MICROPROCESSOR-PART2

- **8086 -Architecture-registers, RAM organization segment-offset addressing, real & protected modes, addressing modes, instructions – arithmetic, data movement, control, I/O string, logical. Subroutine call & return.**
- **Features of Pentium processor**.

8086

- 8086 Microprocessor is an enhanced version of 8085Microprocessor that was designed by Intel in 1976.
- It is a 16-bit Microprocessor having **20 address lines** and16 **data lines** that provides up to 1MB storage.
- It consists of powerful instruction set, which provides operations like multiplication and division easily.
- It supports two modes of operation, i.e. **Maximum mode and Minimum mode**.
- Maximum mode is suitable for system having multiple processors and Minimum mode is suitable for system having a single processor.

Features of 8086

- The most prominent features of a 8086 microprocessor are as follows −

➢ It has an instruction queue, which is capable of storing six instruction bytes from the memory resulting in faster processing.

➢ It was the first 16-bit processor having 16-bit ALU, 16-bit registers, internal data bus, and 16-bit external data bus resulting in faster processing.

➢ It is available in 3 versions based on the frequency of operation −

- ✓ 8086 → 5MHz
- ✓ 8086-2 → 8MHz
- ✓ (c)8086-1 → 10 MHz

➢ It uses two stages of pipelining, i.e. Fetch Stage and Execute Stage, which improves performance.

➢ Fetch stage can prefetch up to 6 bytes of instructions and stores them in the queue.

➢ Execute stage executes these instructions.

➢ It has 256 vectored interrupts.

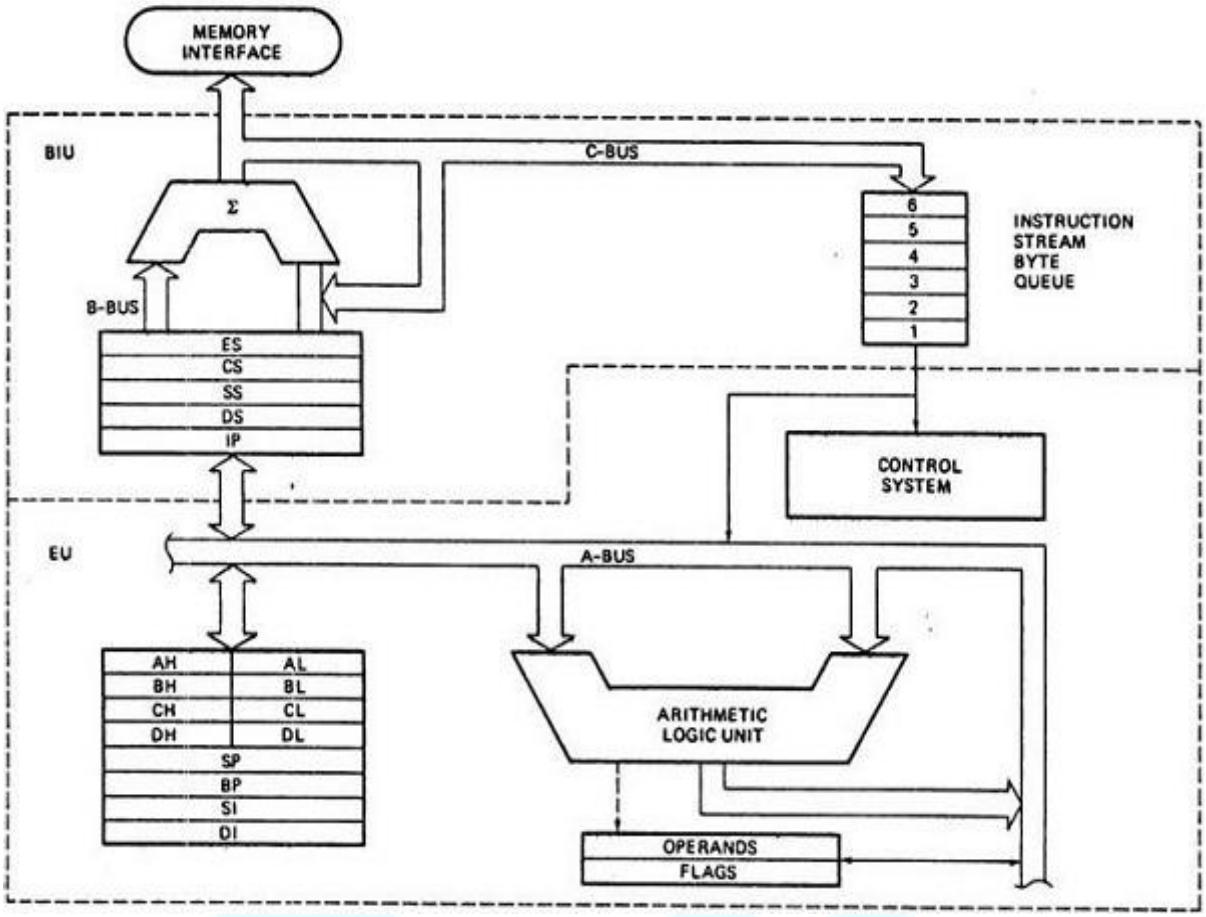
➢ It consists of 29,000 transistors.

Comparison between 8085 & 8086 Microprocessor

- Size − 8085 is 8-bit microprocessor, whereas 8086 is 16-bit microprocessor.
- Address Bus − 8085 has 16-bit address bus while 8086 has 20-bit address bus.
- Memory − 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.
- Instruction − 8085 doesn't have an instruction queue, whereas 8086 has an instruction queue.
- Pipelining − 8085 doesn't support a pipelined architecture while 8086 supports a pipelined architecture.
- I/O − 8085 can address 2^8 = 256 I/O's, whereas 8086 can access 2^16 = 65,536 I/O's.
- Cost − The cost of 8085 is low whereas that of 8086 is high.

