CHAPTER SEVENTEEN

Intel 80x86 Assembly Language

In Chapter 15, we developed a generic assembly language and its associated machine code. This language was presented to create a few simple programs and present how the CPU executed code. In this chapter, the assembly language of the Intel 80x86 processor family is introduced along with the typical syntax for writing 80x86 assembly language programs. This information is then used to write a sample program for the 80x86 processor.

This chapter is meant to serve as an introduction to programming the Intel 80x86 using assembly language. For more detailed instruction, refer to one of the resources listed at the end of this chapter.

17.1 Assemblers versus Compilers

For a high-level programming language such as C, there is a two-step process to produce an application from source code. To begin with, a program called a ***compiler*** takes the source code and converts it into machine language instructions. This is a complex task that requires a detailed understanding of the architecture of the processor. The compiler outputs the resulting sequence of machine code instructions to a file called an ***object file***. The second step takes one or more object files and combines them by merging addressing information and generating necessary support code to make the final unit operate as an application. The program that does this is called a ***linker***.

In order for the linker to operate properly, the object files must follow certain rules for format and addressing to clearly show how one object file interrelates with the others.

A similar two-step process is used to convert assembly language source code into an application. It begins with a program called an ***assembler***. The assembler takes an assembly language program, and using a one-to-one conversion process, converts each line of assembly language to a single machine code instruction. Because of this one-to-one relation between assembly language instructions and machine code instructions, the assembly language programmer must have a clear understanding of how the processor will execute the machine code. In other words, the programmer must take the place of the compiler by converting abstract processes to the step-by-step processor instructions.

As with the compiler, the output of the assembler is an object file. The format and addressing information of the assembler's object file should mimic that of the compiler making it possible for the same linker to be used to generate the final application. This means that as long as the assembly language programmer follows certain rules when identifying shared addressing, the object file from an assembler should be capable of being linked to the object files of a high-level language compiler.

The format of an assembly language program depends on the assembler being used. There are, however, some general formatting patterns that are typically followed. This section presents some of those standards.

Like most programming languages, assembly language source code must follow a well-defined syntax and structure. Unlike most programming languages, the lines of assembly language are not structurally interrelated. In a language such as C, for example, components such as functions, if-statements, loops, and switch/case blocks utilize syntax to indicate the beginning and end of a block of code that is to be treated as a unit. Blocks of code may be contained within larger blocks of code producing a hierarchy of execution. In assembly language, there is no syntax to define blocks of code; formatting only applies to a single line of code. It is the execution of the code itself that is used to logically define blocks within the program.

17.2 Components of a Line of Assembly Language

As shown in Figure 17-1, a line of assembly language code has four fields: a label, an opcode, a set of operands, and comments. Each of these fields must be separated by horizontal white space, i.e., spaces or tabs. No carriage returns are allowed as they identify the beginning of a new line of code. Depending on the function of a particular line, one or more of the fields may be omitted.

The first field of a line is an optional ***label field.*** A label is used to identify a specific line of code or the memory location of a piece of data so that it may be referenced by other lines of assembly language. The assembler will translate the label into an address for use in the object file. As far as the programmer is concerned, however, the label may be used any time an address reference is needed to that particular line. It is not necessary to label all lines of assembly language code, only the ones that are referred to by other lines of code.

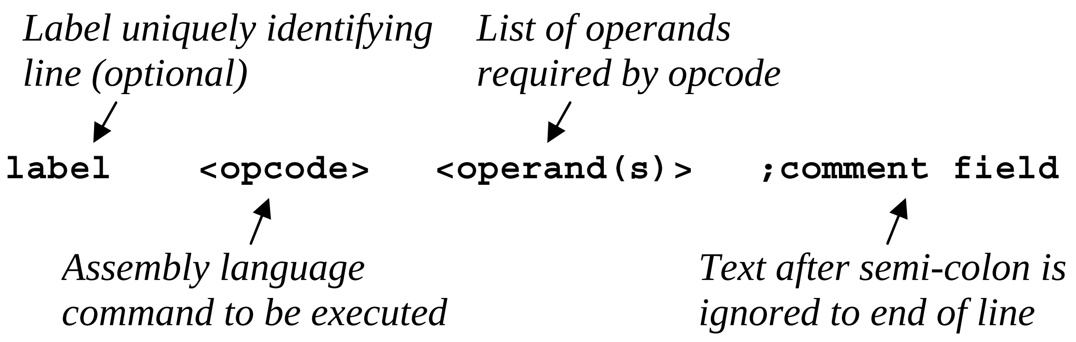
Label uniquely identifying line (optional)
List of operands required by opcode
label <opcode> <operand(s)> ;comment field
Assembly language command to be executed
Text after semi-colon is ignored to end of line

Figure 17-1 Format of a Line of Assembly Language Code

A label is a text string much like a variable name in a high-level language. There are some rules to be obeyed when defining a label.

