XC800

Microcontroller Family

Architecture and Instruction Set

Microcontrollers

Never stop thinking

Edition 2006-02

**Published by Infineon Technologies AG,
81726 München, Germany**

Attention please!

The information herein is given to describe certain components and shall not be considered as a guarantee of characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

XC800

Microcontroller Family

Architecture and Instruction Set

Microcontrollers

Never stop thinking

Revision History:	**2006-02**	V 0.2
Previous Version:	V 0.1	

Page	Subjects (major changes since last revision)
	Only minor enhancements; syntax corrections

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents **Page**

Table of Contents **Page**

1 Fundamental Structure

This manual provides an overview of the architecture and functional characteristics of the XC800 microcontroller family. It also includes a complete description of the XC800 CPU instruction set. For detailed information on the different derivatives of the XC800 8-bit microcontrollers, refer to the respective user's manuals.

1.1 Introduction

The Infineon XC800 microcontroller family has a CPU which is functionally upward compatible to the 8051. While the standard 8051 CPU is designed around a 12-clock machine cycle, the XC800 CPU uses a two-clock period machine cycle.

The instruction set consists of 45% one-byte, 41% two-byte, and 14% three-byte instructions. Each instruction takes 1, 2 or 4 machine cycles to execute. In case of access to slower memory, the access time may be extended by wait states.

The XC800 microcontrollers support via the dedicated JTAG interface or the standard UART interface, a range of debugging features including basic stop/start, single-step execution, breakpoint support and read/write access to the data memory, program memory and special function registers.

The key features of the XC800 microcontrollers are listed below.

- Two clocks per machine cycle
- Up to 1 Mbyte of external data memory; up to 256 bytes of internal data memory
- Up to 1 Mbyte of program memory
- Support for synchronous or asynchronous program and data memory
- Wait state support for slow memory
- Program memory download option
- 15-source, 4-level interrupt controller
- Up to eight data pointers
- Power saving modes
- Dedicated debug mode via the standard JTAG interface or UART
- Two 16-bit timers (Timer 0 and Timer 1)
- Full-duplex serial port (UART)

1.2 Memory Organization

The memory partitioning of the XC800 microcontrollers is typical of the Harvard architecture where data and program areas are held in separate memory space. The on-chip peripheral units are accessed using an internal Special Function Register (SFR) memory area that occupies 128 bytes of address, which can be mapped or paged to increase the number of addressable SFRs.

A typical memory map of the code space consists of internal ROM/Flash, on-chip Boot ROM, an on-chip XRAM and/or external memory. The memory map of the data space is

typical of the standard 8051 architecture: the internal data memory consists of 128 bytes of directly addressable Internal RAM (IRAM) and 128 bytes of indirect addressable IRAM. On-chip ‘external’ RAM (XRAM) is also supported. External data memory may be supported outside of the internal range. **Figure 1-1** provides a general overview of the XC800 memory space and a typical memory map in user mode.

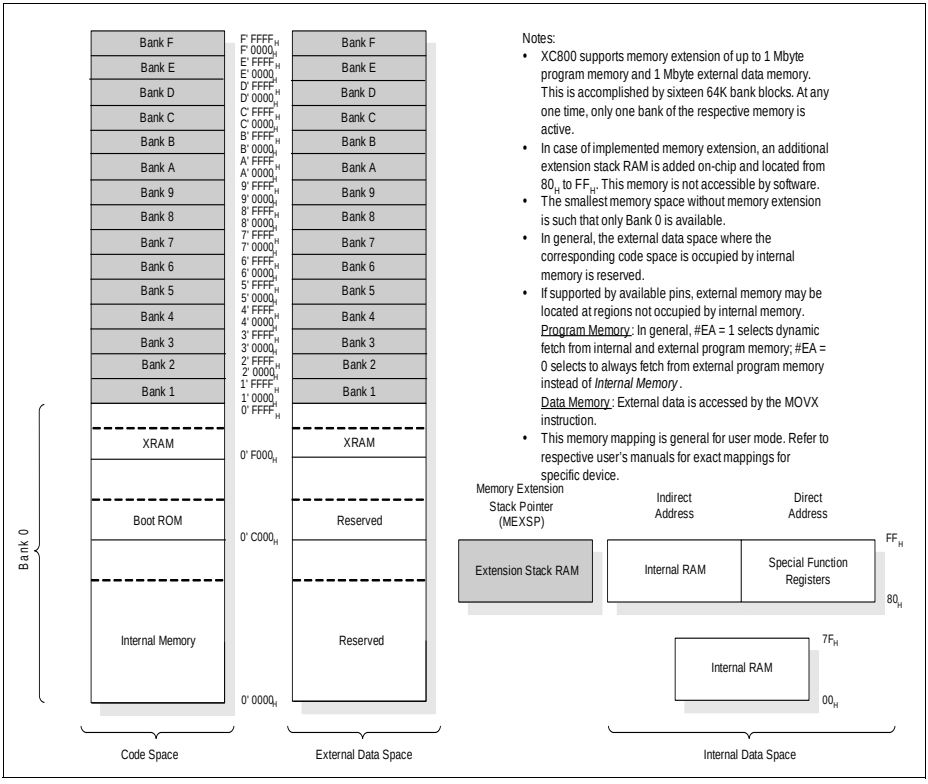


Figure 1-1 XC800 Memory Space and Typical Memory Map in user mode

In derivatives with memory extension, an additional 128 bytes of memory extension stack RAM is available from 80_H to FF_H. Access to this memory is only possible by the hardware, so the memory is effectively transparent to the user. By default after reset, the memory extension stack pointer (MEXSP) points to $7F_H$. It is pre-incremented by call instructions and post-decremented by return instructions.

1.2.1 Memory Extension

The standard amount of addressable program or external data memory in an 8051 system is 64 Kbytes. The XC800 core supports memory expansion of up to 1 Mbyte and this is enabled by the availability of a Memory Management Unit (MMU) and a Memory Extension Stack. The MMU adds a set of Memory Extension registers (MEX1, MEX2, and MEX3) to control access to the extended memory space by different addressing modes. The Memory Extension Stack is used by the hardware to ‘push’ and ‘pop’ values of MEX1.

Program Code is always fetched from the 64-Kbyte block pointed to by the 4-bit Current Bank (CB) register bit field. It is updated from a 4-bit Next Bank (NB) bit field upon execution of long jump (LJMP) and call instructions. CB and NB together constitute the MEX1 register. The programmer simply writes the new bank number to NB before a jump or call instruction.

Interrupt service routines are always executed from code in the 64-Kbyte block pointed to by the Interrupt Bank (IB) register bit field. Further, memory constant data reads (in code space) and external data accesses may take place in banks other than the current bank. These banks are pointed to by the Memory Constant Bank pointer (MCB) and XRAM Bank pointer (MX). These bit fields are located in MEX2 and MEX3 registers.

1.2.1.1 Memory Extension Stack

Interrupts and Calls in Memory Extension mode make use of a Memory Extension Stack, which is updated at the same time as the standard stack.

The Memory Extension Stack is addressed using the SFR Memory Extension Stack Pointer MEXSP. This read/write register provides for a stack depth of up to 128 bytes (Bit 7 is always 0). The SFR is pre-incremented by each call instruction that is executed, and post-decremented by return instructions. MEXSP is by default reset to $7F_H$ so that the first increment selects the bottom of the stack. No indication of stack overflow is provided.

1.2.1.2 Memory Extension Effects

The following instructions can change the 64-Kbyte block pointed to: MOVC, MOVX, LJMP, LCALL, ACALL, RET and RETI.

Relative jumps (e.g. SJMP), indirect jumps (JMP @A+DPTR) and absolute jumps within 2-Kbyte regions (AJMPs), however, will in no way change the current bank. In other words, these instructions do not deselect the active 64-Kbyte bank block.

Move Constant Instructions (MOVC)

MOVC instructions access data bytes in either the Current bank (CB19 – CB16) or a ‘Memory Constant’ bank, defined by the MCB19 – MCB16 bit field in MEX3 and MEX2. The bank selection is done by the MCM bit in MEX2 (MEX2.7).

Move External Data Instructions (MOVX)

MOVX instructions can either access data in the Current bank or a ‘Data Memory’ bank, defined by the MX19 – MX16 bits in MEX3. The bank selection is done by the MXM bit in MEX3 (MEX3.3).

Long Jump Instructions (LJMP)

When a jump to another bank of the Memory Extension is required, the Next Bank bits NB19 – NB16 in MEX1 (MEX1.3 – MEX1.0) must be set to the appropriate bank address before the LJMP instruction is executed. When the LJMP is encountered in the code, the Next Bank bits (NB19 – 16) are copied to the Current Bank bits CB19 – CB16 in MEX1 (MEX1.7 – MEX1.4) and appear on address bus at the beginning of the next program fetch cycle.

Note: The Next Bank Bits (NB19 – 16) are not changed by the jump.

CALL Instructions (LCALL and ACALL)

Whenever an LCALL occurs, the MMU carries out the following sequence of actions:

1. The Memory Extension Stack Pointer is incremented.
2. The MEX1 register bits are made available on data bus.
3. The MEXSP register bits [6:0] are made available on address lines.
4. The Memory Extension Stack read and write signals are set for a write operation.
5. A write is performed to the Memory Extension Stack.
6. The Next Bank bits NB19 – NB16 (MEX1.3 – MEX1.0) are copied to the CB19 – CB16 bits (MEX1.7 – MEX.4).

Return Instructions (RET and RETI)

On leaving a subroutine, the MMU carries out the following sequence of actions:

1. The MEXSP register bits [6:0] are made available on address.
2. The Memory Extension Stack read and write signals are set for a read operation.
3. A read is performed on the Memory Extension Stack.
4. Memory Extension Stack data is written to the MEX1 register.
5. The Memory Extension Stack Pointer is decremented.

1.2.2 Program Memory

Up to 1 Mbyte of synchronous or asynchronous internal and/or external program memory is supported. Program memory extension, if supported by the XC800 derivative, is accomplished with a 4-bit Current Bank pointer (CB). The program code is fetched from the 64-Kbyte block pointed to by CB. The minimum supported code space is therefore 64 Kbytes.

If the internal program memory is used, the $\overline{EA}$ (External Access) pin must be held at high level. With $\overline{EA}$ held high, the microcontroller executes instructions internally unless the address (Program Counter) is outside the range of the internal program memory. In this case, dynamic code fetch from internal and external program memory is supported if the external memory bus is available on the derivative. If the $\overline{EA}$ pin is held at low-level, the microcontroller executes program code from external program memory, instead of from internal memory. The general exception is for accesses to address ranges of the active Boot ROM, internal XRAM and code-space data (e.g., Data Flash), where fetch is always from the internal memory regardless of the status of $\overline{EA}$ pin.

Most XC800 derivatives include a section for Boot ROM code, the size of which depends on the derivative. Usually, the Boot ROM code is executed first after reset where the Boot ROM is mapped starting from base address 0000_H of the code space. The Boot ROM code will switch the memory mapping so that before control is passed to the user code, the standard memory map (of the derivative) is active where user code could run starting from address 0000_H.

For program memory implemented as RAM, the XC800 core supports write to program memory with the instruction ‘MOVC @(DPTR++),A’. This is generally supported by the XC800 derivatives for writes to internal memory only.

1.2.3 Data Memory

The data memory space consists of internal and external memory portions. The internal data memory area is addressed using 8-bit addresses. The external data memory and the internal XRAM data memory are addressable by 8-bit or 16-bit indirect address with ‘MOVX’, additionally with up to 4-bit for selection of extended memory bank (maximum 1 Mbytes).

1.2.3.1 Internal Data Memory IRAM

The internal data memory is divided into two physically separate and distinct blocks: the 256-byte IRAM and the 128-byte SFR area. While the upper 128 bytes of IRAM and the SFR area share the same address locations, they are accessed through different addressing modes. The lower 128 bytes of IRAM can be accessed through either direct or register indirect addressing while the upper 128 bytes of IRAM can be accessed through register indirect addressing only. The special function registers are accessible through direct addressing.

The 16 bytes of IRAM that occupy addresses from 20_H to $2F_H$ are bitaddressable. Bit 0 of the internal data byte at 20_H has the bit address 00_H, while bit 7 of the internal data byte at $2F_H$ has the bit address $7F_H$.

By default after reset, the stack pointer points to address 07_H. The stack may reside anywhere in the IRAM.

IRAM occupying direct addresses from 30_H to $7F_H$ can be used as scratch pad.

1.2.3.2 On-Chip External Data Memory XRAM

The size of the on-chip XRAM is not fixed and varies depending on XC800 derivative. The XRAM is mapped to both the external data space and the code space because it can be accessed using both ‘MOVX’ and ‘MOVC’ instructions. When accessed using the 8-bit MOVX instruction via register R0 or R1, the SFR XADDRH must be initialized to specify the upper address byte.

If the derivative supports only on-chip XRAM or the application only access on-chip XRAM, the external interface ports (if available) can be used for other alternate function or as general purpose I/O. No external bus cycles are generated for on-chip XRAM access.

1.2.3.3 External Data Memory

Up to 1 Mbyte of synchronous or asynchronous external data memory is supported. External data memory extension, if supported by the XC800 derivative, is accomplished with either the 4-bit Current Bank pointer (CB) or the 4-bit XRAM Bank pointer (MX), selected by the MXM bit. The data is fetched from the 64-Kbyte block pointed to by CB or MX. Not all XC800 derivatives support access to external data memory.

1.2.4 Registers

All registers, except the program counter and the four general purpose register banks, reside in the SFR area.

The lower 32 locations of the IRAM are assigned to four banks with eight general purpose registers (GPRs) each. At any one time, only one of these banks can be enabled by two bits in the program status word (PSW): RS0 (PSW.3) and RS1 (PSW.4). This allows fast context switching, which is useful when entering subroutines or interrupt service routines. The eight general purpose registers of the selected register bank may be accessed by register addressing. For indirect addressing modes, the registers R0 and R1 are used as pointer or index register to address internal or external memory.

The Special Function Registers (SFRs) are mapped to the internal data space in the range 80_H to FF_H. The SFRs are accessible through direct addressing. The SFRs that are located at addresses with address bit 0-2 equal to 0 (addresses 80_H, 88_H, 90_H, ..., $F8_H$) are bitaddressable. Each bit of the bitaddressable SFRs has bit address

corresponding to the SFR byte address and its position within the SFR byte. For example, bit 7 of SFR at byte address 80_H has a bit address of 87_H. The bit addresses of the SFR bits span from 80_H to FF_H.

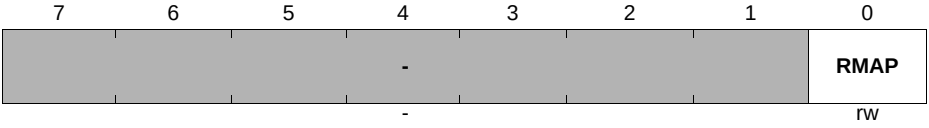
As the 128-SFR range is less than the total number of registers required, register extension mechanisms are implemented to increase the number of addressable SFRs. These mechanisms include:

- Mapping
- Paging

1.2.4.1 Special Function Register Extension by Mapping

SFR extension is performed at the system level by mapping. The SFR area is extended into two portions: the standard (non-mapped) SFR area and the mapped SFR area. Each portion supports the same address range 80H to FF_H, bringing the number of addressable SFRs to 256. To access SFRs in the mapped area, bit RMAP in SFR SYSCON0 must be set by software. The mapped SFR area provides the same addressing capabilities (direct addressing, bit addressing) as the standard SFR area. Bit RMAP must be cleared by software to access the SFRs in the standard area. The hardware does not automatically clear/set the bit.

SYSCON0
System Control Register 0 **Reset Value: XXXX XXX0$_B$**



Field	Bits	Type	Description
RMAP	0	rw	**Special Function Register Map Control** 0 The access to the standard SFR area is enabled. 1 The access to the mapped SFR area is enabled.

1.2.4.2 Special Function Register Extension by Paging

The number of SFRs may be further extended for some on-chip peripherals at the module level via a paging scheme. These peripherals have a built-in local SFR extension mechanism for increasing the number of addressable SFRs. The control is via bit field PAGE in the module page register MOD_PAGE. The bit field PAGE must be programmed before accessing the SFR of the target module. Each module may contain different number of pages and different number of SFRs per page, depending on the requirement. Besides setting the correct RMAP bit value to select the standard or mapped SFR area, the user must also ensure that a valid PAGE is selected to access the desired SFR. The paging mechanism is illustrated in **Figure 1-2**.

