Q1) Explain the execution steps of 32 bit and 64 bit.

Ans:

If disp.asm is our program file then steps for execution are as below :-

$ nasm –f elf64 -o disp.o disp.asm

This command creates an object file disp.o with elf32 file format.

$ ld –o disp disp.o

This command creates an executable with a name hello from disp.o object file

$ ./disp

This command executes disp program

Q2) Give the explanation of section .data, section .text, section .bss

Ans: https://www.tutorialspoint.com/assembly_programming/assembly_basic_syntax.htm

https://mcuoneclipse.com/2013/04/14/text-data-and-bss-code-and-data-size-explained/

The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

The syntax for declaring data section is −

```
section.data
```

The *bss* Section

The **bss** section is used for declaring variables. The syntax for declaring bss section is −

```
section.bss
```

The *text* section

The **text** section is used for keeping the actual code. This section must begin with the declaration **global _start**, which tells the kernel where the program execution begins.

The syntax for declaring text section is −

```
section.text
   global _start
_start:
```

Q3) Why we define “2”? in macro disp_msg 2

Ans: macro declaration with 4 parameters

Q4) Why we define %1 and %2 in macro?

Ans:

Q5) Explain the type of directives?

Ans: https://www.codemiles.com/pic-assembly/assembler-directives-types-t10907.html?mobile=off

1.Control Directives:

- **INCLUDE** Include an additional file in a program
An application of this directive has the effect as though the entire file was copied to a place where the "include" directive was found.

Code:

```
#include <file_name>          ; System file
#include “file_name”          ; User file
```

2. Data Directives:

- **DB** Defining one byte data
Syntax:

Code:

```
[<label>]db <term> [, <term>,.....,<term>]
```

3. Object File Directives:

- **PROCESSOR**
Defining microcontroller model
Syntax:

Code:

```
Processor <microcontroller_type>
```

4. macro directives:

- macro
- endm
- exitm

Sum_of_3 macro arg1, arg2, arg3 ; WREG <- [arg1]+[arg2]+[arg3]

Code:

```
movf arg1,W,A
addwf arg2,W,A
addwf arg3,W,A
endm
```

Q6) What is difference bet ax,eax, rax register?

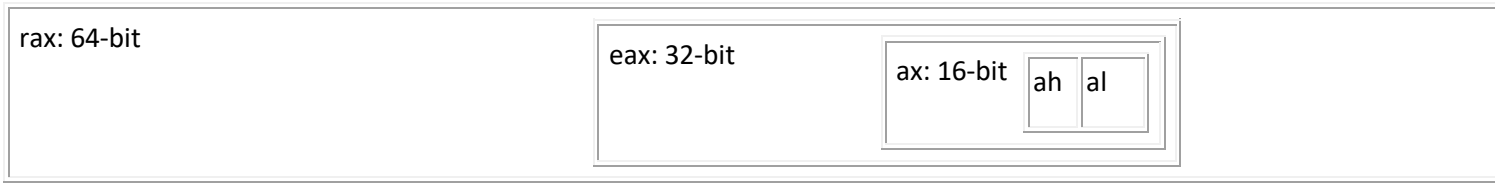
Ans: Like C++ variables, registers are actually available in several sizes:

- rax is the 64-bit, "long" size register. It was added in 2003 during the transition to 64-bit processors.
- eax is the 32-bit, "int" size register. It was added in 1985 during the transition to 32-bit processors with the 80386 CPU. I'm in the habit of using this register size, since they

also work in 32 bit mode, although I'm trying to use the longer rax registers for everything.

- ax is the 16-bit, "short" size register. It was added in 1979 with the 8086 CPU, but is used in DOS or BIOS code to this day.
- al and ah are the 8-bit, "char" size registers. al is the low 8 bits, ah is the high 8 bits. They're pretty similar to the old 8-bit registers of the 8008 back in 1972.

Curiously, you can write a 64-bit value into rax, then read off the low 32 bits from eax, or the low 16 bitx from ax, or the low 8 bits from al--it's just one register, but they keep on extending it!



Q7) What is Full Form of NASM?

Ans: The **Netwide Assembler** (**NASM**) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. **NASM** is considered to be one of the most popular assemblers for Linux.

The Netwide Assembler (**NASM**) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs. **NASM** is considered to be one of the most popular assemblers for Linux.

Q8) Explain the string Operations?

Ans: ba pg 63

https://www.geeksforgeeks.org/string-manipulation-instructions-8086-microprocessor/

https://www.tutorialspoint.com/string-manipulation-instructions-in-8086-microprocessor

OPCODE	OPERAND	EXPLANATION	EXAMPLE
REP	instruction	repeat the given instruction till CX != 0	REP MOVSB
REPE	instruction	repeat the given instruction while CX = 0	REPE
REPZ	instruction	repeat the given instruction while ZF = 1	REPZ
REPNE	instruction	repeat the given instruction while CX != 0	REPNE
REPNZ	instruction	repeat the given instruction while ZF = 0	REPNZ

OPCODE	OPERAND	EXPLANATION	EXAMPLE
MOVSB	none	moves contents of byte given by DS:SI into ES:DI	MOVSB
MOVSW	none	moves contents of word given by DS:SI into ES:DI	MOVSW
MOVD	none	moves contents of double word given by DS:SI into ES:DI	MOVD
LODSB	none	moves the byte at address DS:SI into AL; SI is incr/decr by 1	LODSB
LODSW	none	moves the word at address DS: SI into AX; SI is incr/decr by 2	LODSW
LODSD	none	moves the double word at address DS:SI into EAX; SI is incr/decr by 4	LODSD
STOSB	none	moves contents of AL to byte address given by ES:DI; DI is incr/dec by 1	STOSB
STOSW	none	moves the contents of AX to the word address given by ES:DI; DI is incr/decr by 2	STOSW
STOSD	none	moves contents of EAX to the DOUBLE WORD address given by ES:DI; DI is incr/decr by 4	STOSD
SCASB	none	compares byte at ES:DI with AL and sets flags according to result	SCASB
SCASW	none	compares word at ES:DI with AX and sets flags	SCASW
SCASD	none	compares double word at ES:DI with EAX and sets flags	SCASD
CMPSB	none	compares byte at ES:DI with byte at DS:SI and sets flags	CMPSB
CMPSW	none	compares word at ES:DI with word at DS:SI and sets flags	CMPSW
CMPSD	none	compares double word at ES:DI with double word at DS:SI and sets flags	CMPSD

Q9) What is meaning of resb , resw?

Ans: `RESB`, `RESW`, `RESD`, `RESQ` and `REST` are designed to be used in the BSS section of a module: they declare *uninitialised* storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. As stated in section 2.2.7, NASM does not support the MASM/TASM syntax of reserving uninitialised space by writing `DW ?` or similar things: this is what it does instead. The operand to a `RESB`-type pseudo-instruction is a *critical expression*: see section 3.7.

For example:

```
buffer:   resb 64               ; reserve 64 bytes
wordvar:  resw 1                ; reserve a word
realarray resq 10               ; array of ten reals
```

Q10) Why we use global_start?

Ans: The **text** section is used for keeping the actual code. This section must begin with the declaration **global** _**start**, which tells the kernel where the program execution begins.

The syntax for declaring text section is −

```
section.text
   global _start
_start:
```

Q11) Why we use macro?

Ans: https://jbwyatt.com/253/emu/asm_tutorial_10.html

- A Macro is similar to a procedure but is not invoked by the main program. Instead, the Macro code is pasted into the main program wherever the macro name is written in the main program
- A Macro is simply accessed by writing its name. The entire macro code is pasted at the location by the assemble
- Increases the size of the program
- Executes faster as return address is not needed to be stored into the stack, hence push and pop is not needed
- Does not depend on the stack

A **macro** is **used** to automate a task that you perform repeatedly or on a regular basis. ... You can record or build a **macro** and then **run** it to automatically repeat that series of steps or actions. Tasks performed by **macros** are typically repetitive in nature and can provide significant time savings

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it.

- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making

the output executable file larger and larger, each time all instructions of a macro are inserted.

Q12) Difference between macros and procedure

Ans:

	PROCEDURE (FUNCTION)	MACRO
1	A procedure (Subroutine/ Function) is a set of instruction needed repeatedly by the program. It is **stored as a subroutine and invoked from several places by the main program.**	A Macro is similar to a procedure but is not invoked by the main program. Instead, the **Macro code is pasted into the main program wherever the macro name is written in the main program.**
2	A subroutine is **invoked by a CALL** instruction and control returns by a RET instruction.	A Macro is simply accessed by **writing its name**. The entire macro code is pasted at the location by the assembler.
3	**Reduces the size** of the program	**Increases the size** of the program
4	**Executes slower** as time is wasted to push and pop the return address in the stack.	**Executes faster** as return address is not needed to be stored into the stack, hence push and pop is not needed.
5	**Depends on the stack**	**Does not depend on the stack**

Difference between Macro and Procedure :

S.No.	MACRO	PROCEDURE
01.	Macro definition contains a set of instruction to support modular programming.	Procedure contains a set of instructions which can be called repetitively which can perform a specific task.
02.	It is used for small set of instructions mostly less than ten instructions.	It is used for large set of instructions mostly more than ten instructions.
03.	In case of macro memory requirement is high.	In case of procedure memory requirement is less.
04.	CALL and RET instruction/statements are not required in macro.	CALL and RET instruction/statements are required in procedure.
05.	Assembler directive MACRO is used to define macro and assembler directive ENDM is used to indicate the body is over.	Assembler directive PROC is used to define procedure and assembler directive ENDP is used to indicate the body is over.
06.	Execution time of macro is less than it executes faster than procedure.	Execution time of procedures is high as it executes slower than macro.
07.	Here machine code is created multiple times as each time machine code is generated when macro is called.	Here machine code is created only once, it is generated only once when the procedure is defined.
08.	In a macro parameter is passed as part of statement that calls macro.	In a procedure parameters are passed in registers and memory locations of stack.
09.	Overhead time does not take place as there is no calling and returning.	Overhead time takes place during calling procedure and returning control to calling program.

Q13) Explain Divide Instruction

Ans: BA – 45,48

https://microcontrollerslab.com/8086-integer-division-instructions-assembly-programming/

Q14+

18)AAD [Binary Adjust before Division]

This instruction converts the unpacked BCD digits in AH and Al into a Packed BCD in AL.
AAD updates PF, SF ZF; But **OF, AF, CF** are **undefined** after the instruction.
Eg: Assume

CL = 07H.
AH = 04.
AL = 03.
∴ AX = 0403H … unpacked BCD for $(43)_{10}$

Then **AAD** gives

AX = 002BH … i.e. $(43)_{10}$

Now **DIV CL** gives (divide AX by unpacked BCD in CL)

AL = Quotient = 06 … unpacked BCD
AH = Remainder = 01 … unpacked BCD

The DIV/IDIV Instructions

The division operation generates two elements - a **quotient** and a **remainder**. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.

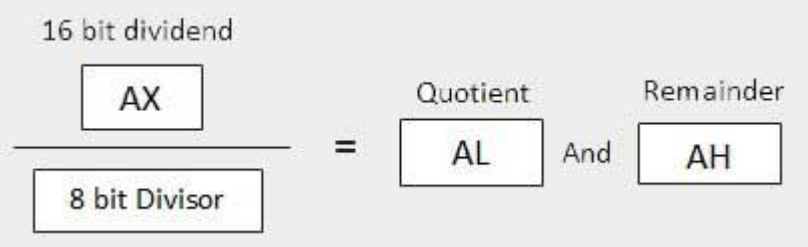
The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.

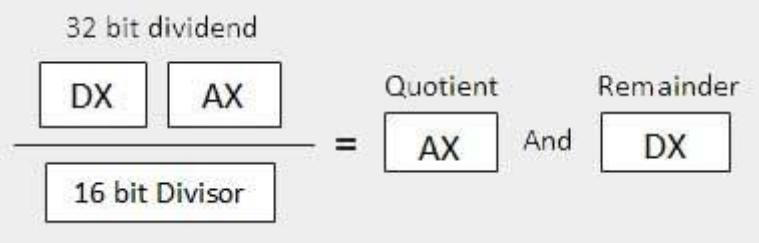
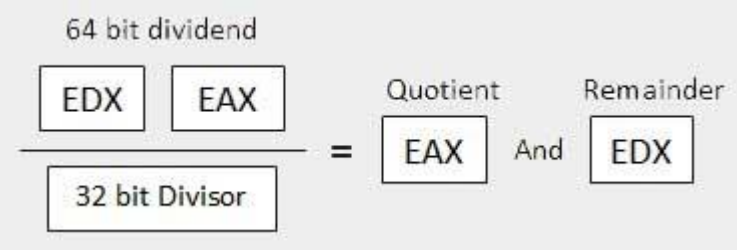
Syntax

The format for the DIV/IDIV instruction −

```
DIV/IDIV      divisor
```

The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size −

Sr.No.	Scenarios
1	**When the divisor is 1 byte −** The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register. 

2	**When the divisor is 1 word −** The dividend is assumed to be 32 bits long and in the DX:AX registers. The high-order 16 bits are in DX and the low-order 16 bits are in AX. After division, the 16-bit quotient goes to the AX register and the 16-bit remainder goes to the DX register. 
3	**When the divisor is doubleword −** The dividend is assumed to be 64 bits long and in the EDX:EAX registers. The high-order 32 bits are in EDX and the low-order 32 bits are in EAX. After division, the 32-bit quotient goes to the EAX register and the 32-bit remainder goes to the EDX register. 

Example

The following example divides 8 with 2. The **dividend 8** is stored in the **16-bit AX register** and the **divisor 2** is stored in the **8-bit BL register**.

Live Demo

```
section      .text
   global _start     ;must be declared for using gcc

_start:              ;tell linker entry point
   mov ax,'8'
   sub     ax, '0'

   mov       bl, '2'
   sub     bl, '0'
   div       bl
   add ax, '0'

   mov       [res], ax
   mov ecx,msg
   mov edx, len
   mov ebx,1  ;file descriptor (stdout)
   mov eax,4  ;system call number (sys_write)
   int 0x80   ;call kernel

   mov ecx,res
   mov edx, 1
   mov ebx,1  ;file descriptor (stdout)
```

```
   mov eax,4  ;system call number (sys_write)
   int 0x80   ;call kernel

   mov eax,1  ;system call number (sys_exit)
   int 0x80   ;call kernel

section .data
msg db "The result is:", 0xA,0xD
len equ $- msg
segment .bss
res resb 1
```

When the above code is compiled and executed, it produces the following result −

```
The result is:
4
```

The DIV instruction divides unsigned numbers, and IDIV divides signed numbers. Both return a quotient and a remainder. Table 4.1 summarizes the division operations. The dividend is the number to be divided, and the divisor is the number to divide by. The quotient is the result. The divisor can be in any register or memory location but cannot be an immediate value.

Size of Operand	Dividend Register	Size of Divisor	Quotient	Remainder
16 bits	AX	8 bits	AL	AH
32 bits	DX:AX	16 bits	AX	DX
64 bits (80386and 80486)	EDX:EAX	32 bits	EAX	EDX

Q14) Write DIV instruction syntax

Ans: BA 45

7) **DIV source**(**unsigned** 8/16-bit register – divisor)
This instruction is used for **UNSIGNED** division.
Divides a **WORD by a BYTE**, OR a **DOUBLE WORD by a WORD.**
If the **divisor is 8-bit** then the **dividend is in AX** register.
After division, the **quotient is in AL** and the **Remainder in AH**.
If the **divisor is 16-bit** then the **dividend is in DX-AX** registers.
After division, the **quotient is in AX** and the **Remainder in DX**.

Source: Register, Memory Location ☺ **For doubts contact Bharat Sir on 98204 08217**
ALL flags are **undefined** after DIV instruction.

Eg: **DIV BL** ; *AX ÷ BL :- AL ← Quotient; AH ← Remainder*
DIV BX ; *{DX,AX} ÷ BX :- AX ← Quotient; DX ← Remainder*

Please Note: If the divisor is 0 or the result is too large to fit in AL (or AX for 16-bit divisor), them 8086 does a Type 0 interrupt (Divide Error).

8) **IDIV source**(**signed** 8/16-bit register – divisor)
Same as DIV except that it is used for **SIGNED** division.

https://microcontrollerslab.com/8086-integer-division-instructions-assembly-programming/#:~:text=AAD-,8086%20DIV%20Instruction%20(%20Unsigned%20Operands),register%20or%20a%20memory%20location.

DIV source(unsigned 8/16-bit register – divisor) This instruction is used for UNSIGNED division. Divides a WORD by a BYTE, OR a DOUBLE WORD by a WORD. If the divisor is 8-bit then the dividend is in AX register. After division, the

quotient is in AL and the Remainder in AH. If the divisor is 16-bit then the dividend is in DX-AX registers. After division, the quotient is in AX and the Remainder in DX

Syntax: DIV divisor IDIV divisor Byte Form: The divisor is eight bit register or memory byte The 16 – bit dividend is assumed to be in AX. After division 8-bit quotient is in AL and 8-bit remainder in AH.

