UNIT II

The IA-32 Pentium Example: Registers and Addressing, IA-32 Instructions, IA-32 Assembly Language, Program Flow Control, Logic and Shift/Rotate Instructions, I/O Operations, Subroutines, Other Instructions, Program Examples.

2 Marks

1. Write about IA-32

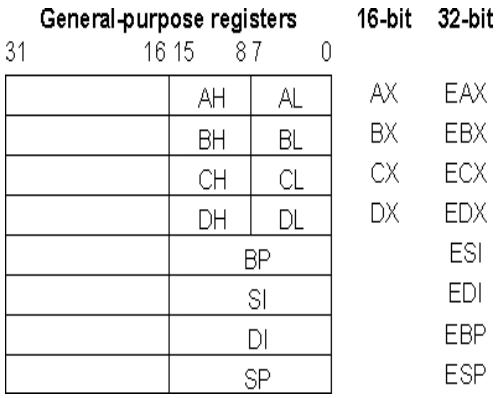
The Intel architecture (IA) processors operate with 32-bit memory address and 32-bit data operands. They are referred to as IA-32 processors, and the most recent Pentium series.

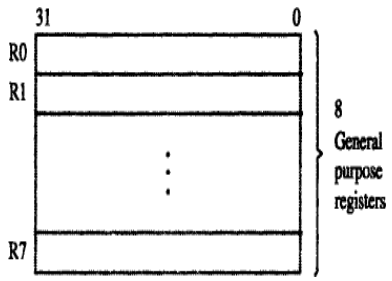
The first IA-32 processor was 80386 then 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4 has been implemented. Theses processors has increasing level of performance, achieved through a number of architectural and microelectronic technology improvements. The latest members of the family have specialized instructions for handling multimedia information and for vector data processing.

2. List out the various register of IA- 32 Register structure

- General purpose registers
- floating point registers
- segment registers
- instruction pointer
- status registers

3. Write about General purpose registers

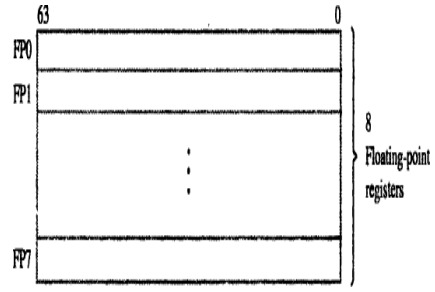




The above diagram shows the general purpose registers. The eight 32-bit registers labeled R0 through R7 are general purpose registers that can be used to bold either data operands or addressing information.

4. Write about floating point registers

There are eight floating point registers for holding double word or quad word (64 bits) floating point data operands. The floating point registers have an extension field to provide a total length of 80 bits, the extra bits are used for increased accuracy while floating point numbers are operated on in the processor.



5. What is Segment registers

There are six segment registers that hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. The six segment registers are:

- CS: code segment register
- SS: stack segment register
- DS, ES, FS, GS: data segment registers

Code segment (CS)

The code segment holds the instructions of a program.

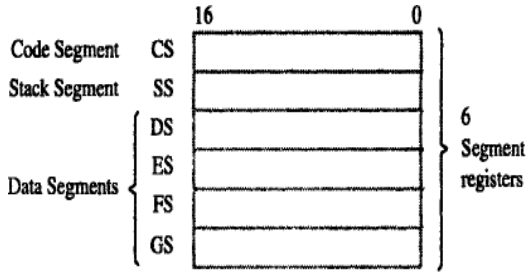
Stack segment (SS)

The stack segment contains the processor stack.

Data segment (DS, RD, FS, and GS)

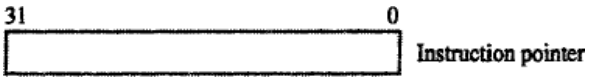
Four data segments are provided for holding data operands. These four data segment registers provide programs with flexible and efficient ways to access data.

The six segment registers below contain selector values that are used in locating these segments in the memory address space.

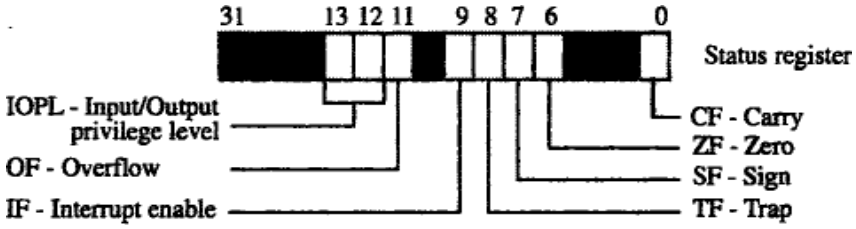


6. Define Instruction pointer

The **EIP** register is the 32-bit instruction pointer. The EIP register (or instruction pointer) can also be called "program counter." It contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP,CALL, RET instructions.



7. Write about status registers



The status register holds the condition code flags (CF,ZF,SF,OF).These flags contain information about the results of arithmetic operations.

8. What is EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. When it is used in the conditional control instructions look at the condition code bits (in the EFLAGS register) to make a decision on whether to take the jump or not.

9. Write about ESP register and EBP register

The **ESP** register is the 32-bit stack pointer, used to manage push and pop operations.

The **EBP** register is the 32-bit base pointer. In connection with the ESP, the EBP is used to manage the stack frame, that part of the stack used to communicate with subprograms and store local variables.

10. Write about the Index Registers

The **ESI** and **EDI** registers are used as source and destination addresses for string and array operations.

The ESI "**Extended Source Index**" and EDI "**Extended Destination Index**" facilitate high–speed memory transfers.

11. List the various addressing modes of IA-32

The addressing mode of IA-32 includes:

1. Immediate mode
2. Direct mode
3. Register mode
4. Register indirect mode
5. Base with displacement mode
6. Index with displacement mode
7. Base with index mode
8. Base with index and displacement mode

12. What is immediate mode? Give example

The operand is contained in the instruction. It is a signed 8-bit or 32-bit number, with the length being specified by a bit in the OP code of the instruction. Thus bit is 0 for the short version and 1 for the long version.

Instruction format:

Opcode	Register	Value

Example:

MOV EAX, 25

The above instruction moves the decimal value 25 into the EAX register. A number given in this form using the digits 0 through 9 is assumed to be in decimal notation .the suffix B and H are used to specify binary and hexadecimal numbers, respectively .For example the instruction,

MOV EAX, 3FAOOH

Moves the hex number 3FA00 into EAX.

13. What is direct mode? Give example

The memory address of the operand is given by a 32-bit value in the instruction

Instruction format:

Opcode	Register	Location

Example:

MOV EAX, LOCATION

The above instruction uses the direct addressing mode to move the doubleword at the memory location specified by the address label LOCATION into register EAX.This assumes that the LOCATION has been defined as an address label for a memory location in the data declaration. If LOCATION represents the address 1000 then this instruction moves the doubleword at 1000 into EAX.

In the IA-32 assembly language square brackets can be used to explicitly indicate the direct addressing mode as in the instruction.

MOV EAX, [LOCATION]

14. What is Register mode? Give example

The operand is contained in one of the general purpose register specified in the instruction.

Instruction format:

Opcode	Dest.Register	Src.Register

Example:

MOV EAX, ECX

Both operands use register mode. The contents of register **ECX** are copied to register EAX.

Before execution of the above instruction, the contents of ECX and EAX are:

ECX

50

EAX

00

After execution

ECX

50

EAX

50

15. What is Register indirect mode? Give example

The memory address of the operand is contained in one of the eight general purpose register specified in the instruction.

Instruction format:

Opcode	Register	[Register]

Example:

MOV EAX,[EBX]

The above instruction moves the contents of LOCATION specified by the register EBX into the register EAX.

Before execution of the above instruction, the contents of EAX and EBX are:

EAX

00

EBX

1000

20

1000

After execution

EAX

20

EBX

1000

20

1000

16. What is Base with displacement mode? Give example

An 8-bit or 32-bit signed displacement and one of the eight general purpose register to be used as a base register are specified in the instruction. The effective address of the operand in the sum of the contents of the base register and the displacement.

