Introduction to 8086 Programming

(The 8086 microprocessor is one of the family of 8086,80286,80386,80486,Pentium,PentiumI,II,III …. also referred to as the X86 family.)

Learning any imperative programming language involves mastering a number of common concepts:

Variables:	declaration/definition
Assignment:	assigning values to variables
Input/Output:	Displaying messages
	Displaying variable values
Control flow:	if-then
	Loops
Subprograms:	Definition and Usage

Programming in assembly language involves mastering the same concepts and a few other issues.

Variables
For the moment we will skip details of variable declaration and simply use the 8086 registers as the variables in our programs. Registers have predefined names and do not need to be declared.

The 8086 has 14 registers. Each of these is a 16-bit register. Initially, we will use four of them – the so called the general purpose registers:

ax, bx, cx, dx

These four 16-bit registers can also be treated as eight 8-bit registers:
ah, al, bh, bl, ch, cl, dh, dl

Assignment

In Java, assignment takes the form:

```
x = 42 ;
y = 24;
z = x + y;
```

In assembly language we carry out the same operation but we use an instruction to denote the assignment operator ("=" in Java). The above assignments would be carried out in 8086 assembly langauge as follows

```
mov     x, 42
mov     y, 24
add     z, x
add     z, y
```

The **mov** instruction carries out assignment.

It which allows us place a number in a register or in a memory location (a variable) i.e. it assigns a value to a register or variable.

Example: **Store the ASCII code for the letter A in register bx.**

```
        mov     bx,  'A'
```

The **mov** instruction also allows you to copy the contents of one register into another register.

Example:

```
                mov             bx, 2
                mov             cx, bx
```

The first instruction loads the value 2 into bx where it is stored as a binary number. [a number such as 2 is called an **integer** constant]

The Mov instruction takes two **operands,** representing the ***destination*** where data is to be placed and the ***source*** of that data.

General Form of Mov Instruction

```
        mov             destination,  source
```

where *destination* must be either a register or memory location and
source may be a constant, another register or a memory location.

Note: The comma is essential. It is used to separate the two operands.

A missing comma is a common syntax error.

Comments

Anything that follows semi-colon (**;**) is ignored by the assembler. It is called a **comment**. Comments are used to make your programs readable. You use them to explain what you are doing in English.

More 8086 Instructions

add, inc, dec and sub instructions

The 8086 provides a variety of arithmetic instructions. For the moment, we only consider a few of them. To carry out arithmetic such as addition or subtraction, you use the appropriate instruction.

In assembly language you can only carry out a single arithmetic operation at a time. This means that if you wish to evaluate an expression such as :

z = x + y + w – v

You will have to use 3 assembly language instructions – one for each arithmetic operation.

These instructions combine assignment with the arithmetic operation.

Example:

```
mov        ax, 5     ;    load 5 into ax

add    ax, 3     ;    add 3 to the contents of ax,
                 ;    ax now contains 8

inc    ax        ;    add 1 to ax
                 ;    ax now contains 9

dec    ax        ;    subtract 1 from ax
                 ;    ax now contains 8

sub    ax, 6     ;    subtract 4 from ax
                 ;    ax now contains 2
```

The **add** instruction adds the source operand to the destination operand, leaving the result in the destination operand.

The destination operand is always the first operand in 8086 assembly language.

The **inc** instruction takes one operand and adds 1 to it. It is provided because of the frequency of adding 1 to an operand in programming.

The **dec** instruction like **inc** takes one operand and subtracts 1 from it. This is also a frequent operation in programming.

The **sub** instruction subtracts the source operand from the destination operand leaving the result in the destination operand.

Exercises:

1) Write instructions to:

    Load character ? into register bx
    Load space character into register cx
    Load 26 (decimal) into register cx
    Copy contents of ax to bx and dx

2) What errors are present in the following :

```
mov     ax   3d
mov     23,  ax
mov     cx, ch
move    ax, 1h
add     2, cx
add     3, 6
inc     ax, 2
```

3) Write instructions to evaluate the arithmetic expression 5 + (6-2) leaving the result in ax using (a) 1 register, (b) 2 registers, (c) 3 registers

4) Write instructions to evaluate the expressions:

a = b + c –d

z = x + y + w – v +u

5) Rewrite the expression in 4) above but using the registers ah, al, bh, bl and so on to represent the variables: a, b, c, z, x, y, w, u, and v.

Implementing a loop: The **jmp** instruction

```
Label_X:    add ax, 2
            add bx, 3
            jmp Label_X
```

The jmp instruction causes the program to start executing from the position in the program indicated by the label Label_X. This is an example of an endless loop.

We could implement a while loop using a conditional jump instruction such as JL which means jumi-if-less-than. It is used in combination with a comparision instruction – cmp.

```
            mov ax, 0
Label_X:    add ax, 2
            add bx, 3
            cmp ax, 10
            jl Label_X
```

The above loop continues while the value of ax is less than 10. The cmp instruction compares ax to 0 and records the result. The jl instruction uses this result to determine whether to jump to the point indicated by Label_X.

Input/Output

Each microprocessor provides instructions for I/O with the devices that are attached to it, e.g. the keyboard and screen.

The 8086 provides the instructions in for input and out for output. These instructions are quite complicated to use, so we usually use the operating system to do I/O for us instead.

In assembly language we must have a mechanism to call the operating system to carry out I/O.

In addition we must be able to tell the operating system what kind of I/O operation we wish to carry out, e.g. to read a character from the keyboard, to display a character or string on the screen or to do disk I/O.

In 8086 assembly language, we do not call operating system subprograms by name, instead, we use a software interrupt mechanism

An interrupt signals the processor to suspend its current activity (i.e. running your program) and to pass control to an interrupt service program (i.e. part of the operating system).