Q15) Explain ADD instruction

Ans: BA 45,47

1) **ADD/ADC destination, source**
Adds the source to the destination and stores the **result** back **in** the **destination**.
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.
Eg: **ADD AL, 25H** ; *AL ← AL + 25H*
ADD BL, CL ; *BL ← BL + CL*
ADD BX, CX ; *BX ← BX + CX*
ADC BX, CX ; *BX ← BX + CX + Carry Flag*

2) **SUB/SBB destination, source**
It is similar to ADD/ADC except that it does subtraction.

https://microcontrollerslab.com/8086-integer-arithmetic-instructions-assembly-language-programming/

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.
- This instruction adds the data of destination and source operand and stores the result in destination. Both operands should be of same type i.e. words or bytes otherwise assembler will generate an error. It supports following operands:

Source	Destination	Example
Register	Register	ADD AX, BX
Register	Immediate	ADD CL, 2DH
Register	Memory Address	ADD [204CH], AL ADD [BP][SI]+23, DX

Memory Address	Register	ADD CL, [BX+15]

- If the sum of two operands exceeds the size of destination operand, then it would set the carry flag to 1. The ADD instruction can affect AF, CF, OF, PF, SF, ZF flags depending upon the result. If the result is zero, the ZF=1. Negative result sets SF to 1.

Q16) Explain 80386 Flag Register

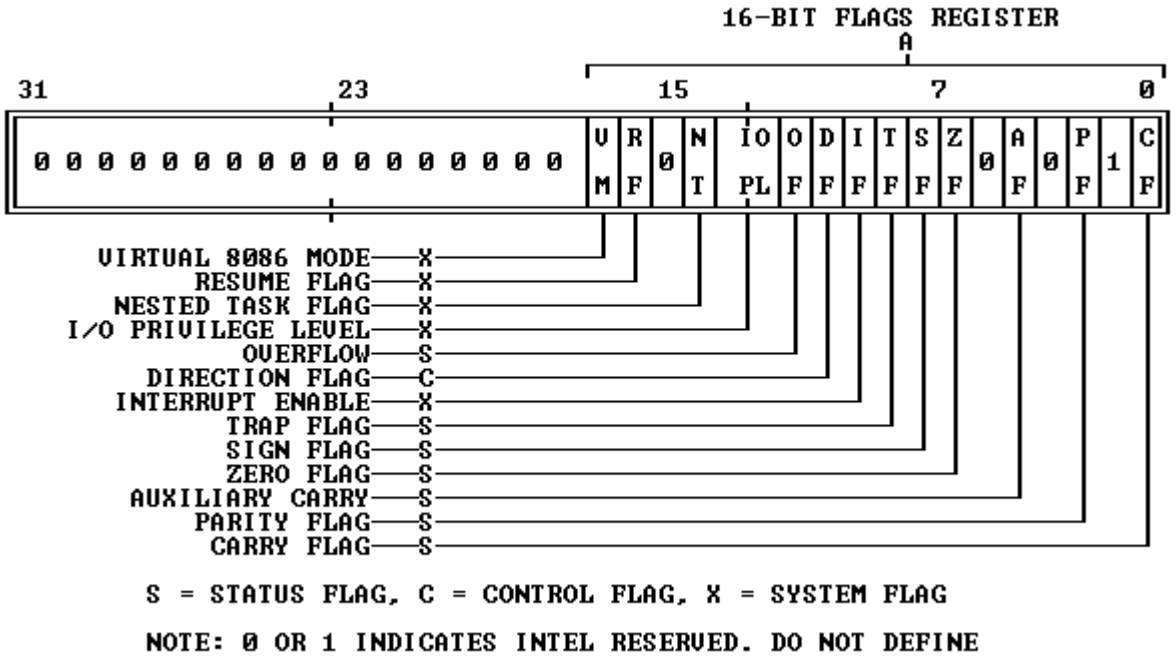
Ans: BA Vid

The flags register is a 32-bit register named EFLAGS. Figure 2-8 defines the bits within this register. The flags control certain operations and indicate the status of the 80386.

The low-order 16 bits of EFLAGS is named FLAGS and can be treated as a unit. This feature is useful when executing 8086 and 80286 code, because this part of EFLAGS is identical to the FLAGS register of the 8086 and the 80286.

The flags may be considered in three groups: the status flags, the control flags, and the systems flags. Discussion of the systems flags is delayed until Part II.

Figure 2-8. EFLAGS Register



2.3.4.1 Status Flags

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

2.3.4.2 Control Flag

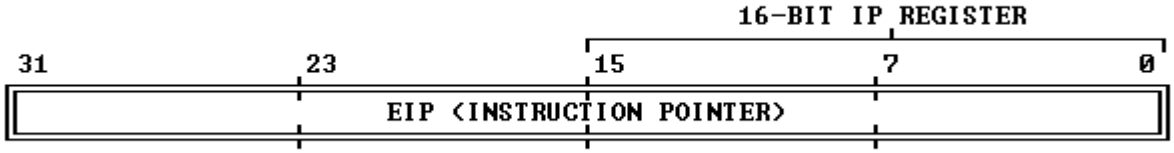
The control flag DF of the EFLAGS register controls string instructions. DF (Direction Flag, bit 10) Setting DF causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses. Clearing DF causes string instructions to auto-increment, or to process strings from low addresses to high addresses.

2.3.4.3 Instruction Pointer

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

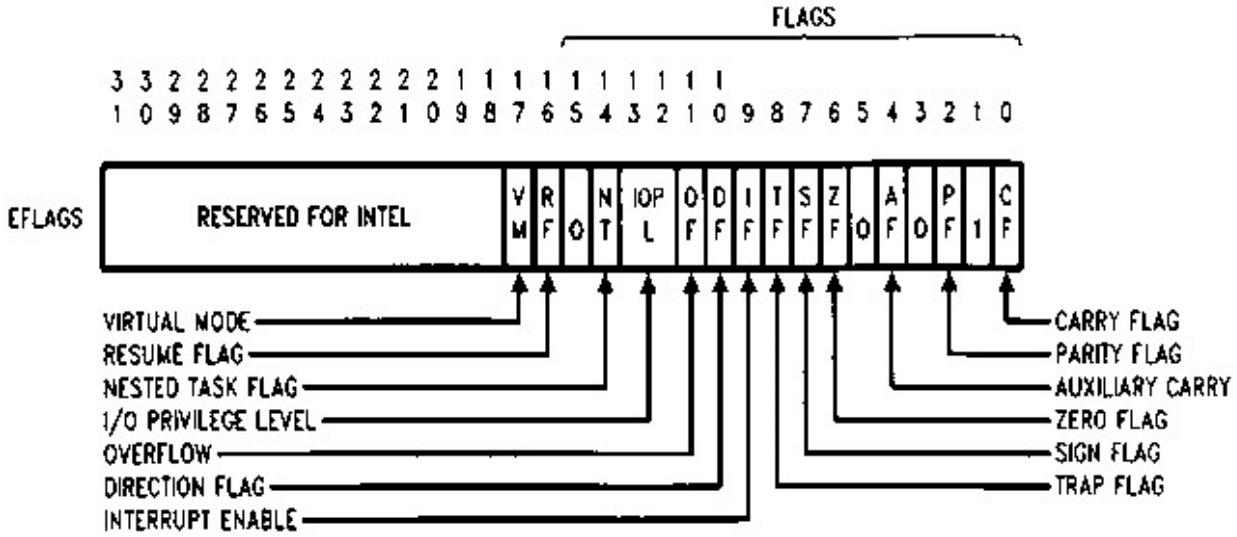
As Figure 2-9 shows, the low-order 16 bits of EIP is named IP and can be used by the processor as a unit. This feature is useful when executing instructions designed for the 8086 and 80286 processors.

Figure 2-9. Instruction Pointer Register



Flag Register of 80386 micro processor

Flag Register of 80386



[source: Intel386 DX Manual]

Note in these descriptions, ``set" means ``set to 1," and ``reset" means ``reset to 0."

VM (Virtual 8086 Mode, bit 17)

The VM bit provides Virtual 8086 Mode within Protected Mode. If set while the 80386 is in Protected Mode, the 80386 will switch to Virtual 8086 operation, handling segment loads as the 8086 does, but generating exception 13 faults on privileged opcodes.

RF (Resume Flag, bit 16)

The RF flag is used in conjunction with the debug register breakpoints. It is checked at instruction boundaries before breakpoint processing. When RF is set, it causes any debug fault to be ignored on the next instruction. RF is then automatically reset at the successful completion of every instruction.

NT (Nested Task, bit 14)

This flag applies to Protected Mode. NT is set to indicate that the execution of this task is nested within another task. If set, it indicates that the current nested task's Task State Segment (TSS) has a valid back link to the previous task's TSS. This bit is set or reset by control transfers to other tasks.

IOPL (Input/Output Privilege Level, bits 12-13)

This two-bit field applies to Protected Mode. IOPL indicates the numerically maximum CPL (current privilege level) value permitted to execute I/O instructions without generating an exception 13 fault or consulting the I/O Permission Bitmap. It also indicates the maximum CPL value allowing alteration of the IF (INTR Enable Flag) bit when new values are popped into the EFLAG register.

OF (Overflow Flag, bit 11)

OF is set if the operation resulted in a signed overflow. Signed overflow occurs when the operation resulted in carry/borrow into the sign bit (high-order bit) of the result but did not result in a carry/borrow out of the high order bit, or vice-versa. For 8/16/32 bit operations, OF is set according to overflow at bit 7/15/31, respectively.

DF (Direction Flag, bit 10)

DF defines whether ESI and/or EDI registers post decrement or post increment during the string instructions. Post increment occurs if DF is reset. Post decrement occurs if DF is set.

IF (INTR Enable Flag, bit 9)

The IF flag, when set, allows recognition of external interrupts signaled on the INTR pin.

When IF is reset, external interrupts signaled on the INTR are not recognized. IOPL indicates the maximum CPL value allowing alteration of the IF bit when new values are popped into EFLAGS or FLAGS.

TF (Trap Enable Flag, bit 8)

TF controls the generation of exception 1trap when single-stepping through code. When TF is set, the 80386 generates an exception 1 trap after the next instruction is executed. When TF is reset, exception 1 traps occur only as a function of the breakpoint addresses loaded into debug registers DR0±DR3.

SF (Sign Flag, bit 7)

SF is set if the high-order bit of the result is set, it is reset otherwise. For 8-, 16-, 32-bit operations, SF reflects the state of bit 7, 15, 31 respectively.

ZF (Zero Flag, bit 6)

ZF is set if all bits of the result are 0. Otherwise it is reset.

AF (Auxiliary Carry Flag, bit 4)

The Auxiliary Flag is used to simplify the addition and subtraction of packed BCD quantities. AF is set if the operation resulted in a carry out of bit 3 (addition) or a borrow into bit 3 (subtraction). Otherwise AF is reset. AF is affected by carry out of, or borrow into bit 3 only, regardless of overall operand length: 8, 16 or 32 bits.

PF (Parity Flags, bit 2)

PF is set if the low-order eight bits of the operation contains an even number of ``1's'' (even parity). PF is reset if the low-order eight bits have odd parity. PF is a function of only the low-order eight bits, regardless of operand size.

CF (Carry Flag, bit 0)

CF is set if the operation resulted in a carry out of (addition), or a borrow into (subtraction) the high-order bit. Otherwise CF is reset. For 8-, 16- or 32-bit operations, CF is set according to carry/borrow at bit 7, 15 or 31, respectively.

https://processorarch.blogspot.com/2013/01/v-behaviorurldefaultvmlo.html

1. Flag register is a part of EU (Execution Unit). It is a 16 bit register with each bit corresponding to a flip-flop.
2. A flag is a flip-flop. It indicates some condition produced by the execution of an instruction. For example the zero flag (ZF) will set if the result of execution of an instruction is zero

The **FLAGS** register is the status register in Intel x86 microprocessors that contains the current state of the processor. This register is 16 bits wide. Its successors, the **EFLAGS** and **RFLAGS** registers, are 32 bits and 64 bits wide, respectively. The wider registers retain compatibility with their smaller predecessors.

The fixed bits at bit positions 1, 3 and 5, and carry, parity, adjust, zero and sign flags are inherited from an even earlier architecture, 8080. The adjust flag used to be called auxiliary carry bit in 8080 and half-carry bit in the Zilog Z80 architecture.

Q17) Explain SUB Instruction. With syntax.

Ans: BA 45

1) **ADD/ADC destination, source**
Adds the source to the destination and stores the **result** back **in** the **destination**.
Source: Register, Memory Location, Immediate Number
Destination: Register
Both, source and destination have to be of the same size.
ADC also adds the carry into the result.
Eg: **ADD AL, 25H** ; *AL ← AL + 25H*
ADD BL, CL ; *BL ← BL + CL*
ADD BX, CX ; *BX ← BX + CX*
ADC BX, CX ; *BX ← BX + CX + Carry Flag*

2) **SUB/SBB destination, source**
It is similar to ADD/ADC except that it does subtraction.

https://microcontrollerslab.com/8086-integer-arithmetic-instructions-assembly-language-programming/

8086 Integer Subtraction Instructions

8086 microprocessor supports the following subtraction Instructions:

1. SUB
2. SBB
3. DEC
4. AAS
5. DAS

Subtraction Instruction without Carry

Subtraction instruction takes two operands. Subtract the data in the source operand from the data of destination operand and then store the result back to

the destination operand. Just like ADD instruction, both operands should be either in bytes or words. If one operand is in a byte and the other in words then insert two zero's at the start of bytes. But they should be of the same type. The subtraction instruction supports the following operands:

Source	Destination	Example
Register	Register	SUB CH, AL
Register	Immediate	SUB AX, 16H
Register	Memory Address	SUB [SI], AX SUB 19[BP][DI], CX
Memory Address	Register	SUB AL, 10[CX]

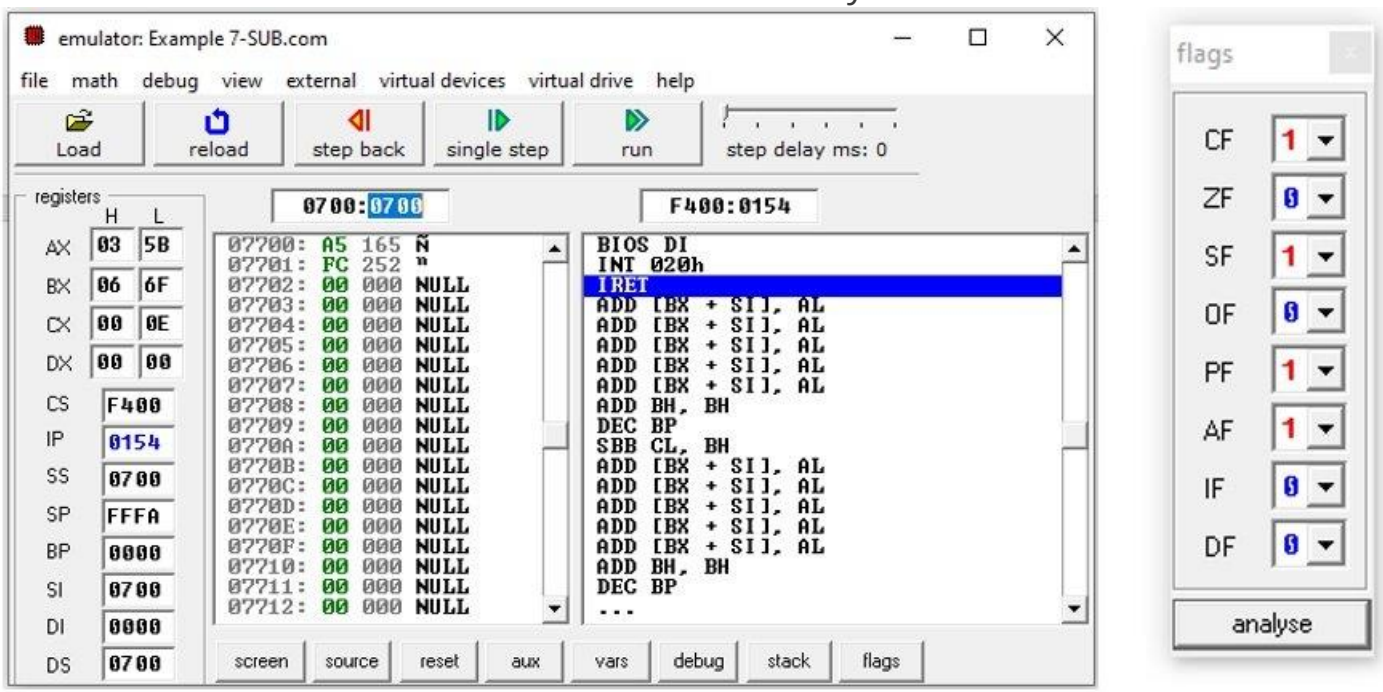
Example Assembly Code

```
ORG 100h
MOV AX, 2506     ;Sets AX to 2506
MOV BX, 1647     ;Sets BX to 1647
SUB AX, BX       ;AX=AX-BX
SUB [SI], AX     ;DS:SI=DS:SI-AX
RET              ;Stop the program
```

Output

The SUB AX, BX instruction subtracts BX from AX and store the difference in AX which is:

AX = 9CA – 66F = 35B. In the next instruction, SI is loaded with 0700H. The 6th instruction SUB [SI], AX subtracts AX from the data stored at memory location DS:SI and stores the difference at the same memory address.



