### Notes on Data Addressing Modes

| | | |
|---|---|---|
| Rn | - | Working register R0-R7 |
| direct | - | 128 internal RAM locations, any I/O port, control or status register |
| @Ri | - | Indirect internal or external RAM location addressed by register R0 or R1 |
| #data | - | 8-bit constant included in instruction |
| #data 16 | - | 16-bit constant included as bytes 2 and 3 of instruction |
| bit | - | 128 software flags, any bitaddressable I/O pin, control or status bit |
| A | - | Accumulator |

### Notes on Program Addressing Modes

| | | |
|---|---|---|
| addr16 | - | Destination address for LCALL and LJMP may be anywhere within the 64-Kbyte program memory address space. |
| addr11 | - | Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction. |
| rel | - | SJMP and all conditional jumps include an 8 bit offset byte. Range is + 127/– 128 bytes relative to the first byte of the following instruction. |

**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, op code bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals $07_H$. The label "SUBRTN" is at program memory location $0345_H$. After executing the instruction

ACALL SUBRTN

at location $0123_H$, SP will contain $09_H$, internal RAM location $08_H$ and $09_H$ will contain $25_H$ and $01_H$, respectively, and the PC will contain $0345_H$.

Operation: ACALL
(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC7-0)
(SP) ← (SP) + 1
((SP)) ← (PC15-8)
(PC10-0) ← page address

Encoding:

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

Bytes: 2

Cycles: 2

**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 of 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 $0C3_H$ ($11000011_B$) and register 0 holds $0AA_H$ ($10101010_B$). The instruction

ADD A,R0

will leave $6D_H$ ($01101101_B$) in the accumulator with the AC flag cleared and both the carry flag and OV set to 1.

**ADD A,Rn**

Operation: ADD
(A) ← (A) + (Rn)

Encoding:

| 0 0 1 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**ADD A,direct**

Operation: ADD
(A) ← (A) + (direct)

Encoding:

| 0 0 1 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**ADD A, @Ri**

Operation: ADD
(A) ← (A) + ((Ri))

Encoding:

| | |
|---|---|
| 0 0 1 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

**ADD A, #data**

Operation: ADD
(A) ← (A) + #data

Encoding:

| | | |
|---|---|---|
| 0 0 1 0 | 0 1 0 0 | immediate data |

Bytes: 2

Cycles: 1

**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 of 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 $0C3_H$ (11000011B) and register 0 holds $0AA_H$ (10101010B) with the carry flag set. The instruction

ADDC A,R0

will leave $6E_H$ (01101110B) in the accumulator with AC cleared and both the carry flag and OV set to 1.

**ADDC A,Rn**

Operation: ADDC
(A) ← (A) + (C) + (Rn)

Encoding:

| 0 0 1 1 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**ADDC A,direct**

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

Encoding:

| 0 0 1 1 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**ADDC A, @Ri**

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

Encoding:

| | |
|---|---|
| 0 0 1 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

**ADDC A, #data**

Operation: ADDC
(A) ← (A) + (C) + #data

Encoding:

| | | |
|---|---|---|
| 0 0 1 1 | 0 1 0 0 | immediate data |

Bytes: 2

Cycles: 1

**AJMP addr11**

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), op code 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 $0123_H$. The instruction

AJMP JMPADR

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

Operation: AJM P
(PC) ← (PC) + 2
(PC10-0) ← page address

Encoding:

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

Bytes: 2

Cycles: 2

**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 a 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 $0C3_H$ (11000011B) and register 0 holds $0AA_H$ (10101010B) then the instruction

ANL A,R0

will leave $81_H$ (10000001B) 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**

Operation: ANL
(A) $\leftarrow$ (A) $\wedge$ (Rn)

Encoding:

| | |
|---|---|
| 0 1 0 1 | 1 r r r |

Bytes: 1

Cycles: 1

**ANL A,direct**

Operation: ANL
(A) ← (A) ∧ (direct)

Encoding:

| | | |
|---|---|---|
| 0 1 0 1 | 0 1 0 1 | direct address |

Bytes: 2

Cycles: 1

**ANL A, @Ri**

Operation: ANL
(A) ← (A) ∧ ((Ri))

Encoding:

| | |
|---|---|
| 0 1 0 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

**ANL A, #data**

Operation: ANL
(A) ← (A) ∧ #data

Encoding:

| | | |
|---|---|---|
| 0 1 0 1 | 0 1 0 0 | immediate data |

Bytes: 2

Cycles: 1

**ANL direct,A**

Operation: ANL
(direct) ← (direct) ∧ (A)

Encoding:

| | | |
|---|---|---|
| 0 1 0 1 | 0 1 0 1 | direct address |

Bytes: 2

Cycles: 1

**ANL direct, #data**

Operation: ANL
(direct) ← (direct) ∧ #data

Encoding:

| 0 1 0 1 | 0 0 1 1 | direct address | immediate data |
|---|---|---|---|

Bytes: 3

Cycles: 2

**ANL C, <src-bit>**

Function: Logical AND for bit variables

Description: If the Boolean value of the source bit is a logic 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 bit 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 accumulator bit 7
ANL     C,/OV           ; AND with inverse of overflow flag
```

**ANL C,bit**

Operation: ANL
(C) ← (C) ∧ (bit)

Encoding:

| 1 0 0 0 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 2

**ANL C,/bit**

Operation: ANL
(C) ← (C) ∧ ¬ (bit)

Encoding:

| 1 0 1 1 | 0 0 0 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 2

**CJNE <dest-byte >, < src-byte >, rel**

Function: Compare and jump if not equal

Description: CJNE compares the magnitudes of the tirst 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 $34_H$. Register 7 contains $56_H$. The first instruction in the sequence