ARCHITECTURE OF 8086

- The following diagram depicts the architecture of a 8086 Microprocessor −



- 8086 Microprocessor is divided into two functional units.
  - **EU (Execution Unit)**
  - **BIU (Bus Interface Unit)**

EU (Execution Unit)

- Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions.
- Its function is to control operations on data using the instruction decoder & ALU.
- EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU.

Functional parts of 8086 microprocessors.

ALU

- It handles all arithmetic and logical operations, like +, −, ×, /, OR, AND, NOT operations.

Flag Register

- It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to the result stored in the accumulator.
- It has 9 flags and they are divided into 2 groups.
  - ✓ **Conditional Flags**
  - ✓ **Control Flags.**

Conditional Flags

- It represents the result of the last arithmetic or logical instruction executed.
- Following is the list of conditional flags −

1. **Carry flag** − This flag indicates an overflow condition for arithmetic operations.

2. **Auxiliary flag** − When an operation is performed at ALU, it results in a carry/barrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.

3. **Parity flag** − This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.

4. **Zero flag** − This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.

5. **Sign flag** − This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.

6. **Overflow flag** − This flag represents the result when the system capacity is exceeded.

Control Flags

- Control flags controls the operations of the execution unit.
- Following is the list of control flags −

1. **Trap flag** − It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.

2. **Interrupt flag** − It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

3. **Direction flag** − It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General purpose register

- There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL.
- These registers can be used individually to store 8-bit data and can be used in pairs to store 16bit data.
- The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL.
- It is referred to the AX, BX, CX, and DX respectively.

AX register − It is also known as accumulator register. It is used to store operands for arithmetic operations.

BX register − It is used as a base register. It is used to store the starting base address of the memory area within the data segment.

CX register − It is referred to as counter. It is used in loop instruction to store the loop counter.

DX register − This register is used to hold I/O port address for I/O instruction.

Stack pointer register

- It is a 16-bit register, which holds the address from the start of the segment to the memory location, where a word was most recently stored on the stack.

Base Pointer register

- BP can hold the offset addresses of any location in the stack segment. It is used to access random locations of the stack.

Source Index register

- It holds offset address in Data Segment during string operations.

Destination Index register

- It holds offset address in Extra Segment during string operations.

BIU (Bus Interface Unit)

- BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory.
- EU has no direction connection with System Buses so this is possible with the BIU.
- EU and BIU are connected with the Internal Bus.

It has the following functional parts –

Instruction queue

- BIU contains the instruction queue.
- BIU gets up to 6 bytes of next instructions and stores them in the instruction queue.
- When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.
- Fetching the next instruction while the current instruction executes is called **pipelining.**

Segment register

- BIU has 4 segment buses, i.e. CS, DS, SS & ES.
- It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations.
- It also contains 1 pointer register IP, which holds the address of the next instruction to execute by the EU.

CS − It stands for Code Segment. It is used for addressing a memory location in the code segment of the memory, where the executable program is stored.

DS − It stands for Data Segment. It consists of data used by the program and is accessed in the data segment by an offset address or the content of other register that holds the offset address.

SS − It stands for Stack Segment. It handles memory to store data and addresses during execution.

ES − It stands for Extra Segment. ES is additional data segment, which is used by the string to hold the extra destination data.

Instruction pointer

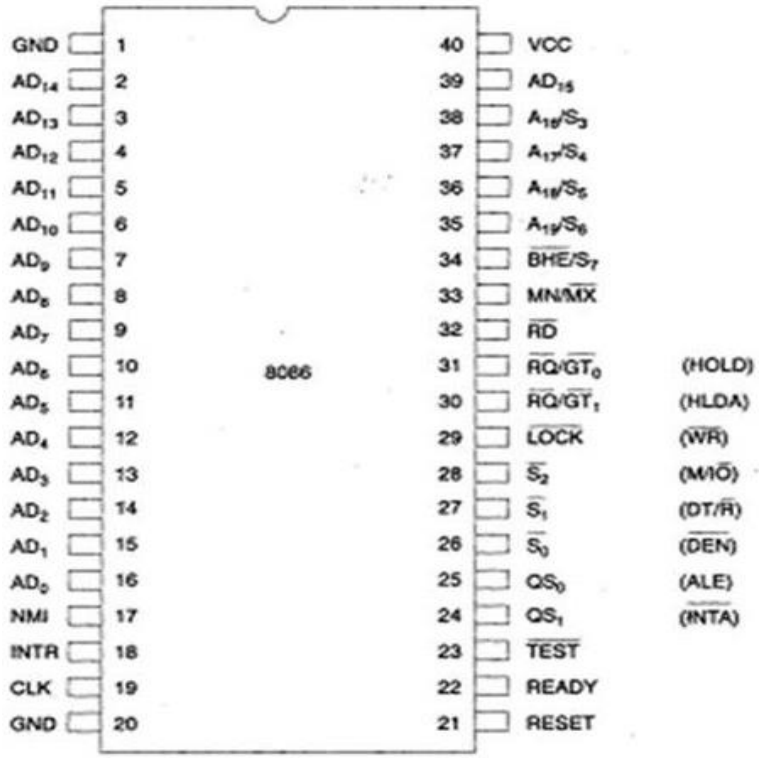
- It is a 16-bit register used to hold the address of the next instruction to be executed.

8086 PIN CONFIGURATION

- 8086 was the first 16-bit microprocessor available in 40-pin DIP (Dual Inline Package) chip.

8086 Pin Diagram

- Here is the pin diagram of 8086 microprocessor −



- **Power supply and frequency signals**

✓ It uses 5V DC supply at VCC pin 40, and uses ground at VSS pin 1 and 20 for its operation.

- **Clock signal**

✓ Clock signal is provided through Pin-19.

✓ It provides timing to the processor for operations.

✓ Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

- **Address/data bus**

✓ AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8AD15 carries higher order byte data.

✓ During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

- **Address/status bus**

✓ A16-A19/S3-S6. These are the 4 address/status buses.
✓ During the first clock cycle, it carries 4-bit address and later it carries status signals.

