PROGRAMMER’S GUIDE AND INSTRUCTION SET

Memory Organization

Program Memory
The 80C51 has separate address spaces for program and data memory. The Program memory can be up to 64k bytes long. The lower 4k can reside on-chip. Figure 1 shows a map of the 80C51 program memory.

The 80C51 can address up to 64k bytes of data memory to the chip. The MOVX instruction is used to access the external data memory.

The 80C51 has 128 bytes of on-chip RAM, plus a number of Special Function Registers (SFRs). The lower 128 bytes of RAM can be accessed either by direct addressing (MOV data addr) or by indirect addressing (MOV @Ri). Figure 2 shows the Data Memory organization.

Direct and Indirect Address Area
The 128 bytes of RAM which can be accessed by both direct and indirect addressing can be divided into three segments as listed below and shown in Figure 3.

1. Register Banks 0-3: Locations 0 through 1FH (32 bytes). The device after reset defaults to register bank 0. To use the other register banks, the user must select them in software. Each register bank contains eight 1-byte registers 0 through 7. Reset initializes the stack pointer to location 07H, and it is incremented once to start from location 08H, which is the first register (R0) of the second register bank. Thus, in order to use more than one register bank, the SP should be initialized to a different location of the RAM where it is not used for data storage (i.e., the higher part of the RAM).

2. Bit Addressable Area: 16 bytes have been assigned for this segment, 20H-2FH. Each one of the 128 bits of this segment can be directly addressed (0-7FH). The bits can be referred to in two ways, both of which are acceptable by most assemblers. One way is to refer to their address (i.e., 0-7FH). The other way is with reference to bytes 20H to 2FH. Thus, bits 0-7 can also be referred to as bits 20.0-20.7, and bits 8-FH are the same as 21.0-21.7, and so on. Each of the 16 bytes in this segment can also be addressed as a byte.

3. Scratch Pad Area: 30H through 7FH are available to the user as data RAM. However, if the stack pointer has been initialized to this area, enough bytes should be left aside to prevent SP data destruction.

Figure 2 shows the different segments of the on-chip RAM.
**Figure 2. 80C51 Data Memory**

![80C51 Data Memory Diagram](image)

**Figure 3. 128 Bytes of RAM Direct and Indirect Addressable**

![128 Bytes of RAM Direct and Indirect Addressable Diagram](image)
### Table 1. 80C51 Special Function Registers

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>DESCRIPTION</th>
<th>DIRECT ADDRESS</th>
<th>BIT ADDRESS, SYMBOL, OR ALTERNATIVE PORT FUNCTION</th>
<th>RESET VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>ACC*</td>
<td>Accumulator</td>
<td>E0H</td>
<td>E7</td>
<td>E6</td>
</tr>
<tr>
<td>B*</td>
<td>B register</td>
<td>F0H</td>
<td>F7</td>
<td>F6</td>
</tr>
<tr>
<td>DPTR</td>
<td>Data pointer (2 bytes)</td>
<td>83H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPH</td>
<td>Data pointer high</td>
<td>82H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPL</td>
<td>Data pointer low</td>
<td></td>
<td>AF</td>
<td>AE</td>
</tr>
<tr>
<td>IE*</td>
<td>Interrupt enable</td>
<td>A8H</td>
<td>EA</td>
<td>–</td>
</tr>
<tr>
<td>IP*</td>
<td>Interrupt priority</td>
<td>B8H</td>
<td>BF</td>
<td>BE</td>
</tr>
<tr>
<td>P0*</td>
<td>Port 0</td>
<td>80H</td>
<td>AD7</td>
<td>AD6</td>
</tr>
<tr>
<td>P1*</td>
<td>Port 1</td>
<td>90H</td>
<td>97</td>
<td>96</td>
</tr>
<tr>
<td>P2*</td>
<td>Port 2</td>
<td>A0H</td>
<td>A15</td>
<td>A14</td>
</tr>
<tr>
<td>P3*</td>
<td>Port 3</td>
<td>B0H</td>
<td>RD</td>
<td>WR</td>
</tr>
<tr>
<td>PCON†</td>
<td>Power control</td>
<td>87H</td>
<td>SMOD</td>
<td>–</td>
</tr>
<tr>
<td>PSW*</td>
<td>Program status word</td>
<td>D0H</td>
<td>CY</td>
<td>AC</td>
</tr>
<tr>
<td>SBUF</td>
<td>Serial data buffer</td>
<td>99H</td>
<td>9F</td>
<td>9E</td>
</tr>
<tr>
<td>SCON*</td>
<td>Serial controller</td>
<td>98H</td>
<td>SM0</td>
<td>SM1</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
<td>81H</td>
<td>8F</td>
<td>8E</td>
</tr>
<tr>
<td>TCON*</td>
<td>Timer control</td>
<td>88H</td>
<td>TF1</td>
<td>TR1</td>
</tr>
<tr>
<td>TH0</td>
<td>Timer high 0</td>
<td>8CH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TH1</td>
<td>Timer high 1</td>
<td>8DH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TL0</td>
<td>Timer low 0</td>
<td>8AH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TL1</td>
<td>Timer low 1</td>
<td>8BH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMOD</td>
<td>Timer mode</td>
<td>89H</td>
<td>GATE</td>
<td>C/T</td>
</tr>
</tbody>
</table>

**NOTES:**
- Bit addressable
- Bits GF1, GF0, PD, and IDL of the PCON register are not implemented on the NMOS 8051/8031.
Figure 4. SFR Memory Map
Those SFRs that have their bits assigned for various functions are listed in this section. A brief description of each bit is provided for quick reference. For more detailed information refer to the Architecture Chapter of this book.

**PSW: PROGRAM STATUS WORD. BIT ADDRESSABLE.**

<table>
<thead>
<tr>
<th>CY</th>
<th>AC</th>
<th>F0</th>
<th>RS1</th>
<th>RS0</th>
<th>OV</th>
<th>–</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>CY</td>
<td>PSW.7</td>
<td>Carry Flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td>PSW.6</td>
<td>Auxiliary Carry Flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>PSW.5</td>
<td>Flag 0 available to the user for general purpose.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS1</td>
<td>PSW.4</td>
<td>Register Bank selector bit 1 (SEE NOTE 1).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS0</td>
<td>PSW.3</td>
<td>Register Bank selector bit 0 (SEE NOTE 1).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OV</td>
<td>PSW.2</td>
<td>Overflow Flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>PSW.1</td>
<td>Usable as a general purpose flag.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>PSW.0</td>
<td>Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of ‘1’ bus in the accumulator.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:**
1. The value presented by RS0 and RS1 selects the corresponding register bank.

<table>
<thead>
<tr>
<th>RS1</th>
<th>RS0</th>
<th>REGISTER BANK</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00H-07H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>08H-0FH</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10H-17H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>18H-1FH</td>
</tr>
</tbody>
</table>

**PCON: POWER CONTROL REGISTER. NOT BIT ADDRESSABLE.**

<table>
<thead>
<tr>
<th>SMOD</th>
<th>–</th>
<th>–</th>
<th>–</th>
<th>GF1</th>
<th>GF0</th>
<th>PD</th>
<th>IDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMOD</td>
<td>Double baud rate bit. If Timer 1 is used to generate baud rate and SMOD = 1, the baud rate is doubled when the Serial Port is used in modes 1, 2, or 3.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>Not implemented, reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>Not implemented reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>Not implemented reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GF1</td>
<td>General purpose flag bit.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GF0</td>
<td>General purpose flag bit.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PD</td>
<td>Power Down Bit. Setting this bit activates Power Down operation in the 80C51. (Available only in CMOS.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td>Idle mode bit. Setting this bit activates Idle Mode operation in the 80C51. (Available only in CMOS.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If 1s are written to PD and IDL at the same time, PD takes precedence.

* User software should not write 1s to reserved bits. These bits may be used in future 8051 products to invoke new features.
INTERRUPTS:

To use any of the interrupts in the 80C51 Family, the following three steps must be taken.

1. Set the EA (enable all) bit in the IE register to 1.
2. Set the corresponding individual interrupt enable bit in the IE register to 1.
3. Begin the interrupt service routine at the corresponding Vector Address of that interrupt. See Table below.

<table>
<thead>
<tr>
<th>INTERRUPT SOURCE</th>
<th>VECTOR ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE0</td>
<td>0003H</td>
</tr>
<tr>
<td>TF0</td>
<td>000BH</td>
</tr>
<tr>
<td>IE1</td>
<td>0013H</td>
</tr>
<tr>
<td>TF1</td>
<td>001BH</td>
</tr>
<tr>
<td>RI &amp; TI</td>
<td>0023H</td>
</tr>
</tbody>
</table>

In addition, for external interrupts, pins INT0 and INT1 (P3.2 and P3.3) must be set to 1, and depending on whether the interrupt is to be level or transition activated, bits IT0 or IT1 in the TCON register may need to be set to 1.

ITx = 0 level activated
ITx = 1 transition activated

IE: INTERRUPT ENABLE REGISTER. BIT ADDRESSABLE.

If the bit is 0, the corresponding interrupt is disabled. If the bit is 1, the corresponding interrupt is enabled.

<table>
<thead>
<tr>
<th>EA</th>
<th>IE.7</th>
<th>−</th>
<th>−</th>
<th>ES</th>
<th>ET1</th>
<th>EX1</th>
<th>ET0</th>
<th>EX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EA</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>ES</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>ES</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>−</td>
<td>IE.6</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>−</td>
<td>IE.5</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>ES</td>
<td>IE.4</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>ET1</td>
<td>IE.3</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>EX1</td>
<td>IE.2</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>ET0</td>
<td>IE.1</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>EX0</td>
<td>IE.0</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

EA: 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.