```
                CJNE    R7, # 60H, NOT_EQ
;               . . .   . . . . .          ; R7 = 60H
NOT_EQ          JC      REQ_LOW            ; If R7 < 60H
;               . . .   . . . . .          ; R7 > 60H
```

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 $60_H$.

If the data being presented to port 1 is also $34_H$, 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 input on P1, the program will loop at this point until the P1 data changes to $34_H$).

**CJNE A,direct,rel**

Operation: (PC) ← (PC) + 3
if (A) < > (direct)
then (PC) ← (PC) + relative offset
if (A) < (direct)
then (C) ←1
else (C) ← 0

Encoding:

| 1 0 1 1 | 0 1 0 1 | direct address | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**CJNE A, #data,rel**

Operation: (PC) ← (PC) + 3
if (A) < > data
then (PC) ← (PC) + relative offset
if (A) ← data
then (C) ←1
else (C) ← 0

Encoding:

| 1 0 1 1 | 0 1 0 0 | immediate data | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**CJNE RN, #data, rel**

Operation: (PC) ← (PC) + 3
if (Rn) < > data
then (PC) ← (PC) + relative offset
if (Rn) < data
then (C) ← 1
else (C) ← 0

Encoding:

| 1 0 1 1 | 1 r r r | immediate data | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**CJNE @Ri, #data,rel**

Operation: (PC) ← (PC) + 3
if ((Ri)) < > data
then (PC) ← (PC) + relative offset
if ((Ri)) < data
then (C) ← 1
else (C) ← 0

Encoding:

| 1 0 1 1 | 0 1 1 i | immediate data | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**CLR A**

Function: Clear accumulator

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

Example: The accumulator contains $5C_H$ (01011100B). The instruction

CLR A

will leave the accumulator set to $00_H$ (00000000B).

Operation: CLR
(A) ← 0

Encoding:

| | |
|---|---|
| 1 1 1 0 | 0 1 0 0 |

Bytes: 1

Cycles: 1

**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 $5D_H$ (01011101B). The instruction

CLR P1.2

will leave the port set to $59_H$ (01011001B).

**CLR C**

Operation: CLR
(C) ← 0

Encoding:

| 1 1 0 0 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 1

**CLR bit**

Operation: CLR
(bit) ← 0

Encoding:

| 1 1 0 0 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 1

**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 zero and vice versa. No flags are affected.

Example: The accumulator contains $5C_H$ (01011100B). The instruction

CPL A

will leave the accumulator set to $0A3_H$ (10100011 B).

Operation: CPL
(A) $\leftarrow \neg$ (A)

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

Bytes: 1

Cycles: 1

**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 $5D_H$ ($01011101_B$). The instruction sequence

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

will leave the port set to $5B_H$ ($01011011_B$).

**CPL C**

Operation: CPL
(C) ← ¬ (C)

Encoding:

| 1 0 1 1 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 1

**CPL bit**

Operation: CPL
(bit) ← ¬ (bit)

Encoding:

| 1 0 1 1 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 1

**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-1111xxxx), 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 $00_H$, $06_H$, $60_H$, or $66_H$ 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 $56_H$ (01010110B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value $67_H$ (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 $0BE_H$ (10111110B) in the accumulator. The carry and auxiliary carry flags will be cleared.

The decimal adjust instruction will then alter the accumulator to the value $24_H$ (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 $01_H$ or $99_H$. If the accumulator initially holds $30_H$ (representing the digits of 30 decimal), then the instruction sequence

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

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

Operation:

DA
contents of accumulator are BCD
if [[(A3-0) > 9] ∨ [(AC) = 1]]
then (A3-0) ← (A3-0) + 6
and
if [[(A7-4) > 9] ∨ [(C) = 1]]
then (A7-4) ← (A7-4) + 6

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

Bytes: 1

Cycles: 1

**DEC byte**

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of $00_H$ will underflow to $0FF_H$. 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 $7F_H$ (01111111B). Internal RAM locations $7E_H$ and $7F_H$ contain $00_H$ and $40_H$, respectively. The instruction sequence

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

will leave register 0 set to $7E_H$ and internal RAM locations $7E_H$ and $7F_H$ set to $0FF_H$ and $3F_H$.

**DEC A**

Operation: DEC
(A) ← (A) – 1

Encoding:

| 0 0 0 1 | 0 1 0 0 |
|---|---|

Bytes: 1

Cycles: 1

**DEC Rn**

Operation: DEC
(Rn) ← (Rn) – 1

Encoding:

| 0 0 0 1 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**DEC direct**

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

Encoding:

| 0 0 0 1 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**DEC @Ri**

Operation: DEC
((Ri)) ← ((Ri)) – 1

Encoding:

| 0 0 0 1 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 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 $00_H$, 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 ($0FB_H$ or 11111011B) and B contains 18 ($12_H$ or 00010010B). The instruction

DIV AB

will leave 13 in the accumulator ($0D_H$ or 00001101 B) and the value 17 ($11_H$ or 00010001B) in B, since 251 = (13x18) + 17. Carry and OV will both be cleared.

Operation: DIV

(A15-8)
(B7-0) $\leftarrow$ (A) / (B)

Encoding:

| | |
|---|---|
| 1 0 0 0 | 0 1 0 0 |

Bytes: 1

Cycles: 4

**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 $00_H$ will underflow to $0FF_H$. 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 $40_H$, $50_H$, and $60_H$ contain the values, $01_H$, $70_H$, and $15_H$, respectively. The instruction sequence

DJNZ $40_H$,LABEL_1
DJNZ $50_H$,LABEL_2
DJNZ $60_H$,LABEL_3

will cause a jump to the instruction at label LABEL_2 with the values $00_H$, $6F_H$, and $15_H$ 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**

