Introduction to Computer Systems

Norman Matloff
University of California, Davis [1]

Contents

Chapter 1

Information Representation and Storage

1.1 Introduction

A computer can store many types of information. A high-level language (HLL) will typically have several data types, such as the C/C++ language's **int**, **float**, and **char**. Yet a computer can not directly store any of these data types. Instead, a computer only stores 0s and 1s. Thus the question arises as to how one can represent the abstract data types of C/C++ or other HLLs in terms of 0s and 1s. What, for example, does a **char** variable look like when viewed from "under the hood"?

A related question is how we can use 0s and 1s to represent our program itself, meaning the machine language instructions that are generated when our C/C++ or other HLL program is compiled. In this chapter, we will discuss how to represent various types of information in terms of 0s and 1s. And, in addition to this question of *how* items are stored, we will also begin to address the question of *where* they are stored, i.e. where they are placed within the structure of a computer's main memory.

1.2 Bits and Bytes

1.2.1 "Binary Digits"

The 0s and 1s used to store information in a computer are called **bits**. The term comes from **b**inary dig**it**, i.e. a digit in the base-2 form of a number (though once again, keep in mind that not all kinds of items that a computer stores are numeric). The physical nature of bit storage, such as using a high voltage to represent a 1 and a low voltage to represent a 0, is beyond the scope of this book, but the point is that every piece of information must be expressed as a string of bits.

For most computers, it is customary to label individual bits within a bit string from right to left, starting with 0. For example, in the bit string 1101, we say Bit 0 = 1, Bit 1 = 0, Bit 2 = 1 and Bit 3 = 1.

If we happen to be using an n-bit string to represent a nonnegative integer, we say that Bit n-1, i.e. the leftmost bit, is the most significant bit (MSB). To see why this terminology makes sense, think of the base-10 case. Suppose the price of an item is $237. A mistake by a sales clerk in the digit 2 would be much more serious than a mistake in the digit 7, i.e. the 2 is the most significant of the three digits in this price. Similarly, in an n-bit string, Bit 0, the rightmost bit, is called the least significant bit (LSB).

A bit is said to be **set** if it is 1, and **cleared** if it is 0.

A string of eight bits is usually called a **byte**. Bit strings of eight bits are important for two reasons. First, in storing characters, we typically store each character as an 8-bit string. Second, computer storage cells are typically composed of an integral number of bytes, i.e. an even multiple of eight bits, with 16 bits and 32 bits being the most commonly encountered cell sizes.

The whimsical pioneers of the computer world extended the pun, "byte" to the term **nibble**, meaning a 4-bit string. So, each hex digit (see below) is called a nibble.

1.2.2 Hex Notation

We will need to define a "shorthand" notation to use for writing long bit strings. For example, imagine how cumbersome it would be for us humans to keep reading and writing a string such as 1001110010101110. So, let us agree to use **hexadecimal** notation, which consists of grouping a bit string into 4-bit substrings, and then giving a single-character name to each substring.

For example, for the string 1001110010101110, the grouping would be

1001 1100 1010 1110

Next, we give a name to each 4-bit substring. To do this, we treat each 4-bit substring as if it were a base-2 number. For example, the leftmost substring above, 1001, is the base-2 representation for the number 9, since

$$1 \cdot (2^3) + 0 \cdot (2^2) + 0 \cdot (2^1) + 1 \cdot (2^0) = 9,$$

so, for convenience we will call that substring "9." The second substring, 1100, is the base-2 form for the number 12, so we will call it "12." However, we want to use a single-character name, so we will call it "c," because we will call 10 "a," 11 "b," 12 "c," and so on, until 15, which we will call "f."

In other words, we will refer to the string 1001110010101110 as 0x9cae. This is certainly much more convenient, since it involves writing only 4 characters, instead of 16 0s and 1s. However, keep in mind that we are doing this only as a quick shorthand form, for use by us humans. The computer is storing the string

in its original form, 1001110010101110, *not* as 0x9cae.

We say 0x9cae is the hexadecimal, or "hex," form of the the bit string 1001110010101110. Often we will use the C-language notation, prepending "0x" to signify hex, in this case 0x9cae.

Recall that we use bit strings to represent many different types of information, with some types being numeric and others being nonnumeric. If we happen to be using a bit string as a nonnegative number, then the hex form of that bit string has an additional meaning, namely the base-16 representation of that number.

For example, the above string 1001110010101110, if representing a nonnegative base-2 number, is equal to

$$\begin{aligned} 1(2^{15}) &+ 0(2^{14}) + 0(2^{13}) + 1(2^{12}) + 1(2^{11}) + 1(2^{10}) + 0(2^9) + 0(2^8) \\ &+ 1(2^7) + 0(2^6) + 1(2^5) + 0(2^4) + 1(2^3) + 1(2^2) + 1(2^1) + 0(2^0) = 40,110. \end{aligned}$$

If the hex form of this bit string, 0x9cae, is treated as a base-16 number, its value is

$$9(16^3) + 12(16^2) + 10(16^1) + 14(16^0) = 40,110,$$

verifying that indeed the hex form is the base-16 version of the number. That is in fact the origin of the term "hexadecimal," which means "pertaining to 16." [But there is no relation of this name to the fact that in this particular example our bit string is 16 bits long; we will use hexadecimal notation for strings of any length.]

The fact that the hex version of a number is also the base-16 representation of that number comes in handy in converting a binary number to its base-10 form. We *could* do such conversion by expanding the powers of 2 as above, but it is much faster to group the binary form into hex, and then expand the powers of 16, as we did in the second equation.

The opposite conversion—from base-10 to binary—can be expedited in the same way, by first converting from base-10 to base-16, and then degrouping the hex into binary form. The conversion of decimal to base-16 is done by repeatedly dividing by 16 until we get a quotient less than 16; the hex digits then are obtained as the remainders and the very last quotient. To make this concrete, let's convert the decimal number 21602 to binary:

```
Divide 21602 by 16, yielding 1350, remainder 2.
Divide 1350 by 16, yielding 84, remainder 6.
Divide 84 by 16, yielding 5, remainder 4.
The hex form of 21602 is thus 5462.
The binary form is thus 0101 0100 0110 0010, i.e. 0101010001100010.
```

The main ingredient here is the repeated division by 16. By dividing by 16 again and again, we are building up powers of 16. For example, in the line

```
Divide 1350 by 16, yielding 84, remainder 6.
```

above, that is our second division by 16, so it is a *cumulative* division by 16^2. [Note that this is why we are dividing by 16, not because the number has 16 bits.]

1.2.3 There Is No Such Thing As "Hex" Storage at the Machine Level!

Remember, hex is merely a convenient notation **for us humans**. It is wrong to say something like "The machine stores the number in hex," "The compiler converts the number to hex," and so on. **It is crucial that you avoid this kind of thinking, as it will lead to major misunderstandings later on.**

1.3 Main Memory Organization

During the time a program is executing, both the program's data and the program itself, i.e. the machine instructions, are stored in main memory. In this section, we will introduce main memory structure. (We will usually refer to main memory as simply "memory.")

1.3.1 Bytes, Words and Addresses

1.3.1.1 The Basics

Memory (this means RAM/ROM) can be viewed as a long string of consecutive bytes. Each byte has an identification number, called an **address**. Again, an address is just an "i.d. number," like a Social Security Number identifies a person, a license number identifies a car, and an account number identifies a bank account. Byte addresses are consecutive integers, so that the memory consists of Byte 0, Byte 1, Byte 2, and so on.

On each machine, a certain number of consecutive bytes is called a **word**. The number of bytes or bits (there are eight times as many bits as bytes, since a byte consists of eight bits) in a word in a given machine is called the machine's **word size**. This is usually defined in terms of the size of number which the CPU addition circuitry can handle, which in recent years has typically been 32 bits. In other words, the CPU's adder inputs two 32-bit numbers, and outputs a 32-bit sum, so we say the word size is 32 bits.

Most CPUs popular today have 32-bit or 64-bit words. As of December 2006, the trend is definitely toward the latter, with many desktop PCs having 64-bit words.

Early members of the Intel CPU family had 16-bit words, while the later ones were extended to 32-bit and then 64-bi size. In order to ensure that programs written for the early chips would run on the later ones, Intel designed the later CPUs to be capable of running in several modes, one for each bit size.

Note carefully that most machines do not allow overlapping words. That means, for example, that on a 32-bit machine, Bytes 0-3 will form a word and Bytes 4-7 will form a word, but Bytes 1-4 do NOT form a word. If your program tries to access the "word" consisting of Bytes 1-4, it may cause an execution error. On UNIX systems, for instance, you may get the error message "bus error." However, an exception to this is Intel chips, which do not require alignment on word boundaries like this.

Just as a bit string has its most significant and least significant bits, a word will have its most significant and least significant bytes. To illustrate this, suppose word size is 32 bits and consider storage of the integer 25, which is

```
00000000000000000000000000011001
```

in bit form and 0x00000019 as hex. Three bytes will each contain 0x00 and the fourth 0x19, with the 0x19 byte being the least significant and the first 0x00 byte being most significant.

Not only does each byte have an address, but also each word has one too. The address of a word will be the address of its lowest-address byte. So for instance Bytes 4-7 comprise Word 4.

1.3.1.2 Word Addresses

1.3.1.3 "Endian-ness"

Recall that the word size of a machine is the size of the largest string on which the hardware is capable of performing addition. A question arises as to whether the lowest-address byte in a word is treated by the hardware as the most or least significant byte.

The Intel family handles this in a **little-endian** manner, meaning that the least significant byte within a word has the lowest address. For instance, consider the above example of the integer 25. Suppose it is stored in Word 204, which on any 32-bit machine will consist of Bytes 204, 205, 206 and 207. On a 32-bit Intel machine (or any other 32-bit little-endian machine), Byte 204 will be the least significant byte, and thus in this example will contain 0x19.

Note carefully that when we say that Byte 204 contains the least significant byte, what this really means is that the arithmetic hardware in our machine will treat it as such. If for example we tell the hardware to add the contents of Word 204 and the contents of Word 520, the hardware will start at Bytes 204 and 520, not at Bytes 207 and 523. First Byte 204 will be added to Byte 520, recording the carry, if any. Then Byte 205 will be added to Byte 521, plus the carry if any from the preceding byte addition, and so on, through Bytes 207 and 523.

SPARC chips, on the other hand, assign the least significant byte to the highest address, a **big-endian** scheme. This is the case for IBM mainframes too, as well as for the Java Virtual Machine.

Some chips, such as MIPS and PowerPC, even give the operating system a choice as to which rules the CPU will follow; when the OS is booted, it establishes which "endian-ness" will be used.

The endian-ness problem also arises on the Internet. If someone is running a Web browser on a little-endian machine but the Web site's server is big-endian, they won't be able to communicate. Thus as a standard, the Internet uses big-endian order. There is a UNIX system call, **htons()**, which takes a byte string and does the conversion, if necessary.

Here is a function that can be used to test the endian-ness of the machine on which it is run:

```
1  int Endian()  // returns 1 if the machine is little-endian, else 0
2
3  {  int X;
4     char *PC;
5
6     X = 1;
7     PC = (char *) &X;
8     return *PC;
9  }
```

As we will discuss in detail later, compilers usually choose to store **int** variables one per word, and **char** variables one per byte. So, in this little program, **X** will occupy four bytes on a 32-bit machine, which we assume here; **PC** will point to one byte.

Suppose for example that **X** is in memory word 4000, so **&X** is 4000. Then **PC** will be 4000 too. Word 4000 consists of bytes 4000, 4001, 4002 and 4003. Since **X** is 1, i.e. $\underbrace{000...00}_{31\ 0s}1$, one of those four bytes will contain 1 and the others will contain 0. The 1 will be in byte 4000 if and only the machine is little-endian. In the `return` line, **PC** is pointing to byte 4000, so the return value will be either 1 or 0, depending on whether the machine is little- or big-endian, just what we wanted.

Note that within a byte there is no endian-ness issue. Remember, the endian-ness issue is defined at the word level, in terms of addresses of bytes within words; the question at hand is, "Which byte within a word is treated as most significant—the lowest-numbered-address byte, or the highest one?" This is because there is no addressing of bits within a byte, thus no such issue at the byte level.

Note that a C compiler treats hex as base-16, and thus the endian-ness of the machine will be an issue. For example, suppose we have the code

```
int Z = 0x12345678;
```

and &Z is 240.

As we will see later, compilers usually store an **int** variable in one word. So, Z will occupy Bytes 240, 241, 242 and 243. So far, all of this holds independent of whether the machine is big- or little-endian. But the endian-ness will affect which bytes of Z are stored in those four addresses.

On a little-endian machine, the byte 0x78, for instance, will be stored in location 240, while on a big-endian machine it would be in location 243.

Similarly, a call to **printf()** with **%x** format will report the highest-address byte first on a little-endian machine, but on a big-endian machine the call would report the lowest-address byte first. The reason for this is that the C standard was written with the assumption that one would want to use **%** format only in situations in which the programmer intends the quantity to be an integer. Thus the endian-ness will be a factor. This is a very important point to remember if you are using a call to **printf()** with **%x** format to determine the actual bit contents of a word which might not be intended as an integer.

There are some situations in which one can exploit the endian-ness of a machine. An example is given in Section 1.10.

1.3.1.4 Other Issues

Many machines insist that words be **aligned**. In a 32-bit machine, this would mean that words only begin at addresses which are multiples of 4 (32 bits is 4 bytes). Thus Bytes 568-571 would form Word 568, but Bytes 569-572 would not be considered a word.

As we saw above, the address of a word is defined to be the address of its lowest-numbered byte. This presents a problem: How can we specify that we want to access, say, Byte 52 instead of Word 52? The answer is that for machine instruction types which allow both byte and word access (some instructions do, others do not), the instruction itself will indicate whether we want to access Byte x or Word x.

For example, we mentioned earlier that the Intel instruction 0xc7070100 in 16-bit mode puts the value 1 into a certain "cell" of memory. Since we now have the terms *word* and *byte* to work with, we can be more specific than simply using the word *cell*: The instruction 0xc7070100 puts the value 1 into a certain *word* of memory; by contrast, the instruction 0xc60701 puts the value 1 into a certain *byte* of memory. You will see the details in later chapters, but for now you can see that differentiating between byte access and word access *is* possible, and is indicated in the bit pattern of the instruction itself.

Note that the word size determines capacity, depending on what type of information we wish to store. For example:

(a) Suppose we are using an n-bit word to store a nonnegative integer. Then the range of numbers that we can store will be 0 to $2^n - 1$, which for n = 16 will be 0 to 65,535, and for n = 32 will be 0 to 4,294,967,295.

(b) If we are storing a signed integer in an n-bit word, then the information presented in Section 1.4.1 will show that the range will be -2^{n-1} to $2^{n-1} - 1$, which will be -32,768 to +32,767 for 16-bit words, and -2,147,483,648 to +2,147,483,647 for 32-bit words.

(c) Suppose we wish to store characters. Recall that an ASCII character will take up seven bits, not eight.

But it is typical that the seven is "rounded off" to eight, with 1 bit being left unused (or used for some other purpose, such as a technique called **parity**, which is used to help detect errors). In that case, machines with 16-bit words can store two characters per word, while 32-bit machines can store four characters per word.

(d) Suppose we are storing machine instructions. Some machines use a fixed instruction length, equal to the word size. These are the so-called RISC machines, to be discussed in a future unit. On the other hand, most older machines have instructions are of variable lengths.

On earlier Intel machines, for instance, instructions were of lengths one to six bytes (and the range has grown much further since then). Since the word size on those machines was 16 bits, i.e. two bytes, we see that a memory word might contain two instructions in some cases, while in some other cases an instruction would be spread out over several words. The instruction 0xc7070100 mentioned earlier, for example, takes up four bytes (count them!), thus two words of memory.[1]

It is helpful to make an analogy of memory cells (bytes or words) to bank accounts, as mentioned above. Each individual bank account has an account number and a balance. Similarly, each memory has its address and its contents.

As with anything else in a computer, an address is given in terms of 0s and 1s, i.e. as a base-2 representation of an unsigned integer. The number of bits in an address is called the **address size**. Among earlier Intel machines, the address size grew from 20 bits on the models based on the 8086 CPU, to 24 bits on the 80286 model, and then to 32 bits for the 80386, 80486 and Pentium. The current trend is to 64 bits.

The address size is crucial, since it puts an upper bound on how many memory bytes our system can have. If the address size is n, then addresses will range from 0 to $2^n - 1$, so we can have at most 2^n bytes of memory in our system. It is similar to the case of automobile license plates. If for example, license plates in a certain state consist of three letters and three digits, then there will be only $26^3 10^3 = 17,560,000$ possible plates. That would mean we could have only 17,560,000 cars and trucks in the state.

Keep in mind that an address is considered as unsigned integer. For example, suppose our address size is, to keep the example simple, four bits. Then the address 1111 is considered to be +15, not -1.

IMPORTANT NOTATION:

We will use the notation c( ) to mean "contents of," e.g. c(0x2b410) means the contents of memory word 0x2b410. Keep this in mind, as we will use it throughout the course

Today it is customary to design machines with address size equal to word size. To see why this makes sense, consider this code:

```
int X,*P;
...
P = &X;
```

[1] Intel machinese today are still of the CISC type.

The variable **P** is a pointer and thus contains an address. But **P** is a variable in its own right, and thus will be stored in some word. For example, we may have **&X** and **P** equal to 200 and 344, respectively. Then we will have c(344) = 200, i.e. an address will be stored in a word. So it makes sense to have address size equal to word size.

1.4 Representing Information as Bit Strings

We may now address the questions raised at the beginning of the chapter. How can the various abstract data types used in HLLs, and also the computer's machine instructions, be represented using strings of 0s and 1s?[2]

1.4.1 Representing Integer Data

Representing nonnegative integer values is straightforward: We just use the base-2 representation, such as 010 for the number +2. For example, the C/C++ language data type **unsigned int** (also called simply **unsigned**) interprets bit strings using this representation.

But what about integers which can be either positive or negative, i.e. which are signed? For example, what about the data type **int** in C?

Suppose for simplicity that we will be using 3-bit strings to store integer variables. [Note: We will assume this size for bit strings in the next few paragraphs.] Since each bit can take on either of two values, 0 or 1, there are $2^3 = 8$ possible 3-bit strings. So, we can represent eight different integer values. In other words, we could, for example, represent any integer from -4 to +3, or -2 to +5, or whatever. Most systems opt for a range in which about half the representable numbers are positive and about half are negative. The range -2 to +5, for example, has many more representable positive numbers than negative numbers. This might be useful in some applications, but since most computers are designed as general-purpose machines, they use integer representation schemes which are as symmetric around 0 as possible. The two major systems below use ranges of -3 to +3 and -4 to +3.

But this still leaves open the question as to which bit strings represent which numbers. The two major systems, **signed-magnitude** and **2s complement**, answer this question in different ways. Both systems store the nonnegative numbers in the same way, by storing the base-2 form of the number: 000 represents 0, 001 represents +1, 010 represents +2, and 011 represents +3. However, the two systems differ in the way they store the negative numbers, in the following way.

The signed-magnitude system stores a 3-bit negative number first as a 1 bit, followed by the base-2 representation of the magnitude, i.e. absolute value, of that number. For example, consider how the number -3 would be stored. The magnitude of this number is 3, whose base-2 representation is 11. So, the 3-bit,

[2]The word *string* here does not refer to a character string. It simply means a group of bits.

signed-magnitude representation of -3 is 1 followed by 11, i.e. 111. The number -2 would be stored as 1 followed by 10, i.e. 110, and so on. The reader should verify that the resulting range of numbers representable in three bits under this system would then be -3 to +3. The reader should also note that the number 0 actually has *two* representations, 000 and 100. The latter could be considered "-0," which of course has no meaning, and 000 and 100 should be considered to be identical. Note too that we see that 100, which in an unsigned system would represent +4, does *not* do so here; indeed, +4 is not representable at all, since our range is -3 to +3.

The 2s complement system handles the negative numbers differently. To explain how, first think of strings of three decimal digits, instead of three bits. For concreteness, think of a 3-digit odometer or trip meter in an automobile. Think about how we could store positive and negative numbers on this trip meter, if we had the desire to do so. Since there are 10 choices for each digit (0,1,...,9), and there are three digits, there are $10^3 = 1000$ possible patterns. So, we would be able to store numbers which are approximately in the range -500 to +500.

Suppose we can wind the odometer forward or backward with some manual control. Let us initially set the odometer to 000, i.e. set all three digits to 0. If we were to wind *forward* from 000 once, we would get 001; if we were to wind forward from 000 twice, we would get 002; and so on. So we would use the odometer pattern 000 to represent 0, 001 to represent +1, 002 to represent +2, ..., and 499 to represent +499. If we were to wind *backward* from 000 once, we would get 999; if we were to wind backward twice, we would get 998; and so on. So we would use the odometer pattern 999 to represent -1, use 998 to represent -2, ..., and use 500 to represent -500 (since the odometer would read 500 if we were to wind backward 500 times). This would give us a range -500 to +499 of representable numbers.

Getting back to strings of three binary digits instead of three decimal digits, we apply the same principle. If we wind backward once from 000, we get 111, so we use 111 to represent -1. If we wind backward twice from 000, we get 110, so 110 will be used to represent -2. Similarly, 101 will mean -3, and 100 will mean -4. If we wind backward one more time, we get 011, which we already reserved to represent +3, so -4 will be our most negative representable number. So, under the 2s complement system, 3-bit strings can represent any integer in the range -4 to +3.

This may at first seem to the reader like a strange system, but it has a very powerful advantage: We can do addition of two numbers without worrying about their signs; whether the two addends are both positive, both negative or of mixed signs, we will do addition in the same manner. For example, look at the base-10 case above, and suppose we wish to add +23 and -6. These have the "trip meter" representations 023 and 994. Adding 023 and 994 yields 1017, but since we are working with 3-digit quantities, the leading 1 in 1017 is lost, and we get 017. 017 is the "trip meter" representation of +17, so our answer is +17—exactly as it should be, since we wanted to add +23 and -6. The reason this works is that we have first wound forward 23 times (to 023) but then wound backward 6 times (the 994), for a net winding forward 17 times.

The importance of this is that in building a computer, the hardware to do addition is greatly simplified. The same hardware will work for all cases of signs of addends. For this reason, most modern computers are designed to use the 2s-complement system.

For instance, consider the example above, in which we want to find the representation of -29 in an 8-bit string. We first find the representation of +29, which is 00011101 [note that we remembered to include the three leading 0s, as specified in (a) above]. Applying Step (b) to this, we get 11100010. Adding 1, we get 11100011. So, the 8-bit 2s complement representation of -29 is 11100011. We would get this same string if we wound back from 000000 29 times, but the method here is much quicker.

This transformation is **idempotent**, meaning that it is its own inverse: If you take the 2s complement representation of a negative number -x and apply Steps (b) and (c) above, you will get +x. The reader should verify this in the example in the last paragraph: Apply Steps (b) and (c) to the bit string 11100011 representing -29, and verify that the resulting bit string does represent +29. In this way, one can find the base-10 representation of a negative number for which you have the 2s complement form.

By the way, the n-bit representation of a negative integer -x is equal to the base-2 representation of $2^n - x$. You can see this by noting first that the base-2 representation of 2^n is $1\underbrace{000...00}_{n\ 0s}$. That means that the n-bit 2s complement "representation" of 2^n—it is out of range for n bits, but we can talk about its truncation to n bits—is $\underbrace{000...00}_{n\ 0s}$. Since the 2s complement representation of -x is the result of winding backwards x times from $\underbrace{000...00}_{n\ 0s}$, that is the result of winding backwards x times from 2^n, which is $2^n - x$.

For example, consider 4-bit 2s complement storage. Winding backward 3 times from 0000, we get 1101 for the representation of -3. But taken as an unsigned number, 1101 is 13, which sure enough is $2^4 - 3$.

Although we have used the "winding backward" concept as our informal definition of 2s complement representation of negative integers, it should be noted that in actual computation—both by us humans and by the hardware—it is inconvenient to find representations this way. For example, suppose we are working with 8-bit strings, which allow numbers in the range -128 to +127. Suppose we wish to find the representation of -29. We *could* wind backward from 00000000 29 times, but this would be very tedious.

Fortunately, a "shortcut" method exists: To find the n-bit 2s complement representation of a negative number -x, do the following.

(a) Find the n-bit base-2 representation of +x, making sure to include any leading 0s.

(b) In the result of (a), replace 0s by 1s and 1s by 0s. (This is called the **1s complement** of x.)

(c) Add 1 to the result of (b), ignoring any carry coming out of the Most Significant Bit.

Here's why the shortcut works: Say we have a number x whose 2s complement form we know, and we want to find the 2s complement form for -x. Let x' be the 1s complement of x, i.e. the result of interchanging 1s and 0s in x. Then x+x' is equal to $\underbrace{111...11}_{n\ 1s}$, so x+x' = -1. That means that -x = x'+1, which is exactly the shortcut above.

Here very important properties of 2s-complement storage:

(i) The range of integers which is supported by the n-bit, 2s complement representation is -2^{n-1} to $2^{n-1}-1$.

(ii) The values -2^{n-1} and $2^{n-1}-1$ are represented by 10000...000 and 01111...111, respectively.

(iii) All nonnegative numbers have a 0 in Bit n-1, and all negative numbers have a 1 in that bit position.

The reader should verify these properties with a couple of example in the case of 4-bit strings.

By the way, due to the slight asymmetry in the range in (i) above, you can see that we can not use the "shortcut" method if we need to find the 2s complement representation of the number -2^n-1; Step (a) of that method would be impossible, since the number 2^n-1 is not representable. Instead, we just use (ii).

Now consider the C/C++ statement

```
Sum = X + Y;
```

The value of Sum might become negative, even if both the values X and Y are positive. Here is why, say for 16-bit word size: With X = 28,502 and Y = 12,344, the resulting value of Sum will be -24,690. Most machines have special bits which can be used to detect such situations, so that we do not use misleading information.

Again, most modern machines use the 2s complement system for storing signed integers. We will assume this system from this point on, except where stated otherwise.

1.4.2 Representing Real Number Data

The main idea here is to use **scientific notation**, familiar from physics or chemistry, say 3.2×10^{-4} for the number 0.00032. In this example, 3.2 is called the **mantissa** and -4 is called the **exponent**.

The same idea is used to store real numbers, i.e. numbers which are not necessarily integers (also called **floating-point** numbers), in a computer. The representation is essentially of the form

$$m \times 2^n \tag{1.1}$$

with m and n being stored as individual bit strings.

1.4.2.1 "Toy" Example

Say for example we were to store real numbers as 16-bit strings, we might devote 10 bits, say Bits 15-6, to the mantissa m, and 6 bits, say Bits 5-0, to the exponent n. Then the number 1.25 might be represented as

$$5 \times 2^{-2} \tag{1.2}$$

that is, with m = 5 and n = -2. As a 10-bit 2s complement number, 5 is represented by the bit string 0000000101, while as a 6-bit 2s complement number, -2 is represented by 111110. Thus we would store the number 1.25 as the 16-bit string 0000000101 111110 i.e.

```
0000000101111110 = 0x017e
```

Note the design tradeoff here: The more bits I devote to the exponent, the wider the range of numbers I can store. But the more bits I devote to the mantissa, the less roundoff error I will have during computations. Once I have decided on the string size for my machine, in this example 16 bits, the question of partitioning these bits into mantissa and exponent sections then becomes one of balancing accuracy and range.

1.4.2.2 IEEE Standard

The floating-point representation commonly used on today's machines is a standard of the Institute of Electrical and Electronic Engineers (IEEE). For the C type **float**, the standard uses 32-bit storage, with 64 bits used for **double**. The 32-bit case, which we will study here, follows the same basic principles as with our simple example above, but it has a couple of refinements to the simplest mantissa/exponent format. It consists of a Sign Bit, an 8-bit Exponent Field, and 23-bit Mantissa Field. These fields will now be explained. Keep in mind that there will be a distinction made between the terms *mantissa* and *Mantissa Field*, and between *exponent* and *Exponent Field*.

Recall that in base-10, digits to the right of the decimal point are associated with negative powers of 10. For example, 4.38 means

$$4(10^0) + 3(10^{-1}) + 8(10^{-2}) \tag{1.3}$$

It is the same principle in base-2, of course, with the base-2 number 1.101 meaning

$$1(2^0) + 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) \tag{1.4}$$

that is, 1.625 in base-10.

Under the IEEE format, the mantissa must be in the form $\pm 1.x$, where 'x' is some bit string. In other words, the absolute value of the mantissa must be a number between 1 and 2. The number 1.625 is 1.101 in base-2, as seen above, so it already has this form. Thus we would take the exponent to be 0, that is, we would represent 1.625 as

$$1.101 \times 2^0 \tag{1.5}$$

What about the number 0.375? In base-2 this number is 0.011, so we *could* write 0.375 as

$$0.011 \times 2^0 \tag{1.6}$$

but again, the IEEE format insists on a mantissa of the form $\pm 1.x$ So, we would write 0.375 instead as

$$1.1 \times 2^{-2} \tag{1.7}$$

which of course is equivalent, but the point is that it fits IEEE's convention.

Now since that convention requires that the leading bit of the mantissa be 1, there is no point in storing it! Thus the Mantissa Field only contains the bits to the right of that leading 1, so that the mantissa consists of $\pm 1.x$, where 'x' means the bits stored in the Mantissa field. The sign of the mantissa is given by the Sign Bit, 0 for positive, 1 for negative.[3] The circuitry in the machine will be set up so that it restores the leading "1." at the time a computation is done, but meanwhile we save one bit per **float**.[4]

Note that the Mantissa Field, being 23 bits long, represents the fractional portion of the number to 23 "decimal places," i.e. 23 binary digits. So for our example of 1.625, which is 1.101 base 2, we have to write 1.101 as 1.10100000000000000000000.[5] So the Mantissa Field here would be 10100000000000000000000.

The Exponent Field actually does *not* directly contain the exponent; instead, it stores the exponent plus a **bias** of 127. The Exponent Field itself is considered as an 8-bit unsigned number, and thus has values ranging from 0 to 255. However, the values 0 and 255 are reserved for "special" quantities: 0 means that the floating-point number is 0, and 255 means that it is in a sense "infinity," the result of dividing by 0, for example. Thus the Exponent Field has a range of 1 to 254, which after accounting for the bias term mentioned above means that the exponent is a number in the range -126 to +127 (1-127) = -126 and 254-127 = +127).

[3] Again, keep in mind the distinction between the mantissa and the Mantissa field. Here the mantissa is $\pm 1.x$ while the Mantissa field is just x.

[4] This doesn't actually make storage shorter; it simply gives us an extra bit position to use otherwise, thus increasing accuracy.

[5] Note that trailing 0s do not change things in the fractional part of a number. In base 10, for instance, the number 1.570000 is the same as the number 1.57.

Note that the floating-point number is being stored is (except for the sign) equal to

$$(1 + M/2^{23}) \times 2^{(E-127)} \tag{1.8}$$

where M is the Mantissa and E is the Exponent. Make sure you agree with this.

With all this in mind, let us find the representation for the example number 1.625 mentioned above. We found that the mantissa is 1.101 and the exponent is 0, and as noted earlier, the Mantissa Field is 10100000000000000000000. The Exponent Field is 0 + 127 = 127, or in bit form, 01111111.

The Sign Bit is 0, since 1.625 is a positive number.

So, how are the three fields then stored altogether in one 32-bit string? Well, 32 bits fill four bytes, say at addresses n, n+1, n+2 and n+3. The format for storing the three fields is then as follows:

- Byte n: least significant eight bits of the Mantissa Field
- Byte n+1: middle eight bits of the Mantissa Field
- Byte n+2: least significant bit of the Exponent Field, and most significant seven bits of the Mantissa Field
- Byte n+3: Sign Bit, and most significant seven bits of the Exponent Field

Suppose for example, we have a variable, say T, of type **float** in C, which the compiler has decided to store beginning at Byte 0x304a0. If the current value of T is 1.625, the bit pattern will be

```
Byte 0x304a0:  0x00; Byte 0x304a1:  0x00; Byte 0x304a2:  0xd0; Byte
0x304a3:  0x3f
```

The reader should also verify that if the four bytes' contents are 0xe1 0x7a 0x60 0x42, then the number being represented is 56.12.

Note carefully: The storage we've been discussing here is NOT base-10. It's not even base-2, though certain components within the format are base-2. It's a different kind of representation, not "base-based."

1.4.3 Representing Character Data

This is merely a matter of choosing which bit patterns will represent which characters. The two most famous systems are the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Information Code (EBCDIC). ASCII stores each character as the base-2 form of a

number between 0 and 127. For example, 'A' is stored as 65_{10} (01000001 = 0x41), '%' as 37_{10} (00100101 = 0x25), and so on.

A complete list of standard ASCII codes may be obtained by typing

```
man ascii
```

on most UNIX systems. Note that even keys such as Carriage Return, Line Feed, and so on, are considered characters, and have ASCII codes.

Since ASCII codes are taken from numbers in the range 0 to $2^7 - 1 = 127$, each code consists of seven bits. The EBCDIC system consists of eight bits, and thus can code 256 different characters, as opposed to ASCII's 128. In either system, a character can be stored in one byte. The vast majority of machines today use the ASCII system.

What about characters in languages other than English? Codings exist for them too. Consider for example Chinese. Given that there are tens of thousands of characters, far more than 256, two bytes are used for each Chinese character. Since documents will often contain both Chinese and English text, there needs to be a way to distinguish the two. Big5 and Guobiao, two of the most widely-used coding systems used for Chinese, work as follows. The first of the two bytes in a Chinese character will have its most significant bit set to 1. This distinguishes it from ASCII (English) characters, whose most significant bits are 0s. The software which is being used to read (or write) the document, such as **cemacs**, a Chinese version of the famous **emacs** text editor, will inspect the high bit of a byte in the file. If that bit is 0, then the byte will be interpreted as an ASCII character; if it is 1, then that byte and the one following it will be interpreted as a Chinese character.[6]

1.4.4 Representing Machine Instructions

Each computer type has a set of binary codes used to specify various operations done by the computer's **Central Processing Unit** (CPU). For example, in the Intel CPU chip family, the code 0xc7070100, i.e.

```
11000111000001110000000100000000,
```

means to put the value 1 into a certain cell of the computer's memory. The circuitry in the computer is designed to recognize such patterns and act accordingly. You will learn how to generate these patterns in later chapters, but for now, the thing to keep in mind is that a computer's machine instructions consist of patterns of 0s and 1s.

Note that an instruction can get into the computer in one of two ways:

[6]Though in the Chinese case the character will consist of two bytes whether we use the Big5 or Guobiao systems, with the first bit being 1 in either cased, the remaining 15 bits will be the different under the Guobiao encoding than under the Big5 one.

(a) We write a program in machine language (or assembly language, which we will see is essentially the same), directly producing instructions such as the one above.

(b) We write a program in a high-level language (HLL) such as C, and the compiler translates that program into instructions like the one above.

[By the way, the reader should keep in mind that the compilers themselves are programs. Thus they consist of machine language instructions, though of course these instructions might have themselves been generated from an HLL source too.]

1.4.5 What Type of Information is Stored Here?

A natural question to ask at this point would be how the computer "knows" what kind of information is being stored in a given bit string. For example, suppose we have the 16-bit string 0111010000101011, i.e. in hex form 0x742b, on a machine using an Intel CPU chip in 16-bit mode. Then

(a) if this string is being used by the programmer to store a signed integer, then its value will be 29,739;

(b) if this string is being by the programmer used to store characters, then its contents will be the characters 't' and '+';

(c) if this string is being used by the programmer to store a machine instruction, then the instruction says to "jump" (like a **goto** in C) forward 43 bytes.

So, in this context the question raised above is,

> How does the computer "know" which of the above three kinds (or other kinds) of information is being stored in the bit string 0x742b? Is it 29,739? Is it 't' and '+'? Or is it a jump-ahead-43-bytes machine instruction?

The answer is, "The computer does *not* know!" As far as the computer is concerned, this is just a string of 16 0s and 1s, with *no* special meaning. So, the responsibility rests with the person who writes the program—he or she must remember what kind of information he or she stored in that bit string. If the programmer makes a mistake, the computer will not notice, and will carry out the programmer's instruction, no matter how ridiculous it is. For example, suppose the programmer had stored *characters* in each of two bit strings, but forgets this and mistakenly thinks that he/she had stored *integers* in those strings. If the programmer tells the computer to multiply those two "numbers," the computer will dutifully obey!

The discussion in the last paragraph refers to the case in which we program in machine language directly. What about the case in which we program in an HLL, say C, in which the *compiler* is producing this machine

language from our HLL source? In this case, during the time the compiler is translating the HLL source to machine language, the compiler must "remember" the type of each variable, and react accordingly. In other words, the responsibility for handling various data types properly is now in the hands of the compiler, rather than directly in the hands of the programmer—but still not in the hands of the hardware, which as indicated above, remains ignorant of type.

1.5 Examples of the Theme, "There Are No Types at the Hardware Level"

In the previous sections we mentioned several times that the hardware is ignorant of data type. We found that it is the software which enforces data types (or not), rather than the hardware. This is such an important point that in this section we present a number of examples with this theme. Another theme will be the issue of the roles of hardware and software, and in the latter case, the roles of your own software versus the OS and compiler.

1.5.1 Example

As an example, suppose in a C program X and Y are both declared of type **char**, and the program includes the statement

```
X += Y;
```

Of course, that statement is nonnsense. But the hardware knows nothing about type, so the hardware wouldn't care if the compiler were to generate an add machine instruction from this statement. Thus the only gatekeeper, if any, would be the compiler. The compiler could either (a) just ignore the oddity, and generate the add instruction, or (b) refuse to generate the instruction, and issue an error message. In fact the compiler will do (a), but the main point here is that the compiler is the only possible gatekeeper here; the hardware doesn't care.

So, the compiler won't prevent us from doing the above statement, and will produce machine code from it. However, the compiler will produce different machine code depending on whether the variables are of type **int** or **char**. On an Intel-based machine, for example, there are two[7] forms of the addition instruction, one named **addl** which operates on 32-bit quantities and another named **addb** which works on 8-bit quantities. The compiler will store **int** variables in 32-bit cells but will store **char** variables in 8-bit cells.[8] So, the compiler will react to the C code

```
X += Y;
```

[7] More than two, actually.

[8] Details below.

by generating an **addl** instruction if X and Y are both of type **int** or generating an **addb** instruction, if they are of type **char**.

The point here, again, is that it is the software which is controlling this, not the hardware. The hardware will obey whichever machine instructions you give it, even if they are nonsense.

1.5.2 Example

So, the machine doesn't know whether we humans intend the bit string we have stored in a 4-byte memory cell to be interpreted as an integer or as a 4-element character string or whatever. To the machine, it is just a bit string, 32 bits long.

The place the notion of types arises is at the compiler/language level, not the machine level. The C/C++ language has its notion of types, e.g. **int** and **char**, and the compiler produces machine code accordingly.[9] But that machine code itself does not recognize type. Again, the machine cannot tell whether the contents of a given word are being thought of by the programmer as an integer or as a 4-character string or whatever else.

For example, consider this code:

```
...
int Y;  // local variable
...
strncpy(&Y,"abcd",4);
...
```

At first, you may believe that this code would not even compile successfully, let alone run correctly. After all, the first argument to **strncpy()** is supposed to be of type **char ***, yet we have the argument as type **int ***. But the C compiler, say GCC, will indeed compile this code without error,[10] and the machine code will indeed run correctly, placing "abcd" into Y. The machine won't know about our argument type mismatch.

If we run the same code through a C++ compiler, say **g++**, then the compiler will give us an error message, since C++ is strongly typed. We will then be forced to use a cast:

```
strncpy((char *) &Y,"abcd",4);
```

1.5.3 Example

When we say that the hardware doesn't know types, that includes array types. Consider the following program:

[9] For example, as we saw above, the compiler will generate word-accessing machine instructions for **int**s and byte-accessing machine instructions for **char**s.

[10] It may give a warning message, though.

```
1  main()
2
3  {  int X[5],Y[20],I;
4
5     X[0] = 12;
6     scanf("%d",&I);  // read in I = 20
7     Y[I] = 15;
8     printf("X[0] = %d\n",X[0]);  // prints out 15!
9  }
```

There appears to be a glaring problem with **Y** here. We assign 15 to **Y[20]**, even though to us humans there is no such thing as **Y[20]**; the last element of **Y** is **Y[19]**. Yet the program will indeed run without any error message, and 15 will be printed out.

To understand why, keep in mind that at the machine level there is really no such thing as an array. **Y** is just a name for the first word of the 20 words we humans think of as comprising one package here. When we write the C/C++ expression **Y[I]**, the compiler merely translates that to machine code which accesses "the location **I int**s after **Y**."

This should make sense to you since another way to write **Y[I]** is **Y+I**. So, there is nothing syntactically wrong with the expression **Y[20]**. Now, where is "**Y[20]**"? C/C++ rules require that local variables be stored in reverse order,[11] i.e. **Y** first and then **X**. So, **X[0]** will follow immediately after **Y[19]**. Thus "**Y[20]**" is really **X[0]**, and thus **X[0]** will become equal to 15!

Note that the compiler could be designed to generate machine code which checks for the condition $\mathbf{Y} > 19$. But the official C/C++ standards do not require this, and it is not usually done. In any case, the point is again that it is the software which might do this, not the hardware. Indeed, the hardware doesn't even know that we have variables **X** and **Y**, that **Y** is an array, etc.

1.5.4 Example

As another example, consider the C/C++ library function **printf()**, which is used to write the values of program variables to the screen. Consider the C code

```
1  int W;
2  ...
3  W = -32697;
4  printf("%d %u %c\n",W,W,W);
```

again on a machine using an Intel CPU chip in 16-bit mode. We are printing the bit string in **W** to the screen three times, but are telling **printf()**, "We want this bit string to first be interpreted as a decimal signed integer (%d); then as a decimal unsigned integer (%u); then as an ASCII character (%c). Here is the output that would appear on the screen:

[11] Details below.

```
-32697  32839  G
```

The bit string in **W** is 0x8047. Interpreted as a 16-bit 2s complement number, this string represents the number -32,697. Interpreted as an unsigned number, this string represents 32,839. If the least significant 8 bits of this string are interpreted as an ASCII character (which is the convention for %c), they represent the character 'G'.

But remember, the key point is that the *hardware* is ignorant; it has no idea as to what type of data we intended to be stored in **W**'s memory location. The interpretation of data types was solely in the software. As far as the hardware is concerned, the contents of a memory location is just a bit string, nothing more.

1.5.5 Example

In fact, we can view that bit string without interpretation as some data type, by using the %x format in the call to **printf()**. This will result in the bit string itself being printed out (in hex notation). In other words, we are telling **printf()**, "Just tell me what bits are in this string; don't do any interpretation." Remember, hex notation is just that—notation, a shorthand system to make things easier on us humans, saving us the misery of writing out lengthy bit strings in longhand. So here we are just asking **printf()** to tells us what bits are in the variable being queried.[12]

A similar situation occurs with input. Say on a machine with 32-bit memory cells we have the statement

```
scanf("%x",&X);
```

and we input bbc0a168.[13] Then we are saying, "Put 0xb, i.e. 1011, in the first 4 bits of X (i.e. the most-significant 4 bits of X), then put 1011 in the next 4 bits, then put 1100 in the next 4 bits, etc. Don't do any interpretation of the meaning of the string; just copy these bits to the memory cell named X." So, the memory cell X will consist of the bits 10111011110000010100000101101000.

By contrast, if we have the statement

```
scanf("%d",&X);
```

and we input, say, 168, then we are saying, "Interpret the characters typed at the keyboard as describing the base-10 representation of an integer, then calculate that number (do $1\times100+6\times10+8$), and store that number in base-2 form in X." So, the memory cell X will consist of the bits 00000000000000000000000010101000

So, in summary, in the first of the two **scanf()** calls above we are simply giving the machine specific bits to store in X, while in the second one we are asking the machine to convert what we input into some bit string and place that in X.

[12] But the endian-ness of the machine will play a role, as explained earlier.

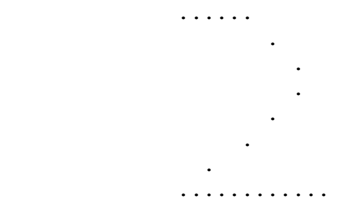
[13] Note that we do not type "0x".

1.5.6 Example

By the way, **printf()** is only a function within the C/C++ library, and thus does not itself directly do I/O. Instead, **printf()** calls another function, **write()**, which is in the operating system (and thus the call is referred to as a **system call**).[14] All **printf()** does is convert what we want to write to the proper sequence of bytes, and then **write()** does the actually writing to the screen.

So,

```
int W = 25;
...
printf("%d",W);
```



Then **printf()** will take the memory cell **W**, which will contain 00000000000000000000000000011001, determine that we want the characters '2' and '5' to be sent to the screen, and then call **write()** with the ASCII codes for those two characters (0110010 and 0110101, respectively). Then **write()** will actually send 0110010 and 0110101 to the screen.

Similarly, when we call **scanf()**, it in turn calls the system function **read()**, which does the actual reading from the keyboard.

1.6 Visual Display

1.6.1 The Basics

Let us continue the 16-bit example above, in which "-32697" is printed, in order to learn more about how computer monitors work.

The machine instructions in **write()** print out the character string "-32697" to the screen. In other words, those instructions will first send 0x2d, the ASCII code for the character '-', to the screen, 0x33 for '3', 0x32 for '2', and so on. It is easy to take this for granted, but there is quite a bit involved here.

[14] The name of the function is write() on UNIX; the name is different in other OSs.

First of all, keep in mind the vast difference between the *number* -32697 and the *character string* "-32697". In our context, it is the former which is stored in **W**, as 0x8047, i.e.

```
1000000001000111
```

(Verify this!) We must convert the part after the negative sign, i.e. convert 32,697, to the characters '3', '2', '6', '9' and '7'. How is this done?

The answer is that the code in **printf()** must do repeated division: It first divides by 10,000, yielding 3 with a remainder of 2,697. The 3, i.e. 11, is changed to '3', i.e. the ASCII 00110011. The 2,697 is then divided by 1,000, yielding 2 with a remainder of 697. The 2 is converted to '2', i.e. 00110010, and so on. (We will then divide by 100 and 10.)

The second question is, how will these characters then be displayed on the screen?

You may already know that a computer monitor consists of many dots called **pixels**. There may be, for instance, 1028 rows of 768 pixels per row. Any image, including characters, are formed from these pixels.

On the old nongraphics screens (often called a **plain—or "dumb"—ASCII terminals**), the hardware directly formed the dots for whichever ASCII characters it received from the computer. Software had no control over individual pixels; it only controlled which ASCII character was printed. In our example above, for instance, the software would send 0x33 along the cable from the computer to the screen, and the screen then wolud form the proper dots to draw the character '3', called the **font** for '3'. The screen was capable only of drawing ASCII characters, no pictures etc.

By contrast, on modern graphics screens, software has control over individual pixels. For concreteness, consider an **xterm** window on UNIX systems. (You need not have used **xterm** to follow this discussion. If you have used Microsoft Windows, you can relate the discussion to that context if you wish.) The **write()** call we saw above will send the value 0x33 not directly to the screen, but rather to the **xterm** program. The latter will look up the font for '3' and then send it to the screen, and the screen hardware then displays it.

1.6.2 Non-English Text

A Chinese version of **xterm**, named **cxterm**, will do the same, except it will first sense whether the character is ASCII or Chinese, and send the appropriate font to the screen.

1.6.3 It's the Software, Not the Hardware

Once again, almost all of this process is controlled by software. For instance, in our example above in which we were storing the integer 32,697 and wished to print it to the screen, it was the software—**printf()** and the

OS—which did almost all the work. The only thing the hardware did was print to the correct pixels after the OS sent it the various fonts (which are pixel patterns).

As another example, what do we mean when we say that "A PC uses the ASCII system" ? We do not mean that, for example, the CPU is designed for ASCII; it isn't. We do mean that the C compiler on a PC will use ASCII for **char** variables, that the windowing software, say **xterm** will use ASCII for specifying fonts, etc.

1.6.4 Text Cursor Movement

A related question is, how does cursor movement work? For example, when you are using the **vi** text editor, you can hit the k key (or the up-arrow key) to make the cursor go up one line. How does this work?

Historically, a problem with all this was that different terminals had different ways in which to specify a given type of cursor motion. For example, if a program needed to make the cursor move up one line on a VT100 terminal, the program would need to send the characters Escape, [, and A:

```
printf("%c%c%c",27,'[','A');
```

(the character code for the Escape key is 27). But for a Televideo 920C terminal, the program would have to send the ctrl-K character, which has code 11:

```
printf("%c",11);
```

Clearly, the authors of programs like **vi** would go crazy trying to write different versions for every terminal, and worse yet, anyone else writing a program which needed cursor movement would have to "re-invent the wheel," i.e. do the same work that the That is why the Curses library was developed. The goal was to alleviate authors of cursor-oriented programs like **vi** of the need to write different code for different terminals. The programs would make calls to the API library, and the library would sort out what to do for the given terminal type. One would simply do an **#include** of the Curses header file and link in the Curses library (**-lcurses**), and then have one's program make the API calls. (To learn more about **curses**, see my tutorial on it, at `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf`.)

The library would know which type of terminal you were using, via the environment variable **TERM**. The library would look up your terminal type in its terminal database (the file **/etc/termcap**). When you, the programmer, would call the Curses API to, say, move the cursor up one line, the API would determine which character sequence was needed to make this happen.

For example, if your program wanted to clear the screen, it would not directly use any character sequences like those above. Instead, it would simply make the call

```
clear();
```

and Curses would do the work on the program's behalf.

Many dazzling GUI programs are popular today. But although the GUI programs may provide more "eye candy," they can take a long time to load into memory, and they occupy large amounts of territory on your screen. So, Curses programs such as **vi** and **emacs** are still in wide usage.[15]

For information on Curses programming, see `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Curses.pdf`.

1.6.5 Mouse Actions

Each time you move the mouse, or click it, the mouse hardware sends a pulse of current to the CPU, called an **interrupt**. This causes the OS to run, and it then sends a signal to whatever application program was associated with the mouse action. That program would then act accordingly. And in the case of a mouse movement, the OS would update the position of the mouse pointer on the screen.

1.6.6 Display of Images

Programs which display images work on individual pixels at an even finer level than the fonts used for text, but the principles are the same.

1.7 There's Really No Such Thing As "Type" for Disk Files Either

You will encounter the terms *text file* and *binary file* quite often in the computer world, so it is important to understand them well—especially they are rather misleading.

1.7.1 Disk Geometry

Files are stored on disks. A disk is a rotating round platter with magnetized spots on its surface.[16] Each magnetized spot records one bit of the file.

[15] Interestingly, even some of those classical Curses programs have also become somewhat GUI-ish. For instance **vim**, the most popular version of **vi** (it's the version which comes with most Linux distributions, for example), can be run in **gvim** mode. There, in addition to having the standard keyboard-based operations, one can also use the mouse. One can move the cursor to another location by clicking the mouse at that point; one can use the mouse to select blocks of text for deletion or movement; etc. There are icons at the top of the editing window, for operations like Find, Make, etc.

[16] Colloquially people refer to the disk as a **disk drive**. However, that term should refer only to the motor which turns the disk.

The magnetized spots are located on concentric rings called **tracks**. Each track is further divided in **sectors**, consisting of, say 512 bytes (4096 bits) each.

When one needs to access part of the disk, the read/write head must first be moved from its current track to the track of interest; this movement is called a **seek**. Then the head must wait until the sector of interest rotates around to the head, so that the read or write of the sector can be performed.

When a file is created, the operating system finds unused sectors on the disk in which to place the bytes of the file. The OS then records the locations (track number, sector number within track) of the sectors of the file, so that the file can be accessed by users later on. Each time a user wants to access the file, the OS will look in its records to determine where the file is on disk.

Again, the basic issue will be that the hardware does not know data types. The bits in a file are just that, bits, and the hardware doesn't know if the creator of the file intended those bits to represent numbers or characters or machine instructions or whatever.

1.7.2 Definitions of "Text File" and "Binary File"

Keep in mind that the term **binary file** is a misnomer. After all, ANY file is "binary," whether it consists of "text" or not, in the sense that it consists of bits no matter what. So what do people mean when they refer to a "binary" file?

First, let's define the term **text file** to mean a file satisifying all of the following conditions:

(a) Each byte in the file is in the ASCII range 00000000-01111111, i.e. 0-127.

(b) Each byte in the file is *intended* to be thought of as an ASCII character.

(c) The file is *intended* to be broken into what we think of (and typically display) as lines. Here the term *line* is defined technically in terms of end-of-line (EOL) markers.

In UNIX, the EOL is a single byte, 0xa, while for Windows it is a pair of bytes, 0xd and 0xa. If you write the character '\n' in a C program, it will write the EOL, whichever it is for that OS.

Any file which does not satisfy these conditions has traditionally been termed a **binary file**.[17]

Say for example I use a text editor, say the **vim** extension of **vi**, to create a file named **FoxStory**, whose contents are

```
The quick brown fox
jumped over the fence.
```

[17] Even this definition is arguably too restrictive. If we produce a non-English file which we intend as "text," it will have some non-ASCII bytes.

Then **vim** will write the ASCII codes for the characters 'T', 'h', 'e' and so on (including the ASCII code for newline in the OS we are using) onto the disk. This is a text file. The first byte of the file, for instance, will be 01010100, the ASCII code for 'T', and we do intend that that 01010100 be thought of as 'T' by humans.

