

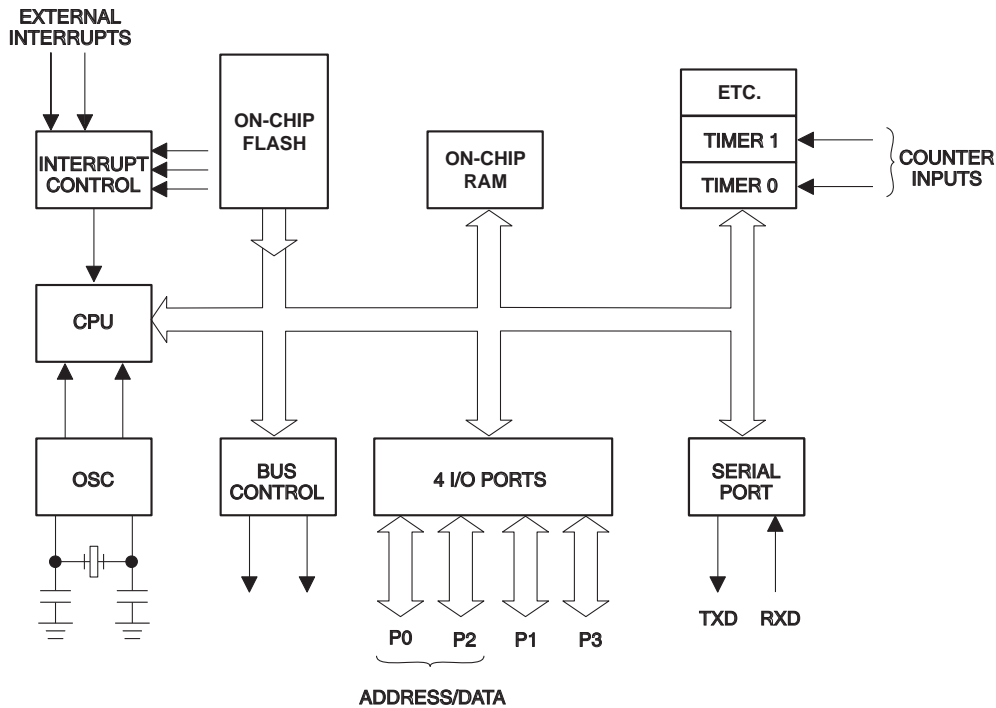
Features

- 8-Bit CPU Optimized for Control Applications
- Extensive Boolean Processing Capabilities (Single-Bit Logic)
- On-Chip Flash Program Memory
- On-Chip Data RAM
- Bidirectional and Individually Addressable I/O Lines
- Multiple 16-Bit Timer/Counters
- Full Duplex UART
- Multiple Source/Vector/Priority Interrupt Structure
- On-Chip Clock Oscillator
- On-chip EEPROM (AT89S series)
- SPI Serial Bus Interface (AT89S Series)
- Watchdog Timer (AT89S Series)

The basic architectural structure of the AT89C51 core is shown in Figure 1.

Block Diagram

Figure 1. Block Diagram of the AT89C core



For more information on the individual devices and features, refer to the Hardware Descriptions and Data Sheets of the specific device.



Flash Microcontroller

Architectural Overview



Figure 2. Block Diagram of the AT89S core

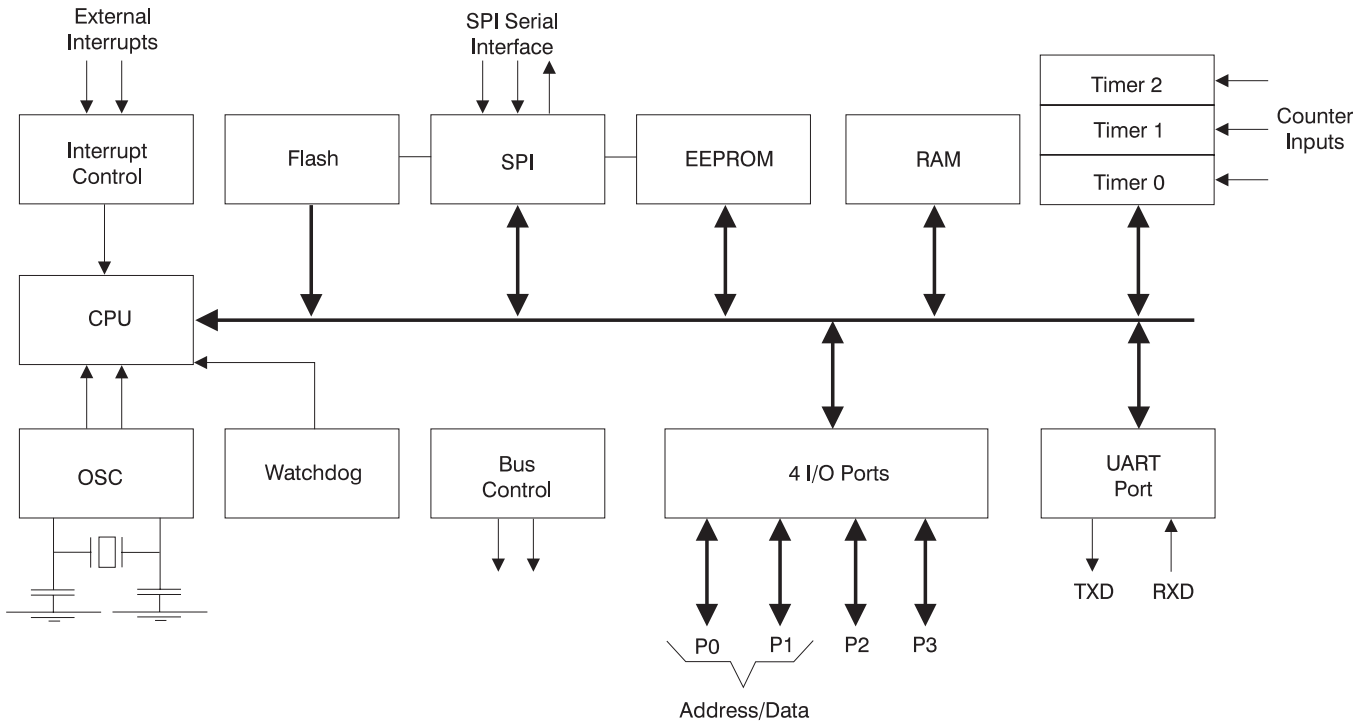
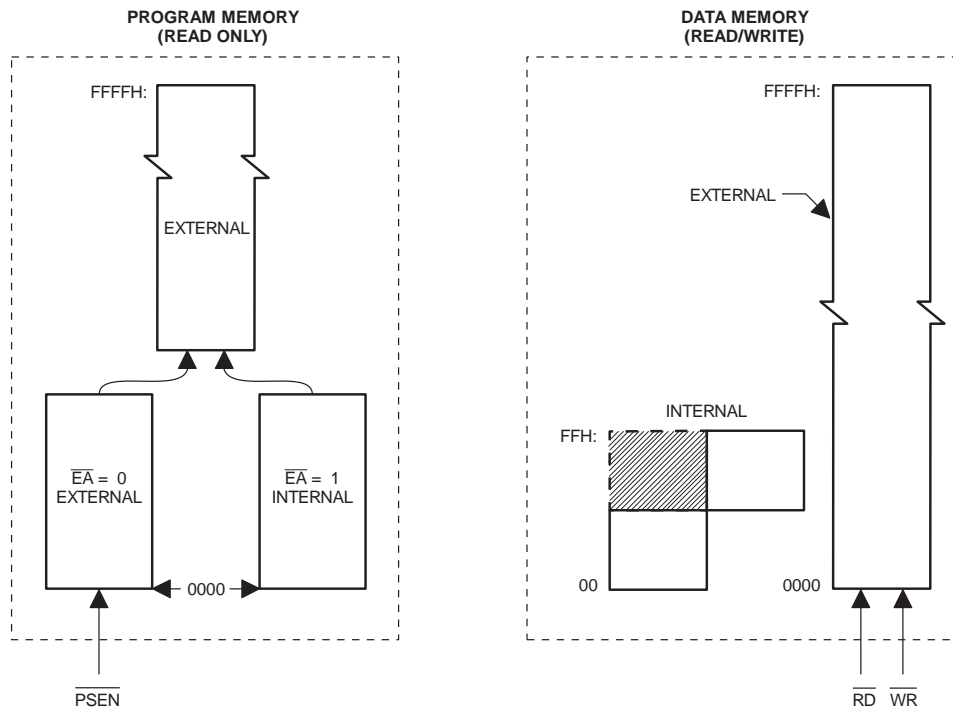