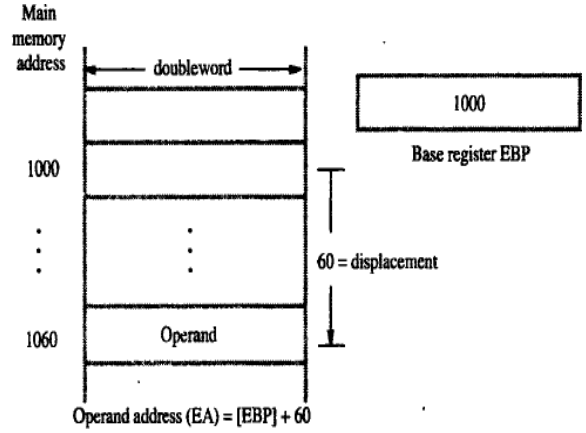
Instruction format:

Opcode	Dest.Register	[Register]+Displacement

Example:

MOV EAX, [EBP + 60]

The second operand uses base displacement mode. The instruction contains a constant. That constant is added to the contents of register EBP to form an effective address. The contents of memory at the effective address are copied into register EAX.



(a) Base with displacement mode, expressed as [EBP + 60]

17. What is Index with displacement mode? Give example

A 32-bit signed displacement, one of the eight general purpose register to be used as an index register, and a scale factor of 1,2,4 or 8 are specified in the instruction. To obtain the effective address of the operand, the contents of the index register are multiplied by the scale factor and then added to the displacement.i.e

Offset = (Index * Scale) + displacement

Instruction format:

Opcode	Dest.Register	[Register] * Scale factor +Displacement

Example:

MOV AL,[EBP * 4 + 10]

18. What is Base with index mode? Give example

Two of the eight general purpose register and a scale factor of 1,2,4 or 8 are specified in the instruction. The register is used as base and index register and the effective address of the operand is calculated as follows: the contents of the index register are multiplied by the scale factor and added to the contents of the base register. i.e

Offset = Base + (Index * Scale)

Instruction format:

Opcode	Dest.Register	[Register1] + [Register2] * Scale factor

Example:

MOV EAX, [ESP+ESI*4]

The contents of registers ESI is multiplied with the scale factor 4 and then the content of ESP is added to form an effective address. The contents of memory at the effective address are copied into register EAX

19. What is Base with index and displacement mode? Give example

An 8 bit or 32 –bit signed displacement two of the eight general purpose registers and a scale factor of 1,2,4 or 8 are specified in the instruction. The register is used as base and index register and the effective address of the operand is calculated as follows: the contents of the index register are multiplied by the scale factor and then added to the contents of the base register and the displacement.

An effective address is computed by:

Offset = Base + (Index * Scale) + displacement

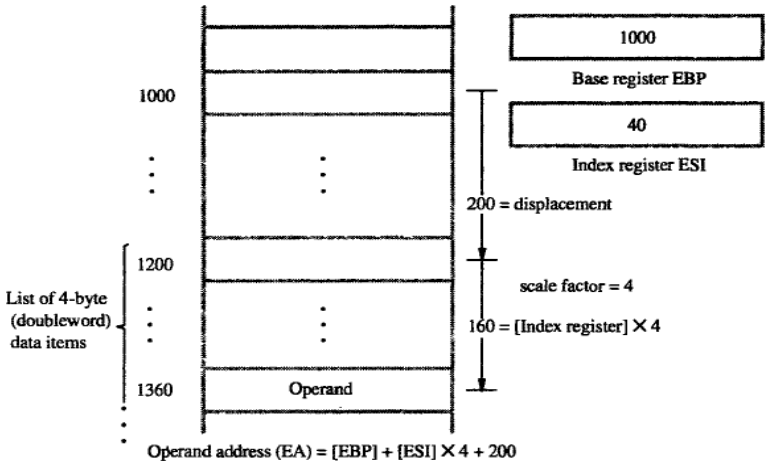
Instruction format:

Opcode	Dest.Register	[Register1] + [Register2] * Scale factor + Displacement

Example:

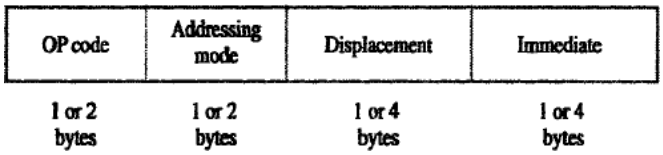
MOV EAX, [EBP+ESI*4 + 200]

The contents of registers ESI is multiplied with the scale factor 4 and then the content of EBP and displacement are added to form an effective address. The contents of memory at the effective address are copied into register EAX



(b) Base with displacement and index mode, expressed as [EBP + ESI * 4 + 200]

20.Write about IA-32 Instruction



The instructions are variable in length ,ranging from one byte to 12 bytes ,consisting of up to four fields .the OP-code field consists of one or two bytes, with most instructions requiring only one byte, the addressing mode information is contained in one or two bytes immediately following the OP code. For instructions that involve the use of only one register in generating the effective address of an operand ,only one byte is needed in the addressing mode field. Two bytes are needed for encoding the last two addressing modes. These modes use two registers to generate the effective address of a memory operand.

21. What is One byte instruction?

Registers can be incremented or decremented by instructions, that occupy one byte. Examples are

INC EDI

And

DEC ECX

In which the general purpose register EDI and ECX are specified by 3-bit codes in the single OP-code byte.

22. What is immediate mode encoding?

The OP-code specifies when the immediate addressing mode is used. For example the instruction

MOV EAX, 820

Is encoded into 5 bytes .a one –byte OP code specifies the move operation, the fact that a 32-bit immediate operand is used and the name of the destination register. The OP code byte is directly followed by the 4-byte immediate value of 820.when an 8-bit immediate operand is used, as in the instruction

MOV DL, 5

Only two bytes are needed to encode the instruction

23. In which situation a program flow control changes.

There are two main ways in which the flow of executing instructions varies from straight –line sequencing, calls to subroutines and returns from them break straight line sequencing. Also, branch instructions, either conditional or unconditional, can cause a break. The branch instructions are called jumps.

24. Write about Conditional jump instruction

The conditional Jump instructions test the four condition code flags in the status register. The instruction

JG LABEL

is an example of a conditional Jump instruction. The condition is *greater-than* as indicated by the G suffix in the OP code.

25. Write about Unconditional Jump Instruction

An unconditional Jump instruction, JMP, causes a branch to the instruction at the target address. In addition to using short (one-byte) or long (four-byte) relative signed offsets to determine the target address, as is done in conditional Jump instructions, the JMP instruction also allows the use of other addressing modes.

26. What is the use of Compare Instructions?

It is often necessary to make conditional jumps in a program based on the results of comparing two numbers. The compare instruction

CMP dst,src

performs the operation

$$dst \leftarrow [src]$$

and sets the condition code flags based on the result obtained. Neither of the operands is changed .the first operand is always compared to the second. For example, the compare instruction by a conditional jump that is based on the “greater than” condition, then the jump will take to the target address if the destination operand is greater than the source operand.

27. What is Logical Shift instruction?

An operand can be shifted right or left, using either logical or arithmetic shifts,by a number of bit positions determined by a specified count. The format of the shift instruction is

OPcode dst , count

Where the destination operand to be shifted is specified by the general addressing modes and the count is given either as an 8-bit immediate value or is contained in the 8-bit register CL.

28. List out the various jump instructions of IA-32

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF ∨ ZF = 0
JBE	Unsigned below or equal	CF ∨ ZF = 1
JGE	Signed greater than or equal	SF ⊕ OF = 0
JL	Signed less than	SF ⊕ OF = 1
JG	Signed greater than	ZF ∨ (SF ⊕ OF) = 0
JLE	Signed less than or equal	ZF ∨ (SF ⊕ OF) = 1

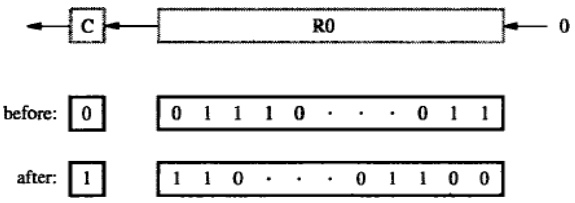
29. List out the various logical shift instruction

There are four shift instructions:

- SHL (Shift left logical)
- SHR (Shift right logical)
- SAL (Shift left arithmetic; operation is identical to SHL)
- SAR (Shift right arithmetic)

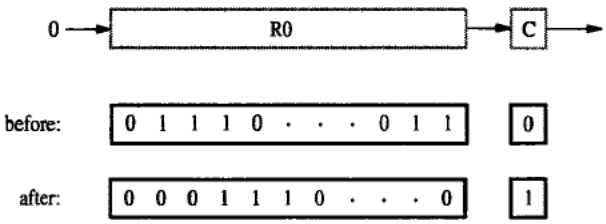
30. What is SHL instruction?