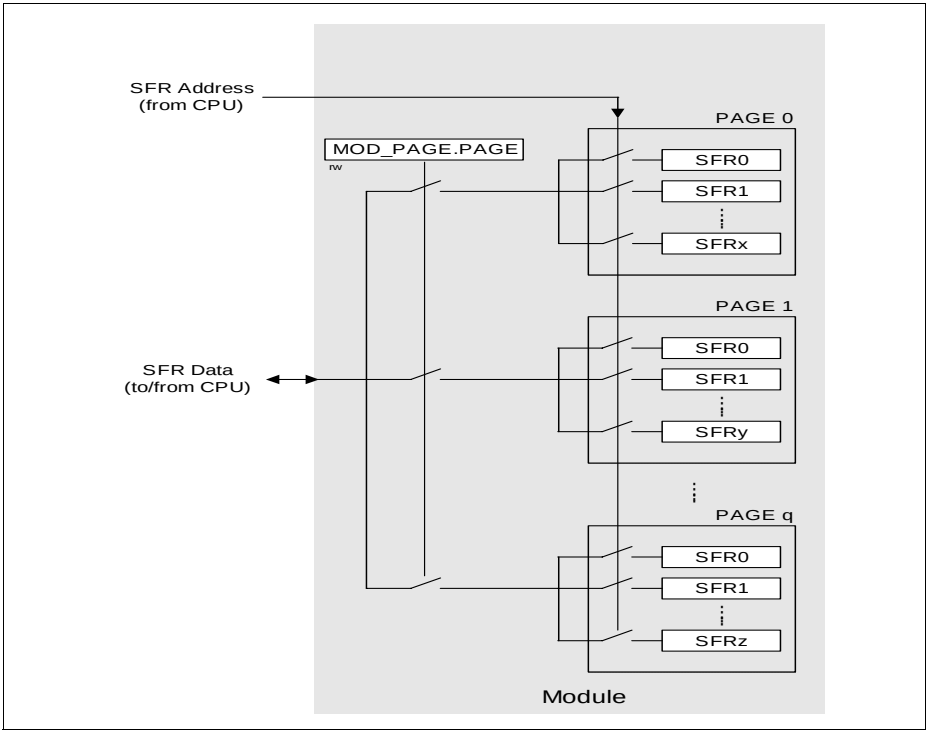


Figure 1-2 SFR Extension by Paging

If an interrupt routine is initiated between the page register access and the module register access, and the interrupt must access a register located in another page, the current page setting can be saved, the new one programmed and finally, the old page

setting restored. This is possible with the storage fields STx (x = 0 - 3) for the save and restore action of the current page setting, as illustrated in **Figure 1-3**. By indicating which storage register should be used in parallel with the new page value, a single write operation can:

- Save the contents of PAGE in STx before overwriting with the new value
  (this is done in the beginning of the interrupt routine to save the current page setting and program the new page number); or
- Overwrite the contents of PAGE with the contents of STx, ignoring the value written to the bit positions of PAGE
  (this is done at the end of the interrupt routine to restore the previous page setting before the interrupt occurred)

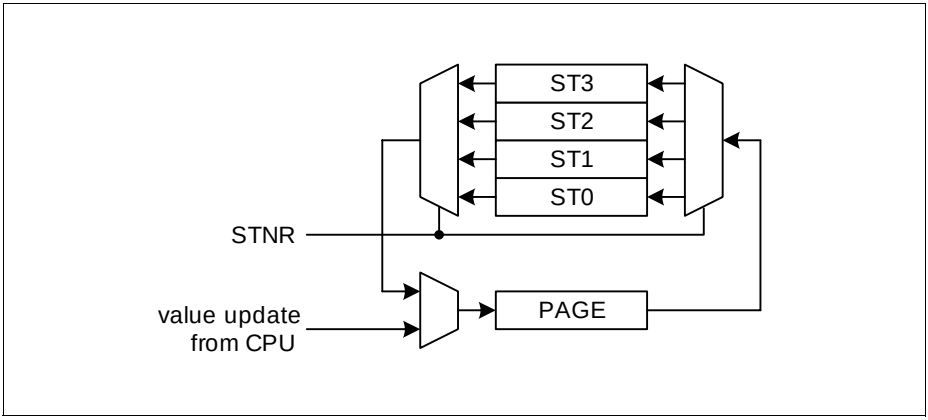


Figure 1-3 Storage Elements for Paging

With this mechanism, a certain number of interrupt routines (or other routines) can perform page changes without reading and storing the previously used page information. The use of only write operations makes the system simpler and faster. Consequently, this mechanism significantly improves the performance of short interrupt routines.

The page register has the following definition:

MOD_PAGE
Page Register for module MOD **Reset Value: 00_H**

7	6	5	4	3	2	1	0
OP		**STNR**		**0**	**PAGE**		
w		w		r	rw		

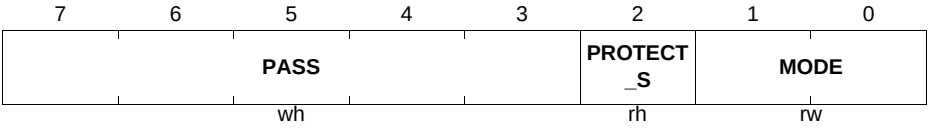
Field	Bits	Type	Description
PAGE	2:0	rw	**Page Bits** When written, the value indicates the new page. When read, the value indicates the currently active page.
STNR	5:4	w	**Storage Number** This number indicates which storage bit field is the target of the operation defined by bit field OP. If OP = 10_B, the contents of PAGE are saved in STx before being overwritten with the new value. If OP = 11_B, the contents of PAGE are overwritten by the contents of STx. The value written to the bit positions of PAGE is ignored. 00 ST0 is selected. 01 ST1 is selected. 10 ST2 is selected. 11 ST3 is selected.
OP	7:6	w	**Operation** 0X Manual page mode. The value of STNR is ignored and PAGE is directly written. 10 New page programming with automatic page saving. The value written to the bit field PAGE is stored. In parallel, the previous contents of PAGE are saved in the storage bit field STx indicated by STNR. 11 Automatic restore page action. The value written to the bit field PAGE is ignored and instead, PAGE is overwritten by the contents of the storage bit field STx indicated by STNR.
0	3	r	**Reserved** Returns 0 if read; should be written with 0.

1.3 Bit Protection Scheme

The bit protection scheme prevents direct software writing of selected bits (i.e., protected bits) by the PASSWD register. When the bit field MODE is 11_B, writing 10011_B to the bit field PASS opens access to writing of all protected bits and writing 10101_B to the bit field PASS closes access to writing of all protected bits. Note that access is opened for maximum 32 CCLKs if the “close access” password is not written. If “open access” password is written again before the end of 32 CCLK cycles, there will be a recount of 32 CCLK cycles.

The bits or bit fields that are protected may differ for the XC800 derivatives.

PASSWD
Password Register **Reset Value: 07_H**



Field	Bits	Type	Description
MODE	1:0	rw	**Bit-Protection Scheme Control bit** 00 Scheme Disabled 11 Scheme Enabled (default) Others: Scheme Enabled These two bits cannot be written directly. To change the value between 11_B and 00_B, the bit field PASS must be written with 11000_B, only then will the MODE[1:0] be registered.
PROTECT_S	2	rh	**Bit-Protection Signal Status bit** This bit shows the status of the protection. 0 Software is able to write to all protected bits. 1 Software is unable to write to any protected bits.
PASS	7:3	wh	**Password bits** The Bit-Protection Scheme recognizes only three patterns. 11000_BEnables writing of the bit field MODE. 10011_BOpens access to writing of all protected bits. 10101_BCloses access to writing of all protected bits.

2 CPU Architecture

Figure 2-1 depicts the typical architecture of an XC800 family microcontroller. It includes the main functional blocks and standard units. The units represented by dotted boxes may not be available, depending on the derivative; these include peripheral units and external memory bus. Memory sizes vary depending on the XC800 microcontroller derivative.

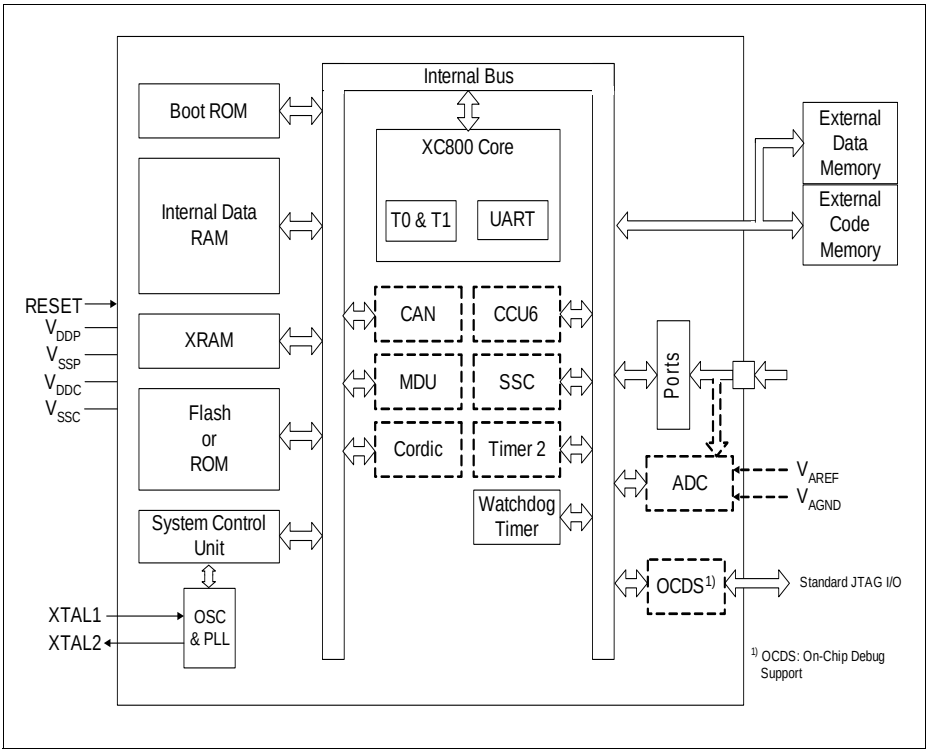


Figure 2-1 Typical Architecture of XC800 Family Microcontroller

The CPU functional blocks are shown in **Figure 2-2**. The CPU consists mainly of the instruction decoder, the arithmetic section, the program control section, the access control section, and the interrupt controller. The CPU also supports power saving modes.

The instruction decoder decodes each instruction and accordingly generates the internal signals required to control the functions of the individual units within the CPU. These internal signals have an effect on the source and destination of data transfers and control the ALU processing.

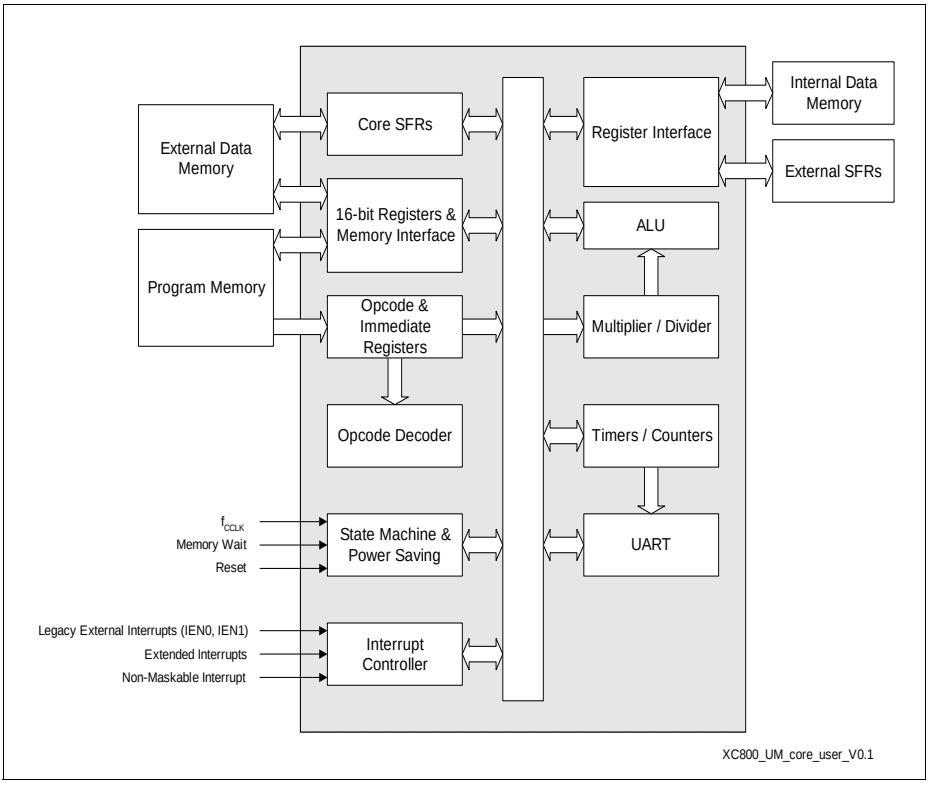


Figure 2-2 XC800 Core Block Diagram

The arithmetic section of the processor performs extensive data manipulation and consists of the arithmetic/logic unit (ALU), A register, B register, and PSW register. The ALU accepts 8-bit data words from one or two sources, and generates an 8-bit result under the control of the instruction decoder. The ALU performs both arithmetic and logic operations. Arithmetic operations include add, subtract, multiply, divide, increment, decrement, BCD-decimal-add-adjust, and compare. Logic operations include AND, OR, Exclusive OR, complement and rotate (right, left or swap nibble (left four)). A Boolean unit is also included for performing the bit operations such as set, clear, complement, jump-if-set, jump-if-not-set, jump-if-set-and-clear, and move to/from carry. The ALU can perform the bit operations of logical AND or logical OR between any addressable bit (or its complement) and the carry flag, and place the new result in the carry flag.

The program control section controls the sequence in which the instructions stored in program memory are executed. The 16-bit program counter (PC) holds the address of

the next instruction to be executed. The conditional branch logic enables internal and external events to the processor to cause a change in the program execution sequence.

The access control unit is responsible for the selection of the on-chip memory resources. The interrupt requests from the peripheral units are handled by the interrupt controller unit.

2.1 CPU Register Description

The CPU registers occupy direct Internal Data Memory space locations in the range 80_H to FF_H.

2.1.1 Stack Pointer (SP)

The SP register contains the Stack Pointer. The Stack Pointer is used to load the program counter into Internal Data Memory during LCALL and ACALL instructions, and to retrieve the program counter from memory during RET and RETI instructions. Data may also be saved on or retrieved from the stack using PUSH and POP instructions. Instructions that use the stack automatically pre-increment or post-decrement the stack pointer so that the stack pointer always points to the last byte written to the stack, i.e. the top of the stack. On reset, the Stack Pointer is reset to 07_H. This causes the stack to begin at a location = 08_H above register bank zero. The SP can be read or written under software control. The programmer must ensure that the location and size of the stack in internal data memory do not interfere with other application data.

2.1.2 Data Pointer (DPTR)

The Data Pointer (DPTR) is stored in registers DPL (Data Pointer Low byte) and DPH (Data Pointer High byte) to form 16-bit addresses for External Data Memory accesses (MOVX A,@DPTR and MOVX @DPTR,A), for program byte moves (MOVC A,@A+DPTR), and for indirect program jumps (JMP @A+DPTR).

Two true 16-bit operations are allowed on the Data Pointer: load immediate (MOV DPTR,#data) and increment (INC DPTR).

The CPU can support up to 8 data pointers. This is useful for high level language programming, which may require the storing of data in large external data memory portions. Selection of the active data pointer is done via the SFR EO (see **Section 2.1.6**). The number of data pointers available is specific to the XC800 derivative.

2.1.3 Accumulator (ACC)

This register is an operand for most ALU operations. ACC is the symbol for the accumulator register. The mnemonics for accumulator-specific instructions, however, refer to the accumulator simply as “A”.

2.1.4 B Register

The B register is used during multiply and divide operations to provide the second operand. For other instructions, it can be treated as another scratch pad register.

2.1.5 Program Status Word

The Program Status Word (PSW) contains several status bits that reflect the current state of the CPU.

2.1.5.1 Program Status Word Register

PSW
Program Status Word Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
CY	**AC**	**F0**	**RS1**	**RS0**	**OV**	**F1**	**P**
rwh	rwh	rw	rw	rw	rwh	rw	rh

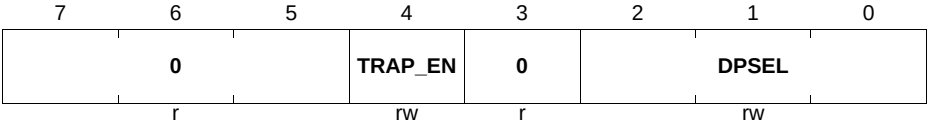
Field	Bits	Type	Description
P	0	rh	**Parity Flag** Set/cleared by hardware after each instruction to indicate an odd/even number of “one” bits in the accumulator, i.e., even parity.
F1	1	rw	**General Purpose Flag**
OV	2	rwh	**Overflow Flag** Used by arithmetic instructions
RS1, RS0	4:3	rw	**Register Bank Select** These bits are used to select one of the four register banks. 00Bank 0 selected, data address 00H-07H 01 Bank 1 selected, data address 08H-0FH 10 Bank 2 selected, data address 10H-17H 11 Bank 3 selected, data address 18H-1FH
F0	5	rw	**General Purpose Flag**
AC	6	rwh	**Auxiliary Carry Flag** Used by instructions that execute BCD operations
CY	7	rwh	**Carry Flag** Used by arithmetic instructions

2.1.6 Extended Operation (EO)

The EO register has two functions. One function is to select the active data pointer where the derivative has multiple data pointers. The other function is to select the instruction executed on opcode $A5_H$. The active instruction is either ‘TRAP’ or ‘MOVC @(DPTR++),A’. The latter instruction is supported for program memory that can be written.

2.1.6.1 Extended Operation Register

EO
Extended Operation Register **Reset Value: 00_H**



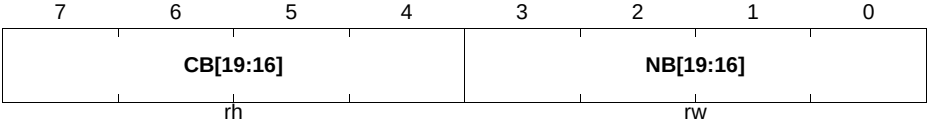
Field	Bits	Type	Description
DPSEL	2:0	rw	**Data Pointer Select** 000 DPTR0 selected 001 DPTR1 selected (if available) 010 DPTR2 selected (if available) 011 DPTR3 selected (if available) 100 DPTR4 selected (if available) 101 DPTR5 selected (if available) 110 DPTR6 selected (if available) 111 DPTR7 selected (if available)
TRAP_EN	4	rw	**TRAP Enable** 0 Select MOVC @(DPTR++),A 1 Select software TRAP instruction
0	3, 7:5	r	**Reserved** Returns 0 if read; should be written with 0.

2.1.7 Memory Extension

These registers support the memory extension feature, which may not be available on certain XC800 microcontroller derivatives.

