51MX Architecture Reference Specification and Instruction Set

Preliminary **July 19, 2002**

Version 0.9

Philips Semiconductors

PHILIPS

1 INTRODUCTION TO THE 51MX ARCHITECTURE

The 51MX is a true binary-level superset of the classic 80C51 architecture, transparently adding access to both code and data memory beyond the original 64 kilobyte (KB) limit, up to the maximum of 8 megabytes (MB) of code and 8 MB minus 64 KB of data, in a non-segmented linear space. The 51MX is completely backward compatible with the 80C51. Code written for the 80C51 may be run on the 51MX with no changes.

1.1 KEY 51MX ARCHITECTURE ENHANCEMENTS

The 51MX architecture provides a seamless and compelling upgrade path from the classic 80C51 by offering easy memory extension combined with enhanced high level language performance and efficiency.

- Program Counter: The Program Counter is extended to 23 bits. Code size up to 8 MB is accessed in linear, non-segmented approach, eliminating inefficient bank-switching solutions when crossing 64 KB boundary.
- Extended Data Pointer: A 23-bit Extended Data Pointer EPTR is added in order to allow simple extension to existing assembly language programs that must be expanded to address more than 64 KB of data memory.
- Stack: Two independent alternate Stack modes are added. The first causes addresses pushed onto the Stack by interrupts to be expanded to 23 bits. The second allows Stack extension into a larger memory space.
- Instruction set: A small number of instructions now have extended addressing modes to allow full use of extended code and data addressing.
- Addressing Modes: A new addressing mode, Universal Pointer mode, is added so that any area of code and data can be accessed with a single instruction. Using this feature, high level languages compilers, such as C, can easily produce more efficient code, both in terms of size and execution time.
- Extended SFR space: additional 128 bytes of Special Function Register addressing allows easy access to the broad range of advanced 51MX peripherals.

1.2 COMPATIBILITY CONSIDERATIONS WITH THE 80C51

An equally important objective for the 51MX is to retain complete compatibility with the familiar classic 80C51. The 80C51 is the most widely used 8-bit microcontroller architecture in the world and as a result, a vast amounts of code have been written. Therefore preservation of customer's investment in code and hardware development is a critical consideration.

- Instruction set: The 51MX implements the entire 80C51 instruction set with identical instruction encoding.
- CPU: The 51MX adds a small number of register and instruction extensions needed for extended addressing.
- Timing: The 51MX has standard 6-clock 80C51 machine cycle timing. If the 51MX replaces an older 80C51 device using 12-clock machine cycles, it will run exactly twice as fast as the original device.
- Memory Map: A major consideration in hardware compatibility of the 51MX with the 80C51 is the memory map. The 51MX implements the entire 80C51 memory map, adding to it in a transparent manner where necessary to extend the address space to 8 MB each of code and data.
- Stack operation: The 51MX duplicates 80C51 stack operations precisely as its default configuration. The 51MX can be configured to extend the stack by increasing the Stack Pointer to 16-bits and allowing it to access the Extended Data memory. When the program size expands beyond 64 KB, the extended interrupt frame mode saves and restores the full 23-bit Program Counter.
- SFR Access: Where 51MX devices implement the same feature set as an earlier 80C51 device, SFR addresses will remain unchanged such that code from the original device will run with no changes. If the 51MX device has additional features, it may not always be possible to use the same SFR addresses for all features.
- Pin-for-pin compatibility. 51MX derivatives are generally intended to be pin-compatible with 80C51 derivatives that have the same feature set. When extended addressing is enabled, additional address lines may replace some port functions.
- Bus Interface: The 51MX external bus operates in the same manner as a classic 80C51. Extended addressing does not take effect unless it is enabled by configuration or by software, so the additional address lines remain inactive unless needed, allowing the pins to be used for other purposes, such as general purpose I/O.

2 MEMORY ORGANIZATION

2.1 51MX VERSUS 80C51 PROGRAMMER'S MODELS AND MEMORY MAPS

The 51MX retains all of the 80C51 memory spaces. Additional memory space has been added transparently as part of the means for allowing extended addressing. The basic memory spaces include code memory (which may be on-chip, off-chip, or both); external data memory (a portion of which is always on-chip); Special Function Registers; and internal data memory, which includes on-chip RAM, registers, and stack. Provision is made for internal data memory to be extended, allowing a larger processor stack.

The classic 80C51 programmer's model and memory map is shown in Figure 1 for reference. The 51MX programmer's model and memory map is shown in Figure 2.

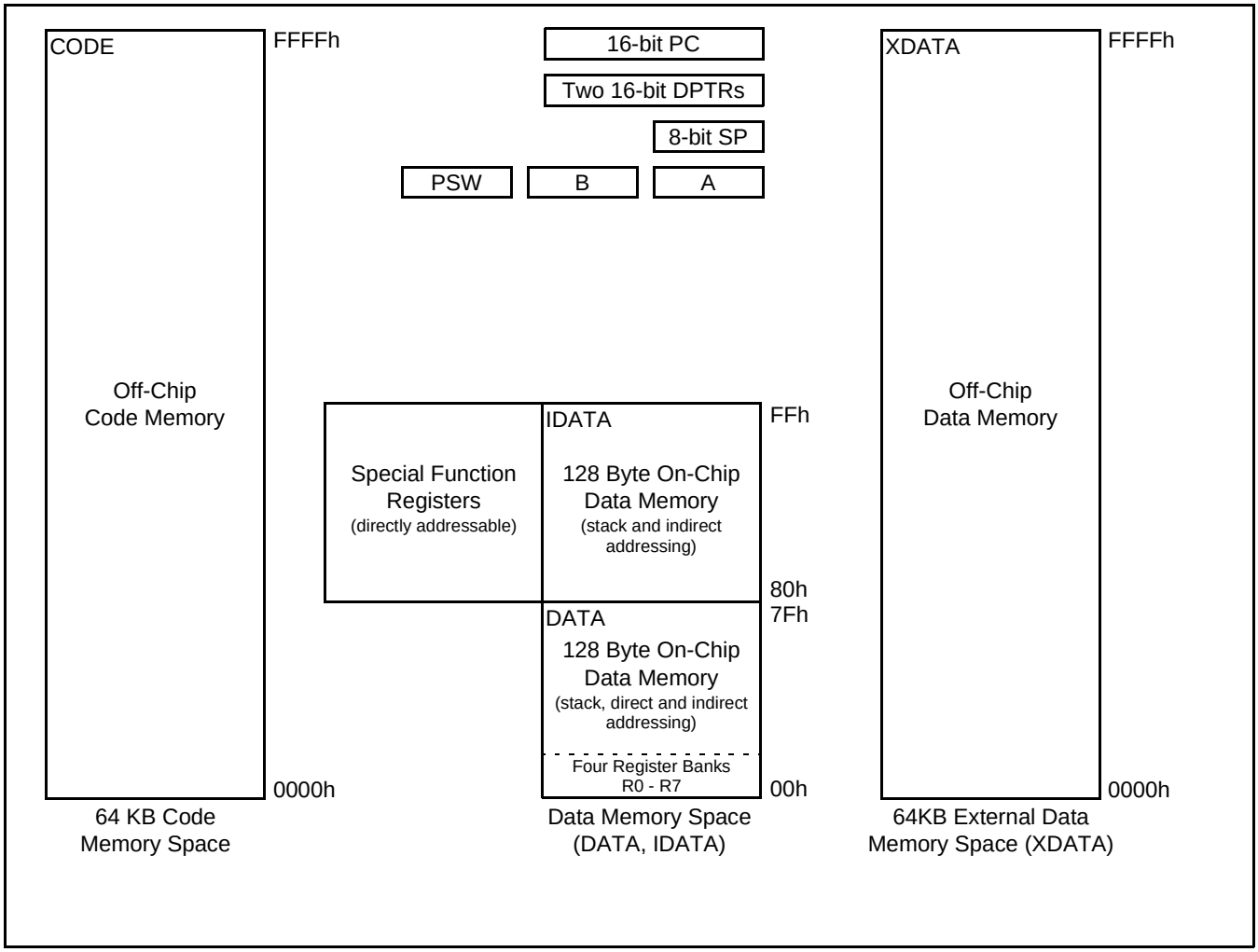


Figure 1: Classic 80C51 Architecture Programmer's Model and Memory Map

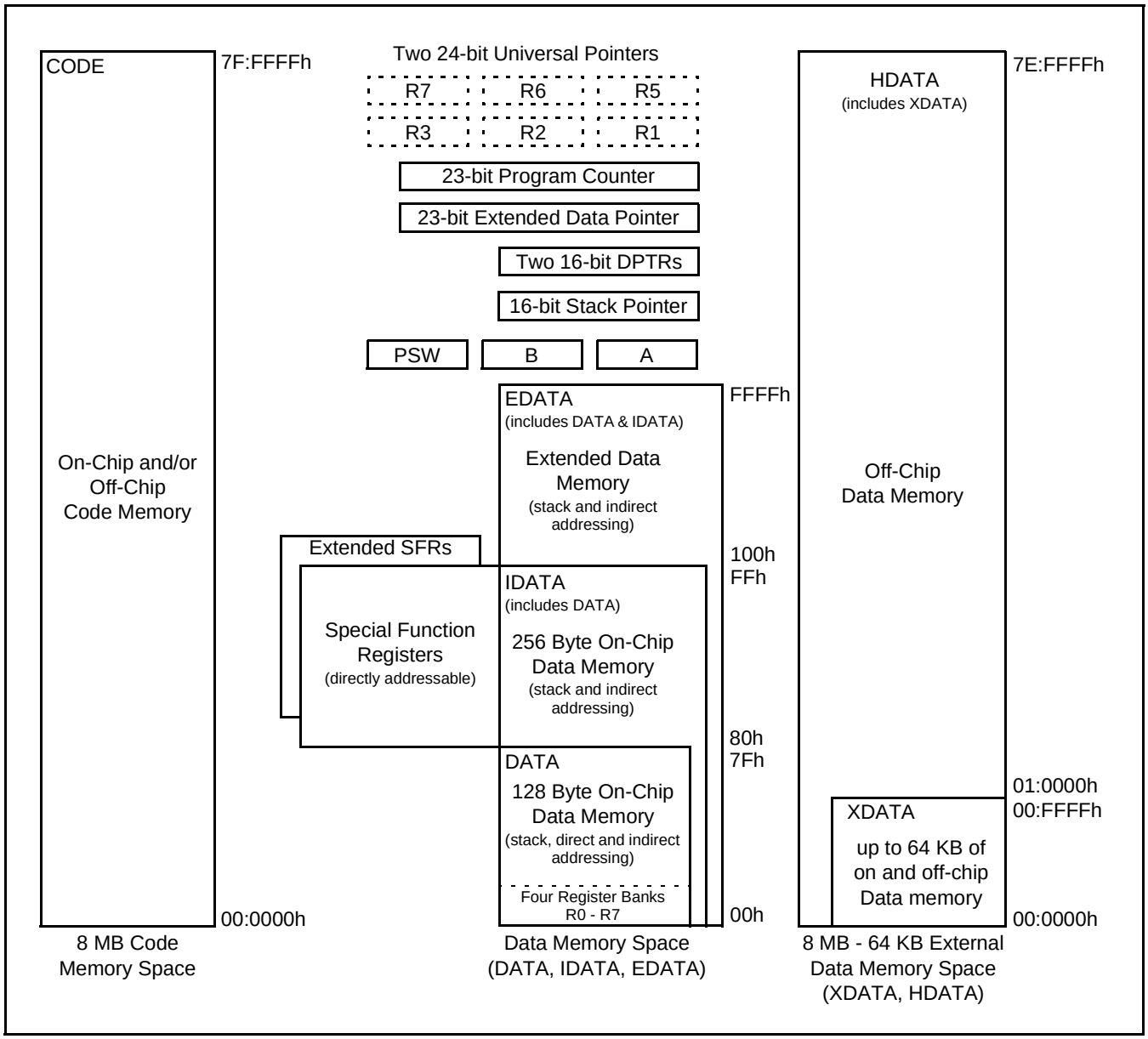


Figure 2: 51MX Programmer's Model and Memory Map

Detailed descriptions of each of the various 51MX memory spaces may be found following this summary.

DATA 128 bytes of internal data memory space (00h...7Fh) accessed via direct or indirect addressing, using instructions other than MOVX and MOVC. All or part of the Stack may be in this area.

IDATA Indirect Data. 256 bytes of internal data memory space (00h...FFh) accessed via indirect addressing using instructions other than MOVX and MOVC. All or part of the Stack may be in this area. This area includes the DATA area and the 128 bytes immediately above it.

EDATA Extended Data. This space is a superset of DATA and IDATA areas and allows up to a potential total of 64 KB (including those areas). The added area may be accessed only as Stack and via indirect addressing using Universal Pointers. Part of this space is always implemented on-chip.

SFR Special Function Registers. Selected CPU registers and peripheral control and status registers, accessible only via direct addressing (addresses in range 80h...FFh). This includes the new 51MX extended SFRs.

XDATA "External" Data. Duplicates the classic 80C51 64 KB memory space addressed via the MOVX instruction using the DPTR, R0, or R1. Part of this space is implemented on-chip. On-chip XDATA can be disabled under program control. Also, XDATA may be placed in external devices.

HDATA "High" Data. This is a superset of XDATA and may include up to 8,323,072 bytes (8 MB - 64 KB) of memory space addressed via the MOVX instruction using the EPTR, DPTR, R0, or R1. Non XDATA portion of HDATA is placed in external devices.

CODE Up to 8 MB of Code memory, accessed as part of program execution and via the MOVC instruction.

All of these spaces except the SFR space may also be accessed through use of Universal Pointer addressing with the EMOV instruction. This feature is detailed in a subsequent section.

2.2 DATA MEMORY (DATA, IDATA, AND EDATA)

The standard 80C51 internal data memory consists of 256 bytes of DATA/IDATA RAM, and is always entirely on-chip. In this space are the data registers R0 through R7, the default stack, a bit addressable RAM area, and general purpose data RAM. On the top of the DATA/IDATA memory space is the rest of Extended Data Memory (EDATA) that provides up to a possible 64 KB of total EDATA memory. Stack by default resides in DATA/IDATA space, but may be configured to extend into the rest of EDATA. The different portions of the data memory are accessed in different manners as described in the following sections.

2.2.1 REGISTERS R0 - R7

General purpose registers R0 through R7 allow quick, efficient access to a small number of internal data memory locations. For example, the instruction:

```
MOV  A,R0
```

uses one byte of code and executes in one machine cycle. Using direct addressing to accomplish the same result as in:

```
MOV  A,10h
```

requires two bytes of code memory and executes in two machine cycles. Indirect addressing further requires setup of the pointer register, etc.

These registers are “banked”. There are four groups of registers, any one of which may be selected to represent R0 through R7 at any particular time. This feature may be used to minimize the time required for context switching during an interrupt service or a subroutine, or to provide more register space for complicated algorithms.

The registers are no different from other internal data memory locations except that they can be addressed in "shorthand" notation as "R0", "R1", etc. Instructions addressing the internal data memory by other means, such as direct or indirect addressing, are quite capable of accessing the same physical locations as the registers in any of the four banks.

2.2.2 BIT ADDRESSABLE RAM

Internal data memory locations 20h through 2Fh may be accessed as both bytes and bits. This allows a convenient and efficient way to manipulate individual flag bits without using much memory space. The bottom bit of the byte at address 20h is bit number 00h, the next bit in the same byte is bit number 01h, etc. The final bit, bit 7 of the byte at address 2Fh, is bit number 7Fh (127 decimal). Bit numbers above this refer to bits in Special Function Registers.

This code:

```
SETB    20h.1
CPL     20h.2
JNB     20h.2, LABEL1
```

sets bit 1 at address 20h, complements bit 2 in the same byte, then branches if the second bit is not equal to 1. In an actual program, these bits would normally be given names and referred to by those names in the bit manipulation instructions.

2.2.3 EXTENDED DATA MEMORY (EDATA)

The 51MX architecture allows for extension of the internal data memory space beyond the traditional 256 byte limit of classic 80C51s. This space can be used as an extended or alternative processor stack space, or can be used as general purpose storage under program control. Other than Stack Pointer based access to this space, which is automatic if Extended Stack Memory Mode is enabled (ESMM=1 in MXCON sfr; see the following Stack section), this memory is addressed only using the new Universal Pointer feature. Universal Pointers are described in a later section.

2.2.4 STACK

The processor stack provides a means to store interrupt and subroutine return addresses, as well as temporary data. The stack grows upwards, from lower addresses towards higher addresses. The current Stack Pointer always points to the last item pushed on the stack, unless the stack is empty. Prior to a push operation, the Stack Pointer is incremented, then data is written to memory. When the stack is popped, the reverse procedure is used. First, data is read from memory, then the Stack Pointer is decremented.

The default configuration of the 51MX stack is identical to the classic 80C51 stack implementation. When interrupt or subroutine addresses are pushed onto the stack, only the lower 16 bits of the Program Counter are stored. This default 80C51 mode stack operation is shown in Figure 3.

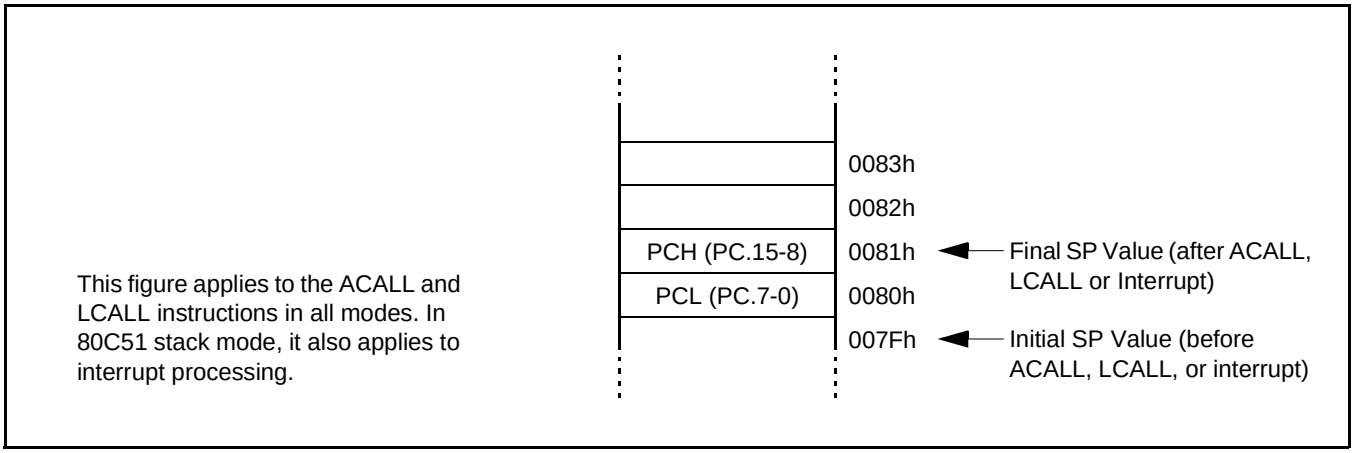


Figure 3: Return Address Storage on the Stack (80C51 Mode)

There are two configuration options for the stack. For purposes of backward compatibility with the classic 80C51, both alternate modes are disabled by a chip reset. The first option, Extended Interrupt Frame Mode, causes interrupts to push the entire 23-bit

Program Counter onto the stack (as three bytes), and the RETI instruction to pop all 23-bits as a return address, as shown in Figure 4. The upper bit of the stack byte containing the most significant byte of the Program Counter is forced to a "1" to be consistent with Universal Pointer addressing.

Storing the full 23-bit Program Counter value is a requirement for systems that include more than 64 KB of program, since an interrupt could occur at any point in the program. The Extended Interrupt Frame Mode changes the operation of interrupts and the RETI instruction only, other calls and returns are not affected. Special extended call and return instructions allow large programs to traverse the entire code space with full 23-bit return addresses. The Extended Interrupt Frame Mode is enabled by setting the EIFM bit in the MXCON register.