The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.For example, **SHL R0, #2**



31. What is SHR instruction?

The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero. For example, **SHR R0,#2**



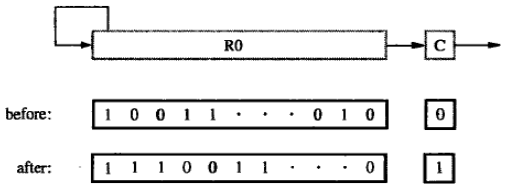
32. What is SAL and SAR instruction

SAL:

The operation of SAL is identical to SHL.

SAR:

SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand. For example, **SAR R0,#2**



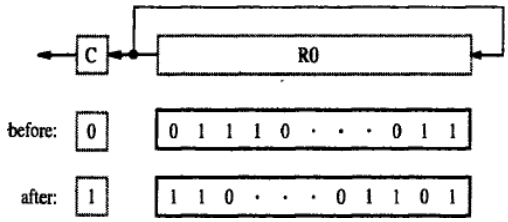
33. List out the various rotate instruction

In addition to the shift instructions, there are also four rotate instructions:

- ROL (Rotate left without the carry flag CF)
- ROR (Rotate right without the carry flag CF)
- RCL (Rotate left including the carry flag CF)
- RCR (Rotate right including the carry flag CF)

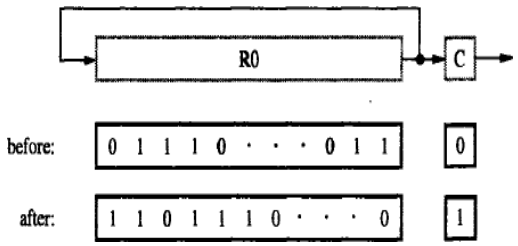
34. What is ROL instruction?

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost
- For example , **ROL R0,#2**



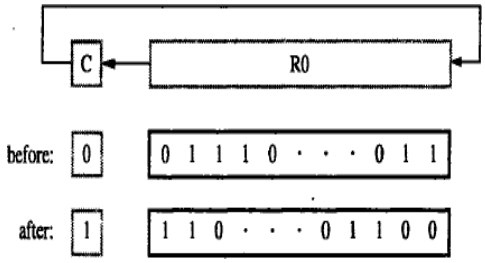
35. What is ROR Instruction?

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost
- For example , **ROR R0,#2**



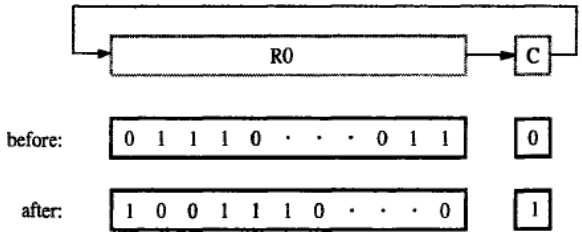
36. What is RCL instruction?

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag
- For example , **RCL R0,#2**



37. What is RCR instruction?

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag
- For example , **ROR R0,#2**



38. What is memory mapped I/O?

- Instead of having special methods for accessing the values to be read or written, just get them from memory or put them into memory.
- The device is connected directly to certain main memory locations.

39. Why use memory mapped I/O?

- Makes programming simpler.
- Do not have special commands to access I/O devices.
  - Just use lw and sw.
- Takes some memory locations
  - Very few compared to the size of main memory.

40. What is Isolated I/O?

The IA-32 instructions set also have two instructions, with OP codes IN and OUT, that are used only for I/O purposes. The addresses issued by these instructions are in an address space that is separate from the memory address space used by the other instructions. This arrangement is called isolated I/O .

41. Write short notes on block transfers

The IA-32 architecture also has two block transfer I/O instructions REPINS and REPOUTS .they transfer a block lf data serially, one item at a time, between the memory and an I/O device. The S suffix in the

OP codes stands for string, and the REP prefix stands for "repeat the item by item transfer until the complex block has been transferred". The instructions themselves do not specify the parameters needed to describe the transfer. These parameters are specified implicitly by processor registers DX,EDI and ECX as follows:

DX contains a 16-bit I/O device address

EDI contains a 32-bit address for the beginning of a block in memory

ECX contains the number of data items to be transferred.

A suffix B or D in the OP-code mnemonic indicates that the item size is either of byte or doubleword length. Thus REPINSB is a byte –block transfer, and REPINSD is a doubleword –block transfer.

42. How block transfer instruction operates?

The block transfer instruction operates as follows: After each data item is transferred, the index register EDI is incremented by 1 or 4 depending on the size of the data items, and the ECX register is decremented by 1.The transfers are repeated until the contents of the counter register ECX have been decremented to 0.The effect of these single instruction is equivalent to a program loop that uses register ECX as the loop counter

43. Write short notes on Subroutines

In the IA-32 architecture, register ESP is used as the stack pointer. It points to the current top element (TOS) in the processor stack. The stack grows toward lower numbered addresses. The width of the stack is 32 bits, that is, all stack entries are double words.

There are two instructions for pushing and popping individual elements onto and off the stack. The instruction

PUSH src

decrements ESP by 4, and then stores the doubleword at location src into the memory location pointed to by ESP. The instruction

POP dst

reverses this process by retrieving the TOS doubleword from the location pointed to by ESP,storing it at location dst, and then incrementing ESP by 4. These instructions implicitly use ESP as the stack pointer. The source and destination operands are specified using the IA-32 addressing modes.

44. List out the other instructions that are available in IA-32

The other instructions available in IA-32 are:

1. Multiplication instruction:
2. Division instruction:
3. Multimedia Extension (MMX) instructions
4. Vector (SIMD) Floating-Point Operations

45. What is Multiplication instruction?

The signed integer multiplication instruction, IMUL, performs 32-bit multiplication. Depending on the form of the instruction that is used, the destination may be implicit and the 64-bit product may be truncated to 32 bits.

One form of this instruction is

IMUL src

which implicitly uses the EAX register as the multiplicand. The multiplier specified by src can be in a register or in the memory. The full 64-bit product is placed in registers EDX (high-order half) and EAX (low-order half).

46. Write about Division instruction:

The integer divide instruction, IDIV, operates on a 64-bit dividend and a 32-bit divisor to generate a 32-bit quotient and a 32-bit remainder. The format of the instruction is

IDIV src

The source operand is the divisor. The 64-bit dividend is formed by the contents of register EDX (high-order half) and register EAX (low-order half). After performing the division, the quotient is placed in EAX and the remainder is placed in EDX.

47. What is MMX instruction?

The IA-32 instruction set has a number of instructions that operate in parallel on such data packed into 64-bit quadwords.(A quadword contains 8 bytes or four 16-bit words).These instructions are called *multimedia extension* (MMX) instructions.

The operands for MMX instructions can be in the memory, or in the eight floating-point registers. Thus, these registers serve a dual purpose. They can hold either floating-point numbers or MMX operands. When used by MMX instructions, the registers are referred to as MM0 through MM7.

48. Write about Vector (SIMD) Floating-Point Operations

A set of instructions that are used to perform arithmetic operations on small group of floating point numbers is provided.SIMD (single-instruction-multiple-data) instructions are useful for vector and matrix calculations in scientific applications.

In Intel terminology, these instructions are called streaming SIMD extension (SSE) instructions. They handle packed 128-bit double quadwords, each consisting of four 32-bit floating-point numbers. Eight additional 128-bit registers, XMM0 to XMM7, are available for holding these operands.

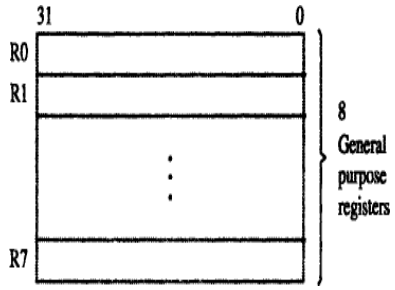
11 Marks

1. Write in detail about IA-32 registers.

IA- 32 Register structure contains the following

- General purpose registers
- floating point registers
- segment registers
- instruction pointer
- status registers

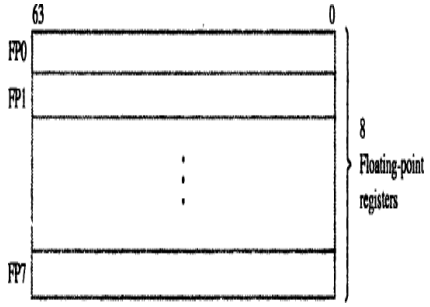
General purpose registers:



The above diagram shows the general purpose registers. The eight 32-bit registers labeled R0 through R7 are general purpose registers that can be used to bold either data operands or addressing information.

