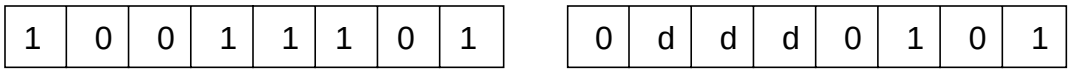
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

AND [Rd+offset16], #data16

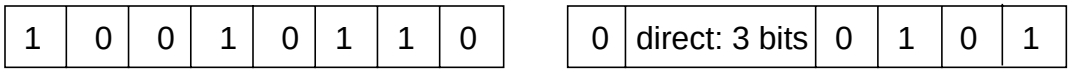
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

AND direct, #data8

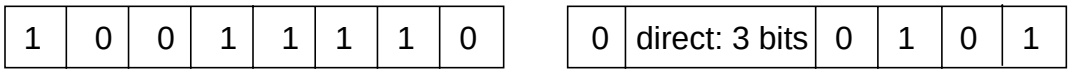
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) • #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

AND direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) • #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ANL C, bit

Operation: C <- C (AND) Bit

Description: Read the specified bit and logically AND it to the Carry flag.

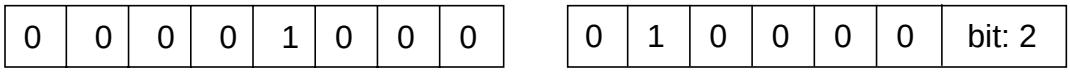
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ANL C, /bit

Operation: Carry <- C (AND) $\overline{\text{bit}}$

Description: Read the specified bit, complement it, and logically AND it to the Carry flag.

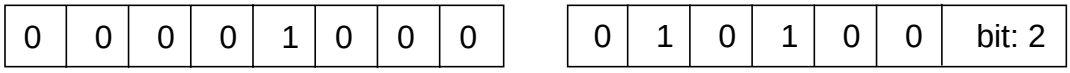
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ASL dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.msb)
(dest.bit n+1) <- (dest.bit n)
count = count-1
if sign change during shift,
(V) <- 1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted left by the number of bits specified by the count operand. The Low-order bits shifted in are zero-filled and the high-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.

The count operand could be:
- An immediate value (#data4 or #data5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count is a positive value which may be from 1 to 31 and the destination operand is a signed integer (twos complement form).The destination operand (data size) may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.The count operand is not affected by the operation.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.

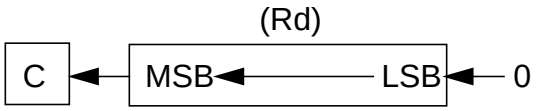
Size: Byte, word, and double word

Flags Updated: C, V, N, Z

Note: The V flag is set if the sign changes at any time during the shift operation and remains set until the end of the shift operation i.e., the V flag does not get cleared even if the sign reverts to its original state because of continued shifts within the same instruction. ASL clears the V flag if the condition to set it does not occur.

ASL Rd, Rs

Operation:

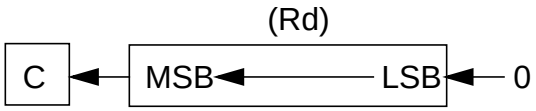


Bytes**:** 2
Clocks**:** For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift
Encoding**:**

1	1	0	0	SZ1	SZ0	0	1	d	d	d	d	s	s	s	s

ASL Rd, #data4
Rd,#data5

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift
Operation:



Encoding: (for byte and word data sizes)

1	1	0	1	SZ1	SZ0	0	1	d	d	d	d	#data4

(for double word data size)

1	1	0	1	1	1	0	1	d	d	d	#data5

Note: SZ1/SZ0 = 00 : byte operation; SZ1/SZ0 = 10 : word operation; SZ1/SZ0 = 11 : double word operation.

Syntax: ASR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
dest.msb <- Sign bit
count = count-1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted right by the number of bits specified by the count operand. The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed. To preserve the sign of the original operand, the MSBs of the result are sign-extended with the sign bit.

The count operand could be:
- An immediate value (#data4/5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count operand could be an immediate value or a register. The count is a positive value which may be from 0 to 31 and the destination operand is a signed integer. The count operand is not affected by the operation. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

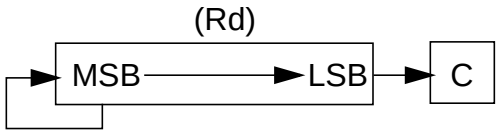
Size: Byte, Word, Double Word

Flags Updated: C, N, Z

ASR Rd, Rs

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Operation:

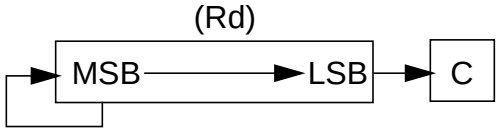


Encoding:

1	1	0	0	SZ1	SZ0	1	0	d	d	d	d	s	s	s	s

ASR Rd, #data4
Rd,#data5

Operation:



Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding: (for byte and word data sizes)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	d	#data4			

(for double word data size)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	#data5				

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

BCC **Branch if carry clear**

Syntax: BCC rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) did not generate a carry (the carry flag contains a 0). If Carry is clear, the program execution branches at the location of the PC, plus the specified displacement, rel8. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

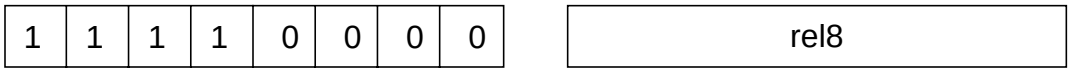
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6 (t) / 3 (nt)

Encoding:



BCS **Branch if carry set**

Syntax: BCS rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) generated a carry (the carry flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	0	0	0	1	rel8

BEQ Branch if zero

Syntax: BEQ rel8

Operation:

(PC) <-- (PC) + 2
if (Z) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a result of zero (the Z flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

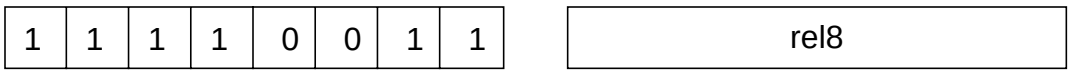
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BG **Branch if greater than (unsigned)**

Syntax: BG rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	1	0	0	0	rel8

BGE **Branch if greater than or equal to (signed)**

Syntax: BGE rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

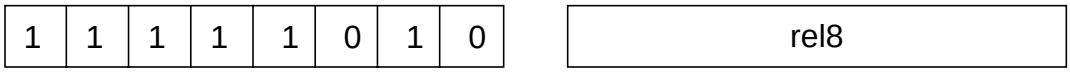
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BGT **Branch if greater than (signed)**

Syntax: BGT rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	1	1	0	0	rel8

Syntax: BKPT

Operation:
(PC) <-- (PC) + 1
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (bkpt vector)
(PC.15-0) <-- code memory (bkpt vector)
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes a breakpoint trap. The breakpoint trap acts like an immediate interrupt, using a vector to call a specific piece of code that will be executed in system mode. This instruction is intended for use in emulator systems to provide a simple method of implementing hardware breakpoints.

For a breakpoint to work properly under all conditions, it must have an instruction length no greater than the smallest other instruction on the processor, in this case the one byte NOP. This requirement exists because a breakpoint may be inserted in place of a NOP that is followed by another instruction that is branched to or otherwise executed without going through the breakpoint. If the breakpoint instruction were longer than the NOP, it would corrupt the next instruction in sequence if that instruction were executed.

The opcode for the breakpoint instruction is specifically assigned to be all ones (FFh). This is so that un-programmed EPROM code memory will contain breakpoints. Similarly, the NOP instruction is assigned to opcode 00 so that both "blank" code states map to innocuous instructions.

Size: None

Flags Updated: none[5]

Bytes: 1
Clocks: 23/19 (PZ)

Encoding:

1	1	1	1	1	1	1	1

5. All flags are affected during the PSW load from the vector table. It is possible that these flags are restored by the debugger, but does not have to be the case.

BL **Branch if less than or equal to (unsigned)**

Syntax: BL rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

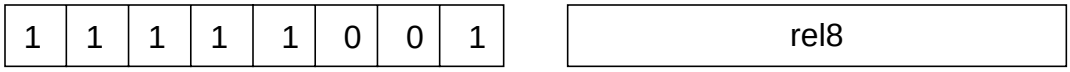
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BLE rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

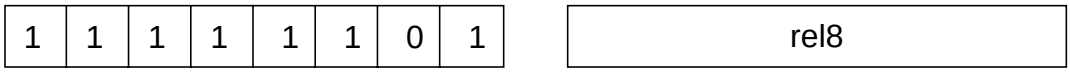
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BLT **Branch if less than (signed)**

Syntax: BLT rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

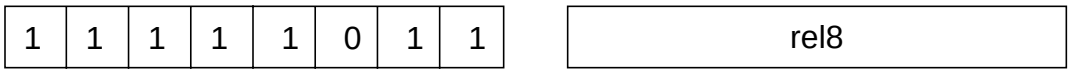
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BMI **Branch if negative**

Syntax: BMI rel8

Operation: (PC) <-- (PC) + 2
if (N) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is less than 0 (the N flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

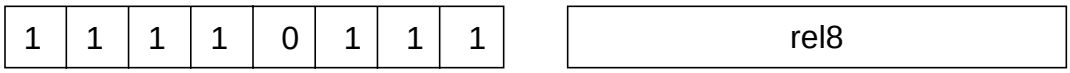
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BNE **Branch if not equal**

Syntax: BNE rel8

Operation: (PC) <-- (PC) + 2
if (Z) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a non-zero result (the Z flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

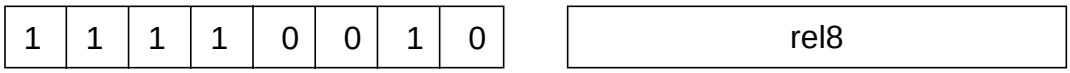
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BNV rel8

Operation: (PC) <-- (PC) + 2
if (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) did not generate an overflow (The V flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

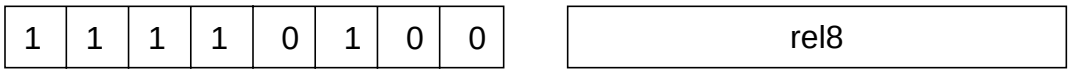
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BOV rel8

Operation: (PC) <-- (PC) + 2
if (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) generated an overflow (the V flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

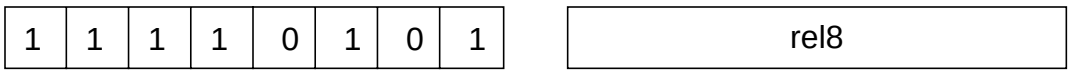
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BPL rel8

Operation: (PC) <-- (PC) + 2
if (N) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is greater than 0 (the N flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

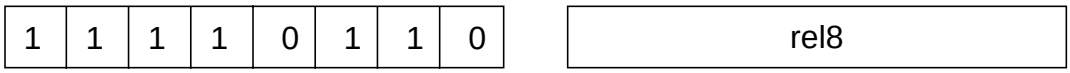
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BR rel8

Operation: (PC) <-- (PC) + 2
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

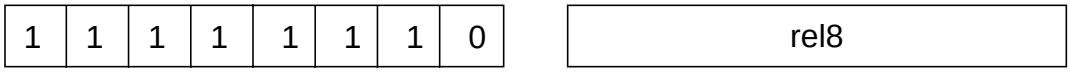
Note: Refer to section 6.3 for details of branch range

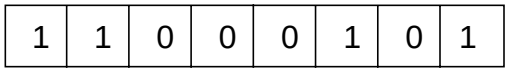
Size: None

Flags Updated: none

Bytes: 2
Clocks: 6

Encoding:





Syntax: CALL rel16

Operation: (PC) <-- (PC) + 3
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +65,534 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory. The 24-bit return address is saved on the stack.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Note: Refer to section 6.3 for details of branch range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 7/4(PZ)

Encoding:

byte 2: upper 8 bits of rel16
byte 3: lower 8 bits of rel16

Syntax: CALL [Rs]

Operation: (PC) <-- (PC) + 2
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC.15-1) <-- (Rs.15-1)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address contained in the operand register, anywhere within the 64K page following the CALL instruction.The return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned, as CALL or branch will force PC.bit0 to 0.

Note:
(1) Since the PC always points to the instruction following the CALL instruction and if that happens to be on a different page, then the called routine should be located in that page (64K)

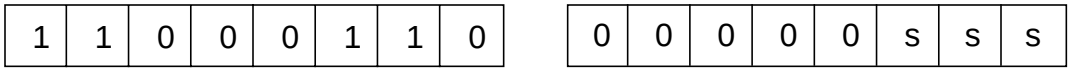
(2) if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/5(PZ)

Encoding:



CJNE **Compare and jump if not equal**

Syntax: CJNE dest, src, rel8

Operation: (PC) <-- (PC) + # of instruction bytes
(dest) - (direct) (result not stored)
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: The byte or word specified by the source operand is compared to the variable specified by the destination operand and the status flags are updated. Jump to the specified address if the values are not equal. The source and destination data are not affected by the operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Byte-Byte, Word-Word

Flags Updated: C, N, Z

(Note: this particular type of compare must not update the V or AC flags to duplicate the 80C51 function.)

CJNE Rd, direct, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

1	1	1	0	SZ	0	1	0

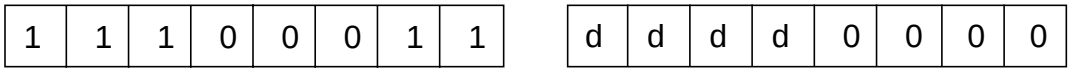
d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct
byte 4: rel8

CJNE Rd, #data8, rel8

Bytes: 4
Clocks: 9t/6nt

Encoding:

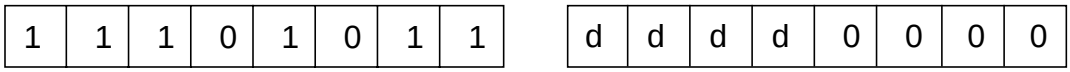


byte 3: rel8
byte 4: data#8

CJNE Rd, #data16, rel8

Bytes: 5
Clocks: 9t/6nt

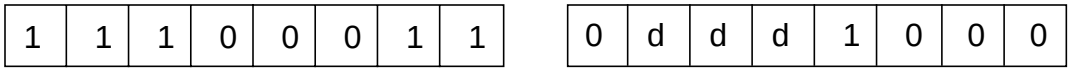
Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CJNE [Rd], #data8, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

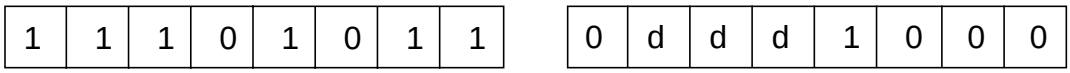


byte 3: rel8
byte 4: #data8

CJNE [Rd], #data16, rel8

Bytes: 5
Clocks: 10t/7nt

Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CLR bit

Operation: (bit) <-- 0

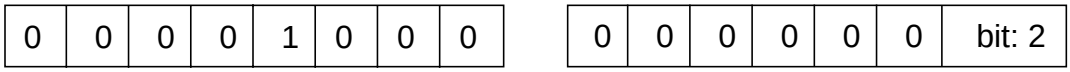
Description: Writes a 0 (clears) to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: CMP dest, src

Operation: dest - src

Description: The byte or word specified by the source operand is compared to the specified destination operand by performing a twos complement binary subtraction of src from dest. The flags are set according to the rules of subtraction. The source and destination data are not affected by the operation.

Size: byte-byte, word-word

Flags Updated: C, AC, V, N, Z

CMP Rd, Rs

Operation**:** (Rd) - (Rs)

Bytes**:** 2
Clocks**:** 3

Encoding:

0	1	0	0	SZ	0	0	1	d	d	d	d	s	s	s	s

CMP Rd, [Rs]

Operation: (Rd) - ((WS:Rs))

Bytes: 2
Clocks: 4

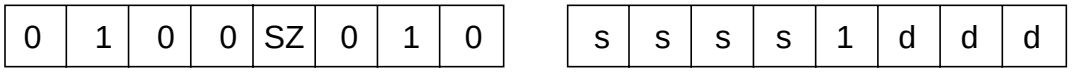
Encoding:

0	1	0	0	SZ	0	1	0	d	d	d	d	0	s	s	s

CMP [Rd], Rs

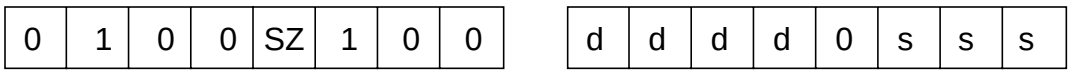
Operation: ((WS:Rd)) - (Rs)

Bytes: 2
Clocks: 4
Encoding**:**



CMP Rd, [Rs+offset8]

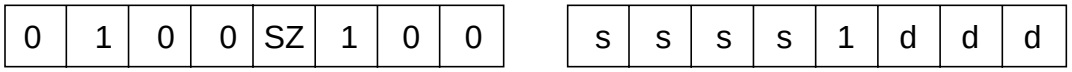
Bytes: 3
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

CMP [Rd+offset8], Rs

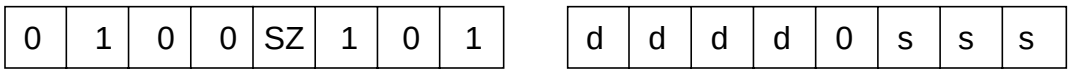
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

CMP Rd, [Rs+offset16]

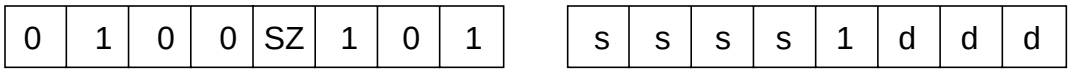
Bytes: 4
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP [Rd+offset16], Rs

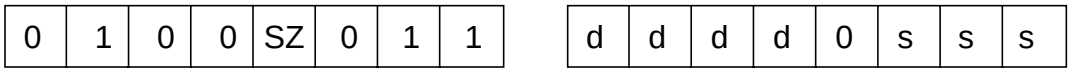
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

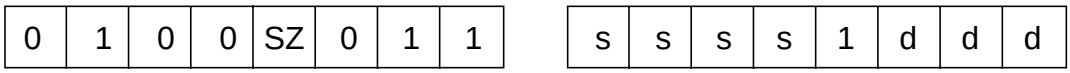


CMP [Rd+], Rs

Bytes: 2
Clocks: 5

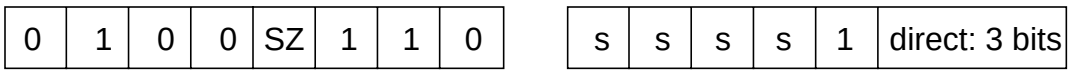
Operation: ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



CMP direct, Rs

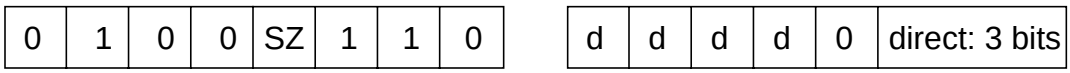
Bytes: 3
Clocks: 4
Operation: (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, direct

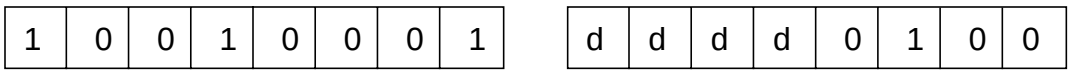
Bytes: 3
Clocks: 4
Operation: (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, #data8

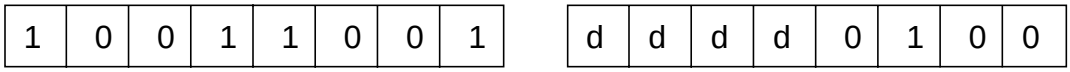
Bytes: 3
Clocks: 3
Operation: (Rd) - #data8
Encoding:



byte 3: #data8

CMP Rd, #data16

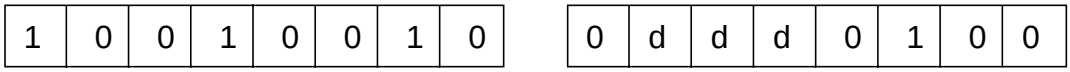
Bytes: 4
Clocks: 3
Operation: (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd], #data8

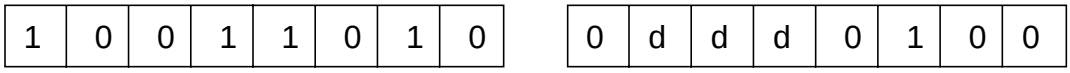
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

CMP [Rd], #data16

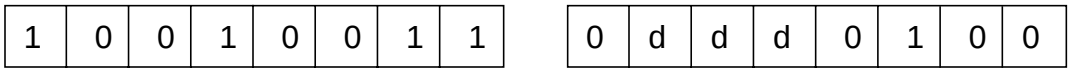
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+], #data8

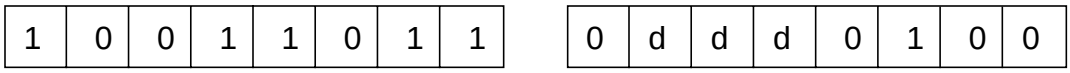
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

CMP [Rd+], #data16

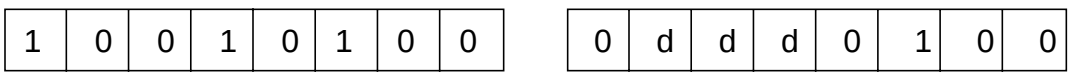
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+offset8], #data8

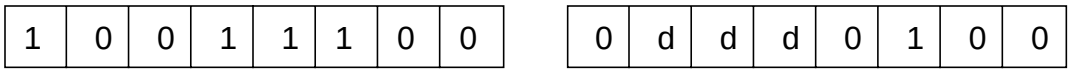
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

CMP [Rd+offset8], #data16

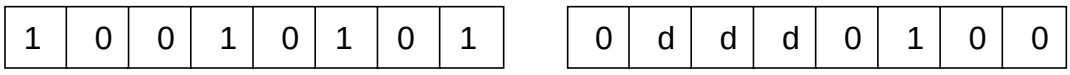
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CMP [Rd+offset16], #data8

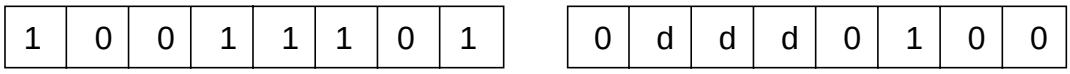
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

CMP [Rd+offset16], #data16

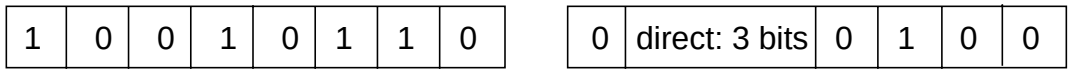
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

CMP direct, #data8

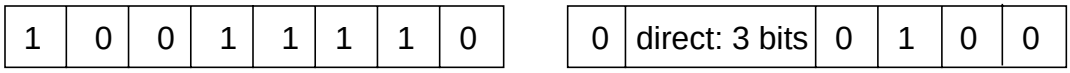
Bytes: 4
Clocks: 4
Operation: (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

CMP direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CPL Rd

Operation: Rd <-- ($\overline{Rd}$)

Description: Performs a ones complement of the destination operand specified by the register Rd. The result is stored back into Rd. The destination may be either a byte or a word.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 3

Encoding:

1	0	0	1	SZ	0	0	0

d	d	d	d	1	0	1	0

Syntax: DA Rd

Operation: if (Rd.3-0) > 9 or (AC) = 1
then (Rd.3-0) <-- (Rd.3-0) + 6
if (Rd.7-4) > 9 or (C) = 1
then (Rd.7-4) <-- (Rd.7-4) + 6

Description: Adjusts the destination register to BCD format (binary-coded decimal) following an ADD or ADDC operation on BCD values. This operation may only be done on a byte register.

