Microprocessors & Microcontrollers : Unit-1

Prof. Prajay P. Raikar
Visiting Faculty
Department of Computer Engineering
Goa College of Engineering, Farmagudi-Ponda, Goa
prajayraikar@gmail.com

Table of Contents

Overview

Intel offers a variety of processor architectures; each designed to meet different needs and use cases. Here is an overview of some of the key types:

- **x86 Architecture**

- Applications: Desktop and laptop computers, servers, workstations.

- Popular Processors: Intel Pentium, Intel Celeron.

- **x86-64 Architecture**

- Applications: High-performance desktops, servers, gaming PCs.

- Popular Processors: Intel Core i7-8700K, Intel Core i9-9900K.

- **Intel Core Series** (Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Skylake, etc.)

- Applications: Mainstream desktops, laptops, ultrabooks, workstations.

- Popular Processors:

  - Nehalem: Intel Core i7-920.
  - Sandy Bridge: Intel Core i5-2500K.
  - Ivy Bridge: Intel Core i7-3770K.
  - Haswell: Intel Core i7-4770K.
  - Skylake: Intel Core i7-6700K.
  - Kaby Lake: Intel Core i7-7700K.
  - Coffee Lake: Intel Core i7-8700K.
  - Comet Lake: Intel Core i9-10900K.
  - Ice Lake: Intel Core i7-1065G7.
  - Tiger Lake: Intel Core i7-1185G7.
  - Alder Lake: Intel Core i9-12900K.

- **Intel Atom Series**

- Applications: Mobile devices, tablets, low-power servers, embedded systems, IoT devices.

- Popular Processors: Intel Atom Z3740, Intel Atom x7-Z8750.

- **Intel Xeon Series**

- Applications: Servers, data centres, high-performance computing, professional workstations.

- Popular Processors:

  - Westmere: Intel Xeon X5650.

  - Sandy Bridge: Intel Xeon E5-2680.

  - Haswell: Intel Xeon E5-2699 v3.

  - Skylake: Intel Xeon Gold 6148.

  - Cascade Lake: Intel Xeon Platinum 8280.

  - Ice Lake: Intel Xeon Platinum 8375C.

- **Intel Celeron and Pentium Series**

- Applications: Budget desktops, entry-level laptops, educational devices.

- Popular Processors:

  - Celeron: Intel Celeron G3900.

  - Pentium: Intel Pentium G4560.

- **Intel Itanium Series**

- Applications: High-end servers, mission-critical applications, enterprise environments.

- Popular Processors: Intel Itanium 9350, Intel Itanium 9700.

- **Intel Quark Series**

- Applications: Internet of Things (IoT) devices, wearable technology, simple and low-power tasks.

- Popular Processors: Intel Quark X1000, Intel Quark D2000.

- **Intel FPGA** (Field-Programmable Gate Array)

- Applications: Custom hardware acceleration, data processing, signal processing, specialized applications.

- Popular Processors: Intel Stratix 10, Intel Arria 10.

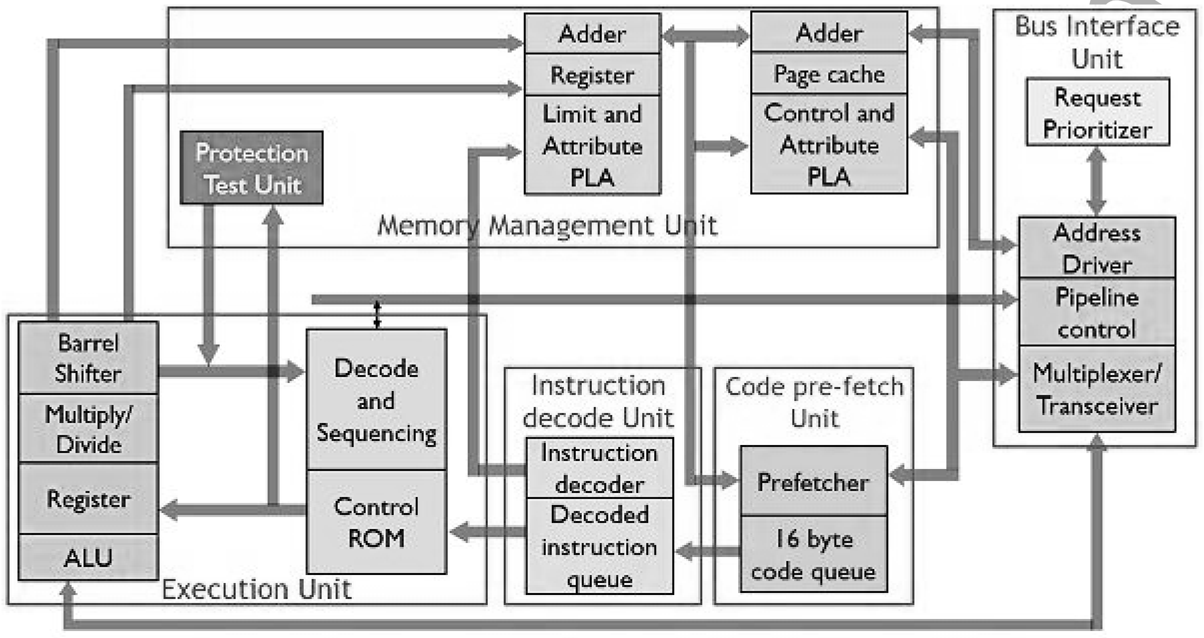
Each of these processor architectures is designed to meet specific performance, power efficiency, and application requirements, providing a wide range of options for various computing needs.

Intel’s 8086 8088 80286 80386 core2 Pentium processors

- **Intel 8086 (1978)**:
    - The first 16-bit microprocessor.
    - Address Bus Width: 20 bits
    - Addressable Memory: 1 MB (Megabyte)
    - Basis for the x86 architecture.
    - Used in early IBM PCs.
- **Intel 8088 (1979)**:
    - Like the 8086, but with an 8-bit external bus.
    - Address Bus Width: 20 bits
    - Addressable Memory: 1 MB (Megabyte)
    - Lower cost, used in the original IBM PC.
- **Intel 80286 (1982)**:
    - Address Bus Width: 24 bits
    - Addressable Memory: 16 MB (Megabytes)
    - Introduced protected mode, allowing for more memory management.
    - Widely used in IBM PC/AT and compatible computers.
- **Intel 80386 (1985)**:
    - First 32-bit processor in the x86 family.
    - Address Bus Width: 32 bits
    - Addressable Memory: 4 GB (Gigabytes)
    - Introduced virtual memory and paging capabilities.
    - Used in high-performance workstations and servers.
- **Intel Pentium (1993)**:
    - Renowned for its superscalar architecture, enabling multiple instructions per clock cycle.
    - Became synonymous with consumer computing in the 90s.
- **Intel Core2 (2006)**:
    - First Intel processor to feature a dual-core architecture.

- Known for its efficiency and performance boost, ideal for multitasking and more demanding applications.

Intel’s 80386-System Architecture



The internal architecture of 80386 is divided into three sections viz.,

Central Processing Unit

Execution unit

- The execution unit has 8 general purpose and 8 special purpose registers.

Instruction unit

- The instruction unit decodes the opcode bytes received from the 16-byte instruction code queue.
- The **barrel shifter** increases the speed of all shift and rotate operations.

Memory Management Unit

Segmentation unit and Paging unit.

The segmentation unit allows the use of two address components, viz. segment and offset for relocability and sharing of code and data. The segmentation unit allows a maximum size of 4 Gbytes segments. The paging unit organizes the physical memory in terms of pages of 4 Kbytes size each. The paging unit works under the control of the segmentation unit, i.e. each segment is further divided into pages. The virtual memory is also organized in terms of segments and pages by the memory management unit. Control & attribute PLA decides on task priority and responsible for allocating memory for high priority tasks.

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.

Bus Interface Unit

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

General Purpose & Special Purpose Registers

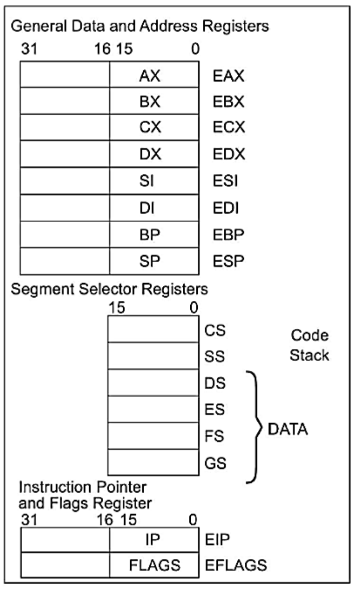
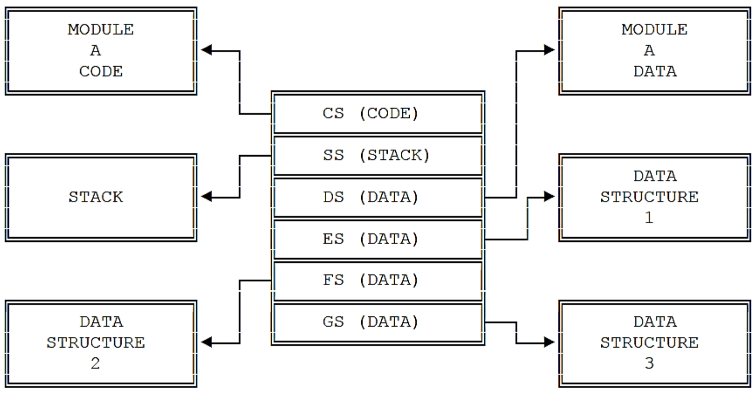


Fig. 10.2(a) *Registers Bank of 80386 (Intel Corp.)*

AX-Accumulator, BX-Base Index Register, CX-Counter Register, DX-Data Register, SI-Source Index Pointer, DI-Destination Index Pointer, SP-Stack Pointer.

E(prefix)-Extended

Figure 2-6. Use of Memory Segmentation



Code Segment (CS)

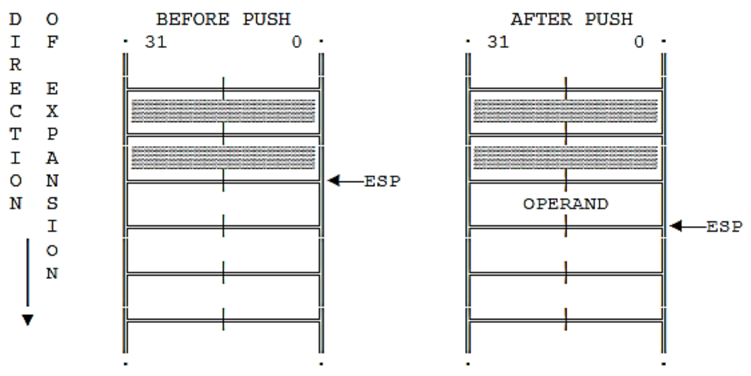
- CS register holds the base/starting memory address of the currently executing program/routine.
- IP register holds the pointer/offset address within code segment of the memory.
- CS & IP is changed implicitly as the result of intersegment control-transfer instructions (for example, CALL and JMP), interrupts, and exceptions.

Stack Segment (SS)

- SS register holds the base/starting memory address of the stack.
- Stack plays crucial role when a subroutine or function or module is called within a program.
- When a subroutine is called, current states of registers can be pushed onto the stack. And reloaded after the end of subroutine.