Figure 3. AT89C51/LV51 and AT89C52/LV52 Memory Structure



Reduced Power Modes

To exploit the power savings available in CMOS circuitry, Atmel's Flash microcontrollers have two software-invoked reduced power modes.

- **Idle Mode.** The CPU is turned off while the RAM and other on-chip peripherals continue operating. In this mode, current draw is reduced to about 15 percent of the current drawn when the device is fully active.
- **Power Down Mode.** All on-chip activities are suspended, while the on-chip RAM continues to hold its data. In this mode, the device typically draws less than 15 μA , and can be as low as 0.6 μA .

In addition, these devices are designed using static logic, which does not require continuous clocking. That is, the clock frequency can be slowed or even stopped while waiting for an internal event.

Memory Organization

Logical Separation of Program Data Memory

All Atmel Flash microcontrollers have separate address spaces for program and data memory, as shown in Figure 3. 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. There can be up to 64K bytes of directly addressable program memory. The read strobe for external program memory is the Program Store Enable signal ($\overline{\text{PSEN}}$).

Data memory occupies a separate address space from program memory. Up to 64K bytes of external memory can be directly addressed in the external data memory space. The CPU generates read and write signals, $\overline{\text{RD}}$ and $\overline{\text{WR}}$, during external data memory accesses.

External program memory and external data memory can be combined by applying the $\overline{\text{RD}}$ and $\overline{\text{PSEN}}$ signals to the input of an AND gate and using the output of the gate as the read strobe to the external program/data memory.

Program Memory

Figure 4 shows a map of the lower part of the program memory. After reset, the CPU begins execution from location 0000H.

As shown in Figure 4, each interrupt is assigned a fixed location in program memory. The interrupt causes the CPU to jump to that location, where it executes the service routine. External Interrupt 0, for example, is assigned to location 0003H. If External Interrupt 0 is used, its service routine must begin at location 0003H. If the interrupt is not used, its service location is available as general purpose 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, and so on. 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 addresses of program memory can be either in the on-chip Flash or in an external memory. To make this selection, strap the External Access ($\overline{\text{EA}}$) pin to either V_{CC} or GND.

For example, in the AT89C51 with 4K bytes of on-chip Flash, if the $\overline{\text{EA}}$ pin is strapped to V_{CC} , program fetches to addresses 0000H through 0FFFH are directed to the internal Flash. Program fetches to addresses 1000H through FFFFH are directed to external memory.

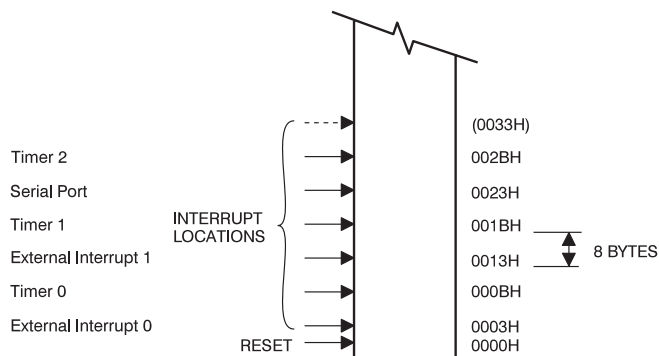
In the AT89C52 (8K bytes Flash), $\overline{\text{EA}} = V_{\text{CC}}$ selects addresses 0000H through 1FFFH to be internal and addresses 2000H through FFFFH to be external.

If the $\overline{\text{EA}}$ pin is strapped to GND, all program fetches are directed to external memory.

The read strobe to external memory, $\overline{\text{PSEN}}$, is used for all external program fetches. Internal program fetches do not activate $\overline{\text{PSEN}}$.

