

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

- constant expression: number, character, or arithmetic

expression

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

Register Operands

- 8, 16 or 32 bit registers

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

- fast and efficient

Memory Operands

- Many ways to access:

addressing modes

Assembly Language Addressing Modes

- Memory Addressing modes:
 - Direct
 - Direct-Offset
 - Register Indirect
 - Based or Indexed
 - Base-indexed
 - Base-indexed with displacement
(displacement is a number or offset from a variable)

Direct Operands

- Memory location specified using a label

```
.data
count db 10
...
.code
...
    mov al, count
```

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 label's offset.

Addition Example

- Irvine, p. 79

Subtraction Example

- p. 79 in Irvine

- Where have we seen this type of addressing before?

Lecture 7 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

- Offset of a variable in a register
- Register “points to” the data
- Registers allowed: BX, SI, DI, BP
 - BX: base register
 - SI, DI: index registers
 - BP: base pointer (contains an offset from the SS register)

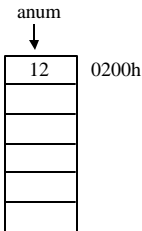
OFFSET Operator

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
    
```

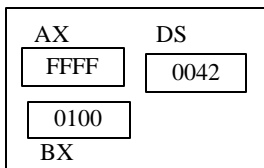


Why is this useful?

Another way to get the address

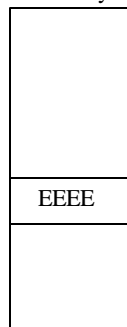
- LEA BX, anum
- Difference:
 - LEA calculates the label's offset at *runtime*
 - MOV ... OFFSET moves an immediate value that is known at *assembly time*
- Use LEA if the effective address of an operand *must* be calculated at runtime.

CPU



0042:0100

Memory



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
```

- With indirect addressing:

```

;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

```

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

```

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.

- You wouldn't need to make a distinction for:
MOV AX, [DI]
- Why?

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

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
 BX
 DX

address of 1st word
 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!