- It works on LIFO (Last-In-First-Out) principle.
- Stack grows downward in memory.
- SP address decrements when a byte is pushed onto stack.
- SP always points to the top of the stack.
- Hence, SP is influenced by implicit PUSH, POP instructions.
- After pushing the data is placed at Top of the Stack (TOS).
- Another pointer BP (Base Pointer) offset can be used in relative to SS base address.
- BP is used to accessing the dynamically allocated work space within the stack which may contain data structures, variables, etc.
- Stack overflow can occur as a result of infinite recursion or excessive local variables create by a function/subroutine.

Figure 3-1. PUSH



Data Segments (DS, ES, FS, GS)

- DS, ES, FS, GS holds base/starting memory addresses of multiple data structures.
- For 80386, all general-purpose registers can work as pointers.

```
MOV   BX,WORD PTR [ EAX]   ;move into BX word pointed to by EAX
MOV   BX,WORD PTR [ AX]    ;invalid AX can't be used as pointer
MOV   EAX,DWORD PTR [ ECX] ;move into EAX DWORD pointed to by ECX
MOV   AL,BYTE PTR [ EDX]   ;move into AL BYTE pointed to by EDX
MOV   EBX,WORD PTR [ CX]   ;invalid CX can't be used as pointer
MOV   EAX,DWORD PTR [ EDI] ;move into EAX DWORD pointed to by EDI
```

Segment Registers and their default pointers

80386 allows all general-purpose registers to be used as pointers in data segment. Unlike, 8086 which allowed only BX (general purpose register) to be used as pointer.

80386 (32-bit)	8086 (16-bit) (Real-mode of 80386)
CS >> EIP	CS >> IP
SS >> ESP, EBP	SS >> SP, BP
DS >> SI,DI, BX ESI, EDI, EAX, EBX, ECX, EDX	DS >> SI, DI, BX
ES >> SI,DI, BX, ESI, EDI, EAX, EBX, ECX, EDX	ES >> SI, DI, BX
FS >> SI,DI, BX, ESI, EDI, EAX, EBX, ECX, EDX	Not applicable
GS >> SI,DI, BX, ESI, EDI, EAX, EBX, ECX, EDX	Not applicable

Control Registers

The 80386 has three 32-bit control registers CR, CR2 and CR3 to hold global machine status independent of the executed task. The load and store instructions are available to access these registers. The control register CR, is reserved for use in future Intel processors.

System Address Registers

Four special registers are defined to refer to the descriptor tables supported by 80386. The 80386 supports four types of descriptor tables, viz. Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), Local Descriptor Table (LDT) and Task State Segment Descriptor (TSS).

The system address registers and system segment registers hold the addresses of these descriptor tables and the corresponding segments. These registers are known as **GDTR, IDTR, LDTR and TR** respectively.

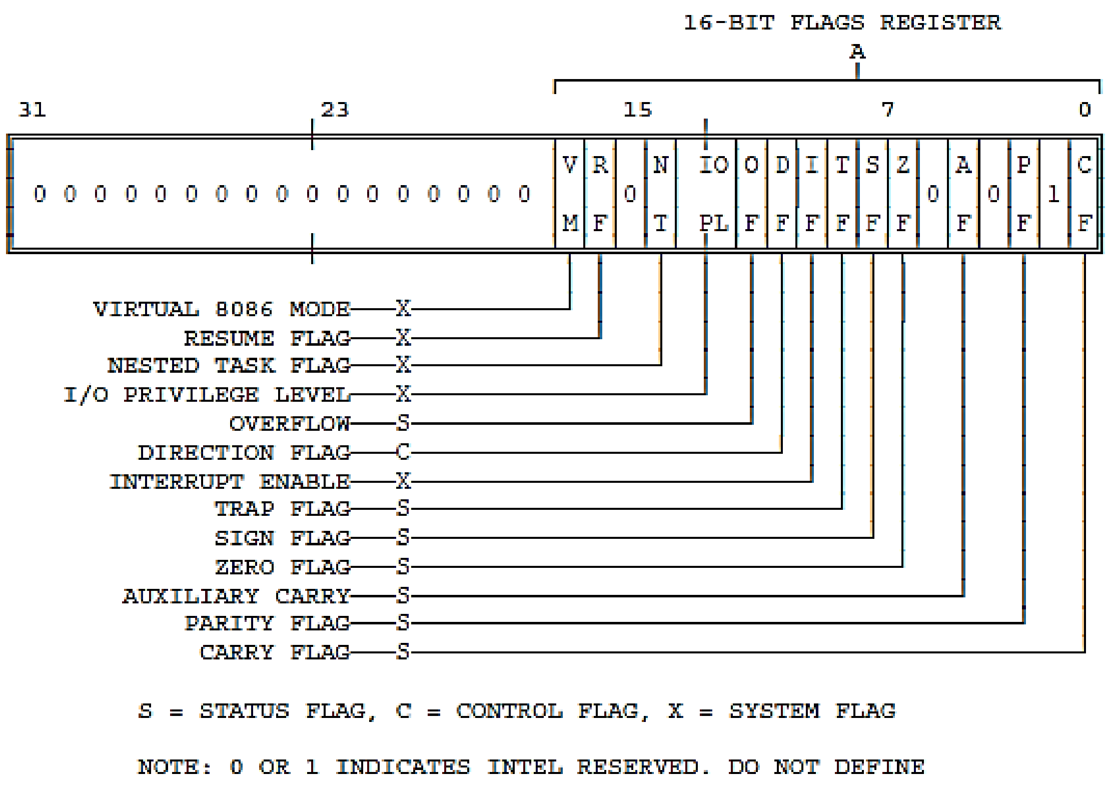
The GDTR and IDTR are called as system address and LDTR and TR are called as system segment registers.

Debug and Test Registers

8 debug registers for hardware debugging. Hold breakpoint status and breakpoint control information. Two more test registers are provided by 80386 for page caching, namely test control and test status registers.

Flag Register

Figure 2-8. EFLAGS Register



C-Carry Flag

This flag is set when there is a carry out of MSB in case of addition or a borrow in case of subtraction. For example, when two numbers are added, a carry may be generated out of the most significant bit position. The carry flag, in this case, will be set to '1'. In case, no carry is generated, it will be '0'. Some other instructions also affect or use this flag and will be discussed later in this text.

P-Parity Flag

This flag is set to 1 if the lower byte of the result contains **even** number of 1s.

AC-Auxiliary Carry Flag

This is set if there is a carry from the **lowest nibble**, i.e. bit three, during addition or borrow for the lowest nibble, i.e. bit three, during subtraction. Used in **BCD arithmetic**.

Z-Zero Flag

This flag is set if the result of the computation or comparison performed by the previous instruction/instructions is zero.

S-Sign Flag

This flag is set when the result of any computation is negative. For signed computations, the **sign flag equals the MSB** of the result.

T-Trap

Flag If this flag is set, the processor enters the single step execution mode. In other words, a trap interrupt is generated after execution of each instruction. The processor executes the current instruction and the control is transferred to the Trap interrupt service routine.

IF (Interrupt-Enable Flag, bit 9)

Setting IF allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no effect on either exceptions or non-maskable external interrupts.

DF-Direction Flag

This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the **lowest address to the highest address**, i.e. autoincrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e. autodecrementing mode.

OF-Overflow Flag

This flag is set if an overflow occurs, i.e. if the result of a signed operation is large enough to be accommodated in a destination register. For example, in case of the addition of two signed numbers, if the result overflows into the sign bit, i.e. the result is of more than 7-bits in size in case of 8-bit signed operations and more than 15-bits in size in case of 16-bit signed operations, then the overflow flag will be set.

IOPL (IO Privilege level Flag, bit 12-13):

The two bits in the IOPL are used by the processor and the operating system to determine your application's access to I/O facilities. It holds a privilege level, from 0 to 3, at which the current code is running to execute any I/O-related instruction.

NT (Nested Task, bit 14)

Sets when one task invokes another task. The processor uses the nested task flag to control chaining of interrupted and called tasks. NT influences the operation of the IRET instruction. Refer to Chapter 7 and Chapter 9 for more information on nested tasks.

RF (Resume Flag, bit 16)

The RF flag temporarily disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception. Refer to Chapter 12 for details.

TF (Trap Flag, bit 8)

Setting TF puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an exception after each instruction, allowing a program to be inspected as it executes each instruction. Single-stepping is just one of several debugging features of the 80386. Refer to Chapter 12 for additional information.

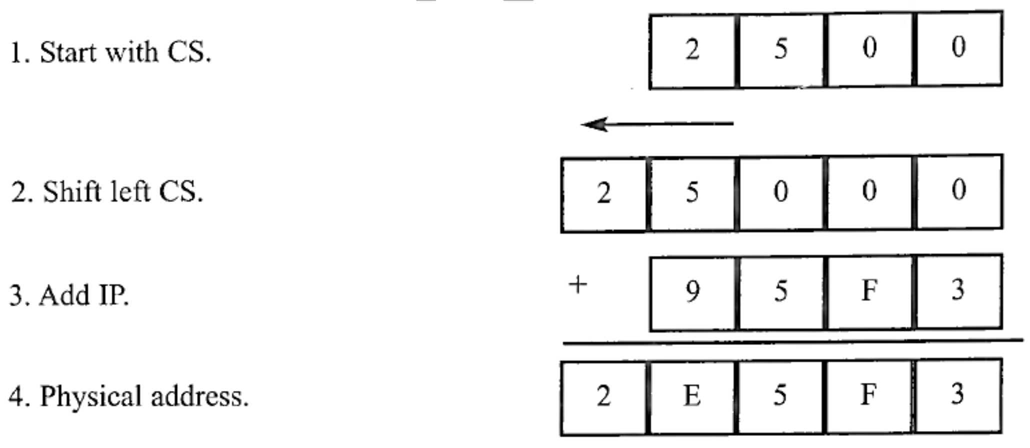
VM (Virtual 8086 Mode, bit 17)

When set, the VM flag indicates that the task is executing an 8086 program.

Operating Modes of 80386

Real Mode

- Also known as Real addressing mode.
- In real mode 80386 operates as 16-bit 8086 processor.
- Address bus size reduces to 20-bits. So can address 1MB of RAM only.
- Since processor internal registers are only 16-bits wide, generation of 20-bit address using 16-bit segment selector registers and corresponding offset takes place as follows.



- The same address generated is called physical address of the RAM
- Whereas, the logical address in real mode can be expressed as CS:IP or DS:SI, etc.