A software interrupt is one generated by a program (as opposed to one generated by hardware).

The 8086 **int** instruction generates a software interrupt.

It uses a single operand which is a number indicating which MS-DOS subprogram is to be invoked.

For I/O and some other operations, the number used is **21h**.

Thus, the instruction **int 21h** transfers control to the operating system, to a subprogram that handles I/O operations.

This subprogram handles a variety of I/O operations by calling appropriate subprograms.

This means that you must also specify which I/O operation (e.g. read a character, display a character) you wish to carry out. This is done by placing a specific number in a register.

The *ah* register is used to pass this information.

For example, the subprogram to display a character is subprogram number **2h**.

This number must be stored in the ah register. We are now in a position to describe character output.

When the I/O operation is finished, the interrupt service program terminates and our program will be resumed at the instruction following *int*.

3.3.1 Character Output

The task here is to display a single character on the screen. There are three elements involved in carrying out this operation using the *int* instruction:

1. We specify the character to be displayed. This is done by storing the character's ASCII code in a specific 8086 register. In this case we use the **dl** register, i.e. we use dl to pass a parameter to the output subprogram.

2. We specify which of MS-DOS's I/O subprograms we wish to use. The subprogram to display a character is subprogram number **2h**. This number is stored in the ah register.

3. We request MS-DOS to carry out the I/O operation using the int instruction. This means that we **interrupt** our program and transfer control to the MS-DOS subprogram that we have specified using the ah register.

Example 1: Write a code fragment to display the character 'a' on the screen:

```
mov     dl, 'a'     ; dl = 'a'
mov     ah, 2h      ; character output subprogram
int     21h         ; call ms-dos output character
```

As you can see, this simple task is quite complicated in assembly language.

3.3.2 Character Input

The task here is to read a single character from the keyboard. There are also three elements involved in performing character input:

1. As for character output, we specify which of MS-DOS's I/O subprograms we wish to use, i.e. the character input from the keyboard subprogram. This is MS-DOS subprogram number **1h**. This number must be stored in the ah register.

2. We call MS-DOS to carry out the I/O operation using the int instruction as for character output.

3. The MS-DOS subprogram uses the al register to store the character it reads from the keyboard.

Example 2: Write a code fragment to read a character from the keyboard:

```
mov     ah, 1h    ; keyboard input subprogram
int     21h       ; character input
                ; character is stored in al
```

The following example combines the two previous ones, by reading a character from the keyboard and displaying it.

Example 3: Reading and displaying a character:

```
mov     ah, 1h    ; keyboard input subprogram
int     21h       ; read character into al

mov     dl, al    ; copy character to dl

mov     ah, 2h    ; character output subprogram
int     21h       ; display character in dl
```

A Complete Program

We are now in a position to write a complete 8086 program. You must use an **editor** to enter the program into a file. The process of using the editor (**editing**) is a basic form of word processing. This skill has no relevance to programming.

We use Microsoft's MASM and LINK programs for assembling and linking 8086 assembly language programs. MASM program files should have names with the **extension** (3 characters after period) ***asm***.

We will call our first program ***prog1.asm***, it displays the letter 'a' on the screen. (You may use any name you wish. It is a good idea to choose a meaningful file name). Having entered and saved the program using an editor, you must then use the MASM and LINK commands to translate it to machine code so that it may be executed as follows:

```
C> masm prog1
```

If you have syntax errors, you will get error messages at this point. You then have to edit your program, correct them and repeat the above command, otherwise proceed to the link command, pressing Return in response to prompts for file names from masm or link.

```
H:\> link prog1
```

To execute the program, simply enter the program name and press the Return key:

```
H:\> prog1
a
H:\>
```

Example 4: A complete program to display the letter ‘a‘ on the screen:

```
; prog1.asm: displays the character ‘a’ on the screen
; Author: Joe Carthy
; Date:   March 1994

          .model small
          .stack 100h

          .code
start:
          mov     dl, ‘a’     ; store ascii code of ‘a’ in dl

          mov     ah, 2h      ; ms-dos character output function
          int     21h         ; displays character in dl register

          mov     ax, 4c00h ; return to ms-dos
          int     21h
          end start
```

The first three lines of the program are comments to give the name of the file containing the program, explain its purpose, give the name of the author and the date the program was written.

The first two are directives, ***.model*** and ***.stack***. They are concerned with how your program will be stored in memory and how large a stack it requires. The third directive, ***.code***, indicates where the program instructions (i.e. the program code) begin.

For the moment, suffice it to say that you need to start all assembly languages programs in a particular format (not necessarily that given above. Your program must also finish in a particular format, the end directive indicates where your program finishes.

In the middle comes the code that you write yourself.

You must also specify where your program starts, i.e. which is the **first** instruction to be executed. This is the purpose of the label, ***start***.

(Note: We could use any label, e.g. *begin* in place of *start*).

This same label is also used by the ***end*** directive.

When a program has finished, we return to the operating system. Like carrying out an I/O operation, this is also accomplished by using the int instruction. This time MS-DOS subprogram number **4c00h** is used.

It is the subprogram to terminate a program and return to MS-DOS. Hence, the instructions:

```
    mov     ax, 4c00h     ; Code for return to MS-DOS
    int     21H           ; Terminates program
```

terminate a program and return you to MS-DOS.

Time-saving Tip

Since your programs will start and finish using the same format, you can save yourself time entering this code for each program. You create a template program called for example, template.asm, which contains the standard code to start and finish your assembly language programs. Then, when you wish to write a new program, you copy this template program to a new file, say for example, prog2.asm, as follows (e.g. using the MS-DOS copy command):

```
H:\> copy template.asm prog2.asm
```

You then edit prog2.asm and enter your code in the appropriate place.