Operation: DJNZ
(PC) ← (PC) + 2
(Rn) ← (Rn) – 1
if (Rn) > 0 or (Rn) < 0
then (PC) ← (PC) + rel

Encoding:

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

Bytes: 2

Cycles: 2

**DJNZ direct,rel**

Operation: DJNZ
(PC) ← (PC) + 2
(direct) ← (direct) – 1
if (direct) > 0 or (direct) < 0
then (PC) ← (PC) + rel

Encoding:

| 1 1 0 1 | 0 1 0 1 | direct address | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**INC <byte>**

Function: Increment

Description: INC increments the indicated variable by 1. An original value of $0FF_H$ will overflow to $00_H$. 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 $7E_H$ (01111110B). Internal RAM locations $7E_H$ and $7F_H$ contain $0FF_H$ and $40_H$, respectively. The instruction sequence

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

will leave register 0 set to $7F_H$ and internal RAM locations $7E_H$ and $7F_H$ holding (respectively) $00_H$ and $41_H$.

**INC A**

Operation: INC
(A) ← (A) + 1

Encoding:

| 0 0 0 0 | 0 1 0 0 |
|---|---|

Bytes: 1

Cycles: 1

**INC Rn**

Operation: INC
(Rn) ← (Rn) + 1

Encoding:

| 0 0 0 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**INC direct**

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

Encoding:

| 0 0 0 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**INC @Ri**

Operation: INC
((Ri)) ← ((Ri)) + 1

Encoding:

| 0 0 0 0 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 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 $0FF_H$ to $00_H$ 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 $12_H$ and $0FE_H$, respectively. The instruction sequence

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

will change DPH and DPL to $13_H$ and $01_H$.

Operation: INC
(DPTR) ← (DPTR) + 1

Encoding:

| 1 0 1 0 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 2

**JB bit,rel**

Function: Jump if bit is set

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

Example: The data present at 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.

Operation:
JB
(PC) ← (PC) + 3
if (bit) = 1
then (PC) ← (PC) + rel

Encoding:

| 0 0 1 0 | 0 0 0 0 | bit address | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**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. *In either case, clear the designated bit.* 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 $56_H$ (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 $52_H$ (01010010B).

Operation:
JBC
(PC) ← (PC) + 3
if (bit) = 1
then (bit) ← 0
(PC) ← (PC) + rel

Encoding:

| 0 0 0 1 | 0 0 0 0 | bit address | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**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.

Operation:
JC
(PC) ← (PC) + 2
if (C) = 1
then (PC) ← (PC) + rel

Encoding:

| 0 1 0 0 | 0 0 0 0 | rel. address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 $04_H$ 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.

Operation: JMP
(PC) ← (A) + (DPTR)

Encoding:

| 0 1 1 1 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 2

**JNB bit,rel**

Function: Jump if bit is not set

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

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

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

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

Operation: JNB
(PC) ← (PC) + 3
if (bit) = 0
then (PC) ← (PC) + rel.

Encoding:

| 0 0 1 1 | 0 0 0 0 | bit address | rel. address |
|---|---|---|---|

Bytes: 3

Cycles: 2

**JNC rel**

Function: Jump if carry is not set

Description: If the carry flag is a zero, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC 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.

Operation:
JNC
(PC) ← (PC) + 2
if (C) = 0
then (PC) ← (PC) + rel

Encoding:

| 0 1 0 1 | 0 0 0 0 | rel. address |
|---|---|---|

Bytes: 2

Cycles: 2

**JNZ rel**

Function: Jump if accumulator is 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 $00_H$. The instruction sequence

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

will set the accumulator to $01_H$ and continue at label LABEL2.

Operation:
JNZ
(PC) ← (PC) + 2
if (A) ≠ 0
then (PC) ← (PC) + rel.

Encoding:

| 0 1 1 1 | 0 0 0 0 | rel. address |
|---|---|---|

Bytes: 2

Cycles: 2

**JZ rel**

Function: Jump if accumulator is 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 $01_H$. The instruction sequence

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

will change the accumulator to $00_H$ and cause program execution to continue at the instruction identified by the label LABEL2.

Operation:
JZ
(PC) ← (PC) + 2
if (A) = 0
then (PC) ← (PC) + rel

Encoding:

| 0 1 1 0 | 0 0 0 0 | rel. address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 64 Kbyte program memory address space. No flags are affected.

Example: Initially the stack pointer equals $07_H$. The label "SUBRTN" is assigned to program memory location $1234_H$. After executing the instruction

LCALL SUBRTN

at location $0123_H$, the stack pointer will contain $09_H$, internal RAM locations $08_H$ and $09_H$ will contain $26_H$ and $01_H$, and the PC will contain $1234_H$.

Operation: LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC7-0)
(SP) ← (SP) + 1
((SP)) ← (PC15-8)
(PC) ← addr15-0

Encoding:

| 0 0 0 1 | 0 0 1 0 | addr15 . . addr8 | addr7 . . addr0 |
|---|---|---|---|

Bytes: 3

Cycles: 2

**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 $1234_H$. The instruction

LJMP JMPADR

at location $0123_H$ will load the program counter with $1234_H$.

Operation: LJMP
(PC) ← addr15-0

Encoding:

| | | | |
|---|---|---|---|
| 0 0 0 0 \| 0 0 1 0 | addr15 . . . addr8 | addr7 . . . addr0 |

Bytes: 3

Cycles: 2

**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 $30_H$ holds $40_H$. The value of RAM location $40_H$ is $10_H$. The data present at input port 1 is 11001010B ($0CA_H$).