- **S7/BHE**

✓ BHE stands for Bus High Enable.
✓ It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15.
✓ This signal is low during the first clock cycle, thereafter it is active.

- **Read($\overline{RD}$)**

✓ It is available at pin 32 and is used to read signal for Read operation.

- **Ready**

✓ It is available at pin 22.
✓ It is an acknowledgement signal from I/O devices that data is transferred.
✓ It is an active high signal.
✓ When it is high, it indicates that the device is ready to transfer data.
✓ When it is low, it indicates wait state.

- **RESET**

✓ It is available at pin 21 and is used to restart the execution.
✓ It causes the processor to immediately terminate its present activity.
✓ This signal is active high for the first 4 clock cycles to RESET the microprocessor.

- **INTR**

✓ It is available at pin 18.
✓ It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

- **NMI**

✓ It stands for non-maskable interrupt and is available at pin 17.
✓ It is an edge triggered input, which causes an interrupt request to the microprocessor.

✓ **$\overline{TEST}$**
✓ This signal is like wait state and is available at pin 23.
✓ When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

- **MN/$\overline{MX}$**

✓ It stands for Minimum/Maximum and is available at pin 33.
✓ It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

- **INTA**
  - ✓ It is an interrupt acknowledgement signal and id available at pin 24.
  - ✓ When the microprocessor receives this signal, it acknowledges the interrupt.

- **ALE**

✓ It stands for address enable latch and is available at pin 25.
✓ A positive pulse is generated each time the processor begins any operation.
✓ This signal indicates the availability of a valid address on the address/data lines.

- **DEN**

✓ It stands for Data Enable and is available at pin 26.
✓ It is used to enable Transreceiver 8286.
✓ The transreceiver is a device used to separate data from the address/data bus.

- **DT/R**

✓ It stands for Data Transmit/Receive signal and is available at pin 27.
✓ It decides the direction of data flow through the transreceiver.
✓ When it is high, data is transmitted out and vice-a-versa.

- **M/IO**

✓ This signal is used to distinguish between memory and I/O operations.
✓ When it is high, it indicates I/O operation and when it is low indicates the memory operation.
✓ It is available at pin 28.

- WR

✓ It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

- HLDA

✓ It stands for Hold Acknowledgement signal and is available at pin 30.
✓ This signal acknowledges the HOLD signal.

- HOLD

✓ This signal indicates to the processor that external devices are requesting to access the address/data buses.
✓ It is available at pin 31.

- **QS1 and QS0**

✓ These are queue status signals and are available at pin 24 and 25.
✓ These signals provide the status of instruction queue.
✓ Their conditions are shown in the following table −

QS0	QS1	Status
0	0	No operation
0	1	First byte of opcode from the queue

1	0	Empty the queue
1	1	Subsequent byte from the queue

- **S0, S1, S2**

✓ These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals.

✓ These are available at pin 26, 27, and 28. Following is the table showing their status −

S2	S1	S0	Status
0	0	0	Interrupt acknowledgement
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write

- **LOCK**

✓ When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus.

✓ It is activated using the LOCK prefix on any instruction and is available at pin 29.

- **RQ/GT_1 and RQ/GT_0**

✓ These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus.

✓ When the signal is received by CPU, then it sends acknowledgment. RQ/GT_0 has a higher priority than RQ/GT_1.

RAM ORGANIZATION SEGMENT

- Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address.
- It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.

Need for Segmentation –

- The Bus Interface Unit (BIU) contains four 16 bit special purpose registers (mentioned below) called as **Segment Registers**.

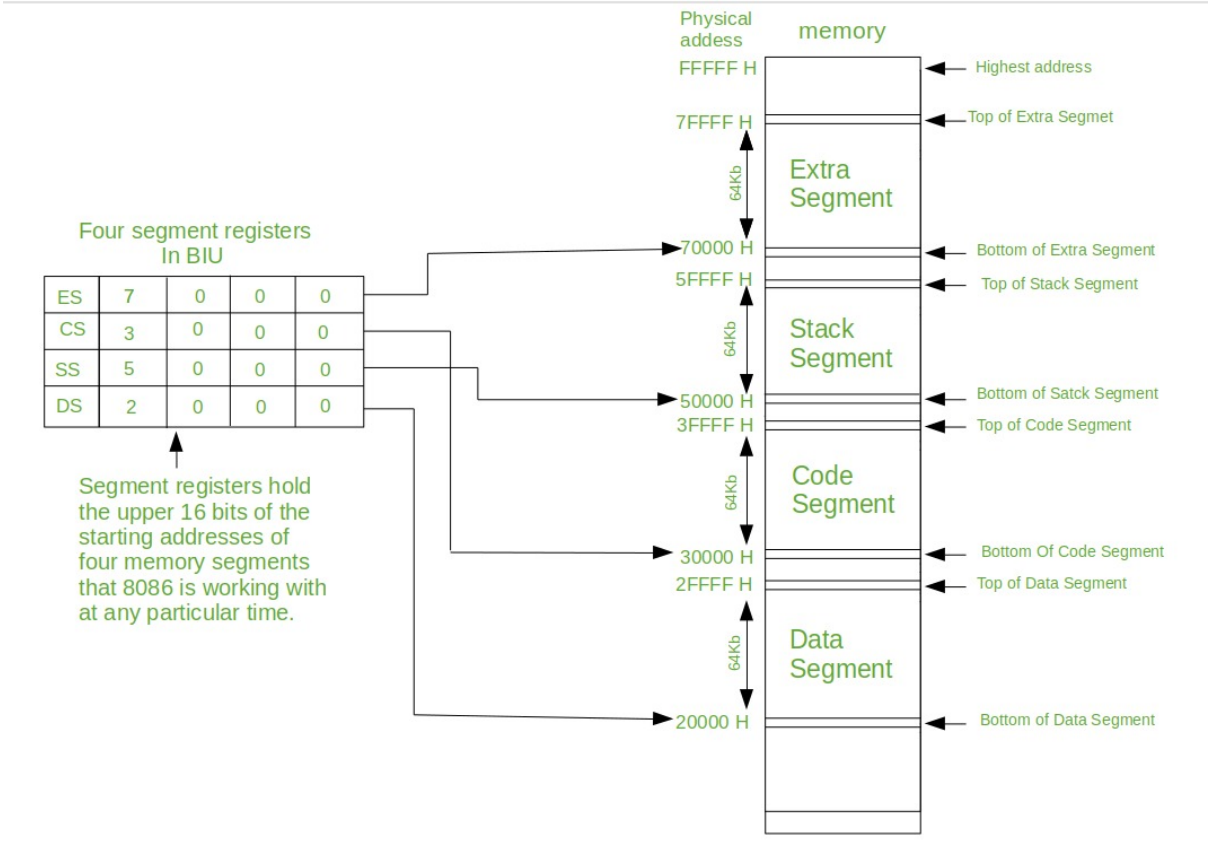
  ✓ **Code segment register (CS)**: is used for addressing memory location in the code segment of the memory, where the executable program is stored.

