BITS Pilani

Microprocessors & Interfacing

MASM & INSTRUCTION SET

Dr. Gargi Prabhu
Department of CS & IS

Directives

- Indicate how an operand or section of a program is to be processed by the assembler.
- By default the assembler accepts only 8086/8088 instructions, unless a program is preceded by the .686 or .686P directive

Storing Data in a Memory Segment

innovate achieve lead

```
LIST_SEG      SEGMENT
DATA1 DB   1,2,3              ;define bytes
      DB   45H                ;hexadecimal
      DB   'A'                ;ASCII
      DB   11110000B          ;binary
DATA2 DW   12,13              ;define words
      DW   LIST1              ;symbolic
      DW   2345H              ;hexadecimal
DATA3 DD   300H               ;define doubleword
      DD   2.123              ;real
      DD   3.34E+12           ;real
LISTA DB   ?                  ;reserve 1 byte
LISTB DB   10 DUP(?)          ;reserve 10 bytes

      ALIGN 2                 ;set word boundary

LISTC DW   100H DUP(0)        ;reserve 100H words
LISTD DD   22 DUP(?)          ;reserve 22 doublewords

SIXES DB   100 DUP(6)         ;reserve 100 bytes

LIST_SEG      ENDS
```

MASM Directives

Directive	*Function*
.286	Selects the 80286 instruction set
.286P	Selects the 80286 protected mode instruction set
.386	Selects the 80386 instruction set
.386P	Selects the 80386 protected mode instruction set
.486	Selects the 80486 instruction set
.486P	Selects the 80498 protected mode instruction set
.586	Selects the Pentium instruction set
.586P	Selects the Pentium protected mode instruction set
.686	Selects the Pentium Pro–Core2 instruction set
.686P	Selects the Pentium Pro–Core2 protected mode instruction set
.287	Selects the 80287 math coprocessor
.387	Selects the 80387 math coprocessor

MASM Directives

innovate achieve lead

Directive	*Function*
.CODE	Indicates the start of the code segment (models only)
.DATA	Indicates the start of the data segment (models only)
.EXIT	Exits to DOS (models only)
.MODEL	Selects the programming model
.STACK	Selects the start of the stack segment (models only)
.STARTUP	Indicates the starting instruction in a program (models only)
ALIGN n	Align to boundary n (n = 2 for words, n = 4 for doublewords)
ASSUME	Informs the assembler to name each segment (full segments only)
BYTE	Indicates byte-sized as in BYTE PTR
DB	Defines byte(s) (8 bits)
DD	Defines doubleword(s) (32 bits)
DQ	Defines quadwords(s) (64 bits)
DT	Defines ten byte(s) (80 bits)
DUP	Generates duplicates
DW	Define word(s) (16 bits)
DWORD	Indicates doubleword-sized, as in DWORD PTR
END	Ends a program file
ENDM	Ends a MACRO sequence
ENDP	Ends a procedure

MASM Directives

Directive	*Function*
ENDS	Ends a segment or data structure
EQU	Equates data or a label to a label
FAR	Defines a far pointer, as in FAR PTR
MACRO	Designates the start of a MACRO sequence
NEAR	Defines a near pointer, as in NEAR PTR
OFFSET	Specifies an offset address
ORG	Sets the origin within a segment
OWORD	Indicates octalwords, as in OWORD PTR
PROC	Starts a procedure
PTR	Designates a pointer
QWORD	Indicates quadwords, as in QWORD PTR
SEGMENT	Starts a segment for full segments
STACK	Starts a stack segment for full segments
STRUC	Defines the start of a data structure
USES	Automatically pushes and pops registers
USE16	Uses 16-bit instruction mode
USE32	Uses 32-bit instruction mode
WORD	Indicates word-sized, as in WORD PTR

EQU Directive

- EQU equates a numeric, ASCII, or label to another label.
  Syntax: identifier EQU expression.

  Example:

```
TEN     EQU 10
NINE    EQU 9

        MOV AL,TEN
        ADD AL,NINE
```

- It assigns a fixed value to a symbol, which cannot be changed during program execution.
- Useful for defining constants such as numerical values, memory addresses, or bit masks.

