LECTURE 1

COMP 211 ASSEMBLY LANGUAGE PROGRAMMING

Introduction

Objectives

- To introduce the assembly language and explain where it fits in the hierarchy of computer languages
- To discuss the advantages and disadvantages associated with programming in the assembly language
- To provide motivation to learn the assembly language
- To demonstrate performance advantages of the assembly language

1.1 A User's View of Computer Systems

A user's view of a computer system depends on the degree of abstraction provided by the underlying software. Figure 1.1 shows a hierarchy of levels at which users can interact with a computer system. Moving to the top of the hierarchy shields the user from the lower level details. At the highest level, the user interaction is limited to the interface provided by application software such as a spreadsheet, word processor, and so on. The user is expected to have only a rudimentary knowledge of how the system operates. Problem solving at this level, for example, involves composing a letter using the word processor software. At the next level, problem solving is done in one of the high-level languages such as C and Java. A user interacting with the system at this level should have detailed knowledge of software development. Typically, these users are application programmers. Level 4 users are knowledgeable about the application and the high-level language that they would use to write the application software. They may not, however, know internal details of the system unless they also happen to be involved in developing system software such as device drivers, assemblers, linkers, and so on. Both levels 4 and 5 are system-independent, i.e., independent of a particular processor used in the system. For example, an application program written in C can be executed on a system with an Intel processor or a PowerPC processor without modifying the source code. All we have to do is recompile the program with a C compiler native to the target system. By contrast, software development done at all levels below level 4 is system-dependent. Assembly language programming is referred to as low-level programming because each assembly language instruction performs a much lower-level task compared to an instruction in a high-level language. As a consequence, to perform the same task, assembly language code tends to be much larger than the equivalent high-level language code. Assembly language instructions are native to the processor used in the system. For example, a program written in the Pentium assembly language cannot be executed on the PowerPC processor. Programming in the assembly language also requires knowledge about system internal details such as the processor architecture, memory organization, and so on. Machine language is a close relative of the assembly language. Typically, there is a one-to-one correspondence between the assembly language and machine language instructions. The processor understands only the machine language, whose instructions consist of strings of 1's and 0's. We say more on these two languages in the next section. Even though the

assembly language is considered a low-level language, programming in the assembly language will not expose you to all the nuts and bolts of the system. Our operating system hides several of the low-level details so that the assembly language programmer can breathe easily. For example, if we want to read input from the keyboard, we can rely on the services provided by the operating system. Well, ultimately there has to be something to execute the machine language instructions. This is the system hardware, which consists of digital logic circuits and the associated support electronics.

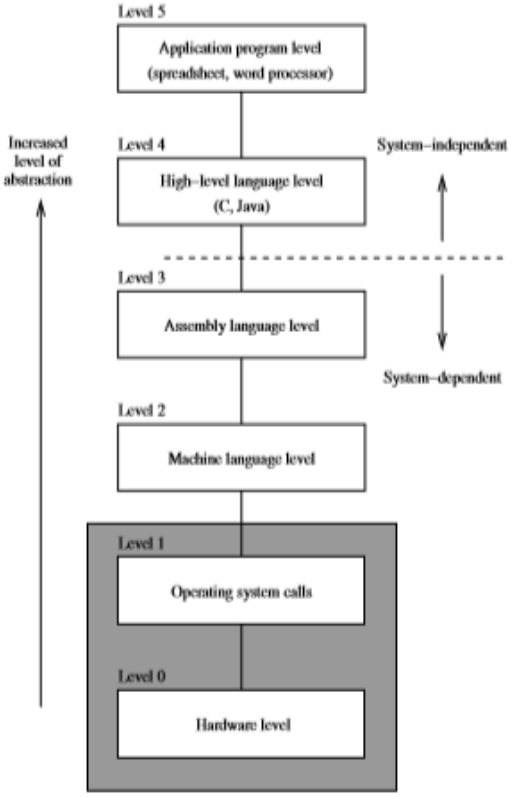

Figure 1.1 A user's view of a computer system.

1.2 What Is Assembly Language?

Assembly language is directly influenced by the instruction set and architecture of the processor. There are two basic types of processors: CISC (Complex Instruction Set Computers) and RISC (Reduced Instruction Set Computers). The Pentium is an example of a CISC processor.