The SUB instruction can affect AF, CF, OF, PF, SF and ZE flags depending upon the result obtained from difference.

Q18) What is DEC instruction?

Ans: BA 45

3) INC destination
Adds "1" to the specified **destination**.
Destination: Register, Memory Location
Note: Carry Flag is NOT affected.

```
Eg: INC AX                  ; AX ← AX + 1
    INC BL                  ; BL ← BL + 1
    INC BYTE PTR [BX]       ; Increment the byte pointed by BX in the Data Segment
                            ; i.e. DS:[BX] ← DS;[BX] + 1
    INC WORD PTR [BX]       ; Increment word pointed by BX in the Data Segment
                            ; {DS:[BX], DS:[BX+1]} ← {DS:[BX], DS:[BX+1]}+1
```

4) DEC destination
It is similar to INC. Here also Carry Flag is NOT affected.

- **DEC** − Used to decrement the provided byte/word by 1.

The DEC instruction subtracts one from the destination operand,
while preserving the state of the CF flag.

- (To perform a decrement operation that does update the CF flag, use a SUB instruction with an **immediate operand** of **1**.)

The **DEC** instruction decrements the specified operand by 1. An original value of 00h underflows to 0FFh. No flags are affected by this instruction.

Example

```
DEC R7
```

Q19) Explain CMP instruction.

Ans: BA 46

10)CMP destination, source
This instruction **compares the source with the destination**.
The source and the destination must be of the same size.
Comparison is **done by internally SUBTRACTING** the **SOURCE form DESTINATION**.
The result of this subtraction is NOT stored anywhere, instead the Flag bits are affected.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
ALL condition flags are **updated**.

```
Eg: CMP BL, 55H         ; BL compared with 55H i.e. BL – 55H.
    CMP CX, BX          ; CX compared with BX i.e. CX – BX.
```

CMP Instruction

The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.

Syntax

```
CMP destination, source
```

CMP compares two numeric data fields. The destination operand could be either in register or in memory. The source operand could be a constant (immediate) data, register or memory.

Example

```
CMP DX,       00   ; Compare the DX value with zero
JE  L7       ; If yes, then jump to label L7
.
.
L7: ...
```

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

```
INC    EDX
CMP    EDX, 10       ; Compares whether the counter has reached 10
JLE    LP1      ; If it is less than or equal to 10, then jump to LP1
```

Q20) Explain OR instruction.

Ans: BA 49

https://microcontrollerslab.com/8086-logical-instructions-assembly-examples/

3) OR destination, source
This instruction **logically Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **OR BL, CL** ; *BL ← BL OR CL*

8086 OR Logical Instruction

It performs the OR operation between two operands and stores the result back into the destination operand. The destination operand can be a register or a memory location whereas the source can be immediate, register, or a memory location. But Keep in mind, both operands should not be a memory location. The OR instruction clears the CF and OF flags to 0 and update PF, ZF, and SF flags. The OR operation gives 1 at output if any one input is 1 and give 0 on output only when both inputs are 0.

Format : OR Destination, Source

OR Logic Example

Suppose two numbers 3527 and 2968. The OR operation between these two numbers give:

```
3527 :   1101 1100 0111
2968 :   1011 1001 1000
4063:    1111 1101 1111
```

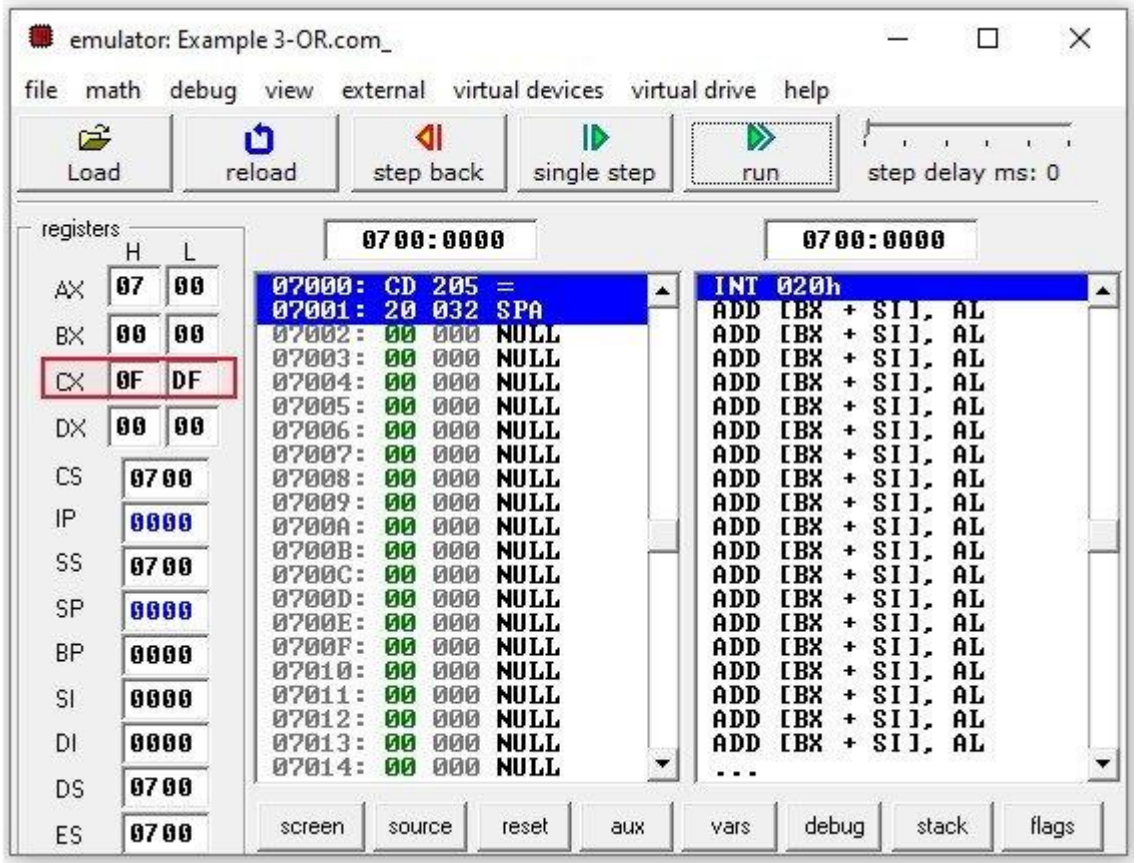
The hexadecimal value of 4063 is FDF.

OR Assembly Example

The code given performs the OR operation between contents of CX register and the data stored at offset address 102h.

```
ORG 100h
.MODEL SMALL
.DATA
VAR_1 DW 3527
.CODE
MOV AX, 0700h
MOV DS, AX
MOV CX, 2968
OR CX, [102h]
RET
```

Output:



Q21) Explain XOR instruction.

Ans: BA 49

4) XOR destination, source
This instruction **logically X-Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **XOR BL, CL** ; *BL ← BL XOR CL*

XOR Instruction 8086

This instruction performs the XOR operation between bits of source and destination operands. The XOR operation gives 1 when both inputs are different. When both inputs are same then the output will be zero. The source operand can be either a register or memory address whereas destination operand can be immediate, register or memory location.

Format: XOR Destination, Source

XOR Logic Example

Suppose AH is loaded with 68H. The XOR instruction performs the XOR operation between the contents of AH and the immediate value of 9CH.

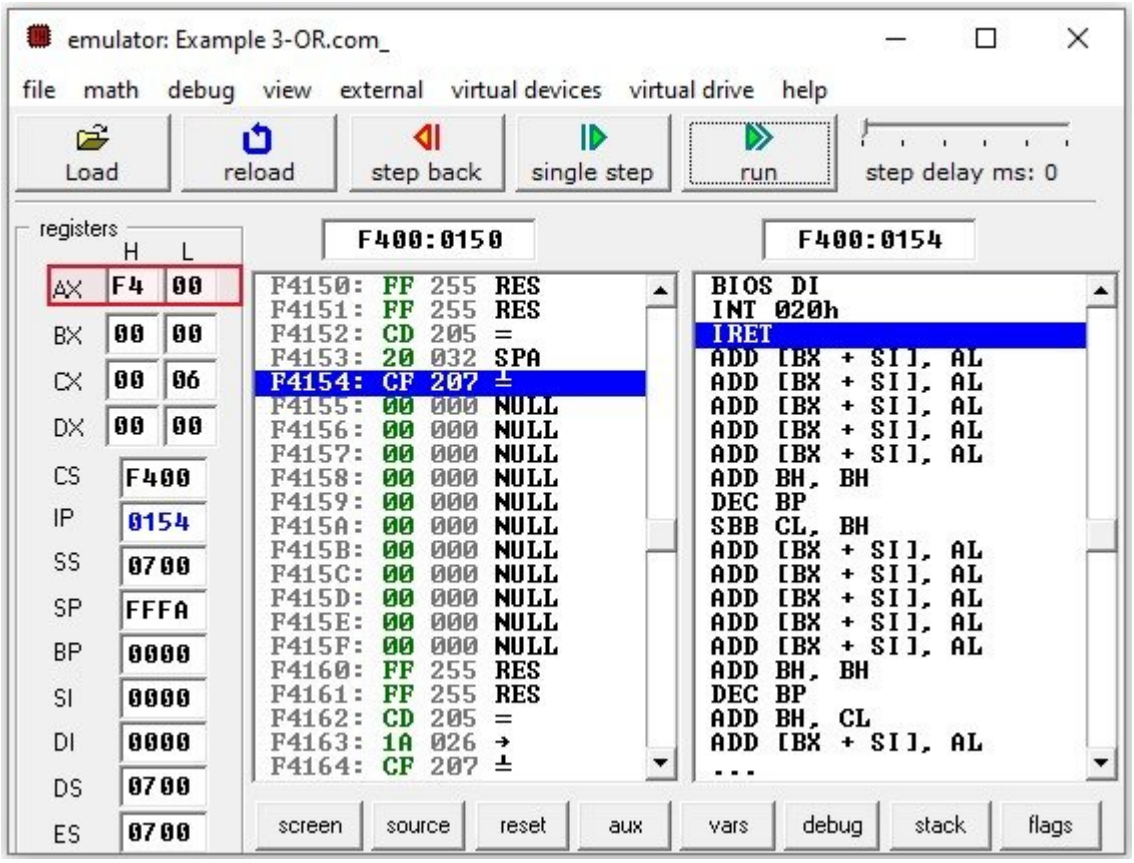
```
68H : 0110 1000

9CH : 1001 1100

F4H : 1111 0100
```

Assembly Example

```
ORG 100h
.MODEL SMALL
.CODE
MOV AH,68H
XOR AH,9CH
RET
```

Output:

The red box in the output indicates the result produced by XOR operation.



Q22) Explain Addressing Modes.

Ans: BA 34-36

https://www.tutorialspoint.com/microprocessor/microprocessor_8086_addressing_modes.htm

https://www.geeksforgeeks.org/addressing-modes-8086-microprocessor/

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 −

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
```

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
```

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]
```

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}
```

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]
```

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]
```

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]
```

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]
```

Q23) Write Syntax of Addressing Modes.

Ans: Q22

Q24) Give Example of addressing Modes.

Ans:Q22

Q25) What is meaning of “[ ]” bracket? (mov eax,[1234H])

Ans: [] to show that we need of pass the address of 1234H location and not 1234H

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]
```

Q26) Which are conditional instruction?

Ans: https://microcontrollerslab.com/8086-conditional-branch-instructions-assembly-examples/

https://www.geeksforgeeks.org/program-execution-transfer-instructions-8086-microprocessor/

Conditional Branch Instructions

On the other hand, the conditional branches are those instructions whose execution is based on some condition. It checks one or more flag conditions and transfers the control to a new memory location. There are two types of jumps namely Far and Near. In the far jumps, the program counter jumps to the memory location which lies outside the current code segment whereas in near jumps, the IP points to the memory address inside the current code segment and that is why the CS register remains unchanged in near jumps. The conditional jumps are also near jumps. The syntax of these instructions is:

```
Opcode LABEL
```

It is a 2-byte instruction consisting of 1-Byte Opcode and 1-byte Label. Label value is between 00 and FF. It is sign-extended to 16-bits and added to the contents of IP register. The target address must be within the -128 to +127 bytes of IP. The flag conditions are checked depending upon the instruction, if they are true the program control is transferred to the memory address pointed by the IP register. If the condition is not satisfied, then the program continues in a sequential manner.

List of 8086 Conditional Branch Instructions

Table below shows the mnemonics of all the conditional branches instructions.

Instructions	Operation	Testing Condition
JA/JNBE	Jump if Above/Not Below or Equal	C=0 and Z=0
JAE/JNB/JNC	Jump if Above or Equal/ Jump if Not Below/Jump if No Carry	C=0
JB/JNAE	Jump if Below/ Jump if Not Above or Equal	C=1

JBE/JNA	Jump if Below or Equal/ Jump if Not Above	(C or Z) =1
JC	Jump if Carry	C=1
JNC	Jump if Not Carry	C=0
JCXZ	Jump if the CX register=0	CX=0
JE/JZ	Jump if Equal/Jump if Zero	Z=1
JG/JNLE	Jump if Greater/Jump if Not Less Than or Equal	((S xor O) or Z) = 0
JGE/JNL	Jump if Greater or Equal/Jump if Not Less Than	(S xor O)=0
JL/JNGE	Jump if Less Than/Jump if Not Greater Than or Equal	(S xor O)=1
JLE/JNG	Jump if Less than or Equal/Jump if Not Greater	((S xor O) or Z) = 1
JNE/JNZ	Jump if Not Equal/ Jump if Not Zero	Z=0
JNP/JPO	Jump if Not Parity/Jump if Parity Odd	P=0
JNS	Jump if Not Signed/Jump if Positive	S=0
JO	Jump if Overflow	O=1
JNO	Jump if Not Overflow	O=0
JP/JPE	Jump if Parity/ Jump if Parity Even	P=1
JS	Jump if Signed/ Jump if Negative	S=1

These instructions are executed after some other instructions which affects the content of flag registers. Let's discuss these instructions in detail through examples.

Q27) Which are Unconditional instruction?

Ans: https://microcontrollerslab.com/8086-conditional-branch-instructions-assembly-examples/

https://www.geeksforgeeks.org/program-execution-transfer-instructions-8086-microprocessor/

Unconditional Branch Instructions

The unconditional branches are those in which the program counter jumps to the label address provided within the instruction.

For example