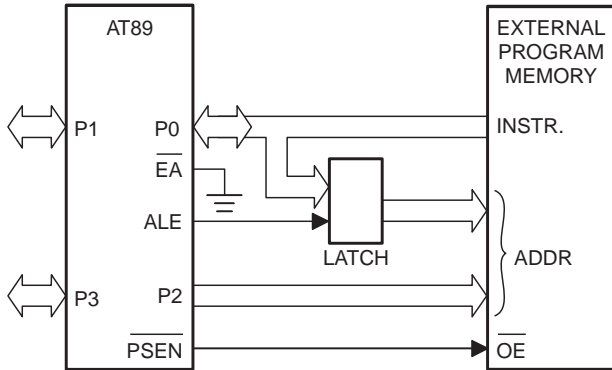
The hardware configuration for external program execution is shown in Figure 5. Note that 16 I/O lines (Ports 0 and 2) are dedicated to bus functions during external program memory fetches. Port 0 (P0 in Figure 5) serves as a multiplexed address/data bus. It emits the low byte of the Program Counter (PCL) as an address and then goes into a float state while waiting for 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 5) emits the high byte of the Program Counter (PCH). Then $\overline{\text{PSEN}}$ strobes the external memory, and the microcontroller reads the code byte.

Figure 4. Program Memory



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

Figure 5. Executing from External Program Memory



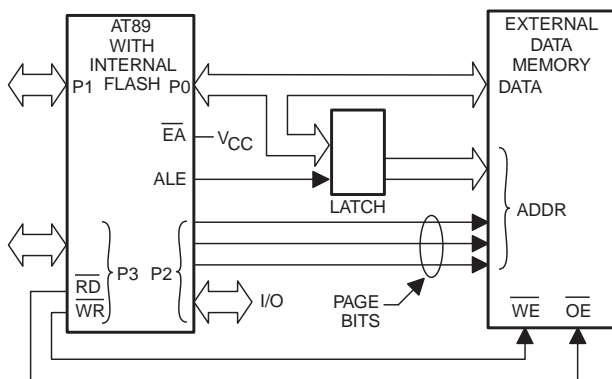
Data Memory

The right half of Figure 3 shows the internal and external data memory spaces available on Atmel's Flash microcontrollers.

Figure 6 shows a hardware configuration for accessing up to 2K bytes of external RAM. In this case, the CPU executes from internal Flash. Port 0 serves as a multiplexed address/data bus to the RAM, and 3 lines of Port 2 are used to page the RAM. The CPU generates \overline{RD} and \overline{WR} signals as needed during external RAM accesses.

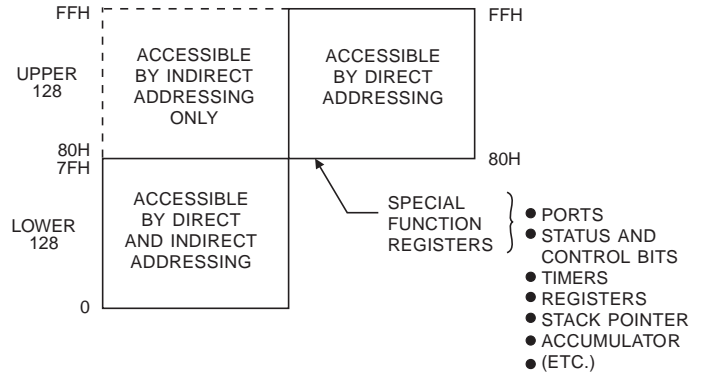
You can assign up to 64K bytes of external data memory. External data memory addresses can be either 1 or 2 bytes wide. One-byte addresses are often used in conjunction with one or more other I/O lines to page the RAM, as shown in Figure 6. Two-byte addresses can also be used, in which case the high address byte is emitted at Port 2.

Figure 6. Accessing external data memory. If the program memory is internal, the other bits of P2 are available as I/O.



Internal data memory is shown in Figure 7. The memory space is divided into three blocks, which are generally referred to as the Lower 128, the Upper 128, and SFR space.

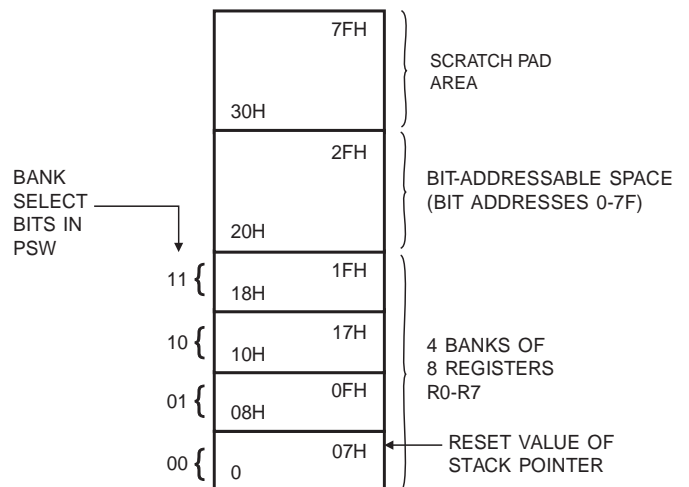
Figure 7. Internal Data Memory



Internal data memory addresses are always 1 byte wide, which implies an address space of only 256 bytes. However, the addressing modes for internal RAM can in fact accommodate 384 bytes. Direct addresses higher than 7FH access one memory space, and indirect addresses higher than 7FH access a different memory space. Thus, Figure 7 shows the Upper 128 and SFR space occupying the same block of addresses, 80H through FFH, although they are physically separate entities.

Figure 8 shows how the lower 128 bytes of RAM are mapped. 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 architecture allows more efficient use of code space, since register instructions are shorter than instructions that use direct addressing.

Figure 8. The Lower 128 Bytes of Internal RAM



The next 16 bytes above the register banks form a block of bit-addressable memory space. The microcontroller instruction set includes a wide selection of single-bit instructions, and these instructions can directly address the 128 bits in this area. These bit addresses 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 9) can only be accessed by indirect addressing. The Upper 128 bytes of RAM are only in the devices with 256 bytes of RAM.