Most current processors, however, follow the RISC design philosophy. As the name suggests, CISC processors use complex instructions. What is a complex instruction? For example, adding two integers is considered a simple instruction. But, an instruction that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction. RISC systems use only simple instructions. Furthermore, RISC systems assume that the required operands are in the processor's registers, not in the main memory. A CISC processor, on the other hand, does not impose such restrictions. So what? It turns out that characteristics like simple instructions and restrictions like register-based operands not only simplify the processor design but also result in a processor that provides improved application performance. The assembly language code must be processed by a program in order to generate the machine language code. Assembler is the program

that translates assembly language code into the machine language. NASM (Netwide Assembler), MASM (Microsoft Assembler), and TASM (Borland Turbo Assembler) are some of the popular assemblers for the Pentium processors. We will use the SPIM simulator to run the MIPS assembly language programs. Here are some Pentium language examples:

```
inc result
 mov class_size,45
and mask1,128
add marks,10
```

The first instruction increments the variable result. This assembly language instruction is equivalent to

```
result++;
```

in C. The second instruction initializes class_size to 45. The equivalent statement in C is

```
class_size = 45;
```

The third instruction performs the bitwise and operation on mask1 and can be expressed in C as

```
mask1 = mask1 & 128;
```

The last instruction updates marks by adding 10. This is equivalent to

```
marks = marks + 10;
```

in C.

As you can see from these examples, most Pentium instructions use two addresses. In these instructions, one operand doubles as a source and destination (for example, marks and class_size). In contrast, the MIPS instructions use three addresses as shown below:

```
andi $t2,$t1,15
addu $t3,$t1,$t2
 move $t2,$t1
```

The operands in these instructions are in processor registers. The processor registers t1, t2, andt3 are identified by $. The andi instruction performs bitwise and of t1 contents with 15 and writes the result in t2. The second instruction adds contents of t1 and t2 and stores the result in t3. The last instruction copies the t1 value into t2. In contrast to our claim that MIPS uses three addresses, this instruction seems to use only two addresses. This is not really an instruction supported by MIPS processor—it is a synthesized assembly language instruction. When translated by the MIPS assembler, it is replaced by

```
addu $t2,$0,$t1
```

The second operand in this instruction is a special register that holds constant zero. Thus, copying the t1 value is treated as adding zero to it. These examples illustrate several points:

1. Assembly language instructions are cryptic.
2. Assembly language operations are expressed by using mnemonics (like and, inc, addu, and so on).
3. Assembly language instructions are low-level. For example, we cannot write the following in the Pentium assembly language:

```
add marks,value
```

This instruction is invalid because two variables, marks and value, cannot be used in a single instruction. In MIPS, for example, we cannot even write something like

```
addu class_size,45
```

as it expects all operands in the processor's internal registers.

We appreciate the readability of the assembly language instructions by looking at the equivalent machine language instructions. Here are some Pentium and MIPS machine language examples:

Pentium examples

Assembly language		Operation	Machine language (in hex)
nop		No operation	90
inc	result	Increment	FF060A00
mov	class_size, 45	Copy	C7060C002D00
and	mask, 128	Logical and	80260E0080
add	marks, 10	Integer addition	83060F000A

MIPS examples

Assembly language		Operation	Machine language (in hex)
nop		No operation	00000000
move	$t2,$t15	Copy	000A2021
andi	$t2,$t1,15	Logical and	312A000F
addu	$t3,$t1,$t2	Integer addition	012A5821

In the above tables, machine language instructions are written in the hexadecimal number system. If you are not familiar with this number system, consult Appendix A for a detailed discussion of various number systems. These examples visibly demonstrate one of the key differences between CISC and RISC processors: RISC processors use fixed-length machine language instructions whereas the machine language instructions of CISC processors vary in length. It is obvious from these examples that understanding the code of a program in the machine language is almost impossible. Since there is a one-to-one correspondence between the instructions of assembly language and machine language, it is fairly straightforward to translate instructions from the assembly language to the machine language. However, life was not so easy for some of the early programmers. When microprocessors were first introduced, some programming was in fact done in machine language!

1.3 Advantages of High-Level Languages

High-level languages are preferred to program applications, as they provide a convenient abstraction of the underlying system suitable for problem solving. Here are some advantages of programming in a high-level language:

1. Program development is faster. Many high-level languages provide structures (sequential, selection, iterative) that facilitate program development. Programs written in a high-level language are relatively small compared to the equivalent programs written in an assembly language. These programs are also easier to code and debug.

2. Programs are easier to maintain. Programming a new application can take from several weeks to several months and the life cycle of such an application software can be several years. Therefore, it is critical that software development be done with a view of software maintainability, which involves activities ranging from fixing bugs to generating the next version of the software. Programs written in a high-level language are easier to understand and, when good programming practices are followed, easier to maintain. Assembly language programs tend to be lengthy and take more time to code and debug. As a result, they are also difficult to maintain.

