Question 1:

Briefly discuss the evolution Intel microprocessors.

Answer:

The following table shows Evolution of Intel Microprocessors from 8008 to the 8088:

<table>
<thead>
<tr>
<th>Product</th>
<th>8008</th>
<th>8080</th>
<th>8085</th>
<th>8086</th>
<th>8088</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>PMOS</td>
<td>NMOS</td>
<td>NMOS</td>
<td>NMOS</td>
<td>NMOS</td>
</tr>
<tr>
<td>Number of pins</td>
<td>18</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Number of transistors</td>
<td>3000</td>
<td>4500</td>
<td>6500</td>
<td>29000</td>
<td>29000</td>
</tr>
<tr>
<td>Number of instructions</td>
<td>66</td>
<td>111</td>
<td>113</td>
<td>133</td>
<td>133</td>
</tr>
<tr>
<td>Physical memory</td>
<td>16K</td>
<td>64K</td>
<td>64K</td>
<td>1M</td>
<td>1M</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Internal data bus</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>External data bus</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Address Bus</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Data types</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8/16</td>
<td>8/16</td>
</tr>
</tbody>
</table>

The following table shows Evolution of Intel Microprocessors from 8086 to the Pentium Pro:

<table>
<thead>
<tr>
<th>Product</th>
<th>8086</th>
<th>80286</th>
<th>80386</th>
<th>80486</th>
<th>Pentium</th>
<th>Pentium Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>NMOS</td>
<td>NMOS</td>
<td>CMOS</td>
<td>CMOS</td>
<td>BICMOS</td>
<td>BICMOS</td>
</tr>
<tr>
<td>Clock Rate (MHz)</td>
<td>3-10</td>
<td>10-16</td>
<td>16-33</td>
<td>25-33</td>
<td>66</td>
<td>150</td>
</tr>
<tr>
<td>Number of pins</td>
<td>40</td>
<td>68</td>
<td>132</td>
<td>168</td>
<td>273</td>
<td>387</td>
</tr>
<tr>
<td>Number of transistors</td>
<td>29000</td>
<td>134000</td>
<td>275000</td>
<td>1.2 million</td>
<td>3.1 million</td>
<td>5.5 million</td>
</tr>
<tr>
<td>Physical memory</td>
<td>1M</td>
<td>16M</td>
<td>4G</td>
<td>4G</td>
<td>4G</td>
<td>64G</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>None</td>
<td>1G</td>
<td>64T</td>
<td>64T</td>
<td>64T</td>
<td>64T</td>
</tr>
<tr>
<td>Internal data bus</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>External data bus</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Address Bus</td>
<td>20</td>
<td>24</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>Data types</td>
<td>8/16</td>
<td>8/16</td>
<td>8/16/32</td>
<td>8/16/32</td>
<td>8/16/32</td>
<td>8/16/32</td>
</tr>
</tbody>
</table>

The following table shows Evolution of Intel Microprocessors from Pentium II to the Itanium:

<table>
<thead>
<tr>
<th>Product</th>
<th>Pentium II</th>
<th>Pentium III</th>
<th>Pentium 4</th>
<th>Itanium II</th>
</tr>
</thead>
</table>

Page: 1
<table>
<thead>
<tr>
<th>Year introduced</th>
<th>1997</th>
<th>1999</th>
<th>2000</th>
<th>2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>BICMOS</td>
<td>BICMOS</td>
<td>BICMOS</td>
<td>BICMOS</td>
</tr>
<tr>
<td>Number of transistors</td>
<td>7.5 million</td>
<td>9.5 million</td>
<td>42 million</td>
<td>220 million</td>
</tr>
<tr>
<td>Physical memory</td>
<td>64G</td>
<td>64G</td>
<td>64G</td>
<td>64G</td>
</tr>
<tr>
<td>Virtual memory</td>
<td>64T</td>
<td>64T</td>
<td>64T</td>
<td>64T</td>
</tr>
<tr>
<td>Internal data bus</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>External data bus</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Address Bus</td>
<td>36</td>
<td>36</td>
<td>36</td>
<td>64</td>
</tr>
<tr>
<td>Data types</td>
<td>8/16/32</td>
<td>8/16/32</td>
<td>8/16/32</td>
<td>8/16/32/64</td>
</tr>
</tbody>
</table>

**Question 2:**

Explain the Internal architecture of 8086 (or 8088) with block diagram.

