21/8/20

Module - 2

Addressing modes of 8086 : accessing of operand

I Addressing modes for Data:

1. Immediate addressing modes (data in instruction)

eg: mov CL, 34H CL ← 34H 8bit
mov CX, 2000H CX ← 2000H 16bit

2. Register addressing mode (data in register)

eg: mov CL, BL CL ← BL 8bit.
mov CX, BX
INC BX

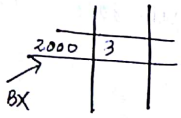
3. Direct addressing mode (address in instruction)

eg: mov CL [2000H] CL ← DS : [2000H] data in 2000.
mov CX [2000H]

4. Indirect addressing mode (address in register)

1. Register indirect (add ← Reg)

eg: mov CL, [BX]
mov CX, [BX]



2. Register Relative. (add ← reg + displacement)

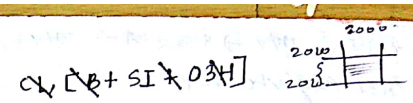
eg: mov CL, [BX + 03H]
mov CL, [BP + 05H]

3. Base indexed (add ← base reg + Index Reg)

eg: mov CL, [BX + SI] (DS)
mov CL, [BP + SI] (SS)

to get a set of instructions

eg: mov CL, [BP + SI + 03H]



4. Base relative plus indexed (add ← base reg + Index + displacement)

eg: mov CL, [BP + SI + 03H]

5. Implied (addressing mode) Unchanged.

Operand is implied

eg: STC, ~~CLC~~ Set Carry
CLC Clear Carry

22/8/2017

Addressing modes for program.

Branch instructions are included

Intra Segment (Near jump) - If jump is in same segment.

```
Code I
L2: jump L1
L1 .....
    Jmp L3
    Code end
```

Inter segment - Jump Blw two different segment ~~Same~~

Addressing modes for Stack.

PUSH BX : - Store POP BX - Remove.

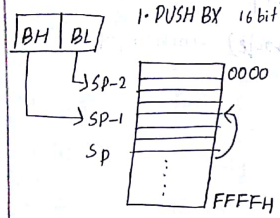
Instruction Set 8086

1. Data Transfer instruction
   - MOV

Mov dest, Src.

- register → r m/y → r r → m/y
- Same size register type

eg: MOV CL, 12H , MOV CX, 1234H,

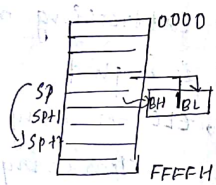


1. PUSH BX 16 bit

SS: [SP-1] ← BH
SS: [SP-2] ← BL
SP ← SP-2.

*SP → decrement by 2.

2. POP BX



SS: [SP] ← BL.
[SP+1] ← BH
SP ← SP+2.

*SP increment by 2.

- XCHG

eg: XCHG AX, BX

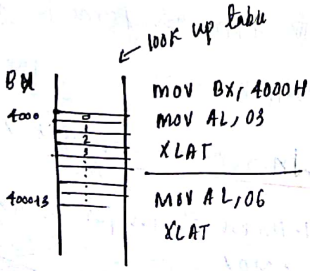
- IN : deskination , SRC
  (Reg) (M/O address)
- OUT : deskination , SRC
  (M/O address) Reg

IN AL, 80H → port add
OUT AL, 20

- XLAT

AL ← DS: [BX + AL]

23/8/17



LEA → load effective address
LDS → load DS
LES → load ES
LAHF → load AH from flag
SAHF → Store AH to flag
PUSHF → Push flag.

- 7-Segment table
- ASCII conversion

2. Arithmetic Instructions :-

1. ADD
   ADD AX, BX
2. ADC — ADC AX, BX — Add with carry AX + BX + CF
3. SUB — SUB AX, BX
4. SBB — SBB AX, BX
5. MUL — mul BX, mul BL [AX ← AL × BL]
6. ~~DIV~~ — Here result is stored in AX
   AX * BX → DX AX
6. DIV — DIV BL (16 ÷ 8) AX / BL → AH (Reminder) AL (Quotient)
   DIV BX (32 ÷ 16) DX·AX / BX → DX (R) AX (Q)

1. Convert Byte to word → CBW:-

Byte - 1 memory locatn, word - 2 memory locatn.

eg: AL

1000 0011

AH contains MSB

AH | AL

11111111 10000011

In the case of AX → DX : MSB — 000000

AX : 0100...

00010

2. Convert word to double.

AX

010....

DX | AX

0000 010....

3. NEG BL BL 10001

2's complement

01111

4. COMP BL, CL

Performs substraction (BL-CL)

Here flag is used for result.

