CS221
More Assembly, Chapter 4 Irvine

Basic Instructions

We are finally at a position where we can start going over some instructions!

MOV
The first is the MOV instruction which moves data from one location to another. The destination comes first, followed by the source. Either may be registers or memory. The sizes of the data you are moving must match (e.g. can't move a word into a byte):

MOV reg, reg	MOV mem, reg
MOV reg, mem	MOV mem, immediate
MOV reg, immediate	

Note the missing MOV instruction – you aren't allowed to move from one memory location directly to another memory location. Instead you must move to a register first. Such is the price one pays for a non-orthogonal architecture.

Here are some examples:

```
.data
x       db      10
y       db      20
total   dw      ?

.code
mov     al, x               ; Move values to AL and BL
mov     bl, y
mov     total, 1000         ; Store 1000 into total
mov     x, y                ; INVALID
mov     ah, x+1             ; move location X+1 into ah, which is Y
```

Note the last example. We can reference memory as offsets from known memory locations. In the last case, we added one to the offset of x. This gives us the address for y, so the contents of y are moved into AH. Although y had a label, this technique lets you access data that may not have a label.

MOVSX, MOVZX
Sometimes we want to move a small-sized piece of data into a larger register. For example, we might want to move an 8 bit value into a 16 bit register. Generally this occurs with numbers. We might have a number that is being represented by a byte, but now we want to move it into a register and have the 16 bit register operate on the data.

One solution is to use AL. We could copy the value that is a byte into AL, and then we would also have to zero out AH. Rather than use two instructions, there is a special instruction to zero out the unfilled bits in the destination. This is the **movzx** instruction (move with zero extend)

```
.data
mynum byte 1
.code
mov ax, 0AAAAh
movzx ax, mynum                ; AX now contains 0001
```

The movzx instruction requires the .386 or higher processor directive.
We can do a similar thing for the extended registers:

```
movzx eax, mynum               ; EAX now contains 00000001
```

In a similar fashion, the **movsx** instruction (move with sign extend) can be used to extend negative numbers. Consider what happens if we use movzx on a negative value:

```
.data
mynum byte –1
.code
mov ax, 0AAAAh
movzx ax, mynum                ; AX now contains 00FF
```

Why did we get 00FF? Because –1 is represented as FF in two's complement.
To correctly get –1 into AX, we need to extend (i.e. copy) the sign bit all the way to the most significant bit. If the sign bit is 1, then the movsx instruction pads with 1's instead of 0's:

```
movsx eax, mynum               ; EAX now contains FFFFFFFF or –1
```

Similarly, if the sign bit contained zero, then we would pad with 0's all the way out to the most significant bit.

XCHG
The next instruction is the XCHG instruction. This exchanges the contents of two registers or a register and a variable:

```
XCHG reg, reg          XCHG reg, mem          XCHG mem, reg
```

This is an efficient way to swap two operands, for example, in sorting some data.

INC and DEC
INC is used to increment an operand by 1, while DEC decrements it by 1.
The operand may be memory or a register.

ADD
The ADD instruction takes a destination and a source of the same size, adds them, and stores the result in the destination:

```
ADD ah, al          ;       Sets AH = AH + AL
ADD var1, 10        ;       Var1 = Var1 + 10
```

Depending on the result of the addition, the zero, negative, sign, overflow, or carry flags are affected.

SUB
The SUB instruction takes a destination and a source of the same size, subtracts them, and stores the result in the destination.

```
SUB ah, bl          ; Sets AH = AH – BL
SUB var1, 10        ; Sets Var1 = Var1 – 10
```

Depending on the result of the subtraction, the zero, negative, sign, overflow, or carry flags are affected.

Types of Operands

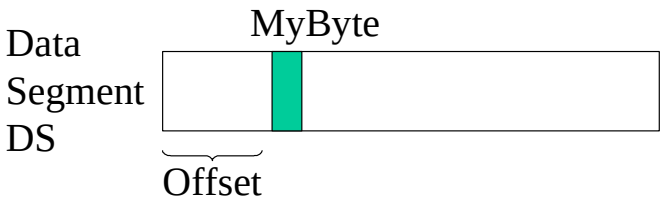
So far we have been dealing primarily with direct addresses and with immediate data. Let's describe for now **direct**, **direct-offset**, and **register indirect** addressing.

