## MCS®-51 INSTRUCTION SET

### Table 10.8051 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 X X</td>
<td>ADDC</td>
<td>X X X</td>
</tr>
<tr>
<td>SUBB</td>
<td>X X X</td>
<td>MUL</td>
<td>O X</td>
</tr>
<tr>
<td>DIV</td>
<td>O X</td>
<td>DIV</td>
<td>X O</td>
</tr>
<tr>
<td>DA</td>
<td>X O</td>
<td>RRC</td>
<td>X</td>
</tr>
<tr>
<td>RLC</td>
<td>X</td>
<td>CJNE</td>
<td>X</td>
</tr>
<tr>
<td>SETB C</td>
<td>1</td>
<td></td>
<td></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.

### Mnemonic Operations

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Oscillator Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,Rn</td>
<td>Add register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ADD A,direct</td>
<td>Add direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ADD A,@Ri</td>
<td>Add indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ADD A,#data</td>
<td>Add immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ADDC A,Rn</td>
<td>Add register to Accumulator with Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ADDC A,direct</td>
<td>Add direct byte to Accumulator with Carry</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ADDC A,@Ri</td>
<td>Add indirect RAM to Accumulator with Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ADDC A,#data</td>
<td>Add immediate data to Acc with borrow</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>SUBB A,Rn</td>
<td>Subtract Register from Acc with borrow</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SUBB A,direct</td>
<td>Subtract direct byte from Acc with borrow</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>SUBB A,@Ri</td>
<td>Subtract indirect RAM from ACC with borrow</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SUBB A,#data</td>
<td>Subtract immediate data from Acc with borrow</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>INC A</td>
<td>Increment Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>INC Rn</td>
<td>Increment register</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>INC direct</td>
<td>Increment direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>INC @Ri</td>
<td>Increment direct RAM</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC A</td>
<td>Decrement Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC Rn</td>
<td>Decrement Register</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>DEC direct</td>
<td>Decrement direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>DEC @Ri</td>
<td>Decrement indirect RAM</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted ©Intel Corporation 1980.
### Table 10. 8051 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 (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC DPTR</td>
<td>Increment Data Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL AB</td>
<td>Multiply A &amp; B</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>DIV AB</td>
<td>Divide A by B</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>DA A</td>
<td>Decimal Adjust</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 A,Rn</td>
<td>AND Register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ANL A,direct</td>
<td>AND direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL A,#data</td>
<td>AND indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ANL A, # data</td>
<td>AND immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL direct A</td>
<td>AND Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL direct, # data</td>
<td>AND immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>ORL A,Rn</td>
<td>OR register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ORL A,direct</td>
<td>OR direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL A,#data</td>
<td>OR indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>ORL A, # data</td>
<td>OR immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL direct A</td>
<td>OR Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ORL direct, # data</td>
<td>OR immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>XRL A,Rn</td>
<td>Exclusive-OR register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XRL A,direct</td>
<td>Exclusive-OR direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL A,#data</td>
<td>Exclusive-OR indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XRL A, # data</td>
<td>Exclusive-OR immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL direct A</td>
<td>Exclusive-OR direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XRL direct, # data</td>
<td>Exclusive-OR immediate data to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>CLR A</td>
<td>Clear Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>CPL A</td>
<td>Complement Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Oscillator Period</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOGICAL OPERATIONS (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RL A</td>
<td>Rotate Accumulator Left</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RLC A</td>
<td>Rotate Accumulator Left through the Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RR A</td>
<td>Rotate Accumulator Right</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>RRC A</td>
<td>Rotate Accumulator Right through the Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SWAP A</td>
<td>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 A,Rn</td>
<td>Move register to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOV A,direct</td>
<td>Move direct byte to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV A,@Ri</td>
<td>Move indirect RAM to Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOV A, # data</td>
<td>Move immediate data to Accumulator</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV Rn,A</td>
<td>Move Accumulator to register</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOV Rn,direct</td>
<td>Move direct byte to register</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>MOV Rn, # data</td>
<td>Move immediate data to register</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV direct,A</td>
<td>Move Accumulator to direct byte</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV direct,Rn</td>
<td>Move register to direct byte</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>MOV direct,direct</td>
<td>Move direct byte to direct byte</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>MOV direct,#data</td>
<td>Move immediate data to direct byte</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>MOV @Ri,A</td>
<td>Move Accumulator to indirect RAM</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted ©Intel Corporation 1980
### Table 10.8051 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 (Continued)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV @Ri,direct</td>
<td>Move direct byte to indirect RAM</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV @Ri,#data</td>
<td>Move immediate data to indirect RAM</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV DPTR,#data16</td>
<td>Load Data Pointer with a 16-bit constant</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>MOVCA,A+DPTR</td>
<td>Move Code byte relative to DPT to Acc</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOVCA,A+PC</td>
<td>Move Code byte relative to PC to Acc</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>MOVX A,@Ri</td>
<td>Move External RAM (8-bit addr) to Acc</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>MOVX A,#DPTR</td>
<td>Move External RAM (16-bit addr) to Acc</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>MOVX @Ri,A</td>
<td>Move Acc to External RAM (8-bit_addr)</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>MOVX @DPTR,A</td>
<td>Move Acc to External RAM (16-bit_addr)</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>PUSH direct</td>
<td>Push direct byte onto stack</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>POP direct</td>
<td>Pop direct byte from stack</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>XCH A,Rn</td>
<td>Exchange register with Accumulator</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XCH A,direct</td>
<td>Exchange direct byte with</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>XCH A,@Ri</td>
<td>Exchange Accumulator with</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>XCHD A,@Ri</td>
<td>Exchange low-order Digit indirect RAM with</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

### BOOLEAN VARIABLE MANIPULATION

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Oscillator Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>Clear Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SETB</td>
<td>Set Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>SETB bit</td>
<td>Set direct bit</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>CPL</td>
<td>Complement Carry</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>CPL bit</td>
<td>Complement direct</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>ANL</td>
<td>AND direct bit to CARRY</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ORL</td>
<td>OR direct bit to Carry</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>ORL bit</td>
<td>OR complement direct bit to Carry</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>MOV C,bit</td>
<td>Move direct bit to Carry</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MOV bit,C</td>
<td>Move Carry to direct bit</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JC rel</td>
<td>Jump if Carry is set</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JNC rel</td>
<td>Jump if Carry not set</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>JB bit,rel</td>
<td>Jump if direct Bit is set</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>JNB bit,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 &amp; clear bit</td>
<td>3</td>
<td>24</td>
</tr>
</tbody>
</table>

### PROGRAM BRANCHING

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Oscillator Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACALL</td>
<td>Absolute Subroutine Call</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>LCALL</td>
<td>Long Subroutine Call</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td>RET</td>
<td>Return from Subroutine Call</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</td>
<td>Absolute Jump</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>LJMP</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>
</tbody>
</table>

