Addressing Modes

Indirect Addressing Mode

```
MOV A, 30H  ] → MOV R0, #30H
              MOV A, @R0
```

Syntax

```
MOV A, @Rp
MOV @Rp, A
MOV @Rp, #n
MOV @Rp, addr
MOV addr, @Rp
MOV addr, @Ri
```

// ALP to copy the data from 70h ~~73h~~ to R0 to R4

```
MOV 70h, #11h
MOV R0, 70h
MOV R1, 70h
MOV R2, 70h
MOV R3, 70h
MOV R4, 70h
```

//ALP to copy data from R0 to 30h to 33h

```
MOV R0, #30h
MOV R1, #31h
MOV 31h, @R0
MOV A, @R1
MOV @R1, 31h
MOV @R0, A
```

→ Direct Addressing Mode

Mov Rx, addr
Mov A, addr
Mov addr, Rx
Mov addr, #n
Mov addr1, addr2
Mov addr, A

→ Immediate Addressing Mode

The operand is present in instruction

Mov A, #n
Mov Rx, #n
Mov DPTR, #nn
↳ 16 bit

→ Register Addressing Mode

Mov A, #30H
Mov R0, A
Mov R1, A
}
Mov Rn, A

Indirect Addressing Mode

$R_p \leftarrow \begin{matrix} R_0 \\ R_1 \end{matrix}$

Mov R_1, #40h $R_1 \leftarrow 40h$
Mov A, #33h $A \leftarrow 33h$
Mov @R1, A

Mov A, 30h → Mov R_0, #30h
Mov A, @R_0

Syntax

Mov A, @R_p
Mov @R_p, A
Mov @R_p, #n
Mov @R_p, addr
Mov addr, @R_p
Mov @R_p, @R_p

// A-lp to copy data at 70h, to R_0 to R_3

```
MOV 70h, #30h
MOV R0, 70h
MOV R1, 70h
MOV R2, 70h
MOV R3, 70h
```

// Alp to copy data from R_0 to 30h to 33h.

```
MOV 30h, R0
MOV 31h, R0
MOV 32h, R0
MOV 33h, R0
END
```

// Alp to load immediate value in register

```
MOV R1, #30h
MOV R2, #30h
END
```

// Alp using indirect Addressing mode & exchange the contents of 30h to 40h

direct →

```
// MOV 31h, 30h
   MOV 30h, 40h
   MOV 40h, 31h
```

Indirect Addressing Mode → //

```
Mov R0, #30h
Mov R1, #40h
Mov A, @R0
Mov B, @R1
Mov @R0, B
Mov @R1, A
```

OR

```
Mov R0, #30h
Mov A, @R0
Mov @R0, 40h
Mov 40h, A
END
```

→ Transfer 5 bytes of data from 30h to 40h onwards using direct Mode

```
Mov A, 30h
Mov 30h, 40h
Mov 40h, A
Mov A, 31h
//
```

Date: / /

Byte level → Logical Operations

AND

ANL A, #n n → Any value (Hexadecimal)
ANL A, Rn
ANL A, addr
ANL A, @Rp
ANL addr, A addr → Internal RAM Address
ANL addr, #n

eg:- MOV A, #54H 54H → 0101 0100
MOV R0, #32H 32H → 0011 0010 &
ANL A, R0 0001 0000
A ← 10H

ORL

ORL A, #n
ORL A, Rn
ORL A, addr
ORL A, @Rp
ORL addr, A
ORL addr, #n

→ eg:- MOV A, #54H 0101 0100
MOV R1, #32H 0001 0010
ORL A, R0 0101 0110
A ← 56H

XOR :- operations

XRL A, #n XRL A, @Rp
XRL A, addr XRL addr, A
XRL A, Rn XRL addr, #n

NOT operation

Cpl A

clear operation

CLR A

→ ALP program that will Invert/Compliment the bit of R_6 of bank 0

```
Mov R6, #10H
Mov A, R6
Cpl A
Mov R6, A
END
```

XOR anything with FFh, we will get its compliment

→ XRL 06H, #0FFH

→ The contents of Port 1 & Port 2, put result into 0100H (External RAM)

```
Mov A, 90H
ORL A, 0A0H
Mov DPTR, #0100H
MovX @DPTR, A
END
```