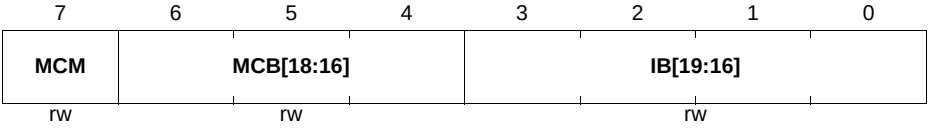
2.1.7.1 Memory Extension Registers

MEX1
Memory Extension Register 1 **Reset Value: 00_H**



Field	Bits	Type	Description
NB[19:16]	3:0	rw	**Next Bank Number**
CB[19:16]	7:4	rh	**Current Bank Number**

MEX2
Memory Extension Register 2 **Reset Value: 00_H**



Field	Bits	Type	Description
IB[19:16]	3:0	rw	**Interrupt Bank Number**
MCB[18:16]	6:4	rw	**Memory Constant Bank Number (with MEX3.7)**
MCM	7	rw	**Memory Constant Mode** 0 MOVC access data in the current bank 1 MOVC access data in the Memory Constant bank

MEX3
Memory Extension Register 3 **Reset Value: 00_H**

7	6	5	4	3	2	1	0
MCB19	**0**		**MX19**	**MXM**	**MX[18:16]**		
rw	r		rw	rw	rw		

Field	Bits	Type	Description
MX[19:16]	4, 2:0	rw	**XRAM Bank Number**
MXM	3	rw	**XRAM Bank Selector** 0 MOVX access data in the current bank 1 MOVX access data in the Memory XRAM bank
MCB19	7	rw	**Memory Constant Bank Number MSB**
0	6:5	r	**Reserved** Returns 0 if read; should be written with 0.

MEXSP
Memory Extension Stack Pointer Register **Reset Value: $7F_H$**

7	6	5	4	3	2	1	0
0	**MXSP**						
r	rwh						

Field	Bits	Type	Description
MXSP	6:0	rwh	**Memory Extension Stack Pointer** It provides for a stack depth of up to 128 bytes. It is pre-incremented by call instructions and post-decremented by return instructions.
0	7	r	**Reserved** Returns 0 if read; should be written with 0.

2.1.8 Power Control (PCON)

The XC800 CPU has two power saving modes: idle mode and power-down mode. In idle mode, the clock to the CPU is disabled while other peripherals may continue to run (possibly at lower frequency). In power-down mode, the clock to the entire CPU is stopped.

2.1.8.1 Power Control Register

PCON
Power Control Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
SMOD		**0**		**GF1**	**GF0**	**0**	**IDLE**
rw		r		rw	rw	r	rw

Field	Bits	Type	Description
IDLE	0	rw	**Idle Mode Enable** 0 Do not enter idle mode 1 Enter idle mode
GF0	2	rw	**General Purpose Flag Bit 0**
GF1	3	rw	**General Purpose Flag Bit 1**
SMOD	7	rw	**Double Baud Rate Enable** 0 Do not double the baud rate of serial interface in mode 2 1 Double baud rate of serial interface in mode 2
0	1, 6:4	r	**Reserved** Returns 0 if read; should be written with 0.

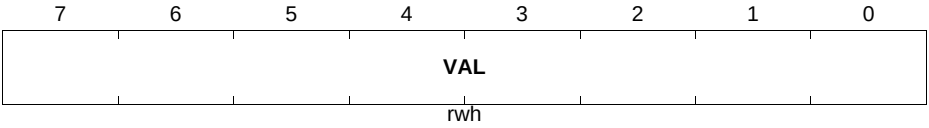
2.1.9 UART

The UART uses two SFRs, SCON and SBUF. SCON is the control register, while SBUF is the data register. The serial port control and status register is the SFR SCON. This register contains not only the mode selection bits, but also the 9th data bit for transmit and receive (TB8 and RB8), and the serial port interrupt bits (TI and RI).

SBUF is the receive and transmit buffer of the serial interface. Writing to SBUF loads the transmit register and initiates transmission. SBUF is read to access the received data from the receive register. The two paths are independent and supports full duplex operation.

2.1.9.1 UART Registers

SBUF
Serial Data Buffer **Reset Value: 00_H**



Field	Bits	Type	Description
VAL	7:0	rwh	**Serial Interface Buffer Register**

SCON
Serial Channel Control Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
SM0	**SM1**	**SM2**	**REN**	**TB8**	**RB8**	**TI**	**RI**
rw	rw	rw	rw	rw	rwh	rwh	rwh

Field	Bits	Type	Description
RI	0	rwh	**Receive Interrupt Flag** This is set by hardware at the end of the 8th bit in mode 0, or at the half point of the stop bit in modes 1, 2, and 3. Must be cleared by software.

Field	Bits	Type	Description
TI	1	rwh	**Transmit Interrupt Flag** This is set by hardware at the end of the 8th bit in mode 0, or at the beginning of the stop bit in modes 1, 2, and 3. Must be cleared by software.
RB8	2	rwh	**Serial Port Receiver Bit 9** In modes 2 and 3, this is the 9th data bit received. In mode 1, if SM2 = 0, this is the stop bit received. In mode 0, RB8 is not used.
TB8	3	rw	**Serial Port Transmitter Bit 9** In modes 2 and 3, this is the 9th data bit sent.
REN	4	rw	**Enable Receiver of Serial Port** 0 Serial reception is disabled 1 Serial reception is enabled
SM2	5	rw	**Enable Serial Port Multiprocessor Communication in Modes 2 and 3** In mode 2 or 3, if SM2 is set to 1, RI will not be activated if the received 9th data bit (RB8) is 0. In mode 1, if SM2 is set to 1, RI will not be activated if a valid stop bit (RB8) was not received. In mode 0, SM2 should be set to 0.
SM0, SM1	7:6	rw	**Serial Port Operating Mode Selection** 00 Mode 0: 8-bit shift register, fixed baud rate = $f_{PCLK}/2$ 01 Mode 1: 8-bit UART, variable baud rate 10 Mode 2: 9-bit UART, fixed baud rate ($f_{PCLK}/32$ or $f_{PCLK}/64$) 11 Mode 3: 9-bit UART, variable baud rate

2.1.10 Timer/Counter

Two 16-bit timers, Timer 0 and Timer 1, are available in the XC800 core.

The SFR TCON controls the running of the timers and generating of interrupts, while SFR TMOD sets the operating modes of the timers. The timer/counter values are stored in two pairs of 8-bit registers: TL0, TH0 and TL1, TH1 (reset value = 0000_H).

2.1.10.1 Timer/Counter Registers

TCON
Timer Control Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**
rwh	rw	rwh	rw	rw	rw	rw	rw

Field	Bits	Type	Description
TR0	4	rw	**Timer 0 Run Control** 0 Timer is halted 1 Timer runs
TF0	5	rwh	**Timer 0 Overflow Flag** Set by hardware when Timer 0 overflows. Cleared by hardware when the processor calls the interrupt service routine.
TR1	6	rw	**Timer 1 Run Control** 0 Timer is halted 1 Timer runs Also affects TH0 if Timer 0 operates in mode 3.
TF1	7	rwh	**Timer 1 Overflow Flag** Set by hardware when Timer 1 overflows. Cleared by hardware when the processor calls the interrupt service routine. *Note: TF1 is set by TH0 instead if Timer 0 operates in mode 3.*

TMOD
Timer Mode Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
GATE1	**CT1**	**T1M**		**GATE0**	**CT0**	**T0M**	
rw	rw	rw		rw	rw	rw	

Field	Bits	Type	Description
T0M, T1M	1:0, 5:4	rw	**Mode select bits** 00 13-bit timer: THx operates as 8-bit timer/counter, TLx is a 5-bit prescaler 01 16-bit timer: THx and TLx are cascaded 10 8-bit auto-reload timer: THx holds the reload value which is reloaded into TLx each time it overflow 11 Timer 0: Timer 0 is divided into two parts. TL0 is an 8-bit timer controlled by the standard Timer 0 control bits, and TH0 is the other 8-bit timer controlled by the standard Timer 1 control bits. Timer 1: TH1 and TL1 are held (Timer 1 is stopped).
CT0, CT1	2, 6	rw	**Counter Selection for Timer x** 0 Timer mode (input from internal system clock) 1 Counter mode (input from Tx input pin)
GATE0, GATE1	3, 7	rw	**Timer x Gating Control** 0 Timer x will only run if TCON.TRx = 1 (software control) 1 Timer x will only run if NINTx pin = 0 (hardware control) and TCON.TRx is set

2.1.11 Interrupt Registers

Each interrupt node can be individually enabled or disabled by setting or clearing the corresponding bit in the bitaddressable interrupt enable registers IEN0 and IEN1. Register IEN0 also contains the global interrupt masking bit (EA), which can be cleared to effectively disable all interrupts at once.

The Non-Maskable Interrupt (NMI) node is always enabled.

After reset, the enable bits of IEN0 and IEN1 are cleared to 0. This implies that all interrupt nodes are disabled by default.

IEN0
Interrupt Enable Register 0 **Reset Value: 00_H**

7	6	5	4	3	2	1	0
EA	**0**	**ET2**	**ES**	**ET1**	**EX1**	**ET0**	**EX0**
rw	r	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
EX0	0	rw	**Enable External Interrupt 0** 0 External Interrupt 0 is disabled 1 External Interrupt 0 is enabled
ET0	1	rw	**Enable Timer 0 Overflow Interrupt** 0 Timer 0 Overflow interrupt is disabled 1 Timer 0 Overflow interrupt is enabled
EX1	2	rw	**Enable External Interrupt 1** 0 External interrupt 1 is disabled 1 External interrupt 1 is enabled
ET1	3	rw	**Enable Timer 1 Overflow Interrupt** 0 Timer 1 Overflow interrupt is disabled 1 Timer 1 Overflow interrupt is enabled
ES	4	rw	**Enable Serial Port Interrupt** 0 Serial Port interrupt is disabled 1 Serial Port interrupt is enabled
ET2	5	rw	**Enable Timer 2 Interrupt** 0 Timer 2 interrupt is disabled 1 Timer 2 interrupt is enabled

Field	Bits	Type	Description
EA	7	rw	**Global Interrupt Mask** 0 All interrupt requests (except NMI) are ignored. 1 Each interrupt node is individually enabled or disabled by setting or clearing its enable bit.
0	6	r	**Reserved** Returns 0 if read; should be written with 0.

The interrupt enable bits of IEN1 are used to enable or disable the corresponding interrupts. The assignment of these bits depends on which peripheral set is available on the derivative.

IEN1
Interrupt Enable Register 1 **Reset Value: 00_H**

7	6	5	4	3	2	1	0
EI13	**EI12**	**EI11**	**EI10**	**EI9**	**EI8**	**EI7**	**EI6**
rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
EIx **(x = 13:6)**	7:0	rw	**Extended Interrupt Node Enable** 0 XINTRx is disabled 1 XINTRx is enabled

Each interrupt source can be individually programmed to one of the four priority levels available via the corresponding IP, IPH or IP1, IPH1 registers. IP and IP1 are bitaddressable, but not IPH and IPH1.

IP(H)
Interrupt Priority (High) Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
0		**PT2(H)**	**PS(H)**	**PT1(H)**	**PX1(H)**	**PT0(H)**	**PX0(H)**
r		rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
PX0, PX0H	0	rw	**Priority Level for External Interrupt 0**
PT0, PT0H	1	rw	**Priority Level for Timer 0 Overflow Interrupt**
PX1, PX1H	2	rw	**Priority Level for External Interrupt 1**
PT1, PT1H	3	rw	**Priority Level for Timer 1 Overflow Interrupt**
PS, PSH	4	rw	**Priority Level for Serial Port Interrupt**
PT2, PT2H	5	rw	**Priority Level for Interrupt Node XINTR5 (Timer 2)**
0	7:6	r	**Reserved** Returns 0 if read; should be written with 0.

IP(H)1
Interrupt Priority 1 (High) Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
PI13(H)	**PI12(H)**	**PI11(H)**	**PI10(H)**	**PI9(H)**	**PI8(H)**	**PI7(H)**	**PI6(H)**
rw	rw	rw	rw	rw	rw	rw	rw

Field	Bits	Type	Description
PI6, PI6H	0	rw	**Priority Level for Interrupt Node XINTR6**
PI7, PI7H	1	rw	**Priority Level for Interrupt Node XINTR7**
PI8, PI8H	2	rw	**Priority Level for Interrupt Node XINTR8**
PI9, PI9H	3	rw	**Priority Level for Interrupt Node XINTR9**
PI10, PI10H	4	rw	**Priority Level for Interrupt Node XINTR10**

Field	Bits	Type	Description
PI11, PI11H	5	rw	**Priority Level for Interrupt Node XINTR11**
PI12, PI12H	6	rw	**Priority Level for Interrupt Node XINTR12**
PI13, PI13H	7	rw	**Priority Level for Interrupt Node XINTR13**

Four bits are available in TCON to control and flag the external interrupts.

TCON
Timer Control Register **Reset Value: 00_H**

7	6	5	4	3	2	1	0
TF1	**TR1**	**TF0**	**TR0**	**IE1**	**IT1**	**IE0**	**IT0**
rwh	rw	rwh	rw	rwh	rw	rwh	rw

Field	Bits	Type	Description
IT0	0	rw	**External Interrupt 0 Level/Edge Trigger Control** 0 Low-level triggered external interrupt 0 is selected 1 Falling edge triggered external interrupt 0 is selected
IE0	1	rwh	**External Interrupt 0 Flag** Set by hardware when external interrupt 0 event is detected. Cleared by hardware when the processor vectors to interrupt routine. Can also be cleared by software.
IT1	2	rw	**External Interrupt 1 Level/Edge Trigger Control** 0 Low-level triggered external interrupt 1 is selected 1 Falling edge triggered external interrupt 1 is selected

Field	Bits	Type	Description
IE1	3	rwh	**External Interrupt 1 Flag** Set by hardware when external interrupt 1 event is detected. Cleared by hardware when the processor vectors to interrupt routine. Can also be cleared by software.

2.2 On-Chip Debug Support System

The XC800 microcontrollers have an On-Chip Debug Support (OCDS) unit that provides basic functionality to support software development and debugging of the XC800-based systems. The debug functionality is usually enabled after the device has been started in OCDS mode.

The debug concept is based on the interaction between the OCDS hardware and a dedicated software (Monitor program) which is usually located in the Boot ROM. Standard interface such as the JTAG or UART is used to communicate with an external host (a debugger).

An overview of the debug system control, access and interfaces is shown in **Figure 2-3**.

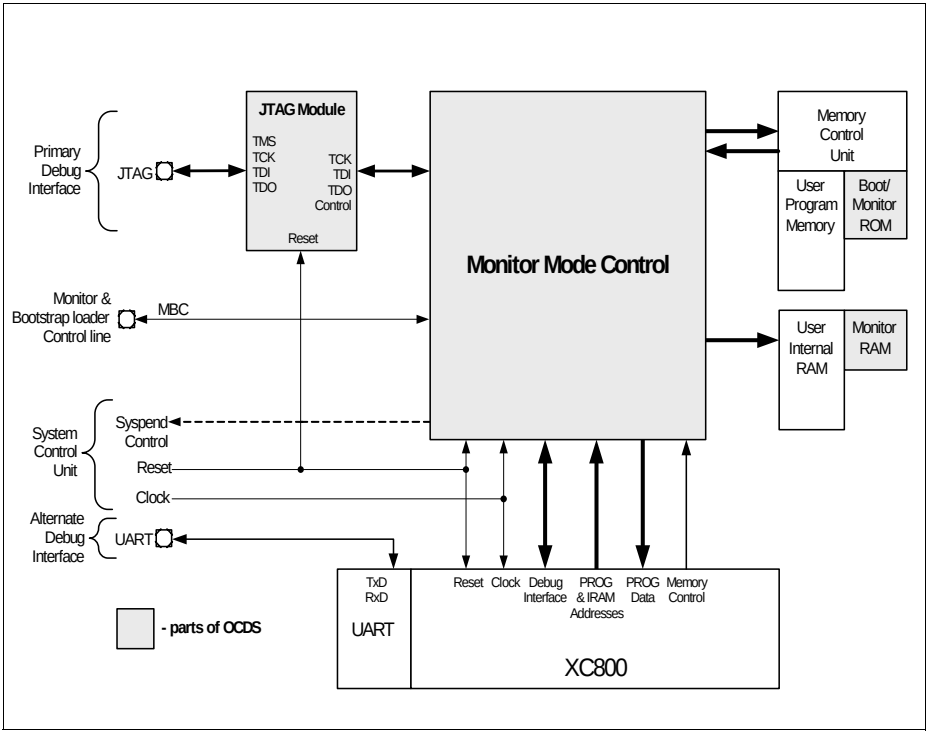


Figure 2-3 XC800 OCDS Block Diagram

- A Monitor Mode Control (MMC) block at the center of the OCDS system brings together control signals and supports the overall functionality
- MMC communicates with the XC800 core primarily via the Debug Interface, and also receives reset and clock signals

- After processing memory address and control signals from the core, MMC provides proper access to the dedicated memories: a Monitor ROM (holding the code) and a Monitor RAM (for work-data and Monitor-stack)
- Two interfaces can be used to access the OCDS system:
  – JTAG as a primary channel; dedicated exclusively to test and debug activities and is not normally used in an application
  – UART as an alternative channel; it has the advantage of needing fewer pins
- A dedicated pin is used as external configuration and control for both the debugging and bootstrap-loading.

The on-chip debug concept is based on the generation and detection of debug events and the corresponding debug actions:

- Debug events:
  – Hardware Breakpoints
  – Software Breakpoints
  – External Breaks
- Debug event actions (non-exclusive):
  – Activate the Monitor Program
  – Activate the MBC pin
- Other debug features:
  – Single step execution
  – Return to user program

2.3 CPU Interrupt System

This section provides an overview of the XC800 interrupt system.

2.3.1 Interrupt Source and Vector Address

Each interrupt event source has an associated interrupt vector address for the interrupt node it belongs to. This vector is accessed to service the corresponding interrupt node request. The interrupt service of each interrupt node can be individually enabled or disabled via an enable bit. The assignment of the XC800 interrupt sources to the interrupt vector address is summarized in **Table 2-1**. The extended interrupts are generally assigned to on-chip peripherals, which vary depending on the XC800 derivative.

