Assembly Language Reference for CompE271 San Diego State University
6/8/06 J. Lovegren

Introduction

This document describes essential concepts of Intel-32 bit assembly language. This reference differs from many on the subject as this course uses in-line assembly within the Microsoft Visual C++ Integrated Design Environment (IDE). All programs for this course will be C/C++ programs with assembly statements and/or assembly blocks used within function main. All data declarations will be done in C/C++. All input/output will be done in C/C++. Since knowledge of C is a prerequisite for this course, this approach eliminates having to learn the language and syntax necessary to work in the typically assembler (MASM, TASM, etc) environment. Since this course uses assembly language as a tool to understand CPU function, we focus on the general purpose instruction set of the 32-bit Intel Architecture (IA-32, commonly known as 32-bit x86 assembly).

This reference is largely an adaptation of material from two sources: Microsoft MASM 6.1 manual and the Intel Architecture (IA-32) Software Development Manual (Vol 1, 2, and 3)

The following sections are found in this reference.

Section 1 IA-32 SYSTEM ARCHITECTURE

- **BIT AND BYTE ORDER**
- **INSTRUCTION OPERANDS**
- **BASIC EXECUTION ENVIRONMENT**
  - **Basic Program Execution Registers**
  - **General-Purpose Registers**
  - **EFLAGS Register**
  - **Status Flags**
  - **Instruction Pointer**
  - **Operand Addressing**
  - **Memory Addressing**

Section 2 OPERANDS

- **REGISTER OPERANDS**
- **DIRECT MEMORY OPERANDS**
- **INDIRECT MEMORY OPERANDS**

Section 3 DATA AND SIMPLE INSTRUCTIONS

- **Declaring Variables**
- **General Purpose Instructions**

Section 4 LOOPS and JUMPS

Section 1: IA-32 System Architecture

This material comes from the IA-32 Intel Architecture Software Developer's Manual.

Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.

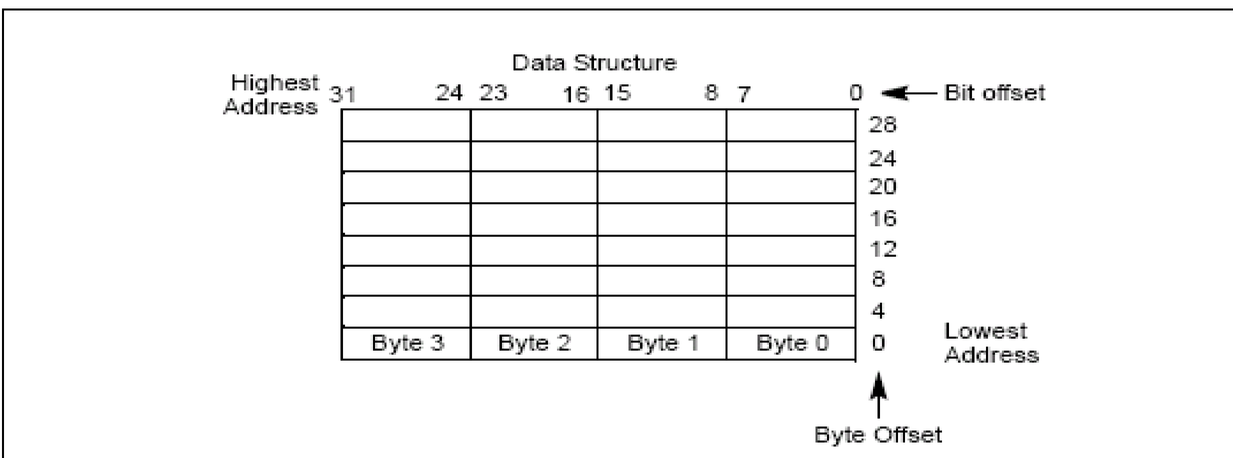

Figure 1-1 Bit Order and Byte Order

Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional.

There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example). When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination. For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order. This course will use the Microsoft Compiler/Assembler instruction syntax as given above.

BASIC EXECUTION ENVIRONMENT

Basic Program Execution Registers

The processor provides 16 basic program execution registers for use in general system and application programming (see Figure x-xx). These registers can be grouped as follows:

- **General-purpose registers**. These eight registers are available for storing operands and pointers.
- **Segment registers**. These registers hold up to six segment selectors.
- **EFLAGS (program status and control) register**. The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
- **EIP (instruction pointer) register**. The EIP register contains a 32-bit pointer to the next instruction to be executed.

General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose. Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

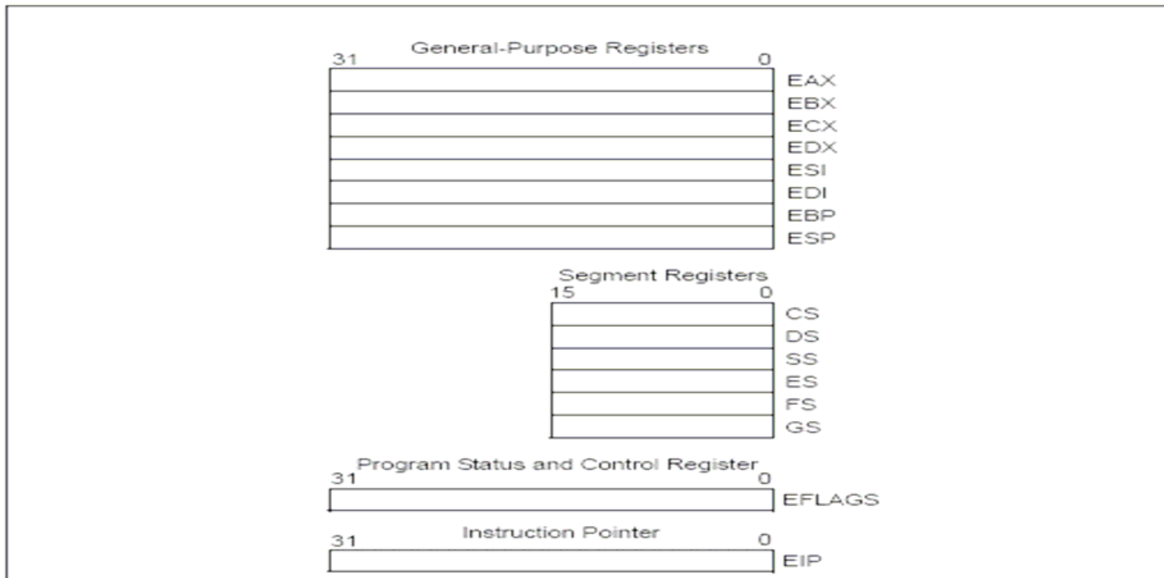

Figure 1-2 User-Accessible Register Names

The special uses of general-purpose registers by instructions are described in Chapter 5, "Instruction Set Summary," in this volume. See also: Chapter 3 and Chapter 4 of *IA-32 Intel® Architecture Software Developer's Manual, Volumes 2A & 2B.* The following is a summary of special uses:

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

General-Purpose Registers

31 — 16	15 — 8	7 — 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 1-3 Alternate (8086) Register Names

As shown in Figure 1-3, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 1-4 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly.

The following instructions can be used to move groups of flags to and from the procedure stack r the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents f the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor's bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor's multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task's TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- **CF (bit 0) Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
- **PF (bit 2) Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.

- **AF (bit 4) Adjust flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
- **ZF (bit 6) Zero flag** — Set if the result is zero; cleared otherwise.

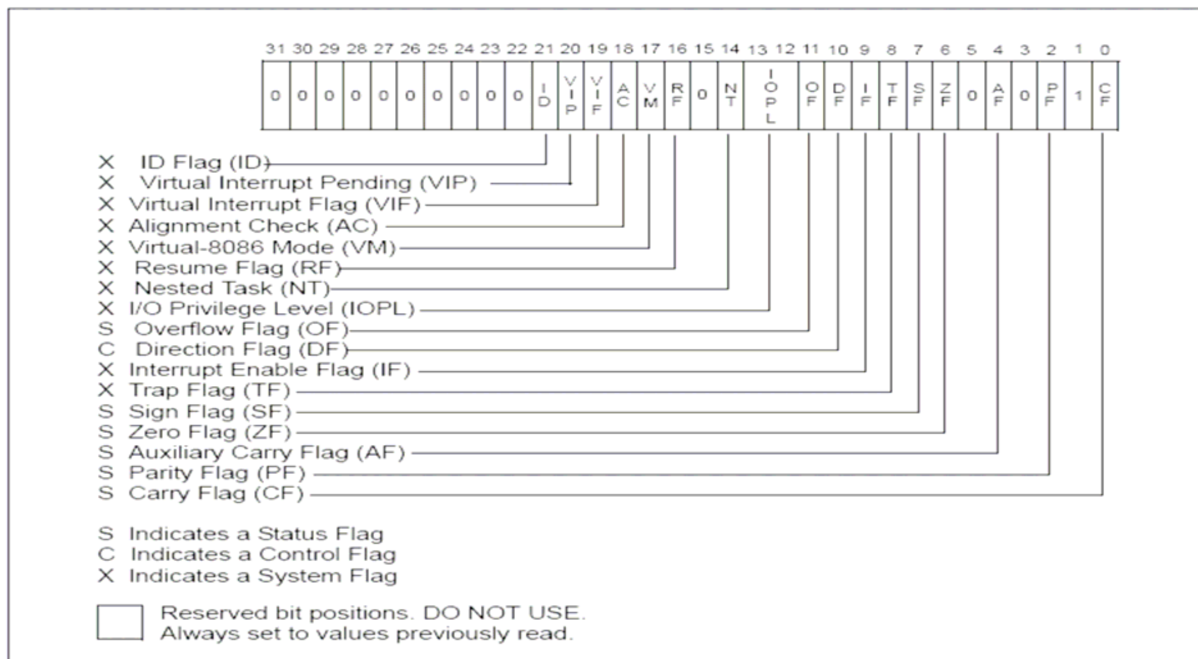

Figure 1-4 EFLAGS Register

- **SF (bit 7) Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
- **OF (bit 11) Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions J*cc* (jump on condition code *cc*), SET*cc* (byte set on condition code *cc*), LOOP*cc*, and CMOV*cc* (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

Instruction Pointer

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straightline code or it is moved ahead or backwards by a number of instructions when executing JMP, J*cc*, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control transfer instructions (such as JMP, J*cc*, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET).