  ✓ **Data segment register (DS)**: points to the data segment of the memory where the data is stored.

- ✓ **Extra Segment Register (ES)**: also refers to a segment in the memory which is another data segment in the memory.
- ✓ **Stack Segment Register (SS)**: is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

- The number of address lines in 8086 is 20, 8086 BIU will send 20bit address, so as to access one of the 1MB memory locations.
- The four segment registers actually contain the upper 16 bits of the starting addresses of the four memory segments of 64 KB each with which the 8086 is working at that instant of time.
- A segment is a logical unit of memory that may be up to 64 kilobytes long.
- Each segment is made up of contiguous memory locations.
- It is an independent, separately addressable unit.
- Starting address will always be changing.
- It will not be fixed.

- Note that the 8086 does not work the whole 1MB memory at any given time.
- However, it works only with four 64KB segments within the whole 1MB memory.
- Below is the one way of positioning four 64 kilobyte segments within the 1M byte memory space of an 8086.



Types of Segmentation

- ✓ **Overlapping Segment** – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts along with this 64kilobytes location of the first segment, then the two are said to be Overlapping Segment.

- ✓ **Non-Overlapped Segment** – A segment starts at a particular address and its maximum size can go up to 64kilobytes. But if another segment starts before this 64kilobytes location of the first segment, then the two segments are said to be Non-Overlapped Segment.

Rules of Segmentation Segmentation process follows some rules as follows:

- The starting address of a segment should be such that it can be evenly divided by 16.
- Minimum size of a segment can be 16 bytes and the maximum can be 64 kB

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Advantages of the Segmentation

The main advantages of segmentation are as follows:

- ✓ It provides a powerful memory management mechanism.
- ✓ Data related or stack related operations can be performed in different segments.
- ✓ Code related operation can be done in separate code segments.
- ✓ It allows to processes to easily share data.
- ✓ It allows extending the address ability of the processor, i.e. segmentation allows the use of 16 bit registers to give an addressing capability of 1 Megabytes. Without segmentation, it would require 20 bit registers.
- ✓ It is possible to enhance the memory size of code data or stack segments beyond 64 KB by allotting more than one segment for each area.

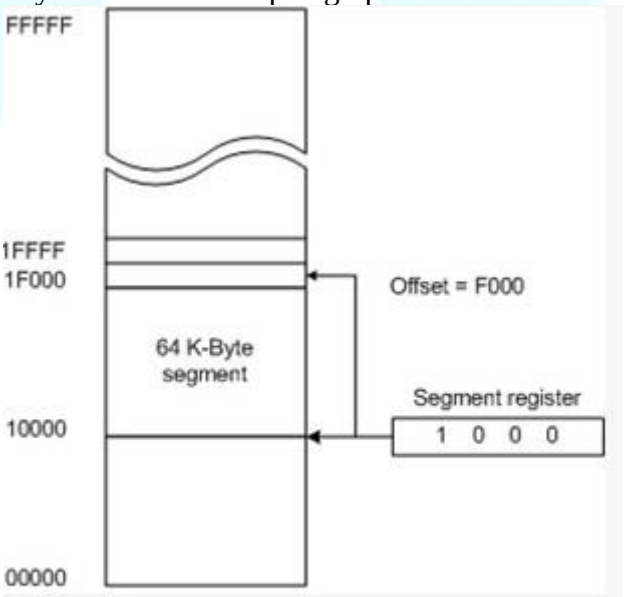
REAL MODE MEMORY ADDRESSING

- The 80286 and above operate in either the real or protected mode.
- Only the 8086 and 8088 op-erate exclusively in the real mode.
- Real mode operation allows the microprocessor to address only the first 1M byte of memory space-even if it is the Pentium II microprocessor.
- Note that the first 1 M byte of memory is called either the real memory or conventional memory system.
- The DOS operating system requires the microprocessor to operate in the real mode.
- Real mode oper-ation allows application software written for the 8086/8088, which contains only 1 M byte of memory, to function in the 80286 and above without changing the software.
- The upward com-patibility of software is partially responsible for the continuing success of the Intel family of mi-croprocessors.
- In all cases, each of these microprocessors begins operation in the real mode by default whenever power is applied or the microprocessor is reset.

SEGMENTS AND OFFSETS

- A combination of a segment address and an offset address, access a memory location in the real mode.

