Chapter 24: Some Compilation Examples

In this chapter, we shall examine the output of a few compilers in order to understand the assembler language code emitted by those compilers. We study this assembler code in order to understand the structure of compilers and gain a deeper understanding of how to use them.

The high–level languages to be considered in this chapter are mostly the older and less used languages, such as FORTRAN and COBOL. The reason for this choice is that they are easier to discuss and do make the points that are the focus of this chapter.

Variable Type

We start with an immediate distinction between high–level languages and assembler language, and then proceed to investigate the implications. The simple, true, and important statement that forms the basic of this chapter is quite simple. Here it is.

Compiled languages use variables; assembler language does not.

Put another way, this chapter focuses on the simple question "What is a variable, and why is it not proper to assume that an assembler language does not use variables?" In order to study this, we must first discuss the idea of labels as used by an assembler and see how these labels are generalized into variables as used by a high–level compiled language.

Assembler language evolved from machine language, which at its basic form is represented as a sequence of binary numbers. Most discussions of machine language employ hexadecimal notation (as pure binary is hard to read) and move towards Assembler Language by substituting mnemonics for binary (or hexadecimal) operation codes. We shall use this hybrid notation in order to investigate the use of labels by Assembler Language.

In our earlier discussions of IBM® 370 Assembler Language, we have mentioned the idea of labels and explained their usage. In this discussion of labels, we shall find it a bit easier to first discuss them within the context of an extremely simple (and fictional) assembly language, such as that for the MARIE, developed by Linda Null and Julia Lobur for their excellent textbook **The Essentials of Computer Organization and Architecture**. This book is published by Jones and Bartlett (Sudbury, MA). The version used as the basis for these notes was published in 2003, with ISBN 0 – 7637 – 2585 – 4.

The MARIE is a single accumulator design, with a very simple instruction set. This single accumulator holds the results of any input operation, as well as the results of any load from memory or arithmetic operation. Here is a table describing the basic instruction set, copied from the textbook by Null and Lobur.

Instruction Number		Instruction	Meaning
Bin	Hex		
0001	1	Load *X*	Load the contents of address *X* into AC.
0010	2	Store *X*	Store the contents of AC at address *X*.
0011	3	Add *X*	Add the contents of address *X* to AC and store the result in AC.
0100	4	Subt *X*	Subtract the contents of address *X* from AC and store the result in AC.
0101	5	Input	Input a value from the keyboard into AC.
0110	6	Output	Output the value in AC to the display.
0111	7	Halt	Terminate the program.
1000	8	Skipcond	Skip the next instruction on condition.
1001	9	Jump *X*	Load the value of *X* into PC.

The MARIE uses 16–bit words, and has 16–bit instructions. The instructions have a uniform 4–bit operation code and possibly 12 bits for operand address; one hexadecimal digit to denote the operation and three hexadecimal digits to denote the address.. While this architecture is extremely restrictive, it suffices to present an excellent example of a stored program computer. More to the point, it exactly illustrates the points important to this chapter. For this reason, our early examples are based on the MARIE.

Consider now the following simple program written in MARIE assembler language. Note that these notes assume that anything following "//" in a line is a comment; in this way it follows the syntax of Java, C++, and possibly the MARIE assembler.

```
LOAD   X    // Value in X is placed into the accumulator
ADD    Y    // Add value in Y to that in the accumulator
STORE  Z    // Store value into location Z.
HALT        // Stop the computer.
```

In a FORTRAN program, the equivalent statement would be **`Z = X + Y`**.

In order to understand the point of this chapter, we must give a plausible machine language rendition of the above simple assembler language program. In order to read this, we must recall the following operation codes, which are each single hexadecimal digits.

