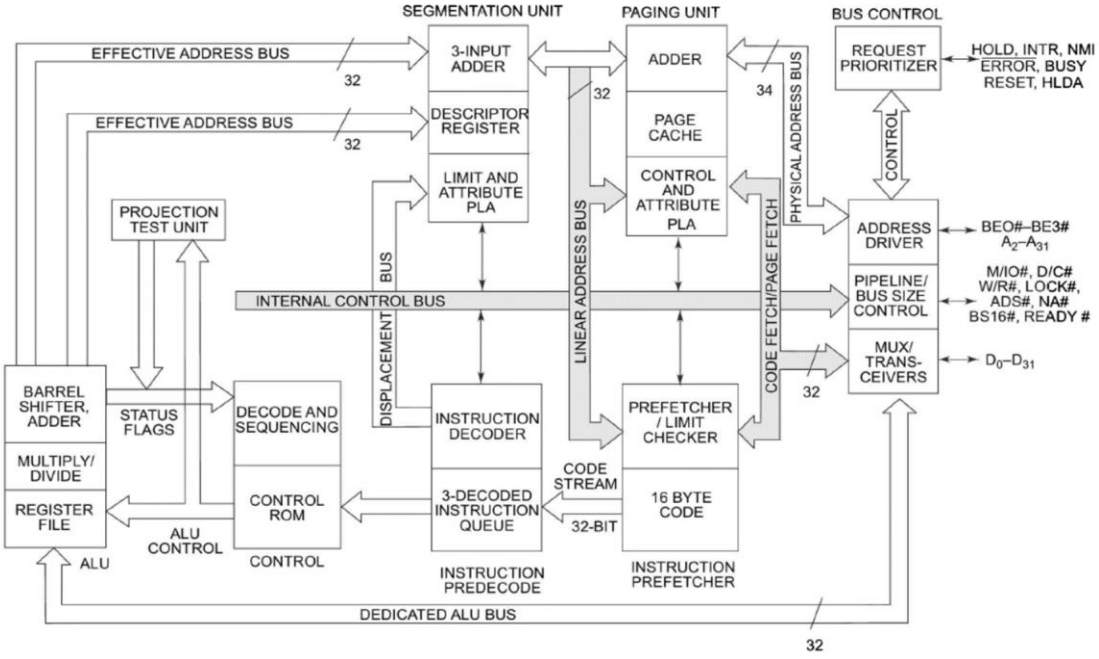
UNIT-1

Features of 80386

- 32-bit microprocessor
- 32-bit ALU
- 32-bit data bus
- 32-bit address bus
- 32-bit registers
- 80386 can address 4GB of primary memory and 64TB of virtual memory.
- Clock speed: 16, 20, 25, 33 MHz
- 3 stage pipeline (Fetch, Decode, Execute)
- 64TB virtual memory
- Segmentation
- Paging
- Protection mechanism
- Multitasking

Architecture of 80386DX



The Internal Architecture of 80386 is divided into 3 sections.

1. Central processing unit (CPU)
    a. Execution Unit
    b. Instruction Unit
2. Memory management unit (MMU)
    a. Segmentation Unit
    b. Paging Unit
3. Bus interface unit (BIU)

The CPU is divided into two units: (a) Execution Unit (b) Instruction Unit

- The execution unit has eight general purpose and eight special purpose registers which are either used for handling data or calculating offset addresses.
- The instruction unit decodes the opcode bytes received from the 16-byte instruction code queue and arranges them into a 3-instruction decoded-instruction queue.
- The barrel shifter increases the speed of all shift and rotate operations.
- The multiply/divide logic implements the bit shift-rotate algorithms to complete the operations in minimum time.

The MMU consists of a segmentation unit and a paging unit.

- The segmentation unit allows the use of two address components i.e. segment and offset for relocatability and sharing of code and data.
- The segmentation unit allows a maximum size of 4GB segments.
- The segmentation unit provides a four level protection mechanism for protecting and isolating the system's code and data from those of the application program.
- The paging unit organizes the physical memory in terms of pages of 4Kbytes size each.
- The paging unit converts linear addresses into physical addresses.

The BIU has a prioritizer to resolve the priority of the various bus requests. This controls the access of the bus.

- It provides a full 32 bit bi-directional data and 32- bit address bus.
- **Instruction Prefetch Unit:** Fetches sequentially the instruction byte stream from the memory. It uses bus control unit to fetch instruction bytes when it is not performing bus cycles. These prefetched instruction bytes are stored in 16 bytes code queue. When jump or call instructions are executed, the contents of the prefetched and decode queues are cleared out.
- **Instruction Predecode Unit :** Takes instruction byte from the instruction prefetch queue and translate them into microcode. The decoded instruction then stored in instruction queue.

Data Types

- Bytes, words, and doublewords are the principal data types.
- A byte is eight bits. The bits are numbered 0 through 7, bit 0 being the least significant bit (LSB).
- A word is two bytes occupying any two consecutive addresses. A word contains 16 bits. The bits of a word are numbered from 0 through 15, bit 0 again being 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. On the 386 DX microprocessor, the low byte is stored in the byte with the lower address.

- A doubleword is four bytes occupying any four consecutive addresses. A doubleword contains 32 bits. The bits of a doubleword are numbered from 0 through 31, bit 0 again being the least significant bit.

