ESTRUCTURA DE COMPUTADORS I
(EC-1)
Enginyeria Informàtica

Manual del 386

Carlos Álvarez
Cristina Barrado
Luis Díaz de Cerio
Montse Fernández
David López
Àngel Olivé
Joan Manel Parcerisa
Jordi Tubella
Miguel Valero

Dept. d'Arquitectura de Computadors

INDEX

Chapter 1. 80386 Architectural Elements

The basic 80386 programming model consists of these architectural elements:

- Memory organization and segmentation
- Data types
- Registers
- Instruction format
- Operand selection
- Input/Output
- Interrupt handling

This chapter contains a section for each aspect of the architecture that is normally visible to applications.

1.1 Memory Organization and Segmentation

The physical memory of an 80386 system is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address that ranges from zero to a maximum of 2^(32) -1 (4 gigabytes).

80386 programs, however, are independent of the physical address space. This means that programs can be written without knowledge of how much physical memory is available and without knowledge of exactly where in physical memory the instructions and data are located.

The model of memory organization seen by applications programmers is determined by systems-software designers. The architecture of the 80386 gives designers the freedom to choose a model for each task. The model of memory organization can range between the following extremes:

- A "flat" address space consisting of a single array of up to 4 gigabytes.
- A segmented address space consisting of a collection of up to 16,383 linear address spaces of up to 4 gigabytes each.

1.1.1 The "Flat" Model

In a "flat" model of memory organization, the applications programmer sees a single array of up to 2^(32) bytes (4 gigabytes). While the physical memory can contain up to 4 gigabytes, it is usually much smaller; the processor maps the 4 gigabyte flat space onto the physical address space by the address translation mechanisms. Applications programmers do not need to know the details of the mapping.

A pointer into this flat address space is a 32-bit ordinal number that may range from 0 to 2^(32) -1. Relocation of separately-compiled modules in this space must be performed by systems software (e.g., linkers, locators, binders, loaders).

1.1.2 The Segmented Model

In a segmented model of memory organization, the address space as viewed by an applications program (called the logical address space) is a much larger terabyte logical address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms. Applications programmers do not need to know the details of this mapping.

Applications programmers view the logical address space of the 80386 as a collection of up to 16,383 one-dimensional subspaces, each with a specified length. Each of these linear subspaces is called a segment. A segment is a unit of contiguous address space. Segment sizes may range from one byte up to a maximum of 2^(32) bytes (4 gigabytes).

A complete pointer in this address space consists of two parts:

1. A segment selector, which is a 16-bit field that identifies a segment.
2. An offset, which is a 32-bit ordinal that addresses to the byte level within a segment.

During execution of a program, the processor associates with a segment selector the physical address of the beginning of the segment. Separately compiled modules can be relocated at run time by changing the base address of their segments. The size of a segment is variable; therefore, a segment can be exactly the size of the module it contains.

1.1.3.1 Physical Address Formation (in Real-Address mode)

The 80386 has three processing modes:

1. Protected Mode.
2. Real-Address Mode.
3. Virtual 8086 Mode.

Protected mode is the natural 32-bit environment of the 80386 processor. In this mode all instructions and features are available.

Real-Address mode (often called just “real mode”) is the mode of the processor immediately after RESET. In real mode the 80386 appears to programmers as a fast 8086 with some new instructions. Most applications of the 80386 will use real mode for initialization only.

Virtual 8086 mode (also called V86 mode) is a dynamic mode in the sense that the processor can switch repeatedly and rapidly between V86 mode and protected mode. The CPU enters V86 mode from protected mode to execute an 8086 program, then leaves V86 mode and enters protected mode to continue executing a native 80386 program.

Along the course we will consider the segmented address space view, using the 80386 microprocessor in Real-Address mode. This publication describes those aspects that affect only to the 80386 programming in Real-Address mode.

Segment relocation in Real-Address mode is performed as in the 8086: the 16-bit value in a segment selector is shifted left by four bits to form the base address of a segment. The effective address is extended with four high order zeros and added to the base to form a linear address. Unlike the 8086 and 80286, 32-bit effective addresses can be generated.

1.2 Data Types

Bytes, words, and doublewords are the fundamental data types. A byte is eight contiguous bits starting at any logical address. The bits are numbered 0 through 7; bit zero is the least significant bit.

A word is two contiguous bytes starting at any byte address. A word thus contains 16 bits. The bits of a word are numbered from 0 through 15; bit 0 is the least significant bit. The byte containing bit 0 of the word is called the low byte; the byte containing bit 15 is called the high byte.

Each byte within a word has its own address, and the smaller of the addresses is the address of

the word. The byte at this lower address contains the eight least significant bits of the word, while the byte at the higher address contains the eight most significant bits.

A doubleword is two contiguous words starting at any byte address. A doubleword thus contains 32 bits. The bits of a doubleword are numbered from 0 through 31; bit 0 is the least significant bit. The word containing bit 0 of the doubleword is called the low word; the word containing bit 31 is called the high word.

Each byte within a doubleword has its own address, and the smallest of the addresses is the address of the doubleword. The byte at this lowest address contains the eight least significant bits of the doubleword, while the byte at the highest address contains the eight most significant bits.

Note that words need not be aligned at even-numbered addresses and doublewords need not be aligned at addresses evenly divisible by four. This allows maximum flexibility in data structures (e.g., records containing mixed byte, word, and doubleword items) and efficiency in memory utilization. When used in a configuration with a 32-bit bus, actual transfers of data between processor and memory take place in units of doublewords beginning at addresses evenly divisible by four; however, the processor converts requests for misaligned words or doublewords into the appropriate sequences of requests acceptable to the memory interface. Such misaligned data transfers reduce performance by requiring extra memory cycles. For maximum performance, data structures (including stacks) should be designed in such a way that, whenever possible, word operands are aligned at even addresses and doubleword operands are aligned at addresses evenly divisible by four. Due to instruction prefetching and queuing within the CPU, there is no requirement for instructions to be aligned on word or doubleword boundaries. (However, a slight increase in speed results if the target addresses of control transfers are evenly divisible by four.)

Although bytes, words, and doublewords are the fundamental types of operands, the processor also supports additional interpretations of these operands. Depending on the instruction referring to the operand, the following additional data types are recognized:

- Integer:

  A signed binary numeric value contained in a 32-bit doubleword,16-bit word, or 8-bit byte. All operations assume a 2's complement representation. The sign bit is located in bit 7 in a byte, bit 15 in a word, and bit 31 in a doubleword. The sign bit has the value zero for positive integers and one for negative. Since the high-order bit is used for a sign, the range of an 8-bit integer is -128 through +127; 16-bit integers may range from -32,768 through +32,767; 32-bit integers may range from -2^(31) through +2^(31) -1. The value zero has a positive sign.

- Ordinal:

  An unsigned binary numeric value contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. All bits are considered in determining magnitude of the number. The value range of an 8-bit ordinal number is 0-255; 16 bits can represent values from 0 through 65,535; 32 bits can represent values from 0 through 2^(32) -1.

- Pointer:

  A 32-bit logical address of two components: a 16-bit segment selector component and a 16-bit offset component. Pointers are used by applications programmers only when systems designers choose a segmented memory organization.

  We only consider this data type regarding pointers, although several kind of pointers

could be used in the 386 family of microprocessors. This view of pointers enforces the restriction to use segments of 64 Kb. Although this restriction could be overcome with the use of appropriate software, it is not needed to deal with larger segments along the course.

The 16-bit offset of a pointer that is stored in memory must be zero-entended to 32 bits before using it to access the address space.

1.3 Registers

The 80386 contains a total of sixteen registers that are of interest to the applications programmer. These registers may be grouped into these basic categories:

1. General registers. These eight 32-bit general-purpose registers are used primarily to contain operands for arithmetic and logical operations.
2. Segment registers. These special-purpose registers permit systems software designers to choose either a flat or segmented model of memory organization. These six registers determine, at any given time, which segments of memory are currently addressable.
3. Status and instruction registers. These special-purpose registers are used to record and alter certain aspects of the 80386 processor state.

1.3.1 General Registers

The general registers of the 80386 are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used interchangeably to contain the operands of logical and arithmetic operations. They may also be used interchangeably for operands of address computations (except that ESP cannot be used as an index operand).

The low-order word of each of these eight registers has a separate name and can be treated as a unit. This feature is useful for handling 16-bit data items and for compatibility with the 8086 and 80286 processors. The word registers are named AX, BX, CX, DX, BP, SP, SI, and DI.

Each byte of the 16-bit registers AX, BX, CX, and DX has a separate name and can be treated as a unit. This feature is useful for handling characters and other 8-bit data items. The byte registers are named AH, BH, CH, and DH (high bytes); and AL, BL, CL, and DL (low bytes).

All of the general-purpose registers are available for addressing calculations and for the results of most arithmetic and logical calculations; however, a few functions are dedicated to certain registers. By implicitly choosing registers for these functions, the 80386 architecture can encode instructions more compactly. The instructions that use specific registers include: double-precision multiply and divide, I/O, string instructions, translate, loop, variable shift and rotate, and stack operations.

1.3.2 Segment Registers

The segment registers of the 80386 give systems software designers the flexibility to choose among various models of memory organization. Designers may choose a model in which applications programs do not need to modify segment registers, in which case applications programmers may skip this section.

Complete programs generally consist of many different modules, each consisting of instructions and data. However, at any given time during program execution, only a small subset of a program's modules are actually in use. The 80386 architecture takes advantage of this by

providing mechanisms to support direct access to the instructions and data of the current module's environment, with access to additional segments on demand.

At any given instant, six segments of memory may be immediately accessible to an executing 80386 program. The segment registers CS, DS, SS, ES, FS, and GS are used to identify these six current segments. Each of these registers specifies a particular kind of segment, as characterized by the associated mnemonics ("code," "data," or "stack"). Each register uniquely determines one particular segment, from among the segments that make up the program, that is to be immediately accessible at highest speed.

The segment containing the currently executing sequence of instructions is known as the current code segment; it is specified by means of the CS register. The 80386 fetches all instructions from this code segment, using as an offset the contents of the instruction pointer. CS is changed implicitly as the result of intersegment control-transfer instructions (for example, CALL and JMP), interrupts, and exceptions.

Subroutine calls, parameters, and procedure activation records usually require that a region of memory be allocated for a stack. All stack operations use the SS register to locate the stack. Unlike CS, the SS register can be loaded explicitly, thereby permitting programmers to define stacks dynamically.

The DS, ES, FS, and GS registers allow the specification of four data segments, each addressable by the currently executing program. Accessibility to four separate data areas helps programs efficiently access different types of data structures; for example, one data segment register can point to the data structures of the current module, another to the exported data of a higher-level module, another to a dynamically created data structure, and another to data shared with another task. An operand within a data segment is addressed by specifying its offset either directly in an instruction or indirectly via general registers.

Depending on the structure of data (e.g., the way data is parceled into one or more segments), a program may require access to more than four data segments. To access additional segments, the DS, ES, FS, and GS registers can be changed under program control during the course of a program's execution. This simply requires that the program execute an instruction to load the appropriate segment register prior to executing instructions that access the data.

The processor associates a base address with each segment selected by a segment register. To address an element within a segment, a 32-bit offset is added to the segment's base address. Once a segment is selected (by loading the segment selector into a segment register), a data manipulation instruction only needs to specify the offset. Simple rules define which segment register is used to form an address when only an offset is specified.

1.3.3 Stack Implementation

Stack operations are facilitated by three registers:

1. The stack segment (SS) register. Stacks are implemented in memory. A system may have a number of stacks that is limited only by the maximum number of segments. A stack may be up to 4 gigabytes long, the maximum length of a segment. One stack is directly addressable at a time, the one located by SS. This is the current stack, often referred to simply as "the" stack. SS is used automatically by the processor for all stack operations.
2. The stack pointer (ESP) register. ESP points to the top of the push-down stack (TOS). It is referenced implicitly by PUSH and POP operations, subroutine calls and returns, and interrupt operations. When an item is pushed onto the stack, the processor decrements ESP, then writes the item at the new TOS. When an item is popped off the stack, the

processor copies it from TOS, then increments ESP. In other words, the stack grows down in memory toward lesser addresses.

3. The stack-frame base pointer (EBP) register. The EBP is the best choice of register for accessing data structures, variables and dynamically allocated work space within the stack. EBP is often used to access elements on the stack relative to a fixed point on the stack rather than relative to the current TOS. It typically identifies the base address of the current stack frame established for the current procedure. When EBP is used as the base register in an offset calculation, the offset is calculated automatically in the current stack segment (i.e., the segment currently selected by SS). Because SS does not have to be explicitly specified, instruction encoding in such cases is more efficient. EBP can also be used to index into segments addressable via other segment registers.

1.3.4 Flags Register

The flags register is a 32-bit register named EFLAGS. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.

The flags may be considered in three groups: the status flags, the control flags, and the systems flags. System flags are not discussed in this publication.

1.3.4.1 Status Flags

The status flags of the EFLAGS register allow the results of one instruction to influence later instructions. The arithmetic instructions use OF, SF, ZF, AF, PF, and CF. There are instructions to set, clear, and complement CF before execution of an arithmetic instruction.

- CF (Carry Flag, bit 0)

  Set on high-order bit carry or borrow; cleared otherwise.

- PF (Parity Flag, bit 2)

  Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.

- AF (Adjust flag, bit 4)

  Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.

- ZF (Zero Flag, bit 6)

  Set if result is zero; cleared otherwise.

- SF (Sign Flag, bit 7)

  Set equal to high-order bit of result (0 is positive, 1 if negative).

- OF (Overflow Flag, bit 11)

  Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

1.3.4.3 Instruction Pointer

The instruction pointer register (EIP) contains the offset address, relative to the start of the

current code segment, of the next sequential instruction to be executed. The instruction pointer is not directly visible to the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions.

The low-order 16 bits of EIP is named IP and can be used by the processor as a unit. This feature is useful when executing instructions designed for the 8086 and 80286 processors.

1.4 Instruction Format

The information encoded in an 80386 instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction must also select, explicitly or implicitly, which of the currently addressable segments contains the operand.

80386 instructions are composed of various elements and have various formats. The exact format of instructions is shown in Appendix B; the elements of instructions are described below. Of these instruction elements, only one, the opcode, is always present. The other elements may or may not be present, depending on the particular operation involved and on the location and type of the operands. The elements of an instruction, in order of occurrence are as follows:

- Prefixes -- one or more bytes preceding an instruction that modify the operation of the instruction. The following types of prefixes can be used by applications programs:
  1. Segment override -- explicitly specifies which segment register an instruction should use, thereby overriding the default segment-register selection used by the 80386 for that instruction.
  2. Address size -- switches between 32-bit and 16-bit address generation.
  3. Operand size -- switches between 32-bit and 16-bit operands.
- Opcode -- specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different variant of the operation.
- Register specifier -- an instruction may specify one or two register operands. Register specifiers may occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.
- Addressing-mode specifier -- when present, specifies whether an operand is a register or memory location; if in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.
- SIB (scale, index, base) byte -- when the addressing-mode specifier indicates that an index register will be used to compute the address of an operand, an SIB byte is included in the instruction to encode the base register, the index register, and a scaling factor.
- Displacement -- when the addressing-mode specifier indicates that a displacement will be used to compute the address of an operand, the displacement is encoded in the instruction. A displacement is a signed integer of 32, 16, or eight bits. The eight-bit form is used in the common case when the displacement is sufficiently small. The processor extends an eight-bit displacement to 16 or 32 bits, taking into account the sign.
- Immediate operand -- when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide. In cases where an eight-bit immediate operand is combined in some way with a 16- or 32-bit operand, the processor automatically extends the size of the eight-bit operand, taking into account the sign.

