Notes on

8086 Microprocessor

Introduction to Microprocessor

A microprocessor is an electronic chip that acts as the central processing unit (CPU) of a computer.

Microprocessor based systems with limited resources are called microcomputer.
Almost all the microprocessors use the basic concept of "**stored program execution**" using which the programs are stored sequentially in memory locations.

The microprocessors fetch the instructions one after the other and execute them in its arithmetic and logic unit.

Each microprocessor will have its own associated set of instructions.

Programs are called Assembly Level Language and then they are converted into binary machine level language. This conversion may be done manually or using an application known as Assembler.

In general, the programs are written by the user for a microprocessor to work with the real world data.

A microprocessor based system needs a set of memory units, a set of interfacing circuits for inputs and a set of interfacing circuits for outputs. All circuits applied along with the microprocessor are called microcomputer systems.

Examples of some common devices that are using microprocessor are computers, printers, automobiles, washing machines, microwave ovens, mobile phones etc.

Generations of Microprocessor

The transistor technology led to the introduction of minicomputers of the 1960s and the personal computer revolution of the 1970s.

Microprocessors evolution is categorized into 5 generations i.e. first, second, third, fourth, and fifth generations.

First Generation (1971-73)

In 1971, Intel Corporation introduced the first 4-bit 4004 microprocessor at 108 kHz.

In 1972, Intel made 8-bit 8008 and 8080 microprocessors.

Second Generation (1974-78)

The second generation microprocessor makes the use of newer semiconductor technology to fabricate the chips.

They were manufactured using NMOS technology.

Some of the popular processors are Motorola’s 6800 and 6809 and Intel’s 8085, Zilog’s Z80.

Third Generation (1979-80)

The third-generation microprocessors were dominated by Intel's 8086 and Zilog Z8000, which were 16-bit processors, 16-bit arithmetic and pipeline instruction processing.

Fourth Generation (1981-95)

In fourth generation microprocessors, Intel introduced 32 bit processors, 80386 and Motorola 68020/68030.

These are fabricated using low-power version of the HMOS technology called HCMOS.

Fifth Generation (1995 - till date)

In fifth generation microprocessors, Intel introduced 64 bit processors.

These carry on-chip functionalities and improvements in the speed of memory and I/O devices. Their design surpassed 10 million transistors per chip.

Some examples are Pentium, Celeron and dual and quad core processors working with up to 3.5GHz speed.

Microprocessor Architecture

Programming Model

Programming models of 8086 through Core2 are considered program visible because its registers are used during application programming and specified by the instructions.

Other registers are considered to be **program invisible** because they are not directly addressable during programming of applications, but can be used indirectly during system programming.

Only 80286 and above contains the program-invisible registers to operate and control the protected memory system and other features of the microprocessor.

Programming models include 8, 16 and 32 bit registers. On operating in 64-bit mode, Pentium 4 and Core 2 have 64-bit registers.

8-bit registers includes AH, AL, BH, BL, CH, CL, DH and DL.

16-bit registers includes AX, BX, CX, DX, SP, BP, DI, SI, IP, FLAGS, CS, DS, ES, SS, FS and GS.

Extended **32-bit registers** includes EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, EIP and EFLAGS.

Note: These 32-bit extended registers and 16-bit registers FS and GS are only available in the 80386 and above.

Multipurpose registers includes EAX, EBX, ECX, EDX, EBP, EDI and ESI.

64-bit registers includes RAX, RBX, RCX, RDX, RSP, RBP, RDI, RSI, RIP, RFLAGS and additional 64-bit registers that are called R8 through R15.

These additional 64-bit registers (R8 through R15) are addressed as a byte, word, double word, or quad word but only the rightmost 8 bits is a byte. R8 through R15 have no provision for directly addressing bits 8 through 15 as a byte.

Flat mode 64-bits access to numbered registers

Register Size	Override	Bits Accessed	Example
8 bits	B	7 – 0	MOV R8B, R9B
16 bits	W	15 – 0	MOV R14W, BX
32 bits	D	31 – 0	MOV R12D, R15D
64 bits	-	63 – 0	MOV R11, R13

General Purpose Registers

AX (Accumulator): AX is addressable as RAX, EAX, AX, AH, or AL. It is **used as accumulator** which multiply, divide, input/output (I/O) and some of the decimal and ASCII adjustment instructions.

BX (Base Index): BX is addressable as RBX, EBX, BX, BH, or BL. It **holds the offset address** of a location in the memory system. It is also **used to refer data in memory**.

CX (Count): CX is addressable as RCX, ECX, CX, CH, or CL. It **holds the count** for various instructions and the **offset address of memory data**. The value of this register indicates the number of times the same instructions has to be executed.

DX (Data): DX is addressable as RDX, EDX, DX, DH, or DL. It **holds a part of the result** from a multiplication or part of the dividend before a division. It also **holds the I/O device address** while executing the IN and OUT instructions. This register can also **address memory data**.

SP (Stack Pointer): SP is addressable as RSP, ESP, or SP. It is used to **hold the offset address** of the data stored at the top of stack segment. It is **used along with SS register** to decide the address at which data is pushed or popped during the execution of PUSH and POP instructions.

BP (Base Pointer): BP is addressable as RBP, EBP, or BP. It is used to **hold the offset address of data** to be read from or write into the stack segment.

SI (Source Index): SI is addressable as RSI, ESI, or SI. It is used to **hold the offset address of source data in data segment** while executing string instructions.

DI (Destination Index): DI is addressable as RDI, EDI, or DI. It is used to **hold the offset address of destination data in extra segment** while executing string instructions.

R8 through R15: These registers are only found in the Pentium 4 and Core2 if 64-bit extensions are enabled. The data in these registers are addressed as 64, 32, 16, or 8 - bit sizes and are of general purpose. The 8-bit portion is the rightmost 8-bit only and **bits from 8 to 15 are not directly addressable as a byte**.

Special-Purpose Registers

The special-purpose registers include RIP, RSP and RFLAGS and the segment registers include CS, DS, ES, SS, FS, and GS.

IP (Instruction Pointer): IP is addressable as RIP, EIP or IP. It points to the next instruction in a section of memory defined as a code segment. It is used by microprocessor to find the next sequential instruction in the program located within the code segment.

Flag registers of 8086

The flag registers can be classified into 2 categories,

1. Status Flags: They indicate the status of the result that is obtained after the execution of the arithmetic or logic instruction.

2. Control Flags: They can control the operation of CPU.

Status Flags

CF (Carry Flag): It **holds the carry after addition or borrow after subtraction** operation. It also **indicates error conditions**.

PF (Parity Flag): It is the **count of ones in a number** expressed as even or odd. It is **logic 0 for odd parity** (i.e. odd number of 1s) and **logic 1 for even parity** (i.e. even number of 1s). For example, if a number contains three binary one bits, it has odd parity and if a number contains no one bits, it has even parity.

AF (Auxiliary carry Flag): It **holds the carry (half-carry)** after addition or borrow after subtraction between bit positions 3 and 4 of the result in a **BCD** operation. This is used by DAA and DAS instructions to adjust the value in AL after a BCD addition or subtraction, respectively.

ZF (Zero Flag): It indicates that the **result of an arithmetic or logic operation is zero**. If **Z = 1**, the **result is zero** and if **Z = 0**, the **result is non zero**.

SF (Sign Flag): It **holds the arithmetic sign of the result** after arithmetic or logic instruction executes. If **S = 1**, the sign bit (leftmost bit of a number) is **set or negative** and if **S = 0**, the sign bit is **cleared or positive**.

TF (Trap Flag): It **enables trapping using single step technique**. If T flag is set (i.e. TF = 1), 8086 gets interrupted after the execution of each instruction in the program. If TF is cleared (i.e. TF = 0), the trapping or debugging feature is disabled.

IF (Interrupt Flag): It **controls the operation of the INTR** (interrupt request) input pin. If **I = 1**, the **INTR pin is enabled** and if **I = 0**, the **INTR pin is disabled**. The state of I flag bit is controlled by **STI** (Set I flag) and **CLI** (Clear I flag) instructions.

DF (Direction Flag): It **selects either the increment or decrement mode** for the DI and/or SI registers during string instructions. If **D = 1**, the registers are automatically **decremented** and if **D = 0**, the registers are automatically **incremented**. The D flag is set with the **STD** (Set Direction) and cleared with the **CLD** (Clear Direction) instructions.

OF (Overflow Flag): It **occurs when signed numbers are added or subtracted**. Signed numbers are represented in 2’s complement form in microprocessor. It **indicates that result has exceeded the capacity of the machine**.

Example: if the 8-bit signed data 7EH (+126) is added with the 8-bit signed data 02H (+2) then the result is 80H (-128 in 2’s complement form). This result indicates an overflow condition and hence the overflow flag is set during the above signed addition.

Note: In an 8-bit register, the minimum and maximum value of the signed number that can be stored is -128 (=80H) and +127 (=7FH) respectively while in a 16 bit register, the minimum and maximum value of the signed number that can be stored is -32768 (=8000H) and +32767 (=7FFFH) respectively.

IOPL (I/O Privilege Level): It is used in protected mode operation to **select the privilege level for I/O devices**. If the current privilege level is higher or more trusted than the IOPL then I/O executes without hindrance. If the current privilege level is lower than the IOPL then an interrupt occurs, causing execution to suspend. An **IOPL of 00 is the highest** or most trusted and an **IOPL of 11 is the lowest** or least trusted.

NT (Nested Task): It indicates that the **current task is nested within another task** in protected mode operation. This flag is set when the task is nested by software.

RF (Resume Flag): It is used with debugging to **control the resumption of execution** after the next instruction.

VM (Virtual Mode): It **selects virtual mode operation** in a protected mode system. A virtual mode system allows multiple DOS memory partitions that are 1M byte in length to coexist in the memory. This allows the system program to execute multiple DOS programs.

AC (Alignment Check): It **activates if a word or double word is addressed** on a non-word or non-double word boundary.

VIF (Virtual Interrupt Flag): It is a **copy of the interrupt flag** bit.

VIP (Virtual Interrupt Pending): It provides information about a virtual mode interrupt. It is **used in multitasking environments** to provide the operating system with virtual interrupt flags and interrupt pending information.

ID (Identification): It indicates that the Pentium – Pentium 4 microprocessors support the **CPU ID** instruction.

Segment Registers

Segment registers generate memory addresses when combined with other registers in the microprocessor. Segment registers functions differently in real mode and in protected mode operations of the microprocessor.

The **minimum size of a segment is 1 byte** and **maximum size of a segment is 64 Kbytes**. A segment begins in memory at a memory address which is divisible by 16.

There are 4 segment registers.

CS (Code Segment): It is a section of memory that **holds the code** (programs and procedures) used by the microprocessor. It **defines starting address** of the section of memory holding code. In real mode operation, it defines the start of a 64Kb section of memory and in protected mode operation, it selects a descriptor that describes the starting address and length of a section of memory holding code.

DS (Data Segment): It is a section of memory that **contains data** used by a program. The data is accessed in the data segment by an offset address or the contents of other registers that hold the offset address.