— Not implemented, reserved for future use.*

ES: Enable or disable the serial port interrupt.

ET1: Enable or disable the Timer 1 overflow interrupt.

EX1: Enable or disable External Interrupt 1.

ET0: Enable or disable the Timer 0 overflow interrupt.

EX0: Enable or disable External Interrupt 0.

* User software should not write 1s to reserved bits. These bits may be used in future 80C51 products to invoke new features.
ASSIGNING HIGHER PRIORITY TO ONE OR MORE INTERRUPTS:
In order to assign higher priority to an interrupt the corresponding bit in the IP register must be set to 1.
Remember that while an interrupt service is in progress, it cannot be interrupted by a lower or same level interrupt.

PRIORITY WITHIN LEVEL:
Priority within level is only to resolve simultaneous requests of the same priority level.
From high to low, interrupt sources are listed below:
IE0
TF0
IE1
TF1
RI or TI

IP: INTERRUPT PRIORITY REGISTER. BIT ADDRESSABLE.
If the bit is 0, the corresponding interrupt has a lower priority and if the bit is 1 the corresponding interrupt has a higher priority.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>PS</th>
<th>PT1</th>
<th>PX1</th>
<th>PT0</th>
<th>PX0</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>–</td>
<td>–</td>
<td>IP.7</td>
<td>Not implemented, reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>–</td>
<td>IP.6</td>
<td>Not implemented, reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>–</td>
<td>IP.5</td>
<td>Not implemented, reserved for future use.*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PS</td>
<td>IP.4</td>
<td>Defines the Serial Port interrupt priority level.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT1</td>
<td>IP.3</td>
<td>Defines the Timer 1 interrupt priority level.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PX1</td>
<td>IP.2</td>
<td>Defines External Interrupt 1 priority level.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PT0</td>
<td>IP.1</td>
<td>Defines the Timer 0 interrupt priority level.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PX0</td>
<td>IP.0</td>
<td>Defines the External Interrupt 0 priority level.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* User software should not write 1s to reserved bits. These bits may be used in future 80C51 products to invoke new features.
TCON: TIMER/COUNTER CONTROL REGISTER. BIT ADDRESSABLE.

<table>
<thead>
<tr>
<th>TF1</th>
<th>TR1</th>
<th>TF0</th>
<th>TR0</th>
<th>IE1</th>
<th>IT1</th>
<th>IE0</th>
<th>IT0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TF1</td>
<td>TCON.7</td>
<td>Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as processor vectors to the interrupt service routine.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR1</td>
<td>TCON.6</td>
<td>Timer 1 run control bit. Set/cleared by software to turn Timer/Counter 1 ON/OFF.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TF0</td>
<td>TCON.5</td>
<td>Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as processor vectors to the service routine.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TR0</td>
<td>TCON.4</td>
<td>Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IE1</td>
<td>TCON.3</td>
<td>External Interrupt 1 edge flag. Set by hardware when External Interrupt edge is detected. Cleared by hardware when interrupt is processed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT1</td>
<td>TCON.2</td>
<td>Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IE0</td>
<td>TCON.1</td>
<td>External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT0</td>
<td>TCON.0</td>
<td>Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TMOD: TIMER/COUNTER MODE CONTROL REGISTER. NOT BIT ADDRESSABLE.

<table>
<thead>
<tr>
<th>GATE</th>
<th>C/T</th>
<th>M1</th>
<th>M0</th>
<th>GATE</th>
<th>C/T</th>
<th>M1</th>
<th>M0</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATE</td>
<td>When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/T</td>
<td>Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from Tx input pin).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>Mode selector bit. (NOTE 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M0</td>
<td>Mode selector bit. (NOTE 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE 1:

<table>
<thead>
<tr>
<th>M1</th>
<th>M0</th>
<th>Operating Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>13-bit Timer (8048 compatible)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>16-bit Timer/Counter</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>8-bit Auto-Reload Timer/Counter</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit Timer and is controlled by Timer 1 control bits.</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(Timer 1) Timer/Counter 1 stopped.</td>
</tr>
</tbody>
</table>
TIMER SET-UP

Tables 2 through 5 give some values for TMOD which can be used to set up Timer 0 in different modes.

It is assumed that only one timer is being used at a time. If it is desired to run Timers 0 and 1 simultaneously, in any mode, the value in TMOD for Timer 0 must be ORed with the value shown for Timer 1 (Tables 5 and 6).

For example, if it is desired to run Timer 0 in mode 1 GATE (external control), and Timer 1 in mode 2 COUNTER, then the value that must be loaded into TMOD is 69H (09H from Table 2 ORed with 60H from Table 5).

Moreover, it is assumed that the user, at this point, is not ready to turn the timers on and will do that at a different point in the program by setting bit TRx (in TCON) to 1.

TIMER/COUNTER 0

Table 2. As a Timer:

<table>
<thead>
<tr>
<th>MODE</th>
<th>TIMER 0 FUNCTION</th>
<th>TMOD</th>
<th>INTERNAL CONTROL (NOTE 1)</th>
<th>EXTERNAL CONTROL (NOTE 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13-bit Timer</td>
<td>00H</td>
<td>08H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16-bit Timer</td>
<td>01H</td>
<td>09H</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8-bit Auto-Reload</td>
<td>02H</td>
<td>0AH</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Two 8-bit Timers</td>
<td>03H</td>
<td>0BH</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. As a Counter:

<table>
<thead>
<tr>
<th>MODE</th>
<th>COUNTER 0 FUNCTION</th>
<th>TMOD</th>
<th>INTERNAL CONTROL (NOTE 1)</th>
<th>EXTERNAL CONTROL (NOTE 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13-bit Timer</td>
<td>04H</td>
<td>0CH</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16-bit Timer</td>
<td>05H</td>
<td>0DH</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8-bit Auto-Reload</td>
<td>06H</td>
<td>0EH</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>One 8-bit Counter</td>
<td>07H</td>
<td>0FH</td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1. The timer is turned ON/OFF by setting/clearing bit TR0 in the software.
2. The Timer is turned ON/OFF by the 1-to-0 transition on INT0 (P3.2) when TR0 = 1 (hardware control).
## TIMER/COUNTER 1

### Table 4. As a Timer:

<table>
<thead>
<tr>
<th>MODE</th>
<th>TIMER 1 FUNCTION</th>
<th>TMOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INTERNAL CONTROL (NOTE 1)</td>
</tr>
<tr>
<td>0</td>
<td>13-bit Timer</td>
<td>00H</td>
</tr>
<tr>
<td>1</td>
<td>16-bit Timer</td>
<td>10H</td>
</tr>
<tr>
<td>2</td>
<td>8-bit Auto-Reload</td>
<td>20H</td>
</tr>
<tr>
<td>3</td>
<td>Does not run</td>
<td>30H</td>
</tr>
</tbody>
</table>

### Table 5. As a Counter:

<table>
<thead>
<tr>
<th>MODE</th>
<th>COUNTER 1 FUNCTION</th>
<th>TMOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INTERNAL CONTROL (NOTE 1)</td>
</tr>
<tr>
<td>0</td>
<td>13-bit Timer</td>
<td>40H</td>
</tr>
<tr>
<td>1</td>
<td>16-bit Timer</td>
<td>50H</td>
</tr>
<tr>
<td>2</td>
<td>8-bit Auto-Reload</td>
<td>60H</td>
</tr>
<tr>
<td>3</td>
<td>Not available</td>
<td>–</td>
</tr>
</tbody>
</table>

**NOTES:**
1. The timer is turned ON/OFF by setting/clearing bit TR1 in the software.
2. The Timer is turned ON/OFF by the 1-to-0 transition on INT1 (P3.2) when TR1 = 1 (hardware control).
80C51 Family

80C51 family programmer’s guide
and instruction set

SCON: SERIAL PORT CONTROL REGISTER. BIT ADDRESSABLE.

<table>
<thead>
<tr>
<th>SM0</th>
<th>SM1</th>
<th>SM2</th>
<th>REN</th>
<th>TB8</th>
<th>RB8</th>
<th>TI</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM0</td>
<td>SCON.7</td>
<td>Serial Port mode specifier. (NOTE 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM1</td>
<td>SCON.6</td>
<td>Serial Port mode specifier. (NOTE 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM2</td>
<td>SCON.5</td>
<td>Enables the multiprocessor communication feature in modes 2 &amp; 3. In mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0. (See Table 6.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REN</td>
<td>SCON.4</td>
<td>Set/Cleared by software to Enable/Disable reception.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB8</td>
<td>SCON.3</td>
<td>The 9th bit that will be transmitted in modes 2 &amp; 3. Set/Cleared by software.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RB8</td>
<td>SCON.2</td>
<td>In modes 2 &amp; 3, is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TI</td>
<td>SCON.1</td>
<td>Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. Must be cleared by software.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RI</td>
<td>SCON.0</td>
<td>Receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in the other modes (except see SM2). Must be cleared by software.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE 1:

<table>
<thead>
<tr>
<th>SM0</th>
<th>SM1</th>
<th>Mode</th>
<th>Description</th>
<th>Baud Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Shift Register</td>
<td>Fosc/12</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8-bit UART</td>
<td>Variable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>9-bit UART</td>
<td>Fosc/64 or Fosc/32</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>9-bit UART</td>
<td>Variable</td>
</tr>
</tbody>
</table>

SERIAL PORT SET-UP:

Table 6.