**Answer:**

The 8086 is divided into two independent functional parts, Execution Unit (EU) and Bus Interface Unit (BIU).

i) **Execution Unit (EU)**
   - The Execution Unit has a 16-bit Arithmetic Logic Unit (ALU) which can add, subtract, AND, OR, XOR, increment, decrement, complement, or shift binary numbers.
   - A decoder in EU translates instructions fetched from memory into series actions.
   - A flag register in EU contains nine active flags. Six out of the nine flags are used to indicate some condition produced by an instruction. These six conditional flags are carry flag (CF), auxiliary carry flag(AF), zero flag(ZF), parity flag (PF), sign flag(SF) and overflow flag (OF). The remaining three flags in the flag register are used to control certain operations of the processor. These control flags are trap flag(TF), interrupt flag (IF) and direction flag(DF).
   - EU has eight general purpose registers. These registers are AH,AL,BH,BL,CH,CL,DH and DL. These registers can be used individually for temporary storage of 8-bits.
   - The Stack Pointer(SP) register in the execution unit holds the 16-bit offset from the start of the segment to the memory location where a word was most recently stored in the stack.

ii) **Bus Interface Unit (BIU)**
   - BIU fetches up to six instructions bytes and store these pre fetched instructions bytes in a first in first out register set called a queue. When EU is ready for its next instruction, it reads the instruction byte from the queue.
   - Four segment registers in the BIU are used to hold the upper 16 bits of the starting addresses of the four memory segments that the processor is working at a particular time. The four segment registers are the code segment(CS), the stack segment(SS) register, the extra segment (ES) register and the data segment(DS) register.
The Instruction Pointer (IP) register holds the 16-bit address (offset address) of the next code byte within the code segment.

Internal Block diagram for 8086 (or 8088) is shown below:
Question 3:

Explain the pipelining in the 8086 (or 8088).

Answer:

The diagram shows the difference between pipeline and non pipeline.

<table>
<thead>
<tr>
<th></th>
<th>fetch1</th>
<th>execution1</th>
<th>fetch2</th>
<th>execution2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonpipelined</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipelined</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In case of non pipeline, CPU could either fetch or execute at a given time. That is, the CPU has to fetch an instruction from memory, then execute it and then fetch again and execute it and so on.
- In the case of pipeline, CPU fetches and execute at the same time.

Intel implements the concept of pipelining in the 8086/8088 by splitting the internal structure of the microprocessor into two sections, the execution unit (EU) and Bus Interface Unit (BIU). These two units work simultaneously. The BIU accesses memory and peripheral while the EU executes instructions previously fetched. BIU of the 8086/8088 has buffer or queue. The queue is 6 bytes long in the 8086 and 4 bytes long in the 8088. BIU fetches a new instruction whenever the queue has room for 2 bytes in 8086 queue and 1 byte in the 8088.
**Question 4:**

Explain briefly about the registers available in 8086/8088.

**Answer:**

The registers of the 8086/8088 fall into six categories as shown in the following table.

<table>
<thead>
<tr>
<th>Category</th>
<th>Bits</th>
<th>Register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Purpose</td>
<td>16</td>
<td>AX, BX, CX, DX</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>AH, AL, BH, BL, CH, CL, DH, DL</td>
</tr>
<tr>
<td>Pointer</td>
<td>16</td>
<td>SP (Stack Pointer), BP (Base Pointer)</td>
</tr>
<tr>
<td>Index</td>
<td>16</td>
<td>SI (Source Index), DI (Destination Index)</td>
</tr>
<tr>
<td>Instruction</td>
<td>16</td>
<td>IP (Instruction Pointer)</td>
</tr>
<tr>
<td>Segment</td>
<td>16</td>
<td>CS (Code segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment)</td>
</tr>
<tr>
<td>Flag</td>
<td>16</td>
<td>FR (Flag Register)</td>
</tr>
</tbody>
</table>

The general purpose registers in 8086/8086 microprocessor can be accessed as either 16 bit or 8 bit register.

**Question 5:**

If CS = 24F6H and IP = 634AH, show (a) the logical address (b) the offset address. Calculate (c) the physical address (d) lower range and upper range of the code segment.

**Answer:**

(a) Logical address = 24F6H:634AH  (b) Offset address = 634AH  
(c) Physical address = 24F60 + 634A = 2B2AAH  
(d) Lower range of the code segment = 24F60 + 0000 = 24F60H  
   Upper range of the code segment = 24F60 + FFFF = 34F5FH

**Question 6:**

If DS = 7FA2H and the offset is 438EH, Calculate (a) the physical address (b) lower range and upper range of the code segment (c) Show the logical address

**Answer:**

(a) Physical address = 7FA20 + 438E = 83DAEH  
(b) Lower range of the code segment = 7FA20 + 0000 = 7FA20H
Upper range of the code segment = 7FA2 + FFFF = 8FA1FH
(c) Logical address = 7FA2H:438EH

**Question 7:**

Assume memory locations with the following contents: DS: 6826H=48H and DS: 6827H=22H. Show the contents of register BX in the instruction MOV BX, [6826]

**Answer:**

According to the little endian convention used in all x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH contain the value from the offset address 6827. Hence BL=48H, BH=22H and BX=2248H

**Question 8:**

Explain briefly the concept of real mode memory addressing.

