Lecture No.1

Lecture Outlines

1.1 Welcome to Assembly Language

Assembly Language for x86 Processors focuses on programming microprocessors compatible with Intel and AMD processors running under 32-bit and 64-bit versions of Microsoft Windows.

The latest version of *Microsoft Macro Assembler* (known as *MASM*) should be used with this book. MASM is included with most versions of Microsoft Visual Studio (Pro, Ultimate, Express, . . .). Please check our web site (*asmirvine.com*) for the latest details about support for MASM in Visual Studio. We also include lots of helpful information about how to set up your software and get started.

Some other well-known assemblers for x86 systems running under Microsoft Windows include TASM (Turbo Assembler), NASM (Netwide Assembler), and MASM32 (a variant of MASM). Two popular Linux-based assemblers are GAS (GNU assembler) and NASM. Of these, NASM's syntax is most similar to that of MASM.

Assembly language is the oldest programming language, and of all languages, bears the closest resemblance to native machine language. It provides direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system.

Educational Value Why read this book? Perhaps you're taking a college course whose title is similar to one of the following courses that often use our book:

- Microcomputer Assembly Language
- Assembly Language Programming
- Introduction to Computer Architecture
- Fundamentals of Computer Systems
- Embedded Systems Programming

This book will help you learn basic principles about computer architecture, machine language, and low-level programming. You will learn enough assembly language to test your knowledge on today's most widely used microprocessor family. You won't be learning to program a "toy" computer using a simulated assembler; MASM is an industrial-strength assembler, used by practicing professionals. You will learn the architecture of the Intel processor family from a programmer's point of view.

If you are planning to be a C or C++ developer, you need to develop an understanding of how memory, address, and instructions work at a low level. A lot of programming errors are not easily recognized at the high-level language level. You will often find it necessary to "drill down" into your program's internals to find out why it isn't working.

If you doubt the value of low-level programming and studying details of computer software and hardware, take note of the following quote from a leading computer scientist, Donald Knuth, in discussing his famous book series, *The Art of Computer Programming*:

> Some people [say] that having machine language, at all, was the great mistake that I made. I really don't think you can write a book for serious computer programmers unless you are able to discuss low-level detail.[1]

Visit this book's web site to get lots of supplemental information, tutorials, and exercises at **www.asmirvine.com**

1.1.1 Questions You Might Ask

What Background Should I Have? Before reading this book, you should have programmed in at least one structured high-level language, such as Java, C, Python, or C++. You should know how to use IF statements, arrays, and functions to solve programming problems.

What Are Assemblers and Linkers? An *assembler* is a utility program that converts source code programs from assembly language into machine language. A *linker* is a utility program that combines individual files created by an assembler into a single executable program. A related utility, called a *debugger*, lets you to step through a program while it's running and examine registers and memory.

What Hardware and Software Do I Need? You need a computer that runs a 32-bit or 64-bit version of Microsoft Windows, along with one of the recent versions of Microsoft Visual Studio.

What Types of Programs Can Be Created Using MASM?

- *32-Bit Protected Mode:* 32-bit protected mode programs run under all 32-bit versions of Microsoft Windows. They are usually easier to write and understand than real-mode programs. From now on, we will simply call this *32-bit mode*.
- *64-Bit Mode:* 64-bit programs run under all 64-bit versions of Microsoft Windows.
- *16-Bit Real-Address Mode:* 16-bit programs run under 32-bit versions of Windows and on embedded systems. Because they are not supported by 64-bit Windows, we will restrict discussions of this mode to Chapters 14 through 17. These chapters are in electronic form, available from the publisher's web site.

What Will I Learn? This book should make you better informed about data representation, debugging, programming, and hardware manipulation. Here's what you will learn:

- Basic principles of computer architecture as applied to x86 processors
- Basic boolean logic and how it applies to programming and computer hardware
- How x86 processors manage memory, using protected mode and virtual mode
- How high-level language compilers (such as C++) translate statements from their language into assembly language and native machine code

- How high-level languages implement arithmetic expressions, loops, and logical structures at the machine level
- Data representation, including signed and unsigned integers, real numbers, and character data
- How to debug programs at the machine level. The need for this skill is vital when you work in languages such as C and C++, which generate native machine code
- How application programs communicate with the computer's operating system via interrupt handlers and system calls
- How to interface assembly language code to C++ programs
- How to create assembly language application programs

How Does Assembly Language Relate to Machine Language? *Machine language* is a numeric language specifically understood by a computer's processor (the CPU). All x86 processors understand a common machine language. *Assembly language* consists of statements written with short mnemonics such as ADD, MOV, SUB, and CALL. Assembly language has a *one-to-one* relationship with machine language: Each assembly language instruction corresponds to a single machine-language instruction.