<table>
<thead>
<tr>
<th>MODE</th>
<th>SCON</th>
<th>SM2 VARIATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10H</td>
<td>Single Processor Environment (SM2 = 0)</td>
</tr>
<tr>
<td>1</td>
<td>50H</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>90H</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D0H</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>NA</td>
<td>Multiprocessor Environment (SM2 = 1)</td>
</tr>
<tr>
<td>1</td>
<td>70H</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B0H</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F0H</td>
<td></td>
</tr>
</tbody>
</table>

GENERATING BAUD RATES

Serial Port in Mode 0:

Mode 0 has a fixed baud rate which is 1/12 of the oscillator frequency. To run the serial port in this mode none of the Timer/Counters need to be set up. Only the SCON register needs to be defined.

\[
\text{Baud Rate} = \frac{\text{Osc Freq}}{12}
\]

Serial Port in Mode 1:

Mode 1 has a variable baud rate. The baud rate is generated by Timer 1.
USING TIMER/COUNTER 1 TO GENERATE BAUD RATES:

For this purpose, Timer 1 is used in mode 2 (Auto-Reload). Refer to Timer Setup section of this chapter.

\[
\text{Baud Rate} = \frac{K \times \text{Osc Freq}}{32 \times 12 \times (256 - (TH1))}
\]

If SMOD = 0, then K = 1.
If SMOD = 1, then K = 2 (SMOD is in the PCON register).

Most of the time the user knows the baud rate and needs to know the reload value for TH1.

\[
TH1 = 256 - \frac{K \times \text{Osc Freq}}{384 \times \text{baud rate}}
\]

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate. In this case, the user may have to choose another crystal frequency.

Since the PCON register is not bit addressable, one way to set the bit is logical ORing the PCON register (i.e., ORL PCON,#80H). The address of PCON is 87H.

SERIAL PORT IN MODE 2:

The baud rate is fixed in this mode and is 1/32 or 1/64 of the oscillator frequency, depending on the value of the SMOD bit in the PCON register.

In this mode none of the Timers are used and the clock comes from the internal phase 2 clock.

SMOD = 1, Baud Rate = 1/32 Osc Freq.
SMOD = 0, Baud Rate = 1/64 Osc Freq.

To set the SMOD bit: ORL PCON,#80H. The address of PCON is 87H.

SERIAL PORT IN MODE 3:

The baud rate in mode 3 is variable and sets up exactly the same as in mode 1.
80C51 FAMILY INSTRUCTION SET

Table 7. 80C51 Instruction Set Summary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Flag</th>
<th>Instruction</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>X</td>
<td>X</td>
<td>AC</td>
</tr>
<tr>
<td>ADDC</td>
<td>X</td>
<td>X</td>
<td>CPL C</td>
</tr>
<tr>
<td>SUBB</td>
<td>X</td>
<td>X</td>
<td>ANL C.bit</td>
</tr>
<tr>
<td>MUL</td>
<td>0</td>
<td>X</td>
<td>ANL C.bit</td>
</tr>
<tr>
<td>DIV</td>
<td>0</td>
<td>X</td>
<td>ORL C.bit</td>
</tr>
<tr>
<td>DA</td>
<td>X</td>
<td>X</td>
<td>ORL C.bit</td>
</tr>
<tr>
<td>RRC</td>
<td>X</td>
<td>MOV C bit</td>
<td></td>
</tr>
<tr>
<td>RLC</td>
<td>X</td>
<td>CJNE</td>
<td></td>
</tr>
<tr>
<td>SETB C</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes on instruction set and addressing modes:

- **Rn**: Register R7-R0 of the currently selected Register Bank.
- **direct**: 8-bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR [i.e., I/O port, control register, status register, etc. (128-255)].
- **@Ri**: 8-bit internal data RAM location (0-255) addressed indirectly through register R1 or R0.
- **#data**: 8-bit constant included in the instruction.
- **#data 16**: 16-bit constant included in the instruction.
- **addr 16**: 16-bit destination address. Used by LCALL and LJMP. A branch can be anywhere within the 64k-byte Program Memory address space.
- **addr 11**: 11-bit destination address. Used by ACALL and AJMP. The branch will be within the same 2k-byte page of program memory as the first byte of the following instruction.
- **rel**: Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is –128 to +127 bytes relative to first byte of the following instruction.
- **bit**: Direct Addressed bit in Internal Data RAM or Special Function Register.

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DESCRIPTION</th>
<th>BYTE</th>
<th>OSCILLATOR PERIOD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETIC OPERATIONS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>A,Rn</td>
<td>Add register to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>A,direct</td>
<td>Add direct byte to Accumulator</td>
<td>2</td>
</tr>
<tr>
<td>ADD</td>
<td>A,@Ri</td>
<td>Add indirect RAM to Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>ADD</td>
<td>A,#data</td>
<td>Add immediate data to Accumulator</td>
<td>2</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,Rn</td>
<td>Add register to Accumulator with carry</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,direct</td>
<td>Add direct byte to Accumulator with carry</td>
<td>2</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,@Ri</td>
<td>Add indirect RAM to Accumulator with carry</td>
<td>1</td>
</tr>
<tr>
<td>ADDC</td>
<td>A,#data</td>
<td>Add immediate data to ACC with carry</td>
<td>2</td>
</tr>
<tr>
<td>SUBB</td>
<td>A,Rn</td>
<td>Subtract Register from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>SUBB</td>
<td>A,direct</td>
<td>Subtract direct byte from ACC with borrow</td>
<td>2</td>
</tr>
<tr>
<td>SUBB</td>
<td>A,@Ri</td>
<td>Subtract indirect RAM from ACC with borrow</td>
<td>1</td>
</tr>
<tr>
<td>SUBB</td>
<td>A,#data</td>
<td>Subtract immediate data from ACC with borrow</td>
<td>2</td>
</tr>
<tr>
<td>INC</td>
<td>A</td>
<td>Increment Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>INC</td>
<td>Rn</td>
<td>Increment register</td>
<td>1</td>
</tr>
</tbody>
</table>

(1) Note that operations on SFR byte address 208 or bit addresses 209-215 (i.e., the PSW or bits in the PSW) will also affect flag settings.

All mnemonics copyrighted © Intel Corporation 1980
### 80C51 Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DESCRIPTION</th>
<th>BYTE</th>
<th>OSCILLATOR PERIOD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARITHMETIC OPERATIONS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>direct Increment direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>INC</td>
<td>@Ri Increment indirect RAM</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC</td>
<td>A Decrement Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC</td>
<td>Rn Decrement Register</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC</td>
<td>direct Decrement direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DEC</td>
<td>@Ri Decrement indirect RAM</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>INC</td>
<td>DPTR Increment Data Pointer</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>MUL</td>
<td>AB Multiply A and B</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>DIV</td>
<td>AB Divide A by B</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>DA</td>
<td>A Decimal Adjust Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td><strong>LOGICAL OPERATIONS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ANL</td>
<td>A,Rn AND Register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>A,direct AND direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>A,@Ri AND indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>A,#data AND immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>direct,A AND Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>direct,#data AND immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>ORL</td>
<td>A,Rn OR register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ORL</td>
<td>A,direct OR direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL</td>
<td>A,@Ri OR indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ORL</td>
<td>A,#data OR immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL</td>
<td>direct,A OR Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL</td>
<td>direct,#data OR immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>XRL</td>
<td>A,Rn Exclusive-OR register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XRL</td>
<td>A,direct Exclusive-OR direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL</td>
<td>A,@Ri Exclusive-OR indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XRL</td>
<td>A,#data Exclusive-OR immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL</td>
<td>direct,A Exclusive-OR Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL</td>
<td>direct,#data Exclusive-OR immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>CLR</td>
<td>A Clear Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>CPL</td>
<td>A Complement Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RL</td>
<td>A Rotate Accumulator left</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RLC</td>
<td>A Rotate Accumulator left through the carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RR</td>
<td>A Rotate Accumulator right</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RRC</td>
<td>A Rotate Accumulator right through the carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SWAP</td>
<td>A Swap nibbles within the Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td><strong>DATA TRANSFER</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>A,Rn Move register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOV</td>
<td>A,direct Move direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV</td>
<td>A,@Ri Move indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1980
### Table 7. 80C51 Instruction Set Summary (Continued)