THIS directive

- Appears as THIS BYTE, THIS WORD, THIS DWORD, or THIS QWORD. In certain cases, data must be referred to as both a byte and a word.
- The assembler can only assign either a byte, word, or doubleword address to a label.

ORG Directive

- The ORG directive specifies the origin (starting address) for a segment in memory.

  Syntax: ORG expression

  Example: ORG 100h

- It sets the starting address for the subsequent code or data segment.
- Allows organization of program memory by specifying where each segment begins.
- Essential for positioning code or data at specific memory locations.

Example

```
                        ;Using the THIS and ORG directives
                        ;
0000                    DATA_SEG      SEGMENT

0300                          ORG     300H

= 0300                  DATA1   EQU   THIS BYTE
0300                    DATA2   DW    ?
0302                    DATA_SEG      ENDS

0000                    CODE_SEG      SEGMENT 'CODE'
                              ASSUME CS:CODE_SEG, DS:DATA_SEG
0000 8A 1E 0300 R             MOV BL,DATA1
0004 A1 0300 R                MOV AX,DATA2
0007 8A 3E 0301 R             MOV BH,DATA1+1

000B                    CODE_SEG      ENDS
```

The ASSUME statement tells the assembler what names have been chosen for the code, data, extra, and stack segments. Without the ASSUME statement, the assembler assumes nothing and automatically uses a segment override prefix on all instructions that address memory data. The ASSUME statement is only used with full-segment definitions

PROC and ENDP

- Indicate the start and end of a procedure (subroutine).
- Directives force structure because the procedure is clearly defined. Note that if structure is to be violated for whatever reason, use the CALLF, CALLN, RETF, and RETN instructions.
- Both require a label to indicate the name of the procedure.
- The PROC directive, must also be followed with a NEAR or FAR.
- NEAR procedure- resides in the same code segment as the program.
- FAR procedure - reside at any location in the memory system
- Often the call NEAR procedure is considered to be local, and the call FAR procedure is considered to be global.

Example

```
;A procedure that adds BX, CX, and DX with the
;sum stored in AX
;
ADDEM PROC  FAR             ;start of procedure

      ADD   BX,CX
      ADD   BX,DX
      MOV   AX,BX
      RET

ADDEM ENDP                  ;end of procedure
```

Example

```
                ;A procedure that includes the USES directive to
                ;save BX, CX, and DX on the stack and restore them
                ;before the return instruction.
        ADDS    PROC    NEAR    USES BX CX DX

*               push    bx
*               push    cx
*               push    dx
                ADD     BX,AX
                ADD     CX,BX
                ADD     DX,CX
                MOV     AX,DX
                RET
*               pop     dx
*               pop     cx
*               pop     bx
*               ret     0000h

        ADDS    ENDP
```

Memory Organization

- Two basic formats for developing software
  - Models – unique MASM
  - Full-segment Definitions – common to most assemblers
- Model is easier to understand for the beginner programmer.
- Models are also used with assembly language procedures that are used by high level languages such as C/C++.
- Full-segment definitions offer better control over the assembly language task and are recommended for complex programs.

Directives

TITLE line (optional)

- Contains a brief heading of the program and the disk file name

.MODEL directive

- Specifies the memory model configuration

TINY: Suitable for small programs that fit within a single code segment and don't require data or stack segments.

SMALL: Suitable for small to moderately sized programs with separate code, data, and stack segments. Code and data segments can be up to 64KB in size.

COMPACT, MEDIUM, LARGE, HUGE: These memory models are suitable for larger programs that require more memory segmentation. They provide increasing levels of memory management and segmentation capabilities.