SS (Stack Segment): It **holds the stack of a program** which is needed while executing **CALL** and **RET** instructions to handle interrupts. Stack entry point is determined by stack segment and stack pointer registers. BP register also addresses data within the stack segment.

ES (Extra Segment): It is an **additional data segment** that is used by some of the string instructions to hold the destination data.

REAL MODE MEMORY ADDRESSING

Real mode operation allows microprocessor to address only **first 1Mb memory space**. The first 1Mb of memory is called the **real memory**, **conventional memory**, or **DOS memory** system.

Segments and Offsets

All real mode memory addresses must include a **segment address** and an **offset address**.

Segment address is located within the segment registers which defines the beginning address of any 64K memory segment.

Offset address selects any location within the 64Kb of memory segment.

In real mode, segments always have a length of **64Kb**.

Offset or displacement is the distance above the beginning of a segment.

In real mode, each segment register is internally appended with **0H** on its rightmost end. This forms a 20 bit memory address, allowing it to access the beginning of a segment.

Example, if a segment register contains **1201H**, it addresses a memory segment beginning at location **12010H**. Because of the internally appended 0H, real mode segments can begin only at **l6 byte boundary** in the memory system. This l6-byte boundary is often called a **paragraph**.

A real mode segment of memory is 64K in length, so once the beginning address is known, the **ending address** is calculated by **adding FFFFH**.

Example, if a segment register contains **3000H** then the **first address** of the segment is **30000H** and the **last address** is **3FFFFH** (30000H + FFFFH).

Offset address is added to the beginning of the segment to address a memory location within the memory segment.

Example, if the segment address is **1000H** and the offset address is **2000H**, the microprocessor addresses memory location **12000H** (10000H+2000H).

The segment and offset address for a segment address of 1000H with an offset of 2000H is sometimes written as **1000:2000**.

Some addressing modes combine multiple registers and one offset value to form an offset address. When this happens then the sum of these values may exceed FFFFH.

Example, the address accessed in a segment whose segment address is **4000H** and whose offset address is specified as the sum of **F000H** and **3000H** will access memory location **42000H instead of location 52000H**. When the F000H and 3000H are added, they form a 16 bit (modulo 16) sum of **2000H** used as the offset address; **not 12000H**, the true sum. Note that the carry of 1 in addition of F000H and 3000H (12000H) is dropped to form the offset address of 2000H. The address is generated as 4000:2000 or 42000H.

Example of real mode segment addresses

Segment Register	**Starting Address**	**Ending Address**
3000H	30000H	3FFFFH
3100H	31000H	40FFFH
AB00H	AB000H	BAFFFH
1234H	12340H	2233FH

<u>Default Segment and Offset Registers</u>

The microprocessor consists of a set of rules that apply to segments when addressing memory. These rules define the segment register and offset register combination.

For example, code segment register is always used with instruction pointer to address the next instruction in a program.

Code segment register defines the **beginning of the code segment** and the **instruction pointer** locates the **next instruction** within the code segment. The combination (CS:IP or CS:EIP) locates the next instruction executed by the microprocessor.

Example, if **CS = 1400H** and **IP = 1200H**, the microprocessor fetches its next instruction from memory location **15200H** (14000H + 1200H).

Stack data is referred to through the stack pointer which is in the memory space addressed by the stack pointers (SP / ESP) or the base pointers (BP / EBP).

Stack data is referenced through **stack segment** at the memory location addressed by either **stack pointer** (SP/ESP) or **base pointer** (BP/EBP).

Example, if **SS = 2000H** and **BP = 3000H**, the microprocessor addresses memory location **23000H** for the stack segment memory location.

A memory segment can overlap if 64Kb of memory are not required for a segment. A program can have more than 4 or 6 segments but can only access 4 or 6 segments at a time.

Default 16-bit segment and offset combinations

Segment	**Offset**	**Special Purpose**
CS	IP	Instruction address
SS	SP or BP	Stack address
DS	BX, DI, SI, an 8- or 16-bit number	Data address
ES	DI for string instructions	String destination address

Default 32-bit segment and offset combinations

Segment	**Offset**	**Special Purpose**
CS	EIP	Instruction address
SS	ESP or EBP	Stack address
DS	EAX, EBX, ECX, EDX, ESI, EDI, an 8- or 32-bit number	Data address
ES	EDI for string instructions	String destination address

Segment and Offset Addressing Scheme Allows Relocation

This scheme of segment plus offset addressing allows DOS programs to be relocated in the memory system. It also allows programs that are written to function in the real mode to operate in a protected mode system.

The segment and offset addressing scheme allows both programs and data to be relocated without changing anything in a program or data.

R**elocatable program** is a program that can be placed into any area of memory and executed without change.

Relocatable data is the data that can be placed in any area of memory and used without any change to the program.

As the memory is addressed within a segment by an offset address, the memory segment can be moved to any place in the memory system without changing any of the offset addresses.

Example: If an instruction is 4 bytes above the start of the segment, its offset address is 4. If the entire program is moved to a new area of memory, this offset address of 4 still points to 4 bytes above the start of the segment. Only content of the segment register must be changed to address the program in new area of memory.

PROTECTED MODE MEMORY ADDRESSING

Protected mode memory addressing allows access to data and programs located above the first 1M byte of memory as well as within the first 1M byte of memory.

Protected mode is where Windows operates.

When data and programs are addressed in extended memory, the offset address is still used to access information located within the memory segment.

The segment address is not present in the protected mode.

In place of segment address, the segment register contains a **selector** that selects a descriptor from the descriptor table. The **descriptor** describes location, length, and access rights of the memory segment. Let’s understand what are these selectors and descriptors.

Selectors and Descriptors

The selector is located in the segment register and selects one of 8192 descriptors from one of the two tables of descriptors. The **descriptor** describes the location, length, and access rights of memory segment.

For example, in the real mode, if CS = 0008H, the CS begins at location 00080H. In protected mode, this segment number can address any memory location in the entire system for code segment.

There are 2 descriptor tables used with the segment registers:

1. Global descriptors (System descriptor): They contain segment definitions that apply to all programs.

2. Local descriptors (application descriptor): They are usually unique to an application.

Each descriptor table has 8192 descriptors, so a total of 16,384 descriptors are available to an application at a time. As the descriptor describes a memory segment, this allows up to 16,384 memory segments to be described for each application. Since a memory segment can be up to 4G bytes in length, this means that an application could have access to 4G * 16,384 bytes or 64T bytes of memory.

Base address of a descriptor indicates the beginning location of the memory segment.

Segment limit contains the last offset address found in a memory segment.

For example, if a segment begins at memory location F00000H and ends at location F000FFH, the base address is F00000H and the limit is FFH.

Granularity bit or G bit: If G = 0, the limit specifies a segment limit of 00000H to FFFFFH and if G = 1, the value of the limit is multiplied by 4K bytes (appended with FFFH) so the segment limit is from 00000FFFFH to FFFFFFFFH.

Example: If G = 0, base address = 10000000H and limit = 001FFH, then

Segment Base = Start = 10000000H

Segment End = Base + Limit

= 10000000H + 001FFH

= 100001FFH

Example: If G = 1, base address = 10000000H and limit = 001FFH, then segment

Segment Base = Start = 10000000H

Segment End = Base + Limit (appended with FFFH)

= 10000000H + 001FFFFFH

= 101FFFFFH

Program-Invisible Registers

The global and local descriptor tables are found in the memory system. In order to access and specify the address of these tables, the microprocessor 80286–Core2 contains program-invisible registers. The program-invisible registers are not directly addressed by software so they are given this name.

Each of the segment registers contains a program-invisible portion used in the protected mode. The program-invisible portion of these registers is often called **cache memory** because cache is any memory that stores information.

The program-invisible portion of the segment register is loaded with the base address, limit, and access rights each time the number segment register is changed.

When a new segment number is placed in a segment register, the microprocessor accesses a descriptor table and loads the descriptor into the program-invisible portion of the segment register. It is held there and used to access the memory segment until the segment number is again changed. This allows the microprocessor to repeatedly access a memory segment without referring to the descriptor table.

GDTR (**Global Descriptor Table Register**) and **IDTR** (**Interrupt Descriptor Table Register**) contain the base address of the descriptor table and its limit. The limit of each descriptor table is 16 bits because the maximum table length is 64K bytes.

When the protected mode operation is desired, the address of the global descriptor table and its limit are loaded into the GDTR. Before using the protected mode, the interrupt descriptor table and the IDTR must also be initialized.

The location of the local descriptor table is selected from the global descriptor table. One of the global descriptors is set up to address the local descriptor table.

To access the local descriptor table, the **LDTR** (**Local Descriptor Table Register**) is loaded with a selector. This selector accesses the global descriptor table and loads the address, limit, and access rights of the local descriptor table into the cache portion of the LDTR.

A task is a procedure or application program. The descriptor for the procedure or application program is stored in the global descriptor table, so access can be controlled through the privilege levels.

TR (Task Register) holds a selector which accesses a descriptor that defines a task. It allows a context or task switch in about 17µs. Task switching allows the microprocessor to switch between tasks in a short amount of time. The task switch allows multitasking systems to switch from one task to another in a simple and orderly fashion.

Addressing Modes

Data Addressing Modes

An **opcode** or operation code specifies the microprocessor which operation to perform.

A comma always separates the destination from the source in an instruction.

Memory-to-memory transfers are *not* allowed by any instruction except for the **MOV** instruction.

MOV AX, BX instruction transfers the word contents of the source register (BX) into the destination register (AX).

A MOV instruction always *copies* the source data into the destination.

Effective Address (EA) represents the offset address of the data within a segment.

The data-addressing modes are as follows:

1. Register Addressing Mode
2. Register relative Addressing Mode
3. Register Indirect Addressing Mode
4. RIP relative Addressing Mode
5. Immediate Addressing Mode
6. Direct Data Addressing Mode
7. Index Addressing Mode
8. Index relative Addressing Mode
9. Scaled Index Addressing Mode
10. Base Addressing Mode
11. Base relative Addressing Mode
12. Base plus Index Addressing Mode
13. Base relative plus Index Addressing Mode

1. Register Addressing Mode: In this addressing mode, the data present in register is moved or manipulated and the result is stored in the register.

The microprocessor contains the following 8-bit registers used with register addressing: AX, BX, CX, DX, SP, BP, SI, and DI.

Examples: **MOV AL, BL**; copies content of BL to AL and **ADC BX, DX**; adds the content of BX, carry flag and DX and store result in BX

Note: Never mix an 8-bit register with a 16-bit register, an 8-bit register with a 32-bit register, or a l6-bit register with a 32-bit register because this is not allowed by the microprocessor and this will result in an error when assembled. For example, **MOV AX, AL** instruction is ***not*** allowed because the registers are of different sizes.

None of the MOV instructions affect the flag bits. The flag bits are normally modified by arithmetic or logic instructions.

The code segment register is *not* changed by a MOV instruction because the address of the next instruction is found by both IP/EIP and CS. If only CS were changed, the address of the next instruction would be unpredictable. Therefore, **changing the CS register with a MOV instruction is not allowed.**

Example: Let initially CX = 1234H and BX = 76AFH.