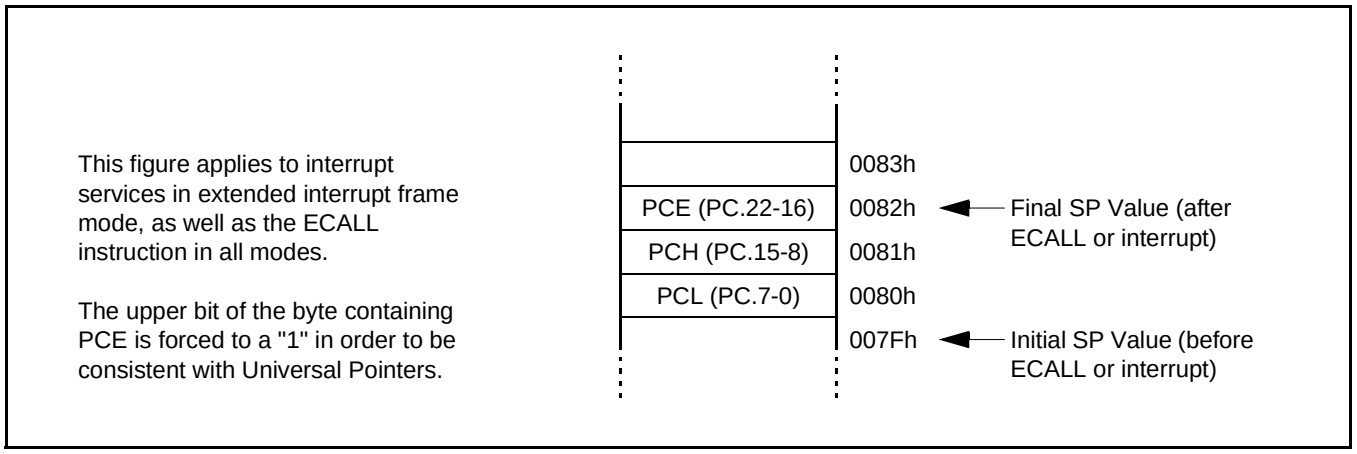


Figure 4: Extended Return Address Storage on the Stack

The second stack option, Extended Stack Memory Mode, allows for stack extension beyond the 256 byte limit of the classic 80C51 family. Stack extension is accomplished by increasing the Stack Pointer to 16 bits in size and allowing it to address the entire EDATA memory rather than just the standard 256 byte internal data memory. Stack extension has no effect on the data that is stored on the stack, it will continue to be stored as shown on in Figure 3 and Figure 4. The Extended Stack Memory Mode is enabled by setting the ESMM bit in the MXCON register.

If the Stack Pointer is not initialized by software, the stack will begin at on-chip RAM address 8, just as for the 80C51. Also note that in Extended Stack Memory Mode, the size of the EDATA space (and therefore its ending address) may not be the same for all 51MX systems, it will depend on the size of the EDATA RAM physically implemented for a specific 51MX derivative.

The stack mode bits ESMM and EIFM are shown in Figure 5. Note that the stack mode bits are intended to be set once during program initialization and not altered after that point. Changing stack modes dynamically may cause stack synchronization problems.

MXCON Address: FFh (51MX Extended SFR Space)

Not bit addressable

Reset Value: 00h

7	6	5	4	3	2	1	0
-	-	-	-	-	EAM	ESMM	EIFM

BIT	SYMBOL	FUNCTION
MXCON.7 - 3	-	Reserved. Programs should not write a 1 to these bits.
MXCON.2	EAM	Enables Extended Addressing Mode, in connection with a non-volatile user configuration bit. The logical OR of the SFR bit and the non-volatile configuration bit determines whether code and data addressing beyond 64 KB is allowed. The same logical OR value will be read from this bit by software. When 0, all addressing (on-chip and off-chip) is limited to 64 KB each of code and data. When 1, 51MX addressing capabilities are extended beyond boundary of 64 KB to 8 MB each of code and data, and upper address bits are multiplexed on Port 2 for external code and/or data accesses. Refer to the External Bus section for additional details. EAM must be set to EAM=1 if at least one of the next two statements is true: - there is executable code or constants in CODE space are above 64 KB - address of data byte that has to be accessed in HDATA is above 64 KB
MXCON.1	ESMM	Enables the Extended Stack Memory Mode. When ESMM = 0, the Stack Pointer is 8 bits in width and the stack is located in the IDATA memory space. When ESMM = 1, the Stack Pointer is increased to 16-bits in width and the stack may be located anywhere in the EDATA space. ESMM is independent of EAM and EIFM bits.
MXCON.0	EIFM	Enables the Extended Interrupt Frame Mode. When EIFM = 0, an interrupt service will cause only the lower 16 bits of the PC to be pushed onto the stack, and a RETI instruction will restore only the lower 16 bits of the PC. When EIFM = 1, an interrupt service will cause all 23 bits of the PC to be pushed onto the stack, while a RETI instruction will restore all 23 bits of the PC. EIFM must be set to one if the application allows execution beyond the first 64 KB of code memory.

Figure 5: 51MX Configuration Register (MXCON)

2.2.5 GENERAL PURPOSE RAM

Portions of the internal data memory that are not used in a particular application as registers, stack, or bit addressable locations may be considered general purpose RAM and used in any desired manner.

The lower 128 bytes of the internal data memory (DATA) may be accessed using either direct or indirect addressing. Direct addressing incorporates the entire address within the instruction. For example, the instruction:

```
MOV     31h,#10
```

will store the value 10 (decimal) in location 31h. Direct addresses above 128 will access the Special Function Registers rather than the internal data memory.

Indirect addressing takes an address from either R0 or R1 of the current register bank and uses it to identify a location in the internal data memory. The entire 256 byte internal data memory space (IDATA) may be accessed using indirect addressing. For example, the instruction sequence:

```
MOV     R0,#90h
MOV     A,@R0
```

will cause the contents of location 90 hex to be loaded into the accumulator. It is typical with the classic 80C51 to cause the stack to be located in the upper area, leaving more general purpose RAM in the lower area that may be accessed using both direct and indirect addressing. With the 51MX, the stack may be extended and moved completely out of the lower 256 bytes of memory.

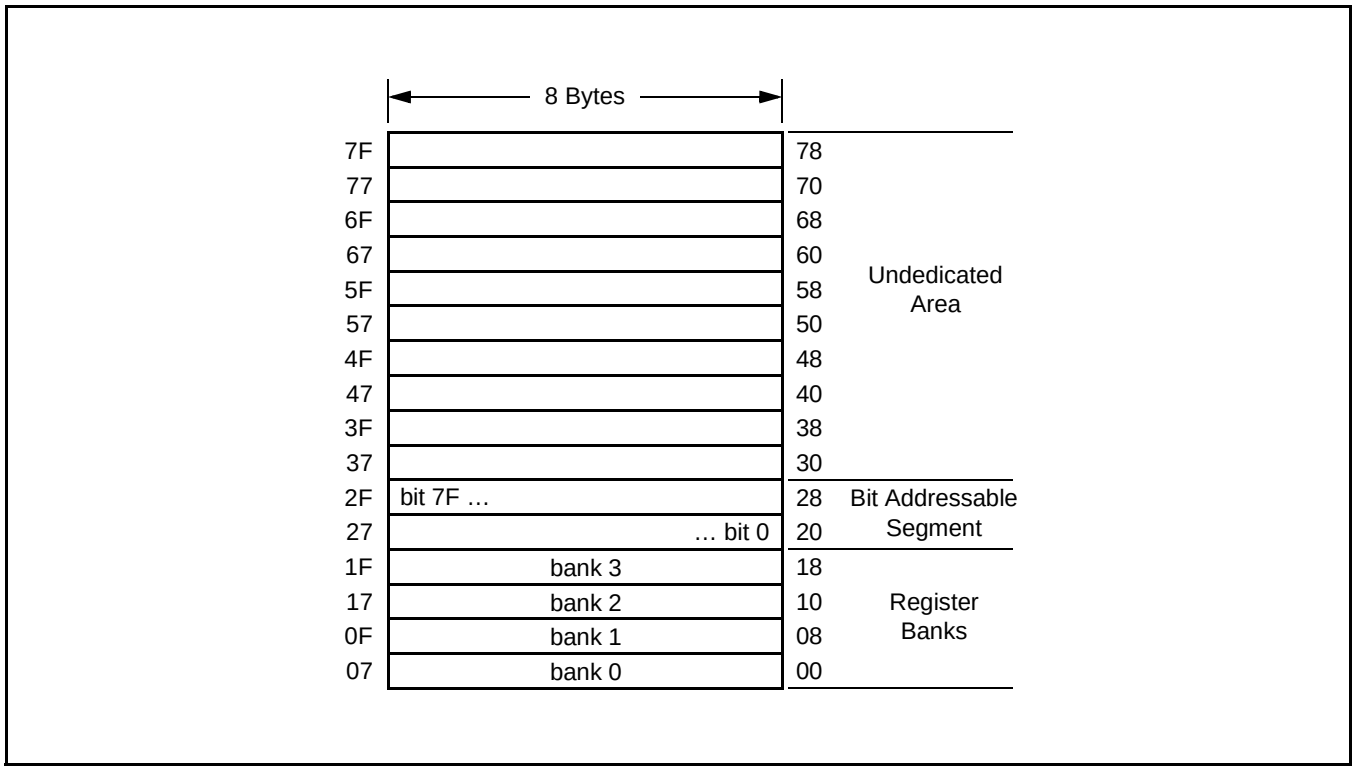


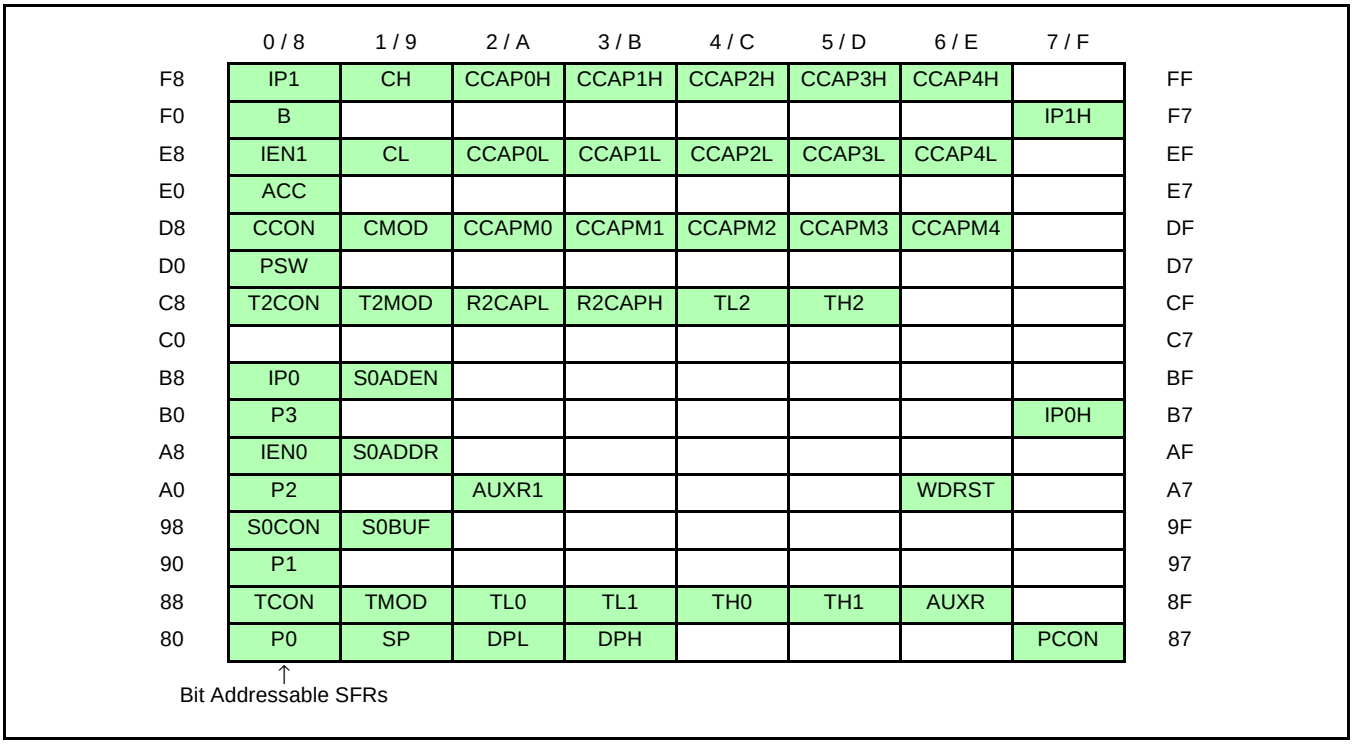
Figure 6: Internal Data Memory, Lower 128 Bytes

2.3 SPECIAL FUNCTION REGISTERS (SFRS)

Special Function Registers (SFRs) provide a means for the processor to access internal control registers, peripheral devices, and I/O ports. An SFR address is always contained entirely within an instruction.

The standard SFR space is 128 bytes in size. SFRs are implemented in each 51MX device as needed in order to provide control for peripherals or access to CPU features and functions. Each 51MX derivative may have a different number of SFRs implemented because each has a different set of peripheral functions. Many SFR addresses will typically be unused on any particular derivative. Those undefined SFRs are considered "reserved" and should not be accessed by user programs. However, if these bits are accessed, they should be filled with "0"s, in order to maintain the code compatibility with future parts.

Sixteen addresses in the SFR space are both byte- and bit-addressable. The bit-addressable SFRs are those whose address ends in 0h or 8h (i.e. 80h, 88h, ..., F8h). Bit addressing allows direct control and testing of bits in those SFRs. Figure 7 shows the SFR map for the first 51MX devices, the 87MC51Mx2 family (87C51MB2, and 87C51MC2).



	0 / 8	1 / 9	2 / A	3 / B	4 / C	5 / D	6 / E	7 / F	
F8	IP1	CH	CCAP0H	CCAP1H	CCAP2H	CCAP3H	CCAP4H		FF
F0	B							IP1H	F7
E8	IEN1	CL	CCAP0L	CCAP1L	CCAP2L	CCAP3L	CCAP4L		EF
E0	ACC								E7
D8	CCON	CMOD	CCAPM0	CCAPM1	CCAPM2	CCAPM3	CCAPM4		DF
D0	PSW								D7
C8	T2CON	T2MOD	R2CAPL	R2CAPH	TL2	TH2			CF
C0									C7
B8	IP0	S0ADEN							BF
B0	P3							IP0H	B7
A8	IEN0	S0ADDR							AF
A0	P2		AUXR1				WDRST		A7
98	S0CON	S0BUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1	AUXR		8F
80	P0	SP	DPL	DPH				PCON	87

Bit Addressable SFRs

Figure 7: Standard SFR Map for the 87C51Mx2

SFR addresses tend to be in very short supply in more complex 80C51 derivatives, especially in the case of bit addressable SFRs. Therefore, a method to extend the SFR space is implemented on the 51MX. When any instruction that references an SFR (including a bit address in an SFR) is preceded by the escape opcode A5h, an alternate bank of SFRs (extended SFRs) will be accessed. Thus, extended SFRs take one machine cycle longer to access and use one additional byte of memory to encode than standard SFRs.

Due to the method by which extended SFRs are accessed, each 51MX instruction may access only standard SFRs or extended SFRs with explicit addressing. This is only an issue for the instruction "MOV direct,direct". When used to access two SFRs, the 51MX restricts this instruction to access only SFRs in the same space (standard SFR space or extended SFR space). Note that instructions such as "MOV A,direct" refer to the Accumulator *implicitly* (the address of the Accumulator is not encoded in the instruction) and can therefore be used with either SFR bank. The complete list of instructions that may access a byte or bit in both a standard and extended SFR (by addressing one implicitly and one explicitly) follows:

```
ADD    A,direct
ADDC   A,direct
SUBB   A,direct
ANL    A,direct
ANL    direct,A
ORL    A,direct
ORL    direct,A
XRL    A,direct
XRL    direct,A
MOV    A,direct
MOV    direct,A
XCH    A,direct
ANL    C,bit
ANL    C,/bit
ORL    C,bit
ORL    C,/bit
```

```
MOV    C,bit
MOV    bit,C
CJNE   A,direct,rel
```

Several SFRs have been defined that will be present on any 51MX derivative. These are: MXCON, EPH, EPM, EPL, and SPE. MXCON contains the bits that control 51MX specific functions, including the Stack Mode control bits and the External Bus Mode control bit. EPH, EPM, and EPL comprise the EPTR. SPE contains the upper byte of the Stack Pointer when the stack is in Extended Stack Memory Mode. All of these SFRs are located in the extended SFR bank. Figure 8 shows the extended SFR map for the 87C51Mx2.

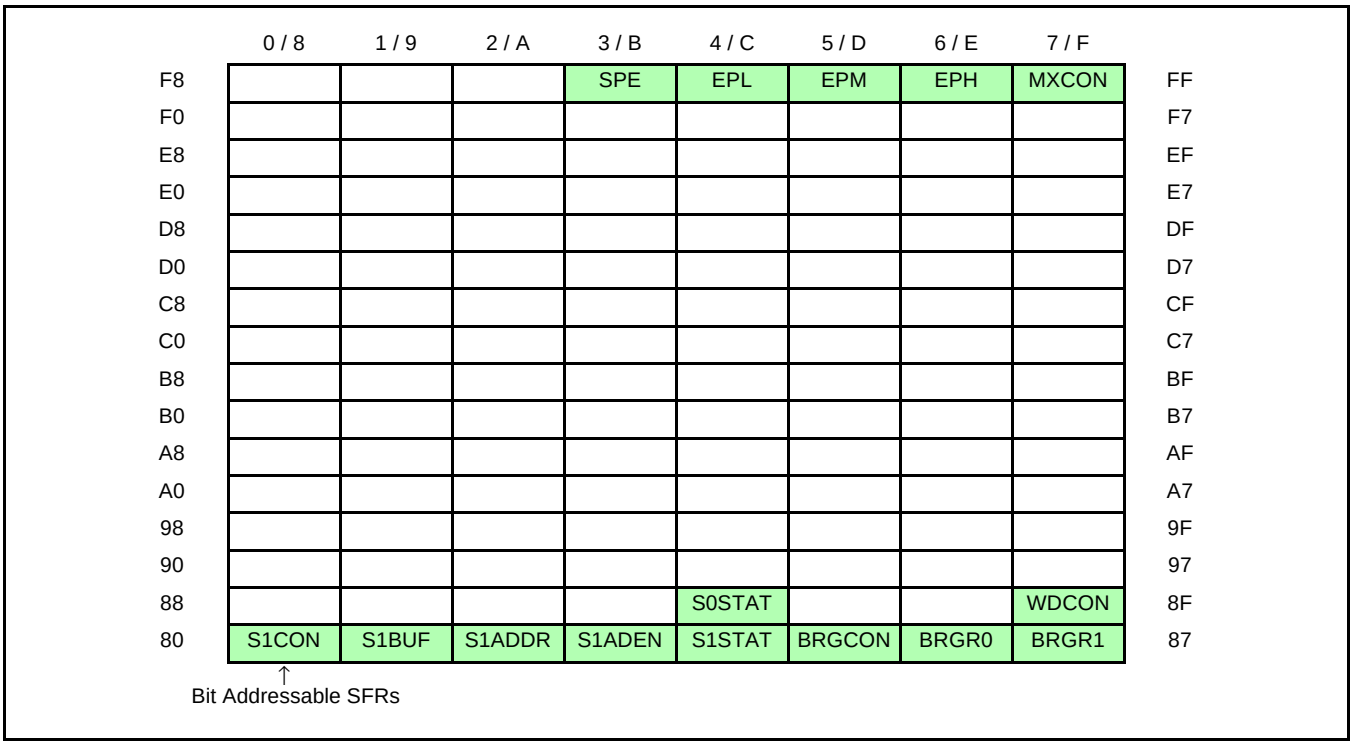


Figure 8: Extended SFRs Map for the 87C51Mx2 (First 51MX Derivative)

2.4 EXTERNAL DATA MEMORY (XDATA)

The XDATA space on the 51MX is the same as the 64 KB external data memory space on the classic 80C51.

On-chip XDATA memory can be disabled under program control via the EXTRAM bit in the AUXR register. Accesses above implemented on-chip XDATA will be routed to the external bus. If on-chip XDATA memory is disabled, all XDATA accesses will be routed to the external bus.

2.5 HIGH DATA MEMORY (HDATA)

The 51MX architecture supports up to an 8 MB data memory space, using 23-bit addressing. The entire 8 MB space except for the 64 KB EDATA space is called HDATA. The XDATA space comprises the lower 64 KB of HDATA.