Table 2-1 Interrupt Vector Address

Interrupt Node	Vector Address	Interrupt Source Assignment
NMI	0073_H	Non-maskable Interrupt
XINTR0	0003_H	External Interrupt 0
XINTR1	$000B_H$	Timer 0
XINTR2	0013_H	External Interrupt 1
XINTR3	$001B_H$	Timer 1
XINTR4	0023_H	UART
XINTR5	$002B_H$	Extended Interrupt 5 (Timer 2)
XINTR6	0033_H	Extended Interrupt 6
XINTR7	$003B_H$	Extended Interrupt 7
XINTR8	0043_H	Extended Interrupt 8
XINTR9	$004B_H$	Extended Interrupt 9
XINTR10	0053_H	Extended Interrupt 10
XINTR11	$005B_H$	Extended Interrupt 11
XINTR12	0063_H	Extended Interrupt 12
XINTR13	$006B_H$	Extended Interrupt 13

2.3.2 Interrupt Handling

The interrupt request signals are sampled at phase 2 in each machine cycle. The sampled requests are then polled during the following machine cycle. If one interrupt node request was active at phase 2 of the preceding cycle, the polling cycle will find it

and the interrupt system will generate a LCALL to the node's service routine, provided this hardware-generated LCALL is not blocked by any of the following conditions:

1. An interrupt of equal or higher priority is already in progress.
2. The current (polling) cycle is not in the final cycle of the instruction in progress.
3. The instruction in progress is RETI or any write access to registers IEN0/IEN1 or IP/IPH or IP1/IPH1.

Any of these three conditions will block the generation of the LCALL to the interrupt service routine. Condition 2 ensures that the instruction in progress is completed before vectoring to any service routine. Condition 3 ensures that if the instruction in progress is RETI or any write access to registers IEN0/IEN1 or IP/IPH or IP1/IPH1, then at least one more instruction will be executed before any interrupt is vectored to; this delay guarantees that changes of the interrupt status can be observed by the CPU.

The polling cycle is repeated with each machine cycle, and the values polled are the values that were present at phase 2 of the previous machine cycle. Note that if any interrupt flag is active but its node interrupt request was not responded to for one of the conditions already mentioned, and if the flag is no longer active at a later time when servicing the interrupt node, the corresponding interrupt source will not be serviced. In other words, the fact that the interrupt flag was once active but not serviced is not remembered. Every polling cycle interrogates only the pending interrupt requests.

The processor acknowledges an interrupt request by executing a hardware generated LCALL to the corresponding service routine. In some cases, hardware also clears the flag that generated the interrupt, while in other cases, the flag must be cleared by the user's software. The hardware-generated LCALL pushes the contents of the Program Counter (PC) onto the stack (but it does not save the PSW) and reloads the PC with an address that depends on the interrupt node being vectored to.

Program execution returns to the next instruction after calling the interrupt when the RETI instruction is encountered. The RETI instruction informs the processor that the interrupt routine is no longer in progress, then pops the two top bytes from the stack and reloads the PC. Execution of the interrupted program continues from the point where it was stopped. Note that the RETI instruction is important because it informs the processor that the program has left the current interrupt priority level. A simple RET instruction would also have returned execution to the interrupted program, but it would have left the interrupt control system on the assumption that an interrupt was still in progress. In this case, no interrupt of the same or lower priority level would be acknowledged.

2.3.3 Interrupt Response Time

Due to an interrupt event of (the various sources of) an interrupt node, its corresponding request signal will be sampled active at phase 2 in every machine cycle. The value is not polled by the circuitry until the next machine cycle. If the request is active and conditions are right for it to be acknowledged, a hardware subroutine call to the requested service

routine will be the next instruction to be executed. The call itself takes two machine cycles. Thus, a minimum of three complete machine cycles will elapse from activation of the interrupt request to the beginning of execution of the first instruction of the service routine as shown in **Figure 2-4**.

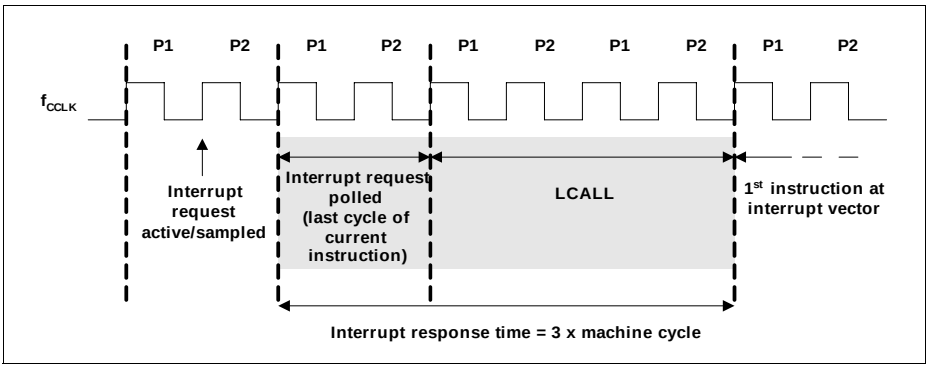


Figure 2-4 Minimum Interrupt Response Time

A longer response time would be obtained if the request is blocked by one of the three previously listed conditions:

1. If an interrupt of equal or higher priority is already in progress, the additional wait time will depend on the nature of the other interrupt's service routine.
2. If the instruction in progress is not in its final cycle, the additional wait time cannot be more than three machine cycles since the longest instructions (MUL and DIV) are only four machine cycles long. See **Figure 2-5**.
3. If the instruction in progress is RETI or a write access to registers IEN0, IEN1 or IP(H), IP1(H), the additional wait time cannot be more than five cycles (a maximum of one more machine cycle to complete the instruction in progress, plus four machine cycles to complete the next instruction, if the instruction is MUL or DIV). See **Figure 2-6**.

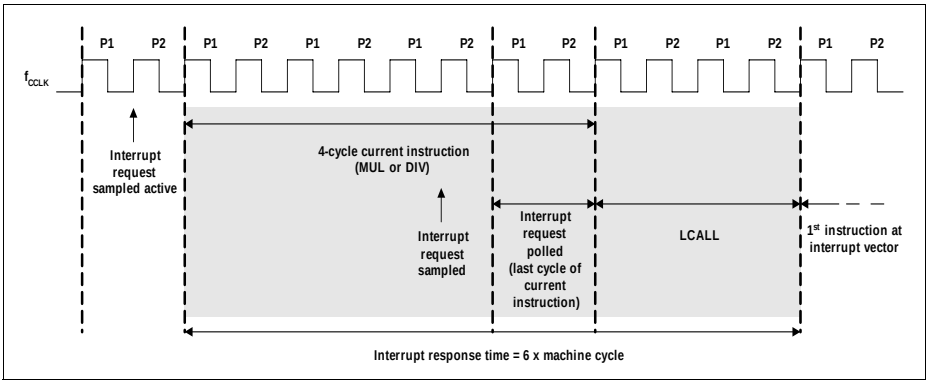


Figure 2-5 Interrupt Response Time for Condition 2

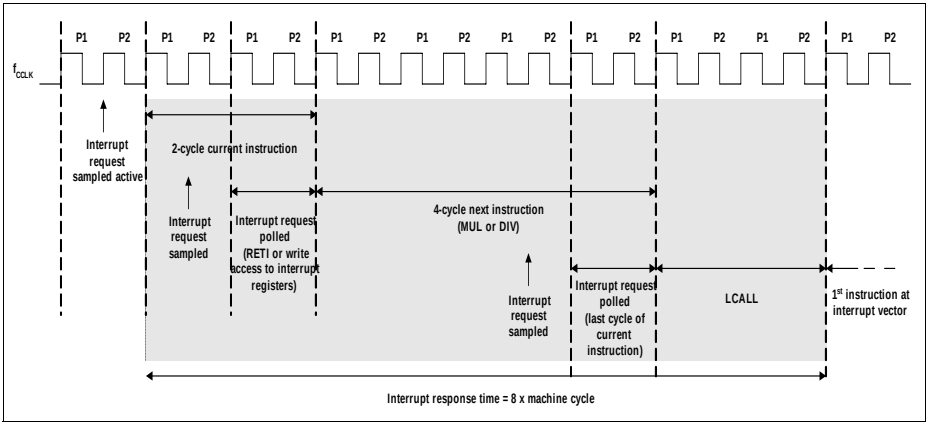


Figure 2-6 Interrupt Response Time for Condition 3

Thus in a single interrupt system, the response time is always more than three machine cycles and less than nine machine cycles (wait states are not considered). When considering wait states, the interrupt response time will be extended depending on the user instructions (also the hardware generated LCALL) being executed during the interrupt response time (shaded region in **Figure 2-5** and **Figure 2-6**).

2.3.4 Interrupt Node Priority

A low-priority interrupt can be interrupted by a high-priority interrupt, but not by another interrupt of the same or lower priority. An interrupt of the highest priority cannot be interrupted by any other interrupt source.

If two or more requests of different priority levels are received simultaneously, the request of the highest priority is serviced first.

The respective bit fields of the interrupt priority registers together select one of the four levels of priority shown in **Table 2-2**.

Table 2-2 Interrupt Priority Level Selection

IPH.x / IPH1.x	IP.x / IP1.x	Priority Level
0	0	Level 0 (lowest)
0	1	Level 1
1	0	Level 2
1	1	Level 3 (highest)

Note: The NMI always takes precedence over all other interrupts.

If requests of the same priority are received simultaneously, an internal polling sequence determines which request is serviced first. Thus, within each priority level there is a second priority structure determined by the polling sequence as shown in **Table 2-3**. The extended interrupts that are applicable, vary depending on the XC800 derivative.

Table 2-3 Priority Structure within Interrupt Level

Source	Level
Non-maskable Interrupt (NMI)	(highest)
External Interrupt 0	1
Timer 0 Interrupt	2
External Interrupt 1	3
Timer 1 Interrupt	4
UART Interrupt	5
Extended Interrupt 5 (Timer 2)	6
Extended Interrupt 6	7
Extended Interrupt 7	8
Extended Interrupt 8	9
Extended Interrupt 9	10
Extended Interrupt 10	11
Extended Interrupt 11	12
Extended Interrupt 12	13
Extended Interrupt 13	14

3 CPU Timing

The following sections describe the CPU instruction timing, and external memory access timing.

3.1 Instruction Timing

A CPU machine cycle comprises two input clock periods, referred to as Phase 1 (P1) and Phase 2 (P2), that correspond to two different CPU states. A CPU state within an instruction is referenced by the machine cycle and state number, e.g., C2P1 means the first clock period within machine cycle 2. Memory access takes place during one or both phases of the machine cycle. SFR writes occur only at the end of P2. Instructions are 1, 2, or 3 bytes long and can take 1, 2 or 4 machine cycles to execute. Registers are generally updated and the next opcode pre-fetched at the end of P2 of the last machine cycle for the current instruction.

The XC800 core supports access to slow (internal) memory by using wait state(s). Each wait state lasts one machine cycle. For example, in case of a memory requiring one wait state, the access time is increased by one machine cycle after every byte of opcode/operand fetched.

Figure 3-1 shows the fetch/execute timing related to the internal states and phases. Execution of an instruction occurs at C1P1. For a 2-byte instruction, the second reading starts at C1P1.

Figure 3-1 (a) shows two timing diagrams for a 1-byte, 1-cycle (1 × machine cycle) instruction. The first diagram shows the instruction being executed within one machine cycle since the opcode (C1P2) is fetched from a memory without wait state. The second diagram shows the corresponding states of the same instruction being executed over two machine cycles (instruction time extended), with one wait state inserted for opcode fetching from a slower memory.

Figure 3-1 (b) shows two timing diagrams for a 2-byte, 1-cycle (1 × machine cycle) instruction. The first diagram shows the instruction being executed within one machine cycle since the second byte (C1P1) and the opcode (C1P2) are fetched from a memory without wait state. The second diagram shows the corresponding states of the same instruction being executed over three machine cycles (instruction time extended), with one wait state inserted for each access to the slow memory (two wait states inserted in total).

Figure 3-1 (c) shows two timing diagrams of a 1-byte, 2-cycle (2 × machine cycle) instruction. The first diagram shows the instruction being executed over two machine cycles with the opcode (C2P2) fetched from a memory without wait state. The second diagram shows the corresponding states of the same instruction being executed over three machine cycles (instruction time extended), with one wait state inserted for opcode fetching from the slow memory.

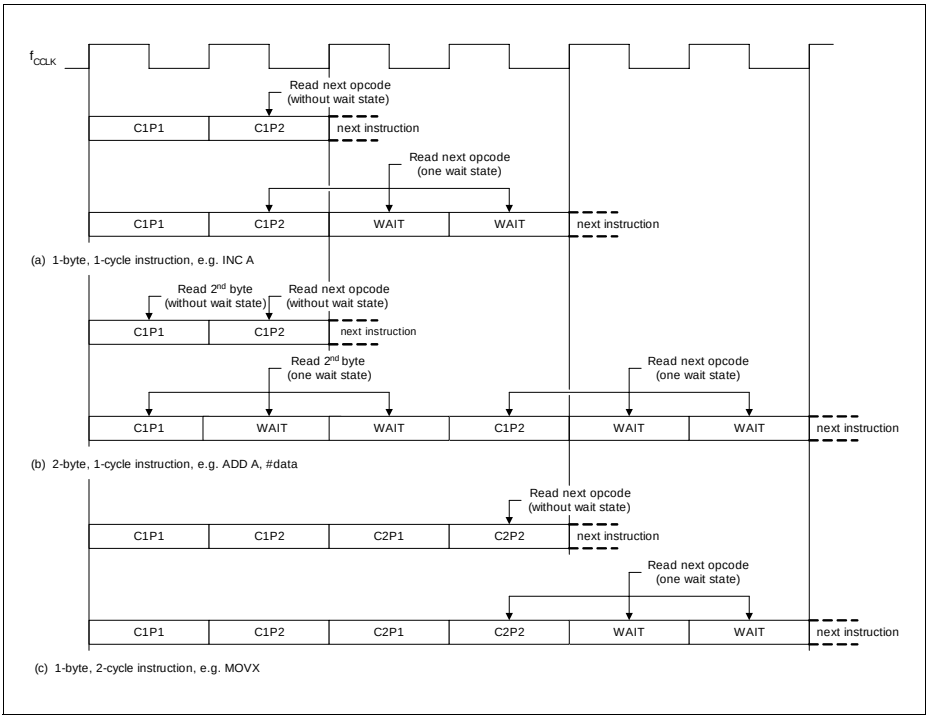


Figure 3-1 CPU Instruction Timing

The time taken for each instruction includes:

- Decoding/executing the fetched opcode
- Fetching the operand/s (for instructions > 1 byte)
- Fetching the first byte (opcode) of the next instruction (due to CPU pipeline)

Note: The XC800 CPU fetches the opcode of the next instruction while executing the current instruction.

Even with one wait state inserted for each byte of operand/opcode fetched, the XC800 CPU executes instructions faster than the standard 8051 processor by a factor of between two (e.g., 2-byte, 1-cycle instructions) to six (e.g., 1-byte, 4-cycle instructions).

3.2 Accessing External Memory

There are two types of external memory accesses: accesses to external program memory and accesses to external data memory. Accesses to external program memory use the signal PSEN as the read strobe, while accesses to external data memory use the RD or WR to read or write the memory. Depending on the derivative that supports external memory accessing, address (Ax) and data (D[7:0]) lines may be multiplexed as alternate function of the available ports.

3.2.1 Accessing External Program Memory

External program memory is generally accessed under two conditions:

- Whenever EA is active (low), or
- Whenever EA is inactive (high) and the program counter (PC) contains an address outside the range of the internal code memories.

Fetches from external program memory use address bus width of 16 bits, and up to 20 bits if memory extension is supported (uppermost 4 bits for bank selection). These address pins are the alternate function of the corresponding ports, and when the CPU is executing from external program memory, should never be used for other alternate port functions.

Figure 3-2 shows the timing of the external program memory access cycle.

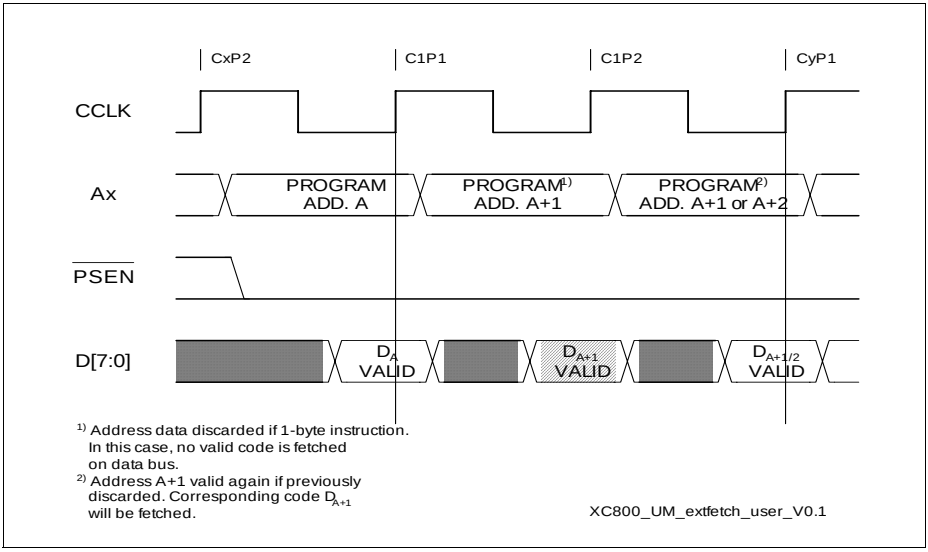


Figure 3-2 External Program Memory Fetches

3.2.2 Accessing External Data Memory

External data memory may generally be accessed only if the corresponding address is not occupied by internal program memory in the code space.