Model Example

```
        .MODEL SMALL            ;select small model
        .STACK 100H             ;define stack
        .DATA                   ;start data segment

LISTA   DB    100 DUP(?)


LISTB   DB    100 DUP(?)



        .CODE                   ;start code segment

HERE:   MOV   AX,@DATA          ;load ES and DS
        MOV   ES,AX
        MOV   DS,AX
        CLD                     ;move data
        MOV   SI,OFFSET LISTA
        MOV   DI,OFFSET LISTB
        MOV   CX,100
        REP   MOVSB

        .EXIT 0                 ;exit to DOS
        END HERE
```

Full-Segment Definitions

```
STACK_SEG       SEGMENT         'STACK'
        DW      100H DUP(?)


STACK_SEG       ENDS

DATA_SEG        SEGMENT         'DATA'
LISTA   DB      100 DUP(?)


LISTB   DB      100 DUP(?)


DATA_SEG        ENDS

CODE_SEG        SEGMENT         'CODE'
        ASSUME CS:CODE_SEG, DS:DATA_SEG
        ASSUME SS:STACK_SEG
MAIN    PROC    FAR
        MOV     AX,DATA_SEG             ;load DS and ES
        MOV     ES,AX
        MOV     DS,AX
        CLD                             ;save data
        MOV     SI,OFFSET LISTA
        MOV     DI,OFFSET LISTB
        MOV     CX,100
        REP     MOVSB
        MOV     AH,4CH                  ;exit to DOS
        INT     21H
MAIN    ENDP
CODE_SEG        ENDS
        END     MAIN
```

Example

innovate achieve lead

MODEL

```
;
.MODEL TINY
.CODE
.STARTUP

MAIN:
        MOV   AH,6              ;read a key
        MOV   DL,0FFH
        INT   21H
        JE    MAIN              ;if no key typed
        CMP   AL, '@'
        JE    MAIN1             ;if an @ key
        MOV   AH,06H            ;display key (echo)
        MOV   DL,AL
        INT   21H
        JMP   MAIN              ;repeat
MAIN1:

.EXIT                           ;exit to DOS
END
```

FULL SEGMENT DEFINITIONS

```
CODE_SEG        SEGMENT 'CODE'
        ASSUME CS:CODE_SEG

MAIN    PROC    FAR

        MOV   AH,06H            ;read a key
        MOV   DL,0FFH
        INT   21H
        JE    MAIN              ;if no key typed
        CMP   AL,'@'
        JE    MAIN1             ;if an @ key
        MOV   AH,06H            ;display key (echo)
        MOV   DL,AL
        INT   21H
        JMP   MAIN              ;repeat
MAIN1:
        MOV   AH,4CH            ;exit to DOS
        INT   21H

MAIN    ENDP
        END   MAIN
```

Program Control Instructions

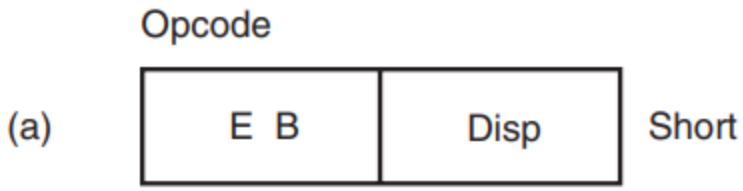
- Direct the flow of a program and allow the flow to change.
- A change in flow often occurs after a decision made with the CMP or TEST instruction is followed by a conditional jump instruction
- jump (JMP), allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction.
- A conditional jump instruction allows the programmer to make decisions based upon numerical tests.
- Results of numerical tests are held in the flag bits, which are then tested by conditional jump instructions

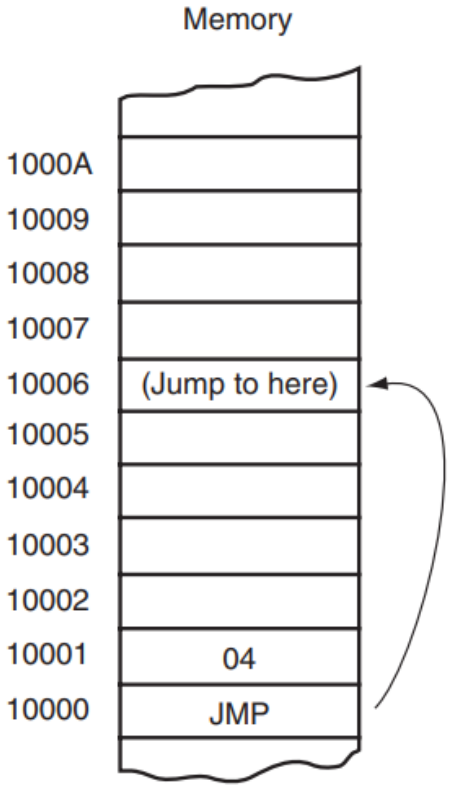
Unconditional Jump (JMP)

- Short jumps are called relative jumps because they can be moved, along with their related software, to any location in the current code segment without a change.

