Lecture 11: Addressing Modes (Part 1)

- Operand Types (a review)
- Addressing Modes
- WORD and BYTE PTR

Basic Operand Types

- Three basic types:
  - immediate – a constant
  - register – a CPU register
  - memory – a reference to a location in memory

Immediate Operands

- An immediate operand is a constant expression such as a number, a character, or an arithmetic expression.

```
mov   al, 10          ; al = 10
mov   bl, 'A'         ; bl = 'A'
mov   cx, 'AB'        ; cx = 'AB'
mov   dx, 123h        ; dx = 123h
```

- The assembler calculates the value of the immediate operand and inserts it directly into the machine instruction.

Register Operands

- As we've seen, register operands are eight or sixteen bit registers (or 32 if using the extended registers).

```
mov     ax, bx
mov     al, bl
```

- Register addressing is very efficient because no memory access is required.

Memory Operands

- For memory operands, there are a number of different ways that they can be accessed using assembly language.
- These different ways are *addressing modes*.

Assembly Language Addressing Modes

- Memory is accessed by calculating its *effective address*, using the distance (or offset) of the data from the beginning of a segment (usually the data segment).
- Memory Addressing modes:
  Direct
  Register Indirect
  Based or Indexed
  Base-indexed
  Base-indexed with displacement (displacement is a number or offset from a variable)

Direct Operands

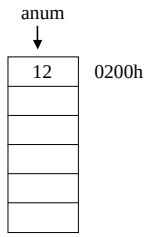
- A direct operand refers to the contents of memory at a location identified by a label in the data segment.
- We've seen many examples of this already. Here are a few more:
  – Irvine, p. 78

OFFSET Operator

The *offset operator* is used to move the offset of a label into a register or variable.

```
anum    db      12
        .
        .
        mov     bx, offset anum

-- assembles as:
        mov     bx, 0200h
```



Why is this useful?
In homework 2 we saw how storing an address in BX could be used to step through a list of numbers.

Another way to get the address

- LEA BX, anum
- LEA stands for load effective address
- There is a difference between LEA and MOV … OFFSET:
  - LEA calculates the label's offset at *runtime*
  - MOV … OFFSET moves an immediate value that is known at *assembly time*
- Use LEA if the effective address of an operand *must* be calculated at runtime.

Direct-Offset Operands

- You can use the addition and subtraction operators to access a list of values.
  - The + operator adds to the offset of a variable.
  - The minus operator subtracts from the labels offset.

Addition Example

- Irvine, p. 79

Subtraction Example

- p. 76 in Irvine
- Where have we seen this type of addressing before?

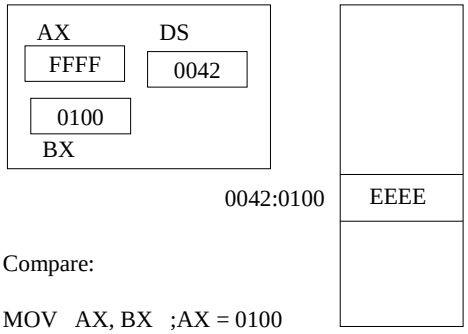
Lecture 6 Example of Direct Offset Addressing

- Example:

```
arrayB   db  10h, 20h
arrayW   dw 100h, 200h
….
mov al, arrayB    ;AL = 10h
mov al, arrayB+1  ;AL = 20h
mov ax, arrayW    ;AX = 100h
mov ax, arrayW+2  ;AX = 200h
mov ax, arrayW+1  ;AX = ?
```

Indirect Addressing

- An indirect operand is a register that contains the offset of data in memory.
- When the offset of the variable is placed in a register, the register becomes a pointer to the label.
- You can use SI, DI, BX, and BP to hold indirect operands.
  - BX: base register
  - SI, DI: index registers
  - BP: base pointer (contains an offset from the SS register)



Compare:

```
MOV   AX, BX   ;AX = 0100
MOV   AX, [BX]  ; AX = EEEE
```

So why do we have to use indirect addressing? We could just put a label at location 0042:0100 and do a

```
MOV   AX, label
```

```
;increment each word in an array list
    .data
LIST    DW     5FFFh, 0Ah, 12h, 17h, 4h
    .code
    .startup
;step through each element and increment
    MOV         AX, LIST
    INC AX
    MOV         LIST, AX
    MOV         AX, LIST+2
    INC AX
    MOV         LIST+2, AX
etc… for list+4, list+6, list+8
```

- This could get pretty long! You need a way to modify the address at *execution time* so you can put the above code in a loop.

- If you put the address of the base of the list in BX, you can use indirect addressing and a loop!

```
;increment each word in an array list
    .data
LIST    DW      5FFFh, 0Ah, 12h, 17h, 4h
    .code
    .startup
    MOV  BX, offset LIST
    MOV  CX, 5              ;5 elements
LUP: MOV AX, [BX]       ; get item pointed to by BX
    INC  AX             ;add one to it
    MOV  [BX], AX       ;put back into same place
    ADD  BX, 2          ;increment address by 2
    LOOP LUP            ;loop back to top
```

- Quite an improvement!