- Labels must begin in the first column with an alphabetic character. Subsequent characters may be numeric.
- It must not be a reserved string, i.e., it cannot be an assembly language instruction nor can it be a command to the assembler.
- Although a label may be referenced by other lines of assembly language, it cannot be reused to identify a second line of code within the same file.
- In some cases, a special format for a label may be required if the label's function goes beyond identification of a line within a file. A special format may be needed, for example, if a high-level programming language will be referencing one of the assembly language program's functions.

The next field is the ***instruction*** or ***opcode field***. The instruction field contains the assembly language command that the processor is supposed to execute for this line of code. An instruction must be either an assembly language instruction (an opcode) or an instruction to the assembler (an assembler directive).

The third field is the ***operand field***. The operand field contains the data or operands that the assembly language instruction needs for its execution. This includes items such as memory addresses, constants, or register names. Depending on the instruction, there may be zero, one, two, or three operands, the syntax and organization of which also depends on the instruction.

The last field in a line of assembly language is the ***comment field***. As was mentioned earlier, assembly language has no structure in the syntax to represent blocks of code. Although the specific operation of a line of assembly language should be clear to a programmer, its purpose within the program usually is not. It is therefore imperative to comment assembly language programs. In addition to the standard use of comments, comments in assembly language can be used to:

- show where functions or blocks of code begin and end;
- explain the order or selection of commands (e.g., where a shift left has replaced a multiplication by a power of two); or
- identify obscure values (e.g., that address 0378_{16} represents the data registers of the parallel port).

A comment is identified with a preceding semi-colon, ';'. All text from the semi-colon to the end of the line is ignored. This is much like the double-slash, "//", used in C++ or the quote used in Visual Basic to comment out the remaining text of a line. A comment may be alone in a line or it may follow the last necessary field of a line of code.

17.3 Assembly Language Directives

There are exceptions in an assembly language program to the opcode/operand lines described in the previous section. One of the primary exceptions is the ***assembler directive***. Assembler directives are instructions to the assembler or the linker indicating how the program should be created. Although they have the same format as an assembly language instruction, they do not translate to object code. This section will only address a few of the available directives. Please refer to one of the resources listed at the end of this chapter for more information on the assembler directives used with the Intel 80x86.

17.3.1 SEGMENT Directive

One of the most important directives with respect to the final addressing and organization of the application is **SEGMENT**. This directive is used to define the characteristics and or contents of a segment. (See Chapter 16 for a description of segments and their use with the 80x86 processor.)

There are three main segments: the code segment, the data segment, and the stack segment. To define these segments, the assembly language file is divided into areas using the SEGMENT directive. The beginning of the segment is defined with the keyword SEGMENT while its end is defined using the keyword ENDS. Figure 17-2 presents the format and parameters used to define a segment.

```
label     SEGMENT     alignment   combine    'class'
              .           .
              .           .
              .           .
label     ENDS
```

Figure 17-2 Format and Parameters Used to Define a Segment

The ***label*** uniquely identifies the segment. The SEGMENT directive label must match the corresponding ENDS directive label.

The ***alignment*** attribute indicates the "multiple" of the starting address for the segment. For a number of reasons, either the processor or the operating system may require that a segment begin on an address that is divisible by a certain power of two. The align attribute is used to tell the assembler what multiple of a power of two is required. The following is a list of the available settings for alignment.

- BYTE – There is no restriction on the starting address.
- WORD – The starting address must be even, i.e., the binary address must end in a zero.
- DWORD – The starting address must be divisible by four, i.e., the binary address must end in two zeros.
- PARA – The starting address must be divisible by 16, i.e., the binary address must end in four zeros.
- PAGE – The starting address must be divisible by 256, i.e., the binary address must end in eight zeros.

The ***combine*** attribute is used to tell the linker if segments can be combined with other segments. The following is a list of a few of the available settings for the combine attribute.

- NONE – The segment is to be located independently of the other segments and is logically considered separate.
- PUBLIC or COMMON – The segment may be combined with other segments of the same name and class.
- STACK – Works like PUBLIC for stack segments.

The ***class*** attribute helps the assembler classify the information contained in the segment. This is important in order to organize the data, code, and other information that the linker will be partitioning into segments when it comes time to create the final application. Typical values are 'Data', 'Code', or 'Stack'. Note that the apostrophes are to be included as part of the attribute value.

17.3.2 .MODEL, .STACK, .DATA, and .CODE Directives

Instead of going to the trouble of defining the segments with the SEGMENT directive, a programmer may select a memory model. By defining the memory model for the program, a basic set of segment definitions is assumed. The directive **.MODEL** can do this. Figure 17-3 presents the format of the .MODEL directive.

```
.MODEL    memory_model
```

Figure 17-3 Format of the .MODEL Directive

Table 17-1 presents the different types of memory models that can be used with the directive. The memory models LARGE and HUGE are the same except that HUGE may contain single variables that use more than 64K of memory.

There are three more directives that can be used to simplify the definition of the segments. They are **.STACK**, **.DATA**, and **.CODE**. When the assembler encounters one of these directives, it assumes that it is the beginning of a new segment, the type being defined by the specific directive used (stack, data, or code). It includes everything that follows the directive in the same segment until a different segment directive is encountered.