```
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 $30_H$ in register 0, $40_H$ in both the accumulator and register 1, $10_H$ in register B, and $0CA_H$ (11001010B) both in RAM location $40_H$ and output on port 2.

**MOV A,Rn**

Operation: MOV
(A) ← (Rn)

Encoding:

| 1 1 1 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**MOV A,direct *)**

Operation: MOV
(A) ← (direct)

Encoding:

| 1 1 1 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

*) MOV A,ACC is not a valid instruction.

**MOV A,@Ri**

Operation: MOV
(A) ← ((Ri))

Encoding:

| 1 1 1 0 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 1

**MOV A, #data**

Operation: MOV
(A) ← #data

Encoding:

| 0 1 1 1 | 0 1 0 0 | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**MOV Rn,A**

Operation: MOV
(Rn) ← (A)

Encoding:

| 1 1 1 1 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**MOV Rn,direct**

Operation: MOV
(Rn) ← (direct)

Encoding:

| 1 0 1 0 | 1 r r r | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**MOV Rn, #data**

Operation: MOV
(Rn) ← #data

Encoding:

| 0 1 1 1 | 1 r r r | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**MOV direct,A**

Operation: MOV
(direct) ← (A)

Encoding:

| 1 1 1 1 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**MOV direct,Rn**

Operation: MOV
(direct) ← (Rn)

Encoding:

| 1 0 0 0 | 1 r r r | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**MOV direct,direct**

Operation: MOV
(direct) ← (direct)

Encoding:

| 1 0 0 0 | 0 1 0 1 | dir.addr. (src) | dir.addr. (dest) |
|---|---|---|---|

Bytes: 3

Cycles: 2

**MOV direct, @ Ri**

Operation: MOV
(direct) ← ((Ri))

Encoding:

| 1 0 0 0 | 0 1 1 i | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**MOV direct, #data**

Operation: MOV
(direct) ← #data

Encoding:

| 0 1 1 1 | 0 1 0 1 | direct address | immediate data |
|---|---|---|---|

Bytes: 3

Cycles: 2

**MOV @ Ri,A**

Operation: MOV
((Ri)) ← (A)

Encoding:

| 1 1 1 1 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 1

**MOV @ Ri,direct**

Ooeration: MOV
((Ri)) ← (direct)

Encoding:

| 1 0 1 0 | 0 1 1 i | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**MOV @ Ri,#data**

Operation: MOV
((Ri)) ← #data

Encoding:

| 0 1 1 1 | 0 1 1 i | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**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 $35_H$ (00110101B).

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

will leave the carry cleared and change port 1 to $39_H$ (00111001 B).

**MOV C,bit**

Operation: MOV
(C) ← (bit)

Encoding:

| 1 0 1 0 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 1

**MOV bit,C**

Operation: MOV
(bit) ← (C)

Encoding:

| 1 0 0 1 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 2

**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, #$1234_H$

will load the value $1234_H$ into the data pointer: DPH will hold $12_H$ and DPL will hold $34_H$.

Operation: MOV
(DPTR) ← #data15-0
DPH ☐ DPL ← #data15-8 ☐ #data7-0

Encoding:

| 1 0 0 1 | 0 0 0 0 | immed. data 15 . . . 8 | immed. data 7 . . . 0 |
|---|---|---|---|

Bytes: 3

Cycles: 2

**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 to 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 $01_H$, it will return with $77_H$ 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**

Operation: MOVC
(A) ← ((A) + (DPTR))

Encoding:

| 1 0 0 1 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 2

**MOVC A, @A + PC**

Operation: MOVC
(PC) ← (PC) + 1
(A) ← ((A) + (PC))

Encoding:

| 1 0 0 0 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 2

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

Function: Move external

Description: The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence the ”X” 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.

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 l/O expansion decoding or 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 instructions, 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 64 Kbyte), 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 addressed via the data pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example: An external 256 byte RAM using multiplexed address/data lines (e.g. an SAB 8155 RAM/I/O/timer) is connected to the SAB 80(c)5XX port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal l/O. Registers 0 and 1 contain $12_H$ and $34_H$. Location $34_H$ of the external RAM holds the value $56_H$. The instruction sequence

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

copies the value $56_H$ into both the accumulator and external RAM location $12_H$.