0x1	LOAD
0x2	STORE
0x3	ADD
0x7	HALT

Here is the machine language program, rendered with hexadecimal digits. While comments never form a part of a machine language program, your author indulges himself a bit here.

```
1402     // Load the accumulator from address 0x402
3404     // Add the contents of address 0x404
2406     // Store the results into address 0x406
7000     // Stop the computer.  The contents of the right
         // three digits, here "000", are irrelevant.
```

In the very first era of computer programming (late 1940's), the above machine language program was the standard. The programmer had to reserve specific addresses in the memory for data storage, and be sure that these were properly used. This became tedious very quickly. Almost immediately, assembler language (also called "assembly language") was developed and used. The first use was to allow programmers to identify storage locations by label and have the assembler allocate addresses to these labels.

If the assembler is to allocate memory to these labels, the question of how much storage for each symbol immediately suggests itself. More specifically, each label is supposed to denote storage for some sort of data. How much storage is required?

The basic integer storage in the MARIE architecture is a 16–bit integer; this corresponds to the halfword fixed–point binary in IBM 370 Assembler Language. For this reason, I elect to extend the MARIE assembler language to use IBM–style data definitions. Also, I am using byte addressability, though the MARIE is word addressable, in order to make the 16–bit word addresses more similar to the IBM halfword addresses.

The assembler language program above might now be written in the following way.

```
      LOAD   X    // Value in X is placed into the accumulator
      ADD    Y    // Add value in Y to that in the accumulator
      STORE  Z    // Store value into location Z.
      HALT        // Stop the computer.


X     DS     H    // Sixteen bits (two bytes) for label X
Y     DS     H    // Sixteen bits (two bytes) for label Y
Z     DS     H    // Sixteen bits (two bytes) for label Z
```

Look again at the raw machine code, written in hexadecimal. Assuming a load address of 0x100 (hexadecimal 100) for the code, the two fragments might resemble the following.

```
100    1402       // Load the accumulator from address 0x402
102    3404       // Add the contents of address 0x404
104    2406       // Store the results into address 0x406
106    7000       // Stop the computer.

    More stuff

402    0000       // Two bytes associated with label X
404    0000       // Two bytes associated with label Y
406    0000       // Two bytes associated with label Z
```

In the early days of computer programming, one might write assembler in the fashion above or in the IBM Assembler Language equivalent (to be used below), but one then needed to decide on the storage allocation and convert everything to binary by hand.

The idea of an assembler that processed a slightly–higher–level language dates at least to the late 1940's (with the EDSAC), and probably predates that. The two main features of early assembler languages both related to the interpretation of symbols, as either:

1. Operation labels to be translated into opcodes, or
2. Labels that were to identify addresses, either of data or locations in the code.

Most early assemblers used two passes. The first pass would identify the symbols and the second pass would generate the machine language. Consider again the above code.

```
      LOAD   X    // Value in X is placed into the accumulator
      ADD    Y    // Add value in Y to that in the accumulator
      STORE  Z    // Store value into location Z.
      HALT        // Stop the computer.

      // More code here.

X     DS     H    // Sixteen bits (two bytes) for label X
Y     DS     H    // Sixteen bits (two bytes) for label Y
Z     DS     H    // Sixteen bits (two bytes) for label Z
```

The first pass would identify the tokens (**LOAD**, **ADD**, **STORE**, and **HALT**) as instructions. It would identify the labels (**X**, **Y**, and **Z**) as being associated with addresses. It is important to note what the assembler will not do.

Consider the processing of the three data definitions. In following the process, we need to note what information the assembler can be considered to store in its **symbol table**, and how it processes the explicit length for each type. Each declaration calls for two bytes.

The first pass of the assembler is based on a value called the **location counter (LC)**. The assembler assumes a start address (which will be adjusted by the loader), and allocates storage for each instruction and data item relative to this start address. The above example is repeated here, to show how the LC would be used if byte addressing were in use.

```
100    1402       // Load the accumulator from address 0x402
102    3404       // Add the contents of address 0x404
104    2406       // Store the results into address 0x406
106    7000       // Stop the computer.
```

The convention calls for the first instruction to be assigned to location 0x100. Remember that all numbers in this discussion are shown in hexadecimal format. This instruction has a length of two bytes, so it will occupy addresses 0x100 and 0x101.