An even shorter way!

```
;increment each word in an array list
    .data
LIST    DW      5FFFh, 0Ah, 12h, 17h, 4h
    .code
    .startup
        LEA     BX, LIST
        MOV     CX, 5
LUP:    INC WORD PTR [BX]       ;*
        ADD BX, 2
        LOOP LUP
```

- * You can't just say INC [BX] because there is nothing in the instruction to indicate if BX has the address of a word (like in this example) or a byte.
- You can specify which one it is using WORD PTR or BYTE PTR

More on WORD and BYTE PTR

```
;increment each word in an array list
    .data
LIST    DW      5FFFh, 0Ah, 12h, 17h, 4h
    .code
    .startup
        LEA     BX, LIST
        MOV     CX, 5
LUP:    INC WORD PTR [BX]    ; List's 1st entry
                             ;after INC would
                             ;be 6000h
        vs.
….
LUP:    INC BYTE PTR [BX]    ;Lists 1st entry
                             ;after INC would
                             ;be 5F00h
```

only the low byte (FF) was incremented. It wrapped around from FF back to 0; the upper byte was unaffected.

- So, the pointer is the same (always addresses a byte) but if declared as a word pointer, the operation is done on a 16-bit value; if a byte pointer, on an 8-bit value.
- You wouldn't need to make a distinction for:

  MOV AX, [DI]
- Why?
  - The use of AX indicates that DI should be treated as a word pointer.

We've seen indirect addressing before!

- Remember homework 2 with our array of numbers:
  - -14 = 0FFF2h
  - 0 = 0000h
  - -6 = 0FFFAh
  - -42 = 0FFD6h
  - 17 = 0011h
  - 2 = 0002h
- We used BX as a pointer to each element in the array in order to step through and add them up (in part 1).
- We used DX as a pointer to the last element in the array and compared it to BX in order to tell if we were done.
- DS pointed to the start of the data segment.

EA	Data	Offset from DS
1C554	F2	0004
1C555	FF	0005
1C556	00	0006
1C557	00	0007
1C558	FA	0008
1C559	FF	0009
1C55A	D6	000A
1C55B	FF	000B
1C55C	11	000C
1C55D	00	000D
1C55E	02	000E
1C55F	00	000F

DS	1C55	
BX		address of 1st word
DX		address of last word

Data	Offset from DS	
F2	0004	mov bx, 0004h
FF	0005	mov ax, [bx] ; ax = FFF2
00	0006	add bx, 2 ; bx = 0006
00	0007	mov ax, [bx] ; ax = 0000
FA	0008	add bx, 2 ; bx = 0008
FF	0009	mov ax, [bx] ; ax = fffa
D6	000A	add bx, 2 ; bx = 000A
FF	000B	mov ax, [bx] ; ax = ffd6
11	000C	add bx, 2 ; bx = 000C
00	000D	mov ax, [bx] ; ax = 0011
02	000E	add bx, 2 ; bx = 000E
00	000F	mov ax, [bx] ; ax = 0002

BX serves as a pointer into the array. In HW2, we used compare and conditional jumps to traverse the array in a loop.

Not just to access words.

- example from p. 106, Irvine

Segment Defaults

- The offset created by an indirect operand is assumed to be from DS unless BP (or EBP) is part of the indirect operand.
- If BP is involved, then the offset is from the stack segment (SS register).
- You can override the default segment if necessary:

```
mov   al, cs:[si] ;offset from CS
```

Another Example

- Example 3, part 1 from Irvine p. 107

Example, continued

- You can avoid the separate instructions that increment BX:
- Example 3 from Irvine, part 2, p. 108

- This takes advantage of sum being stored after the data. This is NOT a good approach!

In-Class Exercise

- The next slide will give an example of a program that uses indirect addressing. Answer the following questions:
  - This program will not compile. What lines need to be changed in order to fix it?
  - What is COUNT? (look in Irvine)
  - What does this program do?
  - If you fix the compilation errors and run the program, there's still a problem with it. What is it?

```
        TITLE lecture 11  -- Indirect Addressing
        .model small
        .stack 100h
        .data
oldstr db"Indirect addressing is Useful!"
COUNT = ($ - oldstr)      ;what does this do?
                          ;Ch 4 in Irvine
newstr db       COUNT
        .code
        .startup
        mov     bx, offset oldstr       ;
        mov     dx, offset newstr       ;
        mov     cx, count               ;
top:    mov     al, [bx]                ;
        cmp     al, 60h                 ;
        jbe     ok                      ;
        sub     al, 20h                 ;
ok:     mov     [dx], al                ;
        inc     bx                      ;
        inc     dx
        loop    top                     ;
        mov     ah, 9
        mov     dx, offset newstr       ;
        int     21h                     ;
        .exit
        end
```