The .STACK directive takes an integer as its operand allowing the programmer to define the size of the segment reserved for the stack.

The .CODE segment takes a label as its operand indicating the segment's name.

Table 17-1 Memory Models Available for use with .MODEL

Memory Model	Segment Definitions
TINY	Code, data, and, stack in one 64K segment
SMALL	One code segment less than or equal to 64K One data segment less than or equal to 64K
MEDIUM	Multiple code segments of any size One data segment less than or equal to 64K
COMPACT	One code segment less than or equal to 64K Multiple data segments of any size
LARGE	Multiple code segments of any size Multiple data segments of any size
HUGE	Multiple code segments of any size Multiple data segments of any size
FLAT	One 4 Gig memory space

17.3.3 PROC Directive

The next directive, **PROC**, is used to define the beginning of a block of code within a code segment. It is paired with the directive **ENDP** which defines the end of the block. The code defined between PROC and ENDP should be treated like a procedure or a function of a high-level language. This means that jumping from one block of code to another is done by calling it like a procedure.

```
label    PROC   NEAR or FAR
  .        .
  .        .
  .        .
label    ENDP
```

Figure 17-4 Format and Parameters Used to Define a Procedure

As with the SEGMENT directive, the labels for the PROC directive and the ENDP directive must match. The attribute for PROC is either

NEAR or FAR. A procedure that has been defined as NEAR uses only an offset within the segment for addressing. Procedures defined as FAR need both the segment and offset for addressing.

17.3.4 END Directive

Another directive, **END**, is used to tell the assembler when it has reached the end of ***all*** of the code. Unlike the directive pairs SEGMENT and ENDS and PROC and ENDP, there is no corresponding directive to indicate the beginning of the code.

17.3.5 Data Definition Directives

The previous directives are used to tell the assembler how to organize the code and data. The next class of directives is used to define entities that the assembler will convert directly to components to be used by the code. They do not represent code; rather they are used to define data or constants on which the application will operate.

Many of these directives use integers as their operands. As an aid to programmers, the assembler allows these integers to be defined in binary, decimal, or hexadecimal. Without some indication as to their base, however, some values could be interpreted as hex, decimal, or binary (e.g., 100). Hexadecimal values have an 'H' appended to the end of the number, binary values have a 'B' appended to the end, and decimal values are left without any suffix.

Note also that the first digit of any number must be a numeric digit. Any value beginning with a letter will be interpreted by the assembler as a label instead of a number. This means that when using hexadecimal values, a leading zero must be placed in front of any number that begins with A, B, C, D, E, or F.

The first of the defining directives is actually a set of directives used for reserving and initializing memory. These directives are used to reserve memory space to hold elements of data that will be used by the application. These memory spaces may either be initialized or left undefined, but their size will always be specified.

The primary form of these directives is **Dx** where a character is substituted for the 'x' to indicate the incremental size of memory that is being reserved. For example, a single byte can be reserved using the directive DB. Figure 17-5 presents some of the define directives and their format.

```
label   DB   expression    ;define a byte
label   DW   expression    ;define a word (2 bytes)
label   DD   expression    ;define a double word
label   DQ   expression    ;define a quad word
```

Figure 17-5 Format and Parameters of Some Define Directives

The label, which is to follow the formatting guidelines of the label field defined earlier, is not required. When it is used, the assembler assigns it the address corresponding to the next element of memory being reserved. The programmer may then use it throughout their code to refer back to that address.

The expression after the directive is required. The expression is used to tell the assembler how much memory is to be reserved and if it is to be initialized. There are four primary formats for the expression.

- Constants – The expression can be a list of one or more constants. These constants will be converted to binary and stored in the order that they were defined.
- String – The expression can be a string. The assembler will divide the string into its characters and store each character in the incremental space required by the selected define directive, i.e., DB reserves memory a byte at a time, DW reserves memory a word at a time, DD reserves memory a double word at a time, and DQ reserves memory a quad word at a time.
- Undefined – A question mark (?) can be used to tell the assembler that the memory is to be reserved, but left undefined.
- Duplicated elements – The keyword DUP may be used to replicate the same value in order to fill a block of memory.

Figure 17-6 presents some examples of the define directives where the comment field is used to describe what will be stored in the reserved memory.

17.3.6 EQU Directive

The next directive, **EQU**, is in the same class as the define directives. It is like the #define directive used in C, and like #define, it is used to define strings or constants to be used during assembly. The format of the EQU directive is shown in Figure 17-7.

```
VAR01   DB  23H       ;Reserve byte/initialized to
                      ;hexadecimal 23
VAR01   DB  10010110B ;Reserve byte/initialized to
                      ;binary 10010110
VAR02   DB  ?         ;Reserve byte/undefined
STR01   DB  'hello'   ;Store 'h', 'e', 'l', 'l',
                      ;and 'o' in 5 sequential bytes
ARR01   DB  3, 2, 6   'Store the numbers 3, 2, and 6
                      ;in 3 sequential bytes
ARR02   DB  4 DUP(?)  ;Reserve 4 bytes/undefined
ARR03   DW  4 DUP(0)  ;Reserve 4 words (8 bytes) and
                      ;initialize to 0
```