If equal flag = 1

5. Decimal adjust accumulator (DAA)

eg:1 AL = 53, CL = 29

ADD AL, CL ; AL ← AL + CL

AL ← 53 + 29

AL ← 7C

DAA ; AL ← 7C + 06 (as C > 9)

AL ← 82

eg:2 AL = 73, CL = 29

AL ← AL + CL

AL ← 73 + 29

AL ← 9C

DAA AL ← 02 CF = 1

AL = 73

29

9C

9C + 6 = A2 + 60

CF = 102

1. If $D_3 - D_0 > 9$, AF is set

add 06H to AL

2. If $D_7 - D_4 > 9$ CF is set

add 60H to AL

6. AAA ASCII adjust after addition.

mov AH, 0

mov AL, 6 BCD 6 in AL.

ADD AL, 05 add BCD 5 to digit in AC

AAA, AH = 1, AC = 1 → represent BCD 11

3. LOGICAL INSTRUCTIONS

1. AND

AND Operand 1, Operand 2

Operand 1 : 0101
Operand 2 : 0011
0001

eg: BL → 0011 1010
AND BL, 0FH
BL ← 0000 1010.

2. OR
3. XOR
4. SHL
5. SHR
6. ROL
7. ROR.

CF 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 Operand
1 0 1 0 1 1 0 0 1 0 1 0 0 1 0 1 0 SHL Res
Inserted
SHL Result 2nd

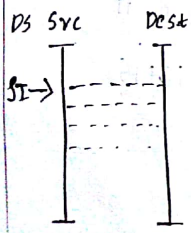
SHL - MSB moved to carry bit and LSB is inserted as zero.

SHR - LSB moved to carry flag and MSB is inserted as zero.

25/8/17

String manipulation Instructions.

- movs
- a set of instructions can be moved.
- src is always DS and dest is ES then only movement is possible.



I MOVS
ES[DI] ← DS:[SI]

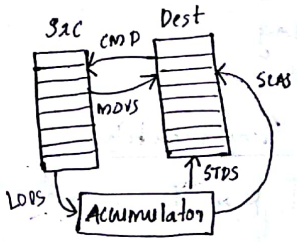
1. MOVSB (8bit)
CLD / STD
DI ← DI ± 1 automatically increment upto count limit
CX ← CX − 1

2. MOVSW 2 m/y locat^{n}
ES:[DI], ES:[DI+1] ← DS:[SI], DS:[SI+1]
SI ← SI ± 2
DI ← DI ± 2

3. LODSB
AL ← DS:[SI]
SI ← SI ± 1
LODSW
AL ← DS[SI], DS:[SI±1]
SI ← SI ± 2

3. STOSB:- ES:[DI] ← AL
DI ← DI ± 1

4. STOSW:



CMPSB

REPE DS:[SI] with ES:[DI]

SI ← SI ± 1

REPNE DI ← DI ± 1

SCASB

REP - Repeat and REPNE : Repeat not equal

Conditional Branch Instructions:

JE	Unconditional
JNZ	JMP
JZ	CALL
JG	LOOP
JBE	Cmp, ax, bx

30/8/17

Flag manipulation:

CLC - Clear carry flag

CMC - Complement carry flag

STC - Set carry flag

CLD - Clear direction flag

STD - Set direction flag

CLI - Clear Interrupt flag

STI - Set " "

Processor Control Instruction

- Wait : for test input pin to go low.

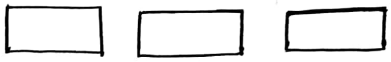
HLT : Hault the processor

NOP : No operation.

ESC : escape - es to external device.

Lock : bus lock instruction prefix.

Assembly Language Programming (ALP):



User gives the instructions to assembler such as load, move, .. assemble gives its opcode to the processor.

→ MASM :-

→ Program structure.

Data Segment

code Segment

End

- .model Small → is a assembler directive. To understand the structure Programmer → assembler
- Assume cs: code Segment or .data
  DS : Data Segment. .code.

- data includes data concept
  db, dw, dR, dT

eg: msg db "Hai$"
msg dw "1234, 123, 564$"
↓ 2 memory location / 16 bit.

~~code~~

(array name array size array type)

- code
  - Initialization.
    move data to data segment

```
mov ax, @data
mov ds, ax
```
} initialize.

- address of the data segment.

```
lea dx, msg
```
offset address of msg is moved to dx.

```
Mov ah, 01h
int 21h
```
} character read.

1 moved to ah and interrupt is read.
Value read from keyboard will be in al.

```
mov ah, 02h
int 21h
```
} character print

```
mov ah, 4ch
int 21h
```
} Exit.