The second instruction is to be placed at location 0x102. It also has two bytes.

The third instruction is to be placed at location 0x104, and the fourth at 0x106.

We assume that more code follows, so that by the time the labels (**X**, **Y**, and **Z**) are read, the location counter has value 0x402; the next item is to be placed as address 0x402. Recall the declarations, each of which states how many bytes are to be set aside.

```
X      DS     H    // Sixteen bits (two bytes) for label X
Y      DS     H    // Sixteen bits (two bytes) for label Y
Z      DS     H    // Sixteen bits (two bytes) for label Z
```

Label **X** is associated with address 0x402. It calls for an allocation of two bytes, so that the 16–bit number will be stored in bytes 0x402 and 0x403. The next available location is 0x404.

Label **Y** is associated with address 0x404, and label **Z** is associated with address 0x406. The address for each is generated by allowing the proper storage for the preceding label. After this much of the assembler process, we have the following symbol table.

Label	Address
X	0x402
Y	0x404
Z	0x406

But note that the table does not carry any information on the length of the storage space allocated to each symbol, much less any on its data type. The only use made of the data definitions is in the placement of the next label. Specifically, there is no indication of the types of operations that are appropriate for data contained in these locations; the one writing the program is responsible to see to that and to use only those operations that are appropriate.

The idea of a **variable**, as used in a higher level language, comprises far more information than just the location to be associated with the data. It includes the type, which dictates not only the size of the storage space, but also the operations appropriate for the data.

Most high–level languages specify that each variable has a type associated. Early languages, such as FORTRAN allowed the variable type to be explicit in the name. Names that began with the letters I, J, K, L, M, or N were implicitly integers, the rest were implicitly single precision floating point numbers. Explicit type declaration was available, but little used.

Experience in software engineering caused **explicit data typing** to take hold; a variable could not be used until it had been explicitly declared and given a data type. The reason for this change in policy can be seen in the following fragment of old–style FORTRAN code, which represents a part of a commercial program that had been in use six years before the problem was found. Folks, this was your defense dollars at work.

```
      SUBROUTINE CLOUDCOLOR (LAT, LONG, C1, C2, C3)
C     FIRST GET THE CLOUD COVER DENSITY
      DENSITY = CLOUDDENSITY (LAT, L0NG)
C     NOW GET THE COLOR OF THE REFLECTED LIGHT
      GETCOLOR (DENSITY, C1, C2, C3)
      RETURN
```

Before reading the explanation of the problem, the reader should attempt to scan the code above and discover the problem. Code such as this would compile under the old FORTRAN, and is unusual only for having comments (denoted by the "C" in column 1). While there is no explicit variable typing, none was required. Variables beginning with "L" were integers and those beginning with "C" and "D" were real numbers. This was as intended by the design.

This is a map–oriented problem. The variable "**LAT**" appears to reference a latitude (in degrees) on a map, and is actually supposed to do so. But note the reference to longitude on the map. It appears to be denoted by "**LONG**", but a careful reader will note that there are **two variables** associated longitude; these are "**LONG**" and "**L0NG**". Reader, be honest. Did you really note the two spellings, one with the letter "**O**" and the other with the digit "**0**"?

Within the context of a FORTRAN subroutine, the appearance of a variable as an argument in the line defining the subroutine immediately gives it a definition. Thus, the appearance of the variable **LONG** in the first line implicitly declared it as an integer and made it useable.

What about the stray variable **L0NG**? The semantics of older FORTRAN allowed a variable to be declared by simple use. On first occurrence, it was initialized to a variant of zero and all further use would develop that value. In the code fragment paraphrased above, there was only one use of "**L0NG**", which occurred in the call to the cloud map. So, while the simulation was attempting to compute cloud covers and spectral densities over the mid Pacific, it was always returning the data for either London or a location in Western Africa (Longitude = 0°).

