Architectural Overview of the C51 Family

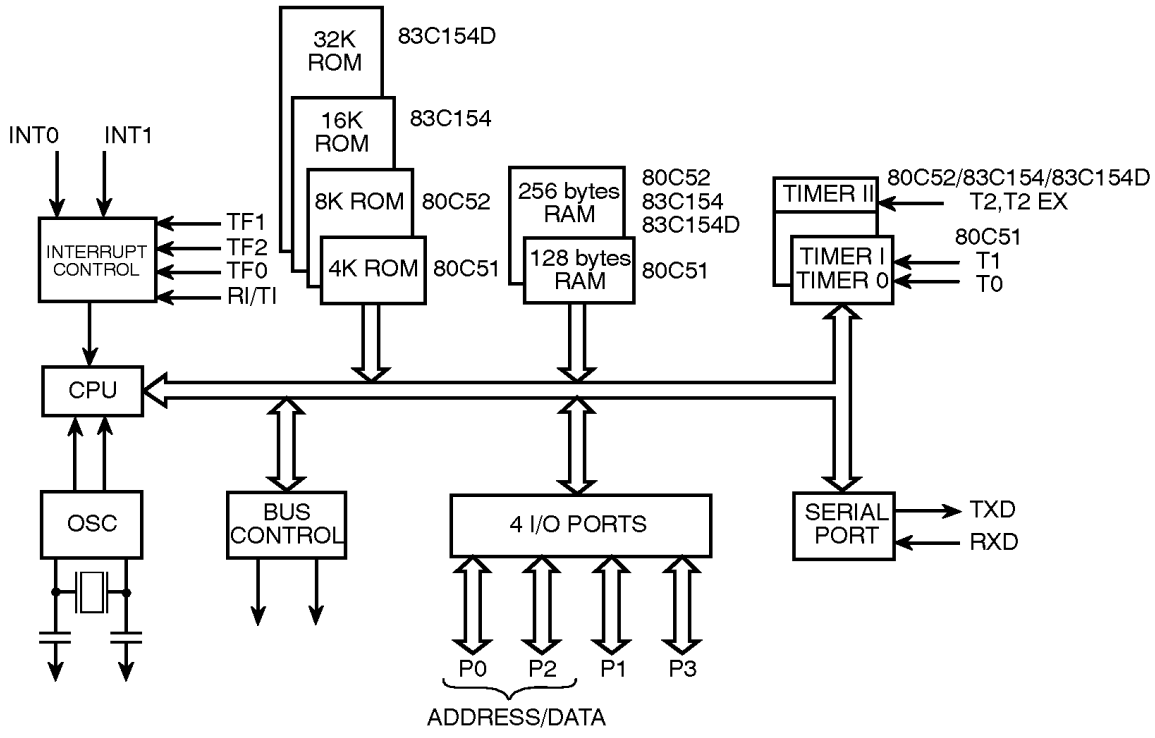
Introduction

The MHS C51 microcontroller family is based on the 80C51 core which features are :

- 8-bit CPU optimized for control applications
- Extensive boolean processing (single-bit logic) capabilities
- 64 K Program Memory address space
- 64 K Data Memory address space
- 4 K bytes of on chip Program Memory
- 128 bytes of on chip data RAM
- 32 bidirectionnal and individually addressable I/O lines
- two 16-bit timers/counters
- Full duplex UART
- 6 sources / 5-vector interrupt structure with 2 priority levels
- on chip clock oscillators

The basic architectural structure of the C51 microcontroller family is shown in figure 1.

Figure 1. Block Diagram



C51 Family

Each device of the C51 family is listed in Table 1.

Table 1 : C51 Family of Microcontrollers.

DEVICE NAME	ROMLESS VERSION	ROM BYTES	RAM BYTES	16-BIT TIMERS	SPEED UP TO	PROCESS
80C51	80C31	4 K	128	2	42 MHz	CMOS
80C52	80C32	8 K	256	3	44 MHz	CMOS
83C154	80C154	16 K	256	3	36 MHz	CMOS
83C154D	—	32 K	256	3	36 MHz	CMOS
80C51PX	—	—	128/256	2/3	12 MHz	CMOS

80C51/80C31

The 80C51 is the CMOS version of the 8051. Functionally, it is fully compatible with the 8051, but being CMOS it draws less current than its HMOS counterpart. To further exploit the power savings available in CMOS circuitry, two reduced power modes are added ;

- Software-invoked Idle Mode, during which the CPU is turned off while the RAM and other onchip peripherals continue operating. In this mode, current draw is reduced to about 15 % of the current drawn when the device is fully active.
- Software-invoked Power Down Mode, during which all on-chip activities are suspensed. The on-chip RAM continues to hold its data. In this mode the device typically draws less than 10 µA.

Although the 80C51 is functionally compatible with its HMOS counterpart, specific differences between the two types of devices must be considered in the design of an application circuit if one wishes to ensure complete interchangeability between the HMOS and CMOS devices.

The ROMless version of the 80C51 is the 80C31.

80C52/80C32

The 80C52 is an enhanced 80C51. It is produced with CMOS technology, and is compatible with the 80C51. Its enhancements over the 80C51 are as follows :

- 256 bytes of on-chip RAM
- Three timer/counters
- 6-source interrupt structure
- 8 K bytes of on-chip Program ROM

The ROMless version of the 80C52 is the 80C32.

83C154/80C154

The 83C154 is an enhanced 80C52. It is produced with CMOS technology, and is compatible with the 80C51 and 80C52. Its enhancements over the 80C51 are as follows :

- 256 bytes of on-chip data RAM
- Three timer/counters (included watchdog and 32 bits timer/counters)
- 6 source interrupt structure
- Serial reception error detection
- New modes of power reduction consumption
- Programmable impedance port
- 16 K bytes of on-chip ROM
- Asynchronous Counter/Serial port mode during power-down

The ROMless version of the 83C154 is the 80C154.

83C154D

The 83C154D is an enhanced 80C154. It is produced with CMOS technology, and is compatible with the 83C154. Its enhancements over the 80C51 are as follows :

- 256 bytes of on-chip data RAM
- Three timer/counters (included watchdog and 32 bits timer/counters)
- 6 source interrupt structure
- Serial reception error detection
- New modes of power reduction consumption
- Programmable impedance port
- 32 K bytes of on-chip ROM
- Asynchronous Counter/Serial port mode during power-down.

80C51PX

- The MHS 80C51PX are PIGGY BACK devices for all the MHS microcontrollers. 4 Different versions are available :
- 80C51P4 for the 80C51 (4 K ROM)
- 80C51P8 for the 80C52 (8 K ROM)
- 80C51P16 for the 80C154 (16 K ROM)
- 80C51P32 for the 80C154D (32 K ROM)

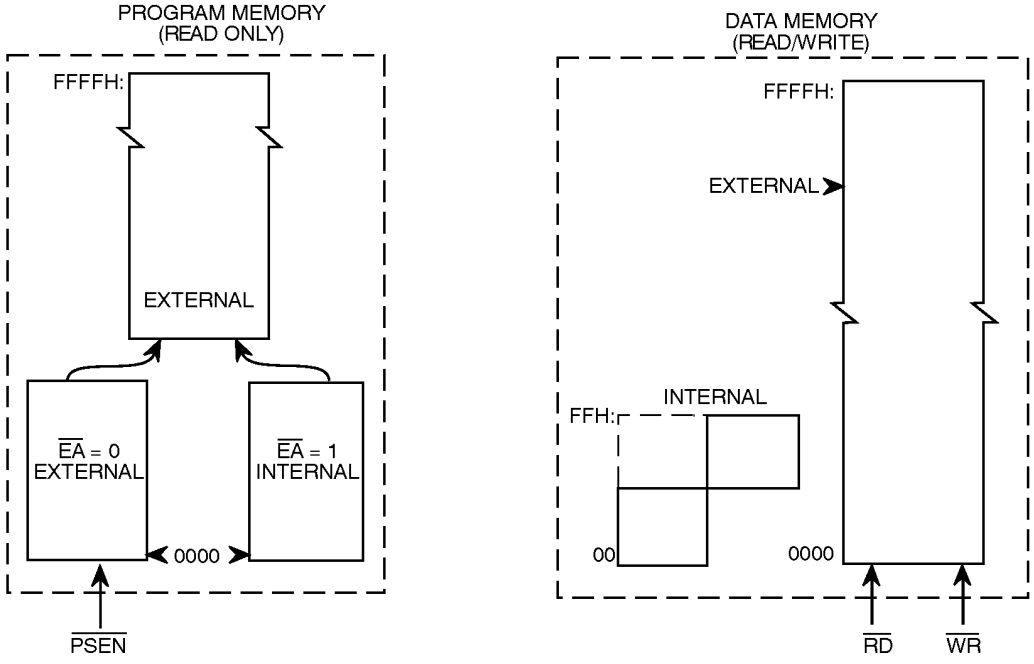
Memory Organization in C51 Devices

Logical Separation Of Program And Data Memory

All C51 devices have separated address spaces for program and Data Memory, as shown in figure 2. The logical separation of Program and Data Memory allows the Data Memory to be accessed by 8-bit addresses, which can be more quickly stored and manipulated by an 8-bit CPU. Nevertheless, 16-bit Data Memory addresses can also be generated through the DPTR register.

