Chapter 1 – Why Study Assembly Language?

This is a textbook for a course in assembly language. More specifically it is a course that covers an older variant (IBM System/ 370 Assembler Language) of the assembly language of the IBM mainframe series of computers from the System/360 of the 1960's to the Z–Series of the present day. The previous statement immediately suggests two questions: what is assembly language and why should one study it?

In answering these two questions, we mention explicitly the one assumption about the intended reader of this book, that he or she has programmed a computer in some higher–level language; possibly Java, C++, Basic, LISP, Python, or COBOL. Other than the fact that each can be used in a beginning course on programming, the common feature of these languages is that each normally considered a **higher–level language**. The structure of such a language is based on the class of problems it is intended to solve; the expressions of such a language facilitate formulation of a solution to the associated problems. Another common feature of such languages is that each is built around a core component common to all implementations, though often with extensions that are specific to a given manufacturer and computer model.

An assembly language and its more primitive variant, machine language, has a structure that reflects the hardware architecture of a specific computer. While an assembly language might have constructs that facilitate solution of a specific class of problems, this reflects only the fact that the underlying hardware architecture has been designed with that goal in mind. As an example, the assembly language of the IBM mainframe computers contains many features to facilitate solution of business–oriented problems; this is due only to the fact that the designers of the computer decided to build a hardware architecture to support these features. It is worth noting that IBM has elected to use the name **"Assembler Language"** for what most others call "assembly language"; the two terms should be viewed as synonymous.

Given the fact that almost all computer programming is now done in a higher–level language, it is unlikely that any student will spend a significant amount of time either writing or modifying an assembly language program. Given that fact, we repeat the question "Why study assembly?". The answer should be developed historically, beginning with an answer that would have been given in 1950 and evolving into an answer that is valid today.

In the earliest digital computers any question about an assembly language would not have been reasonable; assembly language had yet to be invented. Indeed the ENIAC, one of the first general purpose digital computers, was not even a stored program computer; it was programmed by connecting coaxial cables and setting switches. Lacking a program memory, the ENIAC lacked any programming language, including an assembly language.

One of the earliest confirmable uses of assembly language was in the EDSAC, designed by Maurice Wilkes of Cambridge University (Cambridge, England), beginning in late 1946 or 1947. The EDSAC Assembly Language was described in 1951 in a book by Wilkes, Wheeler, and Gill [R_01]. At the time, Wilkes used the term "orders" for what we call "instructions".

The earliest stored–program computers, the EDVAC and EDSAC, were designed in the late 1940's. Beginning with these computers, we assume that each computer is programmed in some sort of language and ask why assembly language might be used. As noted above, the answer depends on the year in which the question is asked.

Here we must introduce a bit of terminology. Both assembly language programs and high–level language programs are written first in text that is readable by humans. From this form, it must be processed into a binary form that can be interpreted and executed by the computer. For assembly language programs, this process is called **assembly** and is done by an **assembler**. High–level languages are said to be **compiled** into binary form by a **compiler**.

In the **1950's** you would study assembly language because high–level languages were yet to be developed. Other than primitive binary machine language, assembly language was the only way to program a computer. FORTRAN (**For**mula **Tran**slation), introduced by IBM in 1957, was one of the first high–level languages that served as an alternative to assembly language.

In the **1960's and 1970's**, one would study assembly language for two purposes: either to maintain a base of legacy code written in assembly language or to enhance the performance of time–critical parts of code generated by a compiler. Most compilers of the time would emit an equivalent assembly language program prior to conversion to binary machine language. It was common practice to edit these intermediate assembly code files and then assemble these, while discarding the original machine language produced by the compiler. In 1972 the author of these notes used that process to program a PDP–9. The resulting assembly language program executed at least twice as fast as the equivalent compiled FORTRAN code.

Writing in 1979, Peter Abel [R_02] was still able to state that "programs (or even parts of programs) written in assembler may be considerably more efficient in storage space and execution time, a useful consideration if such programs are run frequently". One might note the considerable advance in compiler design of the 1980's and early 1990's that increased the efficiency of compiled programs (both time and space) to the level that even the most proficient programmers have difficulty writing assembly language that is more efficient.

