Unit 1-Introduction to 80386

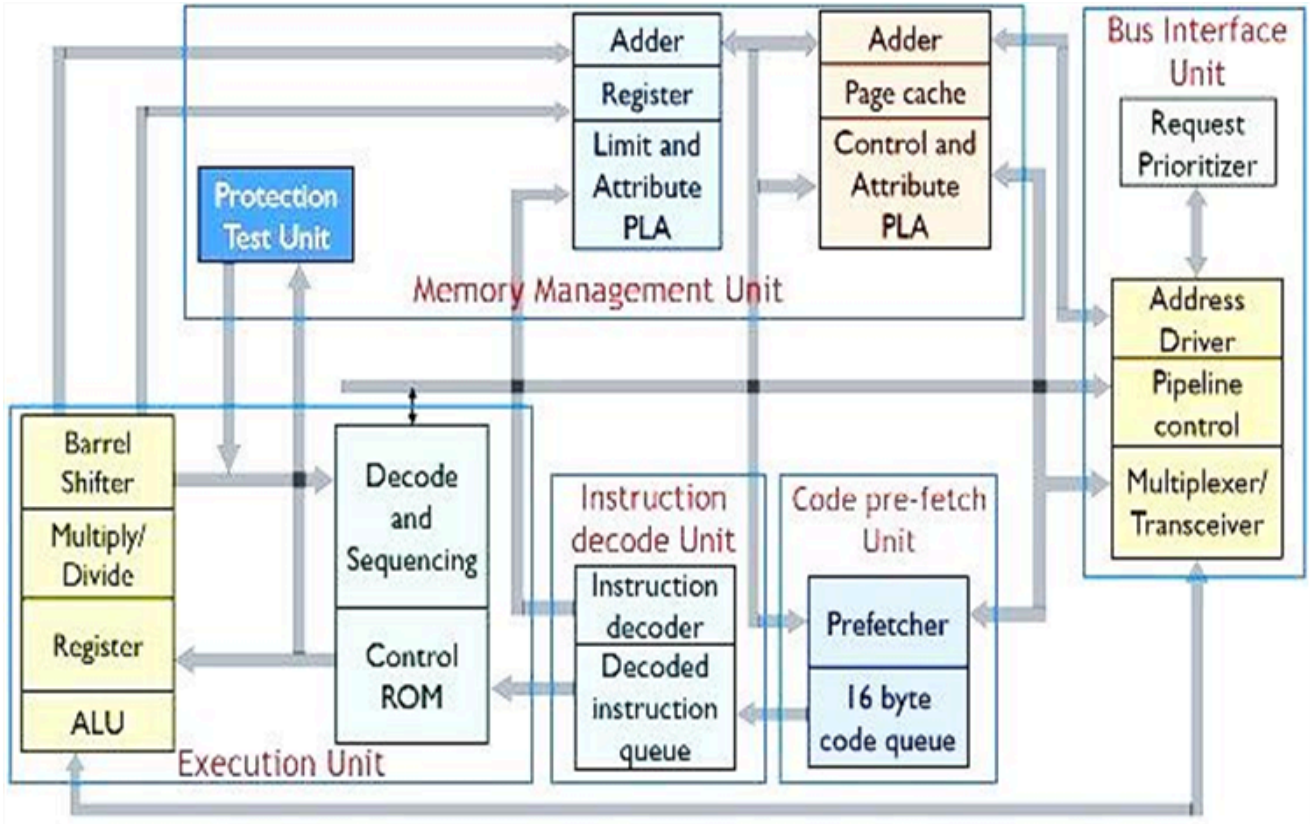
Brief History of Intel Processors, 80386 DX Features and Architecture, Programmers Model, Operating modes, Addressing modes and data types.

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.

Two versions of 80386 are commonly available:
1) 80386DX 2) 80386SX

Sr.No	80386DX (DX means Double word external)	80386SX (SX means Single word external)
1	32 bit address bus	24 bit address bus
2	32bit data bus	16 bit data bus
3	Packaged in 132 pin ceramic pin grid array(PGA)	100 pin flat package
4	Address 4GB of memory	16 MB of memory

80386 DX Architecture :



Basically, it has 5 functional units which are as follows:

- **Bus Interface Unit**
- **Code Fetch Unit**
- **Instruction Decode Unit**
- **Execution Unit**
- **Memory Management Unit**

1. Bus Interface Unit :

1. The bus interface unit or *BIU* holds a 32-bit bidirectional data bus as well as a 32-bit address bus.

2. Whenever a need for instruction or a data fetch is generated by the system then the BIU generates signals (according to the priority) for activating the data and address bus in order to fetch the data from the desired address.
3. The BIU connects the peripheral devices through the memory unit and also controls the interfacing of external buses with the coprocessors.

2. Code Prefetch Unit :

1. This unit fetches the instructions stored in the memory by making use of system buses.
2. Whenever the system generates a need for instruction then the code prefetch unit fetches that instruction from the memory and stores it in a 16-byte prefetch queue.
3. To speed up the operation this unit fetches the instructions in advance and the queue stores these instructions.
4. As this unit fetches one double word in a single access. So, in such a case, it is not necessary that each time only a single instruction will be fetched, as the fetched instruction can be parts of two different instructions.

3. Instruction Decode Unit :

1. We know that instructions in the memory are stored in the form of bits.
2. This unit decodes the instructions stored in the prefetch queue.
3. Basically the decoder changes the machine language code into assembly language and transfers it to the processor for further execution.

4. Execution Unit :

1. The decoded instructions are stored in the decoded instruction queue. These instructions are provided to the execution unit in order to execute the instructions.
2. The execution unit controls the execution of the decoded instructions.
3. This unit has a 32-bit ALU, that performs the operation over 32-bit data in one cycle. Also, it consists of 8 general purpose as well as 8 special purpose

registers.These are used for data handling and calculation of offset addresses.

5. Memory Management Unit :

This unit has two separate units within it. These are

- Segmentation Unit and
- Paging Unit

Segmentation unit:

1. Segmentation unit has the ability to convert the logical address into the linear address at the time of executing an instruction.
2. The segmentation unit compares the effective address for the length limit specified in the segment descriptor.
3. The segmentation unit adds the segment base and effective address to generate a linear address.
4. It gives 4 level protection to the data or code present in the memory. Every information in the memory is assigned a privilege level from PL0 to PL3. Here, PL0 holds the highest priority and PL3 holds the lowest priority.