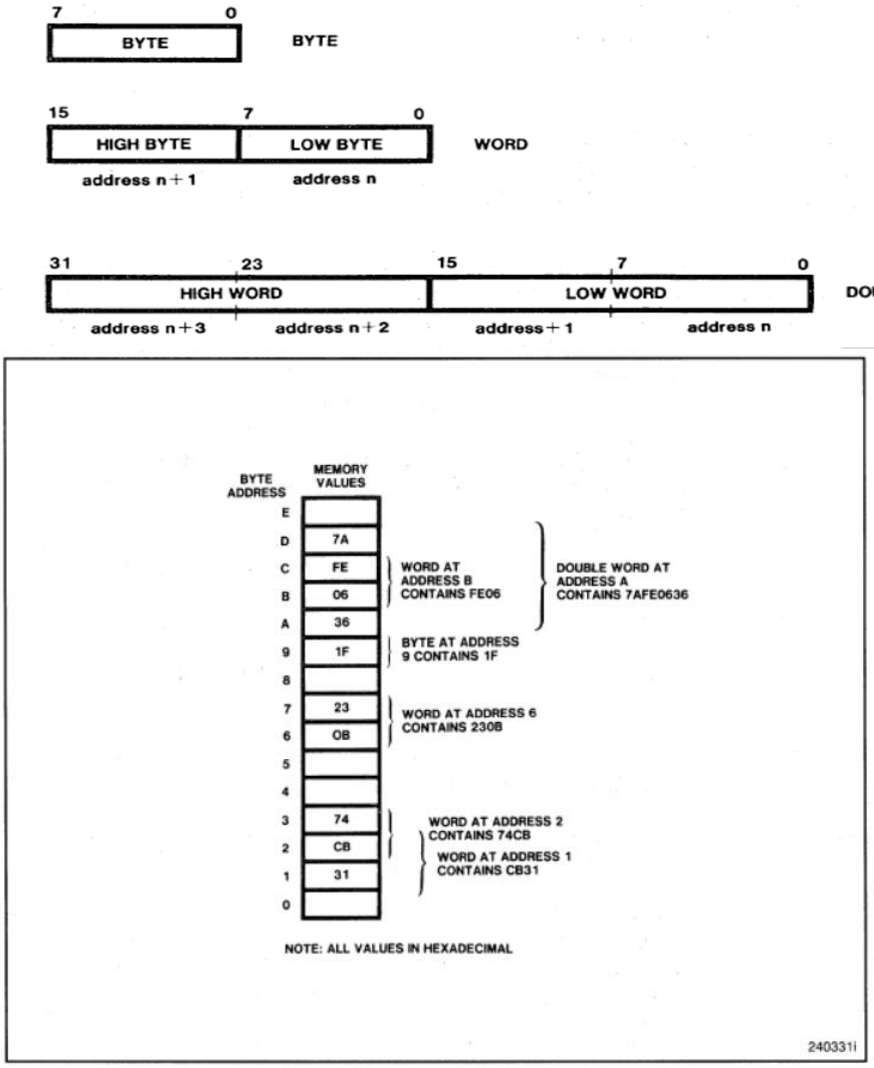


Figure 2-3. Bytes, Words, and Doublewords in Memory

Integer: A signed binary number held in a 32-bit doubleword, 16-bit word, or 8-bit byte. All operations assume a two'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 is set for negative integers, clear for positive integers and zero. The value of an 8-bit integer is from -128 to + 127; a 16-bit integer from - 32,768 to + 32,767; a 32-bit integer from - 2^{31} to $+2^{31-1}$.

Ordinal: An unsigned binary number contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. The value of an 8- bit ordinal is from 0 to 255; a 16-bit ordinal from 0 to 65,535; a 32-bit ordinal from 0 to 2^{32} - 1.

Near Pointer: A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used for all pointers in a flat memory model, or for references within a segment in a segmented model.

Far Pointer: A 48-bit logical address consisting of a 16-bit segment selector and a 32-bit offset. Far pointers are used in a segmented memory model to access other segments.

String: A contiguous sequence of bytes, words, or doublewords. A string may contain from zero to 2^{32} -1 bytes (4 gigabytes).

Bit field: A contiguous sequence of bits. A bit field may begin at any bit position of any byte and may contain up to 32 bits.

Bit string: A contiguous sequence of bits. A bit string may begin at any bit position of any byte and may contain up to 2^{32} - 1 bits.

BCD: A representation of a binary-coded decimal (BCD) digit in the range 0 through 9. Unpacked decimal numbers are stored as unsigned byte quantities. One digit is stored in each byte. The magnitude of the number is the binary value of the low-order half-byte; values 0 to 9 are valid and are interpreted as the value of a digit. The high-order half-byte must be zero during multiplication and division; it may contain any value during addition and subtraction.

Packed BCD: A representation of binary-coded decimal digits, each in the range 0 to 9. One digit is stored in each half-byte, two digits in each byte. The digit in bits 4 to 7 is more significant than the digit in bits 0 to 3. Values 0 to
9 are valid for a digit.

REGISTERS

The 386 DX microprocessor contains sixteen registers which may be used by an application programmer. As Figure 2-5 shows, these registers may be grouped as:

1. **General Registers**. These eight 32-bit registers are free for use by the programmer.

2. **Segment registers**. These registers hold segment selectors associated with different forms of memory access. For example, there are separate segment registers for access to code and stack space. These six registers determine, at any given time, which segments of memory are currently available.

3. **Status and control registers.** These registers report and allow modification of the state of the 386 DX microprocessor.

General Registers

The general registers; are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used to hold operands for logical and arithmetic operations. They also may be used to hold operands for address calculations (except the ESP register cannot be used as an index operand). The names of these registers are derived from the names of the general registers on the 8086 processor, the AX, BX, CX, DX, BP, SP, SI, and DI registers.

Each byte of the 16-bit registers AX, BX, CX, and DX also have other names. 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 address calculations and for the results of most arithmetic and logical operations; however, a few instructions assign specific registers to hold operands.