Direct operands refer to the contents of memory at some known location. For now these locations are specified by a label:

```
.data
countLabel      WORD        1000
.code
mov ax, countLabel                  ; Moves 1000 into AX
inc countLabel
```

Here, countLabel refers to the address that is used to store a word.

If we actually want to access the offset that a variable is stored at, we can use the **offset** operator. In protected mode, an offset is always 32 bits long. In real mode, offsets are 16 bits. To illustrate, the following figure shows a variable named myByte inside the data segment:



The offset essentially gives us the working address of some data variable. Here is a code sample.

```
.data
countLabel      DWORD       1000
.code
mov esi, offset countLabel      ; Moves address of countLabel into ESI
                                ; This would be  0 in real mode, some address
                                ; where the 1000 is stored in protected mode
```

For example, if countLabel is stored at offset 0 of its segment, then the value 0 gets loaded into ESI. If we did not use the offset directive, then the actual value inside countLabel is loaded into ESI (i.e. 1000).

Direct-Offset operands are used to access locations offset up (+) or down (-) from a label. For example:

```
.data
countLabel1    WORD        10
countLabel2    WORD        20
.code
mov ax, countLabel1+2               ; Moves 20 into ax
mov ax, countLabel2-2               ; Moves 10 into ax
```

I subtracted and added 2 because the word size is 2 bytes.

Another example in real mode:

```
.data
bList db 10h, 20h, 30h, 40h         ; Let's say bList begins at offset 0
wList dw 1000h, 2000h, 3000h

.code
mov di, offset bList                ; DI = 0000
mov bx, offset bList +1             ; BX = 0001
mov si, offset wList+2              ; SI = 0006   need to pass up bList data
```

If we try this in protected mode:

```
.data
bList db 10h, 20h, 30h, 40h
wList dw 1000h, 2000h, 3000h
.code
mov eax, offset bList               ; EAX = Address of first byte in bList
mov ebx, offset bList +1            ; EBX = Address of second byte in bList
mov edx, offset wList+2             ; EDX = Address of 2000h
```

Finally, **register indirect** mode is used when a register contains an offset of some memory location. The contents of that memory location are then accessed. For example:

```
.data
val1 BYTE 10h
.code
mov esi, offset val1        ; ESI contains offset of Val1
mov al, [esi]               ; AL gets 10h
```

Using this mode it is possible to access memory outside our data segment. If this occurs then a general protection fault occurs which will crash our program.

In real mode, we can only use the SI, DI, BX, or BP registers. We can access that memory location using brackets around the register, e.g. [BX]. By accessing [BX] we are accessing the effective address of DS:BX, where BX is some offset from the DS.

For example:

```
.data
countLabel1   WORD      1000
countLabel2   WORD      2
.code
mov ebx, offset countLabel1     ; mov offset of countLabel to EBX
mov ax, [ebx]                   ; mov 1000 to AX
mov ax, [ebx+2]                 ; mov 2 to AX, combine with offset
```

We will have more to say about these later… as you can see this could be one way to access successive elements within an array.

More MASM Operators and Directives

There are various addressing operators available: OFFSET, PTR, LABEL, TYPE, and ALIGN We have already discussed OFFSET.

PTR is used to override the default size of an operand. This is used in combination with one of the following data types to indicate the new size: BYTE, SBYTE, WORD, DWORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE. For example:

```
mov al, byte ptr count          ; treat count like a byte
mov ax, word ptr newVal         ; treat newVal like a word
mov eax, dword ptr listPointer  ; treat listPointer like a double word
```

As an example, consider treating val32 below:

```
.data
val32   DWORD       12345678h
.code
mov ax, val32       ; INVALID, ax is 16 bits while val32 is 32bits
mov dx, val32+2     ; INVALID, doesn't get high word, tries to get doubleword
```

The solution is to use PTR to override the size:

```
mov ax, word ptr val32          ; AX = 5678h
mov dx, word ptr val32+2        ; DX = 1234h
```

In the above example we moved a large value into a small one. We can also move a small value into a large one, but we have to make sure that the memory locations after the first small value are set to the appropriate values:

```
.data
wordlist WORD       5678h, 1234h
.code
mov eax, dword ptr wordList         ; EAX = 12345678h
```

This moves data in reverse word order.