MOV BX, CX; This instruction moves (*copies*) 1234H from register CX into register BX. This *overwrites* the old content (76AFH) of register BX, but the contents of CX remain unchanged. The contents of the destination register or destination memory location change for all instructions except the CMP and TEST instructions. Note that this instruction does not affect the leftmost 16 bits of register EBX.

Examples of register-addressed instructions

Assembly Language	**Size**	**Operation**
MOV AL,BL	8 bits	Copies BL into AL
MOV R8B,CH	8 bits	Not allowed
MOV AX,CX	16 bits	Copies CX into AX
MOV ECX,EBX	32 bits	Copies EBX into ECX
MOV RAX,RDX	64 bits	Copies RDX into RAX
MOV ES,DS	—	Not allowed (segment-to-segment)
MOV BL,DX	—	Not allowed (mixed sizes)
MOV CS,AX	—	Not allowed (the code segment register should not be the destination register)

Register Relative Addressing: In this memory addressing mode, the data in a segment of memory are addressed by adding the displacement to the contents of a base or an index register.

Example: **MOV AX, [BX + 4]** This instruction loads AX from the data segment address formed by BX plus 4.and **MOV AX, ARRAY [BX]** this instruction loads AX from the data segment memory location in ARRAY plus the contents of BX.

The size of a real mode segment is 64K bytes long.

The displacement is a number added/subtracted to the register within the [ ]. A displacement also can be an offset address appended to the front of the [ ], as in MOV AL, DATA [DI].

In the 8086–80286 microprocessors, the value of the displacement is limited to a 16-bit signed number with a value ranging between +32,767 (7FFFH) and –32,768 (8000H) whereas in the 80386 and above, a 32-bit displacement is allowed with a value ranging between +2,147,483,647 (7FFFFFFFH) and -2,147,483,648 (80000000H).

Examples of register relative addressing

Assembly Language	**Size**	**Operation**
MOV AX,[DI+100H]	16 bits	Copies the word contents of the data segment memory location addressed by DI plus 100H into AX.
MOV ARRAY[EBX],EAX	32 bits	Copies EAX into the data segment memory location addressed by ARRAY plus EBX

Register Indirect Addressing: In this addressing mode, a byte or word is transferred between a register and a memory location addressed by an index or base register. The index and base registers are BP, BX, DI, and SI.

The [ ] symbols denote indirect addressing in assembly language.

Example: If register BX contains 1000H and the **MOV AX, [BX]** instruction executes, the word contents of data segment offset address 1000H are copied into register AX.

The **data segment** is used by default with register indirect addressing or any other addressing mode that uses BX, DI, or SI to address memory. The **stack segment** is used by default if the BP register addresses memory.

Examples of register indirect addressing

Assembly Language	**Size**	**Operation**
MOV CX,[BX]	16 bits	Copies the word contents of the data segment memory location addressed by BX into CX
MOV [BP],DL	8 bits	Copies DL into the stack segment memory location addressed by BP
MOV [DI],BH	8 bits	Copies BH into the data segment memory location addressed by DI
MOV [DI],[BX]	—	Memory-to-memory transfers are **not allowed** except with string instructions

RIP Relative Addressing: This mode allows access to any location in the memory system by adding a 32-bit displacement to the 64-bit contents of the 64-bit instruction pointer.

Example, if RIP = 1000000000H and a 32-bit displacement is 300H, the location accessed is 1000000300H. **MOV CX, [BX + SI + 50H]** instruction executes a word from the memory address 33050H and stores in CX.

2. Immediate Addressing Mode:

2. Immediate Addressing Mode: In this addressing mode, the data is immediately transferred from source register to the destination register.

Immediate implies that the data immediately follow the hexadecimal opcode in the memory. Immediate data are constant data, whereas the data transferred from a register or memory location are variable data. Immediate addressing operates upon a byte or word of data.

Example: **MOV EAX, 13456H** This instruction copies 13456H from the instruction located in the memory immediately following the hexadecimal opcode into register EAX.

The letter **H** appends hexadecimal data. If hexadecimal data begin with a letter, the assembler requires that the data start with a **0**. For example, to represent a hexadecimal F2, 0F2H is used in assembly language.

An ASCII-coded character or characters may be depicted in the immediate form if the ASCII data are enclosed in apostrophes.

Example: MOV BH, 'A' moves an ASCII-coded letter A [41H] into register BH.

Binary data are represented if the binary number is followed by the letter B.

Examples of immediate addressing using the MOV instruction

Assembly Language	**Size**	**Operation**
MOV BL,44	8 bits	Copies 44 decimal (2CH) into BL
MOV AX,44H	16 bits	Copies 0044H into AX
MOV AX,'AB'	16 bits	Copies ASCII BA into AX
MOV EAX,100B	32 bits	Copies 100 binary into EAX

3. Direct Data Addressing Mode: In this addressing mode, a byte or word is directly moved between a memory location and a register. The instruction set does not support a memory-to-memory transfer, except with the MOV instruction.

Example: **MOV AL, [1000H]**; EA is given within square bracket in the instruction in this addressing mode and hence EA=1000H in this instruction. Since the destination is an 8-bit register (i.e. AL), a byte is taken from memory at the address given by DS * 10H + EA=31000H and stored in AL.

There are two basic forms of direct data addressing:

a.) **Direct addressing**

b.) **Displacement addressing**

Direct Addressing: Direct addressing with a MOV instruction transfers data between a memory location, located within the data segment, and the AL (8-bit), AX (l6-bit), or EAX (32-bit) register.

A MOV instruction using this type of addressing is usually a 3-byte long instruction.

Example: **MOV AL, DS:[1234H]**

[1234H] is an absolute memory location. The effective address is formed by adding 1234H (the offset address) and 10000H (the data segment address of 1000H times 10H) in a system operating in the real mode.

Examples of Direct addressed instructions

Assembly Language	**Size**	**Operation**
MOV AL,NUM	8 bits	Copies the byte contents of data segment memory location NUM into AL
MOV ABHI,EAX	32 bits	Copies EAX into double-word memory location ABHI
MOV ES:[2000H],AL	8 bits	Copies AL into extra segment memory at offset address 2000H

Displacement Addressing: Displacement addressing is almost identical to direct addressing, except that the instruction is 4 bytes wide instead of 3 bytes.

Examples of direct data addressing using a displacement

Assembly Language	**Size**	**Operation**
MOV CH,SRK	8 bits	Copies the byte contents of data segment memory location SRK into CH
MOV CH,DS:[1000H]	8 bits	Copies the byte contents of data segment memory offset address 1000H into CH

4. Index Addressing: In this mode, the EA is the content of SI or DI register which is specified in the instruction. The data is taken from data segment.

Examples: MOV BL, [SI] instruction takes a byte from the memory address 31000H and stores in BL.

EA = (SI) = 1000H

Memory address = DS * 10H + SI

= 30000H + 1000H

= 31000H

Index Relative Addressing: This mode is same as base relative addressing mode except that instead of BP or BX register, SI or DI register is used.

Example: MOV BX, [SI-100H] instruction takes a word from the memory address 30F00H is taken and stores in BX.

EA = (SI) - 100H

Memory Address = DS x 10H + (SI) - 100H

= 30000H + 1000H - 100H

= 30F00H

Scaled-Index Addressing: Scaled-index addressing uses two 32-bit registers (a base register and an index register) to access the memory. The second register (index) is multiplied by a scaling factor. The scaling factor can be 1x, 2x, 4x or 8x.

Example: **MOV AX, [EDI + 2 x ECX]** This instruction uses a scaling factor of 2x, which multiplies the contents of ECX by 2 before adding it to the EDI register to form the memory address. If ECX contains a 00000000H, word-sized memory element 0 is addressed and if ECX contains a 00000001H, word-sized memory element 1 is accessed. This scales the index (ECX) by a factor of 2 for a word-sized memory array.

Examples of scaled-index addressing

Assembly Language	**Size**	**Operation**
MOV [EAX+2*EDI+100H],CX	16 bits	Copies CX into the data segment memory location addressed by the sum of EAX, 100H, and 2 times EDI
MOV EAX,ARRAY[4*ECX]	32 bits	Copies the double-word contents of the data segment memory location addressed by the sum of ARRAY and 4 times ECX into EAX

5. Base Addressing mode: In this mode, the EA is the content of BX or BP register. When BX register is present in the instruction, data is taken from the data segment and if BP is present in the instruction, data is taken from the stack segment.

Example: Suppose DS = 3000H and BX = 2000H and **MOV CL, [BX]** instruction executes, then a byte from the memory address 32000H is read and stored in CL.

EA = (BX) = 2000H

Memory address = DS x 10 + (BX)

= 32000H

Base Relative Addressing: In this mode, the EA is obtained by adding the content of the base register with an 8-bit or 16 bit displacement.

The displacement is a signed number with negative values represented in 2’s complement form. The 16 bit displacement can have value from -32768 to +32767 and 8 bit displacement can have the value from -128 to +127.

Example: MOV AX, [BX+5] instruction takes a word from the memory address 32005H and stores in AX.

EA = (BX) + 5

Memory address = DS x 10H + (BX) + 5

= 30000H + 2000H + 5

= 32005H

Base-Plus-Index Addressing: Base-plus-index addressing transfers a byte or word between a register and the memory location addressed by a base register (BP or BX) plus an index register (DI or SI).

This type of addressing uses one base register (BP or BX) and one index register (DI or SI) to indirectly address memory. The base register often holds the beginning location of a memory array whereas the index register holds the relative position of an element in the array.

A major use of the base-plus-index addressing mode is to address elements in a memory array.

In this mode, the Effective Address (EA) is obtained by adding the content of a base register and index register.

Example: Suppose DS = 3000H, BX = 2000H and SI = 1000H and **MOV AX, [BX+SI]** instruction executes, then a word from the memory address 33000H is taken and stored in AX.

EA = (BX) + (SI)

Memory address=DS x 10H + (BX) + (SI)

= 30000H + 2000H + 1000H

= 33000H

Examples of base-plus-index addressing

Assembly Language	**Size**	**Operation**
MOV CX,[BX+DI]	16 bits	Copies the word contents of the data segment memory location addressed by BX plus DI into CX
MOV [BX+SI],SP	16 bits	Copies SP into the data segment memory location addressed by BX plus SI

Base-Relative-Plus-Index Addressing: It transfers a byte or word between a register and the memory location addressed by a base, an index register plus a displacement. This type of addressing mode addresses a 2-dimensional array of memory data.

It is used to access a byte or a word in a particular record of a particular file in memory.

In this mode, the EA is obtained by adding the content of a base register, an index and a displacement.

Example: Suppose DS = 3000H, BX = 2000H and SI = 1000H and **MOV CX, [BX + SI + 50H]** instruction executes, then a word from the memory address 33050H is taken and stored in CX.

