PC Assembly Language

Paul A. Carter
Adapted for SCC's CISP310 by James R. Town

November 28, 2022

Contents

Chapter 1

Introduction

1.1 Number Systems

Memory in a computer consists of numbers. Computer memory does not store these numbers in decimal (base 10). Because it greatly simplifies the hardware, computers store all information in a binary (base 2) format. First let's review the decimal system.

1.1.1 Decimal

Base 10 numbers are composed of 10 possible digits (0-9). Each digit of a number has a power of 10 associated with it based on its position in the number. For example:

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 Binary

Base 2 numbers are composed of 2 possible digits (0 and 1). Each digit of a number has a power of 2 associated with it based on its position in the number. (A single binary digit is called a bit.) For example[1]:

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

This shows how binary may be converted to decimal. Table 1.1 shows how the first few numbers are represented in binary.

Figure 1.1 shows how individual binary digits (*i.e.*, bits) are added. Here's an example:

[1]The 2 subscript is used to show that the number is represented in binary, not decimal

Decimal	Binary		Decimal	Binary
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Table 1.1: Decimal 0 to 15 in Binary

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

Figure 1.1: Binary addition

If one considers the following decimal division:

$$1234 \div 10 = 123\ r\ 4$$

he can see that this division strips off the rightmost decimal digit of the number and shifts the other decimal digits one position to the right. Dividing by two performs a similar operation, but for the binary digits of the number. Consider the following binary division:

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

This fact can be used to convert a decimal number to its equivalent binary representation as Figure 1.2 shows. This method finds the rightmost digit first, this digit is called the *least significant bit* (lsb). The leftmost digit is called the *most significant bit* (msb). The basic unit of memory consists of 8 bits and is called a *byte*.

1.1.3 Hexadecimal

Hexadecimal numbers use base 16. Hexadecimal (or *hex* for short) can be used as a shorthand for binary numbers. Hex has 16 possible digits. This creates a problem since there are no symbols to use for these extra digits after 9. By convention, letters are used for these extra digits. The 16 hex digits are 0-9 then A, B, C, D, E and F. The digit A is equivalent to 10

Decimal	Binary
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

Thus $25_{10} = 11001_2$

Figure 1.2: Decimal conversion to Binary

$$\begin{array}{rcl} 589 \div 16 & = & 36\ r\ 13 \\ 36 \div 16 & = & 2\ r\ 4 \\ 2 \div 16 & = & 0\ r\ 2 \end{array}$$

Thus $589 = 24\text{D}_{16}$

Figure 1.3: Decimal conversion to Hexadecimal

in decimal, B is 11, etc. Each digit of a hex number has a power of 16 associated with it. Example:

$$\begin{array}{rcl} 2\text{BD}_{16} & = & 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ & = & 512 + 176 + 13 \\ & = & 701 \end{array}$$

To convert from decimal to hex, use the same idea that was used for binary conversion except divide by 16. See Figure 1.3 for an example.

The reason that hex is useful is that there is a very simple way to convert between hex and binary. Binary numbers get large and cumbersome quickly. Hex provides a much more compact way to represent binary.

To convert a hex number to binary, simply convert each hex digit to a 4-bit binary number. For example, 24D_{16} is converted to $0010\ 0100\ 1101_2$. Note that the leading zeros of the 4-bits are important! If the leading zero for the middle digit of 24D_{16} is not used the result is wrong. Converting from binary to hex is just as easy. One does the process in reverse. Convert each 4-bit segments of the binary to hex. Start from the right end, not the left end of the binary number. This ensures that the process uses the correct 4-bit segments[2]. Example:

[2]If it is not clear why the starting point makes a difference, try converting the example

Octal	Binary		Octal	Binary
0	000		10	001 000
1	001		11	001 001
2	010		12	001 010
3	011		13	001 011
4	100		14	001 100
5	101		15	001 101
6	110		16	001 110
7	111		17	001 111

Table 1.2: Octal 0 to 17 in Binary

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

A 4-bit number is called a *nibble* . Thus each hex digit corresponds to a nibble. Two nibbles make a byte and so a byte can be represented by a 2-digit hex number. A byte's value ranges from 0 to 11111111 in binary, 0 to FF in hex and 0 to 255 in decimal.

1.1.4 Octal

Octal numbers use base 8. Less common than hexidecimal, octal can also be used as a shorthand for binary numbers. It is often used when the numbers can be broken into groups of three bits. The first fifteen numbers in octal are shown with their binary equivalents in Table 1.2.

1.2 Computer Organization

1.2.1 Memory

Memory is measured in units of kilobytes ($2^{10} = 1,024$ bytes), megabytes ($2^{20} = 1,048,576$ bytes) and gigabytes ($2^{30} = 1,073,741,824$ bytes).

The basic unit of memory is a byte. A computer with 32 megabytes of memory can hold roughly 32 million bytes of information. Each byte in memory is labeled by a unique number known as its address as Figure 1.4 shows.

Often memory is used in larger chunks than single bytes. On the PC architecture, names have been given to these larger sections of memory as Table 1.3 shows.

All data in memory is numeric. Characters are stored by using a *character code* that maps numbers to characters. One of the most common

starting at the left.

Address	0	1	2	3	4	5	6	7
Memory	2A	45	B8	20	8F	CD	12	2E

Figure 1.4: Memory Addresses

Term	Size
nibble	1/2 bytes
byte	1 byte
word	2 bytes
double word	4 bytes
quad word	8 bytes
paragraph	16 bytes

Table 1.3: Units of Memory

character codes is known as *ASCII* (American Standard Code for Information Interchange). A new, more complete code that is supplanting ASCII is Unicode. One key difference between the two codes is that ASCII uses one byte to encode a character, but Unicode uses multiple bytes. There are several different forms of Unicode. On x86 C/C++ compilers, Unicode is represented in code using the `wchar_t` type and the UTF-16 encoding which uses 16 bits (or a *word*) per character. For example, ASCII maps the byte 41_{16} (65_{10}) to the character capital *A*; UTF-16 maps it to the word 0041_{16}. Since ASCII uses a byte, it is limited to only 256 different characters[3]. Unicode extends the ASCII values and allows many more characters to be represented. This is important for representing characters for all the languages of the world.

1.2.2 The CPU

The Central Processing Unit (CPU) is the physical device that performs instructions. The instructions that CPUs perform are generally very simple. Instructions may require the data they act on to be in special storage locations in the CPU itself called *registers*. The CPU can access data in registers much faster than data in memory. However, the number of registers in a CPU is limited, so the programmer must take care to keep only currently used data in registers.

The instructions a type of CPU executes make up the CPU's *machine language*. Machine programs have a much more basic structure than higher-level languages. Machine language instructions are encoded as raw numbers,

[3]In fact, ASCII only uses the lower 7-bits and so only has 128 different values to use.

not in friendly text formats. A CPU must be able to decode an instruction's purpose very quickly to run efficiently. Machine language is designed with this goal in mind, not to be easily deciphered by humans. Programs written in other languages must be converted to the native machine language of the CPU to run on the computer. A *compiler* is a program that translates programs written in a programming language into the machine language of a particular computer architecture. In general, every type of CPU has its own unique machine language. This is one reason why programs written for a Mac can not run on an IBM-type PC.

GHz stands for gigahertz or one billion cycles per second. A 1.5 GHz CPU has 1.5 billion clock pulses per second.

Computers use a *clock* to synchronize the execution of the instructions. The clock pulses at a fixed frequency (known as the *clock speed*). When you buy a 1.5 GHz computer, 1.5 GHz is the frequency of this clock[4]. The clock does not keep track of minutes and seconds. It simply beats at a constant rate. The electronics of the CPU uses the beats to perform their operations correctly, like how the beats of a metronome help one play music at the correct rhythm. The number of beats (or as they are usually called *cycles*) an instruction requires depends on the CPU generation and model. The number of cycles depends on the instructions before it and other factors as well.

1.2.3 The 80x86 family of CPUs

IBM-type PC's contain a CPU from Intel's 80x86 family (or a clone of one). The CPU's in this family all have some common features including a base machine language. However, the more recent members greatly enhance the features.

8088,8086: These CPU's from the programming standpoint are identical. They were the CPU's used in the earliest PC's. They provide several 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, SS, ES, IP, FLAGS. They only support up to one megabyte of memory and only operate in *real mode*. In this mode, a program may access any memory address, even the memory of other programs! This makes debugging and security very difficult! Also, program memory has to be divided into *segments*. Each segment can not be larger than 64K.

80286: This CPU was used in AT class PC's. It adds some new instructions to the base machine language of the 8088/86. However, its main new feature is *16-bit protected mode*. In this mode, it can access up to 16 megabytes and protect programs from accessing each other's memory. However, programs are still divided into segments that could not be bigger than 64K.

[4] Actually, clock pulses are used in many different components of a computer. The other components often use different clock speeds than the CPU.

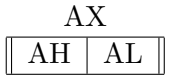

Figure 1.5: The AX register

80386: This CPU greatly enhanced the 80286. First, it extends many of the registers to hold 32-bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) and adds two new 16-bit registers FS and GS. It also adds a new *32-bit protected mode*. In this mode, it can access up to 4 gigabytes. Programs are again divided into segments, but now each segment can also be up to 4 gigabytes in size!

80486/Pentium/Pentium Pro: These members of the 80x86 family add very few new features. They mainly speed up the execution of the instructions.

Pentium MMX: This processor adds the MMX (MultiMedia eXtensions) instructions to the Pentium. These instructions can speed up common graphics operations.

Pentium II: This is the Pentium Pro processor with the MMX instructions added. (The Pentium III is essentially just a faster Pentium II.)

1.2.4 8086 16-bit Registers

The original 8086 CPU provided four 16-bit general purpose registers: AX, BX, CX and DX. Each of these registers could be decomposed into two 8-bit registers. For example, the AX register could be decomposed into the AH and AL registers as Figure 1.5 shows. The AH register contains the upper (or high) 8 bits of AX and AL contains the lower 8 bits of AX. Often AH and AL are used as independent one byte registers; however, it is important to realize that they are not independent of AX. Changing AX's value will change AH and AL and *vice versa*. The general purpose registers are used in many of the data movement and arithmetic instructions.

There are two 16-bit index registers: SI and DI. They are often used as pointers, but can be used for many of the same purposes as the general registers. However, they can not be decomposed into 8-bit registers.

The 16-bit BP and SP registers are used to point to data in the machine language stack and are called the Base Pointer and Stack Pointer, respectively. These will be discussed later.

The 16-bit CS, DS, SS and ES registers are *segment registers*. They denote what memory is used for different parts of a program. CS stands

for Code Segment, DS for Data Segment, SS for Stack Segment and ES for Extra Segment. ES is used as a temporary segment register. The details of these registers are in Sections 1.2.6 and 1.2.7.

The Instruction Pointer (IP) register is used with the CS register to keep track of the address of the next instruction to be executed by the CPU. Normally, as an instruction is executed, IP is advanced to point to the next instruction in memory.

The FLAGS register stores important information about the results of a previous instruction. These results are stored as individual bits in the register. For example, the Z bit is 1 if the result of the previous instruction was zero or 0 if not zero. Not all instructions modify the bits in FLAGS, consult the table in the appendix to see how individual instructions affect the FLAGS register.

1.2.5 80386 32-bit registers

The 80386 and later processors have extended registers. For example, the 16-bit AX register is extended to be 32-bits. To be backward compatible, AX still refers to the 16-bit register and EAX is used to refer to the extended 32-bit register. AX is the lower 16-bits of EAX just as AL is the lower 8-bits of AX (and EAX). There is no way to access the upper 16-bits of EAX directly. The other extended registers are EBX, ECX, EDX, ESI and EDI.

Many of the other registers are extended as well. BP becomes EBP; SP becomes ESP; FLAGS becomes EFLAGS and IP becomes EIP. However, unlike the index and general purpose registers, in 32-bit protected mode (discussed below) only the extended versions of these registers are used.

The segment registers are still 16-bit in the 80386. There are also two new segment registers: FS and GS. Their names do not stand for anything. They are extra temporary segment registers (like ES).

One of definitions of the term *word* refers to the size of the data registers of the CPU. For the 80x86 family, the term is now a little confusing. In Table 1.3, one sees that *word* is defined to be 2 bytes (or 16 bits). It was given this meaning when the 8086 was first released. When the 80386 was developed, it was decided to leave the definition of *word* unchanged, even though the register size changed.

1.2.6 Real Mode

So where did the infamous DOS 640K limit come from? The BIOS required some of the 1M for its code and for hardware devices like the video screen.

In real mode, memory is limited to only one megabyte (2^{20} bytes). Valid address range from (in hex) 00000 to FFFFF. These addresses require a 20-bit number. Obviously, a 20-bit number will not fit into any of the 8086's 16-bit registers. Intel solved this problem, by using two 16-bit values to determine an address. The first 16-bit value is called the *selector*. Selector

values must be stored in segment registers. The second 16-bit value is called the *offset*. The physical address referenced by a 32-bit *selector:offset* pair is computed by the formula

$$16 * \text{selector} + \text{offset}$$

Multiplying by 16 in hex is easy, just add a 0 to the right of the number. For example, the physical addresses referenced by 047C:0048 is given by:

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

In effect, the selector value is a paragraph number (see Table 1.3).

Real segmented addresses have disadvantages:

- A single selector value can only reference 64K of memory (the upper limit of the 16-bit offset). What if a program has more than 64K of code? A single value in CS can not be used for the entire execution of the program. The program must be split up into sections (called *segments*) less than 64K in size. When execution moves from one segment to another, the value of CS must be changed. Similar problems occur with large amounts of data and the DS register. This can be very awkward!

- Each byte in memory does not have a unique segmented address. The physical address 04808 can be referenced by 047C:0048, 047D:0038, 047E:0028 or 047B:0058. This can complicate the comparison of segmented addresses.

1.2.7 16-bit Protected Mode

In the 80286's 16-bit protected mode, selector values are interpreted completely differently than in real mode. In real mode, a selector value is a paragraph number of physical memory. In protected mode, a selector value is an *index* into a *descriptor table*. In both modes, programs are divided into segments. In real mode, these segments are at fixed positions in physical memory and the selector value denotes the paragraph number of the beginning of the segment. In protected mode, the segments are not at fixed positions in physical memory. In fact, they do not have to be in memory at all!

Protected mode uses a technique called *virtual memory* . The basic idea of a virtual memory system is to only keep the data and code in memory that programs are currently using. Other data and code are stored temporarily

on disk until they are needed again. In 16-bit protected mode, segments are moved between memory and disk as needed. When a segment is returned to memory from disk, it is very likely that it will be put into a different area of memory that it was in before being moved to disk. All of this is done transparently by the operating system. The program does not have to be written differently for virtual memory to work.

In protected mode, each segment is assigned an entry in a descriptor table. This entry has all the information that the system needs to know about the segment. This information includes: is it currently in memory; if in memory, where is it; access permissions (*e.g.*, read-only). The index of the entry of the segment is the selector value that is stored in segment registers.

One well-known PC columnist called the 286 CPU "brain dead."

One big disadvantage of 16-bit protected mode is that offsets are still 16-bit quantities. As a consequence of this, segment sizes are still limited to at most 64K. This makes the use of large arrays problematic!

1.2.8 32-bit Protected Mode

The 80386 introduced 32-bit protected mode. There are two major differences between 386 32-bit and 286 16-bit protected modes:

1. Offsets are expanded to be 32-bits. This allows an offset to range up to 4 billion. Thus, segments can have sizes up to 4 gigabytes.

2. Segments can be divided into smaller 4K-sized units called *pages*. The virtual memory system works with pages now instead of segments. This means that only parts of segment may be in memory at any one time. In 286 16-bit mode, either the entire segment is in memory or none of it is. This is not practical with the larger segments that 32-bit mode allows.

In Windows 3.x, *standard mode* referred to 286 16-bit protected mode and *enhanced mode* referred to 32-bit mode. Windows 9X, Windows NT/2000/XP, OS/2 and Linux all run in paged 32-bit protected mode.

1.2.9 Interrupts

Sometimes the ordinary flow of a program must be interrupted to process events that require prompt response. The hardware of a computer provides a mechanism called *interrupts* to handle these events. For example, when a mouse is moved, the mouse hardware interrupts the current program to handle the mouse movement (to move the mouse cursor, *etc.*) Interrupts cause control to be passed to an *interrupt handler*. Interrupt handlers are routines that process the interrupt. Each type of interrupt is assigned an

integer number. At the beginning of physical memory, a table of *interrupt vectors* resides that contain the segmented addresses of the interrupt handlers. The number of interrupt is essentially an index into this table.

External interrupts are raised from outside the CPU. (The mouse is an example of this type.) Many I/O devices raise interrupts (*e.g.*, keyboard, timer, disk drives, CD-ROM and sound cards). Internal interrupts are raised from within the CPU, either from an error or the interrupt instruction. Error interrupts are also called *traps*. Interrupts generated from the interrupt instruction are called *software interrupts*. DOS uses these types of interrupts to implement its API (Application Programming Interface). More modern operating systems (such as Windows and UNIX) use a C based interface. [5]

Many interrupt handlers return control back to the interrupted program when they finish. They restore all the registers to the same values they had before the interrupt occurred. Thus, the interrupted program runs as if nothing happened (except that it lost some CPU cycles). Traps generally do not return. Often they abort the program.

1.3 Assembly Language

1.3.1 Machine language

Every type of CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own unique numeric code called its *operation code* or *opcode* for short. The 80x86 processor's instructions vary in size. The opcode is always at the beginning of the instruction. Many instructions also include data (*e.g.*, constants or addresses) used by the instruction.

Machine language is very difficult to program in directly. Deciphering the meanings of the numerical-coded instructions is tedious for humans. For example, the instruction that says to add the EAX and EBX registers together and store the result back into EAX is encoded by the following hex codes:

```
03 C3
```

This is hardly obvious. Fortunately, a program called an *assembler* can do this tedious work for the programmer.

1.3.2 Assembly language

An assembly language program is stored as text (just as a higher level language program). Each assembly instruction represents exactly one ma-

[5] However, they may use a lower level interface at the kernel level.

chine instruction. For example, the addition instruction described above would be represented in assembly language as:

```
add eax, ebx
```

Here the meaning of the instruction is *much* clearer than in machine code. The word `add` is a *mnemonic* for the addition instruction. The general form of an assembly instruction is:

```
mnemonic operand(s)
```

It took several years for computer scientists to figure out how to even write a compiler!

An *assembler* is a program that reads a text file with assembly instructions and converts the assembly into machine code. *Compilers* are programs that do similar conversions for high-level programming languages. An assembler is much simpler than a compiler. Every assembly language statement directly represents a single machine instruction. High-level language statements are *much* more complex and may require many machine instructions.

Another important difference between assembly and high-level languages is that since every different type of CPU has its own machine language, it also has its own assembly language. Porting assembly programs between different computer architectures is *much* more difficult than in a high-level language.

This book's examples uses the Microsoft's Assembler or MASM for short. It is freely available off the Internet. More common assemblers are Netwide Assembler (NASM) or Borland's Assembler (TASM). There are some differences in the assembly syntax for MASM/TASM and NASM, but the big ideas are the same.

1.3.3 Instruction operands

Machine code instructions have varying number and type of operands; however, in general, each instruction itself will have a fixed number of operands (0 to 3). Operands can have the following types:

register: These operands refer directly to the contents of the CPU's registers.

memory: These refer to data in memory. The address of the data may be a constant hardcoded into the instruction or may be computed using values of registers. Address are always offsets from the beginning of a segment.

immediate: These are fixed values that are listed in the instruction itself. They are stored in the instruction itself (in the code segment), not in the data segment.

implied: These operands are not explicitly shown. For example, the increment instruction adds one to a register or memory. The one is implied.

1.3.4 Basic instructions

The most basic instruction is the `MOV` instruction. It moves data from one location to another (like the assignment operator in a high-level language). It takes two operands:

```
mov dest, src
```

The data specified by *src* is copied to *dest*. One restriction is that both operands may not be memory operands. This points out another quirk of assembly. There are often somewhat arbitrary rules about how the various instructions are used. The operands must also be the same size. The value of AX can not be stored into BL.

Here is an example (semicolons start a comment):

```
1  mov   eax, 3  ; store 3 into EAX register (3 is immediate operand)
2  mov   bx, ax  ; store the value of AX into the BX register
```

The `ADD` instruction is used to add integers.

```
1  add   eax, 4  ; eax = eax + 4
2  add   al, ah  ; al = al + ah
```

The `SUB` instruction subtracts integers.

```
1  sub   bx, 10  ; bx = bx - 10
2  sub   ebx, edi ; ebx = ebx - edi
```

The `INC` and `DEC` instructions increment or decrement values by one. Since the one is an implicit operand, the machine code for `INC` and `DEC` is smaller than for the equivalent `ADD` and `SUB` instructions.

```
1  inc   ecx     ; ecx++
2  dec   dl      ; dl--
```

1.3.5 Directives

A *directive* is an artifact of the assembler not the CPU. They are generally used to either instruct the assembler to do something or inform the assembler of something. They are not translated into machine code. Common uses of directives are:

- define constants

Unit	Letter
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

Table 1.4: Letters for `DX` Directives

- define memory to store data into
- group memory into segments
- conditionally include source code
- include other files

The equ directive

The `equ` directive can be used to define a *symbol*. Symbols are named constants that can be used in the assembly program. The format is:

symbol `equ` *value*

Symbol values can *not* be redefined later.