Date: / /

Nibbles

→ ALP to swap ~~Neighbours~~ of R_0 & R_1 so that low nibble of R_0 swaps with high nibble of R_1 & high nibble of R_0 swaps with low nibble of R_1

SWAP A

A → 33

SWAP 33

=> 23

```
MOV R0, #011H
MOV A, R0
SWAP A, MOV R0, A
MOV R1, # 231H
MOV A, R1
SWAP A
XCH, A, #00H
MOV R1, A
END
```

(OR)

```
MOV R0, #10H
MOV R1, #20H
MOV A, R0
SWAP A
XCH A, R1
SWAP A
MOV R0, A
END
```

→ Store Most Significant Nibble of A in both nibbles of R5.

```
MOV   A, #10H
ANL   A, #0F0H
MOV   R0, A
SWAP  A
ORL   A, 00H
MOV   R5, A
END
```

Arithmetic Operations

Increment operator (INC dest)

INC A INC @Rp INC addr

INC Rn INC DPTR

Decrement operator (DEC dest)

DEC A DEC @Rp DEC addr

DEC Rn DEC DPTR X

Addition Instruction:

ADD A, #n ADDC A, #n

ADD A, Rn ADDC A, Rn

ADD A, addr ADDC A, addr

ADD A, @Rp ADDC A, @Rp

// ALP to add bytes in RAM location 34h & 35h & store result in R5 LSB, in R6 MSB

```
MOV A, 34h
ADD A, 35h
MOV R5, A
CLR A
RLC A
MOV R6, A
```

OR

```
MOV A, 34
ADD A, 35h
MOV R5, A
CLR A
ADDC A, #00h
MOV R6, A
END
```

11h 0001 0001
45h 0100 0101
0101 0110
56h

F1h 1111 0001
F5h 1111 0101
1 1110 0110
1 E 6h

carry=1 low byte = e6
high byte = 1

→ Add the bytes in register R3 & R4, put result in RAM location 4Ah LSB, 4Bh MSB.

```
Mov R3, #10h
Mov R4, #11h
Mov A, R3
ADD A, R4
Mov 4Ah, A
CLR A
RLC A
Mov 4Bh, A
END
```

→ Substraction

SUBB A, #n SUBB A, @Rp
SUBB A, Rn
SUBB A, addr

Date: / /

→ MULTIPLICATION

MUL A, B → MSbyte
↓
LSByte

→ Division

Div AB := A/B
quotient → A
remainder → B

Substract the no° 54h from RAM location 17h & store result in R3 LSB, R4 MSB

→ Substract Contents of R1 from R0, Put back to R7

2) MOV R0, #10h
MOV R1, #20h
SUBB: MOV A, R0
SUBB A, R1
MOV R7, A

13h 0001 0011
-12h 0001 0010
0000 0001
01h

1) MOV A, #17h
SUBB A, #54h
MOV R_3, A
CLR A
RLC A
MOV R_4, A
END

→ ALP to perform 16 bit Addition stored in R_3 & R_4 with RAM location 44h & 45h. Put result in 30h MSB, 31h Middle byte & 32h LSB.

R_3	R_4	11	10
44h	45h	12	13
30h		00	23 23h
↓	↓		
31h	32h		

MOV A, R_4
ADD A, 45h
MOV 32h, A
MOV A, R_3
ADDC A, 44h
MOV 31h, A
CLR A
ADDC A, #00h
MOV 30h, A
END

→ Multiply data from 22h by ram location 15h, put result in 19h LSB, 18h MSB

```
MOV A, 22h
MOV B, 15h
MUL AB
MOV 19h, A
MOV 18h, B
END //
```

→ Square the contents of R5, result into R0 MSB
R1 LSB

Divide the data in RAM 3Eh by number 12h, put quotient in R4, remainder R5

```
MOV A, R5 | MOV B, R5
MUL AB
MOV R0, B
MOV R1, A
END
```

```
MOV A, 3Eh
MOV B, #12h
DIV AB
MOV R4, A
MOV R5, B
END
```