On the other hand, consider a JPEG image file, **FoxJumpedFence.jpg**, showing the fox jumping over the fence. The bytes in this file will represent pixels in the image, under the special format used by JPEG. It's highly likely that some of those bytes will also be 01010100, just by accident; they are certainly not intended as the letter 'T'. And lots of bytes will be in the range 10000000-11111111, i.e. 128-255, outside the ASCII range. So, this is a binary file.

Other examples of (so-called) binary files:

- audio files
- machine language files
- compressed files

1.7.3 Programs That Access of Text Files

Suppose we display the file **FoxStory** on our screen by typing[18]

```
cat FoxStory
```

Your screen will then indeed show the following:

```
The quick brown fox
jumped over the fence.
```

The reason this occurred is that the **cat** program did interpret the contents of the file **FoxStory** to be ASCII codes. What "interpret" means here is the following:

Consider what happens when **cat** reads the first byte of the file, 'T'. The ASCII code for 'T' is 0x54 = 01010100. The program **cat** contains **printf()** calls which use the **%c** format. This format sends the byte, in this case to the screen.[19] The latter looks up the font corresponding to the number 0x54, which is the font for 'T', and that is why you see the 'T' on the screen.

[18] The **cat** command is UNIX, but the same would be true, for instance, if we typed `type FoxStory` into a command window on a Windows machine.

[19] As noted earlier, in modern computer systems the byte is not directly sent to the screen, but rather to the windowing software, which looks up the font and then sends the font to the screen.

Note also that in the example above, **cat** printed out a new line when it encountered the newline character, ASCII 12, 00001100. Again, **cat** was written to do that, but keep in mind that otherwise 00001100 is just another bit pattern, nothing special.

By contrast, consider what would happen if you were to type

```
cat FoxJumpedFence.jpg
```

The **cat** program will NOT know that **FoxJumpedFence.jpg** is not a text file; on the contrary, **cat** assumes that any file given to it will be a text file. Thus **cat** will use **%c** format and the screen hardware will look up and display fonts, etc., even though it is all meaningless garbage.

1.7.4 Programs That Access Binary Files

These programs are quite different for each application, of course, since the interpretation of the bit patterns will be different, for instance, for an image file than for a machine language file.

One point, though, is that when you deal with such files in, say, C/C++, you may need to warn the system that you will be accessing binary files. In the C library function **fopen()**, for example, to read a binary file you may need to specify **"rb"** mode. In the C++ class **ifstream** you may need to specify the mode **ios:binary**.

The main reason for this is apparently due to the Windows situation. Windows text files use ctrl-z as an end-of-file marker. (UNIX has no end-of-file marker at all, and it simply relies on its knowledge of the length of the file to determine whether a given byte is the last one or not.) Apparently if the programmer did not warn the system that a non-text file is being read, the system may interpret a coincidental ASCII 26 (ctrl-z) in the file as being the end of the file.[20]

1.8 Storage of Variables in HLL Programs

1.8.1 What Are HLL Variables, Anyway?

When you execute a program, both its instructions and its data are stored in memory. Keep in mind, the word *instructions* here means machine language instructions. Again, these machine language instructions were either written by the programmer directly, or they were produced by a compiler from a source file written by the programmer in C/C++ or some other HLL. Let us now look at the storage of the *data* in the C case.[21]

[20] I say "apparently," because after much searching I have not been able to confirm this.

[21] The C language became popular in the early 1980s. It was invented by the ATT&T engineers who developed UNIX. The ensuing popularity of UNIX then led to popularity of C, even for non-UNIX users. Later it was extended to C++, by adding class

In an HLL program, we specify our data via names. The compiler will assign each variable a location in memory. Generally (but not always, depending on the compiler), these locations will be consecutive.

1.8.2 Order of Storage

C compilers tend to store local variables in the reverse of the order in which they are declared. For example, consider the following program:

```
1  PrintAddr(char *VarName,int Addr)
2
3  {  printf("the address of %s is %x\n",VarName,Addr);  }
4
5  main()
6
7  {  int X,Y,W[4];
8     char U,V;
9
10    PrintAddr("X",&X);
11    PrintAddr("Y",&Y);
12    PrintAddr("W[3]",&W[3]);
13    PrintAddr("W[0]",&W[0]);
14    PrintAddr("U",&U);
15    PrintAddr("V",&V);
16 }
```

I ran this on a 32-bit Pentium machine running the Linux operating system. The output of the program was

```
the address of X is bffffb84
the address of Y is bffffb80
the address of W[3] is bffffb7c
the address of W[0] is bffffb70
the address of U is bffffb6f
the address of V is bffffb6e
```

Note that the **int**s are 4 bytes apart while the **char**s are 1 byte apart, reflecting the sizes of these types. More on this below.

On the other hand, there is wide variation among C compilers as to how they store global variables.

1.8.2.1 Scalar Types

Virtually all C compilers store variables of type **int** in one word. This is natural, as the word size is the size of integer operands that the hardware works on.

---

structures (it was originally called "C with classes"). If you are wondering whether a certain C++ construct is also part of C, the answer is basically that if it is not something involving classes (or the **new** operator), it is C.

Variables of type **float** are usually stored in 32-bit units, since that is the size of the IEEE floating-point standard. So, in modern 32-bit machines, a **float** is stored in one word. For 16-bit machines, this means that **float** variables must be stored in a pair of words.

Typically **char** variables are stored as one byte each, since they are 7 or 8 bits long and thus fit in one byte.

By the way, C/C++ treats **char** as an integer type, storing a 1-byte signed integer, as opposed to the typically 4-byte **int**. This can save a lot of space in memory if one knows that 8 bits is enough for each of the integers we will store. Thus in C and even C++, the code

```
char G;
...
if (G < 0) ...
```

is legal and usable in the integer sense.

What about **int** variants such as **short** and **long**? Well, first one must note that a compiler need not treat them any differently from **int**s. GCC on Intel machines, for instance, stores a **long** in 4 bytes, just like an **int**. On the other hand, it stores a **short** in 2 bytes. It does offer a type **long long**, which it stores in 8 bytes, i.e. as a 64-bit quantity.

In the C language, the **sizeof** operator tells us how many bytes of storage the compiler will allocate for any variable type.

1.8.2.2 Complex Data Structures

As seen in an example above, array variables are generally implemented by compilers as contiguous blocks of memory. For example, the array declared as

```
int X[100];
```

would be allocated to some set of 100 consecutive words in memory.

What about two-dimensional arrays? For example, consider the four-row, six-column array

```
int G[4][6];
```

Again, this array will be implemented in a block of consecutive words of memory, more specifically 24 consecutive words, since these arrays here consist of $4 \times 6 = 24$ elements. But in what order will those 24 elements be arranged? Most compilers use either **row-major** or **column-major** ordering. To illustrate this, let us assume for the moment that this is a global array and that the compiler stores in non-reverse order, i.e. G[0][0] first and G[3][5] last.

In row-major order, all elements of the first row are stored at the beginning of the block, then all the elements of the second row are stored, then the third row, and so on. In column-major order, it is just the opposite: All of the first column is stored first, then all of the second column, and so on.

In the example above, consider G[1][4]. With row-major order, this element would be stored as the 11th word in the block of words allocated to G. If column-major order were used, the it would be stored in the 18th word. (The reader should verify both of these statements, to make sure to understand the concept.) C/C++ compilers use the row-major system.

Advanced data types are handled similarly, again in contiguous memory locations. For instance, consider the **struct** type in C, say

```
struct S {
   int X;
   char A,B;
};
```

would on a 16-bit machine be stored in four consecutive bytes, first two for X and then one each for A and B. On a 32-bit machine, it would be stored in six consecutive bytes, since X would take up a word and thus four bytes instead of two.

Note, though, that most compilers will store complex objects like this to be **aligned on word boundaries**. Consider the above **struct** S on a 32-bit machine, and suppose we have the local declaration

```
struct S A,B;
```

A and B collectively occupy 12 bytes, thus seemingly three words. But most compilers will now start an instance of S in the middle of a word. Thus although A and B will be stored contiguously in the sense that no other variables will be stored between them, they will be stored in four consecutive words, not three, with their addresses differing by 8, not 6. There will be two bytes of unused space between them.

In C++, an object of a **class** is stored in a manner similar to that of a **struct**.

1.8.2.3 Pointer Variables

We have not yet discussed how **pointer** variables are stored. *A pointer is an address.* In C, for example, suppose we have the declaration

```
int *P,X;
```

We are telling the compiler to allocate memory for two variables, one named P and the other named X. We are also telling the compiler what types these variables will have: X will be of integer type, while the ‘’ says

that P will be of pointer-to-integer type—meaning that P will store the address of some integer. For instance, in our program, we might have the statement

```
P = &X;
```

which would place the address of X in P. (More precisely, we should define this as the lowest-address byte of X.) If one then executed the statement

```
printf("%x\n",P);
```

the address of X would be printed.

As mentioned earlier, most machines today have equal word and address sizes. So, not only will the variable X here be stored in one word, but also the variable P would occupy a word too.

Note carefully that this would be true no matter what type P were to point to. For instance, consider the **struct** S we had above. The code

```
struct S *Q;
```

would *also* result in Q being stored in one word. An address is an address, no matter what kind of data item is stored beginning at that address..

Another aspect of C/C++ pointers which explicitly exposes the nature of storage of program objects in memory is **pointer arithmetic**. For example, suppose P is of pointer type. Then P+1, for example, points to the next consecutive object of the type to which P points.

The following example will bring together a number of different concepts we've looked at in the last few pages (be SURE to followup if you don't understand this well):

Consider the code

```
1  main()
2
3  {  float A,B,C,*Q;
4
5     Q = &B;
6     Q = Q-2;
```

Suppose Q is the memory location, say, 0x278. Then C will be the next word after Q, i.e. word 0x27c, then B will be word 0x280 and A will be word 0x284.

After the first statement,

```
Q = &B;
```

is executed, we will have c(0x278) = 0x280.

Now, what happens when the second statement,

```
Q = Q-2;
```

is executed? The expression Q-2 means "2 objects before Q in memory," where the word *object* means the type of variable pointed to by Q. So, Q-2 is the place 2 **float**s before where Q is pointing in memory. From Section 1.8.2.1 we know that this then means 2 words before where Q is pointing in memory. In other words, Q-2 is pointing to Q itself!

Keep in mind that in C/C++, arrays are equivalent to pointer references. For example, Y[200] means the same thing as Y+200.

1.8.3 Local Variables

Compilers assign global variables to fixed locations determined at compile-time.

On the other hand, variables which are local to a given function storage on the **stack**, which is a section of memory. Thus local variables are in memory too, just like globals. However, a variable's position on the stack won't be known until run-time. The compiler will merely specify the location of a variable relative to the top of the stack.

This is hard to understand given the background you have so far. You will understand it much better when we cover our unit on subroutines, but to give you a brief idea now, think of a recursive function **f()** which has a local variable X. As the function is called recursively multiple times, the stack will expand more and more, and there will be a separate space for X in each call. Thes multiple versions of X will occupy multiple memory locations, so you can see that &X cannot be predicted at compile-time.

1.8.4 Variable Names and Types Are Imaginary

When you compile a program from an HLL source file, the compiler first assigns memory locations for each declared variable.[22] For its own reference purposes, the compiler will set up a **symbol table** which shows which addresses the variables are assigned to. However, it is very important to note is that *the variable names in a C/C++ or other HLL program are just for the convenience of us humans.* In the machine language program produced from our HLL file by the compiler, all references to the variables are through their addresses in memory, with no references whatsoever to the original variables names.

[22] These locations will not actually be used at that time. They will only be used later, when the program is actually run.

For example, suppose we have the statement

```
X = Y + 4;
```

When the compiler produces some machine-language instructions to implement this action, the key point is that these instructions will *not* refer to X and Y. Say X and Y are stored in Words 504 and 1712, respectively. Then the machine instructions the compiler generates from the above statement will only refer to Words 504 and 1712, the locations which the compiler chose for the C/C++ variables X and Y.

For instance, the compiler might produce the following sequence of three machine language instructions:

```
copy Word 1712 to a cell in the CPU
add 4 to the contents of that cell
copy that cell to Word 504
```

There is no mention of X and Y at all! The names X and Y were just *temporary* entities, for communication between you and the compiler. The compiler chose locations for each variable, and then translated all of your C/C++ references to those variables to references to those memory locations, as you see here.

Again, when the compiler is done compiling your program, it will simply discard the symbol table.[23] If say on a UNIX system you type

```
gcc x.c
```

which creates the executable file **a.out**, that file will not contain the symbol table.

There is one exception to this, though, which occurs when you are debugging a program, using a debugging tool such as GDB.[24] Within the debugging tool you will want to be able to refer to variables by name, rather than by memory location, so in this case you do need to tell the compiler to retain the symbol table in **a.out**.[25] You do so with the **-g** option, i.e.

```
gcc -g x.c
```

[23] It may retain a list of function names and names of global variables. For simplicity, though, we'll simply say that the entire table is discarded.

[24] You should always use a debugging tool when you are debugging a program. *Don't debug by just inserting a lot of **printf()** calls!* Use of a debugging tool will save you large amounts of time and effort. Professional software engineers use debugging tools as a matter of course. There is a lot of information on debugging, and debugging tools, on my debug Web page, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`

[25] In fact, you'll need more than the symbol table. You certainly want the debugger to tell you what line of source code you are currently on, so we need to save the memory locations of machine code produced by each line of source code.

But even in this case, the actually machine-language portion (as opposed to the add-on section containing the symbol table) of the executable file **a.out** will not make any references to variable names.

The same comments apply to function names; they're gone too once we create **a.out**. Similarly, there is no such thing as a type of a variable at the machine level, as earlier.

1.8.5 Segmentation Faults and Bus Errors

Most machines today which are at the desktop/laptop level or higher (i.e. excluding handheld PDAs, chips inside cars and other machines, etc.) have **virtual memory** (VM) hardware. We will discuss this in another unit in detail, but the relevance here is that it can be used to detect situations in which a program tries to access a section of memory which was not allocated to it.

In a VM system, memory is (conceptually) broken into chunks called **pages**. When the operating system loads a program, say **a.out**, into memory, the OS also sets up a **page table**, which shows exactly which parts of memory the program is allowed to access. Then when the program is actually run, the hardware monitors each memory access, checking the page table. If the program accesses a part of memory which was not allocated to it, the hardware notices this, and switches control to the OS, which announces an error and kills in the program. This is called a **segmentation fault** in UNIX and a **general protection error** in VM Windows systems.

For example, consider the following program, which we saw in Section 1.4.5:

```
1  main()
2
3  {  int X[5],Y[20];
4
5     X[0] = 12;
6     Y[20] = 15;
7     printf("X[0] = %d\n",X[0]);  // prints out 15!
8  }
```

Say we compile this, say to **a.out**, and run it. What will happen?

As noted earlier, Y[20] is a perfectly legal expression here, which will turn out to be identical to X[0]. That's how X[0] becomes equal to 15.[26]

Even though that is clearly not what the programmer intended, the VM hardware will not detect this problem, and the program will execute without any announcement of error. The reason is that the program does not access memory that is not allocated to it.

On the other hand, if Y[20] above were Y[20000], then the location of "Y[20000]" would be far outside the

[26] C and C++ do not implement array **bounds checking**, i.e. the compiler will not produce code to check for this, which would slow down execution speed.

memory needed by the program, and thus definitely outside the memory allocated to the program.[27] The hardware would detect this, then jump to the OS, which would (in UNIX) shout out to us, "Segmentation fault."

So, remember:
A "seg fault" error message means that your program has tried to access a portion of memory which is outside the memory allocated to it. This is due to a buggy array index or pointer variable.

Note that the case of bad pointer variables includes problems like the one in the classical error made by beginning C programmers:

```
1  scanf("\%d",X);
```

where X is, say, of type **int**; the programmer here has forgotten the ampersand.

The first thing you should do when a seg fault occurs is to find out where in the program it occurred. **Use a debugging tool, say DDD or GDB, to do this!**

Note again that seg faults require that our machine have VM hardware, and that the OS is set up to make use of that hardware. Without both of these, an error like "Y[20000]" will NOT be detected.

Many CPU types require that word accesses be **aligned**. On Intel machines, for example, words must begin at addresses which are multiples of 4. Word 200, for instance, consists of Bytes 200, 201, 202 and 203, and Word 204 consists of Bytes 204-207, but there is no such thing as "Word 201," consisting of Bytes 201-204. Suppose your program has code like

```
1  int X,*P;
2  ...
3  P = (int *) 201;
4  X = *P;
```

The compiler will translate the line

```
X = *P;
```

into some word-accessing machine instruction. When that instruction is executed, the CPU hardware will see that P's value is not a multiple of 4.[28] The CPU will then cause a jump to the operating system, which will (for example in UNIX) announce "Bus error."

As is the case with seg faults, you should use a debugging tool to determine where a bus error has occurred.

[27] Since the memory is allocated in units of pages, the actual amount of memory allocated will be somewhat larger than the program needs; we cannot allocate only a partial page.

[28] Note that this does not require VM capability.

1.9 ASCII Table

This is the output of typing

```
% man ascii
```

on a UNIX machine.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	110	72	48	H
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M
016	14	0E	SO	116	78	4E	N
017	15	0F	SI	117	79	4F	O
020	16	10	DLE	120	80	50	P
021	17	11	DC1	121	81	51	Q
022	18	12	DC2	122	82	52	R
023	19	13	DC3	123	83	53	S
024	20	14	DC4	124	84	54	T
025	21	15	NAK	125	85	55	U
026	22	16	SYN	126	86	56	V
027	23	17	ETB	127	87	57	W
030	24	18	CAN	130	88	58	X
031	25	19	EM	131	89	59	Y
032	26	1A	SUB	132	90	5A	Z
033	27	1B	ESC	133	91	5B	[
034	28	1C	FS	134	92	5C	\ '\\'
035	29	1D	GS	135	93	5D	]
036	30	1E	RS	136	94	5E	^
037	31	1F	US	137	95	5F	
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j

053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

1.10 An Example of How One Can Exploit Big-Endian Machines for Fast Character String Sorting

The endian-ness of a machine can certainly make a difference. As an example, suppose we are writing a program to sort character strings in alphabetical order. The word "card" should come after "car" but before "den", for instance. Suppose our machine has 32-bit word size, variables X and Y are of type **int**, and we have the code

```
strncat(&X,"card",4);
strncat(&Y,"den ",4);
```

Say X and Y are in Words 240 and 804, respectively. This means that Byte 240 contains 0x63, the ASCII code for 'c', Byte 241 contains 0x61, the code for 'a', byte 242 contains 0x72, the code for 'r' and byte 243 contains 0x64. Similarly, bytes 804-807 will contain 0x64, 0x65, 0x6e and 0x20. All of this will be true regardless of whether the machine is big- or little-endian.

But on a big-endian machine, we can actually use word subtraction to determine which of the strings "card" and "den" should precede the other alphabetically, by subtracting Word 240 from Word 804. To see that this works, note that the contents of the two words will be 0x63617264 (1667330660 decimal) and 0x64656e20 (1684368928 decimal), so the result of the subtraction will be negative, and thus we will decide that "card" is less than (i.e. alphabetically precedes) "den"—exactly what we want to happen.

The reader should pause to consider the speed advantage of such a comparison. If we did not use word subtraction, we would have to do a character-by-character comparison, that is four subtractions. (Note that

these are word-based subtractions, even though we are only working with single bytes.) The arithmetic hardware works on a word basis.) Suppose for example that we are dealing with 12-character strings. We can base our sort program on comparing (up to) three pairs of words if we use word subtraction, and thus gain roughly a fourfold speed increase over a more straightforward sort which compares up to 12 pairs of characters.

We could do the same thing on a little-endian machine if we were to store character strings backwards. However, this may make programming inconvenient.

1.11 How to Inspect the Bits of a Floating-Point Variable

Suppose we wish to see the individual bits in a **float** variable**y**. The code

```
printf("%x\n",y);
```

will not work, as it would tell **printf()** to consider the bits in **y** to represent an integer, and would then print that integer in base-16. Why would this be a problem? Recall that **printf()** treats any quantity printed using **%x** format as an integer, and thus the endian-ness will play a role. That would mean the printout would be "backwards" on a little-endian machine.

Instead, we must write something like this:

```
1  float y;
2  char *p;  // char type focuses on individual bytes
3  ...
4  ...
5  p = (char *) &y;
6  for (i = 0; i < 4; i++)  {
7     printf("%y\n",*p);
8     p++;
9  }
10 printf("%f  %x\n",y, y);
```

Actually, even this would have a bit of a problem. Suppose for example that the first byte to be printed is 0x92, which is 10010010 in bit form. Viewed as an 8-bit integer, that is a negative number. As a full 32-bit integer, that number is 11111111111111111111111110010010, i.e 0xffffff92. The latter, not 0x92, would be printed out. We would then just ignore those leading f digits, but it would be annoying. To really make it look nice, we'd have to add some more code.

Chapter 2

Major Components of Computer "Engines"

2.1 Introduction

Recall from Chapter 0 that we discussed the metaphor of the "engine" of a computer. This engine has two main components:

- the hardware, including the central processing unit (CPU) which executes the computer's machine language, and
- the low-level software, consisting of various services that the operating system makes available to programs.

This chapter will present a broad overview of both components of this engine. The details will then unfold in the succeeding chapters.

One of the major goals of this chapter, and a frequent theme in the following chapters, is to develop an understanding of the functions of these two components. In particular, questions which will be addressed both in this chapter and in the chapters which follow concern the various functions of computer systems:

- What functions are typically implemented in hardware?
- What functions are typically implemented in software?
- For which functions are both hardware and software implementations common? Why is hardware implementation generally faster, but software implementation more flexible?

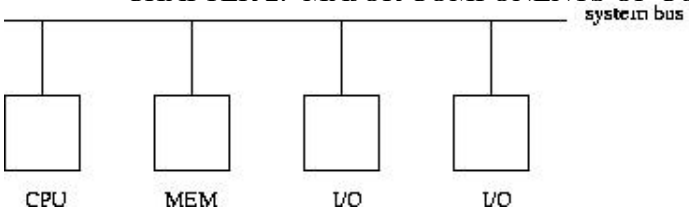


Figure 2.1: system structure

Related to these questions are the following points, concerning the **portability** of a program, i.e. the ability to move a program developed under one computing environment to another. What dependencies, if any, does the program have to that original environment? Potential dependencies are:

- Hardware dependencies:

  A program written in machine language for an older PC's Intel 80386 CPU[1] certainly won't run on Motorola's PowerPC CPU.[2] But that same program *will* run on PC using the Intel Pentium, since the Intel family is designed to be **upward compatible**, meaning that programs written for earlier members of the Intel CPU family are guaranteed to run on later members of the family (though not vice versa).

- Operating system (OS) dependencies:

  A program written to work on a PC under the Windows OS will probably not run on the same machine under the Linux OS (and vice versa), since the program will probably call OS functions.

And finally, we will discuss one of the most important questions of all: What aspects of the hardware and software components of the engine determine the overall speed at which the system will run a given program?

2.2 Major Hardware Components of the Engine

2.2.1 System Components

A block diagram of the basic setup of a typical computer system appears in Figure 2.1.

The major components are as follows:

CPU

[1] Or clones of Intel chips, such as those by AMD. Throughout this section, our term *Intel* will include the clones.

[2] This is what the Apple Macintosh used until 2005.

As mentioned earlier, this is the **central processing unit**, often called simply the **processor**, where the actual execution of a program takes place. (Since only machine language programs can execute on a computer, the word *program* will usually mean a machine language program. Recall that we might write such a program directly, or it might be produced indirectly, as the result of compiling a source program written in a high-level language (HLL) such as C.)

Memory

A program's data and machine instructions are stored here during the time the program is executing. Memory consists of cells called **words**, each of which is identifiable by its **address**.

If the CPU fetches the contents of some word of memory, we say that the CPU **reads** that word. On the other hand, if the CPU stores a value into some word of memory, we say that it **writes** to that word. Reading is analogous to watching a video cassette tape, while writing is analogous to recording onto the tape.

Ordinary memory is called **RAM**, for Random Access Memory, a term which means that the access time is the same for each word.[3] There is also **ROM** (Read-Only Memory), which as its name implies, can be read but not written. ROM is used for programs which need to be stored permanently in main memory, staying there even after the power is turned off. For example, an autofocus camera typically has a computer in it, which runs only one program, a program to control the operation of the camera. Think of how inconvenient—to say the least—it would be if this program had to be loaded from a disk drive everytime you took a picture! It is much better to keep the program in ROM.[4]

I/O Devices

A typical computer system will have several **input/output** devices, possibly even hundreds of them (Figure 2.1 shows two of them). Typical examples are keyboards/monitor screens, floppy and fixed disks, CD-ROMs, modems, printers, mice and so on.

Specialized applications may have their own special I/O devices. For example, consider a vending machine, say for tickets for a regional railway system such as the San Franciso Bay Area's BART, which is capable of accepting dollar bills. The machine is likely to be controlled by a small computer. One of its input devices might be an optical sensor which senses the presence of a bill, and collects data which will be used to analyze whether the bill is genuine. One of the system's output devices will control a motor which is used to pull in the bill; a similar device will control a motor to dispense the railway ticket. Yet another output device will be a screen to give messages to the person buying the ticket, such as "please deposit 25 cents more."

The common feature of all of these examples is that they serve as interfaces between the computer and the "outside world." Note that in all cases, they are communicating with a *program* which is running on

[3] In the market for personal computers, somehow the word "memory" has evolved to mean disk space, quite different from the usage here. In that realm, what we refer to here as "memory" is called "RAM." The terms used in this book are the original ones, and are standard in the computer industry, for instance among programmers and computer hardware designers. However, the reader should keep in mind that the terms are used differently in some other contexts.

[4] Do not confuse ROM with CD-ROMs, in spite of the similar names. A ROM is a series of words with addresses within the computer's memory space, just like RAM, whereas a CD-ROM is an input/output device, like a keyboard or disk.

the computer. Just as you have in the past written programs which input from a keyboard and output to a monitor screen, programs also need to be written in specialized applications to do input/output from special I/O devices, such as the railway ticket machine application above. For example, the optical sensor would collect data about the bill, which would be input by the program. The program would then analyze this data to verify that the bill is genuine.

Note carefully the difference between memory and disk.[5] Memory is purely electronic, while disk is electromechanical, the latter referring to the fact that a disk is a rotating platter. Both memory and disk store bits, but because of the mechanical nature of disk, it is very much slower than memory—memory access is measured in nanoseconds (billionths of seconds) while disk access is measured in milliseconds (thousandths of seconds). The good thing about disk is that it is nonvolatile, so we can store files permanently.[6] Also, disk storage is a lot cheaper, per bit stored, than memory.

System Bus

A **bus** is a set of parallel wires (usually referred to as "lines"), used as communication between components. Our **system bus** plays this role in Figure 2.1— he CPU communicates with memory and I/O devices via the bus. It is also possible for I/O devices to communicate directly with memory, an action which is called **direct memory access** (DMA), and again this is done through the bus.[7]

The bus is broken down into three sub-buses:

- **Data Bus:**

  As its name implies, this is used for sending data. When the CPU reads a memory word, the memory sends the contents of that word along the data bus to the CPU; when the CPU writes a value to a memory word, the value flows along the data bus in the opposite direction.

  Since the word is the basic unit of memory, a data bus usually has as many lines as there are bits in a memory word. For instance, a machine with 32-bit word size would have a data bus consisting of 32 lines.

- **Address Bus:**

  When the CPU wants to read or write a certain word of memory, it needs to have some mechanism with which to tell memory *which* word it wants to read or write. This is the role of the address bus. For example, if the CPU wants to read Word 504 of memory, it will put the value 504 on the address bus, along which it will flow to the memory, thus informing memory that Word 504 is the word the CPU wants.

[5] Often confused by the fact that salespeople at computer stores erroneously call disk "memory."

[6] We couldn't do that with ROM, since we want to be able to modify the files.

[7] This is the basic view, but technically it applies more to older or smaller computers. In a PC, for instance, there will be extra chips which serve as "agents" for the CPU's requests to memory. For a description of how some common chips like this work, and their implications for software execution speed, see Chapter 5 of *Code Optimization: Effective Memory Usage*, by Kris Kaspersky, A-LIST, 2003.

The address bus usually has the same number of lines as there are bits in the computer's addresses.

- **Control Bus:**

  How will the memory know whether the CPU wants to read or write? This is one of the functions of the control bus. For example, the control bus in typical PCs includes lines named MEMR and MEMW, for "memory read" and "memory write." If the CPU wants to read memory, it will **assert** the MEMR line, by putting a low voltage on it, while for a write, it will assert MEMW. Again, this signal will be noticed by the memory, since it too is connected to the control bus, and so it can act accordingly.

As an example, consider a machine with both address and word size equal to 32 bits. Let us denote the 32 lines in the address bus as A_{31} through A_0, corresponding to Bits 31 through 0 of the 32-bit address, and denote the 32 lines in the data bus by D_{31} through D_0, corresponding to Bits 31 through 0 of the word being accessed. Suppose the CPU executes an instruction to fetch the contents of Word 0x000d0126 of memory. This will involve the CPU putting the value 0x000d0126 onto the address bus. Remember, this is hex notation, which is just a shorthand abbreviation for the actual value,

```
00000000000000001101000000100100110
```

So, the CPU will put 0s on lines A_{31} through A_{20}, a 1 on Line A_{19}, a 1 on Line A_{18}, a 0 on Line A_{17}, and so on. At the same time, it will assert the MEMR line in the control bus. The memory, which is attached to these bus lines, will sense these values, and "understand" that we wish to read Word 0x000d0126. Thus the memory will send back the contents of that word on the data bus. If for instance c(0x000d0126) = 0003, then the memory will put 0s on Lines D_{31} through D_2, and 1s on Lines D_1 and D_0, all of which will be sensed by the CPU.

Some computers have several buses, thus enabling more than one bus transaction at a time, improving performance.

2.2.2 CPU Components

2.2.2.1 Intel/Generic Components

We will now look in more detail at the components in a CPU. Figure 2.2 shows the components that make up a typical CPU. Included are an **arithmetic and logic unit** (ALU), and various **registers**.

The ALU, as its name implies, does arithmetic operations, such as addition, subtraction, multiplication and division, and also several **logical** operations. The latter category of operations are similar to the `&&`, `||` and ! operators in the C language) used in logical expressions such as

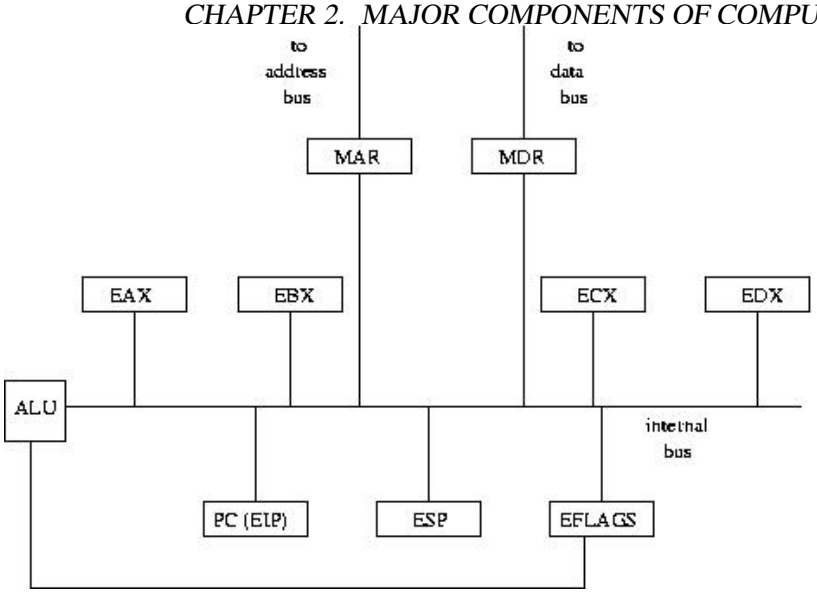


Figure 2.2: CPU internals

```
if (a < b && c == 3) x = y;
```

The ALU does not *store* anything. Values are input to the ALU, and results are then output from it, but it does not store anything, in contrast to memory words, which do store values. An analogy might be made to telephone equipment. A telephone inputs sound, in the form of mechanical vibrations in the air, and converts the sounds to electrical pulses to be sent to the listener's phone, but it does not store these sounds. A telephone tape-recording answering machine, on the other hand, does store the sounds which are input to it.

Registers are storage cells similar in function to memory words. The number of bits in a register is typically the same as that for memory words. We will even use the same c( ) notation for the contents of a register as we have used for the contents of a memory word. For example, c(PC) will denote the contents of the register PC described below, just as, for instance, c(0x22c4) means the contents of memory word 0x22c4. Keep in mind, though, that registers are not in memory; they are inside the CPU. Here are some details concerning the registers shown in Figure 2.2.

- **PC:** This is the **program counter**. Recall that a program's machine instructions must be stored in memory while the program is executing. The PC contains the address of the currently executing instruction. The term *PC* is a generic term; on Intel, the register is called EIP (Extended Instruction Pointer).

- **ESP:** The **stack pointer** contains the address of the "top" of a certain memory region which is called the **stack**. A stack is a type of data structure which the machine uses to keep track of function calls and other information, as we will see in Chapter 5.
- **EAX, EBX, etc.:** These are **data registers**, used for temporary storage of data.
- **EFLAGS:** The **processor status register** (PS), called EFLAGS on Intel machines, contains miscellaneous pieces of information, including the **condition codes**. The latter are indicators of information such as whether the most recent computation produced a negative, positive or zero result. Note that there are wires leading out of the ALU to the EFLAGS (shown as just one line in Figure 2.2). These lines keep the condition codes up to date. Each time the ALU is used, the condition codes are immediately updated according to the results of the ALU operation.

  Generally the PS will contain other information in addition to condition codes. For example, it was mentioned in MIPS and PowerPC processors give the operating system a choice as to whether the machine will run in big-endian or little-endian mode. A bit in the PS will record which mode is used.
- **MAR:** The **memory address register** is used as the CPU's connection to the address bus. For example, if the currently executing instruction needs to read Word 0x0054 from memory, the CPU will put 0x0054 into MAR, from which it will flow onto the address bus.
- **MDR:** The **memory data register** is used as the CPU's connection to the data bus. For example, if the currently executing instruction needs to read Word 0x0054 from memory, the memory will put c(0x0054) onto the data bus, from which it will flow into the MDR in the CPU. On the other hand, if we are writing to Word 0x0054, say writing the value 0x0019, the CPU will put 0x0019 in the MDR, from which it will flow out onto the data bus and then to memory. At the same time, we will put 0x0054 into the MAR, so that the memory will know to which word the 0x0019 is to be written.
- **IR:** This is the **instruction register**. When the CPU is ready to start execution of a new instruction, it fetches the instruction from memory. The instruction is returned along the data bus, and thus is deposited in the MDR. The CPU needs to use the MDR for further accesses to memory, so it enables this by copying the fetched instruction into the IR, so that the original copy in the MDR may be overwritten.

A note on the sizes of the various registers: The PC, ESP, and MAR all contain addresses, and thus typically have sizes equal to the address size of the machine. Similarly, the MDR typically has sizes equal to the word size of the machine. The PS stores miscellaneous information, and thus its size has no particular relation to the machine's address or word size. The IR must be large enough to store the longest possible instruction for that machine.

A CPU also has internal buses, similar in function to the system bus, which serve as pathways with which transfers of data from one register to another can be made. Figure 2.2 shows a CPU having only one such bus, but some CPUs have two or more. Internal buses are beyond the scope of this book, and thus any reference to a "bus" from this point onward will mean the system bus.

The reader should pay particular attention to the MAR and MDR. They will be referred to at a number of points in the following chapters, both in text and in the exercises—not because they are so vital in their own right, but rather because they serve as excellent vehicles for clarifying various concepts that we will cover in this book. In particular, phrasing some discussions in terms of the MAR and MDR will clarify the fact that some CPU instructions access memory while others do not.

Again, the CPU structure shown above should only be considered "typical," and there are many variations. RISC CPUs, not surprisingly, tend to be somewhat simpler than the above model, though still similar.

2.2.2.2 History of Intel CPU Structure

The earliest widely-used Intel processor chip was the 8080. Its word size was 8 bits, and it included registers named A, B, C and D (and a couple of others). Address size was 16 bits.

The next series of Intel chips, the 8086/8088,[8] and then the 80286, featured 16-bit words and 20-bit addresses. The A, B, C and D registers were accordingly extended to 16-bit size and renamed AX, BX, CX and DX ('X' stood for "extended"). Other miscellaneous registers were added. The lower byte of AX was called AL, the higher byte AH, and similarly for BL, BH, etc.

Beginning with the 80386 and extending to the Pentium series, both word and address size were 32 bits. The registers were again extended in size, to 32 bits, and renamed EAX, EBX and so on ('E' for "extended"). As of December 2006, the 64-bit generation is now becoming commonplace.

The pre-32-bit Intel CPUs, starting with 8086/8088, replaced the *single* register PC with a *pair* of registers, CS (for *code segment*) and IP (for *instruction pointer*). A rough description is that the CS register pointed to the **code segment**, which is the place in memory where the program's instructions start, and the IP register then specified the *distance* in bytes from that starting point to the current instruction. Thus by combining the information given in c(CS) and c(IP), we obtained the absolute address of the current instruction.

This is still true today when an Intel CPU runs in 16-bit mode, in which case it generates 20-bit addresses. The CS register is only 16 bits, but it represents a 20-bit address whose least significant four bits are implicitly 0s. (This implies that code segments are allowed to begin only at addresses which are multiples of 16.) The CPU generates the address of the current instruction by concatenating c(CS) with four 0 bits[9] and then adding the result to c(IP).

Suppose for example the current instruction is located at 0x21082, and the code segment begins at 0x21040. Then c(CS) and c(IP) will be 0x2104 and 0x0042, respectively, and when the instruction is to be executed, its address will be generated by combining these two values as shown above.

The situation is similar for stacks and data. For example, instead of having a single SP register as in our model of a typical CPU above, the earlier Intel CPUs (and current CPUs when they are running in 16-bit

[8] The first IBM PC used the 8088.

[9] This is one 0 hex digit. Note too that this concatenation is equivalent to multiplying by 16.

mode) use a pair of registers, SS and SP. SS specifies the start of the **stack segment**, and SP contains the distance from there to the current top-of-stack. For data, the DS register points to the start of the **data segment**, and a 16-bit value contained in the instruction specifies the distance from there to the desired data item.

Since IP, SP and the data-item distance specified within an instruction are all 16-bit quantities, it follows that the code, stack and data segments are limited to $2^{16} = 65,536$ bytes in size. This can make things quite inconvenient for the programmer. If, for instance, we have an array of length, say, 100,000 bytes, we could not fit the array into one data segment. We would need two such segments, and the programmer would have to include in the program lines of code which change the value of c(DS) whenever it needs to access a part of the array in the other data segment.

These problems are avoided by the newer operating systems which run on Intel machines today, such as Windows and Linux, since they run in 32-bit mode. Addresses are also of size 32 bits in that mode, and IP, SP and data-item distance are 32 bits as well. Thus a code segment, for instance, can fill all of memory, and segment switching as illustrated above is unnecessary.

This continues to be the case for 64-bit machines, e.g. with 64-bit addresses.

2.2.3 The CPU Fetch/Execute Cycle

After its power is turned on, the typical CPU pictured in Figure 2.2 will enter its **fetch/execute cycle**, repeatedly cycling through these three steps, i.e. Step A, Step B, Step C, Step A, Step B, and so on:

- **Step A:** The CPU will perform a memory read operation, to fetch the current instruction. This involves copying the contents of the PC to the MAR, asserting a memory-read line in the control bus, and then waiting for the memory to send back the requested instruction along the data bus to the MDR. While waiting, the CPU will update the PC, to point to the following instruction in memory, in preparation for Step A in the next cycle.
- **Step B:** The CPU will copy the contents of MDR to IR, so as to free the MDR for further memory accesses. The CPU will inspect the instruction in the IR, and **decode** it, i.e. decide what kind of operation this instruction performs—addition, subtraction, jump, and so on.
- **Step C:** Here the actual execution of the operation specified by the instruction will be carried out. If any of the operands required by the instruction are in memory, they will be fetched at this time, by putting their addresses into MAR and waiting for them to arrive at MDR. Also, if this instruction stores its result back to memory, this will be accomplished by putting the result in MDR, and putting into MAR the memory address of the word we wish to write to.

After Step C is done, the next cycle is started, i.e. another Step A, then another Step B, and so on. Again, keep in mind that the CPU will continually cycle through these three steps as long as the power remains

on. Note that all of the actions in these steps are functions of the *hardware*; the circuitry is designed to take these actions, while the programmer merely takes advantage of that circuitry, by choosing the proper machine instructions for his/her program.

A more detailed description of CPU operations would involve specifying its **microsteps**. These are beyond the scope of this book, and the three-step cycle described above will give sufficient detail for our purposes, though we *will* use them later to illustrate the term **clock speed**.

2.3 Software Components of the Computer "Engine"

There are many aspects of a computer system which people who are at the learning stage typically take for granted as being controlled by hardware, but which are actually controlled by software. An example of this is the backspace action when you type the backspace key on the keyboard. You are accustomed to seeing the last character you typed now disapppear from the screen, and the cursor moving one position to the left. You might have had the impression that this is an inherent property of the keyboard and the screen, i.e. that their circuitry was designed to do this. However, for most computer systems today this is not the case. The bare hardware will not take any special action when you hit the backspace key. Instead, the special action is taken by whichever **operating system** (OS) is being used on the computer.

The OS is software—a *program*, which a person or group of people wrote to provide various services to user programs. One of those services is to monitor keystrokes for the backspace key, and to take special actions (move the cursor leftward one position, and put a blank where it used to be) when encountering that key. When you write a program, say in C, you do not have to do this monitoring yourself, which is a tremendous convenience. Imagine what a nuisance it would be if you were forced to handle the backspace key yourself: You would have to include some statements in each program you write to check for the backspace key, and to update the screen if this character is encountered. The OS relieves you of this burden.

But if you want this "burden"—say you are writing a game or a text editor and need to have characters read as they are typed, instead of waiting for the user to hit the Enter key—then the OS also will turn off the backspace, echo etc. if you request it. In UNIX, this is done via the ioctl() system call.

This backspace-processing is an example of one of the many services that an OS provides. Another example is maintenance of a file system. Again the theme is convenience. When you create a file, you do not have to burden yourself with knowing the physical location of your file on the disk. You merely give the file a name. The OS finds unused space on the disk to store your file, and enters the name and physical location in a table that the OS maintains. Subsequently, you may access the file merely by specifying the name, and the OS service will translate that into the physical location and access the file on your behalf. In fact, a typical OS will offer a large variety of services for accessing files.

So, a user program will make use of many OS services, usually by calling them as functions. For example, consider the C-language function **scanf()**. Though of course you did not write this function yourself, some-

one did, and and in doing so that person (or group of people) relied heavily on calls to an OS subroutine, **read()**. In terms of our "look under the hood" theme, we might phrase this by saying that a look under the hood of the C **scanf()** source code would reveal system calls to the OS function **read()**. For this reason, the OS is often referred to as "low-level" software. Also, this reliance of user programs on OS services shows why the OS is included in our "computer engine" metaphor—the OS is indeed one of the sources of "power" for user programs, just as the hardware is the other source of power.

To underscore that the OS services do form a vital part of the computer's "engine," consider the following example. Suppose we have a machine-language program—which we either wrote ourselves or produced by compiling from C—for a DEC computer with a MIPS CPU. Could that program be run without modification on a Silicon Graphics machine, which also uses the MIPS chip? The answer is no. Even though both machines do run a Unix OS, there are many different "flavors" of Unix. The DEC version of Unix, called Ultrix, differs somewhat from the SGI version, called IRIX. The program in question would probably include a number of calls to OS services—recall from above that even reads from the keyboard and writes to the screen are implemented as OS services—and those services would be different under the two OSs.

Thus even though individual instructions of the program written for the DEC would make sense on the SGI machine, since both machines would use the same type of CPU, some of those instructions would be devoted to OS calls, which would differ.

Since an OS consists of a program, written to provide a group of services, it follows that several different OSs—i.e. several different programs which offer different groups of services—could be run on the same hardware. For instance, this is the case for PCs. The most widely used OS for these CPUs is Microsoft Windows, but there are also several versions of Unix for PCs, notably the free, public-domain Linux and the commercial SCO.

2.4 Speed of a Computer "Engine"

What factors determine the speed capability of a computer engine? This is an extremely complex question which is still a subject of hot debate in both academia and the computer industry. However, a number of factors are clear, and will be introduced here. The presentation below will just consist of overviews, and the interested reader should pursue further details in books on computer architecture and design. However, it is important that the reader get some understanding of the main issues now; at the very least enough to be able to understand newspaper PC ads! The discussion below is aimed at that goal.

2.4.1 CPU Architecture

Different CPU types have different instruction and register sets. The Intel family, for example, has a very nice set of character-string manipulation instructions, so it does such operations especially quickly. This is counterbalanced somewhat by the fact that CPUs in this family have fewer registers than do CPUs in some

other families, such as those in most of the newer machines.[10] Since registers serve as local—and thus fast—memory, the more of them we have, the better, and thus the Intel family might be at a disadvantage in this respect.

2.4.2 Parallel Operations

One way to get more computing done per unit time is do several things at one time, i.e. in parallel. Most modern machines, even inexpensive ones such as personal computers, include some forms of parallelism.

For example, most CPUs perform **instruction prefetch**: During execution of the current instruction, the CPU will attempt to fetch one or more of the instructions which follow it sequentially—i.e. at increasing addresses—in memory.[11] For example, consider an Intel chip running in 16-bit mode. In this mode the chip has 20-bit addresses. Consider a two-byte instruction at location 0x21082. During the time we are executing that instruction the CPU might attempt to start fetching the next instruction, at 0x21084. The success or failure of this attempt will depend on the duration of the instruction at 0x21082. If this attempt is successful, then Step A can be skipped in the next cycle, i.e. the instruction at 0x21084 can be executed earlier.

Of course, the last statement holds only if we actually do end up executing the instruction at 0x21084. If the instruction at 0x21082 turns out to be a jump instruction, which moves us to some other place in memory, we will not execute the instruction at 0x21084 at all. In this case, the prefetching of this instruction during the execution of the one at 0x21082 would turn out to be wasted. Modern CPUs have elaborate jump-prediction mechanisms to try to deal with this problem.

Intel CPUs will often fetch *several* downstream instructions, while for RISC CPUs we might be able to fetch only one. The reason for this is that RISC instructions, being so simple, have such short duration that there just is not enough time to fetch more than one instruction.

Instruction prefetching is a special case of a more general form of parallelism, called **pipelining**. This concept treats the actions of an instruction as being like those of an assembly line in a factory. Consider an automobile factory, for instance. Its operation is highly parallel, which the construction of many cars being done simultaneously. At any given time, one car, at any early stage in the assembly line, might be having its engine installed, while another car, at a later stage in the line, is having its transmission installed. Pipelined CPUs (which includes virtually every modern CPU) break instruction execution down into several stages, and have several instructions executing at the same time, in different stages. In the simple case of instruction prefetch, there would be only two stages, one for the fetch (Step A) and the other for execution (Steps B and C).

Most recently-designed CPUs are **superscalar**, meaning that a CPU will have several ALUs. This is another

[10] This does not include the newer Intel chips, such as the Pentium, because these chips had to be designed for compatibility with the older ones.

[11] The prefetch may insted be from the **cache**, which we will discuss later.

way in which we can get more than one instruction executing at a time.

Yet another way to do this is to have multiple CPUs! In **multiprocessor** systems with n CPUs, we can execute n instructions at once, instead of one, thus potentially improving system speed by a factor of n. Typical speedup factors in real applications are usually much less than n, due to such overhead as the need for the several CPUs to coordinate actions with each other, and not get in each other's way as they access memory.

The classical way of constructing multiprocessor systems was to connect several CPUs to the system bus. However, as of 2006, it is common for even many low-end CPU chips to be **dual core**, meaning that the chip actually contains *two* CPUs. This has brought multiprocessor computing into the home.

It used to be very expensive to own a multiprocessor machine. The CPUs would all be connected to the bus, which increases manufacturing costs, etc. But by having more than once CPU on a chip, the expense is small.

Note that chip manufacturers have found that it makes better economic sense now for them to use chip space for extra processors, rather than continuing to increase processor speed.

2.4.3 Clock Rate

Recall that each machine instruction is implemented as a series of **microsteps**. Each microstep has the same duration, namely one **clock cycle**, which is typically set as the time needed to transfer a value from one CPU register to another. The CPU is paced by a **CPU clock**, a crystal oscillator; each pulse of the clock triggers one microstep.

In 2002, clock rates of over 1 **gigahertz**, i.e. 1 billion cycles per second, became common in PCs. This is quite a contrast to the 4.77 megahertz clock speed of the first IBM PC, introduced in 1981.

Each instruction takes a certain number of clock cycles. For example, an addition instruction with both operands in CPU registers might take 2 clock cycles on a given CISC CPU, while a multiply instruction with these operands takes 21 clock cyles on the same CPU. If one of the operands is in memory, the time needed will increase beyond these values.

RISC machines, due to the fact that they perform only simple operations (and usually involving only registers), tend to have instructions which operate in a single clock cylce.

The time to execute a given instruction will be highly affected by clock rate, even among CPUs of the same type. For example, as mentioned above, a register-to-register addition instruction might typically take 2 microsteps on a CISC CPU. Suppose the clock rate is 1 gigahertz, so that a clock cycle takes 10^{-9} seconds, i.e. 1 **nanosecond**.[12] Then the instruction would take 2 nanoseconds to execute.

[12] A nanosecond is a billionth of a second.

Within a CPU family, say the Intel family, the later members of the family typically run a given program much faster than the earlier members, for two reasons related to clock cycles:

- Due to advances in fabrication of electronic circuitry, later members of a CPU family tend to have much faster clock rates than do the earlier ones.
- Due to more clever algorithms, pipelining, and so on, the later members of the family often can accomplish the same operation in fewer microsteps than can the earlier ones.

2.4.4 Memory Caches

2.4.4.1 Need for Caching

In recent years CPU speeds have been increasing at very impressive rates, but memory access speeds have not kept pace with these increases. Remember, memory is *outside* the CPU, not in it. This causes slow access, for various reasons.

- The physical distance from the CPU is on the order of inches, whereas it is typically less than an inch within the CPU. Even though signals travel at roughly 2/3 the speed of light, there are so many signals sent per second that even this distance is a factor causing slowdown.
- An electrical signal propagates more slowly when it leaves a CPU chip and goes onto the bus toward memory. This is due to the bus wires being far thicker than the very fine wires within the CPU.
- Control buses typically must include **handshaking** lines, in which the various components attached to the bus assert to coordinate their actions. One component says to the other, "Are you ready," and that component cannot proceed until it gets a response from the other. This causes delay.

2.4.4.2 Basic Idea of a Cache

To solve these problems, a storage area, either within the CPU or near it or both, is designed to act as a **memory cache**. The cache functions as a temporary storage area for a *copy* of some small subset of memory, which can be accessed locally, thus avoiding a long trip to memory for the desired byte or word.[13]

Say for example the program currently running on the CPU had in its source file the line

```
char x;
```

[13] Modern computers typically have two caches, an **L1** cache inside the CPU, and an **L2** cache just outside it. When the CPU wants to access a byte or word, it looks in L1 first; if that fails, it looks in L2; and if that fails, it goes to memory. Here, though, we will assume only an L1 cache.

and suppose that **x** is stored in byte 1205 of memory, i.e. **&x** is 1205. Consider what occurs in each instruction in the program that reads **x**. Without a cache, the CPU would put 1205 on the address bus, and assert the MEMR line in the control bus. The memory would see these things on the bus, and copy whatever is in location 1205 to the data bus, from which the CPU would receive it.

If we did have a cache, the CPU would look first to see if the cache contains a copy of location 1205. If so, the CPU does not have to access memory, a huge time savings. This is called a cache **hit**. If the CPU does not find that the cache contains a copy of location 1205—a cache **miss**—then the CPU must go to memory in the normal manner.

2.4.4.3 Blocks and Lines

Memory is partitioned into **blocks**, of fixed size. For concreteness, we will assume here that the block size is 512 bytes. Then Bytes 00000-000511 form Block 0, Bytes 00512-001023 form Block 1, and so on. A memory byte's block number is therefore its address divided by the block size. Equivalently, since $log_2 512 = 9$, if our address size is 32 bits, then the location's block number is given in the most-significant 23 bits of the address, i.e. all the bits of the address except for the lower 9. The lower 9 bits then give location of the byte within the block.[14] Note that the partitioning is just conceptual; there is no "fence" or any other physical boundary separating one block from another.

This organization by blocks comes into play in the following manner. Consider our example above in which the CPU wanted to read location 1205. The block number is $\lfloor 1205/512 \rfloor = 2$. The CPU would not really search the cache for a copy of byte 1205 or even word 1205. Instead, the CPU would search for a copy of the entire block containing location 1205, i.e. a copy of block 2.

The cache is partitioned into a fixed number of slots, called **lines**. Each slot has room for a copy of one block of memory, plus an extra word in which additional information is stored. This information includes a **tag**, which states which block has a copy in memory right now. This is vital, of course, as otherwise when the CPU looked in the cache, it wouldn't know whether this line contains a copy of the desired block, as opposed to a copy of some other block.