Figure 9. The Upper 128 Bytes of Internal RAM

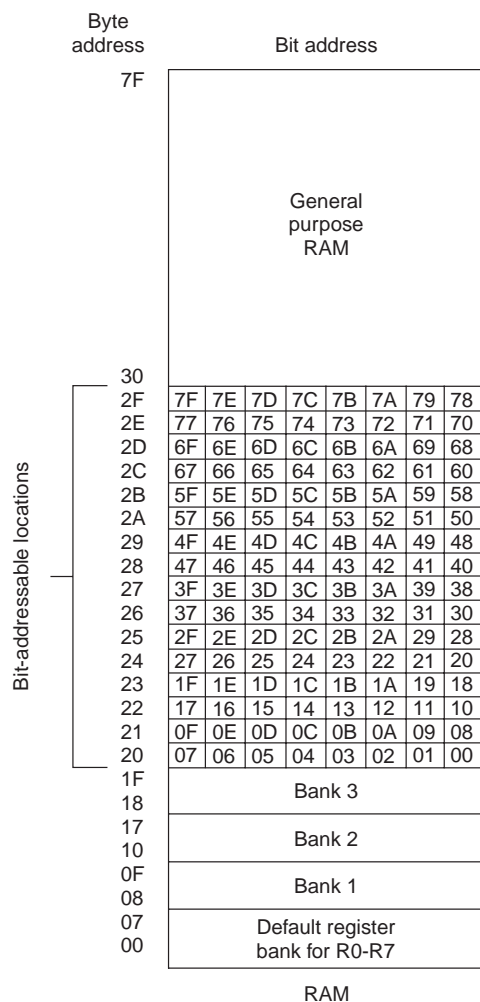


Figure 10 gives a brief look at the Special Function Register (SFR) space. SFRs include Port latches, timers, peripheral controls, etc. These registers can only be accessed by direct addressing. In general, all Atmel microcontrollers have the same SFRs at the same addresses in SFR space as the AT89C51 and other compatible microcontrollers. However, upgrades to the AT89C51 have additional SFRs. Sixteen addresses in SFR space are both byte- and bit-addressable. The bit-addressable SFRs are those whose address ends in 000B. The bit addresses in this area are 80H through FFH.

Figure 10. SFR Space

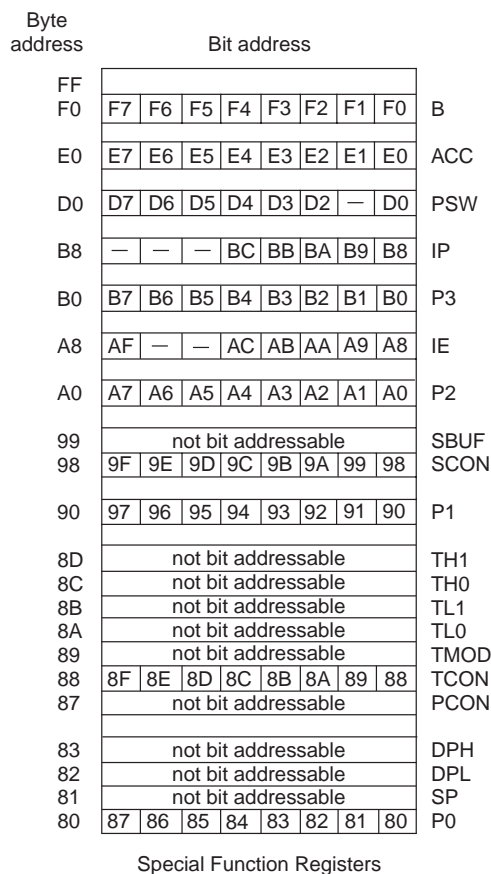
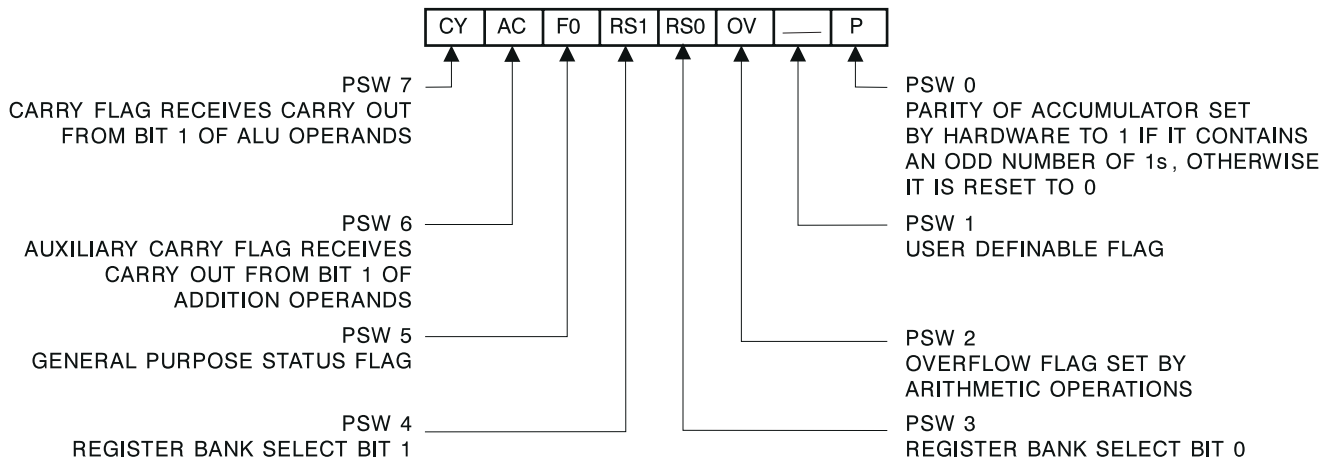


Figure 11. PSW (Program Status Word) Register in Atmel Flash Microcontrollers



The Instruction Set

All members of the Atmel microcontroller family execute the same instruction set. This instruction set is optimized for 8-bit control applications and 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 1-bit variables as a separate data type, allowing direct bit manipulation in control and logic systems that require Boolean processing.

The following overview of the instruction set gives a brief description of how certain instructions can be used.

Program Status Word

The Program Status Word (PSW) contains status bits that reflect the current state of the CPU. The PSW, shown in Figure 11, resides in SFR space. The PSW 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, in addition to serving as a Carry bit in arithmetic operations, also serves as the “Accumulator” for a number of Boolean operations.

The bits RS0 and RS1 select one of the four register banks shown in Figure 8. A number of instructions refer to these RAM locations as R0 through R7. The status of the RS0 and RS1 bits at execution time determines which of the four banks is selected.

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

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

Addressing Modes

The addressing modes in the Flash microcontroller instruction set are as follows.

Direct Addressing

In direct addressing, the operand is specified by an 8-bit address field in the instruction. Only internal data RAM and SFRs can be directly addressed.