If the lower 4 bits of the destination value are greater than 9, or if the AC flag is set, 6 is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value.

If the upper 4 bits of the destination value are greater than 9, or if the carry flag was set by the add to the lower bits, 60 hex is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value. Carry will never be cleared by the DA instruction if it was already set.

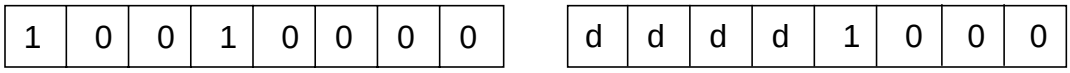
Size: Byte

Flags Updated: C, N, Z

The carry flag may be set but not cleared. See the description of the carry flag update above.

Bytes: 2
Clocks: 4

Encoding:



Note: Please refer to the table on the next page.

The following table shows the possible actions that may occur during the DA instruction, related to the input conditions.

Table 6.6

Low nibble (bits 3-0)	AC	Carry to high nibble	High nibble (bits 7-4)	Initial C flag	Number added to value	Resulting C flag
0 - 9	0	0	0 - 9	0	00	0
A - F	0	1	0 - 8	0	06	0
0 - 3 *	1	0	0 - 9	0	06	0
0 - 9	0	0	A - F	0	60	1
A - F	0	1	9 - F	0	66	1
0 - 3 *	1	0	A - F	0	66	1
0 - 9	0	0	0 - 2 **	1	60	1
A - F	0	1	0 - 2 **	1	66	1
0 - 3 *	1	0	0 - 3 ***	1	66	1

: The largest digit that could result from adding two BCD digits that caused the AC flag to be set is 3. This is with an ADDC instruction where 9 + 9 + 1 (the carry flag) = 13 hex.

** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with no carry from the bottom nibble (the AC flag = 0) is 2. For instance, 98 hex + 97 hex = 12F hex.

*** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with a carry from the bottom nibble (the AC flag = 1) is 3. For instance, 99 hex + 99 hex = 132 hex.

DIV.w	**16x8**	**Signed Division**
DIV.d	**32x16**	**Signed Division**
DIVU.b	**8x8**	**Unsigned Division**
DIVU.w	**16x8**	**Unsigned Division**
DIVU.d	**32x16**	**Unsigned Division**

Description: The byte or word specified by the source operand is divided into the variable specified by the destination operand.

For DIVU.b, the destination operand can be any byte register that is the least significant byte of a word register. For DIV.w and DIVU.w, the destination operand must be a word register, and for DIV.d and DIVU.d, the destination operand must identify a word register that is the low-word of a double-word register (see note below). The result is stored in the destination register as the quotient (8 bits for DIVU.b, DIVU.w, DIV.w, and DIVU.w, and 16-bits for DIV.d and DIVU.d) in the least significant half and the remainder (same size as the quotient), in the most significant half (except for DIVU.b which stores the quotient in the destination as identified by the lower half of a word register and the remainder at upper half of the same word register).

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte-Byte, Word-Byte, Double word-Word

Flags Updated: C, V, N, Z

The carry flag is always cleared. The V flag is set in the following cases, otherwise it is cleared:

- DIVU.b: V is set if a divide by 0 occurred. A divide by 0 also causes a hardware trap to be generated.
- DIV.w, DIVU.w: V is set if the result of the divide is larger than 8 bits (the result does not fit in the destination).
- DIV.d, DIVU.d: V is set if the result of the divide is larger than 16 bits (the result does not fit in the destination).

The Z, and N flags are set based on the quotient (integer) portion of the result only and not on the remainder.

Examples:

a) DIVU.b R4L, R4H - will store the result of the division of R4L by R4H in R4L and R4H (quotient in register R4L, remainder in register R4H).

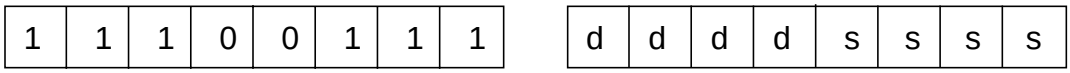
b) DIV.w R0, R2L - will store the result of word register R0 divided by byte register R2L in word register R0 (quotient in register R0L, remainder in register R0H).

c) DIV.d R4,R2 - will store the result of double-word register R5:R4 divided by word register R2 in double-word register R5:R4 (quotient in R4, remainder in R5)

Note: For all divides except DIVU.b, the destination register size is the same as indicated by the instruction (by the ".b", ".w", or ".d") and the source register is half that size.

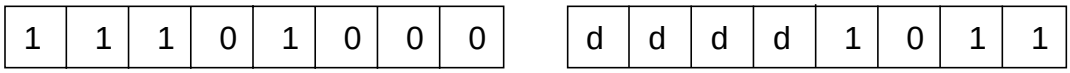
DIV.w Rd, Rs
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIV.w Rd, #data8
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

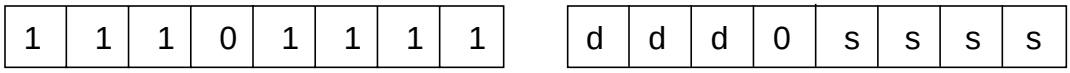
Bytes: 3
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

DIV.d Rd, Rs
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (signed divide)
(Rd+1) <-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIV.d Rd, #data16
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (signed divide)
(Rd+1) <-- 16-bit remainder of (Rd) / #data16
Encoding:

1	1	1	0	1	0	0	1	d	d	d	0	1	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DIVU.b Rd, Rs
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / (Rs)
Encoding:

1	1	1	0	0	0	0	1	d	d	d	d	s	s	s	s

DIVU.b Rd, #data8
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

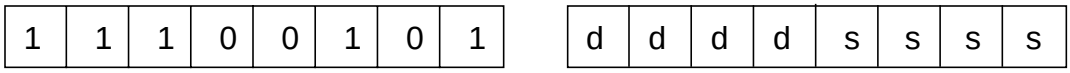
Bytes: 3
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / #data8
Encoding:

1	1	1	0	1	0	0	0	d	d	d	d	0	0	0	1

byte 3: #data8

DIVU.w Rd, Rs
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

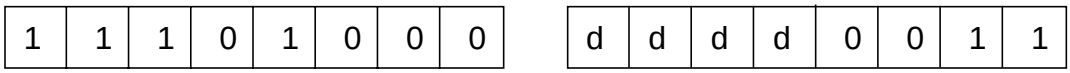
Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.w Rd, #data8
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Clocks: 12

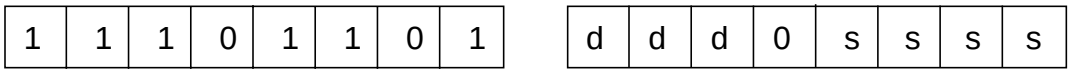
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

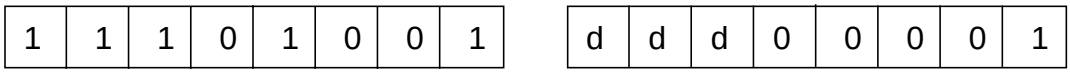
DIVU.d Rd, Rs
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (unsigned divide)
(Rd+1) <-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.d Rd, #data16
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (unsigned divide)
(Rd+1) <-- 16-bit remainder of (Rd) / #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DJNZ Decrement and jump if not zero

Syntax: DJNZ dest, rel8

Operation: (PC) <-- (PC) + 3
(dest) <-- (dest) - 1
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: Controls a loop of instructions. The parameters are: a condition code (Z), a counter (register or memory), and a displacement value. The instruction first decrements the counter by one, tests the condition if the result of decrement is 0 (for termination of the loop); if it is false, execution continues with the next instruction. If true, execution branches to the location indicated by the current value of the PC plus the sign extended displacement. The value in the PC is the address of the instruction following DJNZ.

The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.The destination operand could be byte or word.

Note: Refer to section 6.3 for details of jump range

Size: Byte, Word

Flags Updated: N, Z

DJNZ Rd, rel8

Bytes: 3
Clocks: 8t/5nt
Encoding:

1	0	0	0	SZ	1	1	1	d	d	d	d	1	0	0	0

byte 3: rel8

DJNZ direct, rel8

Bytes: 4
Clocks: 9t/5nt
Encoding:

1	1	1	0	SZ	0	1	0	0	0	0	0	1	direct: 3 bits

byte 3: lower 8 bits of direct
byte 4: rel8

Syntax: FCALL addr24

Operation: (PC) <-- (PC) + 4
(SP) <-- (SP) - 4
((SP)) <-- (PC)
(PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space. The 24-bit return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned as CALL or branch will force PC.bit0 to 0.

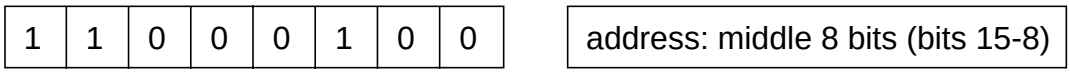
Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 12/8(PZ)

Encoding:



byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: FJMP addr24

Operation: (PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space.
Note: The target address must be word aligned as JMP always forces PC to an even address.

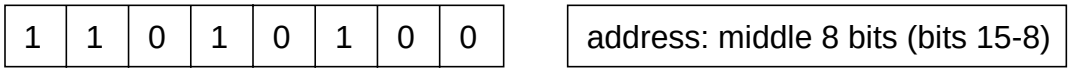
Note: if the XA is in page 0 mode, only 16-bits of the address will be used.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 6

Encoding:



byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: JB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: If the specified bit is a one, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is clear, the instruction following JB is executed.The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

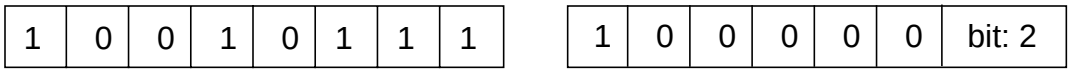
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JBC bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0; (bit) <-- 0

Description: If the bit specified is set, branch to the address pointed to by the PC plus the specified displacement. The specified bit is then cleared allowing implementation of semaphore operations. If the specified bit is clear, the instruction following JBC is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 11t/7nt

Encoding:

1	0	0	1	0	1	1	1

1	1	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JMP rel16

Operation: (PC) <-- (PC) + 3
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Jumps unconditionally. The branch range is +65,535 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory.

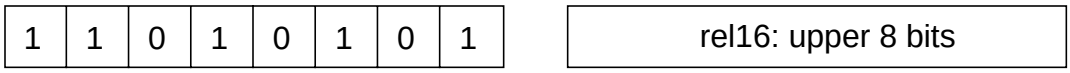
Note: Refer to section 6.3 for details of jump range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 6

Encoding:



byte 3: lower 8 bits of rel16

JMP **Jump Indirect through Register**

Syntax: JMP [Rs]

Operation: (PC) <-- (PC) + 2
(PC.15-1) <-- (Rs.15-1) (note that PC.23-16 is not affected)
(PC.0) <-- 0

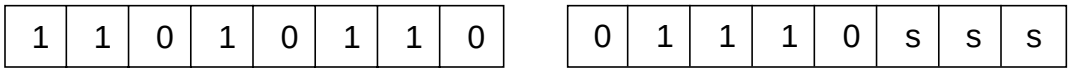
Description: Causes an unconditional branch to the address contained in the operand word register, anywhere within the 64K code page following the JMP instruction.The value of the PC used in the target address calculation is the address of the instruction following the JMP instruction.
The target address must be word aligned as JMP will force PC.bit0 to 0.

Size: none

Flags Updated: none

Bytes: 2
Clocks: 7

Encoding:



Syntax: JMP [A+DPTR]

Operation: (PC) <-- (PC) + 2
(PC15-1) <-- (A) + (DPTR)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address formed by the sum of the 80C51 compatibility registers A and DPTR, anywhere within the 64K code page following the JMP instruction. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: The target address must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 5

Note: A and DPTR are pre-defined registers used for 80C51 code translation.

Encoding:

1	1	0	1	0	1	1	0

0	1	0	0	0	1	1	0

JMP **Jump double indirect**

Syntax: JMP [[Rs+]]

Operation: (PC) <-- (PC) + 2
(PC.15-0) <-- code memory ((WS:Rs))
(PC.0) <-- 0
(Rs) <-- (Rs) + 2

Description: Causes an unconditional branch to the address contained in memory at the address pointed to by the register specified in the instruction. The specified register is post-incremented.

This 2-byte instruction may be used to compress code size by using it to index through a table of procedure addresses that are accessed in sequence. Each procedure would end with another JMP [[R+]] that would immediately go to the next procedure whose address is in the table.

The procedures must be located in the same 64K address page of the executed “Jump Double-indirect” instruction (although the table could be in any page). This instruction can result in substantial code compression and hence cost reduction through smaller memory requirements. The register pointer (index to the table) being automatically post-incremented after the execution of the instruction. The 24-bit address is identified by combining the low order 16-bit of the PC and either of high 8-bits of PC or the contents of a byte-size CS register as chosen by the program through a segment select Special Function Register (SFR).

Note: The subroutine addresses must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 8

Encoding:

1	1	0	1	0	1	1	0

0	1	1	0	0	s	s	s

Syntax: JNB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 0 then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: If the specified bit is a zero, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is set, the instruction following JB is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

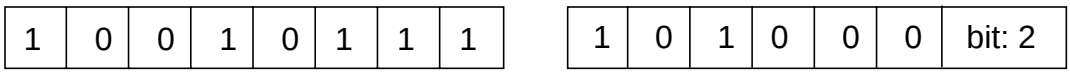
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

JNZ **Jump if the A register is not zero**

Syntax: JNZ rel8

Operation: (PC) <-- (PC) + 2
if (A) not equal to 0, then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are not zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

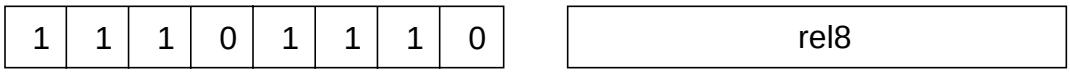
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



JZ **Jump if the A register is zero**

Syntax: JZ rel8

Operation: (PC) <-- (PC) + 2
If (A) = 0 then
(PC.15-0) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

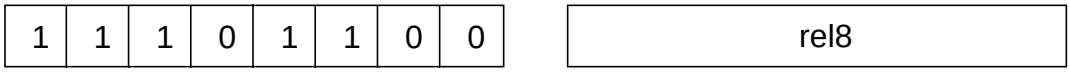
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: LEA Rd, Rs+offset8/16

Operation: (Rd) <-- (Rs)+offset8/16

Description: The word specified by the source operand is added to the offset value and the result is stored into the register specified by the destination operand. The source and destination operands are both registers. The offset value is an immediate data field of either 8 or 16 bits in length. The source data is not affected by the operation.

This instruction mimics the address calculation done during other instructions when the register indirect with offset addressing mode is used, allowing the resulting address to be saved for other purposes.

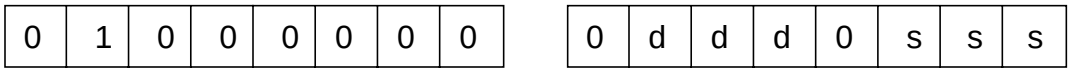
Note: The result of this operation is always a word since it duplicates the calculation of the indirect with offset addressing mode.

Size: Word-Word

Flags Updated: none

LEA Rd, Rs+offset8

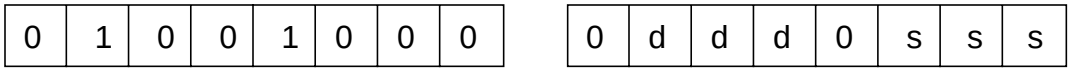
Bytes: 3
Clocks: 3
Encoding:



byte 3: offset8

LEA Rd, Rs+offset16

Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rs)+offset16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

Syntax: LSR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
(dest.msb) <- 0
count = count-1
End While

Description: If the count operand is greater than the variable specified by the destination operand is logically shifted right by the number of bits specified by the count operand. The MSBs of the result are filled with zeroes.The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.The count operand is a positive value which may be from 0 to 31. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register. The count is not affected by the operation.

Note:
- For Logical Shift Left, use ASL ignoring the N flag.
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte, Word, Double Word

Flags Updated: C, N, Z (N = 0 after an LSR unless count = 0, then it is unchanged)

LSR Rd, Rs (Rs = Byte-register)

Operation**:**

Bytes**:** 2
Clocks**:** For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding**:**

1	1	0	0	SZ1	SZ0	0	0

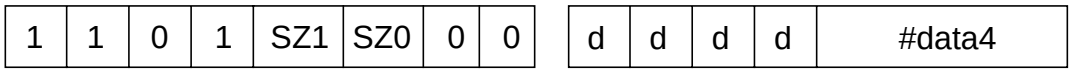
d	d	d	d	s	s	s	s

LSR Rd, #data4
Rd, #data5

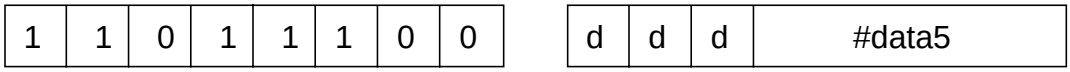
Operation:

Bytes: 2
Clocks: For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding: (for byte and word data sizes)



(for double word data size)



Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: MOV dest, src

Operation: dest <- src

Description: The byte or word specified by the source operand is copied into the variable specified by the destination operand. The source data is not affected by the operation.

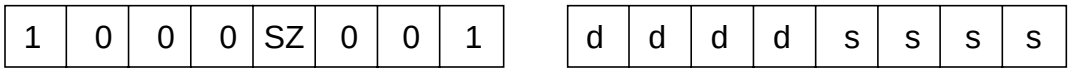
Source and destination operands may be a register in the register file, an indirect address specified by a pointer register, an indirect address specified by a pointer register added to an immediate offset of 8 or 16 bits, or a direct address. Source operands may also be specified as immediate data contained within the instruction. Auto-increment of the indirect pointers is available for simple indirect (not offset) addressing.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

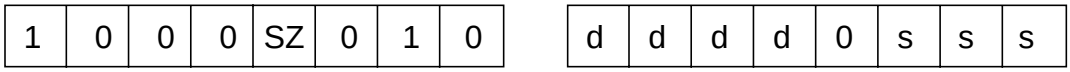
MOV Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rs)
Encoding:



MOV Rd, [Rs]

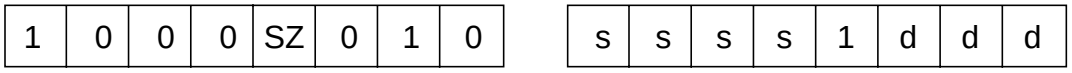
Bytes: 2
Clocks: 3
Operation: (Rd) <-- ((WS:Rs))
Encoding:



MOV [Rd], Rs

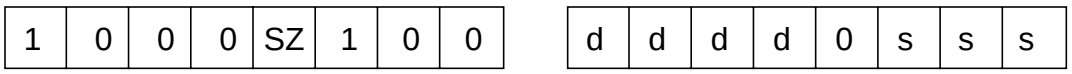
Bytes: 2
Clocks: 3

Operation: ((WS:Rd)) <-- (Rs)
Encoding:



MOV Rd, [Rs+offset8]

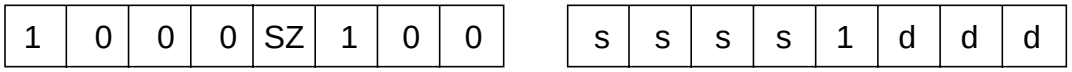
Bytes: 3
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

MOV [Rd+offset8], Rs

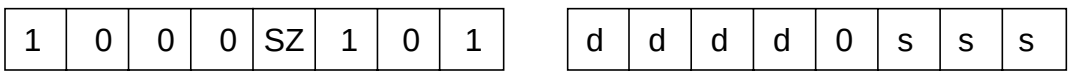
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- (Rs)
Encoding:



byte 3: offset8

MOV Rd, [Rs+offset16]

Bytes: 4
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset16)
Encoding:

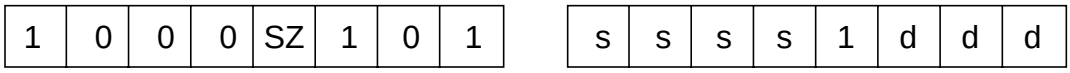


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV [Rd+offset16], Rs

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- (Rs)

Encoding:

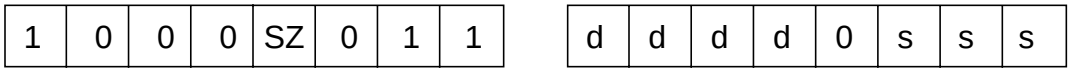


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV Rd, [Rs+]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:

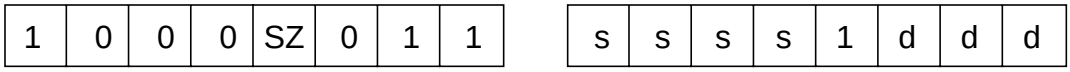


MOV [Rd+], Rs

Bytes: 2
Clocks: 4

Operation: ((WS:Rd)) <-- (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

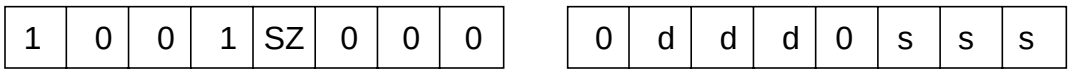
Encoding:



MOV [Rd+], [Rs+]

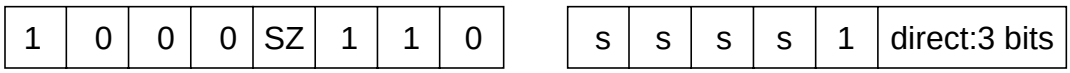
Bytes: 2
Clocks: 6
Operation: ((WS:Rd)) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



MOV direct, Rs

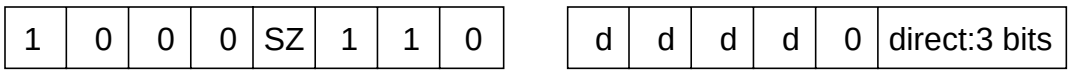
Bytes: 3
Clocks: 4
Operation: (direct) <-- (Rs)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, direct

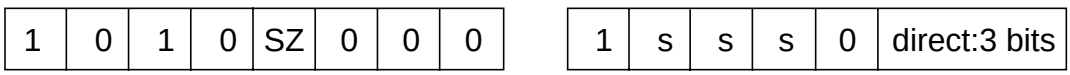
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV direct, [Rs]

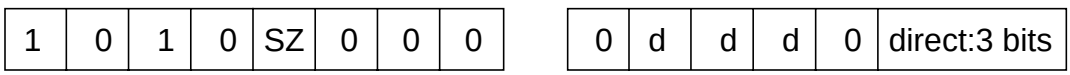
Bytes: 3
Clocks: 4
Operation: (direct) <-- ((WS:Rs))
Encoding:



byte 3: lower 8 bits of direct

MOV [Rd], direct

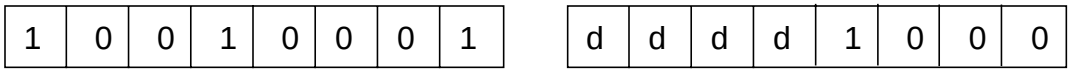
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, #data8