At any given time, the cache contains copies of some blocks from memory, with different blocks being involved at different times. For example, suppose again the CPU needs to read location 1205. It then searches the cache for a copy of block 2. In each line in which it searches, the CPU would check the tag for that line. If the tag says "This line contains a copy of block 2," then the CPU would see that this is a hit. The CPU would then get byte 1205 from this line as follows. As we saw above, byte 1205 is in block $\lfloor 1205/512 \rfloor = 2$, and using the same reasoning,[15] *within* that block, this byte is byte number 1205 mod 512

[14] Suppose we have 10 classrooms, each with 10 kids, sitting in 10 numbered seats, according to the kids' ID numbers. Kids 0-9 are in classroom 0, 10-19 are in classroom 1, and so on. So for instance kid number 28 will be in classroom 2, sitting in seat 8 of that room. Note how the 2 comes from the upper 1 digit of the ID number, and the 8 comes from the lower 1 digit—just like the block number of a byte is given by the upper 23 bits in the address, while the byte number within block is given by the lower 9 bits.

[15] Recall note 14. Make SURE you understand this point.

= 181. So, the CPU would simply look at the 181^{st} byte in this line.[16]

If on the other hand, the CPU finds that block 2 does not have a copy in the cache, the CPU will read the *entire* block 2 from memory, and put it into some cache line. It may have to remove a copy of some other block in doing so; the term used for this is that that latter block copy is **evicted**.

The total number of lines varies with the computer; the number could be as large as the thousands, or on the other extreme less than ten. The larger the cache, the higher the hit rate. On the other hand, bigger caches are slower, take up too much space on a CPU chip in the case of on-chip caches, and are more expensive in the case of off-chip caches.

2.4.4.4 Direct-Mapped Policy

One question is which lines the CPU should search in checking whether the desired block has a copy in the cache. In a **fully-associative** cache, the CPU must inspect all lines. This gives maximum flexibility, but if it is large then it is expensive to implement and the search action is slower, due to the overhead of complex circuitry.

The other extreme is called **direct-mapped**.[17] Here the CPU needs to look in only *one* cache line. By design, the desired block copy will either be in that particular line or not in any line at all. The number of the particular line to be checked is the block number mod the number of lines. Let us suppose for illustration that there are 32 lines in the cache. In our example above, byte 1205 was in block 2, and since 2 mod 32 = 2, then to check whether block 2 has a copy in the cache, the CPU needs to look only at line 2. If the tag in line 2 says "This is a copy of block 2," then the CPU sees we have a hit. If not, the CPU knows it is a miss. The basic design of this direct-mapped cache is that a copy of block 2 will either be in the cache at line 2 or else the block will not have a copy in the cache at all.

Now suppose after successfully reading byte 1205, at some time later in the execution of this program we need to read byte 17589. This is block $\lfloor 17589/512 \rfloor = 34$. Since 34 mod 32 = 2, the CPU will again look at line 2 in the cache. But the tag there will say that this line currently contains a copy of block 2. Thus there will be a miss. The CPU will bring in a copy of block 34, which will replace the copy of block 2, and the tag will be updated accordingly.

2.4.4.5 What About Writes?

One issue to be resolved is what to do when we have a cache hit which is a write. The issue is that now the copy of the block in the cache will be different from the "real" block in memory. Eventually the two must be made consistent, but when? There are two main types of policies:

[16] This means the block begins with its "0^{th}" byte.

[17] A compromise scheme, called **set-associative**, is very common.

- Under a **write-through** policy, any write to the cache would also be accompanied by an immediate corresponding write to memory.
- Under a **write-back** policy, we do not make the memory block consistent with its cache copy until the latter is evicted. At that time, the entire block is written to memory.

Which policy is better will depend on the program being run. Suppose for example we only write to a cache line once before it is evicted. Then write-back would be rather disastrous, since the entire block would be written to memory even though the inconsistency consists of only one byte or word. On the other hand, in code such as

```
for (i = 0; i < 10000; i++)
   sum += x[i];
```

if **sum** is the only part of its cache line being used, we only care about the final value of **sum**, so updating it to memory 10,000 times under a write-through policy would be highly inefficient.

2.4.4.6 Programmability

As noted in Section 2.4.4.5, the behavior of a particular write policy depends on which program is being run. For this reason, many CPUs, e.g. the Pentium, give the programmer the ability to have some blocks subject to a write-through policy and others subject to write-back.

Similarly, the programmer may declare certain blocks not cacheable at all. The programmer may do this, for example, if the programmer can see that this block would have very poor cache behavior. Another reason for designating a block as noncacheable is if the block is also being accessed by a DMA device.

2.4.4.7 Details on the Tag and Misc. Line Information

Consider line 2 in the cache. We will search that line if the number of the block we want is equal to 2, mod 32. That means that the tag, which states the number of the block copy stored in this line, must have 00010 as its last 5 bits. In view of that, it would be wasteful to store those bits. So, we design the tag to consist only of the remainder of the block number, i.e. the upper 18 bits of the block number.

Let's look at our example 17589 above again. Recall that this is in block 34 = 00000000000000000100010. Since the last 5 bits are 00010, the CPU will examine the tag in cache line 2. The CPU will then see whether the tag is equal to 000000000000000001, the upper 18 bits of 34. If so, it is a hit.

At the beginning, a line might not contain any valid data at all. Thus we need another bit in the line, called the Valid Bit, which tells us whether there is valid data in that line (1 means valid).

If a write-back policy is used, we need to know whether a given cache line has been written to. If it has, then the line must be copied back to memory when it is evicted, but if not, we would want to skip that step. For this purpose, in addition to the tag, the line maintains a Dirty Bit, "dirty" meaning "has been written to."

Since we saved some bits in the storage of the tag, we use that space for the Valid and Dirty Bits.

2.4.4.8 Why Caches Usually Work So Well

Experience shows that most programs tend to reuse memory items repeatedly within short periods of time; for example, instructions within a program loop will be accessed repeatedly. Similarly, they tend to use items which neighbor each other (as is true for the data accesses in Version I above), and thus they stay in the same block for some time. This is called the principle of **locality of reference**, with the word "locality" meaning "nearness"—nearness in time or in space.

For these reasons, it turns out that cache misses tend to be rare. A typical hit rate is in the high 90s, say 97%. This is truly remarkable in light of the fact that the size of the cache is tiny compared to the size of memory. Again, keep in mind that this high hit rate is due to the locality, i.e. the fact that programs tend to NOT access memory at randomly-dispersed locations.

2.4.5 Disk Caches

We note also that one of the services an OS might provide is that of a **disk cache**, which is a software analog of the memory caches mentioned earlier. A disk is divided into **sectors**, which on PCs are 512 bytes each in size. The disk cache keeps copies of some sectors in main memory. If the program requests access to a certain disk sector, the disk cache is checked first. If that particular sector is currently present in the cache, then a time-consuming trip to the disk, which is much slower than accessing main memory, can be avoided.[18] Again this is transparent to the programmer. The program makes its request to the disk-access service of the OS, and the OS checks the cache and if necessary performs an actual disk access. (Note therefore that the disk cache is implemented in software, as opposed to the memory cache, which is hardware, though hardware disk caches exist too.)

2.4.6 Web Caches

The same principle is used in the software for Web servers. In order to reduce network traffic, an ISP might cache some of the more popular Web pages. Then when a request for a given page reaches the ISP, instead

[18] Note, though, that the access to the cache will still be somewhat slower than the access to a register. Here is why:

Suppose we have 16 registers. Then the register number would be expressed as a 4-bit string, since $16 = 2^4$. By contrast, say our cache contains 1024 lines, which would mean that we need 10 bits to specify the line number. Digital circuity to decode a 10-bit ID is slower than that for a 4-bit ID. Moreover, there would be considerable internal circuitry delays within the cache itself.

of relaying it to the real server for that Web page, the ISP merely sends the user a copy. Of course, that again gives rise to an update problem; the ISP must have some way of knowing when the real Web page has been changed.

Chapter 3

Introduction to Linux Intel Assembly Language

3.1 Overview of Intel CPUs

3.1.1 Computer Organization

Computer programs execute in the computer's **central processing unit** (CPU). Examples of CPUs are the Pentiums in PCs and the PowerPCs in Macs.[1] The program itself, consisting of both instructions and data are stored in **memory** (RAM) during execution. If you created the program by compiling a C/C++ source file (as opposed to writing in assembly language directly), the instructions are the machine language operations (add, subtract, copy, etc.) generated from your C/C++ code, while the data are your C/C++ variables. The CPU must fetch instructions and data from memory (as well as store data to memory) when needed; this is done via a **bus**, a set of parallel wires connecting the CPU to memory.

The components within the CPU include **registers**. A register consists of bits, with as many bits as the word size of the machine. Thus a register is similar to one word of memory, but with the key difference that a register is inside the CPU, making it much faster to access than memory. Accordingly, when programming in assembly language, we try to store as many of our variables as possible in registers instead of in memory. Similarly, if one invokes a compiler with an "optimize" command, the compiler will also attempt to store variables in registers instead of in memory (and will try other speedup tricks as well).

[1] Slated to be replaced by Intel chips.

3.1.2 CPU Architecture

There are many different different types of CPU chips. The most commonly-known to the public is the Intel Pentium, but other very common ones are the PowerPC chip used in Macintosh computers and IBM UNIX workstations, the MIPS chip used in SGI workstations and so on.

We speak of the **architecture** of a CPU. This means its instruction set and registers, the latter being units of bit storage similar memory words but inside the CPU.

It is highly important that you keep at the forefront of your mind that the term **machine language** does not connote "yet another programming language" like C or C++; instead, it refers to code consisting of instructions for a specific CPU type. A program in Intel machine language will be rejected as nonsense if one attempts to run it on, say, a MIPS machine. For instance, on Intel machines, 0111010111111000 means to jump back 8 bytes, while on MIPS it would either mean something completely different or would mean nothing at all.

3.1.3 The Intel Architecture

Intel CPUs can run in several different modes. On Linux the CPU runs in **protected, flat 32-bit mode**, or the same for 64 bits on machines with such CPUs. For our purposes here, we will not need to define the terms *protected* and *flat*, but do keep in mind that in this mode word size is 32 bits.

For convenience, we will assume from this point onward that we are discussing 32-bit CPUs.

Intel instructions are of variable length. Some are only one byte long, some two bytes and so on.

The main registers of interest to us here are named EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP.[2]

These are all 32-bit registers, but the registers EAX through EDX are also accessible in smaller portions. The less-significant 16 bit portion of EAX is called AX, and the high and low 8 bits of AX are called AH and AL, respectively. Similar statements hold for EBX, ECX and EDX.

Note that EBP and ESP may not be appropriate for general use in a given program. In particular, we should avoid use of ESP in programs with subroutines, for reasons to be explained in our unit on subroutines. If we are writing some assembly language which is to be combined with some C/C++ code, we won't be able to use EBP either.

Also, there is another register, the PC, which for Intel is named EIP (Extended Instruction Pointer). We can never use it to store data, since it is used to specify which instruction is currently executing. Similarly, the

[2]We will use the all-caps notation, EAX, EBX, etc. to discuss registers in the text, even though in program code they appear as **%eax**, **%ebx**, ...

Similarly, we will use all-caps notation when referring to instruction families, such as the MOV family, even though in program code they appear as **%movl**, **%movb**, etc.

flags register EFLAGS, to be discussed below, is reserved as well.

3.2 What Is Assembly Language?

Machine-language is consists of long strings of 0s and 1s. Programming at that level would be extremely tedious, so the idea of assembly language was invented. Just as hex notation is a set of abbreviations for various bit strings in general, assembly language is another set of abbreviations for various bit strings which show up in machine language.

For example, in Intel machine language, the bit string 01100110100111000011 codes the operation in which the contents of the AX register are copied to the BX register. Assembly language notation is much clearer:

```
mov %ax, %bx
```

The abbreviation "mov" stands for "move," which actually means "copy."

An **assembler** is a program which translates assembly language to machine language. Unix custom is that assembly-language file names end with a **.s** suffix. So, the assembler will input a file, say **x.s**, written by the programmer, and produce from it an **object file** named **x.o**. The latter consists of the compiled machine language, e.g. the bit string 0111010111111000 we mentioned earlier for a jump-back-8-bytes instruction. In the Windows world, assembly language soruce code file names end with **.asm**, from which the assembler produces machine code files with names ending with **.obj**.

You probably noticed that an assembler is similar to a compiler. This is true in some respects, but somewhat misleading. If we program in assembly language, we are specifying exactly what machine-language instructions we want, whereas if we program in, say, C/C++, we let the compiler choose what machine-language instructions to produce.

For instance, in the case of the assembly-language line above, we know for sure which machine instruction will be generated by the assembler—we know it will be a 16-bit MOV instruction, that the registers AX and BX will be used, etc. And we know a single machine instruction will be produced. By contrast, with the C statement

```
z = x + y;
```

we don't know which—or even how many—machine instructions will be generated by the compiler. In fact, different compilers might generate different sets of instructions.

3.3 Different Assemblers

Our emphasis will be on the GNU assembler AS, whose executable file is named **as**. This is part of the GCC package. Its syntax is commonly referred to as the "AT&T syntax," alluding to Unix's AT&T Bell Labs origins.

However, we will also be occasionally referring to another commonly-used assembler, NASM. It uses Intel's syntax, which is similar to that of **as** but does differ in some ways. For example, for two-operand instructions, **as** and NASM have us specify the operands in orders which are the reverse of each other, as you will see below.

It is very important to note, though, that the two assemblers will produce the same machine code. Unlike a compiler, whose output is unpredictable, we know ahead of time what machine code an assembler will produce, because the assembly-language **mnemonics** are merely handy abbreviations for specific machine-language bit fields.

Suppose for instance we wish to copy the contents of the AX register to the BX register. In **as** we would write

```
mov %ax,%bx
```

while in NASM it would be

```
mov bx,ax
```

but the same machine-language will be produced in both cases, 0110011010011100011, as mentioned earlier.

3.4 Sample Program

In this very simple example, we find the sum of the elements in a 4-word array, **x**.

```
1  # introductory example; finds the sum of the elements of an array
2
3  .data  # start of data section
4
5  x:
6        .long   1
7        .long   5
8        .long   2
9        .long   18
10
```

```
11  sum:
12        .long 0
13
14  .text  # start of code section
15
16  .globl _start
17  _start:
18        movl $4, %eax  # EAX will serve as a counter for
19                       # the number of words left to be summed
20        movl $0, %ebx  # EBX will store the sum
21        movl $x, %ecx  # ECX will point to the current
22                       # element to be summed
23  top:  addl (%ecx), %ebx
24        addl $4, %ecx  # move pointer to next element
25        decl %eax  # decrement counter
26        jnz top  # if counter not 0, then loop again
27  done: movl %ebx, sum  # done, store result in "sum"
```

3.4.1 Analysis

A source file is divided into **data** and **text** sections, which contain data and machine instructions, respectively.[3] Data consists of variables, as you are accustomed to using in C/C++.

First, we have the line

```
.data   # start of data section
```

The fact that this begins with '.' signals the assembler that this will be a **directive** (also known as a **pseudoinstruction**), meaning a command to the assembler rather than something the assembler will translate into a machine instruction. It is rather analogous to a note one might put in the margin, say "Please double-space here," of a handwritten draft to be given to a secretary for typing; one does NOT want the secretary to type "Please double-space here," but one does want the secretary to take some action there.

This directive here is indicating that what follows will be data rather than code.

The # character means that it and the remainder of the line are to be treated as a comment.

Next,

```
x:

      .long   1
      .long   5
      .long   2
      .long   18
```

[3] There may be other related sections as well. We will describe some of these in a later unit.

tells the assembler to make a note in **x.o** saying that later, when the operating system (OS) loads this program into memory for execution, the OS should set up four consecutive"long" areas in memory, set with initial contents 1, 5, 2 and 18 (decimal). "Long" means 32-bit size, so we are asking the assembler to arrange for four words to be set up in memory, with contents 1, 5, 2 and 18.[4] Moreover, we are telling the assembler that in our assembly code below, the first of these four long words will be referred to as **x**. We say that **x** is a **label** for this word.[5] Similarly, immediately following those four long words in memory will be a long word which we will refer to in our assembly code below as **sum**.

By the way, what if **x** had been an array of 1,000 long words instead of four, with all words to be initialized to, say, 8? Would we need 1,000 lines? No, we could do it this way:

```
x:
      .rept 1000
      .long 8
      .endr
```

The **.rept** directive tells the assembler to act as if the lines following **.rept**, up to the one just before **.endr**, are repeated the specified number of times.

What about an array of characters? We can ask the assembler to leave space for this using the **.space** directive, e.g. for a 6-character array:

```
y:  .space 6  # reserve 6 bytes of space
```

Or if we wish to have that space initialized to some string:

```
y:  .string "hello"
```

This will take up six bytes, including a null byte at the end; the developers of the **.string** directive decided on such a policy in order to be consistent with C. Note carefully, though, that they did NOT have to do this; there is nothing sacrosanct about having null bytes at the ends of character strings.

Getting back to our example here, we next have a directive signalling the start of the **text** section, meaning actual program code. Look at the first two lines:

```
_start:
      movl $4, %eax
```

[4]The term **long** here is a historical vestige from the old days of 16-bit Intel CPUs. The modern 32-bit word size is "long" in comparison to the old 16-bit size. Note that in the Intel syntax the corresponding term is **double**.

[5]Note that **x** is simply a name for the first word in the array, not the set of 4 words. Knowing this, you should now have some insight into why in C or C++, an array name is synonymous with a pointer to the first element of the array.

Keep in mind that whenever we are referring to **x**, both here in the program and also when we run the program via a debugger (see below), **x** will never refer to the entire array; it simply is a name for that first word in the array. Remember, we are working at the machine level, and there is no such thing as a data type here, thus no such thing as an array! Arrays exist only in our imaginations.

Here **_start** is another label, in this case for the location in memory at which execution of the program is to begin, called the **entry point**, in this case that **movl** instruction. We did not choose the name for this label arbitrarily, in contrast to all the others; the Unix linker takes this as the default.[6]

The **movl** instruction copies the constant 4 to the EAX register.[7] You can infer from this example that AS denotes constants by dollar signs.

The 'l' in "movl" means "long." The corresponding Intel syntax,

```
mov eax,4
```

has no such distinction, relying on the fact that EAX is a 32-bit register to implicitly give the same message to the assembler, i.e. to tell the assembler that we mean a 32-bit 4, not say, a 16-bit 4.

Keep in mind that **movl** is just one member of the move family of instructions. Later, for example, you will see another member of that family, **movb**, which does the same thing except that it copies a byte instead of a word.

As noted earlier, we will generally use all-caps notation to refer to instruction families, for example referring to MOV to any of the instructions **movl**, **movb**, etc.

By the way, integer constants are taken to be base-10 by default. If you wish to state a constant in hex instead, use the C "0x" notation.[8]

The second instruction is similar, but there is something noteworthy in the third:

```
movl $x, %ecx
```

In the token $4 in the first instruction, the dollar sign meant a constant, and the same is true for $x. The constant here is the address of **x**. Thus the instruction places the address of **x** in the ECX register, so that EAX serves as a pointer. A later instruction,

```
addl $4, %ecx
```

increments that pointer by 4 bytes, i.e. 1 word, each time we go around the loop, so that we eventually have the sum of all the words.

Note that $x has a completely different meaning that **x** by itself. The instruction

[6] The previous directive, **.globl**, was needed for the linker too. More on this in our unit on subroutines.

[7] We will usually use the present tense in remarks like this, but it should be kept in mind that the action will not actually occur until the program is executed. So, a more precise though rather unwieldy phrasing would be, "When it is later executed, the **movl** instruction will copy..."

[8] Note, though, that the same issues of endian-ness will apply in using the assembler as those related to the C compiler.

```
      movl x, %ecx
```

would copy the contents of the memory location **x**, rather than its address, to ECX.[9]

The next line begins the loop:

```
top:  addl (%ecx), %ebx
```

Here we have another label, **top**, a name which we've chosen to remind us that this is the top of the loop. This instruction takes the word pointed to by ECX and adds it to EBX. The latter is where I am keeping the total.

Recall that eventually we will copy the final sum to the memory location labeled **sum**. **We don't want to do so within the loop, though, because memory access is much slower than register access (since we must leave the CPU to go to memory), and we thus want to avoid it. So, we keep our sum in a register, and copy to memory only when we are done.**[10]

If we were not worried about memory access speed, we might store directly to the variable **sum**, as follows:

```
      movl $sum,%edx  # use %edx as a pointer to "sum"
      movl $0,%ebx
top:  addl (%ecx), %ebx  # old sum is still in %ebx
      movl %ebx,(%edx)
```

Note carefully that we could NOT write

```
      movl $sum,%edx  # use %edx as a pointer to "sum"
top:  addl (%ecx),(%edx)
```

because there is no such instruction in the Intel architecture. Intel chips (like most CPUs) do not allow an instruction to have both its operands in memory.[11] **Note that this is a constraint placed on us by the hardware, not by the assembler.**

The bottom part of the loop is:

```
      decl %eax
      jnz top
```

[9] The Intel syntax is quite different. Under that syntax, x would mean the address of **x**, and the contents of the word **x** would be denoted as [x].

[10] This presumes that we need it in memory for some other reason. If not, we would not do so.

[11] There are actually a couple of exceptions to this on Intel chips, as will be seen in Section 3.17.

The DEC instruction, **decl** ("decrement long"), in AT&T syntax, subtracts 1 from EAX. This instruction, together with the JNZ following it, provides our first illustration of the operation of the EFLAGS register in the CPU:

When the hardware does almost any arithmetic operation, it also records whether the result of this instruction is 0: It sets Zero flag, ZF, in the EFLAGS register to 1 if the result of the operation was 0, and sets that flag to 0 if not.[12] Similarly, the hardware sets the Sign flag to 1 or 0, according to whether the result of the subtraction was negative or not.

Most arithmetic operations do affect the flags, but for instance MOV does not. For a given instruction, you can check by writing a short test program, or look it up in the official Intel CPU manual.[13]

Now, here is how we make use of the Zero flag. The JNZ ("jump if not zero") instruction says, "If the result of the last arithmetic operation was not 0, then jump to the instruction labeled **top**." The circuitry for JNZ implements this by jumping if the Zero flag is 1. (The complementary instruction, JZ, jumps if the Zero flag is 0, i.e. if the result of the last instruction was zero.)

So, the net effect is that we will go around the loop four times, until EAX reaches 0, then exit the loop (where "exiting" the loop merely means going to the next instruction, rather than jumping to the line labeled **top**).

Even though our example jump here is backwards—i.e. to a lower-addressed memory location—forward jumps are just as common. The reader should think about why forward jumps occur often in "if-then-else" situations, for example.

By the way, in computer architecture terminology, the word "branch" is a synonym for "jump." In many architectures, the names of the jump instructions begin with 'B' for this reason.

The EFLAGS register is 32 bits wide (numbered 31, the most significant, to 0, the least significant), like the other registers. Here are some of the flag and enable bits it includes:[14]

Data	Position
Overflow Flag	Bit 11
Interrupt Enable	Bit 9
Sign Flag	Bit 7
Zero Flag	Bit 6
Carry Flag	Bit 0

[12] The reason why the hardware designers chose 1 and 0 as codes this way is that they want us to think of 1 as meaning yes and 0 as meaning no. So, if the Zero flag is 1, the interpretation is "Yes, the result was zero.

[13] This is on the Intel Web site, but also available on our class Web page at `http://heather.cs.ucdavis.edu/~matloff/50/IntelManual.PDF`. Go to Index, then look up the page number for your instruction. Once you reach the instruction, look under the subheading Flags Affected.

[14] The meanings of these should be intuitive, except for the Interrupt Enable bit, which will be described in our unit on input/output.

So, for example there are instructions like JC ("jump if carry"), JNC and so on which jump if the Carry Flag is set.

Note that the label **done** was my choice, not a requirement of **as**, and I didn't need a label for that line at all, since it is not referenced elsewhere in the program. I included it only for the purpose of debugging, as seen later.

Just as there is a DEC instruction for decrementing, there is INC for incrementing.

3.4.2 Source and Destination Operands

In a two-operand instruction, the operand which changes is called the **destination** operand, and the other is called the **source** operand. For example, in the instruction

```
movl %eax, %ebx
```

EAX is the source and EBX is the destination.

3.4.3 Remember: No Names, No Types at the Machine Level

Keep in mind that, just as our variable names in a C/C++ source file do not appear in the compiled machine language, in an assembly language source file labels—in this case, **x**, **sum**, **_start**, **top** and **done**—are just temporary conveniences for us humans. We use them only in order to conveniently refer AS to certain locations in memory, in both the **.data** and **.text** sections. These labels do NOT appear in the machine language produced by AS; only numeric memory addresses will appear there.

For instance, the instruction

```
jnz top
```

mentions the label **top** here in assembly language, but the actual machine language which comes from this is 0111010111111000. You will find in our unit on machine language that the first 8 bits, 01110101, code a jump-if-not-zero operation, and the second 8 bits, 11111000, code that the jump target is 8 bytes backward. Don't worry about those codes for now, but the point at hand now is that neither of those bit strings makes any mention of **top**.

Again, there are no types at the machine level. So for example there would be no difference between writing

```
z:
   .long 0
w:
   .byte 0
```

and

```
z:
   .rept 5
   .byte 0
   .endr
```

Both of these would simply tell AS to arrange for 5 bytes of (zeroed-out) space at that point in the data section. Make sure to avoid the temptation of viewing the first version as "declaring" an integer variable **z** and a character variable **w**. True, the programmer may be interpreting things that way, but the two versions would produce exactly the same **.o** file.

3.4.4 Dynamic Memory Is Just an Illusion

One thing to note about our sample program above is that all memory was allocated statically, i.e. at assembly time.[15] So, we have statically allocated five words in the **.data** section, and a certain number of bytes in the **.text** section (the number of which you'll see in our unit on machine language).

Since C/C++ programs are compiled into machine language, and since assembly language is really just machine language, you might wonder how it can be that memory can be dynamically allocated in C/C++ yet not in assembly language. The answer to this question is that one cannot truly do dynamic allocation in C/C++ either. Here is what occurs:

Suppose you call **malloc()** in C or invoke **new** in C++. The latter, at least for G++, in turn calls **malloc()**, so let's focus on that function. When you run a program whose source code is in C/C++, some memory is set up called the **heap**. Any call to **malloc()** returns a pointer to some memory in the heap; **malloc()**'s internal data structures then record that that portion of memory is in use. Later, if the program calls **free()**, those data structures now mark the area as available for use.

In other words, the heap memory itself was in fact allocated when the program was loaded, i.e. statically. However, **malloc()** and its sister functions such as **free()** manage that memory, recording what is available for use now and what isn't. So the notion of dynamic memory allocation is actually an illusion.

An assembly language programmer could link the C library into his/her program, and thus be able to call **malloc()** if that were useful. But again, it would not actually be dynamic allocation, just like it is not in the C/C++ case.

[15] Again, be careful here. The memory is not actually assigned to the program until the program is actually run. Our word *allocated* here refers to the fact that the asssembler had already planned the memory usage.

3.5 Use of Registers Versus Memory

Recall that registers are located inside the CPU, whereas memory is outside it. Thus, register access is much faster than memory access.

Accordingly, when you do assembly language programming, you should try to minimize your usage of memory, i.e. of items in the **.data** section (and later, of items on the stack), especially if your goal is program speed, which is a common reason for resorting to assembly language.

However, most CPU architectures have only a few registers. Thus in some cases you may run out of registers, and need to store at least some items in memory.[16]

3.6 Another Example

The following example does a type of sort. See the comments for an outline of the algorithm. (This program is merely intended as an example for learning assembly language, not for algorithmic efficiency.)

One of the main new ideas here is that of a **subroutine**, which is similar to the concept of a function in C/C++.[17] In this program, we have subroutines **init()**, **findmin()** and **swap()**.

```
1  # sample program; does a (not very efficient) sort of the array x, using
2  # the algorithm (expressed in pseudo-C code notation):
3
4  # for each element x[i]
5  #    find the smallest element x[j] among x[i+1], x[i+2], ...
6  #    if new min found, swap x[i] and x[j]
7
8  .equ xlength, 7  # number of elements to be sorted
9
10 .data
11 x:
12       .long   1
13       .long   5
14       .long   2
15       .long   18
16       .long   25
17       .long   22
18       .long   4
19
20 .text
21       # register usage in "main()":
22       #    EAX points to next place in sorted array to be determined,
23       #       i.e. "x[i]"
24       #    ECX is "x[i]"
25       #    EBX is our loop counter (number of remaining iterations)
```

[16] Recall that a list of usable registers for Intel was given in Section 3.1.3.

[17] In fact, the compiler translates C/C++ functions to subroutine at the machine-language level.

```
26        #    ESI points to the smallest element found via findmin
27        #    EDI contains the value of that element
28  .globl _start
29  _start:
30        call init  # initialize needed registers
31  top:
32        movl (%eax), %ecx
33        call findmin
34        # need to swap?
35        cmpl %ecx, %edi
36        jge nexti
37        call swap
38  nexti:
39        decl %ebx
40        jz done
41        addl $4, %eax
42        jmp top
43
44  done: movl %eax, %eax  # dummy, just for running in debugger
45
46  init:
47        # initialize EAX to point to "x[0]"
48        movl $x, %eax
49        # we will have xlength-1 iterations
50        movl $xlength, %ebx
51        decl %ebx
52        ret
53
54  findmin:
55        # does the operation described in our pseudocode above:
56        #    find the smallest element x[j], j = i+1, i+2, ...
57        # register usage:
58        #    EDX points to the current element to be compared, i.e. "x[j]"
59        #    EBP serves as our loop counter (number of remaining iterations)
60        #    EDI contains the smallest value found so far
61        #    ESI contains the address of the smallest value found so far
62        # haven't started yet, so set min found so far to "infinity" (taken
63        #    here to be 999999; for simplicity, assume all elements will be
64        #    <= 999999)
65        movl $999999, %edi
66        # start EDX at "x[i+1]"
67        movl %eax, %edx
68        addl $4, %edx
69        # initialize our loop counter (nice coincidence:  number of
70        #    iterations here = number of iterations remaining in "main()")
71        movl %ebx, %ebp
72        # start of loop
73  findminloop:
74        # is this "x[j]" smaller than the smallest we've seen so far?
75        cmpl (%edx), %edi  # compute destination - source, set EFLAGS
76        js nextj
77        # we've found a new minimum, so update EDI and ESI
78        movl (%edx), %edi
79        movl %edx, %esi
80  nextj:  # do next value of "j" in the loop in the pseudocode
81        # if done with loop, leave it
82        decl %ebp
83        jz donefindmin
```

```
84          # point EDX to the new "x[j]"
85          addl $4, %edx
86          jmp findminloop
87    donefindmin:
88          ret
89
90    swap:
91          # copy "x[j]" to "x[i]"
92          movl %edi, (%eax)
93          # copy "x[i]" to "x[j]"
94          movl %ecx, (%esi)
95          ret
```

Note that there are several new instructions used here, as well as a new pseudoinstruction, **.equ**.

Let's deal with the latter first:

```
.equ xlength, 7
```

This tells the assembler that, in every line in which it sees **xlength**, the assembler should assemble that line as if we had typed 7 there instead of **xlength**. In other words **.equ** works like **#define** in C/C++. Note carefully that **xlength** is definitely not the same as a label in the **.data** section, which is the name we've given to some memory location; in other words, **xlength** is not the name of a memory location.

At the beginning of the **.text** section, we see the instruction

```
call init
```

The CALL instruction is a type of jump, with the extra feature that it records the place the jump was made from. That record is made on the **stack**, which we will discuss in detail in our unit on subroutines. Here we will jump to the instruction labeled **init** further down in the source file:

```
init:
      movl $x, %eax
      movl $xlength, %ebx
      decl %ebx
      ret
```

After the CALL instruction brings us to **init**, the instructions there, i.e.

```
      movl $x, %eax
      ...
```

will be executed, just as we've seen before. But the RET ("return") instruction **ret** is another kind of jump; it jumps back to the instruction immediately following the place at which the call to **init** was made. Recall

that that place had been recorded at the time of the call, so the machine does know it, by looking at the stack. Keep in mind that execution of a RET will result in a return to the instruction *following* the CALL. That instruction is the one labeled **top**:

```
top:
      mov (%eax), %ecx
```

Again, you will find out how all this works later, in our unit on subroutines. But it is being introduced here, to encourage you to start using subroutines from the beginning of your learning of assembly language programming. It facilitates a top-down approach. (See Section 3.19.)

As always, remember that the CPU is just a "dumb machine." Suppose for example that we accidentally put a RET instruction somewhere in our code where we don't intend to have one. If execution reaches that line, the RET will indeed be executed, no questions asked. The CPU has no way of knowing that the RET shouldn't be there. It does not know that we are actually not in the midst of a subroutine.

Also, though you might think that the CPU will balk when it tries to execute the RET but finds no record of call location on the stack, the fact is that there is always *something* on the stack even if it is garbage. So, the CPU will return to a garbage point. This is likely to cause various problems, for instance possibly a seg fault, but the point is that the CPU will definitely **not** say, "Whoa, I refuse to do this RET."

For that matter, even the assembler would not balk at our accidentally putting a RET instruction at some random place in our code. Remember, the assembler is just a clerk, so for example it does not care that that RET was not paired with a CALL instruction.

Note by the way that the code beginning at **__start** is analogous to **main()** in C/C++, and the comments in the code use this metaphor.

The **init()** subroutine does what it says, i.e. initialize the various registers to the desired values.

The next instruction is another subroutine call, to **findmin()**. As described in the pseudocode, it finds the smallest element in the remaining portion of the array. Let's not go into the details of how it does this yet—not only should one *write* code in a top-down manner, but one should also *read* code that way. We then check to see whether a swap should be done, and if so, we do it.

Another new instruction is CMP ("compare"), **cmpl**, which is used within **findmin()**. As noted in the comment in the code, this instruction subtracts the source from the destination; the result, i.e. the difference, is not stored anywhere, but the key point is that the EFLAGS register is affected.

Since there is an instruction for addition, it shouldn't be a surprise to know there is one for subtraction too. For example,

```
subl %ebx, %eax
```

subtracts c(EBX) from c(EAX), and places the difference back into EAX. This instruction does the same

thing as CMP, except that the latter does not store the difference back anyway; the computation for the **cmpl** is done only for the purpose of setting the flags.

Following the CMP instruction is JS ("jump if the Sign flag is 1), meaning that we jump if the result of the last arithmetic computation was "signed," i.e. negative. (The complementary instruction, JNS, jumps if the result of the last computation was not negative.)

In other words, the combined effect of the CMP and JS instructions here is that we jump to **nextj** if the contents of EDI is less than that of the memory word pointed to by EDX. In this case, we have not found a new minimum, so we just go to the next iteration of the loop.

The instruction

```
jmp top
```

is conceptually new. All the jump instructions we've seen so far have been **conditional**, i.e. the jump is made only if a certain condition holds. But JMP means to jump unconditionally.

Note my comments on the usage I intend for the various registers, e.g. in **"main()"**:

```
# register usage in "main()":
#    EAX points to next place in sorted array to be determined,
#       i.e. "x[i]"
#    ECX is "x[i]"
#    EBX is our loop counter (number of remaining iterations)
#    ESI points to the smallest element found via findmin
#    EDI contains the value of that element
```

I put these in **BEFORE** I started writing the program, to help keep myself organized, and I found myself repeatedly referring to them during the writing process. **YOU SHOULD DO THE SAME**.

Also note again that we absolutely needed to avoid using ESP as storage here, due to the fact that we are using **call** and **ret**. The reason for this will be explained in our unit on subroutines.

3.7 Addressing Modes

Consider the examples

```
movl $9, %ebx
movl $9, (%ebx)
movl $9, x
```

As you can see, the circuitry which implements **movl** allows for several different versions of the instruction. In the first example above, we copy 9 to a register. In the second and third examples, we copy to places in

memory, but even then, we do it via different ways of specificying the memory location—using a register as a pointer to memory in the second example, versus directly stating the given memory location in the third example.

The manner in which an instruction specifies an operand is called the **addressing mode** for that operand. So, above we see three distinct addressing modes for the second operand of the instruction.

For example, let's look at the instruction

```
movl $9, %ebx
```

Let's look at the destination operand first. Since the operand is in a register, we say that this operand is expressed in **register mode**. We say that the source operand is accessed in **immediate** mode, which means that the operand, in this case the number 9, is right there (i.e. "immediately within") in the instruction itself.[18]

Now consider the instruction

```
movl $9, (%ebx)
```

Here the destination operand is the memory word pointed to by EBX. This is called **indirect mode**; we "indirectly" state the location, by saying, "It's whatever EBX points to."

By contrast, the destination operand in

```
movl $9, x
```

is accessed in **direct mode**; we have directly stated where in memory the operand is.

Other addressing modes will be discussed in Section 3.12.

3.8 Assembling and Linking into an Executable File

3.8.1 Assembler Command-Line Syntax

To assemble an AT&T-syntax source file, say **x.s**, we will type

```
as -a --gstabs -o x.o x.s
```

[18] You'll see this more explicitly when we discuss machine language.

The **-o** option specifies what to call the object file, the machine-language file produced by the assembler. Here we are telling the assembler, "Please call our object file x.o."[19]

The **-a** option tells the assembler to display to the screen the source (i.e. assembly) code, machine code and section offsets[20] side-by-side, for easier viewing by us humans.

The **–gstabs** option (note that there are two hyphens, not one) tells the assembler to retain in **x.o** the **symbol table**, a list of the locations of whatever labels are in **x.s**, in a form usable by symbolic debuggers, in our case GDB or DDD.[21]

Things are similar under other operating systems. The Microsoft assembler, MASM, has similar command-line options, though of course with different names and some difference in functionality. [22]

3.8.2 Linking

Say we have an assembly language source file **x.s**, and then assemble it to produce an object file **x.o**. We would then type, say,

```
ld -o x x.o
```

Here **ld** is the linker, LD, which would take our object file **x.o** and produce an executable file **x**.

We will discuss more on linking in Section 3.13.2.

3.8.3 Makefiles

By the way, you can automate the assembly and linkage process using Makefiles, just as you would for C/C++. Keep in mind that makefiles have nothing to do with C or C++. They simply state which files depend on which other files, and how to generate the former files from the latter files.

So for example the form

```
x: y
<TAB> z
```

[19] **Important note on disaster avoidance:** Suppose we are assembling more than one file, say

as -a –gstabs -o z.o x.s y.s

Suppose we inadvertently forgot to include the z.o on the command line here. Then we would be telling the assembler, "Please call our object file **x.s**." This would result in the assembler overwriting **x.s** with the object file, trashing **x.s**! Be careful to avoid this.

[20] These are essentially addresses, as will be explained later.

[21] The **–gstabs** option is similar to **-g** when running GCC.

[22] By the way, NASM is available for both Unix and MS Windows. For that matter, even AS can be used under Windows, since it is part of the GCC package that is available for Windows under the name Cygwin.

simply says, "The file **x** depends on **y**. If we need to make **x** (or re-make it if we have changed **y**), then we do **z**."

So, for our source file **x.s** above, our Makefile might look like this:

```
x: x.o
<TAB> ld -o x x.o

x.o: x.s
<TAB> as --gstabs -o x.o x.s
```

3.9 How to Execute Those Sample Programs

3.9.1 "Normal" Execution Won't Work

Suppose in our sum-up-4-words example above we name the source file **Total.s**, and then assemble and link it, with the final executable file named, say, **tot**. We could not simply type

```
% tot
```

at the Unix command line. The program would crash with a segmentation fault. Why is this?

The basic problem is that after the last instruction of the program is executed, the processor will attempt to execute the "instruction" at the next location of memory. There is no such instruction, but the CPU won't know that. All the CPU knows is to keep executing instructions, one after the other. So, when your program marches right past its last real instruction, the CPU will try to execute the garbage there. I call this "going off the end of the earth."

Actually, in Linux the linker will arrange for our **.data** section to follow our **.text** section in memory, almost immediately after the end of the latter. It's "almost" because we would like the **.data** section to begin on a word boundary, i.e. at an address which is a multiple of 4. The area in between is padding, consisting of 0s, in this case three bytes of 0s.[23] Thus the "garbage" which we execute when we "go off the end of the earth" is our own data![24]

This doesn't happen with your compiled C/C++ program, because the compiler inserts a **system call**, i.e. a call to a function in the operating system, which in this case is the **exit()** call. (Actually, good programming practice would be to insert this call in your C/C++ programs yourself.) This results in a graceful transition from your program to the OS, after which the OS prints out your familiar command-line prompt.

We could have inserted system calls in our sample assembly language programs above too, but did not done so because that is a topic to be covered later in the course. Note that that also means we cannot do

[23] This can be determined using information presented in our unit on machine language.

[24] Possibly preceded by 1-3 0 bytes.

input and output, which is done via system calls too—so, not only does our program crash if we run it in the straightforward manner above, but also we have no way of knowing whether it ran correctly before crashing!

So, in our initial learning environment here, we will execute our programs via a debugger, either DDD or GDB, which will allow us to have the program stop when it is done and to see the results.

3.9.2 Running Our Assembly Programs Using GDB/DDD

Since a debugger allows us to set breakpoints or single-step through programs, we won't "go off the end of the earth" and cause a seg fault as we would by running our programs directly.[25]

Moreover, since the debuggers allow us to inspect registers and memory contents, we can check the "output" of our program. In our first array-summing program in Section 3.4, for example, the sum was in EBX, so the debugger would enable us to check the program's operation by checking whether EBX contained the correct sum.

3.9.2.1 Using DDD for Executing Our Assembly Programs

Starting the Debugger:

For our array-summing example, we would start by assembling and linking the source code, and then typing

```
% ddd tot
```

Your source file **Total.s** should appear in the DDD Source Window.

Making Sure You Don't "Go Off the End of the World":

You would set a breakpoint at the line labeled **done** by clicking on that line and then on the red stop sign icon at the top of the window. This arranges for your program to stop when it is done.

Running the Program:

You would then run the program by clicking on Run. The program would stop at **done**.

Checking the "Output" of the Program:

You may have written your program so that its "output" is in one or more registers. If so, you can inspect the register contents when you reach **done**. For example, recall that in the **tot** program, the final sum (26) will be stored in the EBX register, so you would inspect the contents of this register in order to check whether the program ran correctly.

[25] Keep in mind, that if we don't set a breakpoint when we run a program within the debugger, we will still "go off the end of the earth."

To do this, click on Status then Registers.

On the other hand, our program's output may be in memory, as is the case for instance for the program in Section 3.6. Here we check output by inspecting memory, as follows:

Hit Data, then Memory. An Examine Memory window will pop up, asking you to state which data items you wish to inspect:

- Fill in the blank on the left with the number of items you want to inspect.
- In the second field, labled "octal" by default, state how you want the contents of the items described—in decimal, hex or whatever.
- In the third field, state the size of each item—byte, word or whatever.
- In the last field, give the address of the first item.

For the purpose of merely checking a program's output we would choose Print here rather than Display. The former shows the items just once, while the latter does so continuously; the latter is useful for actual debugging of the program, while the former is all we need for merely checking the program's output.

For instance, again consider the example in Section 3.6. We wish to inspect all seven array elements, so we fill in the Examine Memory window to indicate that we wish to Print 7 Decimal Words starting at **&x**.

Note on endian-ness, etc.: If you ask the debugger to show a word-sized item (i.e. a memory word or a full register), it will be shown most-significant byte first. Within a byte, the most-significant bit will be shown first.

3.9.2.2 Using GDB for Executing Our Assembly Programs

In some cases, you might find it more convenient to use GDB directly, rather than via the DDD interface. For example, you might be using **telnet**.

(Note: It is assumed here that you have already read the material on using DDD above.)

Starting the Debugger:

For the **tot** program example here, you would start by assembling and linking, and then typing

```
gdb tot
```

Making Sure You Don't "Go Off the End of the World":

Set the breakpoint at **done**:

```
(gdb) b done
Breakpoint 1 at 0x804808b: file sum.s, line 28.
```

Running the Program:

Issue the **r** ("run") command to GDB.

Checking the "Output" of the Program:

To check the value of a register, e.g. EBX, use the **info registers** command:

```
(gdb) info registers ebx
ebx            0x1a     26
```

To check the value of a memory location, use the **x** ("examine") command. In the example in Section 3.6, for instance:

```
x/7w &x
```

This says to inspect 7 words, beginning at **x**, printing out the contents in hex. In some cases, e.g. if you are working with code translated from C to assembly language, GDB will switch to printing individual bytes, in which case use, e.g., **x/7x** instead of **x/7w**.

There are other ways to print out the contents, e.g.

```
x/12c &x
```

would treat the 12 bytes starting at **x** as characters and print them out.

3.10 How to Debug Assembly Language Programs

3.10.1 Use a Debugging Tool for ALL of Your Programming, in EVERY Class

I've found that many students are **shooting themselves in the foot** by not making use of debugging tools. They learn such a tool in their beginning programming class, but treat it as something that was only to be learned for the final exam, rather than for their own benefit. Subsequently they debug their programs with calls to printf() or cout, which is really a slow, painful way to debug. **You should make use of a debugging tool in all of your programming work – for <u>your</u> benefit, not your professors'.** (See my debugging-tutorial slide show, at `http://heather.cs.ucdavis.edu/~matloff/debug.html`.)

For C/C++ programming on Unix machines, many debugging tools exist, some of them commercial products, but the most commonly-used one is GDB. Actually, many people use GDB only indirectly, using DDD as their interface to GDB; DDD provides a very nice GUI to GDB.

3.10.2 General Principles

3.10.2.1 The Principle of Confirmation

Remember (as emphasized in the debugging slide show cited above) **the central principle of debugging is confirmation.** We need to step through our program, at each step confirming that the various registers and memory locations contain what we think they ought to contain, and confirming that jumps occur when we think they ought to occur, and so on. Eventually we will reach a place where something fails to be confirmed, and then that will give us a big hint as to where our bug is.

3.10.2.2 Don't Just Write Top-Down, But Debug That Way Too

Consider code like, say,

```
movl $12, %eax
call xyz
addl %ebx, %ecx
```

When you are single-stepping through your program with the debugging tool and reach the line with the **call** instruction, the tool will give you a choice of going in to the subroutine (e.g. the Step command in DDD) or skipping over it (the Next command in DDD). Choose the latter at first. When you get to the next line (with **addl**), you can check whether the "output" of the subroutine **xyz()** is correct; if so, you will have saved a lot of time and distraction by skipping the detailed line-by-line execution of **xyz()**. If on the other hand you find that the output of the subroutine is wrong, then you have narrowed down the location of your bug; you can then re-run the program, but in this case opt for Step instead of Next when you get to the call.

3.10.3 Assembly Language-Specific Tips

3.10.3.1 Know Where Your Data Is

The first thing you should do during a debugging session is write down the addresses of the labeled items in your **.data** section. And then use those addresses to help you debug.

Consider the program in Section 3.6, which included a data label **x**. We should first determine where **x** is. We can do this in GDB as follows:

```
(gdb) p/x &x
```

In DDD, we might as well do the same thing, issuing a command directly to GDB via DDD's Console window.

Knowing these addresses is extremely important to the debugging process. Again using the program in Section 3.6 for illustration, on the line labeled **top** the EAX register is serving as a pointer to our current position in **x**. In order to verify that it is doing so, we need to know the address of **x**.

3.10.3.2 Seg Faults

Many bugs in assembly language programs lead to seg faults. These typically occur because the programmer has inadvertently used the wrong addressing mode or the wrong register.

For example, consider the instruction

```
movl $39, %edx
```

which copies the number 39 to the EDX register.

Suppose we accidentally writes

```
movl $39, (%edx)
```

This copies 39 to the memory location pointed to by EDX. If EDX contains, say, 12, then the CPU will attempt to copy 39 to memory location 12, which probably will not be in the memory space allocated to this program when the OS loaded it into memory. In other words, we used the wrong addressing mode, which will cause a seg fault.

When we run the program in the debugger, the latter will tell us exactly where—i.e. at what instruction—the seg fault occurred. This is of enormous value, as it pinpoints exactly where our bug is. Of course, the debugger won't tell us, "You dummy, you used the wrong addressing mode"—the programmer then must determine why that particular instruction caused the seg fault—but at least the debugger tells us where the fault occurred.

Suppose on the other hand, we really did need to use indirect addressing mode, but accidentally specified ECX insteade of EDX. In other words, we intended to write

```
movl $39, (%edx)
```

but accidentally wrote

```
movl $39, (%ecx)
```

Say c(EDX) = 0x10402226 but c(ECX) = 12,[26] and that 0x10402226 is in a part of memory which was allocated to our program but 12 is not. Again we will get a seg fault, as above. Again, the debugger won't tell us, "You dummy, you should have written "%edx", not "%ecx", but at least it will pinpoint which instruction caused the seg fault, and we then can think about what we might have done wrong in that particular instruction.

3.10.4 Use of DDD for Debugging Assembly Programs

To examine register contents, hit Status and then Registers. All register contents will be displayed.

The display includes EFLAGS, the flags register. The Carry Flag is bit 0, i.e. the least-significant bit. The other bits we've studied are the Zero Flag, bit 6, the Sign Flag, bit 7, and the Overflow Flag, bit 11.

To display, say, an array **z**, hit Data, then Memory. State how many cells you want displayed, what kind of cells (whole words, individual bytes, etc.), and where to start, such as **&z**. It will also ask whether you want the information "printed," which means displayed just once, or "displayed," which means a continuing display which reflects the changes as the program progresses.

Breakpoints may be set and cleared using the stop sign icons.[27]

You can step through your code line by line in the usual debugging-tool manner. However, in assembly language, make sure that you use Stepi instead of Next or Step, since we are working at the machine instruction level (the 'i' stands for "instruction.")

Note that when you modify your assembly-language source file and reassemble and link, DDD will not automatically reload your source and executable files. To reload, click on File in DDD, then Open Program and click on the executable file.

3.10.5 Use of GDB for Debugging Assembly Programs

3.10.5.1 Assembly-Language Commands

Assuming you already know GDB (see the link to my Web tutorial), here are the two new commands you should learn.

- To view all register contents, type

```
info registers
```

[26] Recall that c() means "contents of."

[27] For some reason, it will not work if we set a breakpoint at the very first instruction of a program, though any other instruction works.

You can view specific registers with the **p** ("print") command, e.g.

```
p/x $pc
p/x $esp
p/x $eax
```

- To view memory, use the **x** ("examine") command. If for example you have a memory location labeled **z** and wish to examine the first four words starting at a data-section label **z**, type

  ```
  x/4w &z
  ```

  Do not include the ampersand in the case of a text-section label. Note that the **x** command differs greatly from the **p** command, in that the latter prints out the contents of only one word.

  Note too that you can do indirection. For example

  ```
  x/4w $ebx
  ```

  would display the four words of memory beginning at the word pointed to by EBX.

- As in the DDD case, use the Stepi mode of single-stepping through code;[28] the command is

  ```
  (gdb) stepi
  ```

  or just

  ```
  (gdb) si
  ```

Unlike DDD, GDB automatically reloads the program's executable file when you change the source.

An obvious drawback of GDB is the amount of typing required. But this can be greatly mitigated by using the "define" command, which allows one to make abbreviations. For example, we can shorten the typing needed to print the contents of EAX as follows:

```
(gdb) define pa
Type commands for definition of "pa".
End with a line saying just "end".
>p/x $eax
>end
```

From then on, whenever we type **pa** in this **gdb** session, the contents of EAX will be printed out.

Moreover, if we want these abbreviations to carry over from one session to another for this program, we can put them in the file **.gdbinit** in the directory containing the program, e.g. by placing these lines

[28] The Nexti mode is apparently unreliable. Of course, you can still hop through the code using breakpoints.

```
define pa
p/x $eax
end
```

in **.gdbinit**, **pa** will automatically be defined in each debugging session for this program.

Use **gdb**'s online help facility to get further details; just type "help" at the prompt.

3.10.5.2 TUI Mode

As mentioned earlier, it is much preferable to use a GUI for debugging, and thus the DDD interface to GDB is highly recommended. As a middle ground, though, you may try GDB's new TUI mode. You will need a relatively newer version of GDB for this, and it will need to have been built to include TUI.[29]

TUI may be invoked with the **-tui** option on the GDB command line. While running GDB, you toggle TUI mode on or off using **ctrl-x a**.

If your source file is purely in assembly language, i.e. you have no **main()**, first issue GDB's **l** ("list") command, and hit Enter an extra time or two. That will make the source-code subwindow appear.

Then, say, set a breakpoint and issue the **r** ("run") command to GDB as usual.

In the subwindow, breakpoints will be marked with asterisks, and your current instruction will be indicated by a $>$ sign.

In addition to displaying a source code subwindow, TUI will also display a register subwindow if you type

```
(gdb) layout reg
```

This way you can watch the register values and the source code at the same time. TUI even highlights a register when it changes values.

Of course, since TUI just adds an interface to GDB, you can use all the GDB commands with TUI.

3.10.5.3 CGDB

Recall that the goal of TUI in our last subsection is to get some of the functionality of a GUI like DDD while staying within the text-only realm. If you are simply Telnetting into to the machine where you are debugging a program, TUI is a big improvement over ordinary GDB. CGDB is another effort in this direction.

[29] If your present version of GDB does not include TUI (i.e. GDB fails when you invoke it with the **-tui** option), you can build your own version of GDB. Download it from `www.gnu.org`, run **configure** with the option **–enable-tui**, etc.