Legacy code continues to be a reason to study assembly language, though increasingly a minor one. The legacy code problem is a side effect of a design choice best represented by the slang expression "If it ain't broke, don't fix it". Many companies had a large installed base of assembly language programs. These programs ran well and produced reliable results. Often these programs required minor modifications or extensions (such as adding Zip Codes to addresses). The choice was always either to redesign the code and implement it in a high–level language, such as COBOL, or modify the assembly language. Since the second option required much less in the way of programming, it was considered to be the lower risk option.

Prior to the middle 1990's, there was another very significant reason to study assembly language. When an executing program encountered an error (such as division by zero, attempting to access an invalid memory address, or trying to open a non–existent file), it would "abort" and produce a "core dump", containing the absolute binary representation of all of the memory address space allocated to the program. The programmer was required to read this absolute binary, reverse engineer it to equivalent assembly language code, and determine the offending instruction and what was to be done to correct the situation. The appearance of modern programming environments with their sophisticated debugging tools has removed this requirement.

For today, almost no new code is written in assembly language and legacy code has become a minor issue. Modern programming environments with their powerful debugging tools have removed the requirement to read "crash dumps" and convert them to assembly language. Other than the rather ethically dubious process of reverse engineering commercial code in order to create unauthorized copies, we must look elsewhere for reasons to study assembly language. For the students of Computer Science as an academic discipline, there remain a few valid reasons.

1. A knowledge of assembly language can help a programmer become more proficient in high–level languages. For example, there are many design peculiarities in the COBOL language that become obvious only when one understands the underlying assembler.
2. An understanding of an assembly language greatly facilitates the study of the architecture and organization of the computer upon which that assembly language is executed. It is your author's opinion that a knowledge of assembly language is absolutely essential to the understanding of how computers are designed and why certain design choices have been made. Indeed, one essential part of the study of a computer architecture is a study of its ISA (Instruction Set Architecture), essentially its assembly language.
3. An ability to program in basic assembly language will help the student to understand and more fully appreciate the services provided by the run–time systems associated with all modern high–level programming languages. Examples of these services include: file handling, management of variable–length strings, allocation of dynamic memory, management of the stack and recursive procedure calls, the function of a relocating loader and assignment of absolute addresses in memory, DLL (Dynamic Link Libraries), and many other common features that are quite useful and often taken for granted.
4. One reason to study IBM System/370 assembler is related to the reason just stated. The System/370 assembler language is a subset of that used on the more recent and powerful IBM mainframe computers, variously called either "zSeries" or "Series Z". The syntax of the language is rather simple and easy to grasp. The choices made in the design of this language reflect the choices dictated by the computer architecture of the day, thus allowing the student to reflect on the interaction of hardware and software design. There is also the fact that the System/370 assembler language provides very few constructs to support directly the higher–level constructs commons in a modern run–time systems. This latter fact allows for programming assignments that use the low–level code to implement these higher–level functions, possibly leading to a greater understanding.
5. We finally note that there might be some geographic reasons to study System/370 assembler language. From a pure didactic view, this author believes it important for every student majoring in Computer Science to study and understand some commonly used assembly language. In the Columbus GA area, there are a few large industries that continue to use IBM mainframe computers and occasionally legacy code written in assembler language to be modified and extended. For Columbus State University, the choice of System/370 assembler language is just a reaction to local demand.

As will become obvious, the focus of this textbook is on writing system code, the interaction of that code with the ISA of the target machine, and on understanding the functioning of the target machine at a very deep level. In this, the book differs fundamentally from other excellent texts, such as the one by Peter Abel [R_02], who appears to expect that his readers will actually use assembler language to write new financial systems. That is what COBOL is for.

Course Objectives (Learning Outcomes)
One of the better ways to explain this textbook is to state the learning objectives for the course for which this text has been written. This course is not a traditional course in assembly language. While the student is expected to become somewhat proficient in IBM Mainframe Assembler by the end of the course, the focus will be on the understanding of 1) the ISA (Instruction Set Architecture) and 2) the services provided by a modern run time system.