EA = (BX) + (SI) + 50H

Memory address = DSx10H + (BX) + (SI) + 50H

= 30000H + 2000H + 1000H + 50H

= 33050H

PROGRAM MEMORY ADDRESSING MODES

Program memory-addressing modes, used with the JMP (jump) and CALL instructions, consist of 3 distinct forms:

1. Direct Program Memory Addressing
2. Relative Program Memory Addressing
3. Indirect Program Memory Addressing

Direct Program Memory Addressing

The direct program memory addressing stores both the segment and offset address where the control has to be transferred with the opcode.

Direct program memory addressing is used for all jumps and calls.

Example: when **JMP 32000H** instruction is executed, the 16 bit offset value 2000H is loaded in IP register and the 16 bit segment value 3000H is loaded in CS. When the microprocessor calculates the memory address from where it has to fetch an instruction using the relation CS * 10H + IP, the address 32000H will be obtained using the above CS and IP values.

An **intersegment jump** is a jump to any memory location within the entire memory system.

The direct jump is called a ***far jump*** because it can jump to any memory location for the next instruction.

Relative Program Memory Addressing

The term relative here means relative to the instruction pointer (IP).

An **intrasegment jump** is a jump anywhere within the current code segment.

Relative JMP and CALL instructions contain either an 8 bit or a 16 bit signed displacement that is added to the current instruction pointer and based on the new value of IP thus obtained, the address of the next instruction to be executed is calculated using the relation CS * 10H + IP.

All assemblers automatically calculate the distance for the displacement and select the proper 1-, 2- or 4-byte form.

The 8-bit or 16 bit signed displacement which allows a forward memory reference or a reverse memory reference. An 8 bit displacement has a jump range of between +127 and -128 bytes from the next instruction while a 16 bit displacement has a jump range of between -32768 and +32767 bytes from the next instruction following the jump instruction in the program.

The 32-bit displacement can only be used in the protected mode.

While using assembler to develop 8086 program, the assembler directive SHORT and NEAR PTR is used to indicate short jump and near jump instruction respectively. The opcode of relative **short jump** and **near jump** instructions are **EBH** and **E9H** respectively.

Examples:

a) JMP SHORT TD

b) JMP NEAR PTR ABHI

In the above examples, TD and ABHI are the labels of memory locations that are present in the same code segment in which the above instructions are present.

Indirect Program Memory Addressing

The indirect jump or CALL instructions use either any 16 bit register (AX, BX, CX, DX, SP, BP, SI or DI) or any relative register ([BP], [BX], [DI] or [SI]) or any relative register with displacement.

The opcode of indirect jump instruction is **FFH**.

If a 16-bit register holds the address of a JMP instruction, the jump is near.

Example: if the CX register contains 2000H and **JMP CX** instruction present in a code segment is executed, the microprocessor jumps to offset address 2000H in the current code segment to take the next instruction for execution.

If a relative register holds the address, the jump is also considered to be an indirect jump.

Example: JMP [BX] instruction refers to the memory location within the data segment at the offset address contained in BX. The offset address is a 16-bit number that is used as the offset address in the intrasegment jump. This type of jump is sometimes called an indirect-indirect or double-indirect jump.

Example: when the instruction **JMP [DI]** is executed, the microprocessor first reads a word in the current data segment from the offset address specified by DI and puts that word in the IP register. Now using this new value of IP, 8086 calculates the address of the memory location where it has to jump using the relation CS * 10H + IP.

STACK MEMORYADDRESSING MODES

The stack holds temporarily data and also stores return address for procedures and interrupt service routines. The stack memory is a last-in-first-out (LIFO) memory. Data are placed into the stack using PUSH instruction and taken out from the stack using POP instruction. The CALL instruction uses the stack to hold the return address for procedures and RET instruction is used to remove return address from stack.

The stack memory is maintained by two registers: the **stack pointer** and the **stack segment**. Always a word is entered into stack. Whenever a word of data is pushed onto the stack, the high-order 8 bits are placed in the location addressed by SP – 1. The low-order 8 bits are placed in the location addressed by SP – 2. The SP is then decremented by 2.

The SP/ESP register always points to an area of memory located within the stack segment. The SP/ESP register adds to SS * 10H to form the stack memory address in the real mode. In protected mode operation, the SS register holds a selector that accesses a descriptor for the base address of the stack segment.

Whenever data are popped from the stack, the low-order 8 bits are removed from the location addressed by SP. The high-order 8 bits are removed from the location addressed by SP + 1. The SP register is then incremented by 2.

Since SP gets decremented for every push operation, the stack segment is said to be growing downwards as for successive push operations, the data are stored in lower memory addresses in stack segment. Due to this, the SP is initialized with highest offset address according to the requirement, at the beginning of the program.

Data may be popped off the stack into any register or any segment register except CS. The reason that data may not be popped from the stack into CS is that this only changes part of the address of the next instruction.

The PUSHA and POPA instructions either push or pop all of the registers, except segment registers, onto the stack.

Example PUSH and POP instructions

Assembly Language	**Operation**
POPF	Removes a word from the stack and places it into the flag register
POPFD	Removes a double-word from the stack and places it into the EFLAG register
PUSHF	Copies the flag register to the stack
PUSHFD	Copies the EFLAG register to the stack
PUSH AX	Copies the AX register to the stack
POP BX	Removes a word from the stack and places it into the BX register
POP CS	This instruction is illegal
PUSHA	Copies AX, CX, DX, BX, SP, BP, DI, and SI to the stack
POPA	Removes the word contents for the following registers from the stack: SI, DI, BP, SP, BX, DX, CX, and AX

Segment Override Prefix

The segment override prefix allows the programmer to deviate from the default segment and offset register mechanism.

The jump and call instructions cannot be prefixed with the segment override prefix since they use only code segment register (CS) for address generation.

Example: MOV AX, [BP] instruction accesses data within stack segment by default since BP is the offset register for stack segment. But if the programmer wants to get data from data segment using BP as offset register in the above instruction, then the instruction is modified as **MOV AX, DS: [BP]**.

Data Movement Instruction

Machine Language Instruction

Machine language is the native binary code that the microprocessor understands and uses as its instructions to control its operation. Machine language instructions vary in length from 1 byte to 13 bytes.

Opcode 1–2 bytes	MOD-REG-R/M 0–1 bytes	Displacement 0–1 bytes	Immediate 0–2 bytes

(a) 16-bit instruction mode

Address size 0–1 bytes	Register size 0–1 bytes	Opcode 1–2 bytes	MOD-REG-R/M 0–1 bytes	Scaled-index 0–1 bytes	Displacement 0–4 bytes	Immediate 0–4 bytes

(b) 32-bit instruction mode

Formats of the 8086–Core2 instructions

The first 2 bytes of the 32-bit instruction mode format are called **override prefixes** because they are not always present. The first modifies the size of the operand address used by the instruction and the second modifies the register size. If the microprocessor operates in 16-bit instruction mode with a 32-bit register, the **register-size prefix** (66H) is appended to the front of the instruction.

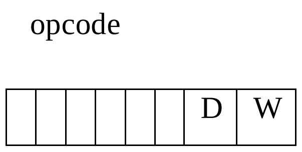
Address size-prefix (67H)

The l6-bit instruction mode uses 8- and l6-bit registers and addressing modes, while the 32-bit instruction mode uses 8- and 32-bit registers and addressing modes by default. The prefixes override these defaults so that a 32-bit register can be used in the l6-bit mode or a l6-bit register can be used in the 32-bit mode.

The **mode of operation** (16 or 32 bits) should be selected to function with the current application. Mode selection is a function of the operating system.

Note: DOS can operate only in the l6-bit mode, where Windows can operate in both modes.

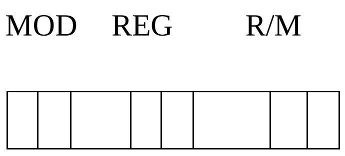
Opcode: The opcode selects the operation that is performed by the microprocessor. The opcode is either 1 or 2 bytes long for most machine language instructions.



Byte 1 of many machine language instructions

The first 6 bits of the first byte are the binary opcode. The remaining 2 bits indicate the **direction** (D) of the data flow, and indicate whether the data are a byte or a word (W). In the 80386 and above, words and double-words are both specified when W = 1. The instruction mode and register-size prefix (66H) determine whether W represents a word or a double-word.

If the direction bit D = 1, data flow to the register REG field from the R/M field located in the second byte of an instruction. If D = 0 in the opcode, data flow to the R/M field from the REG field. If W = 1, the data size is a word or double-word and if W = 0, the data size is always a byte.



Byte 2 of many machine language instructions

MOD Field: The MOD field specifies the addressing mode (MOD) for the selected instruction. The MOD field selects the type of addressing and whether a displacement is present with the selected type.

MOD field for the 16-bit instruction mode

MOD	Function
00	No displacement
01	8-bit sign-extended displacement
10	16-bit signed displacement
11	R/M is a register

Example: MOV AL, [DI] instruction is an example that contains no displacement, **MOV AL,[DI + 2]** instruction uses an 8-bit displacement (+2), and **MOV AL,[DI + 1000H]** instruction uses a 16-bit displacement (+1000H).

All 8-bit displacements are sign-extended into 16-bit displacements when the microprocessor executes the instruction.

(a) If the 8-bit displacement is 00H–7FH (positive), it is sign extended to 0000H–007FH before adding to the offset address.

(b) If the 8-bit displacement is 80H–FFH (negative), it is sign-extended to FF80H–FFFFH.

(c) To sign-extend a number, its sign-bit is copied to the next higher-order byte, which generates either a 00H or an FFH in the next higher-order byte.

MOD field for the 32-bit instruction mode

MOD	Function
00	No displacement
01	8-bit sign-extended displacement
10	32-bit signed displacement
11	R/M is a register

MOD field is interpreted as selected by the address-size override prefix or the operating mode of the microprocessor.

When the MOD field is a 10, this causes the 16-bit displacement to become a 32-bit displacement, to allow any protected mode memory location (4G bytes) to be accessed.

If an 8-bit displacement is selected, it is sign-extended into a 32-bit displacement by the microprocessor.

Register Assignments: Suppose that a 2-byte instruction, **8BECH**, appears in a machine language program and because neither a 67H nor a 66H appears as the first byte, the first byte is the opcode. The opcode is 100010 i.e. opcode of MOV instruction.

If the microprocessor is operated in the 16-bit instruction mode, this instruction is converted to binary and placed in the instruction format of bytes 1 and 2.

opcode						D	W
1	0	0	0	1	0	1	1

MOD		REG			R/M		
1	1	1	0	1	1	0	0

Both the D and W bits are a logic 1, which means that a word moves into the destination register specified in the REG field. The REG field contains a 101, indicating register BP, so the MOV instruction moves data into register BP. The MOD field contains 11, the R/M (R/M = 100) field also indicates a register (SP). Therefore, the instruction is **MOV BP, SP** that moves data from SP into BP.