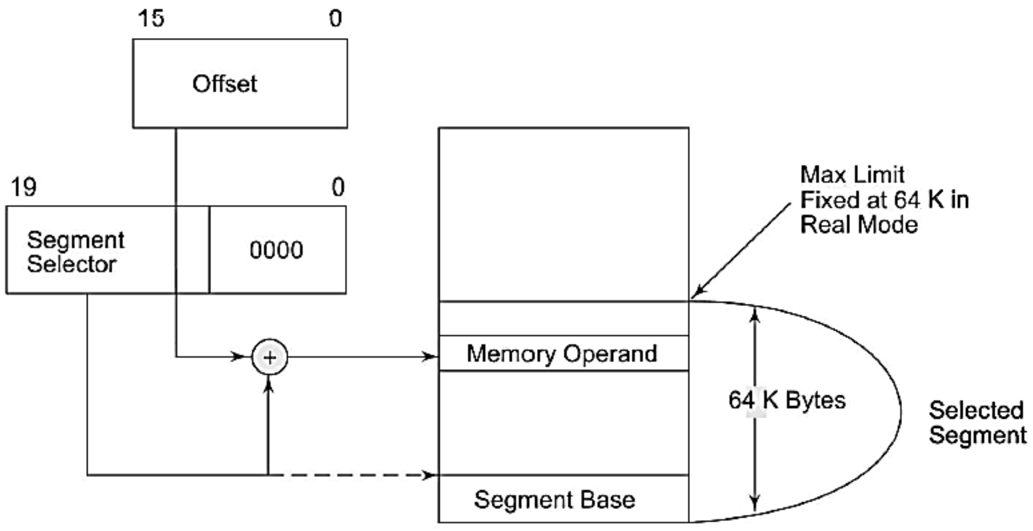


Fig.10.4 *Physical Address Formation in Real Mode of 80386 (Intel Corp.)*

Protected (Virtual) Mode

- Also known as Protected Virtual Addressing Mode (PVAM).
- In protected mode 80386, works as full-fledged 32-bit processor and provides functions like protection of memory segments by preventing unauthorised access, from other application programs, enforced by privilege mechanisms.

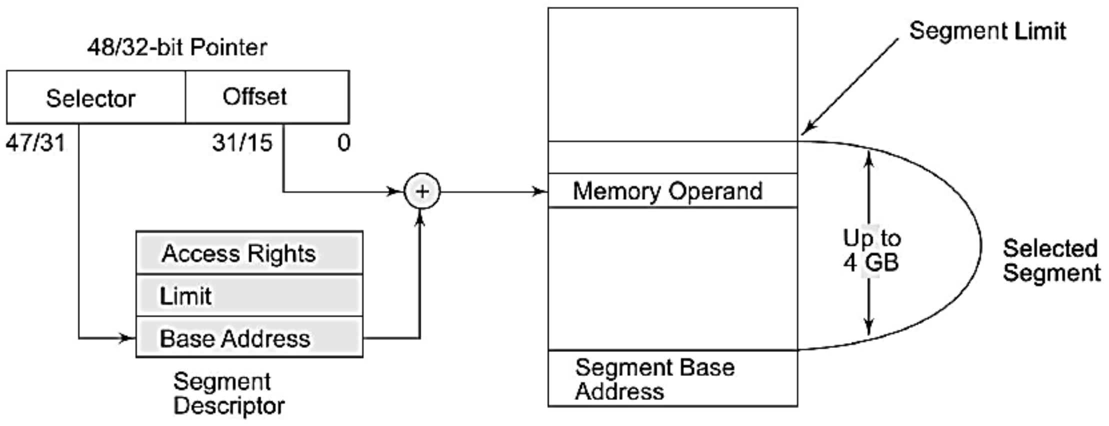


Fig. 10.5(a) *Protected Mode Addressing without Paging Unit (Intel Crop.)*

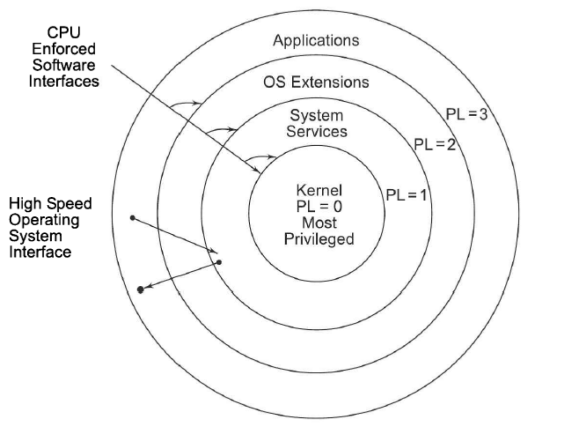

CPU Enforced Software Interfaces
High Speed Operating System Interface
Applications
OS Extensions
System Services
Kernel PL = 0 Most Privileged
PL = 3
PL = 2
PL = 1
Note: PL Becomes Numerically Lower as Privilege Level Increases

Fig. 9.15 ***Four Level Privilege Mechanism***

Virtual 8086 Mode

- Will be discussed in Unit-3

Memory Organization and Segmentation in 80386

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**).

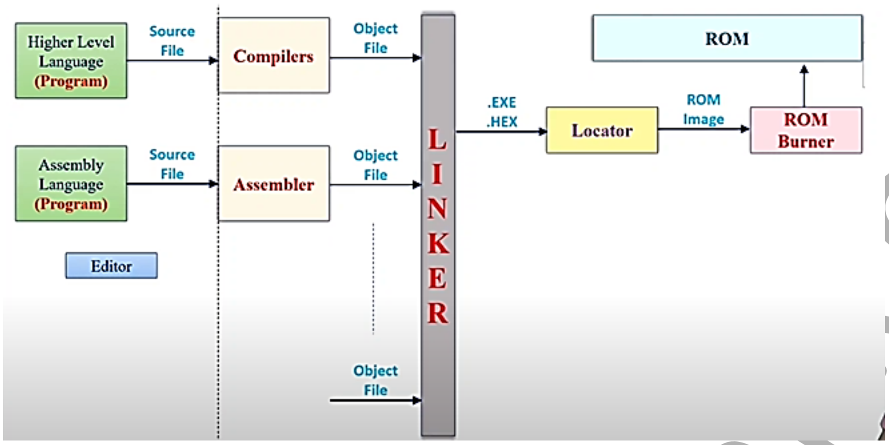
80386 programs, however, are independent of the physical address space. This means that programs can be written without knowledge of how much physical memory is available and without knowledge of exactly where in physical memory the instructions and data are located.

Software designers are given the freedom to choose a model for each task.

Memory Models

Flat Model

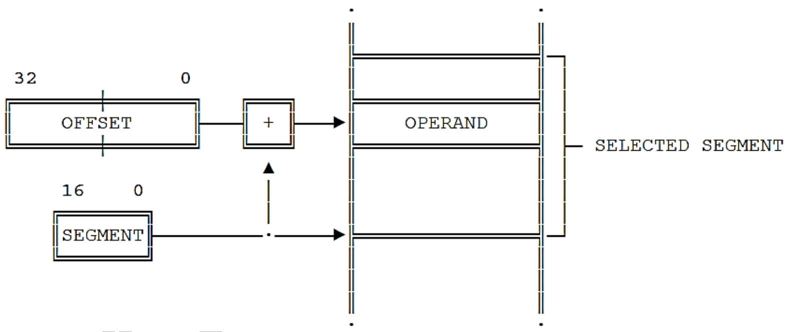
- In a "flat" model of memory organization, the applications programmer sees a single array of up to 2^{32} bytes (4 gigabytes).
- While the physical memory can contain up to 4 gigabytes, it is usually much smaller.
- The processor maps the 4-gigabyte flat space onto the physical address space by the address translation mechanisms described in Chapter 5.
- 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).



Segmented Model

- The processor maps the 64-terabyte logical (virtual) address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms described in Chapter 5.
- 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 (see Figure 2-1):
- A **segment selector**, which is a 16-bit field that identifies a segment.
- An **offset**, which is a 32-bit ordinal that addresses to the byte level within a segment.

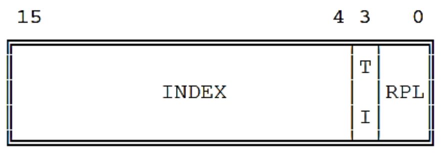
Figure 2-1. Two-Component Pointer



Selector, GDT, LDT

- In protected mode, the processor loads the selector in segment register to access descriptor look-up table available in memory.
- Each descriptor in lookup table is 8-bytes sized.
- The 8192 descriptors, within the table will contain the base address, segment size and privilege levels of the code & data segments stored in some other part of the memory.
- Hence, segment registers are called segment selector registers.
- GDT is used by system.
- LDTs are specific by the individual tasks.
- Operating System creates the descriptor tables.

Figure 5-6. Format of a Selector



TI - TABLE INDICATOR
RPL - REQUESTOR'S PRIVILEGE LEVEL

Index: Selects one of 8192 descriptors in a descriptor table. The processor simply multiplies this index value by 8 (the length of a descriptor), and adds the result to the base address of the descriptor table in order to access the appropriate segment descriptor in the table.

Table Indicator: Specifies to which descriptor table the selector refers. A zero indicates the GDT; a one indicates the current LDT.

Requested Privilege Level: Used by the protection mechanism. (Refer to Chapter 6.)

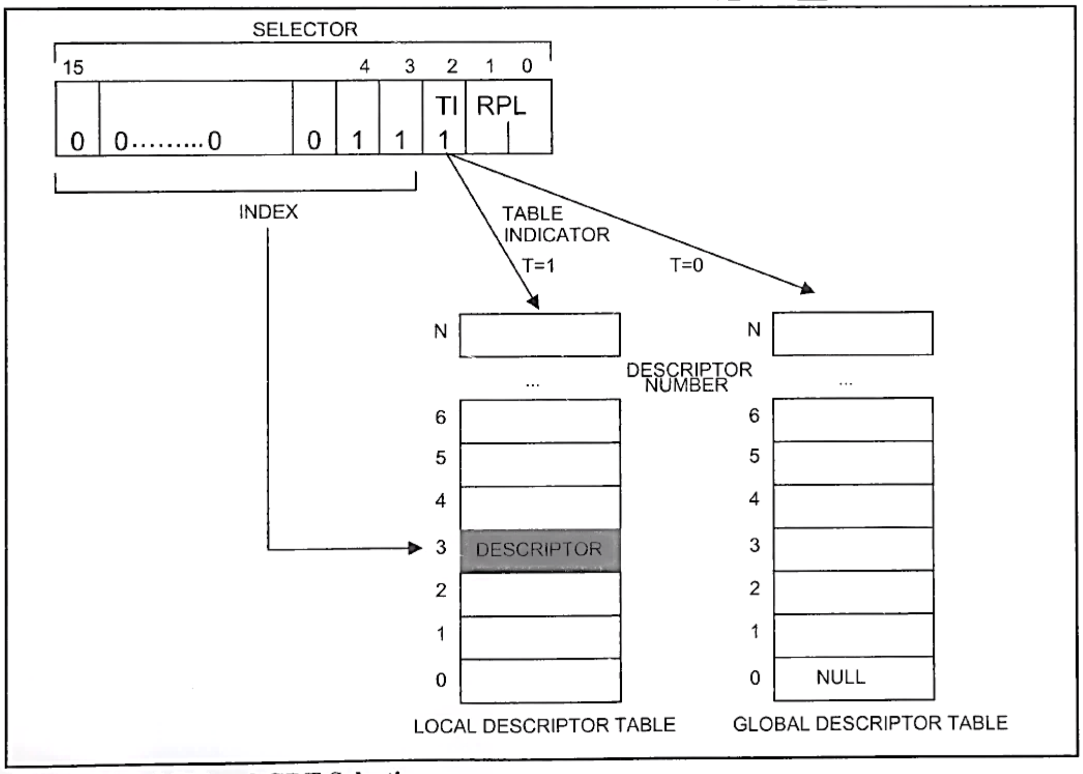
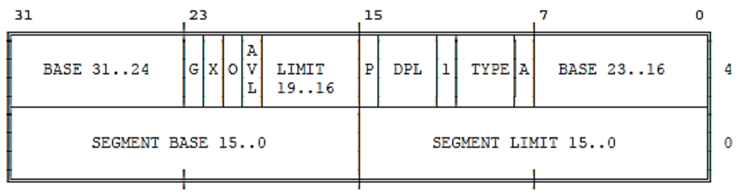


Figure 21-9. LDT and GDT Selection

- The processor locates the GDT and the current LDT in memory by means of the GDTR and LDTR registers.
- These registers store the base addresses of the tables in the linear address space and store the segment limits.
- The instructions LGDT and SGDT give access to the GDTR; the instructions LLDT and SLDT give access to the LDTR.