Data directives

Data directives are used in data segments to define room for memory. There are two ways memory can be reserved. The first way only defines room for data; the second way defines room and an initial value. You can use a ? to reserve uninitialized memory for a given size, or you can give the memory location an initial value. Table 1.4 shows the possible values for the different sizes of data.

It is very common to mark memory locations with *labels*. Labels allow one to easily refer to memory locations in code. Below are several examples:

```
1 L1  db  0       ; byte labeled L1 with initial value 0
2 L2  dw  1000    ; word labeled L2 with initial value 1000
3 L3  db  110101b ; byte initialized to binary 110101 (53 in decimal)
4 L4  db  12h     ; byte initialized to hex 12 (18 in decimal)
5 L5  db  17o     ; byte initialized to octal 17 (15 in decimal)
6 L6  dd  1A92h   ; double word initialized to hex 1A92
7 L7  db  ?       ; 1 uninitialized byte
8 L8  db  "A"     ; byte initialized to ASCII code for A (65)
```

Double quotes and single quotes are treated the same. Consecutive data definitions are stored sequentially in memory. That is, the word L2 is stored

immediately after L1 in memory. Sequences of memory may also be defined, which we'll see later can be treated as arrays.

```
1 L9   db    0, 1, 2, 3            ; defines 4 bytes
2 L10  db    "w", "o", "r", 'd', 0 ; defines a C string = "word"
3 L11  db    'word', 0              ; same as L10
```

The DD directive can be used to define both integer and single precision floating point[6] constants. However, the DQ can only be used to define double precision floating point constants.

For large sequences, DUP directive is often useful. This directive repeats its operand a specified number of times. For example,

```
1 L12  db 100 dup (0)              ; equivalent to 100 (db 0)'s
2 L13  dw 100 dup (?)              ; reserves room for 100 words
```

Remember that labels can be used to refer to data in code. There are two ways that a label can be used. If a plain label is used, it is interpreted as the data at the address (or offset) of the label. If the label is used with the offset operator, it is interpreted as the address of the data. You might notice that labels act like variables in higher level languages. In 32-bit mode, addresses are 32-bit. Here are some examples:

```
1 mov  al, L1     ; copy byte at L1 into AL
2 mov  eax, offset L1   ; EAX = address of byte at L1
3 mov  L1, ah     ; copy AH into byte at L1
4 mov  eax, L6    ; copy double word at L6 into EAX
5 add  eax, L6    ; EAX = EAX + double word at L6
6 add  L6, eax    ; double word at L6 += EAX
7 mov  al, byte ptr L6 ; copy first byte of double word at L6 into AL
```

Line 7 of the examples shows an important property of assembly. The assembler does *not* keep track of the type of data that a label refers to. It is up to the programmer to make sure that he (or she) uses a label correctly. Later it will be common to store addresses of data in registers and use the register like a pointer variable in C. Again, no checking is made that a pointer is used correctly. In this way, assembly is much more error prone than even C.

1.3.6 Input and Output

Input and output are very system dependent activities. It involves interfacing with the system's hardware. High level languages, like C, provide standard libraries of routines that provide a simple, uniform programming interface for I/O. Assembly languages provide no standard libraries. They

[6]Single precision floating point is equivalent to a `float` variable in C.

must either directly access hardware (which is a privileged operation in protected mode) or use whatever low level routines that the operating system provides.

It is very common for assembly routines to be interfaced with C. One advantage of this is that the assembly code can use the standard C library I/O routines. However, one must know the rules for passing information between routines that C uses. These rules are too complicated to cover here.

To start with, you will be creating your own assembly interpreter so you can debug with a language you are already familiar with. Then we'll switch over to "real" assembly and you'll write your own output procedures for debugging.

1.4 Creating a Program

Today, it is unusual to create a stand alone program written completely in assembly language. Assembly is usually used to key certain critical routines. Why? It is *much* easier to program in a higher level language than in assembly. Also, using assembly makes a program very hard to port to other platforms. In fact, it is rare to use assembly at all.

So, why should anyone learn assembly at all?

1. Sometimes code written in assembly can be faster and smaller than compiler generated code.

2. Assembly allows access to direct hardware features of the system that might be difficult or impossible to use from a higher level language.

3. Learning to program in assembly helps one gain a deeper understanding of how computers work.

4. Learning to program in assembly helps one understand better how compilers and high level languages like C work.

These last two points demonstrate that learning assembly can be useful even if one never programs in it later. In fact, the author rarely programs in assembly, but he uses the ideas he learned from it everyday.

1.4.1 First program

As is traditional, here is "Hello, world!" in assembly. We will learn about interrupts (`int`) later, but for now just know that you can now add assembly to your resume.

```
1  title Hello World Program      (hello.asm)
2
3  ;This program displays "Hello, world!"
4  .model small ;small 64k 16 bit
5  .stack 100h ;reserves 256 bytes for the stack
6  .data ;start definition of variables
7  ;define message as an array of characters
8  message db "Hello, world!",0dh,0ah,'$'
9
10 .code ;start code portion of program
11 main proc ;start of the main procedure
12     mov  ax,@data ;load the address of the data segment into ax
13     mov  ds,ax ;load the address of the data segment into ds
14
15     mov  ah,9 ;setting for outputting a string
16     mov  dx,offset message ;point dx to the string
17     int  21h ;output the string
18
19     mov  ax,4C00h ;exit (ah) with code 0 (al)
20     int  21h ;exit
21 main endp ;end procedure
22
23 end main ;end program
```

Line 6 of the program defines a section of the program that specifies memory to be stored in the data segment (whose name is `.data`). On line 7, a string is declared. It will be printed with the interrupt and so must be terminated with a $ character.

The code segment is named `.code`, it is where instructions are placed.

1.4.2 Compiler dependencies

We will be using the MASM assembler in combination with DOSBox both of which can be downloaded from the course website later in the semester.

1.4.3 Assembling the code

The first step is to assemble the code. From the command line, type:

```
ml hello.asm
```

This will assemble and link your program into an executable which will be named hello.exe.

You can run your program (assuming there were no errors) with:

```
hello
```

And it should print:

```
Hello, world!
```

to the console.

1.4.4 Understanding an assembly listing file

The `/l` *`listing-file`* switch can be used to tell `masm` to create a listing file of a given name. This file shows how the code was assembled. Here is how line 8 (in the data segment) appear in the listing file.

```
8  0000 48 65 6C 6C 6F 2C    message db "Hello, world!",0dh,0ah,'$'
9       20 77 6F 72 6C 64
10      21 0D 0A 24
```

The first column in each line is the offset (in hex) of the data in the segment. The second column shows the raw hex values that will be stored. In this case the hex data correspond to ASCII codes. Finally, the text from the source file is displayed on the line. The offsets listed in the second column are very likely *not* the true offsets that the data will be placed at in the complete program. Each module may define its own labels in the data segment (and the other segments, too). In the link step, all these data segment label definitions are combined to form one data segment. The new final offsets are then computed by the linker.

Here is a small section (lines 15 to 16 of the source file) of the text segment in the listing file:

```
15 0005 B4 09          mov  ah,9 ;setting for outputting a string
16 0007 BA 0000 R      mov  dx,offset message ;point dx to the string
```

The second column shows the machine code generated by the assembly. Often the complete code for an instruction can not be computed yet. For example, in line 16 the offset (or address) of `message` is not known until the code is linked. The assembler can compute the op-code for the `mov` instruction (which from the listing is BA), but it doesn't write the offset because the exact value can not be computed yet. In this case, it lists R after to signify that the linker must resolve the address. When the code is linked, the linker will insert the correct offset into the position. Other instructions, like line 15, do not reference any labels. Here the assembler can compute the complete machine code. Here are the same two lines after linking:

```
15 05EF:0005 B4 09           mov  ah,9
16 05EF:0007 BA 0200         mov  dx,0002
```

Notice how the machine code for line 15 is the same, but on line 16 `offset message` is replaced by the offset of the message within the data segment, namely `0002`. Also the `0000 R` has been replaced by `0200` which is the little endian representation of `0002` (see the next section for a description of little endian representation).

Big and Little Endian Representation

Different processors store multibyte integers in different orders in memory. There are two popular methods of storing integers: *big endian* and *little endian*. Big endian is the method that seems the most natural. The biggest (*i.e.* most significant) byte is stored first, then the next biggest, *etc.* For example, the dword 00000004 would be stored as the four bytes 00 00 00 04. IBM mainframes, most RISC processors and Motorola processors all use this big endian method. However, Intel-based processors use the little endian method! Here the least significant byte is stored first. So, 00000004 is stored in memory as 04 00 00 00. This format is hardwired into the CPU and can not be changed. Normally, the programmer does not need to worry about which format is used. However, there are circumstances where it is important.

1. When binary data is transferred between different computers (either from files or through a network).

2. When binary data is written out to memory as a multibyte integer and then read back as individual bytes or *vice versa*.

Endianness does not apply to the order of array elements. The first element of an array is always at the lowest address. This applies to strings (which are just character arrays). Endianness still applies to the individual elements of the arrays.

1.5 Skeleton File

Figure 1.6 shows a skeleton file that can be used as a starting point for writing assembly programs.

```
1  title Title of Program      (skel.asm)
2
3
4  .model small ; one data segment, one code segment
5  .stack 100h ; reserves 256 bytes for the stack
6  .386 ; for 32 bit registers
7  .data ; start definition of variables
8
9     ; your variables go here
10
11 .code ; start code portion of program
12 main proc ; start of the main procedure
13     mov  eax,@data ; load the address of the data segment into eax
14     mov  ds,eax ; load the address of the data segment into ds
15     ; the two previous instructions initalize the data segment
16
17     ; your code goes here
18
19     ; the following two instructions exit cleanly from the program
20     mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar
           to return 0; in C++)
21     int  21h ; exit
22 main endp ; end procedure
23
24 end main ; end program
```

Figure 1.6: Skeleton Program

Chapter 2

Basic Assembly Language

2.1 Working with Integers

2.1.1 Integer representation

Integers come in two flavors: unsigned and signed. Unsigned integers (which are non-negative) are represented in a very straightforward binary manner. The number 200 as an one byte unsigned integer would be represented as by 11001000 (or C8 in hex).

Signed integers (which may be positive or negative) are represented in a more complicated ways. For example, consider -56. $+56$ as a byte would be represented by 00111000. On paper, one could represent -56 as -111000, but how would this be represented in a byte in the computer's memory. How would the minus sign be stored?

There are three general techniques that have been used to represent signed integers in computer memory. All of these methods use the most significant bit of the integer as a *sign bit*. This bit is 0 if the number is positive and 1 if negative.

Signed magnitude

The first method is the simplest and is called *signed magnitude*. It represents the integer as two parts. The first part is the sign bit and the second is the magnitude of the integer. So 56 would be represented as the byte $\underline{0}0111000$ (the sign bit is underlined) and -56 would be $\underline{1}0111000$. The largest byte value would be $\underline{0}1111111$ or $+127$ and the smallest byte value would be $\underline{1}1111111$ or -127. To negate a value, the sign bit is reversed. This method is straightforward, but it does have its drawbacks. First, there are two possible values of zero, $+0$ ($\underline{0}0000000$) and -0 ($\underline{1}0000000$). Since zero is neither positive nor negative, both of these representations should act the same. This complicates the logic of arithmetic for the CPU. Secondly,

general arithmetic is also complicated. If 10 is added to -56, this must be recast as 10 subtracted by 56. Again, this complicates the logic of the CPU.

One's complement

The second method is known as *one's complement* representation. The one's complement of a number is found by reversing each bit in the number. (Another way to look at it is that the new bit value is $1 - \text{oldbitvalue}$.) For example, the one's complement of $\underline{0}0111000$ ($+56$) is $\underline{1}1000111$. In one's complement notation, computing the one's complement is equivalent to negation. Thus, $\underline{1}1000111$ is the representation for -56. Note that the sign bit was automatically changed by one's complement and that as one would expect taking the one's complement twice yields the original number. As for the first method, there are two representations of zero: $\underline{0}0000000$ ($+0$) and $\underline{1}1111111$ (-0). Arithmetic with one's complement numbers is complicated.

There is a handy trick to finding the one's complement of a number in hexadecimal without converting it to binary. The trick is to subtract the hex digit from F (or 15 in decimal). This method assumes that the number of bits in the number is a multiple of 4. Here is an example: $+56$ is represented by 38 in hex. To find the one's complement, subtract each digit from F to get C7 in hex. This agrees with the result above.

Two's complement

The first two methods described were used on early computers. Modern computers use a third method called *two's complement* representation. The two's complement of a number is found by the following two steps:

1. Find the one's complement of the number

2. Add one to the result of step 1

Here's an example using $\underline{0}0111000$ (56). First the one's complement is computed: $\underline{1}1000111$. Then one is added:

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad\quad 1 \\ \hline & \underline{1}1001000 \end{array}$$

In two complement's notation, computing the two's complement is equivalent to negating a number. Thus, $\underline{1}1001000$ is the two's complement representation of -56. Two negations should reproduce the original number. Surprising two's complement does meet this requirement. Take the two's

Number	Hex Representation
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

Table 2.1: Two's Complement Representation

complement of $\underline{1}1001000$ by adding one to the one's complement.

$$\begin{array}{rr} & \underline{0}0110111 \\ + & 1 \\ \hline & \underline{0}0111000 \end{array}$$

When performing the addition in the two's complement operation, the addition of the leftmost bit may produce a carry. This carry is *not* used. Remember that all data on the computer is of some fixed size (in terms of number of bits). Adding two bytes always produces a byte as a result (just as adding two words produces a word, *etc.*) This property is important for two's complement notation. For example, consider zero as a one byte two's complement number ($\underline{0}0000000$). Computing its two complement produces the sum:

$$\begin{array}{rr} & \underline{1}1111111 \\ + & 1 \\ \hline c & \underline{0}0000000 \end{array}$$

where c represents a carry. (Later it will be shown how to detect this carry, but it is not stored in the result.) Thus, in two's complement notation there is only one zero. This makes two's complement arithmetic simpler than the previous methods.

Using two's complement notation, a signed byte can be used to represent the numbers -128 to $+127$. Table 2.1 shows some selected values. If 16 bits are used, the signed numbers $-32,768$ to $+32,767$ can be represented. $+32,767$ is represented by 7FFF, $-32,768$ by 8000, -128 as FF80 and -1 as FFFF. 32 bit two's complement numbers range from -2 billion to $+2$ billion approximately.

The CPU has no idea what a particular byte (or word or double word) is supposed to represent. Assembly does not have the idea of types that a high level language has. How data is interpreted depends on what instruction is used on the data. Whether the hex value FF is considered to represent a signed -1 or a unsigned $+255$ depends on the programmer. The C language

defines signed and unsigned integer types. This allows a C compiler to determine the correct instructions to use with the data.

2.1.2 Sign extension

In assembly, all data has a specified size. It is not uncommon to need to change the size of data to use it with other data. Decreasing size is the easiest.

Decreasing size of data

To decrease the size of data, simply remove the more significant bits of the data. Here's a trivial example:

```
1       mov    ax, 0034h     ; ax = 52 (stored in 16 bits)
2       mov    cl, al        ; cl = lower 8-bits of ax
```

Of course, if the number can not be represented correctly in the smaller size, decreasing the size does not work. For example, if `AX` were 0134h (or 308 in decimal) then the above code would still set `CL` to 34h. This method works with both signed and unsigned numbers. Consider signed numbers, if `AX` was FFFFh (-1 as a word), then `CL` would be FFh (-1 as a byte). However, note that this is not correct if the value in `AX` was unsigned!

The rule for unsigned numbers is that all the bits being removed must be 0 for the conversion to be correct. The rule for signed numbers is that the bits being removed must be either all 1's or all 0's. In addition, the first bit not being removed must have the same value as the removed bits. This bit will be the new sign bit of the smaller value. It is important that it be same as the original sign bit!

Increasing size of data

Increasing the size of data is more complicated than decreasing. Consider the hex byte FF. If it is extended to a word, what value should the word have? It depends on how FF is interpreted. If FF is a unsigned byte (255 in decimal), then the word should be 00FF; however, if it is a signed byte (-1 in decimal), then the word should be FFFF.

In general, to extend an unsigned number, one makes all the new bits of the expanded number 0. Thus, FF becomes 00FF. However, to extend a signed number, one must *extend* the sign bit. This means that the new bits become copies of the sign bit. Since the sign bit of FF is 1, the new bits must also be all ones, to produce FFFF. If the signed number 5A (90 in decimal) was extended, the result would be 005A.

There are several instructions that the 80386 provides for extension of numbers. Remember that the computer does not know whether a number is signed or unsigned. It is up to the programmer to use the correct instruction.

For unsigned numbers, one can simply put zeros in the upper bits using a `MOV` instruction. For example, to extend the byte in AL to an unsigned word in AX:

```
1       mov    ah, 0   ; zero out upper 8-bits
```

However, it is not possible to use a `MOV` instruction to convert the unsigned word in AX to an unsigned double word in EAX. Why not? There is no way to specify the upper 16 bits of EAX in a `MOV`. The 80386 solves this problem by providing a new instruction `MOVZX`. This instruction has two operands. The destination (first operand) must be a 16 or 32 bit register. The source (second operand) may be an 8 or 16 bit register or a byte or word of memory. The other restriction is that the destination must be larger than the source. (Most instructions require the source and destination to be the same size.) Here are some examples:

```
1       movzx  eax, ax     ; extends ax into eax
2       movzx  eax, al     ; extends al into eax
3       movzx  ax, al      ; extends al into ax
4       movzx  ebx, ax     ; extends ax into ebx
```

For signed numbers, there is no easy way to use the `MOV` instruction for any case. The 8086 provided several instructions to extend signed numbers. The `CBW` (Convert Byte to Word) instruction sign extends the AL register into AX. The operands are implicit. The `CWD` (Convert Word to Double word) instruction sign extends AX into DX:AX. The notation DX:AX means to think of the DX and AX registers as one 32 bit register with the upper 16 bits in DX and the lower bits in AX. (Remember that the 8086 did not have any 32 bit registers!) The 80386 added several new instructions. The `CWDE` (Convert Word to Double word Extended) instruction sign extends AX into EAX. The `CDQ` (Convert Double word to Quad word) instruction sign extends EAX into EDX:EAX (64 bits!). Finally, the `MOVSX` instruction works like `MOVZX` except it uses the rules for signed numbers.

Application to C programming

Extending of unsigned and signed integers also occurs in C. Variables in C may be declared as either signed or unsigned (`int` is signed). Consider the code in Figure 2.1. In line 3, the variable `a` is extended using the rules for unsigned values (using `MOVZX`), but in line 4, the signed rules are used for `b` (using `MOVSX`).

ANSI C does not define whether the `char` *type is signed or not, it is up to each individual compiler to decide this. That is why the type is explicitly defined in Figure 2.1.*

```
1 unsigned char uchar = 0xFF;
2 signed char  schar = 0xFF;
3 int a = (int) uchar;   /* a = 255 (0x000000FF) */
4 int b = (int) schar;   /* b = -1 (0xFFFFFFFF) */
```

Figure 2.1:

```
char ch;
while( (ch = fgetc(fp)) != EOF ) {
  /* do something with ch */
}
```

Figure 2.2:

There is a common C programming bug that directly relates to this subject. Consider the code in Figure 2.2. The prototype of `fgetc()`is:

```
int fgetc( FILE * );
```

One might question why does the function return back an `int` since it reads characters? The reason is that it normally does return back an `char` (extended to an `int` value using zero extension). However, there is one value that it may return that is not a character, `EOF`. This is a macro that is usually defined as -1. Thus, `fgetc()` either returns back a `char` extended to an `int` value (which looks like `000000`xx in hex) or `EOF` (which looks like `FFFFFFFF` in hex).

The basic problem with the program in Figure 2.2 is that `fgetc()` returns an `int`, but this value is stored in a `char`. C will truncate the higher order bits to fit the `int` value into the `char`. The only problem is that the numbers (in hex) `000000FF` and `FFFFFFFF` both will be truncated to the byte `FF`. Thus, the while loop can not distinguish between reading the byte `FF` from the file and end of file.