Paging Unit:

1. The paging unit changes the linear address into a physical address.
2. The paging unit supports multitasking. This is so because the physical memory is not required to hold the whole segment of any task.
3. Despite this, only that part of the segment which is needed to be currently executed must be stored in that memory whose physical address is calculated by the paging unit.
4. This resultantly reduces the memory requirement and hence this frees the memory for other tasks. Thus by this we get an effective way for managing the memory to support multitasking.

Features of 80386 :

A. It is a **32-bit** microprocessor. Thus has a **32-bit ALU**.

B. 80386 has a **32-bit data bus**.

C. It holds an **address bus of 32 bits**.

D. It supports physical memory addressability of **4 GB** and virtual memory addressability of **64 TB**.

E. 80386 supports a variety of operating clock frequencies, which are **16 MHz, 20 MHz, 25 MHz, and 33 MHz.**

F. It offers a 3 stage pipeline: **fetch, decode and execute**. As it supports simultaneous fetching, decoding, and execution inside the system.

G. Integrated Memory Management Unit

H. Segmentation and Paging support

I. 4 Levels of Protection

J. Fully Compatible with 80286

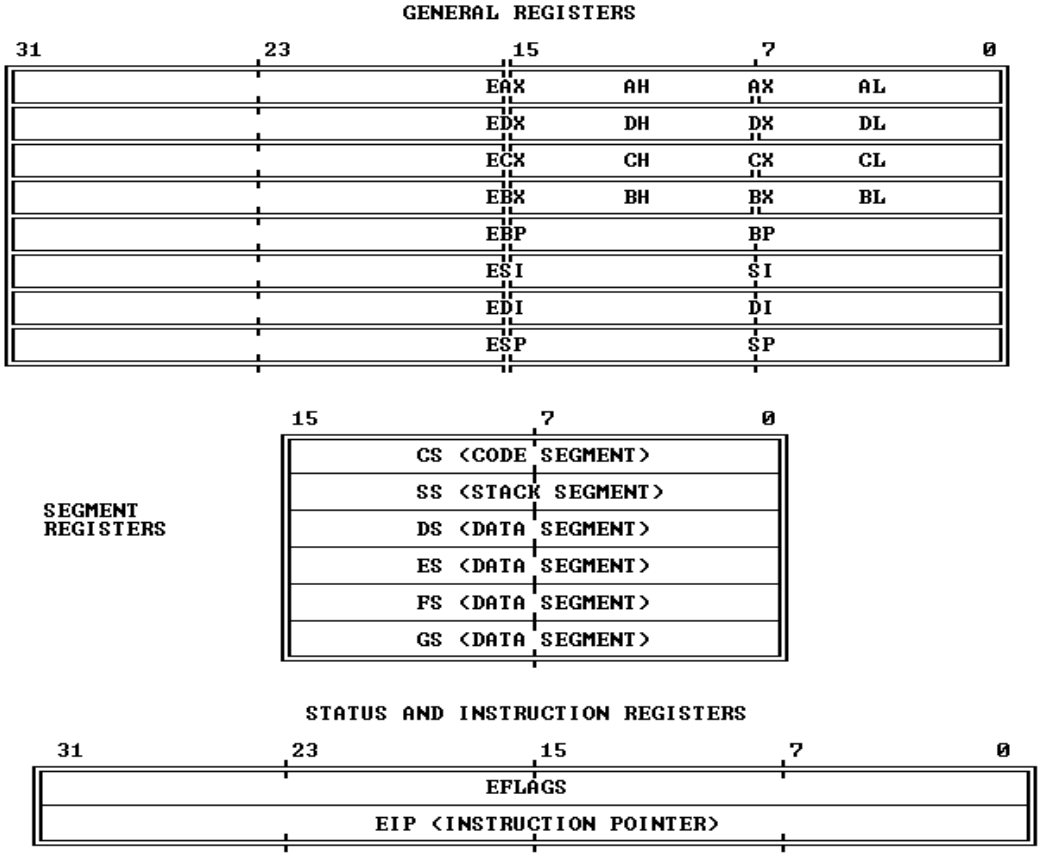
K. Operates in **Real, Protected and Virtual 8086 mode**

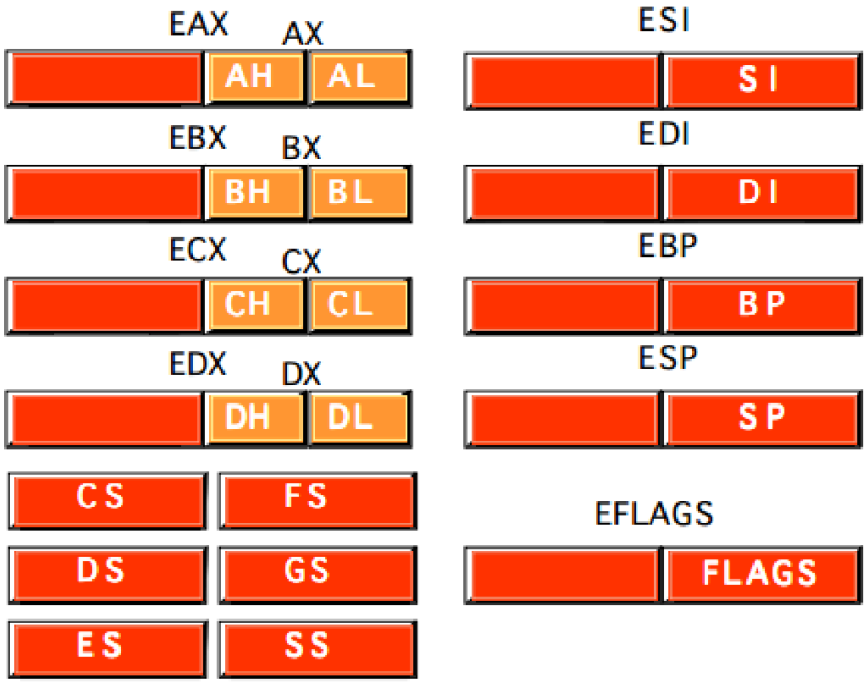
L. It can operate on **17 data types** like bits, byte, word, double word, Quadword etc

M. It allows user to **switch between different OS** such as DOS and UNIX

Programmers Model :

Register Organization :





Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

General Purpose Registers:

1. The general registers of the 80386 are the **32-bit** registers **EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI**. These registers are used interchangeably to contain the operands of logical and arithmetic operations.
2. The low-order word of each of these eight registers has a separate name and can be treated as a unit. This feature is useful for handling **16-bit** data. The word registers are named **AX, BX, CX, DX, BP, SP, SI, and DI.**
3. Each byte of the 16-bit registers further separate to handling characters and other **8-bit** data items. The byte registers are named **AH, BH, CH, and DH (high bytes); and AL, BL, CL, and DL (low bytes).**

Segment Registers:

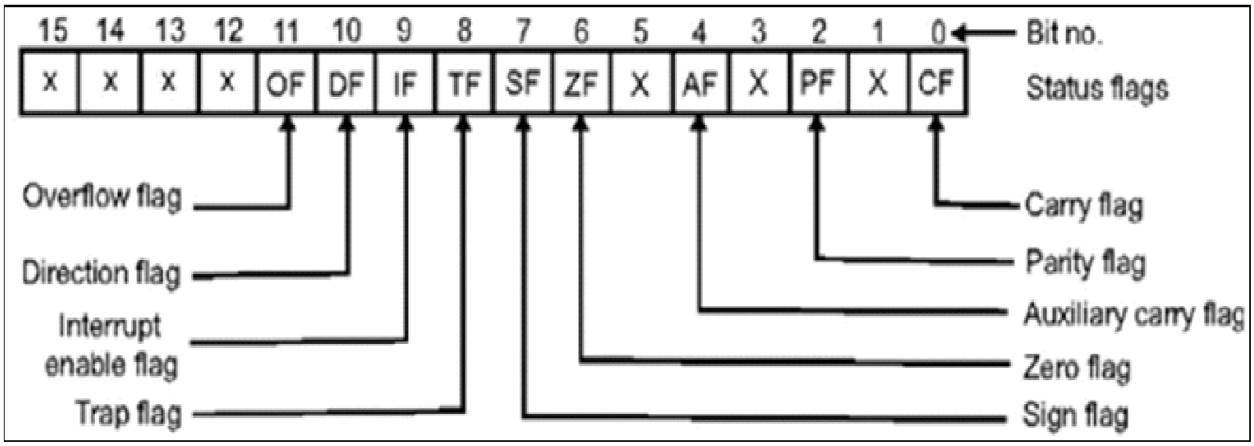
1. The segment registers **CS, DS, SS, ES, FS, and GS** are used to identify these six current segments.
2. **CS(code segment):** The segment containing the currently executing sequence of instructions is known as the current code segment.
3. **SS(Stack segment):** All stack operations use the SS register to locate the stack. Subroutine calls, parameters, and procedure activation records usually require that a region of memory be allocated for a stack.
4. **DS(Data segment):** The **DS, ES, FS, and GS** registers allow the specification of four data segments, each addressable by the currently executing program.

Instruction Pointer:

1. 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.
2. The instruction pointer is not directly visible to the programmer; it is controlled implicitly by control-transfer instructions, interrupts, and exceptions.

   The 32 bit Instruction pointer (EIP)

Flag Register of 80386



These flags can be categorized into three different groups.

1. **Status flags:** These flags reflect the state of a particular program.
2. **Control flags:** These flags directly affect the operation of a few instructions.

3. **System flags:** These flags reflect the current status of the machine and are usually used by the operating systems than by application programs.

Status flags :

1. **CF (Carry flag)**: This bit is set by arithmetic instructions that generate either a carry or a borrow. This bit can also be set, cleared, or inverted with the STC, CLC, or CMC instructions.

   Carry flag is also used in shift and rotate instructions to contain the bit shifted or rotated out of the register.

2. **PF (Parity flag):** The parity bit is set by most instructions if the least significant 8-bit of the result contains an even number of one's.

3. **AF (Auxiliary carry flag):** This bit is set when there is a carry or borrow after a nibble addition or subtraction, respectively. The programmer can't access this bit directly, but this bit is internally used for BCD arithmetic.

4. **ZF (Zero flag):** Zero flags is set to 1 if the result of an operation is zero.

5. **SF (Sign flag):** The signed numbers are represented by a combination of sign and magnitude The Most Significant Bit (MSB) indicates a sign of the number. For a negative number, MSB is 1. The sign flag is set to 1 if the result of an operation is negative (MSB=1).

6. **OF (Overflow flag):** In 2's complemented arithmetic, the most significant bit is used to represent a sign, and the remaining bits are used to represent the magnitude of a number. This flag is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for an 8-bit number) to represent it.

Control Flags:

DF (Direction flag): The direction flag controls the direction of string operations. When the D flag is cleared these operations process strings from low memory up to high memory.

This means that offset pointers (usually SI and DI) are incremented by 1 after each operation in the string instructions when the D flag is cleared. If the D flag is set, then SI and DI are decremented by 1 after each operation to process strings from high to low memory

IF (Interrupt Flag): When the interrupt flag is set, the 80386 recognizes and handles external hardware interrupts on its INTR pin.

If the interrupt flag is cleared, 80386 ignores any inputs on this pin. The IF flag is set and cleared with the STI and CLI instructions, respectively.

TF (Trap Flag): Trap flag allows users to single-step through programs. When an 80386 detects that this flag is set, it executes one instruction and then automatically generates an internal exception 1. After servicing the exception, the processor executes the next instruction and repeats the process.

System Flags:

1. **VM (Virtual Memory) flag:** This flag indicates the operating mode of 80386. When the VM flag is set, 80386 switches from protected mode to virtual 8086 modes.

2. **R (Resume) flag/Restart flag:** This flag, when set allows selective masking of some exceptions at the time of debugging

3. **NT (Nested flag):** This flag is set when one system task invokes another task.

4. **IOPL (VO Privilege level):** 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 in order to execute any I/O-related instruction.

Q. Explain Paging and Segmentation Unit of 80386?

Ans:

Segmentation unit:

1. Segmentation unit has the ability to convert the logical address into the linear address at the time of executing an instruction.
2. The segmentation unit compares the effective address for the length limit specified in the segment descriptor.
3. The segmentation unit adds the segment base and effective address to generate a linear address.
4. It gives 4 level protection to the data or code present in the memory. Every information in the memory is assigned a privilege level from PL0 to PL3. Here, PL0 holds the highest priority and PL3 holds the lowest priority.

Paging Unit:

5. The paging unit changes the linear address into a physical address.
6. The paging unit supports multitasking. This is so because the physical memory is not required to hold the whole segment of any task.
7. Despite this, only that part of the segment which is needed to be currently executed must be stored in that memory whose physical address is calculated by the paging unit.

8. This resultantly reduces the memory requirement and hence this frees the memory for other tasks. Thus by this we get an effective way for managing the memory to support multitasking.