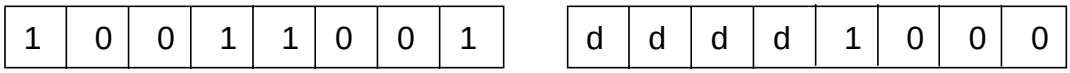
Bytes: 3
Clocks: 3
Operation: (Rd) <-- #data8
Encoding:



byte 3: #data8

MOV Rd, #data16

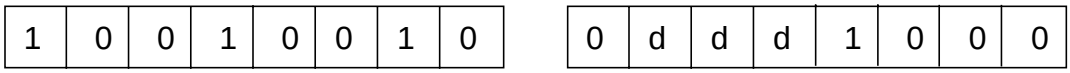
Bytes: 4
Clocks: 3
Operation: (Rd) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd], #data8

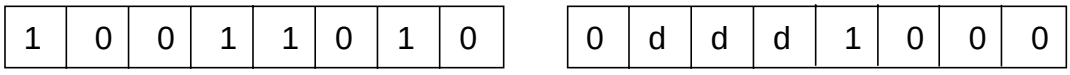
Bytes: 3
Clocks: 3
Operation: ((WS:Rd)) <-- #data8
Encoding:



byte 3: #data8

MOV [Rd], #data16

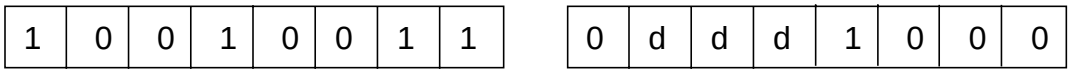
Bytes: 4
Clocks: 3
Operation: ((WS:Rd)) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+], #data8

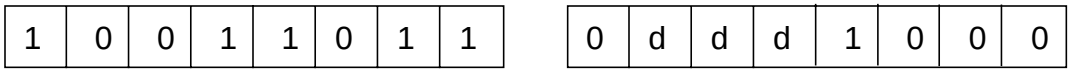
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

MOV [Rd+], #data16

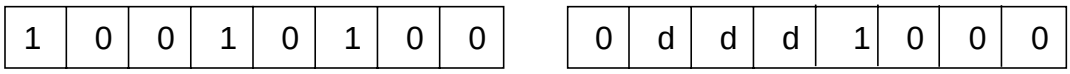
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+offset8], #data8

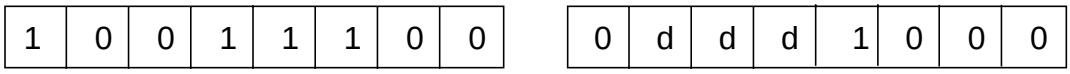
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data8
Encoding:



byte 3: offset8
byte 4: #data8

MOV [Rd+offset8], #data16

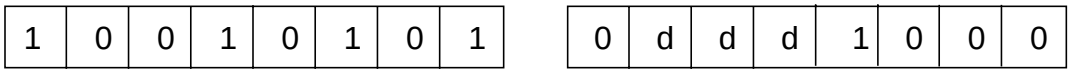
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV [Rd+offset16], #data8

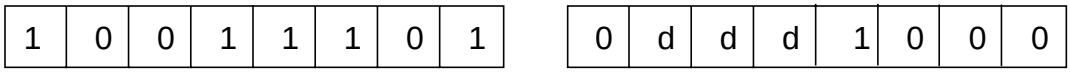
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

MOV [Rd+offset16], #data16

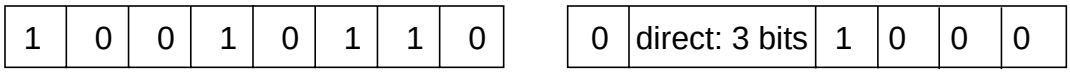
Bytes: 6
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

MOV direct, #data8

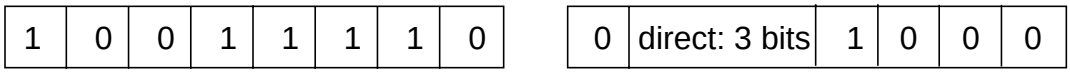
Bytes: 4
Clocks: 3
Operation: (direct) <-- #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

MOV direct, #data16

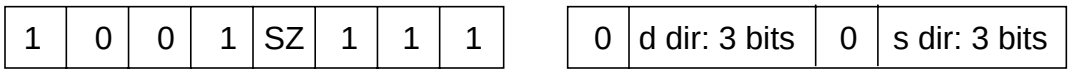
Bytes: 5
Clocks: 3
Operation: (direct) <-- #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV direct, direct

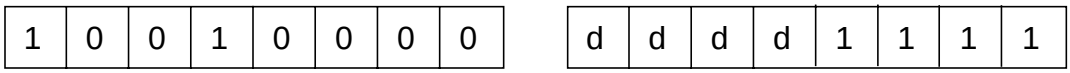
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct (dest)
byte 4: lower 8 bits of direct (src)

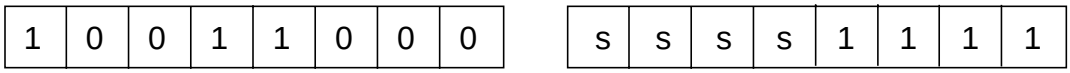
MOV Rd, USP (move from user stack pointer)

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (USP)
Encoding:



MOV USP, Rs (move to user stack pointer)

Bytes: 2
Clocks: 3
Operation: (USP) <-- (Rs)
Encoding:



Syntax: MOV C, bit

Operation: (C) <-- (bit)

Description: Copies the specified bit to the carry flag.

Size: Bit

Flags Updated: none

Note: C is written as the destination of the move, not as a status flag

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: MOV bit, C

Operation: (bit) <-- (C)

Description: Copies the carry flag to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: MOVC Rd, [Rs+]

Operation: (Rd) <-- code memory ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

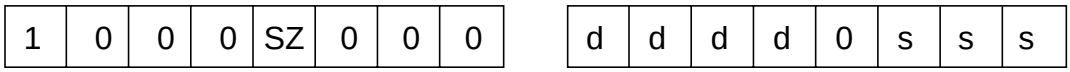
Description: Contents of code memory are copied to an internal register. The byte or word specified by the source operand is copied to the variable specified by the destination operand. In the case of MOVC, the pointer segment selection gives the choices of PC_{23-16} or CS segment (current *working segment* referred here as WS), rather than DS or ES as is used for all other instructions.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4

Encoding:



Syntax: MOVC A, [A+DPTR]

Operation: PC <- PC+2
(A) <-- code memory (PC.23-16:(A) + (DPTR))

Description: The byte located at the code memory address formed by the sum of A and the DPTR is copied to the A register. The A and DPTR registers are pre-defined registers used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	1	0

.

Syntax: MOVC A, [A+PC]

Operation: PC <- PC+2
(A) <-- code memory [PC.23-16: (A +PC.15-0)]

Note: Only 16-bits of A+PC are used

Description: The byte located at the code memory address formed by the sum of A and the current Program Counter value is copied to the A register. The A register is a pre-defined register used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	0	0

Syntax: MOVS dest, #data

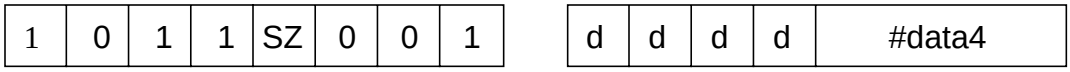
Description: Four bits of signed immediate data are moved to the destination. The immediate data is sign-extended to the proper size, then moved to the variable specified by the destination operand, which may be a byte or a word. The immediate data range is +7 to -8. This instruction is used to save time and code space for the many instances where a small data constant is moved to a destination.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

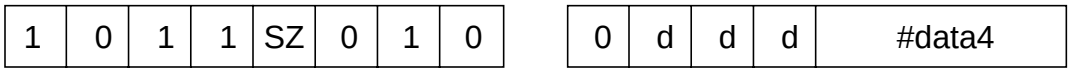
MOVS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- sign-extended #data4
Encoding:



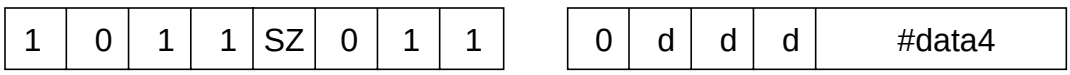
MOVS [Rd], #data4

Bytes: 2
Clocks: 3
Operation: ((WS:Rd)) <-- sign-extended #data4
Encoding:



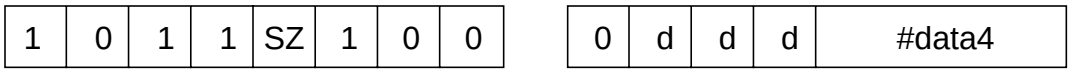
MOVS [Rd+], #data4

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- sign-extended #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



MOVS [Rd+offset8], #data4

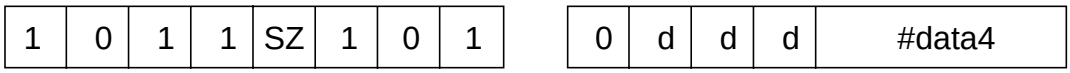
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- sign-extended #data4
Encoding:



byte 3: offset8

MOVS [Rd+offset16], #data4

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- sign-extended #data4
Encoding:

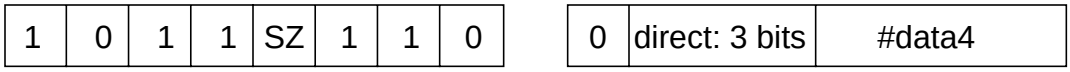


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOVS direct, #data4

Bytes: 3
Clocks: 3
Operation: (direct) <-- sign-extended #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: MOVX dest, src

Description: Move external data to or from an internal register. The byte or word specified by the source operand is copied into the variable specified by the destination operand. This instruction allows access to data external to the microcontroller in the address range of 0 to 64K. The standard indirect move may access external data only above the boundary where internal data RAM ends, whereas MOVX always forces an external access. MOVX only operates on the first 64K of external data memory. This instruction is included to allow compatibility with 80C51 code.

Note that in the 80C51 MOVX instruction using @Ri as a pointer (where i could be 0 or 1), the pointer was eight bits in length and the upper address lines were not driven on the external bus. The XA always drives all of the enabled external bus address lines. The use of the pointer depends on whether compatibility mode is in use. If CM = 0 (compatibility mode off, the default), 16 bits of R0 or R1 are used as the address within data segment 0. If CM = 1 (compatibility mode on), 8 bits of R0L or R0H are used as the bottom eight bits of the address, while the remainder of the address bits, including those corresponding to the data segment are 0.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOVX Rd, [Rs]

Bytes: 2
Clocks: 6
Operation: (Rd) <-- external data memory ((Rs))

Encoding:

1	0	1	0	SZ	1	1	1		d	d	d	d	0	s	s	s

MOVX [Rd], Rs

Bytes: 2
Clocks: 6
Operation: external data memory ((Rd)) <-- (Rs)
Encoding:

1	0	1	0	SZ	1	1	1		s	s	s	s	1	d	d	d

MUL.w 16x16 Signed Multiply
MULU.b 8x8 Unsigned Multiply
MULU.w 16x16 Unsigned Multiply

Description: The byte or word specified by the source operand is multiplied by the variable specified by the destination operand.

The destination operand must be the first half of a double size register (word for a byte multiply and double word for a word multiply). The result is stored in the double size register.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, and R7:R6).

Size: Byte-Byte, Word-Word

Flags Updated: C, V, N, Z

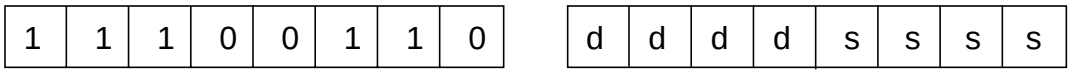
The carry flag is always cleared by a multiply instruction. The V flag is set in the following cases, otherwise it is cleared:
- MULU.b: V is set if the result of the multiply is greater than FFh (the upper byte is not equal to 0).
- MULU.w: V is set if the result of the multiply is greater than FFFFh (the upper word is not equal to 0).
- MUL.w: V is set if the absolute value of the result of the multiply is greater than 7FFFh (the upper word is not a sign extension of the lower word).

Examples:
a) MUL.w R0,R5 stores the product of word register 0 and word register 5 in double word register 0 (least significant word in word register R0, most significant word in word register R1).

b) MULU.b R4L, R4H will store the MS byte of the product of R4L and R4H in R4H and the LS byte in R4L.

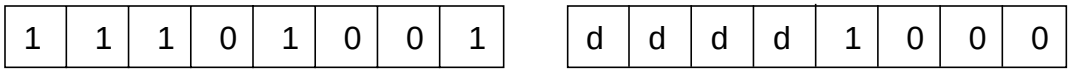
MUL.w Rd, Rs
(signed 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * (Rs) (signed multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:



MUL.w Rd, #data16
(signed 16 bits * 16 bits --> 32 bits)

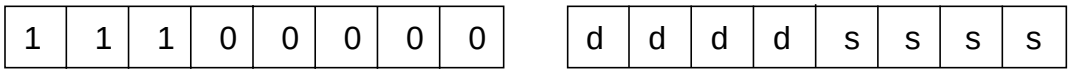
Bytes: 4
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * #data16 (signed multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MULU.b Rd, Rs
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 2
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * (Rs) (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * (Rs)
Encoding:



MULU.b Rd, #data8
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 3
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * #data8 (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * #data8
Encoding:

1	1	1	0	1	0	0	0	d	d	d	d	0	0	0	0

byte 3: #data8

MULU.w Rd, Rs
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * (Rs) (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:

1	1	1	0	0	1	0	0	d	d	d	d	s	s	s	s

MULU.w Rd, #data16
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 4
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * #data16 (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:

1	1	1	0	1	0	0	1	d	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

NEG **Negate**

Syntax: NEG Rd

Operation: Rd <-- ($\overline{Rd}$) + 1

Description: The destination register is negated (twos complement). The destination may be a byte or a word.

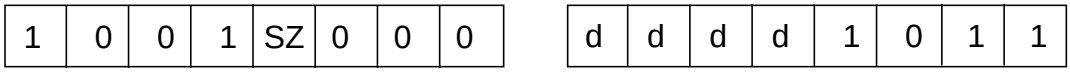
Size: Byte, Word

Flags Updated: V, N, Z

The V flag is set if a twos complement overflow occurred: the original value = result = 8000 hex for a word operation or 80 hex for a byte operation.

Bytes: 2
Clocks: 3

Encoding:



NOP **No Operation**

Syntax: NOP

Operation: PC <- PC + 1

Description: Execution resumes at the following instruction. This instruction is defined as being one byte in length in order to allow it to be used to force word alignment of instructions that are branch targets, or for any other purpose. It may also be used to as a delay for a predictable amount of time.

Size: None

Flags Updated: none

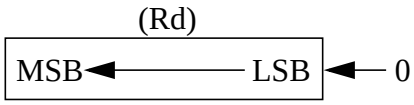
Bytes: 1
Clocks: 3

Encoding:

0	0	0	0	0	0	0	0

Syntax: NORM Rd, Rs

Operation:



Description: Logically shifts left the contents of the destination until the MSB is set, storing the number of shifts performed in the count (source) register. The data size may be 8, 16, or 32 bits.

If the destination value already has the MSB set, the count returned will be 0. If the destination value is 0, the count returned will be 0, the N flag will be cleared, and the Z flag will be set. For all other conditions, the N flag will be 1 and the Z flag will be 0.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
The last pair, i.e, R7:R6 is probably not a good idea as R7 is the current stack pointer.

Size: Byte, Word, Double Word

Flags Updated: N, Z

Bytes: 2
Clocks: For 8 or 16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding:

1	1	0	0	SZ1	SZ0	1	1

d	d	d	d	s	s	s	s

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: OR dest, src

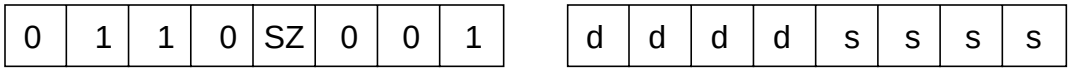
Description: Bitwise logical OR the contents of the source to the destination. The byte or word specified by the source operand is logically ORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

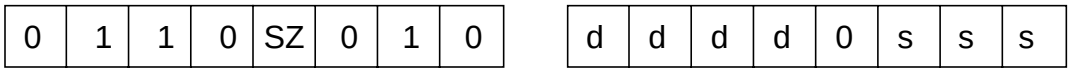
OR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:



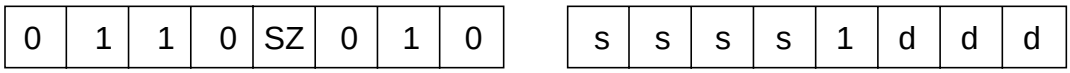
OR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



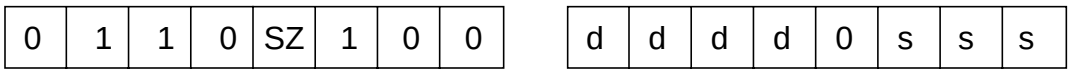
OR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
Encoding:



OR Rd, [Rs+offset8]

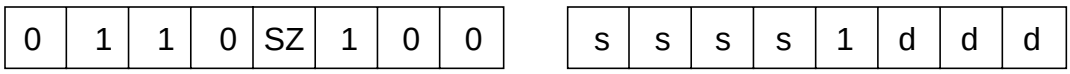
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

OR [Rd+offset8], Rs

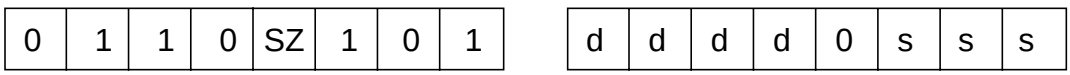
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

OR Rd, [Rs+offset16]

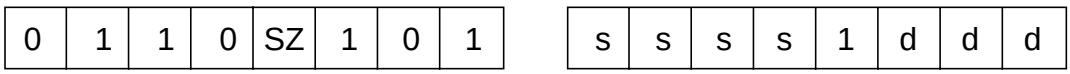
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

OR [Rd+offset16], Rs

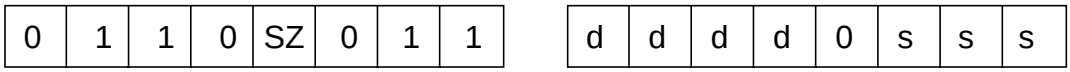
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

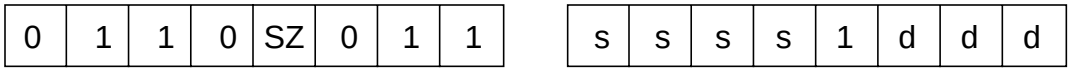
OR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



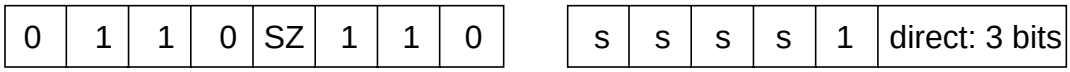
OR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



OR direct, Rs

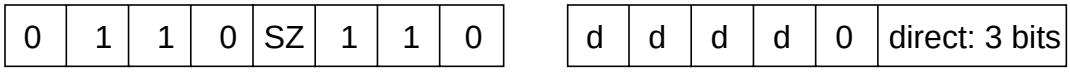
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, direct

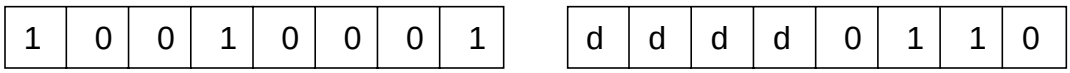
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, #data8

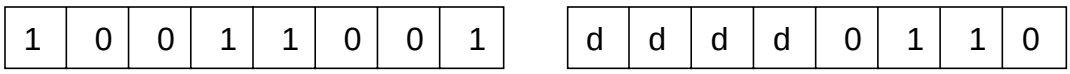
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

OR Rd, #data16

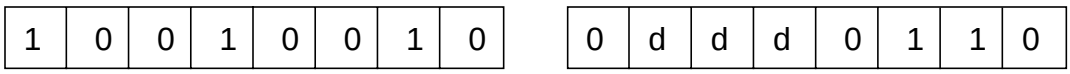
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd], #data8

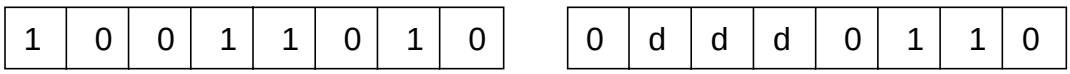
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

OR [Rd], #data16

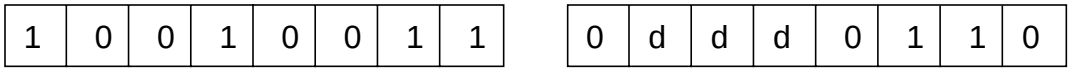
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+], #data8

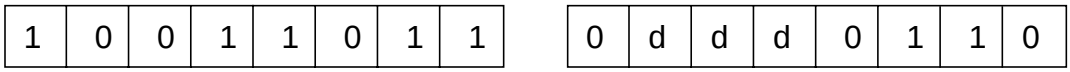
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

OR [Rd+], #data16

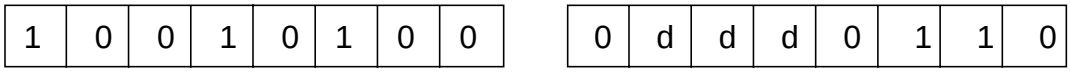
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+offset8], #data8

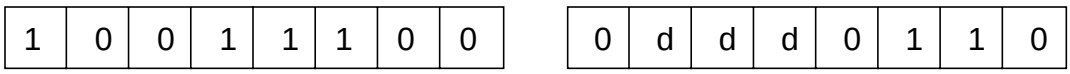
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

OR [Rd+offset8], #data16

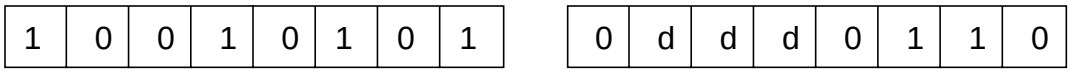
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

OR [Rd+offset16], #data8

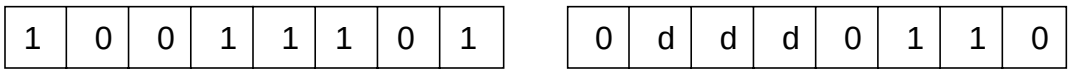
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

OR [Rd+offset16], #data16

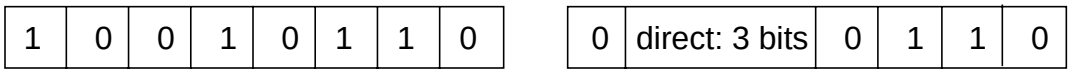
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

OR direct, #data8

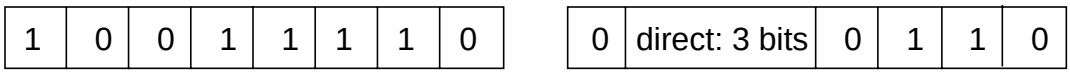
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

OR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ORL C, bit

Operation:(C) <-- (C) + (bit)

Description: Logical (inclusive) OR a bit to the Carry flag. Read the specified bit and logically OR it to the Carry flag.
(C is written as the destination of the ORL, not as a status flag)

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

ORL **Logical OR complement of bit**

Syntax: ORL C, /bit

Operation: (C) <-- (C) + ($\overline{bit}$)

Description: Logically OR the complement of a bit to the Carry flag. Read the specified bit, complement it, and logically OR it to the Carry flag.
(C is written as the destination of the move, not as a status flag)

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

POP **Pop**
POPU **Pop User**

Syntax: POP dest

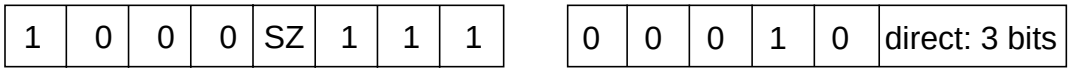
Description: The stack is popped and the data written to the specified directly addressed location. The data size may be byte or word. POP uses the current stack pointer, while POPU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

POP direct

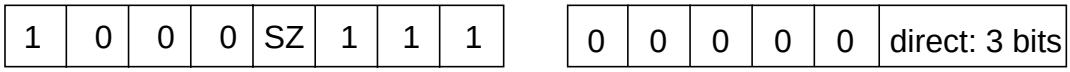
Bytes: 3
Clocks: 5
Operation: (direct) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



byte 3: 8 bits of direct

POPU direct

Bytes: 3
Clocks: 5
Operation: (direct) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



byte 3: 8 bits of direct

POP **Pop Multiple**
POPU **Pop User Multiple**

Syntax: POP Rlist
POPU Rlist

Description: Pop the specified registers (one or more) from the stack. The stack is popped (from 1 to 8 times) and the data stored in the specified registers. Any combination of word registers in the group R0 to R7 may be popped in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be popped in a single instruction in a byte operation. POP uses the current stack pointer, while POPU forces an access to the user stack.