How Do C++ and Java Relate to Assembly Language? High-level languages such as Python, C++, and Java have a *one-to-many* relationship with assembly language and machine language. A single statement in C++, for example, expands into multiple assembly language or machine instructions. Most people cannot read raw machine code, so in this book, we examine its closest relative, assembly language. For example, the following C++ code carries out two arithmetic operations and assigns the result to a variable. Assume X and Y are integers:

```
int   Y;
int   X = (Y + 4) * 3;
```

Following is the equivalent translation to assembly language. The translation requires multiple statements because each assembly language statement corresponds to a single machine instruction:

```
mov   eax,Y                    ; move Y to the EAX register
add   eax,4                    ; add 4 to the EAX register
mov   ebx,3                    ; move 3 to the EBX register
imul  ebx                      ; multiply EAX by EBX
mov   X,eax                    ; move EAX to X
```

(*Registers* are named storage locations in the CPU that hold intermediate results of operations.) The point of this example is not to claim that C++ is superior to assembly language or vice versa, but to show their relationship.

Is Assembly Language Portable? A language whose source programs can be compiled and run on a wide variety of computer systems is said to be *portable*. A C++ program, for example, will compile and run on just about any computer, unless it makes specific references to library functions that exist under a single operating system. A major feature of the Java language is that compiled programs run on nearly any computer system.

Assembly language is not portable, because it is designed for a specific processor family. There are a number of different assembly languages widely used today, each based on a processor family.

Some well-known processor families are Motorola 68x00, x86, SUN Sparc, Vax, and IBM-370. The instructions in assembly language may directly match the computer's architecture or they may be translated during execution by a program inside the processor known as a *microcode interpreter*.

Why Learn Assembly Language? If you're still not convinced that you should learn assembly language, consider the following points:

- If you study computer engineering, you may likely be asked to write *embedded* programs. They are short programs stored in a small amount of memory in single-purpose devices such as telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, data acquisition instruments, video cards, sound cards, hard drives, modems, and printers. Assembly language is an ideal tool for writing embedded programs because of its economical use of memory.
- Real-time applications dealing with simulation and hardware monitoring require precise timing and responses. High-level languages do not give programmers exact control over machine code generated by compilers. Assembly language permits you to precisely specify a program's executable code.
- Computer game consoles require their software to be highly optimized for small code size and fast execution. Game programmers are experts at writing code that takes full advantage of hardware features in a target system. They often use assembly language as their tool of choice because it permits direct access to computer hardware, and code can be hand optimized for speed.
- Assembly language helps you to gain an overall understanding of the interaction between computer hardware, operating systems, and application programs. Using assembly language, you can apply and test theoretical information you are given in computer architecture and operating systems courses.
- Some high-level languages abstract their data representation to the point that it becomes awkward to perform low-level tasks such as bit manipulation. In such an environment, programmers will often call subroutines written in assembly language to accomplish their goal.
- Hardware manufacturers create device drivers for the equipment they sell. *Device drivers* are programs that translate general operating system commands into specific references to hardware details. Printer manufacturers, for example, create a different MS-Windows device driver for each model they sell. Often these device drivers contain significant amounts of assembly language code.

Are There Rules in Assembly Language? Most rules in assembly language are based on physical limitations of the target processor and its machine language. The CPU, for example, requires two instruction operands to be the same size. Assembly language has fewer rules than C++ or Java because the latter use syntax rules to reduce unintended logic errors at the expense of low-level data access. Assembly language programmers can easily bypass restrictions characteristic of high-level languages. Java, for example, does not permit access to specific memory addresses. One can work around the restriction by calling a C function using JNI (*Java Native Interface*) classes, but the resulting program can be awkward to maintain. Assembly language, on the other hand, can access any memory address. The price for such freedom is high: Assembly language programmers spend a lot of time debugging!

1.1.2 Assembly Language Applications

In the early days of programming, most applications were written partially or entirely in assembly language. They had to fit in a small area of memory and run as efficiently as possible on slow processors. As memory became more plentiful and processors dramatically increased in speed, programs became more complex. Programmers switched to high-level languages such as C, FORTRAN, and COBOL that contained a certain amount of structuring capability. More recently, object-oriented languages such as Python, C++, C#, and Java have made it possible to write complex programs containing millions of lines of code.

It is rare to see large application programs coded completely in assembly language because they would take too much time to write and maintain. Instead, assembly language is used to optimize certain sections of application programs for speed and to access computer hardware. Table 1-1 compares the adaptability of assembly language to high-level languages in relation to various types of applications.

Table 1-1 Comparison of Assembly Language to High-Level Languages.

Type of Application	High-Level Languages	Assembly Language
Commercial or scientific application, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	The language may not provide for direct hardware access. Even if it does, awkward coding techniques may be required, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Commercial or scientific application written for multiple platforms (different operating systems).	Usually portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	May produce large executable files that exceed the memory capacity of the device.	Ideal, because the executable code is small and runs quickly.