Whereas TUI is an integral part of GDB, CGDB is a separate front end to GDB, not developed by the GDB team. (Recall that DDD is also like this, but as a GUI rather than being text-based.) You can download it from `http://cgdb.sourceforge.net/`.

Like TUI, CGDB will break the original GDB window into several subwindows, one of which is for GDB prompts and the other for viewing the debuggee's source code. CGDB goes a bit further, by allowing easy navigation through the source-code subwindow, and by using a nice colorful interface.

To get into the source-code subwindow, hit Esc. You can then move through that subwindow using the **vi**-like commands, e.g. **j** and **k** to move down or up a line, **/** to search for text, etc.

To set a breakpoint at the line currently highlighted by the cursor, just hit the space bar. Breakpoints are highlighted in red,[30] and the current instruction in green.

Use the **i** command to get to the GDB command subwindow.

CGDB's startup file is **cgdbrc** in a directory named **.cgdb** in your home directory. One setting you should make sure to have there is

```
set autosourcereload
```

which will have CGDB automatically update your source window when you recompile.

3.11 Some More Operand Sizes

Recall the following instruction from our example above:

```
addl (%ecx), %ebx
```

How would this change if we had been storing our numbers in 16-bit memory chunks?

In order to do that, we would probably find it convenient[31] to use **.word** instead of **.long** for initialization in the **.data** section. Also, the above instruction would become

```
addw (%ecx), %bx
```

with the w meaning "word," an allusion to the fact that earlier Intel chips had word size as 16 bits.

[30] When you build CGDB, make sure you do **make install**, not just **make**. As of this early verion of CGDB, in March 2003, this feature does not seem to work for assembly-language source code.

[31] Though not mandatory; recall Section 3.4.3.

The changes here are self-explanatory, but the non-change may seem odd at first: Why are we still using ECX, not CX? The answer is that even though we are accessing a 16-bit item, its address is still 32 bits.

The corresponding items for 8-bit operations are **.byte** in place of **.long**, **movb** instead of **movl**, **%ah** or **%al** (high and low bytes of AX) in place of EAX, etc.

If you wish to have conditional jumps based on 16-bit or 8-bit quantities, be sure to use **cmpw** or **cmpb**, respectively.

If the destination operand for a byte instruction is of word size, the CPU will allow us to "expand" the source byte to a word. For example,

```
movb $-1, %eax
```

will take the byte -1, i.e. 11111111, and convert it to the word -1, i.e. 11111111111111111111111111111111 which it will place in EAX. Note that the sign bit has been extended here, an operation known as **sign extension**.

By the way, though, in this situation the assembler will also give you a warning message, to make sure you really do want to have a word-size destination operand for your byte instruction. And the assembler will give an error message if you write something like, say,

```
movb %eax, %ebx
```

with its source operand being word-sized. The assembler has no choice but to give you the error message, since the Intel architecture has no machine instruction of this type; there is nothing the assembler can assemble the above line to.

3.12 Some More Addressing Modes

Following are examples of various addressing modes, using decrement and move for our example operations. The first four are ones we've seen earlier, and the rest are new. Here **x** is a label in the **.data** section.

```
decl %ebx           # register mode
decl (%ebx)         # indirect mode
decl x              # direct mode
movl $8888, %ebx    # source operand is in immediate mode

decl x(%ebx)        # indexed mode
decl 8(%ebx)        # based mode (really same as indexed)
decl (%eax,%ebx,4)  # scale-factor (my own name for it)
decl x(%eax,%ebx,4) # based scale-factor (my own name for it)
```

Let's look at the indexed mode first.

The expression **x(%ebx)** means that the operand is c(EBX) bytes past **x**. The name "indexed" comes from the fact that EBX here is playing the role of an array index.

For example, consider the C code

```
char x[100];
...
x[12] = 'g';
```

Since this is an array of type **char**, i.e. with each array element being stored in one byte, **x[12]** will be at the memory location 12 bytes past **x**. So, the C compiler might translate this to

```
movl $12, %ebx
movb $'g', x(%ebx)
```

Again, EBX is playing the role of the array index here, hence the name "indexed addressing mode." We say that EBX is the **index register** here. On Intel machines, almost any register can serve as an index register.

This won't work for **int** variables. Such variables occupy 4 bytes each. Thus our machine code would need an extra instruction in the **int** case. The C code

```
int x[100],i;  // suppose these are global
...
x[i] = 8888;
```

would be translated to something like

```
movl i, %ebx
imull $4, %ebx
movl $8888, x(%ebx)
```

(Here **imull** is a multiplication instruction, to be discussed in detail in a later unit.)

So, Intel has another more general mode, which I have called "scale-factor mode" above. Here is how it works:

In the scale-factor mode, the syntax is

```
(register1,register2,scale_factor)
```

and the operand address is register1+scale_factor*register2.

We can now avoid that extra **imull** instruction by using scale-factor mode:

```
movl $x, %eax
movl i, %ebx
movl $8888, (%eax, %ebx, 4)
```

Of course, that still entailed an extra step, to set EAX. But if we were doing a lot of accesses to the array **x**, we would need to set EAX only once, and thus come out ahead.[32]

By the way, if the instruction to be compiled had been

```
x[12] = 8888;
```

then plain old direct mode would have sufficed:

```
movl $8888, x+48
```

The point is that here the destination would be a fixed address, 48 bytes past **x** (remember that the address of **x** is fixed).

What about based mode? Indexed and based modes are actually identical, even though we think of them somewhat differently because we tend to use them in different contexts. In both cases, the syntax is

```
constant(register)
```

The action is then that the operand's location is constant+register contents. If the constant is, say, 5000 and the contents of the register is 240, then the operand will be at location 5240.

Note that in the case of **x(%ebx)**, the **x** is a constant, because **x** here means the address of **x**, which is a constant. Indeed, the expression **(x+200)(%ebx)** is also valid, meaning "EBX bytes past x+200," or if you prefer thinking of it this way, "EBX+200 bytes past x."

We tend to think of based mode a bit differently from indexed mode, though. We think of **x(%ebx)** as meaning "EBX bytes past x," while we think of **8(%ebx)** as meaning "8 bytes past [the place in memory pointed to by] EBX." The former is common in array contexts, as we have seen, while the latter occurs with stacks.

You will see full detail about stacks in our unit on subroutines later on. But for now, let's recall that local variables are stored on the stack. A given local variable may be stored, say, 8 bytes from the beginning of the stack. You will also learn that the ESP register points to the beginning of the stack. So, the local variable is indeed "8 bytes past ESP," explaining why based mode is so useful.

[32] Actually, the product $4 \times i$ must still be computed, but it is now done as part of that third **movl** instruction, rather than as an extra instruction. This can speed things up in various ways, and it makes our code cleaner—EBX really does contain the index, not 4 times the index.

3.13 GCC/Linker Operations

3.13.1 GCC As a Manager of the Compilation Process

Say our C program consists of source files **x.c** and **y.c**, and we compile as follows:

```
gcc -g x.c y.c
```

GCC[33] is basically a "wrapper" program, serving only a manager of the compilation process; it does not do the compilation itself. Instead, GCC will run other programs which actually do the work. We'll illustrate that here, using the **x.c/y.c** example throughout.

3.13.1.1 The C Preprocessor

First GCC will run the C preprocessor, **cpp**, which processes your **#define**, **#include** and other similar statements. For example:

```
% cat y.c
// y.c

#define ZZZZ 7

#include "w.h"

main()
{  int x,y,z;

   scanf("%d%d",&x,&y);
   x *= ZZZZ;
   y += ZZZZ;
   z = bigger(x,y);
   printf("%d\n",z);
}

% cat w.h
// w.h

#define bigger(a,b) (a > b) ? (a) : (b)

% cpp y.c
# 1 "y.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "y.c"
```

[33]The "English" name of this compiler is the GNU C Compiler, with acronym GCC, so it is customary to refer to it that way. The actual name of the file is **gcc**.

```
# 1 "w.h" 1
# 6 "y.c" 2

main()
{ int x,y,z;

   scanf("%d%d",&x,&y);
   x *= 7;
   y += 7;
   z = (x > y) ? (x) : (y);
   printf("%d\n",z);
}
```

The preprocessor's output, seen above, now can be used in the next stage:

3.13.1.2 The Actual Compiler, CC1, and the Assembler, AS

Next, GCC will start up another program, **cc1**, which does the actual code translation. Even **cc1** does not quite do a full compile to machine code. Instead, it compile only to assembly language, producing a file **x.s** GCC will then start up the assembler, AS (file name **as**), which translates **x.s** to a true machine language (1s and 0s) file **x.o**. The latter is called an **object file**.

Then GCC will go through the same process for **y.c**, producing **y.o**.

Finally, GCC will start up the **linker** program, LD (file name **ld**), which will splice together **x.o** and **y.o** into an executable file, **a.out**. GCC will also delete the intermediate **.s** and **.o** files it produced along the way, as they are no longer needed.

3.13.2 The Linker

3.13.2.1 What Is Linked?

Recall our example above of a C program consisting of two source files, **x.c** and **y.c**. We noted that the compilation command,

```
gcc -g x.c y.c
```

would temporarily produce two object files, **x.o** and **y.o**, and that GCC would call the linker program, LD to splice these two files together to form the executable file **a.out**.

Exactly what does the linker do? Well, suppose **main()** in **x.c** calls a function **f()** in **y.c**. When the compiler sees the call to **f()** in **x.c**, it will say, "Hmm...There is no **f()** in **x.c**, so I can't really translate the call, since I don't know the address of **f()**." So, the compiler will make a little note in **x.o** to the linker saying, "Dear

linker: When you link **x.o** with other **.o** files, you will need to determine where **f()** is in one of those files, and then finish translating the call to **f()** in **x.c**."[34]

Similarly, **x.c** may declare some global variable, say **z**, which the code in **y.c** references. The compiler will then leave a little note to the linker in **y.o**, etc.

So, the linker's job is to resolve issues like this before combining the **.o** files into one big executable file, **a.out**.

It doesn't matter whether our original source code was C/C++ or assembly language. Machine code is machine code, no matter what its source is, so the linker won't care which original language was the source of which **.o** file.

Though GCC invokes LD, we can run it directly too, which as we have seen is common in the case of writing assembly language code.[35]

3.13.2.2 Headers in Executable Files

Even if our source code consists of just one file (as opposed to, for instance, our example of **x.c** and **y.c** above), so that there is nothing to link, we must still invoke the linker to to produce an executable file. There are a couple of reasons for this.

First, in the case of C, you are linking in libraries without realizing it. If you run the **ldd** command on an executable which was compiled from C, you'll find that the executable uses the C library, **libc.so**. (More on **ldd** below.) At the very least, that library provides a place to begin execution, the label **_start** which we found in Section 3.4.1 is needed, and sets up your program's stack. That library also includes many basic functions, such as **printf()** and **scanf()**.

But beyond that, executable files consist not only of the actual machine code but also a **header**, a kind of introduction, at the start of the file. The header will state at what memory locations the various sections are to be loaded, how large the sections are, at what address execution is to begin, and so on. The linker will make these decisions, and place them in the header.

The are various standard formats for these headers. In Linux, the ELF format is used. If you are curious, the Linux **readelf** command will tell you what is in the header of any executable file. By running this command with, e.g., the **-s** option, you can find out the final addresses assigned to your labels by the linker.[36]

There are headers in object files as well; the Linux command **objdump** will display these for you.

[34] Or, in our GCC command line we may ask the linker to get other functions from libraries. See Section 3.13.2.3 below.

[35] We can also apply GCC to the assembly-language file. GCC will notice that the file name has a **.s** extension, and thus will invoke the assembler, AS.

[36] A more common command for this is **nm**. It's more general, as it does not apply only to ELF files, but it only gives symbol information.

3.13.2.3 Libraries

A library is a conglomeration of several object files, collected together in one convenient place. Libraries can be either **static** or **dynamic**, as explained below. In Unix, library names end with **.a** (static) or **.so** (dynamic), possibly followed by a version number.[37] The names also usually start with "lib," so that for example **libc.so.6** is version 6 of the C library.

When you need code from a static library, the linker physically places the code in with your machine code. In the dynamic case, though, the linker merely places a note in your machine code, which states which libraries are needed; at run time, the OS will do the actual linking then. Dynamic libraries save a lot of disk space and memory (at the slight cost of a bit of a delay in loading the program into memory at run time), since we only need a single copy of any given library. Here is an overview of how the libraries are created and used:

One creates a static library by applying the **ar** command to the given group of **.o** files, and then possibly running **ranlib**. For example:

```
% gcc -c x.c
% gcc -c y.c
% ar lib8888.a x.o y.o
% ranlib lib8888.a
```

Later, if someone wants to check what's inside a static library, one runs **ar** on the **.a** file, with the **t** option, e.g.

```
% ar t lib8888.a
```

To create a dynamic library, one needs to use the **-fPIC** option when compiling to produce the **.o** files, and then one compiles the library by using GCC's **-shared** option, e.g.

```
% gcc -g -c -fPIC x.c
% gcc -g -c -fPIC y.c
% gcc -shared -o lib8888.so x.o y.o
```

In systems that use ELF, you can check what's in a dynamic library by using **readelf**, e.g.

```
% readelf -s lib8888.so
```

When you link to a library, GCC's **-l** option can be used to state which library to link in. For example, if you have a C program **w.c** which calls **sqrt()**, which is in the math library **libm.so**, you would type

[37] You may have encountered dynamic libraries on Windows systems, which have the suffix **.dll**.

```
% gcc -g w.c -lm
```

The notation "-lxxx" means "link the library named **libxxx.a** or **libxxx.so**."

One point to note, though, is that the library you need may not be in the default directories **/usr/lib**, **/lib** and those listed in **/etc/ld.so.cache**. If for example your library **libqrs.so** is in the directory **/a/b/c** , you would type

```
% gcc -g w.c -lqrs -L/a/b/c
```

This is fine in the static case, but remember that in the dynamic case, the library is not actually acquired at link time. All that is put in our executable file is the name of the library, but NOT its location. In other words, in the dynamic case, the **-L/a/b/c** in the example above is used by the linker to verify that the library does exist, but this location is NOT recorded. When the program is actually run, the OS will still look only in the default directories. There are various ways to tell it to look at others. One of the ways is to specify **/a/b/c** within the Unix environment variable LD_LIBRARY_PATH, e.g.

```
% setenv LD_LIBRARY_PATH /a/b/c
```

If you need to know which dynamic libraries an executable file needs, run **ldd**, e.g.

```
% ldd a.out
```

Clearly this is a complex topic. If you ever need to know more than what I have in this introduction, plug something like "shared library tutorial" into a Web search engine.

3.14 Inline Assembly Code for C++

The C++ language includes an **asm** construct, which allows you to embed assembly language source code right there in the midst of your C++ code.[38]

This feature is useful, for instance, to get access to some of the fast features of the hardware. For example, say you wanted to make use of Intel's fast MOVS string copy instruction. You could write an assembly language subroutine using MOVS and then link it to your C++ program, but that would add the overhead of subroutine call/return. (More on this in our unit on subroutines.) Instead, you could write the MOVS code there in your C++ source file.

Here's a very short, overly simple example:

[38] This is, as far as I know, available in most C++ compilers. Both GCC and Microsoft's C++ compiler allow it.

```
// file a.c

int x;

main()

{  scanf("%d",&x);
   __asm__("pushl x");
}
```

After doing

```
gcc -S a.c
```

the file **a.s** will be

```
        .file   "a.c"
        .section        .rodata
.LC0:
        .string "%d"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $20, %esp
        movl    $x, 4(%esp)
        movl    $.LC0, (%esp)
        call    scanf
#APP
        pushl x
#NO_APP
        addl    $20, %esp
        popl    %ecx
```

Our assembly language is bracketed by APP and NO_APP, and sure enough, it is

```
pushl x
```

For an introduction to how to use this feature, see the tutorials on the Web; just plug "inline assembly tutorial" into Google. For instance, there is one at `http://www.opennet.ru/base/dev/gccasm.txt.html`.

3.15 "Linux Intel Assembly Language": Why "Intel"? Why "Linux"?

The title of this document is "Linux Intel Assembly Language"? Where do those qualifiers "Intel" and "Linux" come in?

First of all, the qualifier "Intel" refers to the fact, discussed earlier, that every CPU type has a different instruction set and register set. Machine code which runs on an Intel CPU will be rejected on a PowerPC CPU, and vice versa.

The "Linux" qualifier is a little more subtle. Suppose we have a C source file, **y.c**, and compile it twice on the same PC, once under Linux and then under Windows, producing executable files **y** and **y.exe**. Both files will contain Intel instructions. But for I/O and other OS services, the calls in **y** will be different from the calls in **y.exe**.

3.16 Viewing the Assembly Language Version of the Compiled Code

We will have often occasion to look at the assembly language which the compiler produces as its first step in translating C code. In the case of GCC we use the **-S** option to do this. For example, if you type

```
gcc -S y.c
```

an assembly language file **y.s** will be created, as it would ordinarily, but the compiler will stop at that point, not creating object or executable files. You could even then apply **as** to this file, producing **y.o**, and then run **ld** on **y.o** to create the same executable file **a.out**, though you would also have to link in the proper C library code file.[39]

3.17 String Operations

The STOS (STore String) family of instructions does an extremely fast copy of a given string to a range of consecutive memory locations, much faster than one could do with MOV instructions in a programmer-written loop.

For example, the **stosl** instruction copies the word in EAX to the memory word pointed to by the EDI register, and increments EDI by 4. If we add the **prefix**, **rep** ("repeat"), i.e. our line of assembly language is now

[39] In order to do the latter automatically, without having to know which file it is and where it is, use GCC to do the linking:
gcc y.o
GCC will call LD for you, also supplying the proper C library code file.

```
rep stosl
```

and then this is where the real power comes in. That single instruction effectively becomes the equivalent of a loop: The **stosl** instruction will be executed repeatedly, with ECX being decremented by 1 each time, until ECX reaches 0. This would mean that c(EAX) would be copied to a series of consecutive words in memory.

Note that the **rep** prefix is in effect an extra op code, prepended before the instruction itself. The instruction

```
stosl
```

translates to the machine code 0xab, but with **rep** prepended, the instruction is 0xabf3.[40]

EDI, EAX and ECX are wired-in operands for this instruction. The programmer has no option here, and thus they do not appear in the assembly code.

The way to remember EDI is that the D stands for "destination."

There is also the MOVS family, which copies one possibly very long string in memory to another place in memory. EDI again plays the destination role, i.e. points to the place to be copied to, and the ESI register points to the source string.[41]

Here is an example of STOS:

```
1  .data
2  x:
3      .space 20  # set up 5 words of space
4
5  .text
6
7  .globl _start
8
9  _start:
10       movl $x,%edi
11       movl $5,%ecx  # we will copy our string to 5 words
12       movl $0x12345678,%eax  # the string to be copied is 0x12345678
13       rep stosl
14 done:
```

Here is an example of MOVS, copying one string to another:

```
1  .data
2  x: .string "abcde"  # 5 characters plus a null
```

[40] The lower-address byte will be f3, and the higher-address byte will be ab. Recall that the C notation "0x" describes a bit string by saying what the (base-16) number would be if the string were representing an integer. Since we are on a little-endian machine, the 0x notation for this instruction would be 0xabf3.

[41] Again, the S here refers to "source."

```
3  y: .space 6
4
5  .text
6  .globl _start
7  _start:
8      movl $x, %esi
9      movl $y, %edi
10     movl $6, %ecx
11     rep movsb
12 done:
```

Again, you must use ESI, EDI and ECX for the source address, destination address and repeat count, respectively. No other registers may be used.

Warning: REP MOVS really is the equivalent (though much faster) of writing a loop to copy the characters from the source to the destination. An implication of this is that if the destination overlaps the source, you won't necessarily get a copy of the original source in the destination. If for instance the above code were

```
1  .data
2  x: .string "abcde"  # 5 characters plus a null
3  y: .space 9
4
5  .text
6  .globl _start
7  _start:
8      movl $x, %esi
9      movl %esi, %edi
10     addl $3, %edi
11     movl $6, %ecx
12     rep movsb
13 done:
```

then the resulting contents of the nine bytes starting at y would be ('a','b','c','a','b','c','a','b','c').

3.18 Useful Web Links

- Linux assembly language Web page: `http://linuxassembly.org/`
- full **as** manual: `http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html` (contains full list of directives, register names, etc.; op code names are same as Intel syntax, except for suffixes, e.g. ''l' in "movl")
- Intel2gas, converter from Intel syntax to AT&T and vice versa: `http://www.niksula.cs.hut.fi/~mtiihone/intel2gas/`
- There are many Web references for the Intel architecture. Just plug "Intel instruction set" into any Web search engine.

One such site is `http://www.penguin.cz/~literakl/intel/intel.html`.

For more detailed information, see the Intel manual, at `ftp://download.intel.com/design/Pentium4/manuals/24547108.pdf`. There is an index at the end.

Also, if you need to learn about a specific instruction, often you can get some examples by plugging the instruction name and the word *example* into Google or any other Web search engine. Use the family name for the instruction, e.g. MOV instead of **movl**, **movb** etc. This will ensure that your search will pick up both Intel-syntax-oriented and AT&T-syntax-oriented sites.

By the way, if you do get information on the Web which is Intel-syntax-oriented, use the **intel2gas** program, mentioned above, to convert it to AT&T.

- NASM assembler home page: `http://nasm.2y.net/` I don't use it myself, but it is useful in that it is usable on both Linux and Windows
- the ALD debugger: `http://ellipse.mcs.drexel.edu/ald.html`
- my tutorials on debugging, featuring my slide show, using **ddd**: `http://heather.cs.ucdavis.edu/~matloff/debug.html`
- Unix tutorial: `http://heather.cs.ucdavis.edu/~matloff/unix.html`
- Linux installation guide: `http://heather.cs.ucdavis.edu/~matloff/linux.html`

3.19 Top-Down Programming

Programming is really mind-boggling work. When one starts a large, complex program, it is really a daunting feeling. One may feel, "Gee, there is so much to do..." It is imperative that you deal with this by using the **top-down** approach to programming, also known as **stepwise refinement**. Here is what it is.

You start by writing **main()** (or in assembly language, **_start()**), and—this is key—making sure that it consists of no more than about a dozen lines of code.[42] Since most programs are much bigger than just 12 lines, this of course means that some, many most, of these 12 lines will be calls to functions (or in the assembly language case, calls to subroutines). It is important that you give the functions good names, in order to remind yourself what tasks the various functions will perform.

In a large, complex program, some of those functions will also have a lot to do, so you should impose upon yourself a 12-line on them too! In other words, some functions will themselves consist of calls to even more functions.

[42] I'm just using 12 lines as an example. You may prefer a smaller limit, especially when working at the assembly language level.

The point is that this is something easy and somewhat quick to do, something that you can do without feeling overwhelmed. Each 12-line module which you write is simple enough so that you can really focus your thoughts.

Chapter 4

More on Intel Arithmetic and Logic Operations

Note: We will assume our machine has a 32-bit word size throughout this chapter.

4.1 Instructions for Multiplication and Division

4.1.1 Multiplication

4.1.1.1 The IMUL Instruction

One of the Intel instructions for multiplication is **imull** ("IMUL long," i.e. for 32-bit operations). One form of IMUL is

```
imull src
```

where **src** is a 32-bit multiplier specified in register mode. The multiplicand is implicitly EAX,[1] and the product is stored in what is described in Intel literature as EDX:EAX. To see what this means, note that when we multiply two 32-bit quantities, we potentially get a 64-bit product. So, the Intel hardware places the upper 32 bits in EDX and the lower 32 bits in EAX.

[1] So, there is only one operand in this form of the instruction, as opposed to the two in the previous form.

4.1.1.2 Issues of Sign

Note that even if the product fits into 32 bits, thus into EAX, we still need to pay attention to EDX. Suppose for instance that the product happens to be -4. In 32-bit form, that is

```
11111111111111111111111111111100
```

while in 64-bit form it is

```
1111111111111111111111111111111111111111111111111111111111111100
```

In other words, the negative status of -4 is indicated by a 1 in the first of those 64 bits. The 64-bit number

```
0000000000000000000000000000000011111111111111111111111111111100
```

is positive. In fact, it has the value $2^{32} - 4$, which you can see because we wind backwards four times from

```
0000000000000000000000000000000100000000000000000000000000000000.
```

4.1.2 Division

4.1.2.1 The IDIV Instruction

For IDIV, one form is

```
idivl src
```

We place the dividend in EDX:EAX and the divisor in **src**, again with the latter being specified in register addressing mode. The quotient then comes out in EAX, and the remainder in EDX. If the quotient is too big for 32-bit storage, then an **exception**, i.e. an execution error will occur.

4.1.2.2 Issues of Sign

As noted for multiplication above, you must be careful with 64-bit quantities. Even if your dividend fits into 32 bits, it will be treated as a 64-bit number, with the high bits being in EDX—whether you put them there or not.

So, if your dividend is nonnegative, make sure to put 0 in EDX; otherwise put -1 there.

A compact and fast way to do this is to use the CWD instruction.

4.1.3 Example

Here is an example of IMUL/IDIV:

```
1  .text
2  .globl _start
3  _start:
4        movl $2, %ecx
5        movl $-1, %eax
6        imull %ecx
7        idivl %ecx
8  done: movl %ebx, %ebx
```

Let's use GDB to explore the behavior of these instructions:

```
1  gdb) r
2  Starting program: /www/matloff/public_html/50/PLN/a.out
3
4  Breakpoint 1, _start () at g.s:4
5  4             imull %ecx
6  Current language:  auto; currently asm
7  (gdb) p/x $eax
8  $1 = 0xffffffff
9  (gdb) p/x $edx
10 $2 = 0x0
11 (gdb) si
12 8             idivl %ecx
13 (gdb) p/x $eax
14 $3 = 0xfffffffe
15 (gdb) p/x $edx
16 $4 = 0xffffffff
17 (gdb) si
18 9       done: movl %ebx, %ebx
19 (gdb) p/x $eax
20 $5 = 0xffffffff
21 (gdb) p/x $edx
22 $6 = 0x0
```

We started with the 32-bit version of -1 in EAX, and 0 in EDX. Multiplying by 2 gives us -2, but IMUL puts its product in 64-bit form in the register pair EDX:EAX. GDB verified this for us. Then we divided by 2, getting -1, but since IDIV puts its result in 32-bit form in the single register EAX, EDX becomes 0.

Or, if we want to view the numbers as unsigned—remember, the nice thing about 2s complement is that we have our choice of interpretation—then our multiplying by 2 changed $2^{32} - 1$ (a number fitting in 32 bits) to $2^{33} - 2$ (a 64-bit number).

4.2 More on Carry and Overflow, and More Jump Instructions

In Chapter 3, we saw how to use instructions like JS, JNZ, etc. to do "if-then-else" tests. This works for simple cases with small or moderate-sized numbers. Unfortunately, it is more complicated than this, though, due to problems with the difference between signed and unsigned numbers. To see why, suppose for convenience that we are working with 1-byte quantities (other than convenience, there is nothing special about using only one byte here), and consider what happens with

```
cmpb %al, %bl
```

when c(AL) = 0x50 and c(BL) = 0xfb. Viewed as signed numbers, c(BL) is smaller, since $80 > -5$, but viewed as unsigned numbers, c(AL) is smaller, since $80 < 251$.

Recall that the hardware doesn't know whether we are considering our numbers to be signed or unsigned. So, for example, there is nothing in the hardware for the CMP instruction to distinguish between the signed and unsigned cases, nor is there such a capability in the ADD and SUB instructions.[2] However, the hardware does store relevant information in the Carry and Overflow flags, and this will allow us as programmers to handle the signed and unsigned cases differently on our own, as we will see below.

Before continuing, let's review at what the hardware does when adding two numbers. It will in any addition simply add the two numbers bit by bit sequentially from right to left, with carries, in grade-school fashion. The hardware is basically treating the numbers as unsigned, but remember, that's OK even if we are considering the numbers to be signed, because the nature of 2s complement storage makies it all work out.

When performing an ADD instruction, the hardware will set or clear the Carry flag (CF), according to whether a carry does occur out of the most significant bit. As noted in Chapter 3, we can programmatically check this by using the JC or JNC instructions. Similarly, the hardware will set or clear the Overflow flag (OF), under certain conditions which we will discuss shortly; our program can check this flag via the JO and JNO instructions.

The term **overflow**—as opposed to the Overflow flag—informally means that the result of an operation is too large to fit into the number of bits allocated to it. The definition of "too large" in turn depends on whether we are treating the numbers as signed or unsigned.

Again taking the 8-bit case as our example, suppose c(AL) = 0x50 and c(BL) = 0x50, i.e. both registers contain decimal 80. Their sum, 160, does fit into 8 bits if we are considering our numbers to be unsigned, but if we are thinking of them as signed, then "160" is really -96. So, in this example an overflow will not occur if the numbers are considered unsigned, but will occur if they are considered signed.

Clearly, if we are considering our numbers to be unsigned, overflow is indicated by the Carry Flag. But what about the signed case? The Carry Flag will not tell us (at least not directly) whether overflow has occurred.

[2] The MIPS chip, by contrast, does have, for instance, both ADD and ADDU instructions for addition. ADD causes an **exception**, i.e. execution error, upon overflow, while ADDU does nothing to report the problem. C/C++ compilers will use ADDU.

In other words, we'll need another flag to check overflow in the signed case, and that's the purpose of the Overflow Flag.

So, how did the Intel engineers design the circuitry which manages the Overflow Flag? Well, let's first note that there won't be an overflow problem if we are adding a nonnegative number to a negative one. The nonnegative number will be in the range 0 to +127, while the negative one will be between -128 and -1. Then the sum must necessarily be between -128 and +127, thus no overflow.

But a problem can occur if we are adding two positive numbers. They are basically 7 bits each (not counting the sign bit), so overflow will occur if they have an 8-bit sum. In such a case, there will be a carry out of the 7th bit, thus placing a 1 in the 8th bit—thus producing a negative number if we are considering everything signed. If you step through this reasoning in the case of adding two negative numbers, you'll see the problem arises if their sum is positive. In other words:

> In signed addition, overflow occurs if and only if the sum of two positive numbers is negative or the sum of two negative numbers is positive.

Knowing this, the developers of the Intel hardware designed the circuitry so that the Overflow flag would be set if it sees that there is a sign change of the nature described above, and clears the flag otherwise.

Note once again that the hardware doesn't know whether we are considering our numbers to be signed or unsigned. The developers of the hardware assumed that most people's typical usage would be signed, and thus they designed the Overflow flag accordingly. If we really are treating our numbers as unsigned, then we would have our program check the Overflow flag. But if we are treating them as unsigned, we would check the Carry flag.

What about subtraction? Recall that it is implemented via addition, i.e. x-y is effected by adding the 2s complement representation of -y to x, so we are really back to the addition case. But if we are using subtraction to set up a conditional jump—typically by using a CMP instruction—we need to worry about the sign flag too, as follows.

- If the Overflow flag is 0, we have nothing to worry about, and can simply look at the Sign flag to determine whether the subtraction resulted in a negative number.
- If the Overflow is 1, things are a little less simple. Recall that when overflow occurs in an addition (which, recall, is performed for the subtraction process), the sign of the sum is opposite to that of the two addends. In other words, an overflow addition changes signs. So, if the subtraction had resulted in a negative difference, which ordinarily would set the Sign flag, the overflow will make the Sign flag 0 instead. Conversely, if the difference were not negative, the overflow will make the Sign flag 1, falsely making the difference look negative.

From these considerations, we see that:

> In a subtraction of signed numbers x-y, y will be the larger of the two if and only if the OF $\neq$ SF.

For this reason, the developers of the Intel chip including the JL ("jump if less than") which jumps if OF $\neq$ SF. This is then the safe way to do less-than comparisons on signed numbers; JS is not safe unless you know your numbers are small enough that overflow won't occur. There are also others. Here is an incomplete list (here "dst" and "src" refer to the destination and source done, for example, in an immediately-previous CMP):

mnemonic	action	flags checked
JL	jump if dst $<$ src	SF $\neq$ OF
JLE	jump if dst $\leq$ src	SF = 1 or SF $\neq$ OF
JG	jump if dst $>$ src	ZF = 0 and SF = OF
JGE	jump if dst $\geq$ src	SF = OF

There are corresponding instructions for the unsigned case: JB ("jump if below"), JBE ("jump if below or equal"), JA ("jump if above"), and JAE ("jump if above or equal"),

For **imull**, if the product cannot fit into 32 bits and the high bits go into EDX, both CF and OF are set. (SF and ZF are undefined.) With **idivl**, all of these flages are undefined.

4.3 Logical Instructions

This section will illustrate one of the most important tools available to the assembly language programmer—data operations at the bit level. Keep these in mind in all your assembly language programming.

In Chapter 3, we saw how to access 16-bit and 8-bit operands. But what can we do in smaller cases, for example 1-bit operands? The answer is that we cannot do this in a manner directly analogous to the 16-bit and 8-bit cases. The reason for this is that although individual bytes have addresses, individual bits do not. Thus for example there is no 1-bit MOV instruction comparable to **movl** and **movb**.

However, we can access individual bits, or groups of bits, in an indirect manner by using the instructions introduced in this section.

A logical AND operation does the following: At each bit position in the two operands, the result is 1 if both operand bits are 1, and 0 otherwise.

For example,

```
      1011
AND   1101
-----------
      1001
```

In the leftmost bit position of the two operands here, we see two 1s, so there is a 1 in the result. In the second-to-the left position, there is a 0 in the first operand and 1 in the second, so there is a 0 in the result, etc.

The **andl** ("AND long") instruction performs an AND operation on the source and destination, placing the result back in the destination. It is important to note that that means that for every 1 bit in the source operand, every 0 in the source will cause the corresponding bit in the destination to become 0. In other words, **andl** is used to place 0s in desired spots within the destination while keeping the other destination bits unchanged.

Suppose for instance that we wish to put a 0 in bit position 1 (second from the right) in EAX, but keep all the other bits in EAX unchanged. We would use the instruction

```
andl $0xfffffffd, %eax
```

because 0xfffffffd has a 0 at bit position 2 and all 1s elsewhere.

Often an AND operation is used as a **mask**, which is a mechanism by which we can test for certain bits to have certain values. For example, suppose we wish to jump to a location **w** if bit positions 15 and 4 have the values 1 and 0, respectively, in ECX. We could use the following code to do this:

```
andl $0x00008004, %ecx
cmpl $0x00008000, %ecx
jz w
```

The point is that after the **andl** is executed, c(ECX) will equal 0x00008000 if and only if the original value in ECX had 1 and 0 in bit positions 15 an 4.

The & operator in C (the binary operator, not the "address of" operator) performs an AND operation, and in fact the compiler will probably translate a statement containing & into and **andl** instruction.

In an OR operation, if at least one of the two corresponding bits in the two operands is a 1, then the bit in the result is a 1, while that bit is a 0 otherwise.

The **orl** ("OR long") instruction performs an OR operation on the source and destination, placing the result back in the destination. Again, it is important to note that that means that for every bit in the source operand which is a 0, the corresponding bit in the destination remains unchanged, while every 1 in the source will cause the corresponding bit in the destination to become 1. In other words, **orl** is used to place 1s in desired spots within the destination while keeping the other destination bits unchanged.

So for example to set bit 2 to 1 in EAX, we would use the instruction

```
orl $0x00000004, %eax
```

The | operator in C performs a logical-or operation, and the compiler will usually translate it to **orl** or other instruction from the OR family.

Logical operations are quite prominent in the programming of device drivers. For example, the built-in speaker on a PC has a 1-byte **port** at address 0x61. The least-significant 2 bits must be set to 11 for the speaker to be on, 00 for off; the other bits contain other important information and must not be changed. I/O ports, as we will see later, are accessed on Intel machines via Intel's IN and OUT instructions.[3] The following code turns the PC speaker off and on:

```
1  # to turn speaker on:
2  # copy speaker port to AL register
3  inb $0x61,%al
4  # place 11 in last 2 bits of AL
5  orb $3,%al
6  # copy back to speaker port
7  outb %al,$0x61
8
9  # to turn speaker off:
10 # copy speaker port to AL register
11 inb $0x61,%al
12 # place 0s in last 2 bits of AL
13 andb $0xfc,%al
14 # copy back to speaker port
15 outb %al,$0x61
```

The NOT instruction (e.g. **notl**) changes all 1 bits to 0s and all 0 bits to 1s.

Each bit position in the result of the XOR ("exclusive or") instruction is equal to 1 if and only if exactly one of the operands has a 1 at the same bit position.[4] It is a classic way of zeroing-out a register using a very short instruction. For example,

```
xorl %ecx, %ecx
```

will make all bits of ECX equal to 0.

There are also **shift** operations, which move all bits in the operand to the left or right. The number of bit positions to shift can be specified either in an immediate constant, or in the CL register.[5]

For example,

```
shll $4, %eax
```

would move all bits in EAX leftward by 4 positions. Note that those which move past the left end are lost, and the bit positions vacated near the right end become 0s. So, for instance, if c(EAX) had been 0xd5fffff3, after the shift it would be 0x5fffff30.

[3] Accessing them requires status as privileged user.

[4] You can also think of it as adding two bits and taking the sum modulo 2.

[5] Recall that this is the lower 8 bits of ECX.

The **shrl** instruction does the same thing, but rightward.

Note that if we are considering the contents of the operand to be an unsigned integer, a left shift by 4 bits is equivalent to multiplication by $2^4 = 16$, and a right shift by 4 bits is equivalent to division by 16.

If on the other hand we are considering the operand to be a signed integer, we would use **sall** and **sarl**, since these preserve signs. It is not an issue for multiplication (except for overflow), but it makes a difference for division.

This difference is summarized in the comments in the following code:

```
1  # the bit string 0x80000000 represents 2,147,483,648 if viewed as an
2  # unsigned number, and -2,147,483,647 if viewed as signed
3  movl $0x80000000, %eax
4  movl %eax, %ebx
5  shrl $1, %eax  # c(EAX) becomes 0x40000000, i.e. 1,073,741,824
6  sarl $1, %ebx  # c(EBX) becomes 0xc0000000, i.e. -1,073,741,823
```

In C/C++, left and right shift operations are performed via $<<$ and $>>$. In the latter case, the compiler will translate to **shrl** if the operand is **unsigned**, and **sall** in the signed case.

4.4 Floating-Point

Modern Intel CPUs have a built-in Floating-Point unit, which is hardware to perform floating-point add, subtract and so on. Without this hardware, the operations must be performed in software. For example, consider the code

```
float u,v,w;
...
w = u + v;
```

With an Intel CPU, the compiler would implement the addition by generating an FADD ("floating-point add") instruction. If there were no such instruction, the compiler would have to generate a rather length amount of code in which one would first decompose **u** and **v** into their mantissa and exponent portions, would then change one of the addends in order to match exponent values, would then add the mantissas, etc. The resulting code would run much more slowly would a single FADD instruction.

Intel CPUs have special floating-point registers, each 80 bits wide, thus giving more accuracy than the standard 32-bit size in which C/C++ **float** variables are stored. (The variables are still stored in that 32-bit format, but the actual arithmetic is done in 80 bits.) In ATT&T syntax, the registers are named **%st(0)** (also called simply **%st**), **%st(1)**, **%st(2)** and so on. The "st" here stands for "stack," as the registers are indeed arranged as a **stack**.

We will learn about stacks in detail in Chapter 6, but the main point is that a stack is a "last-in, first-out" data structure, meaning that items are removed from the stack in reverse order from which they are inserted. For instance, say the stack is initially empty, and then we **push**, i.e. insert, the numbers 1.8, 302.5 and -29.5222 onto the stack. We say that -29.5222 is now at the **top** of the stack, followed by 302.5 and then 1.8. If we now **pop** the stack, that means removing the top element, -29.5222, so that the new top is 302.5, with 1.8 next. If we pop again, the stack consists only of 1.8. If we now push 168.8888, then that latter value is on top, with 1.8 next.

Here is some sample code:

```
1  .data
2  x:     .float 1.625
3  y:     .float 0.0
4  point25:
5         .float 0.25
6  .text
7  .globl _start
8  _start:
9     flds x  # push x onto f.p. stack; x now on top
10    flds point25  # push 0.25; 0.25 now on top, followed by x
11    faddp  # add the top 2 elts. of the stack, replace 2nd by sum,
12           # then pop once; x+0.25 now on top
13    fstps y  # pop the stack and store the popped value in y
```

Here we see a new assembler directive, **.float**. This tells the assembler to arrange things so that when this program is run, the initial value at **x** in memory will be the (32-bit) representation of the number 1.625. From Chapter 1, we know that this is the bit pattern 0x3fd00000. In fact, we would get the same result from

```
x:     .long 0x3fd00000
```

and would even save the assembler some work in the process, since it would be spared the effort of converting 1.625.[6]

There are two **flds** instructions. They are from the FLD ("floating-point load") Intel family; the 's' here means "stack," just as 'l' means "long" in **addl** from the ADD family.

Read the comments in the code before continuing.

OK, let's use GDB to learn more about this code:

```
(gdb) l
1       .data
2       x:     .float 1.625
```

[6] That effort would be the same as what **scanf()** would go through with **%f** format: It would count the digits after the decimal point, finding the number to be 3. It would then convert the characters '1', '6' etc. to 1, 6 and so on, and thus form the numbers 1 and 625. Finally it would do something like `a = 1 + 625.0/1000.0`.

```
3       y:      .float 0.0
4       point25:
5               .float 0.25
6       .text
7       .globl _start
8       _start:
9          flds x  # push x onto f.p. stack
10         flds point25  # push 0.25
(gdb)
11         faddp
12         fstps y
(gdb) b 12
Breakpoint 1 at 0x8048082: file fp.s, line 12.
(gdb) r
Starting program:
/fandrhome/matloff/public_html/matloff/public_html/50/PLN/fp

Breakpoint 1, _start () at fp.s:12
12         fstps y
Current language:  auto; currently asm
(gdb) nexti
0x08048088 in ?? ()
(gdb) p/f y
$1 = 1.875
(gdb) p/x y
$2 = 0x3ff00000
```

Nothing new here, except that we've used GDB's **p** ("print") command with **f** format. Obviously, the latter is similar to **%f** for **printf()**. We see that **y** comes out to 1.875, as expected. We also print out the value of **y** as a bit string, which turns out to be 0x3ff00000. You should check that that is indeed the IEEE format representation of 1.875.

Of course, there are also the obvious instruction families such as FSUB for subtraction, FMUL for multiplication and so on. But there are also instruction families that implement transcendental function operations, such as FSIN for sine, FSQRT for square root, etc. Again, by implementing these operations in hardware, we can get large speedups in C/C++ programs that make heavy use of **float** variables, such as programs that do graphics, games, scientific computation and so on.

The Intel hardware also gives the programmer the ability to have fine control over issues such as rounding, via special instructions that control **floating-point status registers**.

Chapter 5

Introduction to Intel Machine Language (and More On Linking)

5.1 Overview

This document serves as a brief introduction to Intel machine language. It assumes some familiarity with Intel assembly language (using the AT&T syntax).

5.2 Relation of Assembly Language to Machine Language

Assembly language (AL) is essentially identical to machine language (ML). The only difference is that ML expresses any machine instruction using the 1s and 0s on which the machine operates, while AL expresses the same instruction using English-like abbreviations for specific bit patterns.

Consider for instance a register-to-register "move" operation, which copies the contents of one register to another.[1] The bit pattern which signals this, called an **op code**, is 1000100111. The circuitry is set up so that when the CPU sees 1000100111 in the first ten bits of an instruction, it knows it must do a register-to-register move.[2]

As humans we get tired of writing long strings of 1s and 0s, and it would also be burdensome to have to remember which ones signify which op codes. As an alternative, we could hire a clerk. We would say to

[1] The term "move" is thus a misnomer, though by now entrenched in computer jargon.

[2] What do we mean by the "first" 10 bits? An Intel instruction can be several bytes in length. When we speak of the "first" byte of an instruction, we mean the one contained in the lowest-numbered memory address. When we speak of the "first" bit within a byte, we mean the most-significant bit within that byte.

the clerk, "Clerk, every time you see me write **movl**, you enter 1000100111." The name **movl** is a lot easier for us to remember than 1000100111, so would be a great convenience for us (though the poor clerk would have to deal with these things).

The circuity in the CPU is also set up to know that register-to-register "move" instructions, i.e. instructions whose first bits are 1000100111, are two bytes long.[3] Also, the circuity is set up to know that the source and destination operands of the move—which register is to be copied to which other register—are stored in the remaining six bits in this instruction: The eleventh, twelth and thirteenth bits specify the source register, and the fourteenth, fifteenth and sixteenth indicate the destination register, according to the following codes:

EAX	000
EBX	011
ECX	001
EDX	010

(The Intel CPU also has a number of other registers, but we will not list their codes here.)

Again, instead of having to remember the code for say, EBX, we would tell the clerk "Whenever I say %ebx, I mean 011," and so on. Well, our "clerk" is the assembler, e.g. **as** in standard Unix systems. We use a text editor to type those same abbreviations into a file, and the assembler converts them to 1s and 0s just as a clerk would.

Remember, though, that we the programmer are still in full control of which instructions (i.e. which operations and operands) to specify, in contrast to a compiler. If for example we type the line

```
movl %eax,%ebx
```

into a file, say **x.s**, we know the assembler will convert it to the machine instruction which does this operation, which we will see below happens to be 1000100111000011, i.e. 0x89c3. By contrast, if we run

```
y = x + 3;
```

through a compiler, we have no idea what machine instructions the compiler will generate from it. Indeed, different compilers would likely generate different machine instructions.

The assembler places the machine code it generates, e.g. 1000100111000011 above, into the object file, **x.o**.

[3] Note that that means that the circuity will advance the Program Counter register by 2, so as to prepare for the fetch of the instruction which follows the move instruction.

5.3 Example Program

5.3.1 The Code

We ran an assembly language source file, **Total.s**, through the **as** assembler with the **-a** option, the latter requesting the assembler to print out for us the machine code it produces, side-by-side with our original assembly language. Here is the output:

```
GAS LISTING Total.s page 1


   1
   2              # finds the sum of the elements of an array
   3
   4              # assembly/linking Instructions:
   5
   6              # as -a --gstabs -o total.o Total.s
   7              # ld -o total total.o
   8
   9              .data  # start of data section
  10
  11              x:
  12 0000 01000000       .long   1
  13 0004 05000000       .long   5
  14 0008 02000000       .long   2
  15 000c 12000000       .long   18
  16
  17              sum:
  18 0010 00000000       .long 0
  19
  20              .text  # start of code section
  21
  22              .globl _start
  23              _start:
  24 0000 B8040000       movl $4, %eax  # EAX will serve as a counter for
  24      00
  25                                    # number of words left
  26 0005 BB000000       movl $0, %ebx  # EBX will store the sum
  26      00
  27 000a B9000000       movl $x, %ecx  # ECX will point to the current
  27      00
  28                                    # element to be summed
  29 000f 0319     top:  addl (%ecx), %ebx
  30 0011 83C104         addl $4, %ecx  # move pointer to next element
  31 0014 48             decl %eax      # decrement counter
  32 0015 75F8           jnz top        # if counter not 0, loop again
  33 0017 891D1000 done: movl %ebx, sum # done, store result in "sum"
  33      0000
  34 001d 8D7600

DEFINED SYMBOLS
             Total.s:11     .data:00000000 x
             Total.s:17     .data:00000010 sum
             Total.s:23     .text:00000000 _start
             Total.s:29     .text:0000000f top
```

```
             Total.s:33     .text:00000017 done

NO UNDEFINED SYMBOLS
```

5.3.2 Feedback from the Assembler

Each line in this output is organized as follows:

Display Line Number	Memory Offset	Contents	Assembly Source Line

The **offset** is a relative memory address, i.e. the distance from the given item to the beginning of its section (**.data**, **.text**, etc.).

Here the clerk is saying to us, "Boss, here is what I did with that assembly-language source line of yours, which I've shown here in the fourth field. I translated it to the coding you see in the third field, and I arranged for this coding to go into the offset I've listed in the second field. For your convenience, I've also shown the line number from your source file in the first field."

What does the word *offset* mean? For instance, look at Line 18. It says that the word which we labeled **sum** in our source code will be located 0x10 = 16 bytes from the beginning of the **.data** section.

5.3.3 A Few Instruction Formats

The Contents field shows us the bytes produced by the assembler from our source code, *displayed byte-by-byte, in order of increasing address*. Note the effect of little-endian-ness, for instance in Line 13. That line says that our assembly code

```
.long 5
```

produced the sequence of bytes 05 00 00 00. That makes sense, right? In the number 00000005, the least significant byte is 05 (recall that a byte is two hex digits), so it will have the lowest address, and thus it will be displayed first.

The actual address of this word will be determined when the program is linked by **ld**. The latter will choose a place in memory at which to begin the **.data** section, and that will determine everything else goes. If that section begins at, say, 0x2020, then for instance the data corresponding to Line 15 will be at case 0x2020+0x0c = 0x202c.

Note that the addresses assigned to these items in the **.data** section of our source file are contiguous and follow the same order in which we "declared" them in the source file.

Now, let's learn a few more Intel machine-language formats, to add to the register-to-register move we saw above. First, let's consider the one on Line 24.

This instruction is an immediate-to-register move, where the word "immediate" means "the source operand is a constant which appears in the instruction." Here the constant is 4; it is a 32-bit integer occupying the last four bytes of this five-byte instruction.

This type of instruction is coded by its first five bits being 10111. The next three bits code the destination register, in this case being 000. So the first byte of the instruction is 10111000, i.e. 0xb8, followed by the constant 04000000 (again, note the effect of Intel CPUs being little-endian). We will describe the format for immediate-to-register move as 10111DDDIMM4, where 10111 is the op code, DDD denotes the destination register and IMM4 denotes a 4-byte immediate constant.[4]

Note also on Line 24 that the offset-field count has been reset back to 0000 again, since we are now in a new section (**.text**).

Lines 26 and 27 are similar, but one aspect must be noted in the latter. The immediate constant here, **$x**, is the address of **x**. But all the assembler knows at this point is that later, when the program is linked, **x** will be 0 bytes from the start of the **.data** section. If **ld** allocates the **.data** section at, say, 0x2000, then **x** will be at that address. So, ideally the immediate constant placed by the assembler into the machine instruction here would be 0x2000 (or wherever else the **.data** section ends up). But, since this value is unknown at assembly time, the assembler just puts the offset, 0, in the instruction on a temporary basis, to be resolved later by the linker.

In Line 29 we see an indirect-to-register add. Its op code is 0000001100, followed by three bits for the destination register and then three bits for the source register (the latter being used in indirect mode). We will state this notationally as 0000001100DDDSSS. An add which is register-to-indirect, e.g. **addl %ecx,(%ebx)**, has the format 0000000100SSSDDD.

Of course, the choice of the codes for operations, registers and addressing modes is arbitrary, decided by the hardware designers. Typically it is chosen according to two considerations:

- Instructions which are expected to be more commonly used should have shorter codes, in order to conserve memory and disk space.

- The circuitry may be easier to design, or made faster, with some codes.

In Line 31 we see a register decrement operation. A little experimentation—changing the register to EBX and then observing how the machine code changes—reveals that the format for this kind of instruction is 01001DDD, where DDD is the destination register.

[4] We describe our earlier register-to-register move format as 1000100111SSSDDD.

5.3.4 Format and Operation of Jump Instructions

There is quite a bit to discuss on Line 32. The format for this jump-if-Zero-flag-not-set instruction is 01110101IMM1, where 01110101 is the op code and IMM1 is a 1-byte immediate signed constant which indicates the **target** of the jump, i.e. where to jump to if the Zero flag is not set. The constant is actually the (signed) distance to the target, as follows.

Recall that any computer has a register whose generic name is the Program Counter (PC) which points to the next instruction to be executed. Let's say that the **.text** part of the program starts at 0x300ba. Then the instruction shown here in Line 32 will be at 0x300ba+0x15 = 0x300cf, and our target (shown in Line 29) will be at address 0x300ba+0xf = 0x300c9.

Recall also that as soon as an instruction is fetched for execution, the PC, which had been pointing to that instruction, is incremented to point to the next instruction. In the case of the instruction in Line 32, here is how the incrementing occurred: The PC had contained 0x300cf, and the CPU fetched the instruction from that location. Then the CPU, seeing that the instruction begins with 01110101, realizes that this is a JNZ instruction, thus 2 bytes in length. So, the CPU increments the PC by 2, to point to the instruction in Line 33, at 0x300cf+2 = 0x300d1.

Now, to execute the JNZ instruction, the CPU checks the Zero flag. If the flag is not set, the CPU adds the IMM1 part of the JNZ instruction to the current PC value, treating IMM1 as an 8-bit 2s complement number. As such, 0xf8 = -8, so the sum in question will be 0x300d1-8 = 0x300c9. This latter quantity is exactly what we want, the address of our target. The CPU puts this number in the PC, which means that the next instruction fetch will be done from location 0x300c9—in other words, we jump to the line we had named **top** in our assembly language source file, just as desired.

Of course, the assembler knows that the CPU will add IMM1 to the current PC value, and thus sets IMM1 accordingly. Specifically, the assembler computes IMM1 to equal the distance from the instruction following the JNZ instruction to the target instruction.[5]

IMM1 is just 8 bits. If the distance to the target is greater than what can be expressed as an 8-bit 2s complement number, the Intel instruction set does have an alternate version of JNZ with format 0000111110000101IMM4.