Program Memory can only be read, not written to. There can be up to 64 K bytes of program Memory. In the ROM versions of these devices the lowest 4 K, 8 K, 16 K or 32 K bytes of Program Memory are provided on-chip. Refer to Table 1 for the amount of on-chip ROM, on each device. In the ROMless versions all Program Memory is external. The read strobe for external Program Memory is the signal $\overline{PSEN}$ (Program Store Enable).

Figure 2. MHS C51 Memory Structure.



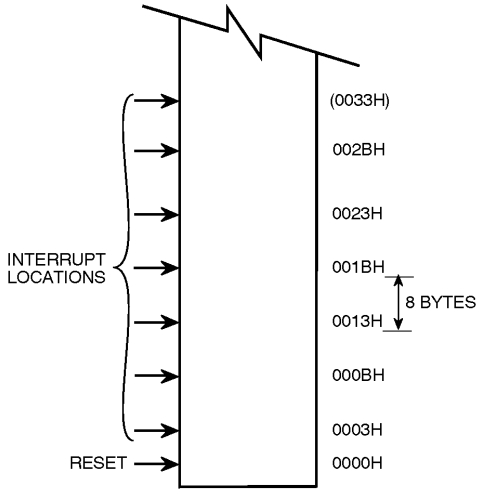
Data Memory occupies a separate address space from Program Memory. Up to 64 K bytes of external RAM can be addressed in the external Data Memory space. The CPU generates read and write signals, $\overline{RD}$ and $\overline{WR}$, as needed during external Data Memory accesses. External Program Memory and external Data Memory may be combined if desired by applying the $\overline{RD}$ and $\overline{PSEN}$ signals to the inputs of an AND gate and using the output of the gate as the read strobe to the external Program/Data memory.

Program Memory

Figure 3 shows a map of the lower part of the Program Memory. After reset, the CPU begins execution from location 0000H.

As shown in Figure 3, each interrupt is assigned a fixed location in Program Memory. The interrupt causes the CPU to jump to that location, where it commences execution of the service routine. External Interrupt 0, for example, is assigned to location 0003H. If External Interrupt 0 is going to be used, its service routine must begin at location 0003H. If the interrupt is not going to be used, its service location is available as general purpose Program Memory.

Figure 3. C51 Program Memory.



The interrupt service locations are spaced at 8-byte intervals : 0003H for External Interrupt 0, 000BH for Timer 0, 0013H for External Interrupt 1, 001BH for Timer 1, etc. If an interrupt service routine is short enough (as is often the case in control applications), it can reside entirely within that 8-byte interval. Longer service routines can use a jump instruction to skip over subsequent interrupt locations, if other interrupts are in use.

The lowest 4 K (or 8 K in the 80C52 or 16 K in the 83C154 or 32 K in the 83C154D) bytes of Program Memory can be either in the on-chip ROM or in an external ROM. This selection is made by strapping the $\overline{EA}$ (External Access) pin to either Vcc or Vss. In the 80C51 and its derivatives, if the $\overline{EA}$ pin is strapped to Vcc, then program fetches to addresses 0000H through 0FFFH are directed to the internal ROM. Program fetches to addresses 1000H through FFFFH are directed to external ROM.

In the 80C52, $\overline{EA}$ = Vcc selects addresses 0000H through 1FFFH to be internal, and addresses 2000H through FFFFH to be external.

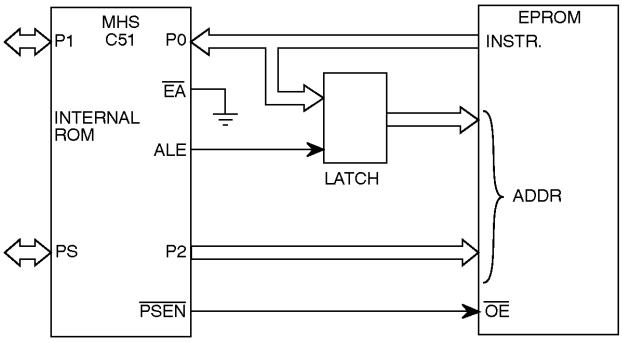
In the 83C154, $\overline{EA}$ = Vcc selects addresses 0000H through 3FFFH to be internal, and addresses 4000H to FFFFH to be external.

In the 83C154D, $\overline{EA}$ = Vcc selects addresses 0000H through 7FFFH to be internal and addresses 8000H to FFFFH to be external.

If the $\overline{EA}$ pin is strapped to Vss, then all program fetches are directed to external ROM. The ROMless parts must have this pin externally strapped to Vss to enable them to execute from external Program Memory.

The read strobe to external ROM, $\overline{PSEN}$, is used for all external program fetches. $\overline{PSEN}$ is not activated for internal program fetches.

Figure 4. Executing from External Program Memory.



The harware configuration for external program execution is shown in figure 4. Note that 16 I/O lines (Ports 0 and 2) are dedicated to bus functions during external Program Memory fetches. Port 0 (P0 in Figure 4) serves as a multiplixed address/data bus. It emits the low byte of the Program Counter (PCL) as an address, and then goes into a float state awaiting the arrival of the code byte from the Program Memory. During the time that the low byte of the Program Counter is valid on P0, the signal ALE (Address Latch Enable) clocks this byte into an address latch. Meanwhile, Port 2 (P2 in Figure 4) emits the high byte of Program Counter (PCH). Then PSEN strobes the EPROM and the code byte is read into the microcontroller.

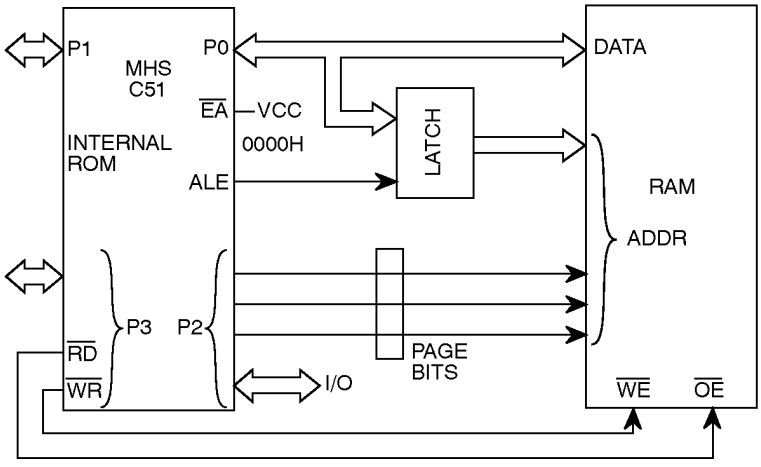
Program Memory addresses are always 16 bits wide, even though the actual amount of Program Memory used may be less than 64 K bytes. External program execution sacrifices two of the 8-bit ports, P0 and P2, to the function of addressing the Program Memory.

Data Memory

The right half of Figure 2 shows the internal and external Data Memory spaces available to the C51 user. Figure 5 shows a hardware configuration for accessing up to 2 K bytes of external RAM. The CPU in this case is executing from internal ROM. Port 0 serves as multiplexed address/data bus to the RAM, and 3 lines of Port 2 are being used to page the RAM. The CPU generates $\overline{RD}$ and $\overline{WR}$ signals as needed during external RAM accesses.

There can be up to 64 K bytes of external Data Memory. External Data Memory addresses can be either 1 or 2 bytes wide. One-byte address is often used in conjunction with one or more other I/O lines to page the RAM, as shown in Figure 5. Two-byte addresses can also be used, in which case the address byte is emitted at Port 2.

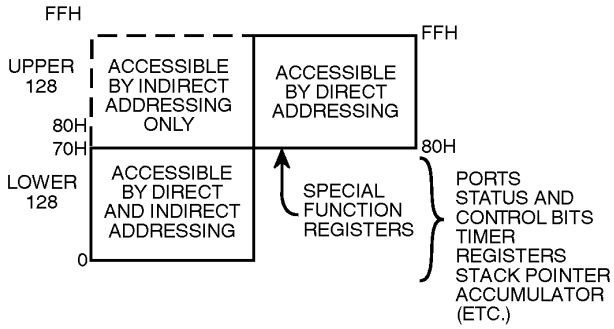
Figure 5. Accessing External Data Memory. If the Program Memory is external, the other bits of P2 are available as I/O.