At the end of the course the student will be able to describe and explain the following:

1. The binary representations used by IBM for character, integer, and floating–point data and how these differ from those used in more common computers.
2. How to use zoned decimal and packed decimal data. How these differ from and extend standard two's–complement arithmetic and standard floating–point formats. Conversions from any one of these formats to any of the other formats.
3. The IBM view of data organization into fields, records and files.
The assembler declaratives that support record definition.
4. How to edit and assemble a program using the older–style tools associated with the IBM Mainframe environment.
5. The basic functions of a two–pass Assembler in producing object code.
6. The basic functions of a Link Editor in producing an executable module.
7. The use of the DS and DC declaratives to define and initialize storage.
Understand the importance of boundary alignment in the use of these declaratives.
8. Addressing modes in the IBM 370, focusing on the use of base registers.
9. How to write simple assembler programs that process character and decimal data.
This will include producing and running a number of small assembler programs.
10. How to link separately assembled programs and pass data among them.
11. The basic design and uses of magnetic tape (obsolete) and disk storage.
12. The basic data architecture of the ISAM and VSAM storage methods.
13. The physical and data architecture of physical I/O and data channels.

The remaining learning goals focus on building a modern run–time system.

14. Several methods to represent and process variable–length strings.
15. How to create and process static arrays with automatic bounds checking.
16. How to create and process a singly linked list.
17. How to create and use a stack to store data, addresses, or both.
18. How to write a simple recursive function by explicit use of a stack.
19, How to write reentrant code, which is required for most systems programs.
20. CPU hardware support for the Operating System.
21. Virtual storage (virtual memory) as implemented on the IBM 370/

The Hierarchy of Programming Languages

Another way to view assembler language is to place it in a hierarchy of programming languages from the very high level languages, down to microcode. Many authors of textbooks on computer architecture and organization use the term "virtual machine" as a method to express this concept, though the term does have other uses that are quite distinct. One use of the term as we intend it here is seen in the name **"JVM"**, for **"Java Virtual Machine"**. Java is a popular high–level language developed by Sun Microsystems, Inc. The method for executing a Java program is first to compile the code into an intermediate language, called **"byte code"**. This byte code is then executed by an emulator called the JVM. To the user, the JVM presents itself as a real computer with real hardware. In fact, it is a program that executes in the native mode of the host machine.

While it would be quite possible to design an architecture for direct execution of Java byte code, it has been thought unnecessary to do so. This use of a lower–level machine to give the appearance of direct execution of a higher–level language is the heart of the virtual machine idea.

The top level of this language hierarchy, though seldom recognized, might be called by a name such as **"computer as appliance"**; it just does its job. As an example, consider the secretary who uses the computer for e–mail, word processing, and financial spreadsheets. The mechanism by which the computer executes each task is almost unimportant; just get it done.

The top level of the traditional language hierarchy comprises problem–oriented languages that are usually called **"high level languages"**. Examples of these languages include Java, C++, Visual Basic, LISP, Snobol, Prolog, FORTRAN, and COBOL. One of the distinguishing features of such languages is that the syntax and semantics reflect the structure of the problems most commonly solved by those languages. Though many of these languages, especially FORTRAN, contained extensions tailored to specific computer architectures, in general the languages are seen as platform–independent.

The language layer below that of high level languages is that of assembly language. The main distinguishing feature of an assembly language is its close correspondence to the hardware architecture of the specific computer. While there is one version of Java that can be executed equally well on a Sun SPARC, Apple Macintosh, Pentium 4, or IBM zSeries; each of these platforms has its distinct assembly language. None of these assembly languages is remotely compatible with an assembly language on the other platform.

Let us consider a simulation problem, such as weather modeling. Suppose the code is to be run on an IBM System/370. One way to highlight the difference between a high–level language and the assembler language is to make the following observation. In order to understand the program in a high–level language, it is necessary to understand the problem being solved. In order to understand the program as written in System/370 assembler, one must also understand the architecture and organization of the underlying hardware.

Assembly language is related to a more primitive variant, called **"machine language"**. Some experts consider machine language to occupy a lower level than assembly language; others place it at the same level. The real difference is that assembly language programs are written to be read by humans, and use mnemonics that are easy to understand. Machine language programs are written as a sequence of binary numbers, which are made marginally more readable by being rendered as hexadecimal (base 16) values.