Floating point registers:

There are eight floating point registers for holding doubleword or quadword (64 bits) floating point data operands. The floating point registers have an extension field to provide a total length of 80 bits, the extra bits are used for increased accuracy while floating point numbers are operated on in the processor.



Segment registers:

IA-32 architectures are based on a memory model that associates different areas of the memory, called segments, with different usages. There are three segments,

Code segment (CS)

The code segment holds the instructions of a program.

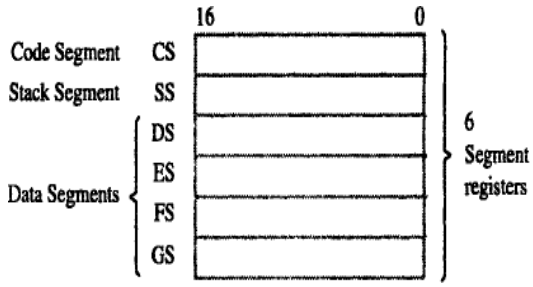
Stack segment (SS)

The stack segment contains the processor stack

Data segment (DS,RD,FS,GS)

Four data segments are provided for holding data operands.

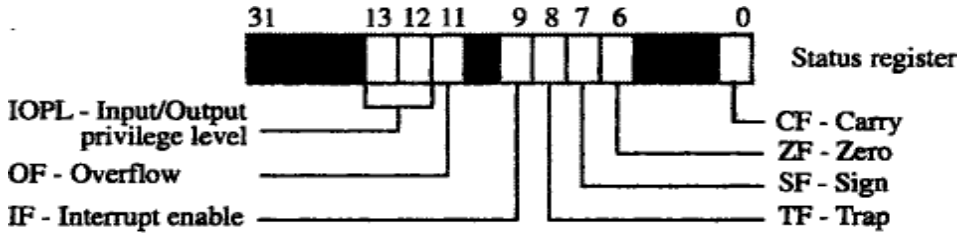
The six segment registers below contain selector values that are used in locating these segments in the memory address space.



Instruction pointer:

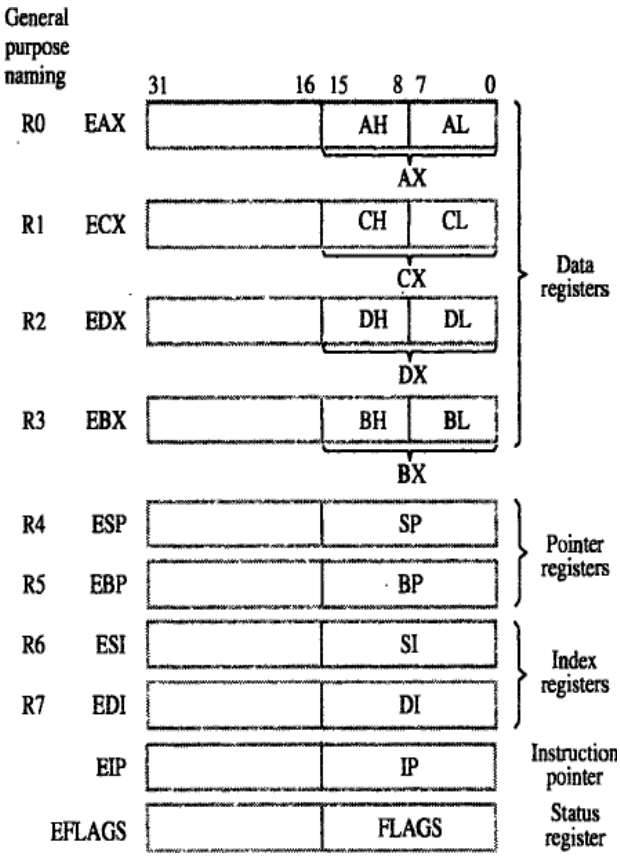
It is a 32-bit register. It serves as the program counter and contains the address of the next instruction to be executed

Status registers:



The status register holds the condition code flags (CF, ZF, SF, OF).These flags contain information about the results of arithmetic operations.

Compatibility of the IA-32 register structure:



The eight general purpose registers are grouped into 3 different types

1. Data register
2. Pointer register
3. Index register

The data register used for holding operands ,and the pointer and index registers for holding address and address indices used to determine the effective address of a memory operand.

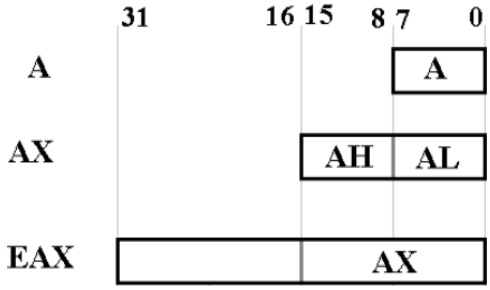
The above figure shows the compatibility of the IA-32 register structure with earlier Intel processor register structures.

The Intel's original 8-bit processors, the data registers were called A,B,C and D.In Intel 16-bit processors, these registers were labeled AX,BX,CX and DX .the high and low-order bytes in each register are identified by suffixes H and L .For example, the two bytes in register AX are referred to as AH and AL.

In IA-32 processors, the prefix E is used to identify the corresponding "extended "32-bit register :EAX,EBX,ECX and EDX.The E –prefix labeling is also used for the other 32-bit registers shown in the above figure. They are the extended versions of the corresponding 16-bit register used in earlier processors. The IA-32 processor state can be switched dynamically between 32-bit operation and 16-bit operation during program execution on an instruction by instruction basis by the use of instruction prefix bytes.

EAX register:

This is the general–purpose register used for arithmetic and logical operations. This division is seen also in the EBX, ECX, and EDX registers; the code can reference BX, BH, CX, CL, etc. This register has an implied role in both multiplication and division.



EBX register:

This can be used as a general–purpose register, but was originally designed to be the base register, holding the address of the base of a data structure. The easiest example of such a data structure is a singly dimensioned array.

ECX register:

This can be used as a general–purpose register, but it is often used in its special role as a counter register for loops or bit shifting operations.

EDX register:

This can be used as a general–purpose register. It also plays a special part in executing integer multiplication and division. For multiplication, DX or EDX store the more significant bits of the product. The 16–bit implementation of multiplication uses AX to hold one of the integers to be multiplied and uses the register pair DX:AX to hold the 32-bit product.

The 32–bit implementation of multiplication uses EAX to hold one of the integers to be multiplied and uses the register pair EDX:EAX to hold the 64-bit product.

The Instruction Pointer(EIP)

The **EIP** register is the 32-bit instruction pointer.The EIP register (or instruction pointer) can also be called "program counter." It contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP,CALL, RET instructions.

The Index Registers

The **ESI** and **EDI** registers are used as source and destination addresses for string and array operations.

The ESI "**Extended Source Index**" and EDI "**Extended Destination Index**" facilitate high–speed memory transfers.

The Other Pointers register's are:

1.ESP register:

The **ESP** register is the 32-bit stack pointer, used to manage push and pop operations.

2.EBP Register:

The **EBP** register is the 32-bit base pointer. In connection with the ESP, the EBP is used to manage the stack frame, that part of the stack used to communicate with subprograms and store local variables.

3.EFLAGS Register:

The **EFLAGS** register holds a collection of at most 32 Boolean flags. The flags are divided into two broad categories: **control flags** and **status flags**.

2. Explain IA-32 Addressing modes

The IA-32 architecture has a large and flexible set of addressing modes. They are designed to access individual data items or data items that are members of an ordered list begins at a specified memory address. There are also several addressing modes that provide mere flexibility in accessing data operands in the memory.

The addressing mode of IA-32 includes:

1. Immediate mode
2. Direct mode
3. Register mode
4. Register indirect mode
5. Base with displacement mode
6. Index with displacement mode
7. Base with index mode
8. Base with index and displacement mode

Immediate mode

The operand is contained in the instruction. It is a signed 8-bit or 32-bit number, with the length being specified by a bit in the OP code of the instruction. Thus bit is 0 for the short version and 1 for the long version.

Instruction format:

Opcode	Register	Value

Example:

MOV EAX,25

The above instruction moves the decimal value 25 into the EAX register. A number given in this form using the digits 0 through 9 is assumed to be in decimal notation .the suffix B and H are used to specify binary and hexadecimal numbers, respectively .For example the instruction,

MOV EAX,3FAOOH

Moves the hex number 3FA00 into EAX.