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

OPERAND ADDRESSING

IA-32 machine-instructions act on zero or more operands. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:

- the instruction itself (an immediate operand)
- a register
- a memory location
- an I/O port

When an instruction returns data to a destination operand, it can be returned to:

- a register
- a memory location
- an I/O port

Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called **immediate** operands (or simply immediates). For example, the following ADDinstruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

Register Operands

Source and destination operands can be any of the following registers, depending on the instruction being executed:

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP)
- 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL) segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the J*cc* instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.

Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset. Segment selectors specify the segment containing the operand. Offsets specify the linear or effective address of the operand. Offsets can be 32 bits (represented by the notation m16:32) or 16 bits (represented by the notation m16:16).
In-line assembly in MSVC++ under Windows uses the flat memory model, so segment registers need not be referenced. Windows has control of the segment registers.

MEMORY ADDRESSING

Specifying an Offset

The offset part of a memory address can be specified directly as a static value (called **displacement**) or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-, 16-, or 32-bit value.
- **Base** — The value in a general-purpose register.
- **Index** — The value in a general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2s complement) value, with the exception of the scaling factor. Figure 1-5 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

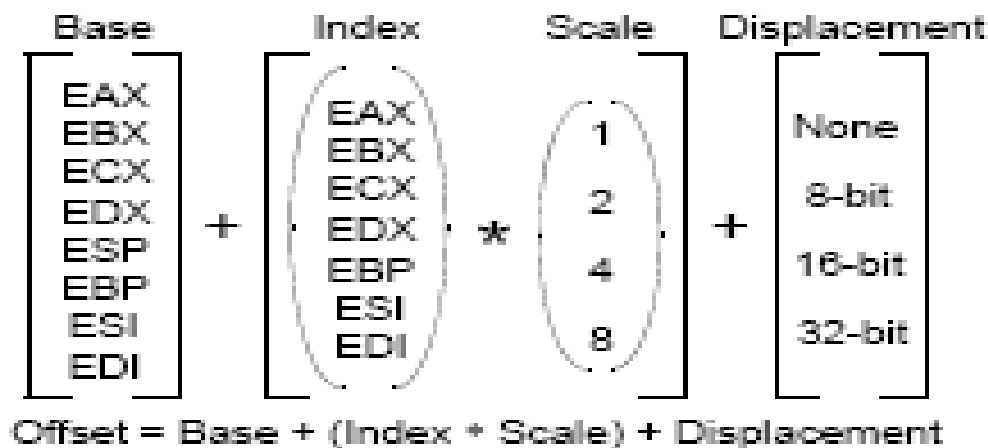

Figure 1-5, Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language.

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address

is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

- **Base** A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** A base register and a displacement can be used together for two distinct purposes:
    - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
    - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

  An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.
- **(Index ∗ Scale) + Displacement** This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.**Base + Index + Displacement** Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index ∗ Scale) + Displacement** Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

Section 2
OPERANDS

(This material adapted from the MASM 6.1 Reference Manual)

With few exceptions, assembly language instructions work on sources of data called operands. In a listing of assembly code, operands appear in the operand field immediately to the right of the instructions.

This section describes the four kinds of instruction operands: register, immediate, direct memory, and indirect memory. Some instructions, such as POPF and STI, have implied operands which do not appear in the operand field. Otherwise, an implied operand is just as real as one stated explicitly.

The following four types of operands are described in the rest of this section:

Operand Type	Addressing Mode
Register	An 8-bit, 16-bit or 32-bit register
Immediate	A constant value contained in the instruction itself.
Direct memory	A fixed location in memory.
Indirect memory	A memory location determined at run time by using the address stored in one or two registers.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be read, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be acted on and possibly changed by the instruction.

In two operand instructions, only one of the operands can be from memory. This helps balance instruction execution time as there will never be more than one memory cycle in the execution of a two operand instruction. So, it is not possible to move data from one memory location directly to another. A register must be used as an intermediate location.

The valid combinations of operands in a two operand instruction are shown below, using the MOV instruction as an example.

Operation	Destination	Source	
mov	reg,	immediate	; OK
mov	reg,	reg	; OK
mov	reg,	memory	; OK
mov	memory,	reg	; OK
mov	memory,	memory	; Invalid

Register Operands

Register operands refer to data stored in registers. The following examples show typical register operands:

```
mov   ebx, 10       ; Load constant to EBX
add   eax, ebx      ; Add EBX to EAX
jmp   edi           ; Jump to the address in EDI
```

An offset stored in a base or index register often serves as a pointer into memory. You can store an offset in one of the base or index registers, then use the register as an indirect memory operand. (See "Indirect Memory Operands," following.) For example:

```
mov   [ebx], dl     ; Store DL (8-bit value) in indirect memory operand
inc   ebx           ; Increment register operand
mov   [ebx], dl     ; Store DL (8-bit value) in new indirect memory operand
```

This example moves the value in DL to a byte of a memory location pointed to by EBX. Any instruction that changes the register value also changes the data item pointed to by the register.

Immediate Operands

An immediate operand is a constant or the result of a constant expression. The assembler encodes immediate values into the instruction at assembly time. Here are some typical examples showing immediate operands:

```
mov   ecx, 20        ; Load constant to register
add   var, 1Fh       ; Add hex constant to variable
sub   ebx, 25 * 80   ; Subtract constant expression
```

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either a register or direct memory to provide a place to store the result of the operation.

Immediate expressions often involve the useful **OFFSET** operator, described in the following paragraphs.

The OFFSET Operator

An address constant is a special type of immediate operand that consists of an offset or segment value. The **OFFSET** operator returns the offset of a memory location, as shown here:

```
mov   ebx, OFFSET var  ; Load offset address
```

Since data in different modules may belong to a single segment, the assembler cannot know for each module the true offsets within a segment. Thus, the offset for ***var***, although an immediate value, is not determined until link time.

Direct Memory Operands

Direct memory operands are always variables. These variables are declared as C/C++ variables using the usual data declaration formats as given for C/C++. (See section on Defining and Using Simple Data Types).

A direct memory operand specifies the data at a given address. The instruction acts on the contents of the address, not the address itself. Except when size is implied by another operand, you must specify the size of a direct memory operand so the instruction accesses the correct amount of memory.

In-line assembly within MSVC++ uses C data declarations to define the size of an operand. Data type ***int*** is a 32-bit value (4-bytes). Data type ***char*** is an 8-bit value (1 byte);

Given the following data declaration:

```
char   ch ;
 .
 .
 mov    ch, al          ; Copy AL to byte at ch
```

Given the following data declaration:

```
int   var2 ;
 .
 .
 mov    var2, eax          ; Copy EAX to doubleword at var2
```

Any location in memory can be a direct memory operand as long as a size is specified (or implied) and the location is fixed. The data at the address can change, but the address cannot. You can create an expression that points to a memory location using any of the following operators:

Operator Name	Symbol
Plus	+
Minus	–
Index	[ ]

These operators are discussed in more detail in the following section.

Plus, Minus, and Index

The plus and index operators perform in exactly the same way when applied to direct memory operands. All operators use number values in bytes. For example, both the following statements move the second doubleword value from an array into the EAX register:

```
mov    eax, array[4]
mov    eax, array+4
```

The index operator can contain any direct memory operand. The following statements are equivalent:

```
mov    eax, var
mov    eax, [var]
```

Some programmers prefer to enclose the operand in brackets to show that the contents, not the address, are used.

The minus operator behaves as you would expect. Both the following instructions retrieve the value located at the doubleword preceding ***array***:

```
mov     eax, array[-4]
mov     eax, array-4
```

Indirect Memory Operands

Like direct memory operands, indirect memory operands specify the contents of a given address. However, the processor calculates the address at run time by referring to the contents of registers. Since values in the registers can change at run time, indirect memory operands provide dynamic access to memory.

Indirect memory operands make possible run-time operations such as pointer indirection and dynamic indexing of array elements, including indexing of multidimensional arrays.

INDIRECT OPERANDS WITH 16- AND 32-BIT REGISTERS

Some rules and options for indirect memory operands always apply, regardless of the size of the register. For example, you must always specify the register and operand size for indirect memory operands. But you can use various syntaxes to indicate an indirect memory operand. This section describes the rules that apply to both 16-bit and 32-bit register modes.

Specifying Indirect Memory Operands

The index operator specifies the register or registers for indirect operands. The processor uses the data pointed to by the register. For example, the following instruction moves into EAX the doubleword value at the address in EBX.

```
mov     eax, DWORD PTR [ebx]
```

When you specify more than one register, the processor adds the contents of the two addresses together to determine the effective address (the address of the data to operate on):

```
mov     eax, [ebx+esi]
```

Specifying Displacements

You can specify an address displacement, which is a constant value added to the effective address. A direct memory specifier is the most common displacement:

```
mov     eax, table[esi]
```

In this relocatable expression, the displacement ***table*** is the base address of an array; ESI holds an index to an array element. The ESI value is calculated at run time, often in a loop. The element loaded into EAX depends on the value of ESI at the time the instruction executes.

Each displacement can be an address or numeric constant. If there is more than one displacement, the assembler totals them at assembly time and encodes the total displacement. For example, in the statement

```
int   table[100] =  0 ;   // C data declaration
.
```

```
.
.
mov     eax, table[ebx][edi]+12
```

both ***table*** and ***12*** are displacements. The assembler adds the value of ***12*** to ***table*** to get the total displacement. However, the statement

```
mov eax, mem1[esi] + mem2
```

is not legal, because it attempts to use a single command to join the contents of two different addresses.