Descriptor

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS

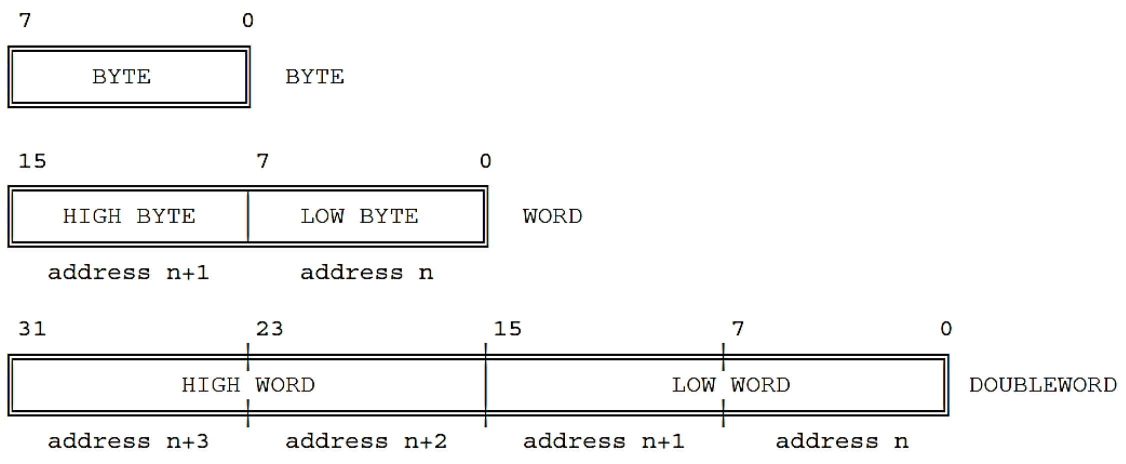


DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS

- **BASE:** Defines the location of the segment within the 4-gigabyte linear address space.
- The processor concatenates the three fragments of the base address to form a single **32-bit** value.
- **LIMIT:** Defines the size of the segment. When the processor concatenates the two parts of the limit field, a 20-bit value results. Hence, can address 1MB memory with **20-bits**.
- But the processor interprets the limit field in one of two ways, depending on the setting of the granularity bit;
- **Granularity bit**: specifies the units with which the LIMIT field is interpreted. When the bit is clear, the limit is interpreted in units of one byte; Hence, addressable memory becomes 1MB×1Byte=1MB
- When set, the limit is interpreted in units of 4 Kilobytes. So addressable memory becomes, 4KB×1MB =4GB
- The limit is shifted left by 12 bits when loaded, and low-order one-bits are inserted.
- **TYPE:** Distinguishes between various kinds of descriptors depending on types of segments (Read/Write, Read Only etc.).
- **DPL (Descriptor Privilege Level):** Used by the protection mechanism (refer to Chapter 6**).**
- **P (Segment-Present Bit):** If this bit is zero, the descriptor is not valid for use in address transformation; The processor will signal an exception when a selector for the descriptor is loaded into a segment register.
- Operating systems that implement segment-based virtual memory clear the present bit in either of these cases;
- When the linear space spanned by the segment is not mapped by the paging mechanism.
- When the segment is not present in memory.
- **Access bit:** The processor sets this bit when the segment is accessed; i.e., a selector for the descriptor is loaded into a segment register.
- Operating systems that implement virtual memory at the segment level may, by periodically testing and clearing this bit, monitor frequency of segment usage.
- **AVL bit:** Determines whether segment is available for OS or user.

Data Types in 80386

Figure 2-2. Fundamental Data Types

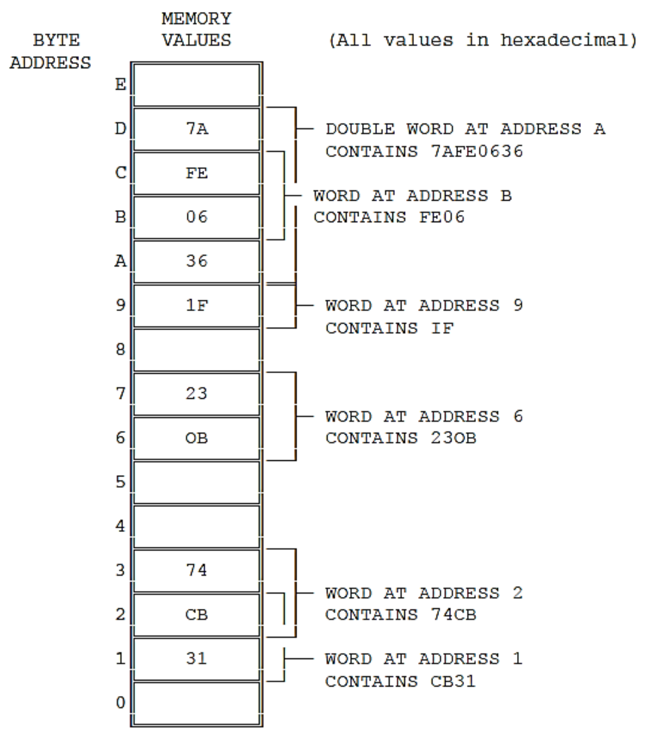


1. A **byte** is eight contiguous bits starting at any logical address. The bits are numbered 0 through 7; bit zero is the least significant bit.

2. A **word** is two contiguous bytes starting at any byte address.

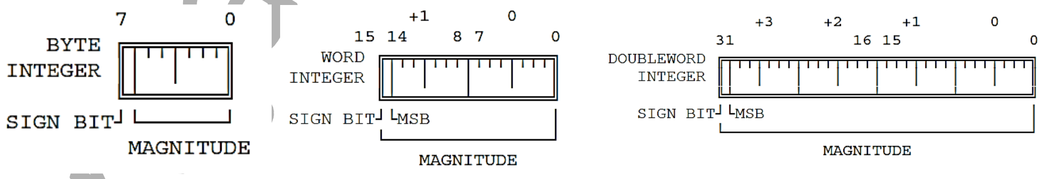
    - A word thus contains 16 bits. The bits of a word are numbered from 0 through 15;
    - Bit 0 is the least significant bit.
    - The byte containing bit 0 of the word is called the **low byte**;
    - the byte containing bit 15 is called the **high byte**.
    - Each **byte within a word has its own 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.
3. 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.
    - Figure 2-3 illustrates the arrangement of bytes within words and doublewords.

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


BYTE ADDRESS
MEMORY VALUES
(All values in hexadecimal)
E
D 7A
C FE
B 06
A 36
9 1F
8
7 23
6 0B
5
4
3 74
2 CB
1 31
0
DOUBLE WORD AT ADDRESS A CONTAINS 7AFE0636
WORD AT ADDRESS B CONTAINS FE06
WORD AT ADDRESS 9 CONTAINS IF
WORD AT ADDRESS 6 CONTAINS 230B
WORD AT ADDRESS 2 CONTAINS 74CB
WORD AT ADDRESS 1 CONTAINS CB31

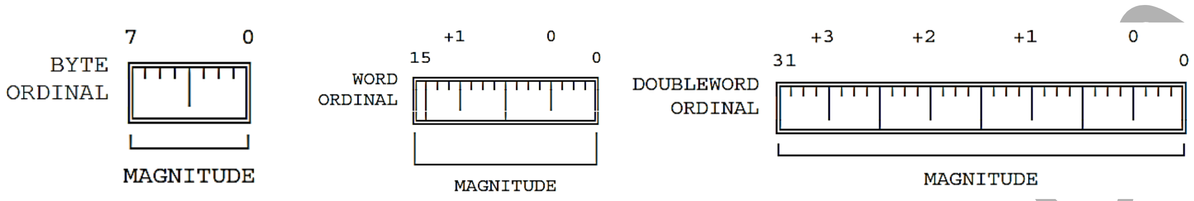
Processor also supports additional interpretations of these operands, given below.

4. **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.

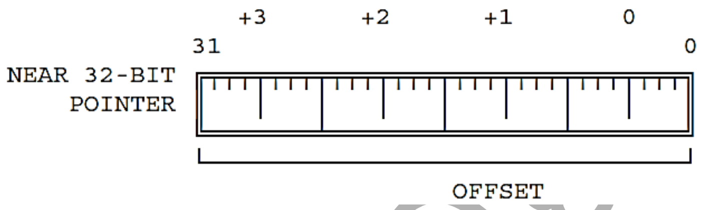


1. **Ordinal:**
    - An unsigned binary numeric value contained in a 32-bit doubleword, 16-bit word, or 8-bit byte.
    - All bits are considered in determining magnitude of the number.
    - The value range of an 8-bit ordinal number is 0-255;
    - 16 bits can represent values from 0 through 65,535;

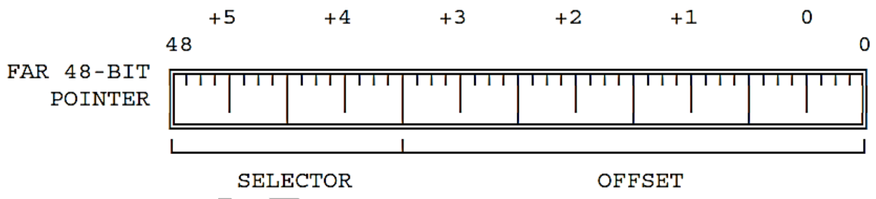
- 32 bits can represent values from 0 through 2^{32}-1.



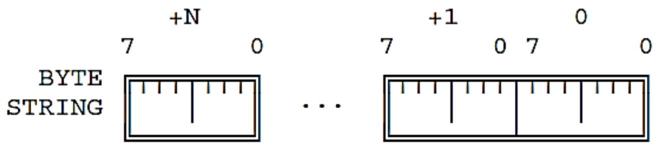
2. **Near Pointer:**
   - A 32-bit logical address.
   - A near pointer is an offset within a segment.
   - Near pointers are used in either a flat or a segmented model of memory organization.



3. **Far Pointer:**
   - A 48-bit logical address of two components:
   - a 16-bit segment selector component and a 32-bit offset component.
   - Far pointers are used by applications programmers only when systems designers choose a segmented memory organization.