Indirect Addressing

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

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

Register Instructions

The register banks, which contain registers R0 through R7, can be accessed by instructions whose opcodes carry a 3-bit register specification. Instructions that access the registers this way make efficient use of code, 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, so no address byte is needed to point to it. In these cases, the opcode itself points to the correct register. 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

Program memory can only be accessed via indexed addressing. 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 1. 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 follows.

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

The execution times listed in Table 1 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 using the Accumulator.

The INC DPTR instruction operates on the 16-bit Data Pointer. The Data Pointer generates 16-bit addresses for external memory, so the ability to be incremented 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.

Note: DIV AB is less useful in arithmetic “divide” routines than in radix conversions and programmable shift operations. 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 and 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 BCD. Note that DA A 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.

Table 1. A List of Atmel Microcontroller Arithmetic Instructions

Mnemonic	Operation	Addressing Modes				Execution Time (μS)
		Dir	Ind	Reg	Imm	
ADD A, <byte>	$A = A + \text{<byte>}$	X	X	X	X	1
ADDC A, <byte>	$A = A + \text{<byte>} + C$	X	X	X	X	1
SUBB A, <byte>	$A = A - \text{<byte>} - C$	X	X	X	X	1
INC A	$A = A + 1$	Accumulator Only				1
INC <byte>	$\text{<byte>} = \text{<byte>} + 1$	X	X	X		1
INC DPTR	$\text{DPTR} = \text{DPTR} + 1$	Data Pointer Only				2
DEC A	$A = A - 1$	Accumulator Only				1
DEC <byte>	$\text{<byte>} = \text{<byte>} - 1$	X	X	X		1
MUL AB	$B:A = B \times A$	ACC and B Only				4
DIV AB	$A = \text{Int}[A/B]$ $B = \text{Mod}[A/B]$	ACC and B Only				4
DA A	Decimal Adjust	Accumulator Only				1

Table 2. 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
CRL A	A = 00H	Accumulator Only				1
CPL 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

Logical Instructions

Table 2 shows the Atmel Flash microcontroller logical instructions. The instructions that perform Boolean operations (AND, OR, Exclusive OR, NOT) on bytes operate on a bit-by-bit basis. That is, if the Accumulator contains 00110101B and <byte> contains 01010011B, then

```
ANL A, <byte>
```

leaves the Accumulator holding 00010001B.

Table 2 also lists the addressing modes that can be used to access the <byte> operand. Thus, the ANL A, <byte> instruction may take any of the following 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 execute in 1 μs (using a 12 MHz clock). The others take 2 μs.

Note that Boolean operations can be performed on any byte in the lower 128 internal data memory space or the SFR space using direct addressing, without using the Accumulator. The XRL <byte>, #data instruction, for example, offers a quick and easy way to invert port bits, as in the following example.

```
XRL P1,#0FFH
```

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

The Rotate instructions (RL A, RLC A, 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 Least Significant Bit (LSB) rolls into the Most Significant Bit (MSB) position.

The SWAP A instruction interchanges the high and low nibbles within the Accumulator. This exchange is useful in BCD manipulations. For example, if the Accumulator contains a binary number that is known to be less than 100, the following code can quickly convert it to BCD.

```
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.

Table 3. 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, #data16	DPTR = 16-bit immediate constant				X	2
PUSH <src>	INC SP : MOV “@SP”, <src>	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

Data Transfers

Internal Ram

Table 3 shows the menu of instructions and associated addressing modes that are available for moving data within the internal memory spaces. 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 transferred between any two internal RAM or SFR locations without going through the Accumulator.

Note that in all Atmel Flash microcontroller 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.

In devices that do not implement the Upper 128, 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 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 exchanges the data in the Accumulator and the addressed byte. The XCHD A, @Ri instruction is similar, but only the low nibbles are exchanged.

To see how XCH and XCHD can facilitate data manipulations, consider the problem of shifting an 8-digit BCD number two digits to the right. Figure 12 compares how direct MOVs and XCH instructions can do this operation. The contents of the registers that hold the BCD number and the content of the Accumulator are shown along side each instruction to indicate their status after the instruction executes.

After the routine executes, the Accumulator contains the two digits that were shifted to the right. Using direct MOVs requires 14 code bytes and 9 μs of execution time (under a 12 MHz clock). Using XCHs for the same operation requires less code and executes almost twice as fast.

To right-shift by an odd number of digits, a one-digit shift must be executed. Figure 13 shows a sample of code that right-shifts a BCD number one digit, using the XCHD instruction.

Figure 12. 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	2D	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						

In this example, pointers R1 and R0 point to the two bytes containing the last four BCD digits. Then a loop leaves the

Figure 13. 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

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.

Note: 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.

Table 4. Data Transfer Instructions that Access External Data Memory

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

Table 5. 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

External Ram

Table 4 lists the Data Transfer instructions that access external data memory. Only indirect addressing can be used. Either a one-byte address, @Ri, where Ri can be either R0 or R1 of the selected register bank, or a two-byte address, @DPTR, can be used. The disadvantage of using 16-bit addresses when only a few Kbytes 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 a few Kbytes of RAM to be used without sacrificing all of Port 2, as shown in Figure 6.

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 if they are not going to be used at all, their pins are available as extra I/O lines.

Lookup Tables

Table 5 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 only be read, not updated. The mnemonic for “move constant” is MOVC.

If the table access is to external program memory, then the read strobe is PSEN.

The first MOVC instruction in Table 5 can accommodate 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 the following instruction copies the desired table entry into the Accumulator.

```
MOVC A, @A+ DPTR
```

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

```
MOV A,ENTRY__NUMBER
CALL TABLE
```

The subroutine TABLE would look like the following example.

```
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.

Boolean Instructions

Atmel's Flash microcontrollers 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.

Table 6. 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

The instruction set for the Boolean processor is shown in Table 6. 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.

The following example shows 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 this 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.

The Boolean instruction set includes ANL and ORL, but not the XRL (Exclusive OR) operation. Implementing XRL in software is simple. Suppose, for example, that an application requires the Exclusive OR of two bits.

```
C = bit1 .XRL. bit2
```

The software to do this operation could be as follows.

```
MOV C,bit1
JNB bit2,OVER
CPL C
```

OVER (continue)

First, bit1 is moved to the Carry. If bit2 = 0, then C now contains the correct result. That is, bit1 .XRL. bit2 = bit1 if bit2 = 0. On the other hand, if bit2 = 1, C now contains the complement of the correct result. C CARRY 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.