Keep in mind that the circuitry which implements JNZ will be quite simple. It will check the Zero Flag, and if that flag is 1, the circuitry will then add the second byte of the instruction (-8 in our example here) to the EIP register. That's all!

[5]This implies that it doesn't matter that the execution-time address of **top** is unknown at assembly time, unlike the case in Line 27, where the lack of knowledge of the execution-time address of **x** caused a problem.

5.3.5 Other Issues

The assembler goes through the **.s** source file line by line, translating each line to the corresponding machine code. By the time the assembler sees this JNZ instruction, it has already seen the line labeled **top**. Thus it knows the offset of **top** within the **.text** section, and can compute the distance to that target, as we saw here. But what if the jump target were to come after this line with JNZ? In this case, the assembler won't know the offset of the target yet, and thus will not be able to completely translate the instruction yet. Instead, the assembler will make a note to itself, saying that it will come back and finish this line after it finishes its first pass through the **.s** file. So, assemblers are **two-pass**, going through the **.s** file twice.

In doing the computation 0x300d1-8, the CPU must first make the two quantities compatible in terms of size. So, the CPU will do a **sign extension**, extending the 8-bit version of -8, i.e. 0xf8, to the 32-bit version, i.e. 0xfffffff8. So, technically, the CPU is performing 0x000300d1+0xfffffff8, which does indeed come out to 0x300c9.

The formats for some other conditional jump instructions are: JZ, 01110100IMM1; JS format 01111000IMM1; JNS 01111001IMM1; and so on. The unconditional jump JMP has format 11101011IMM1.

The Intel instruction set contains hundreds of instructions, and their formats become quite complex. We will not pursue this any further.

Note, though, that even in the few formats we've seen here, there is a large variation in instruction length. Some instructions are only one byte long, some are two bytes, and so on, with lengths ranging up to six bytes in the instructions here. This is in great contrast to RISC CPUs such as MIPS, in which all instructions are four bytes long. RISC architectures aim for uniformity of various kinds, which helps make a "lean and mean" machine.

5.4 It Really Is Just a Mechanical Process

As pointed out earlier, the assembler is just acting like a clerk, mechanically translating English-like codes to the proper bits. To illustrate this, note that we could "write" our own machine language in hex, right there in the middle of our assembly code! For instance, recall from our earlier example that the instruction **decl %eax** has the machine language 0x48. We could set that 0x48 ourselves:

```
... # same code as before, but not shown here
      movl $x, %ecx
top:  addl (%ecx), %ebx
      addl $4, %ecx
      .byte 0x48  # we write "directly" in machine language
      jnz top
done: movl %ebx, sum
```

With that **.byte** directive, we are simply saying to the assembler, "After you translate **addl $4, %ecx** to machine code, put a byte 0x48 right after that code." Since this is what the assembler would have done anyway in assembling **decl %eax**, we get exactly the same result. Try assembling and running the program with this change; you'll see that it works fine.

5.5 You Could Write an Assembler!

A common exercise in assembly language courses is to write a simple assembler. Even if you are not assigned to do so, it would be a very enriching exercise if you were to at least give some thought as to how you would do it. For example, just stick to the few instructions covered in this unit. Your job would be to write a program, say in C, which translates assembly code for the instructions and addressing modes covered in this unit to machine language.

5.6 A Look at the Final Product

The linker chooses addresses for the sections. We can determine those, for example, by running

```
% readelf -s tot
```

on our executable file **tot**. The relevant excerpt of the output is

```
     9: 08049094     0 NOTYPE  LOCAL  DEFAULT    2 x
    10: 080490a4     0 NOTYPE  LOCAL  DEFAULT    2 sum
    11: 08048083     0 NOTYPE  LOCAL  DEFAULT    1 top
    12: 0804808b     0 NOTYPE  LOCAL  DEFAULT    1 done
    13: 08048074     0 NOTYPE  GLOBAL DEFAULT    1 _start
```

So we see that **ld** has arranged things so that, when our program is loaded into memory at run time, the **.data** section will start at 0x08049094 and the **.text** section at 0x08048074.

We can use GDB to confirm that these addresses really hold at run time:

```
% gdb tot
(gdb) p/x &x
$1 = 0x8049094
(gdb) p/x &_start
$2 = 0x8048074
```

We can also confirm that the linker really did fix the temporary machine code the assembler had produced from

```
movl $x, %ecx
```

Recall that the assembler didn't know the address of **x**, since the location of the **.data** section would not be set until later, when the linker ran. So, the assembler left a note in the **.o** file, asking the linker to put in the real address. Let's check that it did:

```
(gdb) b 24
Breakpoint 1 at 0x804807e: file tot.s, line 24.
(gdb) r
Starting program: /root/50/tot
Breakpoint 1, _start () at tot.s:24
24            movl $x, %ecx  # ECX will point to the current
Current language:  auto; currently asm
(gdb) disassemble
Dump of assembler code for function _start:
0x08048074 <_start+0>:  mov    $0x4,%eax
0x08048079 <_start+5>:  mov    $0x0,%ebx
0x0804807e <_start+10>: mov    $0x8049094,%ecx
End of assembler dump.
(gdb) x/5b 0x0804807e
0x804807e <_start+10>:  0xb9    0x94    0x90    0x04    0x08
```

We see that the machine code for the instruction really does contain the actual address of **x**.

Chapter 6

Subroutines on Intel CPUs

6.1 Overview

Programming classes usually urge the students to use **top-down** and **modular** design in their coding. This in turn means using a lot of calls to **functions** or **methods** in C-like languages, which are called **subroutines** at the machine/assembly level.[1]

This document serves as a brief introduction to subroutine calls in Intel machine language. It assumes some familiarity with Intel 32-bit assembly language, using the AT&T syntax with Linux.

6.2 Stacks

Most CPU types base subroutine calls on the notion of a **stack**. An executing program will typically have an area of memory defined for use as a **stack**. Most CPUs have a special register called the **stack pointer** (SP in general, ESP on 32-bit Intel machines), which points to the **top** of the stack.

Note carefully that the stack is not a "special" area of memory, say with a "fence" around; wherever SP points, that is the stack. And that is true even if we aren't using a stack at all, in which case the "stack" is garbage.

Stacks typically grow toward memory address 0.[2] If an item is added—**pushed**—onto the stack, the SP is decremented to point to the word preceding the word it originally pointed to, i.e. the word with the next-lower address than that of the word SP had pointed to beforehand. Similarly SP is incremented if the item at the top of the stack is **popped**, i.e. removed. Thus the words following the top of the stack, i.e. with

[1] This latter term is also used in various high-level languages, such as FORTRAN and Perl.

[2] For this reason, we usually draw the stack with memory address 0 at the top of the picture, rather than the bottom.

numerically larger addresses, are considered the interior of the stack.

Say for example we wish to push the value 35 onto the stack. Intel assembly language code to do this would be, for instance,

```
subl $4,%esp  # expand the stack by one word
movl $35,(%esp)  # put 35 in the word which is the new top-of-stack
```

However, typically a CPU will include a special instruction to do pushes. On Intel machines, for example, we could do the above using just one instruction:[3]

```
push $35
```

(There is also a **pushl** instruction, for consistency with instructions like **movl**, but it is just another name for **push**. Note that this is because the items on the stack must be of word size anyway.)

By the way, instructions like

```
pushl w
```

where **w** is a label in the **.data** section, are legal, a rare exception to Intel's theme of not allowing direct memory-to-memory operations.

Similarly, if we wish to pop the stack and have the popped value placed in, say, ECX, we would write

```
pop %ecx
```

Note carefully that it is the *stack* which is popped, not the ECX register.

Keep in mind that a pop does not change memory. The item is still there; the only change is that it is not considered part of the stack anymore. And the stack pointer merely is a means for indicating what is considered part of the stack; by definition, the stack consists of the word pointed to by ESP plus all words at higher addresses than that. The popped item will still be there in memory until such time, if any, that another push is done, overwriting the item.

6.3 CALL, RET Instructions

The actual call to a subroutine is done via a CALL instruction. Execution of this instruction consists of two actions:

[3] This would be not only easier to program, but more importantly would also execute more quickly.

- The current value of the PC is pushed on the stack.
- The address of the subroutine is placed into the PC.

We say that the first of these two actions pushes the **return address** onto the stack, meaning the place that we wish to return to after we finish executing the subroutine.

Note that both of the actions above, like those of any instruction, are performed by the hardware.

For example, consider the code

```
call abc
addl %eax,%ebx
```

At first the PC is pointing to the **call** instruction. After the instruction is fetched from memory, the CPU, as usual, increments the PC to point to the next instruction, i.e. the **addl**. From the first bullet item above, we see that that latter instruction's address will be pushed onto the stack — which is good, because we do want to save that address, as it is the place we wish to return to when the subroutine is done. So, the terminology used is that the **call** instruction saves the return address on the stack.

What about the second bullet item above? Its effect is that we do a jump to the subroutine. So, what **call** does is save the return address on the stack and then start execution of the subroutine.

The very last instruction which the programmer puts into the subroutine will be **ret**, a return instruction. It pops the stack, and places that popped value into the PC. Since the return address had been earlier pushed onto the stack, we now pop it back off, and the return address will now be in the PC — so we are executing the instruction immediately following the **call**, such as the **addl** in the example above, just as desired.[4]

6.4 Arguments

Functions in C code usually have arguments (another term used is *parameters*, and that is also true in assembly language. The typical way the arguments are passed to the subroutine is again via the stack. We simply push the arguments onto the stack before the call, and then pop them back off after the subroutine is done.

For instance, in the example above, suppose the subroutine **abc** has two integer arguments, and in this particular instance they will have the values 3 and 12. Then the code above might look like

```
push $12
push $3
```

[4]This assumes we've been careful to first pop anything we pushed onto the stack subsequent to the call. More on this later.

```
call abc
pop %edx  # assumes EDX isn't saving some other data at this time
pop %edx
addl %eax,%ebx
```

The subroutine then accesses these arguments via the stack. Say for example **abc** needs to add the two arguments and place their sum in ECX. It could be written this way:

```
...
movl 4(%esp),%ecx
addl 8(%esp),%ecx
...
ret
```

In that **movl**, the source operand is 4 bytes past where SP is pointing to, i.e. the next-to-top element of the stack. In the call example above, this element will be the 3. (The top element of the stack is the return address, since it was the last item pushed.) The second **addl** picks up the 12.

Here's what the stack looks like at the time we execute that **movl**:

	argument 2
	argument 1
ESP →	saved return address
towards 0 ↓	

6.5 Ensuring Correct Access to the Stack

But wait a minute. What if the calling program above had also been storing something important in ECX? We must write **abc** to avoid a clash. So, we first save the old value of ECX—where else but the stack?—and later restore it. So, the beginning and end of **abc** might look like this:

```
push %ecx
...
movl 8(%esp),%ecx
addl 12(%esp),%ecx
... # ECX used here
pop %ecx
ret
```

Note that we had to replace the stack offsets 4 and 8 by 8 and 12, to adjust for the fact that one more item will be on the stack. No one but the programmer can watch out for this kind of thing. The assembler and hardware, for example, would have no way of knowing that we are wrong if we were to fail to make this adjustment; we would access the wrong part of the stack, and neither the assembler nor hardware would complain.

6.6 Cleaning Up the Stack

Also, note that just before the **ret**, we "clean up" the stack by popping it. This is important for several reasons:

- Whatever we push onto the stack, we should eventually pop, to avoid inordinate growth of the stack. For example, we may be using memory near 0 for something else, so if the stack keeps growing, it will eventually overlap that other data, overwriting it.
- We need to restore ECX to its state prior to the call to **abc**.
- We need to ensure that the return instance does return to the correct place.[5]

Note that in our call to **abc** above, we followed the call with some stack cleanup as well:

```
pop %edx  # assumes EDX isn't saving some other data at this time
pop %edx
```

We needed to remove the two arguments which we had pushed before the call, and these two pops do that.

The pop operation insists that the popped value be placed somewhere, so we use EDX in a "garbage can" role here, since we won't be using the popped values. Or, we could do it this way:

```
addl $8,%esp
```

This would be faster-executing and we wouldn't have to worry about EDX.

6.7 Full Examples

6.7.1 First Example

In the following modification of an example from our unit introducing assembly language, we again sum up elements of an array, but we handle initialization of the registers by a subroutine. We also allow starting the summmation in the middle of the array, by specifying the starting point as an argument to the subroutine, and allow specification that only a given number of elements be summed:

[5]The reader should ponder what would happen if the programmer here forgot to include that pop just before the return.

```
1  .data
2  x:
3        .long   1
4        .long   5
5        .long   2
6        .long   18
7        .long   8888
8        .long   168
9  n:    .long 3
10 sum:
11       .long 0
12
13 .text
14 # EAX will contain the number of loop iterations remaining
15 # EBX will contain the sum
16 # ECX will point to the current item to be summed
17
18 .globl _start
19 _start:
20       # push the arguments before call:  place to start summing, and
21       # number of items to be summed
22       push $x+4  # here we specify to start summing at the 5
23       push n  # here we specify how many to sum
24       call init
25       addl $8, $esp  # clean up the stack
26 top:  addl (%ecx), %ebx
27       addl $4, %ecx
28       decl %eax
29       jnz top
30 done: movl %ebx, sum
31
32 init:
33       movl $0, %ebx
34       # pick up arguments from the stack
35       mov 8(%esp),  %ecx
36       mov 4(%esp),  %eax
37       ret
```

Here is the output assembly listing:

```
GAS LISTING full.s                          page 1
 1                      .data
 2                      x:
 3 0000 01000000               .long   1
 4 0004 05000000               .long   5
 5 0008 02000000               .long   2
 6 000c 12000000               .long   18
 7 0010 B8220000               .long   8888
 8 0014 A8000000               .long   168
 9 0018 03000000         n:    .long 3
10                      sum:
11 001c 00000000               .long 0
12
13                      .text
14                      # EAX will contain the number of loop iterations remaining
15                      # EBX will contain the sum
```

```
16                      # ECX will point to the current item to be summed
17
18                      .globl _start
19                      _start:
20                            # push the arguments before call:  place to start summing, and
21                            # number of items to be summed
22 0000 68040000               push $x+4  # here we specify to start summing at the 5
22      00
23 0005 FF351800               push n  # here we specify how many to sum
23      0000
24 000b E8110000               call init
24      00
25 0010 83C408                 addl $8, %esp  # clean up the stack
26 0013 0319             top:  addl (%ecx), %ebx
27 0015 83C104                 addl $4, %ecx
28 0018 48                     decl %eax
29 0019 75F8                   jnz top
30 001b 891D1C00         done: movl %ebx, sum
30      0000
31
32                      init:
33 0021 BB000000               movl $0, %ebx
33      00
34                            # pick up arguments from the stack
35 0026 8B4C2408               mov 8(%esp),  %ecx
36 002a 8B442404               mov 4(%esp),  %eax
37 002e C3                     ret

DEFINED SYMBOLS
              full.s:2      .data:00000000 x
              full.s:9      .data:00000018 n
              full.s:10     .data:0000001c sum
              full.s:19     .text:00000000 _start
              full.s:32     .text:00000021 init
              full.s:26     .text:00000013 top
              full.s:30     .text:0000001b done

NO UNDEFINED SYMBOLS
```

Note from line 24 of the assembly output listing that the CALL instruction assembled to e811000000. That breaks down to an op code of e8 and a distance field of 11000000. The latter (after accounting for little-endianness) is the distance from the instruction after the CALL to the subroutine, which from lines 25 and 33 can be seen to be 0x0021-0x0010 = 0x11.[6]

To gain a more concrete understanding of how the stack is used in subroutine calls, let's use GDB to inspect what happens with the stack when this program is run:

```
% gdb a.out
GNU gdb Red Hat Linux (5.2.1-4)
```

[6]This is what is known for Intel machines as a **near call**. Back in the days when Intel CPUs could only access 64K segment of memory at a time, they needed "near" (within-segment) and "far" (inter-segment) calls, but with the flat 32-bit addressing model used today, this is outmoded, and everything is "near."

```
Copyright 2002 Free Software Foundation, Inc.
...
Breakpoint 1, _start () at SubArgsEx.s:24
24              call init
Current language:  auto; currently asm
(gdb) p/x $eip
$1 = 0x804807f
(gdb) p/x $esp
$2 = 0xbfffcec8
(gdb) si
33              movl $0, %ebx
(gdb) p/x $eip
$3 = 0x8048093
(gdb) p/x $esp
$4 = 0xbfffcec4
(gdb) x/3w $esp
0xbfffcec4:        0x08048084        0x00000003        0x080490a8
(gdb) p/x &x
$5 = 0x80490a4
```

So, the PC value changed from 0x804807f to 0x0x8048093, reflecting the fact that a CALL does do a jump.

The CALL also does a push (of the return address), and sure enough, the stack did expand by one word, as can be seen by the fact that ESP decreased by 4, from 0xbfffcec8 to 0xbfffcec4.

The stack should now have the return address at the top, followed by the value of **n** and the address of **x** plus 4. Let's check that return address. The PC value just before the call had been, according our GDB output above, 0x804807f, and since (as discovered above from the output of **as -a**) the CALL was a 5-byte instruction, the saved return address should be 0x804807f+5 = 0x8048084, which is indeed what we see on the top of the stack. By the way, you should be able to deduce the address at which the **.text** section begins.

Now look at the rest of our GDB session:

```
(gdb) b done
Breakpoint 2 at 0x804808d: file SubArgsEx.s, line 30.
(gdb) c
Continuing.

Breakpoint 2, done () at SubArgsEx.s:30
30        done: movl %ebx, sum
(gdb) si
33              movl $0, %ebx
(gdb) p sum
$6 = 25
(gdb) si
35              mov 8(%esp),  %ecx
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

After executing the instruction at **done**, I checked to see if **sum** was correct, which it was. At that point, I really should have stopped, since the program was indeed done. But I continued anyway, issuing another **si**

command to GDB. Did GDB refuse? Did "the little man inside the computer" refuse? Heck, no! Remember, it's just a dumb machine. The CPU has no idea that the instruction at **done** was the last instruction in the program. So, it keeps going, executing the next instruction in memory. That instruction is the MOV at the beginning of the subroutine **init**()! So, you can see that there is nothing "special" about a subroutine. There is no physical boundary between modules of code, in this case between the calling module and **init**(). It's all just a bunch of instructions.

6.7.2 Second Example

The next example speaks for itself, through the comments at the top of the file. By the way, when you read through it, make sure you understand what we are doing with the SHR instruction.

```
1  # the subroutine findfirst(v,w,b) finds the first instance of a value v in a
2  # block of w consecutive words of memory beginning at b, returning
3  # either the index of the word where v was found (0, 1, 2, ...) or -1 if
4  # v was not found; beginning with _start, we have a short test of the
5  # subroutine
6
7  .data  # data section
8  x:
9        .long   1
10       .long   5
11       .long   3
12       .long   168
13       .long   8888
14 .text  # code section
15 .globl _start  # required
16 _start:  # required to use this label unless special action taken
17       # push the arguments on the stack, then make the call
18       push $x+4  # start search at the 5
19       push $168  # search for 168 (deliberately out of order)
20       push $4  # search 4 words
21       call findfirst
22 done:
23       movl %edi, %edi  # dummy instruction for breakpoint
24 findfirst:
25       # finds first instance of a specified value in a block of words
26       # EBX will contain the value to be searched for
27       # ECX will contain the number of words to be searched
28       # EAX will point to the current word to search
29       # return value (EAX) will be index of the word found (-1 if not found)
30       # fetch the arguments from the stack
31       movl 4(%esp), %ebx
32       movl 8(%esp), %ecx
33       movl 12(%esp), %eax
34       movl %eax, %edx # save block start location
35       # top of loop; compare the current word to the search value
36 top:  cmpl (%eax), %ebx
37       jz found
38       decl %ecx  # decrement counter of number of words left to search
39       jz notthere  # if counter has reached 0, the search value isn't there
```

```
40        addl $4, %eax  # otherwise, go on to the next word
41        jmp top
42  found:
43        subl %edx, %eax  # get offset from start of block
44        shrl $2, %eax  # divide by 4, to convert from byte offset to index
45        ret
46  notthere:
47        movl $-1, %eax
48        ret
```

6.8 Interfacing C/C++ to Assembly Language

Programming in assembly language is very slow, tedious and unclear, so we try to avoid it, sticking to high-level languages such as C. But in some cases we need to write *part* of a program in assembly language, either because there is something which is highly machine-dependent,[7] or because we need extra program speed and this part of the program is the main time consumer.

A good example is Linux. Most of it is written in C, for convenience and portability across machines, but small portions are written in assembly language, to get access to certain specific features of the given hardware. When Linux is ported to a new type of hardware, these portions must be rewritten, but fortunately they are small.

At first it might seem "unnatural" to combine C and assembly language. But remember, both **.c** and **.s** files are translated to machine language, so we are just combining machine language with machine language, not unnatural at all.

6.8.1 Example

So, here we will see how we can interface C code to an assembly language subroutine. Here is our C code:

```
1  // TryAddOne.c, example of interfacing C to assembly language
2  // paired with AddOne.s, which contains the function addone()
3  // compile by assembling AddOne.s first, and then typing
4  //
5  //    gcc -g -o tryaddone TryAddOne.c AddOne.o
6  //
7  // to link the two .o files into an executable file tryaddone
8  // (recall the gcc invokes ld)
9
10 int x;
11
12 main()
```

[7] Note that to a large extent we can deal with machine-dependent aspects even from C. We can deal with the fact that different machines have different word sizes, for example, by using C's **sizeof()** construct. However, this is not the case for some other situations.

```
13
14 {  x = 7;
15    addone(&x);
16    printf("%d\n",x);  // should print out 8
17    exit(1);
18 }
19
```

I wrote the function **addone()** in assembly language. In order to do so, I needed to know how the C compiler was going to translate the call to **addone()** to machine/assembly language. In order to determine this, I compiled the C module with the -S option, which produces an assembly language version of the compiled code:

```
1  % gcc -S TryAddOne.c
2  % more TryAddOne.s
3          .file   "TryAddOne.c"
4          .version        "01.01"
5  gcc2_compiled.:
6                  .section        .rodata
7  .LC0:
8          .string "%d\n"
9  .text
10         .align 4
11 .globl main
12         .type    main,@function
13 main:
14         pushl   %ebp
15         movl    %esp, %ebp
16         subl    $8, %esp
17         movl    $7, x
18         subl    $12, %esp
19         pushl   $x
20         call    addone
21         addl    $16, %esp
22         subl    $8, %esp
23         pushl   x
24         pushl   $.LC0
25         call    printf
26         addl    $16, %esp
27         subl    $12, %esp
28         pushl   $1
29         call    exit
30 .Lfe1:
31         .size    main,.Lfe1-main
32         .comm   x,4,4
33         .ident  "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-85)"
34 %
```

There is quite a lot in that **.s** output file, but the part of interest to us here, for the purpose of writing **addone()**, is this:

```
        pushl   $x
        call    addone
```

This confirms what we expected, that the compiler would produce code that transmits the argument to **addone()** by pushing it onto the stack before the call.[8] So, we write **addone()** so as to get the argument from the stack:

```
1  # AddOne.s, example of interfacing C to assembly language
2  #
3  # paired with TryAddOne.c, which calls addone
4  #
5  # assemble by typing
6  #
7  #    as --gstabs -o AddOne.o AddOne.s
8  #
9
10 # note that we do not have a .data section here, and normally would not;
11 # but we could do so, and if we wanted that .data section to be visible
12 # from the calling C program, we would have .globl lines for whatever
13 # labels we have in that section
14
15 .text
16
17 # need .globl to make addone visible to ld
18 .globl addone
19
20 addone:
21
22    # will use EBX for temporary storage below, and since the calling
23    # module might have a value there, better save the latter on the stack
24    # and restore it when we leave
25    push %ebx
26
27    # at this point the old EBX is on the top of the stack, then the
28    # return address, then the argument, so the latter is at ESP+8
29
30    movl 8(%esp), %ebx
31
32    incl (%ebx)  # need the (), since the argument was an address
33
34    # restore value of EBX in the calling module
35    pop %ebx
36
37    ret
```

The comments here say we should save, and later restore, the EBX register contents as they were prior to the call. Actually, our **main()** here does not use EBX in this case (see the assembly language version above), but since we might in the future want to call **addone()** from another program, it's best practice to protect EBX as we have done here.

So, by the time we reach the **movl** instruction, the stack looks like this:

[8] Note, though, that the compiler also first expanded the stack by 12 bytes, for no apparent reason. More on this below.

```
                argument
                saved return address
ESP →           saved EBX value
towards 0 ↓
```

The **movl** instruction thus needs to go 8 bytes deep into the stack to pick up the argument, which was the address of the word to be incremented. Then the **incl** instruction does the incrementing via indirect addressing, again since the argument is an address.

The reader is urged to compile/assemble/link as indicated in the comments in the **.c** and **.s** files above, and then execute the program, to see that it does indeed work. Seeing it actually happen will increase your understanding!

Note carefully that **TryAddOne.c** and **AddOne.s** are not "programs" by themselves. They are each the source code for a portion of the same program, the executable file **tryaddone**. This is no different from the situation in which we have C source code in two **.c** files, compile them into two **.o** files, and then link to make a single executable.

6.8.2 Cleaning Up the Stack?

Let's check whether the compiler fulfilled its "civic responsibility " by cleaning up the stack after the call. In the **.s** file produced by **gcc -S** above, we saw this code:

```
subl    $12, %esp
pushl   $x
call    addone
addl    $16, %esp
subl    $8, %esp
```

This is a bit odd. Since we had only pushed one argument, i.e. one word, the natural cleanup would have been to then pop one word, as follows:

```
call    addone
addl    $4, %esp
```

(or use a **pop** instruction instead of the **addl**). In other words, we should shrink the stack by 4 bytes after the call. Yet the net effect of the code generated by the compiler is to shrink the stack by 16 - 8 = 8 bytes, not 4.

On the other hand, recall that with the instruction

```
subl    $12, %esp
```

the compiler expanded the stack by 12 bytes *too much* before the call. In other words, the overall effect of the call has been to expand the stack by 12 - (16-8) = 4 bytes!

Supposedly the GCC people designed things this way, to leave gaps between one call and the next on the stack. This has makes things a little safer, as it is harder for a return address to be accidentally overwritten.

6.8.3 More Segments

By the way, the compiler has used the **.comm** directive to set up storage for our global variable **x**. It asks for 4 bytes of space, aligned on an address which is a multiple of 4. The linker will later set things up so that the variable **x** will be stored in the **.bss** section, which is like the **.data** section except that the data is unitialized. Recall that in our program above, the declaration of **x**, a global variable, was simply

```
int x;
```

If instead the declaration had been

```
int x = 28;
```

then **x** would be in the **.data** section.

Also, the label **.LC0** is in yet another kind of section, **.rodata** ("read-only data").

The reader is strongly encouraged to run the Unix **nm** command on any executable file, say the compiled version of the C program here. In the output of that command, symbols (names of functions and global variables) are marked as T, D, B, R and so on, indicating that the item is in the **.text** section, **.data** section, **.bss** section, **.rodata** section, etc., and the addresses assigned to them by the linker are shown.

In Linux, the stack section begins at address 0xbfffffff, and as mentioned, grows toward 0. The **heap**, from which space is allocated dynamically when your program calls **malloc()** or invokes the C++ operator **new**, starts at 0xbffff000 and grows away from 0.

6.8.4 Multiple Arguments

What if **addone()** were to have two arguments?[9] A look at the output of the compilation of the call to **printf()** above shows that arguments are pushed in reverse order, in this case the second before the first. So, if **addone()** were to have two arguments, we would have to write the code for the function accordingly, making use of the fact that the first argument will be closer to the top of the stack than the second.

[9]Note that when the C compiler compiles **TryAddOne.c** above, it does not know how many arguments **addone()** really has, since **addone()** is in a separate source file. (The compiler won't even know whether that source file is C or assembly code.) It simply knows how many arguments we have used in this call, which need not be the same.

6.8.5 Nonvoid Return Values

The above discussion presumes that return value for the assembly language function is **void**. If this is not the case, then the assembly language function must place the return value in the EAX register. This is because the C compiler will place code after the call to the function which picks up the return value from EAX.

That assumes that the return value fits in EAX, which is the case for integer, character or pointer values. It is not the case for the type **long long** in GCC, implemented on 32-bit Intel machines in 8 bytes. If a function has type **long long**, GCC will return the value in the EDX:EAX pair, similar to the IMUL case.

The case of **float** return values is more complicated. The Intel chip has separate registers and stack for floating-point operations. See our unit on arithmetic and logic.

6.8.6 Calling C and the C Library from Assembly Language

The same principles apply if one has a C function which one wishes to call from assembly language. We merely have to take into account the order of parameters, etc., verifying by running **gcc -S** on the C function.

A bit more care must be taken if we wish to call a C library function, e.g. **printf()**, from assembly language. Your call is the same, of course, but the question is how to do the linking. The easiest way to do this is to actually use GCC, because it will automatically handle linking in the C library, etc.

Here is an example, again from our first assembly language example of summing up four array elements:

```
1  .data
2
3  x:
4        .long   1
5        .long   5
6        .long   2
7        .long   18
8
9  sum:
10       .long 0
11
12 fmt:  .string "%d\n"
13
14 .text
15
16 .globl main
17 main:
18       movl $4, %eax  # EAX will serve as a counter for
19                      # the number of words left to be summed
20       movl $0, %ebx  # EBX will store the sum
21       movl $x, %ecx  # ECX will point to the current
22                      # element to be summed
23 top:  addl (%ecx), %ebx
24       addl $4, %ecx  # move pointer to next element
25       decl %eax  # decrement counter
```

```
26        jnz top  # if counter not 0, then loop again
27  printsum:
28        pushl %ebx
29        pushl $fmt
30        call printf
31  done: movl %eax, %eax
```

We wanted to print out the sum which was in EBX. We know that **printf()** has as its first parameter the output format, which is a character string, with the other parameters being the items to print. We only have one such item here, EBX. So we push it, then push the address of the format string, then call.

We then run it through GCC.

```
gcc -g -o sum sum.s
```

Given that we have chosen to use GCC (which, recall, we did in order to avoid having to learn where the C library is, etc.), this forced us to chose **main** as the label for the entry point in our program, instead of **_start**. This is because GCC will link in the C startup library. The label **_start** is actually in that library, and when our program here, or any other C program, begins execution, it actually will be at that label. The code at that label will do some initialization and then will call **main**. So, we had better have a label **main** in our code!

Note that the C library function that you call may use EAX, ECX and EDX! For example, most C library functions have return values, typically success codes, and of course they are returned in EAX. Recall from Section 6.8.8.1 that if you write a C function, you don't have to worry about the calling module having "live" values in EAX, ECX and EDX—that is, if the calling module is in C. Here it isn't.

6.8.7 Local Variables

Not only does the compiler use the stack for storage of arguments, it also stores local variables there. Consider for example

```
int sum(int *x, int n)
{  int i=0,s=0;

   for (i = 0; i < n; i++)
      s += x[i];
   return s;
}
```

As mentioned in a previous unit, the locals will be stored in "reverse" order: The two variables will be in adjacent words of memory, but the first one, i.e. the one at the lower address, will be **s**, not **i**. As you will see below, this will be done by in essence pushing **i** and then pushing **s** onto the stack.

Let's explore this by viewing the compiled code (in assembly language form) for this by running **gcc -S**. Here is an excerpt from the output:[10]

```
...
sum:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $0, -8(%ebp)
        movl    $0, -4(%ebp)
```

You can see that the compiler first put in code to copy ESP to EBP, a standard operation which is explained below in Section 6.8.8, and then put in code to expand the stack by 8 bytes for the two local variables. set both locals to 0.[11] Note that the local variables are not really "pushed" onto the stack; there is simply room made for them on the stack. Note also that later in the compiled code for **sum()** (not shown here), the compiler needed to insert code to pop off those local variables from the stack before the **ret** instruction; without this, the **ret** would try to jump to the place pointed to by the first local variable—total nonsense.

6.8.8 Use of EBP

6.8.8.1 GCC Calling Convention

Recall that in our unit on assembly language programming, we mentioned that you should not use ESP for general storage if you are using subroutines. It should now be clear why we noted that restriction. Now, here is a new restriction: If you are interfacing assembly language to C/C++, you should avoid using the EBP register for general storage. In this section, we'll see why.

Note the first three instructions in the implementation of **sum()** above:

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
```

This **prologue** is standard for C compilers on Intel machines. The old value of EBP is pushed on the stack; the current value of ESP is saved in EBP; and the stack is expanded (by decreasing ESP) to allocate space for the locals.

If you use GCC, then GCC will make sure that the calling module will not have any "live" values in EAX, ECX or EDX. So, the called module need not save the values in any of these registers. But if the called module uses ESI, EDI or EBX, the called module must save these in the prologue too.

[10] Different versions of GCC may produce somewhat different code.

[11] Different versions of **gcc** may not produce the same code. Always run **gcc -S** before doing any interfacing of C to assembly language.

These are referred in the computer world as **calling conventions**, a set of guidelines to follow in writing subroutines for a given machine under a given compiler.

6.8.8.2 The Stack Frame for a Given Call

The portion of the stack which begins at the place pointed to by EBP immediately after execution of the prologue of any given C function **g()**, and ends at the place pointed to by ESP, is called the **stack frame** for **g()**. This is basically **g()**'s current portion of the stack. Let's see what this consists of. Let's say that **g()** had been called by **f()**.

From the prologue code, we can see that the beginning of **g()**'s frame will consist of the saved value that had been in EBP before the call to **g()**. Well, that was the address of the beginning of **f()**'s stack frame! So, we see that the first element in **g()**'s frame will be a pointer to **f()**'s frame (which of course is at higher addresses than **g()**'s).

We also see that **g()**'s stack frame will contain **g()**'s local variables. And if **g()** saves some other register values on the stack, to protect **f()**, those will be part of **g()**'s frame too.

During the time **g()** is executing, the end of **g()**'s stack frame is pointed to by ESP. However, if **g()** in turn makes a call to some other function, say **h()**, that function will also have a stack frame, at lower addresses than **g()**'s frame. However, As **g()** executes, this position may move. This can occur for instance, if **g()** does call another function, say **h()**. ESP will decrease due to expansion of the stack by the called function's arguments and return address, which are considered part of **g()**'s stack frame, not **h()**'s. Here is what **g()**'s stack frame (and a bit of **f()**'s and **h()**'s) will consist of, just after the CALL is executed but before **h()**'s prologue begins executing:

	address in **f()** to return to when **g()** is done (end of **f()**'s frame)
EBP →	address of **f()**'s stack frame (start of **g()**'s frame)
	g()'s first declared local variable
	g()'s second declared local variable
	...
	g()'s last declared local variable
	any stack space **g()** is currently using as "scratch area"
	last argument in **g()**'s call to **h()**
	...
	second argument in **g()**'s call to **h()**
	first argument in **g()**'s call to **h()**
ESP →	address in **g()** to return to when **h()** is done (end of **g()'s frame**)
	address of **g()**'s stack frame (start of **h()**'s frame)
towards 0 ↓	

Of course, after we execute **h()**'s prologue, ESP will change again, and will demarcate the end of **h()**'s stack

frame (and EBP will demarcate the beginning of that frame). Once **h()** finishes execution, the stack will shrink back again, and ESP will increase and demarcate the end of **g()**'s frame again.

6.8.8.3 The Stack Frames Are Chained

An implication of the structure depicted above is that in any function's stack frame, the first (i.e. highest-address) element will contain a pointer to the beginning of the caller's stack frame. In that sense, the beginning elements of the various stack frames form a linked list, enabling one to trace through the stack information in a chain of nested calls. And since we know that EBP points to the current frame, we can use that as our starting point in traversing this chain.

The authors of GDB made use of this fact when they implemented GDB's **bt** ("backtrace") command. Let's review that command. Consider the following example:

```
1  void h(int *w)
2  {  int z;
3     *w = 13 * *w;
4  }
5
6  int g(int u)
7  {  int v;
8     h(&u);
9     v = u + 12;
10    return v;
11 }
12
13 main()
14 {  int x,y;
15    x = 5;
16    y = g(x);
17 }
```

Let's execute it in GDB:

```
(gdb) b h
Breakpoint 1 at 0x804833a: file bt.c, line 3.
(gdb) r
Starting program: /root/50/a.out

Breakpoint 1, h (w=0xfef5d2ac) at bt.c:3
3          *w = 13 * *w;
(gdb) bt
#0  h (w=0xfef5d2ac) at bt.c:3
#1  0x08048360 in g (u=5) at bt.c:8
#2  0x0804839c in main () at bt.c:16
```

We are now in **h()** (shown as frame 0 in the **bt** output), having called it from location 0x08048360 in **g()** (frame 1), which in turn had been called from location 0x0804839c in **main()** (frame 2).

And GDB allows us to temporarily change our context to another frame, say frame 1, and take a look around:

```
(gdb) f 1
#1  0x08048360 in g (u=5) at bt.c:8
8           h(&u);
(gdb) p u
$1 = 5
(gdb) p v
$2 = 0
```

Make sure you understand how GDB—which, remember, is itself a program—is able to do this. It does it by inspecting the stack frames of the various functions, and the way it gets from one frame to another within the stack is by the fact that the first element in a function's stack frame is a pointer to the first element of the caller's stack frame.

By using the structure shown above, the compiler is ensuring that we will always be able to get to return addresses, arguments and so on of "ancestral" calls.

6.8.8.4 ENTER and LEAVE Instructions

We've noted before that after return from a function call, the caller should clean up the stack, i.e. remove the arguments it had pushed onto the stack before the call. Similarly, within the function itself there should be a cleanup, to remove the locals and make sure that EBP is adjusted properly. That means: restoring ESP and EBP to the values they had before the prologue:

```
movl %ebp, %esp
popl %ebp
```

Let's call this the "epilogue." The prologue and epilogue codes are some common that Intel included ENTER and LEAVE instructions in the chip to do them. The above prologue, a three-instruction sequence which set 8 bytes of space for the locals, would be done, for instance, by the single instruction

```
enter 8, 0
```

(The 0 is never used in flat memory mode.)

The two-instruction epilogue above can be done with the single instruction

```
leave
```

Currently GCC uses LEAVE but not ENTER. The latter actually turns out to be slower on modern machines than just using the original three-instruction sequence, so it isn't used anymore.

6.8.8.5 Application of Stack Frame Structure

The following code implements a "backtrace" like GDB's. A user program calls **getchain()**, which will return a list of the return addresses of all the current stack frames. It is required that the user program first call the function **initbase()**, which determines the address of **main()**'s stack frame and stores it in **base**. It is assumed that **getchain()** will not be called from **main()** itself.

The C prototype is

```
void getchain(int *chain, int *nchain)
```

with the space for the array **chain** and its length **nchain** provided by the caller.

In reading the code, keep in mind this picture of the stack after the three pushes are done:

	address of **nchain**
	address of **chain**
	saved return address from **getchain()** back to the caller
	saved EAX
	saved EBX
ESP →	saved ECX
towards 0 ↓	

```
1  .data
2  base:  # address of main()'s stack frame
3     .long 0
4
5  .text
6  .globl initbase, getchain
7  initbase:  # initializes base
8     movl %ebp, base
9     ret
10 getchain:
11    # register usage:
12    #    EAX will point to the current element of chain to be filled
13    #    EBX will point to the frame currently being examined
14    push %eax
15    push %ebx
16    push %ecx
17    movl 16(%esp), %eax
18    # put in the return address from this subroutine, getchain() as
19    # the first element of the chain
20    movl 12(%esp), %ecx
21    movl %ecx, (%eax)
22    addl $4, %eax
23    # EBP still points to the caller's frame, perfect since we don't want
24    # to include the frame for this subroutine
25    movl %ebp, %ebx
26 top:
```

```
27      # if this is main()'s frame, then leave
28      cmpl %ebx, base
29      jz done
30      # get return address and add it to chain
31      movl 4(%ebx), %ecx
32      movl %ecx, (%eax)
33      addl $4, %eax
34      # go to next frame
35      movl (%ebx), %ebx
36      jmp top
37  done:
38      # find nchain, by subtracting start of chain from EAX and adjusting
39      subl 16(%esp), %eax
40      shrl $2, %eax
41      movl 20(%esp), %ebx
42      movl %eax, (%ebx)
43      pop %ecx
44      pop %ebx
45      pop %eax
46      ret
```

Example of calls:

```
1   int chain[5],nchain;
2
3   void h(int *w)
4   {  int z;
5      int chain[10],nchain;
6      z = *w +28;
7      getchain(chain,&nchain);
8      *w = 13 * z;
9   }
10
11  int g(int u)
12  {  int v;
13     h(&u);
14     v = u + 12;
15     getchain(chain,&nchain);
16     return v;
17  }
18
19  main()
20  {  int x,y;
21     initbase();
22     x = 5;
23     y = g(x);
24  }
```

6.8.9 The LEA Instruction Family

The name of the LEA instruction family stands for Load Effective Address. Its action is to compute the memory address of the first operand, and store that address in the second.

For example, the instruction

```
leal    -4(%ebp), %eax
```

computes -4 + c(EBP) and places the result in EAX. If we had a single local variable **z** within a C function, the above instruction would be computing **&z** and placing it in EAX. The compiler often uses this technique.

6.8.10 The Function main() IS a Function, So It Too Has a Stack Frame

It's important to keep in mind that **main()** is a function too, so it has information stored on the stack. Recall that a typical declaration of **main()** is

```
int main(int argc, char **argv)
```

This clearly shows that **main()** is a function. Note that we've given **main()** an **int** return value, typically used as a success code, again illustrating the fact that **main()** is a function.

Let's take a closer look. When you compile your program, the compiler (actually linker) puts in some C library code which is used for startup. Just like your assembly language programs, there is a label there, **_start**, at which execution begins. The code there will prepare the stack, including the **argc** and **argv** arguments, and will then call **main()**.

Note by the way that we are not required to give these formal parameters the names **argc** and **argv**. It is merely customary. As a veteran C programmer, you know that you can name the formal parameters whatever you want. When any subroutine is called, the caller, in this case the C library startup code, neither knows nor cares what the names of the formal parameters are in the module in which the subroutine is defined.

Accordingly, upon entry to **main()**, the stack will consist of the return address, then **argc**, then **argv**, and space will be made on the stack for any local variables **main()** may have. To illustrate that, consider the code

```
1  main(int argc, char **argv)
2  {  int i;
3     printf("%d  %s\n",argc,argv[1]);
4  }
```

After applying **gcc -S** to this, we get

```
1          .file        "argv.c"
2          .section        .rodata
3  .LC0:
4          .string        "%d  %s\n"
```

```
5           .text
6           .align 2
7   .globl main
8           .type    main,@function
9   main:
10          pushl    %ebp
11          movl     %esp, %ebp
12          subl     $8, %esp
13          andl     $-16, %esp
14          movl     $0, %eax
15          subl     %eax, %esp
16          subl     $4, %esp
17          movl     12(%ebp), %eax
18          addl     $4, %eax
19          pushl    (%eax)
20          pushl    8(%ebp)
21          pushl    $.LC0
22          call     printf
23          addl     $16, %esp
24          leave
25          ret
26  .Lfe1:
27          .size    main,.Lfe1-main
28          .ident    "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

We see that **argc** and **argv** are 8 and 12 bytes from EBP, respectively. This makes sense, since from Section 6.8.8 we know that upon entry to **main()**, the stack headed by the place pointed to by EBP will look like this:

```
argv
argc
return address
saved EBP
```

At the end of **main()**, the GCC compiler will put the return value in EAX, and will produce a LEAVE instruction.

Let's look at this from one more angle, by running this program via GDB:

```
1   % gcc -S argv.c
2   % as --gstabs -o argv.o argv.s
3   % gcc -g argv.o
4   % gdb -q a.out
5   (gdb) b 10
6   Breakpoint 1 at 0x8048328: file argv.s, line 10.
7   (gdb) r abc def
8   Starting program:
9   /fandrhome/matloff/public_html/matloff/public_html/50/PLN/a.out abc def
10
11  Breakpoint 1, main () at argv.s:10
12  10              pushl   %ebp
13  Current language:  auto; currently asm
```

```
(gdb) x/3x $esp
0xbfffe33c:     0x42015967      0x00000003      0xbfffe384
(gdb) x/3x 0xbfffe384
0xbfffe384:     0xbfffe95c      0xbfffe99c      0xbfffe9a0
(gdb) x/s 0xbfffe95c
0xbfffe95c:
"/fandrhome/matloff/public_html/matloff/public_html/50/PLN/a.out"
(gdb) x/s 0xbfffe99c
0xbfffe99c:      "abc"
(gdb) x/s 0xbfffe9a0
0xbfffe9a0:      "def"
```

So we see that the return address to the C library is 0x42015967, **argc** is 3 and **argv** is 0xbfffe384. The latter should be a pointer to an array of three strings, which is confirmed in the subsequent GDB commands.

By the way, the C library code provides a third argument to **main()** as well, which is a pointer to the environment variables (current working directory, executable search path, username, etc.). Then the declaration would be

```
int main(int argc, char **argv, char **envp)
```

6.8.11 Once Again, There Are No Types at the Hardware Level!

You have been schooled—properly so—in your beginning programming classes about the importance of **scope**, meaning which variables are accessible or inaccessible from which parts of a program. But if a variable is inaccessible from a certain point in a program, that is merely the compiler doing its gatekeeper work for that language. It is NOT a restriction by the hardware. There is no such thing as scope at the hardware level. ANY instruction in a program—remember, all C/C++ and assembly code gets translated to machine instructions—can access ANY data item ANYWHERE in the program.

In C++, you were probably taught a slogan of the form

> "A **private** member of a class cannot be accessed from anywhere outside the class."

But that is not true. The correct form of the statement should be

> "The compiler *will refuse to compile* any C++ code you write which attempts to access *by name* a **private** member of a class from anywhere outside the class."

Again, the gatekeeper here is the compiler, not the hardware. The compiler's actions are desirable, because the notion of scope helps us to organize our data, but it has no physical meaning. Consider this example:

```
#include <iostream.h>

class c {
   private:
      int x;
   public:
      c();  // constructor
      // printx() prints out x
      void printx() { cout << x << endl; }
};

c::c()
{  x = 5;  }

int main(int argc, char *argv[])
{  c ci;
   ci.printx();  // prints 5
   // now point p to ci, thus to the first word at ci, i.e. x
   int *p = (int *) &ci;
   *p = 29;  // change x to 29
   ci.printx();  // prints 29
}
```

The point is that the member variable **x** in the class **c**, though **private**, is nevertheless in memory, and we can get to any memory location by setting a pointer variable to that location. That is what we have done in this example. We've managed to change the value of **x** in an instance of the class through code which is outside the class.

The point alos applies to local variables. We can actually access a local variable in one function from code in another function. The following example demonstrates that (make sure to review Section 6.8.8 before reading the example):

```
void g()
{  int i,*p;
   p = &i;
   p = p + 1;  // p now points to main()'s EBP
   p = *p;  // p now equals main()'s EBP value
   p = p - 1;  // p now points to main()'s x
   *p = 29;  // changes x to 29
}

main()
{  int x = 5;  // x is local to main()
   g();
   printf("%d\n",x);  // prints out 29
}
```

6.8.12 What About C++?

If the calling program is C++ instead of C, you must inform the compiler that the assembly language routine is "C style," by inserting

```
extern "C" void addone(int *);
```

in the C++ source file. You need to do this, because your assembly language routine will be in the C style, i.e. utilize C conventions such as that concerning EBP above.

For **instance** (i.e. non-**static**) functions, note that the **this** pointer is essentially an argument too. The convention is that it is pushed last, i.e. it is treated as the first argument.

6.8.13 Putting It All Together

To illustrate a number of the concepts we've covered here, let's look at the full assembly code produced from the function **sum()** in Section 6.8.7. Here is the original C and then the compiled code:

```
1  int sum(int *x, int n)
2  {  int i=0,s=0;
3
4     for (i = 0; i < n; i++)
5        s += x[i];
6     return s;
7  }
```

```
1           .file   "sum.c"
2           .text
3           .align 2
4  .globl sum
5           .type   sum,@function
6  sum:
7           pushl   %ebp
8           movl    %esp, %ebp
9           subl    $8, %esp
10          movl    $0, -4(%ebp)
11          movl    $0, -8(%ebp)
12          movl    $0, -4(%ebp)
13 .L2:
14          movl    -4(%ebp), %eax
15          cmpl    12(%ebp), %eax
16          jl      .L5
17          jmp     .L3
18 .L5:
19          movl    -4(%ebp), %eax
20          leal    0(,%eax,4), %edx
21          movl    8(%ebp), %eax
22          movl    (%eax,%edx), %edx
```

```
23          leal    -8(%ebp), %eax
24          addl    %edx, (%eax)
25          leal    -4(%ebp), %eax
26          incl    (%eax)
27          jmp     .L2
28  .L3:
29          movl    -8(%ebp), %eax
30          leave
31          ret
32  .Lfe1:
33          .size   sum,.Lfe1-sum
34          .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

We see that the compiler has first placed the standard prologue at the beginning of the code, allowing for two local variables:

```
1   sum:
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $8, %esp
```

It then initializes both of those locals to 0.

```
1           movl    $0, -4(%ebp)
2           movl    $0, -8(%ebp)
```

Since our code sets **i** to 0 twice, the compiler does so too, since we didn't ask the compiler to optimize.

Our **for** loop compares **i** to **n**, which is done here:

```
1          movl    -4(%ebp), %eax
2          cmpl    12(%ebp), %eax
3          jl      .L5
4          jmp     .L3
```

The code must add **x[i]** to the sum:

```
1   .L5:
2           movl    -4(%ebp), %eax
3           leal    0(,%eax,4), %edx
4           movl    8(%ebp), %eax
5           movl    (%eax,%edx), %edx
6           leal    -8(%ebp), %eax
7           addl    %edx, (%eax)
```

Note the use of the LEA instruction and advanced addressing modes.

We need to increment **i** and go to the top of the loop:

```
1           leal    -4(%ebp), %eax
2           incl    (%eax)
3           jmp     .L2
```

When the function is finished, it needs to put the sum in EAX, as described in Section 6.8.5, clean up the stack and return:

```
1   .L3:
2           movl    -8(%ebp), %eax
3           leave
4           ret
```

Note the use of the LEAVE instruction.

Now, what if we declare that local variable **s** as **static**?

```
1   int sum(int *x, int n)
2   {  int i=0; static s=0;
3
4      for (i = 0; i < n; i++)
5         s += x[i];
6      return s;
7   }
```

Recall that this means that the variable will retain its value from one call to the next.[12] That means that the compile can't use the stack for storage of this variable, as it would likely get overwritten by calls from other subroutines. So, it must be stored in a **.data** section:

```
1           .file       "Sum.c"
2           .data
3           .align 4
4           .type       s.0,@object
5           .size       s.0,4
6   s.0:
7           .long       0
8           .text
9           .align 2
10  .globl sum
11          .type       sum,@function
12  sum:
13          pushl       %ebp
14          movl        %esp, %ebp
15          subl        $4, %esp
16          movl        $0, -4(%ebp)
17          movl        $0, -4(%ebp)
18  .L2:
19          movl        -4(%ebp), %eax
```

[12] Our code initializes **s** to 0, but this is done only once, as you can see from the assembly language here.

```
20          cmpl        12(%ebp), %eax
21          jl        .L5
22          jmp        .L3
23  .L5:
24          movl        -4(%ebp), %eax
25          leal        0(,%eax,4), %edx
26          movl        8(%ebp), %eax
27          movl        (%eax,%edx), %eax
28          addl        %eax, s.0
29          leal        -4(%ebp), %eax
30          incl        (%eax)
31          jmp        .L2
32  .L3:
33          movl        s.0, %eax
34          leave
35          ret
36  .Lfe1:
37          .size        sum,.Lfe1-sum
38          .ident        "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

You can tell from the code

```
        movl        s.0, %eax
        leave
        ret
```

that our C variable **s** is being stored at a label **s.0** in the **.data** section.

6.9 Subroutine Calls/Returns Are "Expensive"

On most machines, subroutine calls and returns add overhead. Not only does it mean that extra instructions must be executed, but also stack access, being memory access, is slow. So, it makes sense to ask how CPUs could be designed to speed this up.

One approach would be to make a special cache for the stack. Note that since the stack is in memory, some of it may be in an ordinary cache, but by devoting a special cache to the stack, we would decrease the cache miss rate for the stack.

But a more aggressive approach would be to design the CPU so that the top few stack elements are in the CPU itself. This is done in Sun Microsystems' SPARC chip, and was done back in the 60s and 70s on Burroughs mainframes. This would be better than a special cache, since the latter, with its complex circuitry for miss processing and so on, would have more delay.

On the other hand, space on a chip is precious. It may be that such usage of the space may not be as good as some other usage.

6.10 Debugging Assembly Language Subroutines

6.10.1 Focus on the Stack

At first, debugging assembly language subroutines would seem to be similar to debugging assembly language in general, and for that matter, debugging any kinds of code. However, what makes the assembly-language subroutine case special is the use of the stack. Many bugs involve errors in accessing the stack, such as failure to restore a register value which had been saved on the stack.

Recall that the essence of good debugging is confirmation. One uses the debugger, say **ddd**, to step through one's code line by line, and in each line confirms that our stored data (variables in C/C++, register and memory values in assembly language) have the values we expect them to. In stepping through our code, we also confirm that we execute the lines we expect to be executed, such as a conditional branch being taken when we expect it to be taken. Eventually this process will lead us to a line in which we find that our confirmation fails. This then will pinpoint where our bug is, and we can start to analyze what is wrong with that line.

For assembly language subroutines, our confirmation process consists of confirming that the contents of the stack are exactly as we expect them to be. Eventually we will find a line of code at which that confirmation fails, and we then will have pinpointed the location of our bug, and we can start to analyze what is wrong with that line.

To do that, you have to know how to inspect the stack from within your debugger, e.g. **ddd**. The key point is that the stack is part of memory, and the debugger allows you to inspect memory. So, first find out from the debugger the current value of ESP, and then inspect a few words of memory starting at wherever ESP points to, e.g.