Specifying Operand Size

You must give the size of an indirect memory operand in one of three ways:

- By the variable's declared size
- With the **PTR** operator
- Implied by the size of the other operand

The following lines illustrate all three methods. Assume the size of the ***table*** array is D**WORD** (doubleword**)**, as declared earlier.

```
mov     table[ebx], 0           ; 4 bytes - from size of table
mov     BYTE PTR table, 0       ; 1 byte  - specified by BYTE
mov     eax, [ebx]              ; e bytes - implied by AX
```

Syntax Options

The assembler allows a variety of syntaxes for indirect memory operands. However, all registers must be inside brackets. You can enclose each register in its own pair of brackets, or you can place the registers in the same pair of brackets separated by a plus operator (+). All the following variations are legal and assemble the same way:

```
mov     eax, table[ebx][edi]
mov     eax, table[edi][ebx]
mov     eax, table[ebx+edi]
mov     eax, [table+ebx+edi]
mov     eax, [ebx][edi]+table
```

All of these statements move the value in ***table*** indexed by EBX+EDI into EAX.

Scaling Indexes

The value of index registers pointing into arrays must often be adjusted for zero-based arrays and scaled according to the size of the array items. For a doubleword array, the item number must be multiplied by four (shifted left by two places). When using 32-bit registers, you must scale with separate instructions, as shown here:

```
mov   ebx, 5           ; Get sixth element (adjust for 0)
shl   ebx, 2           ; Scale by two (word size)
inc   wtable[ebx]      ; Increment sixth element in table
```

When using 32-bit registers, you can include scaling in the operand, as detailed in the section below and in the following example.

Given the integer array, myArray:

```
int  myArray[];
          .
          .
add   ebx, myArray[esi]   ;  source is location of myArray + contents of esi as index
```

The index register (ESI, in this case) must be changed by 4 since the next element if myArray is 4 bytes from the current element.

```
add   ebx, myArray[esi * 4]   ;  //source is location of myArray + contents of esi (times a
                                 // scaling factor of 4) as index
```

Here, the index register (ESI, in this case) need only be incremented by 1, since the next element if myArray is scaling factor of 4 gives the desired 4 bytes from the current element.

INDIRECT MEMORY OPERANDS WITH 32-BIT REGISTERS

In 32-bit mode, an offset address can be up to 4 gigabytes. (Segments are still represented in 16 bits.) This effectively eliminates size restrictions on each segment, since few programs need 4 gigabytes of memory. Windows NT/2000/XP uses 32-bit mode and flat model, which spans all segments. XENIX 386 uses 32-bit mode with multiple segments.

On the 80386/486/Pentium, the processor allows you to use any general-purpose 32-bit register as a base or index register, except ESP, which can be a base but not an index. Several examples are shown here:

```
add    edx, [eax]                  ; Add doubleword
mov    dl, [esp+10]                ; Copy byte from stack
dec    DWORD PTR [edx][eax]        ; Decrement word
cmp    eax, array[ebx][ecx]        ; Compare doubleword from array
```

Scaling Factors

The index register can have a scaling factor of 1, 2, 4, or 8. Any register except ESP can be the index register and can have a scaling factor. To specify the scaling factor, use the multiplication operator (*) adjacent to the register.

You can use scaling to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword

arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov   eax, darray[edx*4]        ; Load double of double array
mov   eax, [esi*8][edi]         ; Load double of quad array
mov   ax, wtbl[ecx+2][edx*2]    ; Load word of word array
```

If two registers are used, only one can have a scaling factor. The register with the scaling factor is defined as the index register. The other register is defined as the base. If scaling is not used, the first register is the base. The following examples illustrate how to determine the base register:

```
mov   eax, [edx][ebp*4]   ; EDX base
mov   eax, [edx*1][ebp]   ; EBP base
mov   eax, [edx][ebp]     ; EDX base
mov   eax, [ebp][edx]     ; EBP base
mov   eax, [ebp]          ; EBP base
mov   eax, [ebp*4]        ; EBP*4 index
```

Section 3
DEFINING AND USING SIMPLE DATA TYPES

This chapter covers the concepts essential for working with simple data types in assembly-language programs. The first section shows how to declare integer variables. The second section describes basic operations including moving, loading, and sign-extending numbers, as well as calculating. The last section describes how to do various operations with numbers at the bit level, such as using bitwise logical instructions and shifting and rotating bits.

The complex data types introduced in the next chapter—arrays, strings, structures, unions, and records—use many of the operations illustrated in this chapter. Floating-point operations require a different set of instructions and techniques are not covered in this document.

Declaring Variables

An integer is a whole number, such as 4 or 4,444. Integers have no fractional part, as do the real numbers. You can initialize integer variables in several ways with the data allocation directives. This section explains how to use the **SIZE** and **TYPE** operators to provide information to the compiler about the types in your program.

Allocating Memory for Integer Variables

When you declare an integer variable by assigning a label to a data allocation directive, the assembler allocates memory space for the integer. The variable's name becomes a label for the memory space. In the MSVC++ environment, all data declarations will be done in C.

The following is a review of C/C++ data types and declarations:

Data type	Type of information	Code	Number of bytes
int	signed integer	Two's Complement	4
unsigned int	unsigned integer	Base 2	4
short	signed integer	Two's Complement	2
unsigned short	unsigned integer	Base 2	2
float	real number	IEEE Floating Point	4
double	real number	IEEE Floating Point	8
char	text character	ASCII	1

```
int    VAR1 ;          // assigns 4 bytes for variable VAR1
char   ch ;            // assigns 1 byte for variable ch
int    VAR2 = 1234 ;         //puts the initial value 1234 (decimal) into
                             //the memory location assigned to VAR2
int    VAR2 = 0x1234 ;       //puts the initial value 1234 (hex) into
```

```
                                //the memory location assigned to VAR2
int    values[] ;               // defines the variable values as an array of integers
       // the number of elements in the array named values has not been specified.
int    values[20];              // defines the variable values as an array of integers
                                // with twenty elements
```

Working with Simple Variables

Once you have declared integer variables in your program, you can use them to copy, move, and sign-extend integer variables in your assembly code. This section shows how to do these operations as well as how to add, subtract, multiply, and divide numbers and do bit-level manipulations with logical, shift, and rotate instructions.

Since assembly language instructions require operands to be the same size, you may need to operate on data in a size other than that originally declared. You can do this with the **PTR** operator. For example, you can use the **PTR** operator to access the high-order word of a **DWORD**-size variable. The syntax for the **PTR** operator is

type **PTR** *expression*

where the **PTR** operator forces *expression* to be treated as having the type specified. An example of this use is

```
int num;
mov   al, BYTE PTR num[0] ; // Loads a byte-size value from byte 0 of num into AL
```

If there is any ambiguity about the size of the entity to be accessed, the PTR operator is also used.

```
mov   edx, DWORD PTR [ebx] ;    // a doubleword variable loaded into EDX from
                                // the location the EBX currently points to.
mov   eax,  BYTE PTR [esi] ;  // a byte from the location ESI currently points to
                              // copied to EAX
```

GENERAL PURPOSE INSTRUCTIONS

Copying Data

The primary instructions for moving data from operand to operand and loading them into registers are **MOV** (Move), **XCHG** (Exchange), **CWD** (Convert Word to Double), and **CBW** (Convert Byte to Word).

Moving Data

The most common method of moving data, the **MOV** instruction, is essentially a copy instruction, since it always copies the source operand to the destination operand without affecting the source. After a **MOV** instruction, the source and destination operands contain the same value.

The following example illustrates the **MOV** instruction. As explained in section on Operands, you cannot move a value from one location in memory to another in a single operation. In the illustration below, mem has been declared as an integer variable.

```
int     mem ; // variable mem is an integer (4-bytes), 32-bit value

; Immediate value moves
    mov    eax, 7        ; Immediate to register
    mov    mem, 7        ; Immediate to memory direct
    mov    mem[ebx], 7   ; Immediate to memory indirect

; Register moves
    mov    mem, eax      ; Register to memory direct
    mov    mem[ebx], eax ; Register to memory indirect
    mov    eax, ebx      ; Register to register

; Direct memory moves
    mov    eax, mem      ; Memory direct to register
    mov    mem, ebx      ; Memory to segment register

; Indirect memory moves
    mov    eax, mem[ebx] ; Memory indirect to register
    mov    mem[ebx], edx ; Register to Memory indirect
```

The following example shows several common types of moves that require two instructions.

```
; Move memory to memory
    mov    eax, mem1     ; Load EAX with memory value
    mov    mem2, eax     ; Copy EAX to other memory
```

The **MOVSX** and **MOVZX** instructions for the 80386/486 processors extend and copy values in one step. See “Extending Signed and Unsigned Integers,” following.

Exchanging Integers

The **XCHG** (Exchange) instruction exchanges the data in the source and destination operands. You can exchange data between registers or between registers and memory, but not from memory to memory:

```
        xchg    eax, ebx        ; Put EAX in EBX and EBX in EAX
        xchg    memory, eax     ; Put "memory" in EAX and EAX in "memory"
;       xchg    mem1, mem2      ; Illegal- can't exchange memory locations
```

Extending Signed and Unsigned Integers

Since moving data between registers of different sizes is illegal, you must "sign-extend" integers to convert signed data to a larger size. Sign-extending means copying the sign bit of the unextended operand to all bits of the operand's next larger size. This widens the operand while maintaining its sign and value.

8086-based processors provide four instructions specifically for sign-extending. The four instructions act only on the accumulator register (AL, AX, or EAX), as shown in the following list.

Instruction	Sign-extend
CBW (convert byte to word)	AL to AX
CWD (convert word to doubleword)	AX to DX:AX
CWDE (convert word to doubleword extended)*	AX to EAX
CDQ (convert doubleword to quadword)*	EAX to EDX:EAX

*Requires an extended register and applies only to 80386/486/Pentium processors.

On the 80386/486/Pentium processors, the **CWDE** instruction converts a signed 16-bit value in AX to a signed 32-bit value in EAX. The **CDQ** instruction converts a signed 32-bit value in EAX to a signed 64-bit value in the EDX:EAX register pair.

This example converts signed integers using **CBW**, **CWD**, **CWDE**, and **CDQ**.