The problem, as noted above, could have been avoided by use of the **cross reference map** provided by every FORTRAN compiler; indeed it was this tool that was used to find it. This map has a list of every variable name and other label used in a module (program, subroutine, or function), the line at which it was defined or assigned a value, and every line in which it was used. Reading such a listing was tedious; most programmers did not do it. Had our programmer read it, she would have discovered entries similar to the following.

```
L0NG    3*
LONG    1*
```

In the above subroutine, with the incorrectly spelled "**L0NG**" replaced by "**LONG**", the symbol table would have an appearance that might be interpreted to contain the following.

Label	Data Type	Storage	Address
C1	Single Float	4 bytes	Some value
C2	Single Float	4 bytes	Some value
C3	Single Float	4 bytes	Some value
DENSITY	Single Float	4 bytes	Some value
LAT	Integer	2 bytes	Some value
LONG	Integer	2 bytes	Some value

It is the appearance of this type of symbol table, along with a data type reference for each of the labels, that causes the appearance of true variables in a program as opposed to labels. Once a label has been explicitly declared (and all proper declarations are now explicit), any operation on that label will be appropriate for the data type. Put another way, the compiler now has the responsibility for proper data typing; it has been taken from the programmer.

For the remainder of this chapter, we shall be using IBM® 370 Assembler. The following program is roughly equivalent to the MARIE code; the HALT has been removed.

```
          LD   0,X         LOAD REGISTER 0 FROM ADDRESS X
          AD   0,Y         ADD VALUE AT ADDRESS Y
          STD  0,Z         STORE RESULT INTO ADDRESS Z

          More code

X         DC   D'3.0'      DOUBLE-PRECISION FLOAT
Y         DC   D'4.0'
Z         DC   D'0.0'
```

The symbols **LD**, **AD**, and **STD** would be identified as assembler language operations, and the symbol **0** would be identified as a reference to register 0. The S/370 had four floating point registers, numbered 0, 2, 4, and 6. Each had a length of 64 bits, appropriate for double precision floating point format. The symbols **X**, **Y**, and **Z** are declared as double precision floating point, and each is initialized.

In the above fragments, we see **two independent processes** at work.

1) Use of data declarations to reserve space in memory to be associated with labeled addresses.
2) Use of assembly code to perform operations on these data.

Note that these are inherently independent. It is the responsibility of the coder to apply the operations to the correct data types. Occasionally, it is proper to apply a different (and apparently inconsistent) operation to a data type. Consider the following.

```
XX        DS   D     Double-precision floating point
```

All that really says is "Set aside an eight–byte memory area, and associate it with the symbol **XX**." Any eight–byte data item could be placed here, even a 15–digit packed decimal format. (This is commonly done; check your notes on **CVB** and **CVD**.)

To show what could happen, and commonly does in student programs, we rewrite the above fragment, using some operations that are not consistent with the data types.

```
         LD  0,X         LOAD REGISTER 0 FROM ADDRESS X
         AD  0,Y         ADD VALUE AT ADDRESS Y
         STD 0,Z         STORE RESULT INTO ADDRESS Z
X        DC  E'3.0'      SINGLE-PRECISION FLOAT, 4 BYTES
Y        DC  E'4.0'      ANOTHER SINGLE-PRECISION
Z        DC  D'0.0'      A DOUBLE PRECISION
```

The first instruction "**LD 0,X**" will go to address **X** and extract the next eight bytes. This will be four bytes for 3.0 and four bytes for 4.0. The value retrieved, represented in raw hexadecimal will be **0x4130 0000 4140 0000**, which can represent a double–precision number with value slightly larger than 3.0. Had X and Y been properly declared, the value retrieved would have been **0x4130 0000 0000 0000**.

Examples from a Modern Compiler

Consider the following fragments of Java code.

```
double x = 3.0;  // 64 bits or eight bytes
double y = 4.0;  // 64 bits or eight bytes
double z = 0.0;  // 64 bits or eight bytes

// More declarations and code here.

z = x + y;       // Do the addition that is
                 // proper for this data type.

                 // Here, it is double-precision
                 // floating point addition.
```

Note that the compiler will interpret the source–language statement "**z = x + y**" according to the data types of the operands.