1.5 Operand Selection

An instruction can act on zero or more operands, which are the data manipulated by the instruction. An example of a zero-operand instruction is NOP (no operation). An operand can be in any of these locations:

- In the instruction itself (an immediate operand)
- In a register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP in the case of 32-bit operands; AX, BX, CX, DX, SI, DI, SP, or BP in the case of 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL in the case of 8-bit operands; the segment registers; or the EFLAGS register for flag operations)
- In memory
- At an I/O port

Immediate operands and operands in registers can be accessed more rapidly than operands in memory since memory operands must be fetched from memory. Register operands are available in the CPU. Immediate operands are also available in the CPU, because they are prefetched as part of the instruction.

Of the instructions that have operands, some specify operands implicitly; others specify operands explicitly; still others use a combination of implicit and explicit specification; for example:

Implicit operand: CWD

The source operand is AX. The destination operands are DX and AX.

Explicit operand: XCHG EAX, EBX

The operands to be exchanged are encoded in the instruction after the opcode.

Implicit and explicit operands: PUSH COUNTER

The memory variable COUNTER (the explicit operand) is copied to the top of the stack (the implicit operand).

Note that most instructions have implicit operands. All arithmetic instructions, for example, update the EFLAGS register.

An 80386 instruction can explicitly reference one or two operands. Two-operand instructions, such as MOV, ADD, XOR, etc., generally overwrite one of the two participating operands with the result. A distinction can thus be made between the source operand (the one unaffected by the operation) and the destination operand (the one overwritten by the result).

For most instructions, one of the two explicitly specified operands -- either the source or the destination -- can be either in a register or in memory. The other operand must be in a register or be an immediate source operand. Thus, the explicit two-operand instructions of the 80386 permit operations of the following kinds:

- Register-to-register
- Register-to-memory
- Memory-to-register
- Immediate-to-register
- Immediate-to-memory

Stack manipulation instructions, however, transfer data from memory to memory. Push and pop stack operations allow transfer between memory operands and the memory-based stack.

1.5.1 Immediate Operands

Certain instructions use data from the instruction itself as one (and sometimes two) of the operands. Such an operand is called an immediate operand. The operand may be 32-, 16-, or 8-bits long. For example:

```
SHR PATTERN, 2
```

One byte of the instruction holds the value 2, the number of bits by which to shift the variable PATTERN.

```
TEST PATTERN, 0FFFF00FFH
```

A doubleword of the instruction holds the mask that is used to test the variable PATTERN.

1.5.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL,or DL).

The 80386 has instructions for referencing the segment registers (CS, DS, ES, SS, FS, GS). These instructions are used by applications programs only if systems designers have chosen a segmented memory model.

The 80386 also has instructions for referring to the flag register. The flags may be stored on the stack and restored from the stack. Certain instructions change the commonly modified flags directly in the EFLAGS register. Other flags that are seldom modified can be modified indirectly via the flags image in the stack.

1.5.3 Memory Operands

Data-manipulation instructions that address operands in memory must specify (either directly or indirectly) the segment that contains the operand and the offset of the operand within the segment. However, for speed and compact instruction encoding, segment selectors are stored in the high speed segment registers. Therefore, data-manipulation instructions need to specify only the desired segment register and an offset in order to address a memory operand.

An 80386 data-manipulation instruction that accesses memory uses one of the following methods for specifying the offset of a memory operand within its segment:

1. Most data-manipulation instructions that access memory contain a byte that explicitly specifies the addressing method for the operand. A byte, known as the modR/M byte, follows the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement. When an index register is used, the modR/M byte is also followed by another byte that identifies the index register and scaling factor. This addressing method is the most flexible.
2. A few data-manipulation instructions implicitly use specialized addressing methods:
   - For a few short forms of MOV that implicitly use the EAX register, the offset of the operand is coded as a doubleword in the instruction. No base register, index register, or scaling factor are used.

- Stack operations implicitly address operands via SS:ESP registers; e.g., PUSH, POP, CALL, RET, IRET, exceptions, and interrupts.

1.5.3.1 Segment Selection

Data-manipulation instructions need not explicitly specify which segment register is used. For all of these instructions, specification of a segment register is optional. For all memory accesses, if a segment is not explicitly specified by the instruction, the processor automatically chooses a segment register according to the following rules. (If systems designers have chosen a flat model of memory organization, the segment registers and the rules that the processor uses in choosing them are not apparent to applications programs.):

Memory Reference Needed	Segment Register Used	Implicit Segment Selection Rule
Instructions	Code (CS)	Automatic with instruction prefetch
Stack	Stack (SS)	All stack pushes and pops. Any memory reference that uses ESP or EBP as a base register.
Local Data	Data (DS)	All data references except when relative to stack.

There is a close connection between the kind of memory reference and the segment in which that operand resides. As a rule, a memory reference implies the current data segment (i.e., the implicit segment selector is in DS). However, ESP and EBP are used to access items on the stack; therefore, when the ESP or EBP register is used as a base register, the current stack segment is implied (i.e., SS contains the selector).

Special instruction prefix elements may be used to override the default segment selection. Segment-override prefixes allow an explicit segment selection. The 80386 has a segment-override prefix for each of the segment registers. Only in the following special cases is there an implied segment selection that a segment prefix cannot override:

- The use of SS in stack instructions.
- The use of CS for instruction fetches.

1.5.3.2 Effective-Address Computation

The modR/M byte provides the most flexible of the addressing methods, and instructions that require a modR/M byte as the second byte of the instruction are the most common in the 80386 instruction set. For memory operands defined by modR/M, the offset within the desired segment is calculated by taking the sum of up to three components:

- A displacement element in the instruction.
- A base register.
- An index register. The index register may be automatically multiplied by a scaling factor of 2, 4, or 8.

The offset that results from adding these components is called an effective address. Each of these components of an effective address may have either a positive or negative value. If the sum of all the components exceeds 2^(32), the effective address is truncated to 32 bits.

The displacement component, because it is encoded in the instruction, is useful for fixed aspects of addressing; for example:

- Location of simple scalar operands.
- Beginning of a statically allocated array.
- Offset of an item within a record.

The base and index components have similar functions. Both utilize the same set of general registers. Both can be used for aspects of addressing that are determined dynamically; for example:

- Location of procedure parameters and local variables in stack.
- The beginning of one record among several occurrences of the same record type or in an array of records.
- The beginning of one dimension of multiple dimension array.
- The beginning of a dynamically allocated array.

The uses of general registers as base or index components differ in the following respects:

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

The scaling factor permits efficient indexing into an array in the common cases when array elements are 2, 4, or 8 bytes wide. The shifting of the index register is done by the processor at the time the address is evaluated with no performance loss. This eliminates the need for a separate shift or multiply instruction.

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

BASE	+	(INDEX * SCALE)		+	DISPLACEMENT

EAX		EAX	1		
ECX		ECX			
EDX		EDX	2		NO DISPLACEMENT
EBX		EBX			8-BIT DISPLACEMENT
ESP		---	4		32-BIT DISPLACEMENT
EBP		EBP			
ESI		ESI	8		
EDI		EDI			

Following are possible uses for some of the various combinations of address components.

DISPLACEMENT

The displacement alone indicates the offset of the operand. This combination is used to directly address a statically allocated scalar operand. An 8-bit, 16-bit, or 32-bit displacement can be used.

BASE

The offset of the operand is specified indirectly in one of the general registers, as for "based" variables.

BASE + DISPLACEMENT

A register and a displacement can be used together for two distinct purposes:

1. Index into static array when element size is not 2, 4, or 8 bytes. The displacement component encodes the offset of the beginning of the array. The register holds the results of a calculation to determine the offset of a specific element within the array.
2. Access item of a record. The displacement component locates an item within record. The base register selects one of several occurrences of record, thereby providing a compact encoding for this common function.

An important special case of this combination, is to access parameters in the procedure activation record in the stack. In this case, EBP is the best choice for the base register, because when EBP is used as a base register, the processor automatically uses the stack segment register (SS) to locate the operand, thereby providing a compact encoding for this common function.

(INDEX * SCALE) + DISPLACEMENT

This combination provides efficient indexing into a static array when the element size is 2, 4, or 8 bytes. The displacement addresses 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

Two registers used together support either a two-dimensional array (the displacement determining the beginning of the array) or one of several instances of an array of records (the displacement indicating an item in the record).

BASE + (INDEX * SCALE) + DISPLACEMENT

This combination provides efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes wide.

1.6 Input/Output

This section presents the I/O features of the 80386 from the following perspectives:

- Methods of addressing I/O ports
- Instructions that cause I/O operations

1.6.1 I/O Addressing

The 80386 allows input/output to be performed in either of two ways:

- By means of a separate I/O address space (using specific I/O instructions)
- By means of memory-mapped I/O (using general-purpose operand manipulation instructions).

1.6.1.1 I/O Address Space

The 80386 provides a separate I/O address space, distinct from physical memory, that can be used to address the input/output ports that are used for external 16 devices. The I/O address space consists of 2^(16) (64K) individually addressable 8-bit ports; any two consecutive 8-bit ports can be treated as a 16-bit port; and four consecutive 8-bit ports can be treated as a 32-bit port. Thus, the I/O address space can accommodate up to 64K 8-bit ports, up to 32K 16-bit ports, or up to 16K 32-bit ports.

The program can specify the address of the port in two ways. Using an immediate byte constant,

the program can specify:

- 256 8-bit ports numbered 0 through 255.
- 128 16-bit ports numbered 0, 2, 4, . . . , 252, 254.
- 64 32-bit ports numbered 0, 4, 8, . . . , 248, 252.

Using a value in DX, the program can specify:

- 8-bit ports numbered 0 through 65535
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

The 80386 can transfer 32, 16, or 8 bits at a time to a device located in the I/O space. Like doublewords in memory, 32-bit ports should be aligned at addresses evenly divisible by four so that the 32 bits can be transferred in a single bus access. Like words in memory, 16-bit ports should be aligned at even-numbered addresses so that the 16 bits can be transferred in a single bus access. An 8-bit port may be located at either an even or odd address.

The instructions IN and OUT move data between a register and a port in the I/O address space.

1.6.1.2 Memory-Mapped I/O

I/O devices also may be placed in the 80386 memory address space. As long as the devices respond like memory components, they are indistinguishable to the processor.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV instruction can transfer data between any register and a port; and the AND, OR, and TEST instructions may be used to manipulate bits in the internal registers of a device. Memory-mapped I/O performed via the full instruction set maintains the full complement of addressing modes for selecting the desired I/O device (e.g., direct address, indirect address, base register, index register, scaling).

1.7 Interrupt Handling

Interrupts and exceptions in 80386 real-address mode work as much as they do on an 8086. Interrupts and exceptions vector to interrupt procedures via an interrupt table. The processor multiplies the interrupt or exception identifier by four to obtain an index into the interrupt table. The entries of the interrupt table are far pointers to the entry points of interrupt or exception handler procedures. When an interrupt occurs, the processor pushes the current values of CS:IP onto the stack, disables interrupts, clears TF (the single-step flag), then transfers control to the location specified in the interrupt table. An IRET instruction at the end of the handler procedure reverses these steps before returning control to the interrupted procedure.

Chapter 2. Instruction Set Classification

This chapter presents an overview of a subset of instructions which programmers can use to write application software for the 80386. The instructions are grouped by categories of related functions.

The instruction dictionary in Appendix B contains more detailed descriptions of these instructions, including encoding, operation, timing, effect on flags, and exceptions.

2.1 Data Movement Instructions

These instructions provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture. They fall into the following classes:

1. General-purpose data movement instructions.
2. Stack manipulation instructions.
3. Type-conversion instructions.

2.1.1 General-Purpose Data Movement Instructions

MOV (Move) transfers a byte, word, or doubleword from the source operand to the destination operand. The MOV instruction is useful for transferring data along any of these paths. There are also variants of MOV that operate on segment registers. These are covered in a later section of this chapter:

- To a register from memory
- To memory from a register
- Between general registers
- Immediate data to a register
- Immediate data to a memory

The MOV instruction cannot move from memory to memory or from segment register to segment register.

XCHG (Exchange) swaps the contents of two operands. This instruction takes the place of three MOV instructions. It does not require a temporary location to save the contents of one operand while load the other is being loaded.

The XCHG instruction can swap two byte operands, two word operands, or two doubleword operands. The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand.

2.1.2 Stack Manipulation Instructions

PUSH (Push) decrements the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP. PUSH is often used to place parameters on the stack before calling a procedure; it is also the basic means of storing temporary variables on the stack. The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers).

POP (Pop) transfers the word or doubleword at the current top of stack (indicated by ESP) to

the destination operand, and then increments ESP to point to the new top of stack. POP moves information from the stack to a general register, or to memory. There are also a variant of POP that operates on segment registers.

2.1.3 Type Conversion Instructions

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into 64-bit items (quad-words). These instructions are especially useful for converting signed integers, because they automatically fill the extra bits of the larger item with the value of the sign bit of the smaller item. This kind of conversion is called sign extension.

There are two classes of type conversion instructions:

1. The forms CWD and CDQ which operate only on data in the EAX register.
2. The forms MOVSX and MOVZX, which permit one operand to be in any general register while permitting the other operand to be in memory or in a register.

CWD (Convert Word to Doubleword) and **CDQ (Convert Doubleword to Quad-Word)** double the size of the source operand. CWD extends the sign of the word in register AX throughout register DX. CDQ extends the sign of the doubleword in EAX throughout EDX. CWD can be used to produce a doubleword dividend from a word before a word division, and CDQ can be used to produce a quad-word dividend from a doubleword before doubleword division.

MOVSX (Move with Sign Extension) sign-extends an 8-bit value to a 16-bit value and a 8- or 16-bit value to 32-bit value.

MOVZX (Move with Zero Extension) extends an 8-bit value to a 16-bit value and an 8- or 16-bit value to 32-bit value by inserting high-order zeros.

2.2 Binary Arithmetic Instructions

The arithmetic instructions of the 80386 processor simplify the manipulation of numeric data that is encoded in binary. Operations include the standard add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign. Both signed and unsigned binary integers are supported. The binary arithmetic instructions may also be used as one step in the process of performing arithmetic on decimal integers.

Many of the arithmetic instructions operate on both signed and unsigned integers. These instructions update the flags ZF, CF, SF, and OF in such a manner that subsequent instructions can interpret the results of the arithmetic as either signed or unsigned. CF contains information relevant to unsigned integers; SF and OF contain information relevant to signed integers. ZF is relevant to both signed and unsigned integers; ZF is set when all bits of the result are zero.

If the integer is unsigned, CF may be tested after one of these arithmetic operations to determine whether the operation required a carry or borrow of a one-bit in the high-order position of the destination operand. CF is set if a one-bit was carried out of the high-order position (addition instructions ADD and ADC) or if a one-bit was carried (i.e. borrowed) into the high-order bit (subtraction instructions SUB, SBB, CMP, and NEG).

If the integer is signed, both SF and OF should be tested. SF always has the same value as the sign bit of the result. The most significant bit (MSB) of a signed integer is the bit next to the sign -- bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword. OF is set in either of these cases:

- A one-bit was carried out of the MSB into the sign bit but no one bit was carried out of the sign bit (addition instructions ADD, ADC and INC). In other words, the result was greater than the greatest positive number that could be contained in the destination operand.
- A one-bit was carried from the sign bit into the MSB but no one bit was carried into the sign bit (subtraction instructions SUB, SBB, DEC, CMP, and NEG). In other words, the result was smaller that the smallest negative number that could be contained in the destination operand.

These status flags are tested by executing a conditional instruction: Jcc (jump on condition cc).

2.2.1 Addition and Subtraction Instructions

ADD (Add Integers) replaces the destination operand with the sum of the source and destination operands. Sets CF if overflow.

ADC (Add Integers with Carry) sums the operands, adds one if CF is set, and replaces the destination operand with the result. If CF is cleared, ADC performs the same operation as the ADD instruction. An ADD followed by multiple ADC instructions can be used to add numbers longer than 32 bits.