Exactly what the code does in this case, depends on whether `char` is signed or unsigned. Why? Because in line 2, `ch` is compared with `EOF`. Since `EOF` is an `int` value[1], `ch` will be extended to an `int` so that two values being compared are of the same size[2]. As Figure 2.1 showed, where the variable is signed or unsigned is very important.

If `char` is unsigned, `FF` is extended to be `000000FF`. This is compared to `EOF` (`FFFFFFFF`) and found to be not equal. Thus, the loop never ends!

[1]It is a common misconception that files have an EOF character at their end. This is *not* true!

[2]The reason for this requirement will be shown later.

If `char` is signed, `FF` is extended to `FFFFFFFF`. This does compare as equal and the loop ends. However, since the byte `FF` may have been read from the file, the loop could be ending prematurely.

The solution to this problem is to define the `ch` variable as an `int`, not a `char`. When this is done, no truncating or extension is done in line 2. Inside the loop, it is safe to truncate the value since `ch` *must* actually be a simple byte there.

2.1.3 Two's complement arithmetic

As was seen earlier, the `add` instruction performs addition and the `sub` instruction performs subtraction. Two of the bits in the FLAGS register that these instructions set are the *overflow* and *carry flag*. The overflow flag is set if the true result of the operation is too big to fit into the destination for signed arithmetic. The carry flag is set if there is a carry in the msb of an addition or a borrow in the msb of a subtraction. Thus, it can be used to detect overflow for unsigned arithmetic. The uses of the carry flag for signed arithmetic will be seen shortly. One of the great advantages of 2's complement is that the rules for addition and subtraction are exactly the same as for unsigned integers. Thus, `add` and `sub` may be used on signed or unsigned integers.

$$\begin{array}{rr} 002C & 44 \\ +\ \text{FFFF} & +\ (-1) \\ \hline 002B & 43 \end{array}$$

There is a carry generated, but it is not part of the answer.

There are two different multiply and divide instructions. First, to multiply use either the `MUL` or `IMUL` instruction. The `MUL` instruction is used to multiply unsigned numbers and `IMUL` is used to multiply signed integers. Why are two different instructions needed? The rules for multiplication are different for unsigned and 2's complement signed numbers. How so? Consider the multiplication of the byte FF with itself yielding a word result. Using unsigned multiplication this is 255 times 255 or 65025 (or FE01 in hex). Using signed multiplication this is -1 times -1 or 1 (or 0001 in hex).

There are several forms of the multiplication instructions. The oldest form looks like:

```
mul   source
```

The *source* is either a register or a memory reference. It can not be an immediate value. Exactly what multiplication is performed depends on the size of the source operand. If the operand is byte sized, it is multiplied by the byte in the AL register and the result is stored in the 16 bits of AX. If the source is 16-bit, it is multiplied by the word in AX and the 32-bit result

dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

Table 2.2: `imul` Instructions

is stored in DX:AX. If the source is 32-bit, it is multiplied by EAX and the 64-bit result is stored into EDX:EAX.

The `IMUL` instruction has the same formats as `MUL`, but also adds some other instruction formats. There are two and three operand formats:

```
imul   dest, source1
imul   dest, source1, source2
```

Table 2.2 shows the possible combinations.

The two division operators are `DIV` and `IDIV`. They perform unsigned and signed integer division respectively. The general format is:

```
div   source
```

If the source is 8-bit, then AX is divided by the operand. The quotient is stored in AL and the remainder in AH. If the source is 16-bit, then DX:AX is divided by the operand. The quotient is stored into AX and remainder into DX. If the source is 32-bit, then EDX:EAX is divided by the operand and the quotient is stored into EAX and the remainder into EDX. The `IDIV` instruction works the same way. There are no special `IDIV` instructions like the special `IMUL` ones. If the quotient is too big to fit into its register or the divisor is zero, the program is interrupted and terminates. A very common error is to forget to initialize DX or EDX before division.

The `NEG` instruction negates its single operand by computing its two's complement. Its operand may be any 8-bit, 16-bit, or 32-bit register or memory location.

2.1.4 Example program

```
1  title Math Example      (math.asm)
2
3
4  .model small ; one data segment, one code segment
5  .stack 100h ; reserves 256 bytes for the stack
6  .386 ; for 32 bits
7  .data ; start definition of variables
8    multiplicand dw 6 ;16 bis
9    multiplier db 5 ;8 bits
10   divisor dd 492 ;16 bits
11   dividend dd 5678 ;32 bits
12
13 .code ; start code portion of program
14 main proc ; start of the main procedure
15     mov  eax,@data ; load the address of the data segment into eax
16     mov  ds,eax ; load the address of the data segment into ds
17     ; the two previous instructions initalize the data segment
18     mov ax,multiplicand
19     mul multiplier ;do ax=ax*multiplier
20     ;ax is now 30
21
22     mov edx,0 ;clear dx
23     mov eax,dividend
24     div divisor ; edx:eax/divisor => eax remainder edx
25     ; now eax = 11 and edx = 266
26     ; since 5678/492 = 11 R 266
27
28
29     ; the following two instructions exit cleanly from the program
30     mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar
           to return 0; in C++)
31     int  21h ; exit
32 main endp ; end procedure
33
34 end main ; end program
```

2.1.5 Extended precision arithmetic

Assembly language also provides instructions that allow one to perform addition and subtraction of numbers larger than double words. These instructions use the carry flag. As stated above, both the `ADD` and `SUB` instructions modify the carry flag if a carry or borrow are generated, respectively. This information stored in the carry flag can be used to add or subtract large numbers by breaking up the operation into smaller double word (or smaller) pieces.

The ADC and SBB instructions use this information in the carry flag. The ADC instruction performs the following operation:

operand1 = *operand1* + carry flag + *operand2*

The SBB instruction performs:

operand1 = *operand1* - carry flag - *operand2*

How are these used? Consider the sum of 64-bit integers in EDX:EAX and EBX:ECX. The following code would store the sum in EDX:EAX:

```
1       add    eax, ecx      ; add lower 32-bits
2       adc    edx, ebx      ; add upper 32-bits and carry from
            previous sum
```

Subtraction is very similar. The following code subtracts EBX:ECX from EDX:EAX:

```
1       sub    eax, ecx      ; subtract lower 32-bits
2       sbb    edx, ebx      ; subtract upper 32-bits and borrow
```

For *really* large numbers, a loop could be used (see Section 2.2). For a sum loop, it would be convenient to use ADC instruction for every iteration (instead of all but the first iteration). This can be done by using the CLC (CLear Carry) instruction right before the loop starts to initialize the carry flag to 0. If the carry flag is 0, there is no difference between the ADD and ADC instructions. The same idea can be used for subtraction, too.

2.2 Control Structures

High level languages provide high level control structures (*e.g.*, the *if* and *while* statements) that control the thread of execution. Assembly language does not provide such complex control structures. It instead uses the infamous *goto* and used inappropriately can result in spaghetti code! However, it *is* possible to write structured assembly language programs. The basic procedure is to design the program logic using the familiar high level control structures and translate the design into the appropriate assembly language (much like a compiler would do).

2.2.1 Comparisons

Control structures decide what to do based on comparisons of data. In assembly, the result of a comparison is stored in the FLAGS register to be used later. The 80x86 provides the CMP instruction to perform comparisons. The FLAGS register is set based on the difference of the two operands of the CMP instruction, see Table 2.3 for a list of common flags. The operands

Flag	Commonly set when:	Abbreviation
Carry Flag	a carry occurs in addition or a borrow in subtraction.	CF
Zero Flag	the value after an operation is zero.	ZF
Overflow Flag	the sign of the result after addition or subtraction is different than the signs of the operands.	OF
Sign Flag	the result of an operation is negative.	SF
Parity Flag	the parity of the lower 8 bits of the result of an operation.	PF

Table 2.3: Common Flags

are subtracted and the FLAGS are set based on the result, but the result is *not* stored anywhere. If you need the result use the SUB instead of the `CMP` instruction.

For unsigned integers, there are two flags (bits in the FLAGS register) that are important: the zero (ZF) and carry (CF) flags. The zero flag is set (1) if the resulting difference would be zero. The carry flag is used as a borrow flag for subtraction. Consider a comparison like:

```
cmp    vleft, vright
```

The difference of `vleft - vright` is computed and the flags are set accordingly. If the difference of the of `CMP` is zero, `vleft = vright`, then ZF is set (*i.e.* 1) and the CF is unset (*i.e.* 0). If `vleft > vright`, then ZF is unset and CF is unset (no borrow). If `vleft < vright`, then ZF is unset and CF is set (borrow).

For signed integers, there are three flags that are important: the zero (ZF) flag, the overflow (OF) flag and the sign (SF) flag. The overflow flag is set if the result of an operation overflows (or underflows). The sign flag is set if the result of an operation is negative. If `vleft = vright`, the ZF is set (just as for unsigned integers). If `vleft > vright`, ZF is unset and SF = OF. If `vleft < vright`, ZF is unset and SF $\neq$ OF.

Why does SF = OF if `vleft > vright`*? If there is no overflow, then the difference will have the correct value and must be non-negative. Thus, SF = OF = 0. However, if there is an overflow, the difference will not have the correct value (and in fact will be negative). Thus, SF = OF = 1.*

Do not forget that other instructions can also change the FLAGS register, not just `CMP`.

2.2.2 Branch instructions

Branch instructions can transfer execution to arbitrary points of a program. In other words, they act like a *goto*. There are two types of branches:

unconditional and conditional. An unconditional branch is just like a goto, it always makes the branch. A conditional branch may or may not make the branch depending on the flags in the FLAGS register. If a conditional branch does not make the branch, control passes to the next instruction.

The `JMP` (short for *jump*) instruction makes unconditional branches. Its single argument is usually a *code label* to the instruction to branch to. The assembler or linker will replace the label with correct address of the instruction. This is another one of the tedious operations that the assembler does to make the programmer's life easier. It is important to realize that the statement immediately after the `JMP` instruction will never be executed unless another instruction branches to it!

There are several variations of the jump instruction:

SHORT This jump is very limited in range. It can only move up or down 128 bytes in memory. The advantage of this type is that it uses less memory than the others. It uses a single signed byte to store the *displacement* of the jump. The displacement is how many bytes to move ahead or behind. (The displacement is added to EIP). To specify a short jump, use the `SHORT` keyword immediately before the label in the `JMP` instruction.

NEAR This jump is the default type for both unconditional and conditional branches, it can be used to jump to any location in a segment. Actually, the 80386 supports two types of near jumps. One uses two bytes for the displacement. This allows one to move up or down roughly 32,000 bytes. The other type uses four bytes for the displacement, which of course allows one to move to any location in the code segment. The four byte type is the default in 386 protected mode. The two byte type can be specified by putting the `WORD` keyword before the label in the `JMP` instruction.

FAR This jump allows control to move to another code segment. This is a very rare thing to do in 386 protected mode.

Valid code labels follow the same rules as data labels. Code labels are defined by placing them in the code segment in front of the statement they label. A colon is placed at the end of the label at its point of definition. The colon is *not* part of the name.

There are many different conditional branch instructions. They also take a code label as their single operand. The simplest ones just look at a single flag in the FLAGS register to determine whether to branch or not. See Table 2.4 for a list of these instructions. (PF is the *parity flag* which indicates the odd or evenness of the number of bits set in the lower 8-bits of the result.)

JZ	branches only if ZF is set
JNZ	branches only if ZF is unset
JO	branches only if OF is set
JNO	branches only if OF is unset
JS	branches only if SF is set
JNS	branches only if SF is unset
JC	branches only if CF is set
JNC	branches only if CF is unset
JP	branches only if PF is set
JNP	branches only if PF is unset

Table 2.4: Simple Conditional Branches

The following pseudo-code:

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

could be written in assembly as:

```
1       cmp    eax, 0       ; set flags (ZF set if eax - 0 = 0)
2       jz     thenblock    ; if ZF is set branch to thenblock
3       mov    ebx, 2       ; ELSE part of IF
4       jmp    next         ; jump over THEN part of IF
5   thenblock:
6       mov    ebx, 1       ; THEN part of IF
7   next:
```

Other comparisons are not so easy using the conditional branches in Table 2.4. To illustrate, consider the following pseudo-code:

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

If EAX is greater than or equal to five, the ZF may be set or unset and SF will equal OF. Here is assembly code that tests for these conditions (assuming that EAX is signed):

```
1       cmp    eax, 5
2       js     signon           ; goto signon if SF = 1
3       jo     elseblock        ; goto elseblock if OF = 1 and SF = 0
4       jmp    thenblock        ; goto thenblock if SF = 0 and OF = 0
5   signon:
```

Signed		Unsigned	
JE	branches if `vleft = vright`	JE	branches if `vleft = vright`
JNE	branches if `vleft` $\neq$ `vright`	JNE	branches if `vleft` $\neq$ `vright`
JL, JNGE	branches if `vleft < vright`	JB, JNAE	branches if `vleft < vright`
JLE, JNG	branches if `vleft` $\leq$ `vright`	JBE, JNA	branches if `vleft` $\leq$ `vright`
JG, JNLE	branches if `vleft > vright`	JA, JNBE	branches if `vleft > vright`
JGE, JNL	branches if `vleft` $\geq$ `vright`	JAE, JNB	branches if `vleft` $\geq$ `vright`

Table 2.5: Signed and Unsigned Comparison Instructions

```
6       jo      thenblock       ; goto thenblock if SF = 1 and OF = 1
7   elseblock:
8       mov     ebx, 2
9       jmp     next
10  thenblock:
11      mov     ebx, 1
12  next:
```

The above code is very awkward. Fortunately, the 80x86 provides additional branch instructions to make these type of tests *much* easier. There are signed and unsigned versions of each. Table 2.5 shows these instructions. The equal and not equal branches (JE and JNE) are the same for both signed and unsigned integers. (In fact, JE and JNE are really identical to JZ and JNZ, respectively.) Each of the other branch instructions have two synonyms. For example, look at JL (jump less than) and JNGE (jump not greater than or equal to). These are the same instruction because:

$$x < y \Longrightarrow \textbf{not}(x \geq y)$$

The unsigned branches use A for *above* and B for *below* instead of L and G.

Using these new branch instructions, the pseudo-code above can be translated to assembly much easier.

```
1       cmp     eax, 5
2       jge     thenblock
3       mov     ebx, 2
4       jmp     next
5   thenblock:
6       mov     ebx, 1
7   next:
```

2.2.3 The loop instructions

The 80x86 provides several instructions designed to implement *for*-like loops. Each of these instructions takes a code label as its single operand.

LOOP Decrements ECX, if ECX $\neq 0$, branches to label

LOOPE, LOOPZ Decrements ECX (FLAGS register is not modified), if ECX $\neq 0$ and ZF $= 1$, branches

LOOPNE, LOOPNZ Decrements ECX (FLAGS unchanged), if ECX $\neq 0$ and ZF $= 0$, branches

The last two loop instructions are useful for sequential search loops. The following pseudo-code:

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

could be translated into assembly as:

```
1       mov   eax, 0        ; eax is sum
2       mov   ecx, 10       ; ecx is i
3   loop_start:
4       add   eax, ecx
5       loop  loop_start
```

2.3 Translating Standard Control Structures

This section looks at how the standard control structures of high level languages can be implemented in assembly language.

2.3.1 If statements

The following pseudo-code:

```
if ( condition )
  then_block;
else
  else_block;
```

could be implemented as:

```
1     ; code to set FLAGS
2     jxx   else_block ; select xx so that branches if condition false
3     ; code for then block
4     jmp   end_if
5   else_block:
6     ; code for else block
7   end_if:
```

If there is no else, then the `else_block` branch can be replaced by a branch to `end_if`.

```
    ; code to set FLAGS
    jxx   end_if ; select xx so that branches if condition false
    ; code for then block
end_if:
```

2.3.2 While loops

The *while* loop is a top tested loop:

```
while( condition ) {
  body of loop;
}
```

This could be translated into:

```
while:
    ; code to set FLAGS based on condition
    jxx   endwhile      ; select xx so that branches if false
    ; body of loop
    jmp   while
endwhile:
```

2.3.3 Do while loops

The *do while* loop is a bottom tested loop:

```
do {
  body of loop;
} while( condition );
```

This could be translated into:

```
do:
    ; body of loop
    ; code to set FLAGS based on condition
    jxx   do         ; select xx so that branches if true
```

2.4 Example: Finding Prime Numbers

This section looks at a program that finds prime numbers. Recall that prime numbers are evenly divisible by only 1 and themselves. There is no formula for doing this. The basic method this program uses is to find the

factors of a number below a given limit. If no factor can be found, it is prime. Here is the basic algorithm written in C:

```c
#include <stdio.h>

int main(){
  unsigned int guess; //current guess for prime
  unsigned int factor; // possible factor of guess
  unsigned int limit; // find primes up to this value

  limit=30; //find the primes up to 30
  guess = 1; //initial guess
  while (guess < limit) { // outer while loop
      guess++;
      factor=1;
      do { //inner while loop
        factor++;
        if (factor==guess){ //isPrime
           printf("found a prime\n");
           break; //jump to outer while loop

        }
     } while(guess % factor != 0);
  }
}
```

Here's the assembly version:

```
title Prime Finder (prime.asm)

.model small ; one data segment, one code segment
.stack 100h ; reserves 256 bytes for the stack
.386 ; for 32 bits
.data ; start definition of variables
    limit dd 30;
    output db "found a prime",0dh,0ah,'$'
; your variables go here
.code ; start code portion of program
main proc ; start of the main procedure
    mov  eax,@data ; load the address of the data segment into ax
    mov  ds,eax ; load the address of the data segment into ds
    ; the two previous instructions initalize the data segment
    mov eax,0
    mov ebx,1 ;ebx is guess
    mov edx,0
    mov edi,limit
outerWhileLoop:
    inc ebx ; increment guess
```

```
21      cmp ebx,edi
22      jg exit
23      mov ecx,1 ;ecx is factor
24  innerWhileLoop:
25      inc ecx ; increment factor
26      cmp ecx,ebx
27      jge isPrime
28      mov eax,ebx ;put guess in eax
29      mov edx,0 ;zero out edx
30      div ecx ;divide guess and factor and put the remainder in edx
31      cmp edx,0
32      jne innerWhileLoop
33      jmp outerWhileLoop
34  isPrime:
35      ;the prime value is in ebx
36      mov  ah,9 ;setting for outputting a string
37      mov  dx,offset output ;point dx to the string
38      int  21h ;output the string
39      jmp outerWhileLoop
40
41
42  exit:
43
44      ;the following two instructions exit cleanly from the program
45      mov  eax,4C00h ; exit (ah) with code 0 (al) (similar to return
            0; in C++)
46      int  21h ; exit
47  main endp ; end procedure
48
49  end main ; end program
```

Chapter 3

Bit Operations

3.1 Shift Operations

Assembly language allows the programmer to manipulate the individual bits of data. One common bit operation is called a *shift*. A shift operation moves the position of the bits of some data. Shifts can be either toward the left (*i.e.* toward the most significant bits) or toward the right (the least significant bits).

3.1.1 Logical shifts

A logical shift is the simplest type of shift. It shifts in a very straightforward manner. Figure 3.1 shows an example of a shifted single byte number.

Operation	**Bit Locations After Op**							
Original	1	1	1	0	1	0	1	0
Left shifted	1	1	0	1	0	1	0	0
Right shifted	0	1	1	1	0	1	0	1

Figure 3.1: Logical shifts

Note that new, incoming bits are always zero. The `SHL` and `SHR` instructions are used to perform logical left and right shifts respectively. These instructions allow one to shift by any number of positions. The number of positions to shift can either be a constant or can be stored in the `CL` register. The last bit shifted out of the data is stored in the carry flag. Here are some code examples:

```
1       mov    ax, 0C123H
2       shl    ax, 1          ; shift 1 bit to left, ax = 8246H, CF = 1
3       shr    ax, 1          ; shift 1 bit to right, ax = 4123H, CF = 0
```

```
4       shr    ax, 1          ; shift 1 bit to right, ax = 2091H, CF = 1
5       mov    ax, 0C123H
6       shl    ax, 2          ; shift 2 bits to left, ax = 048CH, CF = 1
7       mov    cl, 3
8       shr    ax, cl         ; shift 3 bits to right, ax = 0091H, CF =
            1
```

3.1.2 Use of shifts

Fast multiplication and division are the most common uses of a shift operations. Recall that in the decimal system, multiplication and division by a power of ten are simple, just shift digits. The same is true for powers of two in binary. For example, to double the binary number 1011_2 (or 11 in decimal), shift once to the left to get 10110_2 (or 22). The quotient of a division by a power of two is the result of a right shift. To divide by just 2, use a single right shift; to divide by 4 (2^2), shift right 2 places; to divide by 8 (2^3), shift 3 places to the right, *etc.* Shift instructions are very basic and are *much* faster than the corresponding `MUL` and `DIV` instructions!