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DESCRIPTION</th>
<th>BYTE</th>
<th>OSCILLATOR PERIOD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA TRANSFER</strong> (Continued)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>A,#data</td>
<td>Move immediate data to Accumulator</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>Rn,A</td>
<td>Move Accumulator to register</td>
<td>1</td>
</tr>
<tr>
<td>MOV</td>
<td>Rn,direct</td>
<td>Move direct byte to register</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>RN,#data</td>
<td>Move immediate data to register</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>direct,A</td>
<td>Move Accumulator to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>direct,Rn</td>
<td>Move register to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>direct,direct</td>
<td>Move direct byte to direct</td>
<td>3</td>
</tr>
<tr>
<td>MOV</td>
<td>direct,@Ri</td>
<td>Move indirect RAM to direct byte</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>direct,#data</td>
<td>Move immediate data to direct byte</td>
<td>3</td>
</tr>
<tr>
<td>MOV</td>
<td>@Ri,A</td>
<td>Move Accumulator to indirect RAM</td>
<td>1</td>
</tr>
<tr>
<td>MOV</td>
<td>@Ri,direct</td>
<td>Move direct byte to indirect RAM</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>@Ri,#data</td>
<td>Move immediate data to indirect RAM</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>DPTR,#data16</td>
<td>Load Data Pointer with a 16-bit constant</td>
<td>3</td>
</tr>
<tr>
<td>MOVC</td>
<td>A,@A+DPTR</td>
<td>Move Code byte relative to DPTR to A&lt;sub&gt;CC&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td>MOVC</td>
<td>A,@A+PC</td>
<td>Move Code byte relative to PC to A&lt;sub&gt;CC&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td>MOVCX</td>
<td>A,Ri</td>
<td>Move external RAM (8-bit addr) to A&lt;sub&gt;CC&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td>MOVCX</td>
<td>A,DPTR</td>
<td>Move external RAM (16-bit addr) to A&lt;sub&gt;CC&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td>MOVCX</td>
<td>A,Ri,A</td>
<td>Move A&lt;sub&gt;CC&lt;/sub&gt; to external RAM (8-bit addr)</td>
<td>1</td>
</tr>
<tr>
<td>MOVCX</td>
<td>@DPTR,A</td>
<td>Move A&lt;sub&gt;CC&lt;/sub&gt; to external RAM (16-bit addr)</td>
<td>1</td>
</tr>
<tr>
<td>PUSH</td>
<td>direct</td>
<td>Push direct byte onto stack</td>
<td>2</td>
</tr>
<tr>
<td>POP</td>
<td>direct</td>
<td>Pop direct byte from stack</td>
<td>2</td>
</tr>
<tr>
<td>XCH</td>
<td>A,Rn</td>
<td>Exchange register with Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XCH</td>
<td>A,direct</td>
<td>Exchange direct byte with Accumulator</td>
<td>2</td>
</tr>
<tr>
<td>XCH</td>
<td>A,Ri</td>
<td>Exchange indirect RAM with Accumulator</td>
<td>1</td>
</tr>
<tr>
<td>XCHD</td>
<td>A,Ri</td>
<td>Exchange low-order digit indirect RAM with A&lt;sub&gt;CC&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td><strong>BOOLEAN VARIABLE MANIPULATION</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>C</td>
<td>Clear carry</td>
<td>1</td>
</tr>
<tr>
<td>CLR</td>
<td>bit</td>
<td>Clear direct bit</td>
<td>2</td>
</tr>
<tr>
<td>SETB</td>
<td>C</td>
<td>Set carry</td>
<td>1</td>
</tr>
<tr>
<td>SETB</td>
<td>bit</td>
<td>Set direct bit</td>
<td>2</td>
</tr>
<tr>
<td>CPL</td>
<td>C</td>
<td>Complement carry</td>
<td>1</td>
</tr>
<tr>
<td>CPL</td>
<td>bit</td>
<td>Complement direct bit</td>
<td>2</td>
</tr>
<tr>
<td>ANL</td>
<td>C,bit</td>
<td>AND direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ANL</td>
<td>C/,bit</td>
<td>AND complement of direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ORL</td>
<td>C,bit</td>
<td>OR direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>ORL</td>
<td>C/,bit</td>
<td>OR complement of direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>C,bit</td>
<td>Move direct bit to carry</td>
<td>2</td>
</tr>
<tr>
<td>MOV</td>
<td>bit,C</td>
<td>Move carry to direct bit</td>
<td>2</td>
</tr>
<tr>
<td>JC</td>
<td>rel</td>
<td>Jump if carry is set</td>
<td>2</td>
</tr>
<tr>
<td>JNC</td>
<td>rel</td>
<td>Jump if carry not set</td>
<td>2</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1980
<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>DESCRIPTION</th>
<th>BYTE</th>
<th>OSCILLATOR PERIOD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BOOLEAN VARIABLE MANIPULATION (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JB rel</td>
<td>Jump if direct bit is set</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>JNB rel</td>
<td>Jump if direct bit is not set</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>JBC bit,rel</td>
<td>Jump if direct bit is set and clear bit</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>PROGRAM BRANCHING</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACALL addr11</td>
<td>Absolute subroutine call</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>LCALL addr16</td>
<td>Long subroutine call</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>RET</td>
<td>Return from subroutine</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>RETI</td>
<td>Return from interrupt</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>AJMP addr11</td>
<td>Absolute jump</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>LJMP addr16</td>
<td>Long jump</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>SJMP rel</td>
<td>Short jump (relative addr)</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JMP @A+DPTR</td>
<td>Jump indirect relative to the DPTR</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>JZ rel</td>
<td>Jump if Accumulator is zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JNZ rel</td>
<td>Jump if Accumulator is not zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>CJNE A,direct,rel</td>
<td>Compare direct byte to ACC and jump if not equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>CJNE A,#data,rel</td>
<td>Compare immediate to ACC and jump if not equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>CJNE RN,#data,rel</td>
<td>Compare immediate to register and jump if not equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>CJNE @Ri,#data,rel</td>
<td>Compare immediate to indirect and jump if not equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>DJNZ Rn,rel</td>
<td>Decrement register and jump if not zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>DJNZ direct,rel</td>
<td>Decrement direct byte and jump if not zero</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted © Intel Corporation 1980
INSTRUCTION DEFINITIONS

**ACALL  addr11**

<table>
<thead>
<tr>
<th>Function:</th>
<th>Absolute Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2k block of the program memory as the first byte of the instruction following ACALL. No flags are affected.</td>
</tr>
<tr>
<td>Example:</td>
<td>Initially SP equals 07H. The label “SUBRTN” is at program memory location 0345 H. After executing the instruction, ACALL SUBRTN at location 0123H, SP will contain 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC will contain 0345H.</td>
</tr>
<tr>
<td>Bytes:</td>
<td>2</td>
</tr>
<tr>
<td>Cycles:</td>
<td>2</td>
</tr>
</tbody>
</table>
| Encoding: | \[
<table>
<thead>
<tr>
<th>a10</th>
<th>a9</th>
<th>a8</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>a7</th>
<th>a6</th>
<th>a5</th>
<th>a4</th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
</tr>
<tr>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>---</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
<th>ACALL &lt;br&gt;(PC) ← (PC) + 2 &lt;br&gt;(SP) ← (SP) + 1 &lt;br&gt;(SP) ← (SP) + 1 &lt;br&gt;(SP) ← (PC15-8) &lt;br&gt;(PC10-0) ← page address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(PC) ← (PC) + 2 &lt;br&gt;(SP) ← (SP) + 1 &lt;br&gt;(SP) ← (SP) + 1 &lt;br&gt;(SP) ← (PC15-8) &lt;br&gt;(PC10-0) ← page address</td>
</tr>
<tr>
<td></td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>--------------------------------------------------</td>
</tr>
</tbody>
</table>
**ADD A,<src-byte>**

**Function:** Add

**Description:** ADD adds the byte variable indicated to the Accumulator, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

**Example:** The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B). The instruction, ADD A,R0

will leave 6DH (01101101B) in the Accumulator with the AC flag cleared and both the Carry flag and OV set to 1.

**ADD A,Rn**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** 0 0 1 0 1 r r r
- **Operation:** ADD (A) ← (A) + (Rn)

**ADD A,direct**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** 0 0 1 0 0 1 0 1 direct address
- **Operation:** ADD (A) ← (A) + (direct)

**ADD A,@Ri**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** 0 0 1 0 0 1 1 i
- **Operation:** ADD (A) ← (A) + ((Ri))

**ADD A,#data**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** 0 0 1 0 0 1 0 0 immediate data
- **Operation:** ADD (A) ← (A) + #data
ADD C A,<src-byte>

**Function:** Add with Carry

**Description:** ADDC simultaneously adds the byte variable indicated, the carry flag and the Accumulator contents, leaving the result in the Accumulator. The carry and auxiliary-carry flags are set, respectively, if there is a carry-out from bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry-out of bit 6 but not out of bit 7, or a carry-out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

**Example:** The Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The instruction,

`ADD C A,R0`

will leave 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.

**ADD C A,Rn**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:**
  - 00111
  - r r r
- **Operation:**
  
  $\text{ADD C } (A) \leftarrow (A) + (C) + (R_n)$

**ADD C A,direct**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:**
  - 0011001011
  - direct address
- **Operation:**
  
  $\text{ADD C } (A) \leftarrow (A) + (C) + (\text{direct})$

**ADD C A,@Ri**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:**
  - 00110111
  - i
- **Operation:**
  
  $\text{ADD C } (A) \leftarrow (A) + (C) + ((R_i))$

**ADD C A,#data**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:**
  - 00110100
  - immediate data
- **Operation:**
  
  $\text{ADD C } (A) \leftarrow (A) + (C) + \#\text{data}$
AJMP  addr11

<table>
<thead>
<tr>
<th>Function:</th>
<th>Absolute Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2k block of program memory as the first byte of the instruction following AJMP.</td>
</tr>
<tr>
<td>Example:</td>
<td>The label “JMPADR” is at program memory location 0123H. The instruction, AJMP JMPADR is at location 0345H and will load the PC with 0123H.</td>
</tr>
<tr>
<td>Bytes:</td>
<td>2</td>
</tr>
<tr>
<td>Cycles:</td>
<td>2</td>
</tr>
<tr>
<td>Encoding:</td>
<td>a10 a9 a8 0 0 0 1 a7 a6 a5 a4 a3 a2 a1 a0</td>
</tr>
<tr>
<td>Operation:</td>
<td>AJMP (PC) ← (PC) + 2</td>
</tr>
<tr>
<td></td>
<td>(PC&lt;10-0) ← page address</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function:</th>
<th>Logical-AND for byte variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>ANL performs the bitwise logical-AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.</td>
</tr>
<tr>
<td>Example:</td>
<td>If the Accumulator holds 0C3H (11000011B) and register 0 holds 55H (01010101B) then the instruction, ANL A,R0 will leave 41H (01000011B) in the Accumulator. When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the Accumulator at run-time. The instruction, ANL P1,#01110011B will clear bits 7, 3, and 2 of output port 1.</td>
</tr>
</tbody>
</table>
80C51 family programmer’s guide
and instruction set

**ANL A,Rn**
- Bytes: 1
- Cycles: 1
- Encoding: 01011rrr
- Operation: ANL \((A) \leftarrow (A) \land (R_n)\)

**ANL A,direct**
- Bytes: 2
- Cycles: 1
- Encoding: 01010101
- Operation: ANL \((A) \leftarrow (A) \land (\text{direct})\)

**ANL A,@Ri**
- Bytes: 1
- Cycles: 1
- Encoding: 0101011i
- Operation: ANL \((A) \leftarrow (A) \land ((R_i))\)

**ANL A,#data**
- Bytes: 2
- Cycles: 1
- Encoding: 01010100
- Operation: ANL \((A) \leftarrow (A) \land (#\text{data})\)

**ANL direct,A**
- Bytes: 2
- Cycles: 1
- Encoding: 01010010
- Operation: ANL \(\text{direct} \leftarrow (\text{direct}) \land (A)\)

**ANL direct,#data**
- Bytes: 3
- Cycles: 2
- Encoding: 01010011
- Operation: ANL \(\text{direct} \leftarrow (\text{direct}) \land (#\text{data})\)
ANL  C,<src-bit>

**Function:** Logical-AND for bit variables

**Description:** If the Boolean value of the source bit is a logical 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Only direct addressing is allowed for the source operand.