INC (Increment) adds one to the destination operand. INC does not affect CF. Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.

SUB (Subtract Integers) subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. The operands may be signed or unsigned bytes, words, or doublewords.

SBB (Subtract Integers with Borrow) subtracts the source operand from the destination operand, subtracts 1 if CF is set, and returns the result to the destination operand. If CF is cleared, SBB performs the same operation as SUB. SUB followed by multiple SBB instructions may be used to subtract numbers longer than 32 bits. If CF is cleared, SBB performs the same operation as SUB.

DEC (Decrement) subtracts 1 from the destination operand. DEC does not update CF. Use SUB with an immediate value of 1 to perform a decrement that affects carry.

2.2.2 Comparison and Sign Change Instruction

CMP (Compare) subtracts the source operand from the destination operand. It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands. A subsequent Jcc instruction can test the appropriate flags.

NEG (Negate) subtracts a signed integer operand from zero. The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.

2.2.3 Multiplication Instructions

The 80386 has separate multiply instructions for unsigned and signed operands. MUL operates on unsigned numbers, while IMUL operates on signed integers as well as unsigned.

MUL (Unsigned Integer Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, the processor multiplies it by the contents of AL and returns the double-length result to AH and AL. If the source operand is a word, the processor multiplies it by the contents of AX and returns the double-length result to DX and AX. If the source operand is a doubleword, the processor multiplies it by the contents of EAX

and returns the 64-bit result in EDX and EAX. MUL sets CF and OF when the upper half of the result is nonzero; otherwise, they are cleared.

IMUL (Signed Integer Multiply) performs a signed multiplication operation. IMUL has three variations:

1. A one-operand form. The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.
2. A two-operand form. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.
3. A three-operand form; two are source and one is the destination operand. One of the source operands is an immediate value stored in the instruction; the second may be in memory or in any general register. The product may be stored in any general register. The immediate operand is treated as signed. If the immediate operand is a byte, the processor automatically sign-extends it to the size of the second operand before performing the multiplication.

The three forms are similar in most respects:

- The length of the product is calculated to twice the length of the operands.
- The CF and OF flags are set when significant bits are carried into the high-order half of the result. CF and OF are cleared when the high-order half of the result is the sign-extension of the low-order half.

However, forms 2 and 3 differ in that the product is truncated to the length of the operands before it is stored in the destination register. Because of this truncation, OF should be tested to ensure that no significant bits are lost.

Forms 2 and 3 of IMUL may also be used with unsigned operands because, whether the operands are signed or unsigned, the low-order half of the product is the same.

2.2.4 Division Instructions

The 80386 has separate division instructions for unsigned and signed operands. DIV operates on unsigned numbers, while IDIV operates on signed integers as well as unsigned. In either case, an exception (interrupt zero) occurs if the divisor is zero or if the quotient is too large for AL, AX, or EAX.

DIV (Unsigned Integer Divide) performs an unsigned division of the accumulator by the source operand. The dividend (the accumulator) is twice the size of the divisor (the source operand); the quotient and remainder have the same size as the divisor, as the following table shows.

Size of Source Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
Doubleword	EDX:EAX	EAX	EDX

Non-integral quotients are truncated to integers toward 0. The remainder is always less than the divisor. For unsigned byte division, the largest quotient is 255. For unsigned word division, the largest quotient is 65,535. For unsigned doubleword division the largest quotient is 2^(32) -1.

IDIV (Signed Integer Divide) performs a signed division of the accumulator by the source

operand. IDIV uses the same registers as the DIV instruction.

For signed byte division, the maximum positive quotient is +127, and the minimum negative quotient is -128. For signed word division, the maximum positive quotient is +32,767, and the minimum negative quotient is -32,768. For signed doubleword division the maximum positive quotient is 2^(31) -1, the minimum negative quotient is -2^(31). Non-integral results are truncated towards 0. The remainder always has the same sign as the dividend and is less than the divisor in magnitude.

2.3 Logical Instructions

The group of logical instructions includes:

- The Boolean operation instructions
- Bit test and modify instructions
- Bit scan instructions
- Rotate and shift instructions
- Byte set on condition

2.3.1 Boolean Operation Instructions

The logical operations are AND, OR, XOR, and NOT.

NOT (Not) inverts the bits in the specified operand to form a one's complement of the operand. The NOT instruction is a unary operation that uses a single operand in a register or memory. NOT has no effect on the flags.

The **AND**, **OR**, and **XOR** instructions perform the standard logical operations "and", "(inclusive) or", and "exclusive or". These instructions can use the following combinations of operands:

- Two register operands
- A general register operand with a memory operand
- An immediate operand with either a general register operand or a memory operand.

AND, OR, and XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.

2.3.2 Bit Test Instruction

This instruction operates on a single bit which can be in memory or in a general register. The location of the bit is specified as an offset from the low-order end of the operand. The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register.

This instruction assign the value of the selected bit to CF, the carry flag. OF, SF, ZF, AF, PF are left in an undefined state.

2.3.3 Shift and Rotate Instructions

The shift and rotate instructions reposition the bits within the specified operand.

These instructions fall into the following classes:

- Shift instructions

- Rotate instructions

2.3.3.1 Shift Instructions

The bits in bytes, words, and doublewords may be shifted arithmetically or logically. Depending on the value of a specified count, bits can be shifted up to 31 places.

A shift instruction can specify the count in one of three ways. One form of shift instruction implicitly specifies the count as a single shift. The second form specifies the count as an immediate value. The third form specifies the count as the value contained in CL. This last form allows the shift count to be a variable that the program supplies during execution. Only the low order 5 bits of CL are used.

CF always contains the value of the last bit shifted out of the destination operand. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation. Otherwise, OF is cleared. Following a multibit shift, however, the content of OF is always undefined.

The shift instructions provide a convenient way to accomplish division or multiplication by binary power. Note however that division of signed numbers by shifting right is not the same kind of division performed by the IDIV instruction.

SAL (Shift Arithmetic Left) shifts the destination byte, word, or doubleword operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor shifts zeros in from the right (low-order) side of the operand as bits exit from the left (high-order) side.

SHL (Shift Logical Left) is a synonym for SAL (refer to SAL):

	OF	CF	OPERAND
BEFORE SHL OR SAL	X	X	`10001000100010001000100010001111`
AFTER (BY 1)1	1	1 ←	`00010001000100010001000100011110` ← 0
AFTER (BY 10)	X	0 ←	`00100010001000100011110000000000` ← 0

SHR (Shift Logical Right) shifts the destination byte, word, or doubleword operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor shifts zeros in from the left side of the operand as bits exit from the right side:

		OPERAND	CF
BEFORE SHR		`10001000100010001000100010001111`	X
AFTER (BY 1)	0 →	`01000100010001000100010001000111` →	1
AFTER (BY 10)	0 →	`00000000001000100010001000100010` →	0

SAR (Shift Arithmetic Right) shifts the destination byte, word, or doubleword operand to the right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). The processor preserves the sign of the operand by shifting in zeros on the left (high-order) side if the value is positive or by shifting by ones if the value is negative:

		POSITIVE OPERAND	CF
BEFORE SAR		`01000100010001000100010001000111`	X
AFTER (BY 1)	0 →	`00100010001000100010001000100011` →	1
		NEGATIVE OPERAND	CF
BEFORE SAR		`11000100010001000100010001000111`	X
AFTER (BY 1)	0 →	`11100010001000100010001000100011` →	1

Even though this instruction can be used to divide integers by a power of two, the type of division is not the same as that produced by the IDIV instruction. The quotient of IDIV is rounded toward zero, whereas the "quotient" of SAR is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when IDIV is used to divide -9 by 4, the result is -2 with a remainder of -1. If SAR is used to shift -9 right by two bits, the result is -3. The "remainder" of this kind of division is +3; however, the SAR instruction stores only the high-order bit of the remainder (in CF).

The following code produces the same result as IDIV for any M = 2^(N), where 0 < N < 32.

```
; assuming N is in ECX, and the dividend is in EAX
;
              CMP   EAX, 0        ; to set sign flag
              JGE   NoAdjust      ; jump if sign is zero
              ADD   EAX, ECX      ;
              DEC   EAX           ; EAX := EAX + (N-1)
NoAdjust:     SAR   EAX, CL       ;
                                  ;
```

2.3.3.2 Rotate Instructions

Rotate instructions allow bits in bytes, words, and doublewords to be rotated. Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand.

Rotates affect only the carry and overflow flags. CF may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated and then tested by a conditional jump instruction (JC or JNC). CF always contains the value of the last bit rotated out, even if the instruction does not use this bit as an extension of the rotated operand.

In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared. On multibit rotates, the value of OF is always undefined.

ROL (Rotate Left) rotates the byte, word, or doubleword destination operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). For each rotation specified, the high-order bit that exits from the left of the operand returns at the right to become the new low-order bit of the operand.

ROR (Rotate Right) rotates the byte, word, or doubleword destination operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL). For each rotation specified, the low-order bit that exits from the right of the operand returns at the left to become the new high-order bit of the operand.

RCL (Rotate Through Carry Left) rotates bits in the byte, word, or doubleword destination operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL).

This instruction differs from ROL in that it treats CF as a high-order one-bit extension of the destination operand. Each high-order bit that exits from the left side of the operand moves to CF before it returns to the operand as the low-order bit on the next rotation cycle.

RCR (Rotate Through Carry Right) rotates bits in the byte, word, or doubleword destination operand right by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL).

This instruction differs from ROR in that it treats CF as a low-order one-bit extension of the destination operand. Each low-order bit that exits from the right side of the operand moves to CF before it returns to the operand as the high-order bit on the next rotation cycle.

2.3.4 Test Instruction

TEST (Test) performs the logical "and" of the two operands, clears OF and CF, leaves AF undefined, and updates SF, ZF, and PF. The flags can be tested by conditional control transfer instructions or by the byte-set-on-condition instructions. The operands may be doublewords, words, or bytes.

The difference between TEST and AND is that TEST does not alter the destination operand. TEST differs from BT in that TEST is useful for testing the value of multiple bits in one operations, whereas BT tests a single bit.

2.4 Control Transfer Instructions

The 80386 provides both conditional and unconditional control transfer instructions to direct the flow of execution. Conditional control transfers depend on the results of operations that affect the flag register. Unconditional control transfers are always executed.

2.4.1 Unconditional Transfer Instructions

JMP, CALL, RET and IRET instructions transfer control from one code segment location to another. These locations can be within the same code segment (near control transfers) or in different code segments (far control transfers).

2.4.1.1 Jump Instruction

JMP (Jump) unconditionally transfers control to the target location. JMP is a one-way transfer of execution; it does not save a return address on the stack.

The JMP instruction always performs the same basic function of transferring control from the current location to a new location. Its implementation varies depending on whether the address is specified directly within the instruction or indirectly through a register or memory.

A direct JMP instruction includes the destination address as part of the instruction. An indirect JMP instruction obtains the destination address indirectly through a register or a pointer variable.

2.4.1.2 Call Instruction

CALL (Call Procedure) activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction. CALL places the current value of EIP on the stack. The RET instruction in the called procedure uses this address to transfer control back to the calling program.

CALL instructions, like JMP instructions have direct, and indirect versions.

Indirect CALL instructions specify an absolute address in one of these ways:

1. The program can CALL a location specified by a general register (any of EAX, EDX, ECX, EBX, EBP, ESI, or EDI). The processor moves this 32-bit value into EIP.
2. The processor can obtain the destination address from a memory operand specified in the instruction.

2.4.1.3 Return and Return-From-Interrupt Instruction

RET (Return From Procedure) terminates the execution of a procedure and transfers control

through a back-link on the stack to the program that originally invoked the procedure. RET restores the value of EIP that was saved on the stack by the previous CALL instruction.

RET instructions may optionally specify an immediate operand. By adding this constant to the new top-of-stack pointer, RET effectively removes any arguments that the calling program pushed on the stack before the execution of the CALL instruction.

IRET (Return From Interrupt) returns control to an interrupted procedure. IRET differs from RET in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism.

2.4.2 Conditional Transfer Instructions

The conditional transfer instructions are jumps that may or may not transfer control, depending on the state of the CPU flags when the instruction executes.

2.4.2.1 Conditional Jump Instructions

Table 3-2 shows the conditional transfer mnemonics and their interpretations. The conditional jumps that are listed as pairs are actually the same instruction. The assembler provides the alternate mnemonics for greater clarity within a program listing.

Conditional jump instructions contain a displacement which is added to the EIP register if the condition is true. The displacement may be a byte, a word, or a doubleword. The displacement is signed; therefore, it can be used to jump forward or backward.

Unsigned Conditional Transfers

Mnemonic	Condition Tested	"Jump If..."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

Signed Conditional Transfers

Mnemonic	Condition Tested	"Jump If..."
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

2.5 Flag Control Instructions

The flag control instructions provide a method for directly changing the state of bits in the flag register.

2.5.1 Carry Flag Control Instructions

The carry flag instructions are useful in conjunction with rotate-with-carry instructions RCL and RCR. They can initialize the carry flag, CF, to a known state before execution of a rotate that moves the carry bit into one end of the rotated operand.

Flag Control Instruction	Effect
STC (Set Carry Flag)	CF ← 1
CLC (Clear Carry Flag)	CF ← 0
CMC (Complement Carry Flag)	CF ← NOT (CF)

2.5.2 Interrupt Flag Control Instructions

The IF (interrupt-enable flag) controls the acceptance of external interrupts signalled via the INTR pin. When IF=0, INTR interrupts are inhibited; when IF=1, INTR interrupts are enabled. As with the other flag bits, the processor clears IF in response to a RESET signal. The instructions CLI and STI alter the setting of IF.

CLI (Clear Interrupt-Enable Flag) and **STI (Set Interrupt-Enable Flag)** explicitly alter IF (bit 9 in the flag register).

The IF is also affected implicitly by the following operations:

- The instruction PUSHF stores all flags, including IF, in the stack where they can be examined.
- The instructions POPF and IRET load the flags register; therefore, they can be used to modify IF.

2.6 I/O Instructions

The I/O instructions of the 80386 provide access to the processor's I/O ports for the transfer of data to and from peripheral devices. These instructions have as one operand the address of a port in the I/O address space. There are two classes of I/O instruction:

1. Those that transfer a single item (byte, word, or doubleword) located in a register.
2. Those that transfer strings of items (strings of bytes, words, or doublewords) located in memory. These are known as "string I/O instructions" or "block I/O instructions".

2.6.1 Register I/O Instructions

The I/O instructions IN and OUT are provided to move data between I/O ports and the EAX (32-bit I/O), the AX (16-bit I/O), or AL (8-bit I/O) general registers. IN and OUT instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the DX register to one of up to 64K port addresses.

IN (Input from Port) transfers a byte, word, or doubleword from an input port to AL, AX, or EAX. If a program specifies AL with the IN instruction, the processor transfers 8 bits from the selected port to AL. If a program specifies AX with the IN instruction, the processor transfers 16 bits from the port to AX. If a program specifies EAX with the IN instruction, the processor transfers 32 bits from the port to EAX.

OUT (Output to Port) transfers a byte, word, or doubleword to an output port from AL, AX, or EAX. The program can specify the number of the port using the same methods as the IN instruction.

2.7 Segment Register Instructions

This category actually includes several distinct types of instructions. These various types are grouped together here because, if systems designers choose an unsegmented model of memory organization, none of these instructions is used by applications programmers. The instructions that deal with segment registers are:

1. Segment-register transfer instructions.

   MOV SegReg, ... MOV ..., SegReg PUSH SegReg POP SegReg

2. Control transfers to another executable segment.

   JMP far ; direct and indirect CALL far RET far

The MOV, POP, and PUSH instructions also serve to load and store segment registers. These variants operate similarly to their general-register counterparts except that one operand can be a segment register. MOV cannot move segment register to a segment register. Neither POP nor MOV can place a value in the code-segment register CS; only the far control-transfer instructions can change CS.