All mnemonics copyrighted ©Intel Corporation 1980
## Table 10.8051 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>PROGRAM BRANCHING</strong> (Continued)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>JMP</strong> @A+DPTR</td>
<td>Jump indirect relative to the DPTR</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td><strong>JZ</strong> rel</td>
<td>Jump if Accumulator is Zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td><strong>JNZ</strong> rel</td>
<td>Jump if Accumulator is Not Zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td><strong>CJNE A,direct,rel</strong></td>
<td>Compare direct byte to Acc and Jump if Not Equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>CJNE A,#data,rel</strong></td>
<td>Compare immediate to Acc and Jump if Not Equal</td>
<td>3</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Byte</th>
<th>Oscillator Period</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROGRAM BRANCHING</strong> (Continued)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CJNE Rn,#data,rel</strong></td>
<td>Compare immediate to register and Jump if Not Equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>CJNE @Ri,#data,rel</strong></td>
<td>Compare immediate to indirect and Jump if Not Equal</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>DJNZ Rn,rel</strong></td>
<td>Decrement register and Jump if Not Zero</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td><strong>DJNZ direct,rel</strong></td>
<td>Decrement direct byte and Jump if Not Zero</td>
<td>3</td>
<td>24</td>
</tr>
<tr>
<td><strong>NOP</strong></td>
<td>No Operation</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

All mnemonics copyrighted ©Intel Corporation 1980
### Table 11. Instruction Opcodes in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>NOP</td>
<td></td>
<td>39</td>
<td>1</td>
<td>RLC</td>
<td>A</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td>34</td>
<td>2</td>
<td>ADDC</td>
<td>A, #data</td>
</tr>
<tr>
<td>02</td>
<td>3</td>
<td>LJMP</td>
<td>code addr</td>
<td>35</td>
<td>2</td>
<td>ADDC</td>
<td>A, data addr</td>
</tr>
<tr>
<td>03</td>
<td>1</td>
<td>RR</td>
<td>A</td>
<td>36</td>
<td>1</td>
<td>ADDC</td>
<td>A, @R0</td>
</tr>
<tr>
<td>04</td>
<td>1</td>
<td>INC</td>
<td>A</td>
<td>37</td>
<td>1</td>
<td>ADDC</td>
<td>A, @R1</td>
</tr>
<tr>
<td>05</td>
<td>2</td>
<td>INC</td>
<td>data addr</td>
<td>38</td>
<td>1</td>
<td>ADDC</td>
<td>A, R0</td>
</tr>
<tr>
<td>06</td>
<td>1</td>
<td>INC</td>
<td>@R0</td>
<td>39</td>
<td>1</td>
<td>ADDC</td>
<td>A, R1</td>
</tr>
<tr>
<td>07</td>
<td>1</td>
<td>INC</td>
<td>@R1</td>
<td>3A</td>
<td>1</td>
<td>ADDC</td>
<td>A, R2</td>
</tr>
<tr>
<td>08</td>
<td>1</td>
<td>INC</td>
<td>R0</td>
<td>3B</td>
<td>1</td>
<td>ADDC</td>
<td>A, R3</td>
</tr>
<tr>
<td>09</td>
<td>1</td>
<td>INC</td>
<td>R1</td>
<td>3C</td>
<td>1</td>
<td>ADDC</td>
<td>A, R4</td>
</tr>
<tr>
<td>0A</td>
<td>1</td>
<td>INC</td>
<td>R2</td>
<td>3D</td>
<td>1</td>
<td>ADDC</td>
<td>A, R5</td>
</tr>
<tr>
<td>0B</td>
<td>1</td>
<td>INC</td>
<td>R3</td>
<td>3E</td>
<td>1</td>
<td>ADDC</td>
<td>A, R6</td>
</tr>
<tr>
<td>0C</td>
<td>1</td>
<td>INC</td>
<td>R4</td>
<td>3F</td>
<td>1</td>
<td>ADDC</td>
<td>A, R7</td>
</tr>
<tr>
<td>0D</td>
<td>1</td>
<td>INC</td>
<td>R5</td>
<td>40</td>
<td>2</td>
<td>JC</td>
<td>code addr</td>
</tr>
<tr>
<td>0E</td>
<td>1</td>
<td>INC</td>
<td>R6</td>
<td>41</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>0F</td>
<td>1</td>
<td>INC</td>
<td>R7</td>
<td>42</td>
<td>2</td>
<td>ORL</td>
<td>data addr, A</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>JBC</td>
<td>bit addr, code addr</td>
<td>43</td>
<td>3</td>
<td>ORL</td>
<td>data addr, # data</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td>44</td>
<td>2</td>
<td>ORL</td>
<td>A, # data</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>LCALL</td>
<td>code addr</td>
<td>45</td>
<td>2</td>
<td>ORL</td>
<td>A, data addr</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>RRC</td>
<td>A</td>
<td>46</td>
<td>1</td>
<td>ORL</td>
<td>A, @R0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>DEC</td>
<td>A</td>
<td>47</td>
<td>1</td>
<td>ORL</td>
<td>A, @R1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>DEC</td>
<td>data addr</td>
<td>48</td>
<td>1</td>
<td>ORL</td>
<td>A, R0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>DEC</td>
<td>@R0</td>
<td>49</td>
<td>1</td>
<td>ORL</td>
<td>A, R1</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>DEC</td>
<td>@R1</td>
<td>4A</td>
<td>1</td>
<td>ORL</td>
<td>A, R2</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>DEC</td>
<td>R0</td>
<td>4B</td>
<td>1</td>
<td>ORL</td>
<td>A, R3</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>DEC</td>
<td>R1</td>
<td>4C</td>
<td>1</td>
<td>ORL</td>
<td>A, R4</td>
</tr>
<tr>
<td>1A</td>
<td>1</td>
<td>DEC</td>
<td>R2</td>
<td>4D</td>
<td>1</td>
<td>ORL</td>
<td>A, R5</td>
</tr>
<tr>
<td>1B</td>
<td>1</td>
<td>DEC</td>
<td>R3</td>
<td>4E</td>
<td>1</td>
<td>ORL</td>
<td>A, R6</td>
</tr>
<tr>
<td>1C</td>
<td>1</td>
<td>DEC</td>
<td>R4</td>
<td>4F</td>
<td>1</td>
<td>ORL</td>
<td>A, R7</td>
</tr>
<tr>
<td>1D</td>
<td>1</td>
<td>DEC</td>
<td>R5</td>
<td>50</td>
<td>2</td>
<td>JNC</td>
<td>code addr</td>
</tr>
<tr>
<td>1E</td>
<td>1</td>
<td>DEC</td>
<td>R6</td>
<td>51</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>1F</td>
<td>1</td>
<td>DEC</td>
<td>R7</td>
<td>52</td>
<td>2</td>
<td>ANL</td>
<td>data addr, A</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>JB</td>
<td>bit addr, code addr</td>
<td>53</td>
<td>3</td>
<td>ANL</td>
<td>data addr, # data</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
<td>54</td>
<td>2</td>
<td>ANL</td>
<td>A, # data</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>RET</td>
<td>A</td>
<td>55</td>
<td>2</td>
<td>ANL</td>
<td>A, data addr</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>RL</td>
<td>A</td>
<td>56</td>
<td>1</td>
<td>ANL</td>
<td>A, @R0</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>ADD</td>
<td>A, # data</td>
<td>57</td>
<td>1</td>
<td>ANL</td>
<td>A, @R1</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>ADD</td>
<td>A, data addr</td>
<td>58</td>
<td>1</td>
<td>ANL</td>
<td>A, R0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>ADD</td>
<td>A, @R0</td>
<td>59</td>
<td>1</td>
<td>ANL</td>
<td>A, R1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>ADD</td>
<td>A, @R1</td>
<td>5A</td>
<td>1</td>
<td>ANL</td>
<td>A, R2</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>ADD</td>
<td>A, R0</td>
<td>5B</td>
<td>1</td>
<td>ANL</td>
<td>A, R3</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>ADD</td>
<td>A, R1</td>
<td>5C</td>
<td>1</td>
<td>ANL</td>
<td>A, R4</td>
</tr>
<tr>
<td>2A</td>
<td>1</td>
<td>ADD</td>
<td>A, R2</td>
<td>5D</td>
<td>1</td>
<td>ANL</td>
<td>A, R5</td>
</tr>
<tr>
<td>2B</td>
<td>1</td>
<td>ADD</td>
<td>A, R3</td>
<td>5E</td>
<td>1</td>
<td>ANL</td>
<td>A, R6</td>
</tr>
<tr>
<td>2C</td>
<td>1</td>
<td>ADD</td>
<td>A, R4</td>
<td>5F</td>
<td>1</td>
<td>ANL</td>
<td>A, R7</td>
</tr>
<tr>
<td>2D</td>
<td>1</td>
<td>ADD</td>
<td>A, R5</td>
<td>60</td>
<td>2</td>
<td>JZ</td>
<td>code addr</td>
</tr>
<tr>
<td>2E</td>
<td>1</td>
<td>ADD</td>
<td>A, R6</td>
<td>61</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>2F</td>
<td>1</td>
<td>ADD</td>
<td>A, R7</td>
<td>62</td>
<td>2</td>
<td>XRL</td>
<td>data addr, A</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>JNB</td>
<td>bit addr, code addr</td>
<td>63</td>
<td>3</td>
<td>XRL</td>
<td>data addr, # data</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
<td>64</td>
<td>2</td>
<td>XRL</td>
<td>A, # data</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>RETI</td>
<td>code addr</td>
<td>65</td>
<td>2</td>
<td>XRL</td>
<td>A, data addr</td>
</tr>
</tbody>
</table>
Table 11. Instruction Opcodes in Hexadecimal Order (Continued)