```
JMP NEXT
```

Program execution transfer instructions are similar to branching instructions and refer to the act of switching execution to a different instruction sequence as a result of executing a branch instruction.

The two types of program execution transfer instructions are:

1. Unconditional
2. Conditional

1. Unconditional Program Execution Transfer Instructions – These instruction always execute.

Opcode	Operand	Explanation	Example
CALL	address	calls a subroutine and saves the return address on the stack	CALL 2050
RET	none	returns from the subroutine to the main program	RET
JUMP	address	transfers the control of execution to the specified address	JUMP 2050
LOOP	address	loops through a sequence of instructions until CX=0	LOOP 2050

Here the address can be specified directly or indirectly.

Q28) Give example of Conditional JUMP instruction

Ans: Q26

Q29) Give example of Unconditional JUMP instruction

Ans: Q27

Q30) What is command line argument

Ans: Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the `main()` method.

Q31) Explain Rotate instruction.

Ans: BA 52,53

https://www.includehelp.com/embedded-system/shift-and-rotate-instructions-in-8086-microprocessor.aspx

https://www.eeeguide.com/rotate-instruction-in-8086-with-example/

https://care4you.in/logical-and-shift-and-rotate-instructions-8086/

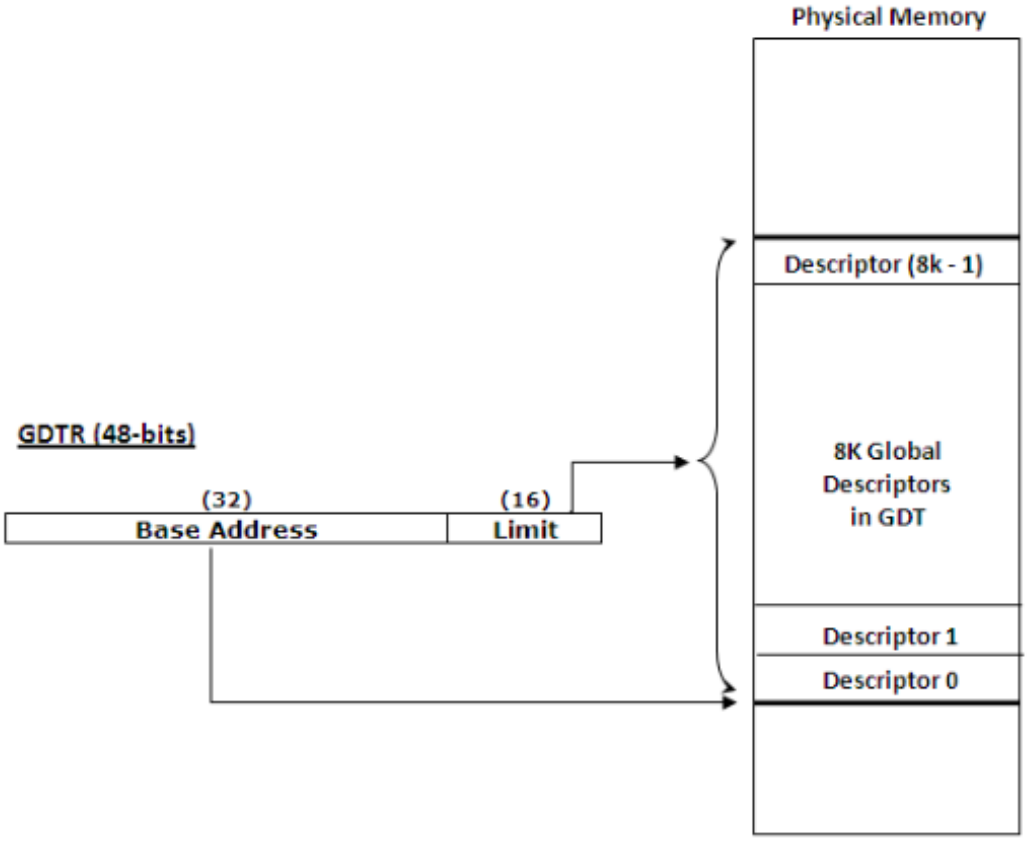
Q32) Give example of all Rotate instructions.

Ans: Q31

Q33) Explain GDTR,LDTR,TR,MSW registers

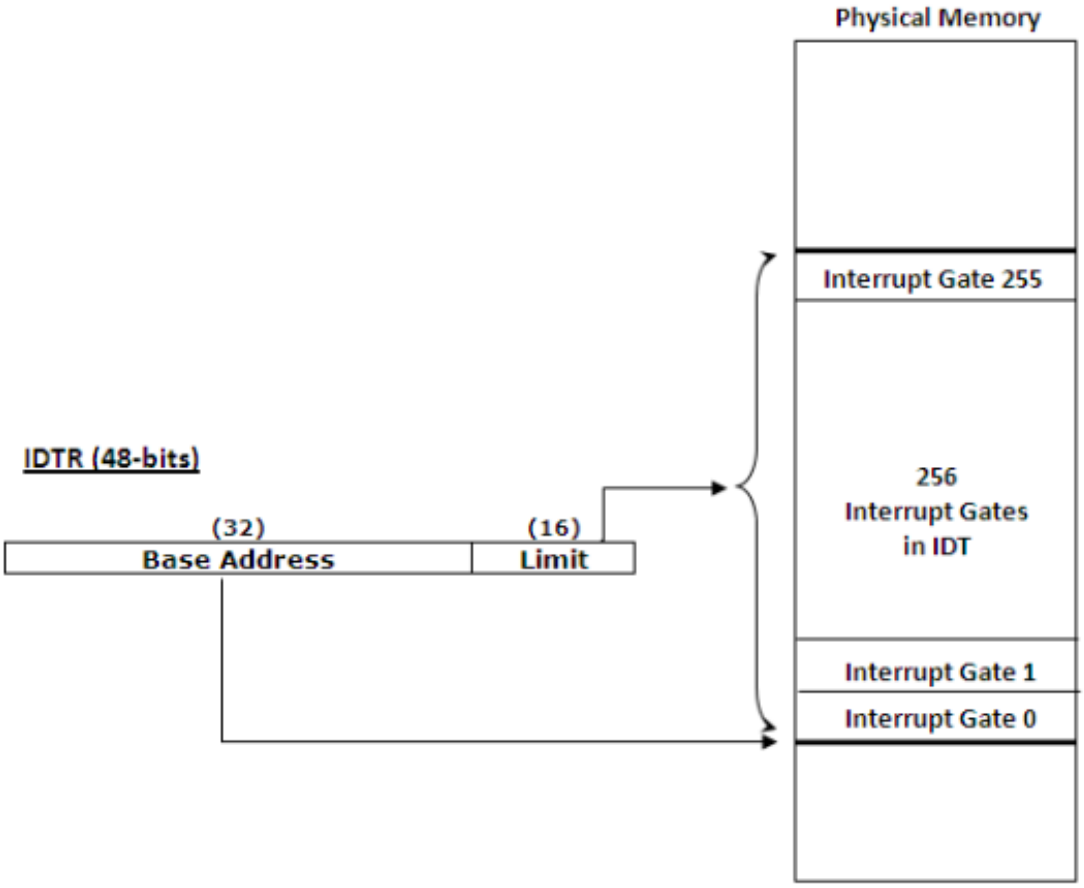
Ans:

GDTR (Global Descriptor Table Register)



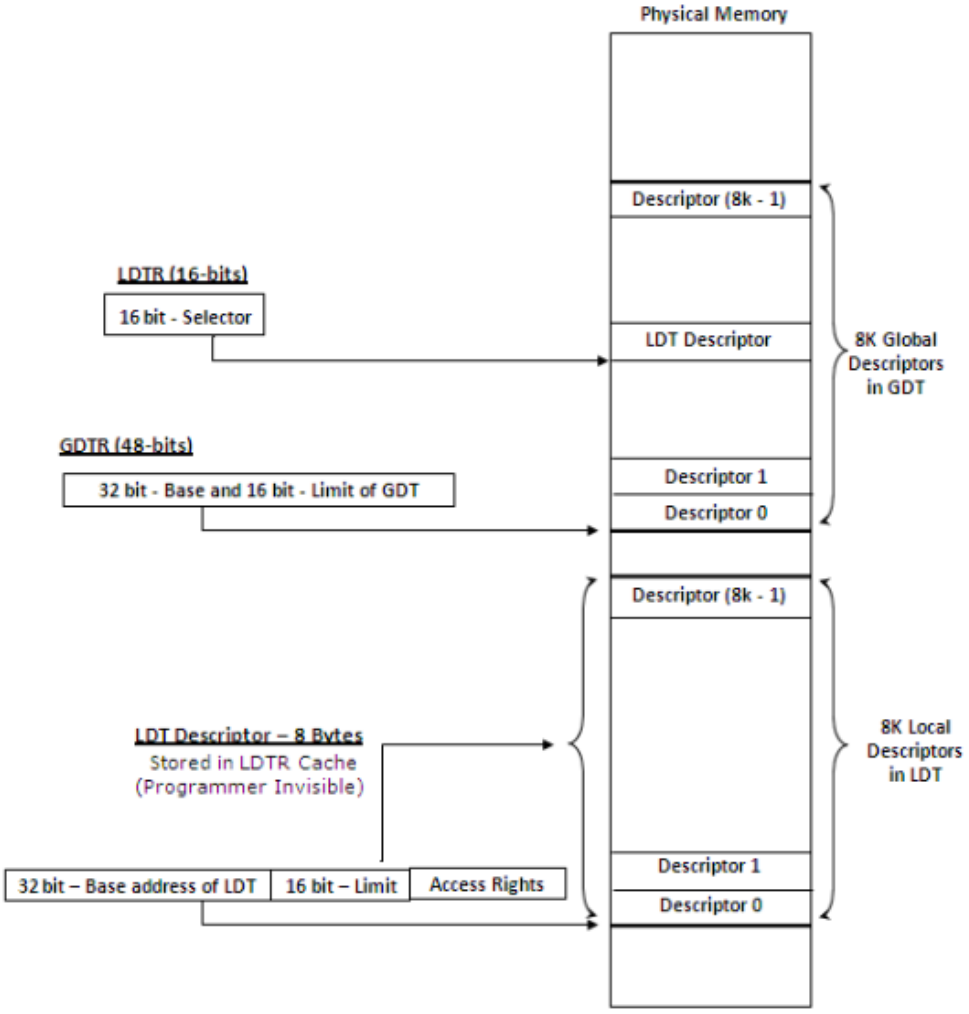
1) GDTR is a **48 bit register. It defines the GDT in the Physical Memory.**
2) The upper 4 bytes (32 bits) give the starting address of the GDT also called the base address.
3) The lower 2 bytes give the 16-bit Limit which decides the size of the GDT.
4) As the Limit field is 16-bits, the **max size of the GDT can be 64 KB.**
   The size of GDT is always Limit + 1.
   If Limit is FFFFH then size of GDT will be 65535+1=65536 i.e. 64KB.
5) **The GDT contains descriptors of Global Segments.**
6) Each descriptor is of 8 bytes.
   It gives the Starting Address, Limit and Access Rights of the segment.
7) **The GDT can contain max 64KB/8 = 8K descriptors (8192 Descriptors).**

IDTR (Interrupt Descriptor Table Register)



1) IDTR is a **48 bit register. It defines the IDT in the Physical Memory.**
2) The upper **4 bytes** (32 bits) give the **starting address** of the IDT also called the base address.
3) The lower **2 bytes** give the 16-bit **Limit** which decides the size of the IDT.
4) The IDT contains **Interrupt Descriptors**.
5) These Descriptors direct the µP towards the respective ISR whenever an interrupt occurs. Hence the Descriptors are also called **"Interrupt Gates"**.
6) Each Interrupt Descriptor is of **8 bytes**.
7) 80386 µP supports **256 Interrupts**.
8) Hence **max size of the IDT is 256 x 8 = 2KB**.
9) The size of IDT is Limit + 1.
10) Hence **max value of Limit must be 07FFH**.

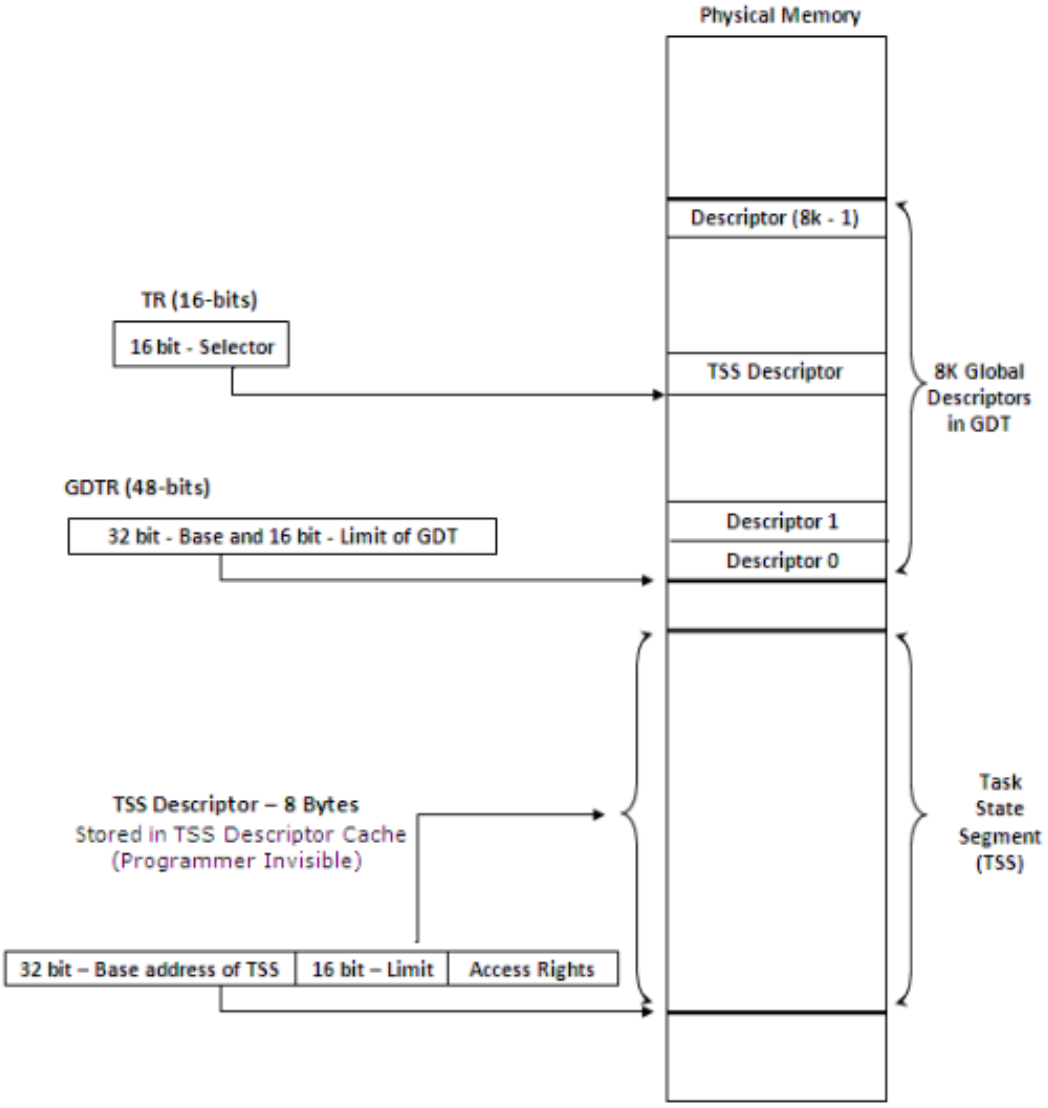
LDTR (LOCAL DESCRIPTOR TABLE REGISTER)



1) LDTR is a **16 bit register**.
2) **It contains a selector, which points to the LDT Descriptor in the GDT.**
3) Every task in 80386 µP has a local memory space. Here it has its local segments, which are only available to that particular task.
4) The **descriptors of all the local segments** are present in the LDT of that task.
5) There is an individual LDT for each task.
6) The **Descriptor for the LDT itself is present in the GDT**.
7) The LDT Descriptor is of 8 bytes.

8) It contains a 32 bit base address of the LDT, 16 bit Limit and Access rights information.
9) Once we load a new selector in the LDTR, the corresponding LDT Descriptor is fetched from the GDT and loaded into a LDT Descriptor Cache and thus it provides the address and limit of the LDT. #Please refer Bharat Sir's Lecture Notes for this ...
10) The structure of LDT is similar to that of GDT. It also has max 8k Descriptors each of size 8 Bytes and hence **max size of LDT is also 64KB**.

TR (TASK REGISTER)



1) TR is a **16 bit register**.
2) **It contains a selector, which points to the TSS Descriptor in the GDT.**
3) Every task in 80386 µP has a TSS (Task State Segment).
4) The TSS contains all the desired information required to start the task such as initial values of all user accessible registers, the entire I/O bitmap, a back link to the previous task etc.
5) The **Descriptors for the TSS of each Task are present in the GDT**.
6) As soon as we load a selector in Task Switch register, the corresponding TSS Descriptor is copied from the GDT into an on chip TSS Descriptor cache.
7) The TSS Descriptor is of 8 bytes.
8) It contains a 32 bit base address of the TSS, 16 bit Limit and Access rights information.
9) As the Limit field is of 16-bits, the max size of TSS can be 64 KB.

Q34) Explain features of 80386

Ans: https://electronicsdesk.com/80386-microprocessor.html

Features of 80386

- As it is a 32-bit microprocessor. Thus has 32-bit ALU.
- 80386 has data bus of 32-bit.
- It holds address bus of 32 bit.
- It supports physical memory addressability of 4 GB and virtual memory addressability of 64 TB.
- 80386 supports variety of operating clock frequency, which are 16 MHz, 20 MHz, 25 MHz and 33 MHz.
- It offers 3 stage pipeline: fetch, decode and execute. As it supports simultaneous fetching, decoding and execution inside the system.

Salient Features of 80386:

1) **Address Bus:**
80386 has a "32 bit" address bus.
This means it can access a total of 2^{32} = 4GB of physical memory.
The memory has an address range of 0000 0000H ... FFFF FFFFH.

Memory Address	Data
0000 0000 h	8-bit
0000 0001 h	8-bit
0000 0002 h	8-bit
0000 0003 h	8-bit
---	---
FFFF FFFF h	8-bit

Though the total address bus is of 32 bits, only the higher 30 bits from A_{31} – A_2 are released by the µP. The lower 2 lines A_1 and A_0 are used internally by the µP to produce the four bank-enable signals $\overline{BE_3}$ ... $\overline{BE_0}$. #Please refer Bharat Sir's Lecture Notes for this ...

2) **Data Bus:**
80386 has a "32-bit" data bus. This means 80386 can transfer 32-bit data at a time.
It also has a 32-bit ALU, which means 80386 can operate on 32-bit numbers in one cycle.
Hence 80386 is called a "32-bit µP".

32-bit data is stored in 4 consecutive locations.
To transfer 32-bit data in one operation 80386 memory is divided into 4 banks of 1 GB each. The banks are enabled by 4 bank-enable signals: $\overline{BE_3}$ ... $\overline{BE_0}$ produced by the µP.

3) **Address Pipelining:**
80386 performs address pipelining, by putting address of the next machine cycle on the address bus, during T2 state of the current machine cycle. This makes the decoder delay transparent and is especially useful for interfacing slower devices as it reduces the number of wait states.

4) **Virtual Memory**:
80386 supports Virtual Memory which is implemented using Segmentation and Paging.
It can access a total Virtual Memory of 64 TB (2^{46}).

5) **Protection:**
80386 uses a protected model for accessing both memory and I/O. It uses 4 Privilege Levels.

6) **Multitasking:**
80386 allows multitasking using timesharing. Here several tasks can execute simultaneously by taking a small time slice of the µP. this gives higher system performance.

7) **I/O Addressing:**
80386 uses a 16-bit I/O address and hence can access up to 2^{16} i.e. 65536 I/O devices with address 0000 h ... FFFF h.

Salient Features of 80386

1) It is a **32 bit Microprocessor**.
2) It has a **32 bit data bus**.
3) It has **4 memory banks**.
4) It has a **32 bit address bus**.
5) It can access **4 GB memory**.
6) It has **3 Pipeline stages**.
7) The Pipeline stages are called: **Fetch, Decode, Execute**.
8) It operates on **16 MHz – 33 MHz** frequency.
9) It has **2,75,000 transistors**.
10) It was released in the year **1985**.

Q35) Explain flag registers of 8086 and 80386

Ans:

https://www.byclb.com/TR/Tutorials/microprocessors/ch2_1.htm

EFLAGS

EFLAGS indicate the condition of the microprocessor and control its operation. Figure 2-2 shows the flag registers of all versions of the microprocessor. Note that the flags are upward-compatible from the 8086/8088 to the Pentium II microprocessor. The 8086-80286 contain a FLAG register (16 bits) and the 80386 and above contain an EFLAG register (32-bit extended flag register).

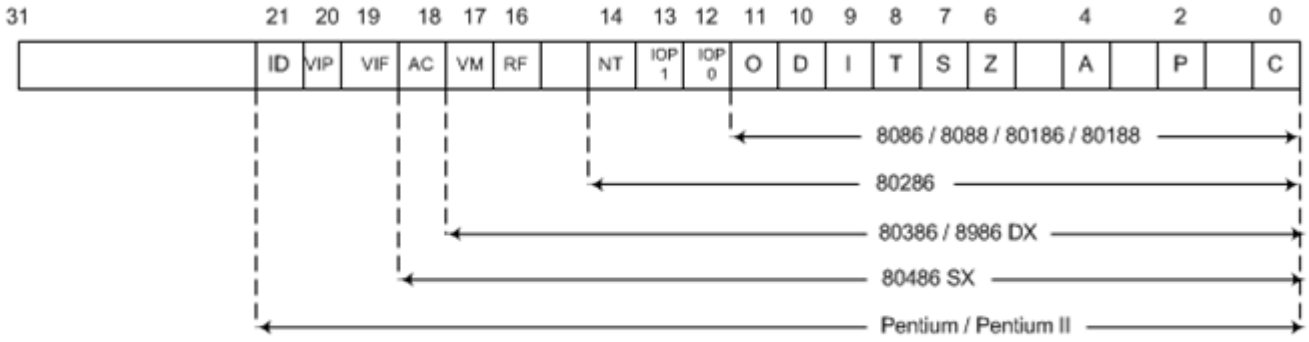


Figure 2-2 : The EFLAGS register.

The rightmost five flag bits and the overflow flag change after many arithmetic and logic instructions execute. The flags never change for any data transfer or program control operation. Some of the flags are also used to control features found in the microprocessor. Following is a list of each flag bit, with a brief description of their function.

C (carry)

Carry holds the carry after addition or the borrow after subtraction. The carry flag also indicates error conditions, as dictated by some programs and procedures. This is especially true of the DOS function calls.

P (parity)

Parity is a logic 0 for odd parity and a logic 1 for even parity. Parity is a count of ones in a number expressed as even or odd.

If a number contains zero one bits, it has even parity. The parity flag finds little application in modern programming and was implemented in early Intel microprocessors for checking data in data communications environments. Today parity checking is often accomplished by the data communications equipment instead of the microprocessor.

A (auxiliary carry)

The auxiliary carry holds the carry (half-carry) after addition or the borrow after subtraction between bits positions 3 and 4 of the result. This highly specialized flag bit is tested by the DAA and DAS instructions to adjust the value of AL after a BCD addition or subtraction. Otherwise, the A flag bit is not used by the microprocessor or any other instructions.

Z (zero)

The zero flag shows that the result of an arithmetic or logic operation is zero. If Z=1, the result is zero; if Z= 0, the result is not zero.

S (sign)

The sign flag holds the arithmetic sign of the result after an arithmetic or logic instruction executes. If S=1, the sign bit (leftmost hit of a number) is set or negative; if S=0, the sign bit is cleared or positive.

T (trap)

The trap flag enables trapping through an on-chip debugging feature. (A program is debugged to find an error or bug.) If the T flag is enabled (1), the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control registers. lf the T flag is a logic 0, the trapping (debugging) feature is disabled.

I (interrupt)

The interrupt flag controls the operation of the INTR (interrupt request) input pin. If I=1. the INTR pin is enabled: if I=0, the INTR pin is disabled. The state of the I flag bit is controlled by the STI (set I flag) and CLI (clear I flag) instructions.

D (direction)

The direction flag selects either the increment or decrement mode for the Dl and/or SI registers during string instructions. If D=1, the registers are automatically decremented: if D=1, the registers are automatically incremented. The D flag is set with the STD (set direction) and cleared with the CLD (clear direction) instructions.

0 (overflow)

Overflows occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine. For unsigned operations, the overflow flag is ignored.

IOPL (I/0 privilege level)

IOPL 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, I/O executes without hindrance. If the IOPL is lower than the current privilege level, an interrupt occurs, causing execution to suspend. Note that an IOPL of 00 is the highest or most trusted: if IOPL is 11, it is the lowest or least trusted.

NT (nested task)

The nested task flag indicates that the current task is nested within another task in protected mode operation. This line is set when the task is nested by software.

RF (resume)

The resume flag is used with debugging to control the resumption of execution after the next instruction.

VM (virtual mode)

The VM flag bit 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 system. Essentially, this allows the system program to execute multiple DOS programs.

AC (alignment check)

The alignment check flag bit activates if a word or douhleword is addressed on a non-word or non-douhleword boundary. Only the 80486SX microprocessor contains the alignment check hit that is primarily used by its companion numeric coprocessor, the 80487SX, for synchronization.

VIF (virtual interrupt flag)

The VIF is a copy of the interrupt flag bit available to the Pentium-Pentium II microprocessors.

VIP (virtual interrupt pending)

VIP provides information about a virtual mode interrupt for the Pentium—Pentium II microprocessors. This is used in multitasking environments to provide the operating system with virtual interrupt flags and interrupt pending information.

ID (identification)

The ID flag indicates that the Pentium—Pentium II microprocessors support the CPUID instruction. The CPUID instruction provides the system with information about the Pentium microprocessor, such as its version number and manufacturer.

Q36) Explain memory size of 8086 and 80386

Ans:

8086

Memory − 8085 can access up to 64Kb, whereas 8086 can access up to 1 Mb of memory.

https://www.geeksforgeeks.org/memory-segmentation-8086-microprocessor/

80386

The physical memory of an 80386 system is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address that ranges from zero to a maximum of 2^(32) -1 (4 gigabytes).

80386 programs, however, are independent of the physical address space. This means that programs can be written without knowledge of how much physical memory is available and without knowledge of exactly where in physical memory the instructions and data are located.

The model of memory organization seen by applications programmers is determined by systems-software designers. The architecture of the 80386 gives designers the freedom to choose a model for each task. The model of memory organization can range between the following extremes:

- A "flat" address space consisting of a single array of up to 4 gigabytes.
- A segmented address space consisting of a collection of up to 16,383 linear address spaces of up to 4 gigabytes each.

Both models can provide memory protection. Different tasks may employ different models of memory organization. The criteria that designers use to determine a memory organization model and the means that systems programmers use to implement that model are covered in Part -- Programming.

https://css.csail.mit.edu/6.858/2014/readings/i386/s02_01.htm

Q37) What is Protected Mode?

Ans: https://www.viralpatel.net/taj/tutorial/protectedmode.php

The processing mode of the 80386 also determines the features that are accessible. The 80386 has three processing modes:

1. Protected Mode.
2. Real-Address Mode.
3. Virtual 8086 Mode.

Protected mode is the natural 32-bit environment of the 80386 processor. In this mode all instructions and features are available.

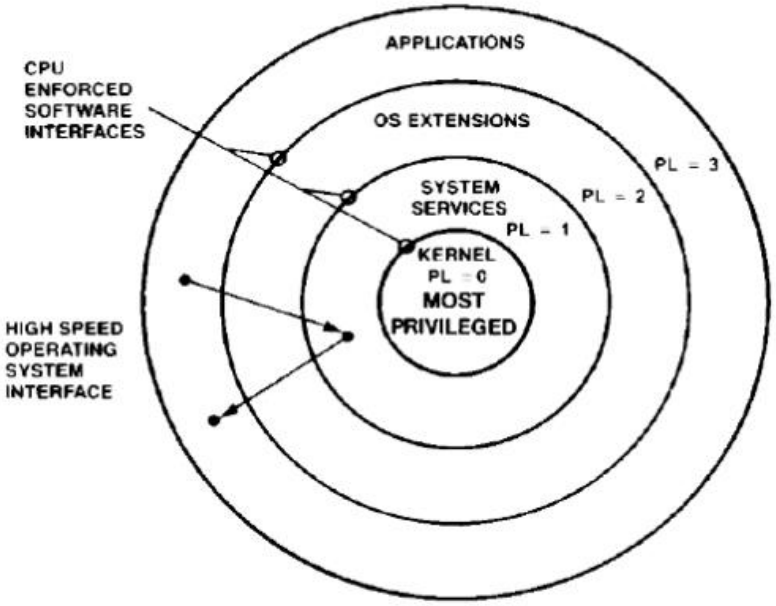
Real-address mode (often called just "real mode") is the mode of the processor immediately after RESET. In real mode the 80386 appears to programmers as a fast 8086 with some new instructions. Most applications of the 80386 will use real mode for initialization only.