The far control-transfer instructions transfer control to a location in another segment by changing the content of the CS register.

2.8 Miscellaneous Instructions

The following instructions do not fit in any of the previous categories, but are nonetheless useful.

2.8.1 Address Calculation Instruction

LEA (Load Effective Address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a general register. This instruction is especially useful for initializing registers before the execution of the string primitives (ESI, EDI) or the XLAT instruction (EBX). The LEA can perform any indexing or scaling that may be needed.

Example: LEA EBX, EBCDIC_TABLE
Causes the processor to place the address of the starting location of the table labeled EBCDIC_TABLE into EBX.

2.8.2 No-Operation Instruction

NOP (No Operation) occupies a byte of storage but affects nothing but the instruction pointer, EIP.

Appendix A. Instruction Format

All instruction encodings are subsets of the general instruction format. Instructions consist of optional instruction prefixes, one or two primary opcode bytes, possibly an address specifier consisting of the ModR/M byte and the SIB (Scale Index Base) byte, a displacement, if required, and an immediate data field, if required.

ADDRESS SIZE PREFIX (0 or 1 byte)
OPERAND SIZE PREFIX (0 or 1 byte)
SEGMENT OVERRIDE (0 or 1 byte)
OPCODE (1 or 2 bytes)
MODR/M (0 or 1 byte)
SIB (0 or 1 byte)
DISPLACEMENT (0, 1, 2 or 4 bytes)
IMMEDIATE (0, 1, 2 or 4 bytes)

Smaller encoding fields can be defined within the primary opcode or opcodes. These fields define the direction of the operation, the size of the displacements, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

Most instructions that can refer to an operand in memory have an addressing form byte following the primary opcode byte(s). This byte, called the ModR/M byte, specifies the address form to be used. Certain encodings of the ModR/M byte indicate a second addressing byte, the SIB (Scale Index Base) byte, which follows the ModR/M byte and is required to fully specify the addressing form.

Addressing forms can include a displacement immediately following either the ModR/M or SIB byte. If a displacement is present, it can be 8-, 16- or 32-bits.

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

A.1 Operand-Size and Address-Size Attributes

When executing an instruction, the 80386 can address memory using either 16 or 32-bit addresses. Consequently, each instruction that uses memory addresses has associated with it an address-size attribute of either 16 or 32 bits. 16-bit addresses imply both the use of a 16-bit displacement in the instruction and the generation of a 16-bit address offset (segment relative address) as the result of the effective address calculation. 32-bit addresses imply the use of a 32-bit displacement and the generation of a 32-bit address offset. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in protected mode) size-specification bits in segment descriptors.

A.1.1 Default Segment Attribute

Programs that execute in real mode have 16-bit addresses and operands by default.

A.1.2 Operand-Size and Address-Size Instruction Prefixes

The internal encoding of an instruction can include two byte-long prefixes: the address-size prefix, 67H, and the operand-size prefix, 66H. These prefixes override the default segment

attributes for the instruction that follows. The following table shows the effect of each possible combination of defaults and overrides.

Operand-Size Prefix 66H	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y
Effective Operand Size	16	16	32	32
Effective Address Size	16	32	16	32

Y = Yes, this instruction prefix is present
N = No, this instruction prefix is not present

A.2 Segment override prefixes

The following are the segment override prefixes:

2EH:	CS segment override prefix
36H:	SS segment override prefix
3EH:	DS segment override prefix
26H:	ES segment override prefix
64H:	FS segment override prefix
65H:	GS segment override prefix

A.3 ModR/M and SIB Bytes

The ModR/M and SIB bytes follow the opcode byte(s) in many of the 80386 instructions. They contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The ModR/M byte contains three fields of information:

MOD (bits 7-6) REG/OPCODE (bits 5-3) R/M (bits 2-0)

- The mod field, which occupies the two most significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes
- The reg field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The r/m field, which occupies the three least significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the field as described above

The based indexed and scaled indexed forms of 32-bit addressing require the SIB byte. The presence of the SIB byte is indicated by certain encodings of the ModR/M byte. The SIB byte then includes the following fields:

SS (bits 7-6) INDEX (bits 5-3) BASE (bits 2-0)

- The ss field, which occupies the two most significant bits of the byte, specifies the scale factor
- The index field, which occupies the next three bits following the ss field and specifies the register number of the index register

- The base field, which occupies the three least significant bits of the byte, specifies the register number of the base register

The following table shows the register addressing forms coded with field REG in the ModR/M byte:

REG =	000	001	010	011	100	101	110	111
r8	AL	CL	DL	BL	AH	CH	DH	BH
r16	AX	CX	DX	BX	SP	BP	SI	DI
r32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI

where r8 or r16/r32 is stated by bit w in the opcode, and r16 or r32 is stated by the operand size.

The following table shows the 16-bit addressing forms coded with MOD and R/M fields in the ModR/M byte:

Effective Address	MOD	R/M
[BX + SI]	00	000
[BX + DI]	00	001
[BP + SI]	00	010
[BP + DI]	00	011
[SI]	00	100
[DI]	00	101
disp16	00	110
[BX]	00	111
[BX+SI]+disp8	01	000
[BX+DI]+disp8	01	001
[BP+SI]+disp8	01	010
[BP+DI]+disp8	01	011
[SI]+disp8	01	100
[DI]+disp8	01	101
[BP]+disp8	01	110
[BX]+disp8	01	111
[BX+SI]+disp16	10	000
[BX+DI]+disp16	10	001
[BX+SI]+disp16	10	010
[BX+DI]+disp16	10	011
[SI]+disp16	10	100
[DI]+disp16	10	101
[BP]+disp16	10	110
[BX]+disp16	10	111
EAX/AX/AL	11	000
ECX/CX/CL	11	001
EDX/DX/DL	11	010
EBX/BX/BL	11	011
ESP/SP/AH	11	100
EBP/BP/CH	11	101
ESI/SI/DH	11	110
EDI/DI/BH	11	111

The following table shows the 32-bit addressing forms coded with MOD and R/M fields in the ModR/M byte.

Effective Address	Mod	R/M
[EAX]	00	000
[ECX]	00	001
[EDX]	00	010
[EBX]	00	011

[--]	00	100
disp32	00	101
[ESI]	00	110
[EDI]	00	111
[EAX]+disp8	01	000
[ECX]+disp8	01	001
[EDX]+disp8	01	010
[EBX]+disp8	01	011
[--]]+disp8	01	100
[EBP]+disp8	01	101
[ESI]+disp8	01	110
[EDI]+disp8	01	111
[EAX]+disp32	10	000
[ECX]+disp32	10	001
[EDX]+disp32	10	010
[EBX]+disp32	10	011
[--]+disp32	10	100
[EBP]+disp32	10	101
[ESI]+disp32	10	110
[EDI]+disp32	10	111
EAX/AX/AL	11	000
ECX/CX/CL	11	001
EDX/DX/DL	11	010
EBX/BX/BL	11	011
ESP/SP/AH	11	100
EBP/BP/CH	11	101
ESI/SI/DH	11	110
EDI/DI/BH	11	111

[--] means a SIB follows the ModR/M byte. disp8 denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index. disp32 denotes a 32-bit displacement following the ModR/M byte, to be added to the index.

The following table shows the 32-bit addressing forms coded with the BASE, INDEX and SS fields in the SIB byte.

BASE =	000	001	010	011	100	101	110	111
r32	EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
Scaled index	INDEX	SS						
[EAX]	00	000						
[ECX]	00	001						
[EDX]	00	010						
[EBX]	00	011						
none	00	100						
[EBP]	00	101						
[ESI]	00	110						
[EDI]	00	111						
[EAX*2]	01	000						
[ECX*2]	01	001						
[EDX*2]	01	010						
[EBX*2]	01	011						
none	01	100						
[EBP*2]	01	101						
[ESI*2]	01	110						
[EDI*2]	01	111						
[EAX*4]	10	000						
[ECX*4]	10	001						

[EDX*4]	10	010
[EBX*4]	10	011
none	10	100
[EBP*4]	10	101
[ESI*4]	10	110
[EDI*4]	10	111
[EAX*8]	11	000
[ECX*8]	11	001
[EDX*8]	11	010
[EBX*8]	11	011
none	11	100
[EBP*8]	11	101
[ESI*8]	11	110
[EDI*8]	11	111

[*] means a disp32 with no base if MOD is 00, [ESP] otherwise. This provides the following addressing modes:
disp32[index] (MOD=00)
disp8[EBP][index] (MOD=01)
disp32[EBP][index] (MOD=10)

Appendix B. Instruction Set

This appendix presents a subset of instructions for the 80386 in alphabetical order. For each instruction, the forms are given for each operand combination, including object code produced, operands required, and a description. For each instruction, there is an operational description and a summary of exceptions generated.

The following is an example of the format used for each 80386 instruction description in this appendix:

CMC -- Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

Operation

CF ← NOT CF;

Description

CMC reverses the setting of the carry flag. No other flags are affected.

Flags Affected

CF as described above

The following sections explain the notational conventions and abbreviations used in these paragraphs of the instruction descriptions.

B.1 Opcode

The “Opcode” column gives the complete object code produced for each form of the instruction. When possible, the codes are given as hexadecimal bytes, in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

/digit: (digit is between 0 and 7) indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction’s opcode.

/r: indicates that the ModR/M byte of the instruction contains both a register operand and an r/m operand.

cb, **cw**, **cd**, **cp**: a 1-byte (cb), 2-byte (cw), 4-byte (cd) or 6-byte (cp) value following the opcode that is used to specify a code offset and possibly a new value for the code segment register.

ib, **iw**, **id**: a 1-byte (ib), 2-byte (iw), or 4-byte (id) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words and doublewords are given with the low-order byte first.

+rb, **+rw**, **+rd**: a register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. The codes are

rb	rw	rd
AL = 0	AX = 0	EAX = 0
CL = 1	CX = 1	ECX = 1
DL = 2	DX = 2	EDX = 2
BL = 3	BX = 3	EBX = 3
AH = 4	SP = 4	ESP = 4
CH = 5	BP = 5	EBP = 5
DH = 6	SI = 6	ESI = 6
BH = 7	DI = 7	EDI = 7

B.2 Instruction

The “Instruction” column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

rel8: a relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.

rel16, **rel32**: a relative address within the same code segment as the instruction assembled. rel16 applies to instructions with an operand-size attribute of 16 bits; rel32 applies to instructions with an operand-size attribute of 32 bits.

ptr16:16: a pointer, typically in a code segment different from that of the instruction. The notation 16:16 indicates that the value of the pointer has two parts. The value to the right of the colon is a 16-bit selector or value destined for the code segment register. The value to the left corresponds to the offset within the destination segment. ptr16:16 is used when the instruction’s operand-size attribute is 16 bits.

r8: one of the byte registers AL, CL, DL, BL, AH, CH, DH, or BH.

r16: one of the word registers AX, CX, DX, BX, SP, BP, SI, or DI.

r32: one of the doubleword registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, or EDI.

imm8: an immediate byte value. imm8 is a signed number between -128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.

imm16: an immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between -32768 and +32767 inclusive.

imm32: an immediate doubleword value used for instructions whose operand-size attribute is 32-bits. It allows the use of a number between +2147483647 and -2147483648.

r/m8: a one-byte operand that is either the contents of a byte register (AL, BL, CL, DL, AH, BH, CH, DH), or a byte from memory.

r/m16: a word register or memory operand used for instructions whose operand-size attribute is 16 bits. The word registers are: AX, BX, CX, DX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation.

r/m32: a doubleword register or memory operand used for instructions whose operand-size attribute is 32-bits. The doubleword registers are: EAX, EBX, ECX, EDX, ESP, EBP,

ESI, EDI. The contents of memory are found at the address provided by the effective address computation.

m16:16: a memory operand containing a pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.

moffs8, **moffs16**, **moffs32**: (memory offset) a simple memory variable of type BYTE, WORD, or DWORD used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.

Sreg: a segment register. The segment register bit assignments are ES=0, CS=1, SS=2, DS=3, FS=4, and GS=5.

B.3 Description

The "Description" column following the "Clocks" column briefly explains the various forms of the instruction. The "Operation" and "Description" sections contain more details of the instruction's operation.

B.4 Operation

The "Operation" section contains an algorithmic description of the instruction which uses a notation similar to the Algol or Pascal language. The algorithms are composed of the following elements:

1. Comments are enclosed within the symbol pairs "(*" and "*)".
2. Compound statements are enclosed between the keywords of the "if" statement (IF, THEN, ELSE, FI) or of the "do" statement (DO, OD), or of the "while" statement (WHILE, DO, OD).
3. A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to SI's default segment (DS) or overridden segment.
4. Brackets also used for memory operands, where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the contents of the source operand is a segment-relative offset.
5. A ← B; indicates that the value of B is assigned to A.
6. The symbols =, <>, ≥, and ≤ are relational operators used to compare two values, meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize** represents the operand-size attribute of the instruction, which is either 16 or 32 bits. **AddressSize** represents the address-size attribute, which is either 16 or 32 bits. Refer for the explanation of address-size and operand-size attributes to appendix A for general guidelines on how these attributes are determined.

- **SRC** represents the source operand. When there are two operands, SRC is the one on the right.
- **DEST** represents the destination operand. When there are two operands, DEST is the one on the left.
- **LeftSRC**, **RightSRC** distinguishes between two operands when both are source operands.

The following functions are used in the algorithmic descriptions:

- **Truncate to 16 bits(value)** reduces the size of the value to fit in 16 bits by discarding the uppermost bits as needed.
- **Addr(operand)** returns the effective address of the operand (the result of the effective address calculation prior to adding the segment base).
- **ZeroExtend(value)** returns a value zero-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, ZeroExtend of a byte value of -10 converts the byte from F6H to doubleword with hexadecimal value 000000F6H. If the value passed to ZeroExtend and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** returns a value sign-extended to the operand-size attribute of the instruction. For example, if OperandSize = 32, SignExtend of a byte containing the value -10 converts the byte from F6H to a doubleword with hexadecimal value FFFFFFF6H. If the value passed to SignExtend and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **Push(value)** pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. The action of Push is as follows:

```
IF OperandSize = 16
THEN
        SP ← SP - 2;
        SS:[SP] ← value; (* 2 bytes assigned starting at byte address in SP *)
ELSE (* OperandSize = 32 *)
        SP ← SP - 4;
        SS:[SP] ← value; (* 4 bytes assigned starting at byte address in SP *)
FI;
```

- **Pop(value)** removes the value from the top of the stack and returns it. The statement EAX Pop( ); assigns to EAX the 32-bit value that Pop took from the top of the stack. Pop will return either a word or a doubleword depending on the operand-size attribute. The action of Pop is as follows:

```
IF OperandSize = 16
THEN
        ret val ← SS:[SP]; (* 2-byte value *)
        SP ← SP + 2;
ELSE (* OperandSize = 32 *)
        ret val ← SS:[SP]; (* 4-byte value *)
        SP ← SP + 4;
FI;
RETURN(ret val); (*returns a word or doubleword*)
```

- **Bit[BitBase, BitOffset]** returns the address of a bit within a bit string, which is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within

registers and within memory bytes. In memory, the two bytes of a word are stored with the low-order byte at the lower address.

If the base operand is a register, the offset can be in the range 0..31. This offset addresses a bit within the indicated register.

If BitBase is a memory address, BitOffset can range from -2 gigabits to 2 gigabits. The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)), where DIV is signed division with rounding towards negative infinity, and MOD returns a positive number.

B.5 Description

The “Description” section contains further explanation of the instruction’s operation.