Here is more code, similar to the first fragment. Note the two data types involved.

```
float  a = 3.0;  // 32 bits or four bytes
float  b = 4.0;  // 32 bits or four bytes
float  c = 0.0;  // 32 bits or four bytes

double x = 3.0;  // 64 bits or eight bytes
double y = 4.0;  // 64 bits or eight bytes
double z = 0.0;  // 64 bits or eight bytes

// More declarations and code here.

c = a + b;       // Single-precision floating-point
                 // addition is done here
z = x + y;       // Double-precision floating-point
                 // addition is done here
```

The operations "**c = a + b**" and "**z = x + y**" have no meaning, apart from the data types recorded by the compiler.

In order to elaborate the above claim that the operations have no meaning apart from the data types, let us consider the assembler language that might be produced were the Java code actually compiled on a S/370 and not interpreted by the JVM (**J**ava **V**irtual **M**achine).

```
// c = a + b ;
        LE  0,A      Load  single precision float
        AE  0,B      Add   single precision float
        STE 0,C      Store single precision float

// z = x + y ;
        LD  2,X      Load  double precision float
        AD  2,Y      Add   double precision float
        STD 2,Z      Store double precision float
```

Note that, when possible, the compiler will avoid immediate reuse of registers, in an attempt to keep as much data in local registers for later use. The code is more efficient, and is less likely to give rise to "register spillage" in which the contents of a register are written back to main memory. Memory reads and writes are time–consuming processes, each possibly taking multiple tens of CPU clock cycles.

Modern compilers devote a large amount of computation to devising a register mapping scheme (allocation of values to registers) that will minimize the register spillage in arithmetic operations of moderate complexity. Consider the following example.

```
double v = 0.0, w = 0.0, x = 3.0, y = 4.0, z = 5.0 ;
w = x + y ;
v = x + y + z ;
```

The example below shows inefficient code of the type actually emitted by an early 1970's era compiler. The modern compiler keeps the sum **`x + y`** in register 0 and reuses it as a partial sum in the next result **`x + y + z`**.

Older Compiler	**Modern Compiler**
`LD  0, X`	`LD  0, X`
`AD  0, Y`	`AD  0, Y`
`STD 0, W`	`STD 0, W`
`LD  0, X`	
`AD  0, Y`	
`AD  0, Z`	`AD  0, Z`
`STD 0, V`	`STD 0, V`

In the above example, code efficiency is obtained by retaining the partial sum "X + Y" in the register and not repeating the two earlier assembly language instructions. Often times, in less sophisticated compilers one may see code such as the following sequence.

```
STD 2, W       Store the value
LD  2, W       Now get the value back into the register.
```

Here we see the silliness of loading a register with a value that it must already contain. This was the main flaw of the early simplistic compilers; each statement was treated individually. Modern compilers are considerably more sophisticated.

Summary
The most obvious conclusion is that it is not appropriate to discuss assembler language code in terms of variables. The name **"variable"** should be reserved for higher–level compiled languages in which a data type is attached to each data symbol. The data type at least indicates the amount of storage space to be associated with the label and what operations are appropriate for use with it; the type may contain much more information.

Here is a brief comparison.

	Assembler	**Compiled HLL**
Data type	Operation, as indicated by the OP Code, such as A, AD, AE, AP, etc.	Data declaration, which determines the operations applied to the data
Attributes of the label	Address	Address
	(Storage size) This is used in Pass 1 of the Assembler, but not kept for future use.	Storage size
		Data type as declared

We closed this chapter with a brief discussion of compiler technology, focusing on the simplicities of earlier compilers that lead to such inefficient code. As mentioned in the first chapter of this textbook, some early compilers were very inefficient and considered each statement of high–level code in isolation from all others. This lead to very inefficient executable code, and encouraged the programmer to rewrite parts of the assembler code emitted by the compiler in order to obtain acceptable performance.