```
__int8      mem8   = -5   ;  // 8-bit value
short       mem16 =  +5 ;  // 16-bit value
int         mem32 = -5   ;  // 32-bit value
  .
  .
  .
  mov     al, mem8     ; Load 8-bit -5 (FBh)
  cbw                  ; Convert to 16-bit -5 (FFFBh) in AX
  mov     ax, mem16    ; Load 16-bit +5
  cwd                  ; Convert to 32-bit +5 (0000:0005h) in DX:AX
  mov     ax, mem16    ; Load 16-bit +5
  cwde                 ; Convert to 32-bit +5 (00000005h) in EAX
  mov     eax, mem32   ; Load 32-bit -5 (FFFFFFFBh)
  cdq                  ; Convert to 64-bit -5
                       ;   (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

These four instructions efficiently convert unsigned values as well, provided the sign bit is zero. This example, for instance, correctly widens mem16 whether you treat the variable as signed or unsigned.

The processor does not differentiate between signed and unsigned values. For instance, the value of mem8 in the previous example is literally 251 (0FBh) to the processor. It ignores the human convention of treating the highest bit as an indicator of sign. The processor can ignore the distinction between signed and unsigned numbers because binary arithmetic works the same in either case.

If you add 7 to mem8, for example, the result is 258 (102h), a value too large to fit into a single byte. The byte-sized mem8 can accommodate only the least-significant digits of the result (02h), and so receives the value of 2. The result is the same whether we treat mem8 as a signed value (-5) or unsigned value (251).

This overview illustrates how the programmer, not the processor, must keep track of which values are signed or unsigned, and treat them accordingly. If AL=127 (01111111y), the instruction **CBW** sets AX=127 because the sign bit is zero. If AL=128 (10000000y), however, the sign bit is 1. **CBW** thus sets AX=65,280 (FF00h), which may not be what you had in mind if you assumed AL originally held an unsigned value. To widen unsigned values, explicitly set the higher register to zero, as shown in the following example:

```
__ int8    mem8    =   251; // 8-bit value
short           mem16  =   251;
        .
        .
        .
        mov     al, mem8  ; Load 251 (FBh) from 8-bit memory
        sub     ah, ah    ; Zero upper half (AH)

        mov     ax, mem16 ; Load 251 (FBh) from 16-bit memory
        sub     dx, dx    ; Zero upper half (DX)

        sub     eax, eax  ; Zero entire extended register (EAX)
        mov     ax, mem16 ; Load 251 (FBh) from 16-bit memory
```

IA-32 processors provide instructions that move and extend a value to a larger data size in a single step. **MOVSX** moves a signed value into a register and sign-extends it. **MOVZX** moves an unsigned value into a register and zero-
extends it.

```
; 80386/486 instructions
        movzx   dx, bl      ; Load unsigned 8-bit value into
                            ;   16-bit register and zero-extend
```

Adding and Subtracting Integers

You can use the **ADD**, **ADC**, **INC**, **SUB**, **SBB**, and **DEC** instructions for adding, incrementing, subtracting, and decrementing values in single registers. You can also combine them to handle larger values that require two registers for storage.

Adding and Subtracting Integers Directly

The **ADD**, **INC** (Increment), **SUB**, and **DEC** (Decrement) instructions operate on 8-, 16-, and 32-bit values. They can be combined with the **ADC** and **SBB** instructions to work on 64-bit values. (See "Adding and Subtracting in Multiple Registers," following.)

These instructions have two requirements:

1. If there are two operands, only one operand can be a memory operand.
2. If there are two operands, both must be the same size.

To meet the second requirement, you can use the **PTR** operator to force an operand to the size required. (See "Working with Simple Variables," previous.) For example, if `Buffer` is an array of bytes and EBX points to an element of the array, you can add a doubleword from `Buffer` with

```
add     eax, DWORD PTR Buffer[ebx] ; Add doubleword from byte array
```

The next example shows 8-bit signed and unsigned addition and subtraction.

```
        __int8  mem8  =  39 ;

; Addition
                              ;              signed    unsigned
        mov    al, 26   ; Start with register   26        26
        inc    al       ; Increment              1         1
        add    al, 76   ; Add immediate         76      + 76
                ;                              ----      ----
                ;                              103       103
        add    al, mem8 ; Add memory            39      + 39
                ;                              ----      ----
        mov    ah, al   ;   Copy to AH        -114       142
                                             +overflow
        add    al, ah   ; Add register                   142
                ;                              ----
                ;                             28+carry
; Subtraction

       mov     mem8, 122  ;  new value for mem8
      ;                            signed    unsigned
        mov    al, 95   ; Load register          95      95
        dec    al       ; Decrement              -1      -1
        sub    al, 23   ; Subtract immediate    -23     -23
                ;                               ----    ----
                ;                                71      71
        sub    al, mem8 ; Subtract memory      -122    -122
```

```
                ;                            ----    ----
                ;                            -51     205 (+sign)

        mov     ah, 119  ; Load register     119
        sub     al, ah   ;  and subtract     -51
                ;                            ----
                ;                          86+overflow
```

The **INC** and **DEC** instructions treat integers as unsigned values and do not update the carry flag for signed carries and borrows.

When the sum of 8-bit signed operands exceeds 127, the processor sets the overflow flag. (The overflow flag is also set if both operands are negative and the sum is less than or equal to -128.) Placing a **JO** (Jump on Overflow) or **INTO** (Interrupt on Overflow) instruction in your program at this point can transfer control to error-recovery statements. When the sum exceeds 255, the processor sets the carry flag. A **JC** (Jump on Carry) instruction at this point can transfer control to error-recovery statements.

In the previous subtraction example, the processor sets the sign flag if the result goes below 0. At this point, you can use a **JS** (Jump on Sign) instruction to transfer control to error-recovery statements. Jump instructions are described in the “Jumps” section in Chapter 7.

MULTIPLYING AND DIVIDING INTEGERS

The IA-32 processors use different multiplication and division instructions for signed and unsigned integers. Multiplication and division instructions also have special requirements depending on the size of the operands and the processor the code runs on.

Using Multiplication Instructions

The **MUL** instruction multiplies unsigned numbers. **IMUL** multiplies signed numbers. For both instructions, one factor must be in the accumulator register (AL for 8-bit numbers, AX for 16-bit numbers, EAX for 32-bit numbers). The other factor can be in any single register or memory operand. The result overwrites the contents of the accumulator register.

Multiplying two 8-bit numbers produces a 16-bit result returned in AX. Multiplying two 16-bit operands yields a 32-bit result in DX:AX. The 80386/486/Pentium processor handles 64-bit products in the same way in the EDX:EAX pair.

This example illustrates multiplication of signed 16- and 32-bit integers.

```
        short mem16  =  -30000 ; // 16-bit value

        .
        .
        .
    ; 8-bit unsigned multiply
        mov     al, 23    ; Load AL              23
        mov     bl, 24    ; Load BL            * 24
        mul     bl        ; Multiply BL        -----
```

```
                        ; Product in AX          552
                        ;   overflow and carry set

    ; 16-bit signed multiply
        mov     ax, 50    ; Load AX                 50
                        ;                    -30000
        imul    mem16     ; Multiply memory         -----
                        ; Product in DX:AX      -1500000
                        ;   overflow and carry set
```

A nonzero number in the upper half of the result (AH for byte, DX or EDX for word) sets the overflow and carry flags.

On the IA-32 processors, the **IMUL** instruction supports three additional operand combinations. The first syntax option allows for 16-bit multipliers producing a 16-bit product or 32-bit multipliers for 32-bit products. The result overwrites the destination. The syntax for this operation is:

IMUL *register16*, *immediate*

The second syntax option specifies three operands for **IMUL**. The first operand must be a 16-bit *register* operand, the second a 16-bit *memory* (or *register*) operand, and the third a 16-bit *immediate* operand. **IMUL** multiplies the memory (or register) and immediate operands and stores the product in the register operand with this syntax:

IMUL *register16*,**{** *memory16* **|** *register16***},** *immediate*

For the 80386/486 only, a third option for **IMUL** allows an additional operand for multiplication of a register value by a register or memory value. The syntax is:

IMUL *register*,{*register* **|** *memory*}

The destination can be any 16-bit or 32-bit register. The source must be the same size as the destination.

In all of these options, products too large to fit in 16 or 32 bits set the overflow and carry flags. The following examples show these three options for **IMUL**.

```
    imul    dx, 456     ; Multiply DX times 456 on 80186-80486
    imul    ax, [bx],6  ; Multiply the value pointed to by BX
                        ;   by 6 and put the result in AX

    imul    dx, ax      ; Multiply DX times AX on 80386
    imul    ax, [bx]    ; Multiply AX by the value pointed to
                        ;   by BX on 80386
```

The **IMUL** instruction with multiple operands can be used for either signed or unsigned multiplication, since the 16-bit product is the same in either case. To get a 32-bit result, you must use the single-operand version of **MUL** or **IMUL**.

Using Division Instructions

The **DIV** instruction divides unsigned numbers, and **IDIV** divides signed numbers. Both return a quotient and a remainder.

Table 4.1 summarizes the division operations. The dividend is the number to be divided, and the divisor is the number to divide by. The quotient is the result. The divisor can be in any register or memory location except the registers where the quotient and remainder are returned.