Example: Suppose that a 668BE8H instruction is operated in the 16-bit instruction mode. The first byte (66H) is the register-size override prefix that selects 32-bit register operands for the 16-bit instruction mode. The remainder of the instruction indicates that the opcode is a MOV with a source operand of EAX and a destination operand of EBP. This instruction is a MOV EBP, EAX. The same instruction becomes a MOV BP, AX instruction if it is operated in the 32-bit instruction mode because the register-size override prefix selects a 16-bit register.

REG and R/M (when) MOD = 11 assignments

Code	W = 0 (Byte)	W = 1 (Word)	W = 1 (Doubleword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

R/M Memory Addressing: If MOD = 00 and R/M = 101, the addressing mode is [DI]. If MOD = 01 or 10, the addressing mode is [DI + 33H] or LIST [DI + 22H] for the 16-bit instruction mode.

Example: MOV DL, [DI] instruction is 2 bytes long and has an opcode 100010, D = 1 (to REG from R/M), W = 0 (byte), MOD = 00 (no displacement), REG = 010 (DL), and R/M = 101 ([DI]).

opcode D W

1	0	0	0	1	0	1	0

MOD REG R/M

0	0	0	1	0	1	0	1

16-bit R/M memory-addressing modes

R/M Code	Addressing Mode
000	DS : 3BX + SI4
001	DS : 3BX + DI4
010	SS : 3BP + SI4
011	SS : 3BP + DI4

100	DS : [SI]
101	DS : [DI]
110	SS : [BP]
111	DS : [BX]

Special Addressing Mode: It occurs whenever memory data are referenced by only the displacement mode of addressing for 16-bit instructions.

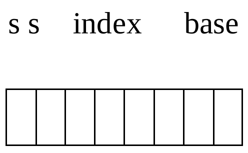
Examples are MOV [1000H], DL and MOV NUMB, DL instructions. Whenever an instruction has only a displacement, the MOD field is always a 00 and the R/M field is always 110.

32-Bit Addressing Modes: The 32-bit addressing modes found in the 80386 and above are obtained by either running these machines in the 32-bit instruction mode or in the 16-bit instruction mode by using the address-size prefix 67H.

When R/M = 100, an additional byte called a scaled-index byte appears in the instruction.

The scaled-index byte is mainly used when two registers are added to specify the memory address in an instruction. Because the scaled-index byte is added to the instruction, there are 7 bits in the opcode and 8 bits in the scaled-index byte to define. This means that a scaled-index instruction has 215 (32K) possible combinations.

Scaled index byte



Above figure shows the format of the scaled-index byte as selected by a value of 100 in the R/M field of an instruction when the 80386 and above use a 32-bit address. The leftmost 2 bits select a scaling factor (multiplier) of 1*, 2*, 4*, 8*. A scaling factor is implicit if none is used in an instruction that contains two 32-bit indirect address registers.

Example: The instruction **MOV EAX, [EBX + 4*ECX]** is encoded as **67668B048BH**. Both the address size (67H) and register size (66H) override prefixes appear in this instruction.

Immediate Instruction: MOV WORD PTR [BX + 1000H], 1234H instruction moves a 1234H into the word-sized memory location addressed by the sum of 1000H, BX, and DS * 10H. This 6-byte instruction uses 2 bytes for the opcode, W, MOD, and R/M fields. Two of the 6 bytes are the data of 1234H and rest of the 2 of 6 bytes are for the displacement of 1000H.

Segment MOV Instructions: If the contents of a segment register are moved by the MOV, PUSH, or POP instructions, a special set of register bits (REG field) selects the segment register.

Segment register selection

Code Segment	Register
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

Note: MOV CS, R/M and POP CS are not allowed.

Example: MOV BX, CS instruction converted to binary machine language

opcode D W

1	0	0	0	1	1	0	0

MOD REG R/M

1	1	0	0	1	0	1	1

64-Bit Mode: In the 64-bit mode, an additional prefix called REX (*register extension*) is added. The REX prefix, which is encoded as a 40H–4FH, follows other prefixes and is placed immediately before the opcode to modify it for 64-bit operation. The purpose of the REX prefix is to modify the REG and R/M fields in the second byte of the instruction. REX is needed to be able to address registers R8 through R15.

The REG field can only contain register assignments as in other modes of operation and the R/M field contains either a register or memory assignment.

As with 32-bit instructions, the modes allowed by the scaled-index byte are fairly all conclusive allowing pairs of registers to address memory and also an index factor of 2*, 4* or 8*.

Example: MOV RAXW, [RDX + RCX – 12] instruction requires the scaled-index byte with an index of 1.

The 64-bit register and memory designators for rrrr and mmmm

Code	**Register**	**Memory**
0000	RAX	[RAX]
0001	RCX	[RCX]
0010	RDX	[RDX]
0011	RBX	[RBX]
0100	RSP	This addressing mode specifies the inclusion of the scaled-index byte
0101	RBP	[RBP]
0110	RSI	[RSI]
0111	RDI	[RDI]
1000	R8	[R8]
1001	R9	[R9]
1010	R10	[R10]
1011	R11	[R11]
1100	R12	[R12]
1101	R13	[R13]
1110	R14	[R14]
1111	R15	[R15]

PUSH/POP

The PUSH and POP instructions stores and retrieves data from the LIFO (last-in-first-out) stack memory.

The microprocessor has 6 forms of the PUSH and POP instructions: register, memory, immediate, segment register, flags, and all registers.

1. **Register addressing:** It allows the contents of any 16-bit register to be transferred to or from the stack.
2. **Memory-addressing:** It stores the content of a 16-bit memory location on the stack into a memory location.
3. **Immediate addressing:** It allows immediate data to be pushed onto the stack, but not popped off the stack.
4. **Segment register addressing:** It allows the contents of any segment register to be pushed onto the stack or removed from the stack
5. **Flags:** It may be pushed or popped from the stack.
6. **All register**: It's content may be pushed or popped from the stack.

PUSH

PUSH instruction always transfers 2 bytes of data to the stack. The source of the data may be any internal 16- or 32-bit register, immediate data, any segment register, or any 2 bytes of memory data.

PUSHF (**Push Flags**) instruction copies the contents of the flag register to the stack.

PUSHAD and **POPAD** instructions push and pop the contents of the 32-bit register set.

Whenever data are pushed onto the stack, the first (most-significant) data byte moves to the stack segment memory location addressed by SP – 1 and the second (least-significant) data byte moves into the stack segment memory location addressed by SP - 2. After the data are stored by a PUSH, the contents of the SP register decrement by 2.

PUSH AX instruction copies the contents of AX onto the stack.

PUSHA (Push All) instruction copies the contents of the internal register set, except the segment registers, to the stack. This instruction copies the registers to the stack in the following order: **AX, CX, DX, BX, SP, BP, SI, and DI**.

PUSHA instruction pushes all the internal 16-bit registers onto the stack. This instruction requires 16 bytes of stack memory space to store all eight 16-bit registers. After all registers are pushed, the contents of the SP register are decremented by 16.

PUSHA instruction is very useful when the entire register set must be saved during a task.

PUSH immediate data instruction has 2 different opcodes, but in both cases, a 16-bit immediate number moves onto the stack and if PUSHD is used, a 32-bit immediate number is pushed. If the values of the immediate data are 00H–FFH, the opcode is a 6AH and if the data are 0100H–FFFFH, the opcode is 68H.

Example of PUSH immediate is the PUSH ‘A’ instruction, which pushes a 0041H onto the stack.

POP

POP instruction removes data from the stack and places it into the target 16-bit register, segment register, or a 16- bit memory location.

POP instruction is not available as an immediate POP.

POPF (**Pop Flags**) instruction removes a 16-bit number from the stack and places it into the flag register.

POPFD removes a 32-bit number from the stack and places it into the extended flag register.

POPA (**Pop All**) instruction removes 16 bytes of data from the stack and places them into the following registers, in the order shown: DI, SI, BP, SP, BX, DX, CX, and AX. This is the reverse order from the way they were placed on the stack by the PUSHA instruction, causing the same data to return to the same registers.

Example: Suppose that a **POP BX** instruction executes. The first byte of data removed from the stack moves into register BL and the second byte moves into register BH. After both bytes are removed from the stack, the SP register is incremented by 2.

POP CS instruction is not a valid instruction in the instruction set. If a POP CS instruction executes, only a portion of the address (CS) of the next instruction changes. This makes the POP CS instruction unpredictable and therefore not allowed.

Initializing the Stack

When the stack area is initialized, load both the stack segment (SS) register and the stack pointer (SP) register. It is normal to designate an area of memory as the stack segment by loading SS with the bottom location of the stack segment.

Example: if the stack segment is to reside in memory locations 10000H–1FFFFH, load SS with a 1000H. Remember that the rightmost end of the stack segment register is appended with a 0H for real mode addressing. To start the stack at the top of this 64K-byte stack segment, the stack pointer (SP) is loaded with a 0000H. Similarly to address the top of the stack at location 10FFFH, use a value of 1000H in SP.

Note: All segments are cyclic in nature i.e. the top location of a segment is contiguous with the bottom location of the segment.

LOAD EFFECTIVE ADDRESS

There are several load-effective address instructions in the microprocessor instruction set. Some of them are LEA, LDS, LES, LSS, LFS and LGS.

LEA

LEA instruction loads any 16-bit register with the offset address, as determined by the addressing mode selected for the instruction.

In other words, this instruction determines the offset address of the variable or memory location named as the source and puts this offset address in the indicated 16 bit register.

The general format of LEA instruction is **LEA register, source.**

Example: LEA BX, ABHI instruction loads BX with offset address of ABHI in data segment where ABHI is the name assigned to a memory location in data segment.

Example: LEA AX, [BX][SI] instruction loads AX with the value equal to (BX)+(SI) where (BX) and (SI) represents content of BX and SI respectively.

Comparison of LEA with MOV instruction,

By comparing LEA with MOV, we observe that **LEA BX,[DI]** loads the offset address specified by [DI] (contents of DI) into the BX register; **MOV BX,[DI]** loads the data stored at the memory location addressed by [DI] into register BX. The OFFSET directive performs the same function as an LEA instruction if the operand is a displacement.

Example: MOV BX, OFFSET COST performs the same function as **LEA BX, COST**. Both instructions load the offset address of memory location LIST into the BX register.

LDS

LDS instruction load any 16-bit or 32-bit register with an offset address and DS segment register with a segment address.

In other words, this instruction copies a word from the memory location specified in the instruction into the register and then copies a word from the next memory location into the DS register.

The general form of LDS instruction is **LDS register, memory address of first word**

LDS is useful for initializing SI and DS registers at the start of a string before using one of the String instructions.

Example: LDS SI,[2000H] instruction copies the content of memory at offset address 2000H in data segment to lower byte of SI, content of 2001H to higher byte of SI, content of 2002H to lower byte of DS and 2003H to higher byte of DS.

LDS instruction is only available in 32-bit register.

LDS instruction use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. This instruction may not use the register addressing mode (MOD = 11).

In the 64-bit mode, LDS instructions are *invalid* and not used because the segments have no function in the flat memory model.

LES

LES instruction load any 16-bit or 32-bit register with an offset address and ES segment register with a segment address.

The general form of LES instruction is **LES register, memory address of first word**

LES instruction is only available in 32-bit register.

