Symbols, Addresses, and Variables

Why Assembler Language Does Not Use Variables

Edward L. Bosworth, Ph.D.

Associate Professor

TSYS Department of Computer Science

Columbus State University

Columbus, GA 31907 – 5645

Sample of Assembler Language Code

Consider the assignment statement **`Z = X + Y`**.

We are using IBM® 370 Series Assembler Language as an example.
Here is a possible translation of the high–level language above.

```
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
```

The first question in examining this text is to determine what it does.
I have already told you much of what it does, but let's start at the beginning.

PLEASE NOTE: This lecture will use much that has yet to be discussed. Please focus on the "big picture". The details, such as how to write the code, will be discussed in due time.

A Two – Pass Assembler

Here, we shall focus only on the **first pass** of either a compiler or assembler. The goal is to read and interpret the symbols found in the text of the code.

Here is what a two–pass assembler would first do with this text.

```
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
```

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 symbols **X**, **Y**, and **Z** would be identified properly only if those symbols had been properly identified. Here is the way to do it for this code.

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

Rereading the Assembler Text

Here is a somewhat literal translation of the text. Note that the spacing has been altered so that I could get more on a line.

```
   LD  0,X     Load with 8-byte value from address X
   AD  0,Y     Load with 8-byte value from address Y
   STD 0,Z     Store result into 8 bytes at address Z

X  DC  D‘3.0’  Set aside eight bytes (64 bits)
               at an address to be associated with
               the label X, initialize it to 3.0
Y  DC  D‘4.0’  Set aside eight bytes (64 bits)
               at an address to be associated with
               the label Y, initialize it to 4.0
Z  DC  D‘0.0’  Set aside eight bytes (64 bits)
               at an address to be associated with
               the label Z, initialize it to 0.0
```

Cautions on the Assembler Process

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)

Reading Some BAD Assembler Text

To show what could happen, and commonly does in student programs, lets rewrite the above fragment.

```
        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 will be **0x4130 0000 4140 0000**, which represents 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**.

What A Modern Compiler Does

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.

Rereading the Java Text

Here is an informal translation of the Java code. We assume that the Java compiler is multi–pass, and that it has a standard first pass.

The standard first pass will identify all of the symbols, allocate storage for each, and assign a **data type for each**.

The compiler, as a system program with input and output, maintains a number of internal tables to assist in generating the appropriate code.

Here is our reading. We assume a table used to describe the variables.
For each variable, this table stores: the character representation,
the storage location allocated, and
the data type for the variable.

When an operation on a variable is called for, it is the description stored in this compiler table that is used to select the proper low–level code.

Rereading the Java Text (page 2)

Let's begin by rereading and interpreting the code we have seen.

```
double x = 3.0;
// Allocate 8 bytes of storage to be associated with
// the symbol "x".  Initialize the value to the
// double-precision floating-point value 3.0.
// Place entries in the compiler tables allocating
// eight bytes for storage and indicating that this
// variable is double-precision floating point.

double y = 4.0;  // Same for this symbol,
                 // but set the value to 4.0.

double z = 0.0;  // Same for this symbol,
                 // but set the value to 4.0.

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

// The data declarations determine the operation.
```

Another Java Code Fragment

Here is more code, similar to the first fragment.

```
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.

More on This Code: IBM® 370 Assembler Equivalents

Here is the sort of thing that might happen. Assume the data declarations given above, and repeated here is somewhat altered fashion.

```
float  a = 3.0, b = 4.0, c = 0.0 ;
double x = 3.0, y = 4.0, z = 0.0 ;

c = a + b ;     // LE  0,A  Instructions appropriate
                // AE  0,B  for single-precision
                // STE 0,C  floating point data.

z = x + y ;     // LD  2,X  Instructions appropriate
                // AD  2,Y  for double-precision
                // STD 2,Z  floating point data.
```

IMPORTANT POINT

The assembler code emitted is entirely dependent on the data types for the operands, as declared earlier in the program.

Another Note: 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.

Side Note: How to Write a Better Compiler

Consider the following Java fragment, focusing on how it might be compiled and how it should be compiled.

```
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`

A Final Comment on Assembler Addresses

Here is some code that will work, though it is strange.

```
        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
        LD   2,Z      RESULT NOW IN REGISTER 2
X       DC   D'3.0'   DOUBLE-PRECISION FLOAT: 8 bytes
Y       DC   D'4.0'   DOUBLE-PRECISION FLOAT: 8 bytes
Z       DC   F        32-BIT INTEGER: 4 BYTES
Z1      DC   H        16-BIT INTEGER: 2 BYTES
Z2      DC   H        16-BIT INTEGER: 2 BYTES
```

The code above will work. At the end, register 2 will have the correct result.

What about the strange declarations for **`Z`**, **`Z1`**, and **`Z2`**? All that matters is that the **`STD 0,Z`** instruction has eight contiguous bytes into which to place its result so that the **`LD 2,Z`** instruction can retrieve it. This is done.

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.

Here is a brief comparison.

Language	**Assembler**	**Compiled**
Data type determined by	Operation	Data Declaration
Attributes of the symbol	Address	Address
	Storage size (the operation may override this)	Storage size
		Data type as declared