- All real mode memory addresses must consist of a segment address plus an offset address.
- The segment address, located within one of the segment registers, defines the beginning address of any 64K-byte memory segment.
- The offset address selects any location within the 64K byte memory segment.
- Segments in the real mode always have a length of 64K bytes.
- Figure 2-3 shows how the segment plus offset addressing scheme selects a memory location.
- This illustra-tion shows a memory segment that begins at location 1 0000H and ends at location 1 FFFEH 64K bytes in length.
- It also shows how an offset address, sometimes called a displacement, and of F000H selects location 1F000H in the memory system.
- Note that the offset or displacement is the distance above the start of the segment, as shown in figure.
- The segment register in figure contains a 1000H, yet it addresses a starting segment at location 10000H.
- In the real mode, each segment register is internally appended with a 0H on its rightmost end.
- This forms a 20-bit memory address, allowing it to access the start of a segment.
- The microprocessor must generate a 20-hit memory address to access a location within the first 1 M of memory.
- For example, when a segment register contains a 1200H, it addresses a 64K-byte memory segment beginning at location 12000H.
- Likewise, if a segment register contains a 1201H, it addresses a memory segment beginning at location 12010H.
- Because of the internally ap-pended 0H, real mode segments can begin only at a 16-byte boundary in the memory system.
- This 16-byte boundary is often called a paragraph.



The real mode memory-addressing scheme

- Because a real mode segment of memory is 64K in length, once the beginning address is known, the ending address is found by adding FFFFH.

- The offset address, which is a part of the address, is added to the start of the segment to ad-dress a memory location within the memory segment.
- For example, if the segment address is 1000H and the offset address is 2000H, the microprocessor addresses memory location 12000H.
- The offset address is always added to the starting address of the segment to locate the data.
- The segment and offset address is sometimes written as 1000:2000 for a segment address of 1000H with an offset of 2000H.
- In the 80286 (with special external circuitry), and the 80386 through the Pentium II, an extra 64K minus 16 bytes of memory is addressable when the segment address is FFFFH and the HIMEM.SYS driver is installed in the system.
- This area of memory (0FFFF0H-10FFEFH) is re-ferred to as high memory.
- Some addressing modes combine more than one register and an offset value to form an offset address.
- When this occurs, the sum of these values may exceed FFFFH.

- For example, the address accessed in a segment whose segment address is 4000H, and whose offset address is specified as the sum of F000H plus 3000H, will access memory location 42000H instead of lo-cation 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 (F000H + 3000H=12000H) is dropped for this addition to form the offset address of 2000H.
- This means that the address is generated as 4000:2000 or 42000H.

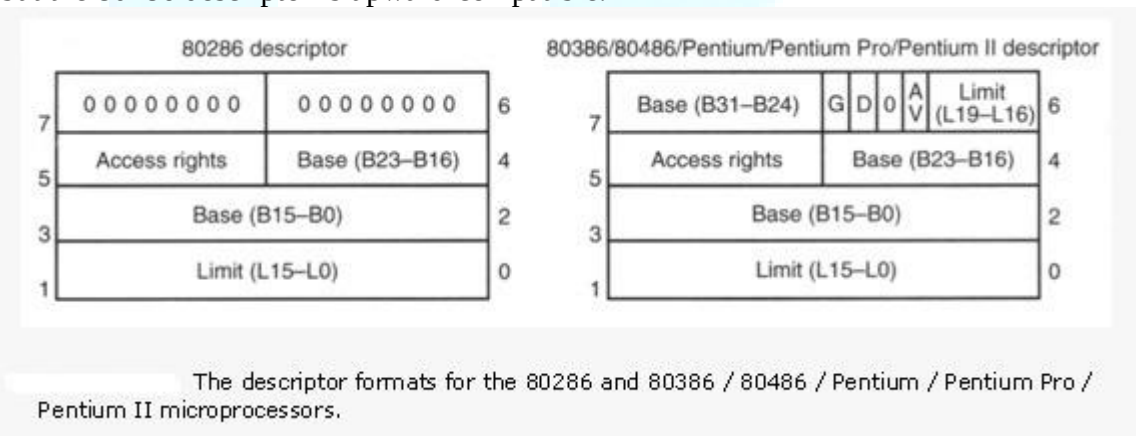
PROTECTED MODE MEMORY ADDRESSING

- Protected mode memory addressing (80286 and above) allows access to data and programs lo-cated above the first 1M byte of memory, as well as within the first 1M byte of memory.
- Ad-dressing this extended section of the memory system requires a change to the segment plus an offset addressing scheme used with real mode memory addressing.
- When data and programs are addressed in extended memory, the offset address is still used to access information located within the memory segment.
- One difference is that the segment address is no longer present in the protected mode. In place of the segment ad-dress, the segment register contains a selector that selects a descriptor from a descriptor table.
- The descriptor describes the memory segment's location, length, and access rights.
- Because the segment register and offset address still access memory, protected mode instructions are iden-tical to real mode instructions.
- In fact, most programs written to function in the real mode will function without change in the protected mode.
- The difference between modes is in the way that the segment register is interpreted by the microprocessor to access the memory segment.
- An-other difference, in the 80386 and above, is that the offset address can be a 32-bit number instead of a 16-bit number in the protected mode.
- A 32-bit offset address allows the microprocessor to access data within a segment that can be up to 4G bytes in length.

Selectors and Descriptors

- The selector, located in the segment register, selects one of 8192 descriptors from one of two ta-bles of descriptors.
- The descriptor describes the location, length, and access rights of the seg-ment of memory.
- Indirectly, the segment register still selects a memory segment, but not directly as in the real mode.
- For example, in the real mode, if CS = 0008H, the code segment begins at lo-cation 00080H.
- In the protected mode, this segment number can address any memory location in the entire system for the code segment.

- There are two descriptor tables used with the segment registers: **one contains global de-scriptors and the other contains local descriptors.**
- The global descriptors contain segment defini-tions that apply to all programs, while the local descriptors are usually unique to an application.
- You might call a global descriptor a system descriptor and call a local descriptor an application descriptor.
- Each descriptor table contains 8192 descriptors, so a total of 16,384 total descriptors are available to an application at any time.
- Because the descriptor describes a memory segment, this allows up to 16,384 memory segments to be described for each application.
- Figure shows the format of a descriptor for the 80286 through the Pentium II.
- Note that each descriptor is 8 bytes in length, so the global and local descriptor tables are each a max-imum of 64K bytes in length.
- Descriptors for the 80286 and the 80386 through the Pentium II differ slightly, but the 80286 descriptor is upward-compatible.