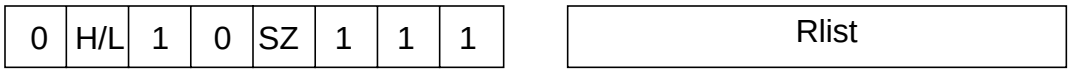
Note: Rlist is a bit map that represents each register to be popped. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The pop order is from right to left, i.e., the register specified by the rightmost one in Rlist will be popped first, etc. The order must be the reverse of that used by the preceding PUSH instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

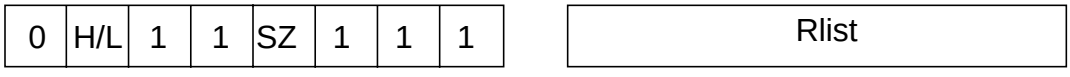
POP Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



POPU Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



Rlist bit definitions for a byte POP from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte POP from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word POP from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

PUSH **Push**
PUSHU **Push User**

Syntax: PUSH src
PUSHU src

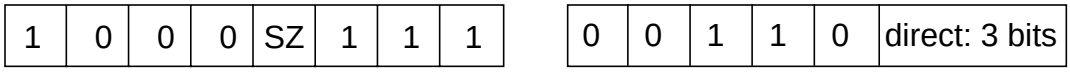
Description: The specified directly addressed data is pushed onto the stack. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

PUSH direct

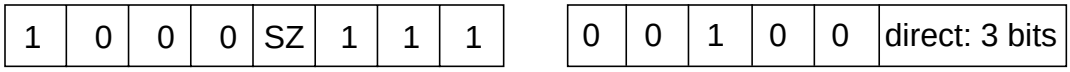
Bytes: 3
Clocks: 5
Operation: (SP) <-- (SP) - 2
((SP)) <-- (direct)
Encoding**:**



byte 3: 8 bits of direct

PUSHU direct

Bytes: 3
Clocks: 5
Operation: (USP) <-- (USP) - 2
((USP)) <-- (direct)
Encoding:



byte 3: 8 bits of direct

PUSH Push Multiple
PUSHU Push User Multiple

Syntax: PUSH Rlist
PUSHU Rlist

Description: Push the specified registers (one or more) onto the stack. The specified registers are pushed onto the stack. Any combination of word registers in the group R0 to R7 may be pushed in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be pushed in a single instruction in a byte operation. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

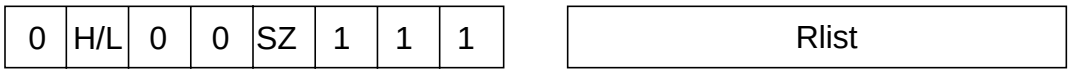
Note: Rlist is a bit map that represents each register to be pushed. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The push order is from left to right, i.e., the register specified by the leftmost one in Rlist will be pushed first, etc. The order must be the reverse of that used by the corresponding POP instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

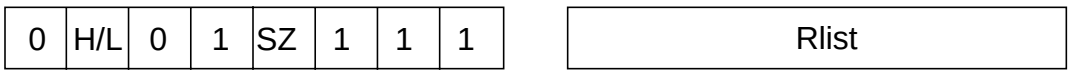
PUSH Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(SP) <-- (SP) - 2
((SP)) <-- (Ri)
Encoding:



PUSHU Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(USP) <-- (USP) - 2
((USP)) <-- (Ri)
Encoding:



Rlist bit definitions for a byte PUSH from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte PUSH from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word PUSH from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

RESET Software Reset

Syntax: RESET

Operation: (PC) <-- vector(0)
(PSW) <-- vector(0)
(SFRs) <-- reset values (refer to the description of reset for details)

Description: The chip is reset exactly as if the external hardware reset has been asserted with the exception that it does not sample inputs for configuration, e.g., $\overline{EA}$, BUSW, etc. When a RESET instruction is executed, the chip is internally reset, but no external $\overline{RESET}$ pulse is generated. The above inputs which are latched during rising edge of a $\overline{RESET}$ pulse, hence does not affect the chip configuration.

Flags Updated: The entire PSW is set to the value specified in the reset vector.

Bytes: 2
Clocks: 18

Encoding:

1	1	0	1	0	1	1	0

0	0	0	1	0	0	0	0

Syntax: RET

Operation: (PC) <-- ((SP))
(SP) <-- (SP) + 4

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value (PC_{23-0}). This instruction is used to return from a subroutine that was called with a CALL or Far Call (FCALL).

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/6 (PZ)

Encoding:

1	1	0	1	0	1	1	0

1	0	0	0	0	0	0	0

Syntax: RETI

Operation: (PSW) <-- ((SSP))
(PC.23-0) <-- ((SSP))
(SSP) <-- (SSP) + 6

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value. The Program Status Word is also restored by being popped from the stack.

This instruction is a privileged instruction (limited to system mode) and is used to return from an interrupt/exception. An attempt to use RETI in user mode will generate a trap.

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: All PSW bits are written by the POP of the PSW value in System mode.

Bytes: 2
Clocks: 10/8 (PZ)

Encoding:

1	1	0	1	0	1	1	0

1	0	0	1	0	0	0	0

Syntax: RL Rd, #data4

Operation:

count <- #data4
Do While (count not equal to 0)
($dest_0$) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
(count) <- count -1
End While

Description: The variable specified by the destination operand is rotated left by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:

1	1	0	1	SZ	0	1	1

d	d	d	d	#data4

Syntax: RLC Rd, #data4

Operation:

count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
($dest_0$) <- (temp)
(count) <- count -1
End While

Description: The variable specified by the destination operand is rotated left through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

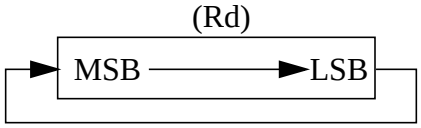
Encoding:

1	1	0	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: RR Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
($dest_{msb}$) <- ($dest_0$)
($dest_{n-1}$) <- ($dest_n$)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15. If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

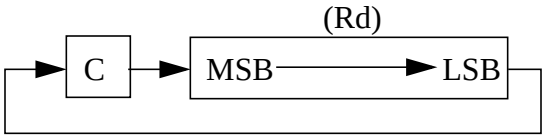
Encoding:

1	0	1	1	SZ	0	0	0

d	d	d	d	#data4

Syntax: RRC Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_0$)
($dest_n$) <- ($dest_{n+1}$)
($dest_{msb}$) <- (temp)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.
If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:

1	0	1	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: SETB bit

Operation: (bit) <-- 1

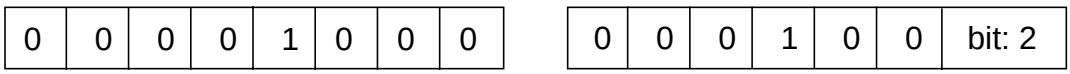
Description: Writes (sets) a 1 to the specified bit.

Size: Bit

Flags Updated:none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: SEXT Rd

Operation: if N = 1
then (Rd) <-- FF in byte mode or FFFF in word mode
if N = 0
then (Rd) <-- 00 in byte mode or 0000 in word mode

Description: Copies the N flag (the sign bit of the last ALU operation) into the destination register. The destination register may be a byte or word register.

Example:
SEXT.b R1
if the result of the previous operation left the N flag set, then R1 <-- FF

Size: Byte, word

Flags Updated: none

Bytes: 2
Clocks: 3

Encoding:

1	0	0	1	SZ	0	0	0

d	d	d	d	1	0	0	1

Syntax: SUB dest, src

Operation: dest <- dest - src

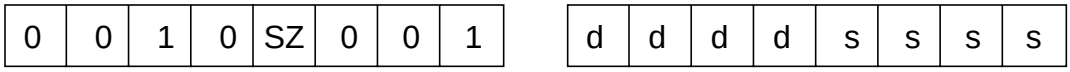
Description: Performs a twos complement binary subtraction of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

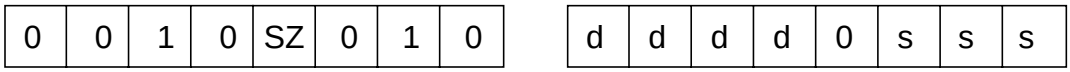
SUB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs)
Encoding:



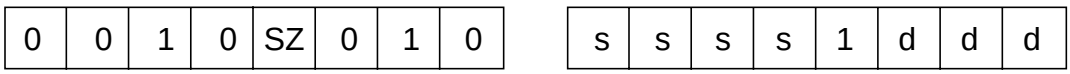
SUB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs))
Encoding:



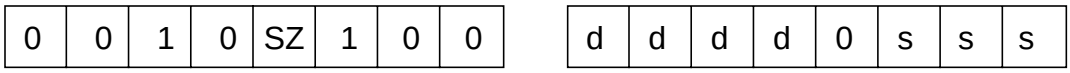
SUB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
Encoding:



SUB Rd, [Rs+offset8]

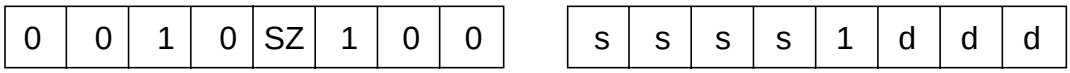
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

SUB [Rd+offset8], Rs

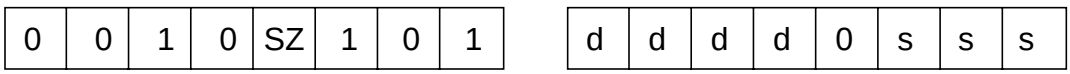
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

SUB Rd, [Rs+offset16]

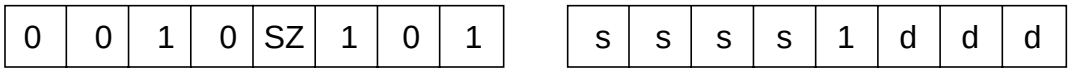
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUB [Rd+offset16], Rs

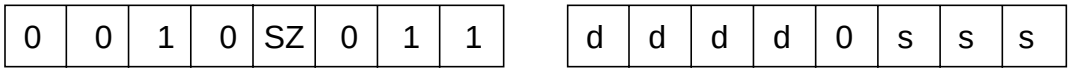
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

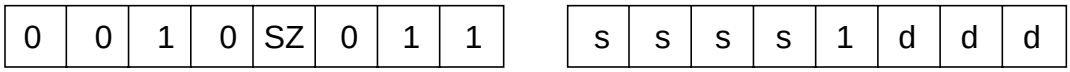
SUB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



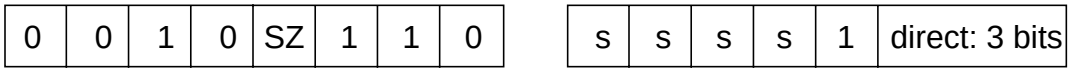
SUB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUB direct, Rs

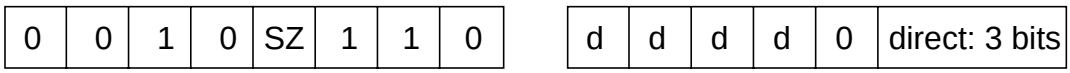
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, direct

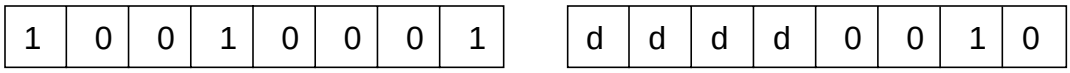
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, #data8

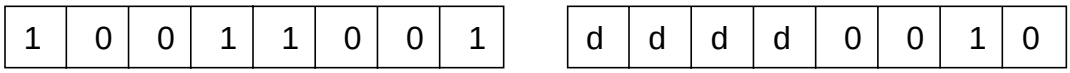
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8
Encoding:



byte 3: #data8

SUB Rd, #data16

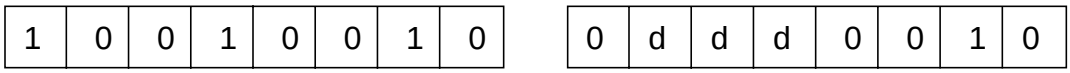
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd], #data8

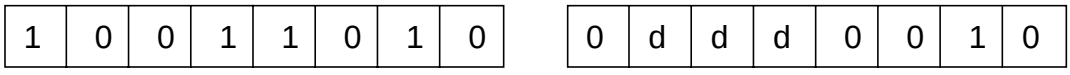
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

SUB [Rd], #data16

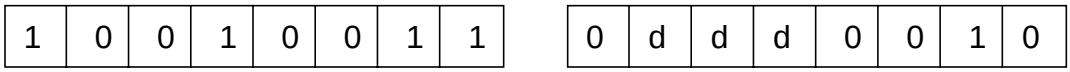
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+], #data8

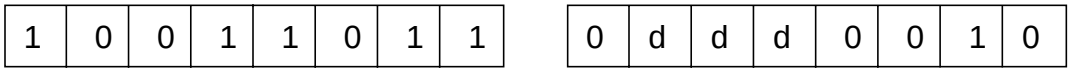
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUB [Rd+], #data16

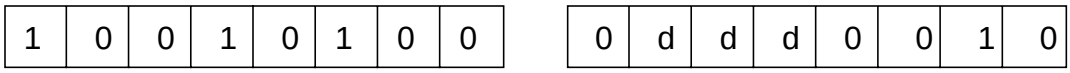
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+offset8], #data8

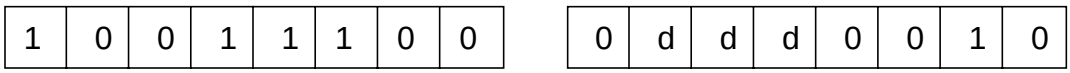
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

SUB [Rd+offset8], #data16

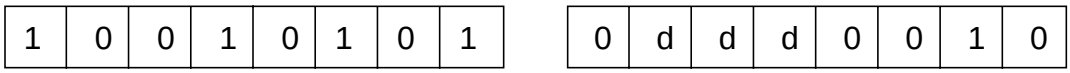
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUB [Rd+offset16], #data8

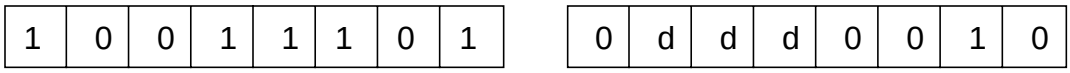
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUB [Rd+offset16], #data16

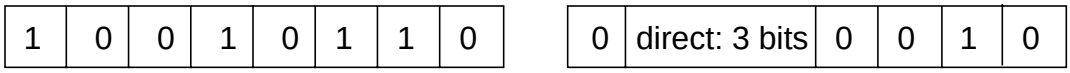
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUB direct, #data8

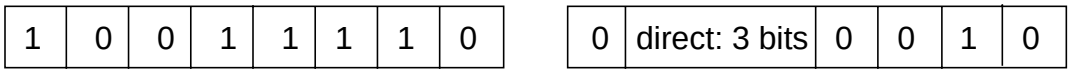
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

SUB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: SUBB dest, src

Operation: dest <- dest - src - C

Description: Performs a twos complement binary addition of the source operand and the previously generated carry bit (borrow) with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is zero (C = 0, i.e., Borrow = 1), the result is exact difference of the operands; if it is one (C = 1, i.e., Borrow = 0), the result is 1 less than the difference in operands.

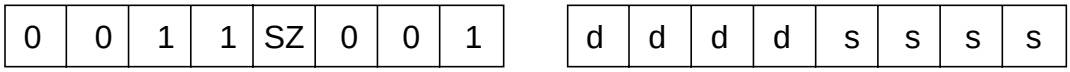
This form of subtraction is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then SUBB is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

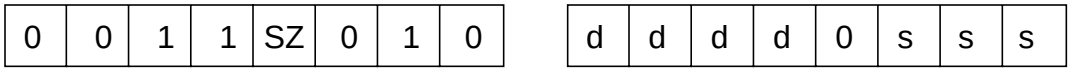
SUBB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs) - (C)
Encoding:



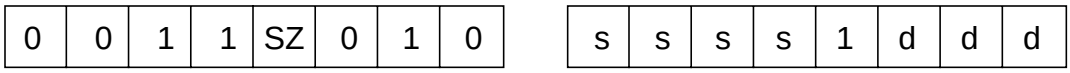
SUBB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
Encoding:



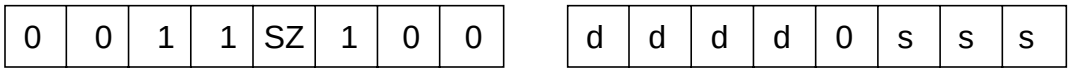
SUBB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
Encoding:



SUBB Rd, [Rs+offset8]

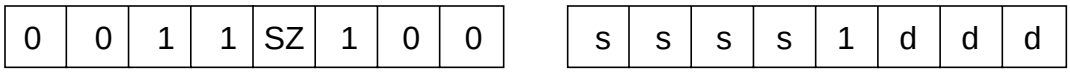
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8) - (C)
Encoding:



byte 3: offset8

SUBB [Rd+offset8], Rs

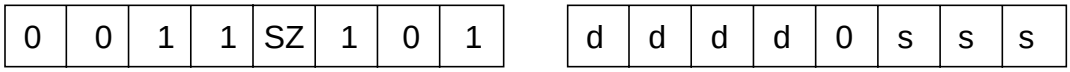
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs) - (C)
Encoding:



byte 3: offset8

SUBB Rd, [Rs+offset16]

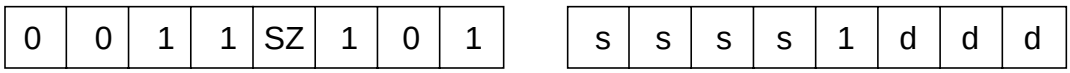
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUBB [Rd+offset16], Rs

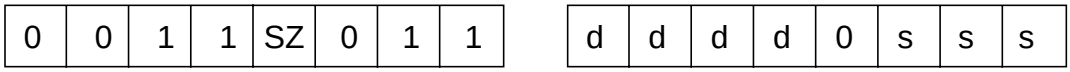
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

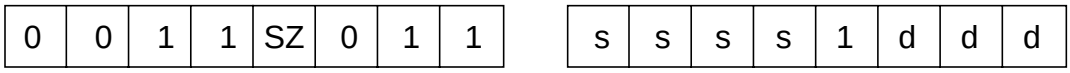
SUBB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



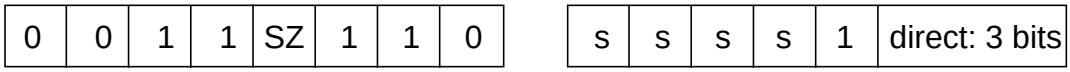
SUBB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUBB direct, Rs

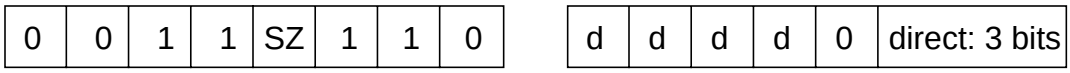
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, direct

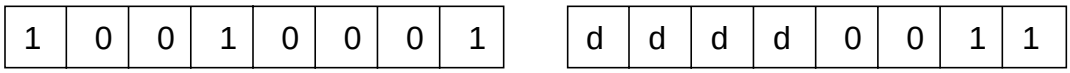
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, #data8

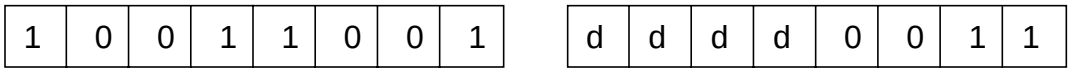
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB Rd, #data16

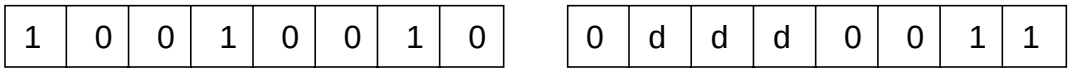
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd], #data8

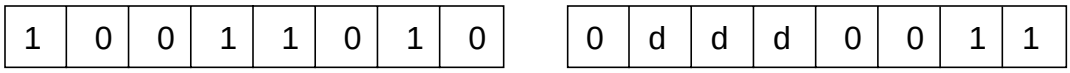
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB [Rd], #data16

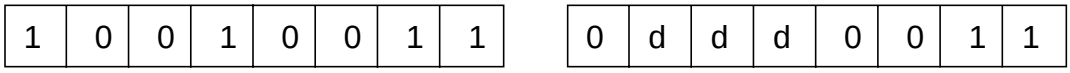
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+], #data8

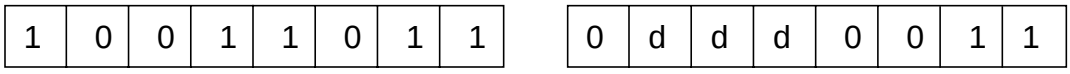
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUBB [Rd+], #data16

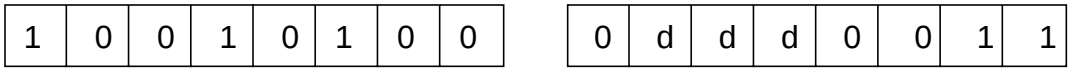
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+offset8], #data8

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8 - (C)
Encoding:



byte 3: offset8
byte 4: #data8

SUBB [Rd+offset8], #data16

Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	0	0	1	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUBB [Rd+offset16], #data8

Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8 - (C)
Encoding:

1	0	0	1	0	1	0	1		0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUBB [Rd+offset16], #data16

Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	1		0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUBB direct, #data8

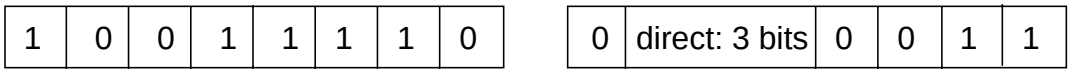
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8 - (C)
Encoding:

1	0	0	1	0	1	1	0		0	direct: 3 bits			0	0	1	1

byte 3: lower 8 bits of direct
byte 4: #data8

SUBB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16 - (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: TRAP #data4

Operation: (PC) <-- (PC) + 2
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (trap vector (#data4))
(PC.15-0) <-- code memory (trap vector (#data4))
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes the specified software trap. The invoked routine is determined by branching to the specified vector table entry point. The RETI, return from interrupt, instruction is used to resume execution after the trap routine has been completed. A trap acts like an immediate interrupt, using a vector to call one of several pieces of code that will be executed in system mode. This may be used to obtain system services for application code, such as altering the data segment register. This is described in more detail in the section on interrupts and exceptions.