```
x/4w $esp
```

to inspect the first four words of the stack in GDB.[13]

Note that if during a debugging session you run your program several times without exiting GDB,[14] the stack will keep growing. Thus any stack addresses which you jot down for later use may become incorrect.

6.10.2 A Special Consideration When Interfacing C/C++ with Assembly Language

When you are interfacing C/C++ to assembly language and while debugging you reach a function call in your C/C++ code, it is often useful to view the assembly language, especially the addresses of the instructions.

[13] You can now see an advantage of designing the hardware so that stacks grow toward 0. This causes consecutive elements of the stack to be in consecutive locations, making it easier to inspect the stack.

[14] And this is the proper way to do it. You SHOULD stay in GDB from one run to the next, so as to retain your breakpoints etc.

You may find GDB's **disassemble** command useful in this regard. It reverse-translates the machine code for the program into assembly language (regardless of whether the program source was originally assembly language or C/C++), and lists the assembly language instructions and their addresses. Note that the latter will be addresses with respect to the linker's assignments of the **.text** section location, not offsets as in the output of **as -a**. Also note that **.data** section symbols won't appear; the actual addresses will show up.

The **disassemble** command has several formats. If you run it without arguments, it will disassemble the code near your current location. You can also run it on any label in the **.text** section, e.g.

```
(gdb) disas _start
```

and

```
(gdb) disas main
```

In the latter case, where we have a function name, the entire function will be disassembled.

You can also specify a range of memory addresses, e.g.

```
(gdb) disas 0x08048083 0x08048099
```

6.11 Macros

A **macro** is like a subroutine in the sense that it makes one's programming more modular, but it actually is **inline code**, meaning that it produces code at the place where it is invoked.

To see what this means, consider this example from an earlier unit:

```
1  .data
2  x:
3        .long   1
4        .long   5
5        .long   2
6        .long   18
7  sum:
8        .long 0
9  .text
10 .globl _start
11 _start:
12       movl $4, %eax
13       movl $0, %ebx
14       movl $x, %ecx
15 top:  addl (%ecx), %ebx
16       addl $4, %ecx
17       decl %eax
18       jnz top
19 done: movl %ebx, sum
```

Those first three lines of executable code,

```
      movl $4, %eax
      movl $0, %ebx
      movl $x, %ecx
```

perform various initializations. To make our program more modular, we could make a subroutine out of them, say as **init()** in Section 6.7 (either with or without the argument). This would be fine, but it would slow down the execution of the program a bit, since the CALL and RET instructions would use some clock cycles and their stack access, being memory accesses, are slow. (And there is further slowdown if **init()** uses an argument).

An alternative would be to convert those three instructions to a macro:

```
1  .macro init
2        movl $4, %eax
3        movl $0, %ebx
4        movl $x, %ecx
5  .endm
6
7  .data
8  x:
9        .long   1
10       .long   5
11       .long   2
12       .long   18
13 sum:
14       .long 0
15
16 .text
17 .globl _start
18 _start:
19       init
20 top:  addl (%ecx), %ebx
21       addl $4, %ecx
22       decl %eax
23       jnz top
24 done: movl %ebx, sum
```

Though the "init" at **_start** has the look of a subroutine call, it really isn't. Instead, when the assembler sees this line at **_start**, it replaces that line with the three lines of code defined for "init" at the top of the file.

In other words, the macro version of the program will produce exactly the same machine code as did the original nonmodular version. We can see this by running both files through **as -a**; here are excerpts from the outputs of this command from the original and macro versions of the program:

```
...
  23                    _start:
  24 0000 B8040000            movl $4, %eax
```

```
  24      00
  25 0005 BB000000            movl $0, %ebx
  25      00
  26 000a B9000000            movl $x, %ecx
  26      00
  27 000f 0319          top:  addl (%ecx), %ebx
...


...
  20                    _start:
  21 0000 B8040000            init
  21      00BB0000
  21      0000B900
  21      000000
  22 000f 0319          top:  addl (%ecx), %ebx
...
```

So, both versions of the program would produce the same **.o** file. By contrast, the subroutine version produces different code:

```
...
  23                    _start:
  24 0000 E80E0000            call init
  24      00
  25 0005 0319          top:  addl (%ecx), %ebx
...
  31                    init:
  32 0013 B8040000            movl $4, %eax
  32      00
  33 0018 BB000000            movl $0, %ebx
  33      00
  34 001d B9000000            movl $x, %ecx
  34      00
  35 0022 C3                  ret
```

Note the machine code for the **call** instruction and the **ret**. So, by using a macro we get the efficiency of the nonmodular version while still getting a modular view for the programmer.[15]

Macros allow arguments too. Within a macro itself, the macro's arguments are rereferred to using backslashes.

For example, the macro **yyy** has two arguments, **u** and **v**:

```
.macro yyy u,v
   movl \u,%eax
   addl \v,%eax
.endm
```

[15] Though this modular view will not appear in a debugging tool, a drawback.

So the call

```
yyy $5,$12
```

will produce the same code as we had written

```
movl $5, %eax
addl $12, %eax
```

Chapter 7

Overview of Input/Output Mechanisms

7.1 Introduction

During the early days of computers, input/output (I/O) was of very little importance. Instead computers, as their name implied, were used to **compute.** A typical application might involve huge mathematical calculations needed for some problem in physics. The I/O done in such an application would be quite minimal, with the machine grinding away for hours at a time with little or no I/O being done.

Today, I/O plays a key role in computer applications. In fact, applications in which I/O plays an important role can arguably be viewed as the most powerful "driving force" behind the revolutionary rise to prominence of computers in the last decade or two. Today we make use of computers in numerous aspects of our business and personal lives–and in the vast majority of such uses, the programs center around I/O operations.

In some of these applications, the use of a computer is fairly obvious: First and foremost, the Internet, but also credit-card billing, airline reservations, word processors, spreadsheets, automatic teller machines, real estate databases, drafting programs for architects, video games, and so on. Possibly less obvious is the existence of computers in automobiles, and in such household items as washing machines and autofocus cameras. However, whether the existence of computers in these applications is obvious or hidden, **the common theme behind them is that they all are running programs in which I/O plays an important role, usually the dominant role.**

To give some concrete of the central role played by I/O, consider the example of the ATM machine. Here are some of the I/O devices the computer in the machine will have:

- The motor used to drag the card in and push it out.
- The scanner which reads the magnetic strip on the card.

- The motors used to prepare and push out the cash.
- The screen and keypad.
- The connection to the network (which leads to a central computer for the bank chain).

7.2 I/O Ports and Device Structure

An I/O device connects to the system bus through **ports**. I/O ports are similar in structure to CPU registers. However, a port is not in the CPU, being instead located between the I/O device and the system bus, in the **interface card** for the device.

Ports are usually 8 bits in width. They have addresses, just like words in memory do, and these addresses are given meaning through the address bus, just as is the case for words in memory.

7.3 Program Access to I/O Ports

Note that a potential ambiguity arises. Suppose we wish to communicate with port 50, which is connected to some I/O device. The CPU will place the value 50 in the MAR to go out onto the address bus. Since all items which are connected to the bus will see what is on the address bus, how can they distinguish this 50, intended for an I/O port, from a value of 50 meant to address 50 of memory? In other words, how can the CPU indicate that it wants to address I/O port 50 rather than memory word 50? There are two fundamental systems for handling this problem, described next.

7.3.1 I/O Address Space Approach

Recall that the system bus of a computer consists of an address bus, a data bus, and a control bus. The control bus in many computers, such as those based on Intel CPUs, has a special line or lines to indicate that the current value on the address bus is meant to indicate an I/O port, rather than a word in memory. We will assume four such lines, MR (memory read), MW (memory write), IOR (I/O read) and IOW (I/O write).

To access I/O port 50, the CPU will assert either IOR or IOW (depending on whether it wants to read from or write to that port), while to access word 50 in memory, the CPU will assert either MR or MW. In either case the CPU will place the value 50 on the address bus, but the values it simultaneously places on these lines in the control bus will distinguish between I/O port 50 and word 50 of memory.

The programmer himself or herself controls which of these lines will be asserted, by choosing which instruction to use.

```
movb 50, %al
```

and

```
inb $50, %al
```

will both cause the value 50 to go out onto the address bus, and in both cases the response will be sent back via the data bus. But the MOV will assert the MR line in the control bus, while the IN instruction will assert the IOR line in that bus. As a result, the MOV will result in copying the memory byte at address 50 to AL, while the IN will copy the byte from I/O port 50 to AL.[1]

Since the lines IOR/IOW and MR/MW allow us to distinguish between I/O ports and memory words having identical addresses, we can describe this by saying that the I/O ports have a separate **address space** from that of memory. Thus we will call this the **I/O address space approach**.

Note the profound hardward dependency here. The asserting of IOR/IOW or MR/MW comes from using IN/OUT or MOV, respectively, and thus the machine must have such instructions.

7.3.2 Memory-Mapped I/O Approach

Another approach to the "duplicate port/memory address" problem described above is called **memory-mapped I/O."**[2]

Under this approach, there are no special lines in the control bus to indicate I/O access versus memory access (though there still must be one or two lines to distinguish a read from a write, of course), and thus no special I/O instructions such as IN and OUT. One simply uses ordinary instructions such as MOV, whether one is accessing I/O ports or memory. Of course, the people who put the computer together must avoid placing memory at the addresses used by I/O ports.

An advantage of memory-mapped I/O is that it can be done directly in C, e.g.:

```
char c,*p;
...
p = 50;
c = *p;
```

Say **c** and **p** were global variables. Then the compiler will likely translate the line

[1] You may wonder why the dollar sign is needed in the second instruction, which seems inconsistent with that of the first instruction. But this comes from Intel tradition, so we are stuck with it.

[2] Unfortunately, many terms in the computer field are "overloaded." It has become common on personal computers to refer to the mapping of video monitor pixels to memory addresses also as "memory-mapped I/O." However, this is not the original use of the term, and should not be confused with what we are discussing here.

```
c = *p;
```

to something like

```
movl p, %ebx
movb (%ebx), %al
movb %al, c
```

So you can see that from the CPU's point of view, reading that I/O port, which is done by the instructions

```
movb (%ebx), %al
```

is the same as reading from memory. Indeed, the CPU is not aware of the fact that we are reading from an I/O port, which has a very different operation and physical structure from memory.

7.4 Wait-Loop I/O

Consider the case of reading a character from the keyboard. The program which will read the character has no idea when the user will hit a key on the keyboard. How should the program be written to deal with this time problem? (Note that we are for the moment discussing programs which directly access the keyboard, rather than going through the operating system, as is the case with programs you usually write; more on this point later.)

On an Intel-based PC, the keyboard data port (KDP) has address 0x60, and its status port (KSP) is at 0x62. Among other things, the KSP tells us whether a key has been typed yet; Bit 4 will be 1 if so, 0 otherwise.[3]

Bit 5 of the KSP plays a role too. When we read the character, we have to notify the keyboard hardware that we've gotten it, so that it can give us another character when one is typed. To do this, we briefly set Bit 5 to 1 and then back to 0. This will cause Bit 4 of the KSP to revert to 0 too.

Wait-loop I/O then consists of writing a loop, in which the program keeps testing Bit 4 of the KSP until that bit indicates "character ready," and then read the character from KDP into some register:[4]

```
1       # loop around until a character is ready
2   lp:
3       inb $0x62, %bl        # get value of KSP
4       andb $0x10, %bl       # check Bit 4
5       jz lp                 # if that bit is 0, try again
6   ready:
```

[3] As usual, we are naming the least significant bit "Bit 0," etc.

[4] By the way we won't have the character in ASCII form. Instead, we will have it as a **scan code**. More on this below.

```
7       # get the character
8       inb $0x60, %cl
9       # acknowlege getting the character
10      # set Bit 5 of KSP to 1
11      orb $0x20, %bl
12      outb %bl, $0x62
13      # reset Bit 5 of KSP to 0
14      andb $0xdf, %bl
15      outb %bl, $0x62
16  done:
```

During the time before the user hits a key, the value read from the KSP will always have a 0 in Bit 4, so that the AND results in 0, and we jump back to `lp`. But eventually the user will hit a key, resulting in Bit 4 of the KSP being 1, and we leave the loop. We then pick up the character in the code starting at `done`.

7.5 PC Keyboards

On Intel- (or Intel clone-) based PCs, the keyboard is not directly wired for ASCII. Instead, each of the keys has its own **scan code**—actually two codes, one which the keyboard emits when a key is pressed and the other, 128 more than the first code, which the keyboard emits when the key is released. For example, the A key has codes 0x1e and 0x9e, respectively. The keyboard driver will then convert to ASCII.

Say for instance that the user wishes to type 'A', i.e. the capital letter. She may hit the left Shift key (code 0x2a), then hit the A key (code 0x1e), then release the A key (code 0x9e) then release the left Shift key (code 0xaa). The keyboard driver can be written to notice that when the A key is hit, the Shift key has not yet been released, and thus the user must mean a capital 'A'. The driver then produces the ASCII code 0x41.

7.6 Interrupt-Driven I/O

7.6.1 Telephone Analogy

Wait-loop I/O is very wasteful. Usually the speed of I/O device is very slow compared to CPU speed. This is particularly true for I/O devices which are mainly of a mechanical rather than a purely electronic nature, such as printers and disk drives, and thus are usually orders of magnitude slower than CPU actions. It is even worse for a keyboard: Not only is human typing extremely slow relative to CPU speeds, but also a lot of the time the user is not typing at all; he/she may be thinking, taking a break, or whatever.

Accordingly, if we use wait-loop I/O, the CPU must execute the wait loop many times between successive I/O actions. This is wasted time and wasted CPU cycles.

An analogy is the following. Suppose you are expecting an important phone call at the office. Imagine how wasteful it would be if phones didn't have bells—-you would have to repeatedly say, "Hello, hello, hello, ..."

into the phone until the call actually came in! This would be a big waste of time. The bell in the phone frees you from this; you can do other things around the office, without paying attention to the phone, because the bell will notify you when a call arrives.

Thus it would be nice to have an analog of a telephone bell in the computer. This does exist, in the form of an **interrupt.** It takes the form of a pulse of current sent to the CPU by an I/O device. This pulse forces the CPU to suspend, i.e. "interrupt," the currently-running program, say "X," and switch execution to another procedure, which we term the **interrupt service routine** (ISR) for that I/O device.[5]

The ISR is usually is known as the *device driver* for that I/O device. It is typically part of the operating system, but it of course exists on systems without an OS too.

7.6.2 What Happens When an Interrupt Occurs?

Think for example of the keyboard or any other I/O device which receives characters.

There will be an interrupt request line (IRQ) in the control bus. When an I/O device receives a character, it will assert IRQ.

Recall that we have described the CPU as repeatedly doing Step A, Step B, Step C, Step A, Step B, etc. where Step A is instruction fetch, Step B is instruction decode, and Step C is instruction execution. Well, in addition, the CPU will check the IRQ line after every Step C it does. If it sees IRQ asserted, it will do a Step D, consisting of the following in the Intel case:

```
CPU pushes current EFLAGS value on stack
CPU pushes current CS value on stack (irrelevant in 32-bit protected mode)
CPU pushes current PC value on stack
CPU does PC <-- ISR addresss
```

The next Step A will, as usual, fetch from wherever the PC points, which in this case will be the ISR. The ISR will start executing.

At the end of the ISR there will be an IRET ("interrupt return") instruction, which "undoes" all of this:

```
CPU pops stack and placed popped value into PC
CPU pops stack and placed popped value into CS
CPU pops stack and placed popped value into EFLAGS
```

Say this occurs when persons X and Y are both using this machine, say X at the console and Y remotely via `ssh`. X's and Y's programs take turns using the machine (details in our unit on OS). Say during Y's turn,

[5] This is called a **hardware interrupt** or **external interrupt**. Those who have done prior assembly-language programming on PCs should not confuse this with the INT instruction. The latter is almost the same as a procedure CALL instruction, and is used on PCs to implement systems calls, i.e. calls to the operating system.

X types a character. That causes an interrupt, which will be noticed by the CPU when it finishes Step C of whatever instruction in Y's program it had been executing when X hit the key.

From the above description, you can see that the CPU will make a sudden jump to the ISR, so Y's program will stop running. But the hardware saves the current PC and EFLAGS values of Y's program on the stack, so that Y's program can resume later in exactly the same conditions it had at the time it was interrupted. Note carefully that all of that will be done by the hardware, without a single instruction being executed.

The IRET instruction at the end of the ISR is what restores those conditions. Note for instance that it is crucial that Y's EFLAGS value be restored. If for example Y was in the midst of executing the first of the pair of instructions

```
subl %eax, %ebx
jz x
```

execution of the second, which will occur when Y's program resumes, will check whether the Z (Zero) flag in EFLAGS had been set by the SUB instruction. So, EFLAGS must be saved by the CPU when the interrupt occurs, and restored by the IRET.

Make absolutely SURE you understand that the jump to the ISR did NOT occur from a jump or call instruction in Y's program. Y did nothing to cause that jump. However, the jump back to Y did occur because the OS programmer put an IRET at the end of the ISR.

Note too what happens if Y hits a key while X's program is running. That will cause an interrupt to be felt at Y's CPU, which will result in it sending the character through the network, eventually ending up in the Ethernet data port of the machine where X is–causing an interrupt on THAT machine.

7.6.3 Alternative Designs

By the way, note that when an interrupt occurs, the CPU doesn't even see it until it finishes Step C of whatever instruction was executing at the time the INTR line was asserted. It could be designed differently, by setting up the circuitry so that instructions would be inerruptible in the middle. That would reduce interrupt response time a bit, which is good, but at a great expense in complexity. Not only would the circuitry have to save the values of the EFLAGS, CS and PC registers on the stack, but it would also have to save information as to just how much of the instruction had been executed at the time it was interrupted. That would be a lot of extra design work, for rather little performance benefit.

However, in the Intel case, they did make one concession to this idea, in the case of the MOVS instruction family. A MOVS instruction may take so long to execute that it is worth interrupting in the middle, and the Intel engineers did design it this way.

7.6.4 Glimpse of an ISR

Here is what a simple keyboard ISR might look like, assuming we wish to store the character at a label `keybdbuf` in memory:

```
1         # save interrupted program's EAX and EBX
2         pushl %eax
3         pushl %ebx
4         # get character
5         inb $0x60, %al
6         # copy it to the keyboard buffer
7         movb %al, keybdbuf
8         # acknowlege getting the character
9         inb $0x62, %bl
10        orb $0x20, %bl
11        outb %bl, $0x62
12        andb $0xdf, %bl
13        outb %bl, $0x62
14        # restore interrupted program's EAX, EBX values
15        popl %ebx
16        popl %eax
17        # back to the interrupted program
18        iret
```

The OS/ISR is not running when the interrupt occurs. The key point is that at some times (whenever someone hits a key, in this case) the ISR will need to be run, and from the point of view of the interrupted program, that time is unpredictable. In the example here, each time a character is typed, a different instruction within the program might get interrupted.

7.6.5 I/O Protection

Linux runs the Intel CPU in **flat protected mode**, which sets 32-bit addressing and enables the hardware to provide various types of security features needed by a modern OS, such as virtual memory. Again for security reasons, we want I/O to be performed only by the OS. The Intel CPU has several modes of operation, which we will simplify here to just User Mode and Kernel (i.e. OS) Mode. The hardware is set up so that I/O instructions such as IN and OUT can be done only in Kernel Mode, and that an interrupt causes entry to Kernel Mode.[6]

[6]By setting a special flag in EFLAGS, the OS can allow user programs to execute IN and OUT for specified devices.

7.6.6 Distinguishing Among Devices

7.6.6.1 How Does the CPU Know Which I/O Device Requested the Interrupt?

Each I/O device is assigned an ID number, known colloquially as its "IRQ number." In PCs, typcally the 8253 timer has IRQ 0, the keyboard has IRQ 1, the mouse IRQ 12, and so on.

After the I/O device asserts the IRQ line in the bus and the CPU notices, the CPU will then assert the INTA, Interrupt Acknowledge, line in the bus. That assertion is basically a query from the CPU to all the I/O devices, saying, "Whoever requested the interrupt, please identify yourself." The requesting I/O device then sends its ID number (e.g. 1 in the case of the keyboard) to the CPU along the data lines in the bus, so that the CPU knows that it was device 1 that requested the interrupt.

7.6.6.2 How Does the CPU Know Where the ISR Is?

The Intel CPU, like many others, uses **vectored interrupts**. Each I/O device will have its own "vector," which consists of a pointer to the I/O device's ISR in memory, plus some additional information.[7] The pointer and the additional information comprise the "vector" for this I/O device.

All the vectors are stored in an interrupt vector table in memory, a table which the OS fills out upon bootup. Each vector is 8 bytes long, so the vector for I/O device i is located at c(IDT)+8*i, where c() means "contents of." The CPU has an Interrupt Descriptor Table register, IDT. Upon bootup, the OS will point this register to the beginning of the table. So for example, upon bootup, the OS initializes the first 4 bytes at c(IDT)+8 to point to the OS's keyboard device driver (ISR).

Note the hardware/software interaction here: The CPU hardware is what reacts to the interrupt, but the place that the CPU jumps to was specified by the software, i.e. the OS, in the interrupt vector table in memory. The ISR is of course also software.

7.6.6.3 Revised Interrupt Sequence

We now see that we have to expand the description in Section 7.6.2 of what the circuitry does when an interrupt occurs:

```
CPU pushes current EFLAGS value on stack
CPU pushes current CS value on stack (irrelevant in 32-bit protected mode)
CPU pushes current PC value on stack
CPU asserts INTA line
CPU reads i from data bus
```

[7] One piece of such "additional information" is the mode in which the ISR is to be run, which typically will be set to Kernel Mode. Later, when the ISR ends by executing an IRET instruction, the mode of the interrupted program will be restored.

```
CPU computes j = c(IDT)+8*i
CPU reads location j+4, sets interrupt hardware options
CPU reads location j; let k = c(j)
CPU does PC <-- k
```

7.6.7 How Do PCs Prioritize Interrupts from Different Devices?

Interrupt systems can get quite complex. What happens, for example, when two I/O devices both issue interrupts at about the same time? Which one gets priority? Similarly, what if during execution of the ISR for one device, another device requests an interrupt? Do we suspend the first ISR, i.e. give priority to the second?

To deal with this problem, PCs also includes another piece of Intel hardware, the Intel 8259A interrupt controller chip. The I/O devices are actually connected to the 8259A, which in turn is connected to the IRQ (Interrupt Request) line, instead of the devices being connected directly to the IRQ. The 8259A has many input pins, one for each I/O device.[8] So, the 8259A acts as an "agent," sending interrupt requests to the CPU "on behalf of" the I/O devices.

If several I/O devices cause interrupts at about the same time, the 8259A can "queue" them, so that all will be processed, and the 8259A can prioritize them. Lots of different priority schemes can be arranged, as the 8259A is a highly configurable, complex chip.

Note that this configuration is done by the OS, at the time of bootup. The 8259A has various ports, e.g. at 0x20 and 0x21, and the OS can set or clear various bits in those ports to command the 8259A to follow a certain priority scheme among the I/O devices.

Today a PC will typically have two 8259A chips, since a single chip can work with only eight devices. The two chips are **cascaded**, meaning that the second will feed into the first. The IRQ output of the second 8259A will feed into one of I/O connection pins of the first 8259A, so that the second one looks like an I/O device from the point of view of the first.

Many other methods of prioritizing I/O devices are possible. For example, we can have the CPU ignore all interrupts until further notice. There is an Interrupt Enable Flag in EFLAGS. The CLI instruction clears it, telling the CPU to ignore interrupts (i.e. skip all Step Ds), while the STI instruction makes the CPU resume responding to interrupts (resume executing a Step D after each Step C).

Designing hardware and software for systems with many I/O devices in which response time is crucial, say for an airplane computer system, is a fine art, and the software aspect is known as **real-time programming**.

[8] These pins are labeled IRQ0, IRQ1, etc., corresponding the IRQ numbers for the devices.

7.7 Direct Memory Access (DMA)

In our keyboard ISR example above, when a user hit a key, the ISR copied the character from the KDP into a register,

```
inb $0x60, %al
```

then copied the register to memory,

```
movb %al, keybdbuf
```

This is wasteful; it would be more efficient if we could arrange for the character to go directly to memory from the KDP. This is called **direct memory access** (DMA).

DMA is not needed for devices which send or receive characters at a slow rate, such as a keyboard, but it is often needed for something like a disk drive, which sends/receives characters at very high rates. So in many machines a disk drive is connected to a DMA controller, which in turn connects to the system bus.

A DMA controller works very much like a CPU, in the sense that it can work as a **bus master** like a CPU does. Specifically, a DMA controller can read from and write to memory, i.e. assert the MR and MW lines in the bus, etc.

So, the device driver program for something like a disk drive will often simply send a command to the DMA controller, and then go dormant. After the DMA controller finishes its assignment, saying writing one disk sector to memory, it causes an interrupt; the ISR here will be the device driver for the disk, though the ISR won't have much work left to do.

7.8 Disk Structure

Your disk files consist of sequences of bytes, which in turn consist of bits. Each bit on a disk is stored as a magnetized spot on the disk (a 1 value being represented by one polarity, and a 0 by the opposite polarity). The spots are arranged in concentric rings called **tracks**. Within each track, the spots are grouped into **sectors**, say 512 bytes each. A disk drive will read or write an entire sector at a time; you cannot request it to access just one bit or byte. The disk rotates at a rate of several thousand RPM. A **read/write head** reads/writes the bits which rotate under it.

Each time we do a disk access there are three components of the access time:

- First, the disk's read/write head must be moved to the proper track, an action called the **seek**. This typically takes on the order of tens of milliseconds.

- Then we must wait for the **rotational delay**, i.e. for the desired sector to rotate around to the read/write head.
- Finally, there is the **transfer time**, i.e. the time while the sector is actually being read/written.

Optimization of disk access is a big business. For example, a large banking chain has millions of credit card transactions to deal with every day. If the database program were to not access the disk as quickly as possible, the bank literally might not be able to keep up with the transaction load. Thus careful placement of files on disk is a necessity.

On large systems, several disks (*platters* might be stacked up, one above another. Think for instance of all the Track 3s on the various disks. They will collectively look like a cylinder, and thus even on single-disk systems a synonym for *track* is *cylinder*.

7.9 USB Devices

The acronym USB stands for **Universal Serial Bus**. The term *bus* here means that many I/O devices can be attached to the same line, i.e. the same USB plug (though on typical home PCs there is no need for this, since there are usually enough USB ports to use separately). The term *serial* refers to the fact that there is only one data line per USB port, as opposed to the 32 or more parallel lines on a system bus.

There are two ways that multiple devices can be attached to the same USB port:

- They can be **daisy-chained**, meaning that, say, device X connects to device Y which in turn is connected to the USB port.
- Several devices can be connected to a USB **hub**. The latter would in turn be connected to a USB port, or into other USB devices or hubs. In this manner, the set of devices connected to a given USB port can has a tree structure.

At most 127 I/O devices can be connected to a given USB port.

The USB idea really makes things convenient, for several reasons:

- We can connect new devices without having to open up the machine.
- We don't need a lot of expansion slots inside the machine.
- Devices can be **hot-swapped**, i.e. we can connect a new device to a machine which is already running.
- Since we are working with the same general structure, it is easier to write device drivers that work across machines.

Chapter 8

Overview of Functions of an Operating System

8.1 Introduction

8.1.1 It's Just a Program!

First and foremost, it is vital to understand that an **operating system** (OS) is just a program—a very large, very complex program, but still just a program. The OS provides support for the loading and execution of other programs (which we will refer to below as "application programs"), and the OS will set things up so that it has some special privileges which user programs don't have, but in the end, the OS is simply a program.

The source code for Linux, for instance, consists of millions of lines, mainly C but some assembly language for certain low-level machine-specific tasks. The binary for the program typically has a name which is some variant of **vmlinuz**. Look for it in your Linux machine, say in the directory **/boot**.

For example, when your program, say **a.out**,[1] is running, the OS is *not* running, Thus the OS has no power to suspend your program while your program is running—since the OS isn't running! This is a key concept, so let's first make sure what the statement even means.

What does it mean for a program to be "running" anyway? Recall that the CPU is constantly performing its fetch/execute/fetch/execute/... cycle. For each fetch, it fetches whatever instruction the Program Counter (PC) is pointing to. If the PC is currently pointing to an instruction in your program, then your program is running! Each time an instruction of your program executes, the circuitry in the CPU will update the PC, having it point to either the next instruction (the usual case) or an instruction located elsewhere in your

[1] Or it could be a program which you didn't write yourself, say **gcc**.

program (in the case of jumps).

The point is that the only way your program can stop running is if the PC is changed to point to another program, say the OS. How might this happen? There are only two ways this can occur:

- Your program can *voluntarily* relinquish the CPU to the OS. It does this via a **system call**.which is a call to some function in the operating system which provides some useful service.
  For example, suppose the C source file from which **a.out** was compiled had a call to **scanf()**. The **scanf()** function is a C library function, which was linked into **a.out**. But **scanf()** itself calls **read()**, a function within the OS. So, when **a.out** reaches the **scanf()** call, that will result in a call to the OS, but after the OS does the read from the keyboard, the OS will return to **a.out**.

- The other possibility is that a hardware interrupt occurs. This is a signal—a physical pulse of current along an **interrupt-request** line in the bus—from some input/output (I/O) device such as the keyboard to the CPU. The circuitry in the CPU is designed to then jump to a place in memory which we designated upon bootup of the machine. This will be a place in the OS, so the OS will now run. The OS will attend to the I/O device, e.g. record the keystroke in the case of the keyboard, and then return to the interrupted program via an IRET instruction.

  Note in our keystroke example that the keystroke may not have been made by you. While your program is running, some other user of the machine may hit a key. The interrupt will cause your program to be suspended; the OS will run the device driver for whichever device caused the interrupt—the keyboard, if the person was sitting at the console of the machine, or the network interface card, if the person was logged in remotely, say via **ssh**—which will record the keystroke in the buffer belonging to that other user; and the OS will execute an **iret** to return from the interrupt, causing your program to resume.

  A hardware interrupt can also be generated by the CPU itself, e.g. if your program attempts to divide by 0, or tries to access memory which the hardware finds is off limits to your program. (The latter case describes a seg fault; more on this later.)

So, when your program is running, it is king. The OS has no power to stop it, since it is NOT running. The only ways your program can stop running is if it voluntarily does so or is forced to stop by action occuring at an I/O device.

8.1.2 What Is an OS for, Anyway?

A computer might not even have an OS, say in embedded applications. A computer embedded in a washing machine will just run one program (in ROM). So, there are no files to worry about, no issues of timesharing, etc., and thus no need for an OS.

But on a general-purpose machine, we need an OS to do:

- Input/output device management:

  If you recall the example in our unit on I/O, we saw that at the machine level it takes about a dozen lines of code just to read one character from the keyboard. We certainly would not want to have to deal with that in our own programs, but fortunately the OS provides this service for us. Let's see what that means in the case of the keyboard.

  First, the OS includes the keyboard device driver, so that we do not need to write our own. Second, the OS provides the **read()** function for us to call in order to pick up the characters the keyboard driver has collected for us. Suppose for instance that you have a **scanf()** call or a **cin** statement in your program. When the user types some characters, each keystroke will cause an interrupt.[2] When the interrupt occurs, the keyboard driver will add the character to an array, typically referred to as a **buffer**. Your program's **scanf()** call or **cin** statement will include to a call to **read()**, a function in the OS. That function will then pick up the characters in the keyboard buffer, parse them according to whatever kind of read format you asked for, and return the result to your program.

  Again, the central point here is that the OS is doing a big service for you, in that it is alleviating you of the burden of writing all this code.

- File management:

  For example, suppose you are writing a program to write a new file. You wouldn't want to have to deal with the details of knowing where the empty space is on the disk, choosing some of it for the file, etc. Again, the OS does this for you.

- Process management:

  When you want to run a program, the OS loads it into memory for you and starts its execution.

  The OS enables **timesharing**, in which many application programs seem to be running simultaneously but are actually "taking turns," *by coordinating with hardware operations*.

- Memory management:

  The OS must keep track of which parts of memory are currently in use, so that it knows what areas it can load a program to when the program is requested for execution.

  The OS also enables **virtual memory** operations for application programs, which both allows flexible use of memory and enforces security, again by *by coordinating with hardware operations*.

How the OS does these things is explained in the following sections. We will model the discussion after a Unix system, but the description here applies to most modern OSs. It is assumed here that the reader is familiar with basic Unix commands; a Unix tutorial is available at `http://heather.cs.ucdavis.edu/~matloff/unix.html`

[2] Actually two interrupts, one for key press and one for release, but let's say one here, for simplicity.

8.2 Application Program Loading

8.2.1 Basic Operations

Suppose you have just compiled a program, producing, say for a Linux system, an executable file **a.out**. (And the following would apply equally well to **gcc** or any other program.) To run it, you type

```
% a.out
```

For the time being, assume a very simple machine/OS combination, with no **virtual memory**. Here is what will occur:

- During your typing of the command, the shell is running, say **tcsh** or **bash**. Again, the shell is just a program (one which could be assigned as a homework problem in a course). It prints out the '%' prompt using **printf()**,[3] and enters a loop in which it reads the command you type using **scanf()**.[4]
- The shell will then make a system call, **execve()**,[5] asking the OS to run **a.out**. The OS is now running.
- The OS will look in its disk directory, to determine where on disk the file **a.out** is. It will read the **header**, i.e. beginning section of **a.out**, which contains information on the size of the program, a list of the data sections used by the program, and so on.[6]
- The OS will check its memory-allocation table (this is just an array in the OS) to find an unused region or regions of memory large enough for **a.out**. (Note that the OS has loaded all currently-active programs in memory up to now, just like it is loading **a.out** now, so it knows exactly which parts of memory are in use and which are free.) We need space both for **a.out**'s instructions, in the **.text** section, its static data (in the **.data** section for unitialized variables, and in the **.bss** section for initialized variables), as well as for stack space and the **heap**.

  The heap is used for calls to **malloc()** or for invocations of the C++ operator **new**. With the GCC compiler, these are one and the same, since the internal implementation of **new** calls **malloc()**. There will be a data structure there which keeps track of which parts of the heap are in use and which are free. Of course, they are initially all free. Each time we need new memory, we use parts of the heap, and they will be marked as in use. Calling **free()** or invoking **delete** will result in the space being marked as free again. All the marking is done by **malloc()**.

[3] Most shells are written in C.

[4] Note that there will be an interrupt each time you type a character. The OS will store these characters until you hit Enter, in which case the OS will finally "wake" the shell program, whose **scanf()** will now finally execute. See also Sleep and Run states later in this unit.

[5] It will also call another system call, **fork()**, but we will not go into that here.

[6] You can read that information yourself, using the **readelf** command in Linux.

- The OS will then load **a.out** (including text and data) into those regions of memory, and will update its memory-allocation table accordingly.
- The OS will check a certain section of the **a.out** file, in which the linker previously recorded **a.out**'s **entry point**, i.e. the instruction in **a.out** at which execution is to begin. (This for example is _start in our previous assembly language examples.)
- The OS is now ready to initiate execution of **a.out**. It will set the stack pointer to point to the place it had chosen earlier for **a.out**'s stack. (The OS will save its own register values, including stack pointer value, beforehand.) Then it will place any command-line arguments on **a.out**'s stack, and then actually start **a.out**, say by executing a JMP instruction (or equivalent) to jump to **a.out**'s entry point.
- The **a.out** program is now running!

8.2.2 Chains of Programs Calling Programs

Note that **a.out** itself may call **execve()** and start other programs running. We mentioned above that the shell does this, and so for example does **gcc**.[7] It first runs the **cpp** C preprocessor (which translates **#include**, **#define** etc. from your C source file). Then it runs **cc1**, which is the "real" compiler (**gcc** itself is just a manager that runs the various components, as you can see now); this produces an assembly language file.[8] Then **gcc** runs **as**, the assembler, to produce a **.o** machine code file. The latter must be linked with some code, e.g. **/usr/lib/crt1.o** and other files which set up the **main()** structure, e.g. access to the **argv** command-line arguments, and also linked to the C library, **/lib/libc.so.6**; so, **gcc** runs the linker, **ld**. In all these cases, **gcc** starts these programs by calling **execve()**.

8.2.3 Static Versus Dynaming Linking

Say your C code includes a call to **printf()**. Even though someone else wrote **printf()**, when you link it in to your program (which GCC does for you when it calls LD), it becomes part of your program. The question is when this occurs.

With **static** linking, LD will put **printf()**'s machine code into your **a.out** executable. However, since most programs make use of the C library, they would collectively take up a lot of disk space if static linking were used.

[7] Remember, a compiler is just a program too. It is a very large and complex program, but in principle no different from the programs you write.

[8] You can view this file by running **gcc** with its -S option. This is often handy if you are going to write an assembly-language subroutine to be called by your C code, so that you can see how the compiler will deal with the parameters in the call to the subroutine.

So, the default is that GCC and LD would set up **dynamic** linking for **printf()** in your program. In your **a.out** file, there will be a message saying that your code calls **printf()**, and that when your program is loaded into memory and is run, **printf()** should be linked in to it at that time.

8.2.4 Making These Concepts Concrete: Commands You Can Try Yourself

8.2.4.1 Mini-Example

Below is a simple illustration of **execve()** in action. The program **run.c** makes this system call to get the program **greeting.c** running. Assume the executable files are named **greeting** and **run**.

```
1  // greeting.c
2
3  main()
4  {  printf("hello\n");  }
```

```
1  // run.c
2
3  main()
4  {  char **a[2],  // arguments to "greeting" (none here; see man page)
5           **p[2];  // environment for "greeting" (none here)
6
7     a[0] = ""; a[1] = 0;
8     p[0] = ""; p[1] = 0;
9     execve("./greeting",a,p);  }
```

```
1  % run
2  hello
```

Note that the shell will do something like this too, when you request that your program **a.out** be run.

8.2.4.2 The strace Command

The Unix **strace** command will report which system calls your program makes. Place it before your program name on the command line, e.g.

```
% strace a.out
```

The output will be rather overwhelming, but if you sift through it[9] You will see that a call **execve()** is made by the shell to launch **a.out**, as well as the various systems call made by **a.out**.

[9] I recommend running the Unix **script** command before you run **strace**. It will keep a record of everything that appears on your screen. After you are done running **strace**, type `exit` to leave the shell created by **script**. Then the record of your session is in a file named **typescript**. You can then browse through it using your favorite text editor.

8.3 OS Bootup

As was explained earlier, when we wish to run an application program, the OS loads the program into memory. But wait! The OS is a program too, so how does *it* get loaded into memory and begin execution? The answer is that another program, the **boot loader**, will load the OS into memory. But then how does the boot loader get loaded in memory?! It sounds like we have an endless, "chicken and egg" problem. The resolution is that the boot loader is permanently in memory, in ROM, so it doesn't have to be loaded.

The process by which the OS is loaded into memory is called **bootup**. We will explain this using the Intel Pentium architecture for illustration.[10]

The circuitry in the CPU hardware will be designed so that upon powerup the Program Counter (PC) is initialized to some specific value, 0xfffffff0 in the case of Intel CPUs. And those who **fabricate** the computer (i.e. who put together the CPU, memory, bus, etc. to form a complete system) will include a small ROM at that same address, again 0xfffffff0 in the Intel case. The contents of the ROM include the boot loader program. So, immediately after powerup, the boot loader program is running!

The goal of the boot loader is to load in the OS from disk to memory. In the simple form, the boot loader reads a specified area of the disk, copies the contents there—which will be part of the OS—to some section of memory, and then finally executes a JMP instruction (or equivalent) to that section of memory—so that now the OS is running. The OS then reads the rest of itself into memory.

For the sake of concreteness, let's look more closely at the Intel case. The program in ROM here is the BIOS, the Basic I/O System. It contains parts of the device drivers for that machine,[11] and also contains the first-stage boot loader program.

The boot loader program in the BIOS has been written to read some information, to be described now, from the first physical sector of the disk.[12] That sector is called the Master Boot Record (MBR).

There will be one or more **partitions** for the disk. If, say, the disk has 1,000 cylinders, then for example we may have the first 200 as one partition and the remaining 800 as the second partition.[13] These simply divide up parts of the disk for different purposes. A typical example would be that there are two partitions, one for Windows and the other for Linux. Your machine was probably shipped with the entire disk consisting of just one partition, but you can split it into several partitions to accommodate different OSs.

These partitions are defined in the **partition table** in the MBR. Note that there is no physical separation from one partition to the next; the boundary between one partition and the next is only defined by the partition table. Exactly one of the primary partitions will be marked in the table as **active**, meaning bootable.

[10] An overview of that architecture is presented in Section 8.9.

[11] These may or may not be used, depending on the OS. Windows uses them, but Linux doesn't.

[12] The boot device could be something else instead of a hard drive, such as a floppy or a CD-ROM. This is set in the BIOS, with a priority ordering of which device to try to boot from first.

[13] You should review the material on disks in our unit on I/O.

The MBR also contains the second-stage boot loader program. The boot loader program in the BIOS will read this second-stage code into memory and then jump to it, so that that program is now running. That program reads the partition table, to determine which partition is the active one. It then goes to the first sector in that partition, reads in the third-stage boot loader program into memory from there, and then jumps to that program.

Now if the machine had originally been shipped with Windows installed, the second-stage code in the MBR had been written to then load into memory the third-stage code from the Windows partition. If on the other hand the machine had been shipped with Linux or some other OS installed, there would be a partition for that OS; the code in the MBR would have been written accordingly, and that OS would now be loaded into memory.

Many people who use Linux retain both Windows and Linux on their hard drives, and have a **dual-boot** setup. They start with a Windows machine, but then install Linux as well, so both OSs are on the machine. As part of the process of installing Linux, the old MBR is copied elsewhere, and a program named LILO (Linux Loader) is written into the MBR.[14] In other words, LILO will be the second-stage boot loader. LILO will ask the user whether he/she wants to boot Linux or Windows, and then go to the corresponding partition to load and execute the third-stage boot loader which is there.[15]

In any case, after the OS is loaded into memory by the third-stage code, that code will perform a jump to the OS, so the OS is running.

8.4 Timesharing

8.4.1 Many Processes, Taking Turns

Suppose you and someone else are both using the computer **pc12** in our lab, one of you at the console and the other logged in remotely. Suppose further that the other person's program will run for five hours! You don't want to wait five hours for the other person's program to end. So, the OS arranges things so that the two programs will take turns running. It won't be visible to you, but that is what happens.

Timesharing involves having several programs running in what appears to be a simultaneous manner.[16] If the system has only one CPU (for simplicity, we will exclude the case of multiprocessor systems in this discussion), this simultaneity is of course only an illusion, since only one program can run at any given time, but it is a worthwhile illusion, as we will see.

[14] A newer, now more popular alternative to LILO is GRUB. But the principle is the same for both, so we will just refer to LILO here for simplicity.

[15] The partition does not have to be the active one. The term *active* applies only from the point of view of the Windows second-stage boot loader which had originally been installed in the MBR.

[16] These programs could be from different users or the same user; it doesn't matter.

First of all, how is this illusion attained? The answer is that we have the programs all take turns running, with each turn—called a **quantum** or **timeslice**—being of very short duration, for example 50 milliseconds. Say we have four programs, **u**, **v**, **x** and **y**, running currently. What will happen is that first **u** runs for 50 milliseconds, then **u** is suspended and **v** runs for 50 milliseconds, then **v** is suspended and **x** runs for 50 milliseconds, and so on. After **y** gets its turn, then **u** gets a second turn, etc. Since the turn-switching, formally known as **context-switching**,[17] is happening so fast (every 50 milliseconds), it appears to us humans that each program is running continuously (though at one-fourth speed), rather than on and off, on and off, etc.[18]

But how can the OS enforce these quanta? For example, how can the OS force the program **u** above to stop after 50 milliseconds? As discussed earlier, the answer is, "It can't! The OS is dead while **u** is running." Instead, the turns are implemented via a timing device, which emits a hardware interrupt at the proper time. For example, we could set the timer to emit an interrupt every 50 milliseconds. We would write a timer device driver, and incorporate it into the OS.

We will make such an assumption here. However, what is more common is to have the timer interrupt more frequently than the desired quantum size. On a PC, the 8253 timer interrupts 100 times per second. Every sixth interrupt, the Linux OS will perform a context switch. That results in a quantum size of 60 milliseconds. But this can be changed, simply by changing the count of interrupts needed to trigger a context switch.

The timer device driver saves all **u**'s current register values, including its PC value and the value in its EFLAGS register. Later, when **u**'s next turn comes, those values will be restored, and **u** will resume execution as if nothing ever happened. For now, though, the OS routine will restore **v**'s's previously-saved register values, making sure to restore the PC value last of all. That last action forces a jump from the OS to **v**, right at the spot in **v** where **v** was suspended at the end of its last quantum. (Again, the CPU just "minds its own business," and does not "know" that one program, the OS, has handed over control to another, **v**; the CPU just keeps performing its fetch/execute cycle, fetching whatever the PC points to, oblivious to which process is running.)

At any given time, there are many different **processes** in memory. These are instances of executions of programs. If for instance there are three users running the **gcc** C compiler right now on a given machine, here one program corresponds to three processes.

8.4.2 Example of OS Code: Linux for Intel CPUs

Here is a bit about how the context switch is done in Linux, in the version for Intel machines.[19]

The OS maintains a **process table**, which is simply an array of **struct**s, one for each process. The **struct** for

[17] We are switching from the "context" of one program to another.

[18] Think of a light bulb turning on and off extremely rapidly, with half the time on and half off. If it is blinking rapidly enough, you won't see it go on and off. You'll simply see it as shining steadily at half brightness.

[19] This was as of Linux kernel version 2.3. I have slightly modified some of this section for the sake of simplicity.

a process is called the Task State Segment (TSS) for that process, and stores various pieces of information about that process, such as the register values which the program had at the time its last turn ended. Remember, these need to be stored somewhere so that we can restore them at the program's next turn; here's where we store them.

As an example of the operations performed, and to show you concretely that the OS is indeed really a program with real code, here is a typical excerpt of code:

```
1  pushl %esi
2  pushl %edi
3  pushl %ebp
4  movl %esp, 532(%ebx)
5  ...
6  movl 532(%ecx), %esp
7  ...
8  popl %ebp
9  popl %edi
10 popl %esi
11 ...
12 iret
```

This code is the ISR for the timer. Upon entry, **u**'s turn has just ended, having been interrupted by the timer. In code not shown here, the OS has pointed the registers EBX and ECX to the TSSs of the process whose turn just ended, **u**, and the process to which we will give the next turn, say **v**.

By the way, how does the OS know that the process whose turn just ended was **u**'s? The answer is that whenever the OS gives a process a turn, it records which process it was. In Linux this is information is stored at address 0xffffe000, and the macro **GET_CURRENT()** is used to fetch it and place it into a register specified by the programmer; in our situation here, the register would be EBX. So the OS knows where to point EBX. The OS will make a decision as to which process to next give a turn to, and point ECX to it.

Here is what that code does. The source code for Linux includes a variable **tss**, which is the TSS **struct** for the current process. In that struct is a field named **esp**. So, **tss.esp** contains the previously-stored value of ESP, the stack pointer, for this process; this field happens to be located 532 bytes past the beginning of the TSS.[20]

Now, upon entry to the above OS code, ESP is still pointing to **u**'s stack, so the three PUSH instructions save **u**'s values of the ESI, EDI and EBP registers on **u**'s own stack.[21] The other register values of **u** must be saved too, including its value of ESP. The latter is done by the MOV, which copies the current ESP value, i.e. **u**'s ESP value, to **tss.esp** in **u**'s TSS. Other register saving is similar, though not shown here.

Now the OS must prepare to start **v**'s next turn. Thus **v**'s previously-saved register values must be restored to the registers. To understand how that is done, you must keep in mind that that same code we see above

[20] In the C source code, this field is referred to as **tss.esp**, but in assembly language we can only use the 532, as field names of **struct**s are not shown.

[21] Note the need to write this in assembly language instead of C, since C would not give us direct access to the registers or the stack. Most of Linux is written in C, but machine-dependent operations like the one here must be done in assembly language.

had been executed when **v**'s last turn ended. Thus **v**'s value of ESP is in **tss.esp** of its TSS, and the second MOV we see above copies that value to ESP. So, now we are using **v**'s stack.

Next, note similarly that at the end of **v**'s last turn, its values of ESI, EDI and EBP were pushed onto its stack, and of course they are still there. So, we just pop them off, and back into the registers, which is what those three POP instructions do.

Finally, what actually get's **v**'s new turn running? To answer this, note that the mechanism which made **v**'s last turn end was a hardware interrupt from the timer. At that time, the values of the Flags Register, CS and PC were pushed onto the stack. Now, the IRET instruction you see here pops all that stuff back into the corresponding registers. Note only does that restore registers, but since **v**'s old PC value is restored to the PC register, **v** is now running!

8.4.3 Process States

The OS maintains a **process table** which shows the state of each process in memory, mainly Run state versus Sleep state. A process which is in Run state means that it is ready to run but simply waiting for its next turn. The OS will repeatedly cycle through the process table, starting turns for processes which are in Run state but skipping over those in Sleep state. The processes in Sleep state are waiting for something, typically an I/O operation, and thus currently ineligible for turns. So, each time a turn ends, the OS will browse through its process table, looking for a process in Run state, and then choosing one for its next turn.

Say our application program **u** above contains a call to **scanf()** to read from the keyboard. Recall that **scanf()** calls the OS function **read()**. The latter is in the OS, so now the OS is running, and this function will check to see whether there are any characters ready in the keyboard buffer. Typically there won't be any characters there yet, because the user has not started typing yet. In this case the OS will place this process in Sleep state, and then start a turn for another process.

How does a process get switched to Run state from Sleep state? Say our application program **u** was in Sleep state because it was waiting for user input from the keyboard, waiting for just a single character, in **raw** mode.[22]

As explained earlier, when the user hits a key, that causes a hardware interrupt from the keyboard, which forces a jump to the OS. Suppose at that time program **v** happened to be in the midst of a quantum. The CPU would temporarily suspend **v** and jump to the keyboard driver in the OS. The latter would notice that the program **u** had been in Sleep state, waiting for keyboard input, and would now move **u** to Run state.

Note, though, that that does not mean that the OS now starts **u**'s next turn; **u** simply becomes eligible to run. Recall that each time one process' turn ends, the OS will select another process to run, from the set of all processes currently in Run state, and **u** will now be in that set. Eventually it will get a turn.

[22] Recall from our earlier unit that in this mode, a character is made available to a program as input the minute the user strikes the key. In ordinary, **cooked**, mode, the program doesn't see the input until the user hits the Enter key.

To make all this concrete, say **u** is running the **vi** text editor, **v** is running a long computational program, and user **w** is also running some long computational program. Remember, **vi** is a program; its source code might have a section like this:

```
while (1)  {
   KeyStroke = getch();
   if (KeyStroke == 'x') DeleteChar();
   else if (KeyStroke = 'j') CursorDown();
   else if ...
}
```

Here you see how **vi** would read in a command from the user, such as **x** (delete a character), **j** (move the cursor down one line) and so on. Of course, **DeleteChar()** etc. are functions, whose code is not shown here.

When during a turn for **u**, **vi** hits the **getch()** line, the latter calls **read()**, which is in the OS, so now the OS is running. Assuming **u** has not hit a key yet (almost certainly the case, since the CPU is a lot faster than **u**'s hand, not to mention the fact that **u** might have gone for a lunch break), the OS will mark **u**'s process as being in Sleep state in the OS's Process Table.

Then the OS will look for a process in Run state to give the next turn to. This might be, say, **v**. It will run until the timer interrupt comes. That pulse of current will cause the CPU to jump to the OS, so the OS is running again. This time, say, **w** will be run.

Say during **w**'s turn, **u** finally hits a key. The resulting interrupt from the keyboard again forces the CPU to jump to the OS. The OS read the character that **u** typed, notices in its Process Table that **u**'s process was in Sleep state pending keyboard input, and thus changes **u**'s entry in the Process Table to Run state. The OS then does IRET, which makes **w**'s process resume execution, but when the next timer interrupt comes, the OS will likely give **u** a turn again.

8.4.4 What About Background Jobs?

For example, you could run the Firefox Web browser by typing

```
% firefox
```

at the command line. But that would tie up the window you typed it into; you wouldn't be able to run any other commands until you quit Firefox, which might be a very long time. To solve this problem, you can run it *in the background*, as follows.

You could type

