

# 210255: Microprocessor

<b>Teaching Scheme</b>	<b>Examination Scheme</b>	<b>Credit</b>
TH: 03 Hours /Week	In-Sem: 30 Marks End-Sem: 70 Marks	3

# 210257: Microprocessor Lab

Teaching Scheme	Examination Scheme	Credit
PR: 02 Hours /Week	TW : 25 Marks PR: 25 Marks	01

# Prerequisites

Digital Electronics & Logic Design

# Course Objectives

1. **Course Objectives:**
2. **To learn the architecture and programmer's model of advanced processor**
3. **To understand the system level features and processes of advanced processor**
4. **To acquaint the learner with application instruction set and logic to build assembly language programs.**
5. **To understand debugging and testing techniques confined to 80386 DX**

# Course Outcomes

1. **On completion of the course, student will be able to-**
2. CO1: To apply the assembly language programming to develop small real life embedded application.
3. CO2: To understand the architecture of the advanced processor thoroughly to use the resources for programming
4. CO3: To understand the higher processor architectures descended from 80386 architecture

# Unit I

80386DX- Basic Programming Model and Applications Instruction Set

**Memory Organization and Segmentation-** Global Descriptor Table, Local Descriptor Table, Interrupt Descriptor Table, Data Types, Registers, Instruction Format, Operand Selection, Interrupts and Exceptions

**Applications Instruction Set-** Data Movement Instructions, Binary Arithmetic Instructions, Decimal Arithmetic Instructions, Logical Instructions, Control Transfer Instructions, String and Character Transfer Instructions, Instructions for Block Structured Language, Flag Control Instructions, Coprocessor Interface Instructions, Segment Register Instructions, Miscellaneous Instructions.

**CO Mapped - CO1 and CO2**



# INTRODUCTION

## *What is Microprocessor ?*

- It is a program controlled semiconductor device (IC), which fetches, decode and executes instructions.





## 2. *What are the basic units of a microprocessor ?*


- The basic units or blocks of a microprocessor are ALU, an array of registers and control unit.

### *3.what is Software and Hardware?*

- The Software is a set of instructions or commands needed for performing a specific task by a programmable device or a computing machine.
- The Hardware refers to the components or devices used to form computing machine in which the software can be run and tested.
- Without software the Hardware is an idle machine.

#### *4. What is assembly language?*

- The language in which the mnemonics (short-hand form of instructions) are used to write a program is called assembly language.
- The manufacturers of microprocessor give the mnemonics.



*5. What are machine language and assembly language programs?*

- The software developed using 1's and 0's are called machine language programs.
- The software developed using mnemonics are called assembly language programs.



*6. What is the drawback in machine language and assembly language programs?*

- The machine language and assembly language programs are machine dependent.
- The programs developed using these languages for a particular machine cannot be directly run on another machine .

## *7. Define bit, byte and word.*

- A digit of the binary number or code is called bit. Also, the bit is the fundamental storage unit of computer memory.
- The 8-bit (8-digit) binary number or code is called byte and 16-bit binary number or code is called word.

(Some microprocessor manufactures refer the basic data size operated by the processor as word).

## *8. What is a bus?*

- Bus is a group of conducting lines that carries data, address and control signals.

## *9. What is the function of microprocessor in a system?*

- The microprocessor is the master in the system, which controls all the activity of the system.
- It issues address and control signals and fetches the instruction and data from memory.
- Then it executes the instruction to take appropriate action.



# Microprocessor :

CPU on a single chip

CPU consists of control unit, ALU and registers.

## Microprocessor based system

-CPU

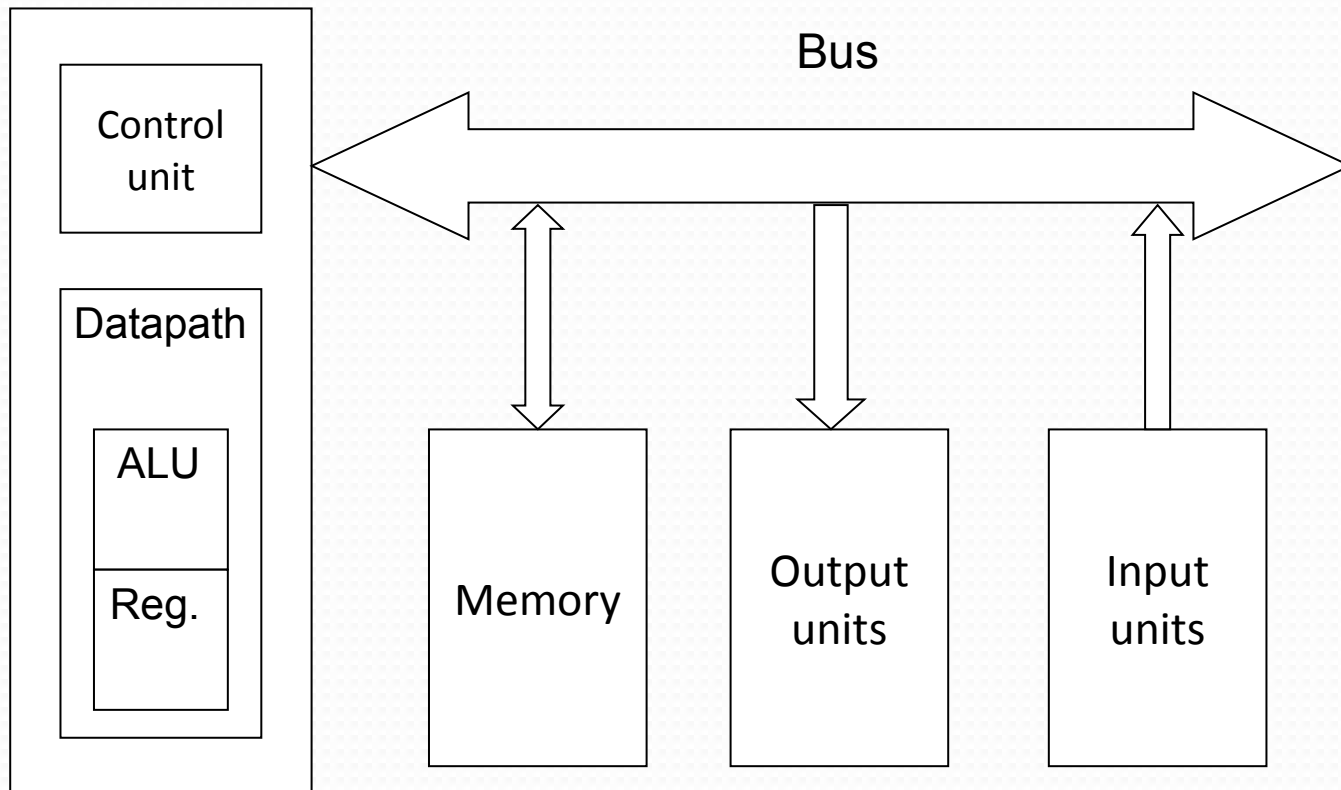
-Input


-Output

-Memory

# *What are microprocessor-based systems?*

Microprocessor





Microprocessor-based systems are electrical systems consisting of microprocessors, memories, I/O units, and other peripherals.

- Microprocessors are the brains of the systems.
- Microprocessors access memories and other units through buses.
- The operations of microprocessors are controlled by instructions stored in memories



# Reference Material

# Text Books

1. Douglas Hall, "Microprocessors & Interfacing", McGraw Hill, Revised 2 Edition, 2006 ISBN 0-07-100462-9
2. A.Ray, K.Bhurchandi, "Advanced Microprocessors and peripherals: Arch, Programming & Interfacing", Tata McGraw Hill, 2004 ISBN 0-07-463841-6
3. Intel 80386 Programmer's Reference Manual 1986, Intel Corporation, Order no.: 231630-011, December 1995.
4. Intel 80386 Hardware Reference Manual 1986, Intel Corporation, Order no.: 231732-001, 1986.
5. James Turley- "Advanced 80386 Programming Techniques", McGraw-Hill, ISBN: 10:0078813425, 13: 978-0078813429.

# 80386

- 32-bit microprocessor
- forms the basis for a high-performance 32-bit system
- Features : multitasking support, memory management, pipelined architecture, address translation caches, and a high-speed bus interface [all on one chip]
- integration of features speeds the execution of instructions
- Paging and dynamic data bus sizing can each be invoked selectively , making the 80386 suitable for a wide variety of system designs and user applications.

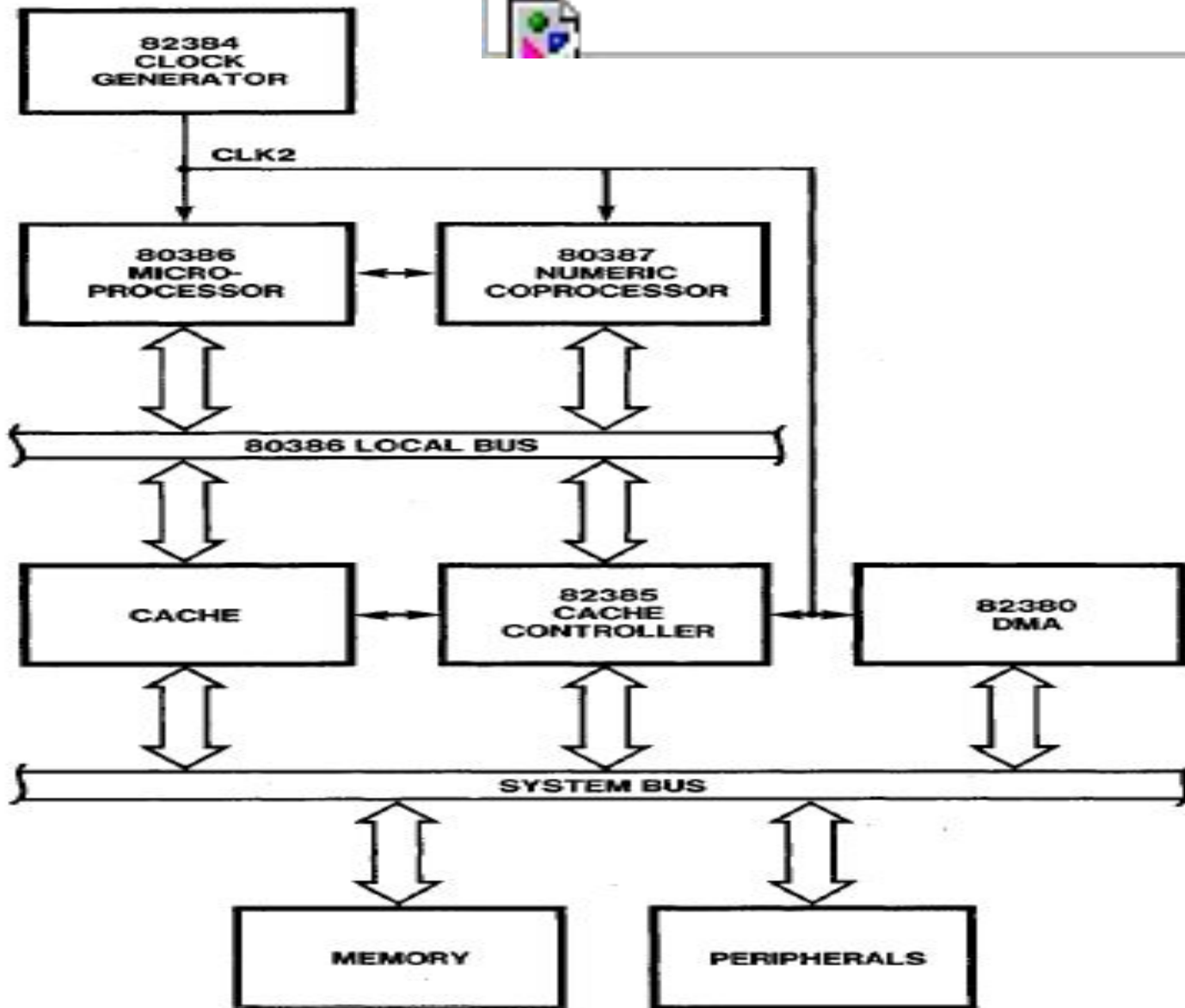
**Table 1-1. 80386 System Components**

<b>Component</b>	<b>Description</b>
80386 Microprocessor	32-bit high-performance microprocessor with on-chip memory management and protection
80287 or 80387 Numeric Coprocessor	Performs numeric instruction in parallel with 80386; expands instruction set
82380 Integrated System Peripheral	Provides 32-bit high-speed direct memory access, interrupt control, and interval timers
82385 Cache Controller	Provides cache directory and management logic
82384 Clock Generator	Generates system clock and RESET signal
8259A Programmable Interrupt Controller	Provides interrupt control and management
82258 Advanced DMA	Performs direct memory controller access (DMA)

# 80386

- 32-bit wide internal and external data paths
- Eight general-purpose 32-bit registers
- The instruction set offers 8-, 16-, and 32-bit data types
- The processor outputs 32-bit physical addresses directly, for a physical memory capacity of four gigabytes.





# 80386 System Overview

- Separate 32-bit data and address paths.
- A 32-bit memory access can be completed in only two clock cycles, enabling the bus to sustain a throughput of 40 megabytes per second (at 20 MHz).
- By making prompt transfers between the microprocessor, memory, and peripherals, the high-speed bus design ensures that the entire system benefits from the processor's increased performance.

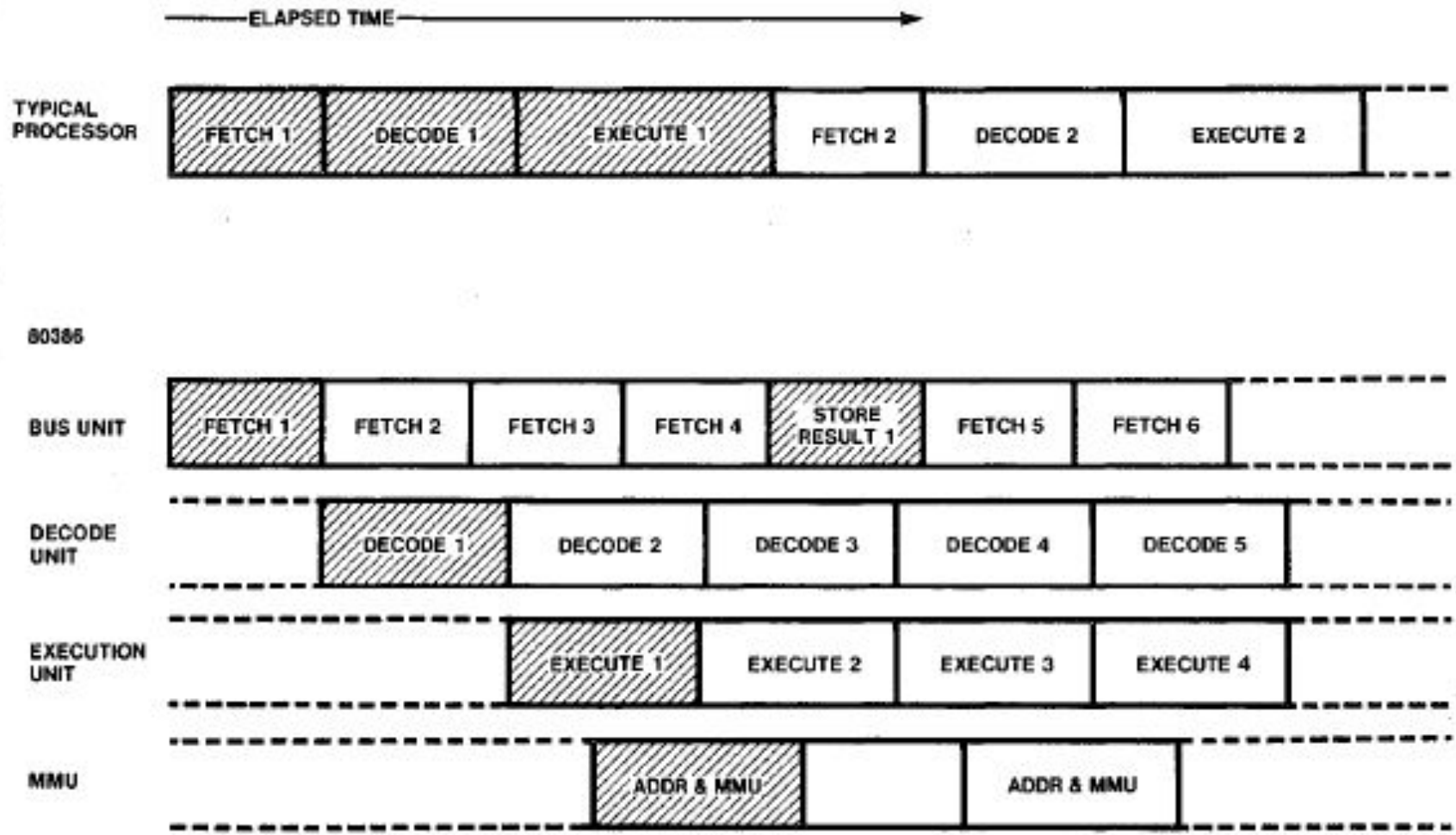
# 80386 System Overview

- Pipelined architecture enables the 80386 to perform instruction fetching, decoding, execution, and memory management functions in parallel.
- Because the 80386 prefetches instructions and queues them internally, instruction fetch and decode times are absorbed in the pipeline; the processor rarely has to wait for an instruction to execute.