The descriptor formats for the 80286 and 80386 / 80486 / Pentium / Pentium Pro / Pentium II microprocessors.

- The base address portion of the descriptor indicates the starting location of the memory segment.
- The segment limit contains the last offset address found in a segment.
- There is another feature found in the 80386 through the Pentium II descriptor that is not found in the 80286 descriptor: the G bit, or granularity bit.
- If G=0, the limit specifies a seg-ment limit of 00000H to FFFFFH. If G = 1, the value of the limit is multiplied by 4K bytes (ap-pended with XXXH).
- The limit is then 00000XXXH to FFFFFXXXH, if G=1.
- This allows a segment length of 4K to 4G bytes in steps of 4K bytes.

- The AV bit, in the 80386 and above descriptor, is used by some operating systems to indi-cate that the segment is available (AV = 1) or not available (AV = 0).
- The D bit indicates how the 80386 through the Pentium II instructions access register and memory data in the protected or real mode.
- If D=0, the instructions are 16-bit instructions, compatible with the 8086—80286 mi-croprocessors.
- This means that the instructions use 16-bit offset addresses and 16-bit registers by default.
- This mode is often called the 16-bit instruction mode. If D=1, the instructions are 32-bit instructions.
- The access rights byte in figure controls access to the protected mode memory segment.

8086 ADDRESSING MODE

- The different ways in which a source operand is denoted in an instruction is known as addressing modes.
- There are 8 different addressing modes in 8086 programming −

1. **Immediate addressing mode**

✓ The addressing mode in which the data operand is a part of the instruction itself is known as immediate addressing mode.

Example
MOV CX, 4929 H, ADD AX, 2387 H, MOV AL, FFH

2. **Register addressing mode**

✓ It means that the register is the source of an operand for an instruction.

Example
MOV CX, AX; (copies the contents of the 16-bit AX register into the 16-bit CX register)
ADD BX, AX

3. **Direct addressing mode**

✓ The addressing mode in which the effective address of the memory location is written directly in the instruction.

Example
MOV AX, [1592H], MOV AL, [0300H]

4. **Register indirect addressing mode**

✓ This addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI & SI.

Example
MOV AX, [BX]; (suppose the register BX contains 4895H, then the contents 4895H are moved to AX)
ADD CX, {BX}

5. **Based addressing mode**

✓ In this addressing mode, the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement.

Example
MOV DX, [BX+04], ADD CL, [BX+08]

6. **Indexed addressing mode**

✓ In this addressing mode, the operands offset address is found by adding the contents of SI or DI register and 8-bit/16-bit displacements.

Example
MOV BX, [SI+16], ADD AL, [DI+16]

7. **Based-index addressing mode**

✓ In this addressing mode, the offset address of the operand is computed by summing the base register to the contents of an Index register.

Example
ADD CX, [AX+SI], MOV AX, [AX+DI]

8. **Based indexed with displacement mode**

✓ In this addressing mode, the operands offset is computed by adding the base register contents.

✓ An Index registers contents and 8 or 16-bit displacement.

Example
MOV AX, [BX+DI+08], ADD CX, [BX+SI+16]

8086 INSTRUCTION SETS

- The 8086 microprocessor supports 8 types of instructions −

- ✓ **Data Transfer Instructions**
- ✓ **Arithmetic Instructions**
- ✓ **Bit Manipulation Instructions**
- ✓ **String Instructions**
- ✓ **Program Execution Transfer Instructions (Branch & Loop Instructions)**
- ✓ **Processor Control Instructions**
- ✓ **Iteration Control Instructions**
- ✓ **Interrupt Instructions**

Data Transfer Instructions

- These instructions are used to transfer the data from the source operand to the destination operand.
- Following are the list of instructions under this group −

Instruction to transfer a word

- ✓ **MOV** − Used to copy the byte or word from the provided source to the provided destination.
- ✓ **PPUSH** − Used to put a word at the top of the stack.
- ✓ **POP** − Used to get a word from the top of the stack to the provided location.
- ✓ **PUSHA** − Used to put all the registers into the stack.
- ✓ **POPA** − Used to get words from the stack to all registers.
- ✓ **XCHG** − Used to exchange the data from two locations.
- ✓ **XLAT** − Used to translate a byte in AL using a table in the memory.

Instructions for input and output port transfer

- ✓ **IN** − Used to read a byte or word from the provided port to the accumulator.
- ✓ **OUT** − Used to send out a byte or word from the accumulator to the provided port.

Instructions to transfer the address

- ✓ **LEA** − Used to load the address of operand into the provided register.
- ✓ **LDS** − Used to load DS register and other provided register from the memory
- ✓ **LES** − Used to load ES register and other provided register from the memory.

Instructions to transfer flag registers

- ✓ **LAHF** − Used to load AH with the low byte of the flag register.

- ✓ **SAHF** − Used to store AH register to low byte of the flag register.
- ✓ **PUSHF** − Used to copy the flag register at the top of the stack.
- ✓ **POPF** − Used to copy a word at the top of the stack to the flag register.

Arithmetic Instructions

- These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.
- Following is the list of instructions under this group −

Instructions to perform addition

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

Instructions to perform subtraction

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

Instruction to perform multiplication

- ✓ **MUL** − Used to multiply unsigned byte by byte/word by word.
- ✓ **IMUL** − Used to multiply signed byte by byte/word by word.
- ✓ **AAM** − Used to adjust ASCII codes after multiplication.

Instructions to perform division

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

- ✓ **IDIV** − Used to divide the signed word by byte or signed double word by word.
- ✓ **AAD** − Used to adjust ASCII codes after division.
- ✓ **CBW** − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.
- ✓ **CWD** − Used to fill the upper word of the double word with the sign bit of the lower word.