The LABEL directive can be used to assign another label to a memory location. Below we assign a label called val16 to the same location we have val32:

```
.data
val16 label word
val32 dword 12345678h
.code
mov ax, val16                       ; AX= 5678h
mov bx, val16+2                     ; BX = 1234h
```

The ALIGN directive aligns a variable on a byte, word, doubleword, or paragraph boundary. The syntax is:

```
ALIGN bound
```

Where bound can be 1 for byte, 2 for word, 4 for doubleword, etc. To do this, the assembler inserts empty bytes before the variable. The purpose of aligning data is because the CPU can process data stored at even-numbered addresses more quickly than those at odd-numbered addresses (recall how blocks of memory are loaded into the cache).

Example:

```
bVal BYTE ?
ALIGN 2
wVal WORD ?              ; This word is now on an even boundary
```

The TYPE operator returns the size, in bytes, of a single element. For example:

```
.data
var1 byte 20h
var2 word 1000h
var3 dword ?
var4 byte 10,20,30,40
msg byte “Hello”, 0
.code
mov ax, type var1                ; AX = 1
mov ax, type var2                ; AX = 2
mov ax, type var3                ; AX = 4
mov ax, type var4                ; AX = 1
mov ax, type msg                 ; AX = 1
```

Note that type does not count the length of a string or multiply defined data.

The LENGTHOF operator returns the number of elements that have been defined using DUP:

```
.data
val1 dw 1000h
arr dw 32 dup(0)
arr2 db 10 dup(0)
.code
mov ax, lengthof val1            ; AX = 1
mov ax, lengthof arr             ; AX = 32
mov ax, lengthof arr2            ; AX = 10
```

The SIZEOF operator multiples the LENGTH by the TYPE:

```
.data
arr dw 32 dup(0)
arr2 db 10 dup(0)
.code
mov ax, sizeof arr               ; 32*2 = 64
mov ax, sizeof arr2              ; 10*1 = 10
```

More x86 Assembly Instructions

We are now at a point to talk about additional x86 assembly instructions. You have already seen how to define, move, and perform mathematical operations on data. The next topic is how to perform unconditional branches and loops. Later we will look at performing conditional branches.

JMP

The JMP instruction tells the CPU to "Jump" to a new location. This is essentially a goto statement. We should load a new IP and possibly a new CS and then start executing code at the new location.

On the x86 we have three formats for the JMP instruction:

```
JMP SHORT destination
JMP NEAR PTR destination
JMP FAR PTR destination
```

Here, destination is a label that is either within +128 or –127 bytes (SHORT), a label that is within the same segment (NEAR), or a label that is in a different segment (FAR). By default, it is assumed that the destination is NEAR unless the assembler can compute that the jump can be short.

Some usage examples:

```
jmp L1                  ; NEAR unless can compute SHORT possible
jmp near ptr L1
jmp short L2
jmp far ptr L3          ; Jump to different segment
```

If it is possible to use SHORT, that is preferred. In a short jump, the machine code includes a 1 byte value that is used as a displacement and added to the IP. For a backward jump, this is a negative value. For a forward jump, this is a positive value. This makes the short jump efficient and doesn't need much space. In the other types of jumps, we'll need to store a 16 or 32 bit address as an operand.

Examples:

```
Label1:     jmp short Label2        ; Short Jump
            …
Label2:     jmp Label1              ; Short jump also since the
                                    ; assembler knows L1 is close
```

We can use JMP to make loops:

```
Label1: inc ax
```

```
        …
        … do processing
        jmp Label1
```

This is of course an infinite loop unless we have a jump somehow to break out of it.

LOOP

For loops, we have a specific LOOP instruction. This is an easy way to repeat a block of statements a specific number of times. The ECX register is automatically used as a counter and is decremented each time the loop repeats. The format is:

```
LOOP destination
```

Here is a loop that repeats 10 times:

```
        mov ecx, 10
        mov eax, 0
start:  inc eax
        …
        loop start                  ; Jump back to start
```

The loop decrements ECX by one each time we are in the loop. When ECX equals zero, the loop stops and no jump takes place. Upon the end of the above loop, ECX =0 and EAX = 10.

You have to be very careful with the LOOP instruction so that you don't change the contents of ECX inside the loop. Otherwise the loop will probably not execute the correct number of iterations.