# 80386 Internal Architecture

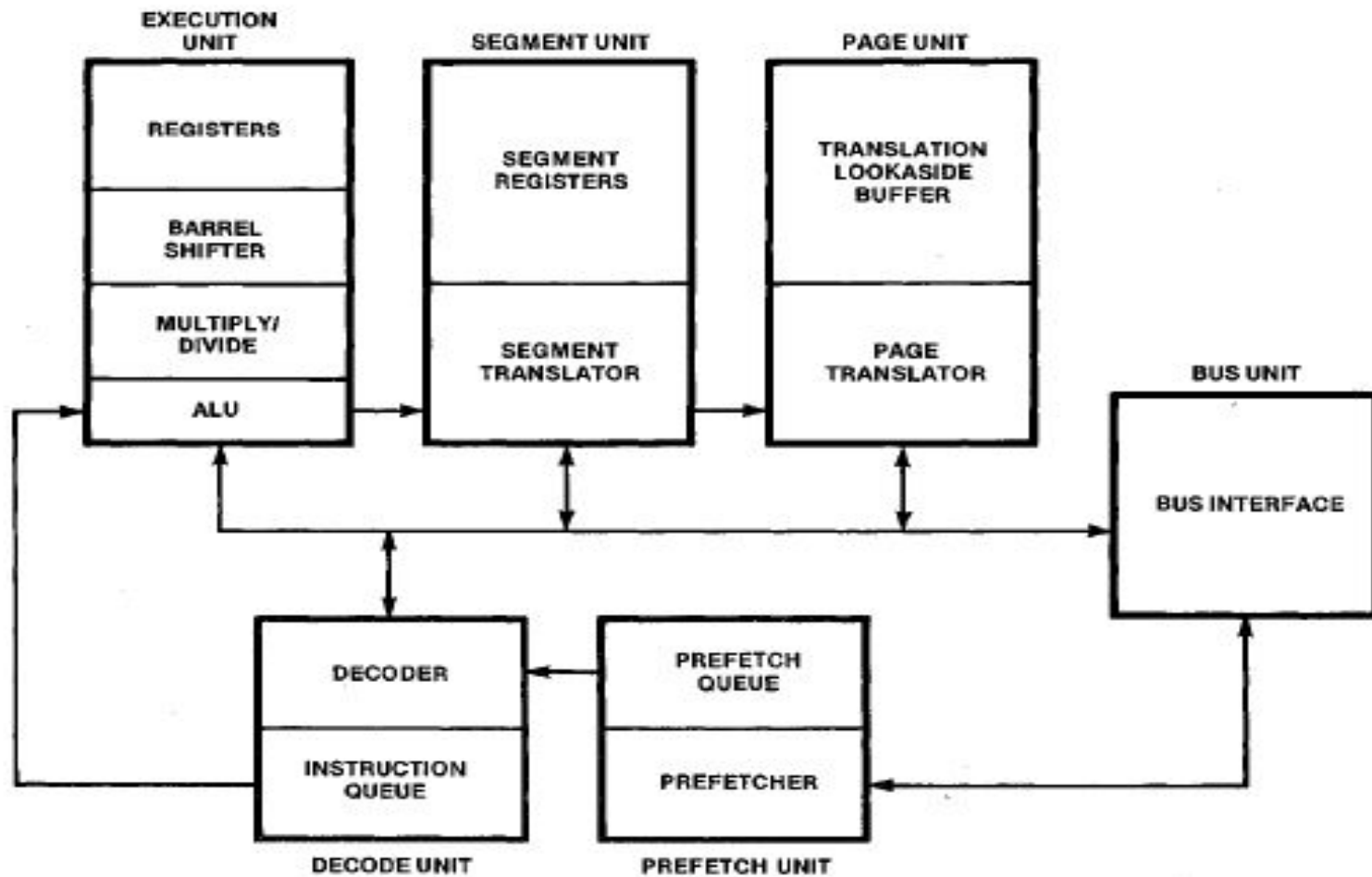
The six functional units of the 80386 are identified as follows:

1. Bus Interface Unit
2. Code Prefetch Unit
3. Instruction Decode Unit
4. Execution Unit
5. Segmentation Unit
6. Paging Unit



210760-11

Figure 2-1. Instruction Pipelining



G30107

Figure 2-2. 80386 Functional Units

# 1. Basic Programming Model x386

The basic programming model consists of these aspects:

1. Memory organization and segmentation
2. Data types
3. Registers
4. Instruction format
5. Operand selection
6. Interrupts and exceptions



## 1.1 Memory organization & segmentation

### **1.1.1 The "Flat" Model**

### **1.1.2 The Segmented Model**



# 1.1 Memory organization & segmentation

- Physical memory organized as sequence of 8-bit bytes
- Each byte is assigned unique address (range 0 to  $2^{32} - 1$ )
- Physical address space : 4 GB (physical memory)
- Logical address space : 64 TB (Virtual memory)
- x386 programs independent of physical address space
- Programmer not known @ the physical memory addresses
- Also, no clue @ exact location of data n code in

# 1.1 continued

- The architecture of the 80386 gives designers the freedom to choose a model for each task.
- The model of memory organization can range between the following :
  1. A "flat" address space consisting of a single array of up to 4 gigabytes.
  2. A segmented address space consisting of a collection of up to 16,383 linear address spaces of up to 4 gigabytes each.

## 1.1.1 The "Flat" Model

- The applications programmer sees a single array of up to  $2^{32}$  bytes (4 gigabytes)
- The processor maps the 4 gigabyte flat space onto the physical address space by the address translation mechanisms
- Applications programmers do not need to know the details of the mapping.
- A pointer into this flat address space is a 32-bit ordinal number that may range from 0 to  $2^{32}-1$ .
- Relocation of separately-compiled modules in this space must be performed by systems software (e.g. linkers, locators, binders, loaders)

## 1.1.2 The Segmented Model

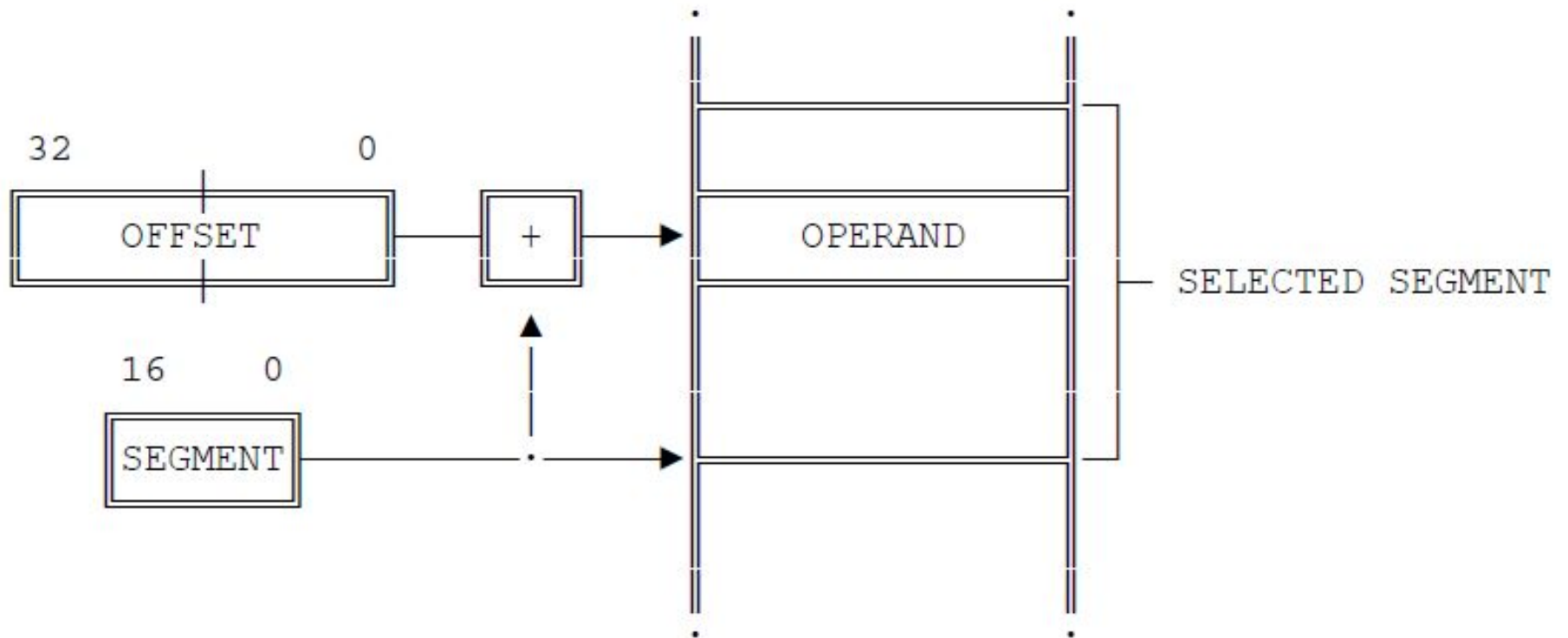
- The address space as viewed by an applications program (called the logical address space) is a much larger space of up to  $2^{46}$  bytes (64 terabytes).
- The processor maps the 64 terabyte logical address space onto the physical address space (up to 4 gigabytes) by the address translation mechanisms.
- Applications programmers do not need to know the details of this mapping.

## 1.1.2 continued

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

# 1.1.2 continued

- A complete pointer in logical address space consists of two parts (Figure Next Slide)
  1. A segment selector, which is a 16-bit field that identifies a segment.
  2. An offset, which is a 32-bit ordinal that addresses to the byte level within a segment.



**Fig: Two-Component Pointer**

## 1.1.2 continued

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



# 1.2 Data Types

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

# Byte....

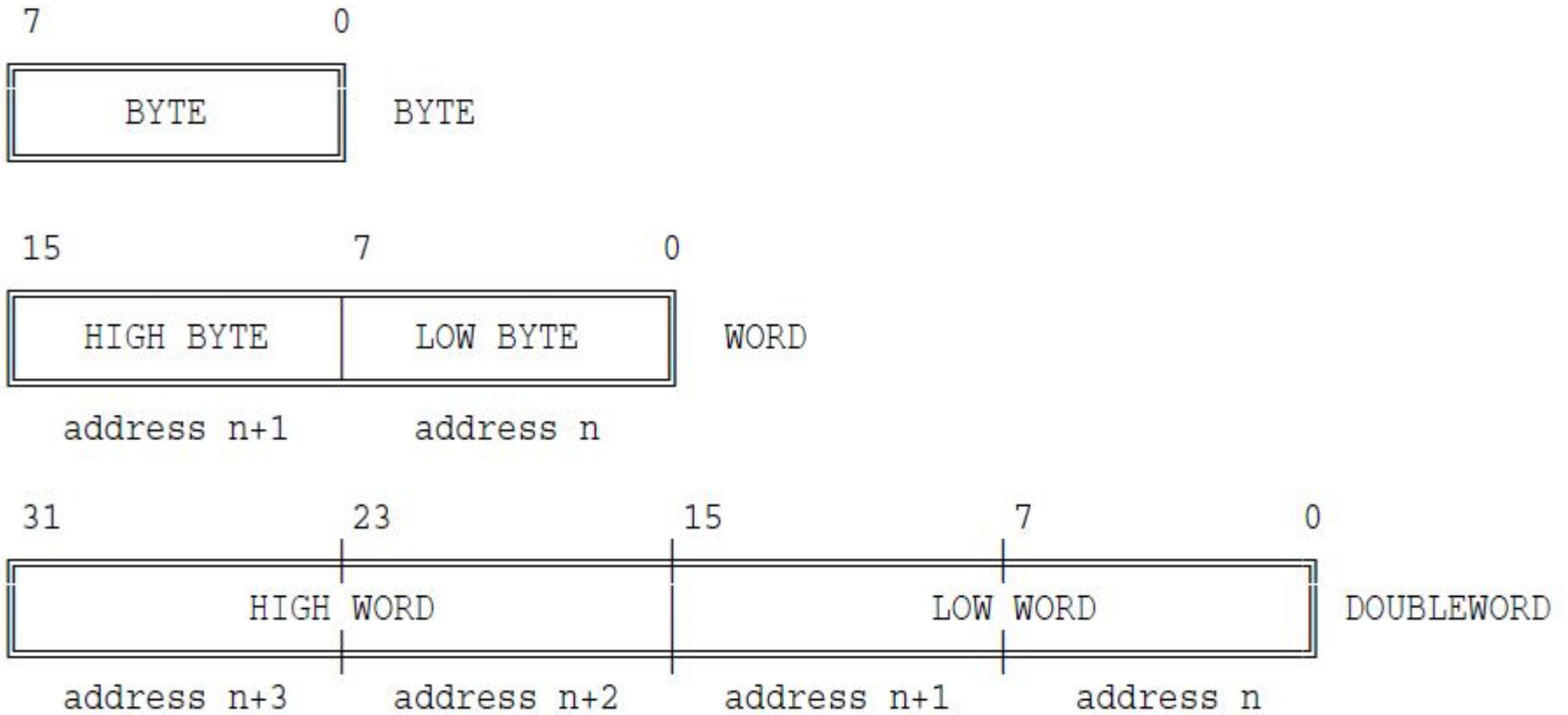
- Each byte within a word has its own address, and the smaller of the addresses is the address of the word.
- The byte at this lower address contains the eight least significant bits of the word
- while the byte at the higher address contains the eight most significant bits.

# Word....


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

# Doubleword

- 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



**Fig: Fundamental Data Types**

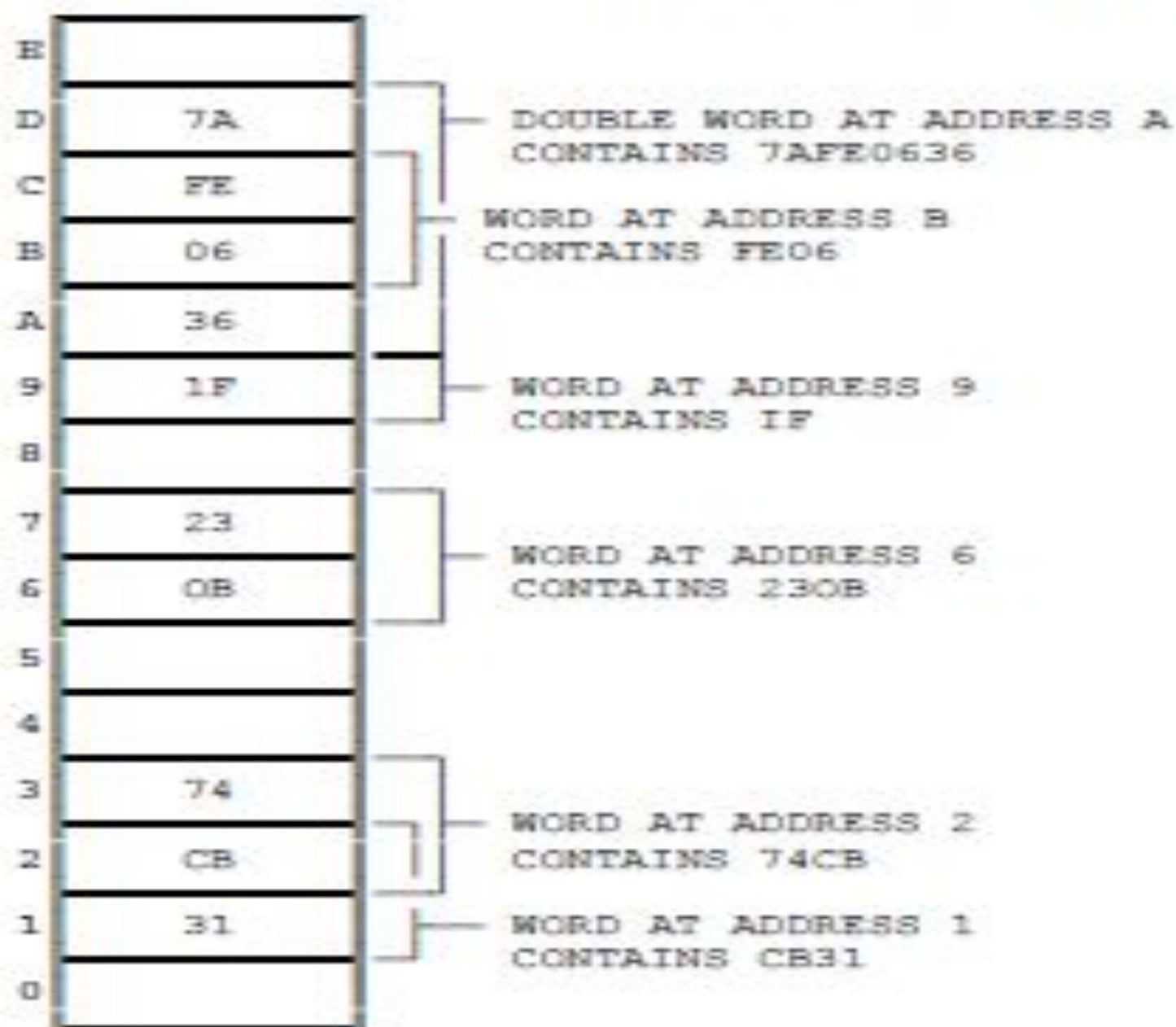


# **Fig: Bytes, Words, and Doublewords in Memory**

BYTE  
ADDRESS

MEMORY  
VALUES

(All values in hexadecimal)



# Additional Data Types

- The processor also supports additional interpretations of these operands.
- Depending on the instruction referring to the operand, the following additional data types are recognized:

1. Integer

2. Ordinal

3. Near Pointer

4. Far Pointer

5. String

6. Bit field

7. Bit string

8. BCD

9. Packed BCD





Fig: 80386 Additional Data Types

# Integer

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

# Ordinal

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

# 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.

# Far Pointer

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

## **String:**

- A contiguous sequence of bytes, words, or doublewords.
- A string may contain from zero bytes 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.