Actually, logical shifts can be used to multiply and divide unsigned values. They do not work in general for signed values. Consider the 2-byte value FFFF (signed -1). If it is logically right shifted once, the result is 7FFF which is $+32,767$! Another type of shift can be used for signed values.

3.1.3 Arithmetic shifts

These shifts are designed to allow signed numbers to be quickly multiplied and divided by powers of 2. They insure that the sign bit is treated correctly.

SAL Shift Arithmetic Left - This instruction is just a synonym for `SHL`. It is assembled into the exactly the same machine code as `SHL`. As long as the sign bit is not changed by the shift, the result will be correct.

SAR Shift Arithmetic Right - This is a new instruction that does not shift the sign bit (*i.e.* the msb) of its operand. The other bits are shifted as normal except that the new bits that enter from the left are copies of the sign bit (that is, if the sign bit is 1, the new bits are also 1). Thus, if a byte is shifted with this instruction, only the lower 7 bits are shifted. As for the other shifts, the last bit shifted out is stored in the carry flag.

```
1       mov    ax, 0C123H
2       sal    ax, 1          ; ax = 8246H, CF = 1
```

```
3       sal    ax, 1          ; ax = 048CH, CF = 1
4       sar    ax, 2          ; ax = 0123H, CF = 0
```

3.1.4 Rotate shifts

The rotate shift instructions work like logical shifts except that bits lost off one end of the data are shifted in on the other side. Thus, the data is treated as if it is a circular structure. The two simplest rotate instructions are `ROL` and `ROR` which make left and right rotations, respectively. Just as for the other shifts, these shifts leave the a copy of the last bit shifted around in the carry flag.

```
1       mov    ax, 0C123H
2       rol    ax, 1          ; ax = 8247H, CF = 1
3       rol    ax, 1          ; ax = 048FH, CF = 1
4       rol    ax, 1          ; ax = 091EH, CF = 0
5       ror    ax, 2          ; ax = 8247H, CF = 1
6       ror    ax, 1          ; ax = C123H, CF = 1
```

There are two additional rotate instructions that shift the bits in the data and the carry flag named `RCL` and `RCR`. For example, if the `AX` register is rotated with these instructions, the 17-bits made up of `AX` and the carry flag are rotated.

```
1       mov    ax, 0C123H
2       clc                   ; clear the carry flag (CF = 0)
3       rcl    ax, 1          ; ax = 8246H, CF = 1
4       rcl    ax, 1          ; ax = 048DH, CF = 1
5       rcl    ax, 1          ; ax = 091BH, CF = 0
6       rcr    ax, 2          ; ax = 8246H, CF = 1
7       rcr    ax, 1          ; ax = C123H, CF = 0
```

3.1.5 Simple application

Here is a code snippet that counts the number of bits that are "on" (*i.e.* 1) in the EAX register.

```
1       mov    bl, 0          ; bl will contain the count of ON bits
2       mov    ecx, 32        ; ecx is the loop counter
3   count_loop:
4       shl    eax, 1         ; shift bit into carry flag
5       jnc    skip_inc       ; if CF == 0, goto skip_inc
6       inc    bl
7   skip_inc:
8       loop   count_loop
```

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: The AND operation

	1	0	1	0	1	0	1	0
AND	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

Figure 3.2: ANDing a byte

The above code destroys the original value of `EAX` (`EAX` is zero at the end of the loop). If one wished to retain the value of `EAX`, line 4 could be replaced with `rol eax, 1`.

3.2 Boolean Bitwise Operations

There are four common boolean operators: *AND*, *OR*, *XOR* and *NOT*. A *truth table* shows the result of each operation for each possible value of its operands.

3.2.1 The *AND* operation

The result of the *AND* of two bits is only 1 if both bits are 1, else the result is 0 as the truth table in Table 3.1 shows.

Processors support these operations as instructions that act independently on all the bits of data in parallel. For example, if the contents of `AL` and `BL` are *AND*ed together, the basic *AND* operation is applied to each of the 8 pairs of corresponding bits in the two registers as Figure 3.2 shows. Below is a code example:

```
1       mov    ax, 0C123H
2       and    ax, 82F6H        ; ax = 8022H
```

3.2.2 The *OR* operation

The inclusive *OR* of 2 bits is 0 only if both bits are 0, else the result is 1 as the truth table in Table 3.2 shows. Below is a code example:

```
1       mov    ax, 0C123H
2       or     ax, 0E831H        ; ax = E933H
```

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Table 3.2: The OR operation

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.3: The XOR operation

3.2.3 The *XOR* operation

The exclusive *OR* of 2 bits is 0 if and only if both bits are equal, else the result is 1 as the truth table in Table 3.3 shows. Below is a code example:

```
1       mov    ax, 0C123H
2       xor    ax, 0E831H          ; ax = 2912H
```

3.2.4 The *NOT* operation

The *NOT* operation is a *unary* operation (*i.e.* it acts on one operand, not two like *binary* operations such as *AND*). The *NOT* of a bit is the opposite value of the bit as the truth table in Table 3.4 shows. Below is a code example:

```
1       mov    ax, 0C123H
2       not    ax                  ; ax = 3EDCH
```

Note that the *NOT* finds the one's complement. Unlike the other bitwise operations, the `NOT` instruction does not change any of the bits in the `FLAGS` register.

3.2.5 The `TEST` instruction

The `TEST` instruction performs an *AND* operation, but does not store the result. It only sets the `FLAGS` register based on what the result would be (much like how the `CMP` instruction performs a subtraction but only sets `FLAGS`). For example, if the result would be zero, `ZF` would be set.

X	NOT X
0	1
1	0

Table 3.4: The NOT operation

Turn on bit i	*OR* the number with 2^i (which is the binary number with just bit i on)
Turn off bit i	*AND* the number with the binary number with only bit i off. This operand is often called a *mask*
Complement bit i	*XOR* the number with 2^i

Table 3.5: Uses of boolean operations

3.3 Uses of Bit Operations

Bit operations are very useful for manipulating individual bits of data without modifying the other bits. Table 3.5 shows three common uses of these operations. Below is some example code, implementing these ideas.

```
1       mov    ax, 0C123H
2       or     ax, 8          ; turn on bit 3, ax = C12BH
3       and    ax, 0FFDFH     ; turn off bit 5, ax = C10BH
4       xor    ax, 8000H      ; invert bit 15, ax = 410BH
5       or     ax, 0F00H      ; turn on nibble, ax = 4F0BH
6       and    ax, 0FFF0H     ; turn off nibble, ax = 4F00H
7       xor    ax, 0F00FH     ; invert nibbles, ax = BF0FH
8       xor    ax, 0FFFFH     ; 1's complement, ax = 40F0H
```

The *AND* operation can also be used to find the remainder of a division by a power of two. To find the remainder of a division by 2^i, *AND* the number with a mask equal to $2^i - 1$. This mask will contain ones from bit 0 up to bit $i - 1$. It is just these bits that contain the remainder. The result of the *AND* will keep these bits and zero out the others. Next is a snippet of code that finds the quotient and remainder of the division of 100 by 16.

```
1       mov    eax, 100       ; 100 = 64H
2       mov    ebx, 0000000FH ; mask = 16 - 1 = 15 or F
3       and    ebx, eax       ; ebx = remainder = 4
4       shr    eax, 4         ; eax = quotient of eax/2^4 = 6
```

Using the `CL` register it is possible to modify arbitrary bits of data. Next is an example that sets (turns on) an arbitrary bit in `EAX`. The number of the bit to set is stored in `BH`.

```
1       mov    bl, 0          ; bl will contain the count of ON bits
2       mov    ecx, 32        ; ecx is the loop counter
3   count_loop:
4       shl    eax, 1         ; shift bit into carry flag
5       adc    bl, 0          ; add just the carry flag to bl
6       loop   count_loop
```

Figure 3.3: Counting bits with `ADC`

```
1       mov    cl, bh         ; first build the number to OR with
2       mov    ebx, 1
3       shl    ebx, cl        ; shift left cl times
4       or     eax, ebx       ; turn on bit
```

Turning a bit off is just a little harder.

```
1       mov    cl, bh         ; first build the number to AND with
2       mov    ebx, 1
3       shl    ebx, cl        ; shift left cl times
4       not    ebx            ; invert bits
5       and    eax, ebx       ; turn off bit
```

Code to complement an arbitrary bit is left as an exercise for the reader.

It is not uncommon to see the following puzzling instruction in a 80x86 program:

```
        xor    eax, eax        ; eax = 0
```

A number *XOR*'ed with itself always results in zero. This instruction is used because its machine code is smaller than the corresponding `MOV` instruction.

3.3.1 Avoiding Conditional Branches

Modern processors use very sophisticated techniques to execute code as quickly as possible. One common technique is known as *speculative execution*. This technique uses the parallel processing capabilities of the CPU to execute multiple instructions at once. Conditional branches present a problem with this idea. The processor, in general, does not know whether the branch will be taken or not. If it is taken, a different set of instructions will be executed than if it is not taken. Processors try to predict whether the branch will be taken. If the prediciton is wrong, the processor has wasted its time executing the wrong code.

One way to avoid this problem is to avoid using conditional branches when possible. The sample code in 3.1.5 provides a simple example of where

one could do this. In the previous example, the "on" bits of the EAX register are counted. It uses a branch to skip the `INC` instruction. Figure 3.3 shows how the branch can be removed by using the `ADC` instruction to add the carry flag directly.

The `SET`*xx* instructions provide a way to remove branches in certain cases. These instructions set the value of a byte register or memory location to zero or one based on the state of the FLAGS register. The characters after `SET` are the same characters used for conditional branches. If the corresponding condition of the `SET`*xx* is true, the result stored is a one, if false a zero is stored. For example,

```
setz  al       ; AL = 1 if Z flag is set, else 0
```

Using these instructions, one can develop some clever techniques that calculate values without branches.

For example, consider the problem of finding the maximum of two values. The standard approach to solving this problem would be to use a `CMP` and use a conditional branch to act on which value was larger. The example program below shows how the maximum can be found without any branches.

```
1  title Tricky Max     (max.asm)
2  .model small ;
3  .stack 100h ; reserves 256 bytes for the stack
4  .386 ; for 32 bits
5  .data ;start data segment
6  firstNumber dd 056h
7  secondNumber dd 0afh
8
9  .code ; start code portion of program
10 main proc ; start of the main procedure
11     mov  eax,@data ; load the address of the data segment into ax
12     mov  ds,eax ; load the address of the data segment into ds
13     ; the two previous instructions initalize the data segment
14     mov eax,firstNumber
15     xor ebx,ebx ; ebx = 0
16     cmp eax,secondNumber ; compare second and first number
17     setg bl     ; ebx = (input2 > input1) ?      1 : 0
18     neg ebx     ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
19     mov ecx,ebx ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
20     and ecx,eax ; ecx = (input2 > input1) ?  input2 : 0
21     not ebx     ; ebx = (input2 > input1) ?      0 : 0xFFFFFFFF
22     and ebx,secondNumber ; ebx = (input2 > input1) ?     0 : input1
23     or  ecx,ebx ; ecx = (input2 > input1) ?  input2 : input1
24     ; ecx now has the greater value
25
26     mov  eax,4C00h ; exit (ah) with code 0 (al)
27     int  21h ; exit
```

```
28 main endp ; end procedure
29
30 end main ; end program
```

The trick is to create a bit mask that can be used to select the correct value for the maximum. The `SETG` instruction in line 17 sets BL to 1 if the second input is the maximum or 0 otherwise. This is not quite the bit mask desired. To create the required bit mask, line 18 uses the `NEG` instruction on the entire EBX register. (Note that EBX was zeroed out earlier.) If EBX is 0, this does nothing; however, if EBX is 1, the result is the two's complement representation of -1 or 0xFFFFFFFF. This is just the bit mask required. The remaining code uses this bit mask to select the correct input as the maximum.

An alternative trick is to use the `DEC` statement. In the above code, if the `NEG` is replaced with a `DEC`, again the result will either be 0 or 0xFFFFFFFF. However, the values are reversed than when using the `NEG` instruction.

3.4 Manipulating Bits in C

3.4.1 The bitwise operators of C

Unlike some high-level languages, C does provide operators for bitwise operations. The *AND* operation is represented by the binary `&` operator[1]. The *OR* operation is represented by the binary `|` operator. The *XOR* operation is represented by the binary `^` operator. And the *NOT* operation is represented by the unary `~` operator.

The shift operations are performed by C's `<<` and `>>` binary operators. The `<<` operator performs left shifts and the `>>` operator performs right shifts. These operators take two operands. The left operand is the value to shift and the right operand is the number of bits to shift by. If the value to shift is an unsigned type, a logical shift is made. If the value is a signed type (like `int`), then an arithmetic shift is used. Below is some example C code using these operators:

```
1 short int s;        /* assume that short int is 16-bit */
2 short unsigned u;
3 s = -1;             /* s = 0xFFFF (2's complement) */
4 u = 100;            /* u = 0x0064 */
5 u = u | 0x0100;     /* u = 0x0164 */
6 s = s & 0xFFF0;     /* s = 0xFFF0 */
7 s = s ^ u;          /* s = 0xFE94 */
8 u = u << 3;         /* u = 0x0B20 (logical shift) */
9 s = s >> 2;         /* s = 0xFFA5 (arithmetic shift) */
```

[1] This operator is different from the binary `&&` and unary `&` operators!

Macro	Meaning
S_IRUSR	user can read
S_IWUSR	user can write
S_IXUSR	user can execute
S_IRGRP	group can read
S_IWGRP	group can write
S_IXGRP	group can execute
S_IROTH	others can read
S_IWOTH	others can write
S_IXOTH	others can execute

Table 3.6: POSIX File Permission Macros

3.4.2 Using bitwise operators in C

The bitwise operators are used in C for the same purposes as they are used in assembly language. They allow one to manipulate individual bits of data and can be used for fast multiplication and division. In fact, a smart C compiler will use a shift for a multiplication like, `x *= 2`, automatically.

Many operating system API[2]'s (such as *POSIX*[3] and Win32) contain functions which use operands that have data encoded as bits. For example, POSIX systems maintain file permissions for three different types of users: *user* (a better name would be *owner*), *group* and *others*. Each type of user can be granted permission to read, write and/or execute a file. To change the permissions of a file requires the C programmer to manipulate individual bits. POSIX defines several macros to help (see Table 3.6). The `chmod` function can be used to set the permissions of file. This function takes two parameters, a string with the name of the file to act on and an integer[4] with the appropriate bits set for the desired permissions. For example, the code below sets the permissions to allow the owner of the file to read and write to it, users in the group to read the file and others have no access.

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

The POSIX `stat` function can be used to find out the current permission bits for the file. Used with the `chmod` function, it is possible to modify some of the permissions without changing others. Here is an example that removes write access to others and adds read access to the owner of the file. The other permissions are not altered.

[2]Application Programming Interface

[3]stands for Portable Operating System Interface for Computer Environments. A standard developed by the IEEE based on UNIX.

[4]Actually a parameter of type `mode_t` which is a typedef to an integral type.

```
1 struct stat file_stats; /* struct used by stat() */
2 stat("foo", &file_stats); /* read file info.
3                             file_stats.st_mode holds permission bits
                                 */
4 chmod("foo", (file_stats.st_mode & ~S_IWOTH) | S_IRUSR);
```

3.5 Big and Little Endian Representations

Chapter 1 introduced the concept of big and little endian representations of multibyte data. However, the author has found that this subject confuses many people. This section covers the topic in more detail.

The reader will recall that endianness refers to the order that the individual bytes (*not* bits) of a multibyte data element is stored in memory. Big endian is the most straightforward method. It stores the most significant byte first, then the next significant byte and so on. In other words the *big* bits are stored first. Little endian stores the bytes in the opposite order (least significant first). The x86 family of processors use little endian representation.

As an example, consider the double word representing 12345678_{16}. In big endian representation, the bytes would be stored as 12 34 56 78. In little endian represenation, the bytes would be stored as 78 56 34 12.

The reader is probably asking himself right now, why any sane chip designer would use little endian representation? Were the engineers at Intel sadists for inflicting this confusing representations on multitudes of programmers? It would seem that the CPU has to do extra work to store the bytes backward in memory like this (and to unreverse them when read back in to memory). The answer is that the CPU does not do any extra work to write and read memory using little endian format. One has to realize that the CPU is composed of many electronic circuits that simply work on bit values. The bits (and bytes) are not in any necessary order in the CPU.

Consider the 2-byte `AX` register. It can be decomposed into the single byte registers: `AH` and `AL`. There are circuits in the CPU that maintain the values of `AH` and `AL`. Circuits are not in any order in a CPU. That is, the circuits for `AH` are not before or after the circuits for `AL`. A `mov` instruction that copies the value of `AX` to memory copies the value of `AL` then `AH`. This is not any harder for the CPU to do than storing `AH` first.

The same argument applies to the individual bits in a byte. They are not really in any order in the circuits of the CPU (or memory for that matter). However, since individual bits can not be addressed in the CPU or memory, there is no way to know (or care about) what order they seem to be kept internally by the CPU.

The C code in Figure 3.4 shows how the endianness of a CPU can be

```c
unsigned short word = 0x1234; /* assumes sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
  printf("Big Endian Machine\n");
else
  printf("Little Endian Machine\n");
```

Figure 3.4: How to Determine Endianness

determined. The `p` pointer treats the `word` variable as a two element character array. Thus, `p[0]` evaluates to the first byte of `word` in memory which depends on the endianness of the CPU.

3.5.1 When to Care About Little and Big Endian

With the advent of multi-byte character sets, like UNICODE, endianness is important for even text data. UNICODE supports either endianness and has a mechanism for specifying which endianness is being used to represent the data.

For typical programming, the endianness of the CPU is not significant. The most common time that it is important is when binary data is transferred between different computer systems. This is usually either using some type of physical data media (such as a disk) or a network. Since ASCII data is single byte, endianness is not an issue for it.

All internal TCP/IP headers store integers in big endian format (called *network byte order*). TCP/IP libraries provide C functions for dealing with endianness issues in a portable way. For example, the `htonl()` function converts a double word (or long integer) from *host* to *network* format. The `ntohl()` function performs the opposite transformation.[5] For a big endian system, the two functions just return their input unchanged. This allows one to write network programs that will compile and run correctly on any system irrespective of its endianness. For more information, about endianness and network programming see W. Richard Steven's excellent book, *UNIX Network Programming*.

Figure 3.5 shows a C function that inverts the endianness of a double word. The 486 processor introduced a new machine instruction named `BSWAP` that reverses the bytes of any 32-bit register. For example,

```
bswap   edx          ; swap bytes of edx
```

The instruction can not be used on 16-bit registers. However, the `XCHG` instruction can be used to swap the bytes of the 16-bit registers that can be decomposed into 8-bit registers. For example:

[5] Actually, reversing the endianness of an integer simply reverses the bytes; thus, converting from big to little or little to big is the same operation. So both of these functions do the same thing.

```
unsigned invert_endian( unsigned x )
{
  unsigned invert;
  const unsigned char * xp = (const unsigned char *) &x;
  unsigned char * ip = (unsigned char *) & invert;

  ip[0] = xp[3]; /* reverse the individual bytes */
  ip[1] = xp[2];
  ip[2] = xp[1];
  ip[3] = xp[0];

  return invert; /* return the bytes reversed */
}
```

Figure 3.5: invert_endian Function

```
int count_bits( unsigned int data )
{
  int cnt = 0;

  while( data != 0 ) {
    data = data & (data - 1);
    cnt++;
  }
  return cnt;
}
```

Figure 3.6: Bit Counting: Method One

```
xchg    ah,al          ; swap bytes of ax
```

3.6 Counting Bits

Earlier a straightforward technique was given for counting the number of bits that are "on" in a double word. This section looks at other less direct methods of doing this as an exercise using the bit operations discussed in this chapter.

3.6.1 Method one

The first method is very simple, but not obvious. Figure 3.6 shows the code.

```
1  static unsigned char byte_bit_count[256]; /* lookup table */
2
3  void initialize_count_bits()
4  {
5    int cnt, i, data;
6
7    for( i = 0; i < 256; i++ ) {
8      cnt = 0;
9      data = i;
10     while( data != 0 ) { /* method one */
11       data = data & (data - 1);
12       cnt++;
13     }
14     byte_bit_count[i] = cnt;
15   }
16 }
17
18 int count_bits( unsigned int data )
19 {
20   const unsigned char * byte = ( unsigned char *) & data;
21
22   return byte_bit_count[byte[0]] + byte_bit_count[byte[1]] +
23          byte_bit_count[byte[2]] + byte_bit_count[byte[3]];
24 }
```

Figure 3.7: Method Two

How does this method work? In every iteration of the loop, one bit is turned off in `data`. When all the bits are off (*i.e.* when `data` is zero), the loop stops. The number of iterations required to make `data` zero is equal to the number of bits in the original value of `data`.