Figure 17-6 Example Uses of Define Directives

```
            label    EQU    expression
```

Figure 17-7 Format and Parameters of the EQU Directive

Both the label and the expression are required fields with the EQU directive. The label, which also is to follow the formatting guidelines of the label field, is made equivalent to the expression. This means that whenever the assembler comes across the label later in the file, the expression is substituted for it. Figure 17-8 presents two sections of code that are equivalent because of the use of the EQU directive.

```
ARRAY    DB     12 DUP(?)
```

a.) Reserving 12 bytes of memory without EQU directive

```
COUNT    EQU    12
ARRAY    DB     COUNT DUP(?)
```

b.) Reserving 12 bytes of memory using EQU directive

Figure 17-8 Sample Code with and without the EQU Directive

Note that EQU only assigns an expression to a name at the time of assembly. No data segment storage area is allocated with this directive.

17.4 80x86 Opcodes

Assembly language instructions can be categorized into four groups: data transfer, data manipulation, program control, and special operations. The next four sections introduce some of the Intel 80x86 instructions by describing their function.

17.4.1 Data Transfer

There is one Intel 80x86 opcode that is used to move data: **MOV**. As shown in Figure 17-9, the MOV opcode takes two operands, *dest* and *src*. MOV copies the value specified by the *src* operand to the memory or register specified by *dest*.

```
MOV     dest, src
```

Figure 17-9 Format and Parameters of the MOV Opcode

Both *dest* and *src* may refer to registers or memory locations. The operand *src* may also specify a constant. These operands may be of either byte or word length, but regardless of what they are specifying, the sizes of *src* and *dest* must match for a single MOV opcode. The assembler will generate an error if they do not.

Section 16.4 showed how the Intel 80x86 uses separate control lines for transferring data to and from its I/O ports. To do this, it uses a pair of special data transfer opcodes: **IN** and **OUT**. The opcode IN reads data from an I/O port address placing the result in either AL or AX depending on whether a byte or a word is being read. The OUT opcode writes data from AL or AX to an I/O port address. Figure 17-10 shows the format of these two instructions using the operand *accum* to identify either AL or AX and *port* to identify the I/O port address of the device.

```
IN      accum, port
OUT     port, accum
```

Figure 17-10 Format and Parameters of the IN and OUT Opcodes

None of the data transfer opcodes modifies the processor's flags.

17.4.2 Data Manipulation

Intel designed the 80x86 family of processors with plenty of instructions to manipulate data. Most of these instructions have two operands, *dest* and *src*, and just like the MOV instruction, they read from *src* and store in *dest*. The difference is that the *src* and *dest* values are combined somehow before being stored in *dest*. Another difference is that the data manipulation opcodes typically affect the flags.

Take for example the **ADD** opcode shown in Figure 17-11. It reads the data identified by *src*, adds it to the data identified by *dest*, then replaces the original contents of *dest* with the result.

```
ADD    dest, src
```

Figure 17-11 Format and Parameters of the ADD Opcode

The ADD opcode modifies the processor's flags including the carry flag (CF), the overflow flag (OF), the sign flag (SF), and the zero flag (ZF). This means that any of the Intel 80x86 conditional jumps can be used after an ADD opcode for program flow control.

Many of the other data manipulation opcodes operate the same way. These include logic operations such as **AND**, **OR**, and **XOR** and mathematical operations such as **SUB** (subtraction) and **ADC** (add with carry). **MUL** (multiplication) and DIV (division) are different in that they each use a single operand, but since two pieces of data are needed to perform these operations, the AX or AL registers are implied.

Some operations by nature only require a single piece of data. For example, **NEG** takes the 2's-complement of a value and stores it back in the same location. The same is true for **NOT** (bit-wise inverse), **DEC** (decrement), and **INC** (increment). These commands all use a single operand identified as *dest*.

```
NEG    dest    ;Take 2's complement of dest
NOT    dest    ;Invert each of the bits of dest
DEC    dest    ;Subtract 1 from dest
INC    dest    ;Add 1 to dest
```

Figure 17-12 Format and Parameters of NEG, NOT, DEC, and INC

As with most processors, the Intel 80x86 processor has a group of opcodes that are used to shift data. There are two ways to classify shift instructions: left versus right and arithmetic versus logical. The area where these classifications are of greatest concern is with a right shift.

Remember from Chapter 3 that left and right shifts are equivalent to multiplication and division by powers of two. When using a right shift to perform a division, the most significant bit must be replicated or the sign of a two's complement value might change from negative to positive. Therefore, if it is important to maintain the sign of a right-shifted value, an arithmetic shift right (**SAR**) should be used, not a logical shift right (**SHR**). Since a left shift doesn't have this constraint, an arithmetic shift left (**SAL**) and logical shift left (**SHL**) perform the same operation and are even identified with the same machine code.