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>1</td>
<td>XRL</td>
<td>A,@RO</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>XRL</td>
<td>A,@R1</td>
</tr>
<tr>
<td>68</td>
<td>1</td>
<td>XRL</td>
<td>A,R0</td>
</tr>
<tr>
<td>69</td>
<td>1</td>
<td>XRL</td>
<td>A,R1</td>
</tr>
<tr>
<td>6A</td>
<td>1</td>
<td>XRL</td>
<td>A,R2</td>
</tr>
<tr>
<td>6B</td>
<td>1</td>
<td>XRL</td>
<td>A,R3</td>
</tr>
<tr>
<td>6C</td>
<td>1</td>
<td>XRL</td>
<td>A,R4</td>
</tr>
<tr>
<td>6D</td>
<td>1</td>
<td>XRL</td>
<td>A,R5</td>
</tr>
<tr>
<td>6E</td>
<td>1</td>
<td>XRL</td>
<td>A,R6</td>
</tr>
<tr>
<td>6F</td>
<td>1</td>
<td>XRL</td>
<td>A,R7</td>
</tr>
<tr>
<td>70</td>
<td>2</td>
<td>JNZ</td>
<td>code addr</td>
</tr>
<tr>
<td>71</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>72</td>
<td>2</td>
<td>ORL</td>
<td>C,bit addr</td>
</tr>
<tr>
<td>73</td>
<td>1</td>
<td>JMP</td>
<td>@A + DPTR</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>MOV</td>
<td>A,#data</td>
</tr>
<tr>
<td>75</td>
<td>3</td>
<td>MOV</td>
<td>data addr,#data</td>
</tr>
<tr>
<td>76</td>
<td>2</td>
<td>MOV</td>
<td>@R0,#data</td>
</tr>
<tr>
<td>77</td>
<td>2</td>
<td>MOV</td>
<td>@R1,#data</td>
</tr>
<tr>
<td>78</td>
<td>2</td>
<td>MOV</td>
<td>R0,#data</td>
</tr>
<tr>
<td>79</td>
<td>2</td>
<td>MOV</td>
<td>R1,#data</td>
</tr>
<tr>
<td>7A</td>
<td>2</td>
<td>MOV</td>
<td>R2,#data</td>
</tr>
<tr>
<td>7B</td>
<td>2</td>
<td>MOV</td>
<td>R3,#data</td>
</tr>
<tr>
<td>7C</td>
<td>2</td>
<td>MOV</td>
<td>R4,#data</td>
</tr>
<tr>
<td>7D</td>
<td>2</td>
<td>MOV</td>
<td>R5,#data</td>
</tr>
<tr>
<td>7E</td>
<td>2</td>
<td>MOV</td>
<td>R6,#data</td>
</tr>
<tr>
<td>7F</td>
<td>2</td>
<td>MOV</td>
<td>R7,#data</td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>SJPMP</td>
<td>code addr</td>
</tr>
<tr>
<td>81</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>82</td>
<td>2</td>
<td>ANL</td>
<td>C,bit addr</td>
</tr>
<tr>
<td>83</td>
<td>1</td>
<td>MOVCA</td>
<td>A,@A + PC</td>
</tr>
<tr>
<td>84</td>
<td>1</td>
<td>DMOV</td>
<td>A5</td>
</tr>
<tr>
<td>85</td>
<td>3</td>
<td>MOV</td>
<td>data addr,#data addr</td>
</tr>
<tr>
<td>86</td>
<td>2</td>
<td>MOV</td>
<td>data addr,@R0</td>
</tr>
<tr>
<td>87</td>
<td>2</td>
<td>MOV</td>
<td>data addr,@R1</td>
</tr>
<tr>
<td>88</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R0</td>
</tr>
<tr>
<td>89</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R1</td>
</tr>
<tr>
<td>8A</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R2</td>
</tr>
<tr>
<td>8B</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R3</td>
</tr>
<tr>
<td>8C</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R4</td>
</tr>
<tr>
<td>8D</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R5</td>
</tr>
<tr>
<td>8E</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R6</td>
</tr>
<tr>
<td>8F</td>
<td>2</td>
<td>MOV</td>
<td>data addr,R7</td>
</tr>
<tr>
<td>90</td>
<td>3</td>
<td>MOVCA</td>
<td>DPTR,#data</td>
</tr>
<tr>
<td>91</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>92</td>
<td>2</td>
<td>MOV</td>
<td>bit addr,C</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
<td>MOVCA</td>
<td>A,#A + DPTR</td>
</tr>
<tr>
<td>94</td>
<td>2</td>
<td>SUBB</td>
<td>A,#data</td>
</tr>
<tr>
<td>95</td>
<td>2</td>
<td>SUBB</td>
<td>A,data addr</td>
</tr>
<tr>
<td>96</td>
<td>1</td>
<td>SUBB</td>
<td>A,@RO</td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>SUBB</td>
