Why Study Assembler?

Why Should a Computer Science Major Learn Assembler Language? (Nobody uses it!)

> Edward L. Bosworth, PhD Columbus State University Columbus, GA

Outline of Talk

- Introduce self
- Discuss levels of languages: high-level and assembler are the levels of interest here.
- Give a history of reasons to study assembler.
- "Behind the scenes" in a high-level language



- Columbus State University (CSU) teaches IBM 370 Assembler Language to all of its Computer Science Majors
- Your speaker (that's me) has taught this course (CPSC 3121) four times and has great enthusiasm for it.
- Some students like the course, most tolerate it, and a few find it silly.

More Background

- I have programmed assembly language on: PDP-11, VAX-11/780, CDC-6600/7600, Xerox Sigma-7, IBM 7090, and now IBM 360/370 (but not professionally)
- I dislike Intel x86 Assembly language; it is too complex. Some have called it "baroque", but this does not fit the standard use of that word.

Languages: High-Level & Low-Level

- High-level languages focus on expression in terms that fit the problem. Some are compiled, and some interpreted.
- Low-level languages (mostly assembler) focus on instructions that fit the architecture of the target machine.
- We examine a part of a standard hierarchy.

High-Level Example 1: SQL

- SQL (Structured Query Language) example: SELECT Company, Country FROM Customers WHERE Country <> 'USA'
- Note that this states what is to be done, and does not prescribe a procedure to accomplish the task.
- SQL is a high-level language.

HLL Example 2: Procedural Language

- SELECT Company, Country FROM Customers WHERE Country <> 'USA'
- The SQL suggests this procedural code Count = 0 For Rec = 1 To Max_Rec_Number If (Table[Rec].Country <> 'USA') Then Count = Count + 1 ; Company[Count] = Table[Rec].Company ; Country [Count] = Table[Rec].Country ; End If ; End If ; End For ;

Assembly Language Example

- At this level, we must assume a record size. Make it 80 (a standard size for card input).
- Here is a fragment of code to start the loop

SR 4,4 Register 4 is counter LH 5,=H'1' Register 5 is loop index LH 6,=H'1' Count by one's L 7,REC_COUNT Number of records LA 8,TABLE Load table address LP Start the comparison and do work More stuff: blah, blah!! BXLE 5,6,LP Loop again, if proper.

First: Why Study Any Assembly Language

- This depends on the decade in which you might first have studied assembly.
- **1940's** You cannot study assembly language. It does not exist yet.
- **1950's** You study assembly language because, other than raw binary machine language, it is the only way to program a computer.
- Ref: The Preparation of Programs for An Electronic Digital Computer, 1951 Maurice Wilkes, David Wheeler, et al.

Why Study An Assembler?

1960's & 1970's Compilers are not that good.

- You study assembly language in order to recode time—critical parts of your code generated by a high-level language compiler.
- In 1972, on a PDP–9, I could write assembly code that executed at least twice as fast as the equivalent compiled FORTRAN code.
- The trick was to have the FORTRAN compiler emit assembly language, and then to edit that assembly language.

Compilers: Simplistic & Modern

- High level code (FORTRAN, etc.)
 M = J + K
 N = L + J + K
- We consider the assembler language emitted by a modern compiler and an older simplistic compiler.
- I do not wish to characterize IBM/360 FORTRAN compilers, as I have not worked that much with them.

Simplistic Compilers

• M = J + K8,J A 8,K ST 8,M // Register 8 has J + K • N = L + J + K9,L A 9,J A 9,K // Register 9 has L + J + K ST 9,N

Modern Compilers

- M = J + K
 L 8,J
 A 8,K
 ST 8,M // Register 8 has J + K
- N = L + J + K
 A 8,L // Register 8 now has L + J + K
 ST 8,N
- The "bad code" reason to avoid compilers and compiled languages is no longer valid.

Why Study Assembly Now

1980's You study assembly language in order to maintain the large base of legacy code, written in assembly language.

Today Legacy code is still an issue, though a minor one. Study assembly language in order to understand the architecture of the computer and the nature of the software services provided.

Focus on compilers and the run-time system.

Why Study IBM Assembler?

Didactically speaking:

- Every CS Major should study some assembly language.
- Assembler language illustrates the ISA (Instruction Set Architecture) of a stored-program computer.
- Assembler language is a functional specification for the computer architecture; it is a good prerequisite for that course.
- IBM Mainframe Assembler can be taught in a variant that is simple and low level. Students see basic operations.

Geographically Speaking:

• Several companies in the area use IBM Mainframe computers.