Table 4.1 Division Operations

Size of Operand	Dividend Register	Size of Divisor	Quotient	Remainder
16 bits	AX	8 bits	AL	AH
32 bits	DX:AX	16 bits	AX	DX
64 bits	EDX:EAX	32 bits	EAX	EDX

Unsigned division does not require careful attention to flags. The following examples illustrate signed division, which can be more complex.

```
short   mem16   =  -2000;   // 16-bit value
int     mem32   =  500000; // 32-bit value
     .CODE
     .
     .
     .
; Divide 16-bit unsigned by 8-bit
     mov    ax, 700              ; Load dividend     700
     mov    bl, 36               ; Load divisor DIV   36
     div    bl                   ; Divide BL       ------
                                 ; Quotient in AL      19
                                 ; Remainder in AH        16

; Divide 32-bit signed by 16-bit
     mov    ax, WORD PTR mem32[0] ; Load into DX:AX
     mov    dx, WORD PTR mem32[2] ;              500000
     idiv   mem16                ;            DIV -2000
                                 ; Divide memory   ------
                                 ; Quotient in AX    -250
                                 ; Remainder in DX         0

; Divide 16-bit signed by 16-bit
     mov    ax, WORD PTR mem16   ; Load into AX     -2000
     cwd                         ; Extend to DX:AX
     mov    bx,-421              ;            DIV -421
     idiv   bx                   ; Divide by BX     -----
                                 ; Quotient in AX       4
                                 ; Remainder in DX       -316
```

If the dividend and divisor are the same size, sign-extend or zero-extend the dividend so that it is the length expected by the division instruction. See “Extending Signed and Unsigned Integers,” earlier in this chapter.

Manipulating Numbers at the Bit Level

The instructions introduced so far in this chapter access numbers at the byte or word level. The logical, shift, and rotate instructions described in this section access individual bits in a number. You can use logical instructions to evaluate characters and do other text and screen operations. The shift and rotate instructions do similar tasks by shifting and rotating bits through registers. This section reviews some applications of these bit-level operations.

Logical Instructions

The logical instructions **AND**, **OR**, and **XOR** compare bits in two operands. Based on the results of the comparisons, the instructions alter bits in the first (destination) operand. The logical instruction **NOT** also changes bits, but operates on a single operand.

The following list summarizes these four logical instructions. The list makes reference to the “destination bit,” meaning the bit in the destination operand. The terms “both bits” and “either bit” refer to the corresponding bits in the source and destination operands. These instructions include:

Instruction	Sets Destination Bit If	Clears Destination Bit If
AND	Both bits set	Either or both bits clear
OR	Either or both bits set	Both bits clear
XOR	Either bit (but not both) set	Both bits set or both clear
NOT	Destination bit clear	Destination bit set

NOTE: Do not confuse logical instructions with the logical operators, which perform these operations at assembly time, not run time. Although the names are the same, the assembler recognizes the difference.

The following example shows the result of the **AND**, **OR**, **XOR**, and **NOT** instructions operating on a value in the AX register and in a mask. A mask is any number with a pattern of bits set for an intended operation.

```
mov     ax, 035h   ; Load value                     00110101
and     ax, 0FBh   ; Clear bit 2                AND 11111011
               ;                                    -----------
                  ; Value is now 31h                00110001
or     ax, 016h   ; Set bits 4,2,1               OR  00010110
               ;                                    -------------
               ; Value is now 37h                   00110111
xor     ax, 0ADh  ;Toggle bits 7,5,3,2,0        XOR 10101101
               ;                                    -------------
               ;    Value is now 9Ah                10011010
not     ax        ; Value is now 65h                01100101
```

The **AND** instruction clears unmasked bits — that is, bits not protected by 1 in the mask. To mask off certain bits in an operand and clear the others, use an appropriate masking value in the source operand. The bits of the mask should be 0 for any bit positions you want to clear and 1 for any bit positions you want to remain unchanged.

The **OR** instruction forces specific bits to 1 regardless of their current settings. The bits of the mask should be 1 for any bit positions you want to set and 0 for any bit positions you want to remain unchanged.

The **XOR** instruction toggles the value of specific bits on and off—that is, reverses them from their current settings. This instruction sets a bit to 1 if the corresponding bits are different or to 0 if they are the same. The bits of the mask should be 1 for any bit positions you want to toggle and 0 for any bit positions you want to remain unchanged.

The following examples show an application for each of these instructions. The code illustrating the **AND** instruction converts a “y” or “n” read from the keyboard to uppercase, since bit 5 is always clear in uppercase letters. In the example for **OR**, the first statement is faster and uses fewer bytes than `cmp bx, 0`. When the operands for **XOR** are identical, each bit cancels itself, producing 0.

```
;AND example - converts characters to uppercase
        mov     al, in_char        ; The character input is in the variable in_char
        and     al, 11011111b   ; Convert to uppercase by clearing bit 5
        cmp     al, 'Y'        ; Is it Y?
        je      yes            ; If so, do Yes actions
        .                      ; Else do No actions
        .
yes:    .

;OR example - compares operand to 0
        or      ebx, ebx        ; Compare to 0
        jg      positive       ;  EBX is positive
        jl      negative       ;  EBX is negative
                               ; else EBX is zero

;XOR example - sets a register to 0
        xor     cx, cx         ;  CX = 0
        sub     cx, cx         ;  CX = 0
        mov     cx, 0          ;  CX = 0
```

The TEST instruction performs an implied logical AND of two operands and sets the SF, ZF, and PF flags according to the results. The flags can then be tested by the conditional jump or loop instructions. The TEST instruction differs from the AND instruction in that it does not alter either of the operands.

Bit Test and Modify Instructions

The bit test and modify instructions (see Table 7-3) operate on a single bit, which can be in an operand. The location of the bit is specified as an offset from the least significant bit of the operand. When the processor identifies the bit to be tested and modified, it first loads the CF flag with the current value of the bit. Then it assigns a new value to the selected bit, as determined by the modify operation for the instruction.

Instruction	Effect on CF Flag	Effect on Selected Bit
BT (Bit Test)	CF flag ← Selected Bit	No effect
BTS (Bit Test and Set)	CF flag ← Selected Bit	Selected Bit ← 1
BTR (Bit Test and Reset)	CF flag ← Selected Bit	Selected Bit ← 0
BTC (Bit Test and Complement)	CF flag ← Selected Bit	Selected Bit ← NOT (Selected Bit)

Table 7-3. Bit Test and Modify Instructions

Bit Scan Instructions

The **BSF** (Bit Scan Forward) and the **BSR** (Bit Scan Reverse) instructions perform operations like those of the logical instructions. They scan the contents of a register to find the first-set or last-set bit. You can use **BSF** or **BSR** to find the position of a set bit in a mask or to check if a register value is 0.

The BSF (bit scan forward) and BSR (bit scan reverse) instructions scan a bit string in a source operand for a set bit and store the bit index of the first set bit found in a destination register. The bit index is the offset from the least significant bit (bit 0) in the bit string to the first set bit. The BSF instruction scans the source operand low-to-high (from bit 0 of the source operand toward the most significant bit); the BSR instruction scans high-to-low (from the most significant bit toward the least significant bit).

Shifting and Rotating Bits

The shift and rotate instructions rearrange the bits within an operand. For the purpose of this discussion, these instructions are further divided subordinate subgroups of instructions that:

- Shift bits
- Double-shift bits (move them between operands)
- Rotate bits

The IA-32 processors provide a complete set of instructions for shifting and rotating bits. Shift instructions move bits a specified number of places to the right or left. The last bit in the direction of the shift goes into the carry flag, and the first bit is filled with 0 or with the previous value of the first bit.

Rotate instructions also move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate operation moves into the first bit position at the other end of the operand. With some variations, the carry bit is used as an additional bit of the operand. Figure 4.2 illustrates the eight variations of shift and rotate instructions for 8-bit operands. Notice that **SHL** and **SAL** are identical.

Shift Instructions

The SAL (shift arithmetic left), SHL (shift logical left), SAR (shift arithmetic right), SHR (shift logical right) instructions perform an arithmetic or logical shift of the bits in a byte, word, or doubleword.

The SAL and SHL instructions perform the same operation (see Figure 7-6). They shift the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.

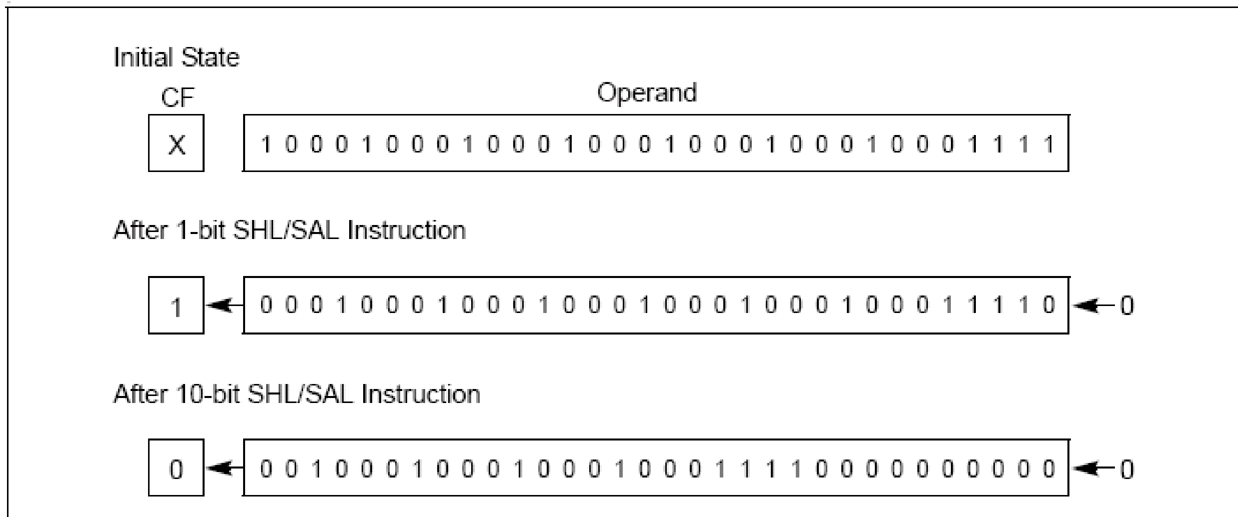

Figure 7-6. SHL/SAL Instruction Operation

The SHR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-7). As with the SHL/SAL instruction, the empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.