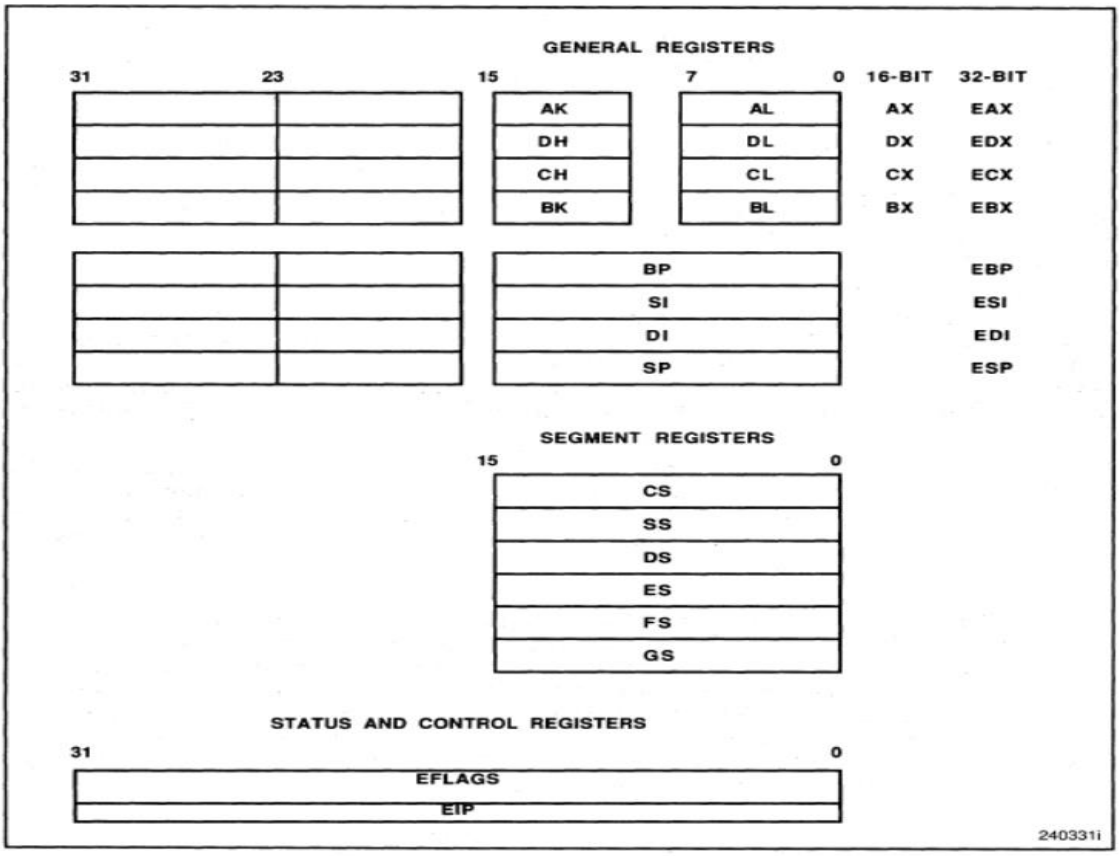


Figure 2-5. Application Register Set

Segment Registers

Segmentation gives system designers the flexibility to choose among various models of memory organization. The segment registers contain 16-bit segment selectors, which index into tables in memory. The tables hold the base address for each segment, as well as other information regarding memory access.

At any instant, up to six segments of memory are immediately available. The segment registers CS, DS, SS, ES, FS, and GS hold the segment selectors for these six segments. Each register is associated with a particular kind of memory access (code, data, or stack). Each register specifies a segment, from among the segments used by the program, which is used for its kind of access.

The segment containing the instructions being executed is called the code segment. Its segment selector is held in the CS register. The 386 DX microprocessor fetches instructions from the code segment, using the contents of the ElP register as an offset into the segment.

Table 2-1. Register Names

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

Stack Implementation

Stack operations are supported by three registers:

1. Stack Segment (SS) Register: Stacks reside in memory. The number of stacks in a system is limited only by the maximum number of segments. A stack may be up to 4 gigabytes long, the maximum size of a segment on the 386 DX microprocessor. One stack is available at a time - the stack whose segment selector is held in the SS register. This is the current stack, often referred to simply as "the" stack. The SS register is used automatically by the processor for all stack operations.

2. Stack Pointer (ESP) Register: The ESP register holds an offset to the top-of-stack (TOS) in the current stack segment. It is used by PUSH and POP operations, subroutine calls and returns, exceptions, and interrupts. When an item is pushed onto the stack, the processor decrements the ESP register, then writes the item at the new TOS. When an item is popped off the stack, the processor copies it from the TOS, then increments the ESP register. In other words, the stack grows down in memory toward lesser addresses.

3. Stack-Frame Base Pointer (EBP) Register: The EBP register typically is used to access data structures passed on the stack. For example, on entering a subroutine the stack contains the return address and some number of data structures passed to the subroutine. The subroutine adds to the stack whenever it needs to create space for temporary local variables. As a result, the stack pointer moves around as temporary variables. are pushed and popped.

Flags Register

Condition codes (e.g., carry, sign, overflow) and mode bits are kept in a 32-bit register named EFLAGS. Figure 2-9 defines the bits within this register. The flags control certain operations and indicate the status of the 386 DX microprocessor.

The flags may be considered in three groups: status flags, control flags, and system flags.

Status Flags

The status flags of the EFLAGS register report the kind of result produced from the execution of arithmetic instructions. For example, when the counter controlling a loop is decremented to zero, the state of the ZF flag changes, and this change can be used to suppress the conditional jump to the start of the loop.

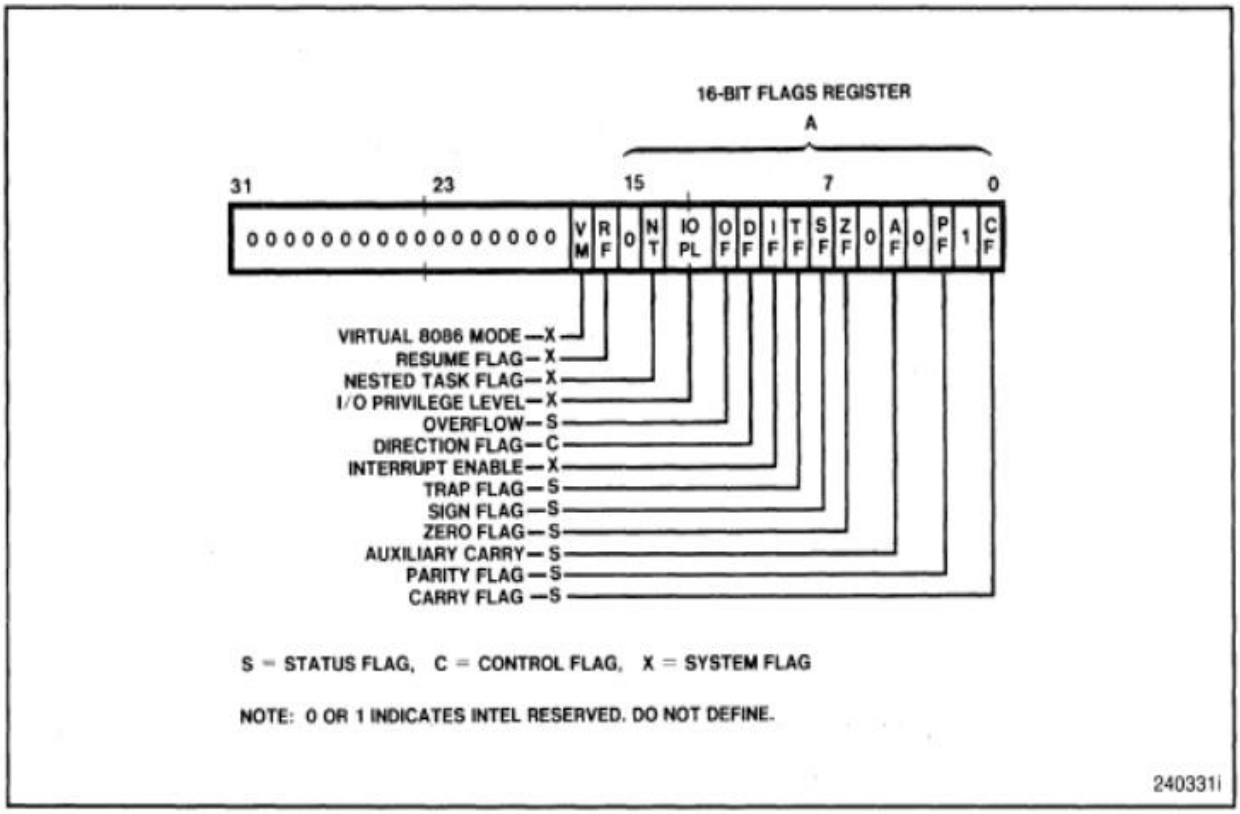


Figure 2-9. EFLAGS Register

Table 2-2. Status Flags

Name	Purpose	Condition Reported
OF	overflow	Result exceeds positive or negative limit of number range
SF	sign	Result is negative (less than zero)
ZF	zero	Result is zero
AF	auxiliary carry	Carry out of bit position 3 (used for BCD)
PF	parity	Low byte of result has even parity (even number of set bits)
CF	carry flag	Carry out of most significant bit of result