LES instruction use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. This instruction may not use the register addressing mode (MOD = 11).

In the 64-bit mode, LES instructions are *invalid* and not used because the segments have no function in the flat memory model.

LSS

LSS instruction load any 16-bit or 32-bit register with an offset address and SS segment register with a segment address.

The general form of LSS instruction is **LSS register, memory address of first word**

LSS instruction is only available in 32-bit register.

LSS instruction use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. This instruction may not use the register addressing mode (MOD = 11).

LFS

LFS instruction load any 16-bit or 32-bit register with an offset address and FS segment register with a segment address.

LFS instruction is only available in 32-bit register.

LFS instruction use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. This instruction may not use the register addressing mode (MOD = 11).

LGS

LGS instruction load any 16-bit or 32-bit register with an offset address and GS segment register with a segment address.

LGS instruction is only available in 32-bit register.

LGS instruction use any of the memory-addressing modes to access a 32-bit or 48-bit section of memory that contains both the segment and offset address. This instruction may not use the register addressing mode (MOD = 11).

The LDS, LES, LFS, LGS, and LSS instructions obtain a new **far address** from memory. The offset address appears first, followed by the segment address. This format is used for storing all 32-bit memory addresses.

A far address can be stored in memory by the assembler.

Example: ADDR DD FAR PTR HORSE instruction stores the offset and segment address (far address) of HORSE in 32 bits of memory at location ADDR. The DD directive tells the assembler to store a double-word (32-bit number) in memory address ADDR.

Q: Why is the LEA instruction available if the OFFSET directive accomplishes the same task?

A: OFFSET only functions with simple operands such as LIST. It may not be used for an operand such as [DI], LIST [SI], and so on. The OFFSET directive is more efficient than the LEA instruction for simple operands. It takes the microprocessor longer to execute the **LEA BX, LIST** instruction than the **MOV BX, OFFSET LIST** because the assembler calculates the offset address of LIST, whereas the microprocessor calculates the address for the LEA instruction.

Suppose that the microprocessor executes an LEA BX,[DI] instruction and DI contains a 1000H. Because DI contains the offset address, the microprocessor transfers a copy of DI into BX. Thus, **MOV BX, DI** instruction performs this task in less time and is often preferred to the **LEA BX,[DI]** instruction.

STRING DATA TRANSFERS

There are 5 string data transfer instructions: LODS, STOS, MOVS, INS, and OUTS. Each string instruction allows data transfers that are single byte, word, or double-word.

Direction Flag

Direction flag (D) selects the auto-increment (D = 0) or the auto decrement (D = 1) operation for the DI and SI registers during string operations.

The direction flag is used only with the string instructions.

The CLD instruction clears the D flag (D = 0) and the STD instruction sets it (D = 1). Therefore, the CLD instruction selects the auto-increment mode (D = 0) and STD selects the auto-decrement mode (D = 1).

Whenever a string instruction transfers a byte, the contents of DI and/or SI are incremented or decremented by 1. If a word is transferred, the contents of DI and/or SI are incremented or decremented by 2. Double-word transfers cause DI and/or SI to increment or decrement by 4.

Only the actual registers used by the string instruction are incremented or decremented.

Example: STOSB instruction uses the DI register to address a memory location. When STOSB executes, only the DI register is incremented or decremented without affecting SI register. The same is true for **LODSB** instruction, which uses the SI register to address memory data. A LODSB instruction will only increment or decrement SI without affecting DI register.

DI and SI

During the execution of a string instruction, memory accesses occur through either or both of the DI and SI registers.

The DI offset address accesses data, by default, in the extra segment while the SI offset address accesses data, by default, in the data segment for all string instructions that use it.

The segment assignment of SI may be changed with a segment override prefix. The DI segment assignment is always in the extra segment when a string instruction executes. This assignment cannot be changed.

LODS

The LODS instruction loads AL, AX, or EAX with data stored at the data segment offset address indexed by the SI register. After loading AL with a byte, AX with a word, or EAX with a double-word, the contents of SI increment, if D = 0 or decrement, if D = 1.

Forms of the LODS instruction

Assembly Language	**Operation**
LODSB	AL = DS:[SI]; SI = SI ± 1
LODSW	AX = DS:[SI]; SI = SI ± 2
LODSD	EAX = DS:[SI]; SI = SI ± 4
LODSQ	RAX = [RSI]; RSI = RSI ± 8 (64-bit mode)
LODS DATA1	AL = DS:[SI]; SI = SI ± 1 (if DATA1 is a byte)
LODS DATA2	AX = DS:[SI]; SI = SI ± 2 (if DATA2 is a word)
LODS DATA3	EAX = DS:[SI]; SI = SI ± 4 (if DATA3 is a double-word)

STOS

The STOS instruction stores AL, AX, or EAX at the extra segment memory location addressed by the DI register. STOS instruction may be appended with a B, W, or D for byte, word, or double-word transfers.

Repeat Prefix (**REP**) is added to any string data transfer instruction, except the LODS instruction. REP prefix causes CX to decrement by 1 each time the string instruction executes. After CX decrements, the string instruction repeats. If CX reaches a value of 0, the instruction terminates and the program continues with the next sequential instruction. Thus, if CX is loaded with 100 and a REP STOSB instruction executes, the microprocessor automatically repeats the STOSB instruction 100 times. Because the DI register is automatically incremented or decremented after each datum is stored, this instruction stores the contents of AL in a block of memory instead of a single byte of memory.

Forms of the STOS instruction

Assembly Language	**Operation**
STOSB	ES:[DI] = AL; DI = DI ± 1
STOSW	ES:[DI] = AX; DI = DI ± 2
STOSD	ES:[DI] = EAX; DI = DI ± 4
STOSQ	[RDI] = RAX; RDI = RDI ± 8 (64-bit mode)
STOS DATA1	ES:[DI] = AL; DI = DI ± 1 (if DATA1 is a byte)
STOS DATA2	ES:[DI] = AX; DI = DI ± 2 (if DATA2 is a word)
STOS DATA3	ES:[DI] = EAX; DI = DI ± 4 (if DATA3 is a double-word)

The operands in a program can be modified by using arithmetic or logic operators.

Common operand modifiers

Operator	**Example**	**Comment**
+	MOV AL,6+2	Copies 8 into AL
-	MOV BL,6-4	Copies 2 into BL
*	MOV AL,2*3	Copies 6 into AL
/	MOV BX,12/5	Copies 2 into BX (remainder is lost)
MOD	MOV CX,12 MOD 7	Copies 5 into CX (quotient is lost)
AND	MOV DX,12 AND 4	Copies 4 into DX (1100 AND 0100 = 0100)
OR	MOV EAX,12 OR 1	Copies 13 into EAX (1100 OR 0001 = 1101)
NOT	MOV AL,NOT 1	Copies 254 into AL (NOT 0000 0001 = 1111 1110)

MOVS

MOVS instruction transfers data from one memory location to another. It transfers a byte, word, or double-word from the data segment location addressed by SI to the extra segment location addressed by SI.

Only the source operand (SI), located in the data segment, may be overridden so that another segment may be used. The destination operand (DI) must always be located in the extra segment.

Forms of the MOVS instruction

Assembly Language	**Operation**
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (double-word transferred)
MOVSQ	[RDI] = [RSI]; RDI = RDI ± 8; RSI = RSI ± 8 (64-bit mode)
MOVS BYTE1, BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred if WORD1 and WORD2 are words)
MOVS DWORD1,DWORD2	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (double-word transferred if DWORD1 and DWORD2 are double-words)

INS

INS instruction transfers a byte, word, or double-word of data from an I/O device into the extra segment memory location addressed by the DI register. The I/O address is contained in the DX register.

This instruction is useful for inputting a block of data from an external I/O device directly into the memory.

There are 3 basic forms of the INS.

1. **INSB** instruction inputs data from an 8-bit I/O device and stores it in the byte-sized memory location indexed by SI.
2. **INSW** instruction inputs 16-bit I/O data and stores it in a word-sized memory location.
3. **INSD** instruction inputs a double-word.

These instructions can be repeated using the REP prefix, which allows an entire block of input data to be stored in the memory from an I/O device.

Forms of the INS instruction

Assembly Language	**Operation**
INSB	ES:[DI] = [DX]; DI = DI ± 1 (byte transferred)
INSW	ES:[DI] = [DX]; DI = DI ± 2 (word transferred)
INSD	ES:[DI] = [DX]; DI = DI ± 4 (double-word transferred)
INS BYTE1	ES:[DI] = [DX]; DI = DI ± 1 (if BYTE1 is a byte)
INS WORD1	ES:[DI] = [DX]; DI = DI ± 2 (if WORD1 is a word)
INS DWORD1	ES:[DI] = [DX]; DI = DI ± 4 (if DWORD1 is a double-word)

OUTS

OUTS instruction transfers a byte, word, or double-word of data from the data segment memory location address by SI to an I/O device. The I/O device is addressed by the DX register as it is with the INS instruction.

Forms of the OUTS instruction

Assembly Language	**Operation**
OUTSB	[DX] = DS:[SI]; SI = SI ± 1 (byte transferred)
OUTSW	[DX] = DS:[SI]; SI = SI ± 2 (word transferred)
OUTSD	[DX] = DS:[SI]; SI = SI ± 4 (double-word transferred)
OUTS BYTE1	[DX] = DS:[SI]; SI = SI ± 1 (if BYTE1 is a byte)
OUTS WORD1	[DX] = DS:[SI]; SI = SI ± 2 (if WORD1 is a word)
OUTS DWORD1	[DX] = DS:[SI]; SI = SI ± 4 (if DWORD1 is a double-word)

MISCELLANEOUS DATA TRANSFER INSTRUCTIONS

Some miscellaneous data transfer instructions are XCHG, LAHF, SAHF, XLAT, IN, OUT, BSWAP, MOVSX, MOVZX, and CMOV.

XCHG

XCHG (exchange) instruction exchanges the contents of a register with the contents of any other register or memory location. It cannot exchange directly the contents of two memory locations. Exchanges are byte-, word-, or doubleword-sized and they use any addressing mode except immediate addressing mode. The segment registers cannot be used in this instruction.

Example: XCHG AL, BL instruction exchanges content of AL and BL and **XCHG AX, [BX]** instruction exchanges content of AX with content of memory at [BX].

LAHF

LAHF instruction transfers the low byte (rightmost 8 bits) of the flag register into the AH register.

This instruction does not affect the flag registers.

SAHF

SAHF instruction transfers the AH register into the low byte (rightmost 8 bits) of the flag register.

This instruction affects the flag registers.

XLAT

XLAT (translate) instruction converts the content of the AL register into a number stored in a memory table. This instruction is used to translate a byte in AL from one code to another code. An XLAT instruction first adds the contents of AL to BX to form a memory address within the data segment. It then copies the contents of this address into AL. This is the only instruction that adds an 8-bit number to a 16-bit number.

IN/OUT

IN instruction transfers data from an external I/O device (port) into AL, AX, or EAX whereas OUT instruction transfers data from AL, AX, or EAX to an external I/O device (port).