Note: The address of the exception handling routine must be word aligned as the PC is forced to an even address before vectoring to the service routine.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 23/19 (PZ)

Encoding:

1	1	0	1	0	1	1	0

0	0	1	1	#data4

Syntax: XCH dest, src

Operation: dest <--> src

Description: The data specified by the source and destination operands is exchanged.

Size: Byte-Byte, word-word.

Flags Updated: none

XCH Rd, Rs

Bytes: 2
Clocks: 5
Operation: (Rd) <--> (Rs)
Encoding:

0	1	1	0	SZ	0	0	0	d	d	d	d	s	s	s	s

XCH Rd, [Rs]

Bytes: 2
Clocks: 6
Operation: (Rd) <--> ((WS:Rs))
Encoding:

0	1	0	1	SZ	0	0	0	d	d	d	d	0	s	s	s

XCH Rd, direct

Bytes: 3
Clocks: 6
Operation: (Rd) <--> (direct)
Encoding:

1	0	1	0	SZ	0	0	0	d	d	d	d	1	direct: 3 bits

byte 3: lower 8 bits of direct

Syntax: XOR dest, src

Operation: dest <- dest (XOR) src

Description: The byte or word specified by the source operand is bitwise logically XORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

XOR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) (Rs)
Encoding:

0	1	1	1	SZ	0	0	1	d	d	d	d	s	s	s	s

XOR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
Encoding:

0	1	1	1	SZ	0	1	0	d	d	d	d	0	s	s	s

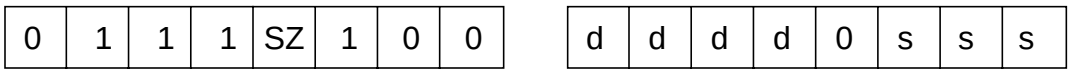
XOR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
Encoding:

0	1	1	1	SZ	0	1	0	s	s	s	s	1	d	d	d

XOR Rd, [Rs+offset8]

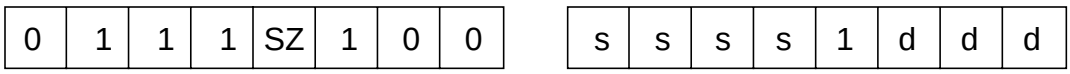
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

XOR [Rd+offset8], Rs

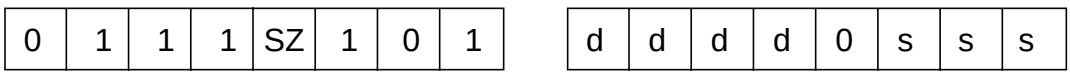
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) (Rs)
Encoding:



byte 3: offset8

XOR Rd, [Rs+offset16]

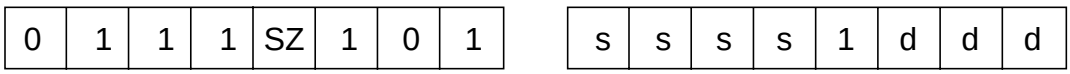
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

XOR [Rd+offset16], Rs

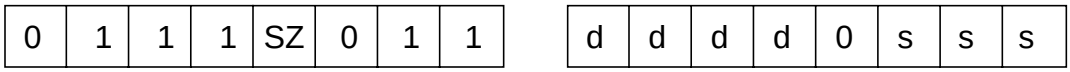
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

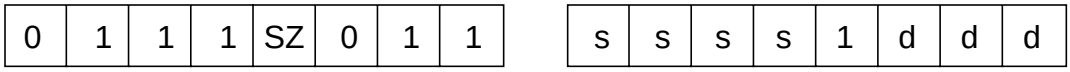
XOR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



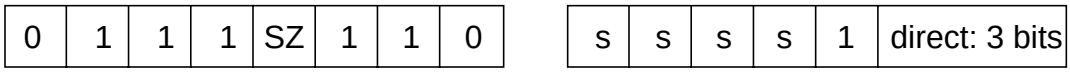
XOR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



XOR direct, Rs

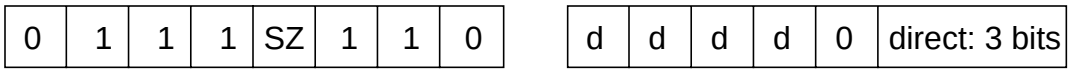
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) (XOR) (Rs)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, direct

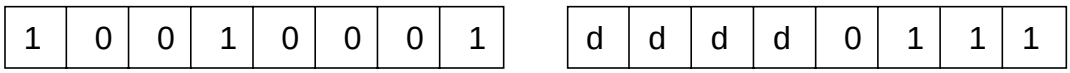
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) (direct)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, #data8

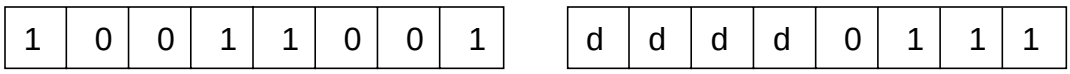
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data8
Encoding:



byte 3: #data8

XOR Rd, #data16

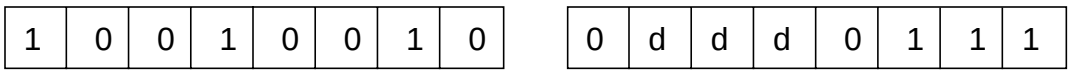
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd], #data8

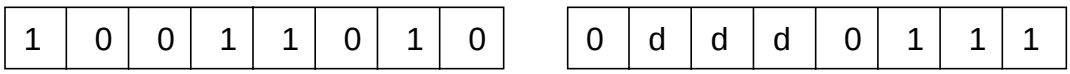
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
Encoding:



byte 3: #data8

XOR [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
Encoding:

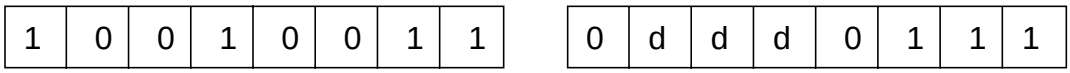


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
(Rd) <-- (Rd) + 1

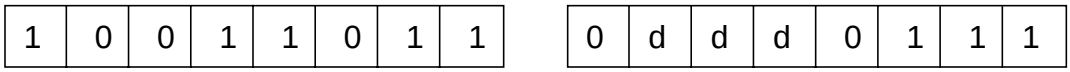
Encoding:



byte 3: #data8

XOR [Rd+], #data16

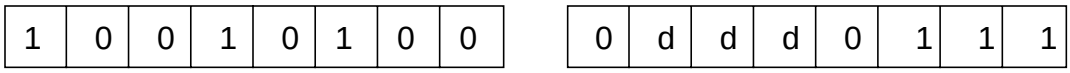
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+offset8], #data8

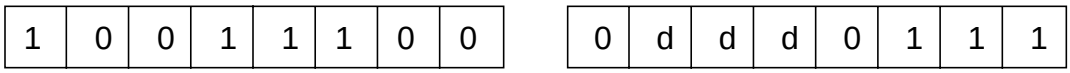
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data8
Encoding:



byte 3: offset8
byte 4: #data8

XOR [Rd+offset8], #data16

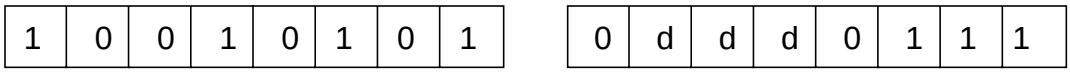
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

XOR [Rd+offset16], #data8

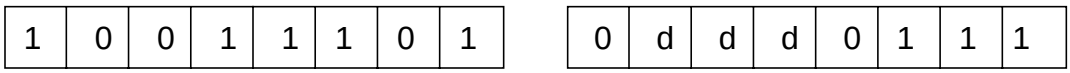
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

XOR [Rd+offset16], #data16

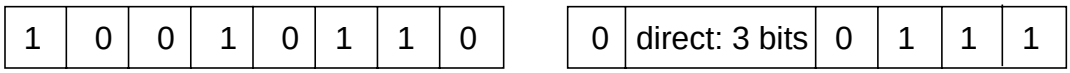
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

XOR direct, #data8

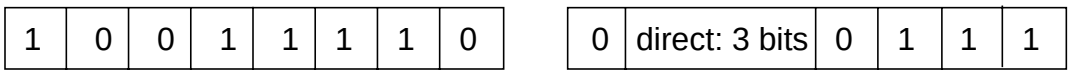
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

XOR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

6.6 Summary Of Illegal Operand Combinations On The XA

All but one case are instructions that specify or imply 2 write operations to a single register file location within a single instruction. The other case is a possible corruption of the source register data by an auto-increment before it is read. These conditions are not detected by XA hardware. The instruction/operand combinations indicated should not be used when writing XA code.

Instruction(s) affected	Reason for illegal combination
(any op) Rx, [Rx+]	Auto-increment plus explicit write [1]
mov [Rx+], [Rx+]	Double auto-increment of one register [2]
(any op) [Rx+], Rx	Auto-increment write may corrupt the source register before it is read [3]
NORM Rx, Rx	Result and shift count stored in the same register [4]
XCH Rx, Rx	Double write of a single register [4]
(any op) [Rx+], Ry	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], [Ry+]	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], #data	Auto-increment plus indirect write to same register [5]
XCH Rx, [Rx]	Indirect write plus explicit write to the same register [6]
XCH Rx, direct	Direct write plus explicit write to the same register [7]
POP R7	Stack pointer auto-increment plus explicit write to R7/SP [8]

NOTES:

1 This addressing mode is illegal when the source and destination are the same register. This would cause both a data write and an auto-increment operation to the same register.

2 This instruction is illegal when the source and destination pointer registers are the same register. This would cause two auto-increment operations to the same register.

3 This instruction is illegal when the source and destination are the same register. The source register would be auto-incremented and read at the same time, with an undefined result.

4 This instruction is illegal when the source and destination are the same register. This would cause two writes to the same register.

5 This addressing mode is illegal when the indirect address of the destination points to the pointer register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause both a data write and an auto-increment operation to the same register.

6 This instruction is illegal when the indirect address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

7 This instruction is illegal when the direct address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

8 A POP to R7 (the stack pointer) would cause both a data write and an auto-increment operation to the same register.

7 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices that are to be accessed, then generates a strobe for the required operation, with data passing in or out on the data bus. Typical bus operations are code read, data read, and data write. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

The following discussion is based on the standard version of the XA external bus. Some specific XA derivatives may have a different implementation of the external bus, or no external bus at all.

7.1 External Bus Signals

For flexibility, the standard XA external bus supports 8 or 16-bit data transfers and a user selectable number of address bits. The maximum number of address lines varies by derivative but may be up to 24. A standard set of bus control signals coordinates activity on the bus. These are described in the following sections.

7.1.1 $\overline{\text{PSEN}}$ - Program Store Enable

The program store enable signal is used to activate an external code memory, such as an EPROM. This active low signal is typically connected to the Output Enable ($\overline{\text{OE}}$) pin of an external EPROM. $\overline{\text{PSEN}}$ remains high when a code read is not in progress.

7.1.2 $\overline{\text{RD}}$ - Read

The bus read signal is also active low. Activity of this signal indicates data read operations on the external bus. $\overline{\text{RD}}$ is typically connected to the pin of the same name on an external peripheral device.

7.1.3 $\overline{\text{WRL}}$ - Write Low Byte

$\overline{\text{WRL}}$ is the external bus data write strobe. It is typically connected to the $\overline{\text{WR}}$ pin of an external peripheral device. When the XA external bus is used in the 16-bit mode, this strobe applies only to the lower data byte, allowing byte writes on the 16-bit bus. The $\overline{\text{WRL}}$ signal is active low.

7.1.4 $\overline{\text{WRH}}$ - Write High Byte

For a 16-bit data bus, a signal similar to $\overline{\text{WRL}}$, but for the upper data byte is needed. The active low signal $\overline{\text{WRH}}$ serves this purpose.

7.1.5 ALE - Address Latch Enable

Since a portion of the XA external bus is used for multiplexed address and data information, that part of the address must be latched outside of the XA so that it will remain constant during the

subsequent read or write operation. The active high ALE signal directs the external latch to allow information to be stored for a data address or a code address. The external latch must close and retain this address when the ALE signal ends, by going low (inactive).

7.1.6 Address Lines

Some of the address lines used by the external bus interface are driven during a complete bus operation and do not need to be latched. In the standard XA bus interface, the lower four address lines are always driven and unlatched in this manner. This is done specifically as part of the optimization of the bus for fetching instructions from external code memory at high speed. This feature will be explained in detail in a later section.

7.1.7 Multiplexed Address and Data Lines

The part of the bus that is used for data transfer is also used for address output from the XA. Prior to asserting the strobe for the bus operation about to be performed, the XA outputs the address for the operation. On the multiplexed portion of the bus, this address is captured by an external latch, as commanded by the ALE signal. After that is done, this part of the bus is free to be used for data transfer either into or out of the XA. The control signals $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, and $\overline{\text{WRH}}$ determine what type of bus operation takes place.

7.1.8 WAIT - Wait

The WAIT input allows wait states to be inserted into any external bus operation. If WAIT is asserted (high) after a bus control strobe ($\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, or $\overline{\text{WRH}}$) is driven by the XA, that bus operation is stretched, and that control strobe continues to be driven by the XA until WAIT goes low again. For this feature to be used, an external circuit must be present to generate the WAIT signal at the appropriate times.

The XA has an internal bus configuration feature that allows programming the various types of external bus cycles to different lengths, so that in most applications the WAIT line will not be needed. This feature will be explained in detail in a later section.

7.1.9 $\overline{\text{EA}}$ - External Access

The $\overline{\text{EA}}$ input determines whether the XA operates in single-chip mode, or begins running code from the internal program memory after reset. If $\overline{\text{EA}}$ is low as Reset goes high, the first code fetch (and all others after that) is made off-chip. If $\overline{\text{EA}}$ is high as Reset goes high, the XA will execute the on-chip code first, but will still attempt to execute instructions from external memory at addresses above the limit of on-chip code. The level on the $\overline{\text{EA}}$ pin is latched as reset goes high, so whatever mode is selected remains valid until the next reset.

On some XA derivatives, the pin used for the $\overline{\text{EA}}$ function may be shared with another function that becomes active after the XA begins code execution.

7.1.10 BUSW - Bus Width

The external XA bus may be configured to be 8 or 16 bits in width. The XA allows the bus width to be programmed in 2 ways. In a system where instructions are initially fetched from on-chip code memory, the user program can configure the external bus size (and many other aspects of the bus) prior to the bus actually being used.

When the initial code fetches must be done using off-chip code memory, however, the XA must know the bus width before the first off-chip code fetch can begin.

On some XA derivatives, the BUSW function may share a pin with some other function. In this case, the level on the BUSW pin is latched as Reset is released and that selection is kept until the next Reset. The secondary function on that pin will be active after Reset when the processor begins executing code normally.

Unlike the $\overline{EA}$ function, the bus width set by the BUSW pin at reset may be over-ridden by a user program, making setting by use of the BUSW pin unnecessary in most systems. Settings in the Bus Configuration Register allow changing the bus size under program control. This feature is covered in more detail in the next section.

7.2 Bus Configuration

The standard XA external bus has a number of configuration options. In addition to the data bus width selection discussed previously, the number of address lines used for external accesses is programmable, as is the bus timing.

7.2.1 8-Bit and 16-Bit Data Bus Widths

The standard XA external bus allows both 8-bit and 16-bit bus widths. BUSW=0 selects an 8-bit bus and BUSW=1 selects a 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The bus width is determined by the value of the BUSW pin as Reset is released, unless a user program overrides that setting by writing to the Bus Configuration Register (BCR), shown in Figure7.1.

BCR	-	-	-	WAITD	BUSD	BC2	BC1	BC0

WAITD: WAIT disable. Causes the XA external bus interface to ignore the value on the WAIT input. This allows tying the WAIT input high for applications that use internal code and do not need the WAIT function.

BUSD: Bus disable. Causes XA external bus functions to be disabled permanently. The primary purpose of this is to allow prevention of inadvertent activation of the bus by an instruction pre-fetch when the XA is executing code near the end of the on-chip code memory.

BC2 - BC0: These bits select the XA external bus configuration, specifically the number of data bits and the number of address lines. The supported combinations are shown below.

000 : 8-bit data bus, 12 address lines
001 : 8-bit data bus, 16 address lines
010 : 8-bit data bus, 20 address lines
011 : 8-bit data bus, 24 address lines
100 : < function reserved >
101 : < function reserved >
110 : 16-bit data bus, 20 address lines
111 : 16-bit data bus, 24 address lines

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Figure 7.1 Bus Configuration Register (BCR)

Figures 7.2 and 7.3 show the address and data functions present on XA bus related pins when used with each available bus width.

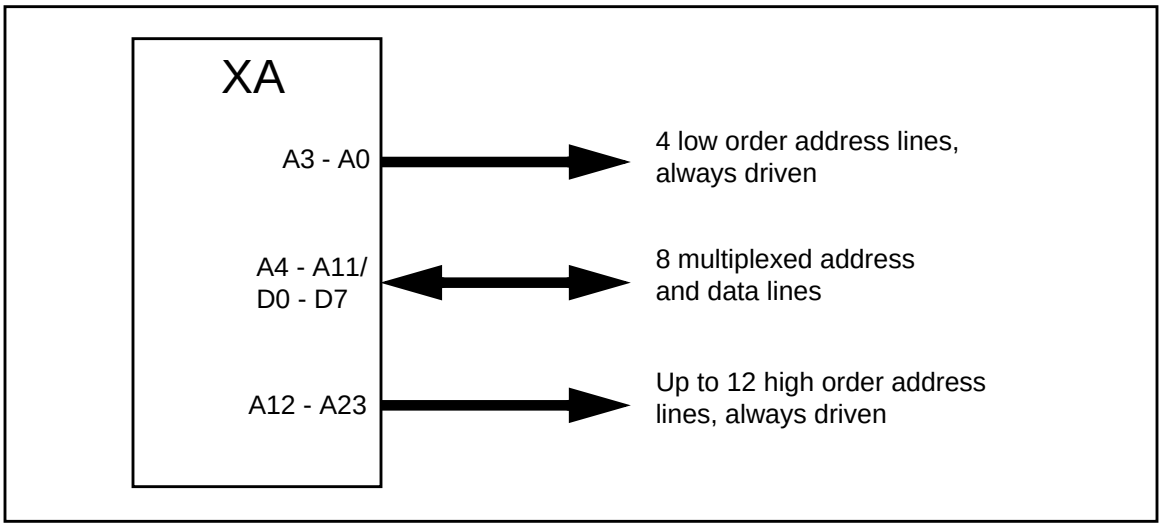


Figure 7.2 8-Bit External Bus Configuration

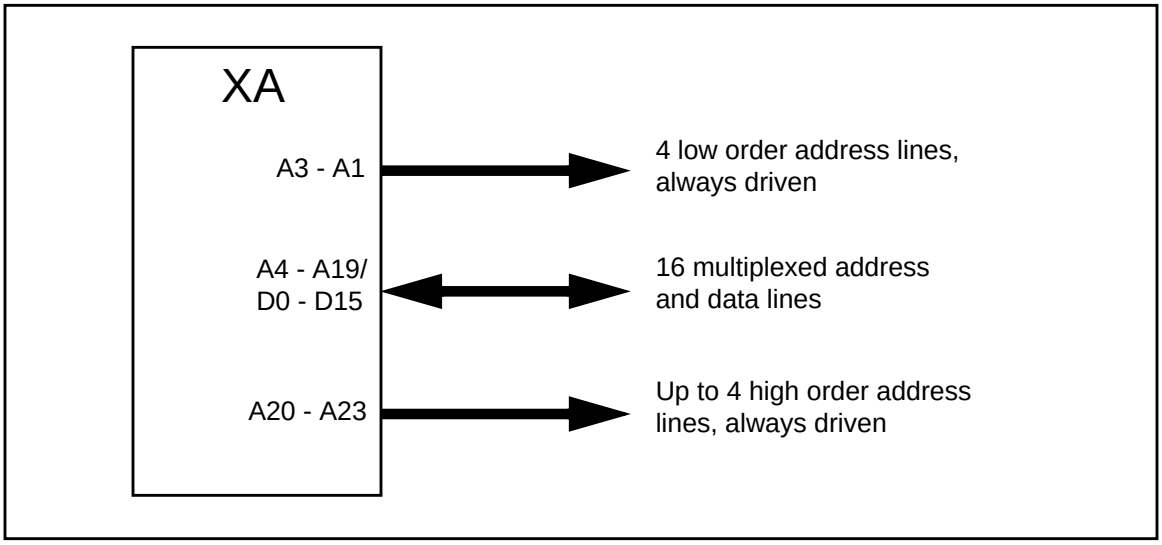


Figure 7.3 16-Bit External Bus Configuration

7.2.2 Typical External Device Connections

Many possibilities exist for connecting and using external devices with the XA bus. The bus will support EPROMs, RAMs, and other memory devices, as well as peripheral devices such as UARTs, and parallel port expanders. The following diagrams show a generalized connection of devices for 8-bit and 16-bit XA bus modes.