Control Flag

The control flag DF of the EFLAGS register controls string instructions. DF (Direction Flag, bit 10) Setting the DF flag causes string instructions to auto-decrement, that is, to process strings from high addresses to low addresses_ Clearing the DF flag causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

Instruction Pointer

The instruction pointer (EIP) register contains the offset into the current code segment for the next instruction to execute. The instruction pointer is not directly available to the programmer; it is

controlled implicitly by control-transfer instructions (jumps, returns, etc.), interrupts, and exceptions.

INSTRUCTION FORMAT

Prefix	Opcode	Register specifier	Addressing-mode specifier	SIB byte	Displacement	Immediate operand

Prefixes: one or more bytes preceding an instruction which modify the operation of the instruction. The following prefixes can be used by application programs:

1. Segment override-explicitly specifies which segment register an instruction should use, instead of the default segment register.
2. Address size-switches between 16- and 32~bit addressing. Either size can be the default; this prefix selects the non-default size.
3. Operand size-switches between 16- and 32-bit data size. Either size can be the default; this prefix selects the non-default size.
4. Repeat-used with a string instruction to cause the instruction to be repeated for each element of the string.

Opcode: specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different form of the operation.

Register specifier: an instruction may specify one or two register operands. Register specifiers 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 an index register will be used to calculate the address of an operand, a 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 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 8 bits. The 8-bit form is used in the common case when the displacement is sufficiently small. The processor extends an 8-bit displacement to 16 or 32 bits, taking into account the sign.

Immediate operand: when present, directly provides the value of an operand. Immediate operands may be bytes, words, or doublewords. In cases where an 8-bit immediate operand is used with a 16- or 32-bit operand, the processor extends the eight-bit operand to an integer of the same sign and magnitude in the larger size. In the same way, a 16-bit operand is extended to 32-bits.

OPERAND SELECTION

An instruction acts on zero or more operands. An example of a zero-operand instruction is the NOP instruction (no operation). An operand can be held in any of these places:
In the instruction itself (an immediate operand).

• **In a register** (in the case of 32-bit operands, EAX, EBX, ECX, ED X, ESI, EDI, ESP, or EBP; in the case of 16-bit operands AX, BX, CX, DX, SI, DI, SP, or BP; in the case of 8-bit operands AH, AL, BH, BL, CH, CL, DH, or DL; the segment registers; or the EFLAGS register for flag operations). Use of 16-bit register operands requires use of the 16-bit operand size prefix (a byte with the value 67H preceding the instruction).
• **In memory**.
• **At an I/O port**.

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. It may be a byte, word, or doubleword. 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.

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, OX, SI, 01, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, or DL). An instruction which uses 16-bit register operands must use the 16-bit operand size prefix (a byte with the value 67H before the remainder of the instruction).

Memory Operands
Instructions with explicit operands in memory must reference the segment containing the operand and the offset from the beginning of the segment to the operand.

Data Movement Instructions
These instructions provide convenient methods for moving bytes, words, or doublewords between memory and the processor registers.
They come in three types:

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

<u>General-Purpose Data Movement Instructions</u>
1.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:

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

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

Operation
DEST ← SRC;

Syntax
MOV Destination,Source

Examples

1.MOV EAX, [N1]
2.MOV [SUM],EBX
3.MOV EAX,EBX
4.MOV EAX,'1'
5.MOV [N2],'2'

2.XCHG (Exchange) swaps the contents of two operands. 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 and a memory operand.

Operation
temp ← DEST
DEST ← SRC
SRC ←~ temp

Syntax
XCHG Operand-1, Operand-2

Stack Manipulation Instructions

1. **PUSH (Push)** decrements the stack pointer (ESP register), then copies the source operand to the top of stack. The PUSH instruction often is used to place parameters on the stack before calling a procedure. Inside a procedure, it can be used to reserve space on the stack for temporary variables. The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers).

2. **POP (Pop)** transfers the word or doubleword at the current top of stack (indicated by the ESP register) to the destination operand, and then increments the ESP register to point to the new top of stack. See Figure 3-3. POP moves information from the stack to a general register, segment register, or to memory. A special form of the POP instruction is available for popping a doubleword from the stack to a general register.

BINARY ARITHMETIC INSTRUCTIONS

The arithmetic instructions of the 386 DX microprocessor operate on numeric data encoded in binary. Operations include the add, subtract, multiply, and divide as well as increment, decrement, compare, and change sign (negate). Both signed and unsigned binary integers are supported.

Source operands can be immediate values, general registers, or memory. Destination operands can be general registers or memory (except when the source operand is in memory). The basic arithmetic instructions have special forms for using an immediate value as the source operand and the AL or EAX registers as the destination operand.

The arithmetic instructions update the ZF, CF, SF, and OF flags to report the kind of result which was produced.
Arithmetic instructions operate on 8-, 16-, or 32-bit data. The flags are updated to reflect the size of the operation. For example, an 8-bit ADD instruction sets the CF flag if the sum of the operands exceeds 255 (decimal).

The INC and DEC instructions do not change the state of the CF flag. This allows the instructions to be used to update counters used for loop control without changing the reported state of arithmetic results. To test the arithmetic state of the counter, the ZF flag can be tested to detect loop termination, or the ADD and SUB instructions can be used to update the value held by the counter.

Addition and Subtraction Instructions
ADD (Add Integers) replaces the destination operand with the sum of the source and destination operands. The OF, SF, ZF, AF, PF, and CF flags are affected.

ADD-Add
ADD Dest,Src
The ADD instruction 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.

Operation: DEST +- DEST + SRC;
Flags Affected: The OF, SF, ZF, AF, CF, and PF flags are set according to the result

SUB: Subtract
SUB Dest,Src
The SUB instruction 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: The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

MUL - Unsigned Multiplication of AL or AX
MUL Src
The MUL instruction performs unsigned multiplication. Its actions depend on the size of its operand, as follows:
A byte operand is multiplied by the AL value; the result is left in the AX register. The CF and OF flags are cleared if the AH value is 0; otherwise, they are set.