Data Pointers

The 51MX adds an additional 23-bit Extended Data Pointer (EPTR) in order to allow a simple method of extending existing 80C51 programs to use more than 64 KB of data memory. To access a single data byte from HDATA RAM located above the first 64 KB, the EAM bit in MXCON must be set to EAM=1.

All 80C51 instructions that use the DPTR have an 51MX variant that uses the EPTR. The 23-bit EPTR is comprised of (in order) EPH, EPM, and EPL. Figures 9 and 10 show examples of indirect accesses to data memory using the DPTR and the EPTR respectively. Since the EPTR is a 23-bit value, the 8th bit of EPH is not used. If read, it will return a 1. Use of the EPTR allows access to the entire HDATA space, including XDATA.

At any point in time, one specific Data Pointer is active and used by instructions that reference the DPTR. The active DPTR may be changed by altering the Data Pointer Select (DPS) bit. The DPS bit occupies the bottom bit of the AUXR1 register. The DPS bit applies only to the two DPTRs, not to the EPTR.

In the indirect addressing mode, the currently active DPTR or the EPTR provides a data memory address for accessing the XDATA and HDATA space respectively. When the DPTR is used for addressing, only the XDATA space is available. When the EPTR is used for addressing, the entire HDATA space (which includes the XDATA space) may be accessed. If the EPTR value exceeds 7E:FFFF (the limit of HDATA), data accesses using EPTR will yield undefined results. The reason for limiting HDATA addresses is to keep the addressing uniform for EPTR addressing and Universal Pointer addressing (which is explained in a later section of this document).

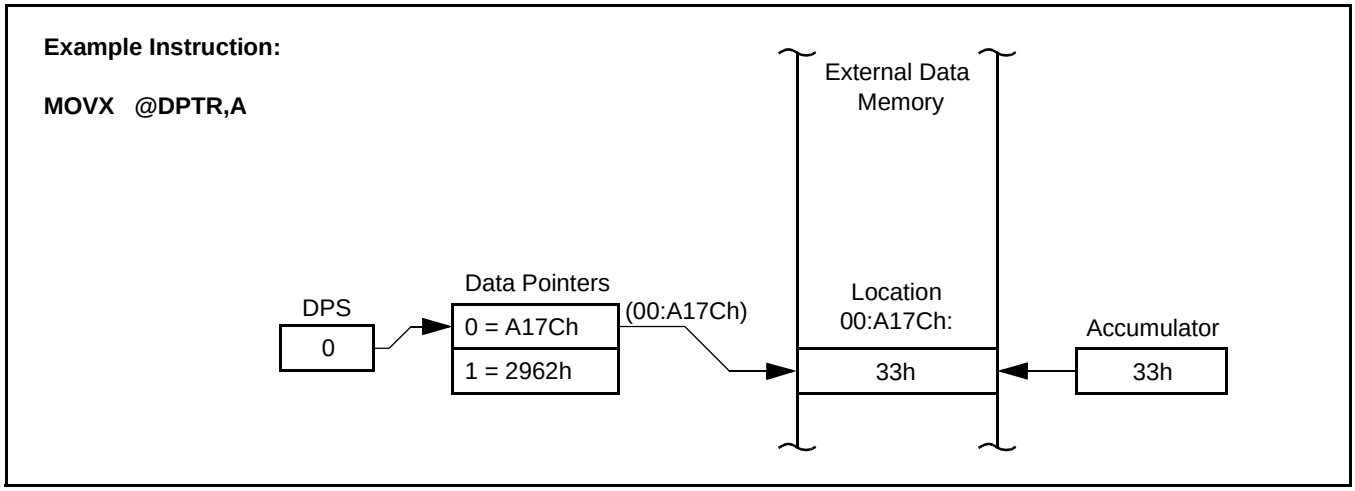


Figure 9: External Data Memory Access using Indirect Addressing with DPTR

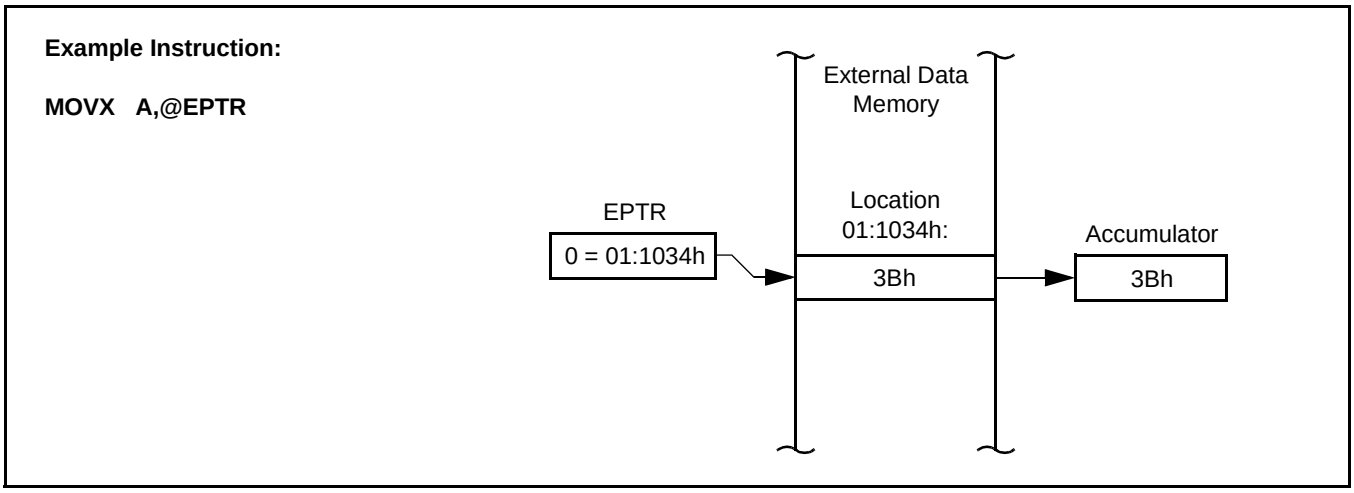


Figure 10: External Data Memory Access using Indirect Addressing with EPTR

2.6 PROGRAM MEMORY (CODE)

The 80C51, and thus the 51MX, are "Harvard" architectures, meaning that the code and data spaces are separated. If there is any portion of executable code above 64 KB that can be interrupted, EAM bit in MXCON must be set to EAM=1. Also, if there is even the slightest amount of constants used by the application and placed in CODE space above 64 KB boundary, EAM must be set to EAM=1.

The 51MX expands the 80C51 Program Counter to 23 bits, providing a continuous, unsegmented linear code space that may be as large as 8 MB. On-chip space begins at code address 0 and extends to the limit of the on-chip code memory. Above that, code will be fetched from external. The 51MX architecture allows for an external bus which supports:

- Mixed mode (some code and/or data memory off-chip).
- Single-chip operation (no external bus connection).
- ROMless operation (no use of on-chip code memory).

In some cases, code memory may be addressed as data. Extended instruction address modes provide access to the entire code space of 8 MB through the use of indexed indirect addressing. The currently active DPTR, the EPTR, a Universal Pointer, or the Program Counter may be used as the base address. Examples of the various code memory addressing modes are shown in figures 11 through 13.

Following a reset, the 51MX begins code execution like a classic 80C51, at address 00:0000h. Similarly, the interrupt vectors are placed just above the reset address, starting at address 00:0003h. It is important to note that the first instruction (located at address 0) must not be an EJMP instruction. EJMP is a 5 byte instruction and would overlap any instructions intended for the external interrupt 0 vector address.

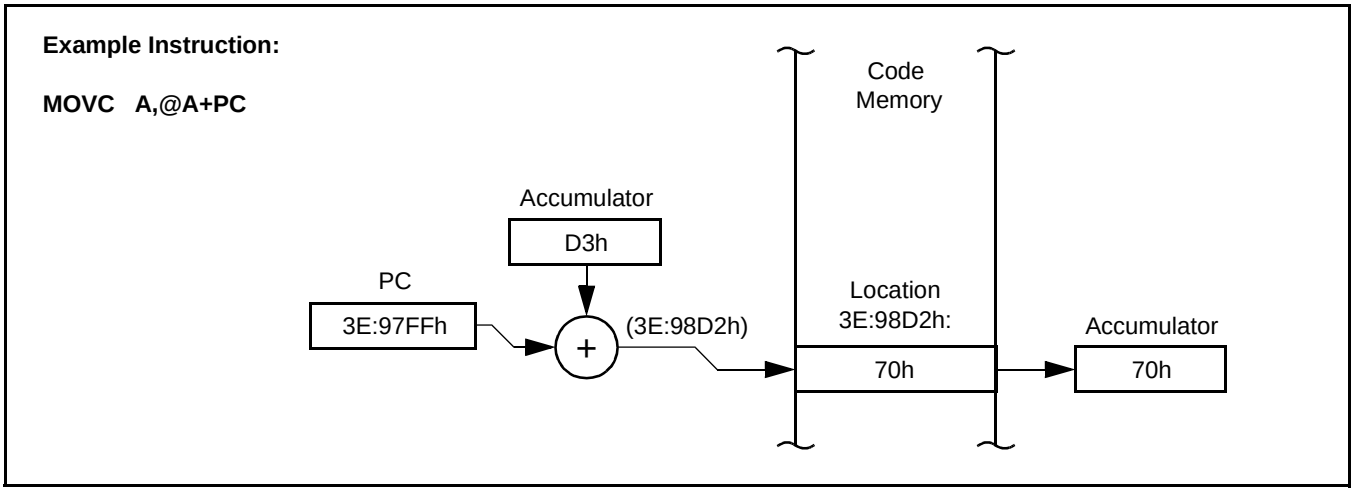


Figure 11: Code Memory Access using Indexed Indirect Addressing with the Program Counter

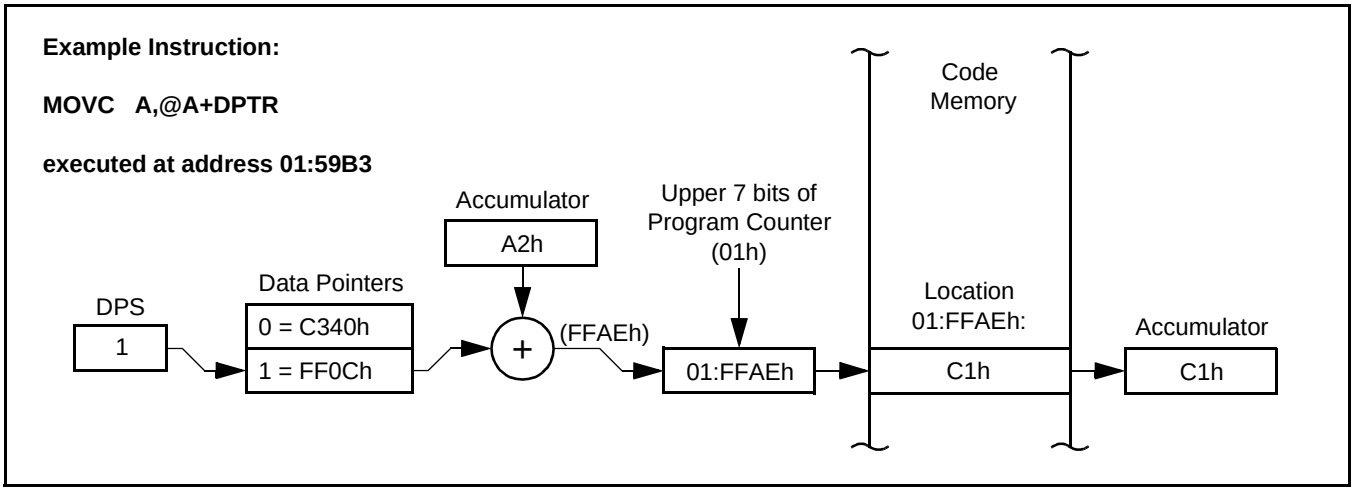


Figure 12: Code Memory Access using Indexed Indirect Addressing with DPTR

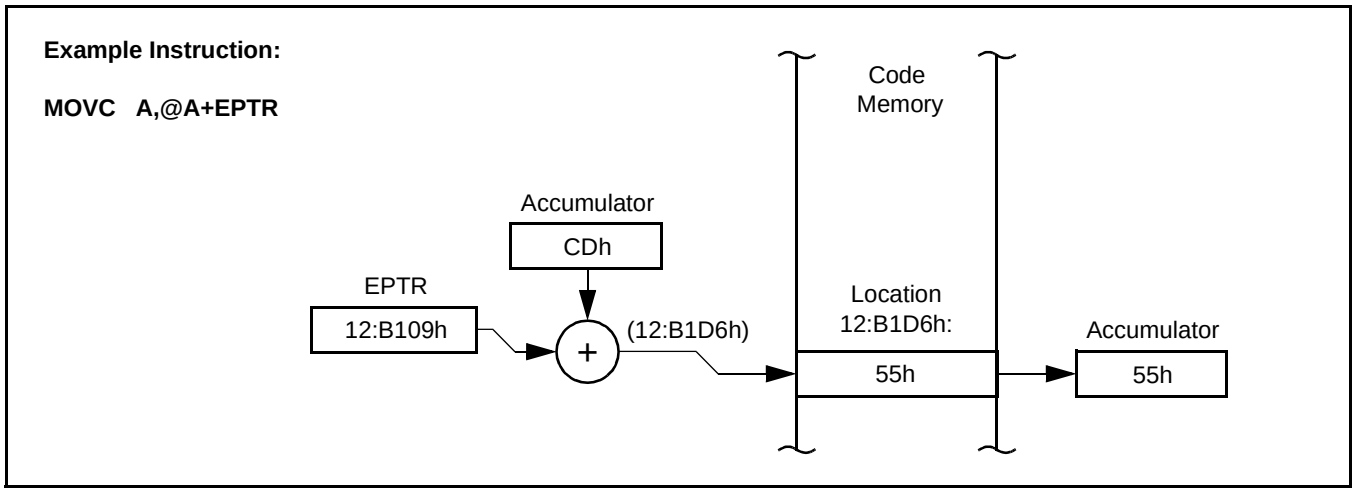


Figure 13: Code Memory Access using Indexed Indirect Addressing with EPTR

2.7 UNIVERSAL POINTERS

A new addressing mode called Universal Pointer mode has been added to the 51MX, specifically for the purpose of greatly enhancing C language code density and performance. This addressing mode allows access to any of the on-chip or off-chip code and data spaces using one instruction, without the need to know in advance which of the different spaces the data will reside in. This includes the DATA, IDATA, EDATA, XDATA, HDATA, and CODE spaces. The SFR space is the only space that may not be accessed using the Universal Pointer mode.

The Universal Pointer addressing mode uses a new set of pointer registers for two reasons. The first is that 24-bit pointers are needed in order to allow addressing both the 8 MB code space and the 8 MB data space. The other reason is that it is much more efficient to manipulate multi-byte pointer values in registers than it is in SFRs. C compilers typically already perform pointer manipulation in registers, then move the result to a Data Pointer for use.

Two Universal Pointers are supported: PR0 and PR1. The pointer PR0 is composed of registers R1, R2, and R3 of the current register bank, while PR1 is composed of registers R5, R6, and R7 of the current register bank, as shown in Figure 14.

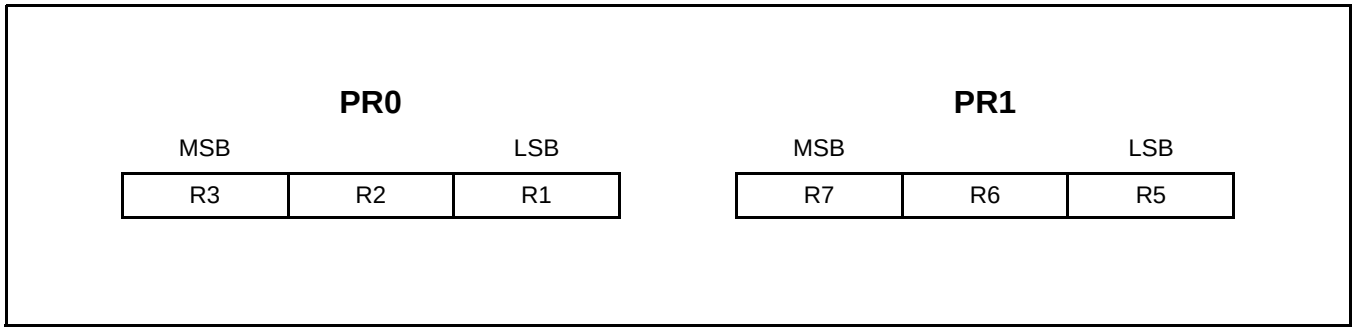


Figure 14: Universal Pointer Registers

In order to access all of the various memory spaces in a single unified manner, they must all be mapped into a new "view" that allows 16 B of total memory space. This new view is called the Universal Memory Map.

The XDATA space is placed at the bottom of this new address map. The HDATA space continues above XDATA. The standard internal data memory spaces (DATA and IDATA) are above HDATA, followed by the remainder of the EDATA space. Finally, the code memory occupies the top of the map.

Thus, the most significant bit of the Universal Pointer determines whether code or data memory is accessed. By placing the XDATA space at the bottom of the Universal Memory Map, Universal Pointer addresses 00:0000 through 00:FFFF can correspond to the classic 80C51 external data memory space. This allows for full backward compatibility for code that does not need more than 64k of external data space. The Universal Memory Map is shown in Figure 15, while the standard view of the memory spaces and how they relate to Universal Pointer values are shown in Figure 16.

The Universal Pointers are used only by a new 51MX instruction called EMOV. The EMOV instruction allows moving data via one of the Universal Pointers into or out of the accumulator. In either case, a displacement of 0, 1, 2, or 3 may also be specified, which is added to the pointer prior to its use. The displacement allows C compiler access of variables of up to 4 bytes in size (e.g. Long Integers) without the need to alter the pointer value. An example of Universal Pointer usage is shown in Figure 17. Note that it is not possible to store a value to the CODE area of the Universal Memory Map.

Another new instruction is added to allow incrementing one of the Universal Pointers by a value from 1 to 4. This allows the pointer to be advanced past the last data element accessed, to the next data element.