Internal Data Memory is mapped in figure 6. The memory space is shown divided into three blocks, which are generally referred to as the lower 128, the Upper 128, and SFR space.

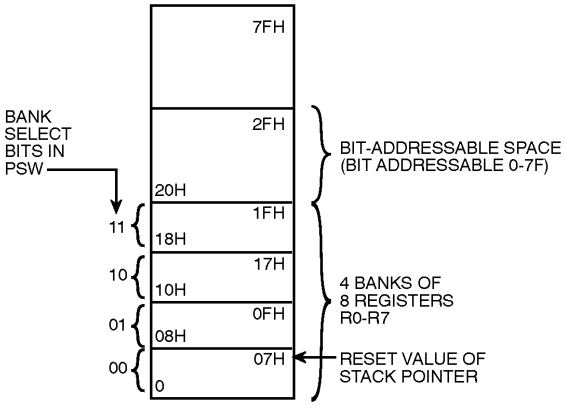
Internal Data Memory addresses are always one byte wide, which implies an address space of only 256 bytes. However, the addressing modes for internal RAM can in fact accomodate 384 bytes, using a simple trick. Direct addresses higher than 7FH access one memory space, and indirect addresses higher than 7FH access a different memory space. Thus figure 6 shows the Upper 128 and SFR space occupying the same block of addresses, 80H through FFH, although they are physically separate entities.

Figure 6. Internal Data Memory.



The Lower 128 bytes of RAM are present in all C51 devices as mapped in Figure 7. The lowest 32 bytes are grouped into 4 banks of 8 registers. Program instructions call out these registers as R0 through R7. Two bits in the Program Status Word (PSW) select which register bank is in use. This allows more efficient use of code space, since register instructions are shorter than instructions that use direct addressing.

Figure 7. The Lower 128 Bytes of Internal RAM.



The next 16 bytes above the register banks form a block of bit-addressable memory space. The C51 instruction set includes a wide selection of single-bit instructions, and the 128 bits in this area can be directly addressed by these instructions. The bit addresses in this area are 00H through 7FH.

All of the bytes in the Lower 128 can be accessed by either direct or indirect addressing. The Upper 128 (Figure 8) can only be accessed by indirect addressing. The Upper 128 bytes of RAM are not implemented in the 80C51 but are in the 80C52, 83C154 and 83C154D.

Figure 8. The Upper 128 Bytes of Internal RAM.

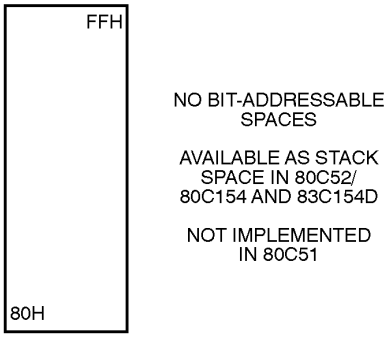
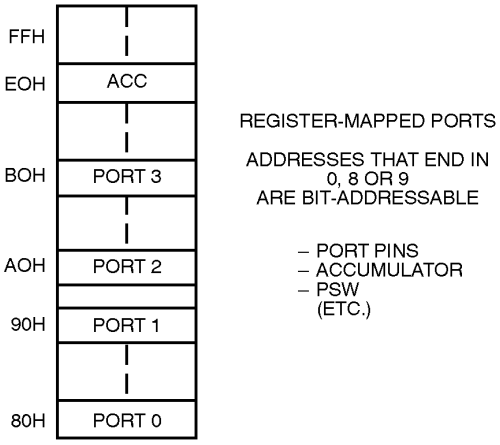


Figure 9 gives a brief look at the Special Function Register (SFR) space. SFRs include the Port latches, timers, peripheral controls, etc. These registers can only be accessed by direct addressing. In general, all C51 microcontrollers have the same SFRs as the 80C51, and at the same addresses in SFR space. However, enhancements to the 80C51 have additional SFRs that are not present in the 80C51, nor perhaps in other proliferation of the family.

Sixteen addresses in SFR space are both byte-and bit-addressable. The bit-addressable SFRs are those whose address ends in 0, 8 or 9. The bit addresses in this area are 80H through FFH.

Figure 9. SFR Space.



The C51 Instruction Set

All members of the C51 family execute the same instruction set. (except code A5H, skip opcode in C51/C52). The C51 instruction set is optimized for 8-bit control applications. It provides a variety of fast addressing modes for accessing the internal RAM to facilitate byte operations on small data structures. The instruction set provides extensive support for one-bit variables as a separate data type, allowing direct bit manipulation in control and logic systems that require Boolean processing.

An overview of the C51 instruction set is presented below, with a brief description of how certain instructions might be used.

Program Status Word

The Program Status Word (PSW) contains several status bits that reflect the current state of the CPU. The PSW, shown in Figure 10, resides in SFR space. It contains the Carry bit, the Auxiliary Carry (for BCD operations), the two register bank select bits, the Overflow flag, a parity bit, and two user-definable status flags.

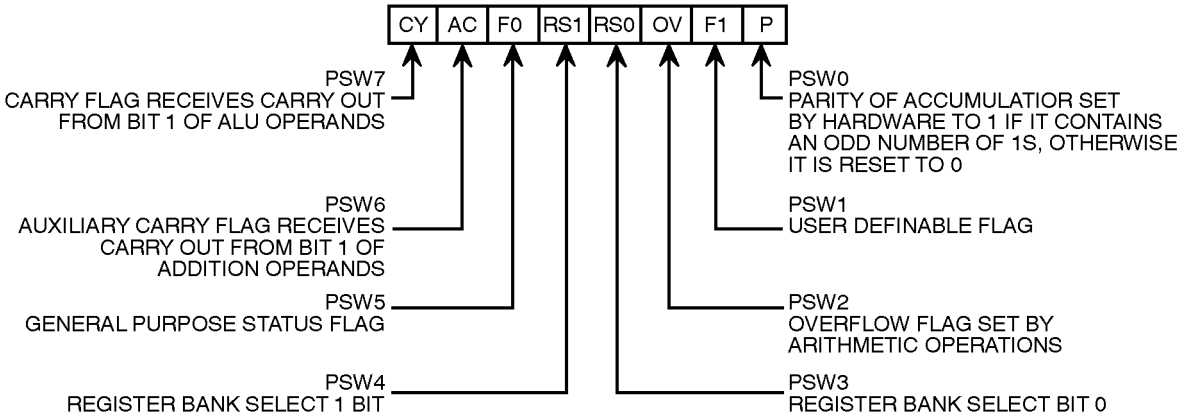
The Carry bit, other than serving the functions of a Carry bit in arithmetic operations, also serves as the “Accumulator” for a number of Boolean operations.

The bits RS0 and RS1 are used to select one of the four register banks shown in Figure 7. A number of instructions refer to these RAM locations as R0 through R7. The selection of which of the four banks is being referred to is made on the basis of the bits RS0 and RS1 at execution time.

The parity bit reflects the number of 1 s in the Accumulator : P = 1 if the Accumulator contains an odd number of 1 s, and P = 0 if the Accumulator contains an even number of 1 s. Thus the number of 1 s in the Accumulator plus P is always even.

Two bits in the PSW are uncommitted and may be used as general purpose status flags.

Figure 10. PSW (Program Status Word) Register in C51 Devices.



Addressing Modes

The addressing modes in the C51 instruction set are as follows :

Direct addressing

In direct addressing the operand is specified by an 8-bit address field in the instruction. Only 128 Lowest bytes of internal Data RAM and SFRs can be directly addressed.

Indirect addressing

In indirect addressing the instruction specifies a register which contains the address of the operand. Both internal and external RAM can be indirectly addressed.

The address register for 8-bit addresses can be R0 or R1 of the selected register bank, or the Stack Pointer. The address register for 16-bit addresses can only be the 16-bit “data pointer” register, DPTR.

Register instructions

The register banks, containing registers R0 through R7, can be accessed by certain instructions which carry a 3-bit register specification within the opcode of the instruction. Instructions that access the registers this way are code efficient, since this mode eliminates an address byte. When the instruction is executed, one of the eight registers in the selected bank is accessed. One of four banks is selected at execution time by the two bank select bits in the PSW.

Register-specific instructions

Some instructions are specific to a certain register. For example, some instructions always operate on the Accumulator, or Data Pointer, etc., so no address byte is needed to point to it. The opcode itself does that. Instructions that refer to the Accumulator as A assemble as accumulator-specific opcodes.

Immediate constants

The value of a constant can follow the opcode in Program Memory. For example,

MOV A, # 100

loads the Accumulator with the decimal number 100. The same number could be specified in hex digits as 64H.