**Answer:**

- 8086 and 8088 microprocessors exclusively operate in real mode. Real mode memory addressing allows the microprocessors to address one Mega Bytes. DOS operating system requires the microprocessor to operate in the real mode.
- Segment and Offset: A combination of a Segment address and Offset address access a memory location in the real mode.
- All real mode memory addresses must consist of a segment address plus offset address. The segment address, located within one of the segment registers, defines the beginning address of 64K-byte memory segment. The offset address selects any location within the 64K byte memory address.
- Segments in the real mode always have a length of 64K bytes.
- Default Segment and Offset registers: Following table shows the combination of segment and offset address

<table>
<thead>
<tr>
<th>Segment</th>
<th>Offset</th>
<th>Special Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>IP</td>
<td>Instruction address</td>
</tr>
<tr>
<td>SS</td>
<td>SP or BP</td>
<td>Stack address</td>
</tr>
<tr>
<td>DS</td>
<td>BX, SI, DI</td>
<td>Data address</td>
</tr>
<tr>
<td>ES</td>
<td>DI</td>
<td>String destination address</td>
</tr>
</tbody>
</table>
**Question 9:**

Explain briefly the memory map of the IBM PC.

**Answer:**

The memory map of the IBM PC is shown below:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>00000H</strong></td>
<td>RAM 640K</td>
</tr>
<tr>
<td><strong>9FFFFH</strong></td>
<td>Video Display</td>
</tr>
<tr>
<td><strong>A0000H</strong></td>
<td>RAM 128K</td>
</tr>
<tr>
<td><strong>BFFFFH</strong></td>
<td>ROM 256K</td>
</tr>
<tr>
<td><strong>C0000H</strong></td>
<td></td>
</tr>
<tr>
<td><strong>FFFFFFH</strong></td>
<td></td>
</tr>
</tbody>
</table>

- The 20-bit address of the 8086/8088 allows a total of 1 megabyte (1024K bytes) of memory space with the address range 00000-FFFFF. Out of the 1 megabyte, 640K bytes with address range
00000-9FFFF were aside for RAM. The 128K bytes with address range A0000-BFFFFF were allocated to video memory.

- The remaining 256K address ranges C0000-FFFFF were allocated to ROM. This memory allocation is called memory map.

- RAM: For a program to be executed on the PC, it must be loaded first into the RAM. Operating system first allocates the available RAM for its own use and then the rest be used for applications. The task of managing RAM memory is left to the operating system.

- Video RAM: The 128K bytes with address range A0000-BFFFFF were allocated to video memory. The amount used depends on the video board installed on the PC.

- ROM: Out of 256K bytes, the 64K bytes with range F0000-FFFFF are used for BIOS (Basic input/output System) ROM. Some of the remaining space is used by various adapter cards such as network card and the rest is free.

**Question 10:**

Why stacks are needed? How are the stacks accessed?

**Answer:**

The stack is a section of read/write memory in the RAM used by the CPU to store information temporarily. The CPU needs this storage area because there are only a limited numbers of registers. There must be some place for the CPU to information safely and temporarily.

The two main registers used to access the stack are:

- SS (Stack Segment) register
- SP (Stack Pointer) registers.

These registers must be loaded before any instruction accessing the stack.

- Every register inside the x86 except segment register and SP can be stored into the stack and brought back into the microprocessor from the stack.

- Storing of microprocessor register in the stack is called a push and loading the content of stack into the microprocessor register is called pop.

- As each push is executed, the contents of the register are saved on the stack and SP is decremented by 2.

- As each pop is executed, top 2 bytes are copied into register specified and SP is incremented by 2.

**Question 11:**

Assuming that SP=1236H, AX=24B6H, DI=85C2H and DX=5F93H, show the contents of the stack as each of each of the following instructions is executed.

PUSH AX
PUSH DI
PUSH DX

**Answer:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Start:</td>
<td>After PUSH AX:</td>
<td>After PUSH DI:</td>
<td>After PUSH DX:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP=1236</td>
<td>SP=1234</td>
<td>SP=1232</td>
<td>SP=1230</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Note: When POP instruction is executed, Stack Pointer (SP) is incremented by 2)

**Question 11:**

If SS=3500H and SP=FFFEH, (a) Calculate the physical address of the stack (b) Calculate the lower range of the stack segment (c) Calculate the upper range of the stack segment (d) Show the logical address.