Virtual 8086 mode (also called V86 mode) is a dynamic mode in the sense that the processor can switch repeatedly and rapidly between V86 mode and protected mode. The CPU enters V86 mode from protected mode to execute an 8086 program, then leaves V86 mode and enters protected mode to continue executing a native 80386 program.

The features that are available to applications programs in protected mode and to all programs in V86 mode are the same. These features form the content of Part I. The additional features that are available to systems software in protected mode form Part II. Part III explains real-address mode and V86 mode, as well as how to execute a mix of 32-bit and 16-bit programs.

Protected Mode of 80386

80386 µP provides a very advanced mode of operations called the Protected Mode.
In Protected Mode, 80386 µP provides dedicated hardware to prevent user programs from affecting other user programs and also safeguards the Operating System from being affected by user programs.
There are Four Privilege Levels, assigned to programs and data to define their privileges.



Level 0: **This level is assigned to the Operating System Kernel (Main part of the Operating System).**
It is the most privileged level.
Any program at this level can access all the data at any Privilege Level, whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0.

Level 1: **This level is assigned to the System Services such as File Handling, Device Drivers.**
It is the 2nd most privileged level. #Please refer Bharat Sir's Lecture Notes for this ...
Any program at this level can access the data at any Privilege Level which is lower than this level (numerically higher), whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0 or Privilege Level 1.

Level 2: **This level is assigned to the Custom Extensions of the OS.**
It is the 3rd most privileged level.
Any program at this level can access the data at any Privilege Level which is lower than this level (numerically higher), whereas a data at this Privilege Level can only be accessed by a program at Privilege Level 0, 1 or 2.

Level 3: **This level is assigned to all the User Application and Programs.**
It is the least privileged level.
Any program at this level can normally access the data at Privilege Level 0, whereas a data at this Privilege Level can only be accessed by a program at any Privilege Level 0...3.

Protected Mode: In computing, protected mode, also called protected virtual address mode is an operational mode of x86-compatible central processing units (CPUs). It allows system software to use features such as virtual memory, paging and safe multi-tasking designed to increase an operating system's control over application software.

When a processor that supports x86 protected mode is powered on, it begins executing instructions in real mode, in order to maintain backward compatibility with earlier x86 processors. Protected mode may only be entered after the system software sets up several descriptor tables and enables the Protection Enable (PE) bit in the control register 0 (CR0).

Q38) What is paging mechanism?

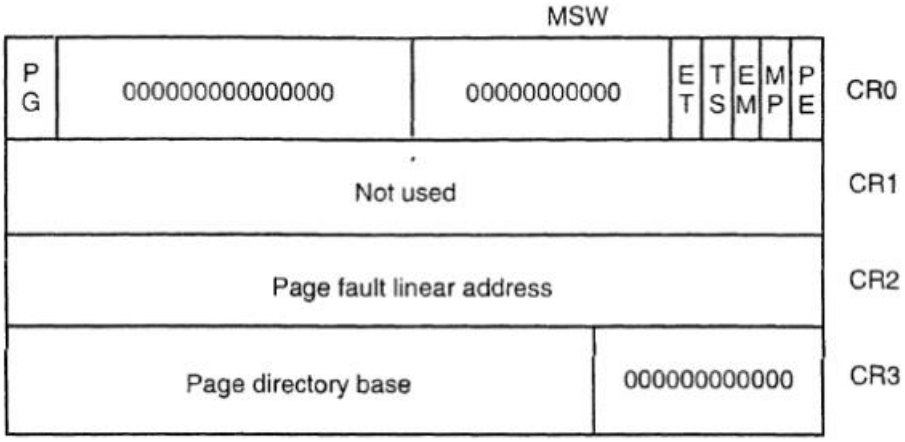
Ans: BA

1. Paging: •Paging Operation: Paging is one of the memory management techniques used for virtual memory multitasking operating system. •The segmentation scheme may divide the physical memory into a variable size segments but the paging divides the memory into a fixed size pages. •The segments are supposed to be the logical segments of the program, but the pages do not have any logical relation with the program. •The pages are just fixed size portions of the program module or data.
2. 26. •The advantage of paging scheme is that the complete segment of a task need not be in the physical memory at any time. •Only a few pages of the segments, which are required currently for the execution need to be available in the physical memory. Thus the memory requirement of the task is substantially reduced, relinquishing the available memory for other tasks. •Whenever the other pages of task are required for execution, they may be fetched from the secondary storage. •The previous page which are executed, need not be available in the memory, and hence the space occupied by them may be relinquished for other tasks.
3. 27. •Thus paging mechanism provides an effective technique to manage the physical memory for multitasking systems. •Paging Unit: The paging unit of 80386 uses a two level table mechanism to convert a linear address provided by segmentation unit into physical addresses. The paging unit converts the complete map of a task into pages, each of size 4K. The task is further handled in terms of its page, rather than segments. The paging unit handles every task in terms of three components namely page directory, page tables and page itself.
4. 28. •Paging Descriptor Base Register: The control register CR2 is used to store the 32-bit linear address at which the previous page fault was detected. The CR3 is used as page directory physical base address register, to store the physical starting address of the page directory. The lower 12 bit of the CR3 are always zero to ensure the page size aligned directory. A move operation to CR3 automatically loads the page table entry caches and a task switch operation, to load CR0 suitably.
5. 29. •Page Directory : This is at the most 4Kbytes in size. Each directory entry is of 4 bytes,thus a total of 1024 entries are allowed in a directory.The upper 10 bits of the linear address are used as an index to the corresponding page directory entry. The page directory entries point to page tables. •Page Tables: Each page table is of 4Kbytes in size and many contain a maximum of 1024 entries. The page table entries contain the starting address of the page and the statistical information about the page. •The upper 20 bit page frame address is combined with the lower 12 bit of the linear address. The address bits A12- A21 are used to select the 1024 page table entries. The page table can be shared between the tasks.
6. 30. •The P bit of the above entries indicate, if the entry can be used in address translation. •If P=1, the entry can be used in address translation, otherwise it cannot be used. •The P bit of the currently executed page is always high. •The accessed bit A is set by 80386 before any access to the page. If A=1, the page is accessed, else unaccessed. •The D bit (Dirty bit) is set before a write operation to the page is carried out. The D-bit is undefined for page director entries. •The OS reserved bits are defined by the operating system software.
7. 31. •The User / Supervisor (U/S) bit and read/write bit are used to provide protection. These bits are decoded to provide protection under the 4 level protection model. •The level 0 is supposed to have the highest privilege, while the level 3 is supposed to have the least privilege. •This protection provide by the paging unit is transparent to the segmentation unit.
8. 32. PAGE TABLE ADDRESS 31..12 OS RESERVED 0 0 D A 0 0 U S R W P 31 12 11 10 9 8 7 6 5 4 3 2 1 0 Page Directory Entry PAGE FRAME ADDRESS 31..12 OS RESEV ED 0 0 D A 0 0 U S R W P 31 12 11 10 9 8 7 6 5 4 3 2 1 0 Page Table Entry

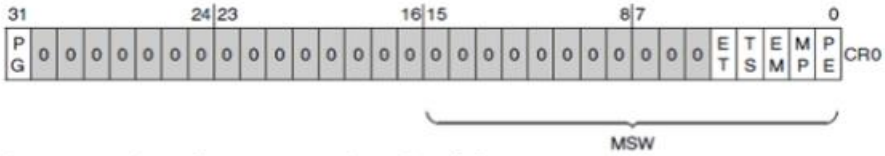
Q39) Explain control registers.

Ans:
https://pdos.csail.mit.edu/6.828/2008/readings/i386/s04_01.htm#:~:text=Four%20registers%20of%20the%2080386,LDTR%20Local%20Descriptor%20Table%20Register

A **control register** is a processor **register** which changes or **controls** the general behavior of a CPU or other digital device. Common tasks performed by **control registers** include interrupt **control**, switching the addressing mode, paging **control**, and coprocessor **control**.



CONTROL REGISTER 0:



The various bits of CR0 are explained as follows:

PG: Paging Enable.
This bit is made "1" to enable paging mode.
80386 µP implements Virtual Memory using the techniques of segmentation and paging. Though segmentation is compulsory, paging is optional. Paging is enabled using the PG bit of CR0. The default value of PG bit is "0".

ET: Extension Type.
This bit is used to indicate the type of Math Co-Processor used with 80386.
If ET = 1, then 80387 Math Co-Processor is used.
If ET = 0, then 80287 Math Co-Processor is used.
Note: This bit was reserved (R) in the initial versions of 80386 and was introduced later once 80387 was developed.

TS: Task Switched.
This bit is made "1" to indicate if a Task Switch is performed.
80386 µP implements "Multitasking", and thus it switches between various tasks, giving the programmer the impression that all tasks are running concurrently.
This significantly improves the overall system performance.
If TS = 1, it means a task switch is performed. Now the TSS of the current task has a back-link to the previous task.
Note: If TS bit = 1, then a type 7 interrupt called Math Co-Processor not available is caused, whenever a coprocessor instruction is encountered.

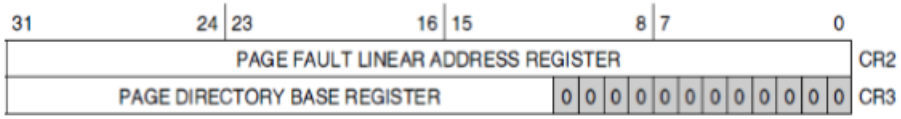
EM: Emulate Coprocessor.
This bit is made "1" in the absence of a Math Co-Processor so that if a coprocessor instruction is encountered, then it will be executed by an on chip emulator. If this bit is 0, then the coprocessor instruction will be executed by 80387/80287 whichever is present in the system.

MP: Math Co-processor Present.
This bit is made "1" to indicate that a Math Co-Processor like 80387 or 80287 is present.
Note: Out of EM bit and MP bit, only one of them must be "1".

PE: Protection Enable.
This bit is made "1" to enter protected mode.
On reset, by default this bit is "0". It is the only bit of CR0 which is also available in Real Mode.

CONTROL REGISTER 1: **NOT USED (RESERVED BY INTEL)**

CONTROL REGISTERS 2 & 3: **(USED ONLY FOR PAGING)**



CONTROL REGISTER 2: **(PAGE FAULT LINEAR ADDRESS)**

This is a 32-bit register giving the Linear Address that caused the last Page fault. A page Fault occurs when the desired page is not present in the Physical Memory.

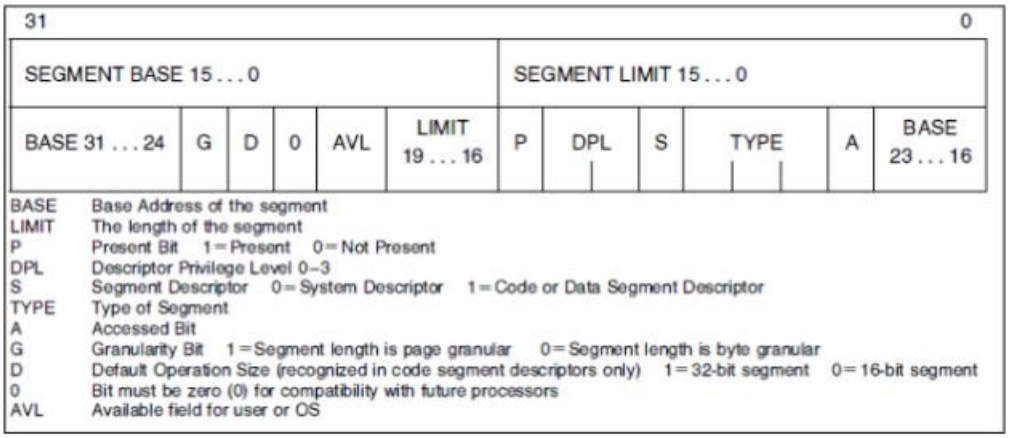
CONTROL REGISTER 3: **(PAGE DIRECTORY BASE REGISTER - PDBR)**

80386 μP implements 2-level Page translation mechanism (explained in later sections). Information about the various pages is stored in various page tables. Addresses of these page tables are stored in the page directory. CR3 gives the base address (starting address) of the Page Directory. Note that PDBR is only 20 bits. It just gives the upper 20-bits of the starting address of the page directory. That's because the page directory is of 4KB and is stored at a 4KB aligned location, so the last 12 bits of the starting address are assumed to be "0".

Q40) Explain descriptors, selector.

Ans:

Basic structure of a segment descriptor



Selectors and descriptors

Descriptors are a bit like real mode segments; they describe (as the name implies) a memory area in protected mode. A descriptor contains information about segment length, its base address and the attributes of it (i.e. type, access rights, ...). These descriptors are stored internally in a so-called descriptor table, which is basically an array of such descriptors. Selectors are roughly an index into this table. Because these 'segments' can be up to 4 GB in size, 32 bits aren't sufficient anymore to describe a single memory location like in real mode. 48 bits are now needed to do this, a 32 bit address and a 16 bit sized selector. The GO32 unit provides the tseginfo record to store such a pointer. But due to the fact that most of the time data is stored and accessed in the %ds selector, FPC assumes that all pointers point to a memory location of this selector. So a single pointer is still only 32 bits in size. This value represents the offset from the data segment base address to this memory location.

https://css.csail.mit.edu/6.858/2014/readings/i386/s05_01.htm

https://tldp.org/LDP/khg/HyperNews/get/memory/80386mm.html

Q41) Explain data bus size and address bus size of 8086

Ans:

IMPORTANT FEATURES OF 8086:

1) **Buses:**

Address Bus: 8086 has a **20-bit address bus**, hence it can access 2^{20} Byte memory i.e. **1MB**. The **address range** for this memory is **00000H ... FFFFFH**.

Data Bus: 8086 has a **16-bit data bus** i.e. it can access 16 bit data in one operation. Its ALU and internal data registers are also 16-bit.
Hence 8086 is called as a **16-bit µP**.

Control Bus: The control bus carries the signals responsible for performing various operations such as $\overline{\textbf{RD}}$, $\overline{\textbf{WR}}$ etc.

Q42) Explain data bus size and address bus size of 80386

Ans:

Salient Features of 80386:

1) **Address Bus:**
80386 has a "32 bit" address bus.
This means it can access a total of 2^{32} = 4GB of physical memory.
The memory has an address range of 0000 0000H ... FFFF FFFFH.

Memory Address	Data
0000 0000 h	8-bit
0000 0001 h	8-bit
0000 0002 h	8-bit
0000 0003 h	8-bit
---	---
FFFF FFFF h	8-bit

Though the total address bus is of 32 bits, only the higher 30 bits from A_{31} – A_2 are released by the µP. The lower 2 lines A_1 and A_0 are used internally by the µP to produce the four bank-enable signals $\overline{BE_3}$ ... $\overline{BE_0}$. #Please refer Bharat Sir's Lecture Notes for this ...

2) **Data Bus:**
80386 has a "32-bit" data bus. This means 80386 can transfer 32-bit data at a time.
It also has a 32-bit ALU, which means 80386 can operate on 32-bit numbers in one cycle.
Hence 80386 is called a "32-bit µP".

32-bit data is stored in 4 consecutive locations.
To transfer 32-bit data in one operation 80386 memory is divided into 4 banks of 1 GB each. The banks are enabled by 4 bank-enable signals: $\overline{BE_3}$ ... $\overline{BE_0}$ produced by the µP.

	80386 DX	80386 SX
1	80386 DX has a 32 bit data bus.	80386 SX has a 16 bit data bus.
2	Due to 32 bit data bus, the execution speed is higher. Hence the name: DX – Double Execution speed.	Due to 16 bit data bus, the execution speed is lower. Hence the name: SX – Single Execution speed.
3	32-bit transfers require 4 Memory Banks.	16-bit transfers require 2 Memory Banks.
4	Has 4 Bank enable signals: $\overline{BE_3}$, $\overline{BE_2}$, $\overline{BE_1}$, $\overline{BE_0}$.	Has only 2 Bank enable signals: $\overline{BHE}$ And $\overline{BLE}$.
5	4 Bytes are fetched at once in the pipelining queue.	2 Bytes are fetched at a time in the pipelining queue.
6	Has dynamic data bus sizing of 16-bit and 32-bit data bus, using $\overline{BS16}$ signal.	No such option available as the data bus is only of 16-bits. Hence $\overline{BS16}$ signal not useful.
7	Used for high performance.	Used for low cost memory and I/O system design.
8	Comes in a 132-pin ceramic PGA (Pin Grid Array) package for higher performance.	Comes in 100 lead plastic quad flat packages (PQFP) to permit lower cost.

Q43) Which are assembly language directives? (DB, DW, DD, DQ and DT)

Ans:

Assembly language has 2 types of statements:

1. **Executable**: Instructions that are translated into Machine Code by the assembler.

2. **Assembler Directives**:

Statements that direct the assembler to do some special task.
No M/C language code is produced for these statements.
Their main task is to inform the assembler about the start/end of a segment, procedure or program, to reserve appropriate space for data storage etc.
Some of the assembler directives are listed below