All four of the shift commands use two operands. The first operand, *dest*, contains the data to be shifted. It is also the location where the result will be stored. The second operand, *count*, indicates the number of bit positions the piece of data will be shifted.

```
SAR    dest, count   ;Arithmetic shift right
SHR    dest, count   ;Logical shift right
SAL    dest, count   ;Arithmetic shift left
SHL    dest, count   ;Logical shift left
```

Figure 17-13 Format and Parameters of SAR, SHR, SAL, and SHL

17.4.3 Program Control

As with the generic processor described in Chapter 15, the 80x86 uses both unconditional and conditional jumps to alter the sequence of instruction execution. When the processor encounters an unconditional jump or "jump always" instruction (**JMP**), it loads the instruction pointer with the address that serves as the JMP's operand. This makes it so that the next instruction to be executed is at the newly loaded address. Figure 17-14 presents an example of the JMP instruction.

```
         JMP  LBL01   ;Always jump to LAB01
          :      :
LBL01:                ;Destination for jump
```

Figure 17-14 Example of a JMP Instruction

The 80x86 has a full set of conditional jumps to provide program control based on the results of execution. Each conditional jump examines the flags before determining whether to load the jump opcode's operand into the instruction pointer or simply move to the next sequential instruction. Table 17-2 presents a summary of most of the 80x86 conditional jumps along with the flag settings that force a jump. (Note that "!=" means "is not equal to")

Table 17-2 Summary of 80x86 Conditional Jumps

Mnemonic	Meaning	Jump Condition
JA	Jump if Above	CF=0 and ZF=0
JAE	Jump if Above or Equal	CF=0
JB	Jump if Below	CF=1
JBE	Jump if Below or Equal	CF=1 or ZF=1
JC	Jump if Carry	CF=1
JE	Jump if Equal	ZF=1
JG	Jump if Greater (signed)	ZF=0 and SF=OF
JGE	Jump if Greater or Equal (signed)	SF=OF
JL	Jump if Less (signed)	SF != OF
JLE	Jump if Less or Equal (signed)	ZF=1 or SF != OF
JNA	Jump if Not Above	CF=1 or ZF=1
JNAE	Jump if Not Above or Equal	CF=1
JNB	Jump if Not Below	CF=0
JNBE	Jump if Not Below or Equal	CF=0 and ZF=0
JNC	Jump if Not Carry	CF=0
JNE	Jump if Not Equal	ZF=0
JNG	Jump if Not Greater (signed)	ZF=1 or SF != OF
JNGE	Jump if Not Greater or Equal (signed)	SF != OF
JNL	Jump if Not Less (signed)	SF=OF
JNLE	Jump if Not Less or Equal (signed)	ZF=0 and SF=OF
JNO	Jump if No Overflow	OF=0
JNS	Jump if Not Signed (signed)	SF=0
JNZ	Jump if Not Zero	ZF=0
JO	Jump if Overflow	OF=1
JPE	Jump if Even Parity	PF=1
JPO	Jump if Odd Parity	PF=0
JS	Jump if Signed (signed)	SF=1
JZ	Jump if Zero	ZF=1

Typically, these conditional jumps come immediately after a compare. In the Intel 80x86 instruction set, the compare function is **CMP**. It uses two operands, setting the flags by subtracting the second operand from the first. Note that the result is not stored.

The 80x86 provides an additional instruction over that of the generic processor discussed in Chapter 15. The **LOOP** instruction was added to support the operation of a for- or a while-loop. It takes as its only operand the address of the first instruction of the loop.

Before entering the loop, the CX register is loaded with a count of the number of times the loop is to be executed. Each time the LOOP opcode is executed, CX is decremented. As long as CX has not yet been decremented to zero, the instruction pointer is set back to the first instruction of the loop, i.e., the address given as the operand of the LOOP instruction. When CX has been decremented to zero, the LOOP instruction does not return to the beginning of the loop; instead, it goes to the instruction after LOOP. Figure 17-15 presents an example where the LOOP instruction executes a loop 25 times.

```
         MOV  CX,25   ;Load CX with the integer 25
LBL02:                ;Beginning of loop
          :     :
         LOOP LBL02   ;Decrement CX and jump to
                      ; LBL02 as long as CX!=0
```

Figure 17-15 Example of a LOOP Instruction

There is one last set of instructions used to control the flow of the program, and although they were not mentioned in Chapter 15, they are common to all processors. These instructions are used to call and return from a procedure or function.

The **CALL** opcode is used to call a procedure. It uses the stack to store the address of the instruction immediately after the CALL opcode. This address is referred to as the ***return address***. This is the address that the processor will jump back to after the procedure is complete.

The CALL instruction takes as its operand the address of the procedure that it is calling. After the return address is stored to the stack, the address of the procedure is loaded into the instruction pointer.