**Answer:**

(a) Physical address= 35000+FFFE=44FFEH
(b) Lower range of the stack segment=35000+0000=35000H
(c) Upper range of the stack segment= 35000+FFFF=44FFF
(d) Logical address=35000H: FFFEH
Question 13:

Explain briefly about flag register of 8086/8088.

Answer:

The following diagram shows bitwise flag register.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>OF</td>
<td>DF</td>
<td>IF</td>
<td>TF</td>
<td>SF</td>
<td>ZF</td>
<td>U</td>
<td>AF</td>
<td>U</td>
<td>PF</td>
</tr>
</tbody>
</table>

- CF, the Carry flag: this flag is set whenever there is a carry out, either from d7 after an 8 bit operation or from d15 after a 16 bit data operation.
- PF, the parity flag: After certain operations, the parity of the low order byte of the result is checked. If the byte has an even number of 1s, the parity is set to 1, otherwise it is cleared.
- AF, the Auxiliary carry flag: If there is a carry from d3 to d4 of an operation, then this bit is set. Otherwise this flag is cleared.
- ZF, the Zero flag: The zero is set to 1 if the result of an arithmetic or logical operation is zero, otherwise this flag is cleared.
- SF, the Sign flag: Binary representation of signed numbers use the most significant bit as the sign bit. After arithmetic or logical operations, the status of the signed bit is copied into the SF, thereby indicate the sign of the result.
- TF, the Trap flag: When this flag is set, it allows the program to single step that is the execution of one instruction at a time.
- IF, the Interrupt flag: This bit is set or cleared to enable or disable the external interrupt request.
- DF, the Direction flag: This bit is used to control the direction of string operations.
- OF, the Overflow flag: This flag is set whenever the result of a signed operation is too large, causing the order bit to overflow into the sign bit.

Question 14:

Show how the flag register is affected by
MOV AL,9CH  
MOV DH,64H  
ADD AL,DH  

**Answer:**

\[
\begin{array}{c}
9CH \quad 10011100 \\
+ \quad 64H \quad 01100100 \\
00 \quad 00000000 \\
\end{array}
\]

CF=1 since there is a carry beyond d7  
AF=1 since there is a carry from d3 to d4  
PF=1 since there is an even number of 1 in the result  
ZF=1 since the result is zero  
SF=0 since d7 of the result is zero

**Question 15:**

Show how the flag register is affected by  
MOV AX,34F5H  
ADD AX,95EBH  

**Answer:**

\[
\begin{array}{c}
34F5H \quad 0011 0100 1111 0101 \\
95EBH \quad 1001 0101 1110 1011 \\
CAE0H \quad 1100 1010 1110 0000 \\
\end{array}
\]

CF=0 since there is no carry beyond d15  
AF=1 since there is a carry from d3 to d4  
PF=0 since there is an odd number of 1s in the above lower byte  
ZF=0 since the result is not zero  
SF=1 since d15 of the result is 1

**Question 16:**

Show the flag register is affected by  
MOV AX,94C2H  
MOV BX,323EH  
ADD AX,BX  
MOV DX, AX  
MOX CX, DX  

Answer:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>94C2H</td>
<td>1001 0100 1100 0010</td>
</tr>
<tr>
<td>323EH</td>
<td>0011 0010 0011 1110</td>
</tr>
<tr>
<td>C700H</td>
<td>1100 0111 0000 0000</td>
</tr>
</tbody>
</table>

CF=0 since there is no carry beyond d15
AF=1 since there is a carry from d3 to d4
PF=1 since there an even number of 1s in the lower byte
ZF=0 since the result is not zero
SF=1 since d15 of the result is 1

Question 17:

Explain briefly x86 addressing modes.

Answer:

**Register addressing mode:**
The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed. Examples of register addressing mode follows:

- MOV BX,DX ; Copy the contents of DX into BX
- MOV ES,AX ; Copy the contents of AX into ES
- ADD AL,BH ; Add the contents of BH to the contents of AL

**Immediate addressing mode:**
In the immediate addressing mode, the source operand is a constant. In immediate addressing mode, the operand comes immediately after the code when the instruction is assembled. Examples of Immediate addressing mode follow:

- MOV AX, 2550H ; Move 2550H into AX
- MOV CX,624AH ; Move 624AH into CX
- MOV BL,45H ; Move 45H into BL

**Direct addressing mode:**
In the direct addressing mode, the data is in some memory location. The address is the offset address. Example of Direct addressing mode follows:

MOV DL,[2400H]    ; Move the contents of DS:2400H into DL

**Register Indirect addressing mode:**

In the register indirect addressing mode, the address of the memory location where operand resides is held by a register. The registers used for the purpose are SI, DI and BX. Example of Register Indirect addressing mode follows:

MOV AL,[BX]     ; The contents of the memory location pointed by DS:BX moves into AL
MOV CL,[SI]      ; The contents of the memory location pointed by DS:SI moves into CL
MOV [DI],AH     ; The contents of AH moves to the memory location pointed by DS:DI

**Based relative addressing mode:**

In the base relative addressing mode, base registers BX, BP and a displacement value, are used to calculate the effective address. The default segment used for calculation of physical address is DS for BX and SS for BP. Example of based relative addressing mode follows:

MOV CL,[BX]+10   ; Move DS:BX+10 into CL
MOV AL,[BP]+5    ; Move SS:BP into AL

**Indexed relative addressing mode:**

The indexed relative addressing mode works the same as the based relative addressing mode except that registers DI and SI holds the offset address. Example of indexed relative addressing mode follows:

MOV CL,[SI]+10    ; Move DS:SI+10 into CL
MOV AL,[DI]+5     ; Move DS:SI into AL

**Based indexed addressing mode:**

By combining the based and indexed addressing mode, a new addressing mode is derived is called based Indexed addressing mode. Example of based indexed addressing mode follows:

MOV CL,[BX][DI]+10  ; Move DS:BXDI+10 into CL
MOV AL,[BP][SI]+5   ; Move SS:BP+SI into AL

**Question 18:**

Explain briefly about Segment Override.

**Answer:**

Following table shows the offset register that can be used with four segment registers.

<table>
<thead>
<tr>
<th>Segment Register</th>
<th>Offset Register</th>
</tr>
</thead>
</table>
- The X86 microprocessor allows the program to override the default segment and use any segment register.
- To do that, the program should specify the segment in the code.
- For example, in the instruction ‘MOV AL, [BX]’, the logical address of operand is DS: BX since DS is the default segment for the pointer BX.
- To override that default, the segment register should be specified in the instruction as ‘MOV AL, ES:[BX]’. Now, the logical address of the operand is ES:BX.

Following table shows more examples for segment override.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Segment Used</th>
<th>Default Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX,CS:[BP]</td>
<td>CS:BP</td>
<td>SS:BP</td>
</tr>
<tr>
<td>MOV DX,SS:[SI]</td>
<td>SS:[SI]</td>
<td>DS:SI</td>
</tr>
</tbody>
</table>

**Question 19:**

Explain briefly the directive ‘MODEL’.

**Answer:**

- **MODEL** directive selects the size of the memory model. Options for the memory model are SMALL, MEDIUM, COMPACT and LARGE.
  - MODEL SMALL  ; Uses maximum of 64K bytes of memory for code and another 64K bytes for data.
  - MODEL MEDIUM  ; Uses maximum of 64K bytes for data but code can exceed 64K bytes of memory.
  - MODEL COMPACT  ; The data can exceed 64K bytes but the code cannot exceed 64K bytes.
  - MODEL LARGE  ; Both data and code can exceed 64K bytes but no single set of data should exceed 64K.
  - MODEL HUGE  ; Both data and code can exceed 64K bytes and a single set of data can exceed 64K.
  - MODEL TINY  ; Used with COM files in which data and code must fit into 64K bytes
Question 20:

Explain briefly the steps required to convert an assembly language program to executable program.

Answer:

Following diagram shows the steps producing an executable file.

- **.asm and .obj files**:  
  - The `.asm` file (the source file, for example file.asm) is the file created with an editor. MASM (or other)  
  - Assembler converts the .asm file into machine language (the .obj object file). In addition, MASM creates .lst, the list file and .crf, the cross reference file.

- **.lst file**:  
  - The .lst file lists all the opcodes and offset address as well as errors that MASM detected.  
  - This file can be displayed on the monitor or can be printed. The programmer uses it to debug the program.

- **.crt file**:
MASM produces .crt file, the cross reference file, which provides an alphabetical list of all symbols and labels used in the program and the program line numbers in which they are referenced. This can be useful in large program with many data segments and code segments.

- .map file:
  - When there are many segments for code or data, there is a need for locating each and number of bytes used by each. This is provided by the map file.

**Question 21:**

Explain briefly the three types unconditional jump instructions.

**Answer:**

Three types of unconditional jump instructions are short jump, near jump and far jump.

**Short Jump:**

- Short jumps are two bytes instruction. First byte is the opcode and the second byte is the displacement of range between 00 to FF.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Displacement</th>
</tr>
</thead>
</table>

- In a short jump, the address of the target must be within -128 to +127 bytes. There are backward jumps (upto -128) and forward jumps (upto +127). For backward jump, the second byte (displacement byte) is 2’s complement of the displacement value.