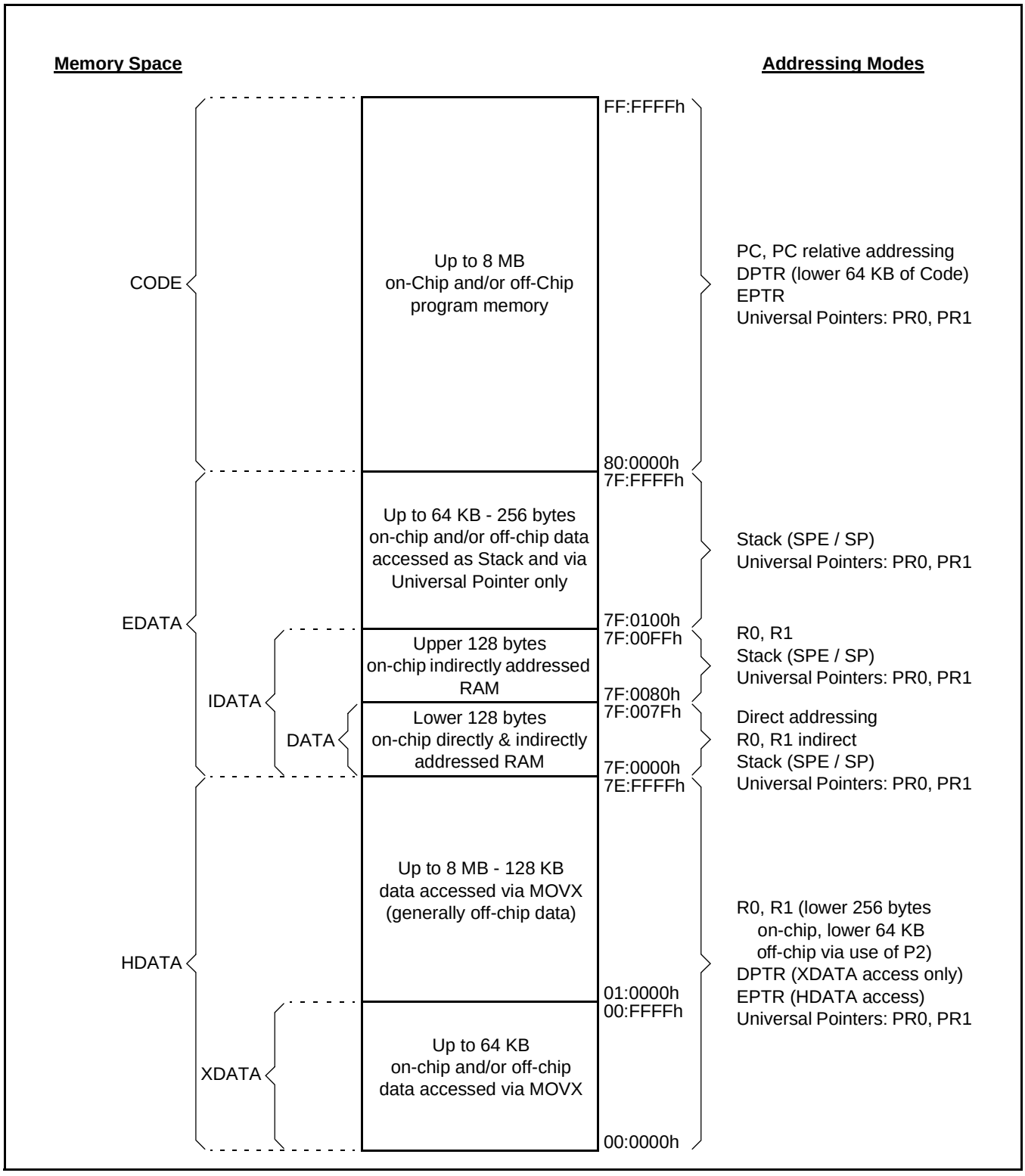


Figure 15: Universal Memory Map

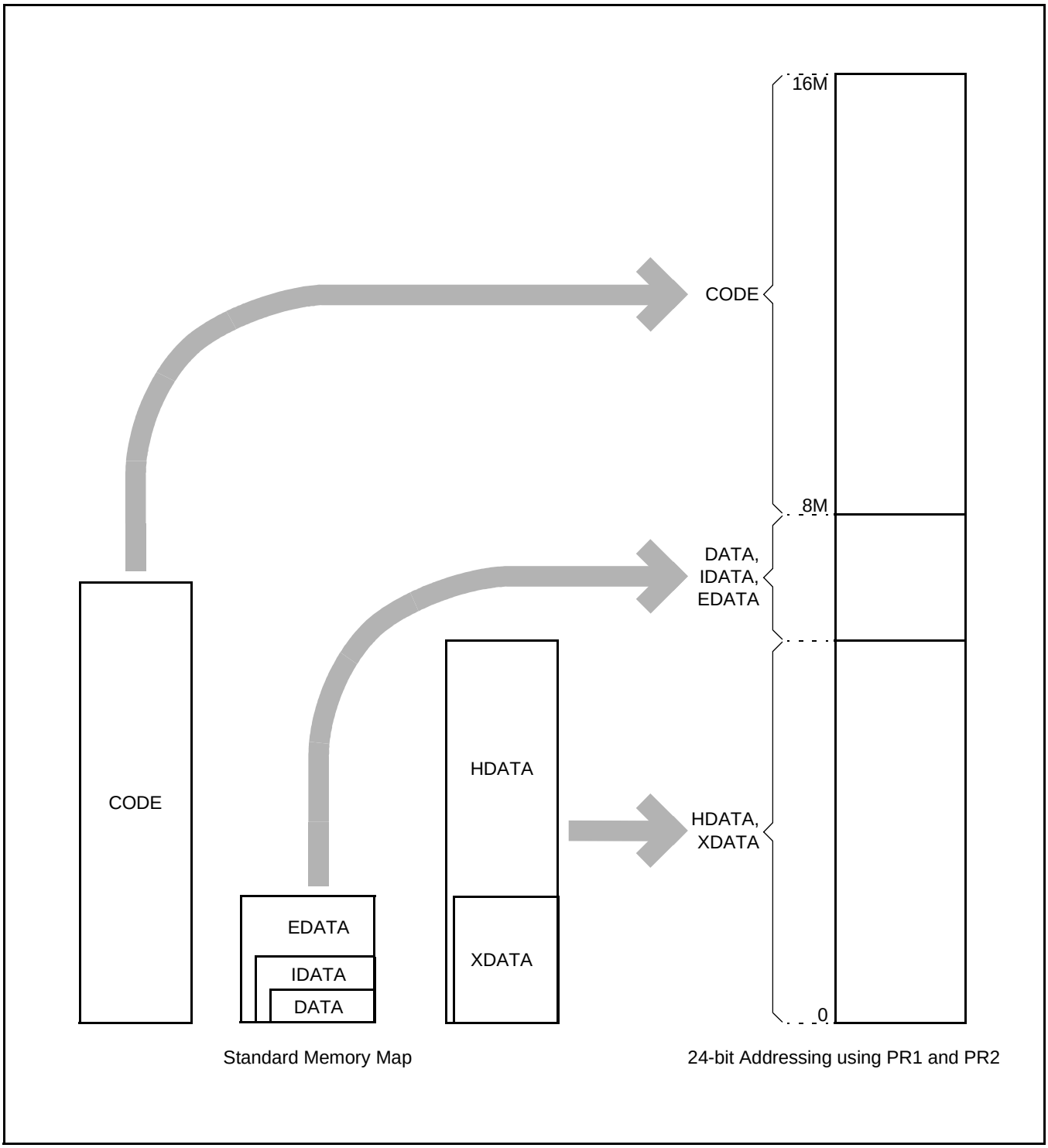


Figure 16: Mapping of other Addressing Modes to Universal Pointer Addressing

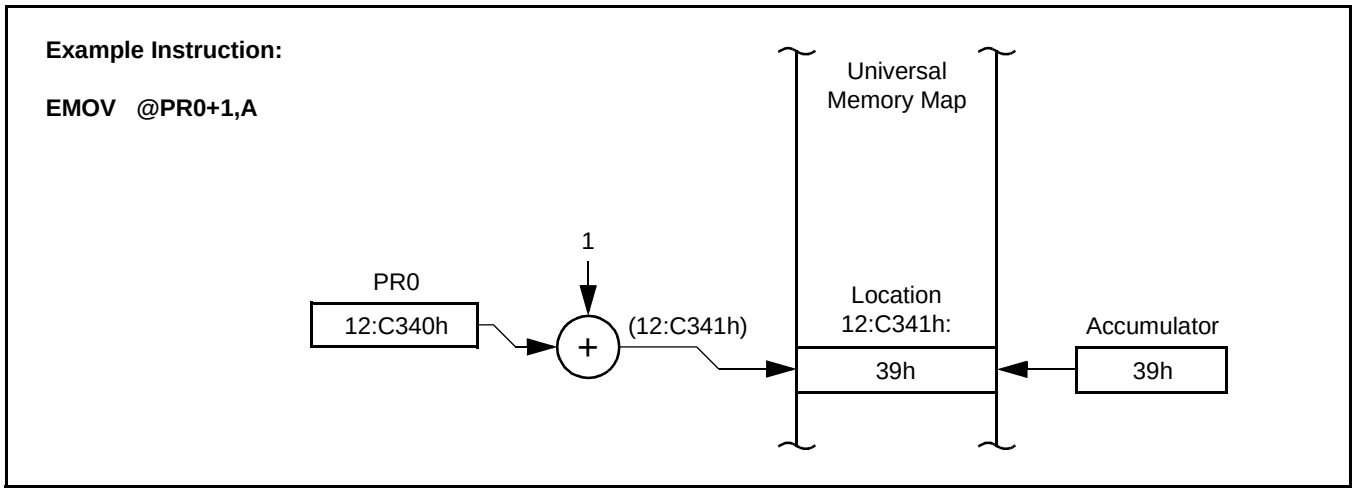


Figure 17: Memory Access using Universal Pointer Addressing

Universal Pointers are designed primarily to facilitate addressing in Extended Addressing Mode, with the EAM bit in MXCON set to one. However, Universal Pointers may still be used when EAM = 0. In this case, Universal Pointer addressing can access only the bottom 64 KB of the Code space, the 64 KB XDATA space, and the 64 KB EDATA space. The Universal Pointer values that point to these areas do not change. When EAM = 0, Universal Pointer accesses outside of these areas are not accessible and will return a value of FF hex.

3 INSTRUCTION SET OVERVIEW

The 51MX instruction set is a a true binary-level superset of the classic 80C51, designed to be fully compatible with previously written 80C51 code. The changes to the instruction set are all related to the expanded address space. Some details of existing instructions have been altered, and some instructions have had an extended mode added. In the latter case, the alternate mode of the instruction is activated by preceding the instruction with a special one-byte prefix code, A5h.

An important goal in the implementation of the 51MX was to keep the same timing relationship of existing 80C51 instructions to existing devices. Any 80C51 instruction executed on the 51MX will take the same number of machine cycles to execute.

Refer to Appendix A for details on 51MX instructions.

80C51 Instruction	Effect of Extended Addressing
All relative branches	Includes SJMP and all conditional branches. These instructions may cross a 64 KB boundary if they are located within branch range of the boundary.
ACALL addr11	This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.
AJMP addr11	This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.
JMP @A+DPTR	The lower 16-bits of the Program Counter are replaced with the value formed by the sum of the Accumulator and the active DPTR. This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.
MOVC A,@A+DPTR	The address formed by replacing the lower 16-bits of the Program Counter with the value formed by the sum of the Accumulator and the active DPTR is used to access code memory. The PC value used is that of the instruction following MOVC.
MOVC A,@A+PC	The sum of the Accumulator and the 23-bit Program Counter forms the 23-bit address used to read the code memory. The PC value used is that of the instruction following MOVC.
MOVX @DPTR,A	The active DPTR points to an address in the 64 KB XDATA memory.
MOVX A,@DPTR	The active DPTR points to an address in the 64 KB XDATA memory.
RET	Replaces the lower 16 bits of the Program Counter with a 16-bit address from the Stack. This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.
RETI	When the extended interrupt frame mode is not enabled, this instruction replaces the lower 16 bits of the Program Counter with a 16-bit address from the Stack. This will cause a 64 KB boundary to be crossed if the instruction is located such that the next instruction in sequence is across the boundary. If the extended interrupt frame mode is enabled, a 23-bit address is loaded into the PC from the stack.
LCALL addr16	Replaces the lower 16 bits of the Program Counter with the 16-bit address. This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.
LJMP addr16	Replaces the lower 16 bits of the Program Counter with the 16-bit address. This instruction will cross a 64 KB boundary if it is located such that the next instruction in sequence is across the boundary.

Table 1: Instructions Affected by Extended Address Space

80C51 Instruction	51MX Effect Without Prefix	51MX Enhancement (these instructions use the prefix byte)	51MX Effect with Prefix
LCALL addr16	Load a 16-bit address into the Program Counter.	ECALL addr23	Load a 23-bit address into the Program Counter.
LJMP addr16	Load a 16-bit address into the Program Counter.	EJMP addr23	Load a 23-bit address into the Program Counter.
JMP @A+DPTR	The lower 16-bits of the Program Counter are replaced with the sum of the Accumulator and the active DPTR.	JMP @A+EPTR	The Program Counter is loaded with the value formed by the sum of the Accumulator and the EPTR.
MOVC A,@A+DPTR	Code memory is accessed using the address formed by replacing the lower 16-bits of the Program Counter with the sum of the Accumulator and the active DPTR.	MOVC A,@A+EPTR	Code memory is accessed using the address formed by the sum of the Accumulator and the EPTR.
MOVX @DPTR,A	The active DPTR points to an address in the 64 KB XDATA memory.	MOVX @EPTR,A	The EPTR points to an address anywhere in HDATA memory (not DATA, IDATA, or EDATA).
MOVX A,@DPTR	The active DPTR points to an address in the 64 KB XDATA memory.	MOVX A,@EPTR	The EPTR points to an address anywhere in HDATA memory (not DATA, IDATA, or EDATA).
INC DPTR	Increment the active Data Pointer.	INC EPTR	Increment the 23 bit EPTR.
MOV DPTR,#data16	Load a 16-bit value into the active Data Pointer.	MOV EPTR,#data23	Load a 23-bit value into the EPTR.
RET	Load a 16-bit address into the Program Counter from the Stack.	ERET	Load a 23-bit address into the Program Counter from the Stack.
ORL A,Rn	Logically OR Register n to the Accumulator.	EMOV A,@PRi+disp	Load the Accumulator with the value from the Universal Memory Map at the address formed by PR0 or PR1plus the displacement (a value from 0 to 3).
ANL A,Rn	Logically AND Register n to the Accumulator.	EMOV @PRi+disp,A	Load the Universal Memory Map address formed by PR0 or PR1 plus the displacement (a value from 0 to 3) with the contents of the Accumulator.
XRL A,Rn	Exclusive OR Register n to the Accumulator.	ADD PRi,#data2	Add an immediate data value from 1 to 4 to the specified Universal Pointer. This is a 24-bit addition.

Table 2: Enhancements to the 80C51 Instruction Set Enabled by the Prefix Byte

4 INTERRUPT PROCESSING

Interrupt processing on the 51MX is the same as on the classic 80C51 but taking extended code addressing capability into account.

When an interrupt occurs, there are two possible actions, depending on the stack mode. If the stack is in 80C51 mode (EIFM=0 in MXCON sfr), the lower 16-bits of the Program Counter are saved on the stack. When the interrupt service routine completes, it returns to the interrupted code by executing the RETI (return from interrupt) instruction. This instruction loads the Program Counter with a 16-bit value, causing execution to resume at the point of interruption.

If the stack is in Extended Interrupt Frame Mode (EIFM=1 in MXCON sfr), the entire 23-bit Program Counter value is saved on the stack. The interrupt is still terminated by executing the RETI instruction. In the Extended Interrupt Frame Mode, this causes the entire 23-bit Program Counter to be loaded from the stack. In effect, the extended interrupt frame mode must be used if the program that can be interrupted extends beyond the 64 KB 80C51 limit.

4.1 INTERRUPT ENABLES AND PRIORITIES

The 51MX family uses four priority level interrupt structure. This allows great flexibility in controlling the handling of the many interrupt sources.

Each interrupt source can be individually enabled or disabled by setting or clearing a bit in dedicated registers. Global disable bit disables and enables all individually enabled interrupts at once.

Each interrupt source can be individually programmed to one of four priority levels by setting or clearing bits in interrupt priority registers. An interrupt service routine in progress can be interrupted by a higher priority interrupt, but not by another interrupt of the same or lower priority. The highest priority interrupt service cannot be interrupted by any other interrupt source. So, if two requests of different priority levels are received simultaneously, the request of higher priority level is serviced.

If requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. This is called the arbitration ranking. Note that the arbitration ranking is only used to resolve simultaneous requests of the same priority level.

For more information on interrupt handling in an 51MX based device, please refer to the corresponding "User’s Manual".

5 EXTERNAL BUS

The external bus provides address information to external devices, and initiates code read, data read, or data write operations. In the 51MX devices, the external bus duplicates the classic 80C51 multiplexed external bus, but allows increasing the address output to 23 bits.

5.1 MULTIPLEXED EXTERNAL BUS

The 51MX external bus supports 8-bit data transfers and up to 23 address lines. The number of address lines available is configurable, and depends on the setting of the EAM in the MXCON register.

The default for an unprogrammed part following reset is 16 address bits. This provides drop-in compatibility in existing 80C51 sockets. A non-volatile configuration bit allows pre-selecting a 23-bit address size at the time that the part is programmed. Software may later enable the Extended Addressing Mode (by setting EAM=1 in MXCON sfr) even if the pre-programmed configuration does not. However, if non-volatile configuration bit is programmed and configures external bus to operate with 23 bits, this interface can not be reversed to 16 bits, characteristic for standard 80C51!

The non-volatile address configuration is implemented using EPROM technology. The configuration is comprised of a single bit that enables multiplexing of the 7 extended address bits on Port 2. If the non-volatile configuration bit is not programmed, extended addressing may be enabled at run time via the EAM bit in the MXCON SFR. Software may write a 1 to MXCON, changing the default configuration. Typically, this would be done a single time. If software reads the EAM bit in MXCON, the value will be the logical OR of the non-volatile configuration bit and the MXCON.EAM bit value. It is not recommended to change the address configuration dynamically during program execution (for example: changing EAM=1 to EAM=0 changes external memory bus interface and prevents core from executing code above the 64 KB boundary). The encoding of the configuration bit is such that an unprogrammed device is configured for 16 address lines. The erased state of the physical cell is interpreted in such a way as to produce that result.

When the full 23-bit address is multiplexed on Port 2 (when the EAM bit in MXCON = 1), the high order address information (bits A22 through A16) must be latched externally in the same manner as the low order bits (A7 through A0) on Port 0. The middle address bits (A15 through A8) appear on Port 2 after ALE goes low. If extended addressing is not enabled, Port 2 behaves just as on a classic 80C51. An example of Port 2 address multiplexing is shown in Figure 18.

There are two special cases for Port 2 multiplexing when extended addressing is enabled: MOVX @Ri and MOVX @DPTR. These instructions do not supply a source for a full 23-bit external address. Where program memory is involved (jumps and MOVC), any "missing" address bits are supplied by the Program Counter (see Table 1). For MOVX, the additional bits are forced to zeroes to complete the address. So, MOVX @Ri will output a 23-bit address composed of seven zeroes for the upper address, Port 2 SFR contents for the middle byte of the address, and Ri contents for the bottom byte. Similarly, MOVX @DPTR will output a 23-bit address composed of seven zeroes for the upper address and the current DPTR contents for the middle and bottom bytes of the address. Figures 19 and 20 present signal waveforms when external DATA memory is accessed with all 23 address bits (EAM=1 in MXCON sfr).

In a single-chip application with code exceeding 64 KB (and thus having EAM=1) and need to preserve an old 80C51 bus interface, the instruction EMOV @PRi,A should be used instead of MOVX @Ri,A. By loading the content of P2 sfr to R3 and R2, execution of instruction EMOV @PR0,A will have exactly the same output in a system with EAM=1 as it is in case of MOVX @R0,A in a design with standard 80C51bus interface.

Some 51MX applications may use extended addressing and rely on software setting the EAM bit in MXCON (i.e. the non-volatile address configuration bit is not programmed). If such an application is set up in a way that the first code executed upon reset is off-chip, then the instruction that sets the EAM bit in MXCON must be located at or below address 00FBh. This is to prevent the external bus from supplying a 16-bit address when a 23-bit address is required. If the Program Counter were to reach address 0100h while EAM = 0, the apparent address (to external hardware that is expecting a 23-bit address) would become 01:0100.