```
% firefox &
```

with the ampersand meaning, "Run this job in the background." The shell would start Firefox, and then print out a prompt, inviting you to run other commands. What is really happening here?

The first point to be made here is that it really means nothing to the OS. The entry for the Firefox process in the OS' process table will not state whether this program is background or foreground; the OS doesn't know.

The only meaning of that ampersand was for the shell. We used it to tell the shell, "Go ahead and launch **a.out** for me, but don't wait for it to finish before giving me your '%' prompt again. Give me the prompt right away, because I want to run some other programs too."[23]

8.4.5 Threads: "Lightweight Processes"

A very important type of application programming is that of **threads**. Thread libraries are offered by any modern OS. On Unix-family systems, for instance, a popular thread package is **pthreads**, which we will assume for concreteness here.

Threading is pervasive in computing today. Most Web servers are threaded, for instance. Also many GUI ("graphical user interface") programs are threaded too.

Threading is also the standard way of programming on multiprocessor systems. Since such systems are now common even at the household level—the term **dual core** means two CPUs—you can see how important threaded programming has become.

8.4.5.1 The Mechanics

Threads, of the "system" type discussed here, are quasi-processes.[24] If your program creates three threads on a Unix-family system such as Linux, for example, you may (depending on the OS) see three entries for them in the output of the **ps** command which displays processes. (For Linux, type **ps axH**. These will correspond to three entries in the process table. Thus the threads take turns running, in the usual timesharing manner. The turn taking includes the other processes too, of course.

From the programmer's point of view, the main difference between threads and processes is that in threaded programs all global variables of the program are shared.[25] This is the way the different threads communicate with each other, by reading and writing these shared variables.

[23] To do this, the shell must first create a new process, by calling the **fork()** system call, and then calling **execve()** from the child process. That way, the parent process—the shell–can keep running and give you the prompt.

[24] What we are describing here are **system-level** threads, i.e. those provided by the OS, which is the case for **pthreads**. There are also **user-level** threads. These, exemplified by the GNU **pth** library, are handled internally to the application program. Basically, the library runs its own private processes, with its own private process table!

[25] It is actually possible to have different processes share memory too, using the system call **shmget()**. However, this is unwieldy and involves a lot of system overhead, and is not commonly done.

This may come as a shock to readers who were taught that shared variables are "evil." In my opinion, shared variables are fine for general programming (within reason), but whether one agrees with that notion or not, **the fact is that in threaded programming one must use global variables.**

Each thread does have its own local variables, though. So, the various threads of a program will share the same **.data** and **ċomm** sections, but each will have its own separate stack.

One creates a thread by calling **pthread_create()**. One of its arguments must be a pointer to a function in the application program. For instance, in a GUI program, we might have one thread for the mouse, one thread for the keyboard, etc. The function which handles mouse events (movement and clicks) would be specified as an argument to **pthread_create()**, and basically the thread would consist of executing that function. There would also be a function for the keyboard, etc.

Key to threaded programming is **locks**. Suppose for instance that an airline reservation system is threaded, and that there is one seat left for a particular flight. We want to make sure to avoid a situation in which two clerks both sell that last seat. (Ignore the issue of overbooking.) So, the two threads running for the two clerks must cooperate, and not try to access that flight's record at the same time. In other words, we want access to that record to be **atomic**. Lock variables, of type **pthread_mutex_t**, are locked and unlocked via calls to **pthread_mutex_lock()** and **pthread_mutex_unlock()**, take care of this for us.

Threads are often called **lightweight processes**. The term *lightweight* here refers to the fact that creating a thread involves much less overhead than creating a process. In **pthreads**, for instance, calling **pthread_create()** has a lot less work to do than would be involved in calling **execve()** to make a new process. Among other things, the latter requires finding memory for the new process, whereas a new thread would simply use the same memory.

8.4.5.2 Threads Example

Examples of threaded programming tend to be rather elaborate, so I've written the following simple (if a bit contrived) example:

```
1  // HelloWorld.c, a baby-level first example of threaded programming.
2
3  // Doesn't actually print out "hello world." :-)  But it's of that level
4  // of simplicity, so I gave it that name anyway.
5
6  // The program reads from each of several keyboard input sources.  To
7  // avoid having to introduce network programming, we will have these
8  // sources be from several windows on the same console on the same
9  // machine.
10
11 // The issue here is that we don't know which keyboard input will have
12 // some activity next.  If we simply call scanf() in nonthreaded fashion
13 // (and don't resort to something called nonblocking I/O), our program
14 // would wait forever for input from one nonactive window when there is
15 // input waiting at another window.  The solution is to use threads,
```

```
16 // with one thread for each window.
17
18 // Each source of keyboard input is considered a different file.  (In
19 // Unix, I/O devices are considered "files.")  We determine the file
20 // name by running the "who am i" command in the given window.  If for
21 // instance the answer is pts/6, the keyboard input there is /dev/pts/6.
22
23 // Each time there is keyboard input from somewhere, the program will
24 // report what number was typed.
25
26 // For simplicity, the program has no mechanism for stopping.
27
28 // Linux compilation:  gcc -g -o hw HelloWorld.c -lpthread
29
30 // usage:  hw first_keybd_input_filename second_keybd_input_filename ...
31 // e.g. hw /dev/pts/4 /dev/pts/6
32
33 #define MAX_THREADS 20
34
35 #include <stdio.h>
36 #include <pthread.h>  // required for threads usage
37
38 int nthreads;  // number of threads to be run
39
40 // lock, needed so that the printf() in one thread does not rudely
41 // interrupt the printf() in another when a context switch occurs
42 pthread_mutex_t numlock = PTHREAD_MUTEX_INITIALIZER;
43 // ID structs for the threads
44 pthread_t id[MAX_THREADS];
45
46 // each thread runs this routine
47 void *worker(char *fname)  // fname is file name
48 {  FILE *kb;  // note separate stacks, so separate kb's
49    int num;  // separate num's too
50    kb = fopen(fname,"r");  // open this keyboard input, read only
51    while(1)  {
52       fscanf(kb,"%d",&num);  // wait for user to type in this window
53       pthread_mutex_lock(&numlock);
54       printf("read %d from %s\n",num,fname);
55       pthread_mutex_unlock(&numlock);
56    }
57 }
58
59 main(int argc, char **argv)
60 {  int i;
61    nthreads = argc - 1;
62    // get threads started
63    for (i = 0; i < nthreads; i++)  {
64       // this next call says to create a thread, record its ID in the array
65       // id, and get the thread started executing the function worker(),
66       // passing the argument argv[i+1] to that function
67       pthread_create(&id[i],NULL,worker,argv[i+1]);
68    }
69    while(1) ;  // dummy loop, to keep main() from exiting, killing
70                // the other threads
71 }
```

The program reads keyboard input from two different windows on the same console on the same machine. This is the contrived part. A typical example would have input coming from keyboards on other machines, with the characters being sent across a network. But I wanted to avoid bringing in network programming.

As you can see, **main()** starts up the threads. Once they are started, they take turns running, like processes. Note that **main()** is considered a thread too, the parent of those created by it.

I ran the program on the windows **/dev/pts/4** and **/dev/pts/6** on my Linux machine at home.[26] In each window I typed

```
% sleep 10000
```

to put the shell to sleep, ensuring that the input I type in those windows would go to my program rather than to the shells.

In a third window, I ran the program. In window 4 I then typed 5 and 12, and then 13 in window 6, then 168 in window 4 again. The output in the third window was

```
% a.out /dev/pts/4 /dev/pts/6
read 5 from /dev/pts/6
read 12 from /dev/pts/6
read 13 from /dev/pts/4
read 168 from /dev/pts/6
```

REMINDER TO THE READER: Don't worry so much about how to do **pthreads** programming; focus on the concepts relevant to our course. Think especially of the role of interrupts here. Threads take turns running, with a turn ending either from a timer interrupt or from a thread voluntarily relinquishing its turn via a system call, in this case **read()** being called by **fscanf()**. In the latter situation, an interrupt from the keyboard will change a thread from Sleep to Run state.

8.4.6 Making These Concepts Concrete: Commands You Can Try Yourself

First, try the **ps** command. On Unix systems, much information on current processes is given by the **ps** command, including:

- state (Run, Sleep, etc.)
- page usage (how many pages, number of page faults, etc.; see material on virtual memory below)
- ancestry (which process is the "parent" of the given process, via a call to **execve()**)

[26] As the comments in the code indicate, I determined their device names by running the **who am i** command in each window.

The reader is urged to try this out. You will understand the concepts presented here much better after seeing some concrete information which the ps command can give you. The format of this command's options varies somewhat from machine to machine (on Linux, I recommend **ps axms**), so check the **man** page for details, but run it with enough options that you get the fullest output possible.

Another command to try is **w.** One of the pieces of information given by the **w** command is the average number of processes in Run state in the last few minutes. The larger this number is, the slower will be the response time of the machine as perceived by a user, as his program is now taking turns together with more programs run by other people.

On Linux (and some other Unix) systems, you can also try the **pstree** command, which graphically shows the "family tree" (ancestry relations) of each process. For example, here is the output I got by running it on one of our CSIF PCs:

```
% pstree
init-+-atd
     |-crond
     |-gpm
     |-inetd---in.rlogind---tcsh---pstree
     |-kdm-+-X
     |     `-kdm---wmaker-+-gnome-terminal-+-gnome-pty-helpe
     |                    |                `-tcsh-+-netscape-commun---netscape-+
     |                    |                       `-vi
     |                    |-2*[gnome-terminal-+-gnome-pty-helpe]
     |                    |                   `-tcsh]
     |                    |-gnome-terminal-+-gnome-pty-helpe
     |                    |                `-tcsh---vi
     |                    `-wmclock
     |-kerneld
     |-kflushd
     |-klogd
     |-kswapd
     |-lpd
     |-6*[mingetty]
     |-2*[netscape-commun---netscape-commun]
     |-4*[nfsiod]
     |-portmap
     |-rpc.rusersd
     |-rwhod
     |-sendmail
     |-sshd
     |-syslogd
     |-update
     |-xconsole
     |-xntpd
     `-ypbind---ypbind
%
```

On Unix systems, the first thing the OS does after bootup is to start a process named **init**, which will be the parent (or grandparent, great-grandparent, etc.) of all processes. That **init** process then starts several OS **daemons**. A **daemon** in Unix parlance means a kind of server program. In Unix, the custom is to name

such programs with a 'd' at the end, standing for "daemon."

For example, you can see above that **init** started **lpd** ("line printer daemon"), which is the print server.[27] When users issue **lpr** and other print commands, the OS refers them to **lpd**, which arranges for the actual printing to be done.

Often daemons spawn further processes. For example, look at the line

```
|-inetd---in.rlogind---tcsh---pstree
```

The **inetd** is an Internet request server. The system administrator can optionally run this daemon in lieu of some other network daemons which are assumed to run only rarely. Here the user (me) had done **rlogin**, a login program like **ssh**[28], to remotely login to this machine. The system administrators had guessed that **rlogin** would be used only rarely, so they did not have **init** start the corresponding daemon, **in.rlogind**, upon bootup. Instead, any network daemon not started at bootup will be launched by **inetd** when the need arises, which as you can see occurred here for **in.rlogind**. The latter then started a shell for the user who logged in (me), who then ran **pstree**. Note again that the shell is the entity which got **pstree** started.

8.5 Virtual Memory

8.5.1 Make Sure You Understand the Goals

Now let us add in the effect of virtual memory (VM). VM has the following basic goals, which are so important to understand that I am describing in separate short sub-subsections:

8.5.1.1 Overcome Limitations on Memory Size

We want to be able to run a program, or collectively several programs, whose memory needs are larger than the amount of physical memory available.

8.5.1.2 Relieve the Compiler and Linker of Having to Deal with Real Addresses

We want to facilitate **relocation** of programs, meaning that the compiler and linker do not have to worry about where in memory a program will be loaded when it is run.

[27] Another common print server daemon is **cupsd**.

[28] But no longer allowed on CSIF.

8.5.1.3 Enable Security

We want to ensure that one program will not accidentally (or intentionally) harm another program's operation by writing to the latter's area of memory, read or write to another program's I/O streams, etc.

8.5.2 The Virtual Nature of Addresses

The word *virtual* means "apparent." It will appear that a program resides entirely in main memory, when in fact only part of it is there. It will appear that your program variables are stored at the places the linker assigned to them,[29] when in fact they actually will be stored somewhere else.

(For the time being, it will be easier to understand VM by assuming there is no cache. We will return to this in Section 8.5.8.)

To make this more concrete, suppose our C source file includes the following statements:

```
int x;
...
x = 8;
printf("%d",&x);
```

Recall that addresses of variables are set in the following manner. Say **x** is global. Then the compiler will generate a **.comm** line in assembly language it produces.[30] The latter will be translated by the assembler, and then the linker will deal with the machine code. Among other things, the linker will assign an address for **x**. It will be this address which is printed out above.

If **x** had been local, it would be stored on the stack, but that is still memory of course, and thus it would have an address too. Again, that would be printed out.

In any case, either of these addresses would be fake. Let's see what this really means. Suppose that **x** is global, and that the compiler and linker assign it address 200, i.e. **&x** is 200. Then the above code

```
x = 8;
```

would correspond to an instruction in **a.out** something like

```
movl $8, 200
```

At the time **a.out** is loaded by the OS into memory, the OS will divide both the **.text** (instructions) and data portions of **a.out** (**.data**, **.bss**, stack, heap, etc.) into chunks, and find unused places in memory at which to

[29] You can determine these by running **readelf** on your executable file, with the **-s** option.

[30] If **x** had been declared in initialzed form, e.g. **int x = 28;** then it would have been in a **.data** section.

place these chunks. The chunks are called **pages** of the program, and the same-sized places in memory in which the OS puts them are called pages of memory.[31] The OS sets up a **page table** for this process, which is an array which is maintained by the OS, in which the OS records the correspondences, i.e. lists which page of the process is stored in which page of memory. Remember, the OS is a program, and the page table is an array in that program, and thus it too is in memory.

What appears to be in word 200 in memory from the program code above may actually be in, say, word 1204. At the time the CPU executes that instruction, the CPU will determine where 'Word 200" really is by doing a lookup in the page table. In our example here, the table will show that the item we want is actually in word 1204, and the CPU will then write to that location. The hardware would put 1204 on the address bus, 8 on the data bus, and would assert the Memory Write line in the control bus.

In this example, we say the **virtual address** is 200, and the **physical address** is 1204.

Note that while virtual addressing is enabled (it is turned off in Kernel Mode), all addresses seen by the hardware will be virtual. For example, the contents of ESP will be a virtual address, with the real top-of-stack address being something else, not what ESP says. A similar statement holds for the PC, etc.

8.5.3 Overview of How the Goals Are Achieved

Let's look at our stated goals in Section 8.5.1 above, and take a first glance at how they are achieved (details will be covered later):

8.5.3.1 Overcoming Limitations on Memory Size

To conserve memory space, the OS will initially load only part of **a.out** into memory, with the remainder being left back on disk. The pages of the program which are not loaded will be marked in the page table as currently being **nonresident**, meaning not in memory, and their locations on disk will be shown in the table. During execution of the program, if the program needs one of the nonresident pages, the CPU will notice that the requested page is nonresident. This is called a **page fault**, and it causes an internal interrupt. The interrupt number is 0xe, i.e. entry 0xe in the IDT. That causes a jump to the OS, which will bring in that page from disk, and then jump back to the program, which will resume at the instruction which accessed the missing page.

Note that pages will often go back and forth between disk and memory in this manner. Each time the program needs a missing page, that page is brought in from disk and a page which had been resident is written back to disk in order to make room for the new one. Note also that a given virtual page might be in different physical pages at different times.

[31] As with the stack, etc., keep in mind that these pages are only conceptual; there is no "fence" or any other physical demarcation between one page in memory and the next. A page is simply a unit of measurement, like acres and liters.

A big issue is the algorithm the OS uses to decide which page to move back to disk (i.e. which page to replace) whenever it brings a page from disk after a page fault. Due to the huge difference in CPU and disk speeds, a page fault is a catastrophic even in terms of program speed. We hope to have as few page faults as possible when our program runs.

So, we want to check the pages to evict very carefully. If we often evict a page which is needed again very soon, our program's performance will really suffer. This is called **thrashing**. Details of page replacement policies are beyond the scope of this document here, but one point to notice is that the policy will be chosen so as to work well on "most" programs. For any given policy, it will do well on some programs (i.e. produce few page faults) while doing poorly on some other programs (produce many page faults).

So, what they try to do is come up with a policy which works reasonably well on a reasonably broad variety of programs. Most policies are some variation on the Least Recently Used policy common for associative caches.

8.5.3.2 Relieving the Compiler and Linker of Having to Deal with Real Addresses

This is clear from the example above, where the location "200" which the compiler and linker set up for **x** was in effect changed by the OS to 1204 at the time the program was loaded. The OS recorded this in the page table, and then during execution of the program, the VM hardware in the CPU does lookups in the page table to get the correct addresses. The point is that the compiler and linker can assign **x** to location 200 without having to worry whether location is actually available at the time the program will be run, because that variable actually **won't** be at 200.

8.5.3.3 Enabling Security

The page table will consist of one entry per page. That entry will, as noted earlier, include information as to where in memory that page of the program currently resides, of if currently nonresident, where on disk the page is stored. But in addition, the entry will also list the permissions the program has to access this particular page—read, write, execute—in a manner analogous to file-access permissions. If . an **access violation** occurs, the VM hardware in the CPU will cause an internal interrupt (again it's interrupt number 0xe, as for page faults), causing the OS to run. The OS will then kill the process, i.e. remove it from the process table.

Normally data-related sections such as **.data** and the stack will be the only ones with write permission. However, you can also arrange for the **.text** section to be writable, via the **-N** option to **ld**.

You might wonder why execute permission is included. One situation in which this would be a useful check is that in which we have a pointer to a function. If for example we forgot to initialize the pointer, a violation will be detected.

Suppose for instance your program tries to read from virtual page number 40000 but that virtual address is out of the ranges of addresses which the program is allocated. The hardware will check the entry for 40000 in the page table, and find that not only is that page not resident, but also it isn't on disk either. Instead, the entry will say that there is no virtual page 40000 among the pages allocated to this program.

We don't want an ordinary user programs to be able to, say maliciously, access the I/O streams of other programs. To accomplish this, we want to forbid a user program from accessing the I/O buffer space of another program, which is easy to to accomplish since that space is in memory; we simply have the OS set up the page table for that program accordingly. But we also need to forbid a user program from directly performing I/O, e.g. the INB and OUTB instructions on Intel machines.

This latter goal is achieved by exploiting the fact that most modern CPUs run in two or more **privilege levels**. The CPU is designed so that certain instructions, for example those which perform I/O such as INB and OUTB, can be executed only at higher privilege levels, say Kernel Mode. (The term **kernel** refers to the OS.)

But wait! Since the OS is a program too, how does *it* get into Kernel Mode? Obviously there can't be some instruction to do this; if there were, then ordinary user programs could use it to put themselves into Kernel Mode. So, how is it done?

When the machine boots up, it starts out in Kernel Mode. Thus the OS starts out in Kernel Mode. Among other things, the interrupts can be configured to change the privileges level of the CPU. Recall that the OS starts a turn for a user program (including that program's first turn) by executing an IRET instruction, a software interrupt. So, the OS can arrange both for user programs to run in non-Kernel Mode and for Kernel Mode to be restored when an interrupt comes during a user program is running. So for example, an interrupt from the timer will not only end a user program's turn, but also will place the CPU in Kernel Mode, thus having the OS run in that mode.

8.5.4 Who Does What When?

Note carefully the roles of the players here: It is the software, the OS, that creates and maintains the page table, but it is the hardware that actually uses the page table to generate physical addresses, check page residency and check security. In the event of a page fault or security violation, the hardware will cause a jump to the OS, which actually responds to those events.

The OS writes to the page table (including creating it in the first place), and the hardware reads it.

The hardware will have a special Page Table Register (PTR) to point to the page table of the current process. When the OS starts a turn for a process, it will restore the previously-saved value of the PTR, and thus this process' page table will now be in effect.

On the Pentium, the name of the PTR is CR3. (Actually, the Pentium uses a two-level hierarchy for its page tables, but we will not pursue that point here.)

8.5.5 Details on Usage of the Page Table

8.5.5.1 Virtual-to-Physical Address Translation, Page Table Lookup

Whenever the running program generates an address—either the address of an instruction, as will be the case for an instruction fetch, or the address of data, as will be the case during the execution of instructions which have memory operands—this address is only virtual. It must be translated to the physical address at which the requested item actually resides. The circuitry in the CPU is designed to do this translation by performing a lookup in the page table.

For convenience, say the page size is 4096 bytes, which is the case for Pentium CPUs. Both the virtual and physical address spaces are broken into pages. For example, consider virtual address 8195. Since

$$8195 = 2 \times 4096 + 3 \tag{8.1}$$

that address would be in virtual page 2 (the first page is page 0). We can then speak of how far into a page a given address is. Here, because of the remainder 3 in Equation (8.1), we see that virtual address 8195 is byte 3 in virtual page 2. We refer to the 3 as the **offset** within the page, i.e. its distance from the beginning of the page.

You can see that for any virtual address, the virtual page number is equal to the address divided by the page size, 4096, and its offset within that page is the address mod 4096. Using our knowledge of the properties of powers of 2 and the fact that $4096 = 2^{12}$, this means that for a 32-bit address (which we'll assume throughout), the upper 20 bits contain the page number and the lower 12 bits contain the offset.

The page/offset description of the position of a byte is quite analogous to a feet/inches description of distance. We could measure everything in inches, but we choose to use a larger unit, the foot, to give a rough idea of the distance, and then use inches to describe the remainder. The concept of offset will be very important in what follows.

Now, to see how the page table is used to convert virtual addresses to physical ones, consider for example the Intel instruction

```
movl $6, 8195
```

This would copy the constant 6 to location 8195. Remember, this is page 2, offset 3. However, it is a virtual address. The hardware would see the 8195,[32] The hardware knows that any address given to it is a virtual one, which must be converted to a physical one. So, the hardware would look at the entry for virtual page 2, and find what physical page that is; suppose it is 5. Then that page starts at physical memory location $5 \times 4096 = 20480$.

[32] Remember, it will be embedded within the instruction itself, as this is direct addressing mode.

What about the offset within that physical page 5? The custom is that an item will have the same offset, no matter whether we are talking about its virtual address or its physical one. So, the offset of our destination in physical page 5 would be 3. In other words, the physical address is

$$5 \times 4096 + 3 = 20483 \tag{8.2}$$

The CPU would now be ready to execute the instruction, which, to refresh your memory, was

```
movl $6, 8195
```

The CPU knows that the real location of the destination is 20483, not 8195. It would put 20483 on the address bus, put 6 on the data bus, and assert the Memory Write line in the control bus, and the instruction would be done.

8.5.5.2 Layout of the Page Table

Suppose the entries in our page table are 32 bits wide, i.e. one word per entry.[33] Let's label the bits of an entry 31 to 0, where Bit 31 is in the most-significant (i.e. leftmost) position and Bit 0 is in the least significant (i.e. rightmost) place. Suppose the format of an entry is as follows:

- Bits 31-12: physical page number if resident, disk location if not
- Bit 11: 1 if page is resident, 0 if not
- Bit 10: 1 if have read permission, 0 if not
- Bit 9: 1 if have write permission, 0 if not
- Bit 8: 1 if have execute permission, 0 if not
- Bit 7: 1 if page is "dirty," 0 if not (see below)
- Bits 6-0: other information, not discussed here

Now, here is what will happen when the CPU executes the instruction

```
movl $6, 8195
```

[33] If we were to look at the source code for the OS, we would probably see that the page table is stored as a very long array of type **unsigned int**, with each array element being one page table entry.

above:

- The CPU does the computation $\times 4096 + 3 = 20483$ and finds that virtual addrss 8195 is virtual page number 2, offset 3.
- Since we are dealing with virtual page 2, the CPU will go to get the entry for that virtual page in the page table, as follows. Suppose the contents of the PTR is 5000. Then since each entry is 4 bytes long, the table entry of interest here, i.e. the entry for virtual page 2, is at location

$$5000 + 4 \times 2 = 5008 \tag{8.3}$$

  The CPU will read the desired entry from that location, getting, say, 0x000005e0.
- The CPU looks at Bits 11-8 of that entry, getting 0xe, finding that the page is resident (Bit 11 is 1) and that the program has read and write permission (Bits 10 and 9 are 1) but no execute permission (Bit 8 is 0). The permission requested was write, so this is OK.
- The CPU looks at Bits 31-12, getting 5, so the hardware would know that virtual page 2 is actually physical page 5. The virtual offset, which we found earlier to be 3, is always retained, so the CPU now knows that the physical address of the virtual location 8195 is

$$5 \times 4096 + 3 = 20483 \tag{8.4}$$

- The CPU puts the latter on the address bus, puts 6 on the data bus, and asserts the Write line in the bus. This writes 6 to memory location 20483, and we are done.

By the way, all this was for Step C of the above MOV instruction. The same actions would take place in Step A. The value in the PC would be broken down into a virtual page number and an offset; the virtual page number would be used as an index into the page table; Bits 10 and 8 in the page table element would be checked to see whether we have permission to read and execute that instruction; assuming the permissions are all right, the physical page number would be obtained from Bits 31-12 of the page table element; the physical page number would be combined with the offset to form the physical address; and the physical address would be placed in the MAR and the instruction fetched.

Recall from above that the upper 20 bits of an address form the page number, and the lower 12 bits form the offset. A similar statement holds for physical addresses and physical page numbers. So, all the hardware need do is: use the upper 20 bits of the virtual address as an index in the page table (i.e. multiply this by 4 and add to c(PTR); take bits 31-12 of from the table entry reached in this manner, to get the physical page number; and finally, concatenate this physical page number with the lower 12 bits of the original virtual address. Then the hardware has the physical address, which it places on the address bus.

8.5.5.3 Page Faults

Suppose in our example above Bit 11 of the page table entry had been 0, indicating that the requested page was not in memory. As mentioned earlier, this event is known as a **page fault**. If that occurs, the CPU will perform an internal interrupt, and will also record the PC value of the instruction which caused the page fault, so that that instruction can be restarted after the page fault is processed. In Pentium CPUs, the CR2 register is used to store this PC value. This will force a jump to the OS.

The OS will first decide which currently-resident page to replace, then will write that page back to disk, if the Dirty Bit is set (see below). The OS will then bring in the requested page from disk. The OS would then update two entries in the page table: (a) it would change the entry for the page which was replaced, changing Bit 11 to 0 to indicate the page is not resident, and changing Bits 31-12 and possible Bit 7; and (b) the OS would update the page table entry of the new item's page, to indicate that the new item is resident now in memory (setting Bit 11 to 1), show where it resides (by filling in Bits 31-12), and setting Bit 7 to 0.

The role of the Dirty Bit is as follows: When a page is brought into memory from disk, this bit will be set to 0. Subsequently, if the page is written to, the bit will be changed to 1. So, when it comes time to evict the page from memory, the Dirty Bit will tell us whether there is any discrepancy between the contents of this page in memory and its copy on disk. If there is a difference, the OS must write the new contents back to disk. That means all 4096 bytes of the page. We must write back the whole page, because we don't know what bytes in the page have changed. The Dirty Bit only tells us that there has been some change(s), not where the change(s) are. So, if the Dirty Bit is 0, we avoid a time-consuming disk write.

Since accessing the disk is far, far slower than accessing memory, a program will run quite slowly if it has too many page faults. If for example your PC at home does not have enough memory, you will find that you often have to wait while a large application program is loading, during which time you can hear the disk drive doing a lot of work, as the OS ejects many currently-resident pages to bring in the new application.

8.5.5.4 Access Violations

If on the other hand an access violation occurs, the OS will announce an error—in Unix, referred to as a **segmentation fault**—and kill the process, i.e. remove it from the process table.

For example, considering the following code:

```
1  int q[200];
2
3  main()
4
5  {  int i;
6
7     for (i = 0; i < 2000; i++) {
8        q[i] = i;
9     }
```

```
10
11  }
```

Notice that the programmer has apparently made an error in the loop, setting up 2000 iterations instead of 200. The C compiler will not catch this at compile time, nor will the machine code generated by the compiler check that the array index is out of bounds at execution time.

If this program is run on a non-VM platform,[34] then it will merrily execute without any apparent error. It will simply write to the 1800 words which follow the end of the array **q**. This may or may not be harmful, depending on what those words had been used for.

But on a VM platform, in our case Unix, an error will indeed be reported, with a "Segmentation fault" message.[35] However, as we look into how this comes about, the timing of the error may surprise you. The error is not likely to occur when **i** = 200; it is likely to be much later than that.

To illustrate this, I ran this program under **gdb** so that I could take a look at the address of **q[199]**.[36] After running this program, I found that the seg fault occurred not at **i** = 200, but actually at **i** = 728. Let's see why.

From queries to **gdb** I found that the array **q** ended at 0x080497bf, i.e. the last byte of **q[199]** was at that address. On Intel machines, the page size is 4096 bytes, so a virtual address breaks down into a 20-bit page number and a 12-bit offset, just as in Section 8.5.5.1 above. In our case here, **q** ends in virtual page number 0x8049 = 32841, offset 0x7bf = 1983. So, after **q[199]**, there are still 4096-1984 = 2112 bytes left in the page. That amount of space holds 2112/4 = 528 **int** variables, i.e. elements "200" through "727" of **q**. Those elements of **q** don't exist, of course, but as discussed in an earlier unit the compiler will not complain. Neither will the hardware, as we will be writing to a page for which we do have write permission. But when **i** becomes 728, that will take us to a new page, one for which we don't have write (or any other) permission; the hardware will detect this and trigger the seg fault.

We could get a seg fault not only by accessing off-limits data items, but also by trying to execute code at an off-limits location. For example, suppose in the above example **q** had been local instead of global. Then it would be on the stack. As we go past the end of **q**, we would go deeper and deeper into the stack. This may not directly cause a seg fault, if the stack already starts out fairly large and is stored in physically contiguous pages in memory. But we would overwrite all the preceding stack frames, including return addresses. When we tried to "return" to those addresses,[37] we would likely attempt to execute in a page for which we do not have execute permission, thus causing a seg fault.

As another example of violating execute permission, consider the following code, with a pointer to a func-

[34] Recall that "VM platform" requires both that our CPU has VM capability, and that our OS uses this capability.

[35] On Microsoft Windows systems, it's called a "general protection error."

[36] Or I could have added a **printf()** statement to get such information. Note by the way that either running under **gdb** or adding **printf()** statement will change the load locations of the program, and thus affect the results.

[37] Recall that **main()** is indeed called by other code, as explained in our unit on subroutines.

tion:[38]

```
1  int f(int x)
2  { return x*x; }
3
4  // review of pointers to functions in C/C++: below p is a pointer to a
5  // function; the first int means that whatever function p points to, it
6  // returns an int value; the second int means that whatever function p
7  // points to, it has one argument, an int
8  int (*p)(int);
9
10 main()
11 {  int u;
12
13    p = f;  // point p to f
14    u = (*p)(5);  // call f with argument 5
15    printf("%d\n",u);  // prints 25
16 }
```

If we were to forget to include the line

```
u = (*p)(5);
```

then the variable **p** would not point to a function, and we would attempt to execute "code" in a location off limits to us. A seg fault would result.

8.5.6 VM and Context Switches

A context switch will be accompanied by a switch to a new page table.

Say for example Process A has been running, but its turn ends and Process B is given a turn. Of course we must now use B's page table. So, before starting B's turn, the OS must change the PTR to point to B's page table.

Thus, there is actually no such thing as "the" page table. There are many page tables. The term "the" page table merely means the page table which PTR currently points to.

8.5.7 Improving Performance—TLBs

Virtual memory comes at a big cost, in the form of overhead incurred by accessing the page tables. For this reason, the hardware will also typically include a **translation lookaside buffer** (TLB). This is a special

[38] If your classes on C/C++ did not cover this important topic of pointers to functions, the comments in the code below should be enough to introduce it to you. Pointers to functions are used in many applications, such as threaded programs, as mentioned earlier.

cache to keep a copy of part of the page table in the CPU, to reduce the number of times one must access memory, where the page table resides.

One might ask, why not store the entire page table in the TLB? First of all, the page table theoretically consists of $2^{32}/2^{12}$ entries. This is somewhat large. Though there are ways around this, but we would need to change this large amount of data at each context switch, which would really slow down context switching. Note by the way that we would need to have special instructions which the OS could use to write to the TLB. That's OK, though, and there are other special instructions for the OS already.

8.5.8 The Role of Caches in VM Systems

Up to now, we have been assuming that the machine doesn't have a cache.[39] But in fact most machines which use VM also have caches, and in such cases, what roles do the caches play?

The central point is that speed is still an issue. The CPU will look for an item in its cache first, since the cache is internal to (or at least near) the CPU. If there is a cache miss, the CPU will go to memory for the item. If the item is resident in memory, the entire block containing the item will be copied to the cache. If the item is not resident, then we have a page fault, and the page must be copied to memory from disk, a very slow process, before the processing of the cache miss can occur.

8.5.8.1 Addressing

An issue that arises is whether the cache is the CPU will **virtually addressed** or **physically addressed**. Suppose for instance the instruction being executed reads from virtual address 200. If the cache is virtually addressed, then the CPU would do its cache lookup using 200 as its index. On the other hand, if the cache is physically addressed, then the CPU would first convert the 200 to its physical address (by checking the TLB, and then the page table if need be), and then do the cache lookup based on that address.

A possible advantage of virtual addressing of caches would be that if we have a cache hit, we eliminate the time delay needed for the virtual-to-physical address translation. Moreover, since we usually will have a hit, we can afford to put the TLB in not-so-fast memory external to the CPU, which is done in some MIPS models.

On the other hand, with physical cache addressing, two different processes could both have separate, unrelated copies of different blocks in the cache at the same time, but with the same virtual addresses. They could coexist in the cache since their physical addresses would be different. That would mean we would not have to flush the cache at each new timeslice, possible increasing performance.

[39] Except for a TLB, which is of course very specialized.

8.5.8.2 Hardware Vs. Software

Note that cache design is entirely a hardware issue. The cache lookup and the block replacement upon a miss is hard-wired into the circuitry. By contrast, in the VM case it is a mixture of hardware and software. The hardware does the page table lookup, and checks page residency and access permissions. But it is the software—the OS—which creates and maintains the page table. Moreover, when a page fault occurs, it is the OS that does the page replacement and updates the page table.

So, you could have two different versions of Unix, say,[40] running on the same machine, using the same compiler, etc. and yet they may have quite different page fault rates. The page replacement policy for one OS may work out better for this particular program than does the one of the other OS.

Note that the OS can tell you how many page faults your program has had (see the **time** command below); each page fault causes the OS to run, so the OS can keep track of how many page faults your program had incurred. By contrast, the OS can NOT keep track of how many cache misses your program has had, since the OS is not involved in handling cache misses; it is done entirely by the hardware.

8.5.9 Making These Concepts Concrete: Commands You Can Try Yourself

The Unix **time** command will report how much time your program takes to run, how many page faults it generated, etc. Place it just before your program's name on the command line. (This program could be either one you wrote, or something like, say, **gcc**.) For example, if you have a program **x** with argument 12, type

```
% time x 12
```

instead of

```
% x 12
```

In fact, trying running this several times in quick succession. You main find a reduction in page faults, since the required pages which caused page faults in the first run might be resident in later runs.

Also, the **top** program is very good, displaying lots of good information on the memory usage of each process.

[40] Or, Linux versus Windows, etc.

8.6 A Bit More on System Calls

Recall that the OS makes available to application programs services such as I/, process management, etc. It does this by making available functions that application programs can call. Since these are function in the OS, calls to them are known as **system calls**.

When you call **printf()**, for instance, it is just in the C library, not the OS, but it in turn calls **write()** which *is* in the OS.[41] The call to **write()** is a system call.

Or you can make system calls in your own code. For example, try compiling and running this code:

```
main()
{ write(1,"abc\n",4); }
```

The function **write()** takes three arguments: the file handle (here 1, for the Unix "file" **stdout**, i.e. the screen); a pointer to the array of characters to be written [note that NULL characters mean nothing to **write()**]; and the number of characters to be written.

Similarly, executing a **cout** statement in C++ ultimately results in a call to **write()** too. In fact, the G++ compiler in GCC translates **cout** statements to calls to **printf()**, which as we saw calls **write()**. Check this by writing a small C++ program with a **cout** in it, and then running the compiled code under **strace**.

A non-I/O example familiar to you is the **execve()** service is used by one program to start the execution of another. Another non-I/O example is **getpid()**, which will return the process number of the program which calls it.

Calling **write()** means the OS is now running, and **write()** will ultimately result in the OS running code which uses the OUT machine instruction. Recall, though, that we want to arrange things so that only can execute instructions like Intel's IN and OUT. So the hardware is designed so that these instructions can be executed only in Kernel Mode.

For this reason, one usually cannot implement a system call using an ordinary subroutine CALL instruction, because we need to have a mechanism that will change the machine to Kernel Mode. (Clearly, we cannot just have an instruction to do this, since ordinary user programs could execute this instruction and thus get into Kernel Mode themselves, wreaking all kinds of havoc!) Another problem is that the linker will not know where in the OS the desired subroutine resides.

Instead, system calls are implemented via an instruction type which is called a **software interrupt**. On Intel machines, this takes the form of the INT instruction, which has one operand.

We will assume Linux in the remainder of this subsection, in which case the operand is 0x80.[42] In other

[41] There is a slight complication here. Calling **write()** from C will take us to a function in the C library of that name. It in turn will call the OS function. We will ignore this distinction here.

[42] Windows uses 0x21.

words, the call to **write()** in your C program (or in **printf()** or **cout**, which call **write()** will execute

```
...  # code to put parameters values into designated registers
int $0x80
```

The INT instruction works like a hardware interrupt, in the sense that it will force a jump to the OS, and change the privilege level to Kernel Mode, enabling the OS to execute the privileged instructions it needs. You should keep in mind, though, that here the "interrupt" is caused deliberately by the program which gets "interrupted," via an INT instruction. This is much different from the case of a hardware interrupt, which is an action totally unrelated to the program which is interrupted.

The operand, 0x80 above, is the analog of the device number in the case of hardware interrupts. The CPU will jump to the location indicated by the vector at c(IDT)+8*0x80.[43]

When the OS is done, it will execute an IRET instruction to return to the application program which made the system call. The **iret** also makes a change back to User Mode.

As indicated above, a system call generally has parameters, just as ordinary subroutine calls do. One parameter is common to all the services—the service number, which is passed to the OS via the EAX register. Other registers may be used too, depending on the service.

As an example, the following Intel Linux assembly language program writes the string "ABC" and an end-of-line to the screen, and then exits:

```
.text
hi:  .string "ABC\n"
_start:

      # write "ABC\n" to the screen
      movl $4, %eax     # the write() system call, number 4 obtained
                        # from /usr/include/asm/unistd.h
      movl $1, %ebx     # 1 = file handle for stdout
      movl $hi, %ecx    # write from where
      movl $4, %edx     # write how many bytes
      int $0x80         # system call

      # call exit()
      movl $1, %eax     # exit() is system call number 1
      int $0x80         # system call
```

For this particular OS service, **write()**, the parameters are passed in the registers EBX, ECX and EDX (and, as mentioned before, with EAX specifying which service we want). Whoever wrote **write()** has forced us to use those registers for those purposes.

[43] By the way, users could cause mischief by changing this area of memory, so the OS would set up their page tables to place it off limits.

Note that we do indeed need to tell **write()** how many bytes to write. It will NOT stop if it encounters a null character.

Here are some examples of the numbers (to be placed in EAX before calling **int $0x80**) of other system services:

```
read              3
file open         5
execve           11
chdir            12
kill             37
```

8.7 OS File Management

The OS will maintain a table showing the starting sectors of all files on the disk. (The table itself is on the disk.) The reason that the table need store only the starting sector of a given file is that the various sectors of the file can be linked together in "linked-list" fashion. In other words, at the end of the first sector of a file, the OS will store information stating the track and sector numbers of the next sector of the file.

The OS must also maintain a table showing unused sectors. When a user creates a new file, the OS checks this table to find space to put the file. Of course, the OS must then update its file table accordingly.

If a user deletes a file, the OS will update both tables, removing the file's entry in the file table, and adding the former file's space to the unused sector table.[44]

As files get created and deleted throughout the usage of a machine, the set of unused sectors typically becomes like a patchwork quilt, with the unused sectors dispersed at random places all around the disk. This means that when a new file is created, then *its* sectors will be dispersed all around the disk. This has a negative impact on performance, especially seek time. Thus OSs will often have some kind of **defragmenting** program, which will rearrange the positioning of files on the disk, so that the sectors of each individual file are physically close to each other on the disk.

8.8 To Learn More

There are several books on the Linux OS kernel. Before you go into such detail, though, you may wish to have a look at the Web pages `http://www.tldp.org/LDP/tlk/tlk.html` and `http://learnlinux.tsf.org.za/courses/build/internals/`, which appear to be excellent overviews.

[44] However, the file contents are still there. This is how "undelete" programs work, by attempting to recover the sectors liberated when the user (accidentally) deleted the file.

8.9 Intel Pentium Architecture

We will assume Pentia with 32-bit word and address size. On Linux the CPU runs in **flat mode**, meaning that the entire address space of 2^{32} bytes is available to the programmer.

The main registers of interest to us here are named EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP.[45] ESP is used as the stack pointer. The program counter is EIP. There is also a condition codes register EFLAGS, which is used to record whether the results of operations are zero, negative and so on.

In this document we use AT&T syntax for assembly code. Here is some sample code, which adds 4 elements of an array pointed to by ECX, with the sum being stored in EBX:

```
1       movl $4, %eax  # copy the number 4 to EAX
2       movl $0, %ebx  # copy the number 0 to EBX
3       movl $x, %ecx  # copy the address of the memory location
4                      # whose label is x to ECX
5  top: addl (%ecx), %ebx  # add the memory word pointed to by ECS to EBX
6       addl $4, %ecx  # add the number 4 to ECS
7       decl %eax  # decrement EAX by 1
8       jnz top  # if EAX not 0, then jump to top
```

Pentia use vector tables for interrupt handlers. Typical Pentium-based computers include one or more Intel 8259A chips for controlling and prioritizing bus access by input/output devices.

As with most modern processor chips, Pentia are highly pipelined and superscalar, the latter term meaning muliple ALU compoents, thus allowing more than one instruction to execute per clock cycle.

For more information see the various PDF files in `http://heather.cs.ucdavis.edu/~matloff/50/PLN`.

[45] We will use the all-caps notation, EAX, EBX, etc. to discuss registers in the text, even though in program code they appear as **%eax**, **%ebx**, ...

Chapter 9

Example of RISC Architecture: MIPS

9.1 Introduction

The term **RISC** is an acronym for reduced instruction set computer, the antonym being **CISC**, for complex instruction set computer.

During the 1970s and early 1980s, computers became more and more CISC-like, with richer and richer instruction sets. The most cited example is that of the VAX family (highly popular in the early 1980s), whose architecture included 304 instructions, including such highly specialized instructions as **poly**, which evaluates polynomials. By contrast, the UC Berkeley RISC I architecture, which later became the basis for what is now the commercial SPARC chip in Sun Microsystems computers, had only 32 instructions.

At the same time, the single-chip CPU was starting out on its path to dominance of the market. The more components a computer has, the more expensive it is to manufacture, so single-chip CPUs became very attractive.[1] But the problem then became that the very limited space on a chip—this space is colloquially called "real estate"—made it very difficult, if not impossible, to fit a CISC architecture onto a single chip.[2]

The VAX was a clear casualty of this. When the Digital Equipment Corporation (DEC) tried to produce a "VAX-on-a-chip," they found that they could not fit the entire instruction set on one chip. They were then forced to implement the remaining instructions in software. For example, when the CPU encountered a **poly** instruction, that would cause an "illegal op code" condition, which triggered a **trap** (i.e. internal interrupt); the trap-service routine would then be a procedure consisting of ordinary **add** and **mul** instructions which compute the polynomial.

[1] And there is a performance gain as well. Multi-chip CPUs suffer from the problem that off-chip communication is slower than within-chip.

[2] It would seem that the solution is to simply make larger chips. However, this idea has not yet worked. For example, larger chips have lower **yield** rates, i.e. lower rates of flaw-free chips. Of course, one can try to make chips denser as well, but there are limits here too.

At this point, some researchers at IBM decided to take a fresh look at the whole question of instruction set formulation. The main point is optimal use of real estate. In a CISC CPU chip, a large portion of the chip is devoted to circuitry which is rarely, if ever used.[3] For example, most users of VAX machines did not make use of the **poly** instruction, and yet the circuitry for that instruction would occupy valuable-but-wasted real estate in the chips comprising a VAX CPU.

True, for those programs which did use the **poly** instruction, execution speed increased somewhat,[4] but for most programs the **poly** instruction was just getting in the way of fast performance.

Other factors came into play. For example, the richer the instruction set, the longer the decode portion of the instruction cycle would take. Even more important is the issue of pipelining, which is much harder to do in CISC machine, due to lack of uniformity in the instruction set (in a CISC machine, instructions tend to have different lengths).

For this reason, the IBM people asked, "Why do we need all this complexity," and they built the first RISC machine. Unfortunately, IBM scrapped their RISC project, but later David Patterson of the Division of Computer Science at UC Berkeley became interested in the idea, and developed two RISC CPU chips, RISC I and RISC II, which later became the basis of the commercial SPARC chip, as mentioned earlier.

Patterson's team did not invent the RISC concept, nor did they invent the specific features of their design such as delayed branch (see below). Nevertheless, these people singlehandedly changed the thinking of the entire computer industry, by showing that RISC could really work well. The team considered the problem as a whole, investigating all aspects, ranging from the questions of chip layout and VLSI electronics technology to issues of compiler writing, and through this integrated approach were able to make a good case for the RISC concept. Manolis Katevenis, a Ph.D. student who wrote his dissertation on RISC under Professor Patterson, won the annual ACM award for the best dissertation in Computer Science.[5]

At about the same time, Professor John Hennessy at Stanford University (who was later to become president of the university) was conducting his own RISC project, leading to his founding of the MIPS Corporation, which produced his RISC chip.

Silicon Graphics, Inc. eventually bought MIPS Inc. and SGI uses MIPS processors in its servers and so on, and though SGI is no longer one of the big players in those segments of the industry, MIPS processors are popular in the world of embedded applications.

Today virtually every computer manufacturer has placed a major emphasis on RISC. Even Intel, the ultimate CISC company, has turned to RISC for its most powerful chips. Sun Microsystems has the SPARC chip,

[3] Actually, most CISC chips use a technique called **microcode**, which would make the use of the word "circuitry" here a bit overly simplistic, but we will not pursue that here.

[4] You should think about this. First, **poly**, being a single instruction, would require only one instruction fetch, compared to many fetches which would be needed if one wrote a function consisting of many instructions which would collectively perform the work that **poly** does. Second, at the circuitry level, we might be able to perform more than one register transfer at once.

[5] Published as *Reduced Instruction Set Computer Architectures for VLSI*, M. Katevenis, MIT Press, 1984.

IBM and Motorola have the Power PC chip series used in Macs,[6] and so on.

Of course, this does not imply that RISC is inherently "superior," and as electronic technologies advance in the future, there may well be some resurgence of CISC ideas, at least to some degree.

9.2 A Definition of RISC

There is no universally-agreed-upon definition of RISC, but most people would agree at least to the following characteristics:

- Most instructions execute in a single clock cycle.
- "Load/store architecture"—the only instructions which access memory are of the LOAD and STORE type, i.e. instructions that merely copy from memory to a register or vice versa, such as Intel's

  ```
  movl (%eax), %ebx
  ```

  and

  ```
  movl %ecx, (%edx)
  ```

  Instructions like Intel's **addl %eax, (%ebx)** are not possible.
- Every instruction has the same length, e.g. 4 bytes. By contrast, Intel instructions range in length from 1 to several bytes.
- **Orthogonal architecture** — any operation can use any operand. This is in contrast, for example, to the Intel instruction STOS, which stores multiple copies of a string.[7] Here the only available operands are the registers EAX, ECX and EDI; one is not allowed to use any other register in place of these.
- The instruction set is limited to only those instructions which are truly justified in terms of performance/space/pipeline tradeoffs. The SPARC chip, for instance, does not even have a multiply instruction. If the compiler sees an expression like, say,

  ```
  i = j * k;
  ```

  in a source code file, the compiler must generate a set of add and shift instructions to synthesize the multiply operation, compared to the Intel case, in which the compiler would simply generate an **imull** instruction.

[6] Though Apple has announced plans to switch to Intel starting 2008.

[7] In this context it must be used with the REP prefix.

- Hardwired implementation, i.e. not microcoded.[8]

All but the last of the traits listed above make pipelining smoother and easier.

9.3 Beneficial Effects for Compiler Writers

Computer architects used to partly justify including instructions like **poly** in the instruction set by saying that this made the job of compiler writers easier. However, it was discovered that this was not the case, for several reasons.

- Compiler writers found it difficult to write compilers which would automatically recognize situations in (say) C source code in which instructions like **poly** could be used.
- Compiler writers found that some specialized instructions that were actually motivated by high-level language constructs did not match the latter well. For example, the VAX had a CASE instruction, obviously motivated by the Pascal **case** and C **switch** constructs; yet it turned out to be too restrictive to use well for compiling those constructs.
- CISC generally led to very nonorthogonal architectures, which made the job of compiler writers quite difficult. For instance, in Intel, the requirement that the EAX, ECX and EDI registers be used in STOS, plus the requirement that some of these registers be used in certain other instructions, implies that compiler writers "don't have any spare registers to work with," and thus they must have the compiler generate code to save and restore "popular" registers such as EAX; this is a headache for the compiler writer.

All of this inhibits development of compilers which produce very fast code.

Most RISC machines do ask that the compiler writers do additional work of other kinds, though, in that they have to try to fill **delay slots**; more on this below.

9.4 Introduction to the MIPS Architecture

MIPS is one of the most popular RISC chips, used for example in SGI workstations and in a number of embedded applications.[9]

[8] Again, we will not discuss microcode here.

[9] MIPS is also the subject of a simulator, SPIM, which is used in many university computer architecture courses. See `http://www.cs.wisc.edu/~larus/spim.html`.

Every MIPS instruction is 32 bits, i.e. one word, in length, and every instruction which does not access memory executes in one cycle. Those that do access memory are assumed to "usually" take two cycles, meaning that this will be the case if the desired memory access results in a cache hit; otherwise, the CPU enters a **stall** mode until the memory access is satisfied.

9.4.1 Register Set

There are 32 general registers, named $0, $1, ..., $31. The assembler also has an alternate set of names for the registers, which show their suggested uses; here is a partial list:

```
$0       zero      contains the hard-wired constant 0
$1-$2    v0-v1     for expression evaluation and function return values
$4-$7    a0-a3     for passing arguments
$8-$15   t0-t7     for temporary results
...
$28      gp        pointer to global area
$29      sp        stack pointer
$30      fp        frame pointer (not used)
$31      ra        return address
```

The usages listed above, e.g. $29 as the stack pointer, are merely suggested; in keeping with the RISC philosophy of orthogonality, the registers are treated uniformly by the architecture.

9.4.2 Example Code

To introduce the MIPS architecture, here is a C program which calls a MIPS assembly language subroutine, **addone**():

TryAddOne.c:

```
int x;

main()

{  x = 7;
   addone(&x);
   printf("%d\n",x);  // should print out 8
   exit(1);
}
```

AddOneMIPS.s:

```
.text
        .globl  addone
```

```
addone:
        lw      $2,0($4)
        addu    $2,$2,1
        sw      $2,0($4)
        j       $31
```

To begin, let's look at the compiled code we obtain by applying **gcc -S** to TryAddOne.c. The source lines

```
x = 7;
addone(&x);
```

translate to:

```
        # x = 7;
        li      $2,7
        sw      $2,x

        # addone(&x);
        la      $4,x
        la      $25,addone
        jal     $31,$25
```

The **li** (Load Immediate) instruction puts 7 into the register $2.[10]

The **sw** (Store Word) instruction then copies what is in $2 to the memory location x.

Now, what about the call? Many RISC architectures are less stack-oriented than are the classical CISC ones. This is because stacks are in memory, and memory access is slow.[11] MIPS is an example of this philosophy. Both the argument to **addone()**, as well as the return address, will be put in registers, rather than on the stack.

The **la** (Load Address) instruction here puts the argument, the address of **x**, into register $4.[12] We then use this same instruction to put the jump target, i.e. the address of **addone()**, into $25. We then call **addone()** with the instruction

```
        jal     $31,$25
```

which says to "jump and link" (i.e. do a subroutine call) to the subroutine pointed to by $25, saving the return address (the address of the instruction following the **jal**) in $31.

[10] Actually, **li** is not a real MIPS instruction, but is instead a macro, from which the assembler will generate an **ori** instruction. More on this later.

[11] Some architectures, such as SPARC, try to solve this problem by actually having space on the CPU chip for the top few elements of the stack.

[12] Recall that registers $4-$7 are used for passing arguments. If there are more arguments, we can store them in memory, and one of the arguments can be a pointer to them.

The return from the subroutine thus makes use of $31:

```
j       $31
```

This is an unconditional jump, to the location pointed to by $31.

Note that operands have different meanings depending on the operation. In **sw $2, x**, the **x** meant the memory location named **x**, whereas in **la $4, x**, **x** meant the address of **x**.

Recall that **main()** had placed the argument in $4. The subroutine then uses it:

```
lw      $2,0($4)
addu    $2,$2,1
sw      $2,0($4)
```

The **lw** (Load Word) instruction is the opposite of **sw**, fetching a word from memory rather than storing it. Both the **lw** and **sw** here use the more general format, which is based addressing. For example,