A word operand is multiplied by the AX value; the result is left in the DX:AX register pair. The DX register contains the high-order 16 bits of the product. The CF and OF flags are cleared if the DX value is 0; otherwise, they are set.
A doubleword operand is multiplied by the EAX value and the result is left in the EDX:EAX register. The EDX register contains the high-order 32 bits of the product. The CF and OF flags are cleared if the EDX value is 0; otherwise, they are set.

DIV- Division

DIV Src

The DIV instruction 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: The OF, SF, ZF, AF, PF, CF flags are undefined.

SHIFT INSTRUCTIONS

- Shift instructions apply an arithmetic or logical shift to bytes, words, and doublewords. An arithmetic shift right copies the sign bit into empty bit positions on the upper end of the operand, while a logical shift right fills clears the empty bit positions.
- There is no difference between an arithmetic shift left and a logical shift left. Two names, SAL and SHL, are supported for this instruction in the assembler.
- A count specifies the number of bit positions to shift an operand. Bits can be shifted up to 31 places. A shift instruction can give the count in any of three ways.
- One form of shift instruction always shifts by one bit position. The second form gives the count as an immediate operand. The third form gives the count as the value contained in the CL register.
- When the number of bit positions to shift is zero, no flags are affected. Otherwise, the CF flag is left with the value of the last bit shifted out of the operand.
- In a single-bit shift, the OF flag is set if the value of the uppermost bit (sign bit) was changed by the operation. Otherwise, the OF flag is cleared.
- After a shift of more than one bit position, the state of the OF flag is undefined. On a shift of one or more bit positions, the SF, ZF, PF, and CF flags are affected, and the state of the AF flag is undefined.

SAL (Shift Arithmetic Left)/SHL (Shift Logical Left)

- SAL (Shift Arithmetic Left) shifts the destination byte, word, or doubleword operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). Empty bit positions are cleared.
- The high-order bit is shifted into the CF flag, and the low-order bit is cleared.
- SHL (Shift Logical Left) is another name for the SAL instruction.
- Examples: SAL EAX, SAL EAX,3, SAL EBX,CL

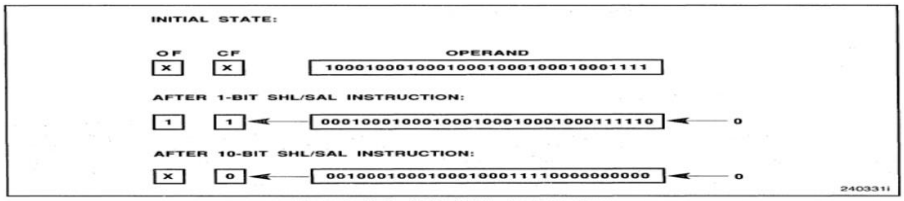


Figure 3-6. SHL/SAL Instruction

SHR (Shift Logical Right)

- SHR (Shift Logical Right) shifts the destination byte, word, or doubleword operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). Empty bit positions are cleared.
- Examples: SHR EBX, SHR EAX, 4, SHR EAX, CL

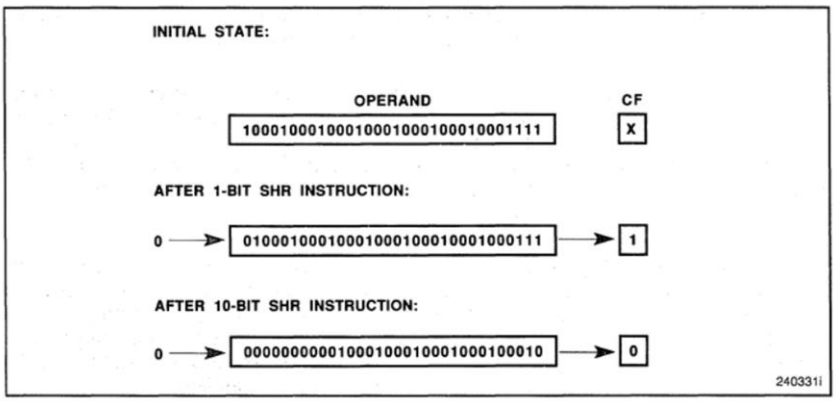


Figure 3-7. SHR Instruction

SAR (Shift Arithmetic Right)

- SAR (Shift Arithmetic Right) shifts the destination byte, word, or doubleword operand to the right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).
- The sign of the operand is preserved by clearing empty bit positions if the operand is positive, or setting the empty bits if the operand is negative.

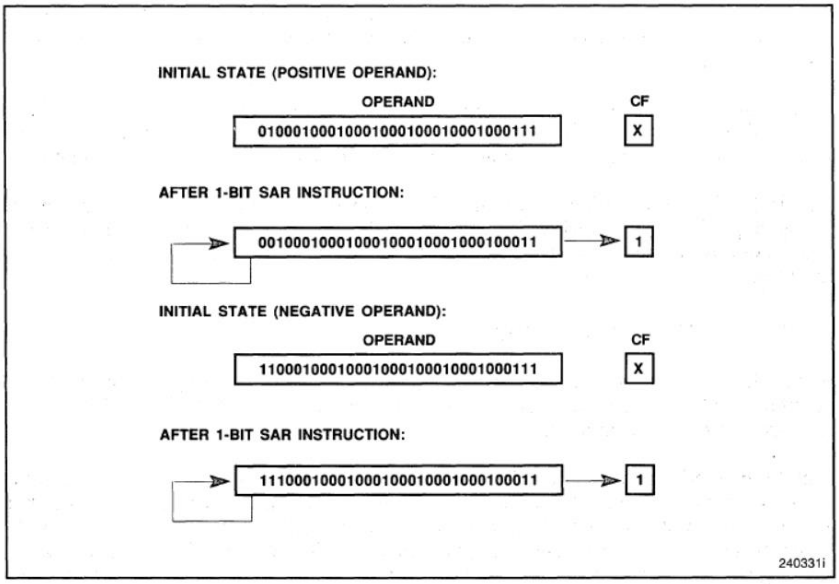


Figure 3-8. SAR Instruction

ROTATE INSTRUCTIONS

- Rotate instructions apply a circular permutation to bytes, words, and doublewords. Bits rotated out of one end of an operand enter through the other end. Unlike a shift, no bits are emptied during a rotation.
- 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.
- Rotate instructions use only the CF and OF flags.
- The CF flag 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).
- The CF flag always contains the value of the last bit rotated out of the operand, even if the instruction does not use the CF flag as an extension of the operand.
- The state of the SF, ZF, AF, and PF flags is not affected.