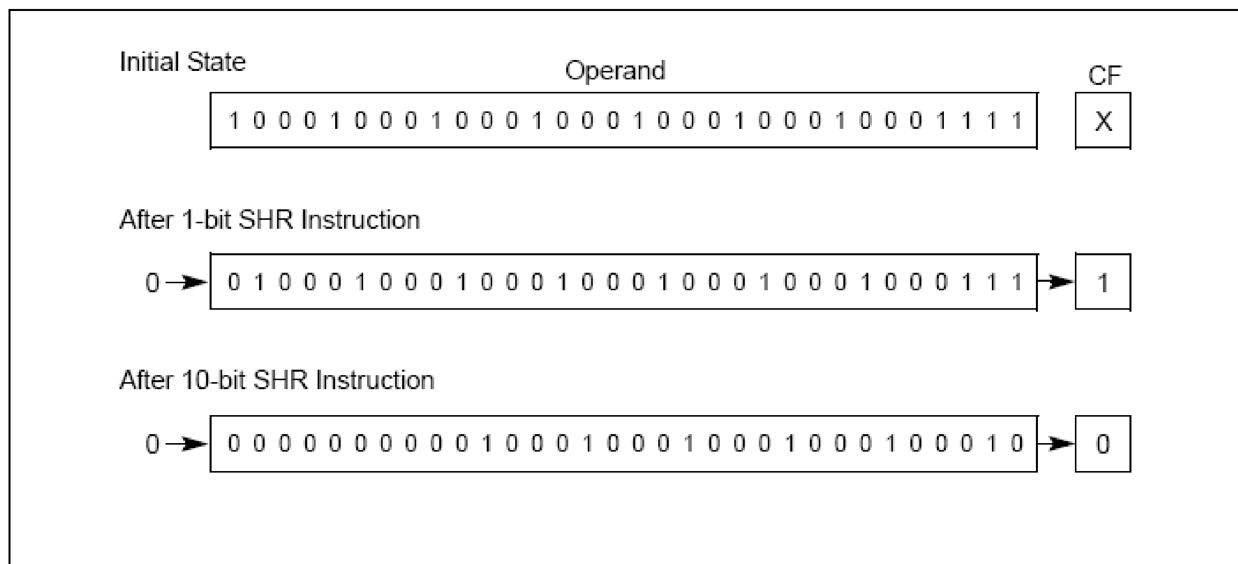

Figure 7-7. SHR Instruction Operation

The SAR instruction shifts the source operand right by from 1 to 31 bit positions (see Figure 7-8). This instruction differs from the SHR instruction in that it preserves the sign of the source operand by clearing empty bit positions if the operand is positive or setting the empty bits if the operand is negative. Again, the CF flag is loaded with the last bit shifted out of the operand.

The SAR and SHR instructions can also be used to perform division by powers of 2.

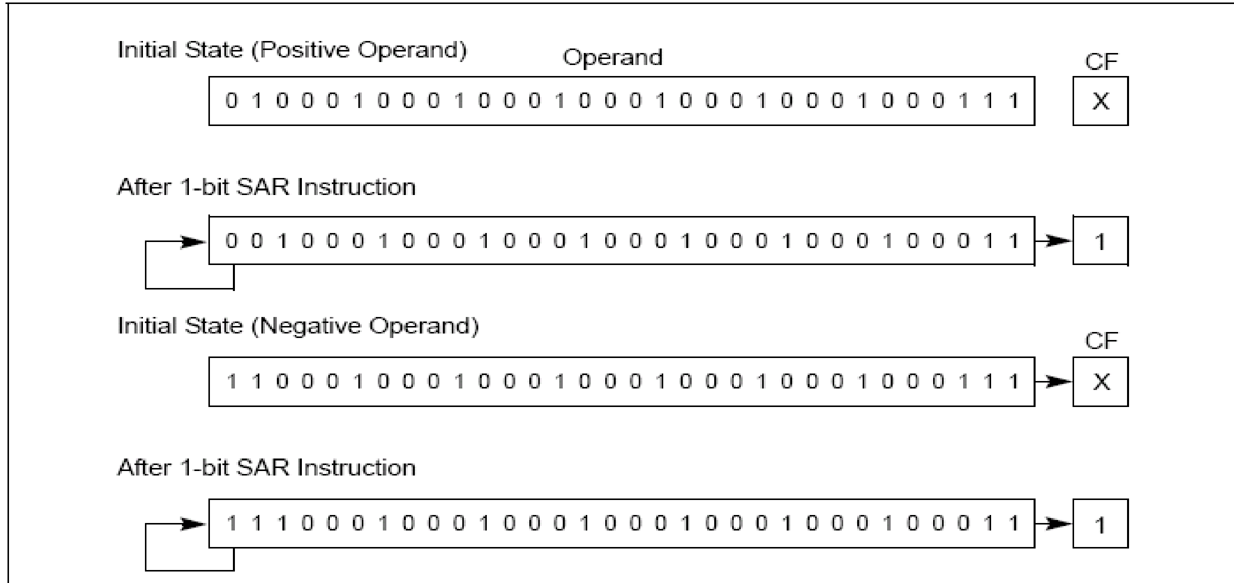

Figure 7-8. SAR Instruction Operation

Double-Shift Instructions

The SHLD (shift left double) and SHRD (shift right double) instructions shift a specified number of bits from one operand to another (see Figure 7-9). They are provided to facilitate operations on unaligned bit strings. They can also be used to implement a variety of bit string move operations.

The SHLD instruction shifts the bits in the destination operand to the left and fills the empty bit positions (in the destination operand) with bits shifted out of the source operand. The destination and source operands must be the same length (either words or doublewords). The shift count can range from 0 to 31 bits. The result of this shift operation is stored in the destination operand, and the source operand is not modified. The CF flag is loaded with the last bit shifted out of the destination operand.

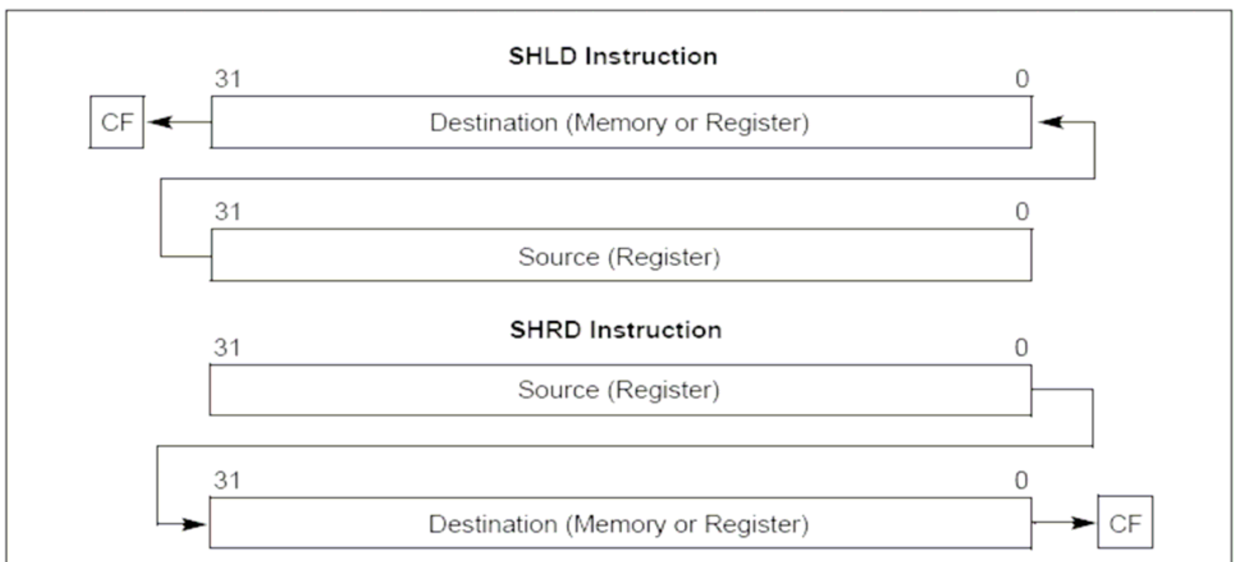

Figure 7-9. SHLD and SHRD Instruction Operations

The SHRD instruction operates the same as the SHLD instruction except bits are shifted to the right in the destination operand, with the empty bit positions filled with bits shifted out of the source operand.

Rotate Instructions

The ROL (rotate left), ROR (rotate right), RCL (rotate through carry left) and RCR (rotate through carry right) instructions rotate the bits in the destination operand out of one end and back through the other end (see Figure 7-10). Unlike a shift, no bits are lost during a rotation. The rotate count can range from 0 to 31.