Lab: 5

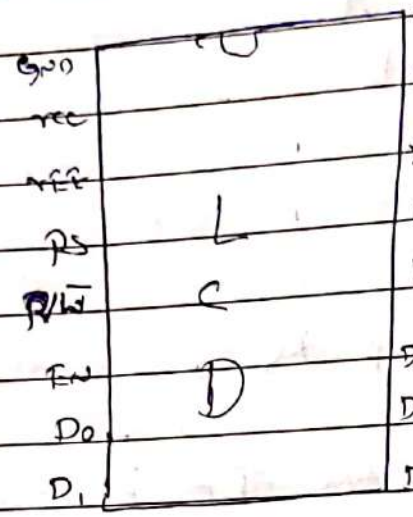
14/2/2020

(Interface LCD with 8051)

16x2 LCD Module is very common type

→ 16 characters, 2 rows

pixel dimension is 5x7 or 5x8 dot

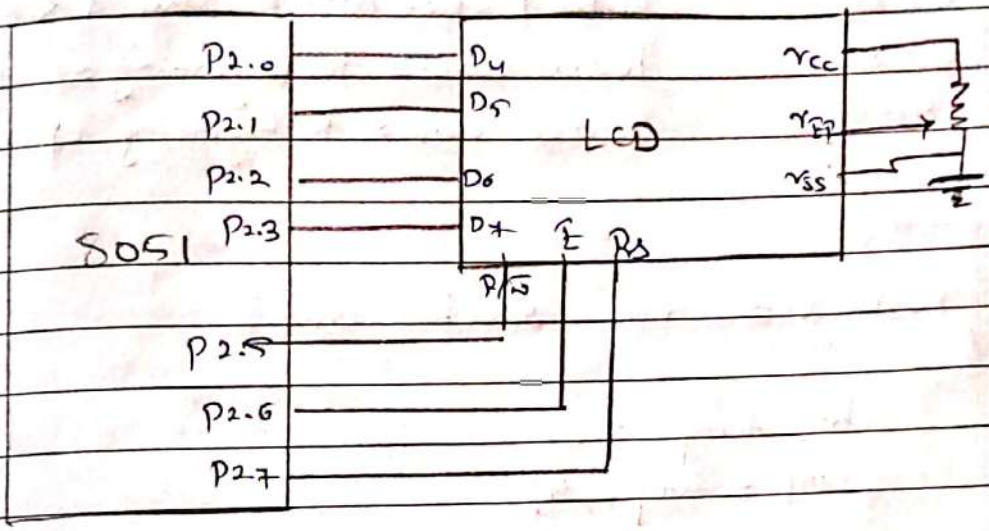


Register Select (RS)

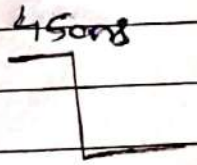
LCD Pin No.	Signal Name	Port line
1	GND	GND
2	Vcc	Vcc
3	VEE (LCD Contrast)	Rq (Volt potentio
4	RS Line	P2.7
5	R/W Line	P2.5
6	EN Line	P2.6
7	Data D0	NC
8	Data D1	NC
9	Data D2	NC
10	Data D3	NC
11	Data D4	P2.0
12	Data D5	P2.1
13	Data D6	P2.2
14	Data D7	P2.3
15	Vcc	

LCD Interfacing

RS → 0 for sending Command to LCD
RS = 1 for sending data to LCD
R/W => 1 for reading from LCD
R/W => 0 for writing to LCD



→ E → enable pin



→ LCD Commands

Command	Instruction
0x01	clear display screen
0x02	return home
0x04	decrement cursor (shift cursor left)
0x06	increment cursor (shift cursor right)
0x07	shift display left
0x08	Display off, cursor off

0x0F	4 bit function set 2 lines & 5x7 dots
0x28	Force
0x80	
0xC0	

0x0F	Display ON, cursor blinking
0x28	4 bit function set 2 lines & 5x7 dots
0x80	Force cursor to beginning of 1st line
0xC0	Force cursor to beginning of 2nd line

```
void lcd_cmd_wr(char cmd)
{
    char temp = cmd;
    temp = temp >> 4;
    temp = temp | 0x40;
    P2 = temp;
    delay(450);
    temp = temp & 0x0F;
    P2 = temp;
    delay(100);
    temp = cmd;
    temp = temp & 0x0F;
    temp = temp | 0x40;
    P2 = temp;
    delay(450);
    temp = temp & 0x0F;
    P2 = temp;
    delay(100);
}
```