Indexed addressing

Only Program Memory can be accessed with indexed addressing, and it can only be read. This addressing mode is intended for reading look-up tables in Program Memory. A 16-bit base register (either DPTR or the Program Counter) points to the base of the table, and the Accumulator is set up with the table entry number. The address of the table entry in Program Memory is formed by adding the Accumulator data to the base pointer.

Another type of indexed addressing is used in the “case jump” instruction. In this case the destination address of a jump instruction is computed as the sum of the base pointer and the Accumulator data.

Arithmetic Instructions

The menu of arithmetic instructions is listed in Table 2. The table indicates the addressing modes that can be used with each instruction to access the <byte> operand. For example, the ADD A, <byte> instruction can be written as :

```
ADD   A,   7FH      (direct addressing)
ADD   A,   @ R0     (indirect addressing)
ADD   A,   R7       (register addressing)
ADD   A,   # 127    (immediate constant)
```

Table 2 : A list of the MHS C51 Arithmetic Instructions.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (µs)
		Dir	Ind	Reg	Imm	
ADD A, <byt>e	A = A + <byte>	X	X	X	X	
ADDC A, <byte>	A = A + <byte> + C	X	X	X	X	1
SUBB A, <byte>	A = A – <byte> – C	X	X	X	X	1
INC A	A = A + 1	Accumulator only				1
INC <byte>	<byte> = <byte> + 1	X	X	X		1
INC DPTR	DPTR = DPTR + 1	Data Pointer only				2
DEC A	A = A – 1	Accumulator only				1
DEC <byte>	<byte> = <byte> – 1	X	X	X		1
MUL AB	B:A = B × A	ACC and B only				4
DIV AB	A = Int [A/B] B = Mod [A/B]	ACC and B only				4
DA A	Decimal Adjust	Accumulator only				1

The execution times listed in Table 2 assume a 12 MHz clock frequency. All of the arithmetic instructions execute in 1 µs except the INC DPTR instruction, which takes 2 µs, and the Multiply and Divide instructions, which take 4 µs.

Note that any byte in the internal Data Memory space can be incremented or decremented without going through the Accumulator.

One of the INC instructions operates on the 16-bit Data Pointer. The Data Pointer is used to generate 16-bit addresses for external memory, so being able to increment it in one 16-bit operation is a useful feature.

The MUL AB instruction multiplies the Accumulator by the data in the B register and puts the 16-bit product into the concatenated B and Accumulator registers.

The DIV AB instruction divides the Accumulator by the data in the B register and leaves the 8-bit quotient in the Accumulator, and the 8-bit remainder in the B register.

Oddly enough, DIV AB finds less use in arithmetic “divide” routines than in radix conversions and programmable shift operations. An example of the use of DIV AB in a radix conversion will be given later. In shift operations, dividing a number by 2^n shifts its n bits to the right. Using DIV AB to perform the division completes the shift in 4 µs leaves the B register holding the bits that were shifted out.

The DA A instruction is for BCD arithmetic operations. In BCD arithmetic ADD and ADDC instructions should always be followed by a DA A operation, to ensure that the result is also in BDC. Note that DAA will not convert a binary number to BCD. The DA A operation produces a meaningful result only as the second step in the addition of two BCD bytes.

Logical Instructions

Table 3 shows the list of MHS C51 logical instructions. The instructions that perform Boolean operations (AND, OR, Exclusive OR, NOT) on bytes perform the operation on a bit-by-bit basis. That is, if the Accumulator contains 00110101B and <byte> contains 01010011B, then

ANL A, <byte>

will leave the Accumulator holding 00010001B.
The addressing modes that can be used to access the <byte> operand are listed in Table 3. Thus, the ANL A, <byte> instruction may take any of the forms.

ANL A, 7FH (direct addressing)
ANL A, @ R1 (indirect addressing)
ANL A, R6 (register addressing)
ANL A, # 53H (immediate constant)

All of the logical instructions that are Accumulator specific in 1 µs (using a 12 MHz clock). The others take 2 µs.

Table 3 : A list of the MHS C51 Logical Instructions.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (µs)
		Dir	Ind	Reg	Imm	
ANL A, <byte>	A = A AND <byte>	X	X	X	X	1
ANL <byte>, A	<byte> = <byte> AND A	X				1
ANL <byte>, # data	<byte> = <byte> AND # data	X				2
ORL A, <byte>	A = A OR <byte>	X	X	X	X	1
ORL <byte>, A	<byte> = <byte> OR A	X				1
ORL <byte>, # data	<byte> = <byte> OR # data	X				2
XRL A, <byte>	A = A XOR <byte>	X	X	X	X	1
XRL <byte>, A	<byte> = <byte> XOR A	X				1
XRL <byte>, # data	<byte> = <byte> XOR # data	X				2
CLR A	A = 00H	Accumulator only				1
CLP A	A = NOT A	Accumulator only				1
RL A	Rotate ACC Left 1 bit	Accumulator only				1
RLC A	Rotate Left through Carry	Accumulator only				1
RR A	Rotate ACC Right 1 bit	Accumulator only				1
RRC A	Rotate Right through Carry	Accumulator only				1
SWAP A	Swap Nibbles in A	Accumulator only				1

Note that Boolean operations can be performed on any byte in the internal Data Memory space without going through the Accumulator. The XRL <byte>, # data instruction, for example, offers a quick and easy way to invert port bits, as in

XRL P1, #OFFH

If the operation is in response to an interrupt, not using the Accumulator saves the time and effort to stack it in the service routine.

The Rotate instructions (RLA, RLCA, etc.) shift the Accumulator 1 bit to the left or right. For a left rotation, the MSB rolls into the LSB position. For a right rotation, the LSB rolls into the MSB position.

The SWAP A instruction interchanges the high and low nibbles within the Accumulator. this is a useful operation in BCD manipulations. For example, if the Accumulator contains a binary number which is known to be less than 100, it can be quickly converted to BCD by the following code :

```
MOV     B, #10
DIV     AB
SWAP    A
ADD     A,B
```

Dividing the number by 10 leaves the tens digit in the low nibble of the Accumulator, and the ones digit in the B register. The SWAP and ADD instructions move the tens digit to the high nibble of the Accumulator, and the ones digit to the low nibble.

Data Transfers

Internal RAM

Table 4 shows the menu of instructions that are available for moving data around within the internal memory spaces, and the addressing modes that can be used with each one. With a 12 MHz clock, all of these instructions execute in either 1 or 2 µs.

The MOV <dest>, <src> instruction allows data to be transfered between any two internal RAM or SFR locations without going through the Accumulator. Remember the Upper 128 bytes of data RAM can be accessed only by indirect, and SFR space only by direct addressing.

Note that in all C51 devices, the stack resides in on-chip RAM, and grows upwards. The PUSH instruction first increments the Stack Pointer (SP), then copies the byte into the stack. PUSH and POP use only direct addressing to identify the byte being saved or restored, but the stack itself is accessed by indirect addressing using the SP register. This means the stack can go into the Upper 128, if they are implemented, but not into SFR space.

Table 4 : A list of the MHS C51 Data Transfer Instructions that Access Internal Data Memory Space.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (µs)
		Dir	Ind	Reg	Imm	
MOV A, <src>	A = <src>	X	X	X	X	1
MOV <dest>, A	<dest> = A	X	X	X		1
MOV <dest>, <src>	<dest> = <src>	X	X	X	X	2
MOV DPTR, # data 16	DPTR = 16-bit immediate constant				X	2
PUSH <src>	INC SP : MOV “@SP”, <scr>	X				2
POP <dest>	MOV <dest>, “@SP” : DEC SP	X				2
XCH A, <byte>	ACC and <byte> Exchange Data	X	X	X		1
XCHD A, @Ri	ACC and @ Ri exchange low nibbles		X			1

The Upper 128 are not implemented in the 80C51, nor in their ROMless. With these devices, if the SP points to the Upper 128 PUSHed bytes are lost, and POPped bytes are indeterminate.

The Data Transfer instructions include a 16-bit MOV that can be used to initialize the Data Pointer (DPTR) for look-up tables in Program Memory, or for 16-bit external Data Memory accesses.

The XCH A, <byte> instruction causes the Accumulator and addressed byte to exchange data.
The XCHD A, @ Ri instruction is similar, but only the low nibbles are involved in the exchange.

The see how XCH and XCHD can be used to facilitate data manipulations, consider first the problem of shifting an 8-digit BCD number two digits to the right. Figure 11 shows how this can be done using direct MOVs, and for comparison how it can be done using XCH instructions. To aid in understanding how the code works, the contents of the registers that are holding the BCD number and the content of the Accumulator are shown alongside each instruction to indicate their status after the instruction has been executed.