Both IN and OUT instructions have 2 forms of I/O device (port): fixed port and variable port.

Fixed-port addressing: It allows data transfer between AL, AX, or EAX using an 8-bit I/O port address. It is called *fixed-port addressing* because the port number follows the instruction's opcode. A fixed port instruction stored in ROM has its port number permanently fixed. With this form, anyone of 256 possible ports can be addressed.

Example: IN AL, 80H instruction input a byte from the port with address 80H to AL.

Variable-port addressing: It allows data transfers between AL, AX, or EAX and a 16-bit port address. It is called *variable-port addressing* because the I/O port number is stored in register DX, which can be changed during the execution of a program. Since DX is a 16 bit register, the port address can be any number between 0000H and FFFFH. Hence up to 65536 ports are addressable in this mode.

Example: MOV DX, 0FE50H instruction initializes DX with port address of FE50H and **IN AX, DX** instruction input a word from 16-bit port with port address FE50H into AX.

MOVSX and MOVZX

These instructions move data and at the same time either sign- or zero- extend it.

When a number is zero-extended, the most significant part fills with zeros.

Zero-extension is used to convert unsigned 8- or 16-bit numbers into unsigned 16- or 32-bit numbers by using the MOVZX instruction.

Example: if an 8-bit 34H is zero-extended into a 16-bit number, it becomes 0034H.

When a number is sign-extended, its sign-bit is copied into the most significant part.

Sign extension is used to convert 8- or 16-bit signed numbers into 16- or 32-bit signed numbers by using the MOVSX instruction.

Example: if an 8-bit 84H is sign-extended into a 16-bit number, it becomes FF84H. The sign-bit of an 84H is 1, which is copied into the most significant part of the sign-extended result.

BSWAP

This instruction takes the contents of any 32-bit register and swaps first byte with fourth byte and second byte with the third byte.

Example: BSWAP EAX instruction with EAX = 00112233H swaps bytes in EAX, resulting in EAX = 33221100H.

CMOV

This instruction (conditional move) moves the data only if the condition is true.

Example: CMOVZ instruction moves data only if the result from some prior instruction was a zero.

Conditional move instructions

Assembly Language	Flag(s) Tested	Operation
CMOVB	C = 1	Move if below
CMOVAE	C = 0	Move if above or equal
CMOVBE	Z = 1 or C = 1	Move if below or equal
CMOVA	Z = 0 and C = 0	Move of above
CMOVE or CMOVZ	Z = 1	Move if equal or move if zero
CMOVNE or CMOVNZ	Z = 0	Move if not equal or move if not zero
CMOVL	S ! = O	Move if less than
CMOVLE	Z = 1 or S ! = O	Move if less than or equal
CMOVG	Z = 0 and S = O	Move if greater than
CMOVGE	S = O	Move if greater than or equal
CMOVS	S = 1	Move if sign (negative)
CMOVNS	S = 0	Move if no sign (positive)
CMOVC	C = 1	Move if carry
CMOVNC	C = 0	Move if no carry
CMOVO	O = 1	Move if overflow
CMOVNO	O = 0	Move if no overflow
CMOVP or CMOVPE	P = 1	Move if parity or move if parity even
CMOVNP or CMOVPO	P = 0	Move if no parity or move if parity odd

Arithmetic and Logic Instructions

Arithmetic Instructions

The arithmetic instructions include addition, subtraction, multiplication, division, comparison, negation, increment, and decrement.

Whenever arithmetic and logic instructions execute, the contents of the flag register change while the contents of the interrupt, trap, and other flags do not change.

Only the flags located in the rightmost 8 bits of the flag register and the overflow flag changes. These rightmost flags denote the result of the arithmetic or a logic operation.

ADD (Addition): This instruction adds the data from the source and destination and the result is placed in the destination.

The source can be an immediate number, a register or a memory location while the destination can be a register or memory location.

But the source and destination both cannot be memory locations.

The data from the source and destination must be of the same type either bytes or words.

Example: ADD AX, CX It adds the content of AX and CX and stores the result in AX.

ADD AL, [BX] It adds the content of AL and byte from memory at [BX] and store result in AL.

AF, CF, OF, PF, SF and ZF flags are affected by the execution of ADD instruction.

Memory-to-memory and segment register additions are not allowed.

ADC (Addition with carry): This instruction adds the data in source and destination along with the content of carry flag and stores the result in the destination.

Example: ADC CX, BX It adds the content of CX, BX and the carry flag and stores result in CX.

ADI (Add Immediate): This addition instruction adds the immediate data from source, which is a constant, to destination and stores the result in the destination.

Example: ADI DL, 15H It adds the content of DL and 15H and stores the result in DL.

ACI (Add Immediate with carry): This instruction adds the immediate data from source to destination along with carry and stores the result in the destination.

Example: ACI DL, 43H It adds the content of DL, 43H and carry and stores result in DL.

INC (Increment Addition): This instruction adds 1 to any register or memory location, except a segment register.

Example: INC BX It adds 1 to the content of BX and stores the result in BX.

XADD (Exchange and Add): This instruction adds the source to destination and stores the result in the destination while copying the original value of destination into source.

Example: If BX = 15H and CX = 28H and **XADD BX, CX** instruction executes, the BX register contains the sum of 43H and CX becomes 15H.

SUB (Subtraction): This instruction subtracts the content of source from the content of destination and stores the result in destination.

For subtraction, the carry flag (CF) functions as borrow flag. If the result is negative after subtraction, CF is set, otherwise it is reset.

AF, CF, OF, PF, SF and ZF flags are affected by SUB instruction.

Example: SUB AX, BX It subtracts the content of BX from AX and stores the result in AX.

SBB (Subtraction with Borrow): This instruction subtracts the content of source and content of carry flag from the content of destination and stores the result in destination.

Example: SBB AX, BX It subtracts the content of BX along with the content of carry flag from AX and stores the result in AX.

SUI (Subtract Immediate): This instruction subtracts the data of source, which is a constant, from the destination and stores the result in destination.

Example: SUI DL, 15H It subtracts the content of 15H from DL and stores result in DL.

SBI (Subtract Immediate with Borrow): This instruction subtracts the immediate data of source as well as content of carry flag from destination and stores the result in destination.

Example: SBI DL, 43H It subtracts 43H and carry flag from DL and stores result in DL.

DEC (Decrement Subtraction): This instruction subtracts 1 from a register or the contents of a memory location.
Example: DEC BX It subtracts 1 from the content of BX and stores the result in BX.

Program Control Instructions

JUMP GROUP

The main program control instruction, **jump** (JMP), allows the programmer to skip sections of a program and branch to any part of the memory for the next instruction.

It includes 2 types of jump instructions:

1. Unconditional Jump
2. Conditional Jump

Unconditional Jump (JMP)

There are 3 types of unconditional jump instructions:

a. Short Jump
b. Near Jump
c. Far Jump

Short Jump

Short jump is a 2-byte instruction that allows jumps or branches to memory locations within +127 and –128 bytes from the address following the jump.

Short jumps are often called as **Intra-segment Jumps**.

Short jumps are called **relative jumps** because they can be moved to any location in the current code segment without a change.

Short jump stores a **distance or displacement** which follows the opcode instead of the jump address. The short jump displacement is a distance represented by a 1-byte signed number whose value ranges between +127 and -128.

When the microprocessor executes a short jump, the displacement is sign-extended and added to the instruction pointer (IP/EIP) to generate the jump address within the current code segment.

A **label** is a symbolic name for a memory address. Whenever a jump instruction references an address, a label normally identifies the address. For example, **JMP NEXT** instruction jumps to label NEXT for the next instruction.

Near Jump

Near jump is a 3-byte instruction that allows a branch or jump within ±32K bytes (or anywhere in the current code segment) from the instruction in the current code segment.

Near jumps are also often called as **Intra-segment Jumps**.

The near jump contains an opcode followed by a signed 16-bit displacement. The signed displacement adds to the instruction pointer (IP) to generate the jump address.

Near jump is also **relocatable** because it is also a relative jump means that if the code segment moves to a new location in the memory then the distance between the jump instruction and the operand address remains the same.

Note: Segments are cyclic in nature, which means that one location above offset address FFFFH is offset address 0000H. So, if we jump 2 bytes ahead in memory and the instruction pointer addresses offset address FFFFH then the flow continues at offset address 0001H.

Far Jump

Far jump is a 5-byte instruction that allows a jump to any memory location within the real memory system.

Far jumps are often called as **Inter-segment Jumps**.

Far jump instruction obtains a new segment and offset address to accomplish the jump.

The bytes 2 and 3 of this 5-byte instruction contain the new offset address while bytes 4 and 5 contain the new segment address.

If the microprocessor is operated in the protected mode, the segment address accesses a descriptor that contains the base address of the far jump segment. The offset address, which is either 16 or 32 bits, contains the offset address within the new code segment.

Jumps with Register Operands

The jump instruction can also use a 16- or 32-bit register as an operand. This automatically sets up the instruction as an **indirect jump** because the address of the jump is in the register specified by the jump instruction and the contents of the register are transferred directly into the instruction pointer.

Example: JMP AX instruction copies the contents of the AX register into the IP when the jump occurs.

Indirect Jumps Using an Index

The jump instruction may also use the **[ ] form** of addressing to directly access the jump table. The jump table can contain offset addresses for near indirect jumps, or segment and offset addresses for far indirect jumps. This type of jump is also known as a **double-indirect jump** if the register jump is called an indirect jump. The assembler assumes that the jump is near unless the FAR PTR directive indicates a far jump instruction.

Example: JMP TABLE [SI] instruction points to a jump address stored at the code segment offset location addressed by SI and jumps to the address stored in the memory at this location.

Conditional Jump

Conditional jump instructions are **always short jumps** in the 8086 through the 80286 microprocessors. This limits the range of the jump to within +127 bytes and -128 bytes from the location following the conditional jump. In the 80386 and above, conditional jumps are either short or near jumps (±32K).

The conditional jump instructions test the following flag bits: sign (S), zero (Z), carry (C), parity (P), and overflow (O). If the condition under test is **true**, a branch to the label associated with the jump instruction occurs and if the condition is **false**, the next sequential step in the program executes.

Example: JC will jump if the carry bit is set.

Conditional jump instructions

Assembly Language	**Tested Condition**	**Operation**
JA	Z = 0 and C = 0	Jump if above
JAE	C = 0	Jump if above or equal
JB	C = 1	Jump if below
JBE	Z = 1 or C = 1	Jump if below or equal
JC	C = 1	Jump if carry
JE or JZ	Z = 1	Jump if equal or jump if zero
JG	Z = 0 and S = 0	Jump if greater than
JGE	S = 0	Jump if greater than or equal
JL	S != 0	Jump if less than
JLE	Z = 1 or S != 0	Jump if less than or equal
JNC	C = 0	Jump if no carry
JNE or JNZ	Z = 0	Jump if not equal or jump if not zero
JNO	O = 0	Jump if no overflow
JNS	S = 0	Jump if no sign (positive)
JNP or JPO	P = 0	Jump if no parity or jump if parity odd
JO	O = 1	Jump if overflow
JP or JPE	P = 1	Jump if parity or jump if parity even
JS	S = 1	Jump if sign (negative)
JCXZ	CX = 0	Jump if CX is zero
JECXZ	ECX = 0	Jump if ECX equals zero
JRCXZ	RCX = 0	Jump if RCX equals zero (64-bit mode)

Note: When signed numbers are compared, use the JG, JL, JGE, JLE, JE, and JNE instructions. The terms greater than and less than refer to signed numbers. When unsigned numbers are compared, use the JA, JB, JAB, JBE, JE, and JNE instructions. The terms above and below refer to unsigned numbers.