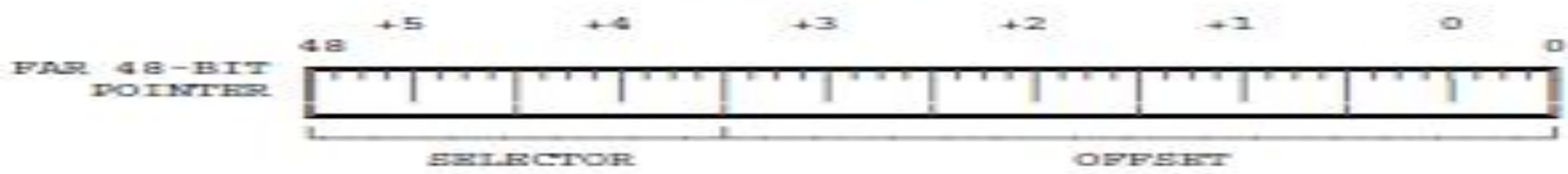
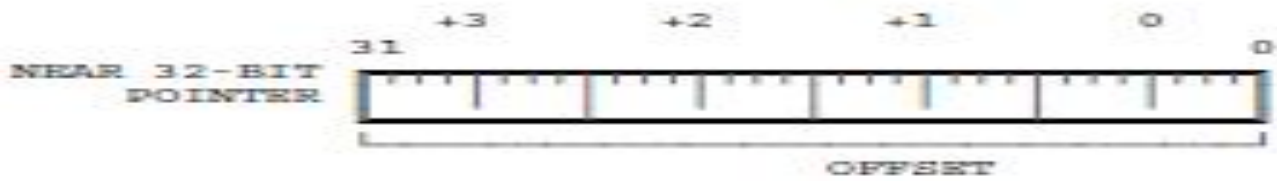
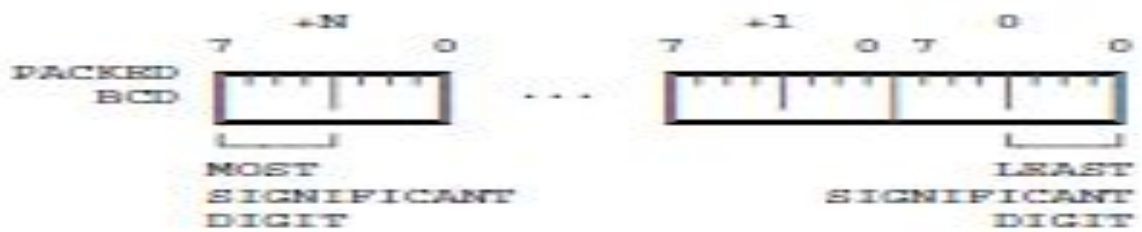
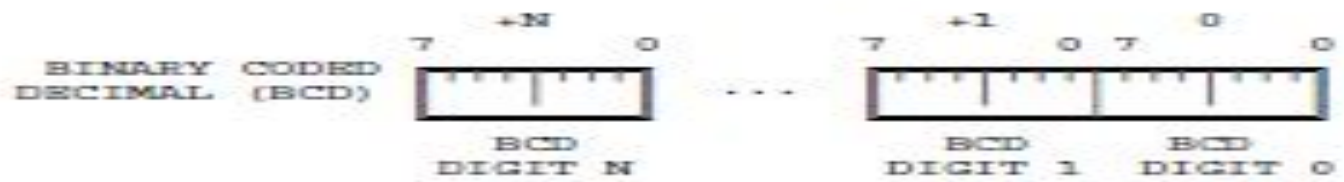
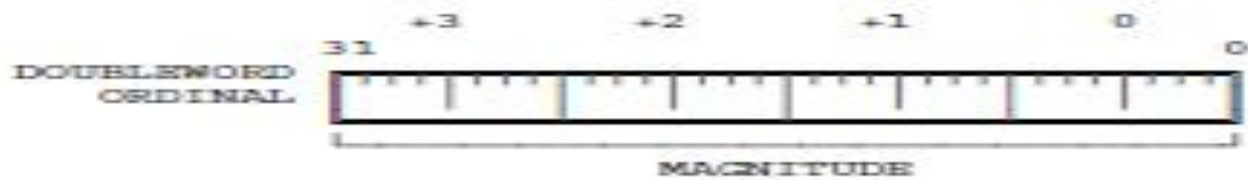
# BCD

- A 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.

# 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.







# 1.3 Registers

1.3.1 General Registers

1.3.2 Segment Registers

1.3.3 Stack Implementation

1.3.4 Flag Register

# Register Categories

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

# 1.3.1 General Registers

- Eight 32-bit registers
- EAX,EBX,ECX, EDX,EBP,ESP, ESI,EDI
- Use: primarily to contain operands for arithmetic and logical operations.
- Can also be used for operands of address computations
- ***Exception: ESP can not be used as an index operand***

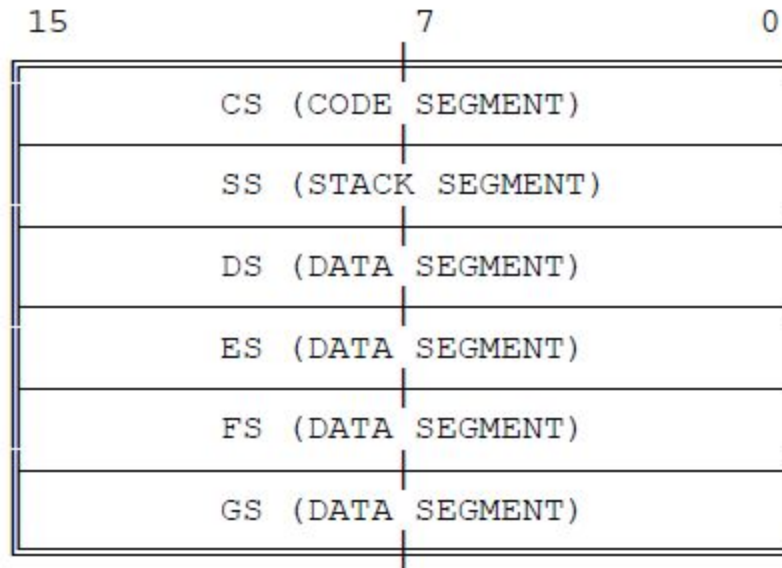


# Fig: 80386 Applications Register Set

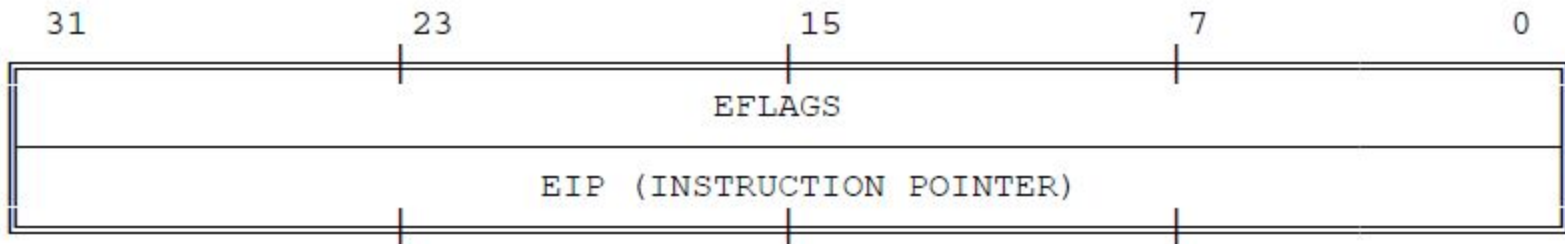
## GENERAL REGISTERS

31	23	15	7	0
		EAX	AH	AX AL
		EDX	DH	DX DL
		ECX	CH	CX CL
		EBX	BH	BX BL
		EBP		BP
		ESI		SI
		EDI		DI
		ESP		SP

SEGMENT  
REGISTERS



STATUS AND INSTRUCTION REGISTERS





# Register Organization

- Eight 32 - bit general purpose registers may be used as either 8 bit or 16 bit registers.
- A 32 - bit register known as an extended register, is represented by the register name with prefix E.
- The 16 bit registers BP, SP, SI and DI in 8086 are available with their extended size of 32 bit and are names as EBP, ESP, ESI and EDI.
- AX represents the lower 16 bit of the 32 bit register EAX.
- BP, SP, SI, DI represents the lower 16 bit of their 32 bit counterparts, and can be used as independent 16 bit registers.
- The six segment registers available in 80386 are CS, SS, DS, ES, FS and GS.
- The CS and SS are the code and the stack segment registers respectively, while DS, ES, FS, GS are 4 data segment registers.
- A 16 bit instruction pointer IP is available along with 32 bit counterpart EIP.

# Use

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

## 1.3.2 Segment Registers

- The segment registers of the 80386 give systems software designers the flexibility to choose among various models of memory organization.

## 1.3.2 Continued

- Complete programs generally consist of many different modules
- Each module consisting of instructions and data
- At any given time during program execution, only a small subset of a program's modules are actually in use.
- The 80386 architecture takes advantage of this by providing mechanisms to support direct access to the instructions and data of the current module's environment, with access to additional segments on demand.

## 1.3.2 Continued

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

# CS

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

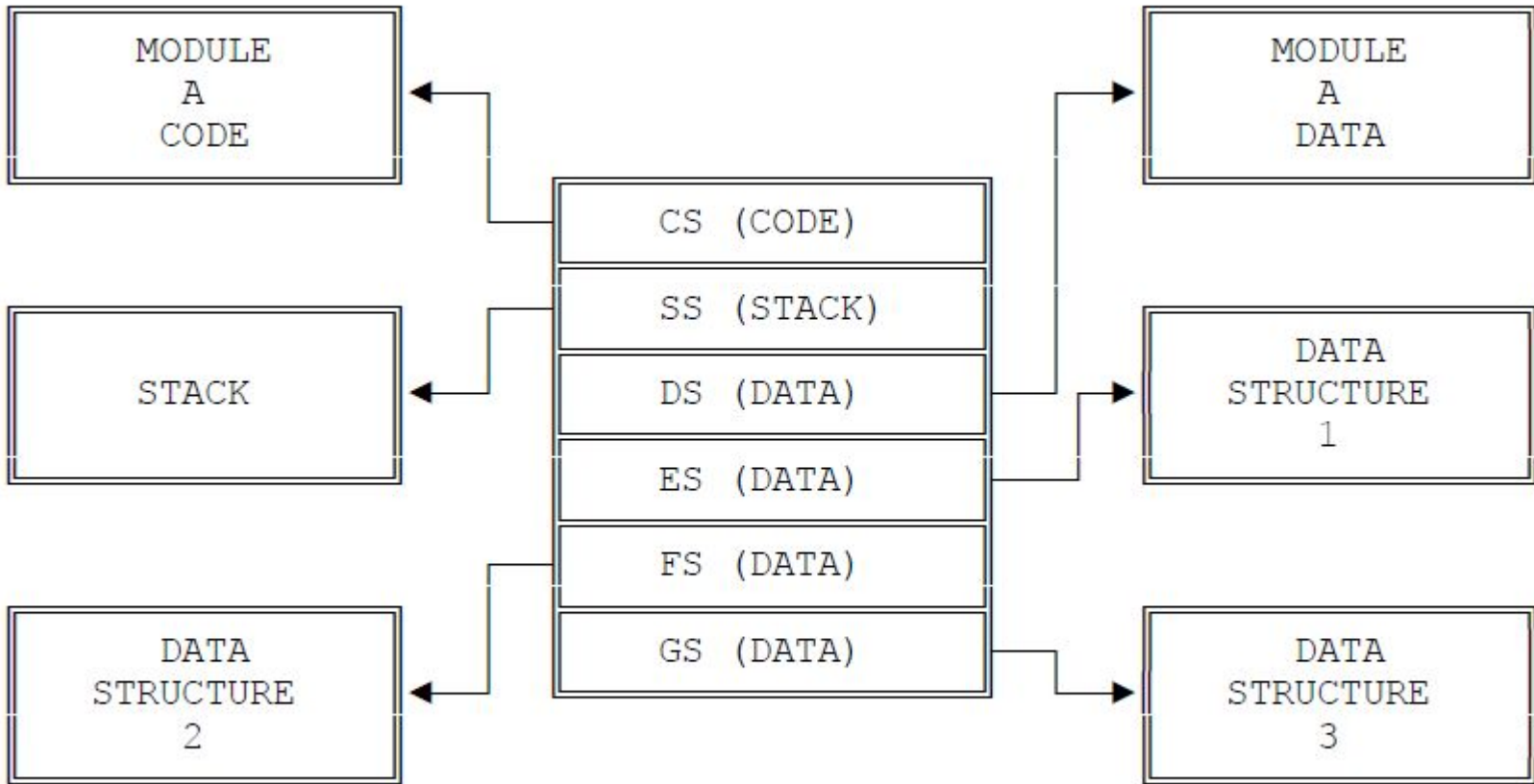
# SS

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

- The DS, ES, FS, and GS registers allow the specification of four data segments, each addressable by the currently executing program.
- Accessibility to four separate data areas helps programs efficiently access different types of data structures; for example, one data segment register can point to the data structures of the current module, another to the exported data of a higher-level module, another to a dynamically created data structure, and another to data shared with another task.



- An operand within a data segment is addressed by specifying its offset either directly in an instruction or indirectly via general registers.



**Fig: Use of Memory Segmentation**

# 1.3.3 Stack Implementation

- Stack operations are facilitated by three registers:
  1. The stack segment (SS) register.
  2. The stack pointer (ESP) register.
  3. The stack-frame base pointer (EBP) register.

# SS Register

- Stacks are implemented in memory.
- A system may have a number of stacks that is limited only by the maximum number of segments.
- A stack may be up to 4 gigabytes long, the maximum length of a segment.
- One stack is directly addressable at a time—the one located by SS.
- This is the current stack, often referred to simply as "the" stack.
- SS is used automatically by the processor for all stack operations.

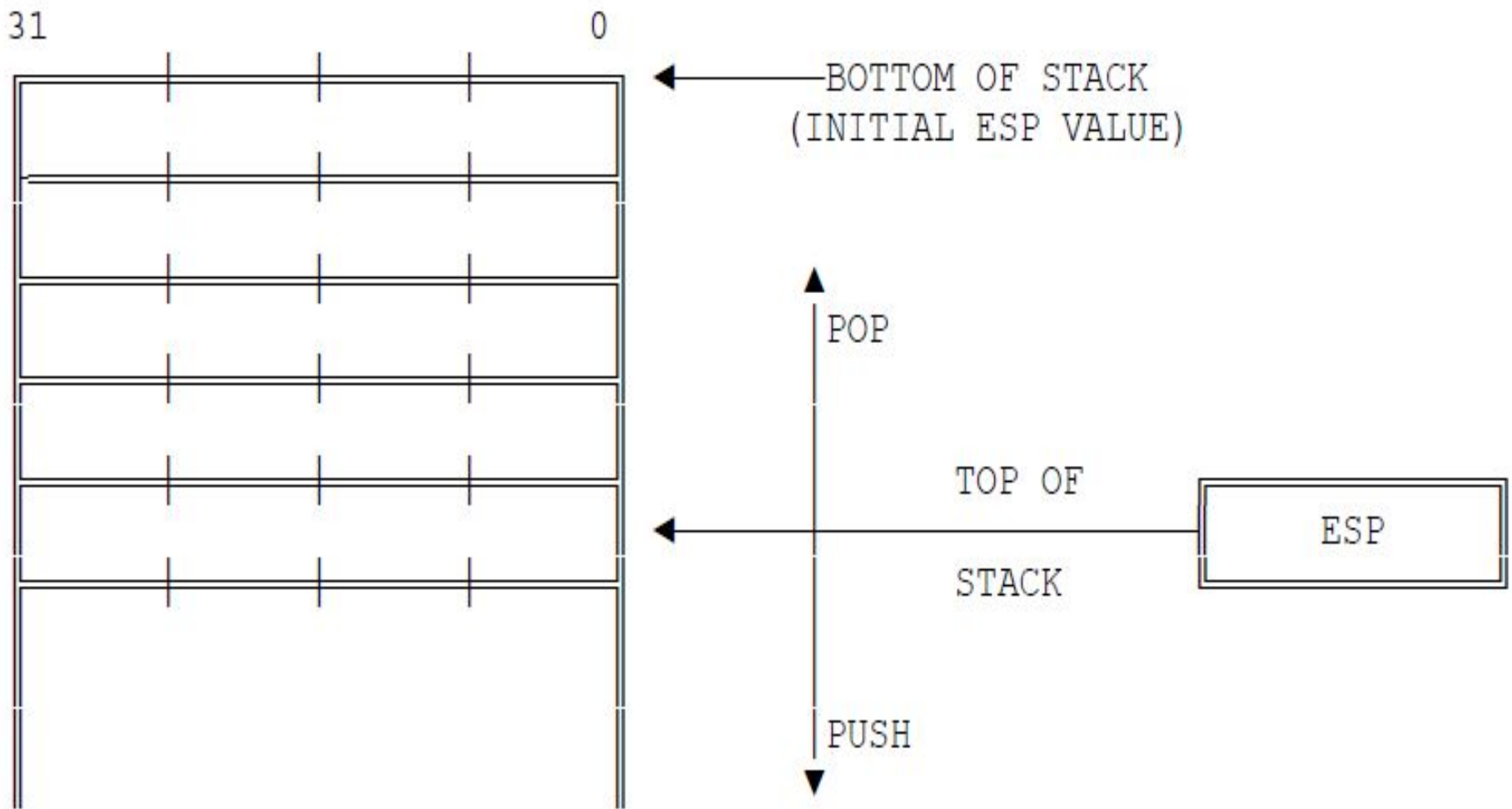


Fig : 80386 Stack

# ESP Register

- ESP points to the top of the push-down stack (TOS).
- It is referenced implicitly by PUSH and POP operations, subroutine calls and returns, and interrupt operations.
- When an item is pushed onto the stack the processor decrements ESP, then writes the item at the new TOS.
- When an item is popped off the stack, the processor copies it from TOS, then increments ESP.
- *The stack grows down in memory toward lesser*

# EBP Register

- The EBP is the best choice of register for accessing data structures, variables and dynamically allocated work space within the stack.
- EBP is often used to access elements on the stack relative to a fixed point on the stack rather than relative to the current TOS.
- It typically identifies the base address of the current stack frame established for the current procedure.
- When EBP is used as the base register in an offset calculation, the offset is calculated automatically in the current stack segment (i.e.,

# EBP Continued

- Because SS does not have to be explicitly specified, instruction encoding in such cases is more efficient.
- EBP can also be used to index into segments addressable via other segment registers.