If the addressed bit is set, JBC executes the jump 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 that 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 7 shows the list of unconditional jumps.

Table 7. Unconditional Jumps in Flash Microcontrollers

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

Table 8. Conditional Jumps in Flash Microcontrollers

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
CJNE A,<byte>,rel	Jump if A ≠ <byte>	X			X	2
CJNE <byte>,#data,rel	Jump if <byte> ≠ #data		X	X		2

Table 7 lists a single JMP addr instruction, but in fact there are three—SJMP, LJMP and AJMP—which differ in the format of the destination address. JMP is a generic mnemonic that can be used if the programmer does not care which way the jump is encoded.

The SJMP instruction encodes the destination address as a 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 a 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 the same way: as a label or as a 16-bit constant. The assembler puts the destination address into the correct format for the given instruction. If

the format required by the instruction does 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 RL A instruction converts the index number (0 through 4) to an even number in the range 0 through 8, because each entry in the jump table is 2 bytes long, as shown in the following example.

```
JUMP__TABLE:
AJMP CASE__0
AJMP CASE__1
AJMP CASE__2
AJMP CASE__3
AJMP CASE__4
```

Table 8 shows a single CALL addr instruction, but there are two CALL instructions—LCALL and ACALL—which differ in the format in which the subroutine address is given to the CPU. CALL is a generic mnemonic that 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 instruction following the ACALL.

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

Subroutines should end with a RET instruction, which returns execution to the instruction 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 finished. If no interrupt is in progress at the time RETI is executed, then the RETI is functionally identical to RET.

Table 8 shows the list of conditional jumps available. 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. 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.

There is no 0 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 a 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 shown in Figure 13. 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 13, 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 decre-

mented, and the looping continued 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 Atmel Flash microcontrollers have an on-chip oscillator, which can be used 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 connect the capacitors to ground as shown in Figure 14.

Examples of how to drive the clock with an external oscillator are shown in Figure 15b.

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

Figure 14. Using the On-Chip Oscillator

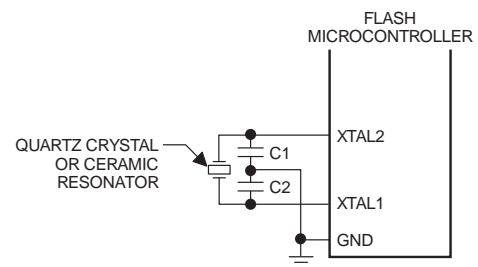


Figure 15. A: Oscillator Connections

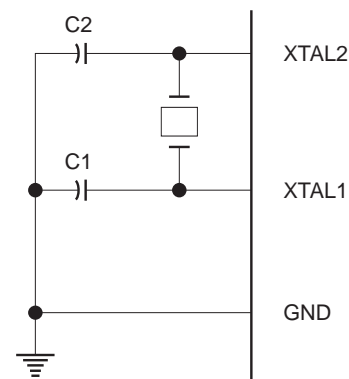
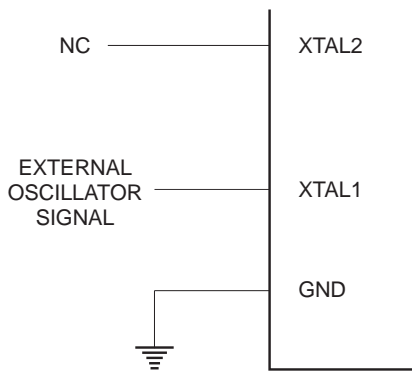


Figure 15. B: External Clock Drive Configuration



A data memory bus cycle takes twice as much time as a program memory bus cycle. Figure 17 shows the relative timing of the addresses being emitted at Ports 0 and 2 and of ALE and PSEN. ALE latches 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 is therefore available as a clock output signal. Note, however, that one ALE is skipped during the execution of the MOVX instruction.

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 lasts 12 oscillator periods or 1 μ s if the oscillator frequency is 12 MHz.

Each state is divided into a Phase 1 half and a Phase 2 half. Figure 16 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 does not require it. If the instruction being executed does not need more code bytes, the CPU ignores the extra fetch, and the Program Counter is not incremented.

Execution of a one-cycle instruction (Figure 16A 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 complete 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 16(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 17 shows the signals and timing involved in program fetches when the program memory is external. If program memory is external, the program memory read strobe PSEN is normally activated twice per machine cycle, as shown in Figure 17(A).

If an access to external data memory occurs, as shown in Figure 17(B), two PSENs are skipped, because the address and data bus are being used for the data memory access.