Operating Modes of 80386 Microprocessors:

- Real Mode
- Protected Mode
- Virtual 8086 Modes

Real Mode:

When the 80386 is turned on for the first time, the Real mode is its default operating mode.

Software created for the 8086 and 8088 processors will work on the 80386 without needing to be modified because it is fully backward-compatible with these older CPUs.

The 80386 has a 20-bit address space in Real mode, giving it access to 1MB of memory. The memory is separated into 64K segments, and a 16-bit segment register can be used to access each section.

The processor uses segment-based memory addressing while operating in this mode and is not given access to sophisticated memory management and security functions.

Real mode likewise has a constrained set of instructions and does not support protected or virtual memory.

Protected Mode:

A 32-bit address space is available in protected mode, a sophisticated operating mode that gives users access to up to 4GB of memory.

Additionally, it offers sophisticated memory management and security features including segmentation and paging. Pages are fixed-size units of memory that can be moved in and out of physical memory as needed. Paging enables this.

Memory can be separated into logical units called segments through segmentation, which can be used to restrict access to particular memory locations.

The 80386 also has access to a number of privileged instructions and registers in a protected mode that are not present in regular mode. The protected mode also supports virtual memory which allows the system to use more memory than the physical memory available by swapping memory pages to and from the disk.

Virtual 8086 Modes:

The 80386 may operate numerous virtual environments that are compatible with the 8086 simultaneously in virtual 8086 modes.

Although they share the same physical memory, each virtual environment has its own set of registers and memory space.

This mode enables the 80386 to mimic the actions of an 8086 processor, which is frequently used to run older applications on more modern computers.

The 80386 may move between many virtual environments as needed thanks to the system for handling interrupts and exceptions that are provided by virtual 8086 modes.

The virtual 8086 mode allows for the simultaneous operation of numerous virtual machines without interfering with one another while simulating the behavior of a real 8086 CPU in a protected virtual memory environment.

Operating Mode	Address Space	Memory Management Features	Backward Compatibility
Real Mode	20-bit	None	Fully compatible with 8086/8088 processors
Protected Mode	32-bit	Segmentation, paging, privileged instructions	Partial compatibility with 8086/8088 processors
Virtual 8086 Mode	20-bit (per virtual machine)	Virtualization of 8086 environment	Fully compatible with 8086/8088 processors

Addressing Modes 80386:

1. Register operand addressing
2. Immediate operand addressing
3. Memory operand addressing

1.Register Operand Addressing:

In this addressing mode, the instruction specifies a register as the operand.

The operand is the value stored within the register. This method of addressing is extremely fast because it only requires the CPU to access the register, which is located directly on the processor.

It is commonly used for arithmetic and logical operations, as well as data transfers between registers.

The value can be saved in a 32-bit general register (EDX, ECX, EBX, EAX, EDI, ESI, EBP, or ESP),

16-bit general register (DX, CX, BX, AX, DI, SI, BP, or SP), or an

8-bit general register (DH, CH, BH, AH, BL, AL, DL, or CL).

2.Immediate Operand Addressing:

In certain instructions, the operand is a constant value embedded within the instruction itself.

This addressing mode is particularly useful for operations that require a fixed value as an operand, such as setting a value or comparing it to a constant.

In some cases, an instruction can use data from within itself as an operand, which is known as an immediate operand.

The length of the operand can be 32, 16, or 8 bits, depending on the instruction.

Example:

MOV DL, 08H

The 8-bit data (08H) given in the instruction is moved to DL

(DL) ← 08H

MOV AX, 0A9FH

The 16-bit data (0A9FH) given in the instruction is moved to AX register

(AX) ← 0A9FH

3.Memory operand Addressing:

In this addressing mode, the instruction includes a memory address as the value. The data stored at that particular memory location is utilized as the value for the instruction.

This mode is used for operations that involve reading from or writing to memory, like transferring data into a register or saving data from a register to memory.

The memory operand addressing mode can be divided into different types:

Direct addressing, indirect addressing; register indirect addressing, base-pointer-indexed addressing, scaled indexed addressing, and relative addressing. Each of these addressing modes has its own specific purposes and use cases.

Addressing Modes

1. 1. Immediate: In this addressing mode, immediate data is a part of instruction. Data byte or Bytes appears in the form of successive byte or byte.

Ex. MOV AX , 1005H

2. Direct: In this addressing mode, physical address is formed by the addition of a segment register and a displacement value that is coded directly in to the instruction. The displacement is usually the address of memory location within a specific segment.

Ex. MOV AX , [1005H]

Here, actual data is residing in memory. 1005H is the offset address of memory location and by default contents of DS (Data Segment) will be considered as segment address.

3. Register Indirect: In this addressing mode, we pass offset address of memory location using processor register within square bracket. These registers can be (BX and BP) Base registers or (SI and DI) Index registers. 16-bit contents of these registers will be added to DS (or SS if BP is used) register to form 20-bit physical address.

Ex. MOV AL , [SI] or MOV [BX] , DX

4.Register Relative: In this addressing mode, one of the two base registers (BX or BP) and 8-bit or 16-bit displacement is included in operand field. To find actual physical address this displacement is also added with segment address and offset address. Displacement may be +ve or -ve.

Ex.MOV AX , [BP+6] or MOV [BX-3] , CX

5. Relative Indexed Addressing Mode or Indexed Addressing Mode: In this addressing mode, one of two index registers (SI or DI) and 8-bit or 16-bit displacement is included in operand field. To find actual physical

address this displacement is also added with segment and offset address. Displacement may be +ve or -ve.

Ex.MOV AX , [SI+16] or MOV [DI-8], CX

6. Based Indexed Addressing Mode: In this addressing mode, combined features of based and Indexed addressing modes, but not allow the use of a displacement. Two registers must be used as the source or destination operand, one from BX/BP and another from SI/DI. This combination of two registers can be written as [BX] [DI] or [BX+DI], both have same meaning and give total offset value with addition of these registers.

Ex. MOV [BP] [SI] , AL

7. Relative Based Indexed or Based Indexed with Displacement: This addressing mode combines the features of Based and Indexed addressing modes and in addition it gives 8-bit or 16-bit displacement too.

Ex.MOV CL , [BX+DI+1587H]

Data Types of 80386 Microprocessor :

Data types are an important part of microprocessor programming. They specify the types of data that can be stored in variables and the operations that can be performed on that data.