# 1.3.4 Flags Register

- The flags register is a 32-bit register named EFLAGS.
- The bits within this register
- The flags control certain operations and indicate the status of the 80386.
- The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit.
- This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.
- The flags may be considered in three groups: the status flags, the control flags, and the systems flags.



## Status Flags' Functions

Bit	Name	Function
0	CF	Carry Flag — Set on high-order bit carry or borrow; cleared otherwise.
2	PF	Parity Flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.
4	AF	Adjust flag — Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.
6	ZF	Zero Flag — Set if result is zero; cleared otherwise.
7	SF	Sign Flag — Set equal to high-order bit of result (0 is positive, 1 if negative).
11	OF	Overflow Flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

## Key to Codes

T = instruction tests flag  
M = instruction modifies flag  
(either sets or resets depending on operands)  
0 = instruction resets flag  
— = instruction's effect on flag is undefined  
blank = instruction does not affect flag

## Definition of Conditions

(For conditional instructions Jcond, and SETcond)

Mnemonic	Meaning	Instruction Subcode	Condition Tested
O	Overflow	0000	OF = 1
NO	No overflow	0001	OF = 0
B NAE	Below Neither above nor equal	0010	CF = 1
NB AE	Not below Above or equal	0011	CF = 0
E Z	Equal Zero	0100	ZF = 1
NE NZ	Not equal Not zero	0101	ZF = 0
BE NA	Below or equal Not above	0110	(CF or ZF) = 1
NBE NA	Neither below nor equal Above	0111	(CF or ZF) = 0
S	Sign	1000	SF = 1
NS	No sign	1001	SF = 0
P PE	Parity Parity even	1010	PF = 1
NP PO	No parity Parity odd	1011	PF = 0
L NGE	Less Neither greater nor equal	1100	(SF xor OF) = 1
NL GE	Not less Greater or equal	1101	(SF xor OF) = 0
LE NG	Less or equal Not greater	1110	((SF xor OF) or ZF) = 1
NLE G	Neither less nor equal Greater	1111	((SF xor OF) or ZF) = 0

# Status Flags

- The status flags of the EFLAGS register allow the results of one instruction to influence later instructions.
- The arithmetic instructions use OF, SF, ZF, AF, PF, and CF.
- The SCAS (Scan String), CMPS (Compare String), and LOOP instructions use ZF to signal that their operations are complete.
- There are instructions to set, clear, and complement CF before execution of an arithmetic instruction.

# Control Flag

- The control flag DF of the EFLAGS register controls string instructions.



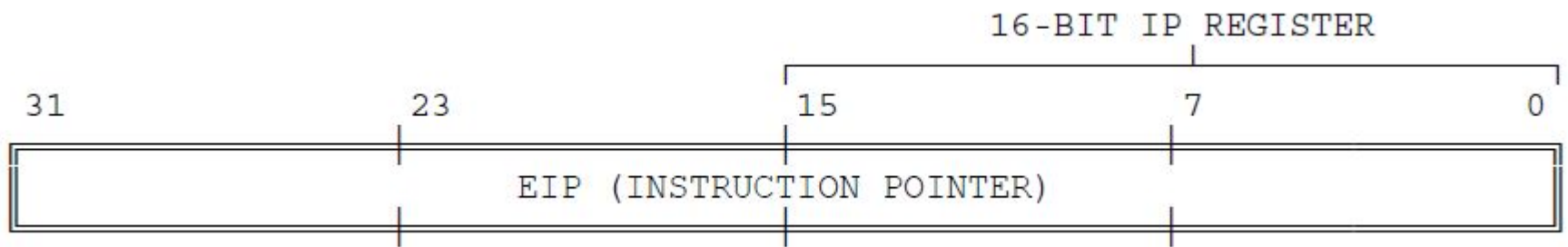
# DF (Direction Flag, bit 10)

- Setting DF causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses.
- Clearing DF causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

# Instruction Pointer Register

- The instruction pointer register (EIP) contains the offset address, relative to the start of the current code segment, of the next sequential instruction to be executed.
- The instruction pointer is not directly visible to the programmer
- It is controlled implicitly by control-transfer instructions, interrupts, and exceptions.





**Fig : Instruction Pointer Register**


# EIP


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

# 1.1.4 Instruction Format

80386 instruction contains :

1. a specification of the operation to be performed,
2. the type of the operands to be manipulated, and
3. the location of these operands  
in encoded format

- 
- If an operand is located in memory, the instruction must also select, explicitly or implicitly, which of the currently addressable segments contains the operand.

- 
- 80386 instructions are composed of various elements and have various formats.
  - Of instruction elements, only one, the opcode, is always present.
  - The other elements may or may not be present, depending on the particular operation involved and on the location and type of the operands.

# Instruction Elements (in order of occurrence)

- I. Prefixes
- II. Opcode
- III. Register Specifier
- IV. Addressing Mode Specifier
- V. SIB (Scale, Index, Base)
- VI. Displacement
- VII. Immediate Operands

# I. Prefixes

- One or more bytes preceding an instruction that modify the operation of the instruction.
- The types of prefixes can be used by applications programs:
  1. Segment Override
  2. Address Size
  3. Operand Size
  4. Repeat

1. **Segment override** :- explicitly specifies which segment register an instruction should use, thereby overriding the default segment-register selection used by the 80386 for that instruction.
2. **Address size** :- switches between 32-bit and 16-bit address generation.
3. **Operand size** :- switches between 32-bit and 16-bit operands.
4. **Repeat** :- used with a string instruction to cause the instruction to act on each element of the string.



## II. Opcode

- Specifies the operation performed by the instruction.
- Some operations have several different opcodes, each specifying a different variant of the operation.

# III. Register Specifier

- An instruction may specify one or two register operands.
- Register specifiers may occur either in the same byte as the opcode or in the same byte as the addressing-mode specifier.

## IV. Addressing Mode Specifier

- When present, specifies whether an operand is a register or memory location;
- If operand in memory, specifies whether a displacement, a base register, an index register, and scaling are to be used.

# V. SIB

- 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.

# VI. 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.

# VII. Immediate Operand

- When present, directly provides the value of an operand of the instruction.
- Immediate operands may be 8, 16, or 32 bits wide.
- In cases where an 8-bit immediate operand is combined in some way with a 16- or 32-bit operand, the processor automatically extends the size of the eight-bit operand, taking into account the sign.

# Operand Selection

- Instruction : 0/ more operands
- 0 operand instruction : NOP
- An operand can be in any of these locations:
  1. In the instruction itself (an immediate operand)
  2. In a register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP in the case of 32-bit operands; AX, BX, CX, DX, SI, DI, SP, or BP in the case of 16-bit operands; AH, AL, BH, BL, CH, CL, DH, or DL in the case of 8-bit operands; the segment registers; or the EFLAGS register for flag operations)
  3. In memory

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



# Implicit/Explicit/Combo Operands

Of the instructions that have operands,

- some specify operands implicitly :

Example of Implicit operand: `AAM`

- others specify operands explicitly :

Example of Explicit operand: `XCHG EAX, EBX`

- still others use a combination of implicit and explicit specification :


Example of Implicit and explicit operands: `PUSH COUNTER`

# Note

- Most instructions have implicit operands.
- All arithmetic instructions, for example, update the EFLAGS register.

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

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

- 
- Certain string instructions and stack manipulation instructions, transfer data from memory to memory.
  - Both operands of some string instructions are in memory and are implicitly specified.
  - Push and pop stack operations allow transfer between memory operands and the memory-based stack.

# Default Segment Register Selection Rules

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

# Effective-Address Computation

- The modR/M byte provides the most flexible of the addressing methods
- instructions that require a modR/M byte as the second byte of the instruction are the most common in the 80386 instruction set.

- For memory operands defined by modR/M, the offset within the desired segment is calculated by taking the sum of up to three components:
  1. A displacement element in the instruction.
  2. A base register.
  3. An index register. The index register may be automatically multiplied by a scaling factor of 2, 4, or 8.



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



***Unit I Part II***  
***Applications Instruction Set***

# Few Instruction Types

- Data Movement Instructions
- Binary Arithmetic Instructions
- Decimal Arithmetic Instructions
- Logical Instructions
- Control Transfer Instructions
- String and Character Transfer Instructions
- Instructions for Block Structured Language
- Flag Control Instructions
- Coprocessor Interface Instructions
- Segment Register Instructions
- Miscellaneous Instructions

# Data Movement Instructions

- provide convenient methods for moving bytes, words, or doublewords of data between memory and the registers of the base architecture
- Types:
  1. General-purpose data movement instructions.
  2. Stack manipulation instructions.
  3. Type-conversion instructions.

# MOV

- Transfers a byte, word, or doubleword from the source operand to the destination operand
  1. To a register from memory
  2. To memory from a register
  3. Between general registers
  4. Immediate data to a register
  5. Immediate data to a memory
- cannot move from memory to memory or from segment register to segment register
- *Exception: string move instruction MOVS*

# XCHG

- Swaps the contents of two operands.
- Takes the place of three MOV instructions
- Does not require a temporary location to save the contents of one operand while load the other is being loaded
- Useful for implementing semaphores or similar data structures for process synchronization
- Can swap two byte /word / doubleword operands
- The operands for the XCHG instruction may be two register operands, or a register operand with a memory operand.

● Will work with both 32-bit and 64-bit registers and memory locations

# Stack Manipulation Instructions

- Push
- Pop

# Push

Function:

1. To decrement the stack pointer (ESP)
2. then to transfer the source operand to the top of stack indicated by ESP

Use

1. to place parameters on the stack before calling a procedure
2. To store temporary variables on the stack

Operands

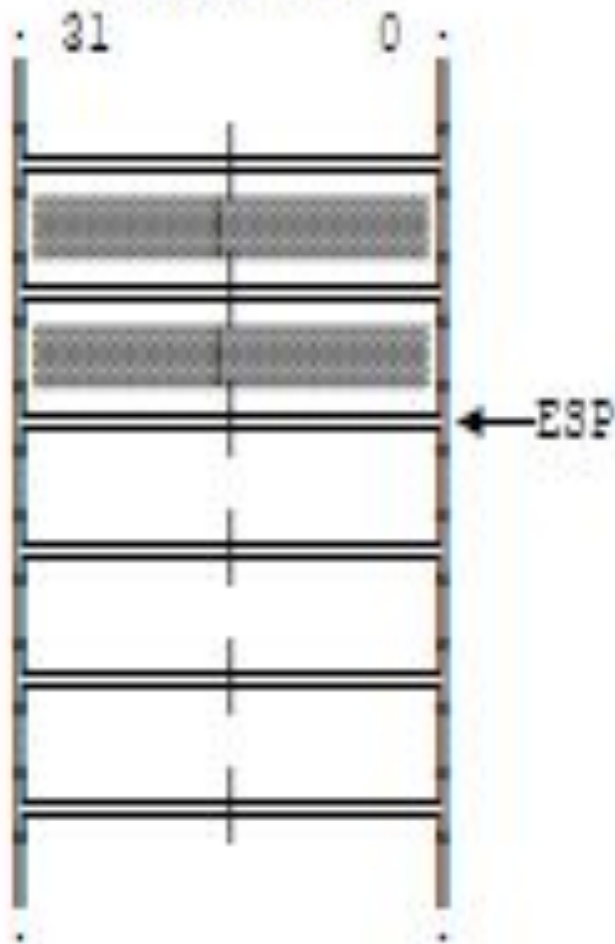
- memory operands, immediate operands, and register operands (including segment registers)



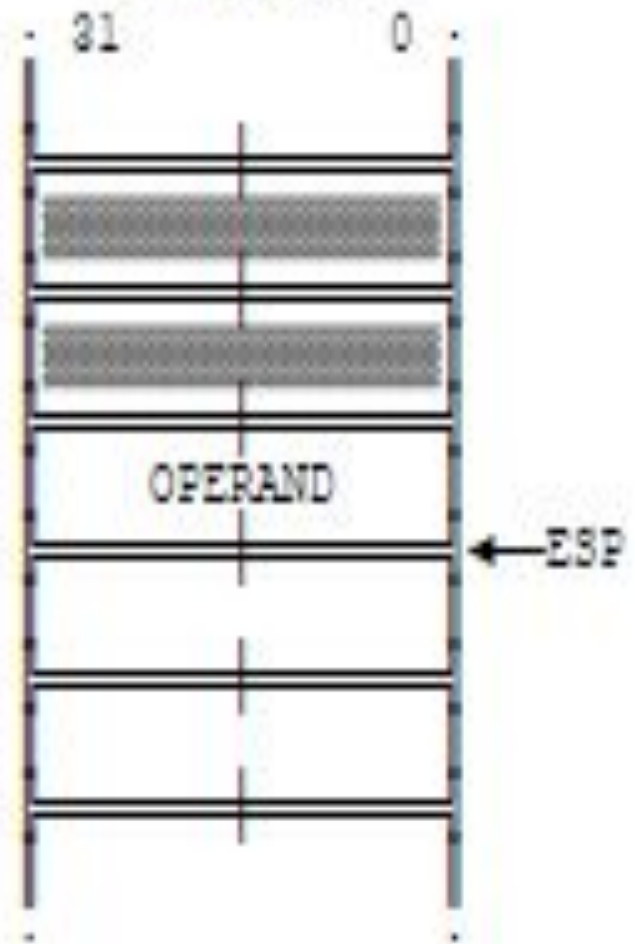
D  
I  
R  
E  
C  
T  
I  
O  
N  
S  
I  
O  
N