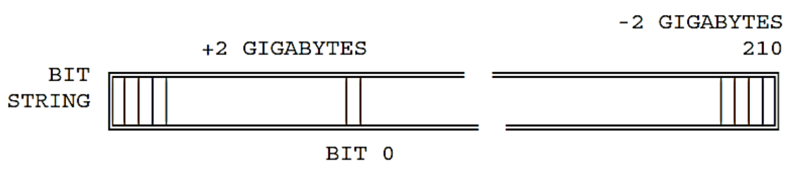
4. **String:**
   - A contiguous sequence of bytes, words, or doublewords.
   - A string may contain from zero bytes to 2^{32}-1 bytes (4 gigabytes).



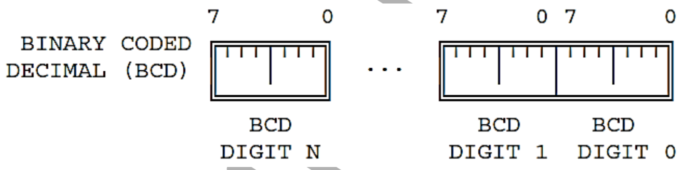
5. **Bit field:**
   - A contiguous sequence of bits.

- A bit field may begin at any bit position of any byte and may contain up to 32 bits.

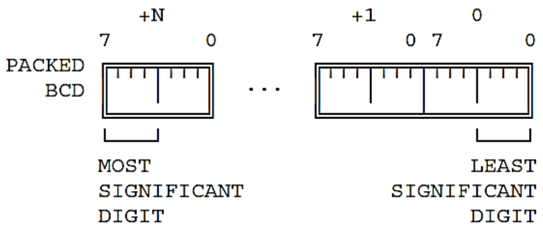
6. **Bit string:**
    - A contiguous sequence of bits.
    - A bit string may begin at any bit position of any byte and may contain up to 2^{32}-1 bits.



7. **Unpacked BCD:**
    - A byte (unpacked) representation of a decimal 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 determined from the low-order half-byte;
    - hexadecimal values 0-9 are valid and are interpreted as decimal numbers.
    - The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction.



8. **Packed BCD:**
    - A byte (packed) representation of two decimal digits, each in the range 0 through 9.
    - One digit is stored in each half-byte.
    - The digit in the high-order half-byte is the most significant.
    - Values 0-9 are valid in each half-byte.
    - The range of a packed decimal byte is 0-99.



Instruction Format of 80386

1. **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:
    a. **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.
    b. **Address size** — switches between 32-bit and 16-bit address generation.
    c. **Operand size** — switches between 32-bit and 16-bit operands.
    d. **Repeat** — used with a string instruction to cause the instruction to act on each element of the string. (REP MOVSB)
2. **Opcode** — specifies the operation performed by the instruction. Some operations have several different opcodes, each specifying a different variant of the operation.
3. **Register specifier** — an instruction may specify one or two register operands.
4. **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.
5. **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.
6. **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.
7. **Immediate operand** — when present, directly provides the value of an operand of the instruction. Immediate operands may be 8, 16, or 32 bits wide.

Figure 17-1. 80386 Instruction Format

INSTRUCTION PREFIX	ADDRESS-SIZE PREFIX	OPERAND-SIZE PREFIX	SEGMENT OVERRIDE
0 OR 1	0 OR 1	0 OR 1	0 OR 1
NUMBER OF BYTES			

OPCODE	MODR/M	SIB	DISPLACEMENT	IMMEDIATE
1 OR 2	0 OR 1	0 OR 1	0,1,2 OR 4	0,1,2 OR 4
NUMBER OF BYTES				

Addressing Modes

- Register & Immediate addressing modes deals with the data available within the processor.
- Rest of the addressing mode are used for data available external to the processor.
- By default, the external data contained in DS & ES segments of the memory are used, unless specified by segment overrides.
- Similarly, use of SP will operate on stack segment in the memory.

Register

- Involves registers as source and destination.
- No memory involvement so faster data transfer.
- Source and destination registers should match in size.
- MOV EBX, EDX ;copies content of EDX to EDX
- MOV ES, AX ;copies content of AX to ES segment register.
- ADD AL,BH ; Adds content of BH to AL and stores the result in AL

Immediate

- Source operand is constant.
- MOV EAX, 76583456H
- Cannot use to load segment registers and flag register.
- Data should first move into general purpose register then loaded into segment register as shown below:
- MOV AX, 2550H
- MOV DS, AX
- MOV DS, 2550H ;illegal!

Direct

- Directly addresses data in memory.
- Data comes immediately into destination register.
- MOV DL, [2400H]; copies content of DS:2400H (DS base address : offset) memory location to DL
- Brackets [] plays critical role otherwise the source operand is treated as data and not memory location.

Register Indirect

- Similar to direct addressing mode, but memory location is held in a register.
- For 80386, any general purpose & index registers can act as pointer/offset register.
- MOV AL, [BX] ;copies content from memory location pointed by DS:BX to AL
- MOV CL, [SI] ;copies content from memory location pointed by DS:SI to CL
- MOV [DI], AH ; copies content from AH register to the memory location pointed by DS:DI

Based Relative

- Similar to register indirect addressing mode, but named so because it typically used BX, BP registers to hold offset address.
- For 80386, any general purpose & index/pointer registers can act as pointer/offset register.
- MOV CX, [BX]+10 ;copies 2-bytes from memory locations DS:BX+10 and DS:BX+10+1
- If **BP** is used, it works with Stack segment.
- BX+10 is also referred as **effective address** in Intel's literature.
- 10 is called displacement.
- A displacement is a signed integer of 8, 16 or 32 bits.
- Effective address is also called as offset.
- If effective address exceeds 32-bits, it is truncated to get 32-bit.

Indexed Relative

- Similar to based relative addressing mode, but named so because only SI & DI index registers hold the offset address.
- MOV DX, [SI]+5 ; copies 2-bytes from memory locations DS:SI+5 and DS:SI+5+1

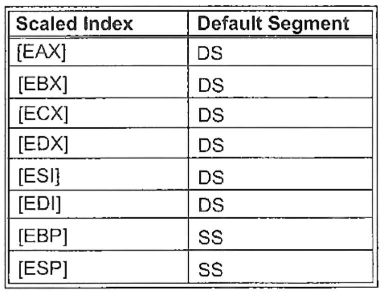
Based Indexed Relative

- Combination of based relative and indexed relative modes.
- Displacement is optional.
- MOV CL, [BX][SI]+8 ;copies a byte from memory location pointed by DS:BX+SI+8
- MOV CL, [BP][SI]+8 ;copies a byte from stack memory location pointed by SS:BP+SI+8
- Coding variation for above instruction can be MOV CL, [BP+SI+8]

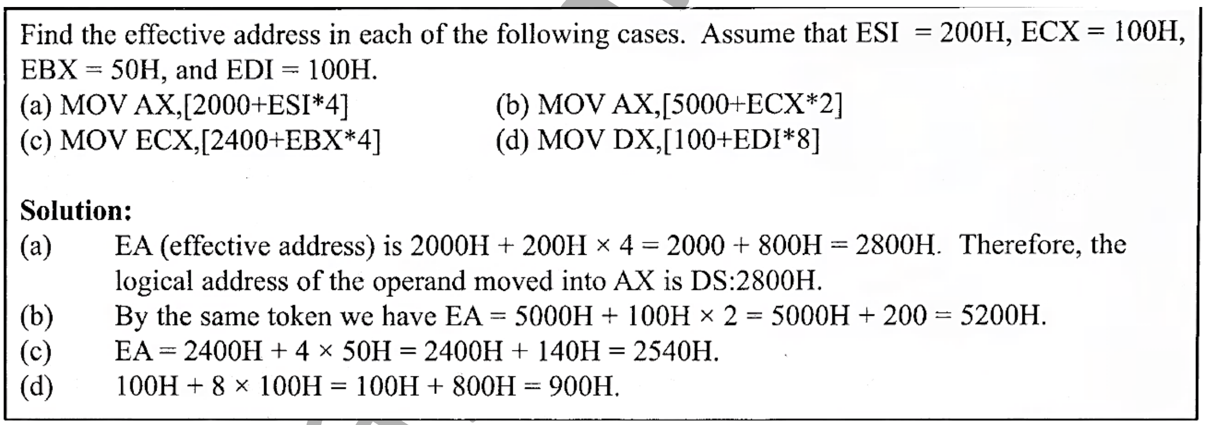
Scaled Indexed (Special 80386 mode)

- Used for accessing multidimensional arrays.
- Any of the 32-bit registers can be used as pointer/offset register.
- These registers can be multiplied/scaled by factor of 1,2,4 & 8.
- Each scaling factor corresponds to number of bytes in an operands.
- 1=byte; 2=word; 4=doubleword; 8=quadword operands.
- 16-bit registers are not allowed. MOV AL, [ESI+BX*4] is illegal.

Table 21-2: 386 Scaled Index Addressing Mode



Scaled Index	Default Segment
[EAX]	DS
[EBX]	DS
[ECX]	DS
[EDX]	DS
[ESI]	DS
[EDI]	DS
[EBP]	SS
[ESP]	SS



Find the effective address in each of the following cases. Assume that ESI = 200H, ECX = 100H, EBX = 50H, and EDI = 100H.

(a) MOV AX,[2000+ESI*4] (b) MOV AX,[5000+ECX*2]

(c) MOV ECX,[2400+EBX*4] (d) MOV DX,[100+EDI*8]

Solution:

(a) EA (effective address) is 2000H + 200H × 4 = 2000 + 800H = 2800H. Therefore, the logical address of the operand moved into AX is DS:2800H.

(b) By the same token we have EA = 5000H + 100H × 2 = 5000H + 200 = 5200H.

(c) EA = 2400H + 4 × 50H = 2400H + 140H = 2540H.

(d) 100H + 8 × 100H = 100H + 800H = 900H.

Table 21-1: Addressing Modes for the 80386

Addressing Mode	Operand	Default Segment
Register	Register	None
Immediate	Data	None
Direct	[OFFSET]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
	[EAX]	DS
	[EBX]	DS
	[ECX]	DS
	[EDX]	DS
	[ESI]	DS
	[EDI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
	[EAX]+disp	DS
	[EBX]+disp	DS
	[ECX]+disp	DS
	[EDX]+disp	DS
	[EBP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	DS
	[EDI]+disp	DS
	[ESI]+disp	DS
Based indexed relative	[R1][R2]+disp R1 and R2 are any of the above	If BP is used, segment is SS; otherwise, DS is the segment

Note: In based indexed relative addressing, disp is optional.

```
MOV   BX,WORD PTR [ EAX]    ;move into BX word pointed to by EAX
MOV   BX,WORD PTR [ AX]     ;invalid AX can't be used as pointer
MOV   EAX,DWORD PTR [ ECX]  ;move into EAX DWORD pointed to by ECX
MOV   AL,BYTE PTR [ EDX]    ;move into AL BYTE pointed to by EDX
MOV   EBX,WORD PTR [ CX]    ;invalid CX can't be used as pointer
MOV   EAX,DWORD PTR [ EDI]  ;move into EAX DWORD pointed to by EDI
```

Instruction Set of 80386

Applicable for 32-bit addressing in protected mode. however, all instructions discussed are also available when 16-bit addressing is in effect in protected mode, real mode, or virtual 8086 mode. For any differences of operation that exist in the various modes, refer to Chapter 13, Chapter 14, or Chapter 15.

