Chapter 6: The Assembly Process

This chapter will present a brief introduction to the functioning of a standard two–pass assembler. It will focus on the conversion of the text of an assembly language program, written in a form that humans can read, into a machine language program that can be loaded into memory and executed by a computer.

Particular attention will be placed on a logical construct called the **location counter**. This is a counter used to allocate memory addresses to each line of machine code, whether it be an executable instruction or a location reserved to store some value.

This chapter will then briefly discuss the process of loading and executing an assembly language program. It will close with a few comments on the essential differences between a compiler processing a high level program and an assembler processing its program.

The Sample Program Fragment

We shall focus on a fragment of assembly language code for the single high–level line of code that adds three variables and places the result in a fourth variable.

```
W = X + Y + Z
```

Before giving the assembly language equivalent of this code fragment, we note that addition is basically a dyadic instruction; it takes two arguments and produces a single result. Due to the lack of a primitive three–input addition instruction, the above code fragment will be processed somewhat as if it were the following lines of high–level code.

```
W = X
W = W + Y
W = W + Z
```

Here is a fragment of assembly language code that includes a translation of the above high–level code. This discussion focuses on allocation of addresses to items in the assembly language; thus it will use a number of concepts without sufficient explanation.

```
          BALR 12,0     LOAD REGISTER 12 WITH CURRENT ADDRESS
          USING *,12    AND USE IT AS A BASE FOR ADDRESSING.
          L  5,X        VALUE OF X INTO REGISTER 3
          A  5,Y        ADD VALUE OF Y TO REGISTER 5
          A  5,Z        ADD VALUE OF Z TO REGISTER 5
          ST 5,W        STORE SUM INTO W
W         DC F'0'       W HAS AN INITIAL VALUE OF 0
X         DC F'1'       X HAS AN INITIAL VALUE OF 1
Y         DC F'2'       Y HAS AN INITIAL VALUE OF 2
Z         DC F'3'       Z HAS AN INITIAL VALUE OF 3
```

At this point, we must notice a major flaw in the code. Every line of the code is correct, and formatted in the standard style. Every instruction will execute correctly. The difficulty will arise after the "**ST 5,W**" has been executed, placing a value of 6 into location **W**.

The program has no STOP or branch statement, so it will start executing the data.

The Two–Pass Assembler

We now examine the action of a two–pass assembler on the above code fragment. Roughly speaking, the functions of the passes are as follows.

Pass 1 This associates addresses with the various labels used. The code fragment above uses four labels: W, X, Y, and Z.

Pass 2 Uses these addresses to generate the machine language code for each line of assembly language.

The assembler assigns addresses by use of the **location counter**, here abbreviated **LC**, to track the size of the address space already allocated. Later discussions of the LC will reflect on its explicit use within the text of an assembler language program and treat it almost as a general purpose register (which it is not), but this discussion treats it as a conceptual device.

We shall now trace the operation of the assembler on these program statements, generally considering only one line at a time. The student should recall that some of these concepts, especially the handling of the first two lines, will be explained very fully at a later time.

The first two lines of the code above should be viewed almost as non–executable. These are used to "establish addressability" in the standard parlance. In effect, they tell the assembler how to set an initial value for the location counter.

```
BALR 12,0      LOAD REGISTER 12 WITH CURRENT ADDRESS
USING *,12     AND USE IT AS A BASE FOR ADDRESSING.
```

Recalling that addresses issued by the assembler may be adjusted when the program is loaded into memory, we note that the affect of this pair of lines is to set the LC.

Pass 1 processes the third line.

```
L  5,X          VALUE OF X INTO REGISTER 3
```

At this point, we have LC = 0, as it was just initialized. The instruction will be assigned to address 0. The instruction is a type RX, requiring four bytes for its storage.

Pass 1 processes line 4

```
A  5,Y          ADD VALUE OF Y TO REGISTER 5
```

Since the previous instruction requires 4 bytes, the location counter has been incremented, so that LC = 4 and this instruction will be placed at address 4. This instruction is also a type RX, and also required four bytes. The next instruction will be at address 8.

Pass 1 processes line 5

```
A  5,Z          ADD VALUE OF Z TO REGISTER 5
```

Since the previous instruction requires 4 bytes, the location counter has been incremented, so that LC = 8 and this instruction will be placed at address 8. This instruction is also a type RX, and also required four bytes. The next instruction will be at address 12 (decimal).

Pass 1 processes line 6

```
ST 5,W          STORE SUM INTO W
```

Since the previous instruction requires 4 bytes, the location counter has been incremented, so that LC = 12 and this instruction will be placed at address 12. This instruction is also a type RX, and also required four bytes. The next instruction will be at address 16 (decimal).

Pass 1 processes line 7

```
W           DC F'0'          W HAS AN INITIAL VALUE OF 0
```

Since the previous instruction requires 4 bytes, the location counter has been incremented, so that LC = 16. This line is not an instruction, but is a **declarative**, used to define a label and allocate storage space for a value to be associated with that label.