↓

BEFORE PUSH



AFTER PUSH



PUSH

# PUSHA

- Push all registers

Function

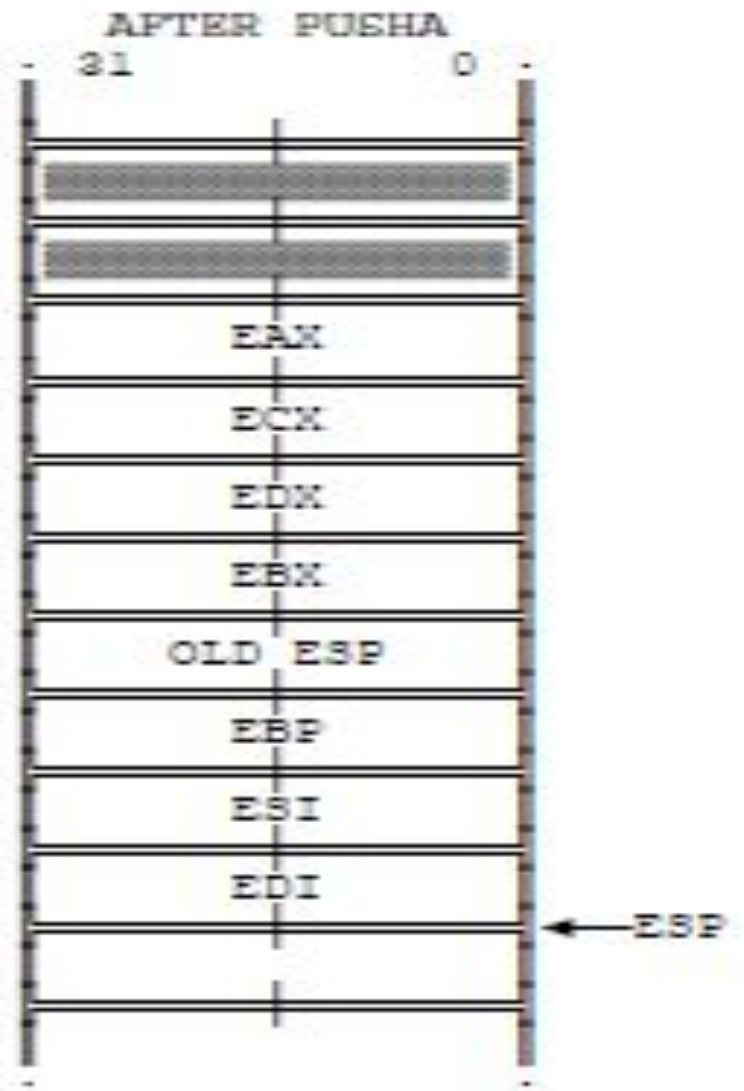
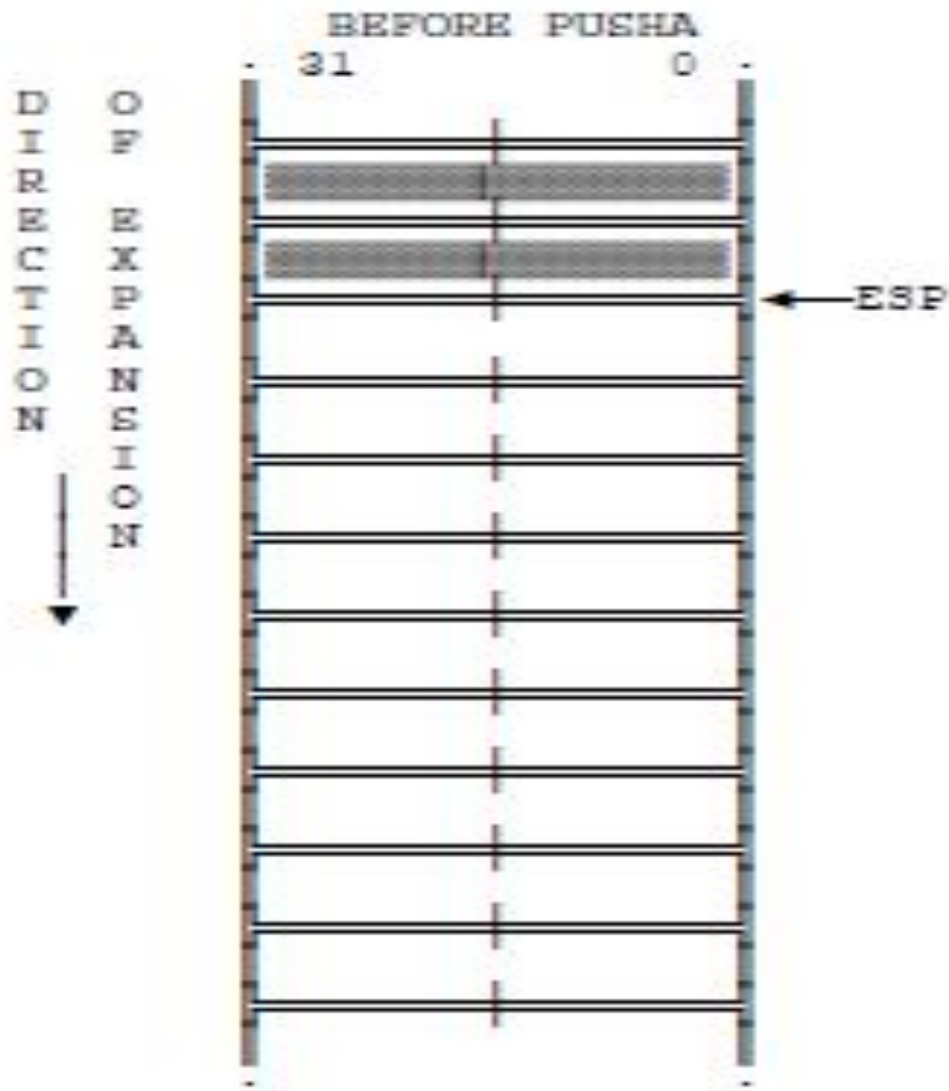
- To save the contents of 8 general registers on the stack

Use

- To simplify procedure calls by reducing the number of instructions required to retain the contents of the general registers for use in a procedure

Order

- general registers : EAX, ECX, EDX, EBX, the initial



PUSHA

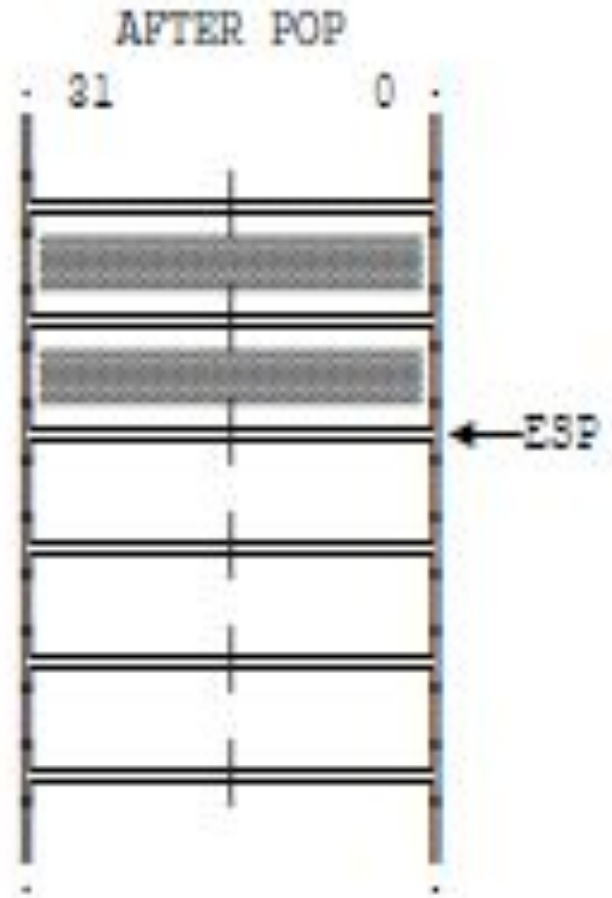
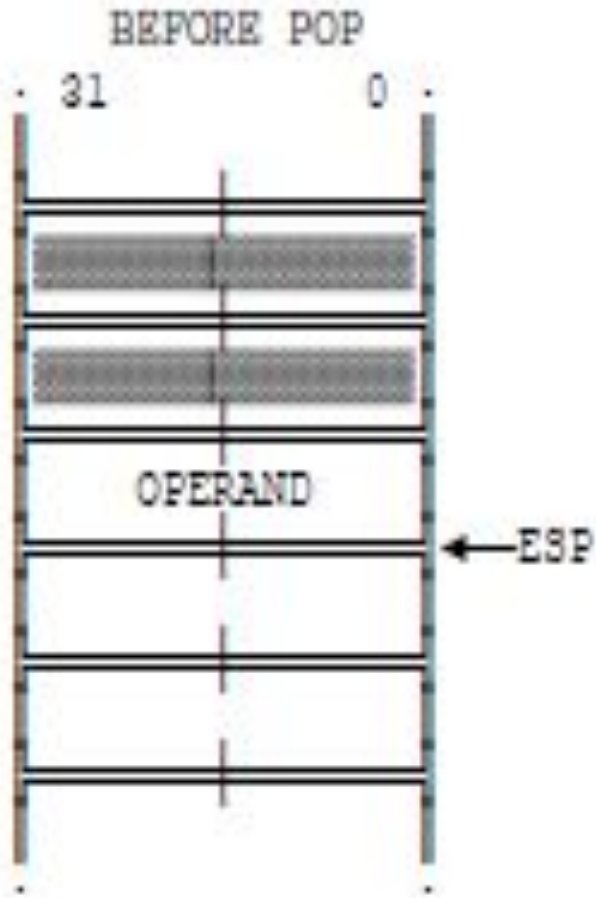
# POP

## Function

1. To transfer the word or doubleword at the current top of stack (indicated by ESP) to the destination operand,
  2. then to increment ESP to point to the new top of stack
- To move information from the stack to a general register, or to memory

D  
I  
R  
E  
C  
T  
I  
O  
N  
↓

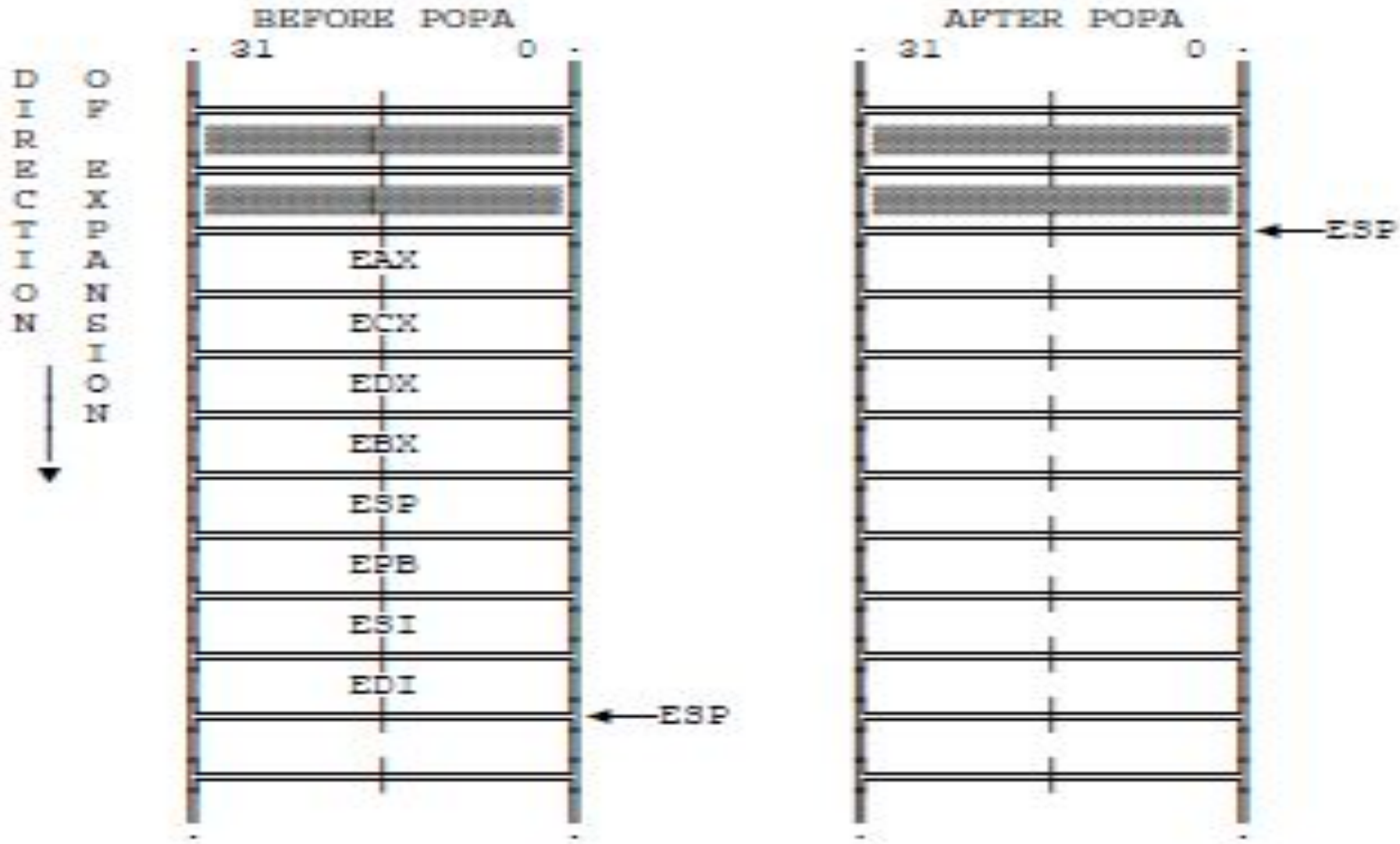
O  
F  
E  
X  
P  
A  
N  
S  
I  
O  
N



**POP**

# POPA

- Pop All Registers
- to restore the registers saved on the stack by PUSHAD
- Exception: it ignores the saved value of ESP



POPA

# Type Conversion Instructions

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





# Sign Extension

# Classes of type conversion instructions

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

# CWD and 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.
- CDQ extends the sign of the doubleword in EAX throughout EDX.
- CWD can be used to produce a doubleword dividend from a word before a word division, and CDQ can be used to produce a quad-word dividend from a doubleword before doubleword division.

# Binary Arithmetic Instructions

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

- Many of the arithmetic instructions operate on both signed and unsigned integers.
- Effect: processor update the flags ZF, CF, SF, and OF in such a manner that subsequent instructions can interpret the results of the arithmetic as either signed or unsigned.
- CF contains information relevant to unsigned integers
- SF and OF contain information relevant to signed integers
- ZF is relevant to both signed and unsigned integers

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

- If the integer is signed, both SF and OF should be tested.
- SF always has the same value as the sign bit of the result.
- The most significant bit of a signed integer is the bit next to the sign—bit 6 of a byte, bit 14 of a word, or bit 30 of a doubleword.
- OF is set in either of these cases:
  1. A one-bit was carried out of the MSB into the sign bit but no one bit was carried out of the sign bit (addition instructions ADD, ADC, INC, AAA, and DAA), i.e. the result was greater than the greatest positive number that could be contained in the destination operand.

# Note

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



# Addition and Subtraction Instructions

1. ADD
2. ADC
3. INC
4. SUB
5. SBB
6. DEC

# 1.ADD

- Add Integers
- to replace the destination operand with the sum of the source and destination operands.
- Sets CF if overflow.

## 2.ADC

- Add Integers with Carry
- To sum 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.

# 3.INC

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

# 4.SUB

- Subtract Integers
- To subtract 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.

# 5.SBB

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

# 6.DEC

- Decrement
- to subtract 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.

## Comparison and Sign Change Instruction

- **CMP** (Compare) subtracts the source operand from the destination operand.
- It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands.
- A subsequent Jcc or SETcc instruction can test the appropriate flags.
- **NEG** (Negate) subtracts a signed integer operand from zero.
- The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.



# Multiplication Instructions

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

# MUL

- Unsigned Integer Multiply
- performs an unsigned multiplication of the source operand and the accumulator.
- If the source is a byte, the processor multiplies it by the contents of *AL* and returns the double-length result to *AH* and *AL*.
- If the source operand is a word, the processor multiplies it by the contents of *AX* and returns the double-length result to *DX* and *AX*.
- If the source operand is a doubleword, the processor multiplies it by the contents of *EAX* and returns the 64-bit result in *EDX* and *EAX*.

# IMUL (Signed Integer Multiply)

- performs a signed multiplication operation.
- IMUL has three variations:
  1. A one-operand form: The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.
  2. A two-operand form. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.
  3. A three-operand form; two are source and one is the destination operand. One of the source

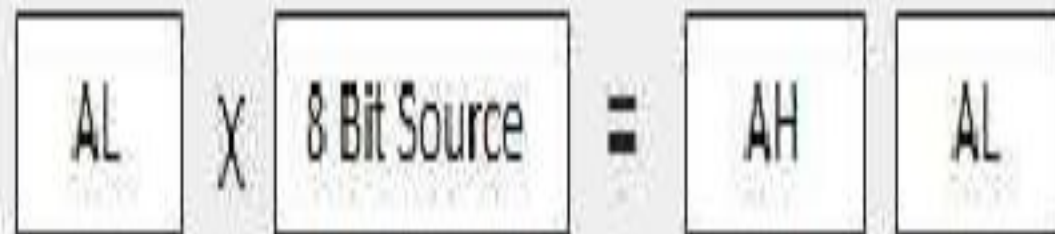
# The MUL/IMUL Instruction

- 2 instructions for multiplying binary data.
- MUL (Multiply) instruction handles unsigned data
- IMUL (Integer Multiply) handles signed data
- Both instructions affect the Carry and Overflow flag.
- SYNTAX:
  - MUL multiplier
  - IMUL multiplier
- Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands.

## SN Scenarios

### When two bytes are multiplied

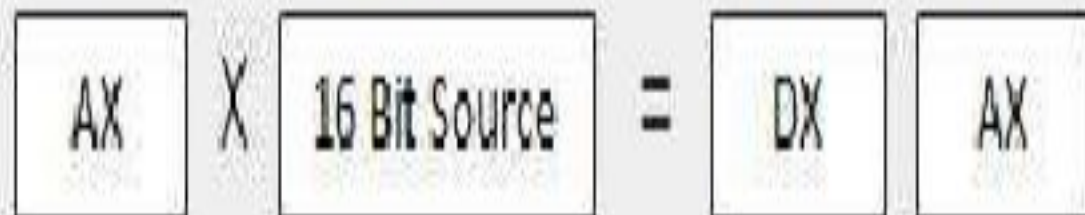
The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High order 8 bits of the product is stored in AH and the low order 8 bits are stored in AL.



## When two one-word values are multiplied

The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX.

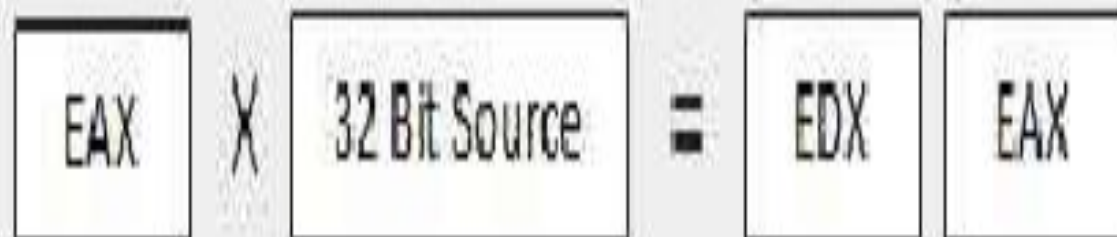
2 The resultant product is a double word, which will need two registers. The High order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX.