- **Bit:** A single-bit quantity.
- **Bit Field:** A group of up to 32 contiguous bits, which spans a maximum of four bytes.
- **Bit String:** A set of contiguous bits, on the Intel386 DX bit strings, can be up to 4 gigabits long.
- **Byte:** A signed 8-bit quantity. (-128 through +127)
- **Unsigned Byte:** An unsigned 8-bit quantity. (0 through 255)
- **Integer (Word):** A signed 16-bit quantity. (-32,768 through +32,767)
- **Unsigned Integer (Word):** An unsigned 16-bit quantity. (0 through 65535)
- **Long Integer (Double Word):** A signed 32-bit quantity. All operations assume a 2's complement representation. (-23 through +231 -1)
- **Unsigned Long Integer (Double Word):** An unsigned 32-bit quantity. (0 through 212 -1)
- **Signed Quad Word:** A signed 64-bit quantity.

- **Unsigned Quad Word:** An unsigned 64-bit quantity.

- **Offset:** A 16 or 32-bit offset-only quantity which indirectly references another memory location.

- **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: A 16-bit segment selector component and a 32-bit offset component. Far pointers are used by application programmers only when systems designers choose a segmented memory organization.

- **Char:** A byte representation of an ASCII Alphanumeric or control character.

- **String:** A contiguous sequence of bytes, words, or dwords.

  A string may contain between 1 byte and 4 Gbytes.

- **BCD:** A byte (unpacked) representation of decimal digits 0-9.

- **Packed BCD:** A byte (packed) representation of lavo decimal digits 0-9 storing one digit in each nibble.
  When the Intel386 DX is coupled with an Intel387 DX Numeric Coprocessor then the following common Floating Point types are supported.

- **Floating Point:** A signed 32-bit, 64-bit, or 80-bit real number representation Floating point numbers are supported by the Intel387 DX numeric coprocessor.

Instruction Set :

1.Data Movement Instructions :

Three Classes of Data Movement Instructions:

1. General-purpose data movement instructions:

MOV and XCHG

2.Stack manipulations instructions:

PUSH,POP,PUSHA,POPA

3.Type-conversion instructions :

CWD,CDQ,CBW,CWDE,MOVSX,MOVZX

1. General-purpose data movement instructions:

- **MOV** – Used to copy the byte or word from the provided source to the provided destination.

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

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

Memory-to-memory moves can be performed, however, by the string move instruction MOVS.

- **XCHG** – Used to exchange the data from two locations.

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

2. Stack manipulations instructions:

- **PUSH** – Used to put a word at the top of the stack.
- **POP** – Used to get a word from the top of the stack to the provided location.
- **PUSHA** – Used to put all the registers into the stack.
- **POPA** – Used to get words from the stack to all registers.

Figure 3-1. PUSH

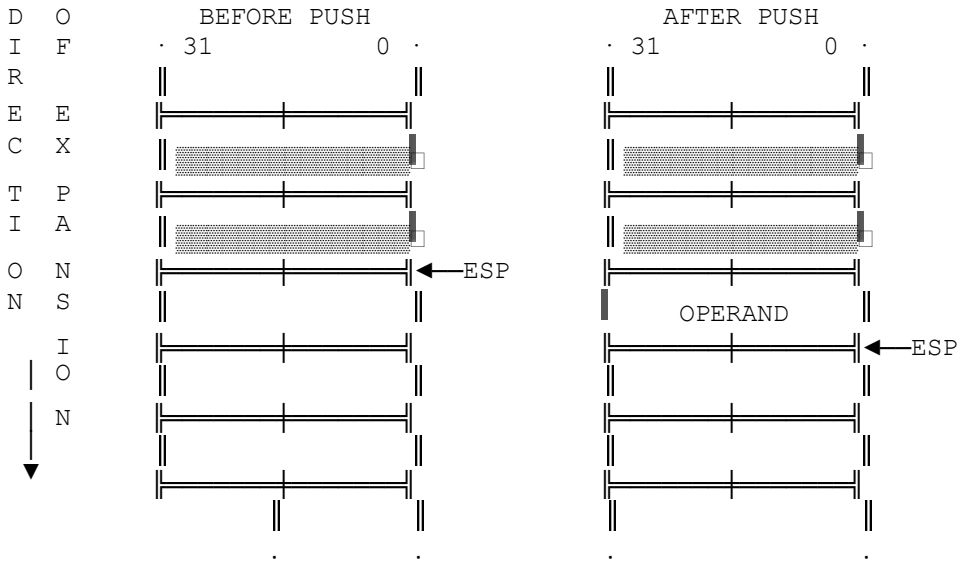
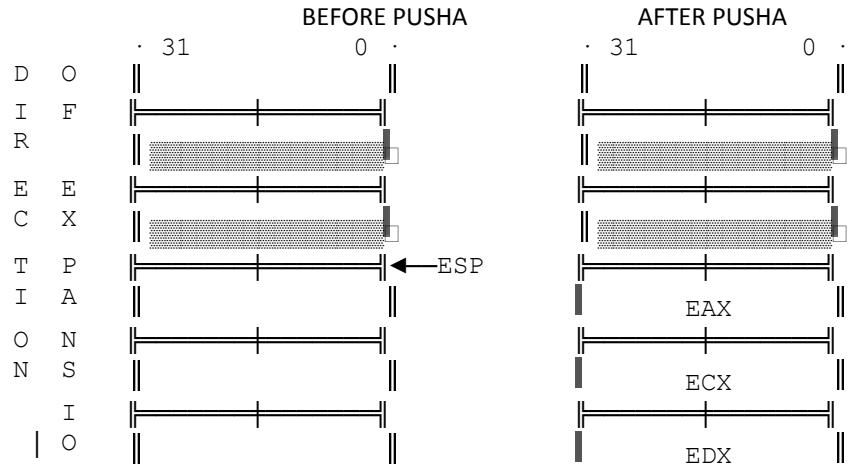