```
1. DB (Define Byte)          ; Used to define a Byte type variable.
Eg: SUM DB 0                 ; Assembler reserves 1 Byte of memory for the variable
                                   ; named SUM and initialize it to 0.

2. DW (Define Word)          ; Used to define a Word type variable (2 Bytes).

3. DD (Double Word)          ; Used to define a Double Word type variable (4 Bytes).

4. DQ (Quad Word)                ; Used to define a Quad Word type variable (8 Bytes).

5. DT (Ten Bytes)                ; Used to define 10 Bytes to a variable (10 Bytes).
```

Data definition directives are used to define the program variables and allocate a specified amount of memory to them. They are of type BYTE, WORD, Double Word, Quad Word and Ten Byte and their size in bytes are 1,2,4,8 and 10 respectively. The data definition directives are DB, DW, DD, DQ, DT.

DB – [define byte]
The DB directive is used to define a byte–type variable or to set aside one or more storage locations of type byte in memory. It can be used to define single or multiple byte variables.

Ex 1) n DB 42H

tells the assembler to reserve 1 byte of memory for a variable named n and to put the value 42H in the that memory location, when the program is loaded into memory to be run.

Ex 2) num DB ?

The above statement informs the assembler to reserve one byte of memory for a variable named num. use of ? in data definition informs the assembler that the value is unknown and hence the variable num is not initialized.

Ex 3) grade DB 'A' or "A"

The above statement informs the assembler to reserve one byte of memory for a variable named grade and initializes with ASCII equivalent of a letter 'A'. The ASCII character should be enclosed within single or double quotes.

Ex 4) num DB 25,50,43,76,34

The above statement reserves 5 bytes of consecutive memory locations for the variable num and initializes them with 5 values.

Ex 5) info DB 'welcome'

The above statement defines a variable info and reserves 7 bytes of consecutive memory locations and initializes with the string 'welcome' during execution of program.

Ex 6) sname db 10 dup('-')

This statement defines a variable sname, reserves 10 bytes of consecutive memory locations and initializes with ASCII equivalent of character '-'.

Ex 7) sum db 25 dup(?)

This statement defines a variable and reserves 25 bytes of consecutive memory locations and are not initialized.

DW – [define word]

The DW directive is used to declare a variable of type word, or to reserve storage locations of type word in memory.

Ex:- MULTIPLIER DW 347AH
num dw 023ah
sum dw ?
items dw 03abh, 0abc4h, 0bbah, 0543ch, 04a4bh
res dw 20 dup(0)

DD – [define double word]

The DD directive is used to declare a variable of type double word.

DQ – **Define Quad word** :

The directive DQ is used to define a quad word (8 bytes) type variable.

DT – **Define Ten bytes** :

The directive DT is used to define a Ten bytes type variable.

Q44) How to convert protected mode to real mode? Which bit is set?

Ans:

PE: Protection Enable.
This bit is made "1" to enter protected mode.
On reset, by default this bit is "0". It is the only bit of CR0 which is also available in Real Mode.

Q45) Which control register is used for paging?

Ans: Q39 – IMP

CR0 – PG bit: Page Enable Bit(1-> Paging Enabled)

CR2, CR3 – Only used for Paging

1. Paging Unit: The paging unit of 80386 uses a two level table mechanism to convert a linear address provided by segmentation unit into physical addresses. The paging unit converts the complete map of a task into pages, each of size 4K. The task is further handled in terms of its page, rather than segments. The paging unit handles every task in terms of three components namely page directory, page tables and page itself.
2. 28. •Paging Descriptor Base Register: The control register CR2 is used to store the 32-bit linear address at which the previous page fault was detected. The CR3 is used as page directory physical base address register, to store the physical starting address of the page directory. The lower 12 bit of the CR3 are always zero to ensure the page size aligned directory. A move operation to CR3 automatically loads the page table entry caches and a task switch operation, to load CR0 suitably.

Q46) Explain size of GDTR and IDTR and LDT register.

Ans: Q33

Q47) Write “1 to 9” hexadecimal numbers into ASCII value.

Ans:

Q48) Write “A to F” hexadecimal numbers into ASCII value

Ans:

Q49) Why we convert HEX to ASCII?

Ans: The logic behind HEX to ASCII conversion is very simple. We are just checking whether the number is in range 0 – 9 or not. When the number is in that range, then the hexadecimal digit is numeric, and we are just simply adding 30H with it to get the ASCII value. When the number is not in range 0 – 9, then the number is range A – F, so for that case, we are converting the number to 41H onwards.

Q50) Why we convert ASCII to HEX?

Ans: The algorithm used in computer systems for the conversion of ASCII values firstly converts the character into its integer equivalent from the lookup table. This integer is called the ASCII value of the given character. This integer is then converted into the Hexadecimal value.

Q51) Which are disadvantage of 8086? And advantage of 80386

Ans:

No protection

2 vs 3 stage pipeline

Less Frequency and other Features

The biggest disadvantage of the 8086 microprocessor is its memory model. Intel set up the 8086 to use memory in segments, rather than using one big, flat address space. This made the 8086 much more difficult to program than it needed to be

The 80386 can deal with more memory space; it also can put data into memory and take it back out faster than the 80286 chip. For these reasons alone, an 80386 computer will be superior to a machine from the AT generation.

The **Segmentation** unit provides a 4 level protection mechanism for protecting and isolating the system code and data from those of the application program. **Paging** unit converts linear addresses into physical addresses. The control and attribute PLA checks the privileges at the page level.

Parameter	8086	80386	Pentium
Year of Introduction	1978	1985	`1993
Data Bus	16 bit	32 bit	128 bit
`Address Bus	20 bit	32 bit	64 bit
Physical memory	1 MB	4 GB	4 GB
Register size	16 bit	32 bit	32 bit
Voltage required	5 V	5 V	3.3V
Clock type	1x	2x	3x
Pipelining	Yes	Yes	Yes
Operating modes	1.,Maximum mode2. Minimum mode	1.,Real mode,2.,Protected mode,Virtual mode	1.,Protected mode,2. Real-Address mode

Q52) Explain in detail Data Transfer Instructions

Ans: BA 41,42

The **data transfer instructions** are used to transfer data from one location to another. This transfer of data can be either from register to register, register to memory or memory to register.

It is important to note here that the memory to memory transfer of data directly is not possible.

Following are some **instructions that are used for data transfer purpose**:

1. MOV
2. PUSH
3. POP
4. XCHG
5. LAHF
6. SAHF
7. IN
8. OUT
9. LDS
10. LES

1) MOV

This instruction simply copies the data from the source to the destination.