## When two doubleword values are multiplied

When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register.





# Summary



**When two bytes are multiplied**

The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High order 8 bits of the product is stored in AH and the low order 8 bits are stored in AL.

1

$$\boxed{\text{AL}} \times \boxed{\text{8 Bit Source}} = \boxed{\text{AH}} \boxed{\text{AL}}$$

**When two one-word values are multiplied**

The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX.

The resultant product is a double word, which will need two registers. The High order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX.

2

$$\boxed{\text{AX}} \times \boxed{\text{16 Bit Source}} = \boxed{\text{DX}} \boxed{\text{AX}}$$

**When two doubleword values are multiplied**

When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register.

3

$$\boxed{\text{EAX}} \times \boxed{\text{32 Bit Source}} = \boxed{\text{EDX}} \boxed{\text{EAX}}$$

# Division Instructions

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

# DIV (Unsigned Integer Divide)

- performs an unsigned division of the accumulator by the source operand.
- The dividend (the accumulator) is twice the size of the divisor (the source operand); the quotient and remainder have the same size as the divisor.
- Non-integral quotients are truncated to integers toward 0.

Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AX	AX
Word	DX:AX	AX	DX
Dword	EDX:EAX	EAX	EDX

- 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 dword division, the largest quotient is 4,294,967,295.

# The DIV/IDIV Instructions

- The division operation generates two elements - a **quotient** and a **remainder**.
- In case of multiplication, overflow does not occur because double-length registers are used to keep the product.
- However, in case of division, overflow may occur.
- The processor generates an interrupt if overflow occurs.
- DIV (Divide) instruction for unsigned data
- IDIV (Integer Divide) for signed data.



- SYNTAX:

  - DIV/IDIV divisor

- The dividend is in an accumulator.
- Both the instructions can work with 8-bit, 16-bit or 32-bit operands.
- The operation affects all six status flags.
- 3 cases

16 bit dividend

AX

Quotient

Remainder

=

AL

And

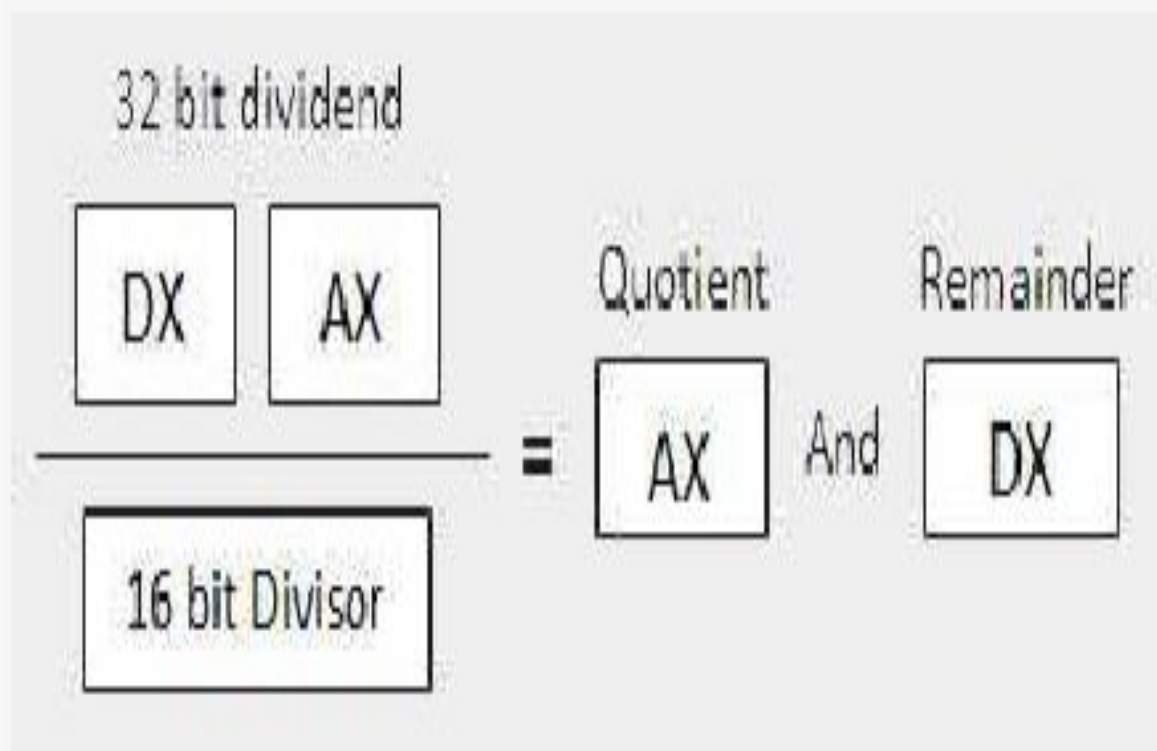
AH

8 bit Divisor

## When the divisor is 1 word

The dividend is assumed to be 32 bits long and in the DX:AX registers. The high order 16 bits are in DX and the low order 16 bits are in AX. After division, the 16 bit quotient goes to the AX register and the 16 bit remainder goes to the DX register.

2

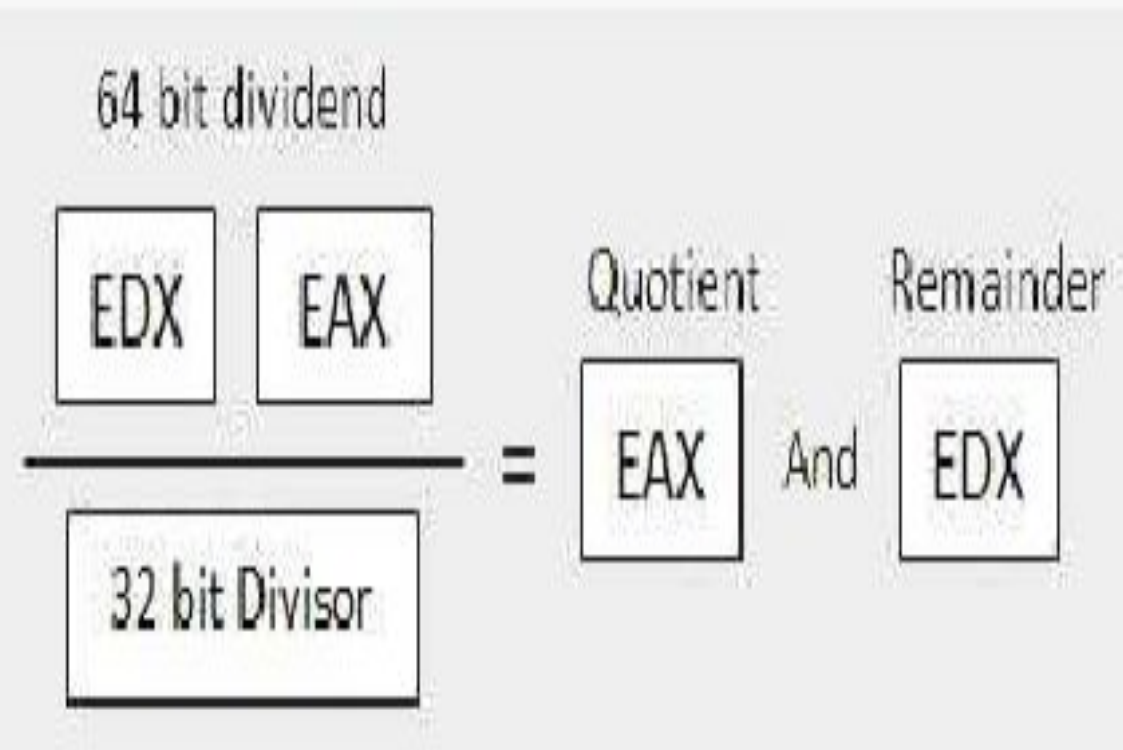




## When the divisor is doubleword

The dividend is assumed to be 64 bits long and in the EDX:EAX registers. The high order 32 bits are in EDX and the low order 32 bits are in EAX. After division, the 32 bit quotient goes to the EAX register and the 32 bit remainder goes to the EDX register.

3





# Decimal Arithmetic Instructions

- **Packed BCD Adjustment Instructions**

1. **DAA**
2. **DAS**

- **Unpacked BCD Adjustment Instructions**

1. **AAA**
2. **AAS**
3. **AAM**
4. **AAD**


# Logical Instructions

The group of logical instructions includes:

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

# Bit Test and Modify Instructions

- 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 as an offset from the low-order end of the operand.
- The value of the offset either may be given by an immediate byte in the instruction or may be contained in a general register.

- 
- 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.

## Bit Test and Modify Instruction

BT (Bit Test ) – reports the status of a bit in the operand by setting or clearing CF to match it. The operand under test may be either a register or a memory location. The second operand specifies which bit in the first operand to test.

### Example

```
BT    EAX, 5    ; test bit 5 of EAX
JC    foo      : jump if bit 5 was set
```

# Bit Test and Modify Instruction

BTC (Bit Test & Complement ) – It operates exactly like the BT, except that the bit being tested is inverted after the test is performed, and its condition is saved in CF.

## Example

```
BTC EAX, 9 ; test & invert bit 9  
JC foo : jump if bit used to be 1
```

# Bit Test and Modify Instruction

BTR (Bit Test & Reset ) – the BTR instruction operates exactly like the BTC instruction, except that it always clears the bit being tested.

Example

```
BTR    EAX , 0    ; test & clear bit 0  
JC    foo        : jump if it was set
```

BTS (Bit Test & Set ) – The BTS instruction operates exactly like the BTC instruction, except that it always sets the bit being tested.

Example

```
BTR    DWORD PTR DS:[840621], 3    ; test & set bit 3  
JC    foo        : jump if it was set
```



# Bit Test & Modify Instructions



# Bit Scan Instructions

- scan a word/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.

1. BSF (Bit Scan Forward) scans from low-order to high-order (starting from bit index zero).
2. 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).

# Logical Instructions

- The processor instruction set provides the instructions AND, OR, XOR, TEST and NOT Boolean logic, which tests, sets and clears the bits according to the need of the program.
- The format for these instructions:
  - AND :           AND operand1, operand2
  - OR:             OR operand1, operand2
  - XOR:           XOR operand1, operand2
  - TEST:          TEST operand1, operand2
  - NOT:           NOT operand1

# The AND Instruction

- The AND instruction is used for supporting logical expressions by performing bitwise AND operation.
- The bitwise AND operation returns 1, if the matching bits from both the operands are 1, otherwise it returns 0. For example:

Operand1: 0101

Operand2: 0011

-----

- After AND -> Operand1: 0001

- The AND operation can be used for clearing one or more bits.
- For example the BL register contains 0011 1010.
- If we need to clear the high order bits to zero, we AND it with 0FH.

AND BL, 0FH ; This sets BL to 0000 1010

# The OR Instruction

- The OR instruction is used for supporting logical expression by performing bitwise OR operation.
- The bitwise OR operator returns 1, if the matching bits from either or both operands are one.
- It returns 0, if both the bits are zero.
- For example,

Operand1: 0101

Operand2: 0011

-----

After OR -> Operand1: 0111

- The OR operation can be used for setting one or more bits.
- For example, let us assume the AL register contains 0011 1010, we need to set the four low order bits, we can OR it with a value 0000 1111, i.e., FH.  
OR BL, 0FH ; This sets BL to 0011 1111

# The XOR Instruction

- The XOR instruction implements the bitwise XOR operation.
- The XOR operation sets the resultant bit to 1, if and only if the bits from the operands are different.
- If the bits from the operands are same (both 0 or both 1), the resultant bit is cleared to 0.



- For example,

Operand1: 0101

Operand2: 0011

-----

After XOR -> Operand1: 0110

- XORing an operand with itself changes the operand to 0.
- This is used to clear a register.

XOR EAX, EAX

# The TEST Instruction

- The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand.
- So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

```
TEST AL, 01H  
JZ EVEN_NUMBER
```

# The NOT Instruction

- The NOT instruction implements the bitwise NOT operation.
- NOT operation reverses the bits in an operand.
- The operand could be either in a register or in the memory.
- For example,

Operand1: 0101 0011

After NOT -> Operand1: 1010 1100

# The CMP Instruction

- This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not.
- It does not disturb the destination or source operands.
- Non destructive subtraction
- It is used along with the conditional jump instruction for decision making.

# Assembly Conditions

- Conditional execution in assembly language is accomplished by several looping and branching instructions.
- These instructions can change the flow of control in a program.
- Conditional execution is observed in two scenarios:

## SN Conditional Instructions

### Unconditional jump

1 This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.

### Conditional jump

2 This is performed by a set of jump instructions  $j\langle\text{condition}\rangle$  depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.



- SYNTAX

CMP destination, source

- The CMP instruction compares two operands.
- It is generally used in conditional execution.
- CMP compares two numeric data fields.

# CMP.....

- The destination operand could be either in register or in memory.
- The source operand could be a constant (immediate) data, register or memory.

- EXAMPLE:

```
cmp dx, 00 ; Compare the DX value  
;with zero
```

```
je L7 ; If yes, then jump to label L7
```

```
.
```

```
.
```

```
L7: ...
```



- CMP is often used for comparing whether a counter value has reached the number of time a loop needs to be run.
- Consider the following typical condition:



# Unconditional Jump

- This is performed by the JMP instruction.
- Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction.
- Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.



- SYNTAX:

`jmp label`

- The `jmp` instruction provides a label name where the flow of control is transferred immediately.

# EXAMPLE

The following code snippet illustrates the JMP instruction:

```
MOV  AX, 00    ; Initializing AX to 0
MOV  BX, 00    ; Initializing BX to 0
MOV  CX, 01    ; Initializing CX to 1
L20:
ADD  AX, 01    ; Increment AX
ADD  BX, AX    ; Add AX to BX
SHL  CX, 1     ; shift left CX, this in turn doubles the CX value
JMP  L20       ; repeats the statements
```

# Conditional Jump

- If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction.
- There are numerous conditional jump instructions, depending upon the condition and data.

# Conditional Jump

- Following are the conditional jump instructions used on signed data used for arithmetic operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

- Following are the conditional jump instructions used on unsigned data used for logical operations:



JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF



- The following conditional jump instructions have special uses and check the value of flags:

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF



- The syntax for the J<condition> set of instructions:
- Example,

```
CMP      AL, BL
JE       EQUAL
CMP      AL, BH
JE       EQUAL
CMP      AL, CL
JE       EQUAL
NON EQUAL: ...
EQUAL:   ...
```

# Example

- Write a program to display the largest of three variables. [The variables need to be double-digit variables. The three variables num1, num2 and num3 have values 47, 22 and 31 respectively]

# Assembly Loops

- The JMP instruction can be used for implementing loops.
- Example, the following code snippet can be used for executing the loop-body 10 times.



# Assembly Loops

- The processor instruction set includes a group of loop instructions for implementing iteration.
- The basic LOOP instruction has the following syntax:

loop label

Where, *label* is the target label that identifies the target instruction as in the jump instructions.

- The loop instruction assumes that the **ECX register contains the loop count**.
- When the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.

# Assembly Loops

- The above code snippet could be written as:

```
mov ECX, 10  
l1:  
<loop body>  
loop l1
```

# Example

- Write a program to print the number 1 to 9 on the screen.

# Assembly Numbers

- Numerical data is generally represented in binary system.
- Arithmetic instructions operate on binary data.
- When numbers are displayed on screen or entered from keyboard, they are in ASCII form.
- Common Practice: Converting input data in ASCII form to binary for arithmetic calculations and converting the result back to binary.



# Decimal Number Representation

- Decimal numbers can be represented in two forms:
  1. ASCII form
  2. BCD or Binary Coded Decimal form

# ASCII Representation

- In ASCII representation, decimal numbers are stored as string of ASCII characters.
- For example, the decimal value 1234 is stored as:

31	32	33	34H
----	----	----	-----

Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on.

● There are the following four instructions for processing numbers in ASCII representation:

1. **AAA** - ASCII Adjust After Addition
2. **AAS** - ASCII Adjust After Subtraction
3. **AAM** - ASCII Adjust After Multiplication
4. **AAD** - ASCII Adjust **Before** Division

● *These instructions do not take any operands and assumes the required operand to be in the AL register.*

- Use AAA only after executing the form of an add instruction that stores a two-BCD-digit byte result in the AL register.
- AAA then adjusts AL to contain the correct decimal result.
- The top nibble of AL is set to 0.
- To convert AL to an ASCII result, follow the AAA instruction with:  
or %AL, 0x30

# How AAA handles a carry

## Carry

## Action

Decimal Carry

AH + 1; CF and AF set to 1

No Decimal Carry

AH unchanged; CF and AF cleared to 0

# BCD Representation

- There are two types of BCD representation:
  1. Unpacked BCD representation
  2. Packed BCD representation
- In unpacked BCD representation, each byte stores the binary equivalent of a decimal digit.
- For example, the number 1234 is stored as:



# Unpacked BCD

- There are two instructions for processing these numbers:
  1. AAM - ASCII Adjust After Multiplication
  2. AAD - ASCII Adjust Before Division
- The four ASCII adjust instructions, AAA, AAS, AAM and AAD can also be used with unpacked BCD representation.