Date: / /

```
void lcd_data_wr(char data)
{
    char temp=data;
    temp = temp>>4;
    temp = temp|0x20;
    P2 = temp;
    delay(450);
    temp = temp & 0x8F;
    P2 = temp
    delay(100);
    temp = data;
    temp = temp & 0x0F;
    temp = temp|0x20;
    P2 = temp;
    delay(450);
    temp = temp & 0x8F;
    P2 = temp;
    delay(100);
}

main()
{
    lcd_cond_wr(0x06);
    lcd_cond_wr(0x28);
    lcd_cond_wr(0x0F);
    lcd_cond_wr(0x01);
    lcd_cond_wr(0x80);
    char a[]="ESCIV";
    for(int i=0; i<strlen(a); i++)
    {
        lcd_data_wr(a[i]);
```

MOVC Syntax

MOVC A, @pc + A
MOVC A, @A + DPTR

// ALP to copy contents of external code Memory 0040H to TF register.

CLR A / MOV #00H
MOV DPTR, #0040H
MOVC A, @DPTR + A
MOV TF A / MOV 0A8H, A
END

// Copy the internal code byte at 0300H to external RAM memory 0300H.

CLR A
MOV DPTR, #0300H
MOVC A, @DPTR + A. //gets code byte
MOVX @DPTR, A //writes into external RAM.
END.

// ALP to copy program bytes 0100H to 0102H, to internal RAM to the location 20H to 22H

CLR A
MOV DPTR, #0100H
MOVC A, @A + DPTR
MOV 030H, A
CLR A

```
MOV DPTR, #010H
MOVC A, @A+DPTR
MOV 021H, A
CLR A
MOV DPTR, #0100H
MOVC A, @A+DPTR
MOV 022H, A.
```

Or

```
MOV DPTR, #0100H
MOV A, #00H
MOVC A, @A+DPTR
MOV 20H, A
MOV A, #01H
MOVC A, @A+DPTR
MOV 21H, A
MOV A, #02H
MOVC A, @A+DPTR
MOV 22H, A
```

PUSH & POP

Push addr

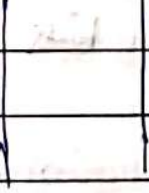
Pop addr

→ push 30H

Contents of 30H will be stored in top's location

→ pop 10H

top's data will be stored in address location of 10H.



increment & then push data

Pop data & then decrement

// Swap contents R_6 & R_7 using push & pop

```
PUSH  R6 (xx)
PUSH  R7 (yy)
pop   R6 (yy)
pop   R7 (xx)
END.
```

30/1/2020

// Logical Instructions

Bit level logical Instructions

AND operation	ANL C, bit ANL C, /bit	MOV C, bit MOV bit, C
OR operation	ORL C, bit ORL C, /bit	CPL C | complement CPL bit
	CLR C | reset CLR bit	SETB bit | sets the value to 1

eg:- Write an Alp to clear bit 3 of RAM location 30H. without affecting other bits

```
MOV  20H, 30H
CLR  03H  // bit addressable
MOV  30H, 20H
END
```

SETB C
SETB b
} Test the values to 1

// ALp to Move bit 6 of R_0 to bit 3 of Port 0

```
Mov 26h. R6
MOV C, 06H
MOV 93H, C
END
```

→ // Sequence of instructions to clear register R_3 of bank 1

```
CPL D3,
MOV R3, #00H

SETB 0D0h.3h } selected the Bank 1
CLR 0D0.4h
MOV R3, #00H
```

5/2/2020

Rotate Instructions

RR A	Rotate Right
RRC A	Rotate Right with Carry
RLC A	
RL A	

Ex:- 80h

RL

1000 0000

0000 0001

01h

RR

1000 0000

0100 0000

40h

C=0 1000 0000

RLC 1 0000 0000

00h

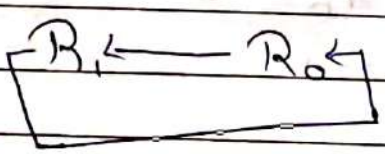
RRC C=0 1000 0000

0 0100 0000

RLC A will double the value (× by 2)