After the routine has been executed, the Accumulator contains the two digits that were shifted out on the right. Doing the routine with direct MOVs uses 14 code bytes and 9 µs of execution time (assuming a 12 MHz clock). The same operation with XCHs uses less code and executes almost twice as fast.

Figure 11. Shifting a BCD Number Two Digits to the Right.

	2A	2B	2C	2D	2E	ACC
MOV A,2EH	00	12	34	56	78	78
MOV 2EH, 2DH	00	12	34	56	56	78
MOV 2DH, 2CH	00	12	34	34	56	78
MOV 2CH, 2BH	00	12	12	34	56	78
MOV 2BH, # 0	00	00	12	34	56	78
(a) Using direct MOVs : 14 bytes, 9 µs						
	2A	2B	2C	2E	2E	ACC
CLR A	00	12	34	56	78	00
XCH A,2BH	00	00	34	56	78	12
XCH A,2CH	00	00	12	56	78	34
XCH A,2DH	00	00	12	34	78	56
XCH A,2EH	00	00	12	34	56	78
(b) Using XCHs : 9 bytes, 5 µs						

Figure 12. Shifting a BCD Number One Digit to the Right.

	2A	2B	2C	2D	2E	ACC
MOV R1,# 2EH	00	12	34	56	78	XX
MOV R0, # 2DH	00	12	34	56	78	XX
loop for R1 = 2EH :						
LOOP : MOV A, @R1	00	12	34	56	78	78
XCHD A, @R0	00	12	34	58	78	76
SWAP A	00	12	34	58	78	67
MOV @R1, A	00	12	34	58	67	67
DEC R1	00	12	34	58	67	67
DEC R0	00	12	34	58	67	67
CJNE R1, #2AH, LOOP						
loop for R1 = 2DH :	00	12	38	45	67	45
loop for R1 = 2CH :	00	18	23	45	67	23
loop for R1 = 2BH :	08	01	23	45	67	01
CLR A	08	01	23	45	67	00
XCH A,2AH	00	01	23	45	67	08

To right-shift by an odd number of digits, a one-digit shift must be executed. Figure 12 shows a sample of code that will right-shift a BCD number one digit, using the XCHD instruction. Again, the contents of the registers holding the number and of the Accumulator are shown alongside each instruction.

First, pointers R1 and R0 are set up to point to the two bytes containing the last four BCD digits. Then a loop is executed which leaves the last byte, location 2EH, holding the last two digits of the shifted number. The pointers are decremented, and the loop is repeated for location 2DH. The CJNE instruction (Compare and Jump if Not Equal) is a loop control that will be described later.

The loop is executed from LOOP to CJNE for R1 = 2EH, 2DH, 2CH and 2BH. At that point the digit that was originally shifted out on the right has propagated to location 2AH. Since that location should be left with 0s, the lost digit is moved to the Accumulator.

External RAM

Table 5 shows a list of the Data Transfer instructions that access external Data Memory. Only indirect addressing can be used. The choice is whether to use a one-byte address, @Ri, where Ri can be either R0 or R1 of the selected register bank, or a two-byte address, @DPTR. The disadvantage to using 16-bit addresses if only a few K bytes of external RAM

are involved is that 16-bit addresses use all 8 bits of Port 2 as address bus. On the other hand, 8-bit addresses allow one to address a few K bytes of RAM, as shown in Figure 5, without having to sacrifice all of Port 2.

All of these instructions execute in 2 µs, with a 12 MHz clock.

Note that in all external Data RAM accesses, the Accumulator is always either the destination or source of the data.

The read and write strobes to external RAM are activated only during the execution of a MOVX instruction. Normally these signals are inactive, and in fact if they're not going to be used at all, their pins are available as extra I/O lines. More about that later.

Table 5 : A list of the MHS C51 Data Transfer Instructions that Access External Data Memory Space.

ADDRESS WIDTH	MNEMONIC	OPERATION	EXECUTION TIME (µs)
8 bits	MOVX A, @Ri	Read external RAM @ Ri	2
8 bits	MOVX @ Ri, A	Write external RAM @ Ri	2
16 bits	MOVX A, @ DPTR	Read external RAM @ DPTR	2
16 bits	MOVX @ DPTR, A	Write external RAM @ DPTR	2

Lookup Tables

Table 6 shows the two instructions that are available for reading lookup tables in Program Memory. Since these instructions access only Program Memory, the lookup tables can be read, not updated. The mnemonic is MOVC for "move constant".

If the table access is to external Program Memory, then the read strobe is $\overline{\text{PSEN}}$.

The first MOVC instruction in Table 6 can accomodate a table of up to 256 entries, numbered 0 through 255. The number of the desired entry is loaded into the Accumulator, and the Data Pointer is set up to point to beginning of the table. Then

MOVC A, @A + DPTR

copies the desired table entry into the Accumulator.

The other MOVC instruction works the same way, except the Program Counter (PC) is used as the table base, and the table is accesses through a subroutine. First the number of the desired entry is loaded into the Accumulator, and the subroutine is called :

```
MOV     A, ENTRY_NUMBER
CALL    TABLE
```

The subroutine "TABLE" would look like this :

```
TABLE :  MOVC A, @A + PC
         RET
```

The table itself immediately follows the RET (return) instruction in Program Memory. This type of table can have up to 255 entries, numbered 1 through 255. Number 0 can not be used, because at the time the MOVC instruction is executed, the PC contains the address of the RET instruction. An entry numbered 0 would be the RET opcode itself.

Table 6 : The C51 Lookup Table Read Instructions.

MNEMONIC	OPERATION	EXECUTION TIME (µs)
MOVC A, @A + DPTR	Read Pgm Memory at (A + DPTR)	2
MOVC A, @A + PC	Read Pgm Memory at (A + PC)	2

Boolean Instructions

C51 devices contain a complete Boolean (single-bit) processor. The internal RAM contains 128 addressable bits, and the SFR space can support up to 128 other addressable bits. All of the port lines are bit-addressable, and each one can be treated as a separate single-bit port. The instructions that access these bits are not just conditional branches, but a complete menu of move, set, clear, complement, OR and AND instructions. These kinds of bit operations are not easily obtained in other architectures with any amount of byte-oriented software.

The instruction set for the Boolean processor is shown in Table 7. All bit accesses are by direct addressing. Bit addresses 00H through 7FH are in the Lower 128, and bit addresses 80H through FFH are in SFR space.

Table 7 : A list of the C51 Boolean Instructions.

MNEMONIC	OPERATION	EXECUTION TIME (µs)
ANL C,bit	C = C AND bit	2
ANL C,/bit	C = C AND (NOT bit)	2
ORL C,bit	C = C OR bit	2
ORL C,/bit	C = C OR (NOT bit)	2
MOV C,bit	C = bit	1
MOV bit,C	bit = C	2
CLR C	C = 0	1
CLR bit	bit = 0	1
SETB C	C = 1	1
SETB bit	bit = 1	1
CPL C	C = NOT C	1
CPL bit	bit = NOT bit	1
JC rel	Jump if C = 1	2
JNC rel	Jump if C = 0	2
JB bit,rel	Jump if bit = 1	2
JNB bit,rel	Jump if bit = 0	2
JBC bit,rel	Jump if bit = 1 ; CLR bit	2

Note how easily an internal flag can be moved to a port pin :

```
MOV     C, FLAG
MOV     P1.0, C
```

In this example, FLAG is the name of any addressable bit in the lower 128 or SFR space. An I/O line (the LSB of Port 1, in the case) is set or cleared depending on whether the flag bit is 1 or 0.

The Carry bit in the PSW is used as the single-bit Accumulator of the Boolean processor. Bit instructions that refer to the Carry bit as C assemble as Carry-specific instructions (CLR C, etc). The Carry bit also has a direct address, since it resides in the PSW register, which is bit-addressable.

Note that the Boolean instruction set includes ANL and ORL operations, but not the XRL (Exclusive OR) operation. An XRL operation is simple to implement in software. Suppose, for example, it is required to form the Exclusive OR of two bits :

C = bit1 XRL bit2

The software to do that could be as follows :

```
MOV     C, bit1
JNB     bit2, OVER
CPL     C
OVER : (continue)
```

First, bit 1 is moved to the Carry. If bit 2 = 0, then C now contains the correct result. That is, bit 1 XRL bit2 = bit1 if bit2 = 0. On the other hand, if bit2 = 1 C now contains the complement of the correct result. It need only be inverted (CPL C) to complete the operation.

This code uses the JNB instruction, one of a series of bit-test instructions which execute a jump if the addressed bit is set (JC, JB, JBC) or if the addressed bit is not set (JNC, JNB). In the above case, bit2 is being tested, and if bit2 = 0 the CPL C instruction is jumped over.