This line declares space to store a 32–bit fullword, so it allocates four bytes for storage. The next line will be associated with address 20 (decimal). This line defines the symbol **W** as being associated with address 16 (decimal).

Pass 1 processes line 8

```
X           DC F'1'          X HAS AN INITIAL VALUE OF 1
```

Since the previous declarative requires 4 bytes, the location counter has been incremented, so that LC = 20, and the label **X** will be associated with address 20 (decimal).

This line also declares space to store a 32–bit fullword, so it allocates four bytes for storage. The next line will be associated with address 24 (decimal).

Pass 1 processes line 9

```
Y           DC F'2'          Y HAS AN INITIAL VALUE OF 2
```

Since the previous declarative requires 4 bytes, the location counter has been incremented, so that LC = 24, and the label **Y** will be associated with address 24 (decimal).

This line also declares space to store a 32–bit fullword, so it allocates four bytes for storage. The next line will be associated with address 28 (decimal).

Pass 1 processes line 10

```
Z           DC F'3'          Z HAS AN INITIAL VALUE OF 3
```

Since the previous declarative requires 4 bytes, the location counter has been incremented, so that LC = 28, and the label **Z** will be associated with address 28 (decimal).

One of the key outputs of pass 1 is a table associating each label with it address. Up to now, we have been using decimal addresses. At this point, we must translate to hexadecimal.

Label	Address	
	Decimal	Hexadecimal
W	16	0x10
X	20	0x14
Y	24	0x18
Z	28	0x1C

One should note that our simple code sample is atypical in one regard. All labels in this fragment are associated with values to be processed. A more typical program would have labels associated with executable statements. The processing will be the same; as each line is processed, the value of the LC will be associated with any label found on the line.

Pass 2 of the assembler will generate the binary machine code associated with each line and prepare it for loading into the computer memory. At this point, we note another artificiality of the simple program fragment; each line corresponds to exactly four bytes. As we shall see later, this is rarely the case for IBM S/370 Assembler Language.

Loading the Machine Code

Before considering the actions of a loader, it would help to give a representation of the results of pass 1 of the assembler. The following listing gives the essence of what was done.

```
Address   Assembly Statement
0000              L  5,X
0004              A  5,Y
0008              A  5,Z
000C              ST 5,W
0010      W       DC F'0'
0014      X       DC F'1'
0018      Y       DC F'2'
001C      Z       DC F'3'
```

Note that the addresses listed here are generated by the assembler and relative to the address assigned to the first one listed here. When the code is loaded into memory and made ready to execute, each of these statements will be moved to another address. One of the jobs of the linking loader is to adjust these addresses.

Suppose that the program were loaded into address 0x1400 of real memory. Here is what the memory map, as seen by the operating system, would appear to be.

```
Address   Assembly Statement
1400              L  5,X
1404              A  5,Y
1408              A  5,Z
140C              ST 5,W
1410      W       DC F'0'
1414      X       DC F'1'
1418      Y       DC F'2'
141C      Z       DC F'3'
```

There are two approaches to handling this issue of addressing, which arises from the fact that the assembler addresses are different from those physical memory addresses into which the program is loaded. The first one relates to altering the machine code generated by pass 2 of the assembler. Since the System/370 does not use this approach, we shall not discuss it.

The second approach depends on the fact that relocating a fragment of code does not change the addresses of any line in the fragment relative to the addresses of any other fragment. Consider each of the two fragments of code listed above. Simple subtraction will show that the address of each statement relative to the first one listed remains constant.

This observation forms the basis of the addressing modes used by the IBM System/370. Consider now the pair of code lines that we have not discussed.

```
          BALR 12,0     LOAD REGISTER 12 WITH CURRENT ADDRESS
          USING *,12    AND USE IT AS A BASE FOR ADDRESSING.
```

The effect of these two lines is to load the absolute memory address of the line of code following the second statement into general purpose register 12, and make all address references relative to that stored value.

Pass 2 of the Assembler

The primary function of the second pass of the assembler is to emit binary machine language. Here we shall jump ahead a bit and show the code generated. The student should be aware that all of this will be explained in great detail in a later chapter; the point now is just to see the association of machine code with addresses.

All of the instructions used in this program fragment are what is called "Type RX". Instructions in this format require four bytes, represented as eight hexadecimal digits. The format of such an instruction is shown below.

Byte 1	Byte 2	Byte 3	Byte 4
OP	R X	B D	D D

The first byte is the opcode for the instruction, represented as two hexadecimal digits. For our instructions they are

L	Load Register	0x58
A	Add to Register	0x5A
ST	Store Register	0x50

The second byte contains two hexadecimal digits. The first is for the affected register, here we are using 5. The second is for an index register. As this example does not use indexed addressing, the value placed in this digit is 0.