**Example:** Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0:

```
MOV  C,P1.0  ;LOAD CARRY WITH INPUT PIN STATE
ANL  C,ACC.7;AND CARRY WITH ACCUM. BIT 7
ANL  C,/OV  ;AND WITH INVERSE OF OVERFLOW FLAG
```

**ANL  C,bit**

| Bytes: | 2 |
| Cycles: | 2 |

**Encoding:**

```
0 0 0 1 0 0 0 1 0
```

**Operation:**

```
(C) ← (C) ∧ (bit)
```

**ANL  C,/bit**

| Bytes: | 2 |
| Cycles: | 2 |

**Encoding:**

```
0 1 1 0 0 0 0 0
```

**Operation:**

```
(C) ← (C) ∧ (bit)
```
CJNE <dest-byte>,<src-byte>,rel

Function: Compare and Jump if Not Equal

Description: CJNE compares the magnitudes of the first two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative-displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the Accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The Accumulator contains 34H. Register 7 contains 56H. The first instruction in the sequence,

```
CJNE R7,#60H,NOT_EQ
```

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H.

If the data being presented to Port 1 is also 34H, then the instruction,

```
WAIT: CJNE A,P1,WAIT
```

clears the carry flag and continues with the next instruction in sequence, since the Accumulator does equal the data read from P1. (If some other value was being input on P1, the program will loop at this point until the P1 data changes to 34H.)

CJNE A,direct,rel

Bytes: 3

Cycles: 2

Encoding:

```
   1 0 1 1 0 1 0 1
```

Operation:

```
(PC) ← (PC) + 3
IF (A) < > (direct)
THEN
   (PC) ← (PC) + relative offset
IF (A) < (direct)
THEN.
   (C) ← 1
ELSE
   (C) ← 0
```
CJNE A,#data,rel
Bytes: 3
Cycles: 2
Encoding: 0x0 101 1 0 1 0 0
Operation: (PC) ← (PC) + 3
  IF (A) <> data
      THEN
          (PC) ← (PC) + relative offset
  IF (A) < data
      THEN
          (C) ← 1
      ELSE
          (C) ← 0

CJNE Rn,#data,rel
Bytes: 3
Cycles: 2
Encoding: 0x0 101 1 1 r r r
Operation: (PC) ← (PC) + 3
  IF (Rn) <> data
      THEN
          (PC) ← (PC) + relative offset
  IF (Rn) < data
      THEN
          (C) ← 1
      ELSE
          (C) ← 0

CJNE @Ri,#data,rel
Bytes: 3
Cycles: 2
Encoding: 0x0 101 1 0 1 0 i
Operation: (PC) ← (PC) + 3
  IF ((Ri)) <> data
      THEN
          (PC) ← (PC) + relative offset
  IF ((Ri)) < data
      THEN
          (C) ← 1
      ELSE
          (C) ← 0
### CLR A

**Function:** Clear Accumulator  
**Description:** The Accumulator is cleared (all bits reset to zero). No flags are affected.  
**Example:** The Accumulator contains 5CH (01011100B). The instruction,  
CLR A  
will leave the Accumulator set to 00H (00000000B).  
**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 0111000110  
**Operation:** CLR (A) ← 0

### CLR bit

**Function:** Clear bit  
**Description:** The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.  
**Example:** Port 1 has previously been written with 5DH (01011101B). The instruction,  
CLR P1.2  
will leave the port set to 59H (01011001B).  
**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 0111000110  
**Operation:** CLR (C) ← 0

### CLR bit

**Function:** Clear bit  
**Description:** The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.  
**Example:** Port 1 has previously been written with 5DH (01011101B). The instruction,  
CLR P1.2  
will leave the port set to 59H (01011001B).  
**Bytes:** 2  
**Cycles:** 1  
**Encoding:** 0110001010  
**Operation:** CLR (bit) ← 0
### CPL A

**Function:** Complement Accumulator

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

**Example:** The Accumulator contains 5CH (01011100B). The instruction,  
\[
\text{CPL A}
\]
will leave the Accumulator set to 0A3H (10100011B).

**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 1111 0100  
**Operation:** \( \text{CPL} (A) \leftarrow \overline{1} (A) \)

### CPL bit

**Function:** Complement bit

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

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

**Example:** Port 1 has previously been written with 5DH (01011101B). The instruction sequence,  
\[
\text{CPL P1.1} \\
\text{CPL P1.2}
\]
will leave the port set to 5BH (01011011B).

### CPL C

**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 1011 0011  
**Operation:** \( \text{CPL} (C) \leftarrow \overline{1} (C) \)

### CPL bit

**Bytes:** 2  
**Cycles:** 1  
**Encoding:** 1011 0010  
**Operation:** \( \text{CPL} \text{ bit address} \)
**Function:** Decimal-adjust Accumulator for Addition

**Description:** 
DA A adjusts the eight-bit value in the Accumulator resulting from the earlier addition of two variable (each in packed-BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If Accumulator bits 3-0 are greater than nine (xxx1010-xxx1111), or if the AC flag is one, six is added to the Accumulator, producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxx-111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn’t clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00H, 06H, 60H, or 66H to the Accumulator, depending on initial Accumulator and PSW conditions.

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

**Example:**
The Accumulator holds the value 56H (01010110B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67H (01100111B) representing the packed BCD digits of the decimal number 67. The carry flag is set. The instruction sequence,

```
ADDC A,R3
DA A
```

will first perform a standard two’s-complement binary addition, resulting in the value 0BEH (101 11110B) in the Accumulator. The carry and auxiliary carry flags will be cleared.

The Decimal Adjust instruction will then alter the Accumulator to the value 24H (00100110B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag will be set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum 56, 67, and 1 is 124.

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

```
ADD A,#99H
DA A
```

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

**Bytes:** 1

**Cycles:** 1

**Encoding:**

```
1 1 0 1 0 1 0 0
```

**Operation:**

- contents of Accumulator are BCD
  
  IF \( [(A_{3:0}) > 9] \lor [(AC) = 1] \)
  THEN \( (A_{3:0}) \leftarrow (A_{3:0}) + 6 \)
  AND

  IF \( [(A_{7:4}) > 9] \lor [(C) = 1] \)
  THEN \( (A_{7:4}) \leftarrow (A_{7:4}) + 6 \)
DEC byte

Function: Decrement

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

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

Example: Register 0 contains 7FH (0111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence,

DEC @R0
DEC R0
DEC @R0

will leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

DEC A

Bytes: 1
Cycles: 1

Encoding: 0 0 0 1 0 1 0 0
Operation: DEC (A) ← (A) − 1

DEC Rn

Bytes: 1
Cycles: 1

Encoding: 0 0 0 1 1 r r r
Operation: DEC (Rn) ← (Rn) − 1

DEC direct

Bytes: 2
Cycles: 1

Encoding: 0 0 0 1 0 1 0 1 direct address
Operation: DEC (direct) ← (direct) − 1

DEC @Ri

Bytes: 1
Cycles: 1

Encoding: 0 0 0 1 0 1 1 i
Operation: DEC ((Rj)) ← ((Rj)) − 1
DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the Accumulator by the unsigned eight-bit integer in register B.

The Accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.

Exception: if B had originally contained 00H, the values returned in the Accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The Accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The instruction,

DIV AB

will leave 13 in the Accumulator (0DH or 00001101B) and the value 17 (11H or 00010001B) in B, since 251 = (13 x 18) + 17. Carry and OV will both be cleared.

Bytes: 1

Cycles: 4

Encoding: 1 0 0 0 0 1 0 0

Operation:

DIV

(A)_{15:8} \leftarrow (A)/(B)
(B)_{7:0}
DJNZ <byte>,<rel-addr>

**Function:** Decrement and Jump if Not Zero

**Description:** DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00H will underflow to 0FFH. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

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

**Example:** Internal RAM locations 40H, 50H, and 60H contain the values 01H, 70H, and 15H, respectively. The instruction sequence,

DJNZ 40H, LABEL_1  
DJNZ 50H, LABEL_2  
DJNZ 60H, LABEL_3

will cause a jump to the instruction at LABEL_2 with the values 00H, 6FH, and 15H in the three RAM locations. The first jump was not taken because the result was zero.

This instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence,

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

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

**DJNZ Rn,rel**

**Bytes:** 2

**Cycles:** 2

**Encoding:**

| 1 1 0 1 | 1 r r r | rel. address |

**Operation:**

DJNZ

\[(PC) \leftarrow (PC) + 2\]

\[(R_n) \leftarrow (R_n) - 1\]

IF \((R_n) > 0 \text{ or } (R_n) < 0\)

\[\text{THEN} \quad (PC) \leftarrow (PC) + \text{rel}\]

**DJNZ direct,rel**

**Bytes:** 3

**Cycles:** 2

**Encoding:**

| 1 1 0 1 | 0 1 0 1 | direct data | rel. address |

**Operation:**

DJNZ

\[(PC) \leftarrow (PC) + 2\]

\[(\text{direct}) \leftarrow (\text{direct}) - 1\]

IF \((\text{direct}) > 0 \text{ or } (\text{direct}) < 0\)

\[\text{THEN} \quad (PC) \leftarrow (PC) + \text{rel}\]
INC <byte>

**Function:** Increment

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

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

**Example:** Register 0 contains 7EH (01111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence,

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

will leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H.

**INC A**

- **Bytes:** 1
- **Cycles:** 1

**Encoding:**

```
 0 0 0 0 0 1 0 0
```

**Operation:**

```
INC (A) ← (A) + 1
```

**INC Rn**

- **Bytes:** 1
- **Cycles:** 1

**Encoding:**

```
 0 0 0 0 1 r r r
```

**Operation:**

```
INC (Rn) ← (Rn) + 1
```

**INC direct**

- **Bytes:** 2
- **Cycles:** 1

**Encoding:**

```
0 0 0 0 0 1 0 1
direct address
```

**Operation:**

```
INC (direct) ← (direct) + 1
```

**INC @Ri**

- **Bytes:** 1
- **Cycles:** 1

**Encoding:**

```
0 0 0 0 0 1 1 i
```

**Operation:**

```
INC ((Ri)) ← ((Ri)) + 1
```
INC DPTR

Function: Increment Data Pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo $2^{16}$) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H will increment the high-order byte (DPH). No flags are affected.

This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The instruction sequence,

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

will change DPH and DPL to 13H and 01H.

Bytes: 1
Cycles: 2

Encoding:

```
1 0 1 0 0 0 1 1
```

Operation:

```
INC DPTR ← (DPTR) + 1
```

JB bit,rel

Function: Jump if Bit set

Description: If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Example: The data present at input port 1 is 11001010B. The Accumulator holds 56 (01010110B). The instruction sequence,

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

will cause program execution to branch to the instruction at label LABEL2.

Bytes: 3
Cycles: 2

Encoding:

```
0 0 1 0 0 0 0 0
```

bit address

rel. address

Operation:

```
JB (PC) ← (PC) + 3
IF (bit) = 1
THEN (PC) ← (PC) + rel
```
### JBC bit,rel

**Function:** Jump if Bit is set and Clear bit

**Description:** If the indicated bit is a one, branch to the address indicated; otherwise proceed with the next instruction. *The bit will not be cleared if it is already a zero.* The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

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

**Example:** The Accumulator holds 56H (01010110B). The instruction sequence,

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

will cause program execution to continue at the instruction identified by the LABEL2, with the Accumulator modified to 52H (01010010B).

**Bytes:** 3  
**Cycles:** 2  
**Encoding:**

```
0 0 0 1 0 0 0 0
```

**Operation:**

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

### JC rel

**Function:** Jump if Carry is set

**Description:** If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

**Example:** The carry flag is cleared. The instruction sequence,

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

will set the carry and cause program execution to continue at the instruction identified by the label LABEL2.

**Bytes:** 2  
**Cycles:** 2  
**Encoding:**

```
0 1 0 0 0 0 0 0
```

**Operation:**

```
JC (PC) ← (PC) + 2
IF (C) = 1
THEN
(PC) ← (PC) + rel
```
**JMP @A+DPTR**

**Function:** Jump indirect

**Description:** Add the eight-bit unsigned contents of the Accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo $2^{16}$): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the Accumulator nor the Data Pointer is altered. No flags are affected.

**Example:** An even number from 0 to 6 is in the Accumulator. The following sequence of instructions will branch to one of four AJMP instructions in a jump table starting at JMP_TBL:

```
MOV DPTR,#JMP_TBL
JMP @A+DPTR
```

JMP_TBL: AJMP LABEL0
AJMP LABEL1
AJMP LABEL2
AJMP LABEL3

If the Accumulator equals 04H when starting this sequence, execution will jump to label LABEL2. Remember that AJMP is a two-byte instruction, so the jump instructions start at every other address.

**Bytes:** 1

**Cycles:** 2

**Encoding:**

```
0 1 1 1 0 0 1 1
```

**Operation:**

```
JMP (PC) ← (A) + (DPTR)
```

---

**JNB bit,rel**

**Function:** Jump if Bit Not set

**Description:** If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

**Example:** The data present at input port 1 is 11001010B. The Accumulator holds 56H (01010110B). The instruction sequence,

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

will cause program execution to continue at the instruction at label LABEL2.

**Bytes:** 3

**Cycles:** 2

**Encoding:**

```
0 0 1 1 0 0 0 0
```

**Operation:**

```
JNB (PC) ← (PC) + 3
IF (bit) = 0
THEN (PC) ← (PC) + rel
```
**JNC rel**

**Function:** Jump if Carry Not set

**Description:** If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice to point to the next instruction. The carry flag is not modified.

**Example:** The carry flag is set. The instruction sequence,

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

will clear the carry and cause program execution to continue at the instruction identified by the label LABEL2.

**Bytes:** 2  
**Cycles:** 2

**Encoding:**

```
0 1 0 1 0 0 0 0
```

**Operation:**

```
JNC
(PC) ← (PC) + 2
IF (C) = 0 THEN
(PC) ← (PC) + rel
```

---

**JNZ rel**

**Function:** Jump if Accumulator Not Zero

**Description:** If any bit of the Accumulator is a one, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

**Example:** The Accumulator originally holds 00H. The instruction sequence,

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

will set the Accumulator to 01H and continue at label LABEL2.

**Bytes:** 2  
**Cycles:** 2

**Encoding:**

```
0 1 1 1 0 0 0 0
```

**Operation:**

```
JNZ
(PC) ← (PC) + 2
IF A ≠ 0 THEN (PC) ← (PC) + rel
```
JZ rel

**Function:** Jump if Accumulator Zero

**Description:** If all bits of the Accumulator are zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. The Accumulator is not modified. No flags are affected.

**Example:** The Accumulator originally holds 01H. The instruction sequence,

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

will change the Accumulator to 00H and cause program execution to continue at the instruction identified by the label LABEL2.

**Bytes:** 2

**Cycles:** 2

**Encoding:**

```
0 1 1 0 0 0 0
```

**Operation:**

```
JZ
(PC) ← (PC) + 2
IF A = 0 THEN (PC) ← (PC) + rel
```

LCALL addr16

**Function:** Long Call

**Description:** LCALL calls a subroutine located at the indicated address. The instruction adds three to the program counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low byte first), incrementing the Stack Pointer by two. The high-order and low-order bytes of the PC are then loaded, respectively, with the second and third bytes of the LCALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 64k-byte program memory address space. No flags are affected.

**Example:** Initially the Stack Pointer equals 07H. The label "SUBRTN" is assigned to program memory location 1234H. After executing the instruction,

```
LCALL SUBRTN
```

at location 0123H, the Stack Pointer will contain 09H, internal RAM locations 08H and 09H will contain 26H and 01H, and the PC will contain 1235H.

**Bytes:** 3

**Cycles:** 2

**Encoding:**

```
0 0 0 1 0 0 1 0
```

**Operation:**

```
LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC7:0)
(SP) ← (SP) + 1
((SP)) ← (PC15:8)
(PC) ← addr15-addr8
addr7-addr0
```
### LJMP addr16

**Function:** Long Jump

**Description:** LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64k program memory address space. No flags are affected.

**Example:** The label "JMPADR" is assigned to the instruction at program memory location 1234H. The instruction, `LJMP JMPADR` at location 0123H will load the program counter with 1234H.

| Bytes: | 3 |
| Cycles: | 2 |

**Encoding:**

```
0 0 0 0 0 1 0
```

**Operation:**

```
LJMP (PC) ← addr15-0
```

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

**Function:** Move byte variable

**Description:** The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

**Example:** Internal RAM location 30H holds 40H. The value of RAM location 40H is 10H. The data present at input port 1 is 11001010B (0CAH). The instruction sequence,

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

leaves the value 30H in register 0, 40H in both the Accumulator and register 1, 10H in register B, and 0CAH (11001010B) both in RAM location 40H and output on port 2.