# Packed BCD

- In packed BCD representation, each digit is stored using four bits.
- Two decimal digits are packed into a byte.
- For example, the number 1234 is stored as:

12

34H

- There are two instructions for processing these numbers:
  1. DAA - Decimal Adjust After Addition
  2. DAS - decimal Adjust After Subtraction
- There is no support for multiplication and division in packed BCD representation.



# Assembly Strings

- We specify the length of the string by either of the two ways:
  1. Explicitly storing string length
  2. Using a sentinel character
- We can store the string length explicitly by using the \$ location counter symbol, that represents the current value of the location counter.

# Example

```
msg db 'Hello, world!', 0xa ; string
```

```
len equ $ - msg ;length of string
```

- \$ points to the byte after the last character of the string variable *msg*.
- Therefore, ***\$-msg*** gives the length of the string.
- We can also write

```
msg db 'Hello world!', 0xa ; string
```

```
len equ 13 ;length of string
```


- Alternatively, we can store strings with a trailing sentinel character to delimit a string instead of storing the string length explicitly.
- The sentinel character should be a special character that does not appear within a string.
- For example:  
message DB 'HELLO WORLD!', 0

# String Instructions

- Each string instruction may require a source operand, a destination operand, or both.
- For 32-bit segments, string instructions use ESI and EDI registers to point to the source and destination operands, respectively.
- For 16-bit segments, however, the SI and the DI registers are used to point to the source and destination respectively.

# String Instructions

- There are five basic instructions for processing strings. They are:
  1. **MOVS** - This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
  2. **LODS** - This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
  3. **STOS** - This instruction stores data from register (AL, AX, or EAX) to memory.
  4. **CMPS** - This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
  5. **SCAS** - This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.



Each of the above instruction has a byte, word and doubleword version and string instructions can be repeated by using a repetition prefix.

# String Instructions

- These instructions use the ES:DI and DS:SI pair of registers, where DI and SI registers contain valid offset addresses that refers to bytes stored in memory.
- SI is normally associated with DS (data segment) and DI is always associated with ES (extra segment).
- The DS:SI (or ESI) and ES:DI (or EDI) registers point to the source and destination operands respectively.
- The source operand is assumed to be at DS:SI (or ESI) and the destination operand at ES:DI (or EDI) in memory.
- For 16-bit addresses the SI and DI registers are used and for 32-bit addresses the ESI and EDI registers are used.

The following table provides various versions of string instructions and the assumed space of the operands.

Basic Instruction	Operands at	Byte Operation	Word Operation	Double word Operation
MOVS	ES:DI, DS: SI	MOVSB	MOVSW	MOVSD
LODS	AX, DS:SI	LODSB	LODSW	LODSD
STOS	ES:DI, AX	STOSB	STOSW	STOSD
CMPS	DS:SI, ES: DI	CMPSB	CMPSW	CMPSD
SCAS	ES:DI, AX	SCASB	SCASW	SCASD



# MOVS

- The MOVS instruction is used to copy a data item (byte, word or doubleword) from the source string to the destination string.
- The source string is pointed by DS:SI and the destination string is pointed by ES:DI.



**LODS**

# STOS

- The STOS instruction copies the data item from AL (for bytes - STOSB), AX (for words - STOSW) or EAX (for doublewords - STOSD) to the destination string, pointed to by ES:DI in memory.

# CMPS

- The CMPS instruction compares two strings.
- This instruction compares two data items of one byte, word or doubleword, pointed to by the DS:SI and ES:DI registers and sets the flags accordingly.
- Use of the conditional jump instructions along with this instruction also possible.

# SCAS

- The SCAS instruction is used for searching a particular character or set of characters in a string.
- The data item to be searched should be in AL (for SCASB), AX (for SCASW) or EAX (for SCASD) registers. The string to be searched should be in memory and pointed by the ES:DI (or EDI) register.

# Repetition Prefixes

- The REP prefix, when set before a string instruction, for example - REP MOVSB, causes repetition of the instruction based on a counter placed at the CX register.
- REP executes the instruction, decreases CX by 1, and checks whether CX is zero. It repeats the instruction processing until CX is zero.
- The Direction Flag (DF) determines the direction of the operation.
- Use CLD (Clear Direction Flag, DF = 0) to make the operation left to right.
- Use STD (Set Direction Flag, DF = 1) to make the operation right to left.

# REP Variants

- The REP prefix also has the following variations:
  1. REP: it is the unconditional repeat. It repeats the operation until CX is zero.
  2. REPE or REPZ: It is conditional repeat. It repeats the operation while the zero flag indicate equal/zero. It stops when the ZF indicates not equal/zero or when CX is zero.
  3. REPNE or REPNZ: It is also conditional repeat. It repeats the operation while the zero flag indicate not equal/not zero. It stops when the ZF indicates equal/zero or when CX is decremented to zero.

# Assembly Arrays

- To define a one dimensional array
- Use of the data definition directives
- To define a one dimensional array of numbers:  
`NUMBERS DW 34, 45, 56, 67, 75, 89`
- This allocates  $2 \times 6 = 12$  bytes of consecutive memory space.
- The symbolic address of the first number will be `NUMBERS` and that of the second number will be `NUMBERS + 2` and so on.



# Define An Array

- We can define an array named *ARR* of size 8, and initialize all the values with zero, as:

```
ARR DW 0
```

```
DW 0
```

```
DW 0
```

```
DW 0
```

```
DW 0
```

```
DW 0
```

```
DW 0
```

```
DW 0
```

- Which, can be abbreviated as:



*Any Shortcut??????*



ARR TIMES 8 DW 0

- Restriction: The TIMES directive can also be used for multiple initializations to the same value

# Assembly Procedures

- Procedures are identified by a name.
- Following this name, the body of the procedure is described, which perform a well-defined job.
- End of the procedure is indicated by a return statement.
- Syntax to define a procedure:



- The procedure is called from another function by using the CALL instruction.
- The CALL instruction should have the name of the called procedure as argument :  
CALL proc\_name
- The called procedure returns the control to the calling procedure by using the RET instruction.

# Stacks Data Structure

- An array-like data structure in the memory
- Data can be stored and removed
- 'top' of the stack
- PUSH and POP operations
- LIFO data structure, i.e., the data stored first is retrieved last.
- Assembly language provides two instructions for stack operations: PUSH and POP.
- Syntax:  
PUSH operand  
POP address/register

# Shift and Rotate Instructions

- The shift and rotate instructions reposition the bits within the specified operand.
- These instructions fall into the following classes:
  - Shift instructions
  - Double shift instructions
  - 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.
- To specify the count in one of three ways:
  1. To specify the count implicitly as a single shift
  2. To specify the count as an immediate value
  3. To specify the count as the value contained in CL. This 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 the content of OF is always undefined.
- The shift instructions provide a convenient way to accomplish division or multiplication by binary power.

*Note : division of signed numbers by shifting right is not the same of division performed by the IDIV instruction.*

# SAL & SHL

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

# SHL

- Synonym SAL
- shifts the bits in the register or memory operand to the left by the specified number of bit positions
- CF receives the last bit shifted out of the left of the operand.
- SHL shifts in zeros to fill the vacated bit locations.
- These instructions operate on byte, word, and doubleword operands.



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

# SHR

- Shift Logical Right
- Shifts the destination byte, word, or doubleword operand right by one or by the number of bits specified in the count operand
- Count: 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.
- SHR shifts the bits of the register or memory operand to the right by the specified number of bit positions.
- CF receives the last bit shifted out of the right of the operand.
- SHRD shifts in ones to fill the specified bit locations

# Shift and Rotate Instruction

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

Shift Logical Right

# SAR

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

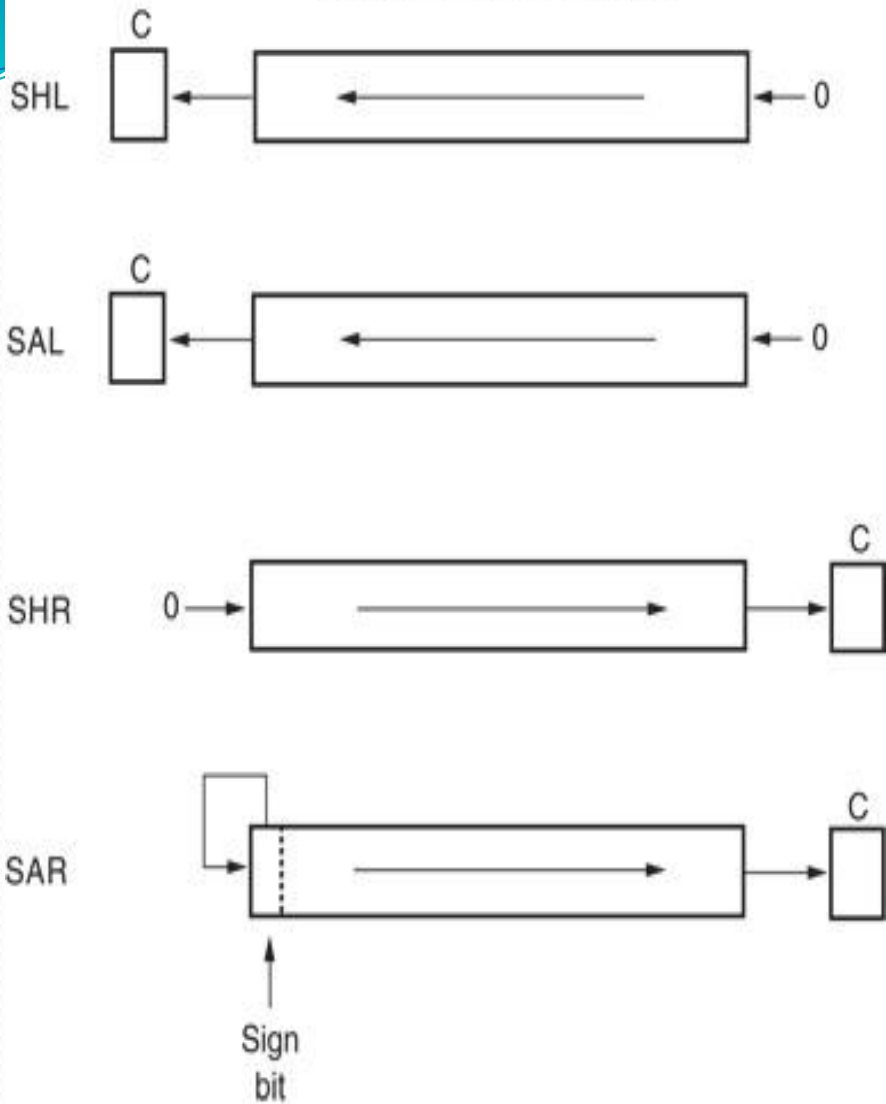
# Shift and Rotate Instruction

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

Shift Arithmetic Right

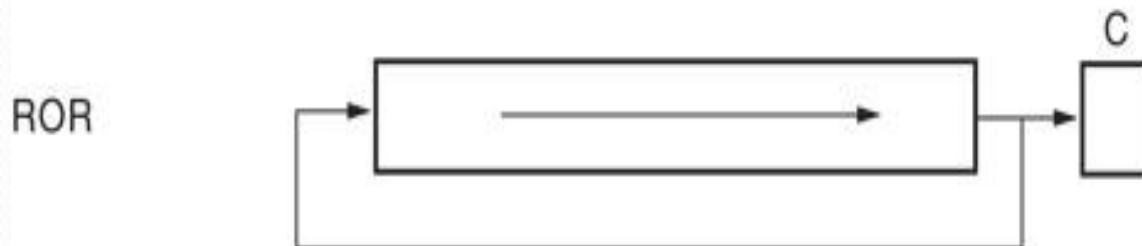
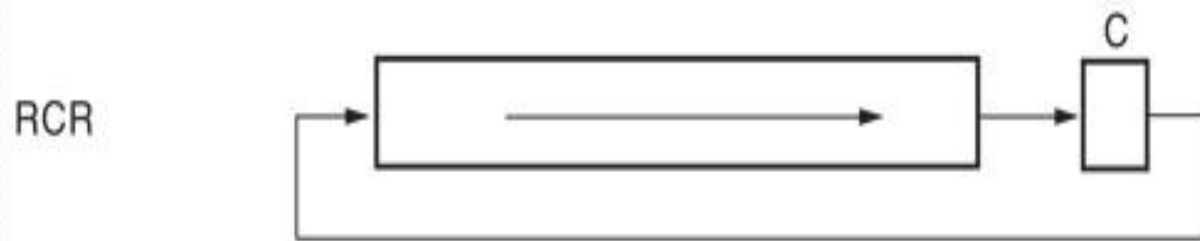
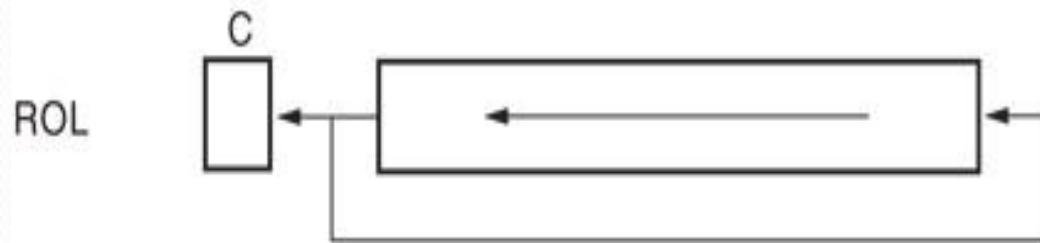
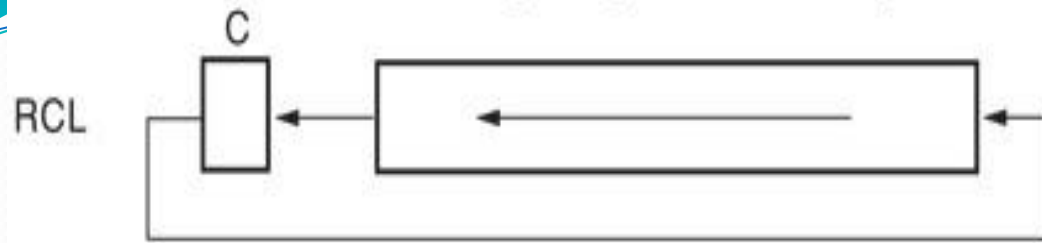


Target register or memory



- logical shifts move 0 in the rightmost bit for a logical left shift;
- 0 to the leftmost bit position for a logical right shift
- arithmetic right shift copies the sign-bit through the number
- logical right shift copies a 0 through the number.


Target register or memory



# Double-Shift Instructions

- These instructions provide the basic operations needed to implement operations on long unaligned bit strings.
- The double shifts operate either on word or doubleword operands, as follows:
  1. Taking two word operands as input and producing a one-word output.
  2. Taking two doubleword operands as input and producing a doubleword output.

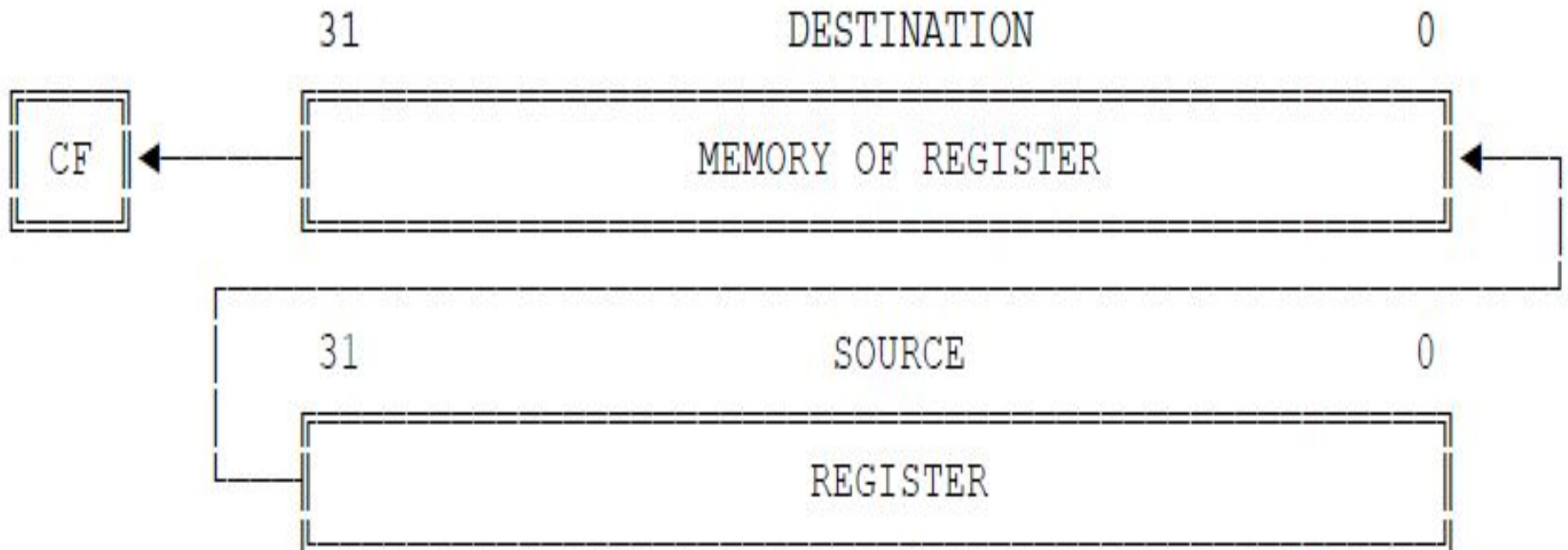
- Of the two input operands, one may either be in a general register or in memory
- the other may only be in a general register.
- The results replace the memory or register operand.
- The number of bits to be shifted is specified either in the CL register or in an immediate byte of the instruction.