3. Programs are portable. High-level language programs contain very few processor-dependent details. As a result, they can be used with little or no modification on different computer systems. By contrast, assembly language programs are processor-specific.

1.4 Why Program in the Assembly Language?

There are two main reasons why programming is still done in the assembly language: (1) efficiency, and (2) accessibility to system hardware. Efficiency refers to how "good" a program is in achieving a given objective. Here we consider two objectives based on space (space efficiency) and time (time efficiency). Space efficiency refers to the memory requirements of a program, i.e., the size of the executable code. Program A is said to be more space-efficient if it takes less memory space than program B to perform the same task. Very often, programs written in an assembly language tend to be more compact than those written in a high-level language. Time efficiency refers to the time taken to execute a program. Obviously a program that runs faster is said to be better from the time-efficiency point of view. If we craft assembly language programs carefully, they tend to run faster than their high-level language counterparts.

The superiority of the assembly language in generating compact code is becoming increasingly less important for several reasons. First, the savings in space pertain only to the program code and not to its data space. Thus, depending on the application, the savings in space obtained by converting an application program from some high-level language to the assembly language may not be substantial. Second, the cost of memory has been decreasing and memory capacity has been increasing. Thus, the size of a program is not a major hurdle anymore. Finally, compilers are becoming "smarter" in generating code that is both space and time-efficient. However, there are systems such as embedded controllers and handheld devices in which space efficiency is important. One of the main reasons for writing programs in the assembly language is to generate code that is time-efficient. The superiority of assembly language programs in producing efficient code is a direct manifestation of specificity. That is, assembly language programs contain only the code that is necessary to perform the given task. Even here, a "smart" compiler can optimize the code that can compete well with its equivalent written in the assembly language. Although the gap is narrowing with improvements in compiler technology, the assembly language still retains its advantage for now. The other main reason for writing assembly language programs is to have direct control over system hardware. High-level languages, on purpose, provide a restricted (abstract) view of the underlying hardware. Because of this, it is almost impossible to perform certain tasks that require access to the system hardware. For example, writing a device driver to a new scanner on the market almost certainly requires programming in the assembly language. Since the assembly language does not impose any restrictions, you can have direct control over the system hardware. If you are developing system software, you cannot avoid writing assembly language programs.

1.5 Typical Applications

We have identified three advantages to programming in an assembly language.

1. Time efficiency
2. Accessibility to hardware
3. Space efficiency

Time efficiency: Applications for which the execution speed is important fall under two categories:

1. Time convenience (to improve performance)
2. Time-critical (to satisfy functionality)

Applications in the first category benefit from time-efficient programs because it is convenient or desirable.

In time-critical applications, tasks have to be completed within a specified time period. These applications, also called real-time applications, include aircraft navigation systems, process control systems, robot control software, communications software, and target acquisition (e.g., missile tracking) software.

Accessibility to hardware: System software often requires direct control over the system hardware. Examples include operating systems, assemblers, compilers, linkers, loaders, device drivers, and network interfaces. Some applications also require hardware control. Video games are an obvious example.

Space efficiency: For most systems, compactness of application code is not a major concern. However, in portable and handheld devices, code compactness is an important factor. Space efficiency is also important in spacecraft control systems.

1.6 Why Learn the Assembly Language?

Programming in the assembly language is a tedious and error-prone process. The natural preference of a programmer is to program in some high-level language. We discussed a few good reasons why some applications cannot be programmed in a high-level language. Even these applications do not require the whole program to be written in the assembly language. In such instances, a small part of the program is written in the assembly language and the rest is written in some high-level language. Such programs are called hybrid or mixed-mode programs. Learning the assembly language has both practical and educational purposes. Even if you don't intend to program in an assembly language, studying it gives you a good understanding of computer systems. When you program in a high-level language, you are provided only a "black-box" view of the system. When programming in the assembly language, you need to understand the internal details of the system (for example, you need to know about processor internal registers). To understand the assembly language is to understand the computer system itself!. We use the popular Pentium processor to represent the CISC category. We study RISC processors by looking at the MIPS assembly language. Studying these two assembly languages gives you a solid foundation to understand the differences between the CISC and RISC design philosophies and how they impact execution speed of your programs. A final reason to learn the assembly language is the personal satisfaction that comes with learning something complex. Sure, learning the assembly language is more difficult than learning Java. But the assembly language gives you complete control over the system hardware. You feel powerful with the assembly language on your side, making the time spent learning assembly language worth your while. The insights provided by the assembly language will benefit you even if you program only in high-level languages.