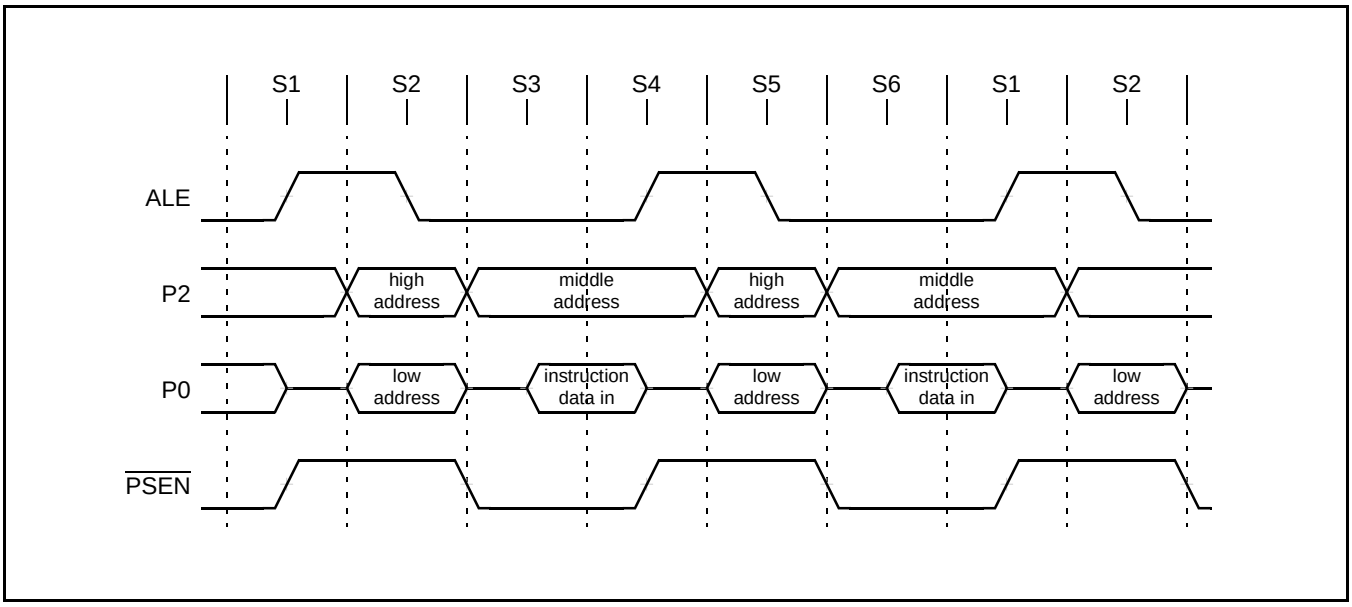


Figure 18: Example of External Code Memory Read Cycles using 23 Address Bits

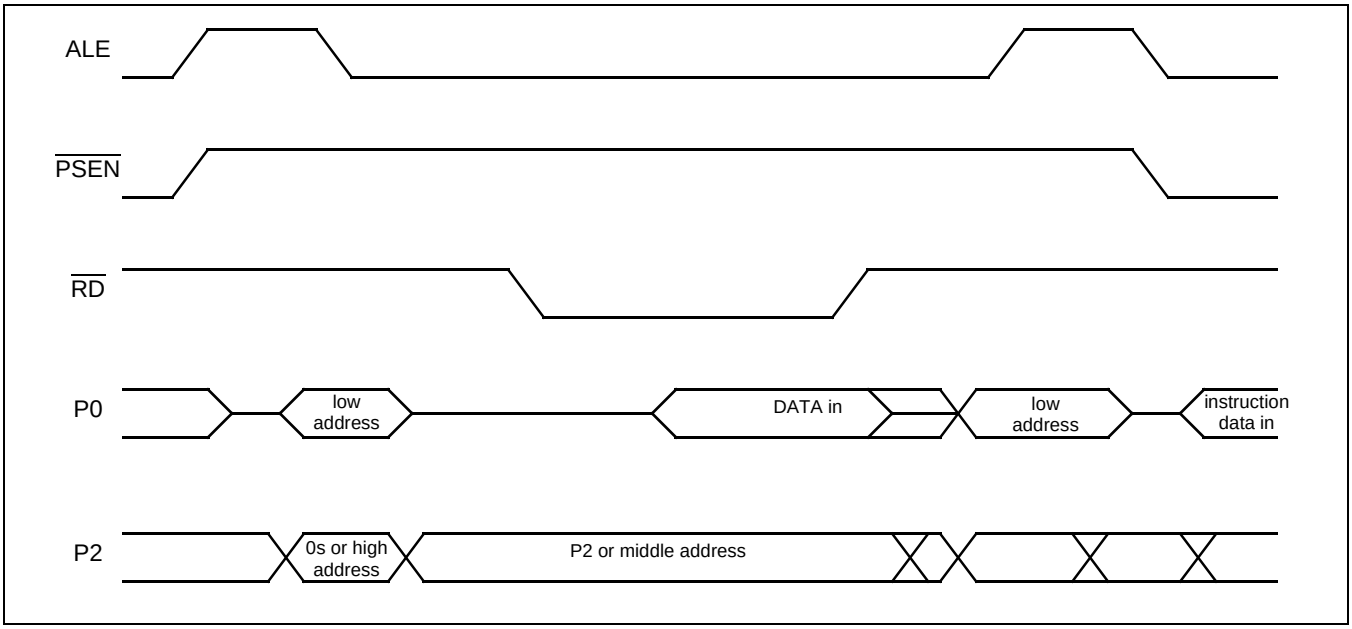


Figure 19: Example of External Data Memory Read Cycle using 23 Address Bits

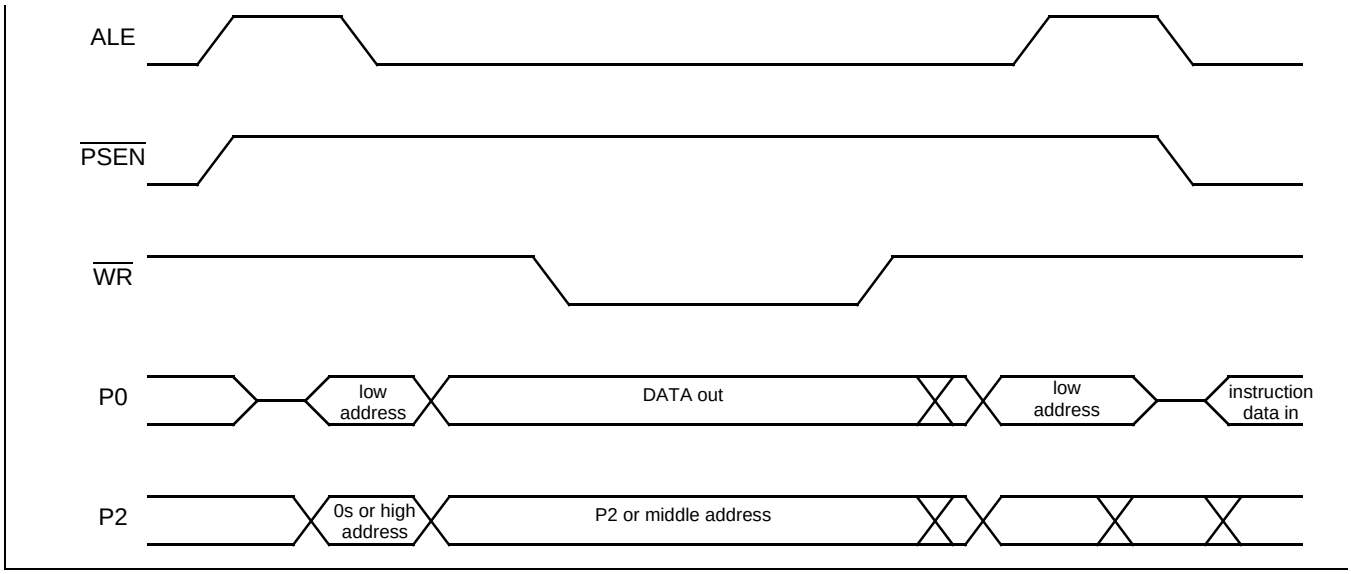


Figure 20: Example of External Data Memory Write Cycle using 23 Address Bits

The standard control signals and their functions for the external bus are as follows:

Signal name	Function
ALE	Address Latch Enable. This signal directs an external address latch to store the multiplexed portion of the address for the next bus operation. This may be either a data address or a code address.
$\overline{\text{PSEN}}$	Program Store Enable. Indicates that the processor is reading code from the bus. Typically connected to the Output Enable pin of external EPROMs or other memory devices. External bus addresses for code memory may range from 00:0000 through 7F:FFFF. In the Universal Memory Map, these correspond to addresses 80:0000 through FF:FFFF
$\overline{\text{RD}}$	Read. The external data read strobe. Typically connected to the $\overline{\text{RD}}$ pin of external peripheral devices.
$\overline{\text{WR}}$	Write. The write strobe for external data. Typically connected to the $\overline{\text{WR}}$ pin of external peripheral devices.

External bus addresses for data memory may range from 00:0000 through 7E:FFFF, which matches Universal Memory Map addresses. If on-chip XDATA is enabled, it will cause an addressing discontinuity in the external data address space. The DATA and IDATA spaces are always on-chip, and therefore always create such an addressing discontinuity.

APPENDIX A 51MX INSTRUCTION SET

This section describes the 51MX instruction set. There is a definition of terms and notations used in this chapter and a summary of the whole instruction set (Table 1), followed by a detailed description of each instruction. The detailed description includes instruction encoding, byte count, timing, pseudocode equivalents, and examples for each instruction.

The 51MX instruction set is identical to the standard 80C51 instruction set with additions and extensions to allow extended addressing capabilities. These extensions are identified at the beginning of the description of each affected instruction type. The instruction set has been extended to allow branching anywhere within the extended 8 MB code address space, to allow using a new 23-bit pointer (EPTR) to access the 8 Megabyte data memory, and to provide a unified method of addressing the entire code and data space from a single 24-bit Universal Pointer.

A.1 GLOSSARY OF TERMS AND NOTATION USED IN THIS SECTION

General:

@	Indicates an indirect reference (e.g.: @R0 refers to the memory address pointed to by the contents of R0).
:	Used to divide a 23 or 24-bit address purely for visual clarity. For example 01:2345h.
A	The Accumulator
AB	The Accumulator and the B register, when both used as operands in the MUL and DIV instructions.
AC	Auxiliary Carry flag from the PSW.
addr11	An 11-bit branch address used in ACALL and AJMP. The branch will be within the same 2 KB block of Code memory as the first byte of the following instruction.
addr16	A 16-bit branch address used in LCALL and LJMP. The branch will be within the same 64 KB block of Code memory as the first byte of the following instruction.
addr23	A 23-bit branch address used in ECALL and EJMP. The branch may be anywhere in the 8 MB Code memory.
B	The B register.
bit	The 8-bit address of an addressable bit in the internal data memory or in a Special Function Register.
/bit	The logical complement (inverse) of an addressable bit.
C	Carry flag from the PSW.
#data	8 bits of immediate data contained in the instruction.
#data2	2 bits of immediate data contained in the instruction.
#data16	16 bits of immediate data contained in the instruction.
#data23	23 bits of immediate data contained in the instruction.
direct	An 8-bit immediate address contained in the instruction. This could be an internal data memory location (0-127) or an SFR [i.e., I/O port, control register, status register, etc. (128-255)].
DPTR	The active 16-bit Data Pointer. One of two Data Pointers, as selected by the Data Pointer Select bits.
EPTR	The 23-bit Extended Pointer.
OV	Overflow flag from the PSW.
PC	The 23-bit Program Counter.
PRi	PR1 or PR0 of the currently selected Register Bank. Used in Universal Pointer addressing. PR1 is comprised of R7:R6:R5 as a 24-bit entity. PR0 is comprised of R3:R2:R1 as a 24-bit entity.
rel	An 8-bit signed relative displacement for branches, used by SJMP and all conditional jumps. Range is +127 to -128 bytes relative to first byte of the following instruction.
Ri	R1 or R0 of the currently selected Register Bank. Used in indirect addressing.
Rn	Any of registers R7 through R0 of the currently selected Register Bank.
SP	The Stack Pointer.

Pseudocode Notation:

←	Pseudocode assignment operator. Occasionally used as ↔ to indicate the interchange of data in both directions.
( )	Used to indicate "contents of" in the instruction operation pseudocode.
	Examples: (R4) is the contents of register 4.
	(C) is the contents of the Carry flag.
(N.x)	Indicates bit x of object N.
(N.x-y)	Indicates a range of bits from bit x to bit y of object N.
	Example: (PC.15-8) is the contents of the Program Counter, bits 8 through 15.
AND	Logical operation: bitwise AND.
OR	Logical operation: bitwise OR.
XOR	Logical operation: bitwise Exclusive-OR.

Memory Spaces:

DATA Lower 128 bytes of standard internal data memory space, accessed via direct addressing using instructions other than MOVX and MOVC. In pseudocode descriptions, the contents of a DATA location are shown as:

(DATA (direct))

IDATA Indirect Data. 256 bytes of internal data memory space, accessed via indirect addressing using instructions other than MOVX and MOVC. This area includes the DATA area and the 128 bytes immediately above it. In pseudocode descriptions, the contents of an IDATA location are shown as:

(IDATA (Ri))

EIDATA Extended Indirect Data. This space is an extension to the DATA and IDATA areas and allows up to a potential total of 64 KB). This area is accessed as Stack or via indirect addressing using Universal Pointers. In pseudocode descriptions, the contents of an EIDATA location are shown as:

(EIDATA (SP))

XDATA "External" Data. Up to 8,323,072 bytes (8 MB - 64 KB) of memory space addressed via the MOVX instruction. In pseudocode descriptions, the contents of an XDATA location is shown in one of the following forms:

(XDATA (Ri))

(XDATA (DPTR))

(XDATA (EPTR))

CODE Up to 8 Megabytes of Code memory, accessed as part of program execution and via the MOVC instruction. In pseudocode descriptions, the contents of a CODE location is shown in one of the following forms:

(CODE ((PC.22-16) : (A) + (DPTR)))

(CODE ((A) + (PC)))

UMEM Universal Memory addressing mode that allows all of the other memory areas to be accessed through the use of a Universal Pointer. In pseudocode descriptions, the contents of a memory location addressed by a Universal Pointer are shown as:

(UMEM ((PRi) + data2))

A.2 51MX INSTRUCTION SET SUMMARY

Table 1 contains a summary of the 51MX instruction set, with a brief description, byte count, and the number of oscillator periods required to execute each instruction. A detailed description of each instruction may be found in the section following the table.

Mnemonic		Description	Bytes	Machine Cycles
Arithmetic Operations				
ADD	A,Rn	Add register to Accumulator	1	1
ADD	A,direct	Add direct byte to Accumulator	2	1
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1
ADD	A,#data	Add immediate data to Accumulator	2	1
ADD	**PRi,#data2**	**Add immediate data to the specified Universal Pointer**	**2**	**4**
ADDC	A,Rn	Add register to Accumulator with Carry	1	1
ADDC	A,direct	Add direct byte to Accumulator with Carry	2	1
ADDC	A,@Ri	Add indirect RAM to Accumulator with Carry	1	1
ADDC	A,#data	Add immediate data to Accumulator with Carry	2	1
SUBB	A,Rn	Subtract Register from Accumulator with borrow	1	1
SUBB	A,direct	Subtract direct byte from Accumulator with borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from Accumulator with borrow	1	1
SUBB	A,#data	Subtract immediate data from Accumulator with borrow	2	1
INC	A	Increment Accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
INC	DPTR	Increment the 16-bit DPTR	1	2
INC	**EPTR**	**Increment the extended 23-bit EPTR**	**2**	**2**
DEC	A	Decrement Accumulator	1	1
DEC	Rn	Decrement Register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
MUL	AB	Multiply A and B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal Adjust Accumulator	1	1
Logical Operations				
ANL	A,Rn	AND Register to Accumulator	1	1
ANL	A,direct	AND direct byte to Accumulator	2	1
ANL	A,@Ri	AND indirect RAM to Accumulator	1	1
ANL	A,#data	AND immediate data to Accumulator	2	1
ANL	direct,A	AND Accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2

Table 1: Summary of the 51MX Instruction Set

Mnemonic		Description	Bytes	Machine Cycles
ORL	A,Rn	OR register to Accumulator	1	1
ORL	A,direct	OR direct byte to Accumulator	2	1
ORL	A,@Ri	OR indirect RAM to Accumulator	1	1
ORL	A,#data	OR immediate data to Accumulator	2	1
ORL	direct,A	OR Accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive-OR register to Accumulator	1	1
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	1
XRL	A,@Ri	Exclusive-OR indirect RAM to Accumulator	1	1
XRL	A,#data	Exclusive-OR immediate data to Accumulator	2	1
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	1
XRL	direct,#data	Exclusive-OR immediate data to direct byte	3	2
CLR	A	Clear Accumulator	1	1
CPL	A	Complement Accumulator	1	1
RL	A	Rotate Accumulator left	1	1
RLC	A	Rotate Accumulator left through the Carry flag	1	1
RR	A	Rotate Accumulator right	1	1
RRC	A	Rotate Accumulator right through the Carry flag	1	1
SWAP	A	Swap nibbles within the Accumulator	1	1
Data Transfer				
MOV	A,Rn	Move register to Accumulator	1	1
MOV	A,direct	Move direct byte to Accumulator	2	1
MOV	A,@Ri	Move indirect RAM to Accumulator	1	1
MOV	A,#data	Move immediate data to Accumulator	2	1
MOV	Rn,A	Move Accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	2
MOV	Rn,#data	Move immediate data to register	2	1
MOV	direct,A	Move Accumulator to direct byte	2	1
MOV	direct,Rn	Move register to direct byte	2	2
MOV	direct,direct	Move direct byte to direct	3	2
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	2
MOV	@Ri,A	Move Accumulator to indirect RAM	1	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2

Table 1: Summary of the 51MX Instruction Set (Continued)

Mnemonic		Description	Bytes	Machine Cycles
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1
MOV	DPTR,#data16	Load the 16-bit DPTR with a constant	3	2
MOV	**EPTR,#data23**	**Load the 23-bit EPTR with a constant**	**5**	**4**
MOVC	A,@A+DPTR	Move code byte relative to DPTR into the Accumulator	1	2
MOVC	**A,@A+EPTR**	**Move code byte relative to EPTR into the Accumulator**	**2**	**4**
MOVC	A,@A+PC	Move code byte relative to the PC into the Accumulator	1	2
MOVX	A,@Ri	Move external RAM (8-bit addr) to Accumulator	1	2
MOVX	@Ri,A	Move Accumulator to external data memory (8-bit addr)	1	2
MOVX	A,@DPTR	Move XDATA memory to Accumulator, using DPTR	1	2
MOVX	@DPTR,A	Move Accumulator to XDATA memory, using DPTR	1	2
MOVX	**A,@EPTR**	**Move XDATA memory to Accumulator, using EPTR**	**2**	**4**
MOVX	**@EPTR,A**	**Move Accumulator to XDATA memory, using EPTR**	**2**	**4**
EMOV	**A,@PRi+data2**	**Move data to the Accumulator from memory, using a Universal Pointer and a 2-bit displacement**	**2**	**4**
EMOV	**@PRi+data2,A**	**Move data to memory from the Accumulator, using a Universal Pointer and a 2-bit displacement**	**2**	**4**
PUSH	direct	Push direct byte onto stack	2	2
POP	direct	Pop direct byte from stack	2	2
XCH	A,Rn	Exchange register with Accumulator	1	1
XCH	A,direct	Exchange direct byte with Accumulator	2	1
XCH	A,@Ri	Exchange indirect RAM with Accumulator	1	1
XCHD	A,@Ri	Exchange low-order digit indirect RAM with ACC	1	1
Bit Manipulation and Test				
CLR	C	Clear Carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set Carry flag	1	1
SETB	bit	Set direct bit	2	1
CPL	C	Complement Carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to Carry flag	2	2
ANL	C,/bit	AND complement of direct bit to Carry flag	2	2
ORL	C,bit	OR direct bit to Carry flag	2	2
ORL	C,/bit	OR complement of direct bit to Carry flag	2	2
MOV	C,bit	Move direct bit to Carry flag	2	1
MOV	bit,C	Move Carry flag to direct bit	2	2

Table 1: Summary of the 51MX Instruction Set (Continued)

Mnemonic		Description	Bytes	Machine Cycles
Program Branching				
SJMP	rel	Short jump (relative addr)	2	2
AJMP	addr11	Absolute jump	2	2
LJMP	addr16	Long jump	3	2
EJMP	**addr23**	**Long jump, using a 23-bit absolute address**	**5**	**4**
JMP	@A+DPTR	Jump indirect relative to the 16-bit DPTR	1	2
JMP	**@A+EPTR**	**Jump indirect relative to the 23-bit EPTR**	**2**	**2**
ACALL	addr11	Absolute subroutine call, saving a 16-bit return address	2	2
LCALL	addr16	Long subroutine call using a 16-bit address, saving a 16-bit return address	3	2
ECALL	**addr23**	**Extended subroutine call using a 23-bit address, and saving a 23-bit return address**	**5**	**4**
RET		Return from subroutine, with 16-bit return address	1	2
ERET		**Return from subroutine, with 23-bit return address**	**2**	**4**
RETI		Return from interrupt, with either a 16-bit or 23-bit return address, depending on the EIFM mode bit in MXCON.	1	2
JZ	rel	Jump if Accumulator is zero	2	2
JNZ	rel	Jump if Accumulator is not zero	2	2
JC	rel	Jump if Carry flag is set	2	2
JNC	rel	Jump if Carry flag not set	2	2
JB	rel	Jump if direct bit is set	3	2
JNB	rel	Jump if direct bit is not set	3	2
JBC	bit,rel	Jump if direct bit is set and clear bit	3	2
CJNE	A,direct,rel	Compare direct byte to Accumulator and jump if not equal	3	2
CJNE	A,#data,rel	Compare immediate to Accumulator and jump if not equal	3	2
CJNE	Rn,#data,rel	Compare immediate to register and jump if not equal	3	2
CJNE	@Ri,#data,rel	Compare immediate to indirect and jump if not equal	3	2
DJNZ	Rn,rel	Decrement register and jump if not zero	2	2
DJNZ	direct,rel	Decrement direct byte and jump if not zero	3	2
NOP		No operation	1	1

Table 1: Summary of the 51MX Instruction Set (Continued)

Instruction(s)	Flags		
	C	AC	OV
ADD ; ADDC ; SUBB	U	U	U
MUL ; DIV	0	-	U
RLC ; RRC ; DA ; CJNE	U	-	-
ANL C,bit ; ANL C,/bit	U	-	-
ORL C,bit ; ORL C,/bit	U	-	-
MOV C,bit ; CPL C	U	-	-
SETB C	1	-	-
CLR C	0	-	-

Table 2: Instructions that affect Flag Settings

Notes:
1. 'U' indicates a flag update, the value is data dependent
2. Operations on the PSW itself or bits in the PSW will also affect flag settings.