ROL (Rotate Left)

- ROL (Rotate Left) rotates the byte, word, or doubleword destination operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).
- For each bit position of the rotation, the bit which exits from the left of the operand returns at the right.
- Examples: ROL EAX, ROL EAX, 4, ROL EAX, CL

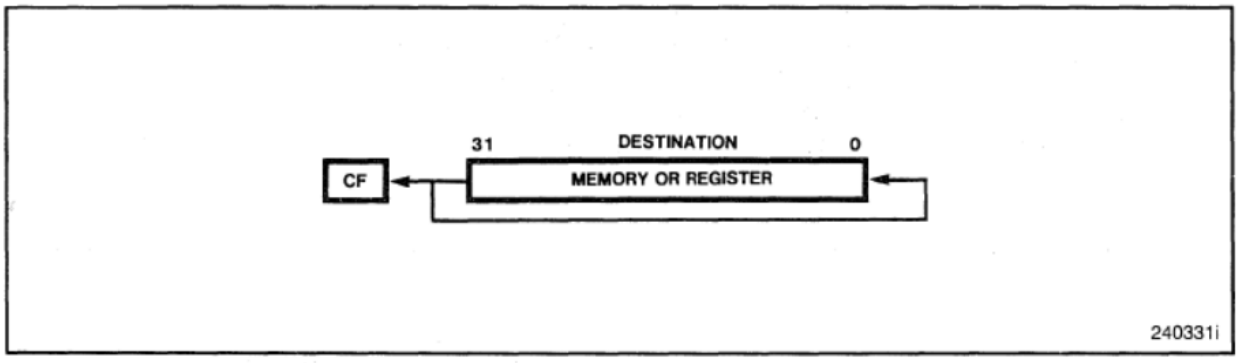


Figure 3-11. ROL Instruction

ROR (Rotate Right)

- ROR (Rotate Right) rotates the byte, word, or doubleword destination operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register). For each bit position of the rotation, the bit which exits from the right of the operand returns at the left.
- Examples: ROR EAX, ROR EAX, 4, ROR EAX, CL

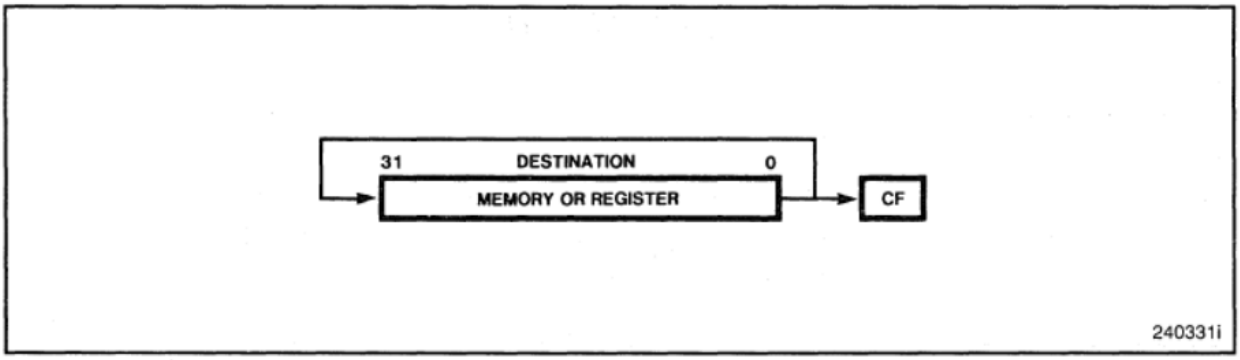


Figure 3-12. ROR Instruction

RCL (Rotate Through Carry Left)

- RCL (Rotate Through Carry Left) rotates bits in the byte, word, or doubleword destination operand left by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).
- This instruction differs from ROL in that it treats the CF flag as a one-bit extension on the upper end of the destination operand.
- Each bit which exits from the left side of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the right side.

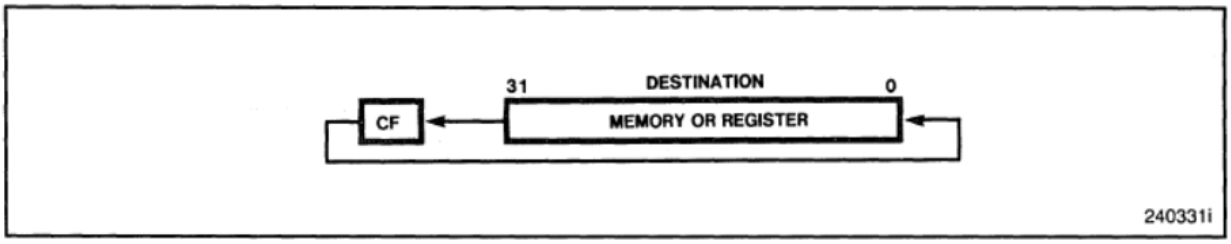


Figure 3-13. RCL Instruction

RCR (Rotate Through Carry Right)

- RCR (Rotate Through Carry Right) rotates bits in the byte, word, or doubleword destination operand right by one bit position or by the number of bits specified in the count operand (an immediate value or a value contained in the CL register).
- This instruction differs from ROR in that it treats CF as a one-bit extension on the lower end of the destination operand. Each bit which exits from the right side of the operand moves into the CF flag. At the same time, the bit in the CF flag enters the left side.

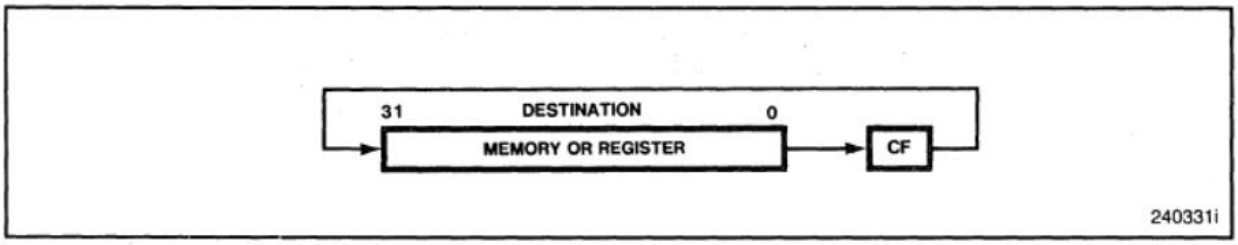


Figure 3-14. RCR Instruction

STRING OPERATIONS

- String operations manipulate large data structures in memory, such as alphanumeric character strings.