**Near Jump:**

- The near jump is similar to the short jump, except that the distance it can move is farther. The near Jump is a three-byte instruction that contains first byte as opcode followed by two bytes as signed 16-bit displacement.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Displacement (Low)</th>
<th>Displacement(High)</th>
</tr>
</thead>
</table>

- A near jump move control to an instruction in the current code segment in the range of -32K bytes And +32K bytes. Near jump can jump to any memory location within the current code segment and called intra-segment (within segment).

**Far Jump:**
A far jump is a 5 byte instruction. First byte contains the opcode, bytes 2 and bytes 3 contain the new Offset address and bytes 4 and 5 contain the new segment address.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IP (Low)</th>
<th>IP (High)</th>
<th>CS (Low)</th>
<th>CS (High)</th>
</tr>
</thead>
</table>

In a far jump, the control can be transferred outside the current code segment. It is an intersegment (between segment) jump. A far jump instruction sometimes appears with the FAR PTR directive.

**Question 22:**

Discuss the conditional jump instructions for 8086 microprocessor.

**Answer:**

Following table shows the various conditional jump instructions for 8086

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Condition tested</th>
<th>Operation</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA/JNBE</td>
<td>CF=0 and ZF=0</td>
<td>Jump if above / Jump if neither below nor equal</td>
<td></td>
</tr>
<tr>
<td>JAE/JNB</td>
<td>CF=0</td>
<td>Jump if above or equal / Jump if not below</td>
<td></td>
</tr>
<tr>
<td>JB/JNAE</td>
<td>CF=1</td>
<td>Jump if below / Jump if neither above nor equal</td>
<td></td>
</tr>
<tr>
<td>JBE/JNA</td>
<td>CF=1 or ZF=1</td>
<td>Jump if below or equal / Jump if not above</td>
<td></td>
</tr>
<tr>
<td>JC</td>
<td>CF=1</td>
<td>Jump if carry flag set</td>
<td></td>
</tr>
<tr>
<td>JE/JZ</td>
<td>ZF=1</td>
<td>Jump if equal / Jump if zero</td>
<td></td>
</tr>
<tr>
<td>JG/JNLE</td>
<td>ZF=0 and SF=0</td>
<td>Jump if greater than</td>
<td></td>
</tr>
<tr>
<td>JGE/JNL</td>
<td>SF=0</td>
<td>Jump if greater than or equal / Jump if not less than</td>
<td></td>
</tr>
<tr>
<td>JL/JNGE</td>
<td>SF&lt;&gt;0</td>
<td>Jump if less than / Jump if not greater</td>
<td></td>
</tr>
<tr>
<td>JLE/JNG</td>
<td>ZF=1 or SF&lt;&gt;0</td>
<td>Jump if less than or equal / Jump if not greater</td>
<td></td>
</tr>
<tr>
<td>JNC</td>
<td>CF=0</td>
<td>Jump if no carry</td>
<td></td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>ZF=0</td>
<td>Jump if not equal / Jump if not zero</td>
<td></td>
</tr>
<tr>
<td>JNO</td>
<td>OF=0</td>
<td>Jump if no overflow</td>
<td></td>
</tr>
<tr>
<td>JNP/JPO</td>
<td>PF=0</td>
<td>Jump if not parity / Jump if parity odd</td>
<td></td>
</tr>
<tr>
<td>JNS</td>
<td>SF=0</td>
<td>Jump if no sign</td>
<td></td>
</tr>
<tr>
<td>JO</td>
<td>OF=1</td>
<td>Jump if overflow is set</td>
<td></td>
</tr>
<tr>
<td>JP/JPE</td>
<td>PF=1</td>
<td>Jump if parity is set / Jump if parity even</td>
<td></td>
</tr>
<tr>
<td>JS</td>
<td>SF=1</td>
<td>Jump if sign is set</td>
<td></td>
</tr>
<tr>
<td>JCXZ</td>
<td>CX=0</td>
<td>Jump if CX is zero</td>
<td></td>
</tr>
</tbody>
</table>

**Question 23:**

Explain briefly the CALL instruction for 8086 microprocessor.

**Answer:**
The CALL instruction transfers the flow of the program to the procedure. The CALL instruction differs the jump instruction because a CALL instruction saves the return address on the stack. CALL instruction returns the control as per the address saved on the stack when RET is executed. There are two types of CALL; these are Near CALL and Far CALL.

