TA - 1

UNIT-1

Q-1]

Instruction set architecture [ISA] is a crucial aspect of computer architecture. They are a set of rules & conventions that define the programming model & functionality of a computer's central processing unit (CPU). It includes set of instructions that a CPU can execute, the data types it supports, the addressing modes etc. ISA serves as an interface betn hardware & software. ISA can be categorized into :1) Zero address (stack) ISA : Instructns have no explicit operands they operate on values stored on a stack or accumulator.

2) One address ISA : instructns take one explicit operand, & other operand is implicitly accumulator or stack top.

3) Two address ISA : Instructns take 2 explicit operands, which can be registers or memory locatn and store result in one of the source operands

4) Three address ISA : Instructns take 3 explicit operands and result is stored in a separate destinatn operand.

- TWO ADDRESS ~~REGISTER~~ format:

```
MOV R1, A        ; R1 ← M[A]
MUL R1, B        ; R1 ← R1 * M[B]
DIV R1, E        ; R1 ← R1 / M[E]
MOV R2, C        ; R2 ← M[C]
MUL R2, E        ; R2 ← R2 * M[E]
ADD R1, R2       ; R1 ← R1 + R2
MOV X, R1        ; M[X] ← R1
```

Date
Page

- ONE ADDRESS format:

```
LOAD A      ; AC ← M[A]
MUL B       ; AC ← AC * M[B]
DIV E       ; AC ← AC / E
STORE T     ; M[T] ← AC
LOAD C      ; AC ← M[C]
MUL E       ; AC ← AC * M[E]
ADD T       ; AC ← AC + M[T]
STORE X     ; M[X] ← AC
```

- ZERO ADDRESS format: AB*E/CE*+

```
PUSH A      ; TOS ← A
PUSH B      ; TOS ← B
MUL         ; TOS ← A * B
PUSH E      ; TOS ← E
DIV         ; TOS ← A*B / E
PUSH C      ; TOS ← C
PUSH E      ; TOS ← E
MUL         ; TOS ← C * E
ADD         ; TOS ← (A*B/E) + (C*E)
POP X       ; M[X] ← TOS
```

Q-2]

Ans→ In a zero address format, the instructions do not explicitly specify operands. Instead they operate on values implicitly stored in a stack or a memory location. It is particularly used in stack-based or accumulator based computer systems. In zero address instruction format the instructns act directly on the data at the top of the stack without needing to specify registers or addresses explicitly. This can simplify instruction set but it requires a well managed stack to keep track of operands.

Example: To add A and B, and POP the result.

CODE:

```
PUSH A    ; TOS ← A
PUSH B    ; TOS ← B
ADD       ; TOS ← A+B
POP X     ; M[X] ← TOS
```

Q-3]

Ans→ A stack is commonly used in subroutine processing to manage function calls and return addresses. Here below is an illustration of how a stack is used in this content:

1] FUNCTION CALL: When a program calls a subroutine (function), it pushes the return address onto the stack. This return address is the memory address of the instruction to execute after the subroutine completes.

2] LOCAL VARIABLES: Any local variables and parameters of the subroutine are also stored on the stack. These variables are pushed onto stack when the subroutine is called and popped off when subroutine exits.

3] EXECUTION: The subroutine executes its instructns, which can include calling other subroutines. Each new subroutine call pushes its return address and local variables onto stack.

4] RETURNING: When a subroutine is finished, it pops its local variables and the return address from stack. This return address is used to continue execution at the correct point in the calling fn.

5] NESTED CALLS: If there are nested subroutines [subroutines calling other subroutines], stack ensures that each return address is correctly stored and retrieved, allowing for proper program flow.

6] STACK POINTER: A stack ptr is used to keep track of the top of the stack. It moves up and down as items are pushed and popped from the stack.

In summary, the stack is a fundamental data structure for managing subroutine calls and return addresses, ensuring that the program's flow is maintained as function are called and return.

Q4]

Ans→ Addressing modes are the techniques used in computer architecture to specify the location of operands in memory or registers when executing instructions. They play a crucial role in how a CPU interacts with memory and data. The various addressing modes are:

1) IMMEDIATE ADDRESSING MODE:

In this, the operand is a constant value that is part of the instruction itself.

ex: MOV R1, #05H

↓ indicates operand is data itself

2) DIRECT ADDRESSING MODE:

The instruction contains the memory address of the operand. The registers are given directly.

ex: MOV R1, R2 or MOV R1, 3000H

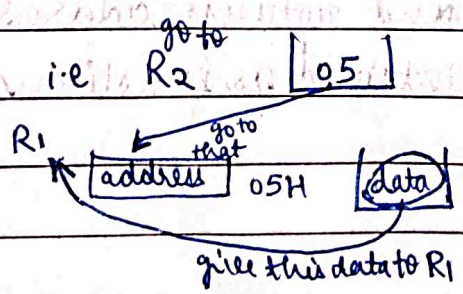
This mode is simple but less flexible because it uses a fixed memory location.

3) REGISTER INDIRECT ADDRESSING MODE:

In this addressing mode, a register contains the memory address where the operand is stored. This mode allows more flexible & dynamic memory access.

ex: MOV R1, [R2]

Example→ loads the value stored at memory address stored in the R2 register into the R1 register.



4] IMPLIED ADDRESSING MODE:

It is the simplest mode in assembly language programming. It is used for instructions where the operand is either predefined or determined by the content, and there is no need to specify any explicit operand within the instruction.

→ no operand is given directly in instruction.

ex: STA
STC
RLC

5] INDEXED ADRESSING MODE:

In this an index/offset is added to a base address to calculate the effective address.

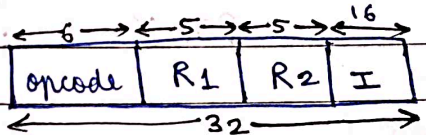
→ used for error finding

ex: MOV R_1, [BL+03]

(8 bit register, indexed Pointer)

Numericals:

Q-1] Given: 1] instruction length = 32 bit
2) no of instructns = 40
3) no of registers = 24



no of bits for opcode can be found out by analyzing no of instructns ; $2^? = 40 \longrightarrow 2^6 = 64 > 40$

$\therefore$ 6 bits

no of bits for registers can be found out by analyzing no of registers ; $2^? = 24 \longrightarrow 2^5 = 32 > 24$

$\therefore$ 5 bits