Conditional Set Instructions

The conditional set instructions either set a byte to 01H or clear a byte to 00H, depending on the outcome of the condition under test.

Conditional set instructions

Assembly Language	**Tested Condition**	**Operation**
SETA	Z = 0 and C = 0	Set if above
SETAE	C = 0	Set if above or equal
SETB	C = 1	Set if below
SETBE	Z = 1 or C = 1	Set if below or equal
SETC	C = 1	Set if carry
SETE or SETZ	Z = 1	Set if equal or set if zero
SETG	Z = 0 and S = 0	Set if greater than
SETGE	S = 0	Set if greater than or equal
SETL	S != 0	Set if less than
SETLE	Z = 1 or S != 0	Set if less than or equal
SETNC	C = 0	Set if no carry
SETNE or SETNZ	Z = 0	Set if not equal or set if not zero
SETNO	O = 0	Set if no overflow
SETNS	S = 0	Set if no sign (positive)
SETNP or SETPO	P = 0	Set if no parity or set if parity odd
SETO	O = 1	Set if overflow
SETP or SETPE	P = 1	Set if parity or set if parity even
SETS	S = 1	Set if sign (negative)

LOOP

LOOP instruction is a combination of a **decrement CX** and **JNZ** conditional jump.

If **CX != 0**, it decrements CX and jumps to the address indicated by the label while if **CX = 0**, the next sequential instruction executes.

Conditional LOOPs

LOOPE (loop while equal) instruction jumps if **CX != 0** while an **equal condition exists**. It will exit the loop if the condition is not equal or if the CX register decrements to 0.

LOOPNE (loop while not equal) instruction jumps if **CX != 0** while a **not-equal condition exists**. It will exit the loop if the condition is equal or if the CX register decrements to 0.

WHILE Loops

The **.WHILE** statement is used with a condition to begin the loop and the **.ENDW** statement ends the loop.

REPEAT-UNTIL Loops

A series of instructions is repeated until some condition occurs. The **.REPEAT** statement defines the start of the loop and the end is defined with the **.UNTIL** statement, which contains a condition.

Procedures

A procedure is a group of instructions that usually performs one task. It is a reusable section of the software that is stored in memory once but used as often as necessary. So, this saves memory space and makes it easier to develop software.

CALL instruction **links** to the procedure and the **RET** (return) instruction **returns** from the procedure.

The **stack** stores the return address whenever a procedure is called during the execution of a program. The **CALL** instruction pushes the address of the instruction following the CALL (return address) on the stack. The **RET** instruction removes an address from the stack so the program returns to the instruction following the CALL.

With the assembler, there are specific rules for storing procedures. A procedure begins with the **PROC** directive and ends with the **ENDP** directive. Each directive appears with the name of the procedure. The PROC directive is followed by the type of procedure: NEAR or FAR.

USES statement allows any number of registers to be automatically pushed to the stack and popped from the stack within the procedure.

The **near return** instruction uses opcode **C3H** and the **far return** uses opcode **CBH**. A near return removes a **16-bit** number from the stack and places it into the instruction pointer to return from the procedure in the current code segment. A far return removes a **32-bit** number from the stack and places it into both IP and CS to return from the procedure to any memory location.

Procedures that are to be used by all software (**global**) should be written as **far** procedures while the procedures that are used by a given task (**local**) are normally defined as **near** procedures.

CALL

The CALL instruction transfers the flow of the program to the procedure. The **CALL** instruction differs from the **jump** instruction because a CALL **saves a return address on**

stack. The return address returns control to the instruction that immediately follows the CALL in a program when a **RET** instruction executes.

Near CALL

The **near CALL** is a **3 bytes** long instruction with first byte contains the **opcode**, and the second and third bytes contain the **displacement, or distance of ±32K**.

When the near CALL executes, it first pushes the offset address of the next instruction onto the stack. The offset address of the next instruction appears in the instruction pointer (IP or EIP). After saving this return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure. There is **no short CALL** instruction.

Q: Why save the IP or EIP on the stack?

Ans: The instruction pointer always points to the next instruction in the program. For the CALL instruction, the contents of IP/EIP are pushed onto the stack, so program control passes to the instruction following the CALL after a procedure ends.

Far CALL

The far CALL is a **5-byte** instruction with bytes 2 and 3 contain the new contents of the **IP**, and bytes 4 and 5 contain the new contents for **CS**.

The far CALL instruction is like a **far jump** because it can call a procedure stored in any memory location in the system.

This instruction places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 through 5 of the instruction. This allows the far CALL to call a procedure located anywhere in the memory and return from that procedure.

RET

The return instruction removes a **16-bit** number (near return) from the stack and places it into **IP**, or removes a **32-bit** number (far return) and places it into **IP** and **CS**.

Far return removes 6 bytes from the stack. The first 4 bytes contain the new value for EIP and the last 2 contain the new value for CS.

Near return removes 4 bytes from the stack and places them into EIP.

Interrupt

An interrupt is either a **hardware-generated CALL** or a **software-generated CALL**. Either type interrupts the program by calling an I**nterrupt Service Procedure** (ISP) or interrupt handler.

Interrupt Vectors

An interrupt vector is a 4-byte number stored in the first 1024 bytes of the memory (00000H–003FFH) when the microprocessor operates in the real mode.

In the protected mode, the vector table is replaced by an interrupt descriptor table that uses 8-byte descriptors to describe each of the interrupts.

There are 256 different interrupt vectors, and each vector contains the address of an interrupt service procedure. Each vector contains a value for IP and CS that forms the address of the interrupt service procedure. The first 2 bytes contain the IP and the last 2 bytes contain the CS.

Interrupt Instructions

The microprocessor has 3 different interrupt instructions that are available to the programmer: **INT, INTO**, and **INT 3**.

In the real mode, each of these instructions fetches a vector from the vector table, and then calls the procedure stored at the location addressed by the vector.

In the protected mode, each of these instructions fetches an interrupt descriptor from the interrupt descriptor table. The descriptor specifies the address of the interrupt service procedure.

The **interrupt call** is similar to a **far CALL** instruction because it places the return address (IP/EIP and CS) on the stack.

INT Interrupt instruction

Each INT instruction has a numeric operand whose range is 0 to 255 (00H–FFH).

Example: INT 100 uses interrupt vector 100, which appears at memory address 190H–193H.

The address of the interrupt vector is determined by multiplying the interrupt type number by 4.

Example: INT 10H instruction calls the interrupt service procedure whose address is stored beginning at memory location 40H (10H * 4) in the real mode.

In the protected mode, the interrupt descriptor is located by multiplying the type number by 8 instead of 4 because each descriptor is 8 bytes long.

Each INT instruction is **2 bytes** long. The first byte contains the **opcode** and the second byte contains the **vector type number**.

Whenever a software interrupt instruction executes, it,

1) pushes the flags onto the stack
2) clears the T and I flag bits
3) pushes CS onto the stack
4) fetches the new value for CS from the interrupt vector
5) pushes IP/EIP onto the stack

6) fetches the new value for IP/EIP from the vector
7) jumps to the new location addressed by CS and IP/EIP.

The INT instruction performs as a **far CALL** except that it not only **pushes CS and IP** onto the stack but it also **pushes the flags** onto the stack.

The INT instruction performs the operation of a **PUSHF**, followed by a **far CALL** instruction.

The **INT** instruction is **2 bytes** long, whereas the far **CALL** is **5 bytes** long.

When the INT instruction executes, it clears the interrupt flag (I), which controls the external hardware interrupt input pin **INTR** (interrupt request). When **I = 0**, the microprocessor **disables** the INTR pin and when **I = 1**, the microprocessor **enables** the INTR pin.

Software interrupts are most commonly used to call system procedures because the address of the system function need not be known. The system procedures are common to all system and application software. The interrupts often control printers, video displays, and disk drives.

IRET/IRETD

Interrupt Return (IRET) instruction is used only with software or hardware interrupt service procedures.

IRET instruction will,

1) pop stack data back into the IP.
2) pop stack data back into CS.
3) pop stack data back into the flag register.

The **IRET** instruction accomplishes the same tasks as the **POPF** followed by a **far RET** instruction.

Whenever an **IRET** instruction executes, it restores the contents of **I** and **T** from the stack because it preserves the state of these flag bits.

IRETD instruction is used to return from an interrupt service procedure that is called in the protected mode.

IRETD differs from **IRET** because it **pops a 32-bit** instruction pointer (EIP) from the stack. **IRET** is used in the **real mode** while **IRETD** is used in the **protected mode**.

INT 3

An INT 3 instruction is a special software interrupt designed to function as a breakpoint.

INT 3 is a **1-byte** instruction while the others are 2-byte instructions.

It is common to insert an INT 3 instruction in software to interrupt or break the flow of the software.

INTO

Interrupt on overflow (INTO) is a conditional software interrupt that tests the overflow flag (O). The INTO instruction appears in software that adds or subtracts signed binary numbers.

If O = 0, the INTO instruction performs no operation and if O = 1, an interrupt occurs via vector type number 4.

Interrupt Control

The **Set Interrupt** (STI) instruction **enables INTR** and the **Clear Interrupt** (CLI) instruction **disables INTR**.

Miscellaneous Instructions

HLT: The halt instruction (HLT) stops the execution of software. There are 3 ways to exit a halt: by an interrupt, by a hardware reset, or during a DMA operation.

NOP: When the microprocessor encounters a no operation instruction (NOP), it takes a short time to execute.

WAIT: This instruction tests the condition of the BUSY or TEST pin on the microprocessor. If BUSY or TEST = 1, WAIT does not wait; but if BUSY or TEST = 0, WAIT continues testing the BUSY or TEST pin until it becomes a logic 1.

LOCK: The LOCK prefix causes the LOCK pin to become logic 0 for the duration of the locked instruction. The ESC instruction passes instruction to the numeric coprocessor.

BOUND: This instruction compares the contents of any 16-bit register against the contents of 2 words of memory: an upper and a lower boundary. If the value in the register compared with memory is not within the upper and lower boundary, a type 5 interrupt ensues.

ENTER and LEAVE: The ENTER and LEAVE instructions are used with stack frames. A stack frame is a mechanism used to pass parameters to a procedure through the stack memory. The stack frame also holds local memory variables for the procedure. The ENTER instruction creates the stack frame and the LEAVE instruction removes the stack frame from the stack. The BP register addresses stack frame data.