Any instruction that alters the accumulator contents will cause an update to the parity flag (P).

DETAILED INSTRUCTION DESCRIPTIONS

ACALL Absolute Call

Function: Absolute call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the next instruction, then pushes the lower 16 bits of the PC value onto the stack, incrementing the Stack Pointer by two in the process. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. The destination address is obtained by successively replacing the bottom 11 bits of the new PC value with opcode bits 7-5 and the second byte of the instruction. The subroutine called must therefore start within the same 2k block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Note that the ACALL and LCALL instruction only push two bytes of address information onto the stack. For this reason, ERET must not be used to return from subroutines that were called using those instructions. A general rule is that any subroutine that must called at any time using the ECALL instruction must always be called using ECALL, and must also terminate with an ERET instruction. Mismatch of call and return types will cause stack de-synchronization and a probable program crash.

Example: Initially SP equals 07h. The label “SUBRTN” is at program memory location 02:0345h. The following instruction, located at address 02:0123h, is executed:

```
ACALL   SUBRTN
```

The SP will contain 09h, internal RAM locations 08h and 09h will contain 25h and 01h, respectively, and the PC will contain 02:0345h. This branch is possible since the PC is within the same 2 KB address range (02:0000 to 02:07FF) as SUBRTN.

ACALL addr11

Bytes: 2

Cycles: 2

Encoding:

a10 a9 a8 1	0 0 0 1	a7 a6 a5 a4	a3 a2 a1 a0

Operation:

(PC) ← (PC) + 2

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.7-0)

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.15-8)

(PC.10-0) ← a10-a0

ADD Add Byte

Function: Add the specified byte to the Accumulator

Description: Adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The Carry flag and Auxiliary Carry flags are set, respectively, if there is a carry-out from 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 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 0C3h (11000011b) and register 0 holds 0AAh (10101010b). The following instruction is executed:

```
ADD   A,R0
```

The Accumulator contains 6Dh (01101101b), the AC flag is cleared and both the Carry flag and OV are set to 1.

ADD A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	1	r	r	r

Operation: (A) ← (A) + (Rn)

ADD A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	1	direct address

Operation: (A) ← (A) + (DATA (direct))

ADD A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	1	1	i

Operation: (A) ← (A) + (IDATA (Ri))

ADD A,#data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	0	0	1	0	0	immediate data

Operation: (A) ← (A) + #data

ADD Add to Universal Pointer (extended instruction)

Function: Add an immediate value to a Universal Pointer

Description: Adds the value defined by the 2-bit immediate data field to the specified Universal Pointer (PR0 or PR1), leaving the result in the same Universal Pointer. The immediate data value may be 1 through 4. No flags are affected.

Example: Registers R7, R6, and R5 (comprising PR1), contain the values 78h, FFh, and FDh, respectively. The following instruction is executed:

```
ADD     PR1,#4
```

R7, R6, and R5 will contain 79h, 00h, and 01h respectively.

ADD PRi,#data2

Bytes: 2

Cycles: (tbd)

Encoding:

A5	0 1 1 0	1 i d1 d0

Operation: (PRi) ← (PRi) + #data2 (when d1, d0 = 00, the value 4 is added to PRi)

ADDC Add with Carry

Function: Add the specified byte and the Carry flag to the Accumulator

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 from 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 0C3h (11000011b) and register 0 holds 0AAh (10101010b) with the Carry flag set. The following instruction is executed:

```
ADDC    A,R0
```

The Accumulator contains 6Eh (01101110b), the AC flag is cleared and both the Carry flag and OV are set to 1.

ADDC A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	1	r	r	r

Operation: (A) ← (A) + (C) + (Rn)

ADDC A,direct

Bytes: 2

Cycles: 1

Encoding:

0	0	1	1	0	1	0	1	direct address

Operation: (A) ← (A) + (C) + (DATA (direct))

ADDC A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	1	1	i

Operation: (A) ← (A) + (C) + (IDATA (Ri))

ADDC A,#data

Bytes: 2

Cycles: 1

Encoding:

0	0	1	1	0	1	0	0	immediate data

Operation: (A) ← (A) + (C) + #data

AJMP Absolute Jump

Function: Absolute jump

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 by two), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2k block of program memory as the first byte of the instruction following AJMP. No flags are affected.

Example: The label “JMPADR” is at program memory location 04:0123h. The following instruction, located at address 03:FFFEh, is executed:

```
AJMP    JMPADR
```

The PC contains 04:0123h. This branch is possible since, after the PC is incremented, it will be within the same 2 KB address range (04:0000 to 04:07FF) as JMPADR.

AJMP addr11

Bytes: 2

Cycles: 2

Encoding:

a10 a9 a8 0	0 0 0 1	a7 a6 a5 a4	a3 a2 a1 a0

Operation: (PC) ← (PC) + 2

(PC.10-0) ← a10-a0

ANL Logical AND Byte

Function: Logical AND the specified bytes

Description: ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected.

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: The Accumulator holds 0C3h (11000011b) and register 0 holds 55h (01010101b). The following instruction is executed:

```
ANL     A,R0
```

The Accumulator contains 41h (01000001b).

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 following instruction is executed:

```
ANL     P1,#01110011b
```

Port 1 bits 7, 3, and 2 are be cleared.

ANL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	0	1	1	r	r	r

Operation: (A) ← (A) AND (Rn)

ANL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	0	1	0	1	0	1	direct address

Operation: (A) ← (A) AND (DATA (direct))

ANL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	1	0	1	0	1	1	i

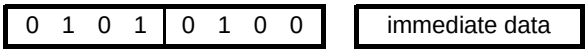
Operation: (A) ← (A) AND (IDATA (Ri))

ANL A,#data

Bytes: 2

Cycles: 1

Encoding:



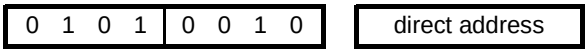
Operation: (A) ← (A) AND #data

ANL direct,A

Bytes: 2

Cycles: 1

Encoding:



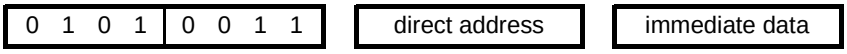
Operation: (A) ← (DATA (direct)) AND (A)

ANL direct,#data

Bytes: 3

Cycles: 2

Encoding:



Operation: (DATA (direct)) ← (DATA (direct)) AND #data

ANL Logical AND Bit

Function: Logical AND the specified bit with the Carry flag

Description: If the Boolean value of the source bit is a logical 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.

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 P1 bit 0
ANL   C,ACC.7     ; AND Carry with Accumulator bit 7
ANL   C,/OV       ; AND Carry with the inverse of the Overflow Flag
```

ANL C,bit

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 0 1 0	bit address

Operation: (C) ← (C) AND (bit)

ANL C,/bit

Bytes: 2

Cycles: 2

Encoding:

1 0 1 1	0 0 0 0	bit address

Operation: (C) ← (C) AND $\overline{(bit)}$

CJNE Compare and Jump if Not Equal

Function: Compare and two values and jump if they are not equal

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 the first operand is less than the unsigned integer value of the second operand; otherwise, the Carry flag is cleared. Neither operand is affected. No flags are 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 34h. Register 7 contains 56h. The following instruction sequence is executed:

```
         CJNE   R7,#60h,NOT_EQ
;        …      …              ; R7 = 60h.

NOT_EQ:  JC     LESS           ; Branch if R7 < 60h.
;        …      …              ; R7 > 60h.

LESS:    ...    ...            ; R7 < 60h.
```

The first instruction causes the Carry flag to be set and branches to the instruction at label NOT_EQ. Testing the Carry flag, that instruction determined that R7 is less than 60h, and therefore branched to the label LESS.

If the data being presented to Port 1 is also 34h, 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 being input on P1, the program will loop at this point until the P1 data changes to 34h.

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

1 0 1 1	0 1 0 1	direct address	rel address

Operation:

(PC) ← (PC) + 3

IF (A) ≠ (DATA (direct))

THEN (PC) ← (PC) + relative offset

IF (A) < (DATA (direct))

THEN (C) ← 1

ELSE (C) ← 0

CJNE A,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	0	1	0	0	immediate data	rel address

Operation: (PC) ← (PC) + 3

IF (A) ≠ #data

THEN (PC) ← (PC) + relative offset

IF (A) < data

THEN (C) ← 1

ELSE (C) ← 0

CJNE Rn,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	1	r	r	r	immediate data	rel address

Operation: (PC) ← (PC) + 3

IF (Rn) ≠ #data

THEN (PC) ← (PC) + relative offset

IF (Rn) < #data

THEN (C) ← 1

ELSE (C) ← 0

CJNE @Ri,#data,rel

Bytes: 3

Cycles: 2

Encoding:

1	0	1	1	0	1	1	i	immediate data	rel address

Operation: (PC) ← (PC) + 3

IF (IDATA (Ri)) ≠ data

THEN (PC) ← (PC) + relative offset

IF (IDATA (Ri)) < data

THEN (C) ← 1

ELSE (C) ← 0

CLR Clear Accumulator

Function: Clear the Accumulator

Description: The Accumulator is cleared (all bits reset to zero). No flags are affected.

Example: The Accumulator contains 5Ch (01011100b). The following instruction is executed:

```
CLR  A
```

The Accumulator contains 00h (00000000b).

CLR A

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	0	1	0	0

Operation: (A) ← 0

CLR Clear Bit

Function: Clear the specified bit

Description: The indicated bit is cleared (reset to zero). CLR can operate on the Carry flag or any directly addressable bit. No other flags are affected.

Example: Port 1 has previously been written with 5Dh (01011101b). The following instruction is executed:

```
CLR   P1.2
```

Port 1 contains the value 59h (01011001b).

CLR C

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	0	1	1

Operation: (C) ← 0

CLR bit

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	0	1	0	bit address

Operation: (bit) ← 0

CPL Complement Accumulator

Function: Complement the Accumulator

Description: Each bit of the Accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to a zero and vice-versa. No flags are affected.

Example: The Accumulator contains 5Ch (01011100b). The following instruction is executed:

```
CPL     A
```

The Accumulator contains 0A3h (10100011b).

CPL A

Bytes: 1

Cycles: 1

Encoding:

1	1	1	1	0	1	0	0

Operation: $(A) \leftarrow \overline{(A)}$

CPL Complement Bit

Function: Complement the specified bit

Description: The bit variable specified is complemented. A bit which had been a one is changed to zero and vice-versa. CPL can operate on the Carry flag or any directly addressable bit. No other flags are affected.

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 5Dh (01011101b). The following instructions are executed:

```
CPL  P1.1
CPL  P1.2
```

Port 1 contains 5Bh (01011011b).

CPL C

Bytes: 1

Cycles: 1

Encoding:

1	0	1	1	0	0	1	1

Operation: $(C) \leftarrow \overline{(C)}$

CPL bit

Bytes: 2

Cycles: 1

Encoding:

1	0	1	1	0	0	1	0	bit address

Operation: $(bit) \leftarrow \overline{(bit)}$

DA Decimal Adjust Accumulator

Function: Decimal adjust the Accumulator

Description: DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variable (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 (xxx1010-xxx1111), 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 (1010xxx-111xxxx), 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 wouldn't clear the Carry flag. 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 00h, 06h, 60h, or 66h to the Accumulator, depending on initial Accumulator and PSW conditions.

Note: DA A cannot simply convert a hexadecimal number in the Accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example: The Accumulator holds the value 56h (01010110b) representing the packed Binary Coded Decimal (BCD) digits of the decimal number 56. Register 3 contains the value 67h (01100111b) 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 two's-complement binary addition, resulting in the value 0BEh (10111110b) in the Accumulator. The Carry and Auxiliary Carry flags will be cleared.

The Decimal Adjust instruction will then alter the Accumulator to the value 24h (00100100b), 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 flag. The Carry flag will be set by the Decimal Adjust instruction, indicating that an overflow occurred. The true sum 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01h or 99h. If the Accumulator initially holds 30h (representing the digits of 30 decimal), the instruction sequence,

```
ADD     A,#99h
DA      A
```

will leave the Carry flag set and 29h in the Accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 – 1 = 29.

DA A

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	1	0	0

Operation:

```
IF   [ [ (A.3-0) > 9 ] or [ (AC) = 1 ] ]
        THEN    (A.3-0) ← (A.3-0) + 6
IF   [ [ (A.7-4) > 9 ] or [ (C) = 1 ] ]
        THEN    (A.7-4) ← (A.7-4) + 6
```

DEC Decrement

Function: Decrement the specified byte

Description: The variable indicated is decremented by 1. An original value of 00h will underflow to 0FFh. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect. No flags are affected.

Note: When this instruction is used to modify an output port, the value used as the original data will be read from the output data latch, not the input pin.

Example: Register 0 contains 7Fh (01111111b). Internal RAM locations 7Eh and 7Fh contain 00h and 40h, respectively. The following instruction sequence is executed:

```
DEC   @R0
DEC   R0
DEC   @R0
```

Register 0 contains 7Eh, internal RAM location 7Eh is set to 0FFh, and internal RAM location 7Fh is set to 3Fh.

DEC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	1	0	0

Operation: (A) ← (A) – 1

DEC Rn

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	1	r	r	r

Operation: (Rn) ← (Rn) – 1

DEC direct

Bytes: 2

Cycles: 1

Encoding:

0	0	0	1	0	1	0	1	direct address

Operation: (DATA (direct)) ← (DATA (direct)) – 1

DEC @Ri

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	1	1	i

Operation: (IDATA (Ri)) ← (IDATA (Ri)) – 1

DIV Divide

Function: Divide the Accumulator by the B register

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 00h, 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 (0FBh or 11111011b) and the B register contains 18 (12h or 00010010b). The following instruction is executed:

```
DIV     AB
```

The Accumulator contains 13 (0Dh or 00001101b) and the B register contains 17 (11h or 00010001b), since 251 = (13 x 18) + 17. The Carry flag and OV will both be cleared.

DIV AB

Bytes: 1

Cycles: 4

Encoding:

1	0	0	0	0	1	0	0

Operation: (A) ← (A) / (B)

(B) ← (remainder)

DJNZ Decrement and Jump if Not Zero

Function: Decrement the specified byte and jump if the result is not zero

Description: DJNZ decrements the indicated location by 1, and branches to the address indicated by the second operand if the resulting value is not zero. The location decremented may be a register or directly addressed byte. An original value of 00h will underflow to 0FFh. 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. No flags are affected.

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 40h, 50h, and 60h contain the values 01h, 70h, and 15h, respectively. The following instruction sequence is executed:

```
        DJNZ    40h,LABEL_1
        DJNZ    50h,LABEL_2
        DJNZ    60h,LABEL_3
```

The first instruction does not cause a branch, because the result is zero. The second instruction causes a branch since the result is not zero. As a result of the branch, the third instruction is not executed. Subsequently, the RAM locations 40h, 50h, and 60h contain the values 00h, 6Fh, and 15h.

This instruction provides a simple was 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 following instructions are executed:

```
        MOV     R2,#8
TOGGLE: CPL     P1.7
        DJNZ    R2,TOGGLE
```

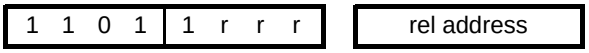
P1.7 is toggled eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse will last three machine cycles, two for DJNZ and one to alter the pin.

DJNZ Rn,rel

Bytes: 2

Cycles: 2

Encoding:



Operation: (PC) ← (PC) + 2

(Rn) ← (Rn) – 1

IF (Rn) ≠ 0

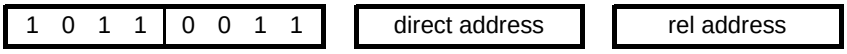
THEN (PC) ← (PC) + rel

DJNZ direct,rel

Bytes: 3

Cycles: 2

Encoding:



Operation: (PC) ← (PC) + 2

(DATA (direct)) ← (DATA (direct)) – 1

IF (DATA (direct)) ≠ 0

THEN (PC) ← (PC) + rel

ECALL Extended Call (extended instruction)

Function: Extended call

Description: ECALL performs a subroutine call to the indicated address. The instruction adds five to the program counter to generate the address of the next instruction and then pushes the 23-bit result onto the stack (low byte first), incrementing the Stack Pointer by three in the process. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. The PC is then loaded with the address from the instruction. Program execution continues with the instruction at this address. ECALL is able to call a subroutine residing anywhere in the full 16 MByte program memory address space. No flags are affected.

Note that since ECALL pushes a full 23-bit address onto the stack, the return from a subroutine called by ECALL must be accomplished with the ERET instruction, which pops a 23-bit address from the stack. A general rule is that any subroutine that is called at any point using the ECALL instruction must always be called using ECALL, and must also terminate with an ERET instruction. Mismatch of call and return types will cause stack de-synchronization and a probable program crash.

Example: Initially the Stack Pointer equals 07h. The label “SUBRTN” is assigned to program memory location 00:1234h. The following instruction, located at address 45:0123h, is executed:

```
ECALL   SUBRTN
```

The Stack Pointer contains 0Ah, internal RAM locations 08h, 09h, and 0Ah contain 28h, 01h, and 45h respectively, and the PC contains 00:1234h. In this example, ECALL must be used because LCALL would not have sufficient range to reach SUBRTN.

ECALL addr23

Bytes: 5

Cycles: (tbd)

Encoding:

A5	0 0 0 1	0 0 1 0	addr.22-16	addr.15-8	addr.7-0

Operation:

(PC) ← (PC) + 5

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.7-0)

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.15-8)

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.22-16)

(PC) ← addr23

EJMP Extended Jump (extended instruction)

Function: Extended jump

Description: EJMP causes an unconditional branch to the indicated address, by loading the PC with the address from the instruction. EJMP can branch anywhere in the code memory. No flags are affected.

Example: The label “JMPADR” is assigned to the instruction at program memory location 12:3456h. The following instruction, located at address 11:9ABCh, is executed:

```
EJMP    JMPADR
```

The PC contains 12:3456h. In this example, LJMP would not be able to complete this branch since the destination address is outside of the 64k block of code memory that follows the branch instruction.

EJMP addr23

Bytes: 5

Cycles: (tbd)

Encoding:

A5	0 0 0 0 0 0 1 0	addr.22-16	addr.15-8	addr.7-0

Operation: (PC) ← addr23

EMOV Move Data Using Universal Pointer (extended instruction)

Function: Move data to or from the Accumulator using a Universal Pointer

Description: Data is moved either to or from the Accumulator and a memory location in the Universal Memory Map addressed using one of the Universal Pointers (PR0 or PR1) plus a displacement. The displacement is in the range of 0 through 3, allowing access to up to four bytes of an item in memory without the need to update the pointer register. The addition of the displacement to the pointer register is a 24-bit addition, so the item being addressed may potentially cross a 64k boundary. No flags are affected.

Example: Registers R3, R2, and R1 (comprising PR0), contain the values 7Fh, 00h, and FFh, respectively. The following instruction is executed:

