CS221
Assembly Language Fundamentals : Irvine Chapter 3

While debug is good for writing very small programs and experimenting with memory, interrupts, and function calls, it is not very good for larger programs. Programmers aren't able to insert new lines of code very easily, reference symbolic names, and other niceties that make programming easier. For this reason we will start to use MASM (the Microsoft Assembler) for most of the rest of the class. MASM is an assembler that has many of the same features that you are probably used to when working with higher-level programming languages.

If your are installing MASM at home on your own computer, see the link from the CS221 web page on "Installing MASM" for help on getting it up and running. In general you have three methods: using the assembler editor, using visual studio as the front-end, or using any editor and then assembling via DOS commands.

Assembly language programs are made up of statements. Each statement may be composed of constants, literals, names, mnemonics, operands, and comments.

Constant Expressions

Numeric literal expressions are represented directly in a program. These may be in scientific notation or not, e.g. the following are valid:

```
100     -100    +100    100.1   10E+2
```

By default, numeric literals in MASM are in decimal. Note that this is different from debug, which used a default of hex. In MASM you have the ability to express numbers in a variety of formats by adding a letter on the end of the literal to indicate the base:

100	decimal
100b	binary
100h	hexadecimal
100q	octal
0FFh	hexadecimal

Note the last example. Hex constants that start with a letter must be preceded by a zero. This is so the assembler doesn't get the hex value confused with a symbolic identifier (e.g., an identifier named "FFh").

We can include mathematical expressions in the constants, e.g.:

```
100 * 2
-3 / 4
1 + 3
```

These expressions are **evaluated at assembly time**, not at runtime. This means in the last example of 1+3, our assembled program will contain the number 4. The assembler does the math, not the program during runtime.

We may wish to refer to a constant value by assigning it a name. We can do this by defining a symbolic constant with the = symbol:

```
pi = 3.14159
rows = 10 * 10
max = 100
```

Although it looks like these are variables, they are not the same! They are constant expressions and may be redefined, but they cannot be used as a storage like a variable can.

Consider the following code fragment:

```
somenum = 0FFh                  ; define constant to FF hex
MOV ah, somenum                 ; move to the AH register
somenum = 0AAh                  ; re-define constant to AA hex
MOV ah, somenum                 ; move to the AH register
```

The above is equivalent to:

```
MOV ah, 0FFh
MOV ah, 0AAh
```

In the above, we redefined the value of the constant somenum. The following code would be invalid:

```
somenum = 0FFh
MOV somenum, ah                 ; INVALID
```

This would be akin to trying to do:

```
MOV 0FFh, ah                    ; Move accumulator to FF? Not valid
                                  since FF is a number, not a storage loc
```

Enclosing the data in either single or double quotation marks can represent character strings. The following are all valid strings. Note embedded quotes:

‘ABC’ ‘Z’ “Z” “Kenrick’s” ‘He said “hi”’ ‘14’

Statements

A statement consists of a name, mnemonic, operands, and comment. There are two types of statements, instructions and directives. **Instructions** are executable statements that includes a mnemonic op code. **Directives** are statements that provide information to the assembler, but do not include executable op codes.

An example of an instruction is the MOV instruction we used previously.
An example of a directive is the redefinable constant, "somenum = 100".

The format for statements is:

```
[name]  [mnemonic]    [operands]   [;comment]
```

These are optional and extra whitespace between columns is ignored.
If you want to continue a long line to the next line, use the backslash character:

```
somenum =  \
55
```

Names

A name identifies a label, variable, symbol, or keyword. It may contain letters, numbers, ?, _, @, or $ and is not case-sensitive. Names may not begin with a number or be a MASM reserved word (e.g., "test" or "mov"). Examples of valid names include "somenum", "somenum55", "_somenum", or "num1".