Line 6 is where a bit of `data` is turned off. How does this work? Consider the general form of the binary representation of `data` and the rightmost 1 in this representation. By definition, every bit after this 1 must be zero. Now, what will be the binary representation of `data - 1`? The bits to the left of the rightmost 1 will be the same as for `data`, but at the point of the rightmost 1 the bits will be the complement of the original bits of `data`. For example:

`data` = xxxxx10000
`data - 1` = xxxxx01111

where the x's are the same for both numbers. When `data` is *AND*'ed with `data - 1`, the result will zero the rightmost 1 in `data` and leave all the other bits unchanged.

3.6.2 Method two

A lookup table can also be used to count the bits of an arbitrary double word. The straightforward approach would be to precompute the number of bits for each double word and store this in an array. However, there are two related problems with this approach. There are roughly *4 billion* double word values! This means that the array will be very big and that initializing it will also be very time consuming. (In fact, unless one is going to actually use the array more than 4 billion times, more time will be taken to initialize the array than it would require to just compute the bit counts using method one!)

A more realistic method would precompute the bit counts for all possible byte values and store these into an array. Then the double word can be split up into four byte values. The bit counts of these four byte values are looked up from the array and sumed to find the bit count of the original double word. Figure 3.7 shows the to code implement this approach.

The `initialize_count_bits` function must be called before the first call to the `count_bits` function. This function initializes the global `byte_bit_count` array. The `count_bits` function looks at the `data` variable not as a double word, but as an array of four bytes. The `dword` pointer acts as a pointer to this four byte array. Thus, `dword[0]` is one of the bytes in `data` (either the least significant or the most significant byte depending on if the hardware is little or big endian, respectively.) Of course, one could use a construction like:

```
(data >> 24) & 0x000000FF
```

to find the most significant byte value and similar ones for the other bytes; however, these constructions will be slower than an array reference.

One last point, a `for` loop could easily be used to compute the sum on lines 22 and 23. But, a `for` loop would include the overhead of initializing a loop index, comparing the index after each iteration and incrementing the index. Computing the sum as the explicit sum of four values will be faster. In fact, a smart compiler would convert the `for` loop version to the explicit sum. This process of reducing or eliminating loop iterations is a compiler optimization technique known as *loop unrolling*.

3.6.3 Method three

There is yet another clever method of counting the bits that are on in data. This method literally adds the one's and zero's of the data together. This sum must equal the number of one's in the data. For example, consider counting the one's in a byte stored in a variable named `data`. The first step is to perform the following operation:

```
1  int count_bits(unsigned int x )
2  {
3    static unsigned int mask[] = { 0x55555555,
4                                   0x33333333,
5                                   0x0F0F0F0F,
6                                   0x00FF00FF,
7                                   0x0000FFFF };
8    int i;
9    int shift;  /* number of positions to shift to right */
10
11   for( i=0, shift=1; i < 5; i++, shift *= 2 )
12     x = (x & mask[i]) + ( (x >> shift) & mask[i] );
13   return x;
14 }
```

Figure 3.8: Method 3

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

What does this do? The hex constant `0x55` is 01010101 in binary. In the first operand of the addition, `data` is *AND*'ed with this, bits at the odd bit positions are pulled out. The second operand `((data >> 1) & 0x55)` first moves all the bits at the even positions to an odd position and uses the same mask to pull out these same bits. Now, the first operand contains the odd bits and the second operand the even bits of `data`. When these two operands are added together, the even and odd bits of `data` are added together. For example, if `data` is 10110011_2, then:

$$\begin{array}{r} \texttt{data \& } 01010101_2 \\ + \texttt{ (data >> 1) \& } 01010101_2 \\ \hline \end{array} \quad \text{or} \quad \begin{array}{r|cccc} & 00 & 01 & 00 & 01 \\ + & 01 & 01 & 00 & 01 \\ \hline & 01 & 10 & 00 & 10 \end{array}$$

The addition on the right shows the actual bits added together. The bits of the byte are divided into four 2-bit fields to show that actually there are four independent additions being performed. Since the most these sums can be is two, there is no possibility that the sum will overflow its field and corrupt one of the other field's sums.

Of course, the total number of bits have not been computed yet. However, the same technique that was used above can be used to compute the total in a series of similar steps. The next step would be:

```
data = (data & 0x33) + ((data >> 2) & 0x33);
```

Continuing the above example (remember that `data` now is 01100010_2):

`data` & 00110011_2			0010	0010
+ (`data >> 2`) & 00110011_2	or	+	0001	0000
			0011	0010

Now there are two 4-bit fields to that are independently added.

The next step is to add these two bit sums together to form the final result:

```
data = (data & 0x0F) + ((data >> 4) & 0x0F);
```

Using the example above (with `data` equal to 00110010_2):

`data` & 00001111_2			00000010
+ (`data >> 4`) & 00001111_2	or	+	00000011
			00000101

Now `data` is 5 which is the correct result. Figure 3.8 shows an implementation of this method that counts the bits in a double word. It uses a `for` loop to compute the sum. It would be faster to unroll the loop; however, the loop makes it clearer how the method generalizes to different sizes of data.

Chapter 4

DOSBox and MASM

4.1 Getting Your Computer Ready

We will be using the Microsoft Assembler (MASM) which can be downloaded from Canvas. Unzip it where you'd like to work on your projects and make a note of the absolute address of the file (`C:\Users\etc`)

4.1.1 Macs

If you have OSX, I recommend downloading the .dmg file from the official DOSBox Downloads page. You can install it as you would install a normal .dmg.

After installation hit 'command' + 'shift' + '.' to show hidden files. Navigate to the hidden `Library` folder in your `Users/username/` folder. In there open Preferences and scroll down to a file called 'DOSBox x.xx-x-x Preferences' and open it as text file (the x's are the version of DOSBox you downloaded).
Scroll to the bottom and add:

```
mount c /Users/yourname/MASM611
set PATH=%PATH%;C:\BIN
c:
```

Don't forget to replace `/Users/yourname/MASM611` with where you unzipped the MASM folder you downloaded.

4.1.2 PCs

If you have a PC, I recommend downloading the .exe file from the official DOSBox Downloads page. You can install it as you would install a normal .exe.
After installation you can go to the Start Menu and select the DosBox folder

and then DOSBox Options. This will open a text file, scroll to the bottom and add:

```
mount c C:\Users\yourname\MASM611
set PATH=%PATH%;C:\BIN
c:
```

Don't forget to replace `C:\Users\yourname\MASM611` with where you unzipped the MASM folder you downloaded.
If you don't have administrative priviledges on your computer there is a Portable Apps version.

4.1.3 Testing DOSBox

Open DOSBox and test it out by typing `ml` at the prompt and then `<enter>`, it should say something like:

```
Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993. All rights reserved.

usage: ML [ options ] filelist [ /link linkoptions]
Run "ML /help" or "ML /?" for more info
```

Congratulations you have successfully installed DOSBox and MASM! One final note, since it is DOS, the preferred number of characters of a filename is 8. While it is possible to have longer names, it will make your life easier if all of your assembly program filenames are less than or equal to eight characters (eg `filename.asm` is fine).

4.2 Using MASM

4.2.1 Assembling

First download "hello.asm" from Canvas and put it in your MASM directory. At the prompt in DOSBox, type in `ml hello.asm` and then `<enter>`, you should see something like this:

```
 Assembling: hello.asm

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

Object Modules [.obj]: hello.obj
Run File [hello.exe]: "hello.exe"
List File [nul.map]: NUL
```

```
Libraries [.lib]:
Definitions File [nul.def]:
```

This is where the assembler would tell you if and where there are errors.

To run the program simply type in the file name without the .asm, in this case it is `hello` then `<enter>`.

```
Hello, world!
```

Hooray!

4.3 Debugging

4.3.1 Assembler Errors

Just like when you compile a program in other languages, the assembler will give you cryptic messages to try to help you debug the errors. First see if you can spot the errors in this code snippet:

```
18    mov ax,tooLarge  ; ax is 16 bits, tooLarge is 32 bits
19    mov eax,tooSmall ; eax is 32 bits, tooSmall is 16 bits
20    mov ecx,ax       ; ecx is 32 bits, ax is 16 bits
21    mov test1,test2  ; test1 is 32 bits, test2 is 32 bits
```

When we try to assemble the code this is what we get:

```
 Assembling: errors.asm
errors.asm(21): error A2070: invalid instruction operands
errors.asm(18): error A2022: instruction operands must be the same size
errors.asm(19): error A2022: instruction operands must be the same size
errors.asm(20): error A2022: instruction operands must be the same size
```

On line 21 there are invalid instruction operands, remember memory to memory operations are not ok. That is usually what this error means. To fix this you'll have to move test2 into a register first:

```
18    mov ax,tooLarge  ; ax is 16 bits, tooLarge is 32 bits
19    mov eax,tooSmall ; eax is 32 bits, tooSmall is 16 bits
20    mov ecx,ax       ; ecx is 32 bits, ax is 16 bits
21    mov ebx,test2    ; store test2 temporarily into ebx
22    mov test1,ebx    ; now move it into test1
```

Then lines 18-20 have all different kinds of size problems with operands. For line 20, you can't move a smaller register into a larger register, if you want

to do that you need to use MOVSX or MOVZX. Helpful comments aside, you don't really know why 18 and 19 have errors until you look at the data segment:

```
6  .data ; start definition of variables
7     tooLarge dd 6 ; 32 bis
8     tooSmall dw 5 ; 16 bits
9     test1 dd 492  ; 32 bits
10    test2 dd 5678 ; 32 bits
```

To fix these errors we could change the size of tooLarge to dw and tooSmall to dd, when we do that they will no longer be the wrong size.
Here is the error free code:

```
title Fixed Errors Example      (noerrors.asm)

.model small ; one data segment, one code segment
.stack 100h ; reserves 256 bytes for the stack
.386 ; for 32 bits
.data ; start definition of variables
   tooLarge dw 6 ;16 bis, no longer too large
   tooSmall dd 5 ;32 bits, no longer too small
   test1 dd 492 ;32 bits
   test2 dd 5678 ;32 bits

.code ; start code portion of program
main proc ; start of the main procedure
   mov  eax,@data ; load the address of the data segment into eax
   mov  ds,eax ; load the address of the data segment into ds
    ; the two previous instructions initalize the data segment

   mov ax,tooLarge ; move 6 into ax
   mov eax,tooSmall ; move 5 into eax
   movzx eax,ax      ; extend ax to eax
   mov ebx,test2    ; store test2 temporarily into ebx
   mov test1,ebx    ; now move it into test1

    ; the following two instructions exit cleanly from the program
   mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar to
       return 0; in C++)
   int  21h ; exit
main endp ; end procedure

end main ; end program
```

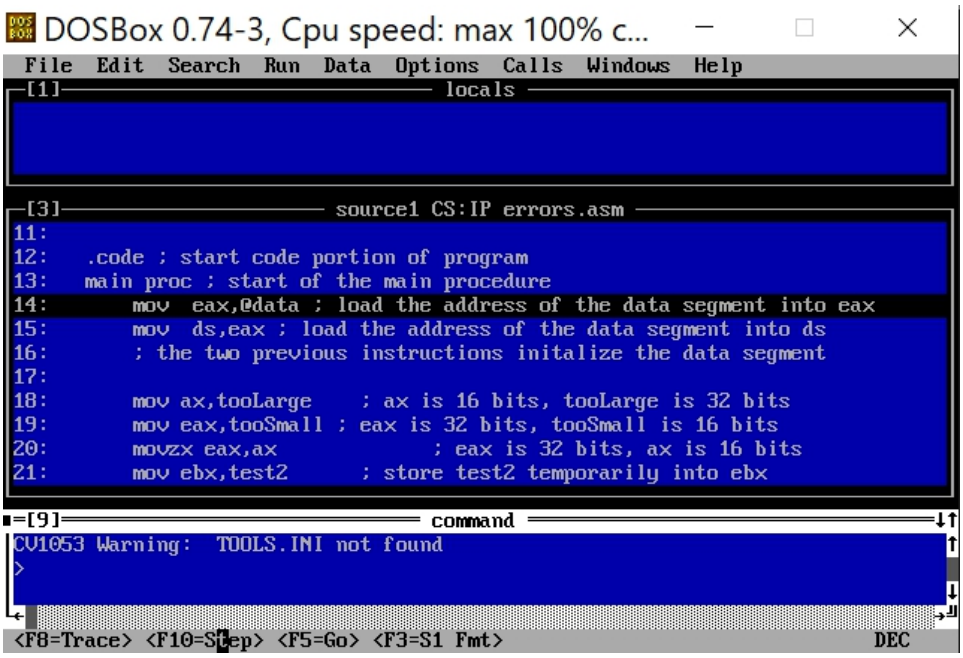

Figure 4.1: Initial CodeView Screen.

4.3.2 Logic Errors

Some of the hardest errors you'll find are logic errors. Generally, the two ways I recommend finding them are print statements and stepping through your code either by hand or with an IDE. The problem is that we don't know how to do print statements in assembly (yet), nor do we have an IDE. Luckily there is a program called CodeView that comes with MASM that helps with that. After your program is assembled and linked you will have a .exe file. That can be run with CodeView to step through your program and see what is happening to the registers. Note: sometimes this version of CodeView misbehaves with 32 bit registers!

If we take the fixed code from section 4.3.1 and link it using `ml /Zi noerrors.asm`, then we can start with CodeView. Start by typing `cv noerrors` and then enter. A screen like this one (Figure 4.1) will pop up. The next thing we will do is open the Register window (click `Windows -> Register`). Since we have one memory location changing we'll add a Watch (click `Data->Add Watch` and type in test1 and hit ok). The window now should look like this (Figure 4.2).

Now we can step through our code and see how the registers change, hit F10 to step through each line. When the program is through, it has the final values of the registers, a garbage value for test1 since the program is

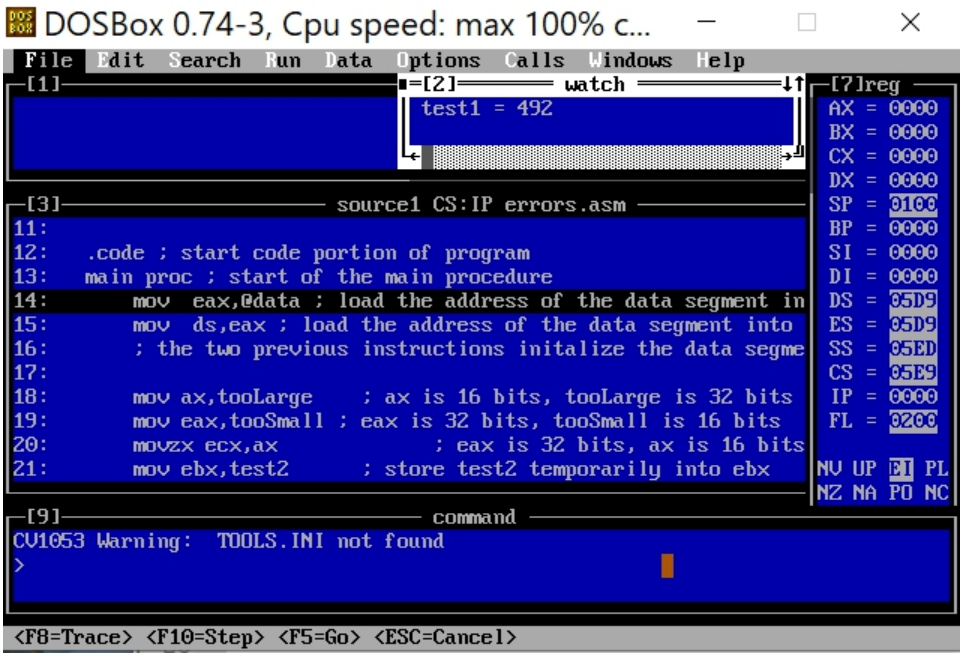

Figure 4.2: CodeView with Registers and a Watch.

over, and gives a peek at the assembled code (Figure 4.3). Just like in most IDEs you are used to you can also set break points instead of running every line and once you have procedures you can use F8 to trace the procedure or F10 to step over it.

4.3.3 DOSBox/MASM Quirks

- Sometimes your mouse will get stuck in the DOSBox window if you hit alt-Tab that will switch to another open window on your computer and will free your mouse.
- If you have an infinite loop or otherwise want to force your program to close, the best way is to just close DOSBox altogether and restart.
- In CodeView sometimes it will stop recognizing the mouse, but the keyboard will still work so you can use the shortcut keys. If you need to choose something from the Menu you can type Alt-F to open the File Menu then you can use the arrow keys and then enter to select what you need. To exit you can simply type Alt-F4.

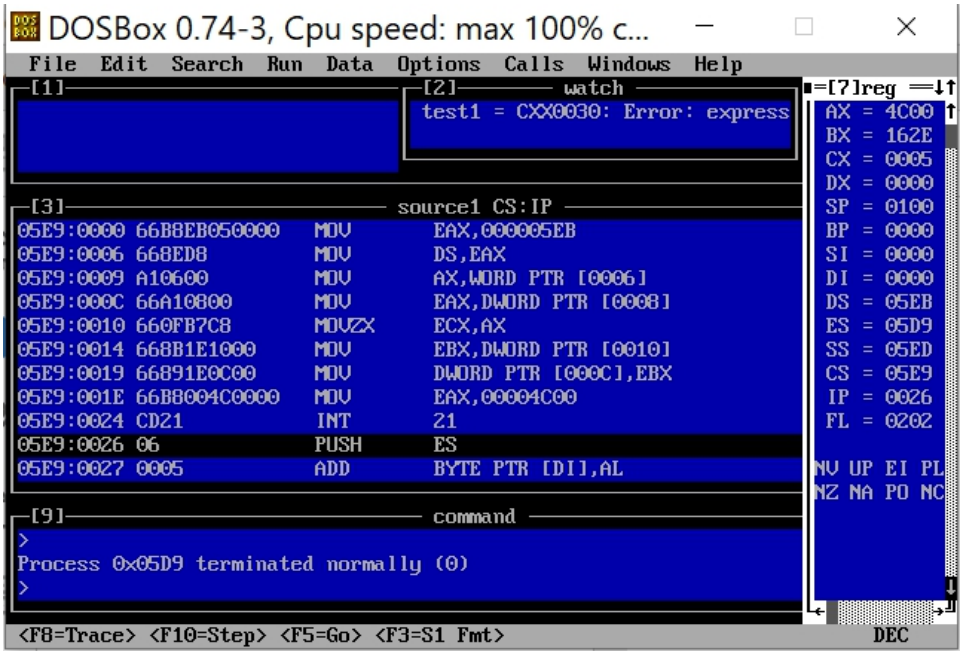

Figure 4.3: CodeView after program runs.

Chapter 5

Subprograms

This chapter looks at using subprograms to make modular programs and to interface with high level languages (like C). Functions and procedures are high level language examples of subprograms.

The code that calls a subprogram and the subprogram itself must agree on how data will be passed between them. These rules on how data will be passed are called *calling conventions*. A large part of this chapter will deal with the standard C calling conventions that can be used to interface assembly subprograms with C programs. This (and other conventions) often pass the addresses of data (*i.e.* pointers) to allow the subprogram to access the data in memory.

5.1 Indirect Addressing

Indirect addressing allows registers to act like pointer variables. To indicate that a register is to be used indirectly as a pointer, it is enclosed in square brackets (`[]`). For example:

```
1       mov    ax, data     ; normal direct memory addressing of a word
2       mov    ebx, offset data     ; ebx = & data
3       mov    ax, [ebx]      ; ax = *ebx
```

Note: Lines 1 and 3 perform the same operation.

Because AX holds a word, line 3 reads a word starting at the address stored in EBX. What EBX is assumed to point to is completely determined by what instructions are used. Furthermore, even the fact that EBX is a pointer is completely determined by the what instructions are used. If EBX is used incorrectly, often there will be no assembler error; however, the program will not work correctly. This is one of the many reasons that assembly programming is more error prone than high level programming.

All the 32-bit general purpose (EAX, EBX, ECX, EDX) and index (ESI, EDI) registers can be used for indirect addressing.