**Bytes:** 1

**Cycles:** 1

**Encoding:**

```
1 1 1 0 1 r r r
```

**Operation:**

```
MOV (A) ← (Rn)
```
*MOV  A,direct
  Bytes:  2
  Cycles: 1
  Encoding:  \[\begin{array}{c}
  1 \ 1 \ 1 \ 0 \\
  0 \ 1 \ 0 \ 1 \\
\end{array}\]
  Operation: MOV  (A) ← (direct)

MOV  A,@Ri
  Bytes:  1
  Cycles: 1
  Encoding:  \[\begin{array}{c}
  1 \ 1 \ 1 \ 0 \\
  0 \ 1 \ 1 \ i \\
\end{array}\]
  Operation: MOV  (A) ← ((Ri))

MOV  A,#data
  Bytes:  2
  Cycles: 1
  Encoding:  \[\begin{array}{c}
  0 \ 1 \ 1 \ 1 \\
  0 \ 1 \ 0 \ 0 \\
\end{array}\]
  Operation: MOV  (A) ← #data

MOV  Rn,A
  Bytes:  1
  Cycles: 1
  Encoding:  \[\begin{array}{c}
  1 \ 1 \ 1 \ 1 \\
  1 \ r \ r \ r \\
\end{array}\]
  Operation: MOV  (Rn) ← (A)

MOV  Rn,direct
  Bytes:  2
  Cycles: 2
  Encoding:  \[\begin{array}{c}
  1 \ 0 \ 1 \ 0 \\
  1 \ r \ r \ r \\
\end{array}\]
  Operation: MOV  (Rn) ← (direct)

MOV  Rn,#data
  Bytes:  2
  Cycles: 1
  Encoding:  \[\begin{array}{c}
  0 \ 1 \ 1 \ 1 \\
  1 \ r \ r \ r \\
\end{array}\]
  Operation: MOV  (Rn) ← #data

*MOV  A,ACC is not a valid instruction.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
<th>Bytes</th>
<th>Cycles</th>
<th>Encoding</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV direct, A</td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>1 1 1 1  0 1 0 1</td>
<td>direct address</td>
</tr>
<tr>
<td>MOV direct, Rn</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>1 0 0 0  1 r r r</td>
<td>direct address</td>
</tr>
<tr>
<td>MOV direct, direct</td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1 0 0 0  0 1 0 1</td>
<td>dir. addr. (src)</td>
</tr>
<tr>
<td>MOV direct, @Ri</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>1 0 0 0  0 1 1 i</td>
<td>direct address</td>
</tr>
<tr>
<td>MOV direct, #data</td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>0 1 1 1  0 1 0 1</td>
<td>direct address</td>
</tr>
<tr>
<td>MOV @Ri, A</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1 1 1 1  0 1 1 i</td>
<td>direct address</td>
</tr>
</tbody>
</table>

Operation: MOV (source) ← (destination)
MOV @Ri,direct
Bytes: 2
Cycles: 2
Encoding: \[0x0\] 101 0 011 i
Operation: MOV ((Ri)) ← (direct)

MOV @Ri,#data
Bytes: 2
Cycles: 1
Encoding: \[0x0\] 011 1 011 i
Operation: MOV ((Ri)) ← #data

MOV <dest-bit>,<src-bit>
Function: Move bit data
Description: The Boolean variable indicated by the second operand is copied into the location specified by the first operand. One of the operands must be the carry flag; the other may be any directly addressable bit. No other register or flag is affected.
Example: The carry flag is originally set. The data present at input Port 3 is 11000101B. The data previously written to output Port 1 is 35H (00110101B). The instruction sequence,
MOV P1.3,C
MOV C,P3.3
MOV P1.2,C
will leave the carry cleared and change Port 1 to 39H (00111001B).

MOV C,bit
Bytes: 2
Cycles: 1
Encoding: \[0x0\] 101 0 001 0
Operation: MOV (C) ← (bit)

MOV bit,C
Bytes: 2
Cycles: 2
Encoding: \[0x0\] 100 1 001 0
Operation: MOV (bit) ← (C)
MOV  DPTR,#data16

Function: Load Data Pointer with a 16-bit constant

Description: The Data Pointer is loaded with the 16-bit constant indicated. The 16-bit constant is loaded into the second and third bytes of the instruction. The second byte (DPH) is the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are affected.

This is the only instruction which moves 16 bits of data at once.

Example: The instruction,
MOV DPTR,#1234H
will load the value 1234H into the Data Pointer: DPH will hold 12H and DPL will hold 34H.

Bytes: 3
Cycles: 2
Encoding: 100 1 000 0 immed. data15-8  immed. data7-0

Operation:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DPH</td>
<td>DPL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>#data15-0</td>
<td>#data15-8</td>
<td>#data7-0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MOVC  A,@A+<base-reg>

Function: Move Code byte

Description: The MOVC instructions load the Accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit Accumulator contents and the contents of a sixteen-bit base register, which may be either the Data Pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the Accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the Accumulator. The following instructions will translate the value in the Accumulator to one of four values defined by the DB (define byte) directive:

REL_PC:
INC A
MOVC A,@A+PC
RET
DB 66H
DB 77H
DB 88H
DB 99H

If the subroutine is called with the Accumulator equal to 01H, it will return with 77H in the Accumulator. The INC A before the MOVC instruction is needed to “get around” the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the Accumulator instead.

MOVC  A,@A+DPTR

Bytes: 1
Cycles: 2
Encoding: 100 1 001 1

Operation:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) ← ((A) + (DPTR))
MOVC A, @A+PC

Bytes: 1
Cycles: 2

Encoding: 1 0 0 0 0 0 1 1

Operation:

```
(PC) ← (PC) + 1
(A) ← ((A) + (PC))
```

MOVC A, @DPTR

Bytes: 1
Cycles: 2

Encoding: 1 1 1 0 0 0 0 0

Operation:

```
(A) ← ((DPTR))
```

**MOVC**

MOVC is a 1-byte instruction that moves the contents of the Accumulator to an external memory location. It is used when the address of the external memory location is known and can be computed using the program counter (PC) and the Accumulator (A). The instruction uses the formula:

\[
\text{Operand} = \text{PC} + (A)
\]

**Example:**

An external 256 byte RAM using multiplexed address/data lines is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

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

copies the value 56H into both the Accumulator and external RAM location 12H.

**MOVC**

MOVC is a 1-byte instruction that moves the contents of the Accumulator to an external memory location. It is used when the address of the external memory location is known and can be computed using the program counter (PC) and the Accumulator (A). The instruction uses the formula:

\[
\text{Operand} = \text{PC} + (A)
\]

**Example:**

An external 256 byte RAM using multiplexed address/data lines is connected to the 8051 Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12H and 34H. Location 34H of the external RAM holds the value 56H. The instruction sequence,

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

copies the value 56H into both the Accumulator and external RAM location 12H.
MOVX @Ri,A
Bytes: 1
Cycles: 2
Encoding: 1111 0 0 1 i
Operation: MOVX ((Ri)) ← (A)

MOVX @DPTR,A
Bytes: 1
Cycles: 2
Encoding: 1111 0 0 0 0
Operation: MOVX ((DPTR)) ← (A)

MUL AB
Function: Multiply
Description: MUL AB multiplies the unsigned eight-bit integers in the Accumulator and register B. The low-order byte of the sixteen-bit product is left in the Accumulator, and the high-order byte in B. If the product is greater than 255 (0FFH) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
Example: Originally the Accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction,
MUL AB
will give the product 12,800 (3200H), so B is changed to 32H (00110010B) and the Accumulator is cleared. The overflow flag is set, carry is cleared.
Bytes: 1
Cycles: 4
Encoding: 1 0 1 0 0 1 0 0
Operation: MUL (A)7-0 ← (A) x (B)
(B)15-8
NOP

**Function:** No Operation

**Description:** Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

**Example:** It is desired to produce a low-going output pulse on bit 7 of Port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming are enabled) with the instruction sequence,

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

**Bytes:** 1

**Cycles:** 1

**Encoding:** 

```
0 0 0 0 0 0 0 0
```

**Operation:**

```
NOP
(PC) ← (PC) + 1
```

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

**Function:** Logical-OR for byte variables

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

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

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

**Example:** If the Accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the instruction,

```
ORL A, R0
```

will leave the Accumulator holding the value 0D7H (11010111B). When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the Accumulator at run-time. The instruction,

```
ORL P1,#00110010B
```

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

**ORL A,Rn**

**Bytes:** 1

**Cycles:** 1

**Encoding:** 

```
0 1 0 0 1 r r r
```

**Operation:**

```
ORL (A) ← (A) ∨ (Rn)
```
ORL A, direct
Bytes: 2
Cycles: 1
Encoding: 01000101
direct address
Operation: ORL (A) ← (A) ∨ (direct)

ORL A, @Ri
Bytes: 1
Cycles: 1
Encoding: 0100011i
Operation: ORL (A) ← (A) ∨ ((Ri))

ORL A, #data
Bytes: 2
Cycles: 1
Encoding: 01000100 immediate data
Operation: ORL (A) ← (A) ∨ #data

ORL direct, A
Bytes: 2
Cycles: 1
Encoding: 01000010 direct address
Operation: ORL (direct) ← (direct) ∨ (A)

ORL direct, #data
Bytes: 3
Cycles: 2
Encoding: 01000011 direct address immediate data
Operation: ORL (direct) ← (direct) ∨ #data
### ORL C,<src-bit>

**Function:** Logical-OR for bit variables

**Description:** Set the carry flag if the Boolean value is a logical 1; leave the carry in its current state otherwise. A slash (/) preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

**Example:** Set the carry flag if and only if P1.0 = 1, ACC.7 = 1, or OV = 0:

ORL C,P1.0 ;LOAD CARRY WITH INPUT PIN P10
ORL C,ACC.7 ;OR CARRY WITH THE ACC. BIT 7
ORL C,/OV ;OR CARRY WITH THE INVERSE OF OV.

### ORL C,bit

| Bytes:  | 2 |
| Cycles: | 2 |

**Encoding:**

```
0 1 1 1 0 0 1 0
```

**Operation:** ORL  
(C) ← (C) ∨ (bit)

### ORL C,/bit

| Bytes:  | 2 |
| Cycles: | 2 |

**Encoding:**

```
1 0 1 0 0 0 0
```

**Operation:** ORL  
(C) ← (C) ∨ (bit)
**POP direct**

**Function:** Pop from stack

**Description:** The contents of the internal RAM location addressed by the Stack Pointer is read, and the Stack Pointer is decremented by one. The value read is then transferred to the directly addressed byte indicated. No flags are affected.

**Example:** The Stack Pointer originally contains the value 32H, and internal RAM locations 30H through 32H contain the values 20H, 23H, and 01H, respectively. The instruction sequence,

| POP | DPH |
| POP | DPL |

will leave the Stack Pointer equal to the value 30H and the Data Pointer set to 0123H. At this point the instruction,

| POP | SP |

will leave the Stack Pointer set to 20H. Note that in this special case the Stack Pointer was decremented to 2FH before being loaded with the value popped (20H).