Data Movement Instructions

- General-purpose data movement instructions.
- Stack manipulation instructions.
- Type-conversion instructions.

General-purpose data movement instructions

MOV

- Format- *MOV destination, source*
- Cannot move from memory to memory or from segment register to segment register.

MOVSx - MOVSB, MOVSW & MOVSD

- Format- *MOVSx*
- MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS
- instructions.
- Can move strings from memory to memory.
- MOVS copies the byte or word at [(E)SI] to the byte or word at ES:[(E)DI].
- After the data is moved, both (E)SI and (E)DI are advanced automatically.
- If the direction flag is 0, the registers are incremented; if the direction flag is 1, the registers are decremented.
- The registers are incremented or decremented by 1 if a byte was moved, 2 if a word was moved, or 4 if a doubleword was moved.
- When used with REP prefix – *REP MOVSB*, (E)CX register can be used as counter register.

XCHG

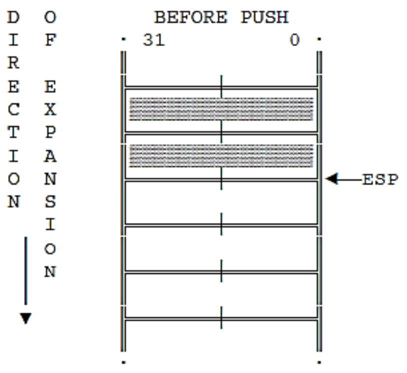
- Format- *XCHG destination, source*
- Swaps the contents of two operands.
- The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand.
- The XCHG instruction can swap two-byte operands, two-word operands, or two doubleword operands.
- This instruction 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.
- When used with a memory operand, XCHG automatically activates the LOCK signal. (Refer to Chapter 11 for more information on the bus lock.)

Stack Manipulation Instructions

PUSH

- Format- *PUSH source*
- It operates on memory operands, immediate operands, and register operands (including segment registers).
- PUSH (Push) **decrements** the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP (see Figure 3-1).

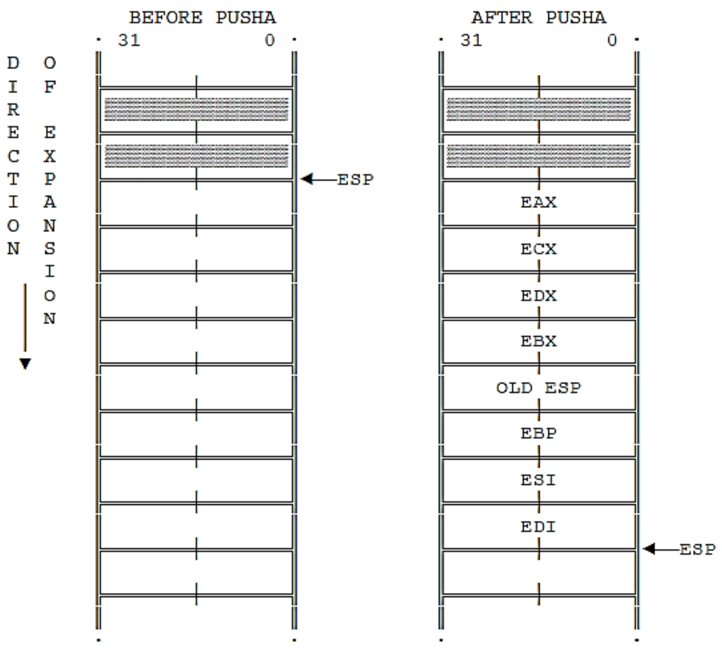
Figure 3-1. PUSH



PUSHA

- Format- *PUSHA*
- (Push All Registers) saves the contents of the eight general registers on the stack (see Figure 3-2).
- The processor pushes the general registers on the stack in the following order: EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI. PUSHA is complemented by the POPA instruction.

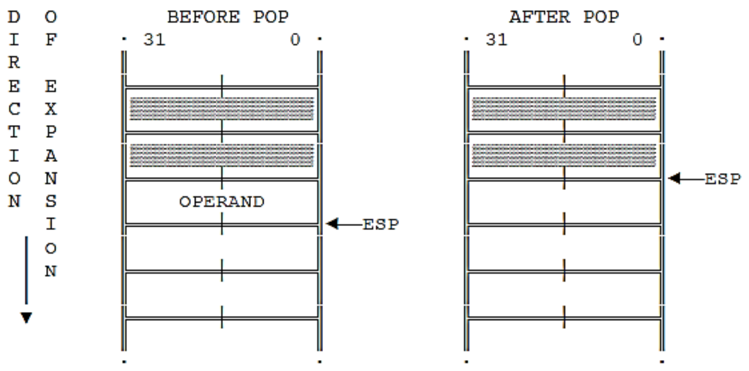
Figure 3-2. PUSHA



POP

- Format- *POP destination*
- (Pop) transfers the word or doubleword at the current top of stack (indicated by ESP) to the destination operand.
- Moves information from the stack to a general register, or to memory.
- Then **increments** ESP to point to the new top of stack.

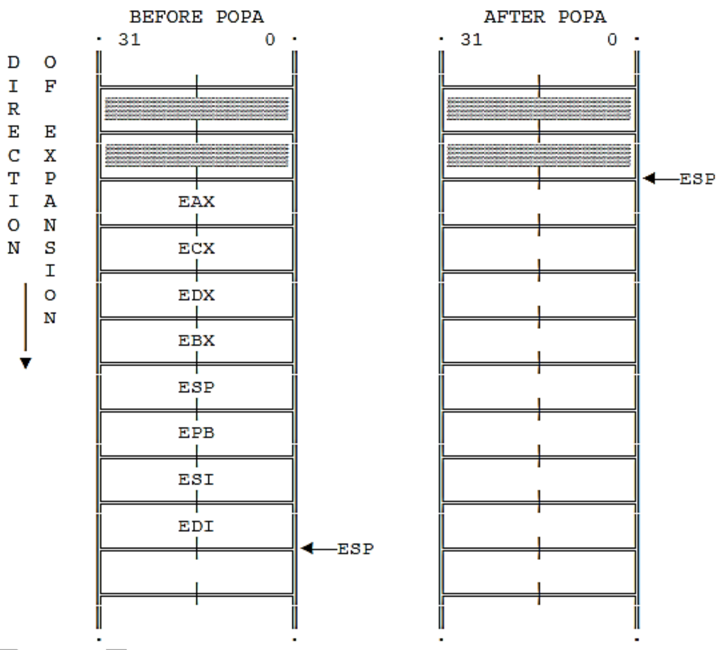
Figure 3-3. POP



POPA

- Format- *POPA*
- (Pop All Registers) restores the registers saved on the stack by PUSHA.
- Except that it ignores the saved value of ESP.

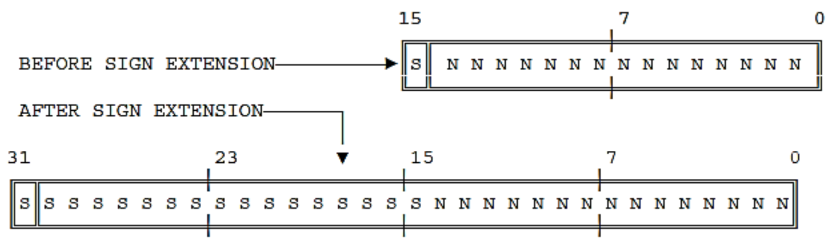
Figure 3-4. POPA



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, illustrated by Figure 3-5, is called sign extension.

Figure 3-5. Sign Extension

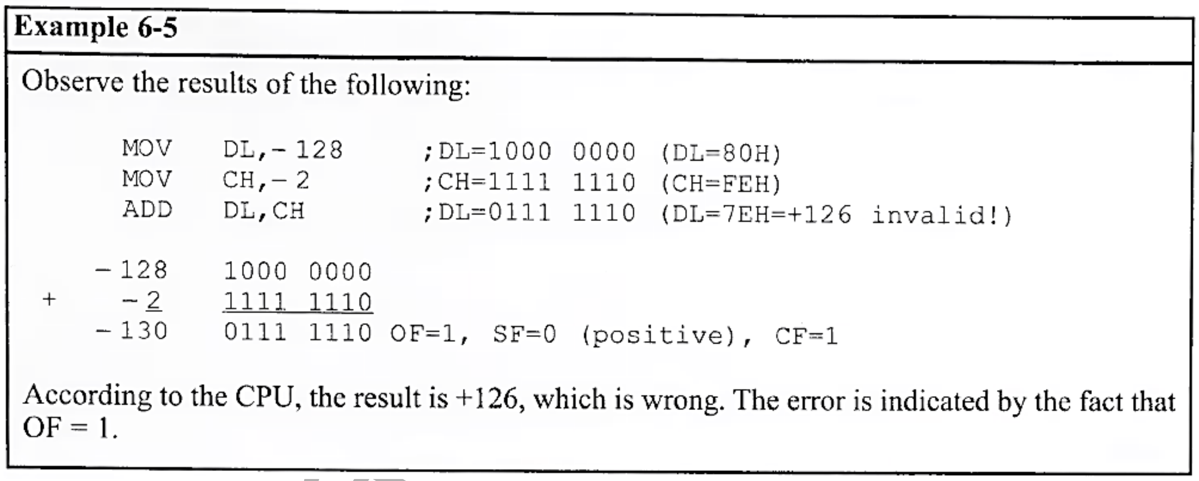

15
7
0
BEFORE SIGN EXTENSION
S
N N N N N N N N N N N N N N N
AFTER SIGN EXTENSION
31
23
15
7
0
S
S S S S S S S S S S S S S S S S N N N N N N N N N N N N N N N N

- **There are two classes of type conversion instructions:**
  The forms **CWD, CDQ, CBW, and CWDE** which operate only on data in the EAX register.
  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.

CBW

- Format- *CBW*
- (Convert Byte to Word) extends the sign of the byte in register AL throughout AX.

Overflows causes sign bit of the result to manipulate, resulting in erroneous results. It usually happens when register size used to store the result is insufficient. Hence, while handling signed numbers if OF flag is set. The operands sizes are extended and arithmetic operation is repeated to get correct result.


Example 6-5

Observe the results of the following:

```
      MOV    DL,-128       ;DL=1000 0000 (DL=80H)
      MOV    CH,-2         ;CH=1111 1110 (CH=FEH)
      ADD    DL,CH         ;DL=0111 1110 (DL=7EH=+126 invalid!)

     -128    1000 0000
  +    -2    1111 1110
     -130    0111 1110 OF=1, SF=0 (positive), CF=1
```

According to the CPU, the result is +126, which is wrong. The error is indicated by the fact that OF = 1.

Crude Assembly Program Implementation

MOV AL,DL ;load first operand in AL

CBW ;sign extend byte to word within AX. AX= (1111 1111 1000 0000)

MOV DX, AX

MOV AL, CH ;load second operand in AL

CBW ;sign extend byte to word within AX= (1111 1111 1111 1110)

MOV CX, AX

ADD CX, DX

CX will contain following result – (**1**111 1111 0111 1110) - 2's complement form

Convert binary to decimal:

$\mathbf{-2^{15}} + 2^{14} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^{9} + 2^{8} + 2^{7} * 0 + 2^{6} + 2^{5} + 2^{4} + 2^{3} + 2^{2} + 2^{1} + 2^{0} *0 = \mathbf{-130}$

CWD & CDQ

- Format- *CWD*
- Format- *CDQ*
- 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. Uses AX:DX pair
- CDQ extends the sign of the doubleword in EAX throughout EDX. Uses EAX:EDX pair.
- CWD can be used to produce a double word 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.