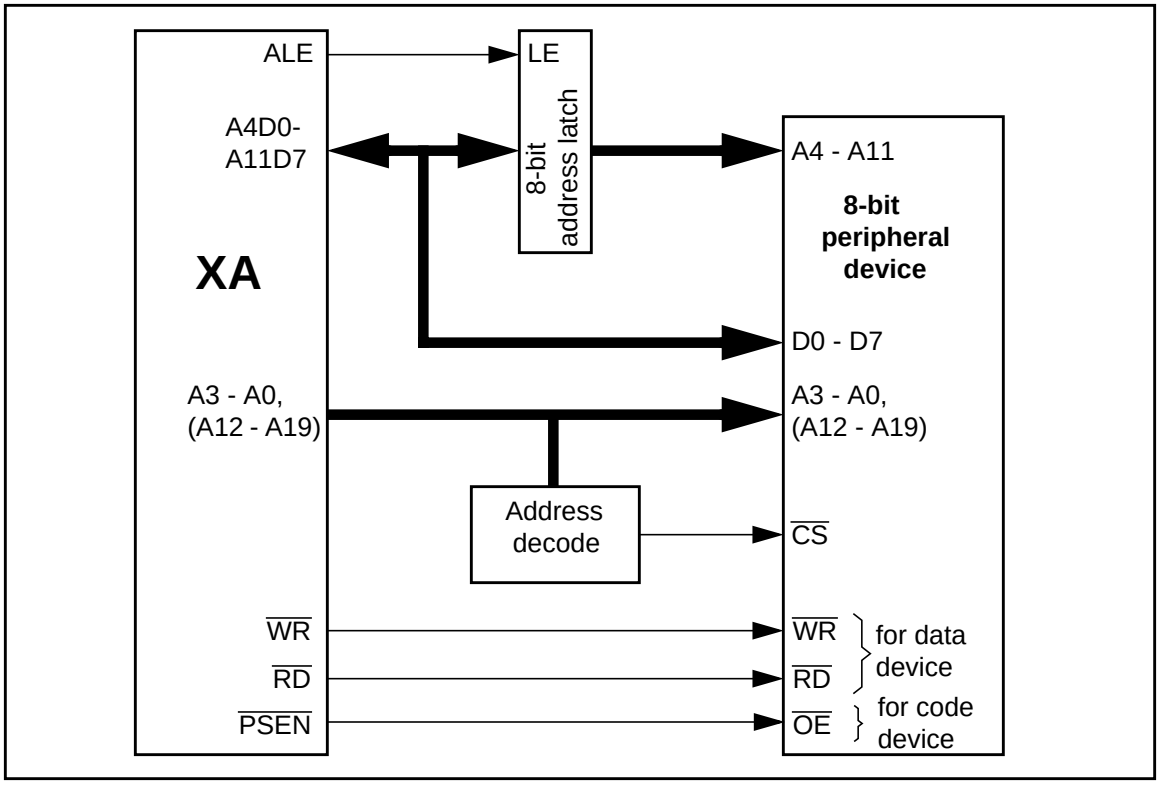


Figure 7.4 Typical XA External Bus Connections for 8-Bit Peripheral Devices

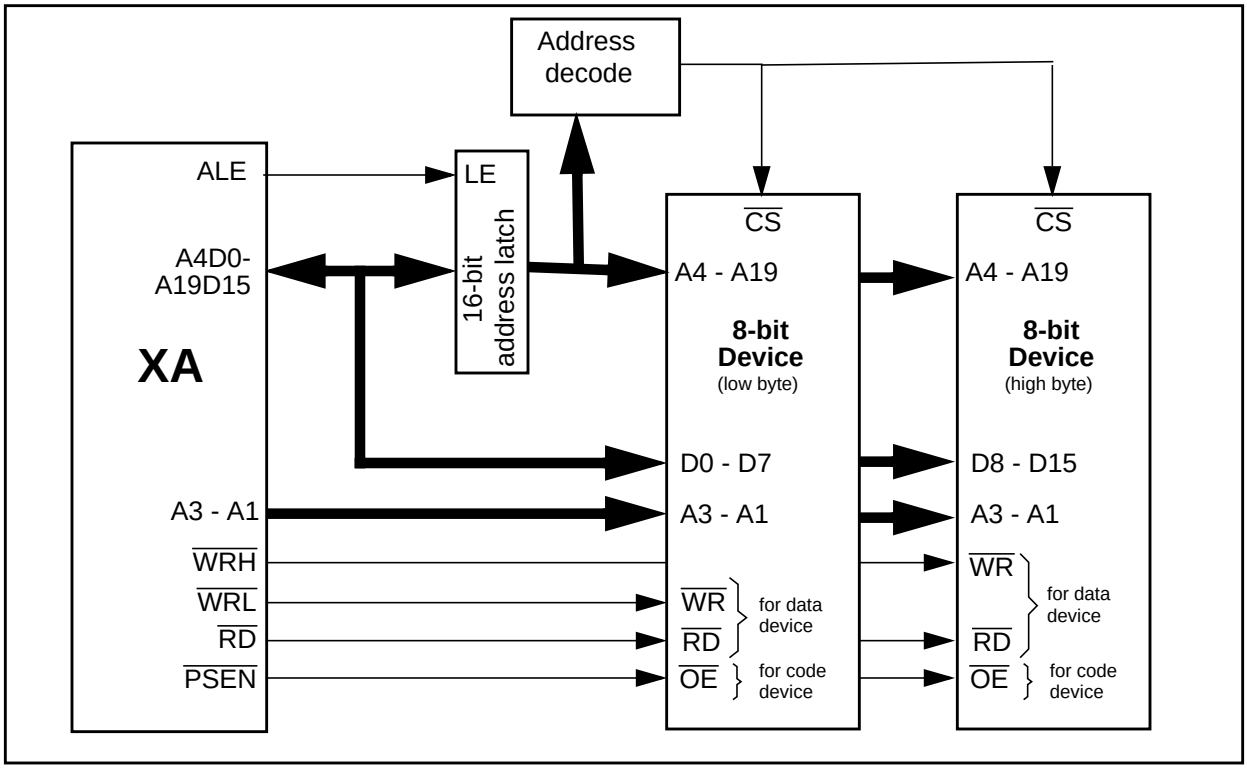


Figure 7.5 Typical XA External Bus Connections for 16-Bit Peripheral Devices

7.3 Bus Timing and Sequences

The standard XA external bus allows programming the widths of the bus control signals ALE, PSEN, WRL, WRH, and RD. There is also an option to extend the data hold time after a write operation. The combinations available will allow interfacing most devices to the XA directly without the need for special buffers or a WAIT state generator. Note that there is always a "rest clock" after any type of bus cycle except part of a burst mode code read. That is, when a bus cycle is completed and the bus strobe de-asserted, no new bus cycle will be begun until one clock has passed with no bus activity.

7.3.1 Code Memory

Interfacing with external code memory, typically in the form of EPROMs, is enabled by the $\overline{\text{PSEN}}$ control signal. If the XA is configured to execute internal code memory at reset, by the setting of the $\overline{\text{EA}}$ pin, it will automatically begin to fetch external code if the program crosses the boundary from internal to external code space. The location of this boundary varies for different XA derivatives, depending on the size of the internal code memory for each part.

Since the XA employs a pre-fetch queue in order to optimize instruction execution times, fetching of external instructions may begin before program execution actually crosses the on/off-chip code memory boundary. If a branch or subroutine return is located near the end of on-chip code memory, the off-chip fetch would be unnecessary, and may in fact cause problems if the XA ports that implement the external bus are being used for other purposes. For this reason, the BUSD (bus disable) bit in the Bus Configuration Register (BCR) is provided to prevent the XA from using the external bus for code or data operations.

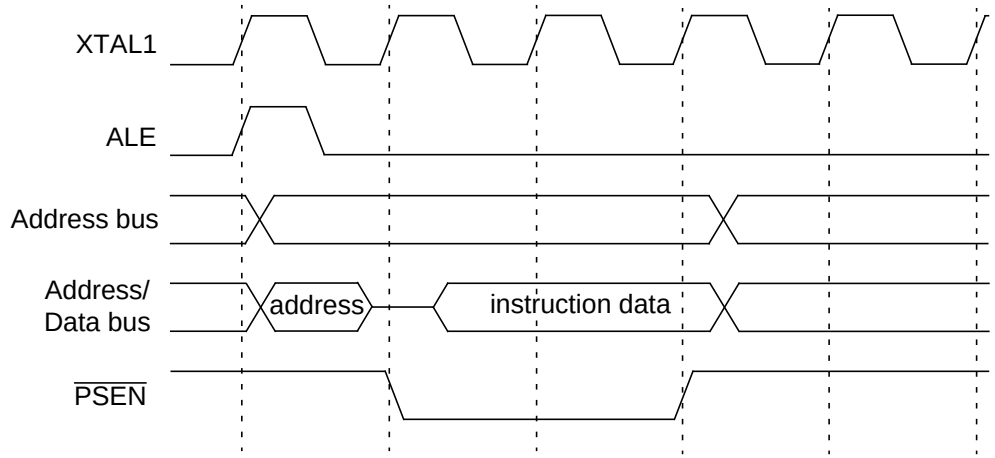
Note also that external code read cycles may sometimes be aborted by the XA. This happens when a code pre-fetch is occurring on the bus and the XA must execute a branch. The instruction data from the code pre-fetch will not be needed, so the bus cycle will be terminated immediately. This may appear as an ALE with no subsequent PSEN strobe, or a PSEN strobe that is shorter than that specified by the bus timing registers.

Code Read with ALE

The classic operation of a multiplexed address and data bus involves the issuance of an address, along with its associated control signal, for every bus cycle. The XA uses the bus control signal ALE to indicate that an address is on the bus that must be latched through the following code or data operation. The following diagram shows a code memory fetch in a cycle using ALE.

Burst Code Read (No ALE)

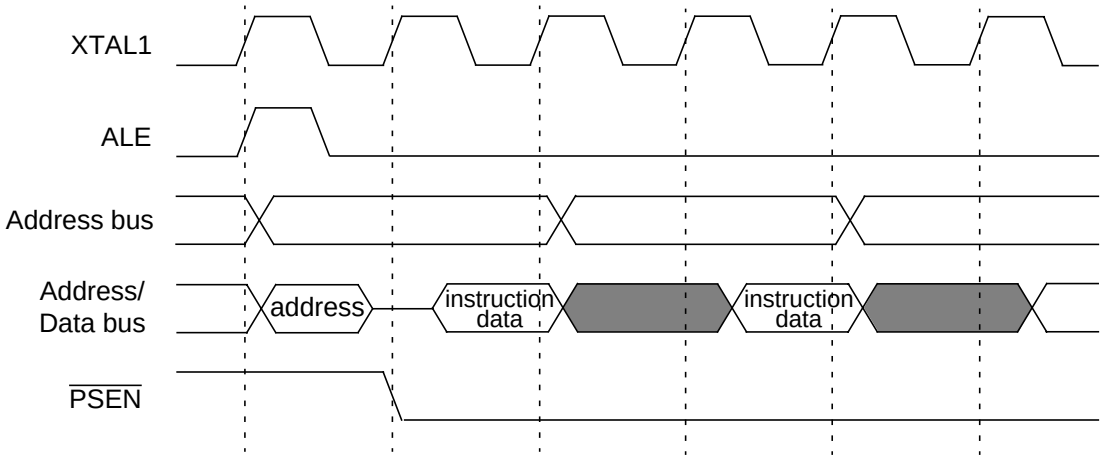
The XA does not always require an ALE cycle for every code fetch. This feature is included specifically to improve performance when the XA executes code from external memory, while increasing the access time available for the external memory device. Because the lower four address lines of the external bus are always driven, not multiplexed, the XA can access up to 16 bytes (or 8 words) of sequential code memory each time an ALE is issued. This type of fast sequential code read is called a burst read. Of course, any type of jump, branch, interrupt, or other change in sequential program flow will require an ALE in order to change the code fetch address in a non-sequential manner. Any data operation (read or write) on the XA external bus also requires an ALE cycle and will cause any subsequent external code fetch to begin with an ALE cycle also.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CRA1/0 = 01.

Figure 7.6 Typical External Code Read Using ALE

The following diagram shows a typical sequential code fetch where no ALE is issued between code reads. Also note that the $\overline{\text{PSEN}}$ bus control signal does not toggle, but remains asserted throughout the burst code read



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CR1/0 = 01, CRA1/0 = 00.

Figure 7.7 Burst Mode (Sequential) External Code Read

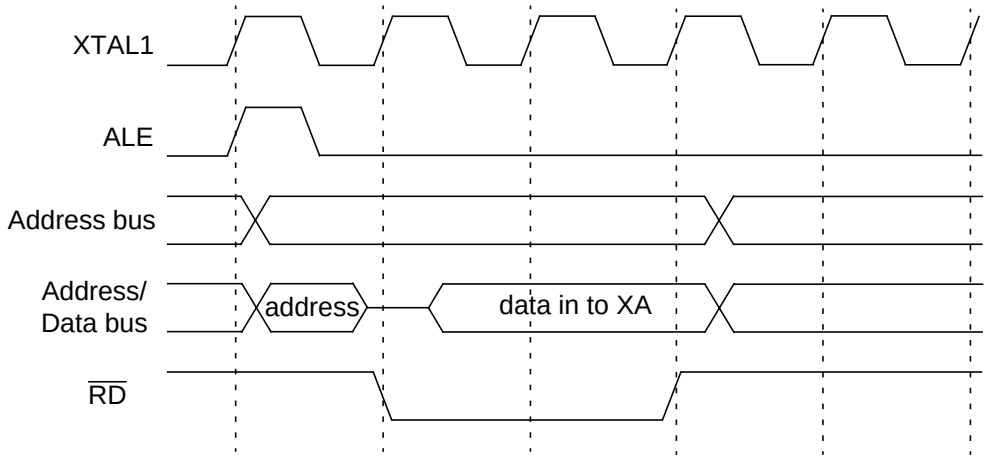
7.3.2 Data Memory

Reads and writes on the XA external bus are controlled through the use of the $\overline{RD}$, $\overline{WRL}$, and $\overline{WRH}$ signals. Since the XA bus supports both 8-bit and 16-bit widths, as well as byte and word read and write operations, several different versions of the basic bus cycles are possible. These are described in the following sections.

Data memory, like code memory, has a boundary where the internal data memory ends, and above which the XA will switch to the external bus in order to act on data memory. This on/off-chip data memory boundary may be in a different place for various XA derivatives, depending upon the amount of internal data memory built into a specific derivative.

Typical Data Read

A simple byte read on an 8-bit bus or any read on a 16-bit bus both begin with an ALE cycle, where the XA presents the address of the data location that is to be read on the bus. This is followed by the assertion of the $\overline{RD}$ strobe, that causes the external device to present its data on the bus. This process is shown in the diagram below.

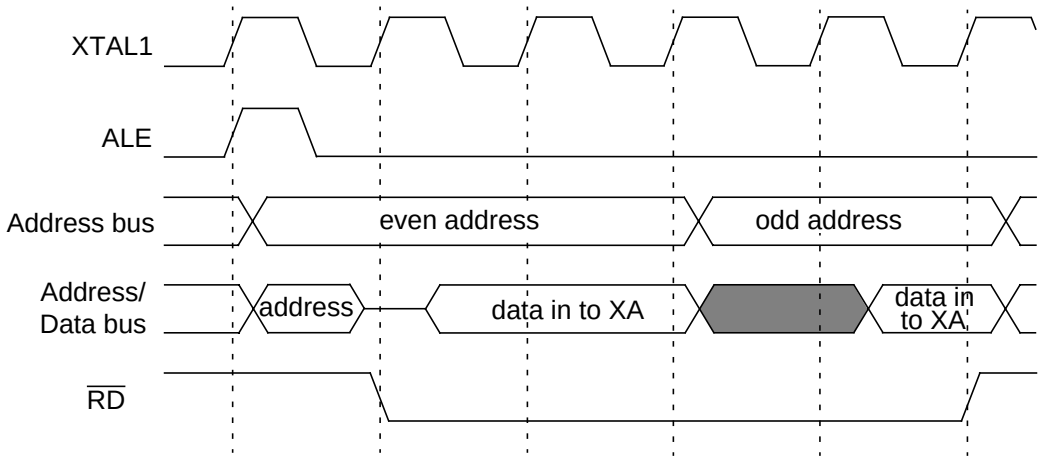


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01.

Figure 7.8 Typical External Data Read

Word Read on an 8-Bit Data Bus

When the XA external bus is configured for an 8-bit data width, a word read operation is automatically performed as two byte reads at sequential addresses. Since the XA CPU requires word operations to be performed at even addresses, the second half of any word read on a byte-wide bus always uses the same upper address latched by ALE. for this operation, the low order byte first is read at the even byte address, then the high order byte is read at the next (odd) address. So, only one ALE is required in this case. The diagram below shows this sequence.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01, DR1/0 = 01.

Figure 7.9 Word Read on 8-Bit Data Bus

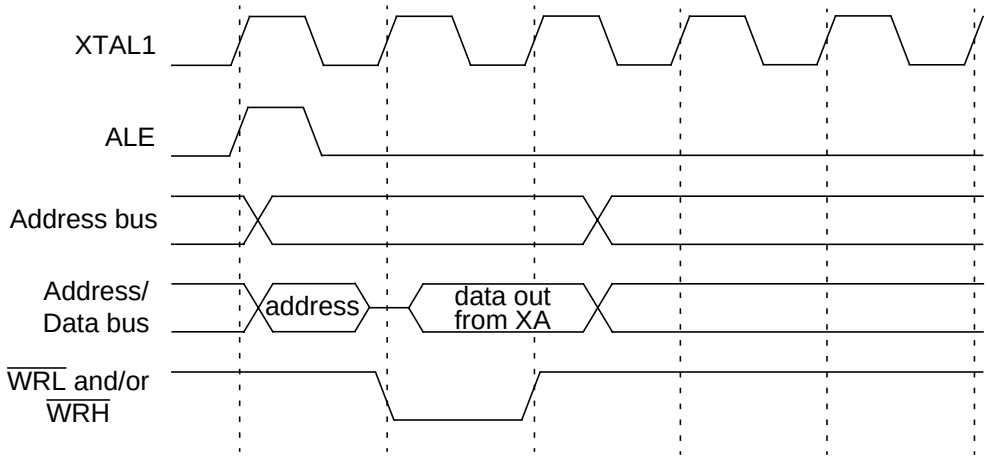
Byte Read on a 16-Bit Data Bus

When an instruction causes a read of one byte of data from the external bus, when it is configured for 16-bit width, a simple read operation is performed. This results in 16 bits of data being received by the XA, which uses only the byte that was requested by the program. There is no way to distinguish a byte read from a word read on the external bus when it is configured for a 16-bit width.

Typical Data Write

A data write operation begins with an ALE cycle, like a read operation, followed by the assertion of one or both of the write strobes, $\overline{WRL}$ and $\overline{WRH}$. This simple bus cycle applies to byte writes on an 8-bit data bus and all writes on a 16-bit data bus.

A byte write on an 8-bit data bus will always use only the $\overline{WRL}$ strobe. A byte write on a 16-bit data bus will always use either the $\overline{WRL}$ or $\overline{WRH}$ strobe, depending on whether the byte is at an even or odd address. A word write on a 16-bit bus requires the assertion of both the $\overline{WRL}$ and $\overline{WRH}$ strobes. The simple data write cycle is shown below.

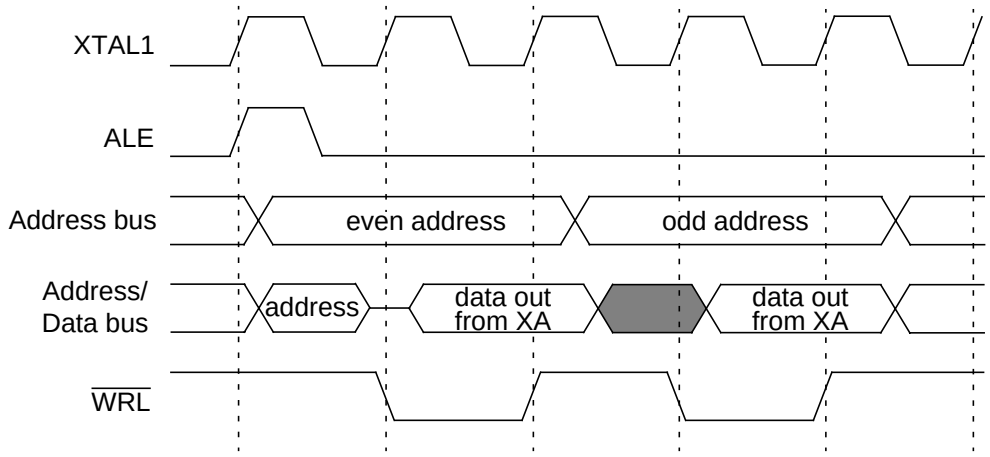


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.10 Typical External Data Write

Word Write on an 8-Bit Data Bus

When a word write operation is done with the bus configured to an 8-bit width, the XA automatically performs two byte writes. First, the low order byte is written (at the even byte address), then the high order byte is written at the next (odd) address. As with a word read on an 8-bit bus, this requires only a single ALE cycle at the beginning of the process. This sequence is shown in the following diagram.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, DW1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.11 Word Write on 8-Bit Data Bus

External Bus Signal Timing Configuration

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, data read and write cycle lengths, and data hold time. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

Programmable bus timing is controlled by settings found in the Bus Timing Register SFRs, named BTRH, and BTRL, shown in Figures 7.12 and 7.13.

BTRH	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0

DW1, DW0: Data Write without ALE. Applies only to the second half of a 16-bit write operation when the bus is configured to 8 bits.
00 : Data write cycle is 2 clock in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DWA1, DWA0: Data Write with ALE. Selects the length (in CPU clocks) of the entire data write cycle, including ALE.
00 : Data write cycle is 2 clocks in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DR1, DR0: Data Read without ALE. Applies only to the second half of a 16-bit read operation when the bus is configured to 8 bits.
00 : Data read cycle is 1 clock in duration.
01 : Data read cycle is 2 clocks in duration.
10 : Data read cycle is 3 clocks in duration.
11 : Data read cycle is 4 clocks in duration.

DRA1, DRA0: Data Read with ALE. Selects the length (in CPU clocks) of the entire data read cycle, including ALE.
00 : Data read cycle is 2 clocks in duration.
01 : Data read cycle is 3 clocks in duration.
10 : Data read cycle is 4 clocks in duration.
11 : Data read cycle is 5 clocks in duration.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.12 Bus Timing Register High Byte (BTRH)

BTRL	WM1	WM0	ALEW	-	CR1	CR0	CRA1	CRA0

WM1: Write Mode 1. Selects the width of the write pulse.
0 : Write pulse (WR) width is 1 CPU clock.
1 : Write pulse (WR) width is 2 CPU clocks.

WM0: Write Mode 0. Selects the data hold time.
0 : Data hold time is minimum (0 clocks).
1 : Data hold time is 1 CPU clock.

ALEW: ALE width selection. Determines the duration of ALE pulses.
0 : ALE width is one half of one CPU clock.
1 : ALE width is one and a half CPU clocks.

CR1, CR0: Code Read. Selects the length of a code read cycle when ALE is not used.
00 : Code read cycle is 1 clocks in duration.
01 : Code read cycle is 2 clocks in duration.
10 : Code read cycle is 3 clocks in duration.
11 : Code read cycle is 4 clocks in duration.

CRA1, CRA0: Code Read with ALE. Selects the length of a code read cycle when ALE is used prior to $\overline{\text{PSEN}}$ being asserted.
00 : Code read cycle is 2 clocks in duration.
01 : Code read cycle is 3 clocks in duration.
10 : Code read cycle is 4 clocks in duration.
11 : Code read cycle is 5 clocks in duration.

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.13 Bus Timing Register Low Byte (BTRL)

Disallowed Bus Timing Configurations

Some possible combinations of bus timing register settings do not make sense and the XA cannot produce working bus signals that match those settings. The disallowed combinations occur where the sum of the specified components of a bus cycle exceed the specified length of the entire cycle. Two simple rules define the allowed/disallowed combinations. Violating these rules may result in incomplete bus cycles, for example a data read cycle in which an address and ALE pulse are output, but no read strobe ($\overline{RD}$) is produced.

For data write cycles on the external bus there are two conditions that must be met. The first applies to data write cycles with no ALE:

$$WM1 + WM0 \leq DW1:0$$

This says that the sum of the timing values defined by the WM1 and WM0 fields must be less than or equal to the timing value defined by the DW field. Note that this is the value of the timing durations that they specify. For example, if the WM1 field specifies a 2 clock write pulse and the WM0 field specifies a 1 clock data hold time, those two times together (3 clocks) must be less than or equal to the timing specified by the DW1:0 field. In this case the DW1:0 field must specify a total bus cycle duration of at least 3 clocks. The other rule uses the same structure, as follows.

A second requirement applies to write cycles with ALE:

$$ALEW + WM1 + WM0 \leq DWA1:0$$

The configuration for data read has only one requirement, which applies to data read cycles with ALE:

$$ALEW + 1 \leq DRA1:0$$

The configuration for code read also has only one requirement, which applies to code read cycles with ALE:

$$ALEW + 1 \leq CRA1:0$$

7.3.3 Reset Configuration

Upon reset, at the time of power up or later, the XA bus is initially configured in certain ways. As previously discussed, the pins $\overline{EA}$ and BUSW select whether the XA will begin operation from internal code, and whether the bus will be 8-bits or 16-bits.