LOOPW, LOOPD

In Real Mode, the LOOP instruction only works using the CX register. Since CX is 16 bits, this only lets you loop 64K times. If you have a 386 or higher processor, you can use the entire ECX register to loop up to 2^{32} times. LOOPD uses the ECX doubleword for the loop counter:

```
        .386  ; in protected mode
        mov ecx, 0A0000000h
L1      .
        .
        loopd L1                    ; loop A0000000h times
```

LOOPW uses a 16 bit word for CX just like LOOP.

Indirect Addressing

An indirect operand is generally a register that contains the offset of data in memory. In other words, the register is a pointer to some data in memory. Typically this data is used to do things like traverse arrays.

In real mode, only the SI, DI, BX, and BP register can be used. By default, SI, DI, and BX are assumed to be offsets from the DS (data segment) register. By default, BP is assumed to be an offset from the SS (stack segment) register.

The format to access the contents of memory pointed to by an indirect register is to enclose the register in square brackets. For example, if BX contains 100, then [BX] refers to the memory at DS:100.

Based on the real mode limitations, many programmers also typically use ESI, EDI, EBX, and EBP in protected mode, although we can also use other registers if we like.

Here is an example that sums three 8 bit values:

```
.data
aList byte 10h, 20h, 30h
sum byte 0
.code
mov ebx, offset aList          ; EBX points to 10h
mov al, [ebx]                  ; move to AL
inc ebx                        ; BX points to 20h
add al, [ebx]                  ; add 20h to AL
inc ebx
add al, [ebx]
mov esi, offset sum            ; same as MOV sum, al
mov [esi], al                  ; in these two lines
exit
```

Here instead we add three 16-bit integers:

```
.data
wordlist word 1000h, 2000h, 3000h
sum word ?
.code
mov ebx, offset wordlist
mov ax,[ebx]
add ax,[ebx+2]                 ; Directly add offset of 2
add ax,[ebx+4]                 ; Directly add offset of 4
mov [ebx+6], ax                ; [ebx+6] is offset for sum
```

Here are some examples in real mode:

```
        .data
        aString db "ABCDEFG", 0
        .code
        mov ax, @data                   ; Set up DS for our data segment
        mov ds, ax                      ; Don't forget to include this

        mov bx, offset aString          ; BX points to "A"
        mov cx, 7
L1:     mov dl, [bx]                    ; Copy char to DL
        mov ah, 2                       ; 2 into AH, code for display char
        int 21h                         ; DOS routine to display
        inc bx                          ; Increment index
        loop L1
```

This loops through and copies A,B,C,D,E,F,G to DL and displays it to the screen.

Here is another example that runs in real mode, can you figure out what it does? Recall that B800 is where video memory begins.

```
        mov ax, 0B800h
        mov ds, ax

        mov cx, 80*25
        mov si, 0
L:      mov [si], word ptr 0F041h   ; need word ptr to tell masm
                                    ; to move just two bytes worth
                                    ; (0F041h could use a dword)
        add si, 2
        loop L
```

Based and Indexed Operands

Based and indexed operands are essentially the same as indirect operands. A register is added to a displacement to generate an effective address. The distinction between based and index is that BX and BP are "base" registers, while SI and DI are "index" registers. As we saw in the previous example, we can use the SI index like it were a base register.

There are many formats for using the base and index registers. One way is to use it as an offset from an identifier much like you would use a traditional array in C or C++:

```
.data
string byte “ABCDE”,0
array byte 1,2,3,4,5
.code
mov ebx, 2
mov ah, array[ebx]                  ; move offset of array +2 to AH
                                    ; this is the number 3
mov ah, string[ebx]                 ; move character C to AH
```

Another technique is to add the registers together explicitly:

```
mov ah, [array + ebx]               ; same as mov ah, array[bx]
mov ah, [string + ebx]              ; same as mov ah, string[bx]
```

We can also add together base registers and index registers:

```
mov bx, offset string
mov si, 2
mov ah, [bx + si]                   ; same as above, number 3 to ah
```

However we cannot combine two base registers and two index registers. This is just another annoyance of non-orthogonality:

```
mov ah, [si + di]                   ; INVALID
mov ah, [bp + bx]                   ; INVALID
```

Finally, one other equivalent format is to put two registers back to back. This has the same effect as adding them:

```
mov ebx, 1
mov esi, 2
mov ah, array[ebx][esi]             ; Moves number 4 to ah, offset+1+2
mov ah, [array+ebx+esi]             ; Also moves 4 to ah
```

Sometimes this format is useful for representing 2D arrays.