**MOVX A,@Ri**

Operation: MOVX
(A) ← ((Ri))

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | i |

Bytes: 1

Cycles: 2

**MOVX A,@DPTR**

Operation: MOVX
(A) ← ((DPTR))

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

Bytes: 1

Cycles: 2

**MOVX @Ri,A**

Operation: MOVX
((Ri)) ← (A)

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | i |

Bytes: 1

Cycles: 2

**MOVX @DPTR,A**

Operation: MOVX
((DPTR)) ← (A)

Encoding:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Bytes: 1

Cycles: 2

**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 ($0FF_H$) the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the accumulator holds the value 80 ($50_H$). Register B holds the value 160 ($0A0_H$). The instruction

MUL AB

will give the product 12,800 ($3200_H$), so B is changed to $32_H$ (00110010B) and the accumulator is cleared. The overflow flag is set, carry is cleared.

Operation: MUL

(A7-0)
(B15-8) $\leftarrow$ (A) x (B)

Encoding:

| 1 0 1 0 | 0 1 0 0 |
|---|---|

Bytes: 1

Cycles: 4

**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
```

Operation: NOP

Encoding:

| | |
|---|---|
| 0 0 0 0 | 0 0 0 0 |

Bytes: 1

Cycles: 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 $0C3_H$ (11000011B) and R0 holds $55_H$ (01010101B) then the instruction

ORL A,R0

will leave the accumulator holding the value $0D7_H$ (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**

Operation: ORL
$(A) \leftarrow (A) \vee (Rn)$

Encoding:

| 0 1 0 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**ORL A,direct**

Operation: ORL
(A) ← (A) ∨ (direct)

Encoding:

| 0 1 0 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**ORL A,@Ri**

Operation: ORL
(A) ← (A) ∨ ((Ri))

Encoding:

| 0 1 0 0 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 1

**ORL A,#data**

Operation: ORL
(A) ← (A) ∨ #data

Encoding:

| 0 1 0 0 | 0 1 0 0 | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**ORL direct,A**

Operation: ORL
(direct) ← (direct) ∨ (A)

Encoding:

| 0 1 0 0 | 0 0 1 0 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**ORL direct, #data**

Operation: ORL
(direct) ← (direct) ∨ #data

Encoding:

| 0 1 0 0 | 0 0 1 1 | direct address | immediate data |
|---|---|---|---|

Bytes: 3

Cycles: 2

**ORL C, <src-bit>**

Function: Logical OR for bit variables

Description: Set the carry flag if the Boolean value is a logic 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 P1.0
ORL   C,ACC.7     ; OR carry with the accumulator bit 7
ORL   C,/OV       ; OR carry with the inverse of OV
```

**ORL C,bit**

Operation: ORL
(C) ← (C) ∨ (bit)

Encoding:

| 0 1 1 1 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 2

**ORL C,/bit**

Operation: ORL
(C) ← (C) ∨ ¬ (bit)

Encoding:

| 1 0 1 0 | 0 0 0 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 the transfer to the directly addressed byte indicated. No flags are affected.

Example: The stack pointer originally contains the value $32_H$, and internal RAM locations $30_H$ through $32_H$ contain the values $20_H$, $23_H$, and $01_H$, respectively. The instruction sequence

```
POP     DPH
POP     DPL
```

will leave the stack pointer equal to the value $30_H$ and the data pointer set to $0123_H$. At this point the instruction

```
POP     SP
```

will leave the stack pointer set to $20_H$. Note that in this special case the stack pointer was decremented to $2F_H$ before being loaded with the value popped ($20_H$).

Operation:
POP
(direct) ← ((SP))
(SP) ← (SP) – 1

Encoding:

| 1 1 0 1 | 0 0 0 0 | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 $09_H$. The data pointer holds the value $0123_H$. The instruction sequence

```
PUSH    DPL
PUSH    DPH
```

will leave the stack pointer set to $0B_H$ and store $23_H$ and $01_H$ in internal RAM locations $0A_H$ and $0B_H$, respectively.

Operation: PUSH
(SP) ← (SP) + 1
((SP)) ← (direct)

Encoding:

| 1 1 0 0 | 0 0 0 0 | direct address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 $0B_H$. Internal RAM locations $0A_H$ and $0B_H$ contain the values $23_H$ and $01_H$, respectively. The instruction

RET

will leave the stack pointer equal to the value $09_H$. Program execution will continue at location $0123_H$.

Operation: RET
(PC15-8) ← ((SP))
(SP) ← (SP) – 1
(PC7-0) ← ((SP))
(SP) ← (SP) – 1

Encoding:

| | |
|---|---|
| 0 0 1 0 | 0 0 1 0 |

Bytes: 1

Cycles: 2

**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 is 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 $0B_H$. An interrupt was detected during the instruction ending at location $0122_H$. Internal RAM locations $0A_H$ and $0B_H$ contain the values $23_H$ and $01_H$, respectively. The instruction

RETI

will leave the stack pointer equal to $09_H$ and return program execution to location $0123_H$.

Operation: RETI
(PC15-8) ← ((SP))
(SP) ← (SP) – 1
(PC7-0) ← ((SP))
(SP) ← (SP) – 1

Encoding:

| | |
|---|---|
| 0 0 1 1 | 0 0 1 0 |

Bytes: 1

Cycles: 2

**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 $0C5_H$ (11000101B). The instruction

RL A

leaves the accumulator holding the value $8B_H$ (10001011B) with the carry unaffected.