B.6 Flags Affected

The “Flags Affected” section lists the flags that are affected by the instruction, as follows:

- If a flag is always cleared or always set by the instruction, the value is given (0 or 1) after the flag name. Arithmetic and logical instructions usually assign values to the status flags in the uniform manner, according to the value of the result. Nonconventional assignments are described in the “Operation” section.
- The values of flags listed as “undefined” may be changed by the instruction in an indeterminate manner.

All flags not listed are unchanged by the instruction.

ADC -- Add with Carry

Opcode	Instruction	Description
14 ib	ADC AL,imm8	Add with CF immediate byte to AL
15 iw	ADC AX,imm16	Add with CF immediate word to AX
15 id	ADC EAX,imm32	Add with CF immediate dword to EAX
80 /2 ib	ADC r/m8,imm8	Add with CF immediate byte to r/m byte
81 /2 iw	ADC r/m16,imm16	Add with CF immediate word to r/m word
81 /2 id	ADC r/m32,imm32	Add with CF immediate dword to r/m dword
83 /2 ib	ADC r/m16,imm8	Add with CF sign-extended immediate byte to r/m word
83 /2 ib	ADC r/m32,imm8	Add with CF sign-extended immediate byte into r/m dword
10 /r	ADC r/m8,r8	Add with CF byte register to r/m byte
11 /r	ADC r/m16,r16	Add with CF word register to r/m word
11 /r	ADC r/m32,r32	Add with CF dword register to r/m dword
12 /r	ADC r8,r/m8	Add with CF r/m byte to byte register
13 /r	ADC r16,r/m16	Add with CF r/m word to word register
13 /r	ADC r32,r/m32	Add with CF r/m dword to dword register

Operation

DEST ⟵ DEST + SRC + CF;

Description

ADC performs an integer addition of the two operands DEST and SRC and the carry flag, CF. The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly. ADC is usually executed as part of a multi-byte or multi-word addition operation. When an immediate byte value is added to a word or doubleword operand, the immediate value is first sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF.

ADD -- Add

Opcode	Instruction	Description
04 ib	ADD AL,imm8	Add immediate byte to AL
05 iw	ADD AX,imm16	Add immediate word to AX
05 id	ADD EAX,imm32	Add immediate dword to EAX
80 /0 ib	ADD r/m8,imm8	Add immediate byte to r/m byte
81 /0 iw	ADD r/m16,imm16	Add immediate word to r/m word
81 /0 id	ADD r/m32,imm32	Add immediate dword to r/m dword
83 /0 ib	ADD r/m16,imm8	Add sign-extended immediate byte to r/m word
83 /0 ib	ADD r/m32,imm8	Add sign-extended immediate byte to r/m dword
00 /r	ADD r/m8,r8	Add byte register to r/m byte
01 /r	ADD r/m16,r16	Add word register to r/m word
01 /r	ADD r/m32,r32	Add dword register to r/m dword
02 /r	ADD r8,r/m8	Add r/m byte to byte register
03 /r	ADD r16,r/m16	Add r/m word to word register
03 /r	ADD r32,r/m32	Add r/m dword to dword register

Operation

DEST ⟵ DEST + SRC;

Description

ADD performs an integer addition of the two operands (DEST and SRC). The result of the addition is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte is added to a word or doubleword operand, the immediate value is sign-extended to the size of the word or doubleword operand.

Flags Affected

OF, SF, ZF, AF, CF, and PF.

AND -- Logical AND

Opcode	Instruction	Description
24 ib	AND AL,imm8	AND immediate byte to AL
25 iw	AND AX,imm16	AND immediate word to AX
25 id	AND EAX,imm32	AND immediate dword to EAX
80 /4 ib	AND r/m8,imm8	AND immediate byte to r/m byte
81 /4 iw	AND r/m16,imm16	AND immediate word to r/m word
81 /4 id	AND r/m32,imm32	AND immediate dword to r/m dword
83 /4 ib	AND r/m16,imm8	AND sign-extended immediate byte with r/m word
83 /4 ib	AND r/m32,imm8	AND sign-extended immediate byte with r/m dword
20 /r	AND r/m8,r8	AND byte register to r/m byte
21 /r	AND r/m16,r16	AND word register to r/m word
21 /r	AND r/m32,r32	AND dword register to r/m dword
22 /r	AND r8,r/m8	AND r/m byte to byte register
23 /r	AND r16,r/m16	AND r/m word to word register
23 /r	AND r32,r/m32	AND r/m dword to dword register

Operation

DEST ⟵ DEST AND SRC;
CF ⟵ 0;
OF ⟵ 0;

Description

Each bit of the result of the AND instruction is a 1 if both corresponding bits of the operands are 1; otherwise, it becomes a 0.

Flags Affected

CF = 0, OF = 0; PF, SF, and ZF.

BT -- Bit Test

Opcode	Instruction	Description
0F A3	BT r/m16,r16	Save bit in carry flag
0F A3	BT r/m32,r32	Save bit in carry flag
0F BA /4 ib	BT r/m16,imm8	Save bit in carry flag
0F BA /4 ib	BT r/m32,imm8	Save bit in carry flag

Operation

CF ← BIT[LeftSRC, RightSRC];

Description

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into the carry flag.

Flags Affected

CF as described above

Notes

The index of the selected bit can be given by the immediate constant in the instruction or by a value in a general register. Only an 8-bit immediate value is used in the instruction. This operand is taken modulo 32, so the range of immediate bit offsets is 0..31. This allows any bit within a register to be selected. For memory bit strings, this immediate field gives only the bit offset within a word or doubleword. Immediate bit offsets larger than 31 are supported by using the immediate bit offset field in combination with the displacement field of the memory operand. The low-order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the high-order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode.

When accessing a bit in memory, the 80386 may access four bytes starting from the memory address given by:

Effective Address + (4 * (BitOffset DIV 32))

for a 32-bit operand size, or two bytes starting from the memory address given by:

Effective Address + (2 * (BitOffset DIV 16))

for a 16-bit operand size. It may do so even when only a single byte needs to be accessed in order to reach the given bit. You must therefore avoid referencing areas of memory close to address space holes. In particular, avoid references to memory-mapped I/O registers. Instead, use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

CALL -- Call Procedure

Opcode	Instruction	Description
E8 cw	CALL rel16	Call near, displacement relative to next instruction
FF /2	CALL r/m16	Call near, register indirect/memory indirect
9A cd	CALL ptr16:16	Call intersegment, to full pointer given
FF /3	CALL m16:16	Call intersegment, address at r/m dword

Operation

```
IF rel16 type of call  THEN (* near relative call *)
        Push(IP);
        EIP ← (EIP + rel16) AND 0000FFFFH;
FI;

IF r/m16 type of call  THEN (* near absolute call *)
        Push(IP);
        EIP ← [r/m16] AND 0000FFFFH;
FI;

IF m16:16 or ptr16:16 type of call  THEN (* far call *)
        Push(CS);
        Push(IP); (* address of next instruction; 16 bits *)
        IF operand type is m16:16  THEN (* indirect far call *)
                CS:IP ← [m16:16];
                EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
        IF operand type is ptr16:16   THEN (* direct far call *)
                CS:IP ← ptr16:16;
                EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
FI;
```

Description

The CALL instruction causes the procedure named in the operand to be executed. The action of the different forms of the instruction are described below.

Near calls are those with destinations of type r/m16 and rel16; changing or saving the segment register value is not necessary. The CALL rel16 form adds a signed offset to the address of the instruction following CALL to determine the destination. The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of EIP are cleared, resulting in an offset whose value does not exceed 16 bits. CALL r/m16 specifies a register or memory location from which the absolute segment offset is fetched. The offset of the instruction following CALL is pushed onto the stack. It will be popped by a near RET instruction within the procedure. The CS register is not changed by this form of CALL.

The far call, CALL ptr16:16, uses a four-byte operand as a long pointer to the procedure called. The CALL m16:16 form fetch the long pointer from the memory location specified (indirection). In Real Address Mode, the long pointer provides 16 bits for the CS register and 16 bits for the EIP register. These forms of the instruction push both CS and IP as a return address.

Flags Affected

No flags are affected.

CLC -- Clear Carry Flag

Opcode	Instruction	Description
F8	CLC	Clear carry flag

Operation

CF ← 0;

Description

CLC sets the carry flag to zero. It does not affect other flags or registers.

Flags Affected

CF = 0

CLI -- Clear Interrupt Flag

Opcode	Instruction	Description
FA	CLI	Clear interrupt flag; interrupts disabled

Operation

IF ⟵ 0;

Description

CLI clears the interrupt flag. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the interrupt flag is set.

Flags Affected

IF = 0

CMC -- Complement Carry Flag

Opcode	Instruction	Description
F5	CMC	Complement carry flag

Operation

CF ← NOT CF;

Description

CMC reverses the setting of the carry flag. No other flags are affected.

Flags Affected

CF as described above

CMP -- Compare Two Operands

Opcode	Instruction	Description
3C ib	CMP AL,imm8	Compare immediate byte to AL
3D iw	CMP AX,imm16	Compare immediate word to AX
3D id	CMP EAX,imm32	Compare immediate dword to EAX
80 /7 ib	CMP r/m8,imm8	Compare immediate byte to r/m byte
81 /7 iw	CMP r/m16,imm16	Compare immediate word to r/m word
81 /7 id	CMP r/m32,imm32	Compare immediate dword to r/m dword
83 /7 ib	CMP r/m16,imm8	Compare sign extended immediate byte to r/m word
83 /7 ib	CMP r/m32,imm8	Compare sign extended immediate byte to r/m dword
38 /r	CMP r/m8,r8	Compare byte register to r/m byte
39 /r	CMP r/m16,r16	Compare word register to r/m word
39 /r	CMP r/m32,r32	Compare dword register to r/m dword
3A /r	CMP r8,r/m8	Compare r/m byte to byte register
3B /r	CMP r16,r/m16	Compare r/m word to word register
3B /r	CMP r32,r/m32	Compare r/m dword to dword register

Operation

LeftSRC - SignExtend(RightSRC); (* CMP does not store a result; its purpose is to set the flags *)

Description

CMP subtracts the second operand from the first but, unlike the SUB instruction, does not store the result; only the flags are changed. CMP is typically used in conjunction with conditional jumps and the SETcc instruction. If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF.

CWD/CDQ -- Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Description
99	CWD	DX:AX ← sign-extend of AX
99	CDQ	EDX:EAX ← sign-extend of EAX

Operation

```
IF OperandSize = 16 (* CWD instruction *)
THEN
        IF AX < 0
        THEN
                DX ← 0FFFFH;
        ELSE
                DX ← 0;
        FI;
ELSE (* OperandSize = 32, CDQ instruction *)
        IF EAX < 0
        THEN
                EDX ← 0FFFFFFFFH;
        ELSE
                EDX ← 0;
        FI;

FI;
```

Description

CWD converts the signed word in AX to a signed doubleword in DX:AX by extending the most significant bit of AX into all the bits of DX. CDQ converts the signed doubleword in EAX to a signed 64-bit integer in the register pair EDX:EAX by extending the most significant bit of EAX (the sign bit) into all the bits of EDX.

Flags Affected

None

DEC -- Decrement by 1

Opcode	Instruction	Description
FE /1	DEC r/m8	Decrement r/m byte by 1
FF /1	DEC r/m16	Decrement r/m word by 1
FF /1	DEC r/m32	Decrement r/m dword by 1
48+rw	DEC r16	Decrement word register by 1
48+rw	DEC r32	Decrement dword register by 1

Operation

DEST ← DEST - 1;

Description

DEC subtracts 1 from the operand. DEC does not change the carry flag. To affect the carry flag, use the SUB instruction with an immediate operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF are modified according to the value of the result.

DIV -- Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV r/m8	Unsigned divide AX by r/m byte (AL=Quo, AH=Rem)
F7 /6	DIV r/m16	Unsigned divide DX:AX by r/m word (AX=Quo, DX=Rem)
F7 /6	DIV r/m32	Unsigned divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem)

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN
        Interrupt 0;
ELSE
        quotient ← temp;
        remainder ← dividend MOD (r/m);
FI;
```

Description

DIV performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX
dword	EDX:EAX	r/m32	EAX	EDX

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Real Address Mode Exceptions

Interrupt 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

IDIV -- Signed Divide

Opcode	Instruction	Description
F6 /7	IDIV r/m8	Signed divide AX by r/m byte (AL=Quo, AH=Rem)
F7 /7	IDIV r/m16	Signed divide DX:AX by r/m word (AX=Quo, DX=Rem)
F7 /7	IDIV r/m32	Signed divide EDX:EAX by r/m dword (EAX=Quo, EDX=Rem)

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN
        Interrupt 0;
ELSE
        quotient ← temp;
        remainder ← dividend MOD (r/m);
FI;
```

Description

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. Only the divisor is given as an explicit r/m operand. The type of the divisor determines which registers to use as follows:

Size	Dividend	Divisor	Quotient	Remainder
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX
dword	EDX:EAX	r/m32	EAX	EDX

If the resulting quotient is too large to fit in the destination, or if the division is 0, an Interrupt 0 is generated. Nonintegral quotients are truncated toward 0. The remainder has the same sign as the dividend and the absolute value of the remainder is always less than the absolute value of the divisor.

Flags Affected

OF, SF, ZF, AR, PF, CF are undefined.

Real Address Mode Exceptions

Interrupt 0 if the quotient is too large to fit in the designated register (AL or AX), or if the divisor is 0; Interrupt 13 if any part of the operand would lie outside of the effective address space from 0 to 0FFFFH

IMUL -- Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL r/m8	AX ← AL * r/m byte
F7 /5	IMUL r/m16	DX:AX ← AX * r/m word
F7 /5	IMUL r/m32	EDX:EAX ← EAX * r/m dword
0F AF /r	IMUL r16,r/m16	word register ← word register * r/m word
0F AF /r	IMUL r32,r/m32	dword register ← dword register * r/m dword
6B /r ib	IMUL r16,r/m16,imm8	word register ← r/m16 * sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	dword register ← r/m32 * sign-extended immediate byte
6B /r ib	IMUL r16,imm8	word register ← word register * sign-extended imm. byte
6B /r ib	IMUL r32,imm8	dword reg. ← dword reg. * sign-extended immediate byte
69 /r iw	IMUL r16,r/m16,imm16	word register ← r/m16 * immediate word
69 /r id	IMUL r32,r/m32,imm32	dword register ← r/m32 * immediate dword
69 /r iw	IMUL r16,imm16	word register ← r/m16 * immediate word
69 /r id	IMUL r32,imm32	dword register ←r/m32 * immediate dword

Operation

result ← multiplicand * multiplier;

Description

IMUL performs signed multiplication. Some forms of the instruction use implicit register operands. The operand combinations for all forms of the instruction are shown in the “Description” column above.

IMUL clears the overflow and carry flags under the following conditions:

Instruction Form	Condition for Clearing CF and OF
r/m8	AL = sign-extend of AL to 16 bits
r/m16	AX = sign-extend of AX to 32 bits
r/m32	EDX:EAX = sign-extend of EAX to 32 bits
r16,r/m16	Result exactly fits within r16
r/32,r/m32	Result exactly fits within r32
r16,r/m16,imm16	Result exactly fits within r16
r32,r/m32,imm32	Result exactly fits within r32

Flags Affected

OF and CF as described above; SF, ZF, AF, and PF are undefined

Notes

When using the accumulator forms (IMUL r/m8, IMUL r/m16, or IMUL r/m32), the result of the multiplication is available even if the overflow flag is set because the result is two times the size of the multiplicand and multiplier. This is large enough to handle any possible result.