JBC executes the jump if the addresed bit is set, and also clears the bit. Thus a flag can be tested and cleared in one operation.

All the PSW bits are directly addressable, so the Parity bit, or the general purpose flags, for example, are also available to the bit-test instructions.

Relative offset

The destination address for these jumps is specified to the assembler by a label or by an actual address in Program Memory. However, the destination address assembles to a relative offset byte. This is a signed (two's complement) offset byte which is added to the PC in two's complement arithmetic if the jump is executed.

The range of the jump is therefore -128 to + 127 Program Memory bytes relative to the first byte following the instruction.

Jump Instructions

Table 8 shows the list of unconditional jumps.

Table 8 : Unconditional Jumps in MHS C51.

MNEMONIC	OPERATION	EXECUTION TIME (µs)
JMP addr	Jump to addr	2
JMP @A + DPTR	Jump to A + DPTR	2
CALL addr	Call subroutine at addr	2
RET	Return from subroutine	2
RETI	Return from interrupt	2
NOP	No operation	1

The table lists a single "JMP addr" instruction, but in fact there are three -SJMP, LJMP, AJMP -which differ in the format of the destination address. JMP is a generic mnemonic which can be used if the programmer does not care which way the jump is encoded.

The SJMP instruction encodes the destination address as relative offset, as described above. The instruction is 2 bytes long, consisting of the opcode and the relative offset byte. The jump distance is limited to range of -128 to + 127 bytes relative to the instruction following the SJMP.

The LJMP instruction encodes the destination address as a 16-bit constant. The instruction is 3 bytes long, consisting of the opcode and two address bytes. The destination address can be anywhere in the 64K Program Memory space.

The AJMP instruction encodes the destination address as an 11-bit constant. The instruction is 2 bytes long, consisting of the opcode, which itself contains 3 of the 11 address bits, followed by another byte containing the low 8 bits of the destination address. When the instruction is executed, these 11 bits are simply substituted for the low 11 bits in the PC. The high 5 bits stay the same. Hence the destination has to be within the same 2K block as the instruction following the AJMP.

In all cases the programmer specifies the destination address to the assembler in the same way : as a label or as a 16-bit constant. The assembler will put the destination address into the correct format for the given instruction. If the format required by the instruction will not support the distance to the specified destination address, a "Destination out of range" message is written, into the list file.

The JMP @ A + DPTR instruction supports case jumps. The destination address is computed at execution time as the sum of the 16-bit DPTR register and the Accumulator. Typically, DPTR is set up with the address of a jump table, and the Accumulator is given an index to the table. In a 5-way branch, for example, an integer 0 through 4 is loaded into the Accumulator.

The code to be executed might be as follows :

```
MOV     DPTR, # JUMP_TABLE
MOV     A, INDEX_NUMBER
RL      A
JMP     @ A + DPTR
```

The RLA instruction converts the index number (0 through 4) to an even number on the range 0 through 8, because each entry in the jump table is 2 bytes long :

```
JUMP_TABLE :
        AJMP CASE_0
        AJMP CASE_1
        AJMP CASE_2
        AJMP CASE_3
        AJMP CASE_4
```

Table 8 shows a single “CALLaddr” instruction, but there are two of them -LCALL and ACALL -which differ in the format in which the subroutine address is given to the CPU. CALL is a generic mnemonic which can be used if the programmer does not care which way the address is encoded.

The LCALL instruction uses the 16-bit address format, and the subroutine can be anywhere in the 64K Program Memory space. The ACALL instruction uses the 11-bit format, and the subroutine must be in the same 2K block as the instructon following the ACALL.

In any case the programmer specifies the subroutine address to the assembler in the same way : as a label or as a 16-bit constant. The assembler will put the address into the correct format for the given instructions.

Subroutines should end a RET instruction, which returns execution following the CALL.

RETI is used to return from an interrupt service routine. The only difference between RET and RETI is that RETI tells the interrupt control system that the interrupt in progress is done. If there is no interrupt in progress at the time RETI is executed, then the RETI is functionnally identical to RET.

Table 9 shows the list of conditional jumps available to the MHS C51 user. All of these jumps specify the destination address by the relative offset method, and so are limited to a jump distance of -128 to + 127 bytes from the instruction following the conditional jump instruction. Important to note, however, the user specifies to the assembler the actual destination address the same way as the other jumps : as a label or a 16-bit constant.

Table 9 : Conditional Jumps in MHS C51 Devices.

MNEMONIC	OPERATION	ADDRESSING MODES				EXECUTION TIME (µs)
		DIR	IND	REG	IMM	
JZ rel	Jump if A = 0	Accumulator only				2
JNZ rel	Jump if A ≠ 0	Accumulator only				2
DJNZ <byte>,rel	Decrement and jump if not zero	X		X		2
CJNZ A,<byte>,rel	Jump if A = <byte>	X			X	2
CJNE <byte>,#data,rel	Jump if <byte> = #data		X	X		2

There is no Zero bit in the PSW. The JZ and JNZ instructions test the Accumulator data for that condition.

The DJNZ instruction (Decrement and Jump if Not Zero) is for loop control. To execute a loop N times, load a counter byte with N and terminate the loop with DJNZ to the beginning of the loop, as shown below for N = 10 :

```
                MOV             COUNTER, # 10
LOOP :          (begin loop)
                *
                *
                *
                (end loop)
                DJNZ            COUNTER, LOOP
                (continue)
```

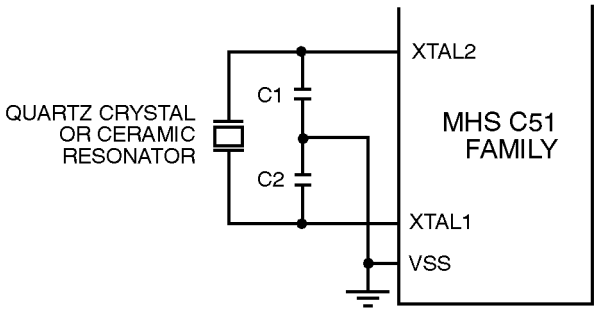
The CJNE instruction (Compare and Jump if Not Equal) can also be used for loop control as in Figure 12. Two bytes are specified in the operand field of the instruction. The jump is executed only if the two bytes are not equal. In the example of Figure 12, the two bytes were the data in R1 and the constant 2AH. The initial data in R1 was 2EH. Every time the loop was executed, R1 was decremented, and the looping was to continue until the R1 data reached 2AH.

Another application of this instruction is in “greater than, less than” comparisons. The two bytes in the operand field are taken as unsigned integers. If the first is less than the second, then the Carry bit is set (1). If the first is greater than or equal to the second, then the Carry bit is cleared.

CPU Timing

All C51 microcontrollers have an on-chip oscillator which can be used if desired as the clock source for the CPU. To use the on-chip oscillator, connect a crystal or ceramic resonator between the XTAL1 and XTAL2 pins of the microcontroller, and capacitors to ground as shown in Figure 13.

Figure 13. Using the On-Chip Oscillator.



Examples of how to drive the clock with an external oscillator are shown in Figure 14. In the MHS C51 devices the signal at the XTAL1 pin drives the internal clock generator. If only one pin is going to be driven with the external oscillator signal, make sure it is the right pin.

The internal clock generator defines the sequence of states that make up the MHS C51 machine cycle.