To return from a procedure, the instruction **RET** is executed. The only function of the RET instruction is to pull the return address from the stack and load it into the instruction pointer. This brings control back to the original sequence. Figure 17-16 presents an example of the organization of a procedure call using the CALL and RET instructions.

```
         CALL PROC01  ;Procedure call to PROC01
         xxx          ;Instruction that is returned
                      ; to after procedure is called
           :     :
PROC01:               ;Beginning of procedure
           :     :
         RET          ;Return to instruction after
                      ; CALL
```

Figure 17-16 Sample Organization of a Procedure Call

17.4.4 Special Operations

The special operations category is for opcodes that do not fit into any of the first three categories, but are necessary to fully utilize the processor's resources. They provide functionality ranging from controlling the processor flags to supporting the 80x86 interrupt system.

To begin with, there are seven instructions that allow the user to manually alter the flags. These are presented in Table 17-3.

Table 17-3 80x86 Instructions for Modifying Flags

Mnemonic	Meaning
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CLI	Clear Interrupt Flag (disables maskable interrupts)
CMC	Complement Carry Flag
STC	Set Carry Flag
STD	Set Direction Flag
STI	Set Interrupt Flag (enables maskable interrupts)

The next two special instructions are **PUSH** and **PULL**. These instructions operate just as they are described in chapters 15 and 16. The Intel 80x86 processor's stack is referred to as a post-increment/ pre-decrement stack. This means that the address in the stack pointer is decremented before data is stored to the stack and incremented after data is retrieved from the stack.

There are also some special instructions that are used to support the operation of the Intel 80x86 interrupts. **IRET**, for example, is the instruction used to return from an interrupt service routine. It is used in the same manner as the RET instruction in a procedure. IRET, however, is required for interrupts because an interrupt on the 80x86 pushes not only the return address onto the stack, but also the code segment and processor flags. IRET is needed to pull these two additional elements off of the stack before returning to the code being executed before the interrupt.

Another special instruction is the software interrupt, **INT**. It is a non-maskable interrupt that calls an interrupt routine just like any hardware interrupt. In a standard PC BIOS, this interrupt has a full array of functions ranging from keyboard input and video output to file storage and retrieval.

The last instruction presented here may not make sense to the novice assembly language programmer. The **NOP** instruction has no operation and it does not affect any flags. Typically, it is used to delete a machine code by replacing it and its operands with this non-executing opcode. In addition, a sequence of NOPs can be inserted to allow a programmer to write over them later with new machine code. This is only necessary under special circumstances.

17.5 Addressing Modes

The previous section described the 80x86 opcodes and their operands. This section shows the format that the programmer needs to use to properly identify the operands. Specifically, the assembler needs to know whether the programmer is referring to a register, a constant, or a memory address. Special syntax is used to do just that.

17.5.1 Register Addressing

To identify a register as an operand, simply use the name of the register for either *src* or *dest*. For the 80x86 architecture described in Chapter 16, these registers include both the 8- and 16-bit general purpose registers (AX, BX, CX, DX, AL, AH, BL, BH, CL, CH, DL, and DH), the address registers (SP, BP, DI, SI, and IP), and the segment registers (CS, DS, SS, and ES). Figure 17-17 presents some examples of instructions using register addressing.

```
MOV  AL,BL    ;Copy the contents of BL to AL
CMP  BX,CX    ;Compare the contents of CX to
              ; the contents of BX
INC  DX       ;Increment the contents of DX
```

Figure 17-17 Examples of Register Addressing

17.5.2 Immediate Addressing

The use of a constant as an operand is referred to as immediate addressing. In this case, a constant is used instead of a stored value such as that retrieved from a register or memory. As with the directives used to define constants in memory, hex, decimal, and binary values must be identified by appending an 'H' to the end of a hexadecimal number, appending a 'B' to the end of a binary number, and leaving the decimal values without any suffix.

Because of the nature of constants, they can only be used as the *src* operand. They reserve no space in the data segment, and therefore cannot have data stored to them. Figure 17-18 presents some examples of instructions using immediate addressing.

```
MOV  AX,67D1H     ;Place the hex value 67D1 in AX
CMP  BL,01101011B ;Compare the contents of BL to
                  ; the binary value 01101011
ADD  CX,9         ;Add decimal 9 to CX
```

Figure 17-18 Examples of Immediate Addressing

17.5.3 Pointer Addressing

It might be misleading not to distinguish between the six different forms used to identify an address as an operand. This chapter, however, is only an introduction to assembly language. At this point it is sufficient to say that an operand is identified as an address by surrounding it with brackets []. For example, to make a reference to hexadecimal address 1000, the operand would be identified as [1000H].

Although the data segment identified by DS is the default segment when using an address as an operand, the segment may still be specified within this notation. By using a colon to separate the segment from the offset, any segment may be used. For example, to ensure that the address 1000 was coming from the data segment, the operand would be identified as [DS:1000H].

The processor can also use the contents of a register as a pointer to an address. In this case, the register name is enclosed in brackets to identify it as a pointer. For example, if the contents of BX are being used as an address pointing to memory, the operand should be entered as [BX] or [DS:BX].

A constant offset can be added to the pointer if necessary by adding a constant within the brackets. For example, if the address of interest is 4 memory locations past the address pointed to by the contents of BX, the operand should be entered as [BX+4] or [DS:BX+4].