Bit Manipulation Instructions

- These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.
- Following is the list of instructions under this group −

Instructions to perform logical operation

- ✓ **NOT** − Used to invert each bit of a byte or word.
- ✓ **AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- ✓ **OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- ✓ **XOR** − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- ✓ **TEST** − Used to add operands to update flags, without affecting operands.

Instructions to perform shift operations

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

Instructions to perform rotate operations

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

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

String Instructions

- String is a group of bytes/words and their memory is always allocated in a sequential order.
- Following is the list of instructions under this group −

✓ **REP** − Used to repeat the given instruction till CX ≠ 0.

✓ **REPE/REPZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

✓ **REPNE/REPNZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

✓ **MOVS/MOVSB/MOVSW** − Used to move the byte/word from one string to another.

✓ **COMS/COMPSB/COMPSW** − Used to compare two string bytes/words.

✓ **INS/INSB/INSW** − Used as an input string/byte/word from the I/O port to the provided memory location.

✓ **OUTS/OUTSB/OUTSW** − Used as an output string/byte/word from the provided memory location to the I/O port.

✓ **SCAS/SCASB/SCASW** − Used to scan a string and compare its byte with a byte in AL or string word with a word in AX.

✓ **LODS/LODSB/LODSW** − Used to store the string byte into AL or string word into AX.

Program Execution Transfer Instructions (Branch and Loop Instructions)

- These instructions are used to transfer/branch the instructions during an execution.
- It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition −

✓ **CALL** − Used to call a procedure and save their return address to the stack.

✓ **RET** − Used to return from the procedure to the main program.

✓ **JMP** − Used to jump to the provided address to proceed to the next instruction.

Instructions to transfer the instruction during an execution with some conditions −

✓ **JA/JNBE** − Used to jump if above/not below/equal instruction satisfies.

✓ **JAE/JNB** − Used to jump if above/not below instruction satisfies.

✓ **JBE/JNA** − Used to jump if below/equal/ not above instruction satisfies.

- ✓ **JC** − Used to jump if carry flag CF = 1
- ✓ **JE/JZ** − Used to jump if equal/zero flag ZF = 1
- ✓ **JG/JNLE** − Used to jump if greater/not less than/equal instruction satisfies.
- ✓ **JGE/JNL** − Used to jump if greater than/equal/not less than instruction satisfies.
- ✓ **JL/JNGE** − Used to jump if less than/not greater than/equal instruction satisfies.
- ✓ **JLE/JNG** − Used to jump if less than/equal/if not greater than instruction satisfies.
- ✓ **JNC** − Used to jump if no carry flag (CF = 0)
- ✓ **JNE/JNZ** − Used to jump if not equal/zero flag ZF = 0
- ✓ **JNO** − Used to jump if no overflow flag OF = 0
- ✓ **JNP/JPO** − Used to jump if not parity/parity odd PF = 0
- ✓ **JNS** − Used to jump if not sign SF = 0
- ✓ **JO** − Used to jump if overflow flag OF = 1
- ✓ **JP/JPE** − Used to jump if parity/parity even PF = 1
- ✓ **JS** − Used to jump if sign flag SF = 1

Processor Control Instructions

- These instructions are used to control the processor action by setting/resetting the flag values.
- Following are the instructions under this group −

- ✓ **STC** − Used to set carry flag CF to 1
- ✓ **CLC** − Used to clear/reset carry flag CF to 0
- ✓ **CMC** − Used to put complement at the state of carry flag CF.
- ✓ **STD** − Used to set the direction flag DF to 1
- ✓ **CLD** − Used to clear/reset the direction flag DF to 0
- ✓ **STI** − Used to set the interrupt enable flag to 1, i.e., enable INTR input.
- ✓ **CLI** − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

Iteration Control Instructions

- These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

✓ **LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0

✓ **LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0

✓ **LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

✓ **JCXZ** − Used to jump to the provided address if CX = 0

Interrupt Instructions

- These instructions are used to call the interrupt during program execution.

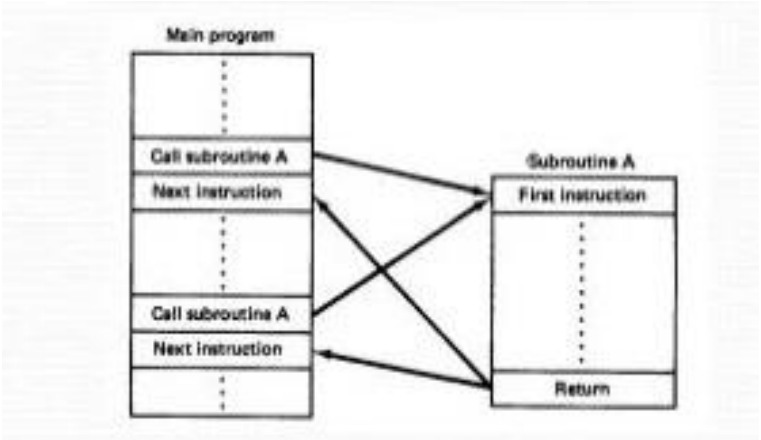
✓ **INT** − Used to interrupt the program during execution and calling service specified.

✓ **INTO** − Used to interrupt the program during execution if OF = 1

✓ **IRET** − Used to return from interrupt service to the main program

SUBROUTINE CALL AND RETURN

- A subroutine is a special segment of program that can be called for execution from any point in a program.
- Figure (a) illustrates the concept of a subroutine.
- Here we see a program structure where one part of the program is called the main program.
- In addition to this, we find a group of instructions attached to the main program, known as a subroutine.
- The subroutine is written to provide a function that must be performed at various points in the main program.
- Instead of including this piece of code in the main program each time the function is needed, it is put into the program just once as a subroutine.
- An assembly language subroutine is also referred to as a procedure.