Compilation vs. Assembly



- Compilers convert high-level language to machine language (maybe assembler as intermediate).
- Assemblers convert assembly language to machine language

Assembly Language: Traditional Goals

These focus on the assumption that the student will soon be writing programs in assembly language. Today, this is rarely true.

- 1. The binary representations used for character, integer, floating point and other decimal data. Floating point is usually given little attention.
- 2. Organization of program data into fields, records and files. The assembler statements that support record definition.
- 3. The basic functions of a two-pass Assembler.
- 4. The basic functions of a Link Editor.
- 5. Addressing modes in the computer; the use of base registers.
- 6. How to write and test simple assembler language programs.

Assembly Language: Additional Goals

Pay attention to the services provided by the compiler and RTS. If we have to "do it ourselves", we shall understand these services better.

Compiler and Loader Services:

Allocation of memory for variables.

Register allocation for more efficient program execution.

Resolution of external references in independently compiled programs.

Adjustment of memory addresses to reflect placement by the loader.

Run–Time Services

Creation and maintenance of dynamic data structures, such as stacks and linked lists.

Support for recursion, including management of the stack.

Sample of Assembler Language Code

Consider the assignment statement z = x + y. We use the old FORTRAN typing standard; these are real numbers. In the code below, they are double-precision real numbers.

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

This uses floating-point register 0, not GPR 0.

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.

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 4 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, in this context it is floating-point register 0.

The symbols \mathbf{x} , \mathbf{y} , and \mathbf{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	DS	D	Declare without initial value

Rereading the Assembler Text A Somewhat Literal Translation

	LD	0,X	Load with 8-byte value from address X, treat as double-precision floating point.
	AD	0,Y	Add 8-byte value from address Y,also treating as double-precision float.
	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	DS	D	Set aside eight bytes (64 bits)at an address to be associated with the label Z; do not initialize the storage.

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 human 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.

// 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.

Another Java Code Fragment

Here is more code, similar to the first fragment.

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 ;	<pre>// LE 0,A // AE 0,B // STE 0,C</pre>	Instructions appropriate for single-precision floating point data.
z = x + y;	<pre>// LD 2,X // AD 2,Y // STD 2,Z</pre>	Instructions appropriate for double-precision 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.

The meaning of "+" depends on the context.

Writing Recursive Subprograms

- IBM Assembler provides mechanisms and conventions to link separately assembled programs. These are impressive.
- My version of the Assembler appears not to support recursive subprograms.
- That's good. It forces my students to address issues of recursion explicitly.

Managing Recursion

- In order to allow recursion, one has to create and manage a stack. This is an opportunity to learn how to write non-trivial macros.
- Use the stack to manage return addresses for each subprogram.
- Use the stack to pass arguments and results.
- Use the stack to manage local variables.

System Routines

- IBM Assembler provides the basic operations of addition, subtraction, multiplication, and division.
- How does a language provide the other standard functions, such as sine, cosine, logarithm, and exponent.
- Integer powers of numbers are handled easily and are covered in my class.

Transcendental Functions

- Strictly speaking, there are no algorithms to calculate the sine or cosine of an angle.
- What makes an algorithm possible is a statement of the precision required in the answer; say 7 digits or 16 digits.
- We illustrate the computation of the sine of an angle in radians, using only basic operations.

Calculating $SIN(\Theta)$

Begin with the series expansion from calculus

SIN(
$$\Theta$$
) = $\Theta - \Theta^3/3! + \Theta^5/5! - \Theta^7/7!$
... + (-1)^N $\Theta^{2N+1}/(2N+1)! + ...$

- Show how to compute the number of terms to achieve the required accuracy.
- Write the code in assembler, using LD, STD, AD, SD, MD, and DD.

HLL vs. Assembler: Summary 1

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.

Another way to see this is to view the **symbol table** used for each tool.

In the assembler, the symbol table associates the following with a symbol:

- 1. an address,
- 2. nothing else
- In a compiler, the symbol table associates the following with a symbol;
- 1. an address,
- 2. a storage size, and
- 3. a data type for use by the compiler in creating operations on the data.

HLL vs. Assembler: Summary 2

Symbols, Addresses, and Variables

Language Data type determined by

Attributes of the symbol

Assembler

Operation

Address

Storage size (the operation may override this) Compiled

Data Declaration

Address

Storage size

Data type as declared

Add operators

A, AH, AD, AE, AP, etc.

"+"

HLL vs. Assembler: Summary 3

- The compiler for a modern HLL (High Level Language) and its RTS (Run Time System) provide significant support for the modern program.
- It benefits the student to understand these services by attempting to replicate them using only the basic Assembler operations.
- Challenge: Convert a D format floating point number to its print representation in EBCDIC