In order to see the difference between the two (or three) levels of languages, we adopt an example from the textbook by Patterson and Hennessy [R_04]. We begin with a fragment of code written in either C or C++ (though some purists claim that neither is high level).

Here is the code fragment, with some reasonable comments added.

```
swap (int v[], int k)
{  int temp ;          // Swap element v[k] with v[k+1]
   temp = v[k];        // Save element v[k]
   v[k] = v[k+1];      // Move v[k+1] down.
   v[k+1] = temp;      // Replace the value of v[k+1]
}
```

Here is the code as written in the assembly language for a computer called MIPS. While most assembly languages, including both MIPS assembler and System/370 assembler, provide for comments, this code will not be commented. This translation of the C++ code above is a bit misleading in that three executable lines are expanded only to seven assembly language lines. Most expansions are four to eight lines of assembly language for each high–level statement.

```
swap:
        muli $2,  $5, 4
        add  $2,  $4, $2
        lw   $15, 0($2)
        lw   $16, 4($2)
        sw   $16, 0($2)
        sw   $15, 4($2)
        jr   $31
```

We now give the translation of this assembly language code into machine language. Each assembly language instruction directly corresponds to a 32–bit binary machine language instruction, which we shall represent in hexadecimal form.

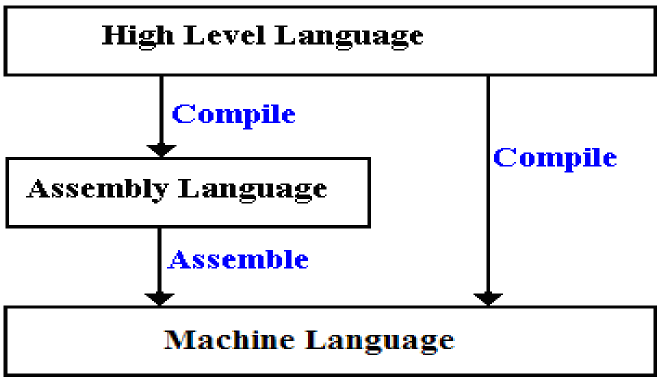
```
        00 A1 00 18
        00 18 18 21
        8C 62 00 00
        8C F2 00 04
        AC F2 00 00
        AC 62 00 04
        03 E0 00 08
```

On all modern stored–program computers, it is a version of the binary machine language that is executed. The basis of this execution is called either **"Fetch/Execute"** or some variant of that name; this is the basic cycle of a modern stored–program computer. Each machine language instruction is fetched from memory and copied into a special register, called the **"Instruction Register"**, where it is decoded and executed. This is the native language of the computer.

As a historical fact, the languages were developed "bottom to top", with machine language being the first developed. Almost immediately, assembly language was developed mostly because it used mnemonics and was much easier for a human to read. High–level languages were a later development. Recent research and development related to high–level languages has focused on more sophisticated compilers and support for parallel processing.

We shall now mention a few of the more obvious differences between high–level languages and assembly languages. We begin with the definitions of the process that converts the language into the machine language that is ready to be loaded into memory and executed. By definition, all high–level language programs are said to be **compiled**, while assembly language programs are said to be **assembled**. While this difference may seem to be just one of semantics, we shall quickly see that compilers are usually much more sophisticated than assemblers.

We now ask about the output of the compiler. In this context, we have two basic options: either machine language or assembly language that is then assembled. The option chosen by IBM is for each compiler to emit assembly language, which is processed by a common assembler.