While this is not a comprehensive list of the methods for using a memory address as an operand, it should be a sufficient introduction. Figure 17-19 presents some examples of using addresses for operands.

```
MOV  AX,[6000H]  ;Load AX w/data from address 6000H
MOV  AX,[BX]     ;Load AX w/data pointed to by the
                 ; address contained in BX
MOV  AX,[BX+4]   ;Load AX w/data 4 memory locations
                 ; past address pointed to by BX
```

Figure 17-19 Examples of an Address being used as an Operand

17.6 Sample 80x86 Assembly Language Programs

Now we need to tie the concepts of assembly language presented in Chapter 15 to the specifics of the 80x86 assembly language. The best way to do this is to create a simple program. We begin with the general framework used to support the program. Figure 17-20 presents the basic skeleton code of an 80x86 assembly language program.

```
        .MODEL SMALL
        .STACK 100H
        .DATA
        .CODE
MAIN    PROC    FAR

MAIN    ENDP
        END     MAIN
```

Figure 17-20 Skeleton Code for a Simple Assembly Program

Let's examine this code line-by-line.

- The first line contains the string ".MODEL SMALL". We see from Table 17-1 that this tells the compiler to use one code segment less than or equal to 64K and one data segment less than or equal to 64K. The program we are writing here is quite small and will easily fit in this memory model.
- The next line, ".STACK 100H", tells the instructor to reserve 256 bytes (hexadecimal 100) for the stack.
- The next line, ".DATA", denotes the beginning of the data segment. All of the data for the application will be defined between the .DATA and .CODE directives.
- The next line, ".CODE", denotes the beginning of the code segment. All of the code will be defined after this directive.
- "MAIN PROC FAR" identifies a block of code named main that will use both the segment and offset for addressing.
- "MAIN ENDP" identifies the end of the block of code named MAIN.
- "END MAIN" tells the assembler when it has reached the end of all of the code.

The next step is to insert the data definitions and code that go after the .DATA and .CODE directives respectively.

The first piece of code we need to write will handle some operating system house keeping. First, we need to start the program by retrieving the address that the operating system has assigned to the data segment. This value needs to be copied to the DS register. We do this with the two lines of code presented in Figure 17-21. These lines need to be placed immediately after the MAIN PROC FAR line.

```
MOV  AX,@DATA   ;Get assigned data segment
                ; address from O/S
MOV  DS,AX      ;Copy it to the DS register
```

Figure 17-21 Code to Assign Data Segment Address to DS Register

When the program ends, we need to transfer control back to the operating system. This is done using a software interrupt. At this point it is not necessary to understand this process other than to say that when the O/S receives this interrupt, it knows that the application is finished and can be removed from memory. Placing the lines from Figure 17-22 immediately before the line MAIN ENDP in the code will do this.

```
MOV  AX,4C00H   ;Load code indicating normal
                ; program termination
INT  21H        ;Call interrupt to end program
```

Figure 17-22 Code to Inform O/S that Program is Terminated

At this point, our skeleton code should look like that shown in Figure 17-23.

```
        .MODEL SMALL
        .STACK 100H
        .DATA
        .CODE
MAIN    PROC    FAR
        MOV     AX,@DATA    ;Load DS with assigned
        MOV     DS,AX       ; data segment address


        MOV  AX,4C00H       ;Use software interrupt
        INT     21H         ; to terminate program
MAIN    ENDP
        END     MAIN
```

Figure 17-23 Skeleton Code with Code Added for O/S Support

Now all we need is a program to write. The program presented here is a simple mathematical calculation using data from the data segment. Specifically, we will be calculating the following algebraic expression where A, B, C, and RESULT are defined to be 16-bit words in the data segment.

$$\text{RESULT} = (A \div 8) + B - C$$

Let's begin by defining what the data segment is going to look like. Each of the variables, A, B, C, and RESULT, need to have a word-sized location reserved in memory for them. Since the first three will be used as inputs to the expression, they will also need to be initialized.

For the sake of this example, let's initialize them to 104_{10}, 100_{10}, and 52_{10} respectively. Since RESULT is where the calculated result will be stored, we may leave that location undefined. Figure 17-24 presents the four lines of directives used to define this memory.

```
A         DW    104
B         DW    100
C         DW    52
RESULT    DW    ?
```

Figure 17-24 Data Defining Directives for Example Code

This code will be inserted between the .DATA and .CODE directives of the code in Figure 17-23.

The next step is to write the code to compute the expression. Begin by assuming the computation will occur in the accumulator register, AX. The process will go something like this.

- Load AX with value stored at the memory location identified by A.
- Divide AX by eight using the arithmetic right shift instruction.
- After dividing AX, add the value stored at the memory location identified by B.
- After adding B to AX, subtract the value stored at the memory location identified by C.
- Lastly, store the result contained in AX to the memory location RESULT.