Displacement between +127 and -128

IP= IP+ Sign Extend(Displacement)



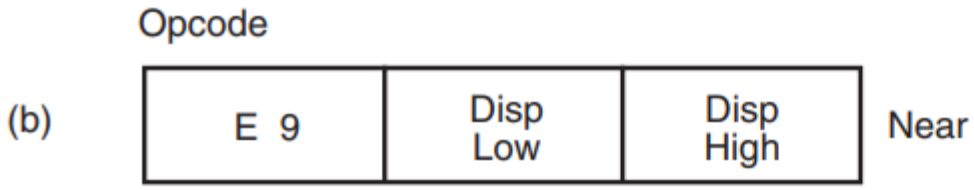


A short jump to four memory locations beyond the address of the next instruction

CS = 1000H
IP = 0002H
New IP = IP + 4
New IP = 0006H

Near Jump

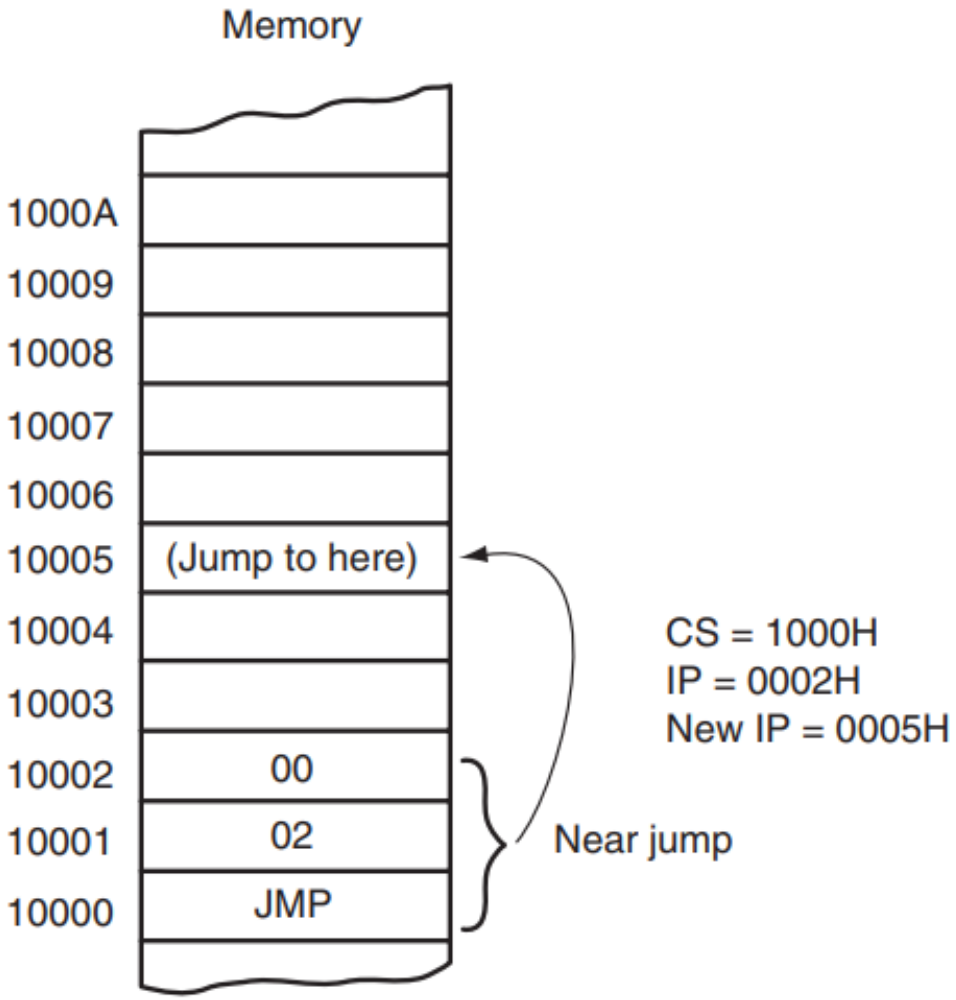
- The near jump is similar to the short jump, except that the distance is farther.
- A near jump passes control to an instruction in the current code segment located within ±32K bytes from the near jump instruction
- The near jump is a 3-byte instruction that contains an opcode followed by a signed 16-bit displacement.