The C and C++ languages have the unique quality of offering a compromise between high-level structure and low-level details. Direct hardware access is possible but completely nonportable. Most C and C++ compilers allow you to embed assembly language statements in their code, providing access to hardware details.

1.2 Virtual Machine Concept

An effective way to explain how a computer's hardware and software are related is called the *virtual machine concept*. A well-known explanation of this model can be found in Andrew Tanenbaum's book, *Structured Computer Organization.* To explain this concept, let us begin with the most basic function of a computer, executing programs.

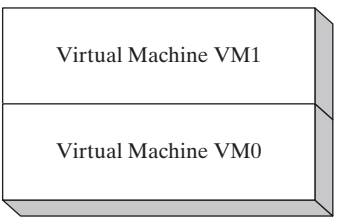
A computer can usually execute programs written in its native *machine language*. Each instruction in this language is simple enough to be executed using a relatively small number of electronic circuits. For simplicity, we will call this language **L0**.

Programmers would have a difficult time writing programs in L0 because it is enormously detailed and consists purely of numbers. If a new language, **L1**, could be constructed that was easier to use, programs could be written in L1. There are two ways to achieve this:

- *Interpretation:* As the L1 program is running, each of its instructions could be decoded and executed by a program written in language L0. The L1 program begins running immediately, but each instruction has to be decoded before it can execute.
- *Translation:* The entire L1 program could be converted into an L0 program by an L0 program specifically designed for this purpose. Then the resulting L0 program could be executed directly on the computer hardware.

Virtual Machines

Rather than using only languages, it is easier to think in terms of a hypothetical computer, or *virtual machine*, at each level. Informally, we can define a virtual machine as a software program that emulates the functions of some other physical or virtual computer. The virtual machine **VM1**, as we will call it, can execute commands written in language L1. The virtual machine **VM0** can execute commands written in language L0:



Each virtual machine can be constructed of either hardware or software. People can write programs for virtual machine VM1, and if it is practical to implement VM1 as an actual computer, programs can be executed directly on the hardware. Or programs written in VM1 can be interpreted/translated and executed on machine VM0.

Machine VM1 cannot be radically different from VM0 because the translation or interpretation would be too time-consuming. What if the language VM1 supports is still not programmer-friendly enough to be used for useful applications? Then another virtual machine, VM2, can be designed that is more easily understood. This process can be repeated until a virtual machine VM*n* can be designed to support a powerful, easy-to-use language.

The Java programming language is based on the virtual machine concept. A program written in the Java language is translated by a Java compiler into *Java byte code*. The latter is a low-level language quickly executed at runtime by a program known as a *Java virtual machine (JVM)*. The JVM has been implemented on many different computer systems, making Java programs relatively system independent.

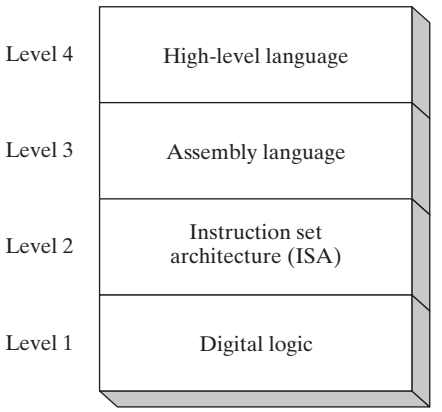
Specific Machines

Let us relate this to actual computers and languages, using names such as **Level 2** for VM2 and **Level 1** for VM1, shown in Figure 1-1. A computer's digital logic hardware represents machine Level 1. Above this is Level 2, called the *instruction set Architecture (ISA)*. This is the first level at which users can typically write programs, although the programs consist of binary values called *machine language.*

Instruction Set Architecture (Level 2) Computer chip manufacturers design into the processor an instruction set to carry out basic operations, such as move, add, or multiply. This set of instructions is also referred to as *machine language*. Each machine-language instruction is executed either directly by the computer's hardware or by a program embedded in the microprocessor chip called a *microprogram*. A discussion of microprograms is beyond the scope of this book, but you can refer to Tanenbaum for more details.

Assembly Language (Level 3) Above the ISA level, programming languages provide translation layers to make large-scale software development practical. Assembly language, which appears at Level 3, uses short mnemonics such as ADD, SUB, and MOV, which are easily translated to the ISA level. Assembly language programs are translated (assembled) in their entirety into machine language before they begin to execute.

FIGURE 1–1 Virtual machine levels.



High-Level Languages (Level 4) At Level 4 are high-level programming languages such as C, C++, and Java. Programs in these languages contain powerful statements that translate into multiple assembly language instructions. You can see such a translation, for example, by examining the listing file output created by a C++ compiler. The assembly language code is automatically assembled by the compiler into machine language.