The values for the programmable bus timing are also set to a default value at reset. All of the timing values are set to their maximum, providing the slowest bus cycles. This setting allows for the slowest external devices that may be sued with the XA without WAIT generation logic. The user program should set the bus timing to the correct values for the specific application in the system initialization code. Refer to the data sheet for a particular XA derivative for details of the values found in registers and SFRs after reset.

7.4 Ports

I/O ports on any microcontroller provide a connection to the outside world. The capabilities of those I/O ports determine how easily the microcontroller can be interfaced to the various external devices that make up a complete application. The standard XA I/O ports provide a high degree of versatility through the use of programmable output modes and allow easy connection to a wide variety of hardware.

7.4.1 I/O Port Access

The standard on-chip I/O ports of the XA are accessed as SFRs. The SFR names used for these ports begin with port 0, called P0. Port numbers and names go up in sequence from there, to the number of ports on a specific XA derivative. Ports are normally identified by their names in assembler source code, such as: "MOV P1,#0". This instruction causes the value 0 to be written to port 1.

XA I/O ports are typically bit addressable, meaning that individual port bits are readable, writable, and testable. An instruction using a port bit looks like this: "SETB P2.1". This particular example would result in the second lowest bit in port 2 (bit 1) having a 1 written to it.

Reading of a Port Pin Versus the Port Latch

Each I/O port has two important logic values associated with it. The first is the contents of the port latch. When data is written to a port, it is stored in the port latch. The second value is the logic level of the actual port pin, which may be different than the port latch value, especially if a port pin is being used as an input.

When a port is explicitly read by an instruction, the value returned is that from the pin. When a port is read intrinsically, in order to perform some operation and store the value back to the port, the port latch is read. This type of operation is called a read-modify-write.

1) The following instructions cause read-modify-write operations, and read the port latch when a port or port bit is specified as the destination:

```
ADD     Px, ...
ADDC    Px, ...
ADDS    Px, ...
AND     Px, ...
DJNZ    Px, ...
OR      Px, ...
SUB     Px, ...
SUBB    Px, ...
XOR     Px, ...

CLR     Px.y
JBC     Px.y, rel8
MOV     Px.y, C
SETB    Px.y
```

2) The following instruction reads the port pins when a port is specified as the destination operand:

```
CMP     Px, ...
```

3) When a port or port bit is specified as a source in any instruction, the port pin is always read.

Figure 7.14 How ports are read.

7.4.2 Port Output Configurations

Standard XA I/O ports provide several different output configurations. One is the 80C51 type quasi-bidirectional port output. Others are open drain, push-pull, and high impedance (input only). It is important to note that the port configuration applies to a pin even if that pin is part of the external bus. Bus pins should normally be configured to push-pull mode. Also, the port latches for pins that are to be used as part of the external bus must be set to one (which is the reset state). A zero in a port latch will override bus operations and force a zero on the corresponding bus position.

The port configuration is controlled by settings in two SFRs for each port. One bit in each port configuration register is associated with a port pin in the corresponding bit position. These port configuration SFRs are called: PnCFGA and PnCFGB, where "n" is the port number. So, the configuration registers for port 1 are named P1CFGA and P1CFGB. The table below shows the port control bit combinations and the associated port output modes.

Table 7.1

PnCFGB	PnCFGA	Port Output Mode
0	0	Open drain.
0	1	Quasi-bidirectional (default).
1	0	High impedance.
1	1	Push-pull.

7.4.3 Quasi-Bidirectional Output

The default port output configuration for standard XA I/O ports is the quasi-bidirectional output that is common on the 80C51 and most of its derivatives. This output type can be used as both an input and output without the need to reconfigure the port. This is possible because when the port outputs a logic high, it is weakly driven, allowing an external device to pull the pin low. When the pin is pulled low, it is driven strongly and able to sink a fairly large current. These features are somewhat similar to an open drain output except that there are three pullup transistors in the quasi-bidirectional output that serve different purposes.

One of these pullups, called the "very weak" pullup, is turned on whenever the port latch for a particular pin contains a logic 1. The very weak pullup sources a very small current that will pull the pin high if it is left floating.

A second pullup, called the "weak" pullup, is turned on when the port latch for its associated pin contains a logic 1 and the pin itself is a logic 1. This pullup provides the primary source current for a pin that is outputting a 1, and can drive several TTL loads. If a pin that has a logic 1 on it is pulled low by an external device, the weak pullup turns off, and only the very weak pullup remains on. In order to pull the pin low under these conditions, the external device has to sink enough current to overpower the weak pullup and pull the voltage on the port pin below its input threshold.

The third (and final) pullup is referred to as the "strong" pullup. This pullup is included to speed up low-to-high transitions on a port pin when the port latch changes from 0 to 1. When this occurs, the strong pullup turns on for a brief time, two CPU clocks, pulling the port pin high quickly, then turning off again.

The quasi-bidirectional output structure normally provides a means to have mixed inputs and outputs on port pins without the need for special configurations. However, it has several drawbacks that can be problems in certain situations. For one thing, quasi-bidirectional outputs have a very small source current and are therefore not well suited to driving certain types of loads. They are especially unsuited to directly drive the bases of external NPN transistors, a common method of boosting the current of I/O pins.

Also, since the weak pullup turns off when a port pin is actually low, and the strong pullup turns on only for a brief time, it is possible that under certain port loading conditions, the port pin will get "stuck" low and cannot be driven high. This tends to happen when an external device being driven by the port pin has some leakage to ground that is larger than the current supplied by the very weak pullup of the quasi-bidirectional port output. If there is also a fairly large capacitance on the pin, from a combination of the wiring itself and the pin capacitance of the device(s) connected to the pin, the strong pullup may not succeed in pulling the pin high enough while it is turned on. When the strong pullup is then turned off, the leakage of the external device pulls the pin low again, since only the very weak pullup is turned on at that point and the leakage is greater than the very weak pullup source current. These issues are the reason for enhancing the port configurations of the XA.

A diagram of the quasi-bidirectional output structure is shown in the figure below.

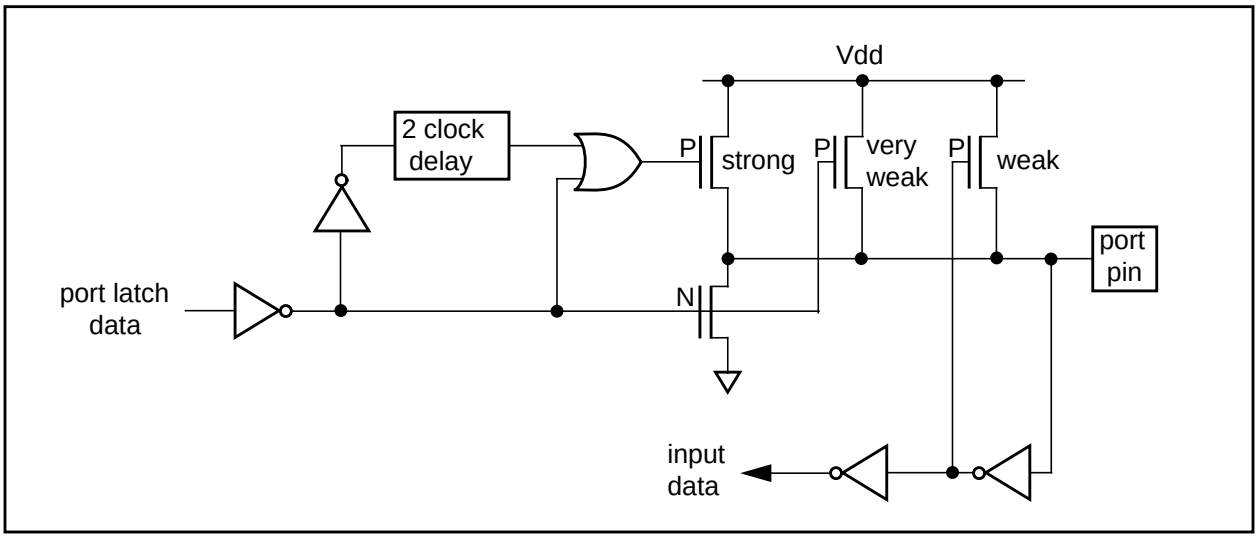


Figure 7.15 Structure of the Quasi-Bidirectional Output Configuration

Open Drain Output

Another port output configuration provided by the standard XA I/O ports is open drain. This configuration turns off all pullups and only drives the pulldown transistor of the port driver when the port latch contains a logic 0. To be used as a logic output, a port configured in this manner must have an external pullup, typically a resistor tied to Vdd. The pulldown for this mode is the same as for the quasi-bidirectional mode.

An advantage of the open drain output is that is may be used to create wired AND logic. Several open drain outputs of various devices can be tied together, and any one of them can drive the wire low, creating a logical AND function without using a logic gate. The figure below show the structure of the open drain output.

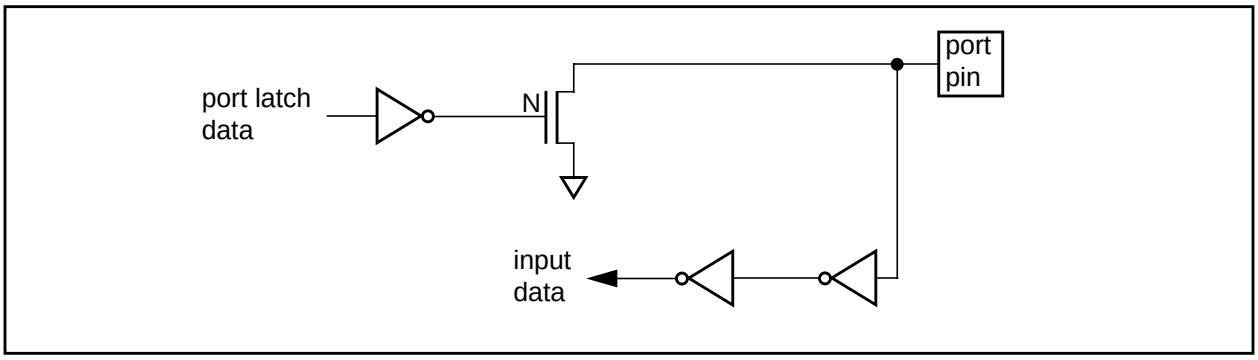


Figure 7.16 Structure of the Open Drain Output Configuration

Push-Pull Output

The push-pull output mode has the same pulldown structure as both the open drain and the quasi-bidirectional output modes, but provides a continuous strong pullup when the port latch contains a logic 1. This mode uses the same pullup as the strong pullup for the quasi-bidirectional mode. The push-pull mode may be used when more source current is needed from a port output. The output structure for this mode is shown below.

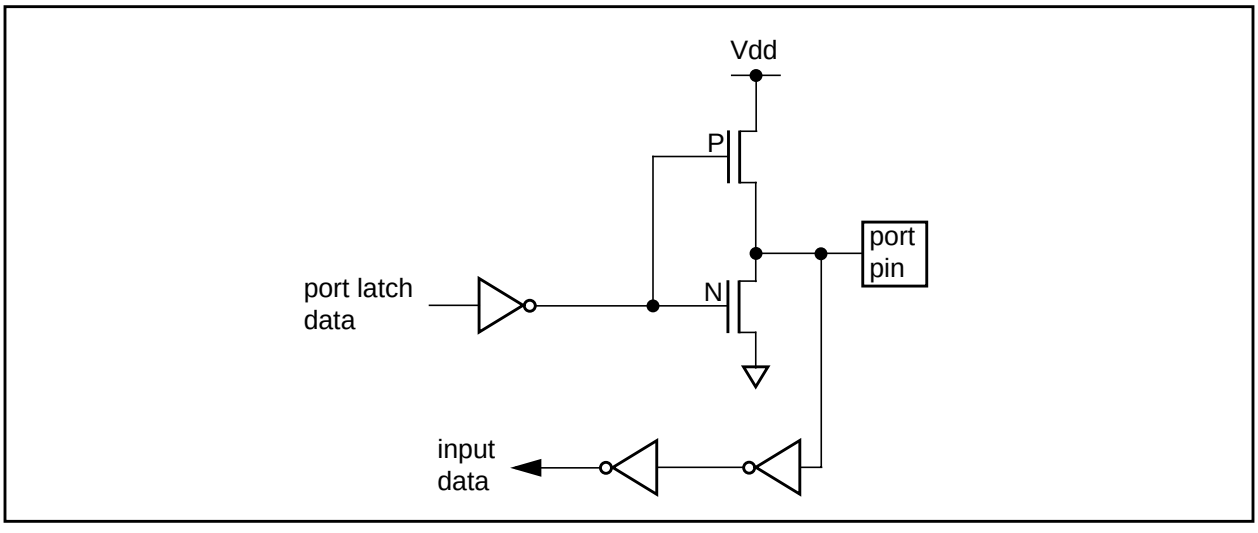


Figure 7.17 Structure of the Push-Pull Output Configuration

High Impedance Output

The final XA port output configuration is called high impedance mode. This mode simply turns all output drivers on a port pin off. Thus, the pin will not source or sink current and may be used effectively as an input-only pin with no internal drivers for an external device to overcome.

7.4.4 Reset State and Initialization

Upon chip reset, all of the port output configurations are set to quasi-bidirectional, and the port latches are written with all ones. The quasi-bidirectional output type is a good default at power-up or reset because it does not source a large amount of current if it is driven by an external device, yet it does not allow the port pin to float. A floating input pin on a CMOS device can cause excess current to flow in the pin's input circuitry, and of course all port pins have input circuits in addition to outputs.

7.4.5 Sharing of I/O Ports with On-Chip Peripherals

Since XA on-chip peripheral devices share device pins with port functions, some care must be taken not to accidentally disable a desired pin function by inadvertently activating another function on the same pin. A peripheral that has an output on a pin will use the I/O port output configuration for that pin (quasi-bidirectional, open drain, push-pull, or high impedance).

The method of sharing multiple functions on a single pin involves a logic AND of all of the functions on a pin. So, if a port latch contains a zero, it will drive that port pin low, and any peripheral output function on that pin is overridden. Conversely, an on-chip peripheral outputting a zero on a pin prevents the contents of the port latch from controlling the output level. It is usually not an issue to avoid turning on an alternate peripheral function on a pin accidentally, since most peripherals must be either explicitly turned on or activated by a write to one of their SFRs. It is more likely that a user program could erroneously write a zero to a port latch bit corresponding to a pin with a peripheral function that is being used and therefore disable that function. The simple rule to follow is: never write a zero to a port bit that is associated with an active on-chip peripheral, or that should only be used an input.

When an XA I/O port pin is used as an input for a peripheral function, it is sampled at the oscillator rate divided by 2. For example, if an XA is running at a 20 MHz clock (giving a 50 ns clock period), an external timer input would have to remain in the same state for at least 100 ns in order to guarantee that it is sampled correctly. This gives a maximum frequency for such inputs as the oscillator rate divided by 4. In this example, the maximum external timer input rate would be 5 MHz.

8 Special Function Register Bus

The Special Function Register Bus or SFR Bus is the means by which all Special Function Registers are connected to the XA CPU so that they may be read and written by user programs. This includes all of the registers contained in peripherals such as Timers and UARTs, as well as some CPU registers such as the PSW. CPU registers communicate functionally with the CPU via direct connections, but read and write operations performed on them are routed through the SFR bus.

The SFR bus provides a common interface for the addition of any new functions to the XA core, thus supplying the means for building a large and varied microcontroller derivative family. This is illustrated in Figure 8.1.

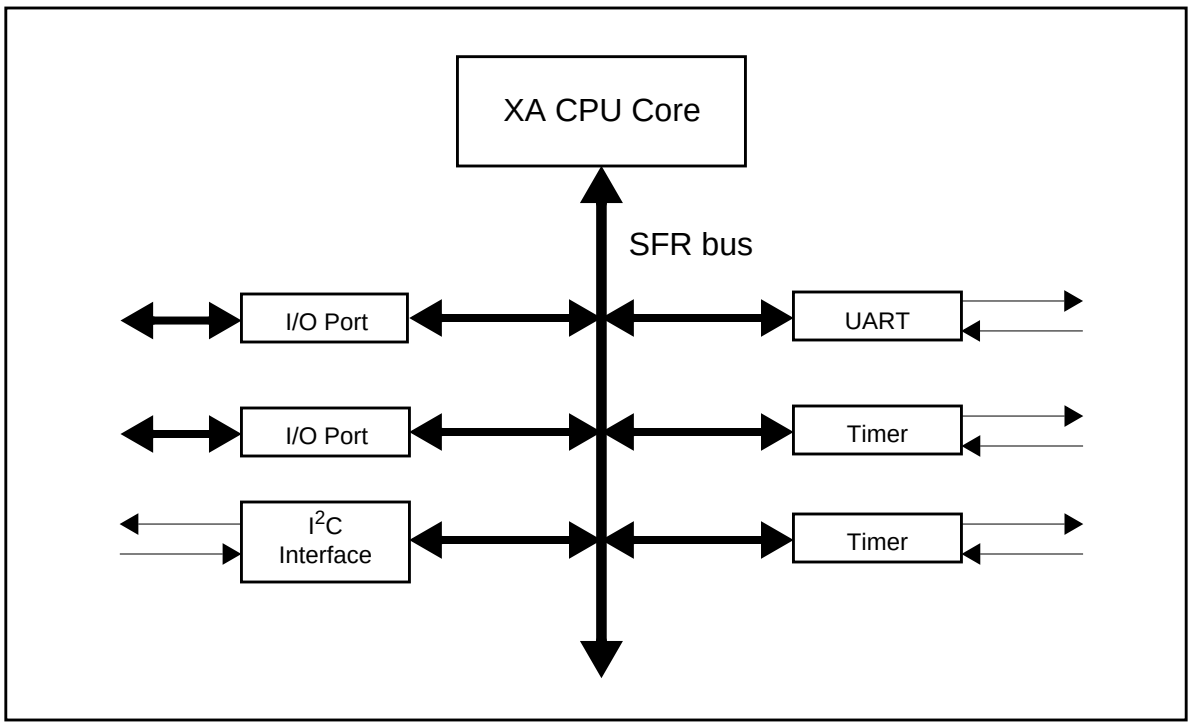


Figure 8.1. Example of peripheral functions connected to the XA SFR bus.

8.1 Implementation and Possible Enhancements

The SFR bus interface is itself not part of the XA CPU core, but a separate functional block. Since the SFR bus controller is a separate block, writes to SFRs may occur simultaneously with the beginning of execution of the next instruction. If the next instruction attempts to access the SFR bus while it is still busy, the instruction execution will stall until the SFR bus becomes available. SFR bus read and write clocks each take 2 CPU clocks to complete. However, the starting time of those 2 clocks has a one clock uncertainty, so the time from the SFR bus controller receiving a request until it is completed can be either 2 or 3 clocks.

The SFR bus implementation on initial XA derivatives is an 8-bit interface. This means that word reads and writes are not allowed. In the future, higher performance XA architecture implementations may expand the capabilities of the SFR bus by supporting 16-bit accesses.

One enhancement to the SFR bus would be to have it divide 16-bit access requests into two 8-bit accesses. This leaves the actual SFR bus width at 8 bits, but allows a user program to act as if it was 16-bits. The highest performance alternative is a full 16-bit SFR bus. This would require extra hardware in the XA to implement, but may eventually become necessary on order to achieve very high performance with some future enhanced XA core implementation.

8.2 Read-Modify-Write Lockout

Some of the SFRs that are accessed via the SFR bus contain interrupt flags and other status bits that are set directly by the peripheral device. When a read-modify-write operation is done on such an SFR, there is a possibility that a peripheral write to a flag bit in the same SFR could occur in the middle of this process. A standard mechanism is defined for the XA to deal with such cases, which is called Read-Modify-Write lockout. A read- modify-write is defined as an operation where a particular SFR is read, altered and written during the execution of a single XA instruction.

The instructions that fit this description are those that write to bits in SFRs and those that modify an entire SFR, except for the MOV instruction. This happens to be the same operations as those that read port latches rather than port pins as specified in Chapter 7, only the SFRs involved are different.

The mechanism used throughout XA peripherals to avoid losing status flags during a read-modify-write operation first involves detecting that such an operation is in progress. A signal from the CPU to the peripherals indicates such a condition. When a peripheral detects this, it prevents the CPU write to just those status flags that the peripheral has already updated since the beginning of the read-modify-write operation. This basically makes it look as if the peripheral flag update happened just after the read-modify-write operation completed, rather than during it. Once the read-modify-write operation is completed, a CPU write may affect all bits in these SFRs.

9 80C51 Compatibility

Many architectural decisions and features were guided by the goal of 80C51 compatibility when the XA core specification was written. The processor's memory configuration, memory addressing modes, instruction set, and many other things had to be taken into account.

9.1 Compatibility Considerations

Source code compatibility of the XA to the 80C51 was chosen as a goal for many reasons. Complete compatibility with an existing processor is not possible if the new processor is to have substantially higher performance.

The XA architecture makes use of a number of rules for 80C51 compatibility. An 80C51 to XA source code translator program is intended to be the means of providing compatibility between the architectures. For the translator software to be fairly simple, a one-to-one translation for all 80C51 instructions is a major consideration. The XA instruction set includes many instructions that are more powerful than 80C51 instructions and yet perform roughly the same function. 80C51 instruction can therefore be translated into those XA instructions. When this is not the case, an 80C51 instruction may be included in its original form in the XA. The XA memory map and memory addressing modes are also a superset of the 80C51, making source code translation easy to accomplish. Other CPU features are made compatible to the extent that such is possible. In rare cases, when this compatibility could not be provided for some important reason, the changes were kept to the minimum while maintaining the XA goals of high performance and low cost.

9.1.1 Compatibility Mode, Memory Map, and Addressing

Specific XA registers are reserved for use as 80C51 registers when translating code. The A register, the B register, and the data pointer all map to a pre-determined place in the XA register file (see figure 9.1). The accumulator (A) is the only one of these that required special hardware support in the XA, because the accumulator can be read or tested directly by certain instructions and in order to generate the parity flag.

The 4 banks of 8 byte registers that are found in the 80C51 are duplicated in the XA. The only difference is that in the XA, these registers do not normally overlap the lower 32 bytes of data memory space as they do in the 80C51. To allow code translation, a special 80C51 compatibility mode causes the XA register file to copy the 80C51 mapping to data memory. This mode is activated by the CM bit in the System Configuration Register (SCR).