The access to external data memory uses address bits 17 up to 20 (if available) for bank selection. Within each bank of external data memory, access can be via either a 16-bit address (MOVX @DPTR) or an 8-bit address (MOVX @R_i). If an 8-bit addressing mode is used, any output port pins can be used to output high-order address bits. Alternatively, the contents of the corresponding port SFR of the high-byte address pins may be initialised to hold the high-byte address on the pins during the external memory access. These pins are therefore used to page the current active bank (selected by MEX1.CBx or MEX3.MXx) of external memory by defining the upper address byte.

In a read cycle, the incoming byte is accepted just before the read strobe $\overline{RD}$ is deactivated.

Figure 3-3 shows the timing of the external data memory read cycle. This timing assumes only data access on the external interface.

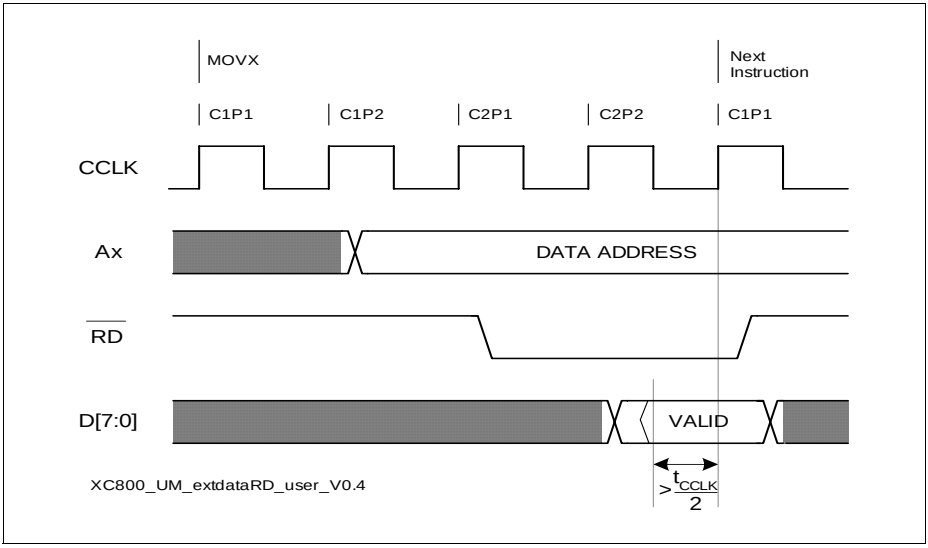


Figure 3-3 External Data Memory Read Cycle

In a write cycle, the data byte to be written appears at the pins before $\overline{WR}$ is activated, and remains there after $\overline{WR}$ is deactivated.

Figure 3-4 shows the timing of the external data memory write cycle. This timing assumes multiplexed program fetch and data access on the external interface.

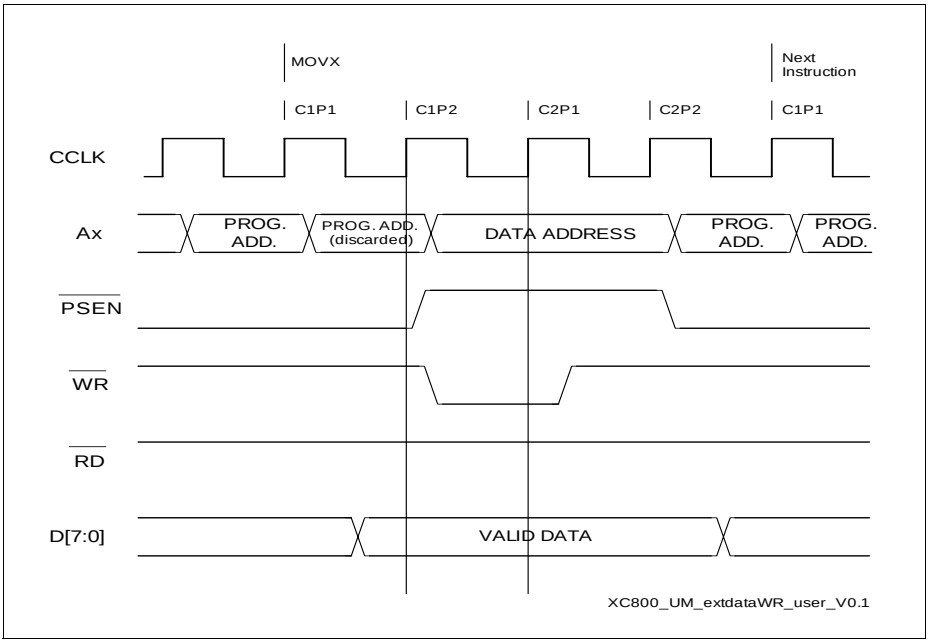


Figure 3-4 External Data Memory Write Cycle

4 Instruction Set

The XC800 8-bit microcontroller family instruction set includes the 111 instructions of the standard 8051, plus 2 additional instructions, ‘MOVC @(DPTR++),A’ and ‘TRAP’, which are multiplexed and selected through the Special Function Register (SFR) EO. Out of the 113 instructions, 51 are single-byte, 46 are two-byte and 16 are three-byte.

The instruction opcode format consists of a function mnemonic that is usually followed by a “destination, source” operand field. This field specifies the data type and addressing method(s) to be used.

4.1 Addressing Modes

The XC800 uses five general addressing modes:

- Register,
- Direct,
- Immediate,
- Register indirect,
- Base register plus index-register indirect,

Including bit addressing for bitaddressable locations.

Table 4-1 summarizes the memory space(s) that may be accessed by each addressing mode.

Table 4-1 Addressing Mode and Associated Memory Space

Addressing Mode	Associated Memory Space
Register addressing	R0 through R7 of selected register bank, ACC, B, CY (Bit), DPTR
Direct addressing	Lower 128 bytes of internal RAM, special function registers
Immediate addressing	Program memory
Register indirect addressing	Internal RAM (@R1, @R0, SP), external data memory (@R1, @R0, @DPTR)
Base register plus index register addressing	Program memory (@A + DPTR, @A + PC)
Bit addressing	Bitaddressable SFRs, 128 bits in the bitaddressable area within the lower internal data RAM

4.1.1 Register Addressing

Register addressing accesses the eight working registers (R0 - R7) of the selected register bank. The least significant bit of the instruction opcode indicates which register is to be used. Some instructions only operate on specific registers such as ACC (A), B, DPTR, or on the bit CY (the Boolean accumulator).

4.1.2 Direct Addressing

Direct addressing is the only method of accessing the SFRs. The lower 128 bytes of internal RAM are also directly addressable. In direct addressing, the operand is specified by an 8-bit address field.

4.1.3 Immediate Addressing

Immediate addressing allows constants to be part of the instruction in program memory. These instructions are 2 or more bytes long.

4.1.4 Register Indirect Addressing

Register indirect addressing uses the contents of either R0 or R1 (in the selected register bank) as a pointer to locations in a 256-byte block: the 256 bytes of internal RAM or the lower 256 bytes of external data memory. Note that the SFRs are not accessible by this method. The upper half of the internal RAM can be accessed by indirect addressing only. Access to the full 64 Kbytes of the active bank of the external data memory address space is accomplished by using the 16-bit data pointer.

4.1.5 Base Register plus Index Register Addressing

Base register plus index register addressing allows a byte to be accessed from program memory via an indirect move from the location whose address is the sum of a base register (DPTR or PC) and index register ACC. This mode facilitates look-up table accesses.

4.1.6 Bit Addressing

Direct bit addressing is supported for bitaddressable locations: bits of bitaddressable SFRs and the 128 bits in the bitaddressable area within the lower internal data RAM.

4.2 Introduction to the Instruction Set

The instruction set is divided into six basic functional groups:

- Arithmetic
- Logic
- Data transfer
- Control transfer (branching)
- Boolean
- Miscellaneous

4.2.1 Arithmetic Instructions

The XC800 microcontrollers have four basic mathematical operations.

- Addition: ADD, ADDC, INC, DA
- Subtraction: SUBB, DEC
- Multiplication: MUL
- Division: DIV

Only 8-bit operations using unsigned arithmetic are supported directly. The overflow flag, however, permits the addition and subtraction operations to handle both unsigned and signed binary integers. Arithmetic can also be performed directly on packed BCD representations.

4.2.2 Logic Instructions

The XC800 microcontrollers perform basic logic operations on both bit and byte operands: ANL, ORL, SRL, CLR, SETB, CPL, RL, RLC, RR, RRC, SWAP.

4.2.3 Data Transfer Instructions

Data transfer operations are divided into three classes:

- General-purpose
- Accumulator-specific
- Address-object

None of these operations affects the PSW flag settings except a POP or MOV directly to the PSW.

4.2.4 Control Transfer Instructions

All control transfer operations, some upon a specific condition, cause the program execution to continue to a non-sequential location in program memory. There are three classes of control transfer operations:

- Unconditional jumps
- Conditional jumps

- Subroutine/interrupt calls and returns

Unconditional jumps transfer control from the current value of the program counter to the target address. These instructions are: AJMP, LJMP, SJMP and JMP @A + DPTR.

Conditional jumps perform a jump contingent upon a specific condition. The destination will be within a 256-byte range centered about the starting address of the next instruction (– 128 to + 127): JZ, JNZ, JC, JNC, JB, JNB, JBC, CJNE, DJNZ.

There are only 2 types of subroutine call: ACALL and LCALL. Interrupt call is controlled by hardware. Return instructions are RET and RETI. RETI is used for return from interrupt, which restores interrupt priority to that of the current priority level.

4.2.5 Boolean Instructions

The bitaddressable registers in both direct and SFR space may be manipulated using Boolean instructions. The bit manipulation instructions allow:

- Set bit
- Clear bit
- Complement bit
- Jump if bit is set
- Jump if bit is not set
- Jump if bit is set and clear bit
- Move bit from / to carry

Addressable bits, or their complements, may be logically AND-ed or OR-ed with the contents of the carry flag. The result is stored in the carry bit.

4.2.6 Miscellaneous Instructions

These instructions are:

- NOP: no operation
- TRAP: software break command

4.3 Instructions

The XC800 instructions can essentially be condensed to 55 basic operations. These operations are described in detail in the following sections.

4.3.1 Affected Flags

Some instructions affect one or more of the PSW flags, as generally shown in **Table 4-2**.

Table 4-2 PSW Flag Modification (CY,OV,AC)

Instruction	Flag			Instruction	Flag		
	CY	OV	AC		CY	OV	AC
ADD	X	X	X	SETB C	1		
ADDC	X	X	X	CLR C	0		
SUBB	X	X	X	CPL C	X		
MUL	0	X		ANL C,bit	X		
DIV	0	X		ANL C,/bit	X		
DA	X			ORL C,bit	X		
RRC	X			ORL C,/bit	X		
RLC	X			MOV C,bit	X		
CJNE	X						

In the above table, a 0 means the flag is always cleared, a 1 means the flag is always set and an “X” means that the state of the flag depends on the result of the operation. A blank cell indicates that the flag is unaffected by the instruction.

Only the carry, auxiliary carry, and overflow flags are discussed above. The parity bit is always computed from the actual content of the accumulator.

- CY is set if the operation causes a carry to or a borrow from the resulting high-order bit; otherwise CY is cleared.
- AC is set if the operation results in a carry from the low-order four bits of the result (during addition), or a borrow from the high-order bits to the low-order bits (during subtraction); otherwise AC is cleared.
- OV is set if the operation results in a carry to the high-order bit of the result but not a carry from the bit, or vice versa; otherwise OV is cleared. OV is used in twos complement arithmetic, because it is set when the signal result cannot be represented in 8 bits.
- P is set if the modulo-2 sum of the eight bits in the accumulator is 1 (odd parity); otherwise P is cleared (even parity). When a value is written to the PSW register, the P bit remains unchanged, as it always reflects the parity of A.

Instructions that directly alter addressed registers could affect the other status flags if the instruction is applied to the PSW. Status flags can also be modified by bit manipulation.

4.3.2 Instruction Table

Table 4-3 lists all the instructions supported by XC800. Instructions are 1, 2 or 3 bytes long as indicated in the 'Bytes' column. Each instruction takes 1, 2 or 4 machine cycles to execute (with no wait state). One machine cycle comprises 2 CCLK clock cycles.

Table 4-3 Instruction Table

Mnemonic	Description	Hex Code	Bytes	Cycles
	ARITHMETIC			
ADD A,Rn	Add register to A	28-2F	1	1
ADD A,direct	Add direct byte to A	25	2	1
ADD A,@Ri	Add indirect memory to A	26-27	1	1
ADD A,#data	Add immediate to A	24	2	1
ADDC A,Rn	Add register to A with carry	38-3F	1	1
ADDC A,direct	Add direct byte to A with carry	35	2	1
ADDC A,@Ri	Add indirect memory to A with carry	36-37	1	1
ADDC A,#data	Add immediate to A with carry	34	2	1
SUBB A,Rn	Subtract register from A with borrow	98-9F	1	1
SUBB A,direct	Subtract direct byte from A with borrow	95	2	1
SUBB A,@Ri	Subtract indirect memory from A with borrow	96-97	1	1
SUBB A,#data	Subtract immediate from A with borrow	94	2	1
INC A	Increment A	04	1	1
INC Rn	Increment register	08-0F	1	1
INC direct	Increment direct byte	05	2	1
INC @Ri	Increment indirect memory	06-07	1	1
DEC A	Decrement A	14	1	1
DEC Rn	Decrement register	18-1F	1	1
DEC direct	Decrement direct byte	15	2	1

Table 4-3 Instruction Table (cont'd)

Mnemonic	Description	Hex Code	Bytes	Cycles
DEC @Ri	Decrement indirect memory	16-17	1	1
INC DPTR	Increment data pointer	A3	1	2
MUL AB	Multiply A by B	A4	1	4
DIV AB	Divide A by B	84	1	4
DA A	Decimal Adjust A	D4	1	1
	LOGICAL			
ANL A,Rn	AND register to A	58-5F	1	1
ANL A,direct	AND direct byte to A	55	2	1
ANL A,@Ri	AND indirect memory to A	56-57	1	1
ANL A,#data	AND immediate to A	54	2	1
ANL direct,A	AND A to direct byte	52	2	1
ANL direct,#data	AND immediate to direct byte	53	3	2
ORL A,Rn	OR register to A	48-4F	1	1
ORL A,direct	OR direct byte to A	45	2	1
ORL A,@Ri	OR indirect memory to A	46-47	1	1
ORL A,#data	OR immediate to A	44	2	1
ORL direct,A	OR A to direct byte	42	2	1
ORL direct,#data	OR immediate to direct byte	43	3	2
XRL A,Rn	Exclusive-OR register to A	68-6F	1	1
XRL A,direct	Exclusive-OR direct byte to A	65	2	1
XRL A,@Ri	Exclusive-OR indirect memory to A	66-67	1	1
XRL A,#data	Exclusive-OR immediate to A	64	2	1
XRL direct,A	Exclusive-OR A to direct byte	62	2	1
XRL direct,#data	Exclusive-OR immediate to direct byte	63	3	2
CLR A	Clear A	E4	1	1
CPL A	Complement A	F4	1	1
SWAP A	Swap Nibbles of A	C4	1	1
RL A	Rotate A left	23	1	1
RLC A	Rotate A left through carry	33	1	1

Table 4-3 Instruction Table (cont'd)

Mnemonic	Description	Hex Code	Bytes	Cycles
RR A	Rotate A right	03	1	1
RRC A	Rotate A right through carry	13	1	1
	DATA TRANSFER			
MOV A,Rn	Move register to A	E8-EF	1	1
MOV A,direct	Move direct byte to A	E5	2	1
MOV A,@Ri	Move indirect memory to A	E6-E7	1	1
MOV A,#data	Move immediate to A	74	2	1
MOV Rn,A	Move A to register	F8-FF	1	1
MOV Rn,direct	Move direct byte to register	A8-AF	2	2
MOV Rn,#data	Move immediate to register	78-7F	2	1
MOV direct,A	Move A to direct byte	F5	2	1
MOV direct,Rn	Move register to direct byte	88-8F	2	2
MOV direct,direct	Move direct byte to direct byte	85	3	2
MOV direct,@Ri	Move indirect memory to direct byte	86-87	2	2
MOV direct,#data	Move immediate to direct byte	75	3	2
MOV @Ri,A	Move A to indirect memory	F6-F7	1	1
MOV @Ri,direct	Move direct byte to indirect memory	A6-A7	2	2
MOV @Ri,#data	Move immediate to indirect memory	76-77	2	1
MOV DPTR,#data16	Move immediate to data pointer	90	3	2
MOVC A,@A+DPTR	Move code byte relative DPTR to A	93	1	2
MOVC A,@A+PC	Move code byte relative PC to A	83	1	2
MOVX A,@Ri	Move external data (A8) to A	E2-E3	1	2
MOVX A,@DPTR	Move external data (A16) to A	E0	1	2
MOVX @Ri,A	Move A to external data (A8)	F2-F3	1	2
MOVX @DPTR,A	Move A to external data (A16)	F0	1	2
PUSH direct	Push direct byte onto stack	C0	2	2
POP direct	Pop direct byte from stack	D0	2	2

Table 4-3 Instruction Table (cont'd)