Operation: RL
$(An + 1) \leftarrow (An)$ n = 0-6
$(A0) \leftarrow (A7)$

Encoding:

| | |
|---|---|
| 0 0 1 0 | 0 0 1 1 |

Bytes: 1

Cycles: 1

**RLC A**

Function: Rotate accumulator left through 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 $0C5_H$ (11000101B), and the carry is zero. The instruction

RLC A

leaves the accumulator holding the value $8A_H$ (10001010B) with the carry set.

Operation: RLC
(An + 1) ← (An) n = 0-6
(A0) ← (C)
(C) ← (A7)

Encoding:

| | |
|---|---|
| 0 0 1 1 | 0 0 1 1 |

Bytes: 1

Cycles: 1

**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 $0C5_H$ (11000101B). The instruction

RR A

leaves the accumulator holding the value $0E2_H$ (11100010B) with the carry unaffected.

Operation: RR
(An) ← (An + 1) n = 0-6
(A7) ← (A0)

Encoding:

| | |
|---|---|
| 0 0 0 0 | 0 0 1 1 |

Bytes: 1

Cycles: 1

**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 $0C5_H$ (11000101B), the carry is zero. The instruction

RRC A

leaves the accumulator holding the value $62_H$ (01100010B) with the carry set.

Operation: RRC
(An) ← (An + 1) n=0-6
(A7) ← (C)
(C) ← (A0)

Encoding:

| | |
|---|---|
| 0 0 0 1 | 0 0 1 1 |

Bytes: 1

Cycles: 1

**SETB <bit>**

Function: Set bit

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

Example: The carry flag is cleared. Output port 1 has been written with the value $34_H$ (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 $35_H$ (00110101B).

**SETB C**

Operation: SETB
(C) ← 1

Encoding:

| 1 1 0 1 | 0 0 1 1 |
|---|---|

Bytes: 1

Cycles: 1

**SETB bit**

Operation: SETB
(bit) ← 1

Encoding:

| 1 1 0 1 | 0 0 1 0 | bit address |
|---|---|---|

Bytes: 2

Cycles: 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 $0123_H$. The instruction

SJMP RELADR

will assemble into location $0100_H$. After the instruction is executed, the PC will contain the value $0123_H$.

**Note:**

Under the above conditions the instruction following SJMP will be at $102_H$. Therefore, the displacement byte of the instruction will be the relative offset ($0123_H$-$0102_H$) = $21_H$. In other words, an SJMP with a displacement of $0FE_H$ would be a one-instruction infinite loop.

Operation: SJMP
(PC) ← (PC) + 2
(PC) ← (PC) + rel

Encoding:

| 1 0 0 0 | 0 0 0 0 | rel. address |
|---|---|---|

Bytes: 2

Cycles: 2

**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 $0C9_H$ (11001001B), register 2 holds $54_H$ (01010100B), and the carry flag is set. The instruction

SUBB A,R2

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

Notice that $0C9_H$ minus $54_H$ is $75_H$. The difference between this and the above result is due to the (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**

Operation: SUBB
(A) ← (A) – (C) – (Rn)

Bytes: 1

Cycles: 1

**SUBB A,direct**

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

Encoding:

| 1 0 0 1 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**SUBB A, @ Ri**

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

Encoding:

| 1 0 0 1 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 1

**SUBB A, #data**

Operation: SUBB
(A) ← (A) – (C) – #data

Encoding:

| 1 0 0 1 | 0 1 0 0 | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**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 $0C5_H$ (11000101B). The instruction

SWAP A

leaves the accumulator holding the value $5C_H$ (01011100B).

Operation: SWAP
(A3-0) $\leftrightarrows$ (A7-4), (A7-4) $\leftarrow$ (A3-0)

Encoding:

| 1 1 0 0 | 0 1 0 0 |
|---|---|

Bytes: 1

Cycles: 1

**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 $20_H$. The accumulator holds the value $3F_H$ (00111111B). Internal RAM location $20_H$ holds the value $75_H$ (01110101B). The instruction

XCH A, @R0

will leave RAM location $20_H$ holding the value $3F_H$ (00111111 B) and $75_H$ (01110101B) in the accumulator.

**XCH A,Rn**

Operation: XCH
(A) ⇆ (Rn)

Encoding:

| 1 1 0 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**XCH A,direct**

Operation: XCH
(A) ⇆ (direct)

Encoding:

| 1 1 0 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**XCH A, @ Ri**

Operation: XCH
(A) ⇆ ((Ri))

Encoding:

| | |
|---|---|
| 1 1 0 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

**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 $20_H$. The accumulator holds the value $36_H$ (00110110B). Internal RAM location $20_H$ holds the value $75_H$ (01110101B). The instruction

XCHD A, @ R0

will leave RAM location $20_H$ holding the value $76_H$ (01110110B) and $35_H$ (00110101B) in the accumulator.