IN -- Input from Port

Opcode	Instruction	Description
E4 ib	IN AL,imm8	Input byte from immediate port into AL
E5 ib	IN AX,imm8	Input word from immediate port into AX
E5 ib	IN EAX,imm8	Input dword from immediate port into EAX
EC	IN AL,DX	Input byte from port DX into AL
ED	IN AX,DX	Input word from port DX into AX
ED	IN EAX,DX	Input dword from port DX into EAX

Operation

DEST ← [SRC]; (* Reads from I/O address space *)

Description

IN transfers a data byte or data word from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with DX as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

Flags Affected

None

INC -- Increment by 1

Opcode	Instruction	Description
FE /0	INC r/m8	Increment r/m byte by 1
FF /0	INC r/m16	Increment r/m word by 1
FF /0	INC r/m32	Increment r/m dword by 1
40 + rw	INC r16	Increment word register by 1
40 + rd	INC r32	Increment dword register by 1

Operation

DEST ← DEST + 1;

Description

INC adds 1 to the operand. It does not change the carry flag. To affect the carry flag, use the ADD instruction with a second operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF are modified according to the value of the result.

IRET -- Interrupt Return

Opcode	Instruction	Clocks	Description
CF	IRET	22	Interrupt return (far return and pop flags)

Operation

IP ← Pop();
CS ← Pop();
FLAGS ← Pop();

Description

In Real Address Mode, IRET pops the instruction pointer, CS, and the flags register from the stack and resumes the interrupted routine.

Flags Affected

All; the flags register is popped from stack

Jcc -- Jump if Condition is Met

Opcode	Instruction	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
74 cb	JE rel8	Jump short if equal (ZF=1)
74 cb	JZ rel8	Jump short if 0 (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=OF)
7D cb	JGE rel8	Jump short if greater or equal (SF=OF)
7C cb	JL rel8	Jump short if less (SF<>OF)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 and SF<>OF)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	JO rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
...		
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

NOTES: rel16/32 indicates that these instructions map to two; one with a 16-bit relative displacement, the other with a 32-bit relative displacement, depending on the operand-size attribute of the instruction.

Operation

```
IF condition  THEN
        EIP ← (EIP + SignExtend(rel8/16/32)) AND 0000FFFFH;
FI;
```

Description

Conditional jumps test the flags which have been set by a previous instruction. The conditions for each mnemonic are given in parentheses after each description above. The terms “less” and “greater” are used for comparisons of signed integers; “above” and “below” are used for unsigned integers.

If the given condition is true, a jump is made to the location provided as the operand.

Instruction coding is most efficient when the target for the conditional jump is in the current code segment and within -128 to +127 bytes of the next instruction's first byte. The jump can also target -32768 thru +32767 (segment size attribute 16) or -2^(31) thru +2^(31) -1 (segment size attribute 32) relative to the next instruction's first byte. When the target for the conditional jump is in a different segment, use the opposite case of the jump instruction (i.e., JE and JNE), and then access the target with an unconditional far jump to the other segment. For example, you cannot code:

```
JZ FARLABEL;
```

You must instead code:

```
    JNZ BEYOND;
    JMP FARLABEL;
BEYOND:
```

Because there can be several ways to interpret a particular state of the flags, ASM386 provides more than one mnemonic for most of the conditional jump opcodes. For example, if you compared two characters in AX and want to jump if they are equal, use JE; or, if you ANDed AX with a bit field mask and only want to jump if the result is 0, use JZ, a synonym for JE.

Flags Affected

None

JMP -- Jump

Opcode	Instruction	Clocks	Description
EB cb	JMP rel8	7+m	Jump short
E9 cw	JMP rel16	7+m	Jump near, displacement relative to next instruction
E9 cd	JMP rel32	7+m	Jump near, displacement relative to next instruction
FF /4	JMP r/m16	7+m/10+m	Jump near indirect
EA cd	JMP ptr16:16	12+m	Jump intersegment, 4-byte immediate address
FF /5	JMP m16:16	43+m	Jump r/m16:16 indirect and intersegment

Operation

```
IF instruction = relative JMP (* i.e. operand is rel8, rel16, or rel32 *) THEN
        EIP ← (EIP + rel8/16/32) AND 0000FFFFH;
FI;
IF instruction = near indirect JMP (* i.e. operand is r/m16  *) THEN
        EIP ← [r/m16] AND 0000FFFFH;
FI;

IF instruction = far JMP (* i.e., op. type is m16:16,  ptr16:16  *) THEN
        IF operand type = m16:16 THEN (* indirect *)
                CS:IP ← [m16:16] ;
                EIP ← EIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
        IF operand type = ptr16:16  THEN
                CS:IP ← ptr16:16;
                EIP ←EIP AND 0000FFFFH; (* clear upper 16 bits *)
        FI;
FI;
```

Description

The JMP instruction transfers control to a different point in the instruction stream without recording return information.

The action of the various forms of the instruction are shown below.

Jumps with destinations of type r/m16, rel16, and rel32 are near jumps and do not involve changing the segment register value.

The JMP rel16 and JMP rel32 forms of the instruction add an offset to the address of the instruction following the JMP to determine the destination. The rel16 form is used when the instruction's operand-size attribute is 16 bits (segment size attribute 16 only); rel32 is used when the operand-size attribute is 32 bits (segment size attribute 32 only). The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of EIP are cleared, which results in an offset whose value does not exceed 16 bits.

JMP r/m16 specifies a register or memory location from which the absolute offset from the procedure is fetched. The offset fetched from r/m is 16 bits for an operand-size attribute of 16 bits (r/m16).

The JMP ptr16:16 form of the instruction use a four-byte operand as a pointer to the destination (indirection). In Real Address Mode, the long pointer provides 16 bits for the CS register and 16 bits for the EIP register.

Flags Affected

None

LEA -- Load Effective Address

Opcode	Instruction	Description
8D /r	LEA r16,m	Store effective address for m in register r16
8D /r	LEA r32,m	Store effective address for m in register r32
8D /r	LEA r16,m	Store effective address for m in register r16
8D /r	LEA r32,m	Store effective address for m in register r32

Operation

```
IF OperandSize = 16 AND AddressSize = 16 THEN
        r16 ← Addr(m);
ELSE IF OperandSize = 16 AND AddressSize = 32 THEN
        r16 ← Truncate_to_16bits(Addr(m)); (* 32-bit address *)
ELSE IF OperandSize = 32 AND AddressSize = 16 THEN
        r32 ← Truncate_to_16bits(Addr(m));
ELSE IF OperandSize = 32 AND AddressSize = 32 THEN
        r32 ← Addr(m);
FI; FI; FI; FI;
```

Description

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction (represented by OperandSize in the algorithm under “Operation” above) is determined by the chosen register. The address-size and operand-size attributes affect the action performed by LEA, as follows:

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Flags Affected

None

Real Address Mode Exceptions

Interrupt 6 if the second operand is a register

MOV -- Move Data

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move byte register to r/m byte
89 /r	MOV r/m16,r16	Move word register to r/m word
89 /r	MOV r/m32,r32	Move dword register to r/m dword
8A /r	MOV r8,r/m8	Move r/m byte to byte register
8B /r	MOV r16,r/m16	Move r/m word to word register
8B /r	MOV r32,r/m32	Move r/m dword to dword register
8C /r	MOV r/m16,Sreg	Move segment register to r/m word
8D /r	MOV Sreg,r/m16	Move r/m word to segment register
A0	MOV AL,moffs8	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32	Move dword at (seg:offset) to EAX
A2	MOV moffs8,AL	Move AL to (seg:offset)
A3	MOV moffs16,AX	Move AX to (seg:offset)
A3	MOV moffs32,EAX	Move EAX to (seg:offset)
B0 + rb	MOV reg8,imm8	Move immediate byte to register
B8 + rw	MOV reg16,imm16	Move immediate word to register
B8 + rd	MOV reg32,imm32	Move immediate dword to register
C6	MOV r/m8,imm8	Move immediate byte to r/m byte
C7	MOV r/m16,imm16	Move immediate word to r/m word
C7	MOV r/m32,imm32	Move immediate dword to r/m dword

NOTES: moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.

Operation

DEST ← SRC;

Description

MOV copies the second operand to the first operand.

A MOV into SS inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into SP).

Flags Affected

None

MOVSX -- Move with Sign-Extend

Opcode	Instruction	Description
0F BE /r	MOVSX r16,r/m8	Move byte to word with sign-extend
0F BE /r	MOVSX r32,r/m8	Move byte to dword, sign-extend
0F BF /r	MOVSX r32,r/m16	Move word to dword, sign-extend

Operation

DEST ← SignExtend(SRC);

Description

MOVSX reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

MOVZX -- Move with Zero-Extend

Opcode	Instruction	Description
0F B6 /r	MOVZX r16,r/m8	Move byte to word with zero-extend
0F B6 /r	MOVZX r32,r/m8	Move byte to dword, zero-extend
0F B7 /r	MOVZX r32,r/m16	Move word to dword, zero-extend

Operation

DEST ← ZeroExtend(SRC);

Description

MOVZX reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

MUL -- Unsigned Multiplication of AL or AX

Opcode	Instruction	Description
F6 /4	MUL r/m8	Unsigned multiply (AX ← AL * r/m byte)
F7 /4	MUL r/m16	Unsigned multiply (DX:AX ← AX * r/m word)
F7 /4	MUL r/m32	Unsigned multiply (EDX:EAX ← EAX * r/m dword)

Operation

```
IF byte-size operation
THEN
        AX ← AL * r/m8
ELSE (* word or doubleword operation *)
        IF OperandSize = 16
        THEN
                DX:AX ← AX * r/m16
        ELSE (* OperandSize = 32 *)
                EDX:EAX ← EAX * r/m32
        FI;
FI;
```

Description

MUL performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by AL; the result is left in AX. The carry and overflow flags are set to 0 if AH is 0; otherwise, they are set to 1.
- A word operand is multiplied by AX; the result is left in DX:AX. DX contains the high-order 16 bits of the product. The carry and overflow flags are set to 0 if DX is 0; otherwise, they are set to 1.
- A doubleword operand is multiplied by EAX and the result is left in EDX:EAX. EDX contains the high-order 32 bits of the product. The carry and overflow flags are set to 0 if EDX is 0; otherwise, they are set to 1.

Flags Affected

OF and CF as described above; SF, ZF, AF, PF, and CF are undefined

NEG -- Two’s Complement Negation

Opcode	Instrction	Description
F6 /3	NEG r/m8	Two’s complement negate r/m byte
F7 /3	NEG r/m16	Two’s complement negate r/m word
F7 /3	NEG r/m32	Two’s complement negate r/m dword

Operation

```
IF r/m = 0
THEN
        CF ← 0
ELSE
        CF ← 1;
FI;
r/m ← -r/m;
```

Description

NEG replaces the value of a register or memory operand with its two’s complement. The operand is subtracted from zero, and the result is placed in the operand.

The carry flag is set to 1, unless the operand is zero, in which case the carry flag is cleared to 0.

Flags Affected

CF as described above; OF, SF, ZF, and PF are modified according to the value of the result.

NOP -- No Operation

Opcode	Instruction	Description
90	NOP	No operation

Description

NOP performs no operation. NOP is a one-byte instruction that takes up space but affects none of the machine context except (E)IP.

NOP is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

Flags Affected

None

NOT -- One’s Complement Negation

Opcode	Instruction	Description
F6 /2	NOT r/m8	Reverse each bit of r/m byte
F7 /2	NOT r/m16	Reverse each bit of r/m word
F7 /2	NOT r/m32	Reverse each bit of r/m dword

Operation

r/m ← NOT r/m;

Description

NOT inverts the operand; every 1 becomes a 0, and vice versa.

Flags Affected

None

OR -- Logical Inclusive OR

Opcode	Instruction	Description
0C ib	OR AL,imm8	OR immediate byte to AL
0D iw	OR AX,imm16	OR immediate word to AX
0D id	OR EAX,imm32	OR immediate dword to EAX
80 /1 ib	OR r/m8,imm8	OR immediate byte to r/m byte
81 /1 iw	OR r/m16,imm16	OR immediate word to r/m word
81 /1 id	OR r/m32,imm32	OR immediate dword to r/m dword
83 /1 ib	OR r/m16,imm8	OR sign-extended immediate byte with r/m word
83 /1 ib	OR r/m32,imm8	OR sign-extended immediate byte with r/m dword
08 /r	OR r/m8,r8	OR byte register to r/m byte
09 /r	OR r/m16,r16	OR word register to r/m word
09 /r	OR r/m32,r32	OR dword register to r/m dword
0A /r	OR r8,r/m8	OR byte register to r/m byte
0B /r	OR r16,r/m16	OR word register to r/m word
0B /r	OR r32,r/m32	OR dword register to r/m dword

Operation

DEST ← DEST OR SRC;
CF ← 0;
OF ← 0

Description

OR computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

Flags Affected

OF = 0, CF = 0; SF, ZF, and PF are modified according to the value of the result. AF is undefined.

OUT -- Output to Port

Opcode	Instruction	Description
E6 ib	OUT imm8,AL	Output byte AL to immediate port number
E7 ib	OUT imm8,AX	Output word AL to immediate port number
E7 ib	OUT imm8,EAX	Output dword AL to immediate port number
EE	OUT DX,AL	Output byte AL to port number in DX
EF	OUT DX,AX	Output word AL to port number in DX
EF	OUT DX,EAX	Output dword AL to port number in DX

Operation

[DEST] ← SRC; (* I/O address space used *)

Description

OUT transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with DX as the first operand. If the instruction contains an eight-bit port ID, that value is zero-extended to 16 bits.

Flags Affected

None

POP -- Pop a Word from the Stack

Opcode	Instruction	Description
8F /0	POP m16	Pop top of stack into memory word
8F /0	POP m32	Pop top of stack into memory dword
58 + rw	POP r16	Pop top of stack into word register
58 + rd	POP r32	Pop top of stack into dword register
1F	POP DS	Pop top of stack into DS
07	POP ES	Pop top of stack into ES
17	POP SS	Pop top of stack into SS
0F A1	POP FS	Pop top of stack into FS
0F A9	POP GS	Pop top of stack into GS

Operation

```
IF OperandSize = 16
THEN
        DEST ← (SS:SP); (* copy a word *)
        SP ← SP + 2;
ELSE (* OperandSize = 32 *)
        DEST ← (SS:SP); (* copy a dword *)
        SP ← SP + 4;
FI;
```

Description

POP replaces the previous contents of the memory, the register, or the segment register operand with the word on the top of the 80386 stack, addressed by SS:SP (address-size attribute of 16 bits). The stack pointer SP is incremented by 2 for an operand-size of 16 bits or by 4 for an operand-size of 32 bits. It then points to the new top of stack.

POP CS is not an 80386 instruction. Popping from the stack into the CS register is accomplished with a RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector.

A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP SP instructions without danger of having an invalid stack during an interrupt.

Flags Affected

None

PUSH -- Push Operand onto the Stack

Opcode	Instruction	Description
FF /6	PUSH m16	Push memory word
FF /6	PUSH m32	Push memory dword
50 + /r	PUSH r16	Push register word
50 + /r	PUSH r32	Push register dword
68	PUSH imm16	Push immediate word
68	PUSH imm32	Push immediate dword
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
OF A8	PUSH GS	Push GS

Operation

```
IF OperandSize = 16 THEN
        SP ← SP - 2;
        (SS:SP) ← (SOURCE); (* word assignment *)
ELSE
        SP ← SP - 4;
        (SS:SP) ← (SOURCE); (* dword assignment *)
FI;
```

Description

PUSH decrements the stack pointer by 2 if the operand-size attribute of the instruction is 16 bits; otherwise, it decrements the stack pointer by 4. PUSH then places the operand on the new top of stack, which is pointed to by the stack pointer.

The 80386 PUSH SP instruction pushes the value of SP as it existed before the instruction. This differs from the 8086, where PUSH SP pushes the new value (decremented by 2).