5.2 Simple Subprogram Example

A subprogram is an independent unit of code that can be used from different parts of a program. In other words, a subprogram is like a function in C. A jump can be used to invoke the subprogram, but returning presents a problem. If the subprogram is to be used by different parts of the program, it must return back to the section of code that invoked it. Thus, the jump back from the subprogram can not be hard coded to a label. The code below shows how this could be done using the indirect form of the `JMP` instruction. This form of the instruction uses the value of a register to determine where to jump to (thus, the register acts much like a *function pointer* in C.) Here is a program that uses a subprogram to get a digit from the user.

```
1  title Subprogram example program first try (subproc1.asm)
2
3  .model small ; one data segment, one code segment
4  .stack 100h ; reserves 256 bytes for the stack
5  .386 ; for 32 bits
6  .data
7      prompt db   "Enter a digit: ", "$"     ; don't forget $ terminator
8      outmsg db   0ah,0dh,"You entered ", "$"
9      input db 0
10
11 .code ; start code portion of program
12 main proc ; start of the main procedure
13     mov  eax,@data ; load the address of the data segment into ax
14     mov  ds,eax ; load the address of the data segment into ds
15     ; the two previous instructions initalize the data segment
16
17         ; read in first digit
18         mov dx, offset prompt ; load address of prompt1
19         mov ah, 9             ; load command
20         int 21h               ; print out prompt1
21         mov ebx, offset input ; store address of input into ebx
22         mov ecx, ret1         ; store return address into ecx
23         jmp short get_digit  ; read digit
24 ret1:   ; set return point
25         ; print the result
26         mov dx, offset outmsg
27         mov ah, 9
28         int 21h ; print outmsg
29
30         mov dl, input
31         or  dl, 30h ;digit to ascii
32         mov ah, 2
33         int 21h ; print out input
34
```

```
35          ; the following two instructions exit cleanly from the
                program
36          mov  eax,4C00h ; 4C in ah means exit with code 0 (al)
                (similar to return 0; in C++)
37          int  21h ; exit
38
39  ; subprogram get_digit
40  ; Parameters:
41  ;   ebx - address of byte to store integer into
42  ;   ecx - address of instruction to return to
43  ; Notes:
44  ;   value of eax is destroyed
45  get_digit:
46          mov ah, 1
47          int 21h
48          and al, 0fh    ; char to int
49          mov [ebx], al  ; store input into memory
50          jmp ecx        ; jump back to caller
51
52  main endp ; end procedure
53  end main ; end program
```

The `get_digit` subprogram uses a simple, register-based calling convention. It expects the EBX register to hold the address of the BYTE to store the digit input into and the ECX register to hold the code address of the instruction to jump back to. In line 22, the `ret1` label is used to compute this return address. The `$` operator could have been used to compute the return address. The `$` operator returns the current address for the line it appears on. The expression `$ + 8` computes the address of the `ret1` label on line 24.

Both of these return address computations are awkward. The first method requires a label to be defined for each subprogram call. The second method does not require a label, but does require careful thought. If a near jump was used instead of a short jump, the number to add to `$` would not be 8! Fortunately, there is a much simpler way to invoke subprograms. This method uses the *stack*.

5.3 The Stack

Many CPUs have built-in support for a stack. A stack is a Last-In First-Out (*LIFO*) list. The stack is an area of memory that is organized in this fashion. The `PUSH` instruction adds data to the stack and the `POP` instruction removes data. The data removed is always the last data added (that is why it is called a last-in first-out list).

The SS segment register specifies the segment that contains the stack

(usually this is the same segment data is stored into). The ESP register contains the address of the data that would be removed from the stack. This data is said to be at the *top* of the stack. Data can only be added in double word units. That is, one can not push a single byte on the stack.

The `PUSH` instruction inserts a double word[1] on the stack by subtracting 4 from ESP and then stores the double word at `[ESP]`. The `POP` instruction reads the double word at `[ESP]` and then adds 4 to ESP. The code below demonstrates how these instructions work and assumes that ESP is initially `0100H`.

```
1    push  dword ptr 1  ; 1 stored at 00FCh, ESP = 00FCh
2    push  dword ptr 2  ; 2 stored at 00F8h, ESP = 00F8h
3    push  dword ptr 3  ; 3 stored at 00F4h, ESP = 00F4h
4    pop   eax          ; EAX = 3, ESP = 00F8h
5    pop   ebx          ; EBX = 2, ESP = 00FCh
6    pop   ecx          ; ECX = 1, ESP = 0100h
```

The stack can be used as a convenient place to store data temporarily. It is also used for making subprogram calls, passing parameters and local variables.

The 80x86 also provides a `PUSHA` instruction (push all) that pushes the values of EAX, EBX, ECX, EDX, ESI, EDI and EBP registers (not in this order). The `POPA` instruction (pop all) can be used to pop them all back off.

5.4 The CALL and RET Instructions

The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy. The CALL instruction makes an unconditional jump to a subprogram and *pushes* the address of the next instruction on the stack. The RET instruction *pops off* an address and jumps to that address. When using these instructions, it is very important that one manage the stack correctly so that the right number is popped off by the RET instruction!

The previous program can be rewritten to use these new instructions by changing lines 21 to 24 to be:

```
21      mov   ebx, offset input
22      ; don't need this anymore
23      call  get_digit
24      ; this either
```

[1] Actually words can be pushed too, but in 32-bit protected mode, it is easier to work with only double words on the stack.

and change the subprogram `get_digit` to:

```
get_digit:
    mov ah, 1
    int 21h
    and al, 0fh    ; char to int
    mov [ebx], al  ; store input into memory
    ret        ; jump back to caller
```

There are several advantages to CALL and RET:

- It is simpler!
- It allows subprograms calls to be nested easily. Notice that `get_digit` could call `read_char`. This call pushes another address on the stack. At the end of `read_char`'s code is a RET that pops off the return address and jumps back to `get_digit`'s code. Then when `get_digit`'s RET is executed, it pops off the return address that jumps back to `main`. This works correctly because of the LIFO property of the stack.

```
get_digit:
    call read_char
    and al, 0fh    ; char to int
    mov [ebx], al  ; store input into memory
    ret        ; jump back to caller

read_char:
    mov ah, 1
    int 21h
    ret
```

Remember it is *very* important to pop off all data that is pushed on the stack. For example, consider the following:

```
get_digit:
    mov ah, 1
    int 21h
    and al, 0fh    ; char to int
    mov [ebx], al  ; store input into memory
    push eax
    ret        ; pops off eax value, not the return address!!
```

This code would not return correctly!

5.5 Calling Conventions

When a subprogram is invoked, the calling code and the subprogram (the *callee*) must agree on how to pass data between them. High-level languages

have standard ways to pass data known as *calling conventions*. For high-level code to interface with assembly language, the assembly language code must use the same conventions as the high-level language. The calling conventions can differ from compiler to compiler or may vary depending on how the code is compiled (*e.g.* if optimizations are on or not). One universal convention is that the code will be invoked with a `CALL` instruction and return via a `RET`.

Calling conventions allow one to create subprograms that are *reentrant*. A reentrant subprogram may be called at any point of a program safely (even inside the subprogram itself).

5.5.1 "Passing" parameters using registers

As we saw above, the easiest way you can "pass" parameters is to use the registers. I say pass in quotes because you aren't doing anything extra like you do in C++. Nevertheless, you can use the existing values in your subprogram. The changes you make will change the registers back in main, similar to passing by reference.

Passing registers by value

Sometimes you might want to change the registers in the procedure, but not have those changes reflected in main. This is similar to a local variable in C++ or passing a parameter by value. There are two common ways to do this, the first is to use push and pop to keep the registers from changing, here is an example from earlier, but now the value of eax is preserved:

```
1  get_digit:
2     push eax
3     mov ah, 1
4     int 21h
5     and al, 0fh    ; char to int
6     mov [ebx], al  ; store input into memory
7     pop eax
8     ret
```

The second way does the exact same thing, but without push and pop cluttering your code (the assembler adds those later). To preserve the registers this way you use PROC with USES, this looks cleaner, and is explicit that EAX is being preserved.

```
1  get_digit proc uses eax
2     mov ah, 1
3     int 21h
4     and al, 0fh    ; char to int
5     mov [ebx], al  ; store input into memory
```

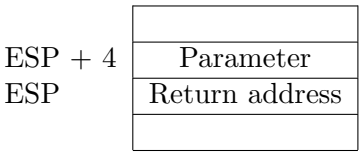

Figure 5.1:

```
6     ret
7 get_digit endp
```

Note the ENDP at the bottom, this is needed to close the procedure. We didn't need to do this when we were using a named location for our procedure, but it becomes necessary when you use PROC. Since we use this we also have to move `MAIN ENDP` above the procedure because we can't have nested procedures.

5.5.2 Passing parameters on the stack

Parameters to a subprogram may be passed on the stack. They are pushed onto the stack before the `CALL` instruction. Just as in C, if the parameter is to be changed by the subprogram, the *address* of the data must be passed, not the *value*. If the parameter's size is less than a double word, it must be converted to a double word before being pushed.

The parameters on the stack are not popped off by the subprogram, instead they are accessed from the stack itself. Why?

- Since they have to be pushed on the stack before the `CALL` instruction, the return address would have to be popped off first (and then pushed back on again).
- Often the parameters will have to be used in several places in the subprogram. Usually, they can not be kept in a register for the entire subprogram and would have to be stored in memory. Leaving them on the stack keeps a copy of the data in memory that can be accessed at any point of the subprogram.

Consider a subprogram that is passed a single parameter on the stack. When the subprogram is invoked, the stack looks like Figure 5.1. The parameter can be accessed using indirect addressing (`[ESP+4]` [2]).

If the stack is also used inside the subprogram to store data, the number needed to be added to ESP will change. For example, Figure 5.2 shows what the stack looks like if a DWORD is pushed the stack. Now the parameter is

[2] It is legal to add a constant to a register when using indirect addressing. More complicated expressions are possible too. This topic is covered in the next chapter

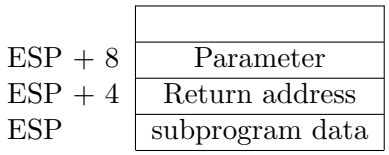

Figure 5.2:

```
1  subprogram_label:
2       push   ebp          ; save original EBP value on stack
3       mov    ebp, esp     ; new EBP = ESP
4  ; subprogram code
5       pop    ebp          ; restore original EBP value
6       ret
```

Figure 5.3: General subprogram form

at `ESP + 8` not `ESP + 4`. Thus, it can be very error prone to use ESP when referencing parameters. To solve this problem, the 80386 supplies another register to use: EBP. This register's only purpose is to reference data on the stack. The C calling convention mandates that a subprogram first save the value of EBP on the stack and then set EBP to be equal to ESP. This allows ESP to change as data is pushed or popped off the stack without modifying EBP. At the end of the subprogram, the original value of EBP must be restored (this is why it is saved at the start of the subprogram.) Figure 5.3 shows the general form of a subprogram that follows these conventions.

Lines 2 and 3 in Figure 5.3 make up the general *prologue* of a subprogram. Lines 5 and 6 make up the *epilogue*. Figure 5.4 shows what the stack looks like immediately after the prologue. Now the parameter can be access with `[EBP + 8]` at any place in the subprogram without worrying about what else has been pushed onto the stack by the subprogram.

After the subprogram is over, the parameters that were pushed on the stack must be removed. The C calling convention specifies that the caller code must do this. Other conventions are different. For example, the Pascal calling convention specifies that the subprogram must remove the parameters. (There is another form of the RET instruction that makes this easy to do.) Some C compilers support this convention too.

Figure 5.5 shows how a subprogram using the C calling convention would be called. Line 3 removes the parameter from the stack by directly manipulating the stack pointer. A `POP` instruction could be used to do this also, but would require the useless result to be stored in a register. Actually, for this particular case, many compilers would use a `POP ECX` instruction to remove the parameter. The compiler would use a `POP` instead of an `ADD` because the

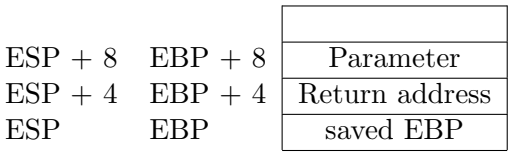

Figure 5.4:

```
1       push   dword ptr 1       ; pass 1 as parameter
2       call   funC
3       add    esp, 4         ; remove parameter from stack
```

Figure 5.5: Sample C style subprogram call

`ADD` requires more bytes for the instruction. However, the `POP` also changes ECX's value! Figure 5.6 show how a subprogram using the Pascal calling convention would remove the parameter from the stack.

Next is another example program with two subprograms that use the C calling conventions discussed above. Line 64 (and other lines) shows that multiple data and code segments may be declared in a single source file. They will be combined into single data and code segments in the linking process. Splitting up the data and code into separate segments allow the data that a subprogram uses to be defined close by the code of the subprogram.

```
1  title Number Sum       (numSum.asm)
2
3
4  .model small ; one data segment, one code segment
5  .stack 100h ; reserves 256 bytes for the stack
6  .386 ; for 32 bit registers
7  includelib printInc.lib ;include external subprocedures
8  ;prototypes for external subprocs
9  extern printDec:proc
10 extern crlf:proc
11 extern get_digit:proc
12
13 .data ; start definition of variables
14 sum    dd   0
15 input  dd   1
16
17 ; pseudo-code algorithm
18 ; i = 1;
19 ; input = 1;
20 ; sum = 0;
```

```
1       push  dword ptr 1      ; pass 1 as parameter
2       call  funPascal
3       ...
4       funPascal PROC
5           ;subprogram code
6           ret 4 ;remove parameter from stack
7        funPascal ENDP
```

Figure 5.6: Sample Pascal style subprogram call

```
21  ; while(input != 0) {
22  ;   input = get_int()
23  ;   sum += input;
24  ;   i++;
25  ; }
26  ; print_sum(num);
27
28  .code ; start code portion of program
29  main proc ; start of the main procedure
30      mov  eax,@data ; load the address of the data segment into eax
31      mov  ds,eax ; load the address of the data segment into ds
32      ; the two previous instructions initalize the data segment
33
34      mov edx, 1         ; edx is 'i' in pseudo-code so i=1
35  while_loop:
36      push edx           ; save i on stack
37      push offset input ; push address of input on stack
38      call get_int
39      add esp, 2         ; remove &input from stack
40      pop edx          ; remove i from the stack and save it in edx
41      mov eax, input
42      cmp eax, 0
43      je end_while
44      add sum, eax       ; sum += input
45      inc edx
46      jmp while_loop
47
48  end_while:
49      push sum      ; push value of sum onto stack
50      call print_sum
51      add esp,4     ; remove sum from stack
52
53      ; the following two instructions exit cleanly from the program
54      mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar
            to return 0; in C++)
55      int  21h ; exit
```

```
56  main endp ; end procedure
57
58  ; subprogram get_int
59  ; Parameters (in order pushed on stack)
60  ;   number of input (at [ebp + 8])
61  ;   address of word to store input into (at [ebp + 6])
62  ; Notes:
63  ;   values of eax, edx, and ebx are destroyed
64  get_int proc
65  .data
66      prompt db ") Enter a digit (0 to quit): ", '$'
67
68  .code
69      push ebp
70      mov ebp, esp
71
72      mov eax, [ebp + 8]
73      call printDec
74
75      push eax
76      mov dx, offset prompt
77      mov  ah,9
78      int 21h
79      pop eax
80
81      call get_digit
82      mov bx, [ebp + 6]
83      mov [bx], eax       ; store input into memory
84
85      pop ebp
86      ret                 ; jump back to caller
87  get_int endp
88  ; subprogram print_sum
89  ; prints out the sum
90  ; Parameter:
91  ;   sum to print out (at [ebp+6])
92  ; Note: destroys value of eax and edx
93  ;
94  print_sum proc
95  .data
96  result db "The sum is ", '$'
97
98  .code
99      push ebp
100     mov ebp, esp
101
102     push eax
103     mov dx, offset result
```

```
subprogram_label:
      push   ebp               ; save original EBP value on stack
      mov    ebp, esp          ; new EBP = ESP
      sub    esp, LOCAL_BYTES  ; = # bytes needed by locals
; subprogram code
      mov    esp, ebp          ; deallocate locals
      pop    ebp               ; restore original EBP value
      ret
```

Figure 5.7: General subprogram form with local variables

```
    mov  ah,9
    int 21h
    pop eax

    mov eax,[ebp+6]
    call printDec
    call crlf
    pop ebp
    ret
print_sum endp
end main ; end program
```

5.5.3 Local variables on the stack

The stack can be used as a convenient location for local variables. This is exactly where C stores normal (or *automatic* in C lingo) variables. Using the stack for variables is important if one wishes subprograms to be reentrant. A reentrant subprogram will work if it is invoked at any place, including the subprogram itself. In other words, reentrant subprograms can be invoked *recursively*. Using the stack for variables also saves memory. Data not stored on the stack is using memory from the beginning of the program until the end of the program (C calls these types of variables *global* or *static*). Data stored on the stack only use memory when the subprogram they are defined for is active.

Local variables are stored right after the saved EBP value in the stack. They are allocated by subtracting the number of bytes required from ESP in the prologue of the subprogram. Figure 5.7 shows the new subprogram skeleton. The EBP register is used to access local variables. Consider the C function in Figure 5.8. Figure 5.9 shows how the equivalent subprogram could be written in assembly.

Figure 5.10 shows what the stack looks like after the prologue of the program in Figure 5.9. This section of the stack that contains the parameters,

```
1 void calc_sum( int n, int *sumP) {
2   int i, sum = 0;
3
4   for( i=1; i <= n; i++ ) {
5     sum += i;
6    }
7   *sumP = sum;
8 }
```

Figure 5.8: C version of sum

return information and local variable storage is called a *stack frame*. Every invocation of a C function creates a new stack frame on the stack.

The prologue and epilogue of a subprogram can be simplified by using two special instructions that are designed specifically for this purpose. The `ENTER` instruction performs the prologue code and the `LEAVE` performs the epilogue. The `ENTER` instruction takes two immediate operands. For the C calling convention, the second operand is always 0. The first operand is the number of bytes needed by local variables. The `LEAVE` instruction has no operands. Figure 5.12 shows how these instructions are used.

5.6 Multi-Module Programs

A *multi-module program* is one composed of more than one object file. The numSum.asm example program above is a multi-module program. It consists of the Assembly driver object file and the assembly object file. Recall that the linker combines the object files into a single executable program. The linker must match up references made to each label in one module (*i.e.* object file) to its definition in another module. In order for module A to use a label defined in module B, the `extern` directive must be used. After the `extern` directive comes a label and then `:proc` to let it know the label is a procedure. The directive tells the assembler to treat these labels as *external* to the module. That is, these are labels that can be used in this module, but are defined in another. The `printInc.lib` file defines the `printDec`, *etc.* routines as external.

5.7 Reentrant and Recursive Subprograms

A reentrant subprogram must satisfy the following properties:

- It must not modify any code instructions. In a high level language this would be difficult, but in assembly it is not hard for a program to

```
1  ; stack has the address of sumP and the value of n
2  cal_sum:
3       push   ebp
4       mov    ebp, esp
5       sub    esp, 4              ; make room for local sum
6
7       mov    dword ptr [ebp - 4], 0 ; sum = 0
8       mov    ebx, 1              ; ebx (i) = 1
9  for_loop:
10      cmp    ebx, [ebp+8]        ; is i <= n?
11      jnle   end_for
12
13      add    [ebp-4], ebx        ; sum += i
14      inc    ebx
15      jmp    short for_loop
16
17 end_for:
18      mov    ebx, [ebp+12]       ; ebx = sumP
19      mov    eax, [ebp-4]        ; eax = sum
20      mov    [ebx], eax          ; *sumP = sum;
21
22      mov    esp, ebp
23      pop    ebp
24      ret
```

Figure 5.9: Assembly version of sum

try to modify its own code. For example:

```
mov word ptr [cs:$+9], 5 ; copy 5 into the word 7 bytes ahead
add ax, 2              ; previous statement changes 2 to 5!
```

This code would work in real mode, but in protected mode operating systems the code segment is marked as read only. When the first line above executes, the program will be aborted on these systems. This type of programming is bad for many reasons. It is confusing, hard to maintain and does not allow code sharing (see below).

- It must not modify global data (such as data in the `data` and the `bss` segments). All variables are stored on the stack.

There are several advantages to writing reentrant code.

- A reentrant subprogram can be called recursively.
- A reentrant program can be shared by multiple processes. On many multi-tasking operating systems, if there are multiple instances of a

ESP + 16	EBP + 12	address of `sumP`
ESP + 12	EBP + 8	`n`
ESP + 8	EBP + 4	Return address
ESP + 4	EBP	saved EBP
ESP	EBP - 4	`sum`

Figure 5.10:

```
1 subprogram_label:
2       enter LOCAL_BYTES, 0    ; = # bytes needed by locals
3 ; subprogram code
4       leave
5       ret
```

Figure 5.11: General subprogram form with local variables using `ENTER` and `LEAVE`

program running, only *one* copy of the code is in memory. Shared libraries and DLL's (*Dynamic Link Libraries*) use this idea as well.

- Reentrant subprograms work much better in *multi-threaded* [3] programs. Windows 9x/NT and most UNIX-like operating systems (Solaris, Linux, *etc.*) support multi-threaded programs.

5.7.1 Recursive subprograms

These types of subprograms call themselves. The recursion can be either *direct* or *indirect*. Direct recursion occurs when a subprogram, say `foo`, calls itself inside `foo`'s body. Indirect recursion occurs when a subprogram is not called by itself directly, but by another subprogram it calls. For example, subprogram `foo` could call `bar` and `bar` could call `foo`.