RRC A will half the value (÷ by 2)

To rotate R_0 & R_1 as 16 bit register & rotate 1 place to left bit 7 of R_0 becomes bit 0 of R_1 & bit 7 of R_1 becomes bit 0 of R_0




```
MOV A, R0
RLC A
MOV R0, A
MOV A, R1
RLC A
MOV R1, A
CLR A
RLC A
ORL 00h, A
```

&

```
MOV 20h, R0
MOV 00h, C
MOV R0, 20h
```

→ // ALP to rotate DPTR 1 place to the right
DPTR (16 bit)

```
MOV A, DPL (82h)
RRC A
MOV DPL, A
MOV A, DPH (83h)
RRC A
MOV DPH, A
MOV 20h, DPL
MOV 07h, C
MOV DPL, 20h
```

→ // ALP shift B register content 1 place to the left if bit 0 is zero

0100 0010

1000 0100

```
MOV A, B
RLC A
MOV B, A
MOV 20h, B
MOV B, 20h
```

```
CLR C
MOV A, B
RLC A
MOV B, A
END
```

→ // ALP swaps every even numbered bit of R3 in bank 0 with every odd numbered bit to its left.

```
MOV A, R3
RR  A
ANL A, #55H
MOV 30h, A      [odd bit → even bit → 30h]
MOV A, R3
RL  A           (even bit → odd bit)
ANL A, #AAh
ORL A, 30h
MOV R3, A
END
```

9/2/2020

```
10h     0001 0000
99h     1001 1001
-------------------
109h    1010 1001
        A    9
        +6 ← Add 6 circuit
        -----------
        10 9
```

No° should be within 00–99 → for B.Decimal

(LSB) (MSB)

ALP to add R3, R4 store result in 30h & 31h. numbers are in BCD format.

```
MOV A, R3
ADD A, R4          DA → decimal adjust after Addition
DA  A
MOV 30h, A
CLR A   ┐          CLR A
ADDC A, #00h ┘     RLC A
MOV 31h, A
```

// ALp to add Numbers found in 30h, 31h, 32h, Store result in R_3 (LSB) R_4 (Mbyte) R_5 (MSB)

```
MOV A, 30h
ADD A, 31h
MOV R3, A
CLR A
ADDC A, #00h
MOV R4, A
MOV A, 32h
ADD A, R3
MOV R3, A
MOV A, R4
ADDC A, #00h
MOV R4, A
END
```

```
MOV A, 30h
Add A, 31h
Addc A, 32h
MOV R3, A
CLR A
RLC A
MOV R4, A
END
```

```
MOV A, 30h
Add A, 31h, MOV R3, A
CLR A
RLC A
MOV R2, A (Carry)
ADDC
MOV A, 32h
ADDC R4, R3
MOV R3, A (LSB)
CLR A
RLC A
Add R2, A
MOV R5, A
END
```

→ 16 bit × 8 bit Multiplication.

R_1 R_0 → 16 bit No.

R_2 → 8 bit No.

Store in 30h, 31h, 32h

12/2/2020

16 bit x 8 bit multiplication

```
MOV A, R0
MOV 0F0h, R2
MUL AB
MOV 31h, A → LSB.
MOV A, B
MOV 0F0h, R1
MUL AB
[MOV 32h, B → MSB.] ×
MOV
ADD A, 31h
MOV 31h, A → Middle byte
CLR A
ADDC A, 0F0h // A+B+ carry
MOV 32h, A // MSB
END.
```

? Subtract byte in the External Ram 02cdh from the internal Ram 19h, put result into external memory address 00c1 MSB

```
MOV DPTR, #02cdh
MOVX A, @DPTR
MOV R0, A
MOV A, 19h
SUBB A, R0
MOV DPTR, #00c0
MOV @DPTR, A
```

```
MOV A, 02cdh
SUBB A, 19h
MOV DPTR, #00c0
MOV @DPTR, A
INC DPTR
```

CLR A
RLC A
INC DPTR
MOV @DPTR, A
END

Jump

Bit

JC radd (Jump on carry)
JNC radd (Jump on no carry)
JB b, radds // if b=1 then jump
JNB b, radds // if b=0, then jump
JBC b, radds // if b=0, then jump, but before going it will clear the bit=0

Flags are not Affected.