**Near CALL:**

- The near CALL instruction is three bytes long; the first byte contains opcode, and second and third bytes contain the displacement in the range of -32K to +32K in the 8086.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Displacement (Low)</th>
<th>Displacement (High)</th>
</tr>
</thead>
</table>

- When the near CALL is executed, it first pushes the offset address of the next instruction on the stack.
- The offset of the next instruction appears in the instruction pointer (IP). After saving the return address, it then adds the displacement from bytes 2 and 3 to the IP to transfer the control to the procedure.

**Far CALL:**

- The far CALL is a five-byte instruction that contains an opcode, followed by new contents of the IP and CS. Bytes 2 and 3 contain the new contents of the IP and bytes 4 and 5 contain the new contents for CS.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IP (Low)</th>
<th>IP (High)</th>
<th>CS (Low)</th>
<th>CS (High)</th>
</tr>
</thead>
</table>

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

**Question 24:**

Explain briefly the following data directives with examples.

(a) ORG  (b) DB  (c) DUP  (d) DW  (e) EQU  (f) DD  (g) DQ  (h) DT

**Answer:**

(a) ORG:
- ORG (Origin) is used to indicate the beginning of the offset address. The number that comes after ORG can be either in hexadecimal or in decimal. For example, ORG 30H
  
  ORG 100D

(b) DB:

- DB (Define Byte) allows allocation of memory in byte-sized chunks. DB can be used to define numbers in decimal, binary, hexadecimal and ASCII. For example, DB 25D
  
  DB 1001001B
  
  DB 'A'

(c) DUP:

- DUP (Duplicate) is used to duplicate a given numbers of characters.
- For example, DATA6 DB 6 DUP(0FFH); Fill 6 bytes with FFH

(d) DW

- DW (Define Word) is used allocate 2 bytes (one word) at a time.
- For example, DATA0 DW 245AH

(e) EQU

- EQU (Equate) is used to define a constant without occupying a memory location. EQU does not set aside storage for the data item but associate a constant value with a data label.
- For example, COUNT EQU 25

(f) DD:

- DD (Define Doubeword) is the directive to allocate memory locations that are 4 bytes in size. The can be in decimal, binary or hexadecimal.
- For example,
  
  DATA1 DD 45731023 ; Decimal
  
  DATA2 DD 10011100110011110011110011001111B ; Binary
  
  DATA3 DD 5CDA67ACH ; Hexadecimal

(g) DQ:

- DQ (Define quadword) is used to locate 8 bytes (four words) in size.
• For example, DATA 4 DQ 34H

(h) DT:

• DT(Define ten bytes) is used memory allocation of packed BCD numbers.
• For example, DATA5 DT 45328965 ;BCD

Question 25:

Write an assembly language program (ALP) to multiply two 16-bit binary numbers to give 32-bit result using full segment definition.

Answer:

Following is an ALP to multiply two 16-bit binary numbers using full segment definition

DATA_HERE SEGMENT
    MULTIPlicAND DW 3B2AH ;First word here
    MULTIPLIER DW 204AH  ; Second word here
    PRODUCT 2 DUP(0)     ; Result of the multiplication here
DATA_HERE ENDS

CODE_HERE SEGMENT
    ASSUME CS:CODE_HERE, DS:DATA_HERE

START:    MOV AX, DATA_HERE  ; initialize DS register
          MOV DS,AX
          MOV AX, MULTIPlicAND ; Get the first word
          MUL MULTIPLIER  ; Multiply by the second word
          MOV PRODUCT, AX ; Store the low word of the result
          MOV PRODUCT+2, DX ; Store the high word of the result
          INT 3 ; Wait for command from user

CODE_HERE ENDS

END START
Question 26:
Write an assembly language program (ALP) adds a profit factor to each of the 8-elements in a COST Array and put the result in a PRICES array using full segment definition.

Answer:
Following is an ALP add a profit factor to each of the 8-elements in a COST Array and put the result in a PRICES array using full segment definition.

```
PROFIT EQU 15H ; profit = 15H

DATA_HERE SEGMENT
COST DB 20H, 28H, 15H, 26H, 19H, 27H, 16H, 29H
PRICES DB 8 DUP(0)
DATA_HERE ENDS

CODE_HERE SEGMENT
ASSUME CS:CODE_HERE, DS:DATA_HERE
START: MOV AX,DATA_HERE ; Initialize the data segment
        MOV DS,AX
        MOV CX,0008H ;Initialize the counter
        MOV BX,0000H ; Initialize the pointer
DO_NEXT: MOV AL, COST[BX] ; Get the cost
          ADD AL,PROFIT ; Add the profit
          DAA ;Decimal adjust
          MOV PRICES[BX],AL ;Store the result in price
          INC BX ;Point to next element
          DEC CX ; decrement the counter
```

JNZ DO_NEXT ;If not the last element, do again

CODE_HERE ENDS

END START

Question 27:

Draw the commonly used flowchart symbols

Answer:

Following are the commonly used Flowchart symbols:
Question 28:

Draw a flowchart to add 5 bytes and store the result in SUM. Also write the pseudo code.