Operation: XCHD
(A3-0) ⇆ ((Ri)3-0)

Encoding:

| | |
|---|---|
| 1 1 0 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

**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 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 $0C3_H$ (11000011B) and register 0 holds $0AA_H$ (10101010B) then the instruction

XRL A,R0

will leave the accumulator holding the value $69_H$ (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**

Operation: XRL2
(A) ← (A) ⩛ (Rn)

Encoding:

| 0 1 1 0 | 1 r r r |
|---|---|

Bytes: 1

Cycles: 1

**XRL A,direct**

Operation: XRL
(A) ← (A) ∀ (direct)

Encoding:

| 0 1 1 0 | 0 1 0 1 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**XRL A, @ Ri**

Operation: XRL
(A) ← (A) ∀ ((Ri))

Encoding:

| 0 1 1 0 | 0 1 1 i |
|---|---|

Bytes: 1

Cycles: 1

**XRL A, #data**

Operation: XRL
(A) ← (A) ∀ #data

Encoding:

| 0 1 1 0 | 0 1 0 0 | immediate data |
|---|---|---|

Bytes: 2

Cycles: 1

**XRL direct,A**

Operation: XRL
(direct) ← (direct) ∀ (A)

Encoding:

| 0 1 1 0 | 0 0 1 0 | direct address |
|---|---|---|

Bytes: 2

Cycles: 1

**XRL direct, #data**

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

Encoding:

| 0 1 1 0 | 0 0 1 1 | direct address | immediate data |
|---|---|---|---|

Bytes: 3

Cycles: 2

### Instruction Set Summary

| Mnemonic | | Description | Byte | Cycle |
|---|---|---|---|---|
| **Arithmetic Operations** | | | | |
| ADD | A,Rn | Add register to accumulator | 1 | 1 |
| ADD | A,direct | Add direct byte to accumulator | 2 | 1 |
| ADD | A, @Ri | Add indirect RAM to accumulator | 1 | 1 |
| ADD | A,#data | Add immediate data to accumulator | 2 | 1 |
| ADDC | A,Rn | Add register to accumulator with carry flag | 1 | 1 |
| ADDC | A,direct | Add direct byte to A with carry flag | 2 | 1 |
| ADDC | A, @Ri | Add indirect RAM to A with carry flag | 1 | 1 |
| ADDC | A, #data | Add immediate data to A with carry flag | 2 | 1 |
| SUBB | A,Rn | Subtract register from A with borrow | 1 | 1 |
| SUBB | A,direct | Subtract direct byte from A with borrow | 2 | 1 |
| SUBB | A,@Ri | Subtract indirect RAM from A with borrow | 1 | 1 |
| SUBB | A,#data | Subtract immediate data from A with borrow | 2 | 1 |
| INC | A | Increment accumulator | 1 | 1 |
| INC | Rn | Increment register | 1 | 1 |
| INC | direct | Increment direct byte | 2 | 1 |
| INC | @Ri | Increment indirect RAM | 1 | 1 |
| DEC | A | Decrement accumulator | 1 | 1 |
| DEC | Rn | Decrement register | 1 | 1 |
| DEC | direct | Decrement direct byte | 2 | 1 |
| DEC | @Ri | Decrement indirect RAM | 1 | 1 |
| INC | DPTR | Increment data pointer | 1 | 2 |
| MUL | AB | Multiply A and B | 1 | 4 |
| DIV | AB | Divide A by B | 1 | 4 |
| DA | A | Decimal adjust accumulator | 1 | 1 |

**Instruction Set Summary** (cont'd)

| Mnemonic | | Description | Byte | Cycle |
|---|---|---|---|---|
| **Logic Operations** | | | | |
| ANL | A,Rn | AND register to accumulator | 1 | 1 |
| ANL | A,direct | AND direct byte to accumulator | 2 | 1 |
| ANL | A,@Ri | AND indirect RAM to accumulator | 1 | 1 |
| ANL | A,#data | AND immediate data to accumulator | 2 | 1 |
| ANL | direct,A | AND accumulator to direct byte | 2 | 1 |
| ANL | direct,#data | AND immediate data to direct byte | 3 | 2 |
| ORL | A,Rn | OR register to accumulator | 1 | 1 |
| ORL | A,direct | OR direct byte to accumulator | 2 | 1 |
| ORL | A,@Ri | OR indirect RAM to accumulator | 1 | 1 |
| ORL | A,#data | OR immediate data to accumulator | 2 | 1 |
| ORL | direct,A | OR accumulator to direct byte | 2 | 1 |
| ORL | direct,#data | OR immediate data to direct byte | 3 | 2 |
| XRL | A,Rn | Exclusive OR register to accumulator | 1 | 1 |
| XRL | A direct | Exclusive OR direct byte to accumulator | 2 | 1 |
| XRL | A,@Ri | Exclusive OR indirect RAM to accumulator | 1 | 1 |
| XRL | A,#data | Exclusive OR immediate data to accumulator | 2 | 1 |
| XRL | direct,A | Exclusive OR accumulator to direct byte | 2 | 1 |
| XRL | direct,#data | Exclusive OR immediate data to direct byte | 3 | 2 |
| CLR | A | Clear accumulator | 1 | 1 |
| CPL | A | Complement accumulator | 1 | 1 |
| RL | A | Rotate accumulator left | 1 | 1 |
| RLC | A | Rotate accumulator left through carry | 1 | 1 |
| RR | A | Rotate accumulator right | 1 | 1 |
| RRC | A | Rotate accumulator right through carry | 1 | 1 |
| SWAP | A | Swap nibbles within the accumulator | 1 | 1 |