Mnemonic	Description	Hex Code	Bytes	Cycles
XCH A,Rn	Exchange A and register	C8-CF	1	1
XCH A,direct	Exchange A and direct byte	C5	2	1
XCH A,@Ri	Exchange A and indirect memory	C6-C7	1	1
XCHD A,@Ri	Exchange A and indirect memory nibble	D6-D7	1	1
BOOLEAN				
CLR C	Clear carry	C3	1	1
CLR bit	Clear direct bit	C2	2	1
SETB C	Set carry	D3	1	1
SETB bit	Set direct bit	D2	2	1
CPL C	Complement carry	B3	1	1
CPL bit	Complement direct bit	B2	2	1
ANL C,bit	AND direct bit to carry	82	2	2
ANL C,/bit	AND direct bit inverse to carry	B0	2	2
ORL C,bit	OR direct bit to carry	72	2	2
ORL C,/bit	OR direct bit inverse to carry	A0	2	2
MOV C,bit	Move direct bit to carry	A2	2	1
MOV bit,C	Move carry to direct bit	92	2	2
BRANCHING				
ACALL addr11	Absolute call within current 2 K	11->F1	2	2
LCALL addr16	Long call to addr16	12	3	2
RET	Return from subroutine	22	1	2
RETI	Return from interrupt routine	32	1	2
AJMP addr11	Absolute jump within current 2 K	01->E1	2	2
LJMP addr16	Long jump unconditional	02	3	2
SJMP rel	Short jump to relative address	80	2	2
JC rel	Jump relative on carry = 1	40	2	2
JNC rel	Jump relative on carry = 0	50	2	2
JB bit,rel	Jump relative on direct bit = 1	20	3	2
JNB bit,rel	Jump relative on direct bit = 0	30	3	2

Table 4-3 Instruction Table (cont'd)

Mnemonic	Description	Hex Code	Bytes	Cycles
JBC bit,rel	Jump relative and clear on direct bit = 1	10	3	2
JMP @A+DPTR	Jump indirect relative DPTR	73	1	2
JZ rel	Jump relative on accumulator = 0	60	2	2
JNZ rel	Jump relative on accumulator = 1	70	2	2
CJNE A,direct,rel	Compare direct memory to accumulator, jump relative if not equal	B5	3	2
CJNE A,#data,rel	Compare immediate to accumulator, jump relative if not equal	B4	3	2
CJNE Rn,#data,rel	Compare immediate to register, jump relative if not equal	B8-BF	3	2
CJNE @Ri,#data,rel	Compare immediate to indirect memory, jump relative if not equal	B6-B7	3	2
DJNZ Rn,rel	Decrement register and jump relative if not zero	D8-DF	2	2
DJNZ direct,rel	Decrement direct memory and jump relative if not zero	D5	3	2
MISCELLANEOUS				
NOP	No operation	00	1	1
ADDITIONAL INSTRUCTIONS (selected through EO[7:4])				
MOVC @(DPTR++),A	XC800-specific instruction for software download into program memory: Copy from accumulator, then increment DPTR	A5	1	2
TRAP	XC800-specific software break command	A5	1	1

The definition of the symbols used in data addressing are:

- Rn: Working register R0-R7
- direct: 128 internal RAM locations, special function registers
- @Ri: Indirect internal or external RAM location addressed by register R0 or R1
- #data: 8-bit constant included in instruction
- #data16: 16-bit constant included in instruction

- bit: 128 bit-addressable bits of lower internal data RAM, any bit-addressable bits of special function registers
- A: Accumulator

The definition of the symbols used in program addressing are:

- addr16: Destination address for LCALL and LJMP may be anywhere within the 64 Kbytes of the active bank located in program space
- addr11: Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction.
- rel: SJMP and all conditional jumps include an 8-bit offset byte. Range is + 127/– 128 bytes relative to the first byte of the following instruction.

4.3.3 Instruction Definitions

The instructions are grouped according to basic operation, and described in alphabetical order according to the operation mnemonic.

Absolute Call

ACALL addr11

Table 4-4 ACALL

Description:	ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2-Kbyte block of program memory as the first byte of the instruction following ACALL. No flags are affected.
Example:	Initially SP equals 07_H. The label “SUBRTN” is at program memory location 0345_H. After executing the instruction ACALL SUBRTN at location 0123_H, SP will contain 09_H, internal RAM location 08_H and 09_H will contain 25_H and 01_H, respectively, and the PC will contain 0345_H.
Instruction:	
ACALL addr11	Operation: (PC) ←(PC) + 2 (SP) ←(SP) + 1 ((SP)) ←(PC7-0) (SP) ←(SP) + 1 ((SP)) ←(PC15-8) (PC10-0) ←page address Bytes: 2 Cycles: 2 Encoding: a10 a9 a8 1 | 0 0 0 1 | a7 a6 a5 a4 | a3 a2 a1 a0

Add

ADD A, <src-byte>

Table 4-5 ADD

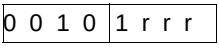
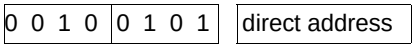
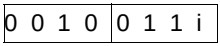
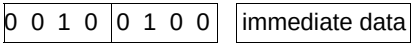
Description:	ADD adds the byte variable indicated to the accumulator, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.
Example:	The accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B). The instruction ADD A,R0 will leave $6D_H$ (01101101_B) in the accumulator with the AC flag cleared and both the carry flag and OV set to 1.
Instruction:	
ADD A,Rn	Operation: (A) ←(A) + (Rn) Bytes: 1 Cycles: 1 Encoding: 
ADD A,direct	Operation: (A) ←(A) + (direct) Bytes: 2 Cycles: 1 Encoding: 

Table 4-5 ADD (cont'd)

ADD A,@Ri	Operation: (A) ←(A) + ((Ri)) Bytes: 1 Cycles: 1 Encoding: 
ADD A,#data	Operation: (A) ←(A) + #data Bytes: 2 Cycles: 1 Encoding: 

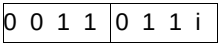
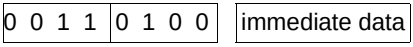
Add with Carry

ADDC A, <src-byte>

Table 4-6 ADDC

Description:	ADDC simultaneously adds the byte variable indicated, the carry flag and the accumulator contents, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.
Example:	The accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B) with the carry flag set. The instruction ADDC A,R0 will leave $6E_H$ (01101110_B) in the accumulator with AC cleared and both the carry flag and OV set to 1.
Instruction:	
ADDC A,Rn	Operation: (A) ←(A) + (C) + (Rn) Bytes: 1 Cycles: 1 Encoding: 0 0 1 1 | 1 r r r
ADDC A,direct	Operation: (A) ←(A) + (C) + (direct) Bytes: 2 Cycles: 1 Encoding: 0 0 1 1 | 0 1 0 1 | direct address

Table 4-6 ADDC (cont'd)

ADDC A,@Ri	Operation: (A) ←(A) + (C) + ((Ri)) Bytes: 1 Cycles: 1 Encoding: 
ADDC A,#data	Operation: (A) ←(A) + (C) + #data Bytes: 2 Cycles: 1 Encoding: 

Absolute Jump

AJMP addr11

Table 4-7 AJMP

Description:	AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2-Kbyte block of program memory as the first byte of the instruction following AJMP.
Example:	The label “JMPADR” is at program memory location 0123_H. The instruction AJMP JMPADR is at location 0345_H and will load the PC with 0123_H.
Instruction:	
AJMP addr11	Operation: (PC) ←(PC) + 2 (PC10-0) ←page address Bytes: 2 Cycles: 2 Encoding: a10 a9 a8 0 0 0 0 1 | a7 a6 a5 a4 a3 a2 a1 a0

Logical Byte AND

ANL <dest-byte>, <src-byte>

Table 4-8 ANL (Byte)

Description:	ANL performs the bitwise logical AND operation between the byte variables indicated and stores the results in the destination variable. No flags are affected (except P, if <dest-byte> = A). The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	If the accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B) then the instruction ANL A,R0 will leave 81_H (10000001_B) in the accumulator. When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the accumulator at run-time. The instruction ANL P1, #01110011_B will clear bits 7, 3, and 2 of output port 1.
Instruction:	
ANL A, Rn	Operation: (A) ← (A) ∧ (Rn) Bytes: 1 Cycles: 1 Encoding: 0 1 0 1 | 1 r r r

Table 4-8 ANL (Byte) (cont'd)

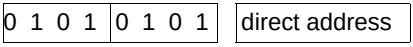
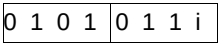
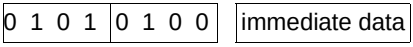
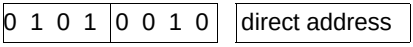
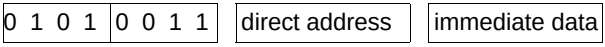
ANL A,direct	Operation: (A) ←(A) ∧ (direct) Bytes: 2 Cycles: 1 Encoding: 
ANL A,@Ri	Operation: (A) ←(A) ∧ ((Ri)) Bytes: 1 Cycles: 1 Encoding: 
ANL A,#data	Operation: (A) ←(A) ∧ #data Bytes: 2 Cycles: 1 Encoding: 

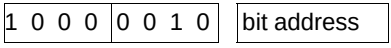
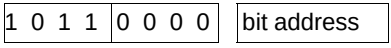
Table 4-8 ANL (Byte) (cont'd)

ANL direct, A	Operation: (direct) ←(direct) ∧ (A) Bytes: 2 Cycles: 1 Encoding: 
ANL direct, #data	Operation: (direct) ←(direct) ∧ #data Bytes: 3 Cycles: 2 Encoding: 

Logical Bit AND

ANL C, <src-bit>

Table 4-9 ANL (Bit)

Description:	If the Boolean value of the source bit is a logic 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash (“/”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected. Only direct bit addressing is allowed for the source operand.
Example:	Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0: MOV C,P1.0 ; Load carry with input pin state ANL C,ACC.7; AND carry with accumulator bit 7 ANL C,/OV; AND with inverse of overflow flag
Instruction:	
ANL C, bit	Operation: (C) ←(C) ∧ (bit) Bytes: 2 Cycles: 2 Encoding: 
ANL C, /bit	Operation: (C) ←(C) ∧ / (bit) Bytes: 2 Cycles: 2 Encoding: 

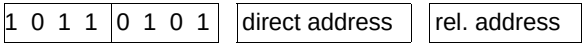
Compare, Jump if not Equal

CJNE <dest-byte>, <src-byte>, rel

Table 4-10 CJNE

Description:	CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected. The first two operands allow four addressing mode combinations: the accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.
Example:	The accumulator contains 34_H. Register 7 contains 56_H. The first instruction in the sequence CJNE R7, # 60H, NOT_EQ ; . . ..; R7 = 60_H NOT_EQ JC REQ_LOW; If R7 < 60_H ; . . ..; R7 > 60_H sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60_H. If the data being presented to port 1 is also 34_H, then the instruction WAIT: CJNE A,P1,WAIT clears the carry flag and continues with the next instruction in sequence, since the accumulator does equal the data read from P1. (If some other value was input on P1, the program will loop at this point until the P1 data changes to 34_H).
Instruction:	

Table 4-10 CJNE (cont'd)

CJNE A, direct, rel	Operation:
	(PC) ←(PC) + 3
	if (A) < > (direct)
	then (PC) ←(PC) + relative offset
	if (A) < (direct)
	then (C) ←1
	else (C) ←0
	Bytes: 3
	Cycles: 2
	Encoding:
	 

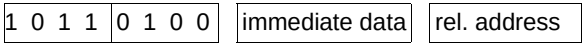
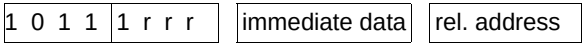
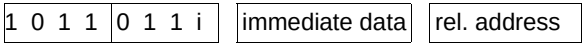
CJNE A, #data, rel	Operation:
	(PC) ←(PC) + 3
	if (A) < > data
	then (PC) ←(PC) + relative offset
	if (A) ←data
	then (C) ←1
	else (C) ←0
	Bytes: 3
	Cycles: 2
	Encoding:
	 

Table 4-10 CJNE (cont'd)

CJNE Rn, #data, rel	Operation:
	(PC) ←(PC) + 3
	if (Rn) < > data
	then (PC) ←(PC) + relative offset
	if (Rn) < data
	then (C) ←1
	else (C) ←0
	Bytes: 3
	Cycles: 2
	Encoding:
	 

CJNE @Ri, #data, rel	Operation:
	(PC) ←(PC) + 3
	if ((Ri)) < > data
	then (PC) ←(PC) + relative offset
	if ((Ri)) < data
	then (C) ←1
	else (C) ←0
	Bytes: 3
	Cycles: 2
	Encoding:
	 

Clear Accumulator

CLR A

Table 4-11 CLR (A)

Description:	The accumulator is cleared (all bits set to zero). No flags are affected.
Example:	The accumulator contains $5C_H$ (01011100_B). The instruction CLR A will leave the accumulator set to 00_H (00000000_B).
Instruction:	
CLR A	Operation: (A) ←0 Bytes: 1 Cycles: 1 Encoding: 1 1 1 0 | 0 1 0 0

Clear Bit

CLR <bit>

Table 4-12 CLR (Bit)

Description:	The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.
Example:	Port 1 has previously been written with $5D_H$ (01011101_B). The instruction CLR P1.2 will leave the port set to 59_H (01011001_B).
Instruction:	
CLR C	Operation: (C) ←0 Bytes: 1 Cycles: 1 Encoding: 1 1 0 0 | 0 0 1 1
CLR bit	Operation: (bit) ←0 Bytes: 2 Cycles: 1 Encoding: 1 1 0 0 | 0 0 1 0 | bit address

Complement Accumulator

CPL A

Table 4-13 CPL (A)

Description:	Each bit of the accumulator is logically complemented (ones complement). Bits that previously contained a one are changed to zero and vice versa. No flags are affected.
Example:	The accumulator contains $5C_H$ (01011100_B). The instruction CPL A will leave the accumulator set to $0A3_H$ (10100011_B).
Instruction:	
CPL A	Operation: (A) ←/ (A) Bytes: 1 Cycles: 1 Encoding: 1 1 1 1 | 0 1 0 0

Complement Bit

CPL <bit>

Table 4-14 CPL (Bit)

Description:	The bit variable specified is complemented. A bit that had been a one is changed to zero and vice versa. No other flags are affected. CPL can operate on the carry or any directly addressable bit. *Note: When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, not the input pin.*
Example:	Port 1 has previously been written with $5D_H$ (01011101_B). The instruction sequence CPL P1.1 CPL P1.2 will leave the port set to $5B_H$ (01011011_B).
Instruction:	
CPL C	Operation: (C) ← / (C) Bytes: 1 Cycles: 1 Encoding: 1 0 1 1 | 0 0 1 1
CPL bit	Operation: (bit) ← / (bit) Bytes: 2 Cycles: 1 Encoding: 1 0 1 1 | 0 0 1 0 | bit address

Decimal Adjust Accumulator for Addition

DA A

Table 4-15 DA

Description:	DA A adjusts the eight-bit value in the accumulator resulting from the earlier addition of two variables (each in packed BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition. If accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise. If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but would not clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected. All of this occurs during the one instruction cycle. Essentially; this instruction performs the decimal conversion by adding 00_H, 06_H, 60_H, or 66_H to the accumulator, depending on initial accumulator and PSW conditions.

Table 4-15 DA (cont'd)

Example:	The accumulator holds the value 56_H (01010110_B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67_H (01100111_B) representing the packed BCD digits of the decimal number 67. The carry flag is set. The instruction sequence
	ADDC A,R3
	DA A
	will first perform a standard twos complement binary addition, resulting in the value $0BE_H$ (10111110_B) in the accumulator. The carry and auxiliary carry flags will be cleared.
	The decimal adjust instruction will then alter the accumulator to the value 24_H (00100100_B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag will be set by the decimal adjust instruction, indicating that a decimal overflow occurred. The true sum 56, 67, and 1 is 124.
	
	BCD variables can be incremented or decremented by adding 01_H or 99_H. If the accumulator initially holds 30_H (representing the digits of 30 decimal), then the instruction sequence
	ADD A, #99H
	DA A
	will leave the carry set and 29_H in the accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 – 1 = 29.