- Wherever the function must be performed, a single instruction is inserted into the main body of the program to "call" the subroutine.
- The logical address CS: IP identifies the next instruction to be fetched for execution. Thus, to branch to a subroutine that starts elsewhere in memory, the value in either " IP "or "CS and IP" must be modified.
- After executing the subroutine, we want to return control to the instruction that immediately follows the one called the subroutine.
- To facilitate this return operation, return linkage is saved when the call takes place. That is, the original value of "IP" or "IP and CS" must be preserved.
- A return instruction is included at the end of the subroutine to initiate the return sequence to the main program environment.
- In this way, program execution resumes in the main program at the point where it left off due to the occurrence of the subroutine call.
- The instructions provided to transfer control from the main program to a subroutine and return control back to the main program are called subroutine-handling instructions.

CALL and RET Instructions

- There are two basic instructions in the instruction set of the 8086 for subroutine handling:
  - **call (CALL) instruction**
  - **return (RET) instruction**
- CALL instruction is used to call the subroutine.
- RET instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.

Call instruction:

- Just like the JMP instruction, CALL allows implementation of two types of operations: **the intra segment call and the intersegment call.**
- It is the operand that initiates either an intersegment or an intra segment call.
- The operands Near-proc, Mem.l6, and Reg.l6 all specify intra segment calls to a subroutine.
- In all three cases, execution of the instruction causes the contents of IP to be saved on the stack.
- The saved value of IP is the offset address of the instruction that immediately follows the CALL instruction.
- After saving this return address, a new 16-bit value, which is specified by the instruction's operand and corresponds to the storage location of the first instruction in the subroutine, is loaded into IP types of operands represent different ways of specifying a new value of IP.

- Using a Near-proc, Mem.l6, and Reg.l6 operand, and the subroutine is located in the same code segment.

  Ex:
  CALL 1234h
  CALL BX
  CALL [BX]

  CALL 1234H
- Here 1234H identifies the starting address of the subroutine.
- It is encoded as the difference between l234H and the updated value of IP-that is, the IP for the instruction following the CALL instruction.

  CALL BX
- When this instruction is executed, the contents of BX are loaded into IP and execution continues with the subroutine starting at the physical address derived from the current CS and the new value of IP.

  CALL [BX]
- By using various addressing modes of the 8086, an operand that resides in memory is used as the call to offset address.
- This represents a Meml6 type of operand.
- This instruction has its subroutine offset address at the memory location whose physical address is derived from the contents of DS and BX.
- The value stored at this memory location is loaded into IP.
- Again the current contents of CS and the new value in IP point to the first instruction of the subroutine.
- The other type of CALL instruction, the intersegment call, permits the subroutine to reside in another code segment.
- It corresponds to the Far-proc and Mem32 operands.
- These operands specify both a new offset address for IP and a new segment address for CS. In both cases, execution of the call instruction causes the contents of the CS and IP registers to be saved on the stack, and the new values are loaded into IP and CS.
- The saved values of CS and IP permit return to the main program from a different Code segment.
- Far-proc represents a 32-bit immediate operand that is stored in the four bytes that follow the opcode of the call instruction in program memory.
- These two words are loaded directly from code segment memory into IP and CS with execution of the CALL instruction.
- On the other hand, when the operand is Mem32, the pointer for the subroutine is stored as four consecutive bytes in data memory.
- The location of the first byte of the pointer can be specified indirectly by one of the 8086's memory addressing modes.

CALL DWORD [DI]

- Here the physical address of the first byte of the 4-byte pointer in memory is derived from the contents of DS and DI.

RET instruction

- Every subroutine must end by executing an instruction that returns control to the main program.
- This is the return (RET) instruction.
- Note that its execution causes the value of IP or both the values of IP and CS that were saved on the stack to be returned back to their corresponding registers.

- In general, an intra segment return results from an intra segment call and an intersegment return results from an intersegment call.
- In this way, program control is returned to the instruction that immediately follows the call instruction in program memory.

FEATURES OF PENTIUM PROCESSOR

Features of Pentium Processor are as follows:

- ✓ 64 bit data bus
- ✓ 8 bytes of data information can be transferred to and from memory in a single bus cycle
- ✓ Supports burst read and burst write back cycles
- ✓ Supports pipelining
- ✓ Instruction cache
- ✓ 8 KB of dedicated instruction cache
- ✓ Two Integer execution units, one Floating point execution unit
- ✓ Dual instruction pipeline
- ✓ 256 lines between instruction cache and prefetch buffers; allows 32 bytes to be transferred from cache to buffer
- ✓ Data cache
- ✓ 8 KB dedicate data cache gives data to execution units
- ✓ 32 byte lines
- ✓ Two parallel integer execution units
- ✓ Allows the execution of two instructions to be executed simultaneously in a single processor clock
- ✓ Floating point unit
- ✓ It includes
- ✓ Faster internal operations
- ✓ Local advanced programmable interrupt controller
- ✓ Speeds up up to 5 times for common operations including add, multiply and load, than 80486
- ✓ Branch Prediction Logic
- ✓ To reduce the time required for a branch caused by internal delays
- ✓ When a branch instruction is encountered, microprocessor begins prefetch instruction at the branch address
- ✓ Data Integrity and Error Detection
- ✓ Has significant error detection and data integrity capability
- ✓ Data parity checking is done on byte – byte basis
- ✓ Address parity checking and internal parity checking features are added
- ✓ Dual Integer Processor
- ✓ Allows execution of two instructions per clock cycle
- ✓ Functional redundancy check
- ✓ To provide maximum error detection of the processor and interface to the processor

- ✓ A second processor ‘checker’ is used to execute in lock step with the ‘master’ processor
- ✓ It checks the master’s output and compares the value with the internal computed values
- ✓ An error signal is generated in case of mismatch
- ✓ Superscalar architecture
- ✓ Three execution units
- ✓ One execution unit executes floating point instructions
- ✓ The other two (U pipe and V pipe) execute integer instructions
- ✓ Parallel execution of several instructions – superscalar processor