- The string operations are made by putting string instructions (which execute only one iteration of an operation) together with other features of the lntel386 architecture, such as repeat prefixes. The string instructions are:
  - ***MOVS*** *- Move String*
  - ***CMPS*** *- Compare string*
  - ***SCAS*** *- Scan string*
  - ***LODS*** *- Load string*
  - ***STOS*** *- Store string*
- After a string instruction executes, the string source and destination registers point to the next elements in their strings. These registers automatically increment or decrement their contents by the number of bytes occupied by each string element.
- A string element can be a byte, word, or doubleword.
- The string registers are: ***ESl - Source index register, EDl - Destination index register***
- String operations can begin at higher addresses and work toward lower ones, or they can begin at lower addresses and work toward higher ones. The direction is controlled by: ***DF -Direction flag***
- If the DF flag is clear, the registers are incremented, If the flag is set, the registers are decremented.
- To operate on more than one element of a string, a repeat prefix must be used, such as:
  - ***REP*** *- Repeat while the ECX register not zero*
  - ***REPE/REPZ*** *- Repeat while the ECX register not zero and the ZF flag is set*
  - ***REPNE/REPNZ*** *- Repeat while the ECX register not zero and the ZF flag is clear*
- These instructions set and clear the flag:
  - ***STD*** *- Set direction flag instruction*
    CLD *- Clear direction flag instruction*

Repeat Prefixes

- The repeat prefixes REP (Repeat While ECX Not Zero), REPE/REPZ (Repeat While Equal/Zero), and REPNE/REPNZ (Repeat While Not Equal/Not Zero) specify repeated operation of a string instruction.
- When a string instruction has a repeat prefix, the operation executes until one of the termination conditions specified by the prefix is satisfied.
- All three prefixes shown in Table 3-4 cause the instruction to repeat until the ECX register is decremented to zero, if no other termination condition is satisfied. The repeat prefixes differ in their other termination condition.
- The REP prefix has no other termination condition.
- The REPE/REPZ and REPNE/REPNZ prefixes are used exclusively with the SCAS (Scan String) and CMPS (Compare String) instructions. The REPE/ REPZ prefix terminates if the ZF flag is clear.
- The REPNE/REPNZ prefix terminates if the ZF flag is set. The ZF flag does not require initialization before execution of a repeated string instruction, because both the SCAS and CMPS instructions affect the ZF flag according to the results of the comparisons they make.

Table 3-4. Repeat Instructions

Repeat Prefix	Termination Condition 1	Termination Condition 2
REP	ECX=0	none
REPE/REPZ	ECX=0	ZF=0
REPNE/REPNZ	ECX=0	ZF=1

Indexing and Direction Flag Control

- The string instructions require the use of two specific registers. The source and destination strings are in memory addressed by the ESI and EDI registers.
- The ESI register points to source operands. By default, the ESI register is used with the DS segment register. A segment-override prefix allows the ESI register to be used with the CS, SS, ES, FS, or GS segment registers.
- The EDI register points to destination operands. It uses the segment indicated by the ES segment register; no segment override is allowed. The use of two different segment registers in one instruction permits operations between strings in different segments.
- When ESI and EDI are used in string instructions, they automatically are incremented or decremented after each iteration. String operations can begin at higher addresses and work toward lower ones, or they can begin at lower addresses and work toward higher ones.
- The direction is controlled by the DF flag. If the flag is clear, the registers are incremented. If the flag is set, the registers are decremented. The STD and CLD instructions set and clear this flag.

String Instructions

MOVS (Move String) moves the string element addressed by the ESI register to the location addressed by the EDI register.

- The MOVSB instruction moves bytes, the MOVSW instruction moves words, and the MOVSD instruction moves doublewords.
- The MOVS instruction, when accompanied by the REP prefix, operates as a memory-to-memory block transfer. To set up this operation, the program must initialize the ECX, ESI, and EDI registers. The ECX register specifies the number of elements in the block.

CMPS (Compare Strings) subtracts the destination string element from the source string element and updates the AF, SF, PF, CF and OF flags.

- Neither string element is written back to memory. If the string elements are equal, the ZF flag is set; otherwise, it is cleared.
- CMPSB compares bytes, CMPSW compares words, and CMPSD compares doublewords.

SCAS (Scan String) subtracts the destination string element from the EAX, AX, or AL register (depending on operand length) and updates the AF, SF, ZF, PF, CF and OF flags.

- The string and the register are not modified.
- If the values are equal, the ZF flag is set; otherwise, it is cleared.
- The SCASB instruction scans bytes; the SCASW instruction scans words; the SCASD instruction scans doublewords.

LODS (Load String) places the source string element addressed by the ESI register into the EAX register for doubleword strings, into the AX register for word strings, or into the AL register for byte strings.

- This instruction usually is used in a loop, where other instructions process each element of the string as they appear in the register.

STOS (Store String) places the source string element from the EAX, AX, or AL register into the string addressed by the EDI register.

- This instruction usually is used in a loop, where it writes to memory the result of processing a string element read from memory with the LODS instruction. A REP STOS instruction is the fastest way to initialize a large block of memory.

INSTRUCTIONS FOR BLOCK-STRUCTURED LANGUAGES

- These instructions provide machine-language support for implementing block-structured languages, such as C and Pascal.
- They include *ENTER and LEAVE*, which simplify procedure entry and exit in compiler-generated code. They support a structure of pointers and local variables on the stack called a stack frame.

- **ENTER (Enter Procedure)** creates· a stack frame compatible with the scope rules of block-structured languages. In these languages, a procedure has access to its own variables and some number of other variables defined elsewhere in the program. The scope of a procedure is the set of variables to which it has access.
  - *The ENTER instruction has two operands. The first specifies the number of bytes to be reserved on the stack for dynamic storage in the procedure being entered. Dynamic storage is the memory allocated for variables created when the procedure is called, also known as automatic variables.*
  - *The second parameter is the lexical nesting level (from 0 to 31) of the procedure. The nesting level is the depth of a procedure in the hierarchy of a block-structured program. The lexical level has no particular relationship to either the protection privilege level or to the I/O privilege level.*
  - ***Example: ENTER 2048,3** Allocates 2K bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame for this procedure.*

- **LEAVE (Leave Procedure)** reverses the action of the previous ENTER instruction. The LEAVE instruction does not have any operands. The LEAVE instruction copies the contents of the EBP register into the ESP register to release all stack space allocated to the procedure. Then the LEAVE instruction restores the old value of the EBP register from the stack. This simultaneously restores the ESP register to its original value. A subsequent RET instruction then can remove any arguments and the return address pushed on the stack by the calling program for use by the procedure.