Flags Affected

None

Real Address Mode Exceptions

None; if SP or ESP is 1, the 80386 shuts down due to a lack of stack space

RCL/RCR/ROL/ROR -- Rotate

Opcode	Instruction	Description
D0 /2	RCL r/m8,1	Rotate 9 bits (CF,r/m byte) left once
D2 /2	RCL r/m8,CL	Rotate 9 bits (CF,r/m byte) left CL times
C0 /2 ib	RCL r/m8,imm8	Rotate 9 bits (CF,r/m byte) left imm8 times
D1 /2	RCL r/m16,1	Rotate 17 bits (CF,r/m word) left once
D3 /2	RCL r/m16,CL	Rotate 17 bits (CF,r/m word) left CL times
C1 /2 ib	RCL r/m16,imm8	Rotate 17 bits (CF,r/m word) left imm8 times
D1 /2	RCL r/m32,1	Rotate 33 bits (CF,r/m dword) left once
D3 /2	RCL r/m32,CL	Rotate 33 bits (CF,r/m dword) left CL times
C1 /2 ib	RCL r/m32,imm8	Rotate 33 bits (CF,r/m dword) left imm8 times
D0 /3	RCR r/m8,1	Rotate 9 bits (CF,r/m byte) right once
D2 /3	RCR r/m8,CL	Rotate 9 bits (CF,r/m byte) right CL times
C0 /3 ib	RCR r/m8,imm8	Rotate 9 bits (CF,r/m byte) right imm8 times
D1 /3	RCR r/m16,1	Rotate 17 bits (CF,r/m word) right once
D3 /3	RCR r/m16,CL	Rotate 17 bits (CF,r/m word) right CL times
C1 /3 ib	RCR r/m16,imm8	Rotate 17 bits (CF,r/m word) right imm8 times
D1 /3	RCR r/m32,1	Rotate 33 bits (CF,r/m dword) right once
D3 /3	RCR r/m32,CL	Rotate 33 bits (CF,r/m dword) right CL times
C1 /3 ib	RCR r/m32,imm8	Rotate 33 bits (CF,r/m dword) right imm8 times
D0 /0	ROL r/m8,1	Rotate 8 bits r/m byte left once
D2 /0	ROL r/m8,CL	Rotate 8 bits r/m byte left CL times
C0 /0 ib	ROL r/m8,imm8	Rotate 8 bits r/m byte left imm8 times
D1 /0	ROL r/m16,1	Rotate 16 bits r/m word left once
D3 /0	ROL r/m16,CL	Rotate 16 bits r/m word left CL times
C1 /0 ib	ROL r/m16,imm8	Rotate 16 bits r/m word left imm8 times
D1 /0	ROL r/m32,1	Rotate 32 bits r/m dword left once
D3 /0	ROL r/m32,CL	Rotate 32 bits r/m dword left CL times
C1 /0 ib	ROL r/m32,imm8	Rotate 32 bits r/m dword left imm8 times
D0 /1	ROR r/m8,1	Rotate 8 bits r/m byte right once
D2 /1	ROR r/m8,CL	Rotate 8 bits r/m byte right CL times
C0 /1 ib	ROR r/m8,imm8	Rotate 8 bits r/m word right imm8 times
D1 /1	ROR r/m16,1	Rotate 16 bits r/m word right once
D3 /1	ROR r/m16,CL	Rotate 16 bits r/m word right CL times
C1 /1 ib	ROR r/m16,imm8	Rotate 16 bits r/m word right imm8 times
D1 /1	ROR r/m32,1	Rotate 32 bits r/m dword right once
D3 /1	ROR r/m32,CL	Rotate 32 bits r/m dword right CL times
C1 /1 ib	ROR r/m32,imm8	Rotate 32 bits r/m dword right imm8 times

Operation

```
(* ROL - Rotate Left *)
temp ← COUNT;
WHILE (temp <> 0) DO
        tmpcf ← high-order bit of (r/m);
        r/m ← r/m * 2 + (tmpcf);
        temp ← temp - 1;
OD;
IF COUNT = 1 THEN
        IF high-order bit of r/m <> CF THEN
                OF ← 1;
        ELSE
                OF ← 0;
        FI;
```

```
ELSE
        OF ← undefined;
FI;
(* ROR - Rotate Right *)
temp ← COUNT;
WHILE (temp <> 0 ) DO
        tmpcf ← low-order bit of (r/m);
        r/m ← r/m / 2 + (tmpcf * 2^(width(r/m)));
        temp ← temp - 1;
DO;
IF COUNT = 1 THEN
        IF (high-order bit of r/m) <> (bit next to high-order bit of r/m) THEN
                OF ← 1;
        ELSE
                OF ← 0;
        FI;
ELSE
        OF ← undefined;
FI;
```

Description

Each rotate instruction shifts the bits of the register or memory operand given. The left rotate instructions shift all the bits upward, except for the top bit, which is returned to the bottom. The right rotate instructions do the reverse: the bits shift downward until the bottom bit arrives at the top.

For the RCL and RCR instructions, the carry flag is part of the rotated quantity. RCL shifts the carry flag into the bottom bit and shifts the top bit into the carry flag; RCR shifts the carry flag into the top bit and shifts the bottom bit into the carry flag. For the ROL and ROR instructions, the original value of the carry flag is not a part of the result, but the carry flag receives a copy of the bit that was shifted from one end to the other.

The rotate is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum instruction execution time, the 80386 does not allow rotation counts greater than 31. If a rotation count greater than 31 is attempted, only the bottom five bits of the rotation are used. The 8086 does not mask rotation counts. The 80386 does mask rotation counts.

The overflow flag is defined only for the single-rotate forms of the instructions (second operand = 1). It is undefined in all other cases. For left shifts/rotates, the CF bit after the shift is XORed with the high-order result bit. For right shifts/rotates, the high-order two bits of the result are XORed to get OF.

Flags Affected

OF only for single rotates; OF is undefined for multi-bit rotates; CF as described above

RET -- Return from Procedure

Opcode	Instruction	Description
CB	RET	Return (far) to caller
CA iw	RET imm16	Return (far), pop imm16 bytes

Operation

```
IP ← Pop(); (* 16-bit pop *)
EIP ← EIP AND 0000FFFFH;
CS ← Pop(); (* 16-bit pop *)
IF instruction has immediate operand THEN SP ← SP + imm16; FI;
```

Description

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL.

The optional numeric parameter to RET gives the number of stack bytes to be released after the return address is popped. These items are typically used as input parameters to the procedure called.

The address on the stack is a pointer. The offset is popped first, followed by the selector.

Flags Affected

None

SAL/SAR/SHL/SHR -- Shift Instructions

Opcode	Instruction	Description
D0 /4	SAL r/m8,1	Multiply r/m byte by 2, once
D2 /4	SAL r/m8,CL	Multiply r/m byte by 2, CL times
C0 /4 ib	SAL r/m8,imm8	Multiply r/m byte by 2, imm8 times
D1 /4	SAL r/m16,1	Multiply r/m word by 2, once
D3 /4	SAL r/m16,CL	Multiply r/m word by 2, CL times
C1 /4 ib	SAL r/m16,imm8	Multiply r/m word by 2, imm8 times
D1 /4	SAL r/m32,1	Multiply r/m dword by 2, once
D3 /4	SAL r/m32,CL	Multiply r/m dword by 2, CL times
C1 /4 ib	SAL r/m32,imm8	Multiply r/m dword by 2, imm8 times
D0 /7	SAR r/m8,1	Signed divide (1) r/m byte by 2, once
D2 /7	SAR r/m8,CL	Signed divide (1) r/m byte by 2, CL times
C0 /7 ib	SAR r/m8,imm8	Signed divide (1) r/m byte by 2, imm8 times
D1 /7	SAR r/m16,1	Signed divide (1) r/m word by 2, once
D3 /7	SAR r/m16,CL	Signed divide (1) r/m word by 2, CL times
C1 /7 ib	SAR r/m16,imm8	Signed divide (1) r/m word by 2, imm8 times
D1 /7	SAR r/m32,1	Signed divide (1) r/m dword by 2, once
D3 /7	SAR r/m32,CL	Signed divide (1) r/m dword by 2, CL times
C1 /7 ib	SAR r/m32,imm8	Signed divide (1) r/m dword by 2, imm8 times
D0 /4	SHL r/m8,1	Multiply r/m byte by 2, once
D2 /4	SHL r/m8,CL	Multiply r/m byte by 2, CL times
C0 /4 ib	SHL r/m8,imm8	Multiply r/m byte by 2, imm8 times
D1 /4	SHL r/m16,1	Multiply r/m word by 2, once
D3 /4	SHL r/m16,CL	Multiply r/m word by 2, CL times
C1 /4 ib	SHL r/m16,imm8	Multiply r/m word by 2, imm8 times
D1 /4	SHL r/m32,1	Multiply r/m dword by 2, once
D3 /4	SHL r/m32,CL	Multiply r/m dword by 2, CL times
C1 /4 ib	SHL r/m32,imm8	Multiply r/m dword by 2, imm8 times
D0 /5	SHR r/m8,1	Unsigned divide r/m byte by 2, once
D2 /5	SHR r/m8,CL	Unsigned divide r/m byte by 2, CL times
C0 /5 ib	SHR r/m8,imm8	Unsigned divide r/m byte by 2, imm8 times
D1 /5	SHR r/m16,1	Unsigned divide r/m word by 2, once
D3 /5	SHR r/m16,CL	Unsigned divide r/m word by 2, CL times
C1 /5 ib	SHR r/m16,imm8	Unsigned divide r/m word by 2, imm8 times
D1 /5	SHR r/m32,1	Unsigned divide r/m dword by 2, once
D3 /5	SHR r/m32,CL	Unsigned divide r/m dword by 2, CL times
C1 /5 ib	SHR r/m32,imm8	Unsigned divide r/m dword by 2, imm8 times

Note: (1) Not the same division as IDIV; rounding is toward negative infinity.

Operation

```
(* COUNT is the second parameter *)
(temp) ← COUNT;
WHILE (temp <> 0) DO
        IF instruction is SAL or SHL THEN
                CF ← high-order bit of r/m;
        FI;
        IF instruction is SAR or SHR THEN
                CF ← low-order bit of r/m;
        FI;
        IF instruction = SAL or SHL THEN
                r/m ← r/m * 2;
        FI;
```

```
        IF instruction = SAR THEN
                r/m ← r/m /2 (*Signed divide, rounding toward negative infinity*);
        FI;
        IF instruction = SHR THEN
                r/m ← r/m / 2; (* Unsigned divide *);
        FI;
        temp ← temp - 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT = 1 THEN
        IF instruction is SAL or SHL THEN
                OF ← high-order bit of r/m <> (CF);
        FI;
        IF instruction is SAR THEN
                OF ← 0;
        FI;
        IF instruction is SHR THEN
                OF ← high-order bit of operand;
        FI;
ELSE
        OF ← undefined;
FI;
```

Description

SAL (or its synonym, SHL) shifts the bits of the operand upward. The high-order bit is shifted into the carry flag, and the low-order bit is set to 0.

SAR and SHR shift the bits of the operand downward. The low-order bit is shifted into the carry flag. The effect is to divide the operand by 2. SAR performs a signed divide with rounding toward negative infinity (not the same as IDIV); the high-order bit remains the same. SHR performs an unsigned divide; the high-order bit is set to 0.

The shift is repeated the number of times indicated by the second operand, which is either an immediate number or the contents of the CL register. To reduce the maximum execution time, the 80386 does not allow shift counts greater than 31. If a shift count greater than 31 is attempted, only the bottom five bits of the shift count are used. (The 8086 uses all eight bits of the shift count.)

The overflow flag is set only if the single-shift forms of the instructions are used. For left shifts, OF is set to 0 if the high bit of the answer is the same as the result of the carry flag (i.e., the top two bits of the original operand were the same); OF is set to 1 if they are different. For SAR, OF is set to 0 for all single shifts. For SHR, OF is set to the high-order bit of the original operand.

Flags Affected

OF for single shifts; OF is undefined for multiple shifts; CF, ZF, PF, and SF are modified according to the value of the result.

SBB -- Integer Subtraction with Borrow

Opcode	Instruction	Description
1C ib	SBB AL,imm8	Subtract with borrow immediate byte from AL
1D iw	SBB AX,imm16	Subtract with borrow immediate word from AX
1D id	SBB EAX,imm32	Subtract with borrow immediate dword from EAX
80 /3 ib	SBB r/m8,imm8	Subtract with borrow immediate byte from r/m byte
81 /3 iw	SBB r/m16,imm16	Subtract with borrow immediate word from r/m word
81 /3 id	SBB r/m32,imm32	Subtract with borrow immediate dword from r/m dword
83 /3 ib	SBB r/m16,imm8	Subtract with borrow sign-ext. immed. byte from r/m word
83 /3 ib	SBB r/m32,imm8	Subtract with borrow sign-ext. immed. byte from r/m dword
18 /r	SBB r/m8,r8	Subtract with borrow byte register from r/m byte
19 /r	SBB r/m16,r16	Subtract with borrow word register from r/m word
19 /r	SBB r/m32,r32	Subtract with borrow dword register from r/m dword
1A /r	SBB r8,r/m8	Subtract with borrow byte register from r/m byte
1B /r	SBB r16,r/m16	Subtract with borrow word register from r/m word
1B /r	SBB r32,r/m32	Subtract with borrow dword register from r/m dword

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN
        DEST ← DEST - (SignExtend(SRC) + CF)
ELSE
        DEST ← DEST - (SRC + CF);
FI;
```

Description

SBB adds the second operand (DEST) to the carry flag (CF) and subtracts the result from the first operand (SRC). The result of the subtraction is assigned to the first operand (DEST), and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.

Flags Affected

OF, SF, ZF, AF, PF, and CF are modified according to the value of the result.

STC -- Set Carry Flag

Opcode	Instruction	Description
F9	STC	Set carry flag

Operation

CF ← 1;

Description

STC sets the carry flag to 1.

Flags Affected

CF = 1

STI -- Set Interrupt Flag

Opcode	Instruction	Description
F13	STI	Set interrupt flag; interrupts enabled at the end of the next instruction

Operation

IF ← 1

Description

STI sets the interrupt flag to 1. The 80386 then responds to external interrupts after executing the next instruction if the next instruction allows the interrupt flag to remain enabled. If external interrupts are disabled and you code STI, RET (such as at the end of a subroutine), the RET is allowed to execute before external interrupts are recognized. Also, if external interrupts are disabled and you code STI, CLI, then external interrupts are not recognized because the CLI instruction clears the interrupt flag during its execution.

Flags Affected

IF = 1

SUB -- Integer Subtraction

Opcode	Instruction	Description
2C ib	SUB AL,imm8	Subtract immediate byte from AL
2D iw	SUB AX,imm16	Subtract immediate word from AX
2D id	SUB EAX,imm32	Subtract immediate dword from EAX
80 /5 ib	SUB r/m8,imm8	Subtract immediate byte from r/m byte
81 /5 iw	SUB r/m16,imm16	Subtract immediate word from r/m word
81 /5 id	SUB r/m32,imm32	Subtract immediate dword from r/m dword
83 /5 ib	SUB r/m16,imm8	Subtract sign-extended immediate byte from r/m word
83 /5 ib	SUB r/m32,imm8	Subtract sign-extended immediate byte from r/m dword
28 /r	SUB r/m8,r8	Subtract byte register from r/m byte
29 /r	SUB r/m16,r16	Subtract word register from r/m word
29 /r	SUB r/m32,r32	Subtract dword register from r/m dword
2A /r	SUB r8,r/m8	Subtract byte register from r/m byte
2B /r	SUB r16,r/m16	Subtract word register from r/m word
2B /r	SUB r32,r/m32	Subtract dword register from r/m dword

Operation