Figure 3-2. PUSHA



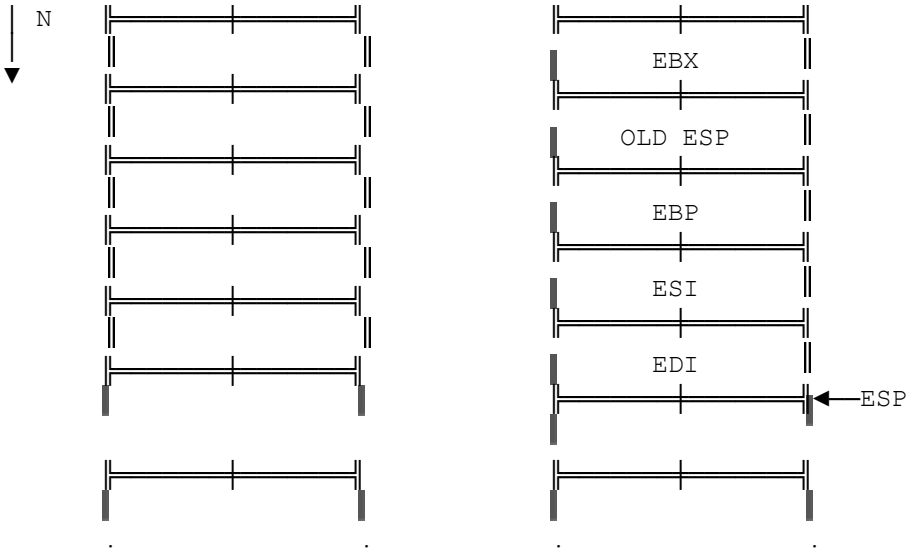


Figure 3-3. POP

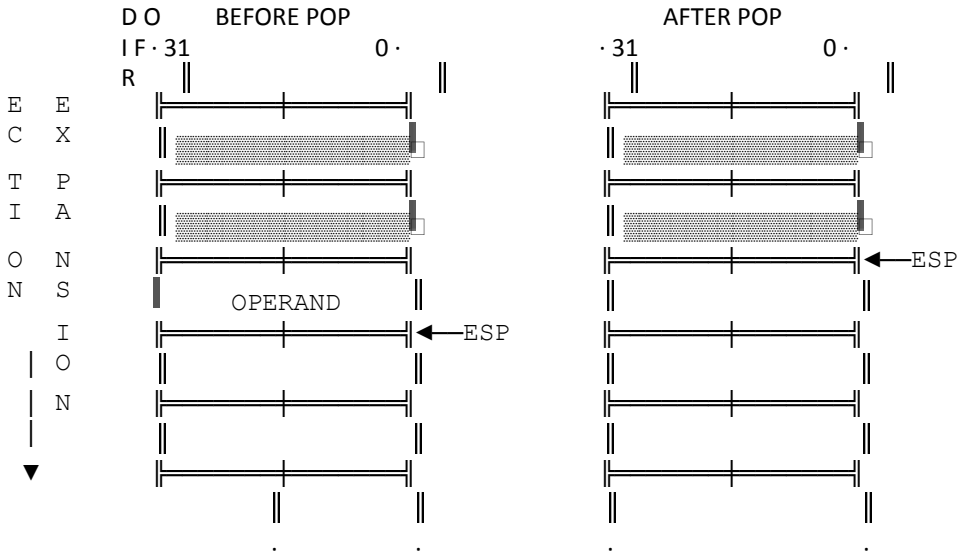
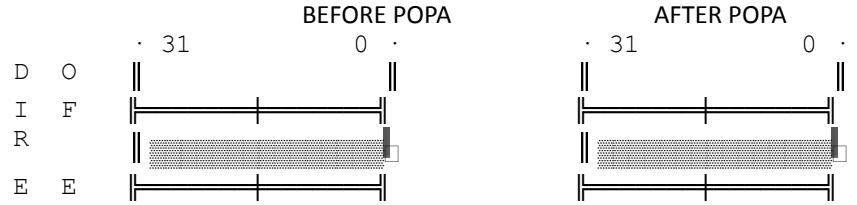
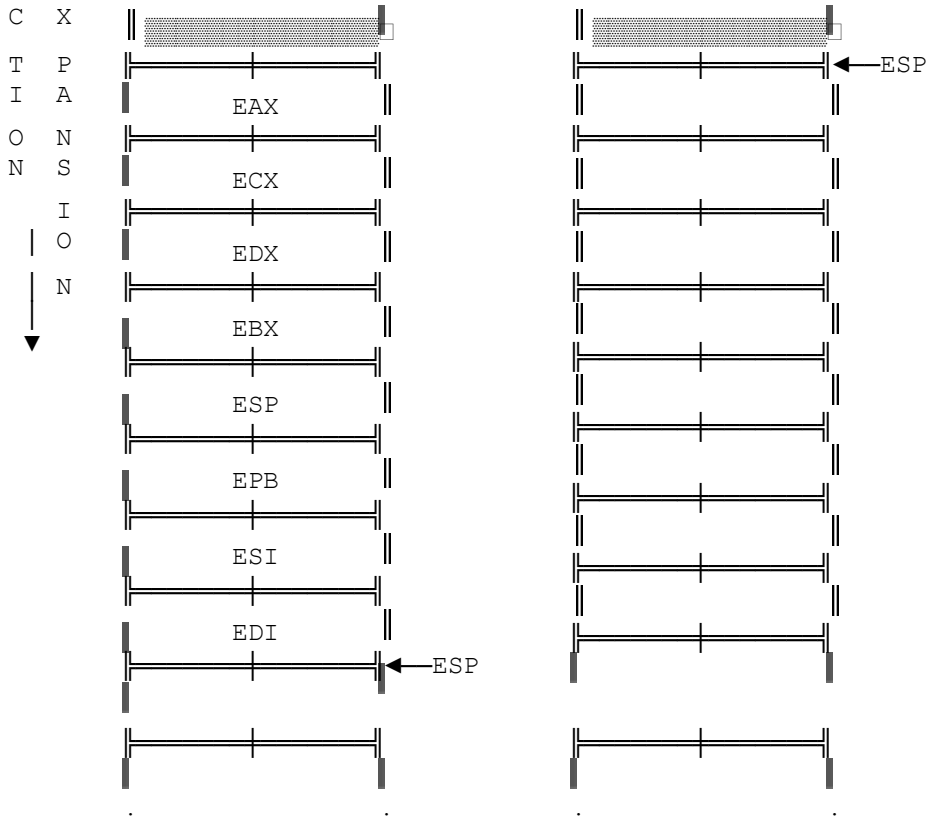


Figure 3-4. POPA





3.Type-conversion instructions :

The type conversion instructions convert bytes into words, words into doublewords, and doublewords into 64-bit items (quad-words).

These instructions are especially useful for converting signed integers, because they automatically fill the extra bits of the larger item with the value of the sign bit of the smaller item. This kind of conversion is called sign extension.

There are two classes of type conversion instructions:

1. The forms CWD, 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 (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.