```
EMOV    A,@PR0+1
```

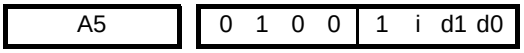
The memory location identified by the value in PR0 plus 1 is read and that value stored into the Accumulator. The address used in this case is 7F:0100h, which is in the EIDATA space, just above the top of IDATA space. The contents of PR0 remain unchanged.

EMOV A,@PRi+data2

Bytes: 2

Cycles: (tbd)

Encoding:



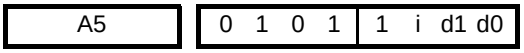
Operation: (A) ← (UMEM ((PRi) + data2))

EMOV @PRi+data2,A

Bytes: 2

Cycles: (tbd)

Encoding:



Operation: (UMEM ((PRi) + data2)) ← (A)

ERET Extended Return from Subroutine (extended instruction)

Function: Extended return from subroutine

Description: ERET pops three address bytes previously stored on the stack into the PC, decrementing the Stack Pointer by three in the process. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. Program execution continues at the resulting address, generally the instruction immediately following an ECALL occurrence. No flags are affected.

Note that since ECALL pushes a full 23-bit address onto the stack, the return from a subroutine called by ECALL must be accomplished with the ERET instruction, which pops a 23-bit address from the stack. A general rule is that any subroutine that is called at any point using the ECALL instruction must always be called using ECALL, and must also terminate with an ERET instruction. Mismatch of call and return types will cause stack de-synchronization and a probable program crash.

Example: The Stack Pointer originally contains the value 0Ch. Internal RAM locations 0Ah, 0Bh, and 0Ch contain the values 45h, 23h, and 01h respectively. The following instruction is executed:

```
ERET
```

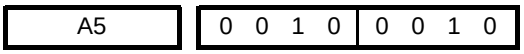
The Stack Pointer contains the value 09h. Program execution continues at location 01:2345h.

ERET

Bytes: 2

Cycles: (tbd)

Encoding:



Operation:

(PC.22-16) ← (EIDATA (SP))

(SP) ← (SP) – 1

(PC.15-8) ← (EIDATA (SP))

(SP) ← (SP) – 1

(PC.7-0) ← (EIDATA (SP))

(SP) ← (SP) – 1

INC Increment

Function: Increment the specified byte

Description: INC increments the indicated variable by 1. An original value of 0FFh will overflow to 00h. Three addressing modes are allowed: register, direct, or register-indirect. No flags are affected.

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 7Eh. Internal RAM locations 7Eh and 7Fh contain 0FFh and 40h, respectively. The following instructions are executed:

```
INC     @R0
INC     R0
INC     @R0
```

Register 0 contains 7Fh and internal RAM locations 7Eh and 7Fh are set to 00h and 41h respectively.

INC A

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 0 0

Operation: (A) ← (A) + 1

INC Rn

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	1 r r r

Operation: (Rn) ← (Rn) + 1

INC direct

Bytes: 2

Cycles: 1

Encoding:

0 0 0 0	0 1 0 1	direct address

Operation: (DATA (direct)) ← (DATA (direct)) + 1

INC @Ri

Bytes: 1

Cycles: 1

Encoding:

0 0 0 0	0 1 1 i

Operation: (IDATA (Ri)) ← (IDATA (Ri)) + 1

INC Increment Data Pointer

Function: Increment the Data Pointer (DPTR)

Description: Increment the 16-bit Data Pointer by 1. A 16-bit increment is performed. An overflow of the low-order byte of the Data Pointer (DPL) from 0FFh to 00h will increment the next byte (DPH). No flags are affected.

Example: Registers DPH, and DPL contain 0FFh, and 0FEh, respectively. The following instructions are executed:

```
INC     DPTR
INC     DPTR
INC     DPTR
```

DPH and DPL now contain the values 00h and 01h.

INC DPTR

Bytes: 1

Cycles: 2

Encoding:

1	0	1	0	0	0	1	1

Operation: (DPTR) ← (DPTR) + 1

INC Increment Extended Pointer (extended instruction)

Function: Increment the Extended Pointer (EPTR)

Description: Increment the 23-bit Extended Pointer by 1. A 23-bit increment is performed. An overflow of the low-order byte of the Extended Pointer (EPL) from 0FFh to 00h will increment the next byte (EPM), and an overflow of EPM from 0FFh to 00h will increment the upper byte (EPH). No flags are affected.

Example: Registers EPH, EPM, and EPL contain 01h, 0FFh, and 0FEh, respectively. The following instruction sequence is executed:

```
INC   EPTR
INC   EPTR
INC   EPTR
```

EPH, EPM, and EPL now contain the values 02h, 00h, and 01h.

INC EPTR

Bytes: 2

Cycles: (tbd)

Encoding:

A5	1 0 1 0	0 0 1 1

Operation: (EPTR) ← (EPTR) + 1

JB Jump if Bit Set

Function: Jump if the specified bit is set

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 the Port 1 pins is CAh (11001010b). The Accumulator holds 56 (01010110b). The following instructions are executed:

```
JB      P1.2,LABEL1
JB      ACC.2,LABEL2
```

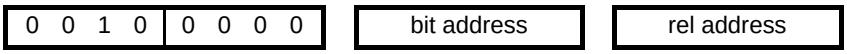
The first branch is not taken because P1.2 is a zero. The second branch is taken (to the instruction at LABEL2) since Accumulator bit 2 is a one.

JB bit,rel

Bytes: 3

Cycles: 2

Encoding:



Operation:

(PC) ← (PC) + 3

IF (bit) = 1

THEN (PC) ← (PC) + rel

JBC Jump if Bit is Set and Clear Bit

Function: Jump if the specified bit is set and clear the bit

Description: If the indicated bit is a one, branch to the address indicated; otherwise proceed with the next instruction. The bit will not be cleared if it is already a zero. 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 read from the output data latch, not the input pin.

Example: The Accumulator holds 56h (01010110b). The following instructions are executed:

```
JBC     ACC.3,LABEL1
JBC     ACC.2,LABEL2
```

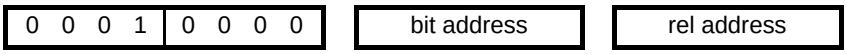
The first branch is not taken, since Accumulator bit 3 is a zero. The second branch (to the instruction at LABEL2) is taken, since Accumulator bit 2 is a 1. Accumulator bit 2 is cleared, so the Accumulator now contains 52h (01010010b).

JBC bit,rel

Bytes: 3

Cycles: 2

Encoding:



Operation:

```
(PC) ← (PC) + 3
IF (bit) = 1
        THEN    (bit) ← 0
                (PC) ← (PC) + rel
```

JC Jump if Carry is Set

Function: Jump if the Carry flag is set

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 by two. No flags are affected.

Example: The Carry flag is contains a zero. The following instructions are executed:

```
JC    LABEL1
CPL   C
JC    LABEL2
```

The first branch is not taken because the Carry flag is a zero. The second branch is taken (to the instruction at LABEL2) because the Carry flag has been complemented and now contains a one.

JC rel

Bytes: 2

Cycles: 2

Encoding:

0 1 0 0	0 0 0 0	rel address

Operation: (PC) ← (PC) + 2

IF (C) = 1

THEN (PC) ← (PC) + rel

JMP Jump Indirect using DPTR

Function: Jump indirect using the Data Pointer (DPTR)

Description: Add the eight-bit unsigned contents of the Accumulator to the contents of the 16-bit Data Pointer, and load the resulting sum into the lower 16 bits of the Program Counter, forming an address for the next instruction fetch. Sixteen-bit addition is performed: a carry-out from the low-order byte propagates through to the next byte. The replacement of the lower 16 bits of the PC occurs after the PC has been incremented to point at the next instruction in sequence. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

Example: A number from 0 to 3 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     ; Load the table address into the DPTR.
        RL    A                 ; Adjust A to account for 2-byte AJMP instructions.
        JMP   @A+DPTR           ; Jump into the table.

JMP_TBL: AJMP  LABEL0
        AJMP  LABEL1
        AJMP  LABEL2
        AJMP  LABEL3
```

This example assumes that the entire table of AJMP instructions is in the same 64 KB block of code memory as the JMP @A+DPTR instruction. If the Accumulator equals 02h when starting this sequence, execution will jump to label LABEL2. Since AJMP is a two-byte instruction, the jump instructions start at every second address in the table.

JMP @A+DPTR

Bytes: 1

Cycles: 2

Encoding:

0	1	1	1	0	0	1	1

Operation: (PC) ← (PC) + 1

(PC.15-0) ← (A) + (DPTR)

JMP Jump Indirect using EPTR (extended instruction)

Function: Jump indirect using the Extended Pointer (EPTR)

Description: Add the eight-bit unsigned contents of the Accumulator to the contents of the 23-bit Extended Pointer, and load the resulting sum into the Program Counter, forming an address for the next instruction fetch. Twenty-three bit addition is performed: a carry-out from the low-order byte propagates through the higher-order bytes. Neither the Accumulator nor the EPTR is altered. No flags are affected.

Example: A number from 0 to 3 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   EPTR,#JMP_TBL     ; Load the table address into the EPTR.
            RL    A                 ; Adjust A to account for 2-byte AJMP instructions.
            JMP   @A+EPTR           ; Jump into the table.
            ...   ...
            ...   ...
            ...   ...
JMP_TBL:    AJMP  LABEL0
            AJMP  LABEL1
            AJMP  LABEL2
            AJMP  LABEL3
```

In this example, the table of AJMP instructions may be located anywhere in the code memory. If the Accumulator equals 02h when starting this sequence, execution will jump to label LABEL2. Since AJMP is a two-byte instruction, the jump instructions start at every second address in the table.

JMP @A+EPTR

Bytes: 2

Cycles: (tbd)

Encoding:

A5	0 1 1 1	0 0 1 1

Operation: (PC) ← (A) + (EPTR)

JNB Jump if Bit is Not Set

Function: Jump if the specified bit is not set

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 CAh (11001010b). The Accumulator holds 56h (01010110b). The following instructions are executed:

```
JNB     P1.3,LABEL1
JNB     ACC.3,LABEL2
```

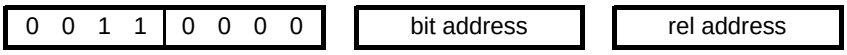
The first branch is not taken because P1.3 is a one. The second branch is taken (to the instruction at LABEL2) since Accumulator bit 3 is a zero.

JNB bit,rel

Bytes: 3

Cycles: 2

Encoding:



Operation: (PC) ← (PC) + 3

IF (bit) = 0

THEN (PC) ← (PC) + rel

JNC Jump if Carry Not Set

Function: Jump if the Carry flag is not set

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 by two to point to the next instruction. No flags are affected.

Example: The Carry flag is contains a one. The following instruction sequence is executed:

```
JNC   LABEL1
CPL   C
JNC   LABEL2
```

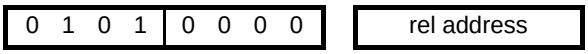
The first branch is not taken because the Carry flag is a one. The second branch is taken (to the instruction at LABEL2) because the Carry flag has been complemented and now contains a zero.

JNC rel

Bytes: 2

Cycles: 2

Encoding:



Operation: (PC) ← (PC) + 2

IF (C) = 0

THEN (PC) ← (PC) + rel

JNZ Jump if Accumulator is Not Zero

Function: Jump if the Accumulator is not equal to zero

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 by two. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally holds 00h. The following instruction sequence is executed:

```
JNZ   LABEL1
INC   A
JNZ   LABEL2
```

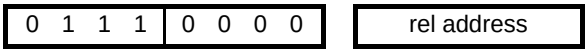
The first branch is not taken because the Accumulator contains a zero. The second branch is taken (to the instruction at LABEL2) because the Accumulator has been incremented and no longer contains a zero.

JNZ rel

Bytes: 2

Cycles: 2

Encoding:



Operation: (PC) ← (PC) + 2

IF A ≠ 0

THEN (PC) ← (PC) + rel

JZ Jump if Accumulator is Zero

Function: Jump if the Accumulator is equal to zero

Description: If all bits of the Accumulator are 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 second instruction byte to the PC, after incrementing the PC by two. The Accumulator is not modified. No flags are affected.

Example: The Accumulator originally holds 01h. The following instruction sequence is executed:

```
JZ    LABEL1
DEC   A
JZ    LABEL2
```

The first branch is not taken because the Accumulator does not contain a zero. The second branch is taken (to the instruction at LABEL2) because the Accumulator has been decremented and now contains a zero.

JZ rel

Bytes: 2

Cycles: 2

Encoding:

0 1 1 0	0 0 0 0	rel address

Operation: (PC) ← (PC) + 2

IF A = 0

THEN (PC) ← (PC) + rel

LCALL Long Call

Function: Long call

Description: LCALL performs a subroutine call to the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the lower 16-bits of the result onto the stack, incrementing the Stack Pointer by two in the process. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. The lower 16-bits of the PC are then loaded with the address from the instruction. Program execution continues with the instruction at this address. LCALL can call a subroutine located anywhere in the 64 KB block of code memory that contains the first byte of the instruction following LCALL. No flags are affected.

Note that the ACALL and LCALL instruction only push two bytes of address information onto the stack. For this reason, ERET must not be used to return from subroutines that were called using those instructions. A general rule is that any subroutine that must called at any time using the ECALL instruction must always be called using ECALL, and must also terminate with an ERET instruction. Mismatch of call and return types will cause stack de-synchronization and a probable program crash.

Example: Initially the Stack Pointer equals 07h. The label “SUBRTN” is assigned to program memory location 00:1234h. The following instruction, located at address 00:54A0h, is executed:

```
LCALL   SUBRTN
```

The Stack Pointer will contain 09h, internal RAM locations 08h and 09h will contain A3h and 54h respectively, and the PC will contain 00:1234h.

LCALL addr16

Bytes: 3

Cycles: 2

Encoding:

0 0 0 1	0 0 1 0	addr15-addr8	addr7-addr0

Operation:

(PC) ← (PC) + 3

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.7-0)

(SP) ← (SP) + 1

(EIDATA (SP)) ← (PC.15-8)

(PC.15-0) ← addr16

LJMP Long Jump

Function: Long jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the PC with the address from the instruction. LJMP can anywhere in the 64 KB block of code memory that contains the first byte of the instruction following LJMP. No flags are affected.

Example: The label “JMPADR” is assigned to the instruction at program memory location 00:1234h. The following instruction, located at address 00:0123h, is executed:

```
LJMP    JMPADR
```

The program counter contains 00:1234h. In this example, the shorter form of the instruction is sufficient to make the branch since the destination address is within the same 64k block of code memory as the instruction following the LJMP.

LJMP addr16

Bytes: 3

Cycles: 2

Encoding:

0 0 0 0	0 0 1 0	addr15-addr8	addr7-addr0

Operation: (PC) ← (PC) + 3

(PC.15-0) ← addr16

MOV Move Byte

Function: Move data 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. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed. No flags are affected.

Example: Internal RAM location 30h holds 40h, and RAM location 40h contains 10h. The data present at input port 1 is 0CAh. The following instruction sequence is executed:

```
MOV   R0,#30h      ; R0 = 30h
MOV   A,@R0        ; A = 40h
MOV   R1,A         ; R1 = 40h
MOV   B,@R1        ; B = 10h
MOV   @R1,P1       ; RAM(40h) = 0CAh
MOV   P2,P1        ; P2 = 0CAh
```

Resulting values are indicated in the comments above.

MOV A,Rn

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	1	r	r	r

Operation: (A) ← (Rn)

***MOV A,direct**

Bytes: 2

Cycles: 1

Encoding:

1	1	1	0	0	1	0	1	direct address

Operation: (A) ← (DATA (direct))

*MOV A,ACC is not a valid instruction.

MOV A,@Ri

Bytes: 1

Cycles: 1

Encoding:

1	1	1	0	0	1	1	i

Operation: (A) ← (IDATA (Ri))

MOV A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 0 0	immediate data

Operation: (A) ← #data

MOV Rn,A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	1 r r r

Operation: (Rn) ← (A)

MOV Rn,direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	1 r r r	direct address

Operation: (Rn) ← (DATA (direct))

MOV Rn,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	1 r r r	immediate data

Operation: (Rn) ← #data

MOV direct,A

Bytes: 2

Cycles: 1

Encoding:

1 1 1 1	0 1 0 1	direct address

Operation: (DATA (direct)) ← (A)

MOV direct,Rn

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	1 r r r	direct address

Operation: (DATA (direct)) ← (Rn)

MOV direct,direct

Bytes: 3

Cycles: 2

Encoding:

1 0 0 0	0 1 0 1	dir addr (src)	dir addr (dest)

Operation: (DATA (direct)) ← (DATA (direct))

MOV direct,@Ri

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 1 1 i	direct address

Operation: (DATA (direct)) ← (IDATA (Ri))

MOV direct,#data

Bytes: 3

Cycles: 2

Encoding:

0 1 1 1	0 1 0 1	direct address	immediate data

Operation: (DATA (direct)) ← #data

MOV @Ri,A

Bytes: 1

Cycles: 1

Encoding:

1 1 1 1	0 1 1 i

Operation: (IDATA (Ri)) ← (A)

MOV @Ri,direct

Bytes: 2

Cycles: 2

Encoding:

1 0 1 0	0 1 1 i	direct address

Operation: (IDATA (Ri)) ← (DATA (direct))

MOV @Ri,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 1	0 1 1 i	immediate data

Operation: (IDATA (Ri)) ← #data

MOV Move Bit

Function: Move the specified bit to or from the Carry flag

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 initially set. The data present at input Port 3 is C5h (11000101b). The data previously written to output Port 1 is 35h (00110101b). The following instruction sequence is executed:

```
MOV  P1.3,C
MOV  C,P3.3
MOV  P1.2,C
```

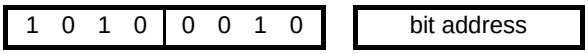
The Carry flag is cleared and Port 1 contains 39h (00111001b).

MOV C,bit

Bytes: 2

Cycles: 1

Encoding:



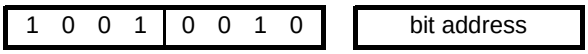
Operation: (C) ← (bit)

MOV bit,C

Bytes: 2

Cycles: 2

Encoding:



Operation: (bit) ← (C)

MOV Move Value to Data Pointer

Function: Move an immediate value to the Data Pointer (DPTR)

Description: The Data Pointer is loaded with the constant indicated. A 16-bit constant is loaded into the active Data Pointer. No flags are affected.

Example: The following instruction is executed:

```
MOV     DPTR,#1234h
```

The value 1234h is loaded into the Data Pointer: DPH contains 12h and DPL contains 34h.

MOV DPTR,#data16

Bytes: 3

Cycles: 2

Encoding:

Operation: (DPH) ← #data.15-8

(DPL) ← #data.7-0

MOV Move Value to Extended Pointer (extended instruction)

Function: Move immediate data value to the Extended Pointer (EPTR)

Description: The Extended Pointer is loaded with the constant indicated. A 23-bit constant is loaded into the Extended Pointer. No flags are affected.

Example: The following instruction is executed:

```
MOV     EPTR,#0AB1234h
```

The value AB:1234h is loaded into the Extended Pointer: EPH contains ABh, EPM contains 12h, and EPL contains 34h.

MOV EPTR,#data23

Bytes: 5

Cycles: (tbd)

Encoding:

A5	1 0 0 1 0 0 0 0	#data.22-16	#data.15-8	#data.7-0

Operation: (EPH) ← #data.22-16

(EPM) ← #data.15-8

(EPL) ← #data.7-0

MOVC Move Code Byte (extended instruction)

Function: Move a Code byte to the Accumulator

Description: The MOVC instructions load the Accumulator with a byte from the program memory. The address of the byte fetched is the sum of the original unsigned eight-bit Accumulator contents and the contents of a base register, which may be either the 16-bit Data Pointer, the 23-bit EPTR, or the 23-bit PC. In the last case, the PC is incremented to the address of the following instruction before being added with the Accumulator. Twenty-three bit addition is performed so a carry-out from the low-order eight bits may propagate through to 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 01h, the value returned in the Accumulator is 77h. The INC A before the MOVC instruction is needed to account for the length of the RET instruction that occurs between the MOVC and the actual start of the lookup table. If several bytes of code separated the MOVC instruction from the table, the adjustment added to the Accumulator must be altered accordingly. It is typically possible to get an assembler to perform this adjustment automatically as shown below.