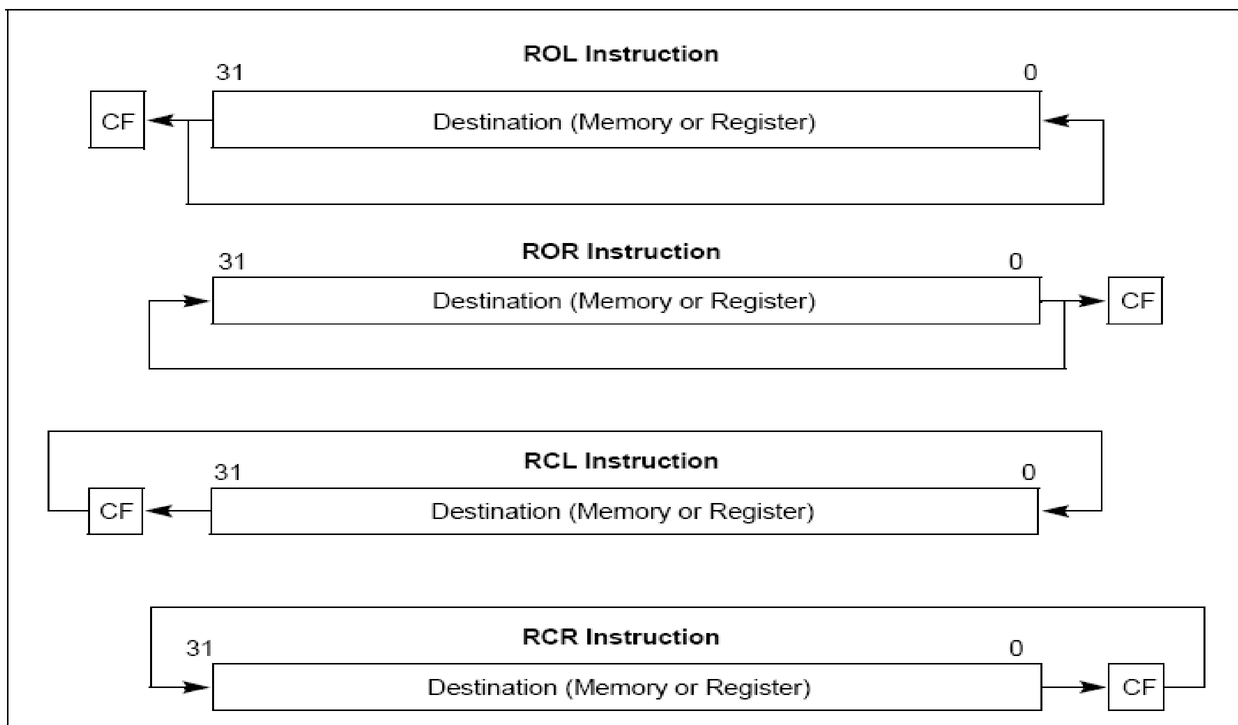

Figure 7-10. ROL, ROR, RCL, and RCR Instruction Operations

The ROL instruction rotates the bits in the operand to the left (toward more significant bit locations).

The ROR instruction rotates the operand right (toward less significant bit locations).

The RCL instruction rotates the bits in the operand to the left, through the CF flag. This instruction treats the CF flag as a one-bit extension on the upper end of the operand. Each bit that exits from the most significant bit location of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the least significant bit location of the operand.

The RCR instruction rotates the bits in the operand to the right through the CF flag. For all the rotate instructions, the CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand. The value of this flag can then be tested by a conditional jump instruction (JC or JNC).

All shift instructions use the same format. Before the instruction executes, the destination operand contains the value to be shifted; after the instruction executes, it contains the shifted operand. The source operand contains the number of bits to shift or rotate. It can be the immediate value 1 or the CL register.

You can use 8-bit immediate values larger than 1 as the source operand for shift or rotate instructions, as shown here:

```
        shr     bx, 4   ;
```

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position, allowing you to mask off a different bit each time the mask is used. This technique, illustrated in the following example, is useful only if the mask value is unknown until run time.

```
__int8  masker     0x02   ; Mask that may change at run time
        .
        .
        .
        mov     cl, 2       ; Rotate two at a time
        mov     bl, 57h     ; Load value to be changed 01010111y
        rol     masker, cl  ; Rotate two to left       00001000y
        or      bl, masker  ; Turn on masked values    ---------
                            ; New value is 05Fh        01011111y
        rol     masker, cl  ; Rotate two more          00100000y
        or      bl, masker  ; Turn on masked values    ---------
                            ; New value is 07Fh        01111111y
```

Arrays and Strings

An array is a sequential collection of variables, all of the same size and type, called “elements.” A string is an array of characters. For example, in the string “ABC,” each letter is an element. You can access the elements in an array or string relative to the first element. This section explains how to handle arrays and strings in your programs.

Declaring and Referencing Arrays

Array elements occupy memory contiguously, so a program references each element relative to the start of the array. Declararing and initializing arrays is done in C following the usual methods. The following examples declare the integer arrays ***y*** and ***z***:

```
int y[10] = {1, 2, 3}; //Allocates memory for 10 int variables, y[0], y[1], ... y[9].
          // Initializes the first three of these to 1, 2 and 3, and the remaining seven  to 0.

int z[] = {1, 2, 3};      //Allocates memory for 3 int variables, z[0], z[1], z[2]
                          //  and initializes them to 1, 2 and 3.
```

Referencing Arrays

Each element in an array is referenced with an index number, beginning with zero. The array index appears in brackets after the array name, as in

array[9]

Assembly-language indexes differ from indexes in high-level languages, where the index number always corresponds to the element's position. In C, for example, ***array[9]*** references the array's tenth element, regardless of whether each element is 1 byte or 8 bytes in size.

In assembly language, an element's index refers to the number of bytes between the element and the start of the array. This distinction can be ignored for arrays of byte-sized elements, since an element's position number matches its index. Data type char occupies one byte. For example, defining the array

char list[] = "97643"

gives a value of 9 to ***list[0]***, a value of 7 to ***list[1]***, and so forth.

However, in arrays with elements larger than 1 byte, index numbers (except zero) do not correspond to an element's position. You must multiply an element's position by its size to determine the element's index. Thus, for the array

int prime[] = {1, 3, 5, 7, 11, 13, 17}

prime[4] represents the second element (3), which is 4 bytes from the beginning of the array. Similarly, the expression ***prime[12]*** represents the fourth element (7) and ***prime[20]*** represents the sixth element (13).

The following example determines an index at run time. It multiplies the position by four (the size of a doubleword element) by shifting it left:

```
mov   esi, ecx          ; ECX holds position number
shl   esi, 2            ; Scale for doubleword (x4) referencing
mov   eax, prime[esi]   ; Move element into EAX
```

It is also possible to use the index scaling capability of the processor to scale an index. The two instructions below are identical to the three instruction sequence above.

```
mov   esi, ecx              ; ECX holds position number
mov   eax, prime[esi * 4]   ; Move element into EAX
```

The offset required to access an array element can be calculated with the following formula:

*n*th element of array = array[(*n*-1) * size of element]

Referencing an array element by distance rather than position is not difficult to master, and is actually very consistent with how assembly language works. Recall that a variable name is a symbol that represents the contents of a particular address in memory. Thus, if the array ***wprime*** begins at address 2400h, the reference ***wprime[8]*** means to the processor "the word value contained in the at offset 2400h-plus-8-bytes."

As described in "Direct Memory Operands," Chapter 3, you can substitute the plus operator (+) for brackets, as in:

```
prime[9]
prime+9
```

Since brackets simply add a number to an address, you don't need them when referencing the first element. Thus, ***prime*** and ***prime[0]*** both refer to the first element of the array ***prime***.

LENGTH, SIZE, and TYPE for Arrays

When applied to arrays, the **LENGTH**, **SIZE**, and **TYPE** operators return information about the length and size of the array and about the type of the initializers.

The **LENGTH** operator returns the number of elements in the array. The **SIZE** operator returns the number of bytes used by the initializers in the array definition. **TYPE** returns the size of the elements of the array. The following examples illustrate these operators:

```
int myArray[40] ;

        LENGTH myArray   ; 40 elements
        TYPE   myArray   ;  4 bytes per element
        SIZE   myArray   ; 160 bytes

int num[] =  {4, 5, 6, 7, 8, 9, 10, 11};
        LENGTH num     ;  8 elements
        TYPE   num     ;  4 bytes per element
        SIZE   num     ; 32 bytes
```

A common use of the LENGTH operator is to initialize a loop counter for code that operates on an array. For example, the following initializes ECX (loop counter) to the number of elements in the variable myArray:

```
mov    ecx, LENGTH myArray;
```

Based on the definition above, ecx now contains 40.

Declaring and Initializing Strings

A string is an array of characters. Initializing a string like `"Hello, there"` allocates and initializes 1 byte for each character in the string and also adds the null terminating character ('\0' or 00h). An initialized string can be no longer than 255 characters.

```
char str1[]  =   "This is a long string that does not "
                 "fit on one line."
```

Strings must be enclosed in double (") quotation marks. A single quotation mark may be used inside a string enclosed by double quotation marks. If you need quotation marks inside a string enclosed by double quotation marks, use the escape sequence \" . These examples show the various uses of quotation marks:

```
char    ch =    'a' ;  // initializes ch with ASCII value for lower case a
char    msg1[] = "That's the message."  ;   //That's the message.
char    msg2[] =     "This \"value\" not found." ;// This "value" not found.
```

LENGTH, SIZE, and TYPE for Strings

Because strings are simply arrays of byte elements, the **LENGTH**, **SIZE**, and **TYPE** operators behave as you would expect, as illustrated in this example:

```
char  mesg[] =     "This string contains many characters, but how many does it have?";

LENGTH msg    ; 37 elements

TYPE       msg    ;  1 byte per element
SIZE        msg    ; 37 bytes
```

Section 4

Loops and Jumps

LOOPS

Loops repeat an action until a termination condition is reached. This condition can be a counter or the result of an expression's evaluation.