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

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

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

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

2) Arithmetic Instructions :

Addition Instruction :

- **ADD** – Used to add the provided byte to byte/word to word.
- **ADC** – Used to add with carry.
- **INC** – Used to increment the provided byte/word by 1.
- **AAA** – Used to adjust ASCII after addition.
- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

Subtraction Instruction :

- **SUB** – Used to subtract the byte from byte/word from word.
- **SBB** – Used to perform subtraction with borrow.
- **DEC** – Used to decrement the provided byte/word by 1.
- **NPG** – Used to negate each bit of the provided byte/word and add 1/2's complement.
- **CMP** – Used to compare 2 provided byte/word.
- **AAS** – Used to adjust ASCII codes after subtraction.
- **DAS** – Used to adjust decimal after subtraction.

Multiplication Instruction :

- **MUL** – Used to multiply unsigned byte by byte/word by word.
- **IMUL** – Used to multiply signed byte by byte/word by word.

- **AAM** – Used to adjust ASCII codes after multiplication.

Division Instruction :

- **DIV** – Used to divide the unsigned word by byte or unsigned double word by word.

- **IDIV** – Used to divide the signed word by byte or signed double word by word.

3) Decimal Arithmetic Instruction :

Packed BCD Adjustment Instruction :

- **DAA** – Used to adjust the decimal after the addition/subtraction operation.

- **DAS** – Used to adjust decimal after subtraction.

Unpacked BCD Adjustment Instruction :

- **AAA** – Used to adjust ASCII after addition.

- **AAS** – Used to adjust ASCII codes after subtraction.

- **AAM** – Used to adjust ASCII codes after multiplication.

4) Logical Instruction :

1.Boolean operation instructions :

- **NOT** – Used to invert each bit of a byte or word.
- **OR** – Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- **AND** - Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- **XOR** – Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.

2.Shift instructions :

- **SHL/SAL** – Used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- **SHR** – Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- **SAR** – Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

3.Rotate instructions :

- **ROL** – Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].

- **ROR** – Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].

- **RCR** – Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.

- **RCL** – Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

4.Test instructions :

- **TEST** – Used to add operands to update flags, without affecting operands.

5) Control Transfer Instructions :

The 8086 provides both conditional and unconditional control transfer instructions to direct the flow of execution.

Conditional control transfers depend on the results of operations that affect the flag register.

Unconditional control transfers are always executed.

Jump Instructions :

- **JMP :** – Used to jump to the provided address to proceed to the next instruction.

Call Instructions :

- **CALL** – Used to call a procedure and save their return address to the stack.

Return Instructions :

- **RET** – Used to return from the procedure to the main program.

Unsigned Conditional Transfers :

- **JA/JNBE** – Used to jump if above/not below/equal instruction satisfies.
- **JAE/JNB** – Used to jump if above/not below instruction satisfies.
- **JBE/JNA** – Used to jump if below/equal/ not above instruction satisfies.
- **JC** – Used to jump if carry flag CF = 1
- **JE/JZ** – Used to jump if equal/zero flag ZF = 1
- **JNE/JNZ** – Used to jump if not equal/zero flag ZF = 0

- **JNP/JPO** – Used to jump if not parity/parity odd PF = 0
- **JP/JPE** – Used to jump if parity/parity even PF = 1

Signed Conditional Transfers :

- **JG/JNLE** – Used to jump if greater/not less than/equal instruction satisfies.
- **JGE/JNL** – Used to jump if greater than/equal/not less than instruction satisfies.
- **JL/JNGE** – Used to jump if less than/not greater than/equal instruction satisfies.
- **JLE/JNG** – Used to jump if less than/equal/if not greater than instruction satisfies.
- **JNO** – Used to jump if no overflow flag OF = 0
- **JO** – Used to jump if overflow flag OF = 1
- **JS** – Used to jump if sign flag SF = 1

Loop Instructions :

- **LOOP** – Used to loop a group of instructions until the condition satisfies, i.e., CX = 0
- **LOOPE/LOOPZ** – Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0
- **LOOPNE/LOOPNZ** – Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

Executing a Loop or Repeat zero Times Instructions :

- **JCXZ** – Used to jump to the provided address if CX = 0

Software-Generated Interrupts :

- **INT** – Used to interrupt the program during execution and calling service specified.
- **INTO** – Used to interrupt the program during execution if OF = 1

6) String and Character Translation Instructions :

The instructions in this category operate on strings rather than on logical or numeric values.

1. A Set of Primitive string Operations:

- MOVS -- Move String

  moves the string element pointed to by ESI to the location pointed to by EDI.

  The MOVS instruction, when accompanied by the REP prefix, operates as a memory-to-memory block transfer.

To set up for this operation, the program must initialize ECX and the register pairs ESI and EDI.

ECX specifies the number of bytes, words, or doublewords in the block.

MOVSB operates on byte elements,

MOVSW operates on word elements, and

MOVSD operates on doublewords.

The destination segment register cannot be overridden by a segment override prefix, but the source segment register can be overridden.

- CMPS -- Compare string

  subtracts the destination string element (at ES:EDI) from the source string element (at ESI) and updates the flags AF, SF, PF, CF and OF.

  If the string elements are equal, ZF=1; otherwise, ZF=0. If DF=0, the processor increments the memory pointers (ESI and EDI) for the two strings.

  CMPSB compares bytes,

  CMPSW compares words, and

  CMPSD compares doublewords.

The segment register used for the source address can be changed with a segment override prefix while the destination segment register cannot be overridden.

- SCAS -- Scan string

  Subtracts the destination string element at ES:EDI from EAX, AX, or AL and updates the flags AF, SF, ZF, PF, CF and OF.

  If the values are equal, ZF=1; otherwise, ZF=0. If DF=0, the processor increments the memory pointer (EDI) for the string.

  SCASB scans bytes;

  SCASW scans words;

  SCASD scans doublewords.

  The destination segment register (ES) cannot be overridden.

- LODS -- Load string

  Places the source string element at ESI into EAX for doubleword strings, into AX for word strings, or into AL for byte strings.

  LODS increments or decrements ESI according to DF.

- STOS -- Store string

Places the source string element from EAX, AX, or AL into the string at ES:DSI.

STOS increments or decrements EDI according to DF.

2. Control flag instructions:

CLD -- Clear direction flag instruction

The instruction CLD puts zero in DF, causing the index registers to be incremented

STD -- Set direction flag instruction

The instruction STD puts one in DF, causing the index registers to be decremented.