Recursive subprograms must have a *termination condition*. When this condition is true, no more recursive calls are made. If a recursive routine does not have a termination condition or the condition never becomes true, the recursion will never end (much like an infinite loop).

Figure 5.12 shows a function that calculates factorials recursively. It could be called from C with:

```
x = factorial(3);       /* find 3! */
```

[3] A multi-threaded program has multiple threads of execution. That is, the program itself is multi-tasked.

```
1  ; finds n!
2  factorial proc
3      enter 0,0
4      mov   eax, [ebp+4]   ; eax = n
5      cmp   eax, 1
6      jbe   term_cond      ; if n <= 1, terminate
7      dec   eax
8      push  eax
9      call  factorial      ; eax = fact(n-1)
10     pop   ecx            ; answer in eax
11     mul   dword ptr [ebp+4] ; edx:eax = eax * [ebp+8]
12     jmp   end_fact
13     term_cond:
14     mov   eax, 1
15     end_fact:
16     leave
17     ret
18 factorial endp
```

Figure 5.12: Recursive factorial function

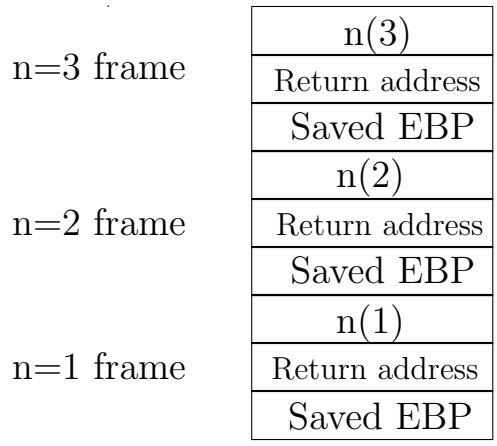

Figure 5.13: Stack frames for factorial function

Figure 5.13 shows what the stack looks like at its deepest point for the above function call.

Chapter 6

Arrays

6.1 Introduction

An *array* is a contiguous block of data in memory. Each element of the list must be the same type and use exactly the same number of bytes of memory for storage. Because of these properties, arrays allow efficient access of the data by its position (or index) in the array. The address of any element can be computed by knowing three facts:

- The address of the first element of the array.
- The number of bytes in each element
- The index of the element

It is convenient to consider the index of the first element of the array to be zero (just as in C). It is possible to use other values for the first index, but it complicates the computations.

6.1.1 Defining arrays

Defining arrays in the `data` segment

To define an initialized array in the `data` segment, use the normal `db`, `dw`, *etc.* directives. MASM also provides a useful directive named `DUP` that can be used to repeat a statement many times without having to duplicate the statements by hand. Figure 6.1 shows several examples of these.

To define an uninitialized array, use the `?` directive. Figure 6.1 also shows examples of these types of definitions.

```
.data
   ; define array of 10 double words initialized to 1,2,..,10
   a1 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
   ; define array of 10 words initialized to 0
   a2 dw 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
   ; same as before using dup
   a3 dw 10 dup (0)
   ; define array of bytes with 200 0's and then 100 1's
   a4 db 200 dup (0)
      db 100 dup (1)
   ; define an array of 10 uninitialized double words
   a5 dd 10 dup (?)
   ; define an array of 100 uninitialized words
   a6 dw 100 dup (?)
```

Figure 6.1: Defining arrays

Defining arrays as local variables on the stack

There is no direct way to define a local array variable on the stack. As before, one computes the total bytes required by *all* local variables, including arrays, and subtracts this from ESP (either directly or using the `ENTER` instruction). For example, if a function needed a character variable, two double word integers and a 50 element word array, one would need $1 + 2 \times 4 + 50 \times 2 = 109$ bytes. However, the number subtracted from ESP should be a multiple of four (112 in this case) to keep ESP on a double word boundary. One could arrange the variables inside this 109 bytes in several ways. Figure 6.2 shows two possible ways. The unused part of the first ordering is there to keep the double words on double word boundaries to speed up memory accesses.

6.1.2 Accessing elements of arrays

The `[ ]` operator in assembly language is much more versatile than C. You can use it similarly to how you might use it in C to access an array, but to access the correct element of an array, its address must be computed. Consider the following two array definitions:

```
array1 db 5, 4, 3, 2, 1   ; array of bytes
array2 dw 5, 4, 3, 2, 1   ; array of words
array3 dd 5, 4, 3, 2, 1   ; array of double words
```

Here are some examples using these arrays:

```
   mov   al, [array1]          ; al = array1[0]
```

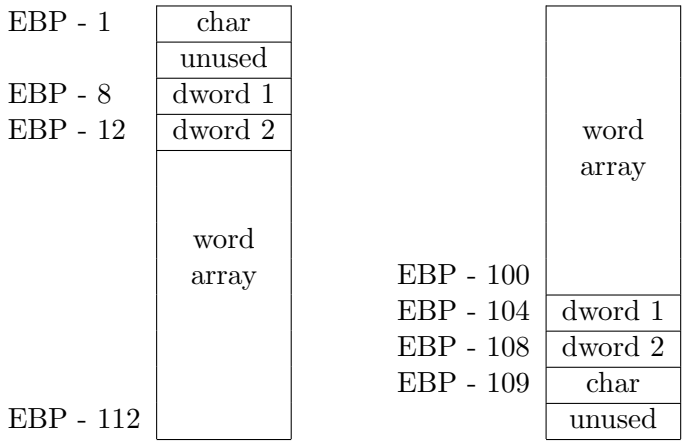

Figure 6.2: Arrangements of the stack

Command	Purpose
type	The number of bytes in each element
lengthof	The number of elements in an array
sizeof	The number of bytes in an array

Table 6.1: Helpful Array Operators

```
2    mov   al, array1[1]       ; al = array1[1]
3    mov   [array1 + 3], al      ; array1[3] = al
4    mov   ax, array2           ; ax = array2[0]
5    mov   ax, array2[2]       ; ax = array2[1] (NOT array2[2]!)
6    mov   [array2 + 6], ax      ; array2[3] = ax
7    mov   ax, [array2 + 1]      ; ax = ??
```

In line 5, element 1 of the word array is referenced, not element 2. Why? Words are two byte units, so to move to the next element of a word array, one must move two bytes ahead, not one. Line 7 will read one byte from the first element and one from the second. In C, the compiler looks at the type of a pointer in determining how many bytes to move in an expression that uses pointer arithmetic so that the programmer does not have to. However, in assembly, it is up to the programmer to take the size of array elements in account when moving from element to element. Luckily, there is a way to calculate the important values associated with an array, see Table 6.1.

Figure 6.3 shows a code snippet that adds all the elements of `array1` in the previous example code. In line 7, AX is added to DX. Why not AL? First, the two operands of the `ADD` instruction must be the same size. Secondly, it would be easy to add up bytes and get a sum that was too big to fit into a byte. By using DX, sums up to 65,535 are allowed. However, it is important to realize that AH is being added also. This is why AH is set

```
1    mov   ebx, offset array1        ; ebx = address of array1
2    mov   dx, 0               ; dx will hold sum
3    mov   ah, 0               ; ?
4    mov   ecx, 5
5  lp:
6    mov   al, [ebx]           ; al = *ebx
7    add   dx, ax              ; dx += ax (not al!)
8    inc   ebx                 ; bx++
9    loop  lp
```

Figure 6.3: Summing elements of an array (Version 1)

```
1       mov   ebx, offset array1        ; ebx = address of array1
2       mov   dx, 0               ; dx will hold sum
3       mov   ecx, 5
4  lp:
5       add   dl, [ebx]           ; dl += *ebx
6       jnc   next                ; if no carry goto next
7       inc   dh                  ; inc dh
8  next:
9       inc   ebx                 ; bx++
10      loop  lp
```

Figure 6.4: Summing elements of an array (Version 2)

to zero[1] in line 3.

Figures 6.4 and 6.5 show two alternative ways to calculate the sum. The differences are mostly around lines 6 and 7 of each Figure.

6.1.3 More advanced indirect addressing

Not surprisingly, indirect addressing is often used with arrays. The most general form of an indirect memory reference is:

[*base reg* + *factor***index reg* + *constant*]

where:

base reg is one of the registers EAX, EBX, ECX, EDX, EBP, ESP, ESI or EDI.

factor is either 1, 2, 4 or 8 . (If 1, factor is omitted.)

[1]Setting AH to zero is implicitly assuming that AL is an unsigned number. If it is signed, the appropriate action would be to insert a `CBW` instruction between lines 6 and 7

```
1       mov    ebx, offset array1        ; ebx = address of array1
2       mov    dx, 0                ; dx will hold sum
3       mov    ecx, 5
4   lp:
5       add    dl, [ebx]            ; dl += *ebx
6       adc    dh, 0                ; dh += carry flag + 0
7       inc    ebx                  ; bx++
8       loop   lp
```

Figure 6.5: Summing elements of an array (Version 3)

```
1     mov    ebx, offset array3   ; ebx = address of array3
2     mov    edx, 0               ; edx will hold sum
3     mov    ecx, 5               ; ecx is our loop counter
4   lp:          ;[baseReg + factor*indexReg + constant]
5     add    edx, [ebx+(type array3)*ecx-(type array3)]; edx +=
          array3[ecx-1]
6     loop   lp
```

Figure 6.6: Summing elements of an array (Version 4)

index reg is one of the registers EAX, EBX, ECX, EDX, EBP, ESI, EDI. (Note that ESP is not in list.)

constant is a 32-bit constant. The constant can be a label (or a label expression).

Figure 6.6 is another sum array example that uses the general form.

The `LEA` instruction

The `LEA` instruction is used to calculate addresses instead of `offset`. In addition to this, you could use it for other purposes. A fairly common one is for fast computations. Consider the following:

```
lea    ebx, [4*eax + eax]
```

This effectively stores the value of $5 \times$ `EAX` into EBX. Using `LEA` to do this is both easier and faster than using `MUL`. However, one must realize that the expression inside the square brackets *must* be a legal indirect address. Thus, for example, this instruction can not be used to multiple by 6 quickly.

6.1.4 Multidimensional Arrays

Multidimensional arrays are not really very different than the plain one dimensional arrays already discussed. In fact, they are represented in memory as just that, a plain one dimensional array.

Two Dimensional Arrays

Not surprisingly, the simplest multidimensional array is a two dimensional one. A two dimensional array is often displayed as a grid of elements. Each element is identified by a pair of indices. By convention, the first index is identified with the row of the element and the second index the column.

Consider an array with three rows and two columns defined as:

```
int a[3][2];
```

The C compiler would reserve room for a 6 ($= 2 \times 3$) integer array and map the elements as follows:

Index	0	1	2	3	4	5
Element	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

What the table attempts to show is that the element referenced as `a[0][0]` is stored at the beginning of the 6 element one dimensional array. Element `a[0][1]` is stored in the next position (index 1) and so on. Each row of the two dimensional array is stored contiguously in memory. The last element of a row is followed by the first element of the next row. This is known as the *rowwise* representation of the array and is how a C/C++ compiler would represent the array.

How does the compiler determine where `a[i][j]` appears in the rowwise representation? A simple formula will compute the index from `i` and `j`. The formula in this case is $2i + j$. It's not too hard to see how this formula is derived. Each row is two elements long; so, the first element of row i is at position $2i$. Then the position of column j is found by adding j to $2i$. This analysis also shows how the formula is generalized to an array with `N` columns: $N \times i + j$. Notice that the formula does *not* depend on the number of rows.

As an example, let us see how *gcc* compiles the following code (using the array `a` defined above):

```
x = a[i][j];
```

The compiler essentially converts the code to:

```
x = *(&a[0][0] + 2*i + j);
```

and in fact, the programmer could write this way with the same result. The example code below shows how you might find the sum of a 2D array in assembly.

```
1  title Sum a 2D Array   (2dSum.asm)
2
3  .model small ; one data segment, one code segment
4  .stack 100h ; reserves 256 bytes for the stack
5  .386 ; for 32 bit registers
6  includelib printInc.lib ;include external subprocedures
7  extern printDec:proc
8  extern crlf:proc
9  .data ; start definition of variables
10 array2d dd  1, 2, 3, 4, 5,  ; 2d array of double words
11             2, 4, 6, 8, 10,
12             3, 6, 9, 12, 15
13 COLS equ 5
14 ROWS equ 3
15
16 .code ; start code portion of program
17 main proc ; start of the main procedure
18     mov  eax,@data ; load the address of the data segment into eax
19     mov  ds,eax ; load the address of the data segment into ds
20     ; the two previous instructions initalize the data segment
21     mov eax,0 ; eax is i
22     mov ebx,0 ; ebx is j
23     mov edx,0 ; edx is the sum
24 while_row:
25     cmp eax,ROWS*COLS ; while i<numRows
26     jge exit ; else exit
27     while_col:
28         cmp ebx,COLS*(type array2d) ; while j<numCols
29         jge while_row_rest ; else return to outer loop
30         add edx,array2d[(type array2d)*eax+ebx] ;array2d[i][j]
31         add ebx,type array2d
32         jmp while_col
33     while_row_rest:
34         add eax,COLS
35         mov ebx,0
36         jmp while_row
37
38 exit:
39     mov eax,edx
40     call printDec
41     call crlf
42     ; the following two instructions exit cleanly from the program
43     mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar
           to return 0; in C++)
```

```
44      int  21h ; exit
45  main endp ; end procedure
46
47  end main ; end program
```

There is nothing magical about the choice of the rowwise representation of the array. A columnwise representation would work just as well:

Index	0	1	2	3	4	5
Element	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

In the columnwise representation, each column is stored contiguously. Element `[i][j]` is stored at position $i + 3j$. Other languages (FORTRAN, for example) use the columnwise representation. This is important when interfacing code with multiple languages.

Dimensions Above Two

For dimensions above two, the same basic idea is applied. Consider a three dimensional array:

```
int b[4][3][2];
```

This array would be stored like it was four two dimensional arrays each of size `[3][2]` consecutively in memory. The table below shows how it starts out:

Index	0	1	2	3	4	5
Element	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
Index	6	7	8	9	10	11
Element	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

The formula for computing the position of `b[i][j][k]` is $6i + 2j + k$. The 6 is determined by the size of the `[3][2]` arrays. In general, for an array dimensioned as `a[L][M][N]` the position of element `a[i][j][k]` will be $M \times N \times i + N \times j + k$. Notice again that the first dimension (`L`) does not appear in the formula.

For higher dimensions, the same process is generalized. For an n dimensional array of dimensions D_1 to D_n, the position of element denoted by the indices i_1 to i_n is given by the formula:

$$D_2 \times D_3 \cdots \times D_n \times i_1 + D_3 \times D_4 \cdots \times D_n \times i_2 + \cdots + D_n \times i_{n-1} + i_n$$

or for the über math geek, it can be written more succinctly as:

$$\sum_{j=1}^{n} \left(\prod_{k=j+1}^{n} D_k \right) i_j$$

LODSB AL = [ESI] ESI = ESI ± 1	STOSB [EDI] = AL EDI = EDI ± 1
LODSW AX = [ESI] ESI = ESI ± 2	STOSW [EDI] = AX EDI = EDI ± 2
LODSD EAX = [ESI] ESI = ESI ± 4	STOSD [EDI] = EAX EDI = EDI ± 4

Figure 6.7: Reading and writing string instructions

The first dimension, D_1, does not appear in the formula.

For the columnwise representation, the general formula would be:

$$i_1 + D_1 \times i_2 + \cdots + D_1 \times D_2 \times \cdots \times D_{n-2} \times i_{n-1} + D_1 \times D_2 \times \cdots \times D_{n-1} \times i_n$$

This is where you can tell the author was a physics major. (Or was the reference to FORTRAN a giveaway?)

or in über math geek notation:

$$\sum_{j=1}^{n} \left(\prod_{k=1}^{j-1} D_k \right) i_j$$

In this case, it is the last dimension, D_n, that does not appear in the formula.

6.2 Array/String Instructions

The 80x86 family of processors provide several instructions that are designed to work with arrays. These instructions are called *string instructions*. They use the index registers (ESI and EDI) to perform an operation and then to automatically increment or decrement one or both of the index registers. The *direction flag* (DF) in the FLAGS register determines where the index registers are incremented or decremented. There are two instructions that modify the direction flag:

CLD clears the direction flag. In this state, the index registers are incremented.

STD sets the direction flag. In this state, the index registers are decremented.

A *very* common mistake in 80x86 programming is to forget to explicitly put the direction flag in the correct state. This often leads to code that works most of the time (when the direction flag happens to be in the desired state), but does not work *all* the time.

```
1  .data ; start definition of variables
2  string db "test",0
3  array1 dd  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
4  array2 dd 10 dup (?)
5
6  .code ; start code portion of program
7  main proc ; start of the main procedure
8      mov  eax,@data ; load the address of the data segment into eax
9      mov  ds,eax ; load the address of the data segment into ds
10     mov  es,eax ; load the address of the data segment into es
11     ; the three previous instructions initalize the data segment and
12
13     ;copies array1 into array2
14       cld      ; don't forget this!
15       mov    esi, offset array1
16       mov    edi, offset array2
17       mov    ecx, lengthof array1
18 lp:
19       lodsd
20       stosd
21       loop  lp
```

Figure 6.8: Load and store example

6.2.1 Reading and writing memory

The simplest string instructions either read or write memory or both. They may read or write a byte, word or double word at a time. Figure 6.7 shows these instructions with a short pseudo-code description of what they do. There are several points to notice here. First, ESI is used for reading and EDI for writing. It is easy to remember this if one remembers that SI stands for *Source Index* and DI stands for *Destination Index*. Next, notice that the register that holds the data is fixed (either AL, AX or EAX). Finally, note that the storing instructions use ES to determine the segment to write to, not DS. In protected mode programming this is not usually a problem, since there is only one data segment and ES should be automatically initialized to reference it (just as DS is). However, in real mode programming, it is *very* important for the programmer to initialize ES to the correct segment selector value. What this means to you is that you need to add `mov es,eax` to the beginning of your code (see line 10 in Figure 6.8). Figure 6.8 shows an example use of these instructions that copies an array into another.

The combination of a `LODSx` and `STOSx` instruction (as in lines 19 and 20 of Figure 6.8) is very common. In fact, this combination can be performed by a single `MOVSx` string instruction. Figure 6.9 describes the operations that

MOVSB	byte [EDI] = byte [ESI] ESI = ESI ± 1 EDI = EDI ± 1
MOVSW	word [EDI] = word [ESI] ESI = ESI ± 2 EDI = EDI ± 2
MOVSD	dword [EDI] = dword [ESI] ESI = ESI ± 4 EDI = EDI ± 4

Figure 6.9: Memory move string instructions

```
1         cld       ; don't forget this!
2         mov   edi, offset array1
3         mov   ecx, 10
4         xor   eax, eax
5         rep stosd
```

Figure 6.10: Zero array example

these instructions perform. Lines 19 and 20 of Figure 6.8 could be replaced with a single `MOVSD` instruction with the same effect. The only difference would be that the EAX register would not be used at all in the loop.

6.2.2 The `REP` instruction prefix

The 80x86 family provides a special instruction prefix[2] called `REP` that can be used with the above string instructions. This prefix tells the CPU to repeat the next string instruction a specified number of times. The ECX register is used to count the iterations (just as for the `LOOP` instruction). Using the `REP` prefix, the loop in Figure 6.8 (lines 18 to 21) could be replaced with a single line:

```
rep movsd
```

Figure 6.10 shows another example that zeroes out the contents of an array.

[2] A instruction prefix is not an instruction, it is a special byte that is placed before a string instruction that modifies the instructions behavior. Other prefixes are also used to override segment defaults of memory accesses

CMPSB	compares byte [ESI] and byte [EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	compares word [ESI] and word [EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	compares dword [ESI] and dword [EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	compares AL and [EDI] EDI ± 1
SCASW	compares AX and [EDI] EDI ± 2
SCASD	compares EAX and [EDI] EDI ± 4

Figure 6.11: Comparison string instructions

6.2.3 Comparison string instructions

Figure 6.11 shows several new string instructions that can be used to compare memory with other memory or a register. They are useful for comparing or searching arrays. They set the FLAGS register just like the `CMP` instruction. The `CMPSx` instructions compare corresponding memory locations and the `SCASx` scan memory locations for a specific value.

Figure 6.12 shows a short code snippet that searches for the number 12 in a double word array. The `SCASD` instruction on line 6 always adds 4 to EDI, even if the value searched for is found. Thus, if one wishes to find the address of the 12 found in the array, it is necessary to subtract 4 from EDI (as line 12 does).