Direct mode

The memory address of the operand is given by a 32-bit value in the instruction

Instruction format:

Opcode	Register	Location

Example:

MOV EAX,LOCATION

The above instruction uses the direct addressing mode to move the doubleword at the memory location specified by the address label LOCATION into register EAX.This assumes that the LOCATION has been defined as an address label for a memory location in the data declaration. If LOCATION represents the address 1000 then this instruction moves the doubleword at 1000 into EAX.

In the IA-32 assembly language square brackets can be used to explicitly indicate the direct addressing mode as in the instruction.

MOV EAX,[LOCATION]

Register mode

The operand is contained in one of the general purpose register specified in the instruction.

Instruction format:

Opcode	Dest.Register	Src.Register

Example:

MOV EAX, ECX

Both operands use register mode. The contents of register **ECX** is copied to register EAX.

Before execution of the above instruction, the contents of ECX and EAX are:

ECX

50

EAX

00

After execution

ECX

50

EAX

50

Register indirect mode

The memory address of the operand is contained in one of the eight general purpose register specified in the instruction.

Instruction format:

Opcode	Register	[Register]

Example:

MOV EAX,[EBX]

The above instruction moves the contents of LOCATION specified by the register EBX into the register EAX.

Before execution of the above instruction, the contents of EAX and EBX are:

EAX

00

EBX

1000

20

1000

After execution

EAX

20

EBX

1000

20

1000

Base with displacement mode

An 8-bit or 32-bit signed displacement and one of the eight general purpose register to be used as a base register are specified in the instruction. The effective address of the operand in the sum of the contents of the base register and the displacement.

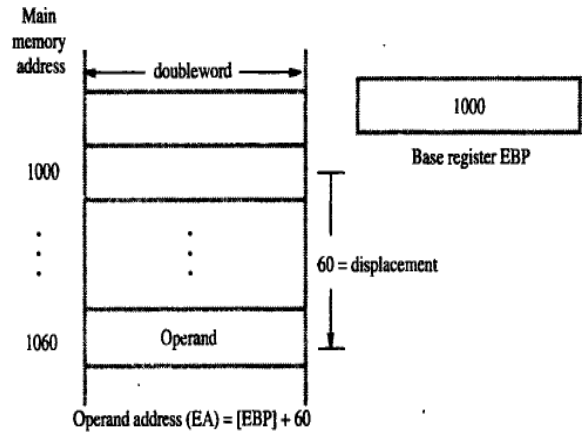
Instruction format:

Opcode	Dest.Register	[Register]+Displacement

Example:

MOV EAX, [EBP + 60]

The second operand uses base displacement mode. The instruction contains a constant. That constant is added to the contents of register EBP to form an effective address. The contents of memory at the effective address are copied into register EAX.



(a) Base with displacement mode, expressed as [EBP + 60]

Index with displacement mode

A 32-bit signed displacement, one of the eight general purpose register to be used as an index register ,and a scale factor of 1,2,4 or 8 are specified in the instruction. To obtain the effective address of the operand, the contents of the index register are multiplied by the scale factor and then added to the displacement.i.e

Offset = (Index * Scale) + displacement

Instruction format:

Opcode	Dest.Register	[Register] * Scale factor +Displacement

Example:

MOV AL,[EBP * 4 + 10]

Base with index mode

Two of the eight general purpose register and a scale factor of 1,2,4 or 8 are specified in the instruction. The registers are used as base and index register and the effective address of the operand is calculated as follows: the contents of the index register are multiplied by the scale factor and added to the contents of the base register.i.e

Offset = Base + (Index * Scale)

Instruction format:

Opcode	Dest.Register	[Register1] + [Register2] * Scale factor

Example:

MOV EAX, [ESP+ESI*4]

The contents of registers ESI is multiplied with the scale factor 4 and then the content of ESP is added to form an effective address. The contents of memory at the effective address are copied into register EAX

Base with index and displacement mode

An 8 bit or 32 –bit signed displacement two of the eight general purpose registers and a scale factor of 1,2,4 or 8 are specified in the instruction. The register is used as base and index register and the effective address of the operand is calculated as follows: the contents of the index register are multiplied by the scale factor and then added to the contents of the base register and the displacement.

An effective address is computed by:

Offset = Base + (Index * Scale) + displacement

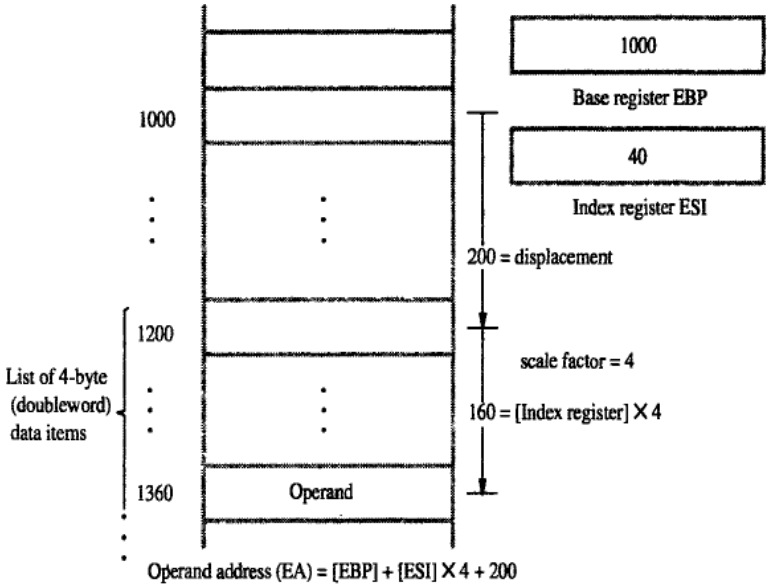
Instruction format:

Opcode	Dest.Register	[Register1] + [Register2] * Scale factor + Displacement

Example:

MOV EAX, [EBP+ESI*4 + 200]

The contents of registers ESI is multiplied with the scale factor 4 and then the content of EBP and displacement are added to form an effective address. The contents of memory at the effective address are copied into register EAX



(b) Base with displacement and index mode, expressed as [EBP + ESI * 4 + 200]

The below table shows the various addressing modes of IA-32 processor:

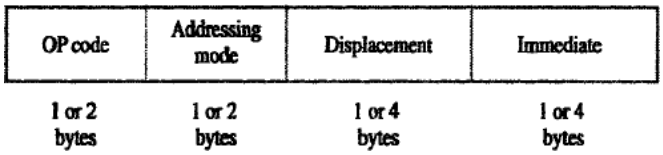
S.No	Name	Assembler syntax	Addressing function
1	Immediate	Value	Operand = Valur
2	Direct	Location	EA = Location
3	Register	Reg	EA = Reg that is Operand=Reg
4	Register indirect	[Reg]	EA = [Reg]
5	Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
6	Index with displacement	[Reg * S + Disp]	EA = [Reg] * S + Disp
7	Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] * S
8	Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] * S + Disp

Where,

Value	an 8 or 32 – bit signed number
Location	a 32-bit address
Reg,Reg1,Reg2	one of the general purpose registers EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI,with the exception that ESP cannot be used as an index register.
Disp	an 8 or 32 – bit signed number except that in the index with displacement mode it can only be 32 bits.
S	a scale factor of 1,2,4 or 8

3. Explain A-32 Instructions

IA-32 Instruction format



The instructions are variable in length ,ranging from one byte to 12 bytes ,consisting of up to four fields .the OP-code field consists of one or two bytes, with most instructions requiring only one byte, the addressing mode information is contained in one or two bytes immediately following the OP code. For instructions that involve the use of only one register in generating the effective address of an operand ,only one byte is needed in the addressing mode field. Two bytes are needed for encoding the last two addressing modes. These modes use two registers to generate the effective address of a memory operand.

If a displacement value is needed in computing an effective address for a memory operand, it is encoded into either one or four bytes in a field that immediately follows the addressing mode field. If one of the operand is an immediate value, then it is placed in the last field of and instruction and it occupies either one or four bytes.

IA-32 instructions have either one or two operands. In the two –operand case, only one of the operands can be in the memory. The other must be in a process register. In addition to the usual instructions for moving data between the memory and the processor register, and for performing arithmetic operations, the instruction set includes a number of different logical and shift/rotate operations on data. Byte string instructions are included for nonnumeric data processing. Push and Pop operations for manipulating the processor stack are directly supported in the instruction set.