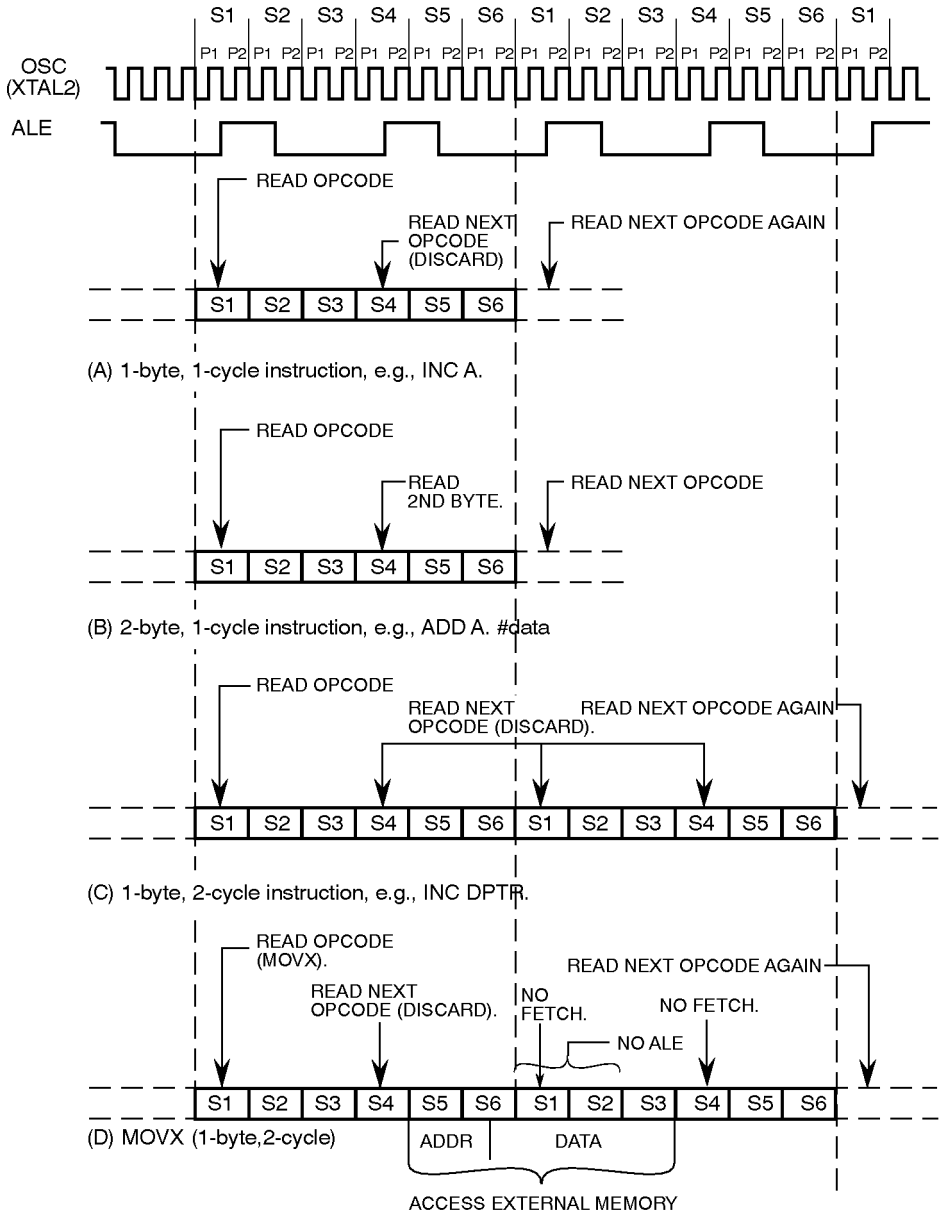
Figure 14. Using an External Clock.

Machine Cycles

A machine cycle consists of a sequence of 6 states, numbered S1 through S6. Each state time lasts for two oscillator periods. Thus a machine cycle takes 12 oscillator periods or 1 µs if the oscillator frequency is 12 MHz.

Each state is divided into a Phase 1 half and a Phase 2 half. Figure 15 shows the fetch/execute sequences in states and phases for various kinds of instructions. Normally two program fetches are generated during each machine cycle, even if the instruction being executed doesn't require it. If the instruction being executed doesn't need more code bytes, the CPU simply ignores the extra fetch, and the Program Counter is not incremented.

Figure 15. State Sequences in MHS C51.



Execution of a one-cycle instruction (Figure 15A and B) begins during State 1 of the machine cycle, when the opcode is latched into the Instruction Register. A second fetch occurs during S4 of the same machine cycle. Execution is completed at the end of State 6 of this machine cycle.

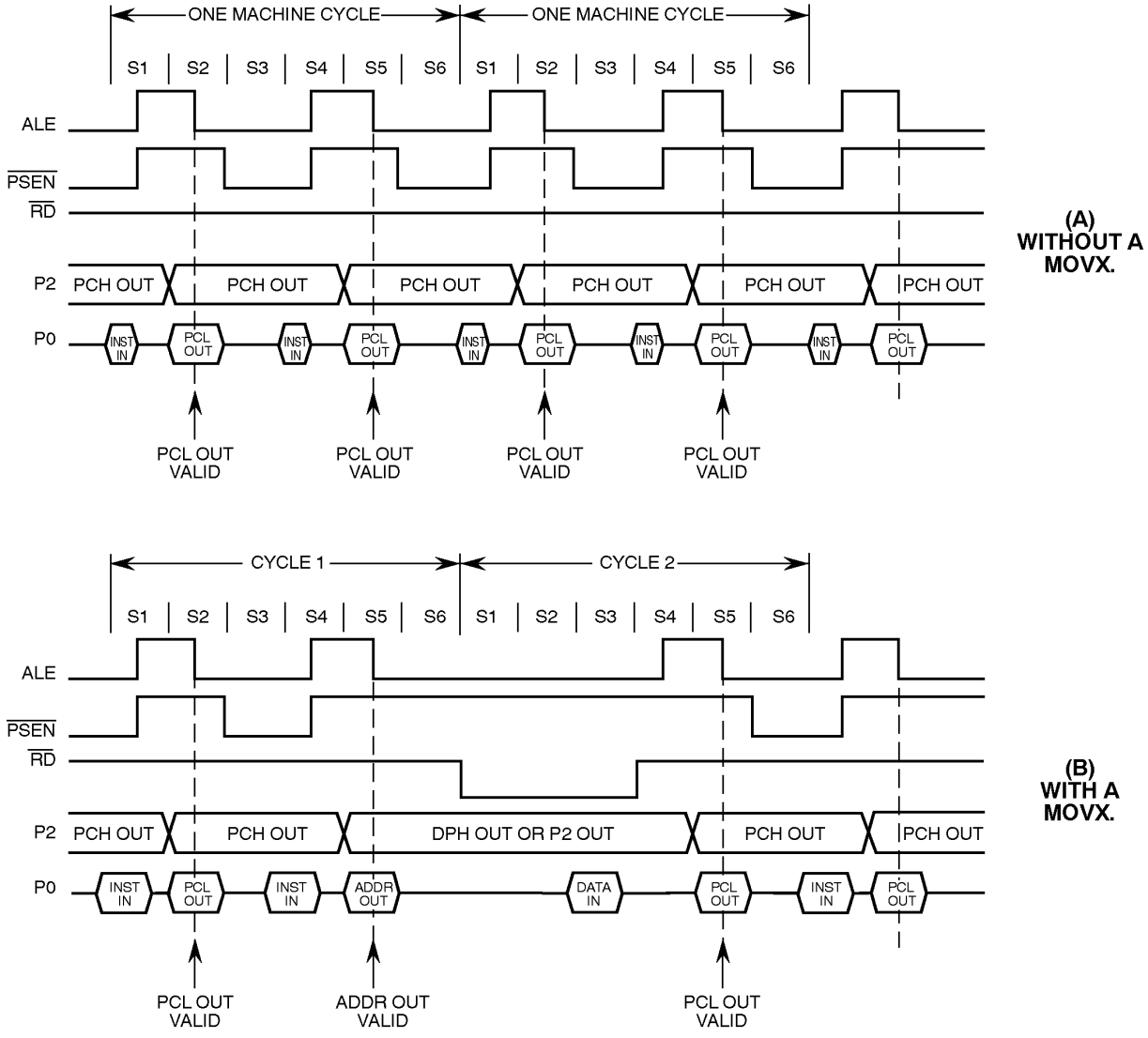
The MOVX instructions take two machine cycles to execute. No program fetch is generated during the second cycle of a MOVX instruction. This is the only time program fetches are skipped. The fetch/execute sequence for MOVX instructions is shown in Figure 15 (D).

The fetch/execute sequences are the same whether the Program Memory is internal or external to the chip. Execution times do not depend on whether the Program Memory is internal or external.

Figure 16 shows the signals and timing involved in program fetches when the Program Memory is external. If Program Memory is external, then, the Program Memory read strobe $\overline{PSEN}$ is normally activated twice per machine cycle, as shown in Figure 16 (A).

If an access to external Data Memory occurs, as shown in Figure 16 (B), two $\overline{PSEN}$s are skipped, because the address and data bus are being used for the Data Memory access.

Figure 16. Bus Cycles in MHS C51 Devices Executing from External Program Memory.



Note that a Data Memory bus cycle takes twice as much time as a Program Memory bus cycle. Figure 16 shows the relative timing of the addresses being emitted at ports 0 and 2, and of ALE and $\overline{PSEN}$. ALE is used to latch the low address byte from P0 into the address latch.

When the CPU is executing from internal Program Memory, PSEN is not activated, and program addresses are not emitted. However, ALE continues to be activated twice per machine cycle and so is available as a clock output signal. Note, however, that one ALE is skipped during the execution of the MOVX instruction.

Interrupt Structure

The 80C51 and his ROMless version provide 5 interrupt sources : 2 external interrupts, 2 timer interrupts, and the serial port interrupt. the 80C52, 83C154 and 83C154D and their ROMless version provide these 5 plus a sixth interrupt that is associated with the third timer/counter which is present in those devices.