Figure 16. State Sequences in Atmel Flash Microcontrollers

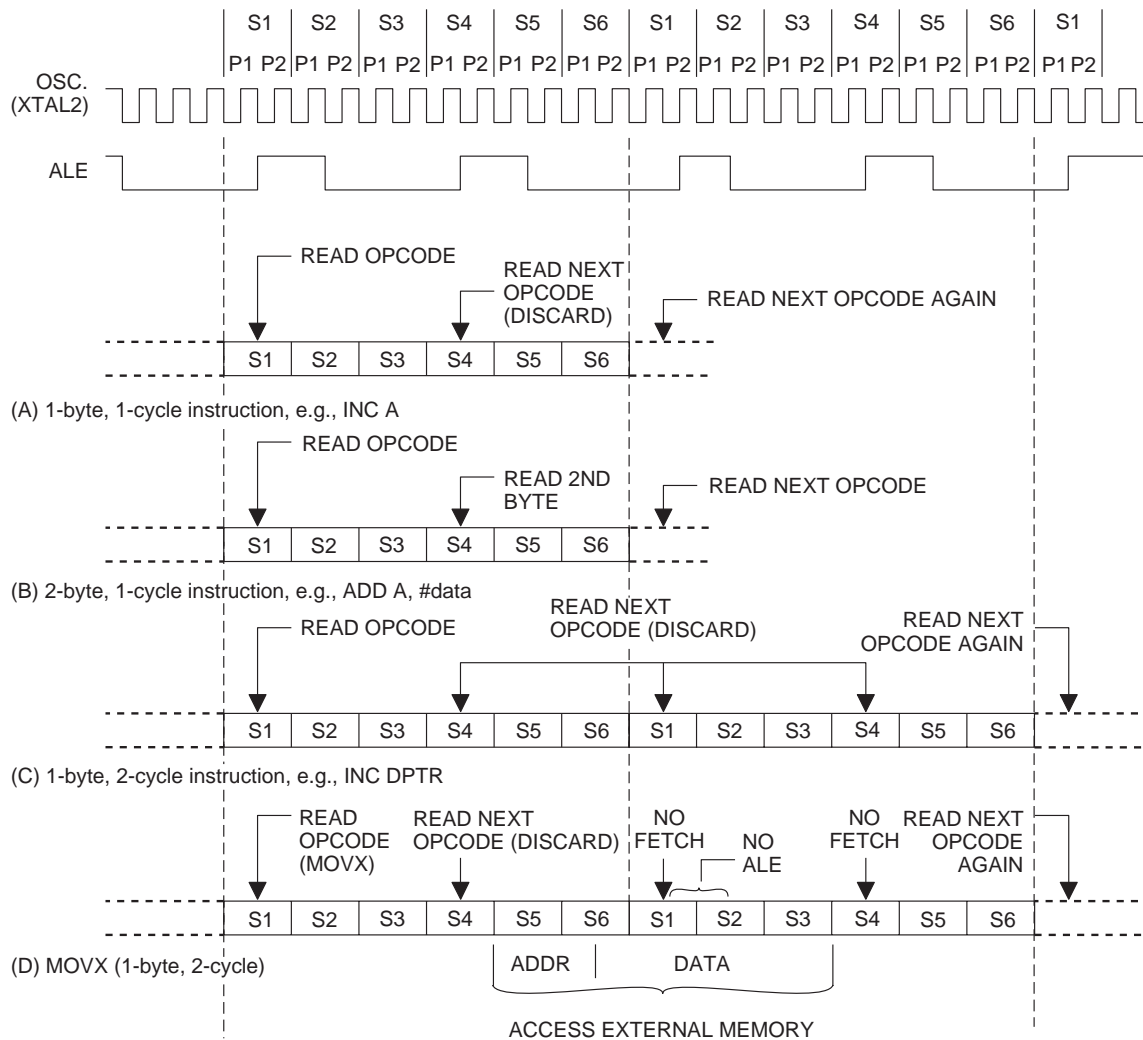
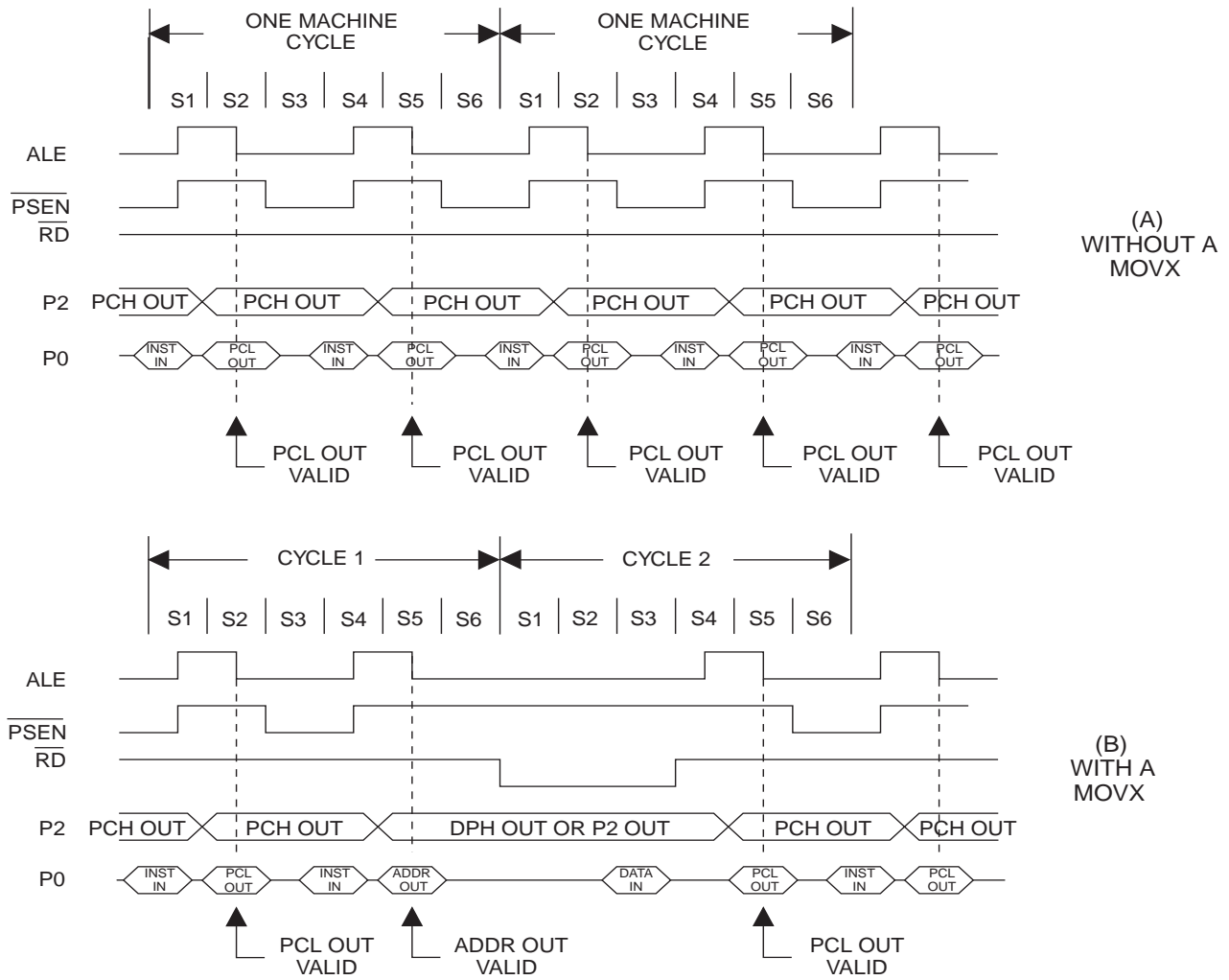


Figure 17. Bus Cycles Executing from External Program Memory



Interrupt Structure

The AT89C51 core provides 5 interrupt sources: 2 external interrupts, 2 timer interrupts, and the serial port interrupt. What follows is an overview of the interrupt structure for the AT89C51. Other Atmel Flash microcontrollers have additional interrupt sources and vectors. Refer to the data sheets on other devices for further information on their interrupts.

