CS221
Assembly Language Fundamentals : Irvine Chapter 3

While we can write assembly directly in machine code, it is not a very convenient method 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) to write x86 assembly programs for 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. I will assume that you are using TextPad as your editor, although you are free to use any editor and debugging environment.

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
10 mod 6
```

These expressions are **evaluated at assembly time**, not at runtime. This means in the 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 (i.e. copy) 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 high 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 in ASCII. The following are all valid strings. Note embedded quotes:

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

Reserved Words

Just as with high level languages, we have a list of words called *reserved words* that have special meaning. You are not allowed to use reserved words for your identifiers. A sample of reserved words is below:

- Instruction mnemonics such as MOV, ADD, etc.
- Directives
- Attributes such as WORD or BYTe
- Operators
- Predefined symbols such as @data

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 include 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
```

Identifiers

An `identifier` 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
```

Comments

We can use the ; character to comment out everything that comes after the semicolon. To comment out an entire block, use:

```
COMMENT !
        This line is a comment
        So is this line
        …
!
```

Sample Program

Here is a sample program that adds and subtracts numbers from chapter 3 of Irvine:

```
1     title Add and Subtract                    (AddSub.asm)
2     ; This program adds and subtracts 32-bit integers.
3
4     INCLUDE Irvine32.inc
5     .code
6     main PROC
7           mov eax, 10000h              ; EAX = 10000h
8           add eax, 40000h              ; EAX = 50000h
9           sub eax, 20000h              ; EAX = 30000h
10          call DumpRegs                ; Display registers
11          exit
12    main ENDP
13    END main
```

The line numbers have been added for your reference only.

When this program runs it will display the contents of the extended registers. EAX should contain 30000h.

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 4 is an include directive. It tells the assembler to copy definitions from the file Irvine32.inc. This file contains macros and useful procedures written by the author of your textbook to perform common tasks such as display the contents of registers, perform file I/O, etc.

Line 5 is the code directive. It marks where the code segment begins in memory.

Line 6 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 7-11 are the body of the code for the main procedure. We have already discussed the MOV instruction. The ADD instruction adds the operand to the EAX register and puts the result into EAX. The SUB instruction subtracts the operand from the EAX register and puts the result into EAX.

Line 10 invokes the DumpRegs procedure. The CALL statement calls a procedure. In this case, the DumpRegs procedure exists in the Irvine32 library – that is why the source code is not in our program. This procedure will be linked into the executable.

Line 11 invokes a macro called Exit in the Irvine32 library. It provides a simple way to end a program in Windows by invoking a Windows function that halts the program.

Line 12 marks the end of the main procedure. The format is to give the procedure name first, followed by ENDP.

Line 13 marks the end of the program. The optional word “main” behind it indicates the location of the program entry point.

For your programs, you can use this program as your basic template for starting out with assembly programming.

Alternate version of AddSub

By including irvine32.inc, we are hiding a number of details. Here is a description of some of these details shown in this alternate version of the same program:

```
1     title Alternate Add and Subtract          (AddSubAlt.asm)
2     ; This program adds and subtracts 32-bit integers.
3
4     .386
5     .MODEL flat,stdcall
6     .STACK 4096
7     ExitProcess PROTO, dwExitCode: DWORD
8     DumpRegs PROTO
9     .code
10    main PROC
11          mov eax, 10000h               ; EAX = 10000h
12          add eax, 40000h               ; EAX = 50000h
13          sub eax, 20000h               ; EAX = 30000h
14          call DumpRegs                 ; Display registers
15          INVOKE ExitProcess, 0
12    main ENDP
13    END main
```

Line 4 is a directive that specifies the minimum target machine, in this case we get all the opcodes available for the 386. If we used ".286" then we could only use opcodes for the 286 processor.

Line 5 is a directive that indicates the memory model.

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
Flat	- Protected mode. Uses 32-bit offsets for code & data in a single 32-bit segment

We can use flat for most of this class.

STDCALL specifies that procedure arguments must be pushed on the stack in reverse order.

Line 6 is the stack directive. It allocates 4096 bytes of stack space out of our data segment.

Line 7-8 specifies that we will be using two external procedures, ExitProcess and DumpRegs. ExitProcess is a Windows function that halts the current process, while DumpRegs is the Irvine32 procedure that displays the register contents. dwExitCode is used by ExitProcess.

Line 15 invokes the ExitProcess function, passing it a return code of zero. INVOKE is an assembler directive that calls a procedure or function.

Real mode sample program

If we are writing a program that is going to run in real mode then we have a bit of extra work to do. We must initialize our segments and remember that offsets (addresses) of code and data are 16 bits rather than 32 bits.

Here is a sample Hello World program in real mode:

```
1     title Hello World Program         (hello.asm)
2
3     ; This program displays "Hello, world!"
4
5     INCLUDE Irvine16.inc
6     .data
7     message BYTE "Hello, world!",0
8     .code
9     main proc
10        mov  ax,@data
11        mov  ds,ax
12
13        mov dx, offset message
14        call WriteString
15        exit
16    main endp
17    end main
```

Line 5 is a directive that indicates we are referencing the 16 bit Irvine library.
More specifically, this declares the small memory model, a stack of 4096 bytes, and uses the .386 processor directive.

Line 8 declares a variable called “message” within the data segment. The “BYTE” means we will declare this variable in chunks of bytes. We can also use “DB” in place of “BYTE” for “Define Byte”. This line allocates a block of memory to hold the string containing “Hello, world!” along with a byte containing the value 0 (NULL) which is delimiter that indicates where the string terminates. If we leave this off, the string procedures will think everything in memory past this string is part of the string, until we happen to hit a zero.

Lines 10-11 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 (remember we are in Real Mode! Addresses are created by combining an offset with a segment register).

Line 13 with “offset message” moves the address of the message variable to the DX register.

Line 14, the WriteString procedure, assumes that the address of the string to write is stored in the DX register. This routine displays all data as ASCII to the screen until the null character is reached.

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/Windows loader.

src.asm →assembler→ src.o → linker→ src.exe→OS→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 symbols, segments, and their respective addresses.

(To do - compare size of assembled programs to a similar C++ or Java program!)

Data Allocation Directives

In the hello world program, we defined a string variable named "message". To do this, we used the BYTE 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.

BYTE or DB	-	Define Byte
SBYTE	-	Signed Byte
WORD or DW	-	Define Word (2 bytes)
SWORD	-	Signed Word
DWORD or DD	-	Define Doubleword (4 bytes)
SDWORD	-	Signed Doubleword
FWORD or DF	-	Define far pointer (6 bytes)
QWORD or DQ	-	Define quadword (8 bytes)
TBYTE or DT	-	Define tenbytes (10 bytes)
REAL4	-	32 bit IEEE real
REAL8	-	64 bit IEEE real

The BYTE, WORD, and DWORD 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] BYTE initial-value [,initialvalue …]
[name] WORD 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   byte    'A'             ; Define the ASCII letter 'A'
char2   byte    'A'+1           ; expression, the ASCII for 'B'
x       byte    255
x2      byte    0FFh
y       word    +32767
z       word    12300 * 2
```

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

```
char1   byte    ?
```

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

```
numlist         byte    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         byte    "Hiya", 0
```

This is equivalent to:

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

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

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

The DWORD 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          BYTE        10h, 20h, 30h  ; Actual list
PtrToList       DWORD       MyList         ; PtrToList holds address 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    byte    20 dup('A')          ; 20 bytes, all equal to 'A'
MyVar       byte    20 dup(?)            ; 20 bytes, all uninitialized
MyVar2      byte    3 dup("ABC")         ; 9 bytes, "ABCABCABC"
MyVar3      word    4 dup(0)             ; 4 words, all zero
MyVar4      word    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    DWORD    2147483640   ; Hold a value using 32 bits
PtrToBigVar DWORD    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).