3. Repeat prefixes:

- REP -- Repeat while ECX not xero
- REPE/REPZ -- Repeat while equal or zero
- REPNE/REPNZ -- Repeat while not equal or not zero

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

7) Instructions for Block-Structured Languages:

The instructions in this section provide machine-language support for functions normally found in high-level languages.

These instructions include ENTER and LEAVE, which simplify the programming of procedures.

ENTER (Enter Procedure) creates a stack frame that may be used to implement the scope rules of block-structured high-level languages.

A LEAVE instruction at the end of a procedure complements an ENTER at the beginning of the procedure to simplify stack management and to control access to variables for nested procedures.

8) Flag Control Instructions :

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

Carry and Direction Flag Control Instructions :

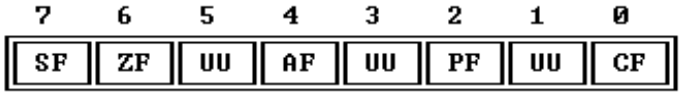
The carry flag instructions are useful in conjunction with rotate-with-carry instructions RCL and RCR.
They can initialize the carry flag, CF, to a known state before execution of a rotate that moves the carry bit into one end of the rotated operand.

The direction flag control instructions are specifically included to set or clear the direction flag, DF, which controls the left-to-right or right-to-left direction of string processing. If DF=0, the processor automatically increments the string index registers, ESI and EDI, after each execution of a string primitive. If DF=1, the processor decrements these index registers.

Flag Control Instruction	Effect
STC (Set Carry Flag)	CF := 1
CLC (Clear Carry Flag)	CF := 0
CMC (Complement Carry Flag)	CF := NOT (CF)
CLD (Clear Direction Flag)	DF := 0
STD (Set Direction Flag)	DF := 1

Flag Transfer Instructions :

Figure 3-22. LAHF and SAHF



LAHF LOADS FIVE FLAGS FROM THE FLAG REGISTER INTO REGISTER AH. SAHF STORES THESE SAME FIVE FLAGS FROM AH INTO THE FLAG REGISTER. THE BIT POSITION OF EACH FLAG IS THE SAME IN AH AS IT IS IN THE FLAG REGISTER. THE REMAINING BITS (MARKED UU) ARE RESERVED; DO NOT DEFINE.

The flag transfer instructions allow a program to alter the other flag bits with the bit manipulation instructions after transferring these flags to the stack or the AH register.

The instructions LAHF and SAHF deal with five of the status flags, which are used primarily by the arithmetic and logical instructions.

LAHF (Load AH from Flags) copies SF, ZF, AF, PF, and CF to AH bits 7, 6, 4, 2, and 0, respectively. The contents of the remaining bits (5, 3, and 1) are undefined. The flags remain unaffected.

SAHF (Store AH into Flags) transfers bits 7, 6, 4, 2, and 0 from AH into SF, ZF, AF, PF, and CF, respectively

The PUSHF and POPF instructions are not only useful for storing the flags in memory where they can be examined and modified but are also useful for preserving the state of the flags register while executing a procedure.

PUSHF (Push Flags) decrements ESP by two and then transfers the low-order word of the flags register to the word at the top of stack pointed to by ESP.

POPF (Pop Flags) transfers specific bits from the word at the top of stack into the low-order byte of the flag register then increments ESP by two.

9) Coprocessor Interface Instructions :

ESC (Escape) is a 5-bit sequence that begins the opcodes that identify floating point numeric instructions. The ESC pattern tells the 80386 to send the opcode and addresses of operands to the numerics coprocessor.

The numerics coprocessor uses the escape instructions to perform high-performance, high-precision floating point arithmetic that conforms to the IEEE floating point standard 754.

WAIT (Wait) is an 80386 instruction that suspends program execution until the 80386 CPU detects that the BUSY pin is inactive. This condition indicates that the coprocessor has completed its processing task and that the CPU may obtain the results.

10) Segment Register Instructions :

This category actually includes several distinct types of instructions. These various types are grouped together here because, if systems

designers choose an unsegmented model of memory organization, none of these instructions is used by applications programmers.

1. **Segment-register transfer instructions.**
    - MOV SegReg, ...
    - MOV ..., SegReg
    - PUSH SegReg
    - POP SegReg

The MOV, POP, and PUSH instructions also serve to load and store segment registers. These variants operate similarly to their general-register counterparts except that one operand can be a segment register.

MOV cannot move segment register to a segment register.

Neither POP nor MOV can place a value in the code-segment register CS; only the far control-transfer instructions can change CS.

2. **Control transfers to another executable segment.**
    - JMP far direct and indirect
    - CALL far
    - RET far

Direct far JMP. Direct JMP instructions that specify a target location outside the current code segment contain a far pointer.

This pointer consists of a selector for the new code segment and an offset within the new segment.

Indirect far JMP. Indirect JMP instructions that specify a target location outside the current code segment use a 48-bit variable to specify the far pointer.

Far CALL. An intersegment CALL places both the value of EIP and CS on the stack.

Far RET. An intersegment RET restores the values of both CS and EIP which were saved on the stack by the previous intersegment CALL instruction.

3. Data pointer instructions.

- LDS :

  LDS (Load Pointer Using DS) transfers a pointer variable from the source operand to DS and the destination register.

- LES :

LES (Load Pointer Using ES) operates identically to LDS except that ES receives the segment selector rather than DS.

Example: LES EDI, DESTINATION_X

- LFS :

LFS (Load Pointer Using FS) operates identically to LDS except that FS receives the segment selector rather than DS.

- LGS :

  LGS (Load Pointer Using GS) operates identically to LDS except that GS receives the segment selector rather than DS.

- LSS :

  LSS (Load Pointer Using SS) operates identically to LDS except that SS receives the segment selector rather than DS.

11). Miscellaneous Instructions :

11.1 Address Calculation Instruction :

LEA (Load Effective Address) transfers the offset of the source operand (rather than its value) to the destination operand.

The source operand must be a memory operand, and the destination operand must be a general register.

This instruction is especially useful for initializing registers before the execution of the string primitives (ESI, EDI) or the XLAT instruction (EBX).

The LEA can perform any indexing or scaling that may be needed.

Example: `LEA EBX, EBCDIC_TABLE`

11.2 No-Operation Instruction :

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

11.3 Translate Instruction :

XLAT (Translate) i replaced a byte in the AL register with a byte from a user-coded translation table.

When XLAT is executed, AL should have the unsigned index to the table addressed by EBX.

XLAT changes the contents of AL from table index to table entry.

The translate table may be up to 256 bytes long. The value placed in the AL register serves as an index to the location of the corresponding translation value.