<td>A,@R0</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>SUBB</td>
<td>A,R0</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>SUBB</td>
<td>A,R1</td>
</tr>
<tr>
<td>9A</td>
<td>1</td>
<td>SUBB</td>
<td>A,R2</td>
</tr>
<tr>
<td>9B</td>
<td>1</td>
<td>SUBB</td>
<td>A,R3</td>
</tr>
<tr>
<td>9C</td>
<td>1</td>
<td>SUBB</td>
<td>A,R4</td>
</tr>
<tr>
<td>9D</td>
<td>1</td>
<td>SUBB</td>
<td>A,R5</td>
</tr>
<tr>
<td>9E</td>
<td>1</td>
<td>SUBB</td>
<td>A,R6</td>
</tr>
<tr>
<td>9F</td>
<td>1</td>
<td>SUBB</td>
<td>A,R7</td>
</tr>
<tr>
<td>A0</td>
<td>2</td>
<td>ORL</td>
<td>C,bit addr</td>
</tr>
<tr>
<td>A1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>A2</td>
<td>2</td>
<td>MOV</td>
<td>C,bit addr</td>
</tr>
<tr>
<td>A3</td>
<td>1</td>
<td>INC</td>
<td>DPTR</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
<td>MUL</td>
<td>AB</td>
</tr>
<tr>
<td>A5</td>
<td>reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>2</td>
<td>MOV</td>
<td>@R0,data addr</td>
</tr>
<tr>
<td>A7</td>
<td>2</td>
<td>MOV</td>
<td>@R1,data addr</td>
</tr>
<tr>
<td>A8</td>
<td>2</td>
<td>MOV</td>
<td>R0,data addr</td>
</tr>
<tr>
<td>A9</td>
<td>2</td>
<td>MOV</td>
<td>R1,data addr</td>
</tr>
<tr>
<td>AA</td>
<td>2</td>
<td>MOV</td>
<td>R2,data addr</td>
</tr>
<tr>
<td>AB</td>
<td>2</td>
<td>MOV</td>
<td>R3,data addr</td>
</tr>
<tr>
<td>AC</td>
<td>2</td>
<td>MOV</td>
<td>R4,data addr</td>
</tr>
<tr>
<td>AD</td>
<td>2</td>
<td>MOV</td>
<td>R5,data addr</td>
</tr>
<tr>
<td>AE</td>
<td>2</td>
<td>MOV</td>
<td>R6,data addr</td>
</tr>
<tr>
<td>AF</td>
<td>2</td>
<td>MOV</td>
<td>R7,data addr</td>
</tr>
<tr>
<td>B0</td>
<td>2</td>
<td>ANL</td>
<td>C,bit addr</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
<td>CPL</td>
<td>bit addr</td>
</tr>
<tr>
<td>B3</td>
<td>1</td>
<td>CPL</td>
<td>C</td>
</tr>
<tr>
<td>B4</td>
<td>3</td>
<td>CJNE</td>
<td>A,#data,code addr</td>
</tr>
<tr>
<td>B5</td>
<td>3</td>
<td>CJNE</td>
<td>A,data addr,code addr</td>
</tr>
<tr>
<td>B6</td>
<td>3</td>
<td>CJNE</td>
<td>@R0,#data,code addr</td>
</tr>
<tr>
<td>B7</td>
<td>3</td>
<td>CJNE</td>
<td>@R1,#data,code addr</td>
</tr>
<tr>
<td>B8</td>
<td>3</td>
<td>CJNE</td>
<td>R0,#data,code addr</td>
</tr>
<tr>
<td>B9</td>
<td>3</td>
<td>CJNE</td>
<td>R1,#data,code addr</td>
</tr>
<tr>
<td>BA</td>
<td>3</td>
<td>CJNE</td>
<td>R2,#data,code addr</td>
</tr>
<tr>
<td>BB</td>
<td>3</td>
<td>CJNE</td>
<td>R3,#data,code addr</td>
</tr>
<tr>
<td>BC</td>
<td>3</td>
<td>CJNE</td>
<td>R4,#data,code addr</td>
</tr>
<tr>
<td>BD</td>
<td>3</td>
<td>CJNE</td>
<td>R5,#data,code addr</td>
</tr>
<tr>
<td>BE</td>
<td>3</td>
<td>CJNE</td>
<td>R6,#data,code addr</td>
</tr>
<tr>
<td>BF</td>
<td>3</td>
<td>CJNE</td>
<td>R7,#data,code addr</td>
</tr>
<tr>
<td>C0</td>
<td>2</td>
<td>PUSH</td>
<td>data addr</td>
</tr>
<tr>
<td>C1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>C2</td>
<td>2</td>
<td>CLR</td>
<td>bit addr</td>
</tr>
<tr>
<td>C3</td>
<td>1</td>
<td>CLR</td>
<td>C</td>
</tr>
<tr>
<td>C4</td>
<td>1</td>
<td>SWAP</td>
<td>A</td>
</tr>
<tr>
<td>C5</td>
<td>2</td>
<td>XCH</td>
<td>A,data addr</td>
</tr>
<tr>
<td>C6</td>
<td>1</td>
<td>XCH</td>
<td>A,@RO</td>
</tr>
<tr>
<td>C7</td>
<td>1</td>
<td>XCH</td>
<td>A,@R1</td>
</tr>
<tr>
<td>C8</td>
<td>1</td>
<td>XCH</td>
<td>A,R0</td>
</tr>
<tr>
<td>C9</td>
<td>1</td>
<td>XCH</td>
<td>A,R1</td>
</tr>
<tr>
<td>CA</td>
<td>1</td>
<td>XCH</td>
<td>A,R2</td>
</tr>
<tr>
<td>CB</td>
<td>1</td>
<td>XCH</td>
<td>A,R3</td>
</tr>
</tbody>
</table>