Instructions	Action
LOOP	Automatically decrements ECX. When ECX = 0, the loop ends. The top of the loop cannot be greater than 128 bytes from the **LOOP** instruction. (This is true for all **LOOP** instructions.)
LOOPE/LOOPZ, LOOPNE/LOOPNZ	Loops while equal or not equal. Checks both ECX and the state of the zero flag. **LOOPZ** ends when either ECX=0 or the zero flag is clear, whichever occurs first. **LOOPNZ** ends when either ECX=0 or the zero flag is set, whichever occurs first. **LOOPE** and **LOOPZ** assemble to the same machine instruction, as do **LOOPNE** and **LOOPNZ**. Use whichever mnemonic best fits the context of your loop. Set ECX to a number out of range if you don't want a count to control the loop.
JCXZ, JECXZ	Branches to a label only if ECX = 0. Unlike other conditional-jump instructions, which can jump to either a near or a short label, **JCXZ** and **JECXZ** always jump to a short label.
Conditional jumps	Acts only if certain conditions met. Necessary if several conditions must be tested. See "Conditional Jumps".

The following examples illustrate these loop constructions.

```
; The LOOP instruction: For 200 to 0 do task
        mov    ecx, 200        ; Set counter
next:   .                      ; Do the task here
        .
        .
        loop   next            ; Do again
```

```
                                        ; Continue after loop

; The LOOPNE instruction: While EAX is not 'Y', do task
        mov     ecx, 256                ; Set count too high to interfere
wend:   .                               ; But don't do more than 256 times
        .                               ; Some statements that change EAX
        .
    cmp     al, 'Y'                     ; Is it Y or too many times?
    loopne  wend                        ; No? Repeat
                                        ; Yes? Continue
```

The **JECXZ** instruction provides an efficient way to avoid executing loops when the loop counter ECX is empty. For example, consider the following loops:

```
        mov     ecx, LoopCount          ; Load loop counter
next:   .                               ; Iterate loop CX times
        .
        .
        loop    next                    ; Do again
```

If `LoopCount` is zero, ECX decrements to -1 on the first pass. It then must decrement 2^32 more times before reaching 0. Use a **JECXZ** to avoid this problem:

```
        mov     ecx, LoopCount          ; Load loop counter
        jcxz    done                    ; Skip loop if count is 0
next:   .                               ; Else iterate loop ECX times
        .
        .
        loop    next                    ; Do again
done:                                   ; Continue after loop
```

JUMPS

Jumps are the most direct way to change program control from one location to another. At the processor level, jumps work by changing the value of the IP (Instruction Pointer) register to a target offset and, for far jumps, by changing the CS register to a new segment address. Jump instructions fall into only two categories: conditional and unconditional.

Unconditional Jumps

The **JMP** instruction transfers control unconditionally to another instruction. **JMP**'s single operand contains the address of the target instruction.

Unconditional jumps skip over code that should not be executed, as shown here:

```
; Handle one case
label1: .
        .
        .
        jmp continue

; Handle second case
label2: .
        .
        .
        jmp continue
        .
        .
        .
continue:
```

The distance of the target from the jump instruction and the size of the operand determine the assembler's encoding of the instruction. The longer the distance, the more bytes the assembler uses to code the instruction.

Conditional Jumps

The most common way to transfer control in assembly language is to use a conditional jump. This is a two-step process:

1. First test the condition.

2. Then jump if the condition is true or continue if it is false.

All conditional jumps except two (**JCXZ** and **JECXZ**) use the processor flags for their criteria. Thus, any statement that sets or clears a flag can serve as a test basis for a conditional jump. The jump statement can be any one of 30 conditional-jump instructions. A conditional-jump instruction takes a single operand containing the target address. You cannot use a pointer value as a target as you can with unconditional jumps.

Jumping Based on the CX Register

JCXZ and **JECXZ** are special conditional jumps that do not consult the processor flags. Instead, as their names imply, these instructions cause a jump only if the CX or ECX register is zero. The use of **JCXZ** and **JECXZ** with program loops is covered in the next section, "Loops."

Jumping Based on the Processor Flags

The remaining conditional jumps in the processor's repertoire all depend on the status of the flags register. As the following list shows, several conditional jumps have two or three names—**JE** (Jump if Equal) and **JZ** (Jump if Zero), for example. Shared names assemble to exactly the

same machine instruction, so you may choose whichever mnemonic seems more appropriate. Jumps that depend on the status of the flags register include:

Instruction	Jumps if
JC/JB/JNAE	Carry flag is set
JNC/JNB/JAE	Carry flag is clear
JBE/JNA	Either carry or zero flag is set
JA/JNBE	Carry and zero flag are clear
JE/JZ	Zero flag is set
JNE/JNZ	Zero flag is clear
JL/JNGE	Sign flag ≠overflow flag
JGE/JNL	Sign flag = overflow flag
JLE/JNG	Zero flag is set or sign ≠overflow
JG/JNLE	Zero flag is clear and sign = overflow
JS	Sign flag is set
JNS	Sign flag is clear
JO	Overflow flag is set
JNO	Overflow flag is clear
JP/JPE	Parity flag is set (even parity)
JNP/JPO	Parity flag is clear (odd parity)

The last two jumps in the list, **JPE** (Jump if Parity Even) and **JPO** (Jump if Parity Odd), are useful only for communications programs. The processor sets the parity flag if an operation produces a result with an even number of set bits. A communications program can compare the flag against the parity bit received through the serial port to test for transmission errors.

The conditional jumps in the preceding list can follow any instruction that changes the processor flags, as these examples show:

```
; Uses JO to handle overflow condition
      add    ax, bx         ; Add two values
      jo     overflow       ; If value too large, adjust

; Uses JNZ to check for zero as the result of subtraction
      sub    ax, bx         ; Subtract
      mov    cx, Count      ; First, initialize CX
```

```
jnz     skip          ; If the result is not zero, continue
call    zhandler      ; Else do special case
```

As the second example shows, the jump does not have to immediately follow the instruction that alters the flags. Since **MOV** does not change the flags, it can appear between the **SUB** instruction and the dependent jump.

There are three categories of conditional jumps:

- Comparison of two values
- Individual bit settings in a value
- Whether a value is zero or nonzero

Jumps Based on Comparison of Two Values

The **CMP** instruction is the most common way to test for conditional jumps. It compares two values without changing either, then sets or clears the processor flags according to the results of the comparison.

Internally, the **CMP** instruction is the same as the **SUB** instruction, except that **CMP** does not change the destination operand. Both set flags according to the result of the subtraction.

You can compare signed or unsigned values, but you must choose the subsequent conditional jump to reflect the correct value type. For example, **JL** (Jump if Less Than) and **JB** (Jump if Below) may seem conceptually similar, but a failure to understand the difference between them can result in program bugs. Table 7.1 shows the correct conditional jumps for comparisons of signed and unsigned values. The table shows the zero, carry, sign, and overflow flags as ZF, CF, SF, and OF, respectively.

Signed Comparisons Instruction	 Jump if True	Unsigned Comparisons Instruction	 Jump if True
JE	ZF = 1	**JE**	ZF = 1
JNE	ZF = 0	**JNE**	ZF = 0
JG/JNLE	ZF = 0 and SF = OF	**JA/JNBE**	CF = 0 and ZF = 0
JLE/JNG	ZF = 1 or SF ≠OF	**JBE/JNA**	CF = 1 or ZF = 1
JL/JNGE	SF ≠OF	**JB/JNAE**	CF = 1
JGE/JNL	SF = OF	**JAE/JNB**	CF = 0

Table 7.1 Conditional Jumps Based on Comparisons of Two Values

The mnemonic names of jumps always refer to the comparison of **CMP**'s first operand (destination) with the second operand (source). For instance, in this example, **JG** tests whether the first operand is greater than the second.

```
cmp     ax, bx ; Compare AX and BX
jg      next1  ; Equivalent to: If ( AX > BX ) goto next1

jl      next2  ; Equivalent to: If ( AX < BX ) goto next2
```

Jumps Based on Bit Settings

The individual bit settings in a single value can also serve as the criteria for a conditional jump. The **TEST** instruction tests whether specific bits in an operand are on or off (set or clear), and sets the zero flag accordingly.

The **TEST** instruction is the same as the **AND** instruction, except that **TEST** changes neither operand. The following example shows an application of **TEST**.

```
        __int8    bits;    // 8-bit variable
        .
        .
    ; If bit 2 or bit 4 is set, then call task_a
                            ; Assume "bits" is 0D3h    11010011
        test    bits, 10100y    ; If 2 or 4 is set     AND 00010100
        jz      skip1           ;                          --------
        call    task_a          ; Then call task_a         00010000
    skip1:                      ; Jump taken
        .
        .
        .
    ; If bits 2 and 4 are clear, then call task_b
                            ; Assume "bits" is 0E9h    11101001
        test    bits, 10100y    ; If 2 and 4 are clear AND 00010100
        jnz     skip2           ;                          --------
        call    task_b          ; Then call task_b         00000000
    skip2:                      ; Jump taken
```

The source operand for **TEST** is often a mask in which the test bits are the only bits set. The destination operand contains the value to be tested. If all the bits set in the mask are clear in the destination operand, **TEST** sets the zero flag. If any of the flags set in the mask are also set in the destination operand, **TEST** clears the zero flag.

The **BT** (Bit Test) series of instructions copy a specified bit from the destination operand to the carry flag. A **JC** or **JNC** can then route program flow depending on the result. For variations on the **BT** instruction, see the *Reference*.

Jumps Based on a Value of Zero

A program often needs to jump based on whether a particular register contains a value of zero. We've seen how the **JCXZ** instruction jumps depending on the value in the CX register. You can test for zero in other data registers nearly as efficiently with the **OR** instruction. A program can **OR** a register with itself without changing the register's contents, then act on the resulting flags status. For example, the following example tests whether BX is zero:

```
or      bx, bx          ; Is BX = 0?
jz      is_zero         ; Jump if so
```

This code is functionally equivalent to:

```
cmp     bx, 0           ; Is BX = 0?
je      is_zero         ; Jump if so
```

but produces smaller and faster code, since it does not use an immediate number as an operand. The same technique also lets you test a register's sign bit:

```
or      dx, dx          ; Is DX sign bit set?
js      sign_set        ; Jump if so
```