**Bytes:** 2

**Cycles:** 2

**Encoding:**

```
1 1 0 1 0 0 0 0
```

**Operation:**

\[
\text{POP (direct)} \leftarrow ((SP)) \\
\text{(SP)} \leftarrow (SP) - 1
\]

---

**PUSH direct**

**Function:** Push onto stack

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

**Example:** On entering an interrupt routine the Stack Pointer contains 09H. The Data Pointer holds the value 0123H. The instruction sequence,

| PUSH | DPL |
| PUSH | DPH |

will leave the Stack Pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

**Bytes:** 2

**Cycles:** 2

**Encoding:**

```
1 1 0 0 0 0 0 0
```

**Operation:**

\[
\text{PUSH (direct)} \leftarrow (SP) + 1 \\
\text{((SP))} \leftarrow \text{(direct)}
\]
**RET**

**Function:** Return from subroutine

**Description:** RET pops the high- and low-order bytes of the PC successively from the stack, decrementing the Stack Pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

**Example:** The Stack Pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction, RET

will leave the Stack Pointer equal to the value 09H. Program execution will continue at location 0123H.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>2</td>
</tr>
</tbody>
</table>

**Encoding:**

```
0 0 1 0 0 0 1 0
```

**Operation:**

```
RET
(PC[15:8]) ← ((SP))
(SP) ← (SP) – 1
(PC[7:0]) ← ((SP))
(SP) ← (SP) – 1
```

---

**RETI**

**Function:** Return from interrupt

**Description:** RETI pops the high- and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The Stack Pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower- or same-level interrupt has been pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.

**Example:** The Stack Pointer originally contains the value 0BH. An interrupt was detected during the instruction ending at location 0122H. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction, RETI

will leave the Stack Pointer equal to 09H and return program execution to location 0123H.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>2</td>
</tr>
</tbody>
</table>

**Encoding:**

```
0 0 1 1 0 0 1 0
```

**Operation:**

```
RETI
(PC[15:8]) ← ((SP))
(SP) ← (SP) – 1
(PC[7:0]) ← ((SP))
(SP) ← (SP) – 1
```
RL A

Function: Rotate Accumulator Left
Description: The eight bits in the Accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B). The instruction,
RL A
leaves the Accumulator holding the value 8BH (10001011B) with the carry unaffected.
Bytes: 1
Cycles: 1
Encoding: 0 0 1 0 0 0 1 1
Operation: RL
(A_{n+1}) ← (A_n), n = 0 – 6
(A0) ← (A7)

RLC A

Function: Rotate Accumulator Left through the Carry flag
Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,
RLC A
leaves the Accumulator holding the value 8AH (10001010B) with the carry set.
Bytes: 1
Cycles: 1
Encoding: 0 0 1 1 0 0 1 1
Operation: RLC
(A_{n+1}) ← (A_n), n = 0 – 6
(A0) ← (C)
(C) ← (A7)
RR A

Function: Rotate Accumulator Right
Description: The eight bits in the Accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B). The instruction,
RR A
leaves the Accumulator holding the value 0E2H (11100010B) with the carry unaffected.
Bytes: 1
Cycles: 1
Encoding: 0 0 0 0 0 0 1 1
Operation: RR
(A_\text{n}) \leftarrow (A_{\text{n+1}}), \text{n} = 0 \text{ to } 6
(A_7) \leftarrow (A_0)

RRC A

Function: Rotate Accumulator Right through the Carry flag
Description: The eight bits in the Accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original state of the carry flag moves into the bit 7 position. No other flags are affected.
Example: The Accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction,
RRC A
leaves the Accumulator holding the value 62 (01100010B) with the carry set.
Bytes: 1
Cycles: 1
Encoding: 0 0 0 1 0 0 1 1
Operation: RRC
(A_\text{n}) \leftarrow (A_{\text{n+1}}), \text{n} = 0 \text{ to } 6
(A_7) \leftarrow (C)
(C) \leftarrow (A_0)
SETB <bit>

Function: Set Bit
Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.
Example: The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The instructions,
        SETB C
        SETB P1.0
    will leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B).

SETB C
Bytes: 1
Cycles: 1
Encoding: 1 1 0 1 0 0 1 1
Operation: SETB (C) ← 1

SETB bit
Bytes: 2
Cycles: 1
Encoding: 1 1 0 1 0 0 1 0
Operation: SETB (bit) ← 1

SJMP rel
Function: Short Jump
Description: Program control branches unconditionally to the address indicated. The branch destination is computed by adding the signed displacement in the second instruction byte to the PC, after incrementing the PC twice. Therefore, the range of destinations allowed is from 128 bytes preceding this instruction to 127 bytes following it.
Example: The label "RELADR" is assigned to an instruction at program memory location 0123H. The instruction,
        SJMP RELADR
    will assemble into location 0100H. After the instruction is executed, the PC will contain the value 0123H.
    (Note: Under the above conditions the instruction following SJMP will be at 102H. Therefore, the displacement byte of the instruction will be the relative offset (0123H-0102H) = 21H. Put another way, an SJMP with a displacement of 0FEH would be a one-instruction infinite loop.)
Bytes: 2
Cycles: 2
Encoding: 1 0 0 0 0 0 0 0
Operation: SJMP (PC) ← (PC) + 2
            (PC) ← (PC) + rel
SUBB A, <src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the Accumulator, leaving the result in the Accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the Accumulator along with the source operand.) AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6, but not into bit 7, or into bit 7, but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The Accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction,

```
SUBB A,R2
```

will leave the value 74H (01110100B) in the Accumulator, with the carry flag and AC cleared but OV set.

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the carry (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction

SUBB A,Rn

Bytes: 1
Cycles: 1

Encoding: 1 0 0 1 1 r r r

Operation: SUBB

(A) ← (A) – (C) – (R_n)

SUBB A,direct

Bytes: 2
Cycles: 1

Encoding: 1 0 0 1 0 1 0 1

Operation: SUBB

(A) ← (A) – (C) – (direct)

SUBB A,@Ri

Bytes: 1
Cycles: 1

Encoding: 1 0 0 1 0 1 1 i

Operation: SUBB

(A) ← (A) – (C) – (R_i)

SUBB A,#data

Bytes: 2
Cycles: 1

Encoding: 1 0 0 1 0 1 0 0

Operation: SUBB

(A) ← (A) – (C) – (#data)
### SWAP A

**Function:** Swap nibbles within the Accumulator

**Description:** SWAP A interchanges the low- and high-order nibbles (four-bit fields) of the Accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.

**Example:** The Accumulator holds the value 0C5H (11000101B). The instruction,

```
SWAP A
```

leaves the Accumulator holding the value 5CH (01011100B).

**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 11000100  
**Operation:** SWAP \((A_{3-0}) \leftrightarrow (A_{7-4})\)

### XCH A,<byte>

**Function:** Exchange Accumulator with byte variable

**Description:** XCH loads the Accumulator with the contents of the indicated variable, at the same time writing the original Accumulator contents to the indicated variable. The source/destination operand can use register, direct, or register-indirect addressing.

**Example:** R0 contains the address 20H. The Accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

```
XCH A,@R0
```

will leave the RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the Accumulator.

### XCH A,Rn

**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 11001rrr  
**Operation:** XCH \((A) \leftrightarrow (R_n)\)

### XCH A,direct

**Bytes:** 2  
**Cycles:** 1  
**Encoding:** 11000101 direct address  
**Operation:** XCH \((A) \leftrightarrow \text{direct}\)

### XCH A,@Ri

**Bytes:** 1  
**Cycles:** 1  
**Encoding:** 11000111  
**Operation:** XCH \((A) \leftrightarrow (R_i)\)
XCHD  A, Ri

Function: Exchange Digit

Description: XCHD exchanges the low-order nibble of the Accumulator (bits 3-0), generally representing a hexadecimal or BCD digit, with that of the internal RAM location indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of each register are not affected. No flags are affected.

Example: R0 contains the address 20H. The Accumulator holds the value 36H (00110110B). Internal RAM location 20H holds the value 75H (01110101B). The instruction,

XCHD  A, @R0

will leave RAM location 20H holding the value 76H (01110110B) and 35H (00110101B) in the Accumulator.

Bytes: 1
Cycles: 1
Encoding: 1 1 0 1 0 1 1 i
Operation:

XCHD (A_{3:0}) \leftrightarrow ((Ri_{3:0}))

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

Function: Logical Exclusive-OR for byte variables

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

The two operands allow six addressing mode combinations. When the destination is the Accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the Accumulator or immediate data.

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

Example: If the Accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction,

XRL  A, R0

will leave the Accumulator holding the value 69H (01101001B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the Accumulator at run-time. The instruction,

XRL  P1, #00110001B

will complement bits 5, 4, and 0 of output Port 1.
### XRL A,Rn

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (A) ← (A) ⊕ (Rn)</td>
<td>0110 1 0 r r</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### XRL A,direct

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (A) ← (A) ⊕ (direct)</td>
<td>0110 010 1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### XRL A,@Ri

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (A) ← (A) ⊕ (Ri)</td>
<td>0110 011 0 i</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### XRL A,#data

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (A) ← (A) ⊕ #data</td>
<td>0110 010 0 0</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

### XRL direct,A

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (direct) ← (direct) ⊕ (A)</td>
<td>0110 001 0 0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

### XRL direct,#data

<table>
<thead>
<tr>
<th>Operation</th>
<th>Encoding</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL (direct) ← (direct) ⊕ #data</td>
<td>0110 001 0 1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>