- 
- Bits are shifted from the register operand into the memory or register operand.
  - CF is set to the value of the last bit shifted out of the destination operand.
  - SF, ZF, and PF are set according to the value of the result.
  - OF and AF are left undefined.

# SHLD

- Shift Left Double
- shifts bits of the R/M field to the left, while shifting high-order bits from the Reg field into the R/M field on the right
- The result is stored back into the R/M operand.
- The Reg field is not modified.

# Shift Left Double

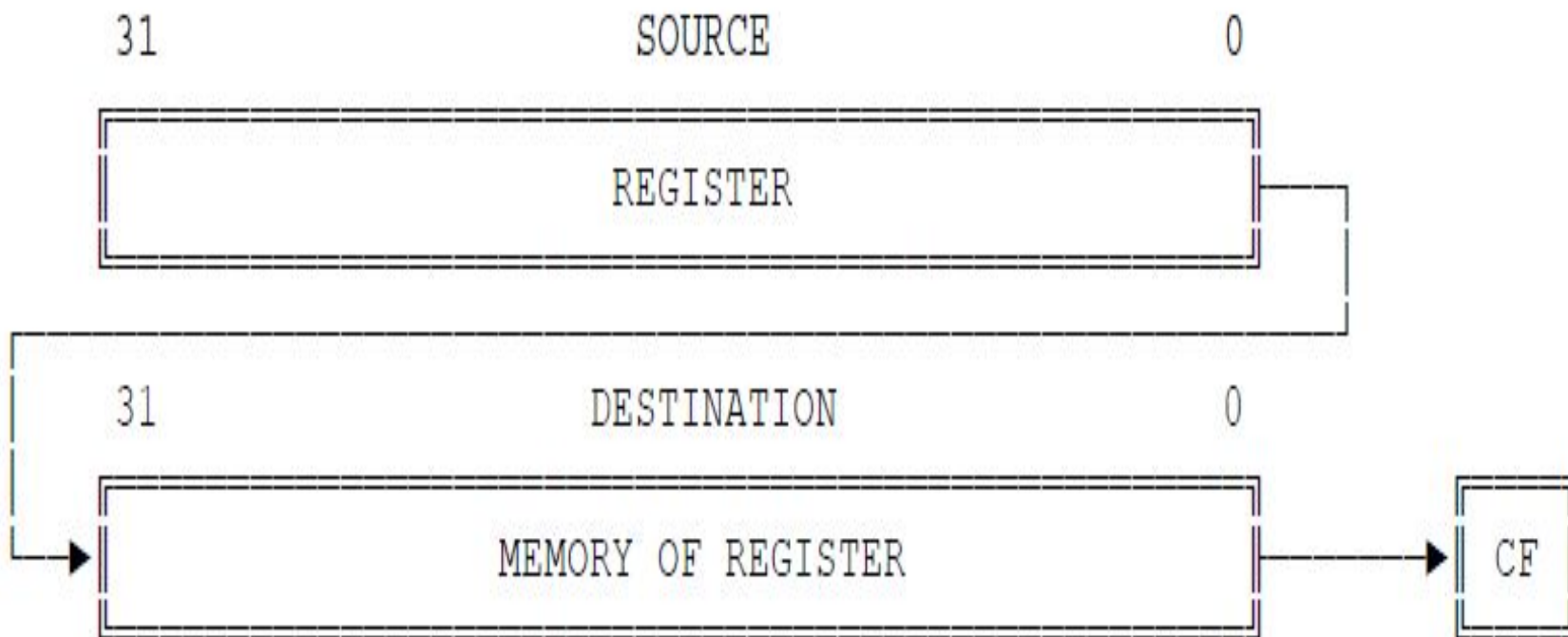


# SHRD

- Shift Right Double
- shifts bits of the R/M field to the right, while shifting low-order bits from the Reg field into the R/M field on the left
- The result is stored back into the R/M operand.
- The Reg field is not modified.




# Shift Right Double



# Rotate Instructions

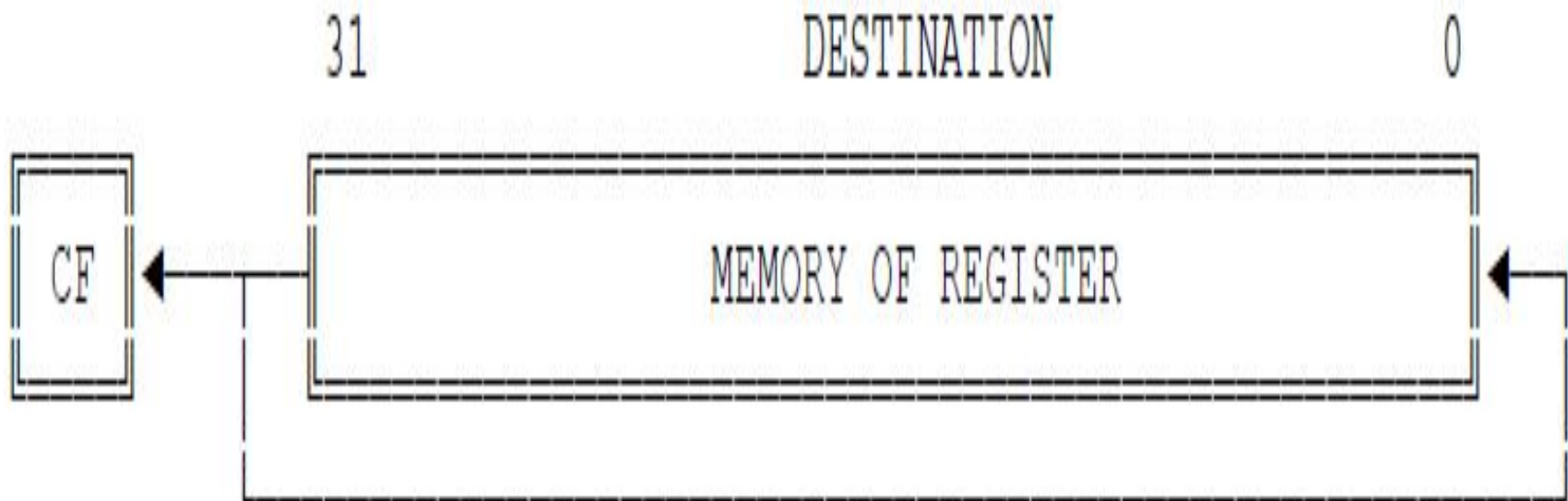
- Allow bits in bytes, words, and doublewords to be rotated
- Bits rotated out of an operand are not lost as in a shift, but are "circled" back into the other "end" of the operand.
- Rotates affect only the carry and overflow flags.
- CF may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated and then tested by a conditional jump instruction (JC /JNC).
- CF always contains the value of the last bit rotated out, even if the instruction does not use this bit as an extension of the rotated operand.

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

# ROL

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

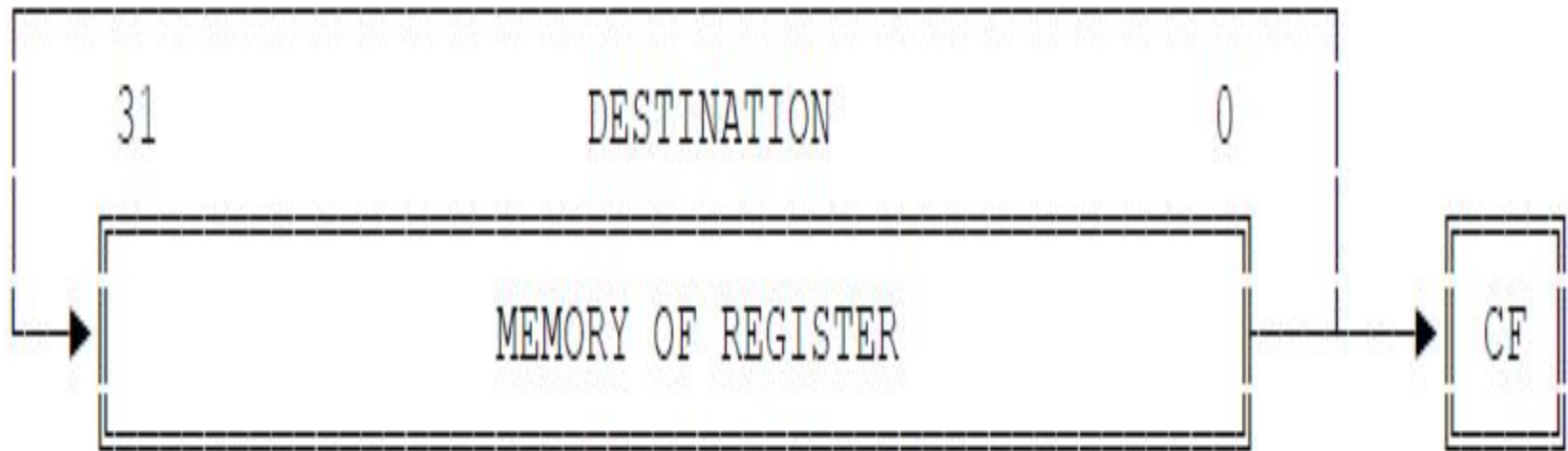
# ROL



# ROR

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

# ROR

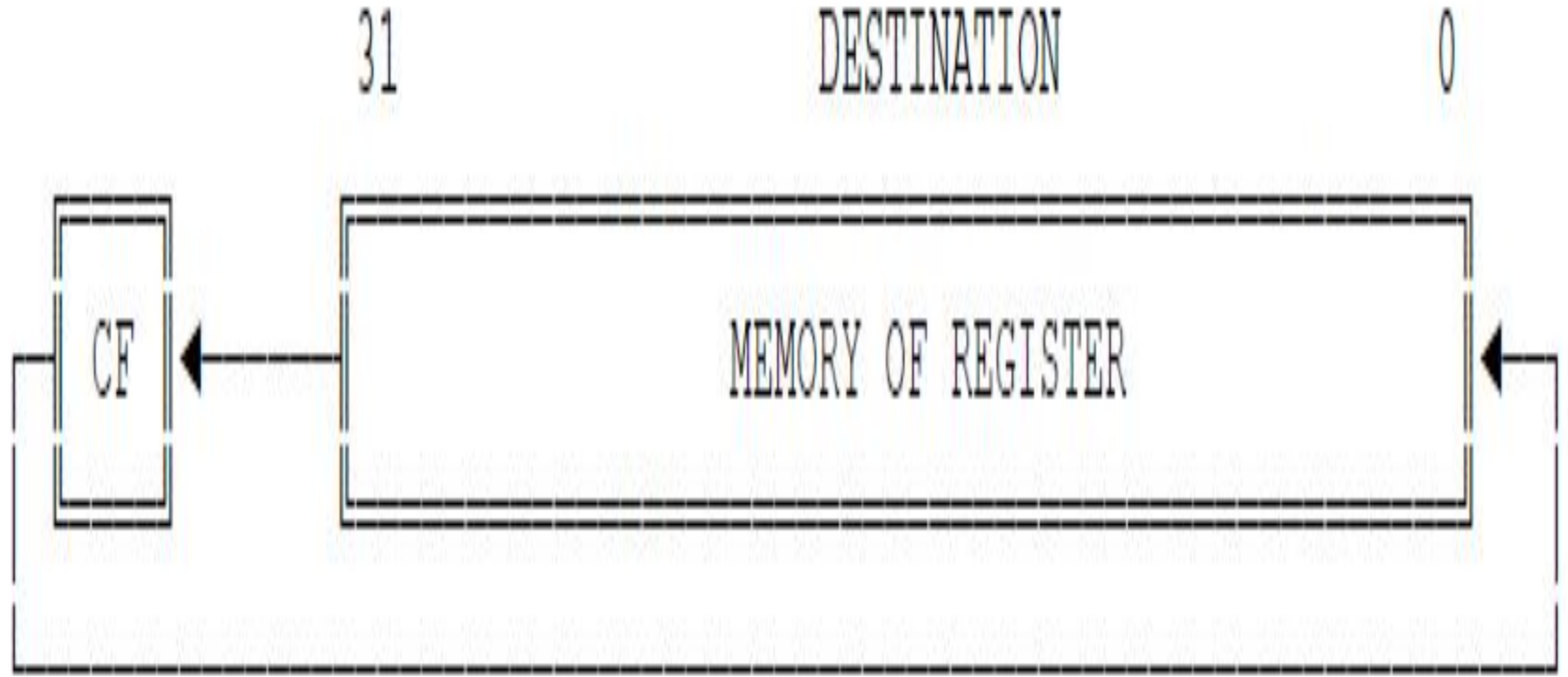


# RCL (Rotate Through Carry Left)

- rotates bits in the byte, word, or doubleword destination operand left by one or by the number of bits specified in the count operand (an immediate value or the value contained in CL)
- differs from ROL
- treats CF as a high-order one-bit extension of the destination operand
- Each high-order bit that exits from the left side of the operand moves to CF before it returns to the operand as the low-order bit on the next rotation



# RCL



**AAA ----- ASCII Adjust after Addition.**

**DAA ----- Decimal Adjust AL after Addition**

Corrects result in AH and AL after addition when working with BCD values.

```
Mov AX,0009h
```

```
Mov BX ,0006h
```

```
Add AX, BX          ; result AX =0fH ( Ah =00 ,Al =0f)
```

```
AAA (DAA)           ; now Ax =0105 h ( Ah =01 ,Al =05)
```

**AAD- ASCII Adjust before Division.**

Prepares two BCD values for division.

```
Mov BX,0003h
```

```
Mov AX, 0105h       ; now Ax =0105 h ( Ah =01 ,Al =05)
```

```
AAD                ; result AX =0fH ( Ah =00 ,Al =0f)
```

```
Div BX              ; AX/BX
```

## **AAM -----ASCII Adjust after Multiplication.**

Corrects the result of multiplication of two BCD values.

```
Mov AX,0003h
```

```
Mov BX ,0005h
```

```
MUL AX, BX ; result AX =0fH ( Ah =00 ,Al =0f)
```

```
AAM ; now Ax =0105 h ( Ah =01 ,Al =05)
```

## **AAS -----ASCII Adjust after Subtraction.**

### **DAS ---- Decimal Adjust AL after Subtraction**

Corrects result in AH and AL after subtraction when working with BCD values.

```
MOV AX, 02FFh ; AH = 02, AL = 0FFh
```

```
AAS ; AH = 01, AL = 09
```

## ARPL -- Adjust RPL Field of Selector

ARPL DEST , SRC

```
IF RPL bits(0,1) of DEST < RPL bits(0,1) of SRC
THEN ZF := 1;
RPL bits(0,1) of DEST := RPL bits(0,1) of SRC;
ELSE ZF := 0;
ENF IF;
```

The ARPL instruction has two operands. The first operand is a 16-bit memory variable or word register that contains the value of a selector. The second operand is a word register. If the RPL field ("requested privilege level"--bottom two bits) of the first operand is less than the RPL field of the second operand, the zero flag is set to 1 and the RPL field of the first operand is increased to match the second operand. Otherwise, the zero flag is set to 0 and no change is made to the first operand. ARPL appears in operating system software, not in application programs. It is used to guarantee that a selector parameter to a subroutine does not request more privilege than the caller is allowed. The second operand of ARPL is normally a register that contains the CS selector value of the caller.

# VERR, VERW -- Verify a Segment for Reading or Writing

VERR eax ; Set ZF=1 if segment can be read,  
selector in eax

VERW eax ;Set ZF=1 if segment can be written,  
selector in eax

## **BOUND -- Check Array Index Against Bounds**

**Bound eax,ffffff1h**

If `eax > fffffff1h` then it call interrupt 5

BOUND ensures that a signed array index is within the limits specified by a block of memory consisting of an upper and a lower bound. Each bound uses one word for an operand-size attribute of 16 bits and a doubleword for an operand-size attribute of 32 bits. The first operand (a register) must be greater than or equal to the first bound in memory (lower bound), and less than or equal to the second bound in memory (upper bound). If the register is not within bounds, an Interrupt 5 occurs; the return EIP points to the BOUND instruction. The bounds limit data structure is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array.

# I/O Port data transfer

IN Input from Port

OUT Output to Port

- `IN al, DX` Input from port DX into AL
- `Out al, DX` output from AL to port DX
- **INS/INSB/INSW/INSD -- Input from Port to String**
- `INS al, DX` Input byte from port  
DX into AL
- `INS ax, DX` Input word from port  
DX into AX

# Flag manipulation instruction

STC -- Set Carry Flag

STD -- Set Direction Flag

STI -- Set Interrupt Flag

CLC -- Clear Carry Flag

CLD -- Clear Direction Flag

CLI -- Clear Interrupt Flag

CMC -- Complement Carry Flag

SAHF -- Store AH into Flags

LAHF --- Load Flags into AH Register