Instruction:	
DA A	Operation: Contents of accumulator are BCD if [[(A3-0) > 9] ∨ [(AC) = 1]] then (A3-0) ←(A3-0) + 6 and if [[(A7-4) > 9] ∨ [(C) = 1]] then (A7-4) ←(A7-4) + 6 Bytes: 1 Cycles: 1 Encoding: 1 1 0 1 | 0 1 0 0

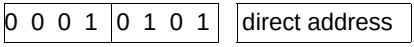
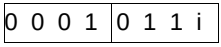
Decrement

DEC <byte>

Table 4-16 DEC

Description:	The variable indicated is decremented by 1. An original value of 00_H will underflow to $0FF_H$. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	Register 0 contains $7F_H$ (01111111_B). Internal RAM locations $7E_H$ and $7F_H$ contain 00_H and 40_H, respectively. The instruction sequence DEC @R0 DEC R0 DEC @R0 will leave register 0 set to $7E_H$ and internal RAM locations $7E_H$ and $7F_H$ set to $0FF_H$ and $3F_H$.
Instruction:	
DEC A	Operation: (A) ←(A) – 1 Bytes: 1 Cycles: 1 Encoding: 0 0 0 1 | 0 1 0 0
DEC Rn	Operation: (Rn) ←(Rn) – 1 Bytes: 1 Cycles: 1 Encoding: 0 0 0 1 | 1 r r r

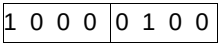
Table 4-16 DEC (cont'd)

DEC direct	Operation: (direct) ←(direct) – 1 Bytes: 2 Cycles: 1 Encoding: 
DEC @Ri	Operation: ((Ri)) ←((Ri)) – 1 Bytes: 1 Cycles: 1 Encoding: 

Divide

DIV AB

Table 4-17 DIV

Description:	DIV AB divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared. Exception: If B had originally contained 00_H, the values returned in the accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.
Example:	The accumulator contains 251 ($0FB_H$ or 11111011_B) and B contains 18 (12_H or 00010010_B). The instruction DIV AB will leave 13 in the accumulator ($0D_H$ or 00001101_B) and the value 17 (11_H or 00010001_B) in B, since 251 = (13x18) + 17. Carry and OV will both be cleared.
Instruction:	
DIV AB	Operation: (A) ←Quo[(A) / (B)] (B) ←Rem[(A) / (B)] Bytes: 1 Cycles: 4 Encoding: 

Decrement, Jump if not Zero

DJNZ <byte>, <rel-addr>

Table 4-18 DJNZ

Description:	DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00_H will underflow to $0FF_H$. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction. The location decremented may be a register or directly addressed byte. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	Internal RAM locations 40_H, 50_H, and 60_H contain the values, 01_H, 70_H, and 15_H, respectively. The instruction sequence DJNZ 40H,LABEL_1 DJNZ 50H,LABEL_2 DJNZ 60H,LABEL_3 will cause a jump to the instruction at label LABEL_2 with the values 00_H, $6F_H$, and 15_H in the three RAM locations. The first jump was not taken because the result was zero. This instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence MOV R2, #8 TOGGLE: CPL P1.7 DJNZ R2,TOGGLE will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.
Instruction:	

Table 4-18 DJNZ (cont'd)

DJNZ Rn, rel	Operation: (PC) ←(PC) + 2 (Rn) ←(Rn) – 1 if (Rn) > 0 or (Rn) < 0 then (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 1 1 0 1 | 1 r r r | rel. address
DJNZ direct, rel	Operation: (PC) ←(PC) + 2 (direct) ←(direct) – 1 if (direct) > 0 or (direct) < 0 then (PC) ←(PC) + rel Bytes: 3 Cycles: 2 Encoding: 1 1 0 1 | 0 1 0 1 | direct address | rel. address

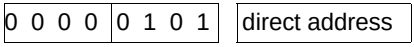
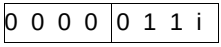
Increment

INC <byte>

Table 4-19 INC (Byte)

Description:	INC increments the indicated variable by 1. An original value of $0FF_H$ will overflow to 00_H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	Register 0 contains $7E_H$ (01111110_B). Internal RAM locations $7E_H$ and $7F_H$ contain $0FF_H$ and 40_H, respectively. The instruction sequence INC @R0 INC R0 INC @R0 will leave register 0 set to $7F_H$ and internal RAM locations $7E_H$ and $7F_H$ holding (respectively) 00_H and 41_H.
Instruction:	
INC A	Operation: (A) ←(A) + 1 Bytes: 1 Cycles: 1 Encoding: 0 0 0 0 | 0 1 0 0
INC Rn	Operation: (Rn) ←(Rn) + 1 Bytes: 1 Cycles: 1 Encoding: 0 0 0 0 | 1 r r r

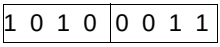
Table 4-19 INC (Byte) (cont'd)

INC direct	Operation: (direct) ←(direct) + 1 Bytes: 2 Cycles: 1 Encoding: 
INC @Ri	Operation: ((Ri)) ←((Ri)) + 1 Bytes: 1 Cycles: 1 Encoding: 

Increment Data Pointer

INC DPTR

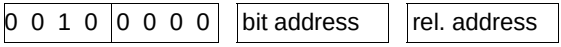
Table 4-20 INC (DPTR)

Description:	Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from $0FF_H$ to 00_H will increment the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented.
Example:	Registers DPH and DPL contain 12_H and $0FE_H$, respectively. The instruction sequence INC DPTR INC DPTR INC DPTR will change DPH and DPL to 13_H and 01_H.
Instruction:	
INC DPTR	Operation: (DPTR) ←(DPTR) + 1 Bytes: 1 Cycles: 2 Encoding: 

Jump if Bit is Set

JB <bit>,rel

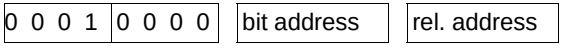
Table 4-21 JB

Description:	If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.
Example:	The data present at input port 1 is 11001010_B. The accumulator holds 56 (01010110_B). The instruction sequence JB P1.2,LABEL1 JB ACC.2,LABEL2 will cause program execution to branch to the instruction at label LABEL2.
Instruction:	
JB bit, rel	Operation: (PC) ←(PC) + 3 if (bit) = 1 then (PC) ←(PC) + rel Bytes: 3 Cycles: 2 Encoding: 

Jump if Bit is Set, and Clear Bit

JBC <bit>,rel

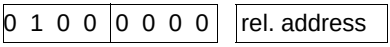
Table 4-22 JBC

Description:	If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. In either case, clear the designated bit. The branch destination is computed by adding the signed relative displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected. *Note: When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.*
Example:	The accumulator holds 56_H (01010110_B). The instruction sequence JBC ACC.3,LABEL1 JBC ACC.2,LABEL2 will cause program execution to continue at the instruction identified by the label LABEL2, with the accumulator modified to 52_H (01010010_B).
Instruction:	
JBC bit, rel	Operation: (PC) ←(PC) + 3 if (bit) = 1 then (bit) ←0 (PC) ←(PC) + rel Bytes: 3 Cycles: 2 Encoding: 

Jump if Carry is Set

JC rel

Table 4-23 JC

Description:	If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.
Example:	The carry flag is cleared. The instruction sequence JC LABEL1 CPL C JC LABEL2 will set the carry and cause program execution to continue at the instruction identified by the label LABEL2.
Instruction:	
JC rel	Operation: (PC) ←(PC) + 2 if (C) = 1 then (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 

Jump Indirect

JMP @A + DPTR

Table 4-24 JMP

Description:	Add the eight-bit unsigned contents of the accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the accumulator nor the data pointer is altered. No flags are affected.
Example:	An even number from 0 to 6 is in the accumulator. The following sequence of instructions will branch to one of four AJMP instructions in a jump table starting at JMP_TBL: MOV DPTR, #JMP_TBL JMP @A + DPTR JMP_TBL: AJMP LABEL0 AJMP LABEL1 AJMP LABEL2 AJMP LABEL3 If the accumulator equals 04_H when starting this sequence, execution will jump to label LABEL2. Remember that AJMP is a two-byte instruction, so the jump instructions start at every other address.
Instruction:	
JMP @A+DPTR	Operation: (PC) ←(A) + (DPTR) Bytes: 1 Cycles: 2 Encoding: 0 1 1 1 | 0 0 1 1

Jump if Bit is Not Set

JNB <bit>,rel

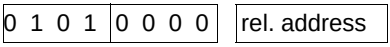
Table 4-25 JNB

Description:	If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.
Example:	The data present at input port 1 is 11001010_B. The accumulator holds 56_H (01010110_B). The instruction sequence JNB P1.3,LABEL1 JNB ACC.3,LABEL2 will cause program execution to continue at the instruction at label LABEL2.
Instruction:	
JNB bit, rel	Operation: (PC) ←(PC) + 3 if (bit) = 0 then (PC) ←(PC) + rel Bytes: 3 Cycles: 2 Encoding: 0 0 1 1 | 0 0 0 0 | bit address | rel. address

Jump if Carry is Not Set

JNC rel

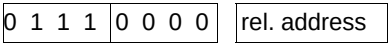
Table 4-26 JNC

Description:	If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.
Example:	The carry flag is set. The instruction sequence JNC LABEL1 CPL C JNC LABEL2 will clear the carry and cause program execution to continue at the instruction identified by the label LABEL2.
Instruction:	
JNC rel	Operation: (PC) ←(PC) + 2 if (C) = 0 then (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 

Jump if Accumulator is Not Zero

JNZ rel

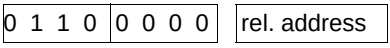
Table 4-27 JNZ

Description:	If any bit of the accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.
Example:	The accumulator originally holds 00_H. The instruction sequence JNZ LABEL1 INC A JNZ LABEL2 will set the accumulator to 01_H and continue at label LABEL2.
Instruction:	
JNZ rel	Operation: (PC) ←(PC) + 2 if (A) ≠ 0 then (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 

Jump if Accumulator is Zero

JZ rel

Table 4-28 JZ

Description:	If all bits of the accumulator are zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The accumulator is not modified. No flags are affected.
Example:	The accumulator originally contains 01_H. The instruction sequence JZ LABEL1 DEC A JZ LABEL2 will change the accumulator to 00_H and cause program execution to continue at the instruction identified by the label LABEL2.
Instruction:	
JZ rel	Operation: (PC) ←(PC) + 2 if (A) = 0 then (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 

Long Call

LCALL addr16

Table 4-29 LCALL

Description:	LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the stack pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64-Kbyte program memory address space. No flags are affected.
Example:	Initially the stack pointer equals 07_H. The label “SUBRTN” is assigned to program memory location 1234_H. After executing the instruction LCALL SUBRTN at location 0123_H, the stack pointer will contain 09_H, internal RAM locations 08_H and 09_H will contain 26_H and 01_H, and the PC will contain 1234_H.
Instruction:	
LCALL addr16	Operation: (PC) ←(PC) + 3 (SP) ←(SP) + 1 ((SP)) ←(PC7-0) (SP) ←(SP) + 1 ((SP)) ←(PC15-8) (PC) ←addr15-0 Bytes: 3 Cycles: 2 Encoding:

0 0 0 1 | 0 0 1 0	addr15..addr8	addr7..addr0	

Long Jump

LJMP addr16

Table 4-30 LJMP

Description:	LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64-Kbyte program memory address space. No flags are affected.
Example:	The label “JMPADR” is assigned to the instruction at program memory location 1234_H. The instruction LJMP JMPADR at location 0123_H will load the program counter with 1234_H.
Instruction:	
LJMP addr16	Operation: (PC) ←addr15-0 Bytes: 3 Cycles: 2 Encoding: 0 0 0 0 | 0 0 1 0 | addr15..addr8 | addr7..addr0

Move Byte Variable

MOV <dest-byte>, <src-byte>

Table 4-31 MOV (Byte)

Description:	The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example:	Internal RAM location 30_H holds 40_H. The value of RAM location 40_H is 10_H. The data present at input port 1 is 11001010B ($0CA_H$).
	MOV R0, #30H ; R0 < = 30H
	MOV A, @R0 ; A < = 40H
	MOV R1,A ; R1 < = 40H
	MOV B, @R1 ; B < = 10H
	MOV R1,A ; R1 < = 40H
	MOV @R1, P1 ; RAM (40H) < = 0CAH
	MOV P2,P1 ; P2 < = 0CAH
	leaves the value 30_H in register 0, 40_H in both the accumulator and register 1, 10_H in register B, and $0CA_H$ (11001010_B) both in RAM location 40_H and output on port 2.

Instruction:	
MOV A, Rn	Operation: (A) ←(Rn) Bytes: 1 Cycles: 1 Encoding: 1 1 1 0 1 r r r

Table 4-31 MOV (Byte) (cont'd)

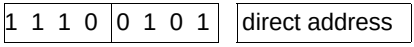
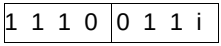
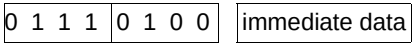
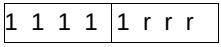
MOV A, direct	Operation: (A) ←(direct) Bytes: 2 Cycles: 1 Encoding:  *Note: MOV A,ACC is not a valid instruction. The content of the accumulator after the execution of this instruction is undefined.*
MOV A, @Ri	Operation: (A) ←((Ri)) Bytes: 1 Cycles: 1 Encoding: 
MOV A, #data	Operation: (A) ←#data Bytes: 2 Cycles: 1 Encoding: 
MOV Rn, A	Operation: (Rn) ←(A) Bytes: 1 Cycles: 1 Encoding: 

Table 4-31 MOV (Byte) (cont'd)

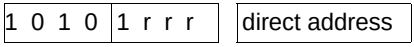
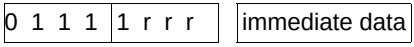
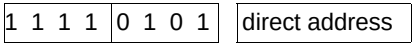
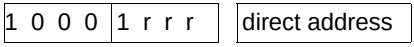
MOV Rn, direct	Operation: (Rn) ←(direct) Bytes: 2 Cycles: 2 Encoding: 
MOV Rn, #data	Operation: (Rn) ←#data Bytes: 2 Cycles: 1 Encoding: 
MOV direct, A	Operation: (direct) ←(A) Bytes: 2 Cycles: 1 Encoding: 
MOV direct, Rn	Operation: (direct) ←(Rn) Bytes: 2 Cycles: 2 Encoding: 

Table 4-31 MOV (Byte) (cont'd)

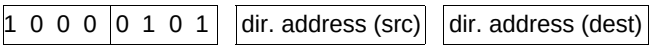
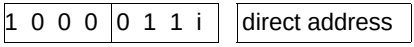
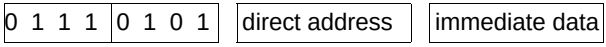
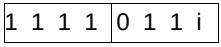
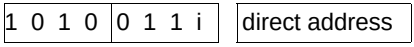
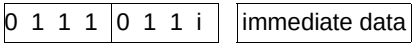
MOV direct, direct	Operation: (direct) ←(direct) Bytes: 3 Cycles: 2 Encoding: 
MOV direct, @Ri	Operation: (direct) ←((Ri)) Bytes: 2 Cycles: 2 Encoding: 
MOV direct, #data	Operation: (direct) ←#data Bytes: 3 Cycles: 2 Encoding: 
MOV @Ri, A	Operation: ((Ri)) ←(A) Bytes: 1 Cycles: 1 Encoding: 

Table 4-31 MOV (Byte) (cont'd)

MOV @Ri, direct	Operation: ((Ri)) ←(direct) Bytes: 2 Cycles: 2 Encoding: 
MOV @Ri, #data	Operation: ((Ri)) ←#data Bytes: 2 Cycles: 1 Encoding: 

Move Bit Data

MOV <dest-bit>, <src-bit>

Table 4-32 MOV (Bit)

Description:	The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.

Example:	The carry flag is originally set. The data present at input port 3 is 11000101_B. The data previously written to output port 1 is 35_H (00110101_B).
	MOV P1.3,C
	MOV C,P3.3
	MOV P1.2,C
	will leave the carry cleared and change port 1 to 39_H (00111001_B).

Instruction:	
MOV C, bit	Operation: (C) ←(bit) Bytes: 2 Cycles: 1 Encoding: 1 0 1 0 | 0 0 1 0 | bit address
MOV bit, C	Operation: (bit) ←(C) Bytes: 2 Cycles: 2 Encoding: 1 0 0 1 | 0 0 1 0 | bit address

Load Data Pointer

MOV DPTR, #data16

Table 4-33 MOV (DPTR)

Description:	The data pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected. This is the only instruction that moves 16 bits of data at once.
Example:	The instruction MOV DPTR, #1234H will load the value 1234_H into the data pointer: DPH will hold 12_H and DPL will hold 34_H.
Instruction:	
MOV DPTR, #data16	Operation: (DPTR) ←#data15-0 DPH, DPL ←#data15-8,#data7-0 Bytes: 3 Cycles: 2 Encoding: 1 0 0 1 | 0 0 0 0 | immed. data 15..8 | immed. data 7..0

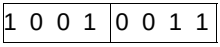
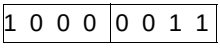
Read Code Byte

MOVC A, @A+<base-reg>

Table 4-34 MOVC (Read)

Description:	Load the accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen-bit base register, which may be either the data pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added to the accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.
Example:	A value between 0 and 3 is in the accumulator. The following instructions will translate the value in the accumulator to one of four values defined by the DB (define byte) directive. REL_PC: INC A MOVC A, @A + PC RET DB 66H DB 77H DB 88H DB 99H If the subroutine is called with the accumulator equal to 01_H, it will return with 77_H in the accumulator. The INC A before the MOVC instruction is needed to “get around” the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the accumulator instead.
Instruction:	

Table 4-34 MOVC (Read) (cont'd)

MOVC A, @A+DPTR	Operation: (A) ←((A) + (DPTR)) Bytes: 1 Cycles: 2 Encoding: 
MOVC A, @A+PC	Operation: (PC) ←(PC) + 1 (A) ←((A) + (PC)) Bytes: 1 Cycles: 2 Encoding: 

Write Code Byte

MOVC @(DPTR++), A

Table 4-35 MOVC (Write)

Description:	Store the byte content of accumulator to program memory. The address in program memory is pointed to by the data pointer. The data pointer is incremented by hardware, after the write. No flags are affected.

Example:	Store value $E4_H$ to program memory at 1000_H. Opcode $E4_H$ is the CLR A instruction.
	MOV A, #E4H
	MOV DPTR,#1000H
	MOVC @(DPTR++), A
	write CLR A to program memory at 1000_H.

Instruction:	
MOVC @(DPTR++), A	Operation: ((DPTR)) ←(A) (DPTR) = (DPTR) + 1 Bytes: 1 Cycles: 2 Encoding: 1 0 1 0 | 0 1 0 1

Note: This instruction is XC800-specific, therefore may not be supported by standard 8051 assembler. In such cases, this can be workaround by direct byte declaration and definition e.g. ".byte #A5$_H$" (syntax is assembler dependent).

Note: This instruction shares the same opcode with another XC800-specific instruction TRAP. MOVC is selected only if EO.TRAP_EN = 0.

Move External

MOVX <dest-byte>, <src-byte>

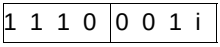
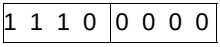
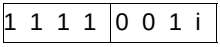
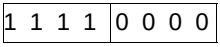
Table 4-36 MOVX

Description:	The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence the “X” appended to MOV. There are two types of instructions, differing in whether they provide an 8-bit or 16-bit indirect address to the external data RAM. In the first type, the contents of R0 or R1 in the current register bank provide an 8-bit address on the low-byte address port. Eight bits are sufficient for external I/O expansion decoding or a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX. In the second type of MOVX instructions, the data pointer generates a 16-bit address. The high-byte address port outputs the high-order eight address bits (the contents of DPH) while the low-byte address port outputs the low-order eight address bits (DPL). The special function registers of the address ports are unaffected and retain the previous contents. This form of access is faster and more efficient when accessing very large data arrays (up to 64 Kbytes), since no additional instructions are needed to set up the output ports. It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven on the address port can be addressed via the data pointer, or with code to output high-order address bits to the high-byte port followed by a MOVX instruction using R0 or R1.

