Introduction to 8086 Programming

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.

We have seen that the 8086 has 14 registers. 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:

\[
\begin{align*}
x &= 42; \\
y &= 24; \\
z &= x + y;
\end{align*}
\]

In assembly language we carry out the same operation but we use an instruction to denote the assignment operator (“=” in Java).

\[
\begin{align*}
\text{mov} & \quad x, 42 \\
\text{mov} & \quad y, 24 \\
\text{add} & \quad z, x \\
\text{add} & \quad z, y
\end{align*}
\]

The \texttt{mov} instruction carries out assignment in 8086 assembly language.

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.

A has ASCII code 65D (01000001B, 41H)

The following \texttt{mov} instruction carries out the task:

\texttt{mov bx, 65d}
We could also write it as:

\[
\text{mov } \text{bx, } 41h \\
\text{or } \text{mov } \text{bx, } 01000001b \\
\text{or } \text{mov } \text{bx, } \text{‘A’}
\]

All of the above are equivalent. They each carry out exactly the same task, namely the binary number representing the ASCII code of A is copied into the bx register.

The value is copied into the right-hand side (low-order byte) of the register. The left-hand side will contain all 0’s.

Thus we could also have written it as:

\[
\text{mov } \text{bl, } 65d \\
\text{mov } \text{bl, } \text{‘A’}
\]

Since register bl represents the low-order byte of register bx.

**Note:** The 8086 Assembler converts a **character constant** i.e. a character in single quotes (e.g. ‘A’) to its ASCII code automatically. This is a very useful feature and it means that you can specify many characters without having to look up their ASCII code. You simply enclose the character in single quotes. You will have to use the
ASCII code for control characters such as carriage return and line feed.

**Notation**

`mov` is one of the many 8086 instructions that we will be using. Most assembly language books use uppercase letters to refer to an instruction e.g. MOV.

However, the assembler will also recognise the instruction if it is written in lowercase or in mixed case e.g. Mov. (In fact, the assembler converts all instructions to uppercase).

It is *my personal* preference to use lower case when writing programs. You may write your programs using which ever notation you find convenient, but you should be consistent and stick to one particular style.

**More about mov**

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

\[ \text{mov } \text{destination}, \text{source} \]

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

In 8086 assembly language, the source and destination cannot both be memory locations in the same instruction.

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

A missing comma is a common syntax error.

We will look at manipulating data in memory at a later stage.

More Examples

The following instructions result in registers ax, bx, and cx all having the value 4:

\[ \text{mov bx, 4} \quad ; \text{copy number 4 into register bx} \]
\[ \text{mov ax, bx} \quad ; \text{copy contents of bx into register ax} \]
\[ \text{mov cx, ax} \quad ; \text{copy contents of ax into register cx} \]
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.

It is recommended that you use comments frequently in your programs, not only so that others can understand them, but also for yourself, when you look back at programs you have previously written.

Every programming language has a facility for defining comments.

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 instruction combine assignment with the arithmetic operation.

Example:

```assembly
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.

(In M68000 assembly language, it is the other way round i.e. the source operand is always the first operand e.g. `move #10, x`)

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.

Some microprocessors do not provide instructions for multiplication or division (e.g. the M6800). With such microprocessors, multiplication and division have to be programmed using repeated additions and subtractions and shift operations (which will be discussed later).

The 8086 provides **mul** and **div** (and others) for multiplication and division.

**Ambiguity**

Suppose you wish to load the hexadecimal value A (decimal 10) written as **ah** in the register bl.

You might be tempted to write:

```
mov bl, ah
```

But we have already seen that there is a register called **ah** (the high-order byte of ax) and so the above does not do what we intend. Instead it copies the contents of register ah into bl. In order to avoid ambiguity when writing hexadecimal numbers that begin with a letter we prefix them with 0. Thus we write:

```
mov bl, 0ah
```

; copy hex number ah into bx
It is common practice to write decimal numbers with the letter D appended so as to distinguish them from hexadecimal.

The 8086 assembler take all numbers to be decimal numbers unless there is a B (binary), H (hex) or O (octal) appended to them.

**Note:**

When data is moved to a register, all 16 bits (or 8 bits) are given a value. The assembler will automatically fill in 0’s on the left-hand side.

Example:

```
mov bx, 42h ; copy 42 hex into bx
```

42H is 100 0001 in binary. This padded out with nine 0-bits on the left-hand side to fill all 16-bits of the register.

```
BX  0000 0000 0100 0001
    15 0
```

The effect of executing MOV BX, 41H is to overwrite the BX register with 41H in binary.
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.