Answer:

Following is the flowchart to add 5 bytes and store the result in SUM.
Pseudo code:

Count=5
Repeat
    Add next byte
    Increment Pointer
    Decrement Count
Until Count=0

Store SUM
Stop
Store SUM

Question 29:

Write ALP program that adds 5 bytes of data and saves the result. The data should be the following hex numbers 25h,12h,15h,1fh and 2bh.

Answer:

In .Model:

.data
    data_in db 25h,12h,15h,1fh,2bh
    sum dw ?
.code
    mov ax,@data
    mov ds,ax
    mov cx,05
    mov bx,offset data_in
    mov ax,0
    again: add ax,[bx]
    inc bx
    dec cx
    jnz again
    mov sum,ax
    mov ah,4ch
    int 21h
.end

In Full segment definitions:

    Data Segment
        data_in db 25h,12h,15h,1fh,2bh
        sum dw ?
    Data ends
    Code segment
    Assume CS:Code, DS:Data
    Start:
        mov ax,Data
        mov ds,ax
mov cx,05
mov bx,offset data_in
mov ax,0
again: add ax,[bx]
    inc bx
dec cx
jnz again
mov sum,ax
mov ah,4ch
int 21h
end start
Code ends

Question 30:

Write ALP program that adds 4 words of data and saves the result. The data should be the following hex numbers 234dh, 1de6h, 3bc7h, 1fh and 566ah.

Answer:

In .Model:
.model small
.stack 64
.data
data_in dw 234dh,1de6h,3bc7h,566ah
sum dw ?
.code
mov ax,@data
mov ds,ax
mov cx,04
mov bx,offset data_in
mov ax,0
again: add ax,[bx]
    inc bx
    inc bx
    dec cx
    jnz again
mov sum,ax
mov ah,4ch
int 21h
In Full segment definitions:

Data Segment
  data_in dw 234dh,1de6h,3bc7h,566ah
  sum dw ?
Data ends
Code segment
Assume CS:Code, DS:Data
Start:
  mov ax,Data
  mov ds,ax
  mov cx,04
  mov bx,offset data_in
  mov ax,0
again: add ax,[bx]
  inc bx
  inc bx
  dec cx
  jnz again
  mov sum,ax
  mov ah,4ch
  int 21h
end start
Code ends

Question 31:

Write ALP program that transfer 6 bytes of data from memory locations with offset of 0010h to memory locations with offset of 0028h.

Answer:

In .Model:

.model small
.stack 64
.data
org 10h
data_in    db  25h, 4fh, 85h, 1fh, 2bh, 1ch
    org 28h
data_copy db 6 dup(?)
.code
mov ax,@data
mov ds,ax
mov si,offset data_in
mov di,offset data_copy
mov cx,06
again:
    mov al,[si]
    mov [di],al
    inc si
    inc di
    dec cx
    jnz again
    mov ah,4ch
    int 21h
.end

**In Full segment definitions:**

Data Segment
    org 10h
    data_in    db  25h, 4fh, 85h, 1fh, 2bh, 1ch
    org 28h
    data_copy db 6 dup(?)

Data ends

Code segment
Assume CS:Code, DS:Data
Start:
    mov ax,Data
    mov ds,ax
    mov si,offset data_in
    mov di,offset data_copy
    mov cx,06
again:
    mov al,[si]
    mov [di],al
    inc si
    inc di
dec cx
jnz again
mov ah,4ch
int 21h
end
Code ends

Question 32:
Write ALP program to count the number of 1s in a word. Provide the count in BCD.

Answer:
In .Model:
.model small
.stack 64
.data
dataw dw 97b2h
count db ?
.code
mov ax,@data
mov ds,ax
mov al,0 ;Clear the AL register to keep the number of 1s in BCD.
mov dl,16 ;Rotate 16 times
mov bx,dataw
again: rol bx,1
jnc next
add al,1
next: dec dl
jnz again
da
mov count,al
mov ah,4ch
int 21h
end

In Full segment definitions:
Data Segment
dataw dw 97b2h
count db ?
Data ends

Code segment
Assume CS:Code, DS:Data

Start:

mov ax,Data
mov ds,ax
mov al,0 ;Clear the AL register to keep the number of 1s in BCD.
mov dl,16 ;Rotate 16 times
mov bx,dataw
again: rol bx,1
jnc next
add al,1
next: dec dl
jnz again
da
mov count,al
mov ah,4ch
int 21h

end start

Code ends