What follows is an overview of the interrupt structure for these devices. More detailed information for specific members of the MHS C51 family is provided in the chapters of this handbook that describe the specific devices.

Interrupt Enables

Each of the interrupt source can be individually enabled or disabled by setting or clearing a bit in the SFR named IE (Interrupt Enable). This register also contains a global disable bit, which can be cleared to disable all interrupts at once. Figure 17 shows the IE register for the 80C51, 80C52 and 83C154 or the 83C154D.

Figure 17. IE (Interrupt Enable) Register in the 80C51, 80C52, 83C154 and 83C154D.

(MSB) (LSB)

EA	X	ET2	ES	ET1	EX1	ET0	EX0

Symbol	Position	Function
EA	IE.7	disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
–	IE.6	reserved
ET2	IE.5	enables or disables the Timer 2 overflow or capture interrupt. If ES = 0, the Timer 2 interrupt is disabled.
ES	IE.4	enables or disables the Serial Port interrupt. If ES = 0, the Serial Port interrupt is disabled.
ET1	IE.3	enables or disables the Timer 1 Overflow interrupt. If ET1 = 0, the Timer 1 interrupt is disabled.
EX1	IE.2	enables or disables External Interrupt 1. If EX1 = 0, External Interrupt 1 is disabled.
ET0	IE.1	enables or disables the Timer 0 Overflow interrupt. If ET0 = 0, the Timer 0 interrupt is disabled.
EX0	IE.0	enables or disables External Interrupt 0. If EX0 = 0, External Interrupt 0 is disabled.

Interrupt priorities

Each interrupt source can also be individually programmed to one of two priority level by setting or clearing a bit in the SFR named IP (Interrupt Priority). Figure 18 shows the IP register in the 80C51, 80C52, *83C154 and 83C154D.

A low-priority interrupt can be interrupted by a high-priority interrupt, but not by another low-priority interrupt.

A high-priority interrupt can’t be interrupted by any other interrupt source.

If two interrupt requests of different priority levels are received simultaneously, the request of higher priority level is serviced. If interrupt requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus within each priority level there is a second priority structure determined by the polling sequence.

Figure 18. IP (Interrupt Priority) Register in the 80C51, 80C52, 83C154 and 83C154D.

(MSB) (LSB)

PCT	X	PT2	PS	PT1	PX1	PT0	PX0

Symbol	Position	Function
PCT	IP.7	83C154/C154D only. Priority interrupt circuit control bit. The priority register contents are valid and priority assigned interrupts can be processed when this bit is “0”. When the bit is “1”, the priority interrupt circuit is stopped, and interrupts can only be controlled by the interrupt enable register (IE).
–	IP.6	reserved
PT2	IP.5	defines the Timer 2 interrupt priority level. PT2 = 1 programs it to the higher priority level.
PS	IP.4	defines the Serial Port interrupt priority level. PS = 1 programs it to the higher priority level.
PT1	IP.3	defines the Timer 1 interrupt priority level. PT1 = 1 programs it to the higher priority level.
PX1	IP.2	defines the External Interrupt 1 priority level. PX1 = 1 programs it to the higher priority level.
PT0	IP.1	defines the Timer 0 interrupt priority level. PT0 = 1 programs it to the higher priority level.
PX0	IP.0	defines the External Interrupt 0 priority level. PX0 = 1 programs it to the higher priority level.

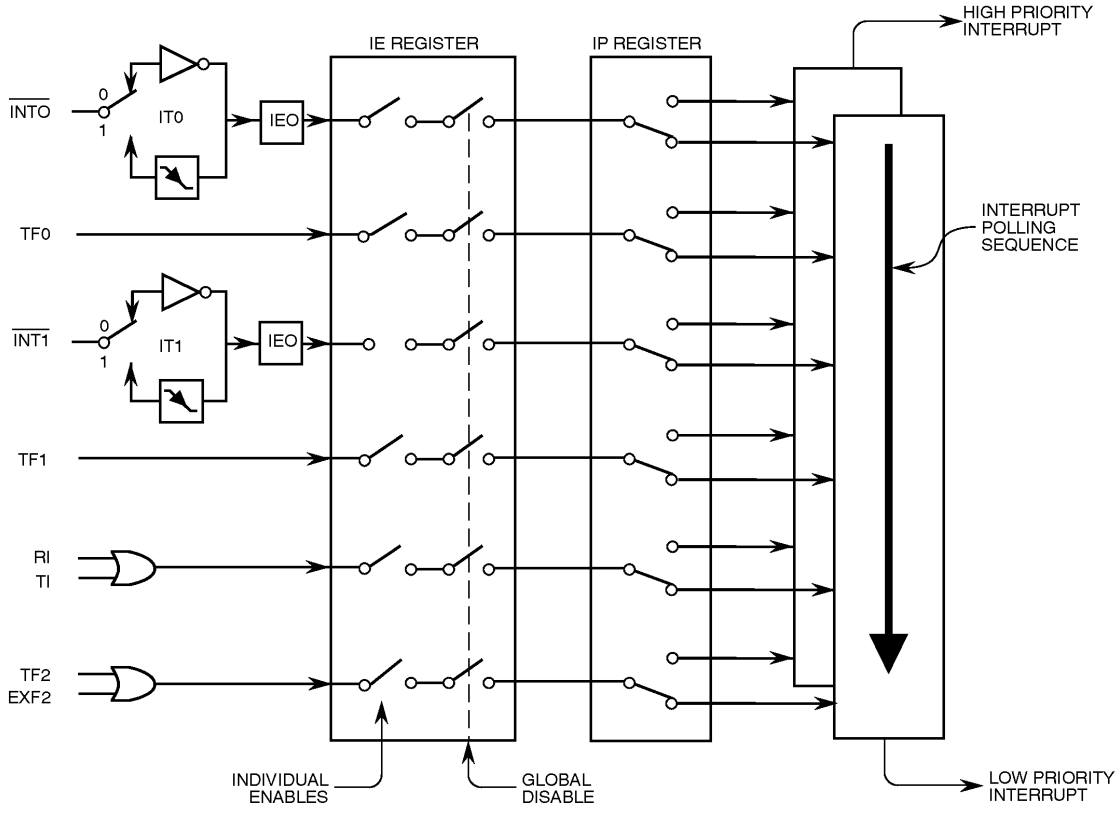
Figure 19 shows, for the 80C51, 80C52, 83C154 and 83C154D, how the IE and IP registers and the polling sequence work to determine which interrupt will be serviced.

In operation, all the interrupt flags are latched into the interrupts control system during State 5 of every machine cycle. The samples are polled during the following machine cycle. If the flag for an enabled interrupt is found to be set (1), the interrupt system generates an LCALL to the appropriate location in Program Memory, unless some other condition blocks the interrupt. Several conditions can block an interrupt, among them that an interrupt of equal or higher priority level is already in progress.

The hardware-generated LCALL causes the contents of the Program Counter to be pushed onto the stack, and reloads the PC with the beginning address of the service routine. As previously noted (Figure 3), the service routine for each interrupt begins at a fixed location.

Only the Program Counter is automatically pushed onto that stack, not the PSW or any other register. Having only the PC be automatically saved allows the programmer to decide how much time to spend saving which other registers. This enhances the interrupt response time, albeit at the expense of increasing the programmer’s burden of responsability. As a result, many interrupt functions that are typical in control applications-toggling a port pin, for example, or reloading a timer, or unloading a serial buffer can often be completed in less time than it takes other architectures to commence them.

Figure 19. 80C51, 80C52, 83C154 and 83C154D Interrupt Control System.



Simulating a third priority level in software

Some applications require more than the two priority levels that are provided by on-chip hardware in C51 devices. In these cases, relatively simple software can be written to produce the same effect as a third priority level. First, interrupts that are to have higher priority than 1 are assigned to priority 1 in the IP (Interrupt Priority) register. The service routines for priority 1 interrupts that are supposed to be interruptible by “priority 2” interrupts are written to include the following code :

```
        PUSH    IE
        MOV     IE, # MASK
        CALL    LABEL
              **********
        (execute service routine)
              ********
        POP     IE
        RET
LABEL : RETI
```

As soon as any priority 1 interrupt is acknowledged, the IE (Interrupt Enable) register is redefined so as to disable all but “priority 2” interrupts. Then, a CALL to LABEL executes the RETI instruction, which clears the priority 1 interrupt-in-progress flip-flop. At this point any priority 1 interrupt that is enabled can be serviced, but only “priority 2” interrupts are enabled.

POPping IE restores the original enable byte. Then a normal RET (rather than another RETI) is used to terminate the service routine. The additional software adds 10 µs (at 12 MHz) to priority 1 interrupts.