Some of the other more common differences between compiled and assembled languages are:

1. Assembly language statements almost always map one–to–one into machine language statements; one assembly language statement generates one machine language word. High–level language statements usually generate a number of machine language words, commonly in the range 3 to 8; with more being possible.
2. High–level languages provide for the declaration of variables by types and associate the proper operations with them. Assembly languages provide only for the allocation of storage space and rely on the assembly language instruction to be specific about the data type. In other words, high–level languages select the operation appropriate for the type of the variable, while the assembly language uses the operation specified.
3. Compilers for high–level languages have become quite sophisticated in the optimal use of system resources, such as the general purpose registers in the CPU. This usually leads to executable code that is quite efficient. It is worth noting that many early compiler writers considered the problem of optimal register allocation to be unsolvable, until one bright designer recognized it as the equivalent of a well known problem in mathematical graph theory. Once this was seen, the problem was easily solved.
4. It is said that compilers are yet to become sufficiently sophisticated in allocating resources for parallel execution on a multi–CPU (or multi–core) computer. While assembly language programs can allocate the resources explicitly, your author expects considerable progress to be made in compilation for parallel execution.
5. The code for every operating system does require some basic operations, such as interrupt management, that are not easily provided in a high level language. For this reason, one may expect assembly language to play a minor part in all future systems.

Let us consider the second point by assuming a machine, such as the System/370, that supports both 32–bit integer arithmetic and 32–bit floating point (real) numbers. Consider the following fragment, written in a FORTRAN–like high level language.

```
X = 2
Y = 3
Z = X + Y
```

The values 2 and 3 are integer constants. The symbols X, Y, and Z represent variables. The compiler allocated 32–bits (four bytes) for the storage of each, as will the assembler. However, the compiler will use the variable type declarations (either implicit or explicit, as in Java) to determine the operations. If all of X, Y, and Z represent integer variables, the first two assignments are quite simple and it is integer addition that is invoked.

Suppose now that each of X, Y, and Z represent real numbers. The first two assignment statements involve conversion of the integer values to the equivalent real number values, 2.0 and 3.0. Most modern compilers will do this at compile time, thus avoiding the overhead of run–time translation. The addition is now the operation appropriate for real numbers.

Note again the fact that it is the type declarations for the variables that determines what type of assembly language is emitted by the compiler for the three statements being considered.

We close this section by mentioning a newer assembler language developed by IBM and stating the reasons that it might be preferable to a high–level language. Here, we must note that the author of this textbook has no experience directly related to these arguments, but finds them quite plausible. These reasons are taken from a presentation by Kristine H. Neely [R_03]

1. HLASM, the new High Level Assembler, now directly supports all of the structures required for structured programming. Multiple direct branches are no longer required.
2. HLASM can create programs that “break” the addressing limits imposed by some high–level languages. This may be called the “2 GB bar”.
3. HLASM has provisions for explicit control of multiple processors operating in parallel. At present, the facilities offered by high–level languages lack the sophistication that can be explicitly achieved by programming in HLASM. [Your author’s note: This may change soon.]

The introduction of this new high level assembler raises the question about the complexity desired in a modern assembly language. A number of modern assemblers, such at the IBM product HLSAM, contain almost enough features to rank them as high–level languages. There are two variants of this complexity problem, only one of which having an obvious answer.

1. The first variant is similar to that seen in the VAX/11–780 and other members of the VAX family (often called “Vaxen” by pundits familiar with German). Here, the complex assembly language is a direct result of the complexity of the underlying machine. One of the standard results of modern computer design theory is that complex machines are less desirable than machines more simply designed; simpler means faster.
2. The second variant calls for the complex assembly language to be processed by sophisticated pre–processor into a simpler standard assembly language. This approach has no implications for the underlying machine.

What Lies Beneath
The student of computer architecture will realize that the hierarchy of languages does not convey the whole story about hardware and software architecture. We have taken this hierarchy down only to the machine language level. Below that level, lie a number of very important levels.

The **microarchitecture level** deals with the control structures that cause the computer to execute the instructions found in the sequence of machine language words.

The **device level** deals with the construction of the circuits used by the microarchitecture in order to execute the machine language. These are built from basic devices, often called "logic gates".

The **electrical engineering** layer deals with how to fabricate the basic devices (or gates) from the basic circuit elements: transistors, resistors, capacitors, inductors, and conducting traces.

The **solid state layer** deals with how to fabricate basic circuit elements with new desirable qualities. Normally the requirement is either that they be faster or dissipate less heat.

A course in Computer Architecture and Design (such as CPSC 5155 taught at Columbus State University in Columbus GA) will address issues in the microarchitecture level and device level. Any course in assembly language must stop at the machine language level.