FLAG CONTROL INSTRUCTIONS

- The flag control instructions change the state of bits in the EFLAGS register.
- **Carry and Direction Flag Control Instructions**
  - *The **carry flag** instructions are useful with instructions like the rotate-with-carry instructions RCL and RCR. They can initialize the carry flag, CF, to a known state before execution of an instruction which copies the flag into an operand.*
  - *The **direction flag** control instructions set or clear the direction flag, DF, which controls the direction of string processing. If the DF flag is clear, the processor increments the string index registers, ESI and EDI, after each iteration of a string instruction. If the DF flag is set, the processor decrements these index registers.*

Table 3-5. Flag Control Instructions

Instruction	Effect
STC (Set Carry Flag)	CF ← 1
CLC (Clear Carry Flag)	CF ← 0
CMC (Complement Carry Flag)	CF ← - (CF)
CLD (Clear Direction Flag)	DF ← 0
STD (Set Direction Flag)	DF ← 1

Flag Transfer Instructions

- *The flag transfer instructions allow a program to change the state of the other flag bits using the bit manipulation instructions once these flags have been moved to the stack or the AH register.*

- **LAHF (Load AH from Flags)** copies the SF, ZF, AF, PF, and CF flags to the AH register bits 7, 6, 4, 2, and 0, respectively (see Figure 3-21). The contents of the remaining bits 5, 3, and 1 are left undefined. The contents of the EFLAGS register remain unchanged.
- **SAHF (Store AH into Flags)** copies bits 7,6,4,2, and 0 from the AH register into the SF, ZF, AF, PF, and CF flags, respectively. The SF, ZF, AF, PF, and CF flags are loaded with values form the AH register.
- The **PUSHF and POPF** instructions are not only useful for storing the flags in memory where they can be examined and modified, but also are useful for preserving the state of the EFLAGS register while executing a subroutine.
- **PUSHF (Push Flags)** pushes the lower word of the EFLAGS register onto the stack. The PUSHFD instruction pushes the entire EFLAGS register onto the stack (the RF flag reads as clear, however).
  - *The PUSHF instruction decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; the PUSHFD instruction decrements the stack pointer by 4, and the 386 DX microprocessor EFLAGS register is copied to the new top of stack which is pointed to by SS:ESP.*
- **POPF (Pop Flags)** pops a word from the stack into the EFLAGS register. Only bits 14, 11, 10, 8, 7, 6, 4, 2, and 0 are affected with all uses of this instruction. If the privilege level of the current code segment is 0 (most privileged), the IOPL bits (bits 13 and 12) also are affected.
- **The POPFD** instruction pops a doubleword into the EFLAGS register, but it only can change the state of the same bits affected by a POPF instruction.
- **The POPF and POPFD** instructions pop the word or doubleword on the top of the stack and store the value in the flags register. If the operand-size attribute of the instruction is 16 bits, then a word is popped and the value is stored in the FLAGS register. If the operand-size attribute is 32 bits, then a doubleword is popped and the value is stored in the EFLAGS register.

COPROCESSOR INTERFACE INSTRUCTIONS

- The 387 DX Numeric Coprocessor provides an extension to the instruction set of the base architecture.
- The coprocessor extends the instruction set of the 386 DX microprocessor to support high-precision integer and floating-point calculations.
- These extensions include arithmetic, comparison, transcendental, and data transfer instructions. The coprocessor also contains frequently-used constants, to enhance the speed of numeric calculations.

- The coprocessor instructions are embedded in the instructions for the 386 DX microprocessor, as though they were being executed by a single processor having both integer and floating-point capabilities. But the coprocessor actually works in parallel with the 386 microprocessor, so the performance is higher.
- The 386 DX microprocessor also has features to support emulation of the numeric coprocessor when the coprocessor is absent. The software emulation of the coprocessor is transparent to application software, but much slower.

- **ESC (Escape)** is a bit pattern which identifies floating-point arithmetic instructions.
  - *The ESC bit pattern tells the processor to send the opcode and operand addresses to the numeric coprocessor. The coprocessor uses instructions containing the ESC bit pattern to perform high-performance, high-precision floating point arithmetic. When the coprocessor is not present, these instructions generate coprocessor-not-available exceptions.*

- **WAIT (Wait)** is an instruction which suspends program execution while the BUSY# pin is active. The signal on this pin indicates that the coprocessor has not completed an operation. When the operation completes, the processor resumes execution and can read the result.
  - *The WAIT instruction is used to synchronize the processor with the coprocessor. Typically, a coprocessor instruction is launched, a WAIT instruction is executed, then the results of the coprocessor instruction are read.*
  - *Between the coprocessor instruction and the WAIT instruction, there is an opportunity to execute some number of non-coprocessor instructions in parallel with the coprocessor instruction.*

SEGMENT REGISTER INSTRUCTIONS

- **Segment-Register Transfer Instructions**
  - *Forms of the MOV, POP, and PUSH instructions also are used to load and store segment registers. These forms operate like the general-register forms, except that one operand is a segment register. The MOV instruction cannot copy the contents of a segment register into another segment register.*
- **Far Control Transfer Instructions**
  - *The far control-transfer instructions transfer execution to a destination in another segment by replacing the contents of the CS register. The destination is specified by a far pointer, which is a 16-bit segment selector and a 32-bit offset into the segment. The far pointer can be an immediate operand or an operand in memory.*
  - *Far CALL. An inter segment CALL instruction places the values held in the EIP and CS registers on the stack.*
  - *Far RET. An intersegment RET instruction restores the values of the CS and EIP registers from the stack.*

Data Pointer Instructions

  - *The data pointer instructions load a far pointer into the processor registers. A far pointer consists of a 16-bit segment selector, which is loaded into a segment register, and a 32-bit offset into the segment, which is loaded into a general register.*

  - *LDS (Load Pointer Using DS) copies a far pointer from the source operand into the DS register and a general register. The source operand must be a memory operand, and the destination operand must be a general register.*
  - *Example: LDS ESI, STRING_X*
- **LES (Load Pointer Using ES)** has the same effect as the LDS instruction, except the segment selector is loaded into the ES register rather than the DS register.
  - *Example: LES EDI, DESTINATION_X*
- **LFS (Load Pointer Using FS)** has the same effect as the LDS instruction, except the FS register receives the segment selector rather than the DS register.
- **LGS (Load Pointer Using GS)** has the same effect as the LDS instruction, except the GS register receives the segment selector rather than the DS register.
- **LSS (Load Pointer Using SS)** has the same effect as the LDS instruction, except the SS register receives the segment selector rather than the DS register.

MISCELLANEOUS INSTRUCTIONS

- **LEA (Load Effective Address)** puts the 32-bit offset to a source operand in memory (rather than its contents) into the destination operand. The source operand must be in memory, and the destination operand must be a general register. This instruction is especially useful for initializing the ESI or EDI registers before the execution of string instructions or initializing the EBX register before an XLAT instruction.
- **No-Operation Instruction NOP** (No Operation) occupies a byte of code space. When executed, it increments the EIP register to point at the next instruction, but affects nothing else.
- **Translate Instruction XLAT** (Translate) replaces the contents of the AL register with a byte read from a translation table in memory. The contents of the AL register are interpreted as an unsigned index into this table, with the contents of the EBX register used as the base address.