The instruction

ADD dst,src

Performs the operation

And

$dst \leftarrow [dst]+[src]$

Performs the operation

MOV dst,src

$dst \leftarrow [src]$

One byte instructions

Registers can be incremented or decremented by instructions,that occupy one boe byte.examples are

INC EDI

And

DEC ECX

In which the general purpose register EDI and ECX are specified by 3-bit codes in the single OP-code byte.

Immediate mode encoding

The OP-code specifies when the immediate addressing mode is used. For example the instruction

MOV EAX,820

Is encoded into 5 bytes .a one –byte OP code specifies the move operation, the fact that a 32-bit immediate operand is used and the name of the destination register. The OP code byte is directly followed by the 4-byte immediate value of 820.when an 8-bit immediate operand is used, as in the instruction

MOV DL,5

Only two bytes are needed to encode the instruction.

Addressing mode and displacement fields

As a general; rule, one operand of a two-operand instruction must be in a register .the other operand can also be in a register, or it can be in the memory. There are two exceptions where both operands can be in the memory. The first is the case where the source operand is an immediate operand, and the destination operand is in the memory. The second is the case of instruction for Push and Pop operations on the processor stack. The stack is located in the stack segment of memory, and it is possible to push a memory operand onto the stack or to pop an operand from the stack into the memory.

When both operands are in registers, only one addressing mode byte is needed. For example ,the instruction

ADD EAX,EDX

is encoded into two bytes. The first byte contains the OP code and the other byte is an addressing mode byte that specifies the two registers. The instruction

MOV ECX,N

is encoded in 6 bytes: one for the OP code, one for the addressing mode byte that specifies both the Direct mode and the destination register ECX,and four bytes for the address of memory location N.

The instruction

ADD EAX,[EBX+EDI*4]

requires two addressing mode bytes because two registers are used to generate the effective address of the source operand. The scale factor of 4 is also included in the second of these two bytes. Thus, the instruction requires a total of 3 bytes, including the OP-code byte.

In the encoding of two-operand instructions, the specification of the register operand and the memory operand are placed in a fixed order, with the register operand always, being specified first. In order to distinguish between the instructions

MOV EAX,LOCATION

Which loads the contents of memory location LOCATION into register EAX ,and the instruction

MOV LOCATION,EAX

Which stores the contents of EAX into LOCATION ,the OP-code byte contains a bit called the direction bit. This bit indicates which operand is the source.

4. Write about IA-32 assembly language.

Basic aspects of the IA-32 assembly language for specifying OP code, addressing modes and instruction address labels are shown in the below program:

```
            LEA     EBX,NUM1            Load base register EBX and
            SUB     EBX,4                 adjust to hold NUM1–4.
            MOV     ECX,N               Initialize counter/index (ECX).
            MOV     EAX,0               Clear the accumulator (EAX).
STARTADD:   ADD     EAX,[EBX + ECX * 4] Add next number into EAX.
            LOOP    STARTADD            Decrement ECX and branch
                                          back if [ECX] > 0.
            MOV     SUM,EAX             Store sum in memory.
```

The assembler directives are needed to define the data area of a program and to define the correspondence between symbolic names for data locations and the actual physical address values. A complete assembly language program is shown below:

Assembler directives

```
.data
NUM1    DD    17, 3,-51, 242,-113
N       DD    5
SUM     DD    0

.code
```

Statements that generate machine instructions

```
MAIN :      LEA     EBX, NUM1
            SUB     EBX, 4
            MOV     ECX, N
            MOV     EAX, 0
STARTADD :  ADD     EAX, [EBX+ECX * 4]
            LOOP    STARTADD
            MOV     SUM, EAX
```

Assembler directive

```
            END     MAIN
```

The data and code assembler directives define the beginning og the data and code sections of the program .In the data section ,the DD directives allocate 4-byte doublewords initialized to the decimal values 17,3,-51,242 and -113.the next two doubleword locations ,initialized to 5 and 0 ,are given the address labels N and SUM.

The 3 symbolic names declared in the data section are used in the addressing modes of the instruction in the code section. The MAIN label is used to specify the location where instruction execution is to begin, and this label is used in the END assembler directive that terminates the text file for the program.

5. Write in detail about IA-32 program flow control

There are two main ways in which the flow of executing instructions varies from straight –line sequencing, calls to subroutines and returns from them break straight line sequencing. Also ,branch instructions, either conditional or unconditional, can cause a break. The branch instructions are called jumps.

Conditional jumps and condition code flags

The conditional Jump instructions test the four condition code flags in the status register. The instruction

JG LABEL

is an example of a conditional Jump instruction. The condition is *greater-than* as indicated by the G suffix in the OP code. The below table summarizes the conditional Jump instructions and the corresponding

combinations of the condition code flags that are tested. The Jump instructions that test the sign flag (SF) are used when the operands of a preceding arithmetic or comparison instruction are signed numbers. For example, the JG instruction tests for the greater-than condition when signed numbers are involved, and it considers the SF flag. When unsigned numbers are involved, the JA (jump-above) instruction tests for the greater than condition without considering the SF flag.

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF ∨ ZF = 0
JBE	Unsigned below or equal	CF ∨ ZF = 1
JGE	Signed greater than or equal	SF ⊕ OF = 0
JL	Signed less than	SF ⊕ OF = 1
JG	Signed greater than	ZF ∨ (SF ⊕ OF) = 0
JLE	Signed less than or equal	ZF ∨ (SF ⊕ OF) = 1

For example, program for adding numbers

```
          LEA     EBX,NUM1          Initialize base (EBX) and
          MOV     ECX,N               counter (ECX) registers.
          MOV     EAX,0             Clear accumulator (EAX)
          MOV     EDI,0               and index (EDI) registers.
STARTADD: ADD     EAX,[EBX + EDI *4] Add next number into EAX.
          INC     EDI               Increment index register.
          DEC     ECX               Decrement counter register.
          JG      STARTADD          Branch back if [ECX] > 0.
          MOV     SUM,EAX           Store sum in memory.
```

the instruction

JG STARTADD

is an example of jump instruction. The condition is “Greater than 0”is tested ,if it is true then the control transfers to STARTADD otherwise control transferred to the next statement which is coming after JG STARTADD.

Unconditional Jump Instruction

An unconditional Jump instruction, JMP, causes a branch to the instruction at the target address. In addition to using short (one-byte) or long (four-byte) relative signed offsets to determine the target address, as is done in conditional Jump instructions, the JMP instruction also allows the use of other addressing modes. This flexibility in generating the target address can be very useful. Consider the Case statement that is found in many high-level languages. It is used to perform one of a number of alternative computations at some point in a program. Each of these alternatives is referred to as a case. Suppose that for each case, a routine is defined to perform the corresponding computation. Suppose also that the 4-byte starting addresses of the routines are stored in a table in the memory, beginning at a location labeled JUMPTABLE. The cases are

numbered with indices 0, 1, 2, ... At execution time, the index of the selected case is loaded into index register ESI. A jump to the routine for the selected case is performed by executing the instruction

JMP [JUMPTABLE + ESI * 4]

which uses the Index with displacement addressing mode.

Compare instructions

It is often necessary to make conditional jumps in a program based on the results of comparing two numbers. The compare instruction

CMP dst,src

performs the operation

dst ← [src]

and sets the condition code flags based on the result obtained. Neither of the operands is changed .the first operand is always compared to the second. For example, the compare instruction by a conditional jump that is based on the "greater than" condition, then the jump will take to the target address if the destination operand is greater than the source operand.

6. Write in detail about logic and shift/rotate instructions

Logic operations:

The IA-32 architecture has instructions that perform the logic operations AND, OR, and XOR. The operation is performed bitwise on two operands, and the result is placed in the destination location.

For example, suppose register EAX contains the hexadecimal pattern 0000FFFF and register EBX contains the pattern 02FA62CA. The instruction

AND EBX, EAX

clears the left half of EBX to all zeroes, and leaves the right half unchanged. The result in EBX will be 000062CA.There is also a NOT instruction which generates the logical complement of all bits of the operand, that is, it changes all 1s to 0s and all 0s to 1s.

Shift instruction:

An operand can be shifted right or left ,using either logical or arithmetic shifts,by a number of bit positions determined by a specified count. The format of the shift instruction is

OPcode dst , count

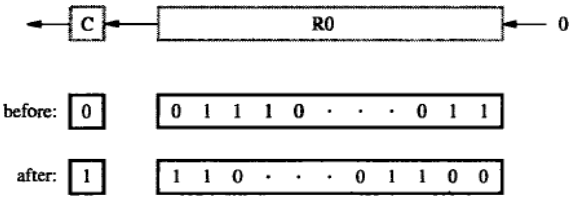
Where the destination operand to be shifted is specified by the general addressing modes and the count is given either as an 8-bit immediate value or is contained in the 8-bit register CL.