```
lw      $2,0($4)
```

adds $4 to 0, and then treats that as an address, fetching the contents of that address and putting it in register $2.

The **addu** (Add Unsigned) instruction has three operands; it adds the second and third, and places the sum in the first.

9.4.3 MIPS Assembler Pseudoinstructions

Recall that most assembly languages allow **macros**. MIPS assemblers go a step further, offering the programmer use of **pseudoinstructions**; these appear to be machine instructions, but are actually like macros, which the assembler translates to one or more real MIPS instructions.

Several "instructions" which we saw in the example above are actually pseudoinstructions. The first of those was **li**, allegedly the Load Immediate instruction. There actually is no such instruction, and our line containing **li** was actually treated by the assembler as

```
ori $2, $0, 7
```

where **ori** is the MIPS' OR Immediate instruction. Remember, the register $0 contained the hardwired value 0, so the instruction above applies the logical OR operation to 0 and 7, yielding 7, and thus placing 7 into

the register $2, just as desired. So, why couldn't the designers of the MIPS architecture include **li** Load Immediate instruction?

To see the answer, suppose we wish to load a large number like 0xabcd1234, which is 32 bits in length. Remember, all MIPS instructions are 32 bits long, so this operand would occupy the entire instruction, with no room for an op code!

Indeed, if our assembly language source code contains a line

```
li $2,0xabcd1234
```

the assembler will actually treat it as if it were two instructions:

```
lui $1,0xabcd
ori $2, $1, 0x1234
```

The MIPS instruction **lui** places the given immediate constant, 0xabcd in this case, and placing it in upper 16 bits of the destination, the register $1 here, and places 0s in the lower 16 bits. The MIPS instruction **ori** then puts 0x1234 in the lower 16 bits of a copy of $1, then placing the result in $2. This gets 0xabcd1234 into $2, as desired.

So, the use of pseudoinstructions here frees the programmer from having to think about the size of the operand she is using.

Similarly, in the line

```
la $4, x
```

from our example code above, **la** is just a pseudoinstruction, not a real MIPS machine instruction. One of the issues here is like the one with **li** above; the address of x is a 32-bit quantity, and thus could not be fit into one instruction. The other issue is that we need a way of letting the assembler know that "x" here means the address of x, and our use of the pseudoinstruction **la** tells the assembler that.

Finally, there is even some chicanery in the line

```
sw $2, x
```

Actually, **sw** IS a real MIPS instruction. But the real instruction has three operands, as in our other line,

```
sw $2, 0($4)
```

(the three operands are $2, 0 and $4), whereas

```
sw $2, x
```

has only two operands. The fact that there are only two operands here is a signal to the assembler that this too is a pseudoinstruction, in spite of the fact that there is a real MIPS instruction **sw**. So, the assembler will convert this to an **lui** and **ori** and then a (real) **sw**.

9.4.4 Programs Tend to Be Longer on RISC Machines

Note in our example above that it took two instructions

```
li      $2,7
sw      $2,x
```

to do what, for example, on Intel would take only one:

```
movl $7,x
```

In fact, we later found that those two instructions were actually pseudoinstructions, so here MIPS is taking four instructions to do what Intel does in one.

This is typical of RISC machines, and means that RISC programs tend to be longer than CISC ones. For example, the compiled version of TryAddOne.c above[13] is 1276 bytes long on an SGI machine but only 1048 on an Intel platform.

9.4.5 Instruction Formats

One very "RISC-y" feature of MIPS is that it has only three instruction formats.[14] To present them, let us number the most- and least-significant bits in a word as 31 and 0, respectively. Also, we will use the following abbreviations:

```
op         op code
--         -------
rs         register operand
rt         register operand
rd         register operand
imm        immediate constant
shamt      shift amount
funct      special function (like an additional op code)
dist       distance factor to branch (i.e. jump) target
```

[13] This is TryAddOne.o, generated from applying **gcc -c** to TryAddOne.c.

[14] By contrast, the Intel chip, a CISC, has a large number of instruction formats, and even the RISC SPARC chip has six.

the formats are as follows.

- I ("immediate") format:
  - op: bits 31-26
  - rs (source): bits 25-21
  - rt (destination): bits 20-16
  - imm: bits 15-0
- J ("jump") format:
  - op: bits 31-26
  - dist: bits 25-0
- R ("register") format:
  - op: bits 31-26
  - rs (source): bits 25-21
  - rt (source): bits 20-16
  - rd (destination): bits 15-11
  - shamt: bits 10-6
  - funct: bits 5-0

The **shamt** field is used if the instruction is to perform a right- or left-shift; the shift will be **shamt** bits in the specified direction.

The **dist** field in J instructions is somewhat unusual, but actually clever: It is equal to 1/4 of the distance to the branch target.[15] Since all instructions are 4 bytes long, then every instruction address is a multiple of 4, and thus the distance from any branch instruction to its target is a multiple of 4, i.e. the distance has two 0 bits at the right end. That last point means that there is no point in storing the last two bits of the distance; they are simply tacked on by the CPU when the instruction is executed. This effectively multiplies the jump range by 4.

The J format is used only for unconditional branches, including **call**. Conditional branches use the I or R format.

Note that the **imm** field can have two interpretations, depending on **op**. The **addi** instruction, for example, treats the immediate constant as a signed number, while **addiu** treats it as unsigned. Since MIPS has word

[15] Recall that *branch* is a synonym for *jump*.

size 32 bits, its ALU deals with 32-bit quantities, not the 16-bit quantity in an **imm** field. Thus the latter must be extended to 32 bits before entering the ALU, by copying the leftmost bit to the leading 16 bits. Thus for instance in the instructions

```
addi $6, $7, 0xb123
addiu $9, $10, 0xb123
```

in the first case the 0xb123 becomes 0xffffb123 while in the second case it becomes 0x0000b123.

9.4.6 Arithmetic and Logic Instruction Set

Here is a partial listing of the MIPS instruction set for arithmetic and logic operations:

```
add rd, rs, rt              # rd <-- rs + rt
addu rd, rs, rt             # rd <-- rs + rt (without overflow)
addi rd, rs, imm            # rd <-- rs + imm

and rd, rs, rt              # rd <-- rs AND rt
ani rd, rs, imm             # rd <-- rs AND imm

not rd, rs                  # rs <-- NOT rd

or rd, rs, rt               # rd <-- rs OR rt
ori rd, rs, imm             # rd <-- rs OR imm

sll rd, rt, shamt           # rd <-- rt << shamt
srl rd, rt, shamt           # rd <-- rt >> shamt

sub rd, rs, rt              # rd <-- rs - rt
subu rd, rs, rt             # rd <-- rs - rt (without overflow)
```

9.4.7 Conditional Branches in MIPS

On a MIPS machine, applying **gcc -S** to

```
if (i == 20) j = 2;
```

produces

```
        lw      $2,i
        li      $3,20
        bne     $2,$3,.L4
        li      $2,2
        sw      $2,j
```

where .L4 is a label of the branch target. The **bne** instruction compares the two registers and then branches to the target if they are not equal. (There is nothing like Intel's Z and S Flags in MIPS.)

9.5 Some MIPS Op Codes

```
ORI    001101
LUI    001111
SW     101011
JAL    000011
J      000010
BNE    000101
ADDI   001000
ADDU   001001
```

9.6 Dealing with Branch Delays

RISC architectures are especially good at having smooth pipeline operation, due to the uniformity of instruction execution time: Every instruction executes in one clock cycle, except for branches and load/store instructions, which access memory and thus take longer. Even to go to the cache takes up an extra clock cycle, and going to full memory takes even longer. Thus the pipeline is delayed.

For example, consider this MIPS code:

```
subiu $2, $2, $1
bne     $4, $3, yyy
addiu $5, $5 12
```

While the fetch/decode/execute cycle for the **bne** is taking place, the CPU is prefetching the **addiu**. But if the branch is taken, the instruction at yyy will be the one we need, and it will not have been prefetched.

9.6.1 Branch Prediction

One possible way to deal with the branch-delay problem is to design the hardware to try to predict whether a conditional branch will be taken or not. The hardware will prefetch from the instruction which it predicts will be executed next. If it is correct, there is no delay; if the prediction is false, we must fetch the other instruction, incurring the delay.

Even non-RISC architectures do this. Intel CPUs, for example do the following when a given conditional branch instruction is executed for the first time: If it is a branch backward, the prediction is that the branch will be taken; this design is motivated by the assumption that backward branches will typically be the last instruction within a loop, and usually such branches will be taken. If the branch is forward, the prediction is that it will not be taken. In subsequent times this particular branch instruction is executed, a complicated prediction algorithm is applied, based on previous history.

9.6.2 Delayed Branch

MIPS—as well as most RISC chips—solves this problem by using something called **delayed branch**. The CPU actually executes the instruction which sequentially follows the branch instruction in memory, in this case the **addiu**, before executing the instruction at the branch target. That way useful work is being done during the delay involving fetching of the branch target.

But wait! This can't work as it stands. After all, if the branch is taken, we do not want to execute that **addiu**! So, the assembler (or compiler, in the case of C/C++) must put in a "do nothing" instruction: **nop**, pronounced "no-op" for "no operation."[16]

```
subiu $2, $2, $1
bne     $4, $3, yyy
nop
addiu $5, $5 12
```

But now we are back to the original problem — no useful work is being done while the instruction at yyy is being fetched. So, next an optimizer within the assembler (or compiler) will try to actually rearrange the instructions. It will look for some instruction elsewhere in the code which can be moved into the slot currently occupied by the **nop**. In this case, the **subiu** can be safely moved:

```
bne $4, $3, yyy
subiu $2, $2, $1
addiu $5, $5, 12
```

The reasoning is this: The **bne** is not affected by the outcome of the **subiu** which precedes it (this would not be true if the **bne** compared $4 with $2, for example), so it doesn't matter if we move the **subiu**. Remember, the latter will still be executed, because the CPU is designed to execute the instruction which immediately follows the branch in memory.

The setup for load/store instructions is similar. The MIPS CPU will execute the instruction which immediately follows a load/store instruction in memory while performing the load/store. This is done because the latter requires a time-consuming memory access.

But as with branches, we need to watch out for dependencies. For example, consider

```
lw $2, 0($4)
addiu $2, $2, 1
```

The CPU would execute the add simultaneously with fetching register $2, with potentially disastrous consequences. Thus the assembler will at first change this to

[16] Almost every CPU architecture has this kind of instruction. On MIPS, its op code is all 0s.

```
lw $2, 0($4)
nop
addiu $2, $2, 1
```

and then the optimizer will try to find some instruction elsewhere to move to the position now occupied by the **nop**.

The instruction position following a branch or load/store is called a **delay slot**. Machines such as MIPS and SPARC have one delay slot, but some other machines have more than one. Clearly, the more delay slots there are, the harder it is to "fill" them, i.e. to find instructions to move to put in the slots, and thus avoid simply putting wasteful NOPs in them.

How hard is it to fill even one delay slot? The Katevenis dissertation on RISC says:

> Measurements have shown [citation given] that the [compiler] optimizer is able to remove about 90% of the no-ops following unconditional transfers and 40% to 60% of those following conditional branches. The unconditional and conditional transfer-instructions each represent approximately 10% of all executed instructions (20% total). Thus, while a conventional pipeline would lose $\approx$ 20% of the cycles, optimized RISC code only loses about 6% of them.

Chapter 10

The Java Virtual Machine

Note: Thomas Fifield, a student in ECS 50 a few years ago, wrote portions of Section 10.7 (excluding the source code for **NumNode.java**), and added the corresponding new instructions to Section 10.8.

10.1 Background Needed

In what follows, it is assumed that the reader has at least a rudimentary knowledge of Java. See the author's Java tutorial at `http://heather.cs.ucdavis.edu/~matloff/java.html` for a 30-minute introduction to the Java language. But really, for our purposes, anyone who understands C++ should have no trouble following the Java code here.

10.2 Goal

Keep in mind that our goal here is to study the Java (virtual) *machine*, not the Java language. We wish to see how the machine works, and—this is very important—see why its developers chose to design certain features of the machine the way they did.

10.3 Why Is It a "Virtual" Machine?

You may have heard about the **Java virtual machine** (JVM), associated with the Java language. What is really going on here?

The name of any Java source file has a **.java** suffix, just like C source file names end in **.c**. Suppose for

example our Java program source code is all in one file, **x.java**, and that for instance we are doing our work on a PC running Linux.

We first compile, using the Java compiler, **javac**:

```
1 % javac -g x.java
```

(The **-g** option saves the symbol table for use by a debugger.)

This produces the executable Java file, **x.class**, which contains machine language, called **byte code**, to run on a "Java machine." But we don't have such a machine.

Instead, we have a program that emulates the operation of such a machine. This, of course, is the reason for the 'V' in "JVM." The emulator (**interpreter**) program is named **java** . Note that in our case here, **java** will be a program running the Intel machine language of our PC.[1]

We then run our program:

```
1 % java x
```

Note that Java not only runs on a virtual machine, it also is in some sense running under a virtual operating system. For example, the analog of C's **printf()** function in Java is **System.out.println()**. Recall that (if our real machine is running UNIX) a **printf()** call in a C program calls the **write()** function in the OS. But this does not happen in the Java case; our OS is running on our real machine, but our Java program is running on the JVM. What actually happens is that **System.out.println()** makes a call to what amounts to an OS in the JVM, and the latter calls **write()** on the real machine.

By the way, JVM chips—i.e. chips that run Java byte code—do exist, but they are not in common use. Also, note that newer versions of GCC include GCJ, which will compile Java to machine code for your machine, e.g. Intel machine language if you are on a PC. This is really nice, as it produces faster-running programs, and it is far more common than Java chips, but still, the vast majority of Java code is executed on emulators.

10.4 The JVM Architecture

The JVM is basically a stack-based machine, with a 32-bit word size, using 2s complement arithmetic.

The term *stack-based machine* means that almost all operands are on the stack, and they are implicit to the instruction. For example, contrast the Intel instruction

```
1 addl %eax, %ebx
```

[1] And, for that matter, **javac** would be an Intel machine-language program too.

to the JVM instruction

```
1 iadd
```

In the Intel case, the programmer has a choice of which two things to add together, and explicitly states them. In the JVM case, the programmer has no choice; the two operands must be the top two elements of the stack. And since those operands must be the top two elements of the stack, there is no need to specify them in the instruction, so the instruction consists only of an op code, no explicit operands.[2]

Before continuing, it's vital that we discuss what the term "the" stack means above. Unlike the case of C/C++/assembly language, where there is a stack for each *process*, here we will have a stack for each *method call*.[3] So, the term "the stack" in the material here must always be thought of as meaning "the stack for whatever method is currently running." This is referred to as the **operand stack**.

On the other hand, there *is* something analogous to the stack you have worked with before in C/C++/assembly language. Recall that in that situation, all the function calls use a common stack, and all the data for a given function call—its arguments, return value and local variables—comprise the **stack frame** for that function call. Java has this too. There is a stack frame for each method call, and they are placed in the overall stack in the same last-in, first-out fashion that use saw for function calls in C/C++/assembly language. Each stack frame in Java will in turn consist of the operand stack for that method call, a section for arguments and local variables, and some other data.

10.4.1 Registers

The JVM register set[4] is fairly small:

- **pc**: program counter
- **optop**: pointer to the top of the operand stack for the currently-active method
- **frame**: pointer to the stack frame of the currently-active method
- **vars**: pointer to the beginning of the local variables of the currently-active method

[2]Historical note: A number of famous machines popular in the past, such as the Burroughs mainframe and HP 3000 minicomputer series, were stack-based.

[3]Recall that in Java, we use the term *method* instead of *function*.

[4]Remember, in the case of a real Java chip, these would be real registers, but in the JVM setting, these registers, as well as the instruction set, are simulated by the **java** program.

10.4.2 Memory Areas

- the (general) Stack:

  A method call produces a new stack frame, which is pushed onto the Stack for that program,[5] and a return pops the frame from the program's Stack.

  A stack frame is subdivided into the following.

  - a Local Variables section:

    All the local variables and arguments for the method are stored here, one per **slot** (i.e. one variable per word), with the arguments stored first and then the locals. The arguments and locals are stored in order of their declaration. In the case in which the method is an instance method,[6] slot 0 will contain a pointer to the object on which this method is operating, i.e. the object referred to as **this** in the program's source code.
  - an Operand Stack section:

    This is the area on which the method's instructions operate. As noted, almost all JVM instructions are stack-based; e.g. an "add" instruction pops the top two elements of the stack, adds them, and pushes the sum back onto the stack. So the operand stack portion of a method's stack frame is what we are referring to when we refer to an instruction as operating on "the" stack.
  - a Frame Data section:

    We will not go into the details of this, but for example the Frame Data section of a called method's stack frame will include a pointer to the caller's stack frame. This enables return to the caller when the called method finishes, and enables the latter to put the return value, if any, into the caller's stack frame.

- the Method Area:

  The classes used by the executing program are stored here. This includes:

  - the bytecode and access types of the methods of the class (similar to the **.text** segment of a UNIX program, except for the access types)
  - the **static** variables of the class (their values and access types, i.e **public**, **private** etc.)

  The **pc** register points to the location within the Method Area of the JVM instruction to be executed next.

  The Method Area also includes the Constant Pool. It contains the string and numeric literals used by the program, e.g. the 1.2 in

[5] More precisely, for that thread, since many Java programs are threaded.

[6] In general object-oriented programming terminology, a function is called an **instance** function if it applies specifically to an instance of a class. In C++ and Java, this is signified by not being declared **static**. The alternate form is **class** methods, which apply to all objects of that class, and is signified in C++/Java by being declared **static**.

```
1   float W = 1.2;
```

and also contains information on where each method is stored in the Method Area, as well as the static variables for the various classes.

- the Heap:

  This is where Java objects exist. Whenever the Java **new** operation is invoked to create an object, the necessary memory for the object is allocated within the heap.[7] This space will hold the instance variables for the object, and a pointer to the location of the object's class in the Method Area.

10.5 First Example

Consider the following source code, **Minimum.java**:

```
1   public class Minimum {
2
3      public static void main(String[] CLArgs)
4
5      {  int X,Y,Z;
6
7         X = Integer.parseInt(CLArgs[0]);
8         Y = Integer.parseInt(CLArgs[1]);
9         Z = Min(X,Y);
10        System.out.println(Z);
11     }
12
13     public static int Min(int U, int V)
14
15     {  int T;
16
17        if (U < V) T = U;
18        else T = V;
19        return T;
20     }
21  }
```

10.5.1 Java Considerations

In Java, there are no global quantities of any kind. That not only means no global variables, but also no free-standing, functions—every function must be part of some class. So for example we see here that even **main()** is part of a class, the class **Minimum**. (That class in turn must be named after the name of the source file, **Minimum.java**.)

[7] Just as in C, where a call to **malloc()** results in memory space being allocated from the heap, and just as in C++, where a call to **new** results in space being taken from the heap.

The argument for **main()**, **CLArgs**, is the set of command-line arguments, i.e. what we normally name **argv** in C/C++.[8] There is nothing like **argc**, though. The reason is that **CLArgs** is of type **String []**, i.e. an array of strings, and in Java even arrays are objects. Array objects include a member variable showing the length of the array, so information analogous to **argc** is incorporated in **CLArgs**, specifically in **CLArgs.length**—the value of the **length** member variable of the array class, for the instance **CLArgs** of this class.

The function **Integer.parseInt()** is similar to **atoi()** in C/C++. Note, though, since Java does not allow free-standing functions, **parseInt()** must be part of a class, and it is—the **Integer** class, which has lots of other functions as well.

We use the compiler, **javac**, to produce the class file, **Minimum.class**. The latter is what is executed, when we run the Java interpreter, **java**.

10.5.2 How to Inspect the JVM Code

We can use another program, **javap**, to disassemble the contents of **Minimum.class**:[9]

```
% javap -c Minimum
Compiled from Minimum.java
public class Minimum extends java.lang.Object {
    public Minimum();
    public static void main(java.lang.String[]);
    public static int Min(int, int);
}

Method Minimum()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 aload_0
   1 iconst_0
   2 aaload
   3 invokestatic #2 <Method int parseInt(java.lang.String)>
   6 istore_1
   7 aload_0
   8 iconst_1
   9 aaload
  10 invokestatic #2 <Method int parseInt(java.lang.String)>
  13 istore_2
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Min(int, int)>
  19 istore_3
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
```

[8] Note that in C/C++ we can name that parameter whatever we want, though it is customary to call it **argv**.

[9] So, this listing here is similar to the output of **gcc -S** in a C/C++ context.

```
  24 invokevirtual #5 <Method void println(int)>
  27 return

Method int Min(int, int)
   0 iload_0
   1 iload_1
   2 if_icmpge 10
   5 iload_0
   6 istore_2
   7 goto 12
  10 iload_1
  11 istore_2
  12 iload_2
  13 ireturn
```

Note that each "line number" is actually an offset, i.e. the distance in bytes of the given instruction from the beginning of the given method.

10.5.3 The Local Variables Section of main()

Here is what the Local Variables Section of **main()**'s stack frame looks like:

slot	variable
0	pointer to CLArgs
1	X
2	Y
3	Z

10.5.4 The Call of Min() from main()

Now consider the call to **Min()**. The code

```
1         Z = Min(X,Y);
```

gets compiled to

```
  14 iload_1
  15 iload_2
  16 invokestatic #3 <Method int Min(int, int)>
  19 istore_3
```

As in a classical architecture, the arguments for a call are pushed onto **main()**'s operand stack, as follows. The **iload_1** ("integer load") instruction in offset 14 of **main()** pushes slot 1 to the operand stack. Since slot

1 in **main()** contains X, this means that X will be pushed onto **main()**'s operand stack. The instruction in offset 15 will then push Y.[10]

The **invokestatic** instruction is the function call. By looking up information which is stored regarding **Min()**, this instruction knows that **Min()** has two arguments.[11] Thus the instruction knows that it must pop two values from **main()**'s operand stack, which are the arguments (placed there earlier by **iload_1** and **iload_2**), and it places them in the first two slots of **Min()**'s Local Variables Area.

Note that in the instruction **iload_1**, for example, the '1' is a part of the instruction name, not an operand. In fact, none of **iload_0**, **iload_1**, **iload_2** and **iload_3** has an operand; there is a separate instruction, i.e. separate op code, for each slot.

But there is no **iload_5**, **iload_6** and so on. What if you need to load from some other slot, say slot 12? Then you would use the **iload** instruction, which is fully general, i.e. can load from any slot, including but not limited to slots 0, 1, 2 and 3. Note that **iload_0** etc. are one-byte instructions, while **iload** is a two-byte instruction, with the second byte being the slot number, so we should use those special instructions for the cases of slots 0, 1, 2 and 3. The designers of the JVM knew that most accesses to local variables would be to slots 0-3, so they decided to include special one-byte instructions to load from those slots, thus making for smaller code. Similar statements hold for **istore**, etc.

10.5.5 Accessing Arguments from within Min()

Now, upon entry to **Min()**, here is what that function's Local Variables Section will look like:

slot	variable
0	U
1	V
2	T

Since the arguments **U** and **V** are in slots 0 and 1, they will be accessed with instructions like **iload_0** and **iload_1**.

The main new instruction in **Min()** is **if_icmpge** ("if integer compare greater-than-or-equal") in offset 2. Let's refer to the top element of the current (i.e. **Min()**'s) operand stack as *op2* and the next-to-top element as *op1*. The instruction pops these two elements off the operand stack, compares them, and then jumps to the branch target if $op1 \geq op2$. Again, keep in mind that these items on **Min()**'s operand stack were placed there by the **iload_0** and **iload_1** instructions.

Min()'s **ireturn** instruction then pops the current (i.e. **Min()**'s) operand stack and pushes the popped value on top of the caller's (i.e. **main()**'s operand stack. In the case of methods which do not have return values,

[10] So, the arguments must be pushed in the order in which they appear in the call, unlike C/C++.

[11] Again unlike C/C++, where the number of arguments in the call is unknown to the compiler.

we use the **return** instruction instead of **ireturn**.

10.5.6 Details on the Action of Jumps

For jump instructions, the branch target is specified as the distance from the current instruction to the target. As can be seen in the JVM assembler code above, the target for the **if_icmpge** instruction in offset 2 of **Min()** is offset 10 (an **iload_1** instruction). Since our **if_icmpge** instruction is in offset 2, the distance will be 8, i.e. 0x0008.[12] Those latter two bytes comprise the second and third bytes of the instruction. Note that the branch target must be within the current method; JVM will announce a runtime error if not.

From the Sun JVM specifications (see below), we know that the op code for the **if_icmpge** instruction is 0xa2. Thus the entire instruction should be a2 00 08, and this string of three bytes should appear in the file **Minimum.class**. Running the command

```
1  % od -t x1 Minimum.class
```

on a UNIX machine, we see that a2 00 08 is indeed in that file, in bytes 1255-1257 octal (685-687 decimal).

10.5.7 Multibyte Numbers Embedded in Instructions

Note in the jump example above, the instruction was a0 00 08, with a0 being the op code and 00 08 being the distance to the jump target. Note that the order of bytes in memory will be that seen above, i.e. a0 00 08, so 00 will be in a lower-address byte than 08. So, the question arises as to whether the embedded constant is intended to be 0008, i.e. 8, or 8000. The answer is the former.

In other words, numbers embedded within instructions are meant to be interpreted as big-endian.

10.5.8 Calling Instance Methods

When an instance method is called, we use the **invokevirtual** instruction instead of **invokestatic**. To understand how that works. recall that in general OOP programming, an instance method includes an extra "unofficial" argument in the form of a pointer to the instance on which the method is invoked, i.e. the famous **this** pointer in C++ (and Java).

Concretely, suppose a class **C** includes an instance member method **M()**, say with two arguments. Then the call

[12] Unlike the Intel (and typical) case, the jump distance in JVM is calculated from the jump instruction, not from the one following it.

```
M(A,B)
```

from within **C** would in essence have three arguments—the two explicit ones, **A** and **B**, and also implicitly, a pointer to the current instance of **C**. As with any stack-based calling system, e.g. C++ running on an Intel machine, that extra parameter must be pushed onto the stack too, just like the explicit ones. This is also done in the case of the JVM.

Thus we can't use **invokestatic** in the JVM case, because we must account for that "extra" argument. The **invokestatic** instruction, as explained earlier, pops the arguments off the caller's stack and places them into the callee's Local Variables Section. That's not good enough in the case of an instance method, because that "extra" argument must be popped and placed into the callee's Local Variables Section too. So, the designers of the JVM also included an **invokevirtual** instruction to do this; it transfers *all* the arguments from the caller's stack to the callee's Local Variables Section, both the explicit arguments and the "extra" one.

So, from the callee's point of view, the item in its slot 0 will be **this**, the pointer to the current object, and in fact the compiler will translate references to **this** to accesses to slot 0.

The only example we have so far of a call made via **invokevirtual** is in the compiled code for our source line in **main**(),

```
System.out.println(Z);
```

The compiled code is

```
  20 getstatic #4 <Field java.io.PrintStream out>
  23 iload_3
  24 invokevirtual #5 <Method void println(int)>
```

That 3 in offset 23 refers to slot 3, the location of our local variable **Z**, our argument to **println**().

The code

```
System.out.println(Z);
```

calls the **println()** method on the object **System.out**. The official Java documentation indicates that the latter is an instance of the class **java.io.PrintStream**, which in turn is a subclass of the class **java.io.FilterOutputStream**. That latter class includes a member variable **out**, which is basically a pointer to the file we are writing to. In this case, we are writing to the screen (in UNIX parlance, **stdout**).

With that in mind, look at the instruction in offset 20. From our discussion above, you can see that this instruction must be placing a pointer to the object **System.out** on the stack, and that is exactly what the **getstatic** instruction is doing. It is getting a class (i.e. **static**) variable from the given object, and pushing it onto the stack.

10.5.9 Creating and Accessing Arrays

Arguments and local variables which are arrays are maintained as addresses within the Local Variables Area. Array read accesses work by pushing the address of the array and the desired index onto the operand stack, and then executing one of the special array instructions. Array writ accesses are done the same way, but with an additional push for the value to be stored.

There are also special array instructions which create the array storage itself when **new** is invoked.

Consider the following example:

```
1  public int gy(int x)
2  {  int y[],z;
3     y = new int[5];
4     z = x + 2;
5     y[3] = x;
6     return 0;
7  }
```

(It does nothing of interest, but will serve as a convenient simple example.)

That code is translated to:

```
Method int gy(int)
   0 iconst_5
   1 newarray int
   3 astore_2
   4 iload_1
   5 iconst_2
   6 iadd
   7 istore_3
   8 aload_2
   9 iconst_3
  10 iload_1
  11 iastore
  12 iconst_0
  13 ireturn
```

Here you see the **newarray** instruction is used to create space for the array, specifically enough for 5 **int**s; the 5 was pushed onto the operand stack by the **iconst_5** ("integer constant") instruction.

The assignment to **y[3]** is done by the code in offsets 7-11. Make sure you see how this works. In particular, the **iastore** ("integer array store") instruction first pops the stack to determine which value to store (slot 1, i.e. **x**), then pops again to determine which array element to store to (index 3), and then pops once more to determine which array to use (slot 2, i.e. **y**.

In the code for **main()** in our **Min** example above, **CLArgs** is an array of strings, thus an array of arrays, just as in C/C++. The **aload_0** ("address load") instruction pushes the address in slot 0 i.e. the address

of **CLArgs**, onto the current operand stack. Similarly, **iconst_0** ("integer constant") pushes the constant 0. All this is preparation for accessing the array element **CLArgs[0]**. However, since that element is a string, thus an address again, we do not use **iastore** as before, instead using the **aastore** instruction ("address array store").

10.5.10 Constructors

When the constructor for a new object is called, we need to execute a **invokespecial** instruction.

10.5.11 Philosophy Behind the Design of the JVM

Remember, we are here to learn about the Java machine, not the Java language. One aspect of that concerns why the designers of the JVM made certain choices as to the structure of the machine.

Before we continue, note that the JVM *is* a machine, just like Intel, MIPS or the others. While it's true that we most often just use an emulator program, JVM chips have indeed been built. So for example one could certainly write a C compiler for the machine, i.e. write a compiler that translates C source code to JVM machine code.

10.5.11.1 Instruction Structure

Most instructions are a single byte in length, but there are a few multi-byte instructions as well. In any case, the first byte is the op code, and any operands are in the bytes that follow. As mentioned earlier, almost all JVM instructions involve stack operations, so there are no explicit operands, which is why most instructions are a single byte long. This has other implications, explained below.

10.5.11.2 Stack Architecture

You may wonder why the developers of the JVM chose the stack-based architecture. Actually, the literature is not clear about this.

One claim is that the choice of this architecture was made in order to make it easy to write JVM interpreters for native machines that had few or no registers. If for example the developers had chosen to have the JVM have 32 registers, then on machines with fewer registers the authors of Java interpreters would not be able to model the JVM virtual registers by the real registers of the native machines. Instead, the JVM registers would have to be modeled by variables in memory. This does not sound like a very convincing argument. For one thing, it makes it more difficult for authors of interpreters on machines that do have a lot of registers to write fast interpreters.

Another claim is that by having most instructions only a single byte in length, the overall program size is smaller. This would be an important consideration if it were true. Java code is often transported over the network. This happens behind the scenes when you access a Java-powered Web page. The Web site will download Java code for some operation to your machine (the one where you are running the browser), and that code will be executed by the Java interpreter on your machine.

So smaller code would mean less network download delay. But is Java code smaller? Recall that RISC instructions are also very simple, but that that actually tends to make code size larger than on CISC machines, since each instruction does less work. Since the stack architecture means a lot of pushes and pops in order to manoevre data items for arithmetic, it may well be the case that code size is larger for the JVM.

However,[13] the JVM situation is very different from the RISC one, because with the former we are worried about conserving network bandwidth. JVM byte code is likely to be highly compressable—certain instructions like, say, **iload_0** are probably quite frequent—so that download times could be made quite low.

10.5.11.3 Safety

Note that there are separate instructions **istore_0** and **astore_0**. Each pops the operand stack and places the popped value into slot 0. But the first assumes the popped value is an integer, while the second assumes an address.[14] In other words, unlike "normal" machines, in which the hardware is unaware of types, the JVM is quite aware and proactive. The JVM will check types (which are stored with the value), and a runtime error will occur if, for example, one tries to execute **astore_0** when the top of the operand stack contains an integer type.

Java does allow **type coercion** as in C/C++. Consider for instance the code

```
int I = 12; float X;
...
X = I + (float) 3.8;
```

Here the variable **I** is converted to type **float** in the compiled code, using the JVM's **i2f** instruction.

Getting back to the safety issue, one concern is that Java code might be "hacked" during its transport over the network. There are various mechanisms within the JVM to check that the code is the same as when it was sent; none of these is perfect, of course, but it does provide some protection.

10.6 Another Example

This one finds the row in a matrix which has minimum sum.

[13] As pointed out by student Daniel Wolfe.

[14] Formally called a **reference**.

```
1  public class Min2 {
2
3     public static int ABC[][];  // elements will be within [0,1000000]
4
5     public static void main(String[] CLArgs)
6
7     {  int I,J;
8
9        ABC = new int[10][10];
10       // fill it with something, just an example
11       for (I = 0; I < 10; I++)
12          for (J = 0; J < 10; J++)
13             ABC[I][J] = 2 * I + J;
14       System.out.println(Min());
15    }
16
17    public static int Min()
18
19    {  int Row,Col,MinSoFar,Sum;
20
21       MinSoFar = 1000000;  // note restrictions on ABC above
22       for (Row = 0; Row < 10; Row++)  {
23          Sum = 0;
24          for (Col = 0; Col < 10; Col++)  {
25             Sum += ABC[Row][Col];
26             if (Sum > MinSoFar) break;
27          }
28          if (Sum < MinSoFar)
29             MinSoFar = Sum;
30       }
31       return MinSoFar;
32    }
33 }
```

```
Compiled from Min2.java
public class Min2 extends java.lang.Object {
    public static int ABC[][];
    public Min2();
    public static void main(java.lang.String[]);
    public static int Min();
}

Method Min2()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 bipush 10
   2 bipush 10
   4 multianewarray #2 dim #2 <Class [[I>
   8 putstatic #3 <Field int ABC[][]>
  11 iconst_0
  12 istore_1
  13 goto 45
  16 iconst_0
  17 istore_2
```

```
  18 goto 36
  21 getstatic #3 <Field int ABC[][]>
  24 iload_1
  25 aaload
  26 iload_2
  27 iconst_2
  28 iload_1
  29 imul
  30 iload_2
  31 iadd
  32 iastore
  33 iinc 2 1
  36 iload_2
  37 bipush 10
  39 if_icmplt 21
  42 iinc 1 1
  45 iload_1
  46 bipush 10
  48 if_icmplt 16
  51 getstatic #4 <Field java.io.PrintStream out>
  54 invokestatic #5 <Method int Min()>
  57 invokevirtual #6 <Method void println(int)>
  60 return

Method int Min()
   0 ldc #7 <Integer 1000000>
   2 istore_2
   3 iconst_0
   4 istore_0
   5 goto 52
   8 iconst_0
   9 istore_3
  10 iconst_0
  11 istore_1
  12 goto 36
  15 iload_3
  16 getstatic #3 <Field int ABC[][]>
  19 iload_0
  20 aaload
  21 iload_1
  22 iaload
  23 iadd
  24 istore_3
  25 iload_3
  26 iload_2
  27 if_icmple 33
  30 goto 42
  33 iinc 1 1
  36 iload_1
  37 bipush 10
  39 if_icmplt 15
  42 iload_3
  43 iload_2
  44 if_icmpge 49
  47 iload_3
  48 istore_2
  49 iinc 0 1
  52 iload_0
```

```
  53 bipush 10
  55 if_icmplt 8
  58 iload_2
  59 ireturn
```

So, for example, look at 13 of the source code,

```
1  ABC[I][J] = 2 * I + J;
```

This single Java statement compiles to a remarkable 10 JVM machine instructions! They are in offsets 21-32 of **main**(). Below is an overview of what they do:

```
21   push address of ABC
24   push I
25   pop I, address of ABC; push address of ABC[I]
26   push J
27   push 2
28   push I
29   pop I, 2; push 2*I
30   push J
31   pop J, 2*I; push 2*I+J
32   pop 2*I+J, J, address of ABC[I]; do ABC[I][J]=2*I+J
```

As you read this, recall that a two-dimensional array is considered an array of arrays. For example, row **I** of **ABC**, i.e. **ABC[I]**, is an array. Recall also that an array name, when used without a subscript, is the address of the beginning of that array. Here **ABC[I]**, considered as an array, has no subscript, while for instance **ABC[I][J]** has the subscript **J**. So, **ABC[I]** is an address. Thus **ABC** is an array of addresses! Hence the use of **aaload** in offset 25.

The **multianewarray** instruction sets up space for the array, pushing the address on the stack. The **putstatic** instruction then pops that address off the stack, and places it in the entry for this **static** variable in the Method Area.

10.7 Yet Another Example

This example builds a linked list of integers:

```
1  // this class is in the file NumNode.java
2
3  public class NumNode
4
5  {  private static NumNode Nodes = null;
6     // valued stored in this node
7     int Value;
```

```
8     // "pointer" to next item in list
9     NumNode Next;
10
11    public NumNode(int V)  {
12       Value = V;
13       Next = null;
14    }
15
16    public static NumNode Head()  {
17          return Nodes;
18    }
19
20    public void Insert()  {
21       if (Nodes == null)  {
22          Nodes = this;
23          return;
24       }
25       if (Value < Nodes.Value)  {
26          Next = Nodes;
27          Nodes = this;
28          return;
29       }
30       else if (Nodes.Next == null)  {
31          Nodes.Next = this;
32          return;
33       }
34       for (NumNode N = Nodes; N.Next != null; N = N.Next)  {
35          if (Value < N.Next.Value)  {
36             Next = N.Next;
37             N.Next = this;
38             return;
39          }
40          else if (N.Next.Next == null)  {
41             N.Next.Next = this;
42             return;
43          }
44       }
45    }
46
47    public static void PrintList()  {
48       if (Nodes == null) return;
49       for (NumNode N = Nodes; N != null; N = N.Next)
50          System.out.println(N.Value);
51    }
52
53 }
```

```
Compiled from "NumNode.java"
public class NumNode extends java.lang.Object{
int Value;

NumNode Next;

public NumNode(int);
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
```

```
   4:   aload_0
   5:   iload_1
   6:   putfield        #2; //Field Value:I
   9:   aload_0
   10:  aconst_null
   11:  putfield        #3; //Field Next:LNumNode;
   14:  return

public static NumNode Head();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   areturn

public void Insert();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       11
   6:   aload_0
   7:   putstatic       #4; //Field Nodes:LNumNode;
   10:  return
   11:  aload_0
   12:  getfield        #2; //Field Value:I
   15:  getstatic       #4; //Field Nodes:LNumNode;
   18:  getfield        #2; //Field Value:I
   21:  if_icmpge       36
   24:  aload_0
   25:  getstatic       #4; //Field Nodes:LNumNode;
   28:  putfield        #3; //Field Next:LNumNode;
   31:  aload_0
   32:  putstatic       #4; //Field Nodes:LNumNode;
   35:  return
   36:  getstatic       #4; //Field Nodes:LNumNode;
   39:  getfield        #3; //Field Next:LNumNode;
   42:  ifnonnull       53
   45:  getstatic       #4; //Field Nodes:LNumNode;
   48:  aload_0
   49:  putfield        #3; //Field Next:LNumNode;
   52:  return
   53:  getstatic       #4; //Field Nodes:LNumNode;
   56:  astore_1
   57:  aload_1
   58:  getfield        #3; //Field Next:LNumNode;
   61:  ifnull  119
   64:  aload_0
   65:  getfield        #2; //Field Value:I
   68:  aload_1
   69:  getfield        #3; //Field Next:LNumNode;
   72:  getfield        #2; //Field Value:I
   75:  if_icmpge       92
   78:  aload_0
   79:  aload_1
   80:  getfield        #3; //Field Next:LNumNode;
   83:  putfield        #3; //Field Next:LNumNode;
   86:  aload_1
   87:  aload_0
   88:  putfield        #3; //Field Next:LNumNode;
   91:  return
   92:  aload_1
```

```
   93:  getfield        #3; //Field Next:LNumNode;
   96:  getfield        #3; //Field Next:LNumNode;
   99:  ifnonnull       111
   102: aload_1
   103: getfield        #3; //Field Next:LNumNode;
   106: aload_0
   107: putfield        #3; //Field Next:LNumNode;
   110: return
   111: aload_1
   112: getfield        #3; //Field Next:LNumNode;
   115: astore_1
   116: goto    57
   119: return

public static void PrintList();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       7
   6:   return
   7:   getstatic       #4; //Field Nodes:LNumNode;
   10:  astore_0
   11:  aload_0
   12:  ifnull  33
   15:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   18:  aload_0
   19:  getfield        #2; //Field Value:I
   22:  invokevirtual   #6; //Method java/io/PrintStream.println:(I)V
   25:  aload_0
   26:  getfield        #3; //Field Next:LNumNode;
   29:  astore_0
   30:  goto    11
   33:  return

static {};
  Code:
   0:   aconst_null
   1:   putstatic       #4; //Field Nodes:LNumNode;
   4:   return

}
```

You may notice some new instructions in this example. There are actually only six of them—**aconst_null**, **ifnull**, **ifnonnull**, **getfield**, **putfield**, and **areturn**.

The first three of these all deal with the value `NULL`, the value for null pointers. The instruction **aconst_null** is very simple. All it does is push the value of `NULL` onto the stack, just like **iconst_4** would push the value of 4 onto the stack.[15]

The two instructions **ifnull** and **ifnonnull** are conditional jumps, just like **if_cmpeq**. However, instead of comparing the first item on the stack to the second one, they jump if the value on the top of the stack is a `NULL` value (for **ifnull**) or not `NULL` (for **ifnonull**).

[15] Recall that the 'a' means "address," while 'i' means "integer."

Let's look at one of these new conditional jumps in action. Here's the first line of the function **PrintList()**:

```
if (Nodes == null) return;
```

This line compiles to the following three instructions:

```
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnonnull       7
   6:   return
```

It's interesting to note that an "If Nodes **is** `NULL`..." source code line generates a "If Nodes is **not** `NULL`..." instruction (**ifnonnull**) in the compiled code! The reason? It takes fewer instructions to do it that way. Consider the alternative:

```
public static void PrintList();
  Code:
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   ifnull          9
   6:   goto            10
   9:   return
```

The instruction **areturn**[16] does basically the same thing as **ireturn**, except that it doesn't return an integer as a return value. Instead, it returns an object referance—a pointer to something other than your usual types like **int** and **float**.[17] For example, the statement

```
1  return Nodes;
```

in **Head()** compiles to

```
   0:   getstatic       #4; //Field Nodes:LNumNode;
   3:   areturn
```

The **getstatic** instruction pushes the value of **Nodes** onto the stack, and then the instruction **areturn** pops that value, and pushes it onto the stack of the function which called **Head()**.

The last two new instructions, **getfield** and **putfield** work basically the same as **getstatic** and **putstatic**, except that they load and save values to and from non-static member variables ("fields") in a class. For instance, in the line

[16] That's *a return*, not *are turn*.

[17] Again, the 'a' stands for "address."

```
1  if (Value < Nodes.Value)  {
```

the fetching of **Value**, whose "full name" is **this.Value**, compiles to

```
1  11:  aload_0
2  12:  getfield        #2; //Field Value:I
```

The **aload_0** instruction pushes the address in Slot 0, which is **this**, onto the stack. The **getfield** instruction—yes, keep in mind that this is a JVM machine instruction—then gets the **Value** field from within the class instance pointed to by **this**.

The file **Intro.java** illustrates the usage of the **NumNode** class:

```
1  // usage:  java Intro nums
2
3  // reads integers from the command line, storing them in a linear linked
4  // list, maintaining ascending order, and then prints out the final list
5  // to the screen
6
7  public class Intro
8
9  {  public static void main(String[] Args) {
10        int NumElements = Args.length;
11        for (int I = 1; I <= NumElements; I++)  {
12           int Num;
13           Num = Integer.parseInt(Args[I-1]);
14           NumNode NN = new NumNode(Num);
15           NN.Insert();
16        }
17        System.out.println("final sorted list:");
18        NumNode.PrintList();
19     }
20 }
```

It compiles to

```
Compiled from Intro.java
public class Intro extends java.lang.Object {
    public Intro();
    public static void main(java.lang.String[]);
}

Method Intro()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 aload_0
   1 arraylength
```

```
 2 istore_1
 3 iconst_1
 4 istore_2
 5 goto 35
 8 aload_0
 9 iload_2
10 iconst_1
11 isub
12 aaload
13 invokestatic #2 <Method int parseInt(java.lang.String)>
16 istore_3
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
27 aload 4
29 invokevirtual #5 <Method void Insert()>
32 iinc 2 1
35 iload_2
36 iload_1
37 if_icmple 8
40 getstatic #6 <Field java.io.PrintStream out>
43 ldc #7 <String "final sorted list:">
45 invokevirtual #8 <Method void println(java.lang.String)>
48 invokestatic #9 <Method void PrintList()>
51 return
```

There are again some new instructions to discuss here. First, consider the Java statement

```
1 int NumElements = Args.length;
```

in **Intro.java**. Java is a more purely object-oriented language than C++, and one illustration of that is that in Java arrays are objects. One of the member variables in the **Array** class is **length**, the number of elements in the array. Thus the compiler translates the fetch of **Args.length** to

```
1 0 aload_0
2 1 arraylength
```

Note again that **arraylength** is a JVM machine instruction. This is not a subroutine call.

Now consider the statement

```
1 NumNode NN = new NumNode(Num);
```

It compiles to

```
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
```

The JVM instruction set includes an instruction **new**, which allocates space from the heap for the object to be created of class **NumNode**. The instruction pushes a pointer to that space. Next, the **dup** ("duplicate") instruction pushes a second copy of that pointer, to be used later in the compiled code for

```
1 NN.Insert();
```

But before then we still need to call the constructor for the **NumNode** class, which is done in offset 22, after pushing the parameter in offset 21. The assignment of the pointer to **NN** then is done in offset 25.

10.8 Overview of JVM Instructions

In the following, *top* will refer to the element at the top of the operand stack, and *nexttop* will refer to the element next to it.

- **aaload**:

  Format: 0x32

  Loads an element of an array of addresses (i.e. from a multidimensional array). Treats *nexttop* as a pointer to an array (i.e. a variable declared of array type), and *top* is an index into the array. The instruction loads the array element and pushes it onto the operand stack. In other words, *nexttop* and *top* are popped, and *nexttop*[*top*] is fetched and pushed onto the operand stack.

- **arraylength**:

  Format: 0xbe

  Pops the operand stack to get the array address, and then pushes the length of the array.

- **aastore**:

  Format: 0x53

  Does the opposite of **aaload**, popping the operand stack first to get *value*, then to get *index*, the finally to get *address*. It then stores *value* into element *index* of the array starting at *address*.

- **aconst_null**:

  Format: 0x01

  Pushes the value `NULL` onto the stack.

- **aload**:

  Format: 0x19 *8bitindex*

  Treats the value in slot *8bitindex* as an address, and pushes it onto the stack.

- **aload_0**, **aload_1**, **aload_2**, **aload_3**:

  Formats: 0x2a, 0x2b, 0x2c

  The instruction **aload_0**, is the same as **aload**, but specifically for slot 0. The others are analogous.

- **areturn**:

  Format: 0xb0

  Does exactly the same thing as **ireturn**, but instead of returning an integer, it returns an object referance.

- **astore**:

  Format: 0x3a, *8bitindex*

  Opposite of **aload**, popping the operand stack and placing the popped value (presumed to be an address) into the specified slot.

- **astore_0**, **astore_1**, **astore_2**, **astore_3**:

  Formats: 0x4b, 0x4c, 0x4d, 0x4e

  Same as **astore**, but specifically for slot 0, slot 1 etc..

- **bipush**:

  Format: 0x10 *8bitinteger*

  Pushes the given 8-bit integer onto the operand stack.

- **dup**:

  Format: 0x59

  Duplicates the top word on the operand stack, so the operand stack now has two copies of that word instead of one.

- **getfield**:

  Format: 0xb5 *16bitindex*

  Opposite of **putfield**. Gets value of the given non-static item, then pushes it onto the operand stack.

- **getstatic**:

  Format: 0xb2 *16bitindex*

  Opposite of **putstatic**. Gets value of the given static item, then pushes it on the operand stack.

- **goto**:

  Format: 0xa7 *16bitjumpdistance*

  Unconditional jump. See **if_icmpeq** for explanation of *16bitjumpdistance*.

- **i2f**:

  Format: 0x86

  Pops the top word on the stack, converts it from **int** to **float**, and pushes the new value back onto the stack.

- **iadd**:

  Format: 0x60

  Pops *nexttop* and *top*, and pushes the sum *nexttop* + *top*.

- **iaload**:

  Format: 0x2e

  Load an element of an integer array. See **aaload** above.

- **iastore**:

  Format: 0x4f

  Store to an element of an integer array. See **aastore** above.

- **iconst_0**, **iconst_1**, **iconst_2**, **iconst_3**, **iconst_4**, **iconst_5**:

  Format: 0x3, 0x4, 0x5, 0x6, 0x7, 0x8

  Pushes the integer constant 0, 1, 2 etc. onto the operand stack.

- **idiv**:

  Format: 0x6c

  Pops *nexttop* and *top*, and pushes the quotient *nexttop* / *top*.

- **if_icmpeq**:

  Format: 0x9f *16bitjumpdistance*

  If *top* = *nexttop*, jumps the given distance to the branch target. The quantity *16bitjumpdistance* is a 2-byte, 2s complement signed number, measured from the jump instruction. Both *top* and *nexttop* must be integers; a runtime error occurs if no.

- **if_icmpge**:

  Format: 0xa2 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if *nexttop* $\geq$ *top*.

- **if_icmple**:

  Format: 0xa4 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if *nexttop* $\leq$ *top*.

- **if_icmplt**:

  Format: 0xa1 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if *nexttop* $<$ *top*.

- **if_icmpne**:

  Format: 0xa0 *16bitjumpdistance*

  Same as **if_icmpeq**, but jumps if *top* $\neq$ *nexttop*.

- **ifnonnull**:

  Format: 0xc7 *16bitjumpdistance*

  Same as the previous jump conditions, but jumps if *top* $\neq$ `NULL`.

- **ifnull**:

  Format: 0xc6 *16bitjumpdistance*

  Same as the previous jump conditions, but jumps if *top* = `NULL`.

- **iinc**:

  Format: 0x84 *8bitindex 8bitinteger*

  Increments slot *8bitindex* by the amount *8bitinteger*.

- **iload**:

  Format: 0x15 *8bitindex*

  Same as **aload** but for integers instead of addresses.

- **iload_0**, **iload_1**, **iload_2**, **iload_3**:

  Formats: 0x1a, 0x1b, 0x1c, 0x1d

  Same as **aload_0** etc. but for integers instead of addresses.

- **imul**:

  Format: 0x68

  Pops *nexttop* and *top*, and pushes the product *nexttop* $\times$ *top*.

- **invokespecial**:

  Format: 0xb7 *16bitindex*

  Like **invokevirtual**, but for constructors, superclass and other special situations.

- **invokestatic**:

  Format: 0xb8 *16bitindex*

  Method call. The quantity *16bitindex* serves as an index into the Constant Pool, pointing to the given method. Creates a new stack frame for the method. Pops the method's arguments from the caller's operand stack and places them into the method's Local Variables Section. Points the **frame** register to the method's stack frame, and jumps to the method.

- **invokevirtual**:

  Format: 0xb6 *16bitindex*

  Same as **invokestatic**, but the arguments include the "hidden" argument **this**, i.e. a pointer to the object this method is being invoked on.

- **ireturn**:

  Format: 0xac

  Return from method with return value. Pops integer from current operand stack, places it on the caller's operand stack, restores the **frame** register to point to the caller's stack frame, and jumps back to the caller.

- **istore**:

  Format: 0x36 *8bitindex*

  Same as **astore** but for integers instead of addresses.

- **istore_0**, **istore_1**, **istore_2**, **istore_3**:

  Formats: 0x3b, 0x3c, 0x3d, 0x3e

  Same as **astore_0** etc. but for integers instead of addresses.

- **isub**:

  Format: 0x64

  Pops *nexttop* and *top*, and pushes the difference *nexttop* - *top*.

- **ldc**:

  Format: 0x12 *8bitindex*

  Gets an item from the Constant Pool and pushes it onto the operand stack.

- **new**:

  Format: 0xbb *16bitindex*

  Performs the Java **new** operation, with *16bitindex* being the index into the Constant Pool, pointing to the given class. Creates the given object using memory from the Heap, and then pushes the address of the new object on the operand stack.

- **putfield**:

  Format: 0xb4 *16bitindex*

  Pops the operand stack, and assigns the popped value to the non-static item given by *16bitindex*.

- **putstatic**:

  Format: 0xb3 *16bitindex*

  Pops the operand stack, and assigns the popped value to the static item given by *16bitindex*.

- **return**:

  Format: 0xb1

  Same as **ireturn**, but for methods without a return value.

- **swap**:

  Format: 0x5f

  Swaps the top two words of the operand stack.

10.9 References

- Bill Venners' book, *Inside the Java Virtual Machine*. Available on the Web (with many helpful URL links) at `www.artima.com`.
- Sun's "official" definition of the JVM: *The Java Virtual Machine Specification*, by Lindholm and Yellin, also available on the Web, at `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`. Chapter 6 gives specs (mnemonics, op codes, actions, etc.) on the entire JVM instruction set.