2-26
### Table 11. Instruction Opcodes in Hexadecimal Order (Continued)

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Number of Bytes</th>
<th>Mnemonic</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>1</td>
<td>XCH</td>
<td>A,R4</td>
</tr>
<tr>
<td>CD</td>
<td>1</td>
<td>XCH</td>
<td>A,R5</td>
</tr>
<tr>
<td>CE</td>
<td>1</td>
<td>XCH</td>
<td>A,R6</td>
</tr>
<tr>
<td>CF</td>
<td>1</td>
<td>XCH</td>
<td>A,R7</td>
</tr>
<tr>
<td>D0</td>
<td>2</td>
<td>POP</td>
<td>data addr</td>
</tr>
<tr>
<td>D1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>D2</td>
<td>2</td>
<td>SETB</td>
<td>bit addr</td>
</tr>
<tr>
<td>D3</td>
<td>1</td>
<td>SETB</td>
<td>C</td>
</tr>
<tr>
<td>D4</td>
<td>1</td>
<td>DA</td>
<td>A</td>
</tr>
<tr>
<td>D5</td>
<td>3</td>
<td>DJNZ</td>
<td>data addr,code addr</td>
</tr>
<tr>
<td>D6</td>
<td>1</td>
<td>XCHD</td>
<td>A,@R0</td>
</tr>
<tr>
<td>D7</td>
<td>1</td>
<td>XCHD</td>
<td>A,@R1</td>
</tr>
<tr>
<td>D8</td>
<td>2</td>
<td>DJNZ</td>
<td>R0,code addr</td>
</tr>
<tr>
<td>D9</td>
<td>2</td>
<td>DJNZ</td>
<td>R1,code addr</td>
</tr>
<tr>
<td>DA</td>
<td>2</td>
<td>DJNZ</td>
<td>R2,code addr</td>
</tr>
<tr>
<td>DB</td>
<td>2</td>
<td>DJNZ</td>
<td>R3,code addr</td>
</tr>
<tr>
<td>DC</td>
<td>2</td>
<td>DJNZ</td>
<td>R4,code addr</td>
</tr>
<tr>
<td>DD</td>
<td>2</td>
<td>DJNZ</td>
<td>R5,code addr</td>
</tr>
<tr>
<td>DE</td>
<td>2</td>
<td>DJNZ</td>
<td>R6,code addr</td>
</tr>
<tr>
<td>DF</td>
<td>2</td>
<td>DJNZ</td>
<td>R7,code addr</td>
</tr>
<tr>
<td>E0</td>
<td>1</td>
<td>MOVX</td>
<td>A,@DPTR</td>
</tr>
<tr>
<td>E1</td>
<td>2</td>
<td>AJMP</td>
<td>code addr</td>
</tr>
<tr>
<td>E2</td>
<td>1</td>
<td>MOVX</td>
<td>A,@R0</td>
</tr>
<tr>
<td>E3</td>
<td>1</td>
<td>MOVX</td>
<td>A,@R1</td>
</tr>
<tr>
<td>E4</td>
<td>1</td>
<td>CLR</td>
<td>A</td>
</tr>
<tr>
<td>E5</td>
<td>2</td>
<td>MOV</td>
<td>A,data addr</td>
</tr>
<tr>
<td>E6</td>
<td>1</td>
<td>MOV</td>
<td>A,@R0</td>
</tr>
<tr>
<td>E7</td>
<td>1</td>
<td>MOV</td>
<td>A,@R1</td>
</tr>
<tr>
<td>E8</td>
<td>1</td>
<td>MOV</td>
<td>A,R0</td>
</tr>
<tr>
<td>E9</td>
<td>1</td>
<td>MOV</td>
<td>A,R1</td>
</tr>
<tr>
<td>EA</td>
<td>1</td>
<td>MOV</td>
<td>A,R2</td>
</tr>
<tr>
<td>EB</td>
<td>1</td>
<td>MOV</td>
<td>A,R3</td>
</tr>
<tr>
<td>EC</td>
<td>1</td>
<td>MOV</td>
<td>A,R4</td>
</tr>
<tr>
<td>ED</td>
<td>1</td>
<td>MOV</td>
<td>A,R5</td>
</tr>
<tr>
<td>EE</td>
<td>1</td>
<td>MOV</td>
<td>A,R6</td>
</tr>
<tr>
<td>EF</td>
<td>1</td>
<td>MOV</td>
<td>A,R7</td>
</tr>
<tr>
<td>F0</td>
<td>1</td>
<td>MOVX</td>
<td>@DPTR,A</td>
</tr>
<tr>
<td>F1</td>
<td>2</td>
<td>ACALL</td>
<td>code addr</td>
</tr>
<tr>
<td>F2</td>
<td>1</td>
<td>MOVX</td>
<td>@R0,A</td>
</tr>
<tr>
<td>F3</td>
<td>1</td>
<td>MOVX</td>
<td>@R1,A</td>
</tr>
<tr>
<td>F4</td>
<td>1</td>
<td>CPL</td>
<td>A</td>
</tr>
<tr>
<td>F5</td>
<td>2</td>
<td>MOV</td>
<td>data addr,A</td>
</tr>
<tr>
<td>F6</td>
<td>1</td>
<td>MOV</td>
<td>@R0,A</td>
</tr>
<tr>
<td>F7</td>
<td>1</td>
<td>MOV</td>
<td>@R1,A</td>
</tr>
<tr>
<td>F8</td>
<td>1</td>
<td>MOV</td>
<td>R0,A</td>
</tr>
<tr>
<td>F9</td>
<td>1</td>
<td>MOV</td>
<td>R1,A</td>
</tr>
<tr>
<td>FA</td>
<td>1</td>
<td>MOV</td>
<td>R2,A</td>
</tr>
<tr>
<td>FB</td>
<td>1</td>
<td>MOV</td>
<td>R3,A</td>
</tr>
<tr>
<td>FC</td>
<td>1</td>
<td>MOV</td>
<td>R4,A</td>
</tr>
<tr>
<td>FD</td>
<td>1</td>
<td>MOV</td>
<td>R5,A</td>
</tr>
<tr>
<td>FE</td>
<td>1</td>
<td>MOV</td>
<td>R6,A</td>
</tr>
<tr>
<td>FF</td>
<td>1</td>
<td>MOV</td>
<td>R7,A</td>
</tr>
</tbody>
</table>
ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the 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.

Example: Initially SP equals 07H. The label “SUBRTN” is at program memory location 0345H. 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.

Bytes: 2
Cycles: 2

Encoding:

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

Operation:

ACALL

(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC7:0)
(SP) ← (SP) + 1
((SP)) ← (PC15:8)
(PC10:0) ← page address
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: 0010 1 r r r

Operation: ADD (A) ← (A) + (Rn)

ADD A, direct

Bytes: 2
Cycles: 1

Encoding: 0010 0 1 0 1

direct address

Operation: ADD (A) ← (A) + (direct)
ADD  A,Ri

Bytes: 1
Cycles: 1

Encoding: 0010 011i
Operation: ADD (A) ← (A) + ((Ri))

ADD  A,#data

Bytes: 2
Cycles: 1

Encoding: 0010 0100
Operation: ADD (A) ← (A) + #data

ADDC  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,

ADDC  A,R0

will leave 6EH (01101110B) in the Accumulator with AC cleared and both the Carry flag and OV set to 1.
MCS®-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET

ADDC $A,Rn$
- Bytes: 1
- Cycles: 1
- Encoding: 0011 1rrr
- Operation: ADDC
  \( (A) \leftarrow (A) + (C) + (R_n) \)

ADDC $A,direct$
- Bytes: 2
- Cycles: 1
- Encoding: 0011 0101 direct address
- Operation: ADDC
  \( (A) \leftarrow (A) + (C) + (direct) \)

ADDC $A,@Ri$
- Bytes: 1
- Cycles: 1
- Encoding: 0011 011i
- Operation: ADDC
  \( (A) \leftarrow (A) + (C) + ([R_i]) \)

ADDC $A,#data$
- Bytes: 2
- Cycles: 1
- Encoding: 0011 0100 immediate data
- Operation: ADDC
  \( (A) \leftarrow (A) + (C) + #data \)
**AJMP addr1**

**Function:** Absolute Jump

**Description:** AJMP transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bits of the PC (after incrementing the PC 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.