6.2.4 The `REPx` instruction prefixes

There are several other `REP`-like instruction prefixes that can be used with the comparison string instructions. Figure 6.13 shows the two new prefixes and describes their operation. `REPE` and `REPZ` are just synonyms for the same prefix (as are `REPNE` and `REPNZ`). If the repeated comparison string instruction stops because of the result of the comparison, the index register or registers are still incremented and ECX decremented; however, the FLAGS register still holds the state that terminated the repetition. Thus, it is possible to use the Z flag to determine if the repeated comparisons stopped because of a comparison or ECX becoming zero.

Why can one not just look to see if ECX is zero after the repeated comparison?

Figure 6.14 shows an example code snippet that determines if two blocks

```
1     cld
2     mov   edi, offset array  ; pointer to start of array
3     mov   ecx, lengthof array ; number of elements
4     mov   eax, 12      ; number to scan for
5  lp:
6     scasd
7     je    found
8     loop  lp
9     mov dx, offset notFoundStr
10    jmp   onward
11 found:
12    sub edi, 4 ; edi now points to 12 in array
13    mov dx, offset foundStr
14 onward:
15    mov ah, 9
16    int 21h
```

Figure 6.12: Search example

REPE, REPZ	repeats instruction while Z flag is set or at most ECX times
REPNE, REPNZ	repeats instruction while Z flag is cleared or at most ECX times

Figure 6.13: REPx instruction prefixes

of memory are equal. The JE on line 6 of the example checks to see the result of the previous instruction. If the repeated comparison stopped because it found two unequal bytes, the Z flag will still be cleared and no branch is made; however, if the comparisons stopped because ECX became zero, the Z flag will still be set and the code branches to the equal label.

```
1     cld
2     mov   esi, offset block1 ; address of first block
3     mov   edi, offset block2 ; address of second block
4     mov   ecx, sizeof block1 ; size of blocks in bytes
5     repe  cmpsb              ; repeat while Z flag is set
6     je    equal              ; if Z set, blocks equal
7     ; code to perform if blocks are not equal
8     jmp   onward
9  equal:
10    ; code to perform if equal
11 onward:
```

Figure 6.14: Comparing memory blocks

Chapter 7

Structures

7.1 Structures

7.1.1 Introduction

Structures are used in C to group together related data into a composite variable. This technique has several advantages:

1. It clarifies the code by showing that the data defined in the structure are intimately related.

2. It simplifies passing the data to functions. Instead of passing multiple variables separately, they can be passed as a single unit.

3. It increases the *locality*[1] of the code.

From the assembly standpoint, a structure can be considered as an array with elements of *varying* size. The elements of real arrays are always the same size and type. This property is what allows one to calculate the address of any element by knowing the starting address of the array, the size of the elements and the desired element's index.

A structure's elements do not have to be the same size (and usually are not). Because of this each element of a structure must be explicitly specified and is given a *tag* (or name) instead of a numerical index.

In assembly, the element of a structure will be accessed in a similar way as in C. To access an element, you give the structure variable name then a dot then the field.

For example, consider the following structure shown in Figure 7.1.

Compare this to Figure 7.2, structures and structure variables are both declared in the data segment in assembly. For both, if we wanted to access

[1] See the virtual memory management section of any Operating System text book for discussion of this term.

```
struct exampleStruct {
  short int x=0;  /* 2-byte integer */
  int       y=1;  /* 4-byte integer */
  long int z=2;   /* 8-byte integer */
};

//down in main
exampleStruct s = {4,5,6};
```

Figure 7.1: Defining and declaring structs in C

```
.data
   exampleStruct struct
      x dw 1
      y dd 2
      z dq 3
   exampleStruct ends
   s exampleStruct <4,5,6>
```

Figure 7.2: Defining and declaring structs in Assembly

member y, we would use `s.y`. If you only wanted to specify values for x and z you can omit the y initializer and it will be assigned the default value: `s2 exampleStruct <7,,9>`.

7.1.2 Structure Example

Here is an example of a structure that holds user data:

```
title Struct Example     (structEx.asm)

.model small ; one data segment, one code segment
.stack 100h ; reserves 256 bytes for the stack
.386 ; for 32 bits
includelib printInc.lib ;include external subprocedures
extern printDec:proc
extern crlf:proc
.data ; start definition of variables
currentID dd 0c123h
userData struct
   fullName db 20 dup (?)
   idNum dd ?
userData ends
userTest userData <"Fake Name",>
```

```
16 userNew userData <>
17 prompt db "Please enter your name: ",'$'
18 replyName db "Your name is: ", '$'
19 replyID db "Your ID number is: ",'$'
20 .code ; start code portion of program
21 main proc ; start of the main procedure
22     mov  eax,@data ; load the address of the data segment into eax
23     mov  ds,eax ; load the address of the data segment into ds
24    mov  es,eax ; load es for string operations
25     ; the two previous instructions initalize the data segment
26    mov eax,currentID
27    mov userTest.idNum,eax ; give Fake Name an ID Number
28    inc currentID ; increment current ID
29     ; get new user set up
30    mov eax,currentID
31    mov userNew.idNum,eax
32    mov ecx,lengthof userNew.fullName
33    mov edi,offset userNew.fullName
34    ; ask for their name
35    mov dx,offset prompt
36    mov ah,9
37    int 21h
38    cld
39 nameLoop: ; read in their name
40    mov ah,1
41    int 21h
42    cmp al,0dh ; enter key
43    je printData
44    stosb
45    loop nameLoop
46 printData: ; print their data
47     mov al,'$'
48    stosb ; add a '$' to the end of their name
49    call crlf
50    mov dx,offset replyName
51    mov ah,9
52    int 21h
53    mov dx,offset userNew.fullName
54    mov ah,9
55    int 21h
56    call crlf
57    mov dx,offset replyID
58    mov ah,9
59    int 21h
60    mov eax,userNew.idNum
61    call printDec
62
63     ; the following two instructions exit cleanly from the program
```

```
64      mov  eax,4C00h ; 4C in ah means exit with code 0 (al) (similar
            to return 0; in C++)
65      int  21h ; exit
66  main endp ; end procedure
67
68  end main ; end program
```

Appendix A

80x86 Instructions

A.1 Non-floating Point Instructions

This section lists and describes the actions and formats of the non-floating point instructions of the Intel 80x86 CPU family.

The formats use the following abbreviations:

R	general register
R8	8-bit register
R16	16-bit register
R32	32-bit register
SR	segment register
M	memory
M8	byte
M16	word
M32	double word
I	immediate value

These can be combined for the multiple operand instructions. For example, the format *R, R* means that the instruction takes two register operands. Many of the two operand instructions allow the same operands. The abbreviation *O2* is used to represent these operands: *R,R R,M R,I M,R M,I*. If a 8-bit register or memory can be used for an operand, the abbreviation, *R/M8* is used.

The table also shows how various bits of the FLAGS register are affected by each instruction. If the column is blank, the corresponding bit is not affected at all. If the bit is always changed to a particular value, a 1 or 0 is shown in the column. If the bit is changed to a value that depends on the operands of the instruction, a *C* is placed in the column. Finally, if the bit is modified in some undefined way a *?* appears in the column. Because the

only instructions that change the direction flag are `CLD` and `STD`, it is not listed under the FLAGS columns.

Name	Description	Formats	Flags					
			O	**S**	**Z**	**A**	**P**	**C**
`ADC`	Add with Carry	O2	C	C	C	C	C	C
`ADD`	Add Integers	O2	C	C	C	C	C	C
`AND`	Bitwise AND	O2	0	C	C	?	C	0
`BSWAP`	Byte Swap	R32						
`CALL`	Call Routine	R M I						
`CBW`	Convert Byte to Word							
`CDQ`	Convert Dword to Qword							
`CLC`	Clear Carry							0
`CLD`	Clear Direction Flag							
`CMC`	Complement Carry							C
`CMP`	Compare Integers	O2	C	C	C	C	C	C
`CMPSB`	Compare Bytes		C	C	C	C	C	C
`CMPSW`	Compare Words		C	C	C	C	C	C
`CMPSD`	Compare Dwords		C	C	C	C	C	C
`CWD`	Convert Word to Dword into DX:AX							
`CWDE`	Convert Word to Dword into EAX							
`DEC`	Decrement Integer	R M	C	C	C	C	C	
`DIV`	Unsigned Divide	R M	?	?	?	?	?	?
`ENTER`	Make stack frame	I,0						
`IDIV`	Signed Divide	R M	?	?	?	?	?	?
`IMUL`	Signed Multiply	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
`INC`	Increment Integer	R M	C	C	C	C	C	
`INT`	Generate Interrupt	I						
`JA`	Jump Above	I						
`JAE`	Jump Above or Equal	I						
`JB`	Jump Below	I						
`JBE`	Jump Below or Equal	I						
`JC`	Jump Carry	I						

Name	Description	Formats	Flags					
			O	S	Z	A	P	C
JCXZ	Jump if CX = 0	I						
JE	Jump Equal	I						
JG	Jump Greater	I						
JGE	Jump Greater or Equal	I						
JL	Jump Less	I						
JLE	Jump Less or Equal	I						
JMP	Unconditional Jump	R M I						
JNA	Jump Not Above	I						
JNAE	Jump Not Above or Equal	I						
JNB	Jump Not Below	I						
JNBE	Jump Not Below or Equal	I						
JNC	Jump No Carry	I						
JNE	Jump Not Equal	I						
JNG	Jump Not Greater	I						
JNGE	Jump Not Greater or Equal	I						
JNL	Jump Not Less	I						
JNLE	Jump Not Less or Equal	I						
JNO	Jump No Overflow	I						
JNS	Jump No Sign	I						
JNZ	Jump Not Zero	I						
JO	Jump Overflow	I						
JPE	Jump Parity Even	I						
JPO	Jump Parity Odd	I						
JS	Jump Sign	I						
JZ	Jump Zero	I						
LAHF	Load FLAGS into AH							
LEA	Load Effective Address	R32,M						
LEAVE	Leave Stack Frame							
LODSB	Load Byte							
LODSW	Load Word							
LODSD	Load Dword							
LOOP	Loop	I						
LOOPE/LOOPZ	Loop If Equal	I						
LOOPNE/LOOPNZ	Loop If Not Equal	I						

Name	Description	Formats	Flags					
			O	S	Z	A	P	C
MOV	Move Data	O2						
		SR,R/M16						
		R/M16,SR						
MOVSB	Move Byte							
MOVSW	Move Word							
MOVSD	Move Dword							
MOVSX	Move Signed	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MOVZX	Move Unsigned	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MUL	Unsigned Multiply	R M	C	?	?	?	?	C
NEG	Negate	R M	C	C	C	C	C	C
NOP	No Operation							
NOT	1's Complement	R M						
OR	Bitwise OR	O2	0	C	C	?	C	0
POP	Pop From Stack	R/M16						
		R/M32						
POPA	Pop All							
POPF	Pop FLAGS		C	C	C	C	C	C
PUSH	Push to Stack	R/M16						
		R/M32 I						
PUSHA	Push All							
PUSHF	Push FLAGS							
RCL	Rotate Left with Carry	R/M,I	C					C
		R/M,CL						
RCR	Rotate Right with Carry	R/M,I	C					C
		R/M,CL						
REP	Repeat							
REPE/REPZ	Repeat If Equal							
REPNE/REPNZ	Repeat If Not Equal							
RET	Return							
ROL	Rotate Left	R/M,I	C					C
		R/M,CL						
ROR	Rotate Right	R/M,I	C					C
		R/M,CL						
SAHF	Copies AH into FLAGS			C	C	C	C	C

Name	Description	Formats	Flags					
			O	S	Z	A	P	C
SAL	Shifts to Left	R/M,I R/M, CL						C
SBB	Subtract with Borrow	O2	C	C	C	C	C	C
SCASB	Scan for Byte		C	C	C	C	C	C
SCASW	Scan for Word		C	C	C	C	C	C
SCASD	Scan for Dword		C	C	C	C	C	C
SETA	Set Above	R/M8						
SETAE	Set Above or Equal	R/M8						
SETB	Set Below	R/M8						
SETBE	Set Below or Equal	R/M8						
SETC	Set Carry	R/M8						
SETE	Set Equal	R/M8						
SETG	Set Greater	R/M8						
SETGE	Set Greater or Equal	R/M8						
SETL	Set Less	R/M8						
SETLE	Set Less or Equal	R/M8						
SETNA	Set Not Above	R/M8						
SETNAE	Set Not Above or Equal	R/M8						
SETNB	Set Not Below	R/M8						
SETNBE	Set Not Below or Equal	R/M8						
SETNC	Set No Carry	R/M8						
SETNE	Set Not Equal	R/M8						
SETNG	Set Not Greater	R/M8						
SETNGE	Set Not Greater or Equal	R/M8						
SETNL	Set Not Less	R/M8						
SETNLE	Set Not LEss or Equal	R/M8						
SETNO	Set No Overflow	R/M8						
SETNS	Set No Sign	R/M8						
SETNZ	Set Not Zero	R/M8						
SETO	Set Overflow	R/M8						
SETPE	Set Parity Even	R/M8						
SETPO	Set Parity Odd	R/M8						
SETS	Set Sign	R/M8						
SETZ	Set Zero	R/M8						
SAR	Arithmetic Shift to Right	R/M,I R/M, CL						C

Name	Description	Formats	Flags					
			O	S	Z	A	P	C
SHR	Logical Shift to Right	R/M,I						C
		R/M, CL						
SHL	Logical Shift to Left	R/M,I						C
		R/M, CL						
STC	Set Carry							1
STD	Set Direction Flag							
STOSB	Store Btye							
STOSW	Store Word							
STOSD	Store Dword							
SUB	Subtract	O2	C	C	C	C	C	C
TEST	Logical Compare	R/M,R	0	C	C	?	C	0
		R/M,I						
XCHG	Exchange	R/M,R						
		R,R/M						
XOR	Bitwise XOR	O2	0	C	C	?	C	0

A.2 Floating Point Instructions

In this section, many of the 80x86 math coprocessor instructions are described. The description section briefly describes the operation of the instruction. To save space, information about whether the instruction pops the stack is not given in the description.

The format column shows what type of operands can be used with each instruction. The following abbreviations are used:

Abbr.	Meaning
STn	A coprocessor register
F	Single precision number in memory
D	Double precision number in memory
E	Extended precision number in memory
I16	Integer word in memory
I32	Integer double word in memory
I64	Integer quad word in memory

Instructions requiring a Pentium Pro or better are marked with an asterisk(*).

Instruction	**Description**	**Format**
`FABS`	`ST0` = |`ST0`|	
`FADD` *src*	`ST0 +=` *src*	STn F D
`FADD` *dest*`, ST0`	*dest* `+= ST0`	STn
`FADDP` *dest*`[,ST0]`	*dest* `+= ST0`	STn
`FCHS`	`ST0` = $-$`ST0`	
`FCOM` *src*	Compare `ST0` and *src*	STn F D
`FCOMP` *src*	Compare `ST0` and *src*	STn F D
`FCOMPP` *src*	Compares `ST0` and `ST1`	
`FCOMI`* *src*	Compares into FLAGS	STn
`FCOMIP`* *src*	Compares into FLAGS	STn
`FDIV` *src*	`ST0 /=` *src*	STn F D
`FDIV` *dest*`, ST0`	*dest* `/= ST0`	STn
`FDIVP` *dest*`[,ST0]`	*dest* `/= ST0`	STn
`FDIVR` *src*	`ST0 =` *src*`/ST0`	STn F D
`FDIVR` *dest*`, ST0`	*dest* `= ST0/`*dest*	STn
`FDIVRP` *dest*`[,ST0]`	*dest* `= ST0/`*dest*	STn
`FFREE` *dest*	Marks as empty	STn
`FIADD` *src*	`ST0 +=` *src*	I16 I32
`FICOM` *src*	Compare `ST0` and *src*	I16 I32
`FICOMP` *src*	Compare `ST0` and *src*	I16 I32
`FIDIV` *src*	`ST0 /=` *src*	I16 I32

Instruction	Description	Format
FIDIVR *src*	ST0 = *src*/ST0	I16 I32
FILD *src*	Push *src* on Stack	I16 I32 I64
FIMUL *src*	ST0 *= *src*	I16 I32
FINIT	Initialize Coprocessor	
FIST *dest*	Store ST0	I16 I32
FISTP *dest*	Store ST0	I16 I32 I64
FISUB *src*	ST0 -= *src*	I16 I32
FISUBR *src*	ST0 = *src* - ST0	I16 I32
FLD *src*	Push *src* on Stack	STn F D E
FLD1	Push 1.0 on Stack	
FLDCW *src*	Load Control Word Register	I16
FLDPI	Push π on Stack	
FLDZ	Push 0.0 on Stack	
FMUL *src*	ST0 *= *src*	STn F D
FMUL *dest*, ST0	*dest* *= ST0	STn
FMULP *dest*[,ST0]	*dest* *= ST0	STn
FRNDINT	Round ST0	
FSCALE	ST0 = ST0 $\times 2^{\lfloor \text{ST1} \rfloor}$	
FSQRT	ST0 = $\sqrt{\text{ST0}}$	
FST *dest*	Store ST0	STn F D
FSTP *dest*	Store ST0	STn F D E
FSTCW *dest*	Store Control Word Register	I16
FSTSW *dest*	Store Status Word Register	I16 AX
FSUB *src*	ST0 -= *src*	STn F D
FSUB *dest*, ST0	*dest* -= ST0	STn
FSUBP *dest*[,ST0]	*dest* -= ST0	STn
FSUBR *src*	ST0 = *src*-ST0	STn F D
FSUBR *dest*, ST0	*dest* = ST0-*dest*	STn
FSUBP *dest*[,ST0]	*dest* = ST0-*dest*	STn
FTST	Compare ST0 with 0.0	
FXCH *dest*	Exchange ST0 and *dest*	STn

Appendix B

Interrupts

B.1 Common Interrupts

We can do a wide variety of useful system interactions using interrupts. There are too many to list, but below are some interesting examples (see Table B.1 [1]).

The format for the interrupts follow the same pattern. You load the values into the appropriate registers and then call the interrupt. Here is an example to output one character:

```
    ;print @ char
    mov ah,2    ; setting for outputting a char
    mov dl,'@'  ; dl = character going out
    int 21h     ; call the interrupt and output the character
```

You can output one character at a time or a whole string. You can also read a character from the keyboard, here is an example of both working together:

```
title Read key       (readkey.asm)

.model small ; one data segment, one code segment
.stack 100h ; reserves 256 bytes for the stack
.386 ; for 32 bits
.data
    greater5 db " is greater than 5",0ah,0dh,'$'
    lessthan5 db " is less than or equal to 5",0ah,0dh,'$'
.code ; start code portion of program
main proc ; start of the main procedure
    mov  eax,@data ; load the address of the data segment into ax
    mov  ds,eax ; load the address of the data segment into ds
    ; the two previous instructions initalize the data segment

```

[1] table originally from `http://www.skynet.ie/~darkstar/assembler/intlist.html`

```
15      mov ah,01h ; read from keyboard with echo
16      int 21h   ; interrupt to read from the keyboard, al=char
17
18      and al,0fh ; numbers 0-9 are ascii 30h-39h, so zero out the 3
19      ;sub al,30h ; this would work too
20      cmp al,5 ; if al>5
21      jbe elseBlock
22      mov dx,offset greater5 ; pointer to string in dx
23      jmp printMessage
24  elseBlock: ; al<=5
25      mov dx,offset lessthan5 ; pointer to string in dx
26  printMessage:
27      mov ah,9 ; output string
28      int 21h ; interrupt to print the string
29
30      ;the following two instructions exit cleanly from the program
31      mov  eax,4C00h ; exit (ah) with code 0 (al) (similar to return
            0; in C++)
32      int  21h ; exit
33  main endp ; end procedure
34
35  end main ; end program
```

Interrupt	**AH= (SubFunction)**	**Input**	**Output**
10h (VIDEO_INTERRUPT)	00 (SET_MODE) Sets the Video mode.	AL=mode number	-
	0Ch (WRITE_DOT) Puts a dot on the screen Graphics modes only	DX=row CX=column AL=color	-
	0Dh (READ_DOT) Reads a dot on screen Graphics modes only	DX=row CX=column	AL=color
16h (KBD_IO)	00 (AWAIT_CHAR) Reads a character from keyboard	-	AL=character in AH=scancode
	01 (PREVIEW_KEY) Checks to see if a key is ready, but does not remove key from buffer.	-	Zero flag = key ready AL=character in AH=scancode
21h (DOS_INTERRUPT)	01 (KEYBOARD_INPUT) Reads and displays one character	-	AL=character in
	02 (DISPLAY_OUTPUT) Displays one character on screen	DL=character out	-
	08 (NO_ECHO_INPUT) Same as 01 but not displayed	-	AL=character in
	09 (PRINT_STRING) Displays a string on screen String must end with "$"	DX=address of output string	-
	4Ch (EXIT)	AL=exit code	-

Figure B.1: Common Interrupts

Index