CWDE

- Format- CWDE
- (Convert Word to Doubleword Extended) extends the sign of the word in register AX throughout EAX.

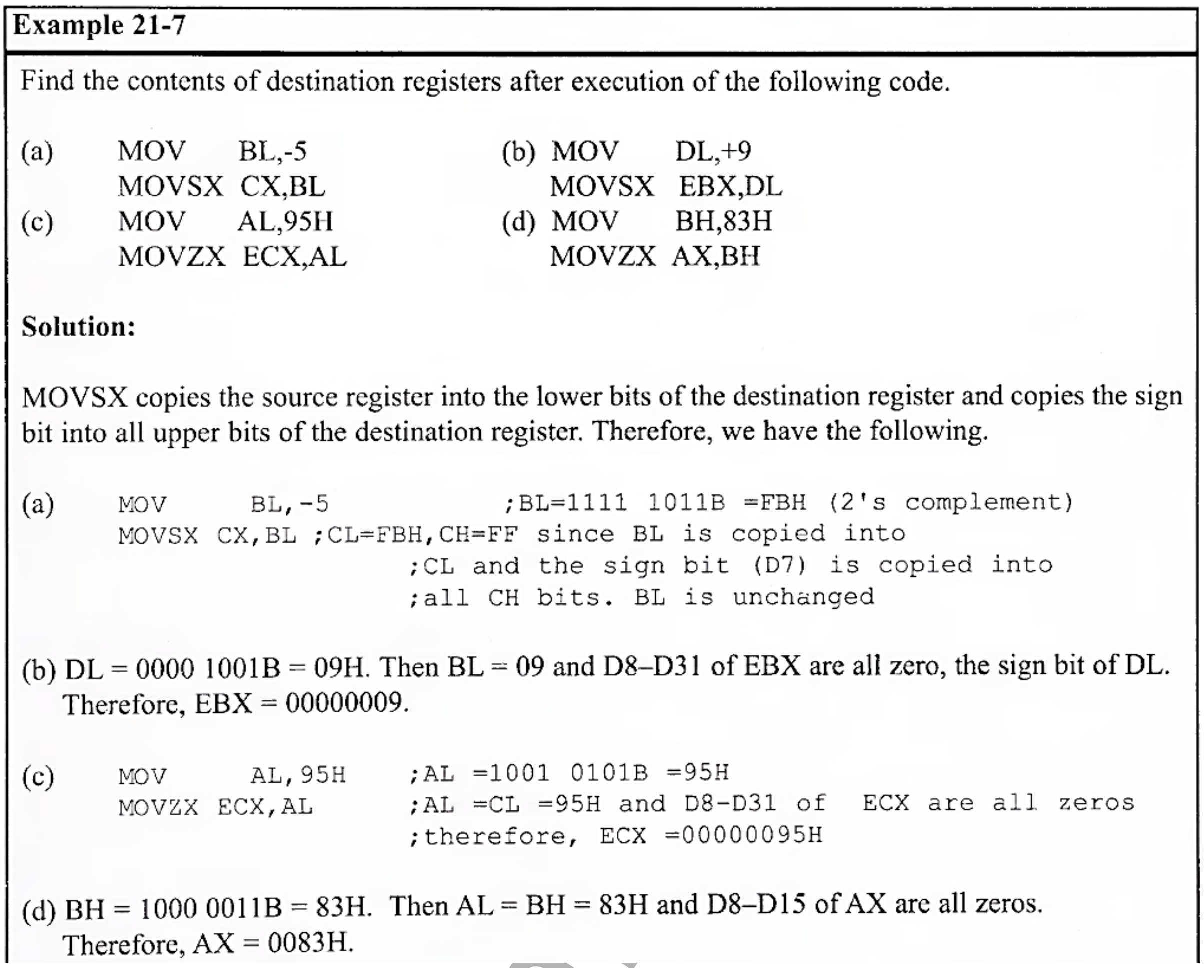
Above instructions work only when operand is in A-register. Instructions below provide same functionality as well as extended functionality by working with any register.

MOVSX (Special 80386 instruction)

- Format- *MOVSX destination, source*
- (Move with Sign Extension) sign-extends an 8-bit value to a 16-bit or 32-bit value. And 16-bit value to 32-bit value.
- Used for signed arithmetic operations.
- MOV BL,-5 ;BL=$(1111\ 1011)_2$ - 2's compliment form
- MOVSX CX, BL ;CX= FFFB =$(1111\ 1111\ 1111\ 1011)_2$

MOVZX (Special 80386 instruction)

- Format- *MOVZX destination, source*
- (Move with Zero Extension) extends an 8-bit value to a 16-bit or 32-bit value. And 16-bit value to 32-bit value.
- Inserts high-order zeros.
- Used for unsigned arithmetic operations.
- MOV AL, 95H
- MOVZX ECX,AL ; ECX=$(0000\ 0095)_{16}$



Example 21-7

Find the contents of destination registers after execution of the following code.

```
(a)   MOV     BL,-5          (b) MOV     DL,+9
      MOVSX  CX,BL               MOVSX  EBX,DL
(c)   MOV     AL,95H         (d) MOV     BH,83H
      MOVZX  ECX,AL              MOVZX  AX,BH
```

Solution:

MOVSX copies the source register into the lower bits of the destination register and copies the sign bit into all upper bits of the destination register. Therefore, we have the following.

```
(a)   MOV     BL,-5          ;BL=1111 1011B =FBH (2's complement)
      MOVSX CX,BL ;CL=FBH,CH=FF since BL is copied into
                       ;CL and the sign bit (D7) is copied into
                       ;all CH bits. BL is unchanged
```

(b) DL = 0000 1001B = 09H. Then BL = 09 and D8–D31 of EBX are all zero, the sign bit of DL. Therefore, EBX = 00000009.

```
(c)   MOV     AL,95H     ;AL =1001 0101B =95H
      MOVZX ECX,AL       ;AL =CL =95H and D8-D31 of  ECX are all zeros
                         ;therefore, ECX =00000095H
```

(d) BH = 1000 0011B = 83H. Then AL = BH = 83H and D8–D15 of AX are all zeros. Therefore, AX = 0083H.

Binary Arithmetic Instruction

- 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.
- 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.
- 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.
- ZF is relevant to both signed and unsigned integers; ZF is set when all bits of the result are zero.

OF is set in either of these cases:

- 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, INC, AAA, and DAA).

- 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, AAS, DAS, CMP, and NEG).

These status flags are tested by executing one of the two families of conditional instructions: Jcc (jump on condition cc) or SETcc (byte set on condition).

ADD

- Format- *ADD destination, source*
- (Add Integers) replaces the destination operand with the sum of the source and destination operands.
- Operands should match in size.
- Only one operand can be in the memory.
- Sets CF if result overflows.
- OF is ignored for unsigned addition. Hence programmer should be sure about the choice of result register.
- Use of signed number adds an advantage that OF can be monitored and result register size can be scaled up accordingly.

ADC

- Format- *ADC destination, source* ;performs destination + source + carry
- (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

- Format- *INC destination*
- Destination can be memory location or register.
- (Increment) adds one to the destination operand. INC does not affect CF.
- Value $(FFFF)_{16}$ is incremented to $(0000)_{16}$ without CF.
- Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.

SUB

- Format- *SUB destination, source ;*performs destination - source
- (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

- Format- *SBB destination, source ;*performs destination - source - carry
- (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.

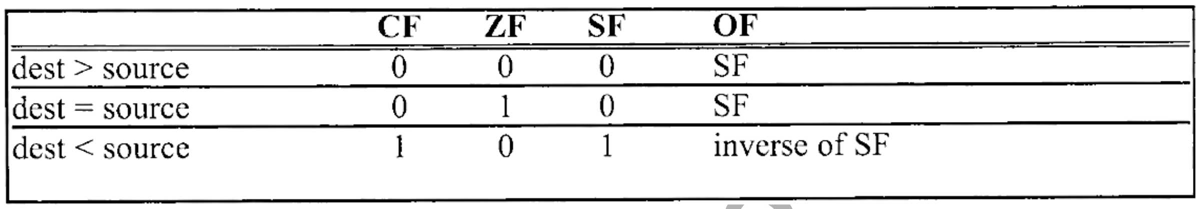
DEC

- Format- *DEC destination*

- (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.
- DEC with destination containing 0000H, results in FFFFH value.

CMP

- Format- *CMP destination, source ;*performs destination – source but does not affect destination or source operands.
- (Compare) subtracts the source operand from the destination operand.
- It does not alter the source and destination operands.
- It updates OF, SF, ZF, AF, PF, and CF. But ZF is mostly used in looping.
- A subsequent Jcc (Jump on condition) or SETcc instruction can test the appropriate flags.



	CF	ZF	SF	OF
dest > source	0	0	0	SF
dest = source	0	1	0	SF
dest < source	1	0	1	inverse of SF

NEG

- Format- *NEG destination*
- (Negate) subtracts a signed integer operand from zero.
- Basically performs 2’s complement of the operand.
- The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.

MUL

- Format- *MUL source*
- (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 AX – (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

- **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.
  Format- *IMUL source*

- **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.
  Format- *IMUL destination register, source register/memory/immediate operand*
- **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.
  Format- *IMUL destination register, source register/memory, immediate source operand*
  - 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:

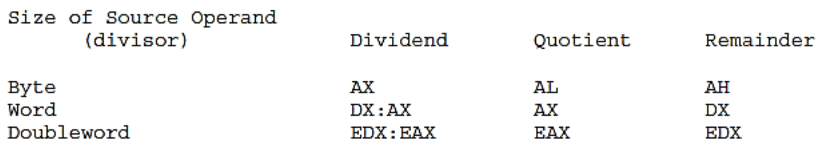
- The length of the product is calculated to twice the length of the operands.
- The CF and OF flags are set when significant bits are carried into the high-order half of the result. CF and OF are cleared when the high-order half of the result is the sign-extension of the low-order half.

However, forms 2 and 3 differ in foll. ways:

- 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. (For ways to test OF, refer to the INTO and PUSHF instructions.)
- 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.

DIV

- Format- *DIV source*
- (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.

Decimal Arithmetic Instruction

Decimal arithmetic is performed in combination with binary arithmetic. The decimal arithmetic instructions are used in one of the following ways:

- To adjust the results of a previous binary arithmetic operation to produce a valid packed or unpacked decimal result.
- To adjust the inputs to a subsequent binary arithmetic operation so that the operation will produce a valid packed or unpacked decimal result.

These instructions operate only on the AL or AH registers. Most utilize the AF flag.