**Example:** The label "JMPADR" is at program memory location 0123H. The instruction,

```
AJMP JMPADR
```

is at location 0345H and will load the PC with 0123H.

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

**Encoding:**

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

**Operation:**

\[
\text{AJMP} \\
(\text{PC}) \leftarrow (\text{PC}) + 2 \\
(\text{PC}_{10}) \leftarrow \text{page address}
\]

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

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

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

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

**Example:** If the Accumulator holds OC3H (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.
**ANL A,Rn**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** \[0101 \text{ } 1 \text{ } r \text{ } r \text{ } r\]
- **Operation:** ANL
  \[(A) \leftarrow (A) \wedge (Rn)\]

**ANL A,direct**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** \[0101 \text{ } 0 \text{ } 1 \text{ } 0 \text{ } 1\]
- **Operation:** ANL
  \[(A) \leftarrow (A) \wedge (\text{direct})\]

**ANL A,@Ri**

- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** \[0101 \text{ } 0 \text{ } 1 \text{ } 1 \text{ } i\]
- **Operation:** ANL
  \[(A) \leftarrow (A) \wedge ((Ri))\]

**ANL A,#data**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** \[0101 \text{ } 0 \text{ } 1 \text{ } 0 \text{ } 0\]
- **Operation:** ANL
  \[(A) \leftarrow (A) \wedge \#\text{data}\]

**ANL direct,A**

- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** \[0101 \text{ } 0 \text{ } 0 \text{ } 1 \text{ } 0\]
- **Operation:** ANL
  \[(\text{direct}) \leftarrow (\text{direct}) \wedge (A)\]
### ANL direct, #data

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

**Encoding:**  
0101 0011  
direct address  
immediate data

**Operation:**  
ANL  
(direct) ← (direct) ∧ #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.

**Example:** Only direct addressing is allowed for the source operand.

- 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:**  
1000 0010  
bit address

**Operation:**  
ANL  
(C) ← (C) ∧ (bit)

### ANL C,/bit

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

**Encoding:**  
1011 0000  
bit address

**Operation:**  
ANL  
(C) ← (C) ∧ (bit)
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 altered.

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: 1011 0101 direct address rel. address

Operation: \[
\begin{align*}
\text{IF (A) <> (direct) THEN} \\
\quad (PC) & \leftarrow (PC) + \text{relative offset} \\
\text{IF (A) < (direct) THEN} \\
\text{ELSE} & \\
(C) & \leftarrow 1 \\
(C) & \leftarrow 0
\end{align*}
\]
CJNE A,#data,rel
Bytes: 3
Cycles: 2
Encoding: 1 0 1 1 0 1 0 0 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: 1 0 1 1 1 r 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: 1 0 1 1 0 1 1 i 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 set on 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: 1 1 1 0 0 1 0 0
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).

CLR C

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

CLR bit

Bytes: 2
Cycles: 1
Encoding: 1 1 0 0 0 0 1 0  bit address
Operation: CLR
(bit) ← 0
### CPL A

<table>
<thead>
<tr>
<th>Function:</th>
<th>Complement Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>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.</td>
</tr>
<tr>
<td>Example:</td>
<td>The Accumulator contains 5CH (01011100B). The instruction, CPL A will leave the Accumulator set to 0A3H (10100011B).</td>
</tr>
<tr>
<td>Bytes:</td>
<td>1</td>
</tr>
<tr>
<td>Cycles:</td>
<td>1</td>
</tr>
<tr>
<td>Encoding:</td>
<td>1111 0100</td>
</tr>
<tr>
<td>Operation:</td>
<td>CPL (A) ← ¬(A)</td>
</tr>
</tbody>
</table>

### CPL bit

<table>
<thead>
<tr>
<th>Function:</th>
<th>Complement bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>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. CLR can operate on the carry or any directly addressable bit.</td>
</tr>
<tr>
<td>Example:</td>
<td>Port 1 has previously been written with 5BH (01011101B). The instruction sequence, CPL P1.1, CPL P1.2 will leave the port set to 5BH (01011011B).</td>
</tr>
<tr>
<td>Note:</td>
<td>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.</td>
</tr>
</tbody>
</table>

### CPL C

| Bytes:        | 1                       |
| Cycles:       | 1                       |
| Encoding:     | 1011 0011               |
| Operation:    | CPL (C) ← ¬(C)          |
DA A

**Function:** Decimal-adjust Accumulator for Addition

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

If Accumulator bits 3–0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the Accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-111xxx), 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 1100, 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.

```
ADDCA, R3
DA A
```

will first perform a standard two's-complement binary addition, resulting in the value 0BEH (10111110) in the Accumulator. The carry and auxiliary carry flags will be cleared.

The Decimal Adjust instruction will then alter the Accumulator to the value 24H (00100100B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-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 001H or 99H. If the Accumulator initially holds 30H (representing the digits of 30 decimal), then 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: 1101 0100
Operation: DA
<br>
- contents of Accumulator are BCD
- IF [(A3,0) > 9] \lor [(AC) = 1]]
  THEN (A3,0) ← (A3,0) + 6
  AND
- IF [(A7,4) > 9] \lor [(C) = 1]]
  THEN (A7,4) ← (A7,4) + 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 port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7FH (01111111B). 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: 0001 0100
Operation: DEC (A) \(\leftarrow (A) - 1\)

DEC Rn

Bytes: 1
Cycles: 1

Encoding: 0001 1rrr
Operation: DEC (Rn) \(\leftarrow (Rn) - 1\)
DEC direct

Bytes: 2
Cycles: 1
Encoding: 0001 0101
Operation: DEC (direct) ← (direct) – 1

DEC @Ri

Bytes: 1
Cycles: 1
Encoding: 0001 0111
Operation: DEC ((Ri)) ← ((Ri)) – 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 (1IH or 00010001B) in B, since 251 = (13 × 18) + 17. Carry and OV will both be cleared.