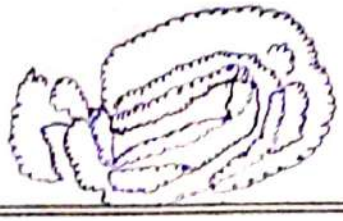
Then C means clear

Byte

CJNE A, add, radd Compare A to add, if not equal, then go
CJNE A, #n, radd Compare A to n, if not equal => Jump
CJNE Rx, #n, radd
CJNE @Rp, #n, radd

A>30h, C=0
A<30h, C=1

// Alp to put a random no° in R3 & increment it until it equals a no° E lb



Mov R_3, #11h
bk, INC R_3
CJNE R_3, #0E1h, bk

//ALp, to put a random no° in R_3 & decrement it until it becomes E1h.

Mov R_3, #11h
bk DEC R_3
CJNE R_3, #0E1h, bk

//ALp to count no° of 1's in numbers in Register B, put the Count in R_5

Mov R5, #00h
Mov B, #50h
Mov R_3, #8h
Mov A, B
bk: RLC A
JNC Next
INC R5
Next DJNZ R_3, bk
END

//ALp Count no° of Zeroes in R_3 & put in R5
Mov R5, #00h
Mov R_3, #8h
Mov A, R_3
bk, RLC A
JC Next
INC R5
Next DJNZ R_3, bk
END

// Alp to exchange bank 0 content with bank 2.

// Alp to find biggest no°. among R1 R2 R3 R4 & put in R6

Bank 0

R0
R1
:
R4

One register to count the values
R0 - base address

Mov 0D0h, #08h // selects bank 1
Setbit 0D0.3
CLR 0D0.4

0D0h - address of PSW

CY	AC	F	RS1	RS0	OV	-	P

Mov 0D0h, #08h
Mov R3, #3
Mov R0, #00h
Mov A, @R0 // A ← 6
back: inc R0 // R0 ← 01
Mov 30h, @R0 // 30h ← 5
CJNE A, 30h, L1
SJMP L2
L1: JNC L2
Mov A, 30h
L2: DJNZ R3, back
END

bank 1
PSW → 0D0h

R0 - 6
R1 - 5
R2 - 13
R3 - 2

CJNE,
6-5 : C=0
1st < 2nd : C=1

biggest element in A

or

L1: JC L2
DJNZ R3, back
L2: Mov A, 30h
END

// ALP to find smallest no. (Among array) of 20 elements stored in Memory location

Not needed ← MOV 0D0h, #40h
MOV R3, #13h (#19) (20 elements ⇒ 19 Comparis
MOV R0, #40h
MOV A, @R0
back INC R0
MOV 30h, @R0
CJNE A, 30h, L1
SJMP L2
L1: JC L2
MOV A, 30h
L2: DJNZ R3, back
END

L1: JNC L2
~~MOV DJNZ R3, back~~
L2: ~~MOV A, 30h~~
~~END~~

// exchange bank 0 content with bank 3

MOV 0D0h, #08h //bank 1 (using bank of 2
MOV R0, #00h //bank 0 starting addr (use diff bank)
MOV R1, #10h //bank 2 starting address
MOV R3, #08h // 8 iteration
back: MOV A, @R0
XCH A, @R1
MOV @R0, A
INC R0
INC R1
DJNZ R3, back
END

Date: / /

// ALp to check whether No. in R5 is even or odd. if even store in FFh, else store 00h.

```
MOV A, R5
MOV B, #02h
DIV AB          A → quotient  B → remainder
```

```
MOV A, R5
RRC A           //LSB is in Carry
JC L2
MOV P0, #FFh
SJMP last       (unconditional jump)
L2: MOV P0, #00h
last:
END
```

SJMP → short
LJMP → long
AJMP → Absolute