There are four shift instructions:

- SHL (Shift left logical)
- SHR (Shift right logical)
- SAL (Shift left arithmetic; operation is identical to SHL)
- SAR (Shift right arithmetic)

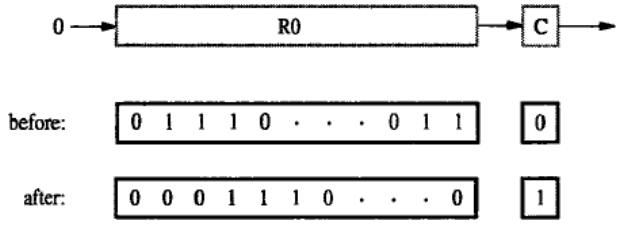
1. SHL (Shift left logical)

The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.For example, **SHL R0,#2**



2. SHR (Shift right logical)

The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero. For example, **SHR R0,#2**

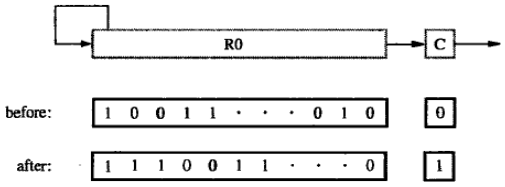


3. SAL (Shift left arithmetic)

The operation of SAL is identical to SHL.

4. SAR (Shift right arithmetic)

SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand. For example, **SAR R0,#2**



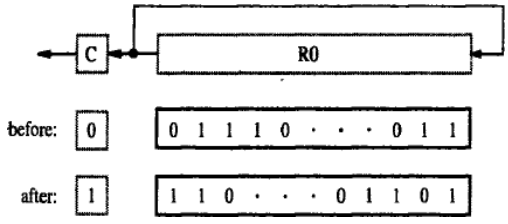
Rotate instruction:

In addition to the shift instructions, there are also four rotate instructions:

- ROL (Rotate left without the carry flag CF)
- ROR (Rotate right without the carry flag CF)
- RCL (Rotate left including the carry flag CF)
- RCR (Rotate right including the carry flag CF)

1. ROL (Rotate left without the carry flag CF)

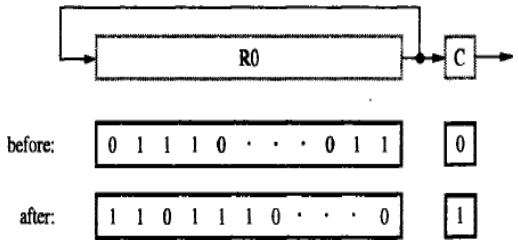
- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost
- For example , **ROL R0,#2**



2. ROR (Rotate right without the carry flag CF)

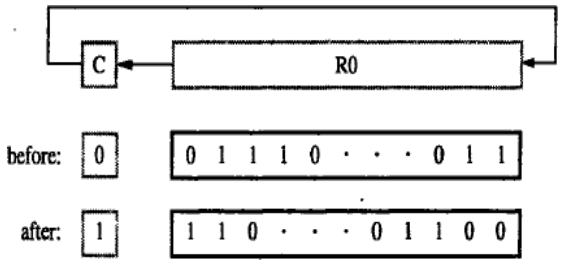
- ROR (rotate right) shifts each bit to the right

- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost
- For example , **ROR R0,#2**



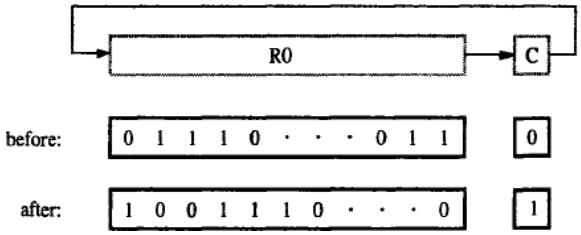
3. RCL (Rotate left including the carry flag CF)

- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag
- For example , **RCL R0,#2**



4. RCR (Rotate right including the carry flag CF)

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag
- For example , **ROR R0,#2**



The rotate instructions require the count argument to be either an 8-bit immediate value or the 8-bit contents of register CL.

7. Explain I/O Operation in IA-32

Memory mapped I/O

The IA-32 Move instruction can be used to transfer directives to I/O devices ,and to transfer data and status information to and from devices. For example, suppose that keyboard and display devices have their synchronization flags SIN and SOUT stored in bit 3 of device status registers INSTATUS and OUTSTATUS,

respectively. Using program controlled I/O; a byte can be read from the keyboard buffer register DATAIN into register AL using the wait loop

```
READWAIT:   BT INSTATUS,3
            INC READWAIT
            MOV ALDATAIN
```

The instruction BT is a bit-test instruction .the value in the destination bit position specified by the source operand is loaded into the carry flag CF .the conditional jump JNC causes a jump to READWAIT if CF = 0. Similarly, an output operation to send a byte from register AL to the display buffer register DATAOUT is performed by

```
WRITEWAIT:  BT OUTSTATUS,3
            INC WRITEWAIT
            MOV DATAOUT,AL
```

An IA-32 program that reads one line of character from a keyboard, stores them in memory starting at address LOC ,and echo's them back out to the display shown below

```
        LEA   EBP,LOC       EBP points to memory area.
READ:   BT    INSTATUS,3    Wait for character to be
        JNC   READ            entered into DATAIN.
        MOV   AL,DATAIN     Transfer character into AL.
        MOV   [EBP],AL      Store the character in memory
        INC   EBP             and increment pointer.
ECHO:   BT    OUTSTATUS,3   Wait for display to
        JNC   ECHO            be ready.
        MOV   DATAOUT,AL    Send character to display.
        CMP   AL,CR         If not carriage return,
        JNE   READ            read more characters.
```

Isolated I/O

The IA-32 instructions set also have two instructions, with OP codes IN and OUT, that are used only for I/O purposes. The addresses issued by these instructions are in an address space that is separate from the memory address space used by the other instructions. This arrangement is called isolated I/O to distinguish it from memory –mapped I/O in which the addressable locations in I/O devices are in the same address space as memory locations. The same address and data lines on Intel processor chips are used for both address spaces. An output control line is used to indicate which address space is reference by the current instruction.

The 16-bit addresses are used in the byte-addressable I/O address space. There are 8-bit and 32-bit I/O device operand location that hold data ,status, and command information. The first 256 addresses can be specified directly using an 8-bit field in the In and Out instructions. The format for the input instruction using this mode is

IN REG ,DEVADDR

Where the destination REG must be register AL or EAX,denoting an 8-bit or a 32-bit operand transfer respectively. The last field in the instruction contains the 8-bit device address DEVADDR.the corresponding output instruction is

OUT DEVADDR,REG

Since the address space is byte addressable, a keyboard device that can send an 8-bit ASCII character to the processor could have its data buffer register at byte address DEVADDR and its 8-bit starts register at address

DEVADDR + 1

The full 16-bit I/O address spans 64K locations; it can be referenced through the DX register using the input instruction

IN REG,DX

Where, as before,REG must be AL or EAX.The 16-bit device address is contained in the DX register ,which is low order 16 bits of the EDX register, and the width of the data transfer is determined by the size of the REG operand. The corresponding output instruction os

OUT DX,REG

Block transfers

In addition to the instruction IN and OUT that transfer a single item of information between the processor and an I/O device ,the IA-32 architecture also has two block transfer I/O instructions REPINS and REPOUTS .they transfer a block lf data serially, one item at a time, between the memory and an I/O device. The S suffix in the OP codes stands for string, and the REP prefix stands for "repeat the item by item transfer until the complex block has been transferred". The instructions themselves do not specify the parameters needed to describe the transfer. These parameters are specified implicitly by processor registers DX,EDI and ECX as follows:

DX contains a 16-bit I/O device address

EDI contains a 32-bit address for the beginning of a block in memory

ECX contains the number of data items to be transferred.

A suffix B or D in the OP-code mnemonic indicates that the item size is either of byte or doubleword length. Thus REPINSB is a byte –block transfer, and REPINSD is a doubleword –block transfer.