Example:	An external 256-byte RAM using multiplexed address/data lines is connected to the low-byte address port. Port 3 provides control lines for the external RAM. Other ports (such as the high-byte address port) are used for normal I/O. Registers 0 and 1 contain 12_H and 34_H. Location 34_H of the external RAM holds the value 56_H. The instruction sequence
	MOVX A, @R1
	MOVX @R0,A
	copies the value 56_H into both the accumulator and external RAM location 12_H.

Instruction:	

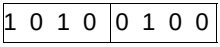
Table 4-36 MOVX (cont'd)

MOVX A, @Ri	Operation: (A) ←((Ri)) Bytes: 1 Cycles: 2 Encoding: 
MOVX A, @DPTR	Operation: (A) ←((DPTR)) Bytes: 1 Cycles: 2 Encoding: 
MOVX @Ri, A	Operation: ((Ri)) ←(A) Bytes: 1 Cycles: 2 Encoding: 
MOVX @DPTR, A	Operation: ((DPTR)) ←(A) Bytes: 1 Cycles: 2 Encoding: 

Multiply

MUL AB

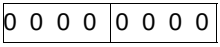
Table 4-37 MUL

Description:	MUL AB multiplies the unsigned eight-bit integers in the accumulator and register B. The low-order byte of the sixteen-bit product is left in the accumulator, and the high-order byte in B. If the product is greater than 255 ($0FF_H$) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
Example:	Originally the accumulator holds the value 80 (50_H). Register B holds the value 160 ($0A0_H$). The instruction MUL AB will give the product 12,800 (3200_H), so B is changed to 32_H (00110010_B) and the accumulator is cleared. The overflow flag is set, carry is cleared.
Instruction:	
MUL AB	Operation: (B) ←High Byte[(A) x (B)] (A) ←Low Byte[(A) x (B)] Bytes: 1 Cycles: 4 Encoding: 

No Operation

NOP

Table 4-38 NOP

Description:	Execution continues at the following instruction. Other than the PC, no registers or flags are affected.
Example:	It is desired to produce a low-going output pulse on bit 7 of port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence CLR P2.7 NOP NOP NOP NOP SETB P2.7
Instruction:	
NOP	Operation: NULL Bytes: 1 Cycles: 1 Encoding: 

Logical Byte OR

ORL <dest-byte>, <src-byte>

Table 4-39 ORL (Byte)

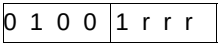
Description:	ORL performs the bitwise logical OR operation between the indicated variables, storing the results in the destination byte. No flags are affected (except P, if <dest-byte> = A). The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	If the accumulator holds $0C3_H$ (11000011_B) and R0 holds 55_H (01010101_B) then the instruction ORL A,R0 will leave the accumulator holding the value $0D7_H$ (11010111_B). When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the accumulator at run-time. The instruction ORL P1,#00110010_B will set bits 5, 4, and 1 of output port 1.
Instruction:	
ORL A, Rn	Operation: (A) ←(A) ∨ (Rn) Bytes: 1 Cycles: 1 Encoding: 

Table 4-39 ORL (Byte) (cont'd)

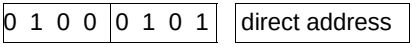
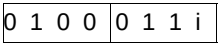
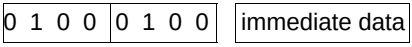
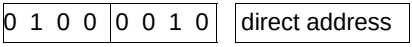
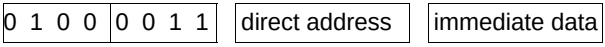
ORL A, direct	Operation: (A) ←(A) ∨ (direct) Bytes: 2 Cycles: 1 Encoding: 
ORL A, @Ri	Operation: (A) ←(A) ∨ ((Ri)) Bytes: 1 Cycles: 1 Encoding: 
ORL A, #data	Operation: (A) ←(A) ∨ #data Bytes: 2 Cycles: 1 Encoding: 

Table 4-39 ORL (Byte) (cont'd)

ORL direct, A	Operation: (direct) ←(direct) ∨ (A) Bytes: 2 Cycles: 1 Encoding: 
ORL direct, #data	Operation: (direct) ←(direct) ∨ #data Bytes: 3 Cycles: 2 Encoding: 

Logical Bit OR

ORL C, <src-bit>

Table 4-40 ORL (Bit)

Description:	Set the carry flag if the Boolean value is a logic 1; leave the carry in its current state otherwise. A slash (“/”) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.
Example:	Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, or OV = 0: MOV C,P1.0 ; Load carry with input pin P1.0 ORL C,ACC.7; OR carry with the accumulator bit 7 ORL C,/OV ; OR carry with the inverse of OV
Instruction:	
ORL C, bit	Operation: (C) ←(C) ∨ (bit) Bytes: 2 Cycles: 2 Encoding: 0 1 1 1 | 0 0 1 0 | bit address
ORL C, /bit	Operation: (C) ←(C) ∨ / (bit) Bytes: 2 Cycles: 2 Encoding: 1 0 1 0 | 0 0 0 0 | bit address

Pop from Stack

POP direct

Table 4-41 POP

Description:	The contents of the internal RAM location addressed by the stack pointer is read, and the stack pointer is decremented by one. The value read is the transfer to the directly addressed byte indicated. No flags are affected.
Example:	The stack pointer originally contains the value 32_H, and internal RAM locations 30_H through 32_H contain the values 20_H, 23_H, and 01_H, respectively. The instruction sequence POP DPH POP DPL will leave the stack pointer equal to the value 30_H and the data pointer set to 0123_H. At this point the instruction POPSP will leave the stack pointer set to 20_H. Note that in this special case the stack pointer was decremented to $2F_H$ before being loaded with the value popped (20_H).
Instruction:	
POP direct	Operation: (direct) ←((SP)) (SP) ←(SP) – 1 Bytes: 2 Cycles: 2 Encoding: 1 1 0 1 | 0 0 0 0 | direct address

Push onto Stack

PUSH direct

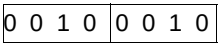
Table 4-42 PUSH

Description:	The stack pointer is incremented by one. The content of the indicated variable is then copied into the internal RAM location addressed by the stack pointer. Otherwise no flags are affected.
Example:	On entering an interrupt routine the stack pointer contains 09_H. The data pointer holds the value 0123_H. The instruction sequence PUSH DPL PUSH DPH will leave the stack pointer set to $0B_H$ and store 23_H and 01_H in internal RAM locations $0A_H$ and $0B_H$, respectively.
Instruction:	
PUSH direct	Operation: (SP) ←(SP) + 1 ((SP)) ←(direct) Bytes: 2 Cycles: 2 Encoding: 1 1 0 0 | 0 0 0 0 | direct address

Return from Subroutine

RET

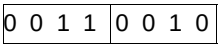
Table 4-43 RET

Description:	RET pops the high and low-order bytes of the PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.
Example:	The stack pointer originally contains the value $0B_H$. Internal RAM locations $0A_H$ and $0B_H$ contain the values 23_H and 01_H, respectively. The instruction RET will leave the stack pointer equal to the value 09_H. Program execution will continue at location 0123_H.
Instruction:	
RET	Operation: (PC15-8) ←((SP)) (SP) ←(SP) – 1 (PC7-0) ←((SP)) (SP) ←(SP) – 1 Bytes: 1 Cycles: 2 Encoding: 

Return from Interrupt

RETI

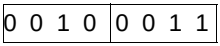
Table 4-44 RETI

Description:	RETI pops the high and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower or same-level interrupt is pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.
Example:	The stack pointer originally contains the value $0B_H$. An interrupt was detected during the instruction ending at location 0122_H. Internal RAM locations $0A_H$ and $0B_H$ contain the values 23_H and 01_H, respectively. The instruction RETI will leave the stack pointer equal to 09_H and return program execution to location 0123_H.
Instruction:	
RETI	Operation: (PC15-8) ←((SP)) (SP) ←(SP) – 1 (PC7-0) ←((SP)) (SP) ←(SP) – 1 Bytes: 1 Cycles: 2 Encoding: 

Rotate Accumulator Left

RL A

Table 4-45 RL

Description:	The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.
Example:	The accumulator holds the value $0C5_H$ (11000101_B). The instruction RL A leaves the accumulator holding the value $8B_H$ (10001011_B) with the carry unaffected.
Instruction:	
RL A	Operation: (An + 1) ←(An) n = 0-6 (A0) ←(A7) Bytes: 1 Cycles: 1 Encoding: 

Rotate Accumulator Left Through Carry Flag

RLC A

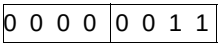
Table 4-46 RLC

Description:	The eight bits in the accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.
Example:	The accumulator holds the value $0C5_H$ (11000101_B), and the carry is zero. The instruction RLC A leaves the accumulator holding the value $8A_H$ (10001010B) with the carry set.
Instruction:	
RL A	Operation: (An + 1) ←(An) n = 0-6 (A0) ←(C) (C) ←(A7) Bytes: 1 Cycles: 1 Encoding: 0 0 1 1 | 0 0 1 1

Rotate Accumulator Right

RR A

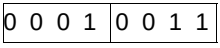
Table 4-47 RR

Description:	The eight bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.
Example:	The accumulator holds the value $0C5_H$ (11000101_B). The instruction RR A leaves the accumulator holding the value $0E2_H$ (11100010_B) with the carry unaffected.
Instruction:	
RR A	Operation: (An) ←(An + 1) n = 0-6 (A7) ←(A0) Bytes: 1 Cycles: 1 Encoding: 

Rotate Accumulator Right Through Carry Flag

RRC A

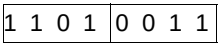
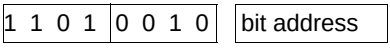
Table 4-48 RRC

Description:	The eight bits in the accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.
Example:	The accumulator holds the value $0C5_H$ (11000101_B), the carry is zero. The instruction RRC A leaves the accumulator holding the value 62_H (01100010_B) with the carry set.
Instruction:	
RRC A	Operation: (An) ←(An + 1) n=0-6 (A7) ←(C) (C) ←(A0) Bytes: 1 Cycles: 1 Encoding: 

Set Bit

SETB <bit>

Table 4-49 SETB

Description:	SETB sets the indicated bit to one. SETB can operate on the carry flag or any directiy addressable bit. No other flags are affected.
Example:	The carry flag is cleared. Output port 1 has been written with the value 34_H (00110100_B). The instructions SETB C SETB P1.0 will leave the carry flag set to 1 and change the data output on port 1 to 35_H (00110101_B).
Instruction:	
SETB C	Operation: (C) ←1 Bytes: 1 Cycles: 1 Encoding: 
SETB bit	Operation: (bit) ←1 Bytes: 2 Cycles: 1 Encoding: 

Short Jump

SJMP rel

Table 4-50 SJMP

Description:	Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.
Example:	The label “RELADR” is assigned to an instruction at program memory location 0123_H. The instruction SJMP RELADR will assemble into location 0100_H. After the instruction is executed, the PC will contain the value 0123_H. *Note: Under the above conditions the instruction following SJMP will be at 102_H. Therefore, the displacement byte of the instruction will be the relative offset (0123_H-0102_H) = 21_H. In other words, an SJMP with a displacement of $0FE_H$ would be a one-instruction infinite loop.*
Instruction:	
SJMP rel	Operation: (PC) ←(PC) + 2 (PC) ←(PC) + rel Bytes: 2 Cycles: 2 Encoding: 1 0 0 0 | 0 0 0 0 | rel. address

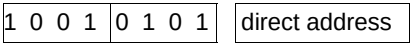
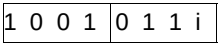
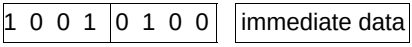
Subtract with Borrow

SUBB A, <src-byte>

Table 4-51 SUBB

Description:	SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the accumulator along with the source operand). AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6 but not into bit 7, or into bit 7 but not bit 6. When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number. The source operand allows four addressing modes: register, direct, register-indirect, or immediate.
Example:	The accumulator holds $0C9_H$ (11001001_B), register 2 holds 54_H (01010100_B), and the carry flag is set. The instruction SUBB A,R2 will leave the value 74_H (01110100_B) in the accumulator, with the carry flag and AC cleared but OV set. Notice that $0C9_H$ minus 54_H is 75_H. The difference between this and the above result is due to the (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.
Instruction:	
SUBB A, Rn	Operation: (A) ←(A) – (C) – (Rn) Bytes: 1 Cycles: 1 Encoding: 1 0 0 1 | 1 r r r

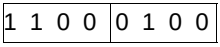
Table 4-51 SUBB (cont'd)

SUBB A, direct	Operation: (A) ←(A) – (C) – (direct) Bytes: 2 Cycles: 1 Encoding: 
SUBB A, @Ri	Operation: (A) ←(A) – (C) – ((Ri)) Bytes: 1 Cycles: 1 Encoding: 
SUBB A, #data	Operation: (A) ←(A) – (C) – #data Bytes: 2 Cycles: 1 Encoding: 

Swap Accumulator Nibbles

SWAP A

Table 4-52 SWAP

Description:	SWAP A interchanges the low and high-order nibbles (four-bit fields) of the accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.
Example:	The accumulator holds the value $0C5_H$ (11000101_B). The instruction SWAP A leaves the accumulator holding the value $5C_H$ (01011100_B).
Instruction:	
SWAP A	Operation: (A3-0) ↔ (A7-4) Bytes: 1 Cycles: 1 Encoding: 

Software Break

TRAP

Table 4-53 TRAP

Description:	Assert a software break. Enters debug mode at the end of phase 1 of the machine cycle. No flags are affected.
Example:	If EO.TRAP_EN = 1, opcode $A5_H$ is a TRAP instruction. MOV A, #55H TRAP ; break INC A
Instruction:	
TRAP	Operation: Break Bytes: 1 Cycles: 1 Encoding: 1 0 1 0 | 0 1 0 1

Note: This instruction is XC800-specific, therefore may not be supported by standard 8051 assembler. In such cases, this can be workaround by direct byte declaration and definition e.g. ".byte #$A5_H$" (syntax is assembler dependent).

Note: This instruction shares the same opcode with another XC800-specific instruction MOVC @(DPTR++),A. TRAP is selected only if EO.TRAP_EN = 1.

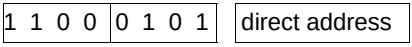
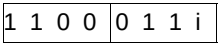
Exchange Accumulator with Byte

XCH A, <byte>

Table 4-54 XCH

Description:	XCH loads the accumulator with the contents of the indicated variable, at the same time writing the original accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.
Example:	R0 contains the address 20_H. The accumulator holds the value $3F_H$ (00111111_B). Internal RAM location 20_H holds the value 75_H (01110101_B). The instruction XCH A, @R0 will leave RAM location 20_H holding the value $3F_H$ (00111111_B) and 75_H (01110101_B) in the accumulator.
Instruction:	
XCH A, Rn	Operation: (A) ↔ (Rn) Bytes: 1 Cycles: 1 Encoding: 1 1 0 0 | 1 r r r

Table 4-54 XCH (cont'd)

XCH A, direct	Operation: (A) ↔ direct Bytes: 2 Cycles: 1 Encoding: 
XCH A, @Ri	Operation: (A) ↔ ((Ri)) Bytes: 1 Cycles: 1 Encoding: 

Exchange Digit

XCHD A,@Ri

Table 4-55 XCHD

Description:	XCHD exchanges the low-order nibble of the accumulator (bits 3-0, generally representing a hexadecimal or BCD digit), with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.
Example:	R0 contains the address 20_H. The accumulator holds the value 36_H (00110110_B). Internal RAM location 20_H holds the value 75_H (01110101_B). The instruction XCHD A, @ R0 will leave RAM location 20_H holding the value 76_H (01110110_B) and 35_H (00110101_B) in the accumulator.
Instruction:	
XCHD A, @Ri	Operation: (A3-0) $\leftrightarrows$ ((Ri)3-0) Bytes: 1 Cycles: 1 Encoding: 1 1 0 1 | 0 1 1 i

Logical Byte Exclusive OR

XRL <dest-byte>, <src-byte>

Table 4-56 XRL

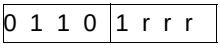
Description:	XRL performs the bitwise logical Exclusive OR operation between the indicated variables, storing the results in the destination. No flags are affected (except P, if <dest-byte> = A). The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be accumulator or immediate data. *Note: When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.*
Example:	If the accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010B) then the instruction XRL A,R0 will leave the accumulator holding the value 69_H (01101001_B). When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the accumulator at run-time. The instruction XRL P1,#00110001B will complement bits 5, 4, and 0 of output port 1.
Instruction:	
XRL A, Rn	Operation: (A) ←(A) ∀(Rn) Bytes: 1 Cycles: 1 Encoding: 

Table 4-56 XRL (cont'd)

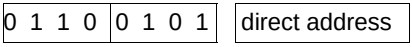
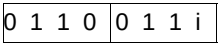
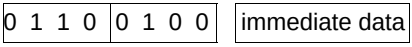
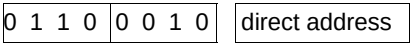
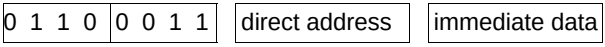
XRL A, direct	Operation: (A) ←(A) ∀(direct) Bytes: 2 Cycles: 1 Encoding: 
XRL A, @Ri	Operation: (A) ←(A) ∀((Ri)) Bytes: 1 Cycles: 1 Encoding: 
XRL A, #data	Operation: (A) ←(A) ∀#data Bytes: 2 Cycles: 1 Encoding: 

Table 4-56 XRL (cont'd)

XRL direct, A	Operation: (direct) ←(direct) ∀(A) Bytes: 2 Cycles: 1 Encoding: 
XRL direct, #data	Operation: (direct) ←(direct) ∀#data Bytes: 3 Cycles: 2 Encoding: 

5 Index

5.1 Keyword Index

This section lists a number of keywords which refer to specific details of the XC800 in terms of its architecture, its functional units, or functions.

w w w . i n f i n e o n . c o m

Published by Infineon Technologies AG