Near Jump

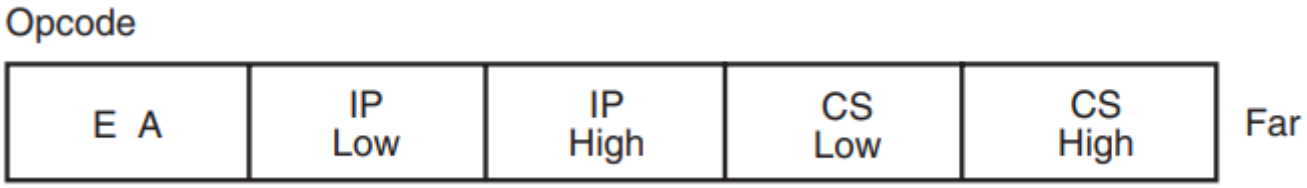
A near jump that adds the displacement (0002H) to the contents of IP.

```
        XOR   BX,BX
START:  MOV   AX,1
        ADD   AX,BX
        JMP   NEXT

<skipped memory locations>

NEXT:   MOV   BX,AX
        JMP   START
```



Far Jump

- A far jump instruction obtains a new segment and offset address to accomplish the jump.
- Bytes 2 and 3 of this 5-byte instruction contain the new offset address; bytes



Conditional Jump

- Conditional jump instructions are always short jumps in the 8086
- This limits the range of the jump to within +127 bytes and -128 bytes from the location following the conditional jump
- The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (0).
- If the condition under test is true, a branch to the label associated with the jump instruction occurs.
- If the condition is false, the next sequential step in the program executes.

E.g. JC Jump if carry is set

Assembly Language	*Tested Condition*	*Operation*
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL	S != O	Jump if less than
JLE	Z = 1 or S != O	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

LOOP

- The LOOP instruction is a combination of a decrement CX and the JNZ conditional jump.
- In the 8086 through the 80286 processors, LOOP decrements CX; if CX != 0, it jumps to the address indicated by the label.
- If CX becomes 0, the next sequential instruction executes.
- In the 80386 and above, LOOP decrements either CX or ECX, depending upon the instruction mode.

Example

```
;A program that sums the contents of BLOCK1 and BLOCK2
;and stores the results on top of the data in BLOCK2.
;
.MODEL SMALL                    ;select SMALL model
.DATA                           ;start data segment
BLOCK1 DW 100 DUP(?)            ;100 words for BLOCK1


BLOCK2 DW 100 DUP(?)            ;100 words for BLOCK2


.CODE                           ;start code segment
.STARTUP                        ;start program
        MOV AX,DS               ;overlap DS and ES
        MOV ES,AX
        CLD                     ;select auto-increment
        MOV CX,100              ;load counter
        MOV SI,OFFSET BLOCK1    ;address BLOCK1
        MOV DI,OFFSET BLOCK2    ;address BLOCK2
L1:     LODSW                   ;load AX with BLOCK1
        ADD AX,ES:[DI]          ;add BLOCK2
        STOSW                   ;save answer
        LOOP L1                 ;repeat 100 times
.EXIT
END
```

Conditional LOOPs

LOOPE (loop while equal) instruction jumps if CX != 0 while an equal condition exists.

- It will exit the loop if the condition is not equal or if the CX register decrements to 0.

LOOPNE (loop while not equal) instruction jumps if CX != 0 while a not-equal condition exists.

- It will exit the loop if the condition is equal or if the CX register decrements to 0.

innovate achieve lead

BITS Pilani

Pilani Campus

Thank You