```
Syntax:   MOV destination , source
Example:  MOV AX, BX
```

2) PUSH

This instruction is used to push data into the stack.

```
Syntax:     PUSH source
Example:    PUSH CX
Working:    SP  <-  SP - 1
            [SP] <- CH
            SP <-  SP - 1
            [SP]  <-  CL
```

3) POP

This instruction is used to get the data from the stack.

```
Syntax:     POP destination
Example:    POP CX
Working:    CL<-  [SP]
            SP <- SP + 1
            CL  <-  [SP]
            SP <-  SP+ 1
```

4) XCHG

It exchanges the contents of the source and the destination.

```
Syntax:     XCHG destination, source
Example:    XCHG BL, AL
```

5) LAHF

It stands for 'Load AHfrom Flag register'. This instruction will, therefore, load the AH register with the content of lower byte of the flag register.

```
Syntax:     LAHF
Working:    AH <- lower byte of the flag register
```

6) SAHF

It stands for 'Store AH to Flag register'. This instruction stores the content of AH register to the lower byte of flag register.

```
Syntax:     SAHF
Working:    Lower Byte of flag register <- AH
```

7) IN

This instruction is used to transfer data from the input unit to accumulator.

```
Syntax:     IN accumulator, Port address
Working:    The content from the input unit whose address in mentioned
            in the instruction is transferred to the accumulator
            which is the AX register.
Example:    IN AX, 1326H
            IN AL, DX
```

8) OUT

This instruction is used to transfer data from accumulator to the output unit.

```
Syntax:     OUT Port address, accumulator
Working:    The content from the accumulator which is the AX registeris
            transferred tothe output unit whose address in mentioned in
            the instruction.
Example:    OUT 1326H, AL
            OUT DX, AX
```

9) LDS

This instruction will load the register that is defined in the instruction and the data segment (DS) from the source.

```
Syntax:     LDS destination, source
Example:    LDS BX , [SI]
Working:    BL <-  [SI]
            BH  <-  [SI + 1]
```

```
DS  <-  [ SI + 3  :  SI + 2]
```

10) LES

The working and syntax of this instruction is the same as the LDS. The difference is only that instead of data segment register (DS), Extra Segment register (ES) is used.

Data Transfer Instructions

1) MOV Destination, Source
Moves a byte/word **from** the **source to** the **destination** specified in the instruction.
Source: *Register, Memory Location, Immediate Number*
Destination: Register, Memory Location
Both, source and destination cannot be memory locations.

```
Eg: MOV CX, 0037H          ; CX ← 0037H
    MOV BL, [4000H]        ; BL ← DS:[4000H]
    MOV AX, BX             ; AX ← BX
    MOV DL, [BX]           ; DL ← DS:[BX]
    MOV DS, BX             ; DS ← BX
```

2) PUSH Source
Push the **source** (word) **into** the **stack** and decrement the stack pointer by two.
The source MUST be a **WORD** (**16 bits**).
Source: *Register, Memory Location*

```
Eg: PUSH CX                ; SS:[SP-1] ← CH, SS:[SP-2] ← CL
                           ; SP ← SP – 2
    PUSH DS                ; SS:[SP-1, SP-2] ← DS
                           ; SP ← SP – 2
```

3) POP Destination
POP a **word from** the **stack** into the given destination and increment the Stack Pointer by 2. The destination MUST be a **WORD** (16 bits).
Destination: *Register [EXCEPT CS], Memory Location*

```
Eg: POP CX                 ; CH ← SS:[SP], CL ← SS:[SP+1]
                           ; SP ← SP + 2
    POP DS                 ; DS ← SS:[SP, SP+1]
                           ; SP ← SP + 2
```

Please Note: **MOV, PUSH, POP** are the ONLY instructions that use the Segment Registers as operands {except CS}.

4) PUSHF
Push value of **Flag Register into stack** and decrement the stack pointer by 2.

```
Eg: PUSHF                  ; SS:[SP-1] ← FlagH, SS:[SP-2] ← FlagL, SP ← SP – 2
```

5) POPF
POP a **word from** the **stack into** the **Flag register**.

```
Eg: POPF                   ; FlagL ← SS:[SP], FlagH ← SS:[SP+1], SP ← SP + 2
```

6) XCHG Destination, Source
Exchanges a byte/word between the **source and** the **destination** specified in the instruction.
Source: *Register, Memory Location*
Destination: Register, Memory Location
Even here, both operands cannot be memory locations.

```
Eg: XCHG CX, BX            ; CX ←→ BX
    XCHG BL, CH            ; BL ←→ CH
```

7) **XLATB / XLAT** (very important)
Move into AL, the **contents of** the **memory location** in Data Segment, **whose** effective **address** is **formed by** the **sum of BX and AL**.
Eg: **XLAT** ; *AL ← DS:[BX + AL]*
; i.e. if DS = 1000H; BX = 0200H; AL = 03H
; ∴ 10000 ... DS × 16
; + 0200 ... BX
; + 03 ... AL
; =**10203** ∴ **AL ← [10203H]**

Note: the difference between XLAT and XLATB

In XLATB there is no operand in the instruction.
E.g.:: XLATB
It works in an implied mode and does exactly what is shown above.

In XLAT, we can specify the name of the look up table in the instruction
E.g.:: XLAT SevenSeg
This will do the translation form the look up table called SevenSeg.
In any case, the base address of the look up table must be given by BX.

8) **LAHF**
Loads AH with **lower byte** of the **Flag** Register.

9) **SAHF**
Stores the contents of **AH into** the **lower byte** of the **Flag** Register.

10) **LEA register, source**
Loads Effective Address (offset address) **of** the **source into** the **given register**.
Eg: **LEA BX, Total** ; *BX ← offset address of Total in Data Segment.*

11) **LDS destination register, source**
Loads the **destination register and DS** register **with offset** address **and segment address** specified by the **source**.
Eg: **LDS BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *DS← {DS: [Total + 2], DS:[Total + 3]}*

12) **LES destination register, source**
Loads the **destination register and ES** register with the **offset address and** the **segment address** indirectly specified by the **source**.
Fg: **LES BX, Total** ; *BX ← {DS:[Total], DS:[Total + 1]},*
; *ES← {DS: [Total + 2], DS:[Total + 3]}*

Q53) Explain in detail Arithmetic Instructions

Ans: BA 45-58

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc. In 8086 the destination address is need not to be the accumulator.

Let us see the arithmetic instructions of 8086 microprocessor. Here the D and S are destination and source respectively. D and S can be either register, data or memory address.

https://www.geeksforgeeks.org/arithmetic-instructions-8086-microprocessor/

https://www.tutorialspoint.com/arithmetic-instructions-in-8086-microprocessor

Opcode	Operand	Description
ADD	D,S	Used to add the provided byte to byte/word to word.
ADC	D,S	Used to add with carry.
INC	D	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.
SUB	D,S	Used to subtract the byte from byte/word from word.
SBB	D,S	Used to perform subtraction with borrow.
DEC	D	Used to decrement the provided byte/word by 1.
NEG	D	Used to negate each bit of the provided byte/word and add 1/2's complement.
CMP	D	Used to compare 2 provided byte/word.
AAS	----	Used to adjust ASCII codes after subtraction.
DAS	----	Used to adjust decimal after subtraction.
MUL	8-bit reg	Used to multiply unsigned byte by byte/word by word.
IMUL	8 or 16-bit reg	Used to multiply signed byte by byte/word by word.
AAM	----	Used to adjust ASCII codes after multiplication.
DIV	8-bit reg	Used to divide the unsigned word by byte or unsigned double word by word.

Opcode	Operand	Description
IDIV	8 or 16-bit reg	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.

Q54) Explain in detail Bit Manipulation Instructions

Ans:

LOGICAL INSTRUCTIONS [BIT MANIPULATION INSTRUCTIONS]

1) **NOT destination**
This instruction forms the **1's complement** of the destination, and stores it back in the destination.
Destination: Register, Memory Location. **No Flags affected**.
Eg: **Assume** AL= 0011 0101
NOT AL ; *AL ← 1100 1010 … i.e. AL = 1's Complement (AL)*

2) **AND destination, source**
This instruction **logically ANDs** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **AND BL, CL** ; *BL ← BL AND CL*

3) **OR destination, source**
This instruction **logically Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **OR BL, CL** ; *BL ← BL OR CL*

4) **XOR destination, source**
This instruction **logically X-Ors** the source with the destination and stores the **result in the destination**. Source and destination have to be of the same size.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **XOR BL, CL** ; *BL ← BL XOR CL*

5) **TEST destination, source**
This instruction **logically ANDs** the source with the destination **BUT** the **RESULT** is **NOT STORED ANYWHERE**. **ONLY** the **FLAG** bits are **AFFECTED**.
Source: Register, Memory Location, Immediate Value
Destination: Register, Memory Location
PF, SF, ZF affected; **CF, OF ← 0**; **AF** becomes **undefined**.
Eg: **TEST BL, CL** ; *BL AND CL ; result not stored; Flags affected.*
*Note: Don't forget this instruction because it will be used later in **multiprocessor systems**!*

https://www.eeeguide.com/bit-manipulation-instructions-in-8086/

Q55) Explain in detail Program Execution Transfer Instructions

Ans:

https://www.geeksforgeeks.org/program-execution-transfer-instructions-8086-microprocessor/

https://www.tutorialspoint.com/microprocessor/microprocessor_8086_instruction_sets.htm

Q56) Explain in detail String Instructions

Ans: BA 63-65

https://www.geeksforgeeks.org/string-manipulation-instructions-8086-microprocessor/

https://tutorialspoint.dev/computer-science/microprocessor/string-manipulation-instructions-8086-microprocessor

While talking about the 8086 microprocessors, the Strings can be defined as the collection of ASCII characters. Each ASCII character is of one byte, and each byte of a String is stored at successive memory locations.

Source Index (SI), Destination Index (DI) and the general-purpose register-CX (which functions as a counter) are the registers involved in performing string operations.

The following are the **various string manipulation instructions in the 8086 microprocessor**:

1) REP

Stands for **'Repeat'**. This instruction repeats the given instruction(s) till **CX** does not becomes zero, i.e. **CX!=0**

```
    REP  instruction_to_be_repeated
```

2) REPE

This instruction repeats the given instruction(s) till **CX** remains zero, i.e. **CX=0**

```
    REPE  instruction_to_be_repeated
```

3) REPZ

It repeats the given instruction(s) while **ZF** remains 1, i.e. **ZF=1**

```
    REPZ  instruction_to_be_repeated
```

4) REPNE

This instruction is same as **REP**. It repeats the given instruction(s) till **CX** remains zero, i.e. **CX=0**

```
    REPZ  instruction_to_be_repeated
```

5) REPNZ

It repeats the instructions while **ZF=0**

```
    REPNZ  instruction_to_be_repeated
```

6) MOVSB

Stands for **'Move String Byte'**. It moves the contents of a byte (8 bits) from **DS:SI** to **ES:DI**

```
    MOVSB
```

7) MOVSW

Stands for **'Move String Word'**. It moves the contents of a word (16 bits) from **DS:SI** to **ES:DI**

```
    MOVSW
```

8) MOVSD

Stands for **'Move String Double Word'**. It moves the contents of a double word (32 bits) from **DS:SI** to **ES:DI**

```
    MOVSD
```

9) CMPSB

Stands for **'Compare String Byte'**. It moves the compares the contents of a byte (8 bits) at **DS:SI** with the contents of a byte at **ES:DI** and sets the flag.

```
    CMPSB
```

10) CMPSW

Stands for **'Compare String Word'**. It moves the compares the contents of the word (16 bits) at **DS:SI** with the contents of the word at **ES:DI** and sets the flag.

```
    CMPSW
```

11) CMPSD

Stands for **'Compare String Double Word'**. It moves the compares the contents of the double word (32 bits) at **DS:SI** with the contents of the double word at **ES:DI** and sets the flag.

```
    CMPSD
```

Q57) Explain in detail Processor Control Instructions

Ans: https://www.tutorialspoint.com/microprocessor/microprocessor_8086_instruction_sets.htm

Type 2) **Processor Control / Machine Control Instructions**
(these are instructions that directly operate on Flag Reg)

In the exam first explain the following instructions: **PUSHF, POPF, LAHF and SAHF**

For Carry Flag

1) STC
This instruction **sets** the **Carry Flag**. No Other Flags are affected.

2) CLC
This instruction **clears** the **Carry Flag**. No Other Flags are affected.

3) CMC
This instruction **complements** the **Carry Flag**. No Other Flags are affected.

For Direction Flag

4) STD
This instruction **sets** the **Direction Flag**. No Other Flags are affected.

5) CLD
This instruction **clears** the **Direction Flag**. No Other Flags are affected.

For Interrupt Enable Flag

6) STI
This instruction **sets** the **Interrupt Enable Flag**. No Other Flags are affected.

7) CLI
This instruction **clears** the **Interrupt Enable Flag**. No Other Flags are affected.

Note: There is no direct way to alter TF. It can be altered through program as follows:

To set TF:

```
PUSHF          ; push contents of Flag register into the stack
POP BX         ; pop contents of flag reg from the stack-top into BX
OR BH, 01H     ; set the bit corresponding to TF, in the BH register
PUSH BX        ; push the modified BX register into the stack
POPF           ; pop the modified contents into flag register.
```

To reset TF:

```
PUSHF          ; push contents of Flag register into the stack
POP BX         ; pop contents of flag reg from the stack-top into BX
AND BH, FEH    ; reset the bit corresponding to TF, in the BH register
PUSH BX        ; push the modified BX register into the stack
POPF           ; pop the modified contents into flag register.
```

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.

Q58) Explain size of page, page table, page directory

Ans:

A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.

Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages (2^(20)). Because each page contains 4K bytes 2^(12) bytes), the tables of one page directory can span the entire physical address space of the 80386 (2^(20) times 2^(12) = 2^(32)).

The physical address of the current page directory is stored in the CPU register CR3, also called the page directory base register (PDBR). Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two. Refer to Chapter 10 for information on initialization of CR3 . Refer to Chapter 7 to see how CR3 can change for each task .

A **page** is simply a contiguous chunk of memory. x86 (32-bit) supports 3 sizes of pages: 4MB, 2MB, and 4KB, with the latter being the most commonly used in mainstream operating systems. A **page table** is an array of 1024 * 32-bit entries (conveniently fitting into a single 4KB page). Each entry points to the physical address of a page. Because a single page table is not able to represent the entire address space on its own (1024 entries * 4KB = only 22-bits of address space), we require a second level page table: a **page directory**. A page directory also consists of 1024 * 32-bit entries (again fitting into a single page), each pointing to a page table. We can see that now 1024 * 1024 * 4KB = 32-bits and with this 3-level structure we are able to map the entire 4GB virtual address space.

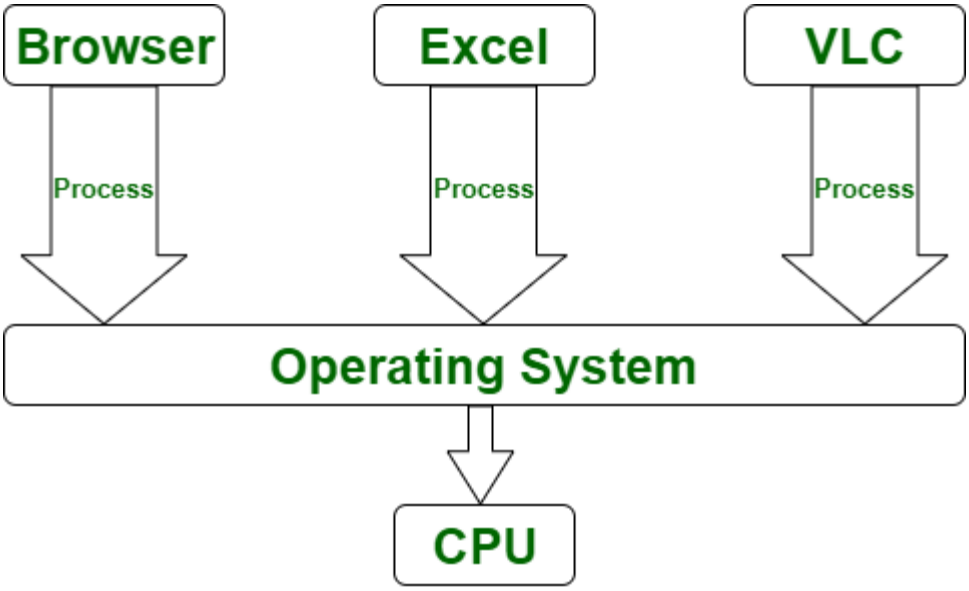
Q59) What is difference between multitasking and multithreading?

Ans:

https://www.geeksforgeeks.org/difference-between-multi-tasking-and-multi-threading/

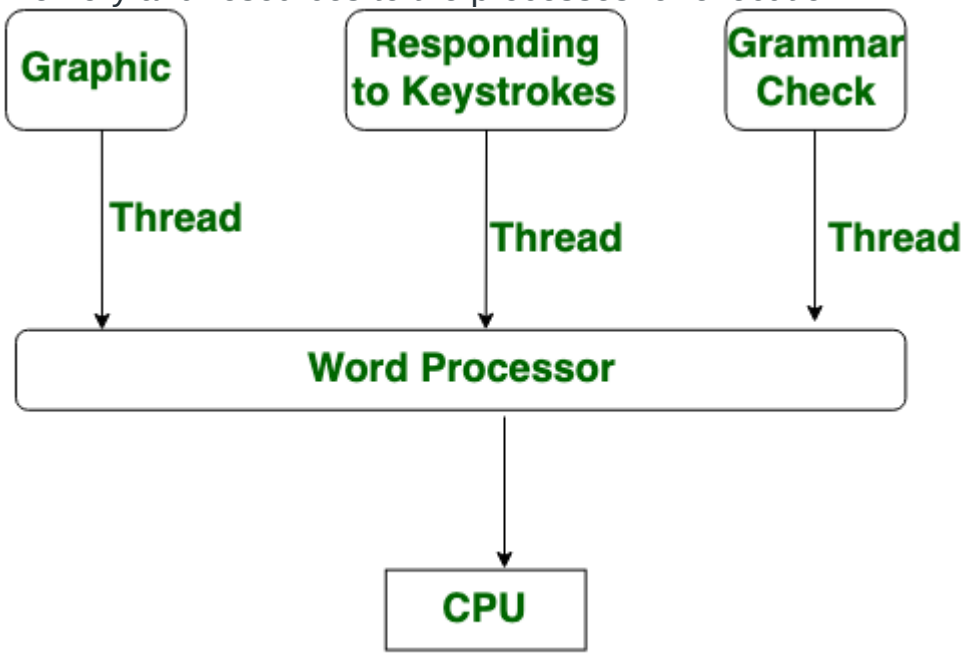
Multitasking:

Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, In multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.



Multitasking

Multithreading:
Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.



Multithreading

Let's see the difference between multitasking and multithreading:

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.

S.NO	Multitasking	Multithreading
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated same memory.
4.	Multitasking component involves multiprocessing.	While multithreading component does not involve multiprocessing.
5.	In multitasking, CPU is provided in order to execute many tasks at a time.	While in multithreading also, CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don't share same resources, each process is allocated separate resources.	While in multithreading, each process share same resources.
7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of process takes more time.	While in multithreading, termination of thread takes less time.

Q60) Explain in detail Privilege Levels.

Ans:

The concept of privilege is implemented by assigning a value from zero to three to key objects recognized by the processor. This value is called the privilege level. The value zero represents the greatest privilege, the value three represents the least privilege. The following processor-recognized objects contain privilege levels:

- Descriptors contain a field called the descriptor privilege level (DPL).
- Selectors contain a field called the requestor's privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that originates a selector.
- An internal processor register records the current privilege level (CPL). Normally the CPL is equal to the DPL of the segment that the processor is currently executing. CPL changes as control is transferred to segments with differing DPLs.

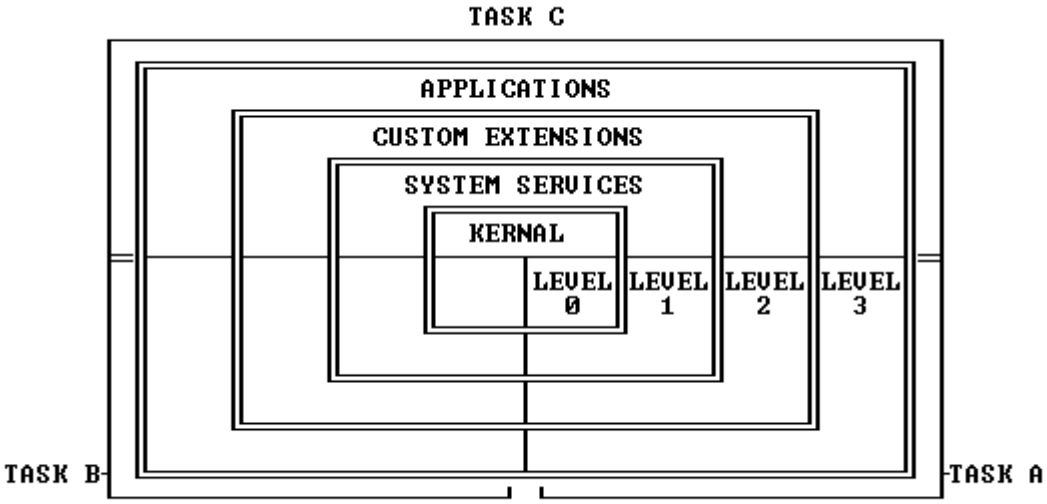
The processor automatically evaluates the right of a procedure to access another segment by comparing the CPL to one or more other privilege levels. The evaluation is performed at the time the selector of a descriptor is loaded into a segment register. The criteria used for evaluating access to data differs from that for evaluating transfers of control to executable

segments; therefore, the two types of access are considered separately in the following sections.

Figure 6-2 shows how these levels of privilege can be interpreted as rings of protection. The center is for the segments containing the most critical software, usually the kernel of the operating system. Outer rings are for the segments of less critical software.

It is not necessary to use all four privilege levels. Existing software that was designed to use only one or two levels of privilege can simply ignore the other levels offered by the 80386. A one-level system should use privilege level zero; a two-level system should use privilege levels zero and three.

Figure 6-2. Levels of Privilege



PRIVILEGE CHECK

80386 µP provides 4 levels of protection called Privilege Levels.

PRIVILEGE LEVEL	ASSIGNED TO	HIERARCHY
PL0	Operating System Kernel	Most Privileged
PL1	System Services	⋮
PL2	Operating System Extensions	
PL3	User Programs	Least Privileged

The following Privilege Levels are checke dbefor granting access to data or code.

RPL (Requestor Privilege Level): RPL is the Privilege Level of the original supplier of the selector. RPL is determined by the two LSBs of the selector.

DPL (Descriptor Privilege Level): DPL is the least Privilege Level at which a task may access that Descriptor and thereby access the segment associated with the Descriptor. DPL is stored at bit 5 and 6 of the access rights by of the Descriptor.

CPL (Current Privilege Level): CPL is the Privilege Level at which the task is currently executing. It is equal to the Privilege Level of the Code Segment currently being executed. It is Stored in the two LSBs of the Code Segment Register (Except for a Confirming Code Segment). A user invisible register stroes the value of CPL. CPL hanges as control is transferred to a segment of a different Privilege Level.

EPL (Effective Privilege Level): It is the least Privilege Level between RPL and CPL (numerically highest).

∴ EPL = max (RPL, CPL).

Q61) What is TLB Buffer?

Ans: A translation lookaside buffer (TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.

When a virtual memory address is referenced by a program, the search starts in the CPU. First, instruction caches are checked. If the required memory is not in these very fast caches, the system has to look up the memory's physical address. At this point, TLB is checked for a quick reference to the location in physical memory.

When an address is searched in the TLB and not found, the physical memory must be searched with a memory page crawl operation. As virtual memory addresses are translated, values referenced are added to TLB. When a value can be retrieved from TLB, speed is enhanced because the memory address is stored in the TLB on processor. Most processors include TLBs to increase the speed of virtual memory operations through the inherent latency-reducing proximity as well as the high-running frequencies of current CPU's.

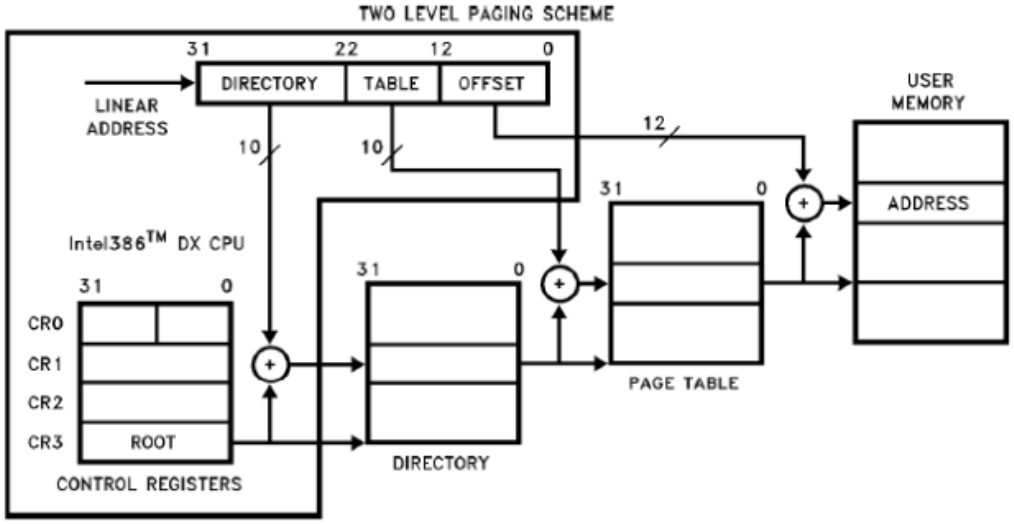
TLBs also add the support required for multi-user computers to keep memory separate, by having a user and a supervisor mode as well as using permissions on read and write bits to enable sharing.

TLBs can suffer performance issues from multitasking and code errors. This performance degradation is called a cache thrash. Cache thrash is caused by an ongoing computer activity that fails to progress due to excessive use of resources or conflicts in the caching system.

https://www.geeksforgeeks.org/translation-lookaside-buffer-tlb-in-paging/

PAGE TRANSLATION

Paging Mechanism:



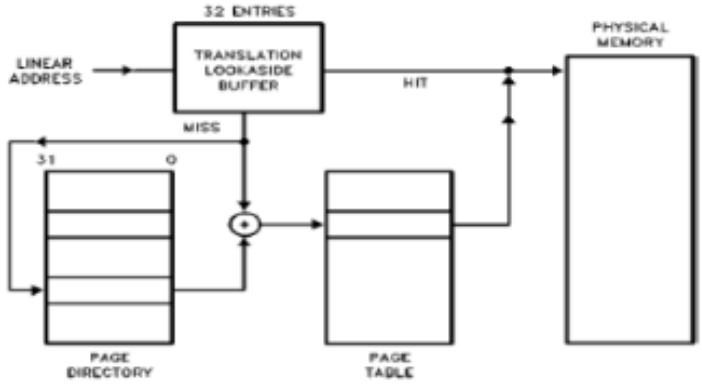
Page Directory Entry:

31 … 12	11 … 9	8	7	6	5	4	3	2	1	0
PAGE TABLE ADDRESS 31..12	OS RESERVED	0	0	D	A	0	0	U/S	R/W	P

Page Table Entry:

31 … 12	11 … 9	8	7	6	5	4	3	2	1	0
PAGE FRAME ADDRESS 31..12	OS RESERVED	0	0	D	A	0	0	U/S	R/W	P

Translation Look-aside Buffer (TLB):



The Virtual Memory space is divided into equal size blocks of 4KB called "Pages".

1) **The Physical Memory space (also called Main Memory) is also divided into equal size blocks of 4 KB called Page Frames (also simply called pages).**
   As Physical Memory is of 4 GB and page size is 4 KB **there are total 1 M pages (2^{20}) in the Physical Memory.**
2) A page from Virtual Memory is loaded into any available page frame of Physical Memory.
3) Whenever a page is required to be accessed, the µP first checks if the desired page is present in the Physical Memory.
   If so, it is called a "HIT" and the operation is performed on the Physical Memory.
4) **A "Page Fault" (MISS) occurs when the desired page is not present in the Physical Memory.**
5) On a Page Fault the desired page is loaded form Virtual Memory into any available page frame of Physical Memory.
6) If no page frame is available, then a **"Page Replacement"** is performed by replacing an old page from the Physical Memory with the new desired page from the Virtual Memory.
   Various algorithms like FIFO (First in first out), LRU (Least recently used) or LFU (Least frequently used) are used to determine which page of the Physical Memory must be replaced.
7) Once the page to be replaced is decided, a **"Dirty Bit"** is checked to determine if the page is modified in the Physical Memory.
8) **If Dirty bit = 1, then the page has been modified** (is "Dirty") and hence must be copied back into Virtual Memory before being replaced else the modified information will be lost.
9) If the Dirty bit = 0, then the page is not modified and hence can be directly replaced without being copied back into Virtual Memory.
10) Since a page of Virtual Memory can be loaded into any page frame of Physical Memory, a **"Page Table"** is required to give the mapping between Virtual Memory page number and Physical Memory page frame number.
11) Simply speaking the **Page table tells which page of Virtual Memory is present in which page of Physical Memory**.
12) But since there are too many page frames in the Physical Memory (2^{20} i.e. 1M), the page table will become too large and searches will become extremely slow.
    Hence the mechanism is further subdivided.
13) Instead of having straight 1M (2^{20}) entries in the page table, **there are 1K (2^{10}) entries in a page table and there are 1K (2^{10}) such page tables.**
    {$2^{20} = 2^{10} \times 2^{10}$ .....!}

14) Each page table is of 4KB and has 1K "Page Table Entries" (PTEs) each of size 4 bytes. Each PTE gives information about a Page Frame.

15) The PTE has following information:
20 bit page frame address: Gives the upper 20 bits of the starting address of the page frame. Lower 12 bits are 0...0 as the page is of 4 KB and starts from a 4 KB aligned location (refer Bharat Sir's lecture notes...)
D: Dirty bit indicates whether the page has been modified (1) or not (0).
A: Accessed Bit tells whether the page has been accessed or not (1 means accessed). This is used by replacement algorithms ☺ For doubts contact Bharat Sir on 98204 08317
U/S: User or Supervisor and R/W: Read or Read and Write give protection information
P: Present bit indicates whether the page is present in the Physical Memory.
If P = 1 then the page is present and the 20 bit address field is valid, else the page is not present in the Physical Memory and the 20 bit address field is obviously invalid.

16) Information about all the page tables is stored in the "Page Directory".

17) The page directory is of 4KB and has 1K "Page Directory Entries" (PDEs) each of size 4 bytes. Each PDE gives information about a Page Table.

18) The PDE has following information:
20 bit page table address: Gives the upper 20 bits of the starting address of the corresponding page table. Lower 12 bits are 0...0 as the page table is of 4 KB and starts from a 4 KB aligned location (refer Bharat Sir's lecture notes...)
D: Dirty bit (explained above)
A: Accessed Bit (explained above)
U/S: User or Supervisor and R/W: Read or Read and Write (explained above)
P: Present bit (explained above)

19) The Page Directory is of 4 KB and begins from a 4 KB aligned location.

20) **The address of the page directory is given by the PGBR (page Directory Base Register) field in CR3.**

21) The 32 bit Linear Address can be divided into three parts.
The higher 10 bits select one PDE out of 1K PDEs in the page directory.
This gives the starting address in the page table.
The next 10 bits select one PTE out of 1K PTEs in the page table.
This gives the starting address of the page frame.
Finally, the lowest 12 bits (offset) select a location within the 4KB page.

22) This means, to access any location, µP must first access a PDE in the page directory then a PTE in the page table, then access the page. This can make the process very slow. To speed up the process a **"Translation Look-aside Buffer"** is used (called TLB).

23) **The TLB is an on chip cache which stores 32 most recently used PTEs and PDEs.** This makes subsequent access to these pages (whose information is cached in the TLB) much faster as there is no need to access the Page directory and the page table. µP can directly obtain the starting address of the page frame from the TLB and hence directly access the page. Due to principle of "Locality of reference" most systems get a Hit ratio of >98% on the TLB, thus making the operations very fast.

U/S	R/W	Permitted Level 3	Permitted Access Levels 0, 1, or 2
0	0	None	Read/Write
0	1	None	Read/Write
1	0	Read-Only	Read/Write
1	1	Read/Write	Read/Write