Bytes: 1
Cycles: 4
Encoding: 1000 0100
Operation: DIV

(A)15-8 ← (A)/(B)
(B)7-0 ← (A)/(B)
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 OOH 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 O1H, 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 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: 1111011 r r r rel. address
Operation: DJNZ
(PC) ← (PC) + 2
(Rn) ← (Rn) - 1
IF (Rn) > 0 or (Rn) < 0
THEN
(PC) ← (PC) + rel

2-43
DJNZ direct,rel

Bytes: 3
Cycles: 2

Encoding: 1 1 0 1 0 1 0 1

Operation:
DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) - 1
IF (direct) > 0 or (direct) < 0
THEN
   (PC) ← (PC) + 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 (0111 11110B). 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: 0000 1111  
Operation: INC (Rn) ← (Rn) + 1  

INC direct

Bytes: 2  
Cycles: 1  
Encoding: 0000 0101 (direct address)  
Operation: INC (direct) ← (direct) + 1  

INC @Ri

Bytes: 1  
Cycles: 1  
Encoding: 0000 0111  
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: 1010 0011  
Operation: INC (DPTR) ← (DPTR) + 1  

2-45
**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

<table>
<thead>
<tr>
<th>Encoding</th>
<th>bit address</th>
<th>rel. address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

**Operation:**

```
(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 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 be 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 label LABEL2, with the Accumulator modified to 52H (01010010B).
MCS®-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET

Bytes: 3
Cycles: 2

Encoding:

Operation: JBC
(PC) ← (PC) + 3
IF (bit) = 1
THEN
(bit) ← 0
(PC) ← (PC) + 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 LABEL 2
```

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

Bytes: 2
Cycles: 2

Encoding:

Operation: JC
(PC) ← (PC) + 2
IF (C) = 1
THEN
(PC) ← (PC) + rel

2-47
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
(\text{PC} \leftarrow (A) + (DPTR))
MCS®-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET

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: 0011 0000 0000 0000

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: 0101 0000 0000

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

2-49
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,

\[
\text{JNZ LABEL1}
\text{INC A}
\text{JNZ LABEL2}
\]

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

**Bytes:** 2  
**Cycles:** 2  
**Encoding:** 0111 0000

**Operation:**  
\[
\text{JNZ (PC)} \rightarrow (PC) + 2
\text{IF } (A) \neq 0
\text{THEN } (PC) \rightarrow (PC) + \text{rel}
\]

JZ rel

**Function:** Jump if Accumulator Zero

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

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

\[
\text{JZ LABEL1}
\text{DEC A}
\text{JZ LABEL2}
\]

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:** 0110 0000

**Operation:**  
\[
\text{JZ (PC)} \rightarrow (PC) + 2
\text{IF } (A) = 0
\text{THEN } (PC) \rightarrow (PC) + \text{rel}
\]

2-50
**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 1234H.

**Bytes:** 3

**Cycles:** 2

**Encoding:**

| 0001 | 0010 | addr15-addr8 | addr7-addr0 |

**Operation:**

```
LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC) + 3
(PC) ← addr15-0
```

---

**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:**

| 0000 | 0010 | addr15-addr8 | addr7-addr0 |

**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).

MOV R0,#30H ;RO < = 30H
MOV A,@RO ;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 O, 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.

MOV A,Rn
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: 1 1 1 0 0 1 0 1
Operation: MOV (A) ← (direct)

MOV A,ACC is not a valid instruction.
### MOV A, Ri
- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** \[1110 \ 011i\]
- **Operation:**
  \[
  \text{MOV (A) } \leftarrow (\text{Ri})
  \]

### MOV A, #data
- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** \[0111 \ 0100\] (immediate data)
- **Operation:**
  \[
  \text{MOV (A) } \leftarrow \#\text{data}
  \]

### MOV Rn, A
- **Bytes:** 1
- **Cycles:** 1
- **Encoding:** \[1111 \ 1rrr\]
- **Operation:**
  \[
  \text{MOV (Rn) } \leftarrow (A)
  \]

### MOV Rn, direct
- **Bytes:** 2
- **Cycles:** 2
- **Encoding:** \[1010 \ 1rrr\] (direct addr.)
- **Operation:**
  \[
  \text{MOV (Rn) } \leftarrow (\text{direct})
  \]

### MOV Rn, #data
- **Bytes:** 2
- **Cycles:** 1
- **Encoding:** \[0111 \ 1rrr\] (immediate data)
- **Operation:**
  \[
  \text{MOV (Rn) } \leftarrow \#\text{data}
  \]
MOV direct,A
Bytes: 2
Cycles: 1
Encoding: 1111 0101
direct address
Operation: MOV (direct) ← (A)

MOV direct,Rn
Bytes: 2
Cycles: 2
Encoding: 1000 1rrr
direct address
Operation: MOV (direct) ← (Rn)

MOV direct,direct
Bytes: 3
Cycles: 2
Encoding: 1000 0101
dir. addr. (src) dir. addr. (dest)
Operation: MOV (direct) ← (direct)

MOV direct,@Ri
Bytes: 2
Cycles: 2
Encoding: 1000 011i
direct addr.
Operation: MOV (direct) ← ((Ri))

MOV direct,#data
Bytes: 3
Cycles: 2
Encoding: 0111 0101
direct address immediate data
Operation: MOV (direct) ← #data
MOV @Ri,A
Bytes: 1
Cycles: 1
Encoding: 1111 011
Operation: MOV ((RI)) ← (A)

MOV @Ri, direct
Bytes: 2
Cycles: 2
Encoding: 1010 011
Operation: MOV ((RI)) ← (direct)

MOV @Ri, #data
Bytes: 2
Cycles: 1
Encoding: 0111 011
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).

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: 1010 0010
Operation: MOV (C) ← (bit)

MOV bit, C

Bytes: 2
Cycles: 2

Encoding: 1001 0010
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.

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: 1001 0000
Operation: MOV
(DPTR) ← #data15-0
DPH, DPL ← #data15-8 #data7-0
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: 10010011

Operation: MOVC (A) ← ((A) + (DPTR))

MOVC A, @A + PC

Bytes: 1
Cycles: 2

Encoding: 10000011

Operation: MOVC (PC) ← (PC) + 1
(A) ← ((A) + (PC))
**MOVX** `<dest-byte>,<src-byte>`

<table>
<thead>
<tr>
<th>Function:</th>
<th>Move External</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The MOVX instructions transfer data between the Accumulator and a byte of external data memory, hence the &quot;X&quot; appended to MOV. There are two types of instructions, differing in whether they provide an eight-bit or sixteen-bit indirect address to the external data RAM.</td>
</tr>
</tbody>
</table>

In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the Data Pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low-order eight bits (DPL) with data. The P2 Special Function Register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64K bytes), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be accessed via the Data Pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

<table>
<thead>
<tr>
<th>Example:</th>
<th>An external 256 byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) 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,</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVX A,@R1</td>
<td></td>
</tr>
<tr>
<td>MOVX @R0,A</td>
<td></td>
</tr>
</tbody>
</table>

copies the value 56H into both the Accumulator and external RAM location 12H.
MO VX  A, @Ri
Bytes: 1
Cycles: 2
Encoding: 1110 0010
Operation: MOVX (A) ← ((Ri))

MO VX  A, @DPTR
Bytes: 1
Cycles: 2
Encoding: 1110 0000
Operation: MOVX (A) ← ((DPTR))

MO VX  @Ri, A
Bytes: 1
Cycles: 2
Encoding: 1111 0010
Operation: MOVX ((Ri)) ← (A)

MO VX  @DPTR, A
Bytes: 1
Cycles: 2
Encoding: 1111 0000
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: 1010 0100

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 no interrupts are enabled) with the instruction sequence,

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

Bytes: 1
Cycles: 1

Encoding: 0000 0000

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 OC3H (11000011B) and R0 holds 55H (01010101B) then the instruction,

ORL A, R0

will leave the Accumulator holding the value OD7H (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: 0100 1 rrr