Packed BCD Adjustment Instructions

DAA

- Format- *DAA*
- (Decimal Adjust after Addition) adjusts the result of adding two valid packed decimal operands in **AL**.
- It adds 6 to the lower 4 bits of AL, if it is greater than 9 or if AF = 1.
- Then it adds 6 to the upper 4 bits of AL, if it is greater than 9 or if CF = 1.

```
Example 1:
        MOV    AL,47H        ;AL=0100 0111
        ADD    AL,38H        ;AL=47H+38H=7FH.   invalid BCD
        DAA                  ;NOW AL=1000 0101 (85H IS VALID BCD)
```

DAS

-
- Format- *DAS*
- (Decimal Adjust after Subtraction) adjusts the result of subtracting two valid packed decimal operands in **AL**.
- If the lower 4 bits of AL represent a number greater than 9 or if AF = 1, then 6 is subtracted from the lower nibble.
- If the upper 4 bits of AL are now greater than 9 or if CF = 1, 6 is subtracted from the upper nibble.

```
Example:
        MOV    AL,45H        ;AL=0100 0101 BCD for 45
        SUB    AL,17H        ;AL=0010 1110 AN INVALID BCD
        DAS                  ;AL=0010 1000 BCD FOR 28(45-17=28)
```

Unpacked BCD Adjustment Instructions

AAA

- Format- *AAA*
- (ASCII Adjust after Addition) changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits.
- AAA must always follow the addition of two unpacked decimal (ASCII) operands in **AL.**
- Zeroes upper 4-bits of AL.
- It adds 6 to the lower 4 bits of AL, if it is greater than 9 or if AF = 1.
- If carry out from lower nibble, the CF & AF flags are set and AH is incremented if a carry is necessary. Hence AH should be cleared before addition.

```
Example 1:
        MOV     AL,31H       ;AL=31 THE ASCII CODE FOR 1
        ADD     AL,37H       ;ADD 37 (ASCII FOR 7) TO AL;   AL=68H
        AAA                  ;AL=08 AND CF=0
```

AAS

- Format- *AAS*
- (ASCII Adjust after Subtraction) changes the contents of register **AL** to a valid unpacked decimal number, and zeros the top 4 bits.
- AAS must always follow the subtraction of one unpacked decimal (ASCII) operands from another in AL.
- Zeroes upper 4-bits of AL.
- If the lower 4 bits of AL represent a number greater than 9 or if AF = 1, then 6 is subtracted from the lower nibble.
- If borrow in from lower nibble, the CF & AF flags are set and AH is decremented if a borrow is necessary. Hence AH should be cleared before subtraction.

```
Example:
        MOV    AL,32H        ;AL=32 ASCII FOR 2
        MOV    DH,37H        ;DH=37 ASCII FOR 7
        SUB    AL,DH         ;AL-DH=32-37=FBH WHICH IS -5 IN 2'S COMP
                             ;CF=1 INDICATING A BORROW
        AAS                  ;NOW AL=05 AND CF=1
```

AAM

- Format- *AAM*
- (ASCII Adjust after Multiplication) corrects the result of a multiplication of two valid unpacked BCD numbers.
- AAM must always follow the multiplication of two decimal (BCD) numbers to produce a valid decimal result.
- The high order digit is left in AH, the low order digit in AL.
- Example-
    - *MOV AL, 04 ;BCD digit not exceeding 9 and higher nibble is zeroed.*
    - *MOV BL, 09 ;BCD digit not exceeding 9 and higher nibble is zeroed.*
    - *MUL BL ;AX=$(36)_{10}$*
    - *AAM*
- AAM zeroes AH bits.
- Divides 36 by 10.

- Assigns quotient to AH=03
- Assigns remainder to AL=06

AAD

- Format- *AAD*
- (ASCII Adjust **before** Division) modifies the dividend in AH and AL to prepare for the division of two valid unpacked BCD operands.
- It ensures that quotient produced by the division will be a valid unpacked BCD number.
- AH should contain the high-order digit and AL the low-order digit.
- This instruction adjusts the value and places the result in AL. AH will contain zero.

```
Example:
        MOV    AX,3435H     ;AX=3435 THE ASCII FOR 45
        AND    AX,0F0FH     ;AX=0405H UNPACKED BCD FOR 45
        AAD                 ;AX=002DH HEX FOR 45
        MOV    DL,07        ;DL=07
        DIV    DL           ;2DH DIV BY 07 GIVES AL=06,AH=03
        OR     AX,3030H     ;AL=36=QUOTIENT AND AH=33=REMAINDER
```

- ANDing converts ASCII value into unpacked BCD.
- Unpacked BCD digits converted by AND is scattered across AX.
- So, AAD makes BCD digits across AX to be available in **AL** as hex value.
- ORing converts the hex value in AH-AL back into ASCII value.

Logical Instructions

Boolean operation instructions

NOT

- Format- NOT destination
- (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.

AND, OR & XOR

Format- AND destination, source. 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:

- register, register
- register , memory location
- An immediate operand , general register/memory operand.
- AND, OR, and XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.

Bit Test and Modify Instructions (Special 80386 Instructions)

- Format- Bit destination, source
- Destination - register or memory location.
- Source - immediate byte value or register.
- This group of instructions operates on a single bit which can be in memory or in a general register.
- The location of the bit is specified by source operand.
- These instructions first assign the value of the selected bit to CF, the carry flag.

- Then a new value is assigned to the selected bit**, as determined by the operation**.
- OF, SF, ZF, AF, PF are left in an undefined state.

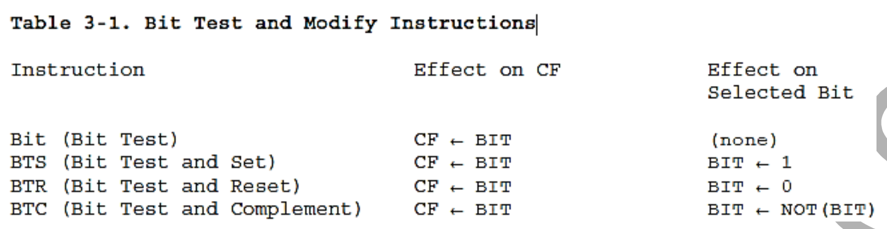


Table 3-1. Bit Test and Modify Instructions

Instruction	Effect on CF	Effect on Selected Bit
Bit (Bit Test)	CF ← BIT	(none)
BTS (Bit Test and Set)	CF ← BIT	BIT ← 1
BTR (Bit Test and Reset)	CF ← BIT	BIT ← 0
BTC (Bit Test and Complement)	CF ← BIT	BIT ← NOT(BIT)

Bit Scan instructions (Special 80386 Instructions)

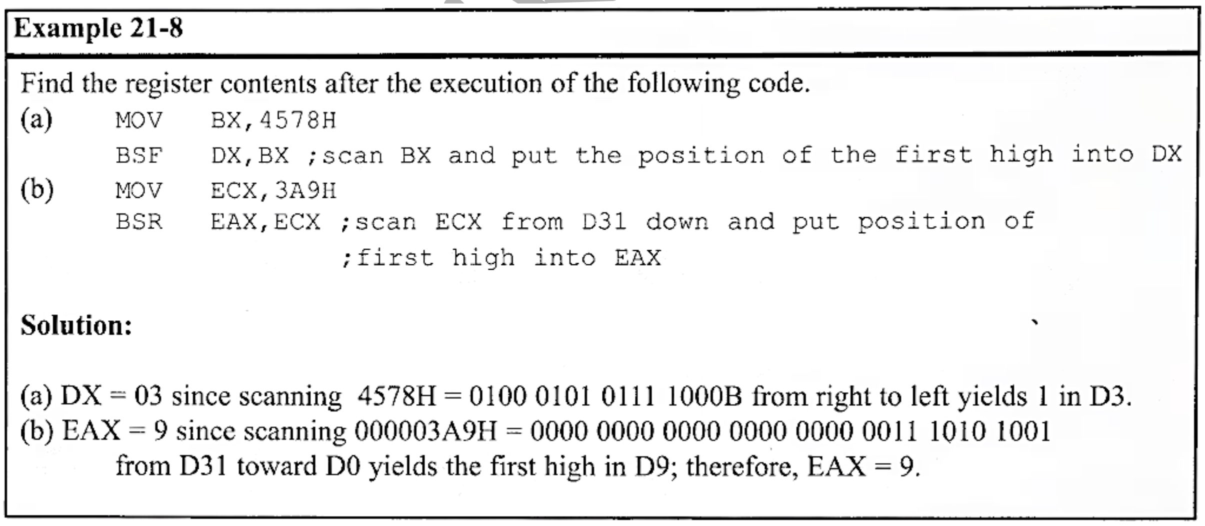
- These instructions scan a word or doubleword for a one-bit and store the index of the first set bit into a register.
- The bit string being scanned may be either in a register or in memory.
- The ZF flag is set if the entire word is zero (no set bits are found); ZF is cleared if a one-bit is found.
- If no set bit is found, the value of the destination register is undefined.

BSF

- Format- BSF destination, source
- Source- bit string to be scanned
- Destination- index of first high bit from left.
- (Bit Scan Forward) scans from low-order to high-order (starting from bit index zero)

BSR

- (Bit Scan Reverse) scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).
- Source- bit string to be scanned
- Destination- index of first high bit from right.



Example 21-8

Find the register contents after the execution of the following code.

```
(a)     MOV    BX,4578H
        BSF    DX,BX ;scan BX and put the position of the first high into DX
(b)     MOV    ECX,3A9H
        BSR    EAX,ECX ;scan ECX from D31 down and put position of
                       ;first high into EAX
```

Solution:

(a) DX = 03 since scanning 4578H = 0100 0101 0111 1000B from right to left yields 1 in D3.
(b) EAX = 9 since scanning 000003A9H = 0000 0000 0000 0000 0000 0011 1010 1001
from D31 toward D0 yields the first high in D9; therefore, EAX = 9.

Shift and Rotate Instructions

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.

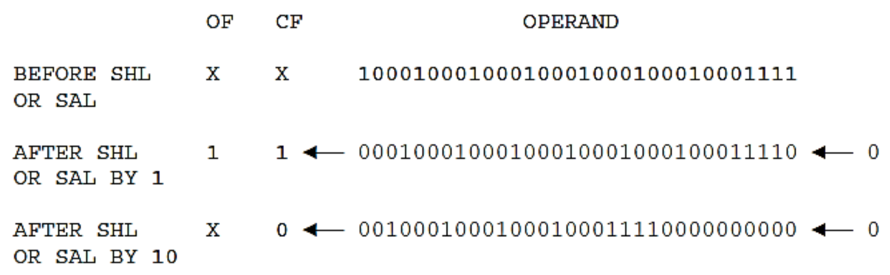
SAL

- Format- *SAL destination, 1* **OR** *SAL destination, ImmediateValue* **OR** *SAL destination, CL*
- (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. See Figure 3-6.
- CF receives the last bit shifted out of the left of the operand.

SHL

- (Shift Logical Left) is a synonym for SAL (refer to SAL).

Figure 3-6. SAL and SHL


```
                 OF   CF                 OPERAND

BEFORE SHL       X    X      10001000100010001000100010001111
OR SAL

AFTER SHL        1    1 <--  00010001000100010001000100011110 <-- 0
OR SAL BY 1

AFTER SHL        X    0 <--  00100010001000100011110000000000 <-- 0
OR SAL BY 10
```

SHR

- Format- *SHR destination, 1* **OR** *SHR destination, ImmediateValue* **OR** *SHR destination, CL*
- (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. See Figure 3-7.
- CF receives the last bit shifted out of the right of the operand.