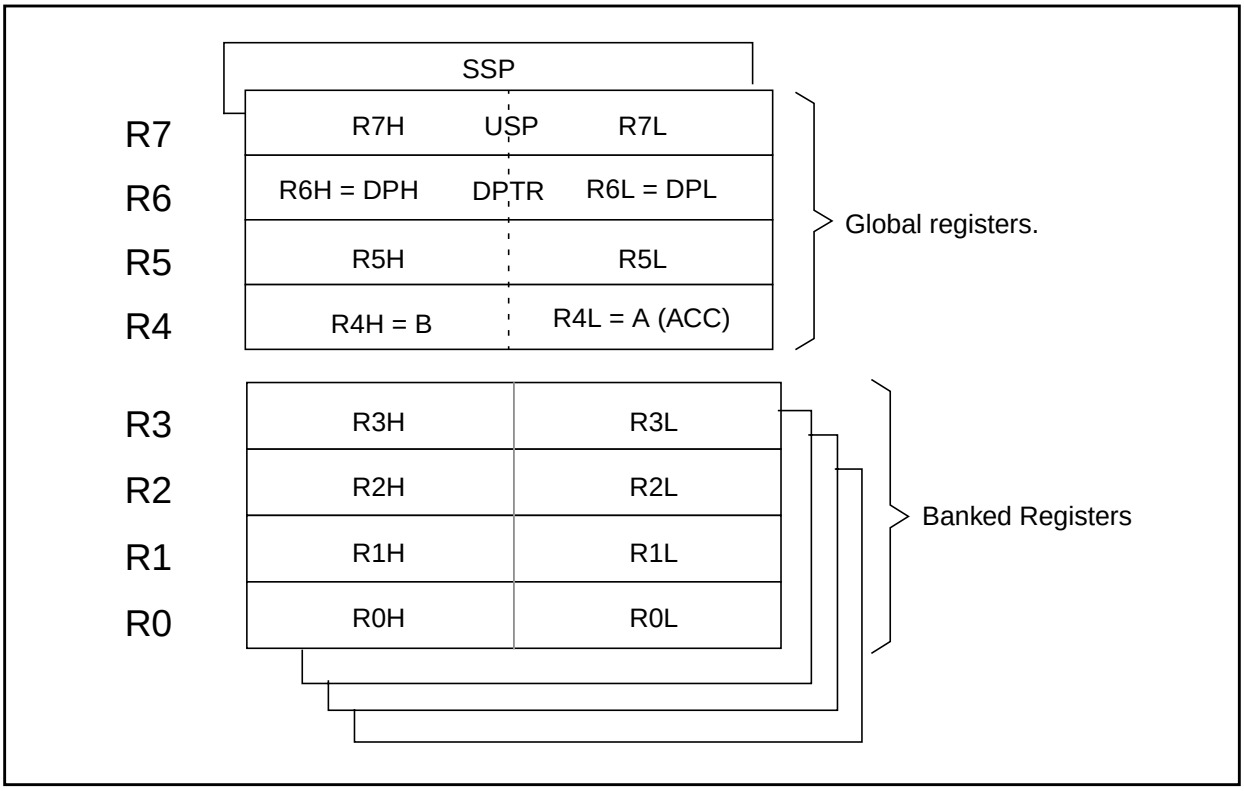


Figure 9.1. XA Register File

Other important registers of the 80C51 are provided in other ways. The program status word (PSW) of the XA is slightly different than the 80C51 PSW, so a special SFR address is reserved to provide an 80C51 compatible "view" of the PSW for use by translated code. This alternate PSW, called PSW51, is shown in the figure 9.2. The F0 flag and the F1 flag are simply readable

Figure 9.2. PSW CPU status flags

and writable bits. The P flag provides an even parity bit for the 80C51 A register and always reflects the current contents of that register. Note that the P flag, the F0 flag, and the F1 flag only appear in the PSW51 register.

The 80C51 indirect data memory access mode, using R0 or R1 as pointers, requires special support on the XA, where pointers are normally 16 bits in length. The 80C51 compatibility mode also causes the XA to mimic the 80C51 indirect scheme, using the first two bytes of the register file as indirect pointers, each zero extended to make a 16-bit address. Due to this and the previously mentioned register overlap to memory feature, the compatibility mode must be turned on in order to execute most translated 80C51 code on the XA. Other than the two aforementioned effects, nothing else about XA functioning is affected by the compatibility mode.

The 80C51 mapped the special function registers (SFRs) into the direct address space, from address 80 hex to FF hex. SFRs were only accessed by instruction that contain the entire SFR address, so translation to the XA is fairly simple. Since references to SFRs are normally done by their name in 80C51 source code, the translation just copies the name into the XA code output. If an SFR happened to be referred to by its address, its name must be found so that it can be inserted into the XA code. This would require that an SFR table be available for the 80C51 derivative for which the code was originally written.

The XA has another mode which may be useful for translated 80C51 code. In order to save stack space as well as speed up execution, a Page Zero (PZ) mode causes return addresses on the stack to be saved as 16 bits only, instead of the usual 24 bits (which occupy 32 bits due to word alignment on the XA stack). All other program and data addresses are also forced to be 16-bits. If an entire 80C51 application program is translated to the XA, it will very likely fit within this 64K limit, allowing the use of this mode.

Other aspects of the processor stack have been altered on the XA. For one, the standard direction of stack growth for 16 bit processors has been adopted. So, the XA stack grows downward, from higher to lower addresses in data memory. The stack can now be nearly 64K in size if necessary, and begin anywhere in its data segment so may be easily moved to a new location for translated 80C51 applications. This stack direction change is important to match the stack contents to normal data memory accesses on the XA.

80C51 code translated to run on the XA will also tend to use more stack space for two reasons. First, the PSW is automatically saved during interrupt and exception processing on the XA. The original 80C51 code should have also saved the PSW explicitly, but the XA PSW is 16 bits in length. Secondly, the initial implementation of the XA allows only word writes to the stack. Both byte and word operations may be performed, but both types of operations use 16 bits of stack space.

The tendency for stack size increase, in addition to the stack growth direction will require some changes to be made if a complete 80C51 application program is translated to run on the XA.

9.1.2 Interrupt and Exception Processing

Interrupt handling on the XA is inherently much more powerful than it was on the 80C51. Along with this added power and flexibility comes some difference that must be taken into account for 80C51 code conversion.

Previously noted was the fact that the XA automatically saves the PSW during interrupt processing. If an 80C51 program relied on this not being the case somehow, it would not work without alteration. This type of reliance is not found in code using common programming practices and should be very rare.

The XA allows up to 15 interrupt priority levels, compared to only 2 in the standard 80C51, although up to 4 levels are available in a few of the newer 80C51 variations. These priorities are stored as 4-bit values, with the priority for 2 interrupts found in the same SFR byte. This is

different (and much more powerful) than any 80C51 derivative, and will require minor changes to code that is translated.

The method of entering an interrupt routine in the XA uses a vector table stored in low addresses of the code memory. Each interrupt or exception source has a vector which consists of the address of the handler routine for that event and a new PSW value that is loaded when the vector is taken. This differs from the 80C51 approach of fixed addresses for the interrupt service routines, and again is a much more flexible and powerful method. So, if a complete 80C51 application program is converted for the XA, the interrupt service routines must be re-located above the XA vector table and the new address stored in the table, a very simple process.

9.1.3 On-Chip Peripherals

Compatibility with standard on-chip peripherals found in the 80C51 has been kept in the XA whenever possible and reasonable, but not to the extent that some enhancements are not made. The set of standard peripheral devices includes the UART, Timers 0 and 1, and Timer 2 from the 80C52.

The XA UART has been enhanced in a way that does not affect translated 80C51 code. Some additional features are added through the use of a new SFR, such as framing error detection, overrun detection, and break detection.

Timers 0 and 1 remain the same except for one difference in the function, and a difference in timing. The functional change was to remove the 8048 timer mode (mode 0) and replace it with something much more useful: a 16-bit auto-reload mode. Sixteen bit reload registers (formed by RTHn and RTLn) had to be added to Timers 0 and 1 to support the new mode 0. In mode 2, RTLn also replaces THn as the 8-bit reload register.

The relationship of all timer count rates to the microcontroller oscillator has also been changed. This adds flexibility since this is now a programmable feature, allowing oscillator divided by 4, 16, or 64 to be used as the base count rate for all of the timers. Since XA performance is much higher (on a clock-by clock basis), an application converted to the XA from the 80C51 would likely not use the same oscillator frequency anyway.

9.1.4 Bus Interface

The customary 80C51 bus control signals are all found on the standard external XA bus. To provide the best performance, the details of some of these signals have changed somewhat, and a few new ones have been added. In addition to the well known ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{EA}}$, there are now also WAIT and $\overline{\text{WRH}}$. The WAIT signal causes wait states to be inserted into any XA bus clock as long as it is asserted. The $\overline{\text{WRH}}$ signal is used to distinguish writes to the high order byte when the XA bus is configured to be 16 bits wide.

The multiplexed address/data bus has undergone some renovations on the XA as well. To get the most performance in a system executing code from the external bus, the XA separates the 4 least significant address lines on to their own pins. Since these lines normally change the most often, an ALE clock would be required on every external code fetch if these lines were multiplexed as they are on the 80C51. The 80C51 had time to do this since its performance was not that high.

The XA, however, uses only as many clocks as are needed to execute each instruction, so an ALE for every fetch would slow things down considerably. With this change, up to 16 bytes (or 8 words) of code may be accessed without the need to insert an ALE cycle on the XA bus.

The number of XA clocks used for each type of bus cycle (code read, data read, or data write) can also be programmed, so that slower peripheral devices can work with the XA without the need for an external WAIT state generator.

Due to the various changes to the bus just mentioned, an XA device cannot be completely pin compatible with an 80C51 derivative if the external bus is used. The changes to application hardware needed are relatively small and easy to make.

9.1.5 Instruction Set

The simplest goal of the XA for instruction set compatibility was to have every 80C51 instruction translate to one XA instruction. That has been achieved but for a single exception. The 80C51 instruction, XCHD or exchange digits, cannot be translated in that manner. XCHD is an instruction that is rarely used on the 80C51 and could not be implemented on the XA, due to its internal architecture, without adding a great deal of extra circuitry. So, if this instruction is encountered when 80C51 source code is being translated, a sequence of XA instructions is used to duplicate the function:

```
PUSH    R4H         ; Save temporary register.
MOV     R4H,(Ri)    ; Get second operand.
RR      R4H,#4      ; Swap one byte.
RR      R4L,#4      ; Swap second byte (the "A" register).
RL      R4,#4       ; Swap word.
                    ; Result is swapped nibbles in A and R4H.
MOV     (Ri),R4H    ; Store result.
POP     R4H         ; Restore temporary register.
```

If the application requires this sequence to not be interruptible, some additional instruction must be added in order to disable and re-enable interrupts. The table at the end of this section shows all of the other XA code replacements for 80C51 instructions.

The XA instruction set is much more powerful than the 80C51 instruction set, and as a direct consequence, the average number of bytes in an instruction is higher on the XA. In code written for the XA, the capability of a single instruction is high, so the size of an entire XA program will normally be smaller than the same program written for an 80C51. Of course, this depends on how much the application can take advantage of XA features. When code is translated from 80C51 source, however, the size change can be an issue.

In the case of a jump table, where the JMP @A+DPTR instruction is used to jump into a table of other jumps composed of the 80C51 AJMP instruction, the XA cannot always duplicate the function of the jumps in the table with instructions that are 2 bytes in length, as in the case of the AJMP instruction. An adjustment to the calculation of the table index will be required to make the translated code work properly. For a data table, accessed using MOVC @A+PC, the distance to the table may change, requiring a similar index adjustment.

Since the XA optimizes the timing of each instruction, there will be very little correspondence to the original 80C51 timing for the same code prior to translation to the XA. If the exact timing of a sequence of instructions is important to the application, the translated code must be altered, perhaps by adding NOPs or delay loops, to provide the necessary timing.

To show how a simple 80C51 to XA source code translator might work, a subroutine was extracted from a working 80C51 program and translated using the table at the end of this document and the other rules presented here. The original 80C51 source code was:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,A          ; Save step target for later use.
          MOV     B,#Steplow       ; Get low byte of step increment.
          MUL     AB               ; Multiply this by the step target.
          MOV     StepResult,B     ; Save high byte as partial result.
          MOV     Temp1,A          ; Save low byte to use for rounding.

          MOV     A,Temp2          ; Get back the step target.
          MOV     B,#StepHigh      ; Get high byte of step increment,
          MUL     AB               ; and multiply the two.

          ADD     A,StepResult     ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          INC     A                ; If so, round up the final result.
Exit:     ADD     A,#MotorBot      ; Add in the 0 step displacement.
          MOV     StepResult,A     ; Save final step target.
          RET
```

The same code as translated for the XA is as follows:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,R4L        ; Save step target for later use.
          MOV     R4H,#Steplow     ; Get low byte of step increment.
          MULU.b  R4,R4H           ; Multiply this by the step target.
          MOV     StepResult,R4H   ; Save high byte as partial result.
          MOV     Temp1,R4L        ; Save low byte to use for rounding.

          MOV     R4L,Temp2        ; Get back the step target.
          MOV     R4H,#StepHigh    ; Get high byte of step increment,
          MULU.b  R4,R4H           ; and multiply the two.

          ADD     R4L,StepResult   ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          ADDS    R4L,#1           ; If so, round up the final result.
Exit:     ADD     R4L,#MotorBot    ; Add in the 0 step displacement.
          MOV     StepResult,R4    ; Save final step target.
          RET
```

In this case, the translated code actually changed very little. Primarily, the 80C51 register names have been replaced by the new ones reserved for them in the XA. The increment (INC) instruction became a short add (ADDS), and the mnemonic for multiply (MUL) changed to MULU8.

Some basic statistical information about these code samples may be found in table 9.1. These statistics show a large performance increase for the XA code. This is significant because the code is only simple translated 80C51 code and therefore does not take any advantage of the XA's unique features.

Table 9.1: 80C51 to XA Code Translation Statistics

Statistic	80C51 code	XA translation	Comments
Code bytes	28	40	- one NOP added for branch alignment on XA
Clocks to execute	300	78	- includes XA pre-fetch queue analysis, raw execution is 66 clocks
Time to execute @ 20MHz	15 µsec	3.9 µsec	- a nearly 4x improvement without any optimization

9.2 Code Translation

Table 9.2 shows every 80C51 instruction type and the XA instruction that replaces it. An actual 80C51 to XA source code translator can make use of this table, but must also flag the compatibility exceptions noted in this section, so that any necessary adjustments may be made to the resulting XA source code.

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Arithmetic operations	

ADD A, Rn
ADD A, #data8
ADD A,dir8
ADD A, @Ri
ADDC A, Rn
ADDC A, #data8
ADDC A,dir8
ADDC A, @Ri
	ADD.b R, R
	ADD.b R, #data8
	ADD.b R, direct
	ADD.b R, [R]
	ADDC.bR, R
	ADDC.bR, #data8
	ADDC.bR, direct
	ADDC.bR, [R]

SUBB A, Rn
SUBB A, #data8
SUBB A, dir8
SUBB A, @Ri
	SUBB.bR, R
	SUBB.bR, #data8
	SUBB.bR, direct
	SUBB.bR, [R]

INC Rn INC dir8 INC @Ri INC A INC DPTR	ADDS.bR, #1 ADDS.bdirect, #1 ADDS.b[R], #1 ADDS.bR, #1 ADDS.wR, #1
DEC Rn DEC dir8 DEC @Ri DEC A	ADDS.bR, #-1 ADDS.bdirect, #-1 ADDS.b[R], #-1 ADDS.bR, #-1
MUL AB DIV AB DA A	MULU.bR, R DIVU.b R, R DA R

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Logical operations	
ANL A, Rn ANL A, #data8 ANL A, dir8 ANL A, @Ri ANL dir8, A ANL dir8, #data8	AND.b R, R AND.b R, #data8 AND.b R, direct AND.b R, [R] AND.b direct, R AND.b direct, #data8
ORL A, Rn ORL A, #data8 ORL A, dir8 ORL A, @Ri ORL dir8, A ORL dir8, #data8	OR.b R, R OR.b R, #data8 OR.b R, direct OR.b R, [R] OR.b direct, R OR.b direct, #data8
XRL A, Rn XRL A, #data8 XRL A, dir8 XRL A, @Ri XRL dir8, A XRL dir8, #data8	XOR.b R, R XOR.b R, #data8 XOR.b R, direct XOR.b R, [R] XOR.b direct, R XOR.b direct, #data8
CLR A CPL A SWAP A	MOVS R, #0 CPL.b R RL.b R, #4
RL A RLC A RR A RRC A	RL.b R, #1 RLC.b R, #1 RR.b R, #1 RRC.b R, #1

CLR C
CLR bit
SETB C
SETB bit
CPL C
CPL bit
ANL C, bit
ANL C, /bit
ORL C, bit
ORL C, /bit
MOV C, bit
MOV bit, C
	CLR bit
	CLR bit
	SETB bit
	SETB bit
	XOR.b PSWL, #data8
	XOR.b direct, #data8
	AND C, bit
	AND C, /bit
	OR C, bit
	OR C, /bit
	MOV C, bit
	MOV bit, C

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Data transfer	
MOV A, Rn	MOV.b R, R
MOV A, #data8	MOV.b R, #data8
MOV A, dir8	MOV.b R, direct
MOV A, @Ri	MOV.b R, [R]
MOV Rn, A	MOV.b R, R
MOV Rn, #data8	MOV.b R, #data8
MOV Rn, dir8	MOV.b R, direct
MOV dir8, A	MOV.b direct, R
MOV dir8, #data8	MOV.b direct, #data8
MOV dir8, Rn	MOV.b direct, R
MOV dir8, dir8	MOV.b direct, direct
MOV dir8, @Ri	MOV.b direct, [R]
MOV @Ri, A	MOV.b [R], R
MOV @Ri, dir8	MOV.b [R], direct
MOV @Ri, #data8	MOV.b [R], #data8
MOV DPTR, #data16	MOV.w R, #data16
XCH A, Rn	XCH.b R, R
XCH A, dir8	XCH.b R, direct
XCH A, @Ri	XCH.b R, R
XCHD A, @Ri	a sequence (see text)
PUSH dir8	PUSH.bdirect
POP dir8	POP.b direct
MOVX A, @Ri	MOVX.bR, [R]
MOVX A, @DPTR	MOVX.bR, [R]
MOVX @Ri, A	MOVX.b[R], R
MOVX @DPTR, A	MOVX.b[R], R
MOVC A, @A+DPTR	MOVC.bA, [A+DPTR]
MOVC A, @A+PC	MOVC.bA, [A+PC]

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Relative branches	
SJMP rel8	BR rel8
CJNE A, dir8, rel CJNE A, #data8, rel CJNE Rn, #data8, rel CJNE @Ri, #data8, rel	CJNE.b R, direct, rel CJNE.b R, #data8, rel CJNE.b R, #data8, rel CJNE.b [R], #data8, rel
DJNZ Rn, rel DJNZ dir8, rel	DJNZ.b R, rel DJNZ.b direct, rel
JZ rel JNZ rel JC rel JNC rel	JZ rel JNZ rel BCS rel BCC rel
Jumps, Calls, Returns, and Misc.	
NOP	NOP
AJMP addr11 LJMP addr16 JMP @A+DPTR	JMP rel16 JMP rel16 JUMP [A+DPTR]
ACALL addr11 LCALL addr16	CALL rel16 CALL rel16
RET RETI	RET RETI

9.3 New Instructions on the XA

While the XA instructions that are similar to 80C51 instructions have a larger addressing range, more status flags, etc., the XA also has many entirely new instructions and addressing modes that make writing new code for the XA much easier and more efficient. The new addressing modes also make the XA work very well with high level language compilers. A complete list of the new XA instructions and addressing modes is shown in Table 9.3.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
alu.w ..., ...	All of the 80C51 arithmetic and logic instructions with a 16-bit data size.
SUBB R,...	Subtract (without borrow), all addressing modes.
alu [R], R	Arithmetic and logic operations (ADD, ADDC, SUB, SUBB, CMPAND, OR, XOR, and MOV) from a register to an indirect address.
alu R, [R+]	Arithmetic and logic operations from an indirect address to a register, with the indirect pointer automatically incremented.
alu R,[R+offset8/16]	Arith/Logic operations from an indirect offset address (with 8 or 16-bit offset) to a register.
alu direct, R	The 80C51 has only MOV direct, R.
alu [R], R	The 80C51 has only MOV [R], R.
alu [R+], R	Arith/Logic operations from a register to an indirect address, with the indirect pointer automatically incremented.
alu [R+offset8/16], R	Arith/Logic operations from a register to an indirect offset address (with 8 or 16-bit offset).
alu direct, #data8/16	Arith/Logic operations to a direct address with 8 or 16-bit immediate data.
alu [R], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data.
alu [R+], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data with the indirect pointer automatically incremented.
alu [R+offset8/16], #data8/16	Arith/Logic operations to an indirect offset address (with 8 or 16-bit offset), with 8 or 16-bit immediate data.
MOV direct, [R]	Move data from an indirect to a direct address.
ADDS R, #data4	The 80C51 can only increment or decrement a register by 1. ADDS has a range of +7 to -8.
ADDS [R], #data4	Add a short value to an indirect address.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
ADDS [R+], #data4	Add a short value to an indirect offset address, with the indirect pointer automatically incremented.
ADDS [R+offset8/16], #data4	Add a short value to an indirect offset address (with 8 or 16-bit offset).
ADDS direct, #data4	Add a short value to a direct address.
MOVS ..., #data4	Move short data to destination using any of the same addressing modes as ADDS.
ASL R, R	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from register.
ASR R, R	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from register.
LSR R, R	Logical shift right a byte, word, or double word, up to 31 places, shift count read from register.
ASL R, #DATA4/5	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from instruction.
ASR R, #DATA4/5	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
LSR R, #DATA4/5	Logical shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
DIV R, R	Signed divide of 32 bits register by 16 bit register, or 16 bit register by 8 bit register.
DIVU R, R	Unsigned divide of 32 bit register by 16 bit register, or 16 bit register by 8 bit register.
MUL R, R	Signed multiply of 16 bit register by 16 bit register, or 8 bit register by 8 bit register.
MULU R, R	Unsigned multiply of 16 bit register by 16 bit register.
DIV R, #data8/16	Signed divide of 32 bits register by 16 bit immediate, or 16 bit register by 8 bit immediate.
DIVU R, #data8/16	Unsigned divide of 32 bit register by 16 bit immediate, or 16 bit register by 8 bit immediate.
MUL R, #data8/16	Signed multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
MULU R, #data8/16	Unsigned multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.
LEA R, R+offset8/16	Load effective address, duplicates the offset8 or 16-bit addressing mode calculation but saves the address in a register.
NEG R	Negate, performs a twos complement operation on a register.
SEXT R	Sign extend, copies the sign flag from the last operation into an 8 or 16-bit register.
NORM R, R	Normalize. Shifts a byte, word, or double word register left until the MSB becomes a 1. The number of shifts used is stored in a register.
RL, RR, RLC, RRC R,#data4	All of the 80C51 rotate modes with 16-bit data size and a variable number of bit positions (up to 15 places).
MOV [R+], [R+]	Block move. Move data from an indirect address to another indirect address, incrementing both pointers.
MOV R, USP and USP, R	Allows system code to move a value to or from the user stack pointer. Handy in multi-tasking applications.
MOVC R, [R+]	Move data from an indirect address in the code space to a register, with the indirect pointer automatically incremented.
PUSH and POP Rlist	PUSH and POP up to 8 word registers in one instruction.
PUSHU and POPU Rlist or direct	Allows system code to write to or read the user stack. Handy in multi-tasking applications.
conditional branches	A complete set of conditional branches, including BEQ, BNE, BG, BGE, BGT, BL, BLE, BMI, BPL, BNV, and BOV.
CALL [R]	Call indirect, to an address contained in a register.
CALL rel16	Call anywhere in a +/- 64K range.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
FCALL addr24	Far call, anywhere within the XA 16Mbyte code address space.
JMP [R]	Jump indirect, to an address contained in a register.
JMP rel16	Jump anywhere in a +/- 64K range.
FJMP addr24	Far jump, anywhere within the XA 16Mbyte code address space.
JMP [[R+]]	Jump double indirect with auto-increment. Used to branch to a sequence of addresses contained in a table.
BKPT	Breakpoint, a debugging feature.
RESET	Allows software to completely reset the XA in one instruction.
TRAP #data4	Call one of up to 16 system services. Acts like an immediate interrupt.