Operation: ORL (A) ← (A) V (Rn)
ORL A, direct
Bytes: 2
Cycles: 1
Encoding: 0100 0101  
Operation: ORL (A) ← (A) V (direct)

ORL A, @Ri
Bytes: 1
Cycles: 1
Encoding: 0100 0111
Operation: ORL (A) ← (A) V (@(Ri))

ORL A, #data
Bytes: 2
Cycles: 1
Encoding: 0100 0100  
Operation: ORL (A) ← (A) V #data

ORL direct, A
Bytes: 2
Cycles: 1
Encoding: 0100 0010  
Operation: ORL (direct) ← (direct) V (A)

ORL direct, #data
Bytes: 3
Cycles: 2
Encoding: 0100 0011  
Operation: ORL (direct) ← (direct) V #data

2-62
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:

```
MOV 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: 0111 0010

Operation: ORL
(C) ← (C) OR (bit)

ORL C,/bit

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

```

Encoding: 1010 0000

Operation: ORL
(C) ← (C) OR (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:** \[11 01 00 00\]  

**Operation:** \[
\begin{align*}
\text{POP (direct) } & \leftarrow ((\text{SP})) \\
\text{(SP) } & \leftarrow (\text{SP}) - 1
\end{align*}
\]

### 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 OBH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

**Bytes:** 2

**Cycles:** 2

**Encoding:** \[11 00 00 00\]  

**Operation:** \[
\begin{align*}
\text{PUSH (SP) } & \leftarrow (\text{SP}) + 1 \\
((\text{SP})) & \leftarrow (\text{direct})
\end{align*}
\]
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.

Bytes: 1
Cycles: 2
Encoding: 0010 0010

Operation:

RET
(PC<15:8>) ← ((SP))
(SP) ← (SP) − 1
(PC7:0) ← ((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 had 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.

Bytes: 1
Cycles: 2
Encoding: 0011 0010

Operation:

RETI
(PC<15:8>) ← ((SP))
(SP) ← (SP) − 1
(PC7:0) ← ((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 OC5H (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) ← (An) 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 OC5H (11000101B), and the carry is zero. The instruction, RLC A
leaves the Accumulator holding the value 8BH (10001010B) with the carry set.

Bytes: 1
Cycles: 1
Encoding: 0 0 1 1 0 0 1 1
Operation: RLC
(A_n + 1) ← (An) 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.

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

**Encoding:** \(0000\ 0011\)

**Operation:**
\[
\begin{align*}
(A_n) \leftarrow (A_n + 1) & \quad n = 0 - 6 \\
(A7) \leftarrow (A0)
\end{align*}
\]

## RRC A

**Function:** Rotate Accumulator Right through 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 value of the carry flag moves into the bit 7 position. No other flags are affected.

**Example:** The Accumulator holds the value 0C5H (11000101B), the carry is zero. The instruction, RRC A, leaves the Accumulator holding the value 62 (01100010B) with the carry set.

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

**Encoding:** \(0001\ 0011\)

**Operation:**
\[
\begin{align*}
(A_n) \leftarrow (A_n + 1) & \quad n = 0 - 6 \\
(A7) \leftarrow (C) \\
(C) \leftarrow (A0)
\end{align*}
\]
<table>
<thead>
<tr>
<th>Function:</th>
<th>Set Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>SETB sets the indicated bit to one. SETB can operate on the carry flag or any directly addressable bit. No other flags are affected.</td>
</tr>
<tr>
<td>Example:</td>
<td>The carry flag is cleared. Output Port 1 has been written with the value 34H (00110100B). The instructions,</td>
</tr>
<tr>
<td></td>
<td>SETB C</td>
</tr>
<tr>
<td></td>
<td>SETB P1.0</td>
</tr>
<tr>
<td></td>
<td>will leave the carry flag set to 1 and change the data output on Port 1 to 35H (00110101B).</td>
</tr>
</tbody>
</table>

**SETB <bit>**

<table>
<thead>
<tr>
<th>Bytes:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>1</td>
</tr>
<tr>
<td>Encoding:</td>
<td>[11010011]</td>
</tr>
<tr>
<td>Operation:</td>
<td>SETB ( (C) \leftarrow 1 )</td>
</tr>
</tbody>
</table>

**SETB C**

<table>
<thead>
<tr>
<th>Bytes:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles:</td>
<td>1</td>
</tr>
<tr>
<td>Encoding:</td>
<td>[11010010]</td>
</tr>
<tr>
<td>Operation:</td>
<td>SETB ( (bit) \leftarrow 1 )</td>
</tr>
</tbody>
</table>
## 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 OFEH would be a one-instruction infinite loop.)

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

**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: | 1001 rrr |
| Operation: | SUBB (A) ← (A) − (C) − (Rn) |
SUBB A, direct

Bytes: 2
Cycles: 1

Encoding: 1001 0101
Operation: SUBB (A) ← (A) - (C) - (direct)

SUBB A, @Ri

Bytes: 1
Cycles: 1

Encoding: 1001 0111
Operation: SUBB (A) ← (A) - (C) - ((Ri))

SUBB A, #data

Bytes: 2
Cycles: 1

Encoding: 1001 0100
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: 1100 0100
Operation: SWAP (A3:0) ← (A7:4)
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 RAM location 20H holding the values 3FH (00111111B) and 75H (01110101B) in the accumulator.

XCH A, Rn

Bytes: 1
Cycles: 1
Encoding: 1100 1 rrr
Operation: XCH (A) \(\rightarrow\) (Rn)

XCH A, direct

Bytes: 2
Cycles: 1
Encoding: 1100 0101 \(\text{direct address}\)
Operation: XCH (A) \(\rightarrow\) (direct)

XCH A, @Ri

Bytes: 1
Cycles: 1
Encoding: 1100 0111
Operation: XCH (A) \(\rightarrow\) ((Ri))
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: `1101 0111`

Operation: XCHD

\[(A_{3:0}) \leftrightarrow (R_{3:0})\]

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 OC3H (11000011B) and register 0 holds OA0H (10101010B) then the instruction,

```
XRL A,R0
```

will leave the Accumulator holding the value 69H (01101011B).

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
Bytes: 1
Cycles: 1
Encoding: 0110 1rrr
Operation: XRL
(A) ← (A) ⊕ (Rn)

XRL A,direct
Bytes: 2
Cycles: 1
Encoding: 0110 0101
Operation: XRL
(A) ← (A) ⊕ (direct)

XRL A,@Ri
Bytes: 1
Cycles: 1
Encoding: 0110 011i
Operation: XRL
(A) ← (A) ⊕ ((Ri))

XRL A,#data
Bytes: 2
Cycles: 1
Encoding: 0110 0100
Operation: XRL
(A) ← (A) ⊕ #data

XRL direct,A
Bytes: 2
Cycles: 1
Encoding: 0110 0010
Operation: XRL
(direct) ← (direct) ⊕ (A)
XRL  direct,#data

Bytes:  3
Cycles:  2

Encoding:  0 1 1 0 0 0 1 1  direct address  immediate data

Operation:  XRL
            (direct) ← (direct) V #data