The block transfer instruction operates as follows :After each data item is transferred ,the index register EDI is incremented by 1 or 4 depending on the size of the data items, and the ECX register is decremented by 1.The transfers are repeated until the contents of the counter register ECX have been decremented to 0.The effect of these single instruction is equivalent to a program loop that uses register ECX as the loop counter.

As an example, suppose that a block lf 128 doublewords is to be transferred from a disk storage device into the memory. Assume that the addressable data buffer register in the disk device contains a doubleword data item, and has the I/O address MEMBLOCK.The counter register ECX has to be initialized to 128.The instruction sequence

```
LEA     EDI,MEMBLOCK
MOV     DX,DISKDATA
MOV     ECX,128
REPINSD
```

can be used to accomplish the transfer. This assumes that MEMBLOCK has been defined as an address label, and DISKDATA has been defined by an EQU assembler directive to represent the 16-bit address of the device data buffer register.

8. Write in detail about Subroutines in IA-32Subroutines:

In the IA-32 architecture, register ESP is used as the stack pointer. It points to the current top element (TOS) in the processor stack. The stack grows toward lower numbered addresses. The width of the stack is 32 bits, that is, all stack entries are doublewords.

There are two instructions for pushing and popping individual elements onto and off the stack. The instruction

PUSH src

decrements ESP by 4, and then stores the doubleword at location src into the memory location pointed to by ESP. The instruction

POP dst

reverses this process by retrieving the TOS doubleword from the location pointed to by ESP,storing it at location dst, and then incrementing ESP by 4. These instructions implicitly use ESP as the stack pointer. The source and destination operands are specified using the IA-32 addressing modes.

There are also two more instructions that push or pop the contents of multiple registers. The instruction

PUSHAD

pushes the contents of all eight general-purpose registers EAX through EDI onto the stack,and the instruction

POPAD

pops them off in the reverse order. When POPAD reaches the old stored value of ESP, it discards those four bytes without loading them into ESP and continues to pop the remaining values into their respective registers. These two instructions are used to efficiently save and restore the contents of all registers as part of implementing subroutines.

For example, The list-addition program shown below

```
            LEA     EBX, NUM1              Use EBX as base register.
            MOV     ECX, N                 Use ECX as counter register.
            MOV     EAX, 0                 Use EAX as accumulator register.
            MOV     EDI, 0                 Use EDI as index register.
STARTADD:   ADD     EAX, [EBX + EDI * 4]   Add next number into EAX.
            INC     EDI                    Increment index register.
            DEC     ECX                    Decrement counter register.
            JG      STARTADD               Branch back if [ECX] > 0.
            MOV     SUM, EAX               Store sum in memory.
```

can be written as a subroutine as shown below:

Calling program

```
            ⋮
            LEA     EBX,NUM1        Load parameters
            MOV     ECX,N             into EBX,ECX.
            CALL    LISTADD         Branch to subroutine.
            MOV     SUM,EAX         Store sum into memory.
            ⋮
```

Subroutine

```
LISTADD:    PUSH    EDI             Save EDI.
            MOV     EDI,0           Use EDI as index register.
            MOV     EAX,0           Use EAX as accumulator register.

STARTADD:   ADD     EAX, [EBX + EDI * 4]   Add next number.
            INC     EDI             Increment index.
            DEC     ECX             Decrement counter.
            JG      STARTADD        Branch back if [ECX] > 0.
            POP     EDI             Restore EDI.
            RET                     Branch back to Calling program.
```

(a) Calling program and subroutine

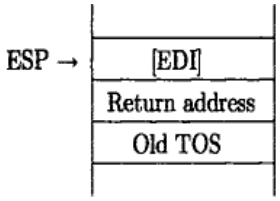
Parameters are passed through registers. Memory address NUM1 of the first number in the list is loaded into register EBX by the calling program.

The number of entries in the list, contained in memory location N, is loaded into register ECX. The calling program expects to get the final sum passed back to it in register EAX. Thus, registers EBX,ECX, and EAX are used for passing parameters.

Register EDI is used by the subroutine as an index register in performing the addition, so its contents have to be saved and restored in the subroutine by PUSH and POP instructions.
The subroutine is called by the instruction

CALL LISTADD

which first pushes the return address onto the stack and then jumps to LISTADD. The return address is the address of the MOV instruction that immediately follows the CALL instruction. The subroutine saves the contents of register EDI on the stack.



(b) Stack contents after saving EDI in subroutine

The above figure shows the stack contents at this point. After executing the loop, the saved contents of register EDI are restored. The instruction RET returns execution control to the calling program by popping the TOS element into the Instruction Pointer (register EIP).

9. Write about the other instructions that are available in IA-32

The other instructions available in IA-32 are:

5. Multiplication instruction:
6. Division instruction:
7. Multimedia Extension (MMX) instructions
8. Vector (SIMD) Floating-Point Operations

1. Multiplication instruction:

The signed integer multiplication instruction, IMUL, performs 32-bit multiplication. Depending on the form of the instruction that is used, the destination may be implicit and the 64-bit product may be truncated to 32 bits.

One form of this instruction is

IMUL src

which implicitly uses the EAX register as the multiplicand. The multiplier specified by src can be in a register or in the memory. The full 64-bit product is placed in registers EDX (high-order half) and EAX (low-order half).

A second form of this instruction is

IMUL REG, src

The destination operand, REG, must be one of the eight general-purpose registers. The source operand can be in a register or in the memory. The product is truncated to 32 bits before it is placed in the destination register REG.

For both forms, the CF and OF flags are set if there are any 1s (including sign bits) in the high-order half of the 64-bit product. Otherwise, the CF and OF flags are cleared. The other flags are undefined.

2. Division instruction:

The integer divide instruction, IDIV, operates on a 64-bit dividend and a 32-bit divisor to generate a 32-bit quotient and a 32-bit remainder. The format of the instruction is

IDIV src

The source operand is the divisor. The 64-bit dividend is formed by the contents of register EDX (high-order half) and register EAX (low-order half). After performing the division, the quotient is placed in EAX and the remainder is placed in EDX.

All of the condition code flags are undefined. Division by zero causes an exception. If the dividend value is represented by 32 bits, it must first be placed in EAX, and then sign-extended to the required 64-bit operand size in registers EAX and EDX. This is done by the instruction CDQ (convert doubleword to uadword), which has no operands because the source and destination are implicitly registers EAX and EDX, respectively

3. Multimedia Extension (MMX) instructions

A two-dimensional graphic or video image can be represented by a large array of sampled image points, called *pixels*. The color and brightness of each point can be encoded into an 8-bit data item. Processing of such data has two main characteristics.

The first is that manipulations of individual pixels often involve very simple arithmetic or logic operations.

The second is that very high computational performance is needed for some real-time display applications. The same characteristics apply to sampled audio signals or speech processing, where a sequence of signed numbers represents samples of a continuous analog
signal taken at periodic intervals.

In such applications, processing efficiency is achieved if the individual data items, which are usually bytes or 16-bit words, are packed into small groups whose elements can
be processed in parallel.

The IA-32 instruction set has a number of instructions that operate in parallel on such data packed into 64-bit quadwords.(A quadword contains 8 bytes or four 16-bit words).These instructions are called *multimedia extension* (MMX) instructions.

The operands for MMX instructions can be in the memory, or in the eight floating-point registers. Thus, these registers serve a dual purpose. They can hold either floating-point numbers or MMX operands. When used by MMX instructions, the registers are referred to as MM0 through MM7.

The MOVE instruction is provided for transferring 64-bit quadword operands between the memory and the MMX registers. The instruction

PADDB MMi,src

Adds the corresponding bytes of two 8-byte operands individually and places eight sums in the destination register. The source can be in the memory or in an MMX register but the destination must be an MMX register. The instruction

PADDB MM2, [EBX]

adds eight corresponding bytes of the quadwords in register MM2 and in the memory location pointed to by register EBX. The eight sums are computed in parallel. The results are placed in register MM2.

4. Vector (SIMD) Floating-Point Operations

A set of instructions that are used to perform arithmetic operations on small group of floating point numbers is provided.SIMD (single-instruction-multiple-data) instructions are useful for vector and matrix calculations in scientific applications.

In Intel terminology, these instructions are called *streaming SIMD extension* (SSE) instructions. They handle packed 128-bit double quadwords, each consisting of four 32-bit floating-point numbers. Eight additional 128-bit registers, XMM0 to XMM7, are available for holding these operands.

Add and multiply are two of the basic instructions are provided in this group .they operate on the four corresponding pairs of 32-bit operands in the 128-bit compound source and destination operands and place the four individual results in the 128-bit destination location.