1.7 Performance: C Versus Assembly Language

Now let's see how much better we can do by writing programs in assembly language. As an example, consider multiplying two 16-bit integers. Our strategy is to write the multiplication procedure in C (a representative high-level language) and in the Pentium assembly language and compare their execution times.

1.7.1 Multiplication Algorithm

The Pentium instruction set has two instructions for multiplication: one for unsigned integers and the other for signed integers. These instructions can be used to multiply two integers that can take up to 32 bits to represent them. Here we consider the algorithm that is based on the

longhand multiplication. This algorithm, shown below, takes two n-bit unsigned integers and produces a 2n-bit product.

product := 0 for (i =1to n) if (least significant bit of the multiplier = 1) then product := product + multiplicand end if Left shift the multiplicand by one bit position Right shift the multiplier by one bit position end for

More details on this algorithm are given in Appendix A. The main program is shown in Program 1.1 on page 16. To avoid the influence of I/O (i.e., the printf and scanf statements), we time only the mult procedure. To do this, we use clock(), which is defined in the time.h header file. When clock() is invoked, it gives the current clock value in terms of number of clock ticks. The number of clock ticks per second is defined by CLOCKS_PER_SEC. Thus, to obtain the multiplication time in seconds, we have to divide the clock ticks by CLOCKS_PER_SEC. The C version of the mult procedure is given in Program 1.2 (page 17). This procedure multiplies two 16-bit integers. As you can see from this program listing, it directly follows the algorithm described before. The assembly language version of the procedure is shown in Program 1.3. As you can see from this code, the assembly language statements are inserted into the procedure using the asm construct. For this reason, this method is called inline assembly. We give more details on this method in Chapter 17. At this time, you are not expected to make any sense out of this program.

Section 1.7 Performance: C Versus Assembly Language

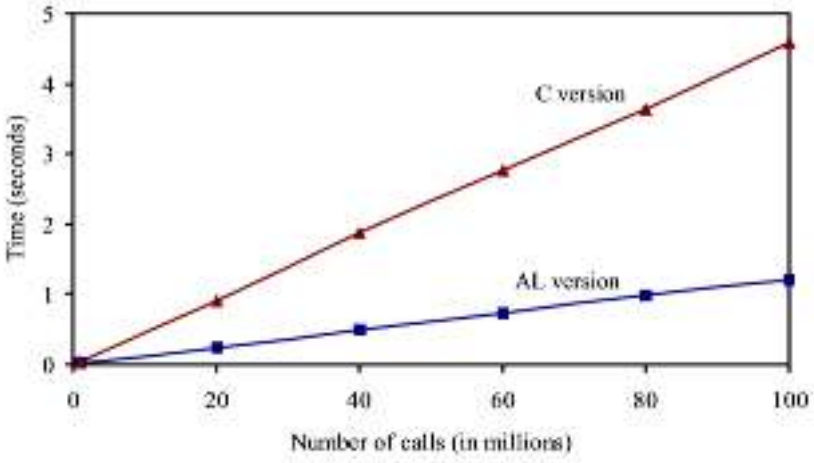

Figure 1.2 Multiplication time comparison on a 2.4-GHz Pentium 4 system: C version uses the multiplication procedure shown in Program 1.2; the assembly language (AL) version uses the assembly language procedure shown in Program 1.3.

Speedup Let us now look at the potential performance benefit we can get from using the assembly language. Toward this end, we present the multiplication times for the C and assembly language versions in Figure 1.2. These timings were obtained on a 2.4-GHz Pentium 4 system running Red Hat Linux 8.0. The y-axis gives the multiplication time in seconds. It is clear from this plot that the assembly language version runs substantially faster than the C version. This plot substantiates our claim that assembly language programs are time efficient. For example, to execute the procedure 100 million times, the C version takes about 3.5 seconds more than the assembly language version. To quantify the performance difference, let us look at the speedup. We define speedup as

Speedup =

Execution time of the C version Execution time of the assembly language version

Speedup values greater than 1 indicate that performance of the assembly language version is better—the higher the speedup, the better the assembly language performance. For our application, we get a speedup of about 4. The reader should be cautioned that the improvement obtained by the assembly language programs depends on the application, compiler, and the type of processor, and so on. It is also important to write an efficient assembly language code in order to get better performance. If we write sloppy assembly language code, it may run slower than the compiler-generated code. This implies that critical analysis and efficient coding are very important to realize the potential performance gains from the assembly language. In practice, assembly language programming is limited to critical sections of a program. When we say critical, we mean either due to the nature of the application (e.g., real-time constraints) or due to performance reasons.