**Instruction Set Summary** (cont'd)

| Mnemonic | | Description | Byte | Cycle |
|---|---|---|---|---|
| **Data Transfer** | | | | |
| MOV | A,Rn | Move register to accumulator | 1 | 1 |
| MOV | A,direct *) | Move direct byte to accumulator | 2 | 1 |
| MOV | A,@Ri | Move indirect RAM to accumulator | 1 | 1 |
| MOV | A,#data | Move immediate data to accumulator | 2 | 1 |
| MOV | Rn,A | Move accumulator to register | 1 | 1 |
| MOV | Rn,direct | Move direct byte to register | 2 | 2 |
| MOV | Rn,#data | Move immediate data to register | 2 | 1 |
| MOV | direct,A | Move accumulator to direct byte | 2 | 1 |
| MOV | direct,Rn | Move register to direct byte | 2 | 2 |
| MOV | direct,direct | Move direct byte to direct byte | 3 | 2 |
| MOV | direct,@Ri | Move indirect RAM to direct byte | 2 | 2 |
| MOV | direct,#data | Move immediate data to direct byte | 3 | 2 |
| MOV | @Ri,A | Move accumulator to indirect RAM | 1 | 1 |
| MOV | @Ri,direct | Move direct byte to indirect RAM | 2 | 2 |
| MOV | @Ri, #data | Move immediate data to indirect RAM | 2 | 1 |
| MOV | DPTR, #data16 | Load data pointer with a 16-bit constant | 3 | 2 |
| MOVC | A,@A + DPTR | Move code byte relative to DPTR to accumulator | 1 | 2 |
| MOVC | A,@A + PC | Move code byte relative to PC to accumulator | 1 | 2 |
| MOVX | A,@Ri | Move external RAM (8-bit addr.) to A | 1 | 2 |
| MOVX | A,@DPTR | Move external RAM (16-bit addr.) to A | 1 | 2 |
| MOVX | @Ri,A | Move A to external RAM (8-bit addr.) | 1 | 2 |
| MOVX | @DPTR,A | Move A to external RAM (16-bit addr.) | 1 | 2 |
| PUSH | direct | Push direct byte onto stack | 2 | 2 |
| POP | direct | Pop direct byte from stack | 2 | 2 |
| XCH | A,Rn | Exchange register with accumulator | 1 | 1 |
| XCH | A,direct | Exchange direct byte with accumulator | 2 | 1 |
| XCH | A,@Ri | Exchange indirect RAM with accumulator | 1 | 1 |
| XCHD | A,@Ri | Exchange low-order nibble indir. RAM with A | 1 | 1 |

*) MOV A,ACC is not a valid instruction

**Instruction Set Summary** (cont'd)

| Mnemonic | | Description | Byte | Cycle |
|---|---|---|---|---|
| **Boolean Variable Manipulation** | | | | |
| CLR | C | Clear carry flag | 1 | 1 |
| CLR | bit | Clear direct bit | 2 | 1 |
| SETB | C | Set carry flag | 1 | 1 |
| SETB | bit | Set direct bit | 2 | 1 |
| CPL | C | Complement carry flag | 1 | 1 |
| CPL | bit | Complement direct bit | 2 | 1 |
| ANL | C,bit | AND direct bit to carry flag | 2 | 2 |
| ANL | C,/bit | AND complement of direct bit to carry | 2 | 2 |
| ORL | C,bit | OR direct bit to carry flag | 2 | 2 |
| ORL | C,/bit | OR complement of direct bit to carry | 2 | 2 |
| MOV | C,bit | Move direct bit to carry flag | 2 | 1 |
| MOV | bit,C | Move carry flag to direct bit | 2 | 2 |
| **Program and Machine Control** | | | | |
| ACALL | addr11 | Absolute subroutine call | 2 | 2 |
| LCALL | addr16 | Long subroutine call | 3 | 2 |
| RET | | Return from subroutine | 1 | 2 |
| RETI | | Return from interrupt | 1 | 2 |
| AJMP | addr11 | Absolute jump | 2 | 2 |
| LJMP | addr16 | Long iump | 3 | 2 |
| SJMP | rel | Short jump (relative addr.) | 2 | 2 |
| JMP | @A + DPTR | Jump indirect relative to the DPTR | 1 | 2 |
| JZ | rel | Jump if accumulator is zero | 2 | 2 |
| JNZ | rel | Jump if accumulator is not zero | 2 | 2 |
| JC | rel | Jump if carry flag is set | 2 | 2 |
| JNC | rel | Jump if carry flag is not set | 2 | 2 |
| JB | bit,rel | Jump if direct bit is set | 3 | 2 |
| JNB | bit,rel | Jump if direct bit is not set | 3 | 2 |
| JBC | bit,rel | Jump if direct bit is set and clear bit | 3 | 2 |
| CJNE | A,direct,rel | Compare direct byte to A and jump if not equal | 3 | 2 |

**Instruction Set Summary** (cont'd)

| Mnemonic | Description | Byte | Cycle |
|---|---|---|---|
| **Program and Machine Control** (cont'd) | | | |
| CJNE A,#data,rel | Compare immediate to A and jump if not equal | 3 | 2 |
| CJNE Rn,#data rel | Compare immed. to reg. and jump if not equal | 3 | 2 |
| CJNE @Ri,#data,rel | Compare immed. to ind. and jump if not equal | 3 | 2 |
| DJNZ Rn,rel | Decrement register and jump if not zero | 2 | 2 |
| DJNZ direct,rel | Decrement direct byte and jump if not zero | 3 | 2 |
| NOP | No operation | 1 | 1 |