Converting this step-by-step sequence into assembly language results in the code presented in Figure 17-25.

```
MOV  AX,A        ;Load A from memory
SAR  AX,3        ;Divide A by 8
ADD  AX,B        ;Add B to (A/8)
SUB  AX,C        ;Subtract C from (A/8)+B
MOV  RESULT,AX   ;Store (A/8)+B-C to RESULT
```

Figure 17-25 Step-by-Step Example Operation Converted to Code

The last step is to insert this code after the two lines of code that load the data segment register but before the two lines of code that perform the program termination in Figure 17-23. Figure 17-26 presents the final program.

```
        .MODEL SMALL
        .STACK 100H
        .DATA
A       DW      104
B       DW      100
C       DW      52
RESULT  DW      ?
        .CODE
MAIN    PROC    FAR
        MOV     AX,@DATA    ;Load DS with assigned
        MOV     DS,AX       ; data segment address
        MOV  AX,A           ;Load A from memory
        SAR  AX,3           ;Divide A by 8
        ADD  AX,B           ;Add B to (A/8)
        SUB  AX,C           ;Subtract C from (A/8)+B
        MOV  RESULT,AX      ;Store A/8+B-C to RESULT
        MOV  AX,4C00H       ;Use software interrupt
        INT     21H         ; to terminate program
MAIN    ENDP
        END     MAIN
```

Figure 17-26 Final Code for Example Assembly Language Program

17.7 Additional 80x86 Programming Resources

This chapter falls short of teaching 80x86 assembly language. It is meant to serve only as an introduction. There are a number of resources available both in print and on the web to learn more about programming the 80x86 in assembly language including:

- Abel, Peter, *IBM PC Assembly Language and Programming*, 5th ed., Prentice-Hall, 2001.
- Hyde, Randall, *The Art of Assembly Language*, No Starch Press, 2003. (Available on-line at http://webster.cs.ucr.edu/AoA/DOS/)
- *Intel(R) 186 Processor – Documentation*, Intel Corp., on-line, http://developer.intel.com/design/intarch/intel186/docs_186.htm.

17.8 What's Next?

Over the past seventeen chapters, I have tried to cover three main areas: representation and manipulation of numbers using digital logic, combinational logic and memory circuit design, and basic computer architecture. The intent of this book was never to make the reader a designer of hardware. Instead, the presentation of hardware was meant to provide the reader with well-established tools for logic design along with an understanding of the internals of the computer. The tools can be applied to software as well as hardware. The understanding of hardware can also be applied to software design allowing for improved performance of software applications.

This, however, is merely a beginning. What's the next step for you the reader? The answer to that question depends on what your interests are. At this point, you should have the foundation necessary to begin a deeper study of topics such as advanced computer architecture, embedded systems design, network design, compiler design, or microprocessor design. The possibilities are endless.

Problems

1. What character/symbol is used to indicate the start of a comment in assembly language for the assembler we used in class?
2. Which of the following four strings would make valid assembly language labels? Explain why the invalid ones are not allowed.

ABC123	123ABC
JUMP HERE	LOOP

3. Assume that the register BX contains 5680_{16} when the instruction **`SAR BL,3`** is executed. What would the new value of BL be?
4. Assuming that CX contains 0055_{16} when the instruction **`DEC CH`** is executed, what will CX contain and how will the flags CF, PF, SF, and ZF be set afterwards?
5. Below is a summary description of the 80x86 shift arithmetic left (SAL) instruction:

   Usage: SAL dest,count
   Modifies flags: CF OF PF SF ZF (AF undefined)
   Operation: Shifts the destination left by "count" bits with zeroes shifted in on right. The Carry Flag contains the last bit shifted out.

Assuming that AX contains 2345_{16} when the instruction SAL AH,2 is executed, what will AX contain and how will the flags CF, PF, SF, and ZF be set afterwards?

6. For each of the assembly language commands below, what is the binary value for the active low signals ^MRDC, ^MWTC, ^IORC, and ^IOWC.

	^MRDC	^MWTC	^IORC	^IOWC
`mov   ah,[5674h]`				
`in    bh,1234h`				
`mov   [ax],bx`				
`out   4af5h,bh`				

7. Assume the register BX contains the value 2000h and the table to the right represents the contents of a short portion of memory. Indicate what value AL contains after each of the following MOV instructions.

```
mov   al, ds:[bx]
mov   al, ds:[bx+1]
mov   ax, bx
mov   ax, 2003
```

Address	Value
DS:2000	17h
DS:2001	28h
DS:2002	39h
DS:2003	4Ah
DS:2004	5Bh
DS:2005	6Ch

8. Of the following jump instructions, indicate which ones will jump to the address LOOP, which ones will simply execute the next address (i.e., not jump), and which ones you don't have enough information to tell.

```
Instruction          Current Flag Settings

 je   loop             sf=0, zf=1, cf=0
 jl   loop             sf=1, zf=0
 jng  loop             sf=0, zf=1, of=0
 jne  loop             sf=0, zf=1, of=1
 jnb  loop             sf=1, zf=0, cf=0
 jmp  loop             sf=0, zf=0, of=0
 jge  loop             zf=0, sf=0, of=1
```

9. Modify the code in Figure 17-26 to calculate the expression $((4 \times A) + B - C) \div 32$ where $A = 41_{10}$, $B = 142_{10}$, and $C = 18_{10}$.