```
REL_PC: ADD     A,#Table - Lookup - 1   ; Calculate adjustment to table index (within a range of FFh).
Lookup: MOVC    A,@A+PC
        ...     ...                     ; Other code, may be several instructions.
        RET
Table:  DB      66h
        DB      77h
        DB      88h
        DB      99h
```

MOVC A,@A+DPTR

Bytes: 1

Cycles: 2

Encoding:

1	0	0	1	0	0	1	1

Operation: (A) ← (CODE ((PC.22-16) : (A) + (DPTR)))

MOVC A,@A+EPTR

Bytes: 2

Cycles: (tbd)

Encoding:

A5

1	0	0	1	0	0	1	1

Operation: (A) ← (CODE ((A) + (EPTR)))

MOVC A,@A+PC

Bytes: 1

Cycles: 2

Encoding:

1	0	0	0	0	0	1	1

Operation: (PC) ← (PC) + 1

(A) ← (CODE ((A) + (PC)))

MOVX Move External Data Byte (extended instruction)

Function: Move a byte to or from an XDATA location and the Accumulator

Description: The MOVX instructions transfer data between the Accumulator and a byte of external data memory (XDATA). There are several addressing modes available. No flags are affected.

In the first type of MOVX instruction, the contents of R0 or R1 in the current register bank provide an eight-bit address which is used to address a byte in the XDATA memory area. This address may map to an on-chip or off-chip location, if the specific 51MX derivative implements any on-chip XDATA memory and whether that memory enabled. If the address is off-chip, the upper address lines (bits 22 through 8) are not driven, the port pins corresponding to those line remain in their previous state. If the address is on-chip, the upper address bits are treated as zeroes.

Eight bits of address are sufficient for external I/O device expansion or for a small amount of memory. For larger memories, port pins could be used to output higher-order address bits. These pins would be controlled by a write to another port prior to executing the MOVX.

In the second type of MOVX instruction, The active Data Pointer provides a 16-bit address. The upper address bits (22 through 16) are always zeroes. If the address corresponds to an off-chip XDATA location, the external bus will present the address in the standard fashion, using the number of address bits that the device is configured for.

The third type of MOVX instruction is similar, but uses the Extended Pointer to provide a full 23-bit address. If the address corresponds to an off-chip XDATA location, the external bus will present the address in the standard fashion, using the number of address bits that the device is configured for.

Examples: 1) Registers 0 and 1 contain 12h and 34h respectively. The following instruction sequence is executed:

```
MOVX  A,@R1
MOVX  @R0,A
```

The value from XDATA address 34h is copied into both the Accumulator and XDATA location 12h. If the access occurs off-chip, the values of the port pins that correspond to the upper address lines (bits 22 through 8) may also be decoded externally.

2) The active DPTR contains 0399h. The XDATA location 00:0399h contains the value ABh. The following instruction is executed:

```
MOVX  A,@DPTR
```

The Accumulator contains the value ABh.

3) The EPTR contains 7C:2B01h and the Accumulator contains the value CDh. The following instruction is executed:

```
MOVX  @EPTR,A
```

The value CDh is written to XDATA location 7C:2B01h.

MOVX A,@Ri

Bytes: 1

Cycles: 2

Encoding:

1	1	1	0	0	0	1	i

Operation: (A) ← (XDATA (Ri))

MOVX @Ri,A

Bytes: 1

Cycles: 2

Encoding:

1	1	1	1	0	0	1	i

Operation: (XDATA (Ri)) ← (A)

MOVX A,@DPTR

Bytes: 1

Cycles: 2

Encoding:

1	1	1	0	0	0	0	0

Operation: (A) ← (XDATA (DPTR))

MOVX @DPTR,A

Bytes: 1

Cycles: 2

Encoding:

1	1	1	1	0	0	0	0

Operation: (XDATA (DPTR)) ← (A)

MOVX A,@EPTR

Bytes: 2

Cycles: (tbd)

Encoding:

A5

1	1	1	0	0	0	0	0

Operation: (A) ← (XDATA (EPTR))

MOVX @EPTR,A

Bytes: 2

Cycles: (tbd)

Encoding:

A5

1	1	1	1	0	0	0	0

Operation: (XDATA (EPTR)) ← (A)

MUL Multiply

Function: Multiply the Accumulator by the B register

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 (0FFh) the overflow flag is set; otherwise it is cleared. The Carry flag is always cleared.

Example: Originally the Accumulator holds the value 80 (50h). The B register holds the value 160 (0A0h).The instruction,

```
MUL   AB
```

will give the product 12,800 (3200h), so the B register is changed to 32h (00110010b) and the Accumulator is cleared. The overflow flag is set, the Carry flag is cleared.

MUL AB

Bytes: 1

Cycles: 4

Encoding:

1	0	1	0	0	1	0	0

Operation: (B) ← high byte of (A) x (B)

(A) ← low byte of (A) x (B)

NOP No Operation

Function: No Operation

Description: Execution continues at the following instruction. No registers or flags are affected.

Example: It is desired to produce a low-going output pulse on bit 7 of Port 2 lasting 5 machine cycles. A simple SETB/CLR sequence would generate a one machine cycle pulse, so four additional machine cycles must be inserted. This may be done (assuming interrupts are disabled or not used) with the instruction sequence,

```
CLR   P2.7
NOP
NOP
NOP
NOP
SETB  P2.7
```

NOP

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	0	0

Operation: (PC) ← (PC) + 1

ORL Logical OR Byte

Function: Logical OR the specified bytes

Description: ORL performs the bitwise logical-OR operation between the indicated variables, storing the results in the destination byte. No flags are affected.

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 0C3h (11000011b) and R0 holds 55h (01010101b) then the instruction,

```
ORL   A,R0
```

will leave the Accumulator holding the value 0D7h (11010111b).

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,#00110010b
```

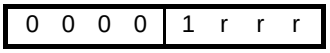
will set bits 5, 4, and 1 of Port 1.

ORL A,Rn

Bytes: 1

Cycles: 1

Encoding: 0 0 0 0 1 r r r



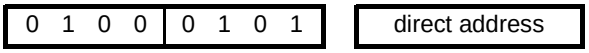
Operation: (A) ← (A) OR (Rn)

ORL A,direct

Bytes: 2

Cycles: 1

Encoding:



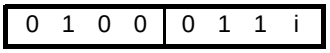
Operation: (A) ← (A) OR (DATA (direct))

ORL A,@Ri

Bytes: 1

Cycles: 1

Encoding:



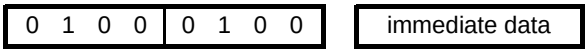
Operation: (A) ← (A) OR (IDATA (Ri))

ORL A,#data

Bytes: 2

Cycles: 1

Encoding:



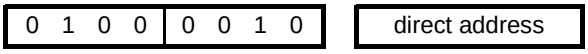
Operation: (A) ← (A) OR #data

ORL direct,A

Bytes: 2

Cycles: 1

Encoding:



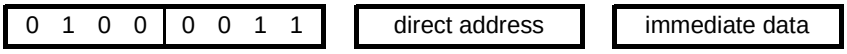
Operation: (DATA (direct)) ← (DATA (direct)) OR (A)

ORL direct,#data

Bytes: 3

Cycles: 2

Encoding:



Operation: (DATA (direct)) ← (DATA (direct)) OR #data

ORL Logical OR Bit

Function: Logical OR the specified bit with the Carry flag

Description: Set the Carry flag if the Boolean value is a logical 1; leave the Carry flag in its current state otherwise. A forward 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 Port 1 bit 0 = 1, Accumulator bit 7 = 1, or the OV flag = 0:

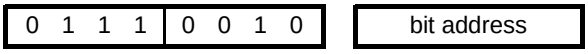
```
ORL   C,P1.0      ; Load Carry with input pin P1 bit 0
ORL   C,ACC.7     ; OR Carry with Accumulator bit 7
ORL   C,/OV       ; OR Carry with the inverse of the OV flag.
```

ORL C,bit

Bytes: 2

Cycles: 2

Encoding:



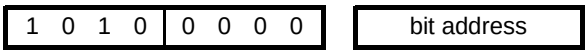
Operation: (C) ← (C) OR (bit)

ORL C,/bit

Bytes: 2

Cycles: 2

Encoding:



Operation: (C) ← (C) OR $\overline{(bit)}$

POP Pop Data from Stack

Function: Pop a data byte from the stack

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 then transferred to the directly addressed byte indicated. No flags are affected.

Example: The Stack Pointer originally contains the value 32h, and internal RAM locations 31h and 32h contain the values 23h and 01h respectively. The following instruction sequence is executed:

```
POP   DPH
POP   DPL
```

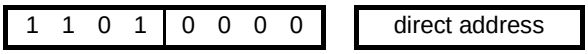
The Stack Pointer contains the value 30h and the Data Pointer is set to 0123h.

POP direct

Bytes: 2

Cycles: 2

Encoding:



Operation: (DATA (direct)) ← (EIDATA (SP))

(SP) ← (SP) – 1

PUSH Push Data onto Stack

PUSH direct

Function: Push a data byte onto the stack

Description: The Stack Pointer is incremented by one. The contents of the indicated variable is then copied into the internal RAM location addressed by the Stack Pointer. No flags are affected.

Example: Upon entering an interrupt routine the Stack Pointer contains 09h. The Data Pointer holds the value 0123h. The following instruction sequence is executed:

```
PUSH  DPL
PUSH  DPH
```

The Stack Pointer contains 0Bh and internal RAM locations 0Ah and 0Bh contain 23h and 01h.

PUSH direct

Bytes: 2

Cycles: 2

Encoding:

1 1 0 0	0 0 0 0	direct address

Operation: (SP) ← (SP) + 1

(EIDATA (SP)) ← (DATA (direct))

RET Return from Subroutine (extended instruction)

Function: Return from subroutine

Description: RET pops two address bytes previously stored on the stack into the lower 16 bits of the PC, decrementing the Stack Pointer by two in the process. The PC is incremented prior to replacing the lower 16-bits. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL occurrence. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. No flags are affected.

Note that the ACALL and LCALL instruction only push two bytes of address information onto the stack. For this reason, ERET must not be used to return from subroutines that were called using those instructions. A general rule is that any subroutine that must called at any time using the ECALL instruction must always be called using ECALL, and must also terminate with an ERET instruction. Mismatch of call and return types will cause stack de-synchronization and a probable program crash.

Example: The Stack Pointer originally contains the value 0Bh. Internal RAM locations 0Ah and 0Bh contain the values 45h and 23h respectively. The following instruction, located at address 67:ABCD, is executed:

```
RET
```

The Stack Pointer contains the value 09h. Program execution continues at address 67:2345h.

RET

Bytes: 1

Cycles: 2

Encoding:

0	0	1	0	0	0	1	0

Operation:

(PC) ← (PC) + 1

(PC.15-8) ← (EIDATA (SP))

(SP) ← (SP) – 1

(PC.7-0) ← (EIDATA (SP))

(SP) ← (SP) – 1

RETI Return from Interrupt (extended instruction)

Function: Return from interrupt

Description: RETI pops three address bytes previously stored on the stack into the PC, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer (which is a word 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 has been pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed. The stack may reside in the IDATA space or anywhere in the entire EIDATA space, depending on the stack mode. No flags are affected.

Example: The Stack Pointer originally contains the value 0Bh. Previously, an interrupt was detected during the instruction ending at location 01:2344h. Internal RAM locations 09h, 0Ah and 0Bh contain the values 45h, 23h, and 01h respectively. The following instruction is executed:

```
RETI
```

The Stack Pointer contains 08h and program execution continues at location 01:2345h.

RETI

Bytes: 1

Cycles: 2 when (MXCON.EIFM) = 0

(tbd) when (MXCON.EIFM) = 1

Encoding:

0	0	1	1	0	0	1	0

Operation:

```
IF (MXCON.EIFM) = 0
        THEN    (PC.15-8) ← ( EIDATA (SP) )
                (SP) ← (SP) – 1
                (PC.7-0) ← ( EIDATA (SP) )
                (SP) ← (SP) – 1
        ELSE    (PC.22-16) ← ( EIDATA (SP) )
                (SP) ← (SP) – 1
                (PC.15-8) ← ( EIDATA (SP) )
                (SP) ← (SP) – 1
                (PC.7-0) ← ( EIDATA (SP) )
                (SP) ← (SP) – 1
```

Note: as indicated above, the affect of RETI is dependent on the stack mode.

RL Rotate Left

Function: Rotate the Accumulator left one position

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 0C5h (11000101b). The following instruction is executed:

```
RL   A
```

The Accumulator contains the value 8Bh (10001011b). The Carry flag is unaffected.

RL A

Bytes: 1

Cycles: 1

Encoding:

0	0	1	0	0	0	1	1

Operation: $(A.n+1) \leftarrow (A.n)$, n = 0 to 6

$(A.0) \leftarrow (A.7)$

RLC Rotate Left through Carry Flag

Function: Rotate the Accumulator with the Carry flag left one position

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 0C5h (11000101b), and the Carry flag is zero. The following instruction is executed:

```
RLC   A
```

The Accumulator contains the value 8Ah (10001010b). The Carry flag is set.

RLC A

Bytes: 1

Cycles: 1

Encoding:

0	0	1	1	0	0	1	1

Operation: $(A.n+1) \leftarrow (A.n)$, n = 0 to 6

$(A.0) \leftarrow (C)$

$(C) \leftarrow (A.7)$

RR Rotate Right

Function: Rotate the Accumulator right one position

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 0C5h (11000101b). The following instruction is executed:

```
RR      A
```

The Accumulator contains the value 0E2h (11100010b). The Carry flag is unaffected.

RR A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	0	0	0	1	1

Operation: $(A.n) \leftarrow (A.n+1)$, n = 0 to 6

$(A.7) \leftarrow (A.0)$

RRC Rotate Right through Carry Flag

Function: Rotate the Accumulator with the Carry flag right one position

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 state of the Carry flag moves into the bit 7 position. No other flags are affected.

Example: The Accumulator holds the value 0C5h (11000101b), and the Carry flag is zero. The following instruction is executed:

```
RRC  A
```

The Accumulator contains the value 62 (01100010b). The Carry flag is set.

RRC A

Bytes: 1

Cycles: 1

Encoding:

0	0	0	1	0	0	1	1

Operation: $(A.n) \leftarrow (A.n+1)$, n = 0 to 6

$(A.7) \leftarrow (C)$

$(C) \leftarrow (A.0)$

SETB Set Bit

Function: Set the specified bit

Description: SETB sets the indicated bit to one. SETB can operate on the Carry flag or any directly addressable bit. No other flags are affected.

Example: The Carry flag is cleared. Port 1 contains the value 34h (00110100b). The following instructions are executed:

```
SETB  C
SETB  P1.0
```

The Carry flag is set to 1 and Port 1 contains 35h (00110101b).

SETB C

Bytes: 1

Cycles: 1

Encoding:

1 1 0 1	0 0 1 1	bit address

Operation: (C) ← 1

SETB bit

Bytes: 2

Cycles: 1

Encoding:

1 1 0 1	0 0 1 0	bit address

Operation: (bit) ← 1

SJMP Short Jump

Function: Short jump

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 by two. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it. No flags are affected.

Example: The label “RELADR” is assigned to an instruction at program memory location 00:0123h. The following instruction, located at address 00:0100h is executed:

```
SJMP    RELADR
```

The PC contains 000123h.

(Note: Under the above conditions the instruction following SJMP will be at 102h. Therefore, the displacement byte of the instruction will be the relative offset (00:0123h-00:0102h) = 21h. Another example is that an SJMP with a displacement of 0FEh would act as a single instruction loop.)

SJMP rel

Bytes: 2

Cycles: 2

Encoding:

1 0 0 0	0 0 0 0	rel address

Operation: (PC) ← (PC) + 2

(PC) ← (PC) + rel

SUBB Subtract with Borrow

Function: Subtract the specified byte and the Carry flag from the Accumulator

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 flag 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 0C9h (11001001b), register 2 holds 54h (01010100b), and the Carry flag is set. The following instruction is executed:

```
SUBB    A,R2
```

The Accumulator contains the value 74h (01110100b). The Carry flag and AC are cleared, the OV flag is set.

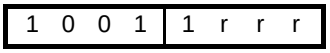
Notice that 0C9h minus 54h is 75h The difference between this and the above result is due to the Carry flag causes a borrow since it is set prior to the operation. If the state of the Carry flag is not known prior to executing a subtract operation, it should be explicitly cleared by a CLR C instruction

SUBB A,Rn

Bytes: 1

Cycles: 1

Encoding:



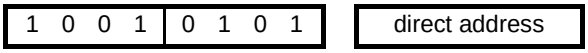
Operation: (A) ← (A) – (C) – (Rn)

SUBB A,direct

Bytes: 2

Cycles: 1

Encoding:



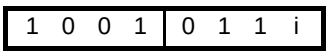
Operation: (A) ← (A) – (C) – (DATA (direct))

SUBB A,@Ri

Bytes: 1

Cycles: 1

Encoding:



Operation: (A) ← (A) – (C) – (IDATA (Ri))

SUBB A,#data

Bytes: 2

Cycles: 1

Encoding:

1	0	0	1	0	1	0	0	immediate data

Operation: (A) ← (A) – (C) – #data

SWAP Swap Nibbles in Accumulator

Function: Swap the upper and lower nibbles of the Accumulator

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 0C5h (11000101b). The following instruction is executed:

```
SWAP    A
```

The Accumulator contains the value 5Ch (01011100b).

SWAP A

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	0	0

Operation: (A.3-0) $\leftrightarrow$ (A.7-4)

XCH Exchange Byte

Function: Exchange the Accumulator with the specified byte

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. No flags are affected.

Example: R0 contains the address 20h. The Accumulator holds the value 3Fh. Internal RAM location 20h holds the value 75h. The following instruction is executed:

```
XCH   A,@R0
```

RAM location 20h contains the value 3Fh, and the Accumulator contains the value 75h.

XCH A,Rn

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	1	r	r	r

Operation: (A) ↔ (Rn)

XCH A,direct

Bytes: 2

Cycles: 1

Encoding:

1	1	0	0	0	1	0	1	direct address

Operation: (A) ↔ (DATA (direct))

XCH A,@Ri

Bytes: 1

Cycles: 1

Encoding:

1	1	0	0	0	1	1	i

Operation: (A) ↔ (IDATA (Ri))

XCHD Exchange Digit

Function: Exchange the lower nibble of the Accumulator with the lower nibble of the specified data

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 20h. The Accumulator holds the value 36h (00110110b). Internal RAM location 20h holds the value 75h (01110101b). The following instruction is executed:

```
XCHD    A,@R0
```

RAM location 20h contains the value 76h (01110110b) and the Accumulator contains the value 35h (00110101b).

XCHD A,@Ri

Bytes: 1

Cycles: 1

Encoding:

1	1	0	1	0	1	1	i

Operation: (A.3-0) $\leftrightarrow$ (IDATA(Ri).3-0)

XRL Logical Exclusive-Or Byte

Function: Logical Exclusive-OR the specified bytes

Description: XRL performs the bitwise logical Exclusive-OR operation between the indicated variables, storing the results in the destination. No flags are affected.

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 0C3h (11000011b) and register 0 holds 0AAh (10101010b) then the instruction,

```
XRL   A,R0
```

will leave the Accumulator holding the value 69h (01101001b).

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 Port 1.

XRL A,Rn

Bytes: 1

Cycles: 1

Encoding:

0	1	1	0	1	r	r	r

Operation: (A) ← (A) XOR (Rn)

XRL A,direct

Bytes: 2

Cycles: 1

Encoding:

0	1	1	0	0	1	0	1	direct address

Operation: (A) ← (A) XOR (DATA (direct))

XRL A,@Ri

Bytes: 1

Cycles: 1

Encoding:

0	1	1	0	0	1	1	i

Operation: (A) ← (A) XOR (IDATA (Ri))

XRL A,#data

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 1 0 0	immediate data

Operation: (A) ← (A) XOR #data

XRL direct,A

Bytes: 2

Cycles: 1

Encoding:

0 1 1 0	0 0 1 0	direct address

Operation: (DATA (direct)) ← (DATA (direct)) XOR (A)

XRL direct,#data

Bytes: 3

Cycles: 2

Encoding:

0 1 1 0	0 0 1 1	direct address	immediate data

Operation: (DATA (direct)) ← (DATA (direct)) XOR #data