- In assembly language first executes the code (It include the initialization of data)
- edit filename.asm, - editor open write the program
- masm filename.asm - Save
- link filename.obj
- filename

Q Write a progm to add a data byte located at offset 0500H in 2000H segment to another data byte available at 600H in the same segment and store the result at 0700H in the same segment.

```
mov ax, 2000H
mov DS, AX
mov AX, [500H]
ADD AX, [600H]
mov [700H], AX
HLT
```
} Segment

Q Move a byte string, 16 bytes long, from the offset 200H to 300H in the segment 7000H

```
mov AX, 7000H
mov DS, AX
Mov SI, 200H
MOV DI, 300H
mov CX, 0010H
```

```
Back  mov AL, [SI]
      mov [DI], AL
      INC SI
      INC DI
      loop back
      HLT
```

Q Findout the largest number from an unordered array of 16-8 bit numbers stored sequentially in the memory location starting at offset 500H in the segment 2000H.

```
      MOV  CX, 0FH
      mov  AX, 2000H
      mov  DS, AX
      mov  SI, 0500H
      mov  AL, [SI]
Back: INC  SI
      CMP  AL, [SI]
      JNC  NEXT
      mov  AL, [SI]
NEXT: Loop back
      HLT
```

write a program for addition of two numb.

Assume CS: code, DS: DATA

ALP using assembler

even and odd

```
·Data
·code
    XOR  BX, BX
    XOR  DX, DX
    mov  AX, @data
    mov  ds, ax
    mov  CL, count
    mov  SI, offset list
Again: mov AX, [SI]
    ROR  AX, 01
    JNZ  again
    mov  AH, 4CH
    INT  21h
    JC   odd
    INC  BY
```

```
Jmp Next
odd: INC DX
Next: INC SI
      DEC CL
      JNZ again
      MOV AH,4CH
      INT 21h
      code end.
```


13/a/17

Q. Differentiate b/w ~~macro~~ procedure and macro (function)

PROCEDURE	Macro
• A procedure is a set of instruction needed repeatedly by the program it is stored as a subroutine and invoked from several places by the main prgm.	A macro is similar to a procedure but it is not invoked by the main prgm instead macro code is pasted to the main pgm where ever the macro name is written in the main pro.
• A subroutine is invoked by a call instruction and control written by RET instruction	A macro is simply accessed by writting its name. The entire macro code is pasted at the location by the assembler.
Reduces the size of the prgm	Increases the size of the pgm
• Executes slower	Execute faster.
• Procedure depends on stack	Not depends on stack.

Differentiate b/w Jump instruct and

Jump instruct^{n}	call instruct^{n}
Jump instruct^{n} is used to a new locat^{n} in the pgm and continue	call instruct^{n} is used to invo a subroutine, execute it an return to main pgm
A jump instruct^{n} simply put branch address	A call first stores the return address onto the stack and th loads the branch address to IP
In 8086 Jump can be either conditional / unconditional	In 8086 calls are only conditional
doesn't use the stack	Uses the stack
It doesn't need a return instruct^{n}	It needs a return instru to return back to the mai program.

Assembly directives / Pseudo opcodes

Statements that direct the assembler to do some special task no machine language code is produced for these stmts their main task is to inform the assembler about the start / End of the progm segment, procedure / pgm. To reserve appropriate space for data storage.. Some of the assembler directives are

DB - Define Byte

eg. SUM DB 00

Used to define a byte type variable. Assembler reserves one byte of mly for the variable.

- DW - Define word - Reserve 2 bytes
- DD - Double word : " 4 bytes.
- DQ - Quard Word : 8 bytes
- DT - Ten Bytes
- DUP() - List DB . TEN DUP() ZERO
  It stores list of a series of 10 byte initialize to zero.
- SEGMENT - used to indicate the beginning of a segment
- END - End of a segment
- ASSUME - eg: ASSUME CS : code.
  associates a logical segment with a processor segment
- PROC - beginning of a procedure.
- ENDP - End of a procedure.
- END - End of a program.
- EQU - to defines a constant
  eg AREA EQU 25
- Offset.
- START
- → Model Directives
  - model small
  - model medium
  - model large.

Parameter Passing techniques used in assembly language programming

A parameter is

more flexible and enhance the usage of the subroutine eg: factorial of 'n' numbers. There are many popular methods passing parameter to subroutine

1. Using registers: here the main pgm store the parameter into a register like DL, and calls the subroutine. Now subroutine takes the parameter value from DL register and works on it.

```
main:
    mov DL, 25H
    call Sub
    - - - - - -
    - - - - - -
Sub: mov AL, DL
    - - - - - -
    RET
```

→ Using memory locations directly

```
main:
    mov [4000H], 25H
    call Sub
Sub mov AL, [4000H]
    - - - - - -
    RET
```

3) Using memory location indirectly

```
main:
    mov SI, 4000
    mov [SI], 25H
    call Sub
Sub:
    mov AL, [SI]
    - - - -
    RET
```

4) Using Stack.

```
main:
    mov BX, 1234H
    Push BX          move to stack
    call Sub         Return address to stack.
Sub:
    POP CX
    POP AX
    - - - -
    - - - -
    RET
```

Here the main pgm push the parameter into the stack ~~to~~ and call the subroutine during call operation microprocessor pushes the return address to the stack this will be placed above the Parameter which is stored in the stack. Hence the subroutine first pop the return address into some register there after the subroutine will pop the parameter and use it. Before Returning the subroutine must first pup the return address to the stack and only then