```
IF SRC is a byte and DEST is a word or dword
THEN
        DEST ← DEST - SignExtend(SRC);
ELSE
        DEST ← DEST - SRC;
FI;
```

Description

SUB subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction, and the flags are set accordingly.

When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

Flags Affected

OF, SF, ZF, AF, PF, and CF as modified according to the value of the result.

TEST -- Logical Compare

Opcode	Instruction	Description
A8 ib	TEST AL,imm8	AND immediate byte with AL
A9 iw	TEST AX,imm16	AND immediate word with AX
A9 id	TEST EAX,imm32	AND immediate dword with EAX
F6 /0 ib	TEST r/m8,imm8	AND immediate byte with r/m byte
F7 /0 iw	TEST r/m16,imm16	AND immediate word with r/m word
F7 /0 id	TEST r/m32,imm32	AND immediate dword with r/m dword
84 /r	TEST r/m8,r8	AND byte register with r/m byte
85 /r	TEST r/m16,r16	AND word register with r/m word
85 /r	TEST r/m32,r32	AND dword register with r/m dword

Operation

DEST ← LeftSRC AND RightSRC;
CF ← 0;
OF ← 0;

Description

TEST computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

Flags Affected

OF = 0, CF = 0; SF, ZF, and PF are modified according to the value of the result.

XCHG -- Exchange Register/Memory with Register

Opcode	Instruction	Description
90 + r	XCHG AX,r16	Exchange word register with AX
90 + r	XCHG r16,AX	Exchange word register with AX
90 + r	XCHG EAX,r32	Exchange dword register with EAX
90 + r	XCHG r32,EAX	Exchange dword register with EAX
86 /r	XCHG r/m8,r8	Exchange byte register with r/m byte
86 /r	XCHG r8,r/m8	Exchange byte register with r/m byte
87 /r	XCHG r/m16,r16	Exchange word register with r/m word
87 /r	XCHG r16,r/m16	Exchange word register with r/m word
87 /r	XCHG r/m32,r32	Exchange dword register with r/m dword
87 /r	XCHG r32,r/m32	Exchange dword register with r/m dword

Operation

```
temp ← DEST;
DEST ← SRC;
SRC ← temp;
```

Description

XCHG exchanges two operands. The operands can be in either order.

Flags Affected

None

XOR -- Logical Exclusive OR

Opcode	Instruction	Description
34 ib	XOR AL,imm8	Exclusive-OR immediate byte to AL
35 iw	XOR AX,imm16	Exclusive-OR immediate word to AX
35 id	XOR EAX,imm32	Exclusive-OR immediate dword to EAX
80 /6 ib	XOR r/m8,imm8	Exclusive-OR immediate byte to r/m byte
81 /6 iw	XOR r/m16,imm16	Exclusive-OR immediate word to r/m word
81 /6 id	XOR r/m32,imm32	Exclusive-OR immediate dword to r/m dword
83 /6 ib	XOR r/m16,imm8	XOR sign-extended immediate byte with r/m word
83 /6 ib	XOR r/m32,imm8	XOR sign-extended immediate byte with r/m dword
30 /r	XOR r/m8,r8	Exclusive-OR byte register to r/m byte
31 /r	XOR r/m16,r16	Exclusive-OR word register to r/m word
31 /r	XOR r/m32,r32	Exclusive-OR dword register to r/m dword
32 /r	XOR r8,r/m8	Exclusive-OR byte register to r/m byte
33 /r	XOR r16,r/m16	Exclusive-OR word register to r/m word
33 /r	XOR r32,r/m32	Exclusive-OR dword register to r/m dword

Operation

```
DEST ← LeftSRC XOR RightSRC;
CF ← 0;
OF ← 0;
```

Description

XOR computes the exclusive OR of the two operands. Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same. The answer replaces the first operand.

Flags Affected

CF = 0, OF = 0. SF, ZF, and PF are modified according to the value of the result. AF is undefined.

Apèndix C. Aspectes bàsics del LA TASM

En aquest apèndix es mostra l'estructura bàsica d'un programa escrit en el llenguatge assemblador anomenat TurboAssembler (TASM) i les principals característiques sintàctiques que permeten fer la declaració de constants, variables i especificar els modes d'adreçament dels operands de les instruccions.

C.1 Estructura bàsica

L'estructura bàsica d'un programa en assemblador és la següent.

```
        .model  large
        .386

; Definicio de constants
        ... constants del programa ...

; Definicio de variables
        .data                       ; Directiva que comença el segment de dades
        ... variables del programa ...

; Definicio de la pila
        .stack  100h                ; Directiva que comença el segment de pila
                                    ; en la que s'hi indica la mida en bytes.

; Definicio de codi del programa
        .code                       ; Directiva que comença el segment de codi
        .startup                    ; Codi per iniciar el programa

        ... instruccions del programa ...

        .exit                       ; Codi per retornar al SO

        end
```

C.2 Declaració de constants

A continuació es mostren varis exemples de declaració de constants.

```
; ... constants del programa ...

        D       EQU 10              ; D es una constant amb valor 10.
                                    ; Per defecte, qualsevol valor numeric
                                    ; es expressat en decimal.

        H1      EQU 10h             ; H1 es una constant amb valor 16 (10h).
```

```
        H2      EQU 0ah             ; H2 es una constant amb valor 10 (0ah)
                                    ; Per expressar un numero en hexadecimal,
                                    ; afegir h. A mes, el primer caràcter
                                    ; ha de ser un digit 0..9
        B       EQU 10b             ; B es una constant amb valor 2 (10b).
                                    ; Per expressar un número en booleà,afegir b.
```

C.2 Declaració de variables

A continuació es mostren diversos exemples de declaració de variables.

```
; ... variables del programa ...

dada1   db      12h                 ; Es defineix la variable dada1 de mida byte
                                    ; i s'inicialitza al valor 12h.
dada2   dw      ?                   ; Es defineix la variable dada2 de mida word
                                    ; i no s'inicialitza.
dada3   dd      5 dup (0)           ; Es reserva espai per a 5 elements de mida dword
                                    ; i s'inicialitzen tots ells a 0.
                                    ; L'etiqueta dada3 apunta a l'adreça del
                                    ; primer element.
dada4   db      1,2,3,4,5           ; Es reserva espai per a 5 elements de mida byte
                                    ; i s'inicialitza el seu contingut.
                                    ; Dada4 apunta al primer element.
car     db      'c'                 ; Es defineix la variable car, i
                                    ; s'inicialitza amb el codi ASCII del caràcter 'c'.
string  db      'Hola, món.'        ; Es defineix la variable string, i
                                    ; s'inicialitza amb els codis ASCII de la cadena.
punter  dd      dada1               ; La variable punter conté un punter cap a la
                                    ; variable dada1.
                                    ; Un punter correspon al selector de segment
                                    ; (16 bits de més pes) i l'adreça efectiva (16 bits
                                    ; de menys pes).
```

C.3 Sintaxi dels modes d'adreçament

A continuació es mostren algunes instruccions on s'utilitzen diferents modes d'adreçament per a especificar l'operand font.

```
        mov     eax,ebx         ; Mode registre
        mov     ax,1            ; Mode immediat
        mov     al,D            ; Mode immediat: D es una constant
        mov     ax,dada2        ; Mode memoria (despl): dada2 es una etiqueta
        mov     eax,[ebx]       ; Mode memoria (base)
        mov     eax,[ebx+esi]   ; Mode memoria (base + index)
        mov     eax,4[ebx]      ; Mode memoria (base + despl)
        mov     eax,dada3[esi*4] ; Mode memoria (index*n + despl)
        mov     ax,4[ebx+esi*2] ; Mode memoria (cas general)
```

L'assemblador determina la mida dels operands a partir del nom del registre (quan s'utilitza mode registre) o pel tipus amb el qual s'ha definit una etiqueta (en mode memòria). Quan no es

fa servir cap de les dues possibilitats anteriors, l'assemblador no pot determinar quina és la mida de l'operand. En aquests casos, per a especificar-la s'ha d'utilitzar la directiva ptr.

```
mov     byte ptr [eax], 0   ; Mou el byte 0 a la posició de memòria
                            ; apuntada pel registre eax.
mov     word ptr [eax], 0   ; Mou el word 0.
mov     dword ptr [eax], 0  ; Mou el dword 0.
```

També es pot fer servir aquesta directiva per modificar el tipus de símbols que ja estan definits

```
mov     ah,byte ptr dada2   ; Mou el byte que hi ha a l'@ dada2 (encara
                            ; que dada2 s'ha definit com una @ a word).
```

C.4 Declaració d'etiquetes del codi i de subrutines

Exemple de creació d'etiquetes de codi i de la seva utilització en instruccions de salt:

```
bucle:                      ; Bucle es una etiqueta que podrà ser
                            ; utilitzada en instruccions de salt.
                            ;
                            ; Instruccions dins el cos d'un bucle
                            ;
        jne    bucle        ; La instrucció jne utilitza una etiqueta
                            ; definida en el fitxer.
```

Estructura d'una subrutina:

```
subr1   proc
                           ; Instruccions dins la subrutina
subr1   endp
```

Exemple d'instrucció de crida a subrutina:

```
call    subr1              ; instrucció que crida una subrutina
```

C.5 Sintaxi de les expressions

Es poden fer servir expressions per referenciar operands en mode immediat o per especificar la part anomenada desplaçament en el mode memòria. En aquestes dues situacions es pot avaluar la constant respectiva mitjançant el càlcul d'una expressió. Evidentment, els operands sempre han de ser constants en temps d'assemblatge.

Els operadors permesos són els següents:

op1 OR op2	operació O lògica
op1 XOR op2	operació O exclusiva
op1 AND op2	operació I lògica
NOT op	complementació
op1 EQ op2	retorna 1 si els operands són iguals (=)
op1 NE op2	id. (<>)
op1 LT op2	id. (<)
op1 LE op2	id. (<=)

op1 GT op2	id. (>)
op1 GE op2	id. (>=)
op1 + op2	suma
op1 - op2	resta
op1 * op2	producte
op1 / op2	quocient de la divisió
op1 MOD op2	reste de la divisió
op1 SHR op2	desplaça op1 a la dreta op2 bits
op1 SHL op2	desplaça op1 a l'esquerra op2 bits
+ op	retorna op
- op	nega op

Exemples d'ús d'expressions:

```
; en la definició d'una constant
        M       EQU 4
        N       EQU 10*M


; en la definició de variables
        V       DD N+1 DUP (?)


; en les instruccions del programa
        MOV     EAX, N MOD 4    ; ... utilitzada en mode immediat.
        MOV     EAX, V-4[EAX*4] ; ... utilitzada en la part del desplaçament del
                                ;     mode memòria.
```

Apèndix D. Temps d'execució de les instruccions

D.1 Introducció

En aquest apèndix s'indica el temps d'execució del subconjunt d'instruccions del processador 386 que considerem al llarg del curs. Utilitzarem els cicles de rellotge del processador per mesurar el temps. Cal mencionar que aquest temps és una aproximació del temps exacte que trigarien en executar-se en un sistema real.

El temps total d'execució d'una instrucció es desglosa en tres parts: (i) el temps degut al fetch, la descodificació i l'execució de la instrucció; (ii) el temps degut al mode d'adreçament utilitzat per referenciar els operands explícits, i (iii) el temps utilitzat per accedir els operands que són a memòria. Les principals característiques que es contemplen en aquestes dades són les següents:

- Respecte al primer punt, la taula que hi ha en aquest apèndix mostra el temps degut al fetch, la descodificació i l'execució de cadascuna de les instruccions. Aquest temps correspon a 8 cicles en la majoria de les instruccions. Hi ha unes quantes instruccions més complexes que triguen més cicles.
- Respecte al segon punt, cal distingir el temps requerit per cadascun dels modes d'adreçament que es poden utilitzar per referenciar els operands explícits de la instrucció. Si s'utilitza el mode registre en un operand, no s'incrementa el temps d'execució de les instruccions. Si s'utilitza el mode immediat, s'incrementa el temps d'execució en 2 cicles per realitzar el fetch de l'operand. Si s'utilitza el mode memòria, només s'haurà d'incrementar el temps en 2 cicles quan es faci servir la quantitat que anomenen desplaçament en la sintaxi d'aquest mode. Això es degut al temps per fer el fetch d'aquesta part de la instrucció.
- Respecte al tercer punt, s'haurà de tenir en compte el temps per llegir o escriure aquells operands que estiguin emmagatzemats a memòria. En concret, s'ha d'incrementar el temps en 5 cicles quan a la instrucció es llegeix o s'escriu un operand a memòria. Quan l'operand es llegeix i s'escriu en la mateixa instrucció, el temps que s'haurà d'incrementar correspon a 7 cicles.

En les instruccions de control Jcc, JMP i CALL, el nombre de cicles que apareix en la taula ja correspon al temps total d'execució. A la instrucció Jcc, el temps es diferencia segons si la instrucció salta o no salta.

En les instruccions de multiplicació i divisió, el temps que apareix en la taula té en compte que els operands siguin de mida byte, word o dword. En el temps que correspongui s'hi haurà de sumar el degut als modes d'adreçament i als possibles accessos a memòria.

D.2 Taula per calcular el temps d'execució de programes

Temps de fetch, decodificació i execució

ADC	8		MUL	19/27/43	(B/W/D)
ADD	8		NEG	8	
AND	8		NOP	8	
BT	8		NOT	8	
CALL	17 *		OR	8	
CDQ	8		OUT	13	
CLC	8		POP	10	
CLI	8		PUSH	10	
CMC	8		RCL	8	
CMP	8		RCR	8	
CWD	8		RET	17*	
DEC	8		ROL	8	
DIV	19/27/43	(B/W/D)	ROR	8	
IDIV	19/27/43	(B/W/D)	SAL	8	
IMUL	19/27/43	(B/W/D)	SAR	8	
IN	13		SBB	8	
INC	8		SHL	8	
IRET	22*		SHR	8	
Jcc	12/8 *	(Salta / No salta)	STC	8	
JMP	12 *		STI	8	
LEA	8		SUB	8	
MOV	8		TEST	8	
MOVSX	8		XCHG	8	
MOVZX	8		XOR	8	

Al cost en cicles de cada instrucció s'ha d'afegir el següent, excepte a les instruccions marcades * on tots els accessos a memòria que es produeixen ja han estat comptabilitzats a la taula.

Mode immediat	+2 cicles
Mode memòria amb desplaçament	+2 cicles
Un accés a memòria	+5 cicles
Dos accessos a la mateixa posició de memòria	+7 cicles

Exemples

S'indica el temps que trigaria l'execució de cadascuna de les següents instruccions, suposant que N és una constant i DADA, SUBRUTINA i DESTI són etiquetes de memòria.

```
ADD     EAX, EBX          ; 8 cicles
ADD     EAX, N            ; 10 cicles (8+2 mode immediat)
ADD     EAX, [EBX]        ; 13 cicles (8+5 lectura de  memòria)
MOV     [EBX], EAX        ; 13 cicles (8+5 escriptura a memòria)
ADD     [EBX], EAX        ; 15 cicles (8+7 dos accessos mateixa posició de memòria)
ADD     EAX, DADA         ; 15 cicles (8+2 desplaçament +5 lectura de memòria)
ADD     EAX, DADA[EBX]    ; 15 cicles (8+2 desplaçament +5 lectura de memòria)
ADD     DADA[EBX], EAX    ; 17 cicles (8+2 desplaçament +7 dos acc. mateixa pos. mem.)
CALL    SUBRUTINA         ; 17 cicles (l'accés a l'@ de subrutina ja està comptabilitzat)
JMP     DESTI             ; 12 cicles (l'accés a l'@ de destí  ja està comptabilitzat)
LEA     EAX, DADA[EBX]    ; 10 cicles (8+2 desplaçament, NO hi ha accés a memòria)
PUSH    EAX               ; 10 cicles
POP     DADA[EBX]         ; 17 cicles (10+2 desplaçament +5 escriptura a memòria)
IN      AL, 60H           ; 15 cicles (13+2 mode immediat)
```