The next two bytes contain the base register used, represented by a single hexadecimal digit. Here we specify 0xC, because register 12 is used as a base. The "DDD" represents a displacement address given as three hexadecimal digits.

Here is the output of pass 2 of the assembler.

```
Address    Contents     Assembly Statement
0000     58 50 C0 10            L  5,X
0004     5A 50 C0 14            A  5,Y
0008     5A 50 C0 18            A  5,Z
000C     50 50 C0 1C            ST 5,W
0010     00 00 00 00   W        DC F'0'
0014     00 00 00 01   X        DC F'1'
0018     00 00 00 02   Y        DC F'2'
001C     00 00 00 03   Z        DC F'3'
```

A few points should be clear in the listing above.

1. The executable instructions produce code that follows a standard format that is known to the assembler.
2. Each constant, here defined by a DC declarative, is just transformed into a hexadecimal number of the proper length.
3. This code gets around the issue of absolute memory addressing by making all addresses relative to the first one in the list.
4. This code lacks any proper termination code, so that it will simply attempt to execute the data. This will cause problems.

Compilers vs. Assemblers: Some Preliminary Remarks

At this point, it would be helpful to note a few differences between the actions of a compiler and the actions of a typical assembler. Each takes a program written in text written by humans and emits a machine language program that is executable on a computer.

The major difference between a compiled language and an assembled language has to do with the construct called a variable in a high–level language. While it is tempting to refer to data labels in assembler code as variables, this is not strictly correct.

The idea of a high–level language variable refers to a binding of three different attributes associated with the label: the storage location, the data type, and the value. It is important to recall that the compiler manages much of this automatically, following some directions that are found in the code. Consider the following Java code, almost correct and roughly equivalent to the assembly code we have just seen.

```
int w = 0;
int x = 1;
int y = 2;
int z = 3;

w = x + y + z;
```

In this language we have true variables. Note that the variables are simply declared and given initial values. There is no need (or ability) to specify the storage locations assigned to each. The compiler will do that automatically.

In the above Java code, it is obvious that each variable is declared as an integer. For this reasons, the "+" sign in the line of code is interpreted by the compiler as integer addition. Put another way, the declaration of the variable type defines the operations applied to it.

The situation in assembler coding is quite different. Each label (what might very loosely be called a variable) must have its storage allocation specifically in the code. Furthermore, each label must be declared in a location that will not interfere with the execution of the code. The student will note this as an acknowledged failure of the assembly code fragment above.

The second major difference between assembler code and high–level code is that, in assembler, the data declarations do not define the operations. Each operation is specific to a data type and can be applied to another data type with amusing results.

Consider the following slight modification of the assembly code above.

```
          LH   5,X          VALUE OF X INTO REGISTER 3
          AH   5,Y          ADD VALUE OF Y TO REGISTER 5
          AH   5,Z          ADD VALUE OF Z TO REGISTER 5
          STH  5,W          STORE SUM INTO W
W         DC   F'0'         W HAS AN INITIAL VALUE OF 0
X         DC   F'1'         X HAS AN INITIAL VALUE OF 1
Y         DC   F'2'         Y HAS AN INITIAL VALUE OF 2
Z         DC   F'3'         Z HAS AN INITIAL VALUE OF 3
```

The data are still declared as 32–bit fullwords, but the operations are specific to 16–bit halfwords. The code will execute, and the answer will be 0.

Since I have mentioned this possibility of mismatching the assembler instruction with the data type of the operands, I think that I should explain what is going on. Recall the address assignments made by the assembler: W at 0x10, X at 0x14, Y at 0x18, and Z at 0x1C.

Consider the hexadecimal representation of the 32–bit number stored at address X. It is **`00 00 00 01`**. Here is the memory map, in which the addresses are byte addresses.

Address	Contents
0x14	00
0x15	00
0x16	00
0x17	01

We now consider the effect of two different instructions.

L 5, X Go to address 0x14. Get the four bytes at addresses 0x14, 0x15, 0x16 and 0x17. Treat these as a 32–bit integer and load into register 5.

LH 5, X Go to address 0x14. Get the two bytes at addresses 0x14 and 0x15. Treat these as a 16–bit integer, sign extend to 32 bits, and load into register 5.

Note that the two–byte value stored at address 0x14 is **`00 00`**, which will be interpreted as decimal 0. The same is true for the values at addresses Y and Z.

To repeat himself, the author invites the student to note the names used in this part of the discussion. The symbol "X", which would be called a variable in a high–level language, will be referred to either as an address or as a label. In assembler language, it is never exactly correct to speak of a variable, though the term does have some valid informal use.

As another obvious example, consider the assembler language that might translate the high–level language statement **`Y = X`**.

```
L  6, X   LOAD REGISTER 6 WITH THE VALUE AT ADDRESS X.
ST 6, Y   STORE THAT VALUE INTO ADDRESS Y.
```