Interrupt Enables

Each of the interrupt sources can be individually enabled or disabled by setting or clearing the Interrupt Enable (IE) bit in the SFR. This register also contains a global disable bit, which can be cleared to disable all interrupts at once. Figure 18 shows the IE register for the AT89C51.

Figure 18. Interrupt Enable (IE) Register in the AT89C51

(MSB)								(LSB)
EA	—	—	ES	ET1	EX1	ET0	EX0	

Enable bit = 1 enables the interrupt.
 Enable bit = 0 disables it.

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.*
—	IE.5	reserved.*
ES	IE.4	Serial Port Interrupt enable bit.
ET1	IE.3	Timer 1 Overflow Interrupt enable bit.
EX1	IE.2	External Interrupt 1 enable bit.
ET0	IE.1	Timer 0 Overflow Interrupt enable bit.
EX0	IE.0	External Interrupt 0 enable bit.

*These reserved bits are used in other Atmel microcontrollers.

Interrupt Priorities

Each interrupt source can also be individually programmed to one of two priority levels by setting or clearing the Interrupt Priority (IP) bit in the SFR. Figure 19 shows the IP register in the AT89C51.

A low-priority interrupt can be interrupted by a high-priority interrupt but not by another low-priority interrupt. A high-priority interrupt can not 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 the polling sequence determines a second priority structure.

Figure 20 shows how the IE and IP registers and the polling sequence work to determine which (if any) interrupt will be serviced.

In operation, all the interrupt flags are latched into the interrupt 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, including an interrupt of equal or higher priority level already in progress.

The hardware-generated LCALL pushes the contents of the Program Counter onto the stack and reloads the PC with the beginning address of the service routine. As previously noted (Figure 4), the service routine for each interrupt begins at a fixed location.

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

Figure 19. IP (Interrupt Priority) Register in the AT89C51

(MSB)							(LSB)
—	—	—	PS	PT1	PX1	PT0	PX0

Priority bit = 1 assigns high priority.
 Priority bit = 0 assigns low priority.

Symbol	Position	Function
—	IP.7	reserved.*
—	IP.6	reserved.*
—	IP.5	reserved.*
PS	IP.4	Serial Port Interrupt priority bit.
PT1	IP.3	Timer 1 Interrupt priority bit.
PX1	IP.2	External Interrupt 1 priority bit.
PT0	IP.1	Timer 0 Interrupt priority bit.
PX0	IP.0	External Interrupt 0 priority bit.

*These reserved bits are used in other Atmel microcontrollers.

Simulating a Third Priority Level in Software

Some applications require more than the two priority levels that are provided by on-chip hardware in Atmel Flash microcontrollers. In these cases, relatively simple software can be written to produce the same effect as a third priority level.

First, interrupts that require higher priority than 1 are assigned to priority 1 in the IP 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)

```
*****
```

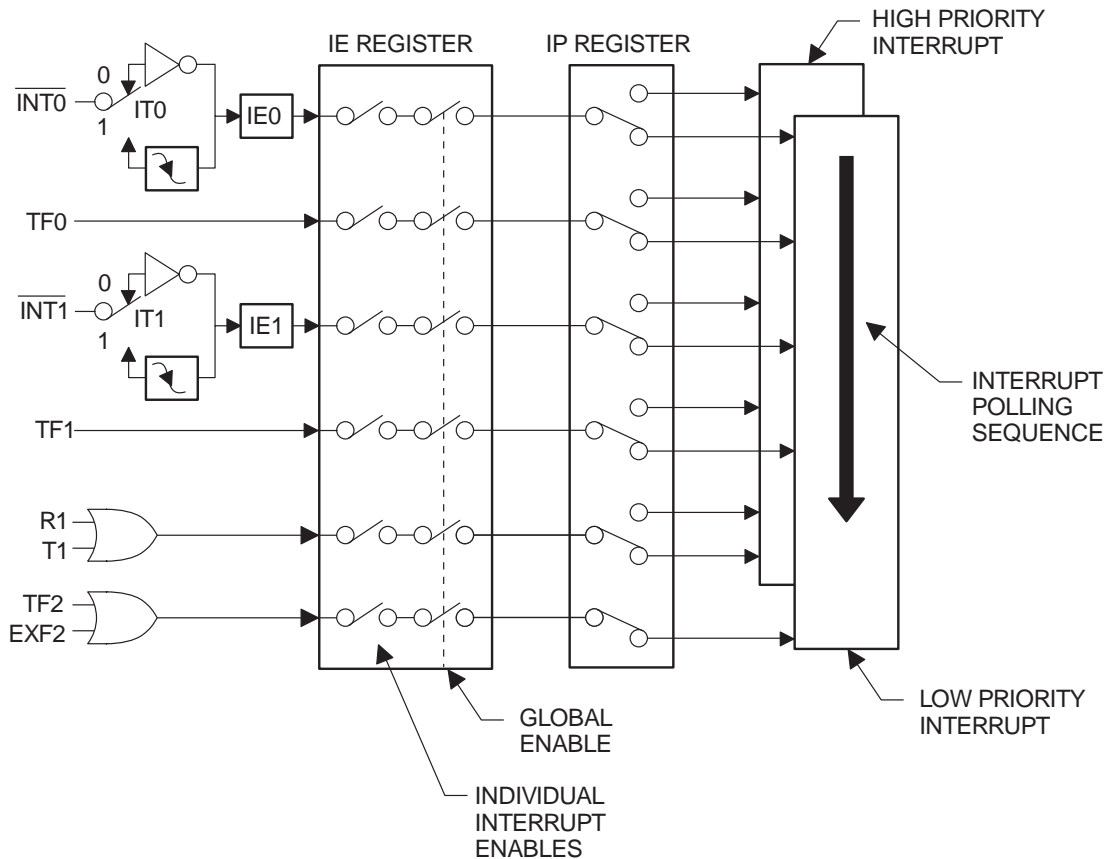
```
POP IE
RET
```

```
LABEL: RETI
```

As soon as any priority 1 interrupt is acknowledged, the IE register is redefined 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 enabled priority 1 interrupt 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 ms (at 12 MHz) to priority 1 interrupts.

Figure 20. AT89 Interrupt Control System



Note: Only on AT89C52/AT89LV52/AT89S8252