A **variable** is a location in the program's data area that has been assigned a name. Here is an example that defines a byte named "count1" and initialized the value to 50:

```
count1 db 50
```

A **label** is a name that appears in the code area of a program. A label serves as a placemarker when a program needs to jump or loop back to some other instruction. Rather than use line numbers that might change when instructions are added or removed, labels remain placeholders and the line number they are on is recalculated when the program is assembled. The following is an example of a label:

```
BeginLabel:    mov ax, 0
               mov bx, 0
               …
               jmp BeginLabel              ; go back to BeginLabel
```

Sample Program

Here is the Hello World program from chapter 3 of Irvine:

```
1     title Hello World Program         (hello.asm)
2
3     ; This program displays "Hello, world!"
4
5     .model small
6     .stack 100h
7     .data
8     message db "Hello, world!",0dh,0ah,'$'
9
10    .code
11    main proc
12        mov  ax,@data
13        mov  ds,ax
14
15        mov  ah,9
16        mov  dx,offset message
17        int  21h
18
19        mov  ax,4C00h
20        int  21h
21    main endp
22
23    end main
```

The line numbers have been added only for reference purposes, they are not part of the program.

Line 1 is the title directive and prints the specified title at the top of the listing to identify the program. It is optional and not necessary to include on all programs.

Line 5 is a directive that indicates the memory model. The small memory model is for a program that uses at most 64K for the code, and 64K for the data.

The options are:

Tiny	- Code and data combined < 64K
Small	- Code <=64K, data <=64K
Medium	- Data <=64K, code any size; multiple code segments
Compact	- Code <=64K, data any size; multiple data segments
Large	- Code, Data > 64K. Multiple code, data segments
Huge	- Same as Large but arrays could be > 64K

We can use tiny or small for most of our programs in the class.

Line 6 is the stack directive. It allocates 100 hex (256) bytes of stack space out of our data segment.

Line 7 is the data directive. It marks the beginning of the data segment, where variables are stored.

Line 8 declares a variable called “message” within the data segment. The “db” means “define byte”. This line allocates a block of memory to hold the string containing “Hello, world!” along with two bytes containing the newline character (0dh, 0ah). The ‘$’ is a required string terminator character for the output subroutine used to display a string to the screen.

Line 10 is the code directive. It marks where code begins.

Line 11 declares a procedure called main. PROC marks the beginning of a procedure. The format is to give the procedure name first, followed by PROC.

Line 12-20 are the body of the code for the main procedure. We have already discussed the MOV instruction. The first two lines move the address of the program’s data segment into the DS register. This is needed so we’ll reference the correct offset for our string. The line with “offset message” moves the address of the message variable to the DX register. The first call to INT21H invokes a DOS routine to display the string and the second call to INT21H invokes a DOS routine to terminate the program.

Line 21 marks the end of the main procedure. The format is to give the procedure name first, followed by ENDP.

Line 23 marks the end of the program. The optional word “main” behind it indicates the location of the program entry point.

Assembling a Program

Assembling a program is similar to compiling a program. There are two stages. First, the source code is run through the assembler to produce an object file. The object files contain assembled machine code, but for individual modules. Object files are sometimes distributed as libraries; for example you have Irvine.lib as an object file containing commonly used subroutines. Next, the object files are linked to produce an executable program. The executable program is then run through the DOS loader.

src.asm → assembler → src.o → linker → src.exe → DOS → output
othersrc.o
library.lib

Other files that may be produced along the way are MAP and LISTING files. The listing files are optionally generated during assembly and contain the source code and translated machine code in a printable format. The map files are generated during linking and contain information about the segments that is useful for debugging.

(Step through assembling this program in class – compare size to a similar C++ program!)

Data Allocation Directives

In the hello world program, we defined a string variable named "message". To do this, we used the db directive. Let's look at the data allocation directives in more detail. These directives determine how much storage to allocated based on some predefined types.

DB - Define Byte
DW - Define Word (2 bytes)
DD - Define Doubleword (4 bytes)
DF,DP - Define far pointer (6 bytes)
DQ - Define quadword (8 bytes)
DT - Define tenbytes (10 bytes)

The DB and DW directives will be the ones we use most commonly. DB allocates storage for one or more 8-bit values. The syntax is as follows:

```
[name] DB initial-value [,initialvalue … ]
[name] DW initial-value [, initialvalue …]
```

We can define multiple initial values by separating them with commas. The initial value must be representable in the amount of space allocated. For a byte, this is a value from 0 to 255 or from –128 to 127.

Here are some examples:

```
char1   db      'A'             ; Define the ASCII letter 'A'
char2   db      'A'+1           ; expression, the ASCII for 'B'
x       db      255
x2      db      0FFh
y       dw      +32767
z       dw      12300 * 2
```

A variable's initial contents may be left undefined by using a question mark for the initializer:

```
char1   db      ?
```

When multiple initializers are used, the data is stored sequentially in memory. Consider:

```
numlist         db      10, 20, 30, 40
```

If numlist is stored at memory location 0000, then the value 10 is stored at location 0000, the value 20 is stored at location 0001, the value 30 is stored at 0002, etc.

ASCII Strings can be represented by separating the letters by commas, or by using quotation marks. The following shows a C-style null terminated string:

```
Cstring     db     "Hiya", 0
```

This is equivalent to:

```
Cstring     db     'H','i','y','a',0
```

If you have a very long string, you can continue on multiple lines:

```
Longstr     db     "This long string"
                   "continues on the next line", 0
```

The DW word storage is enough to store an offset memory location for some other label or variable. To do this we can use reference the variable as a word:

```
MyList      dw     10h, 20h, 30h        ; Actual list
PtrToList   dw     MyList               ; PtrToList holds offset of MyList
```

Don't forget about the reverse byte order – when storing words in memory, the low end is stored first, so a value like 0011h will actually have the 11 stored first and then the 00 second.

One final operator used in defining data storage is the **DUP** operator. DUP appears after a storage directive, such as DB, and with it you can duplicate one or more values. It is most often used when allocating space for a string or array:

```
MyString    db     20 dup('A')          ; 20 bytes, all equal to 'A'
MyVar       db     20 dup(?)            ; 20 bytes, all uninitialized
MyVar2      db     3 dup("ABC")         ; 9 bytes, "ABCABCABC"
MyVar3      dw     4 dup(0)             ; 4 words, all zero
MyVar4      dw     3 (dup (4 dup(0)))   ; 12 words, all zero, for 3x4 table
```

The other data storage directives work similarly, but allocate more space. For example we could use:

```
MyBigVar    dd     2147483640           ; Hold a value using 32 bits
PtrToBigVar dd     MyBigVar             ; 32 bit offset addr of MyBigVar
```

Other Directives

A few other directives are at your disposal:

EQU – This assigns a symbolic name to a string or numeric constant. Unlike using the equal sign, a symbol defined with EQU may not be redefined later:

```
Pi      EQU   3.14159
Max     EQU   10000
```

We can now use Pi or Max like constants. The assembler will complain if we try to change either one to something else later.

TEXTEQU – This directive creates a text macro. A sequence of characters is assigned to the macro name, and then use the name later in the program to substitute the sequence of characters. A symbol define with TEXTEQU cannot be redefined later. Enclose the sequence of characters in angle brackets:

```
SomeMsg     TEXTEQU     <"Continue?">
.data
Prompt      db      SomeMsg
```

Most commonly, the text macros are used to encode bits of code itself:

```
Move        TEXTEQU     <mov>
MyPointer   TEXTEQU     <offset myString>

.data
myString    db      "A String", 0

.code
move        bx, MyPointer          ; same as MOV BX, offset myString
```

In this case, we redefined the MOV instruction to MOVE and MyPointer as "offset myString" (which gives the offset of a variable).

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.

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, al          ; Sets AH = AH – AL
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      dw      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:

```
.data
countLabel      dw      1000
.code
mov ax, offset countLabel           ; Moves offset of countLabel into AX, e.g. 0
```

For example, if countLabel is stored at offset 0 of its segment, then the value 0 gets loaded into AX.

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

```
.data
countLabel1     dw      10
countLabel2     dw      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.

Finally, **register indirect** mode is used when a register (either the SI, DI, or the BX register) contains an offset of some memory location. 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    dw    1000
countLabel2    dw    2
.code
mov bx, offset countLabel      ; mov offset of countLabel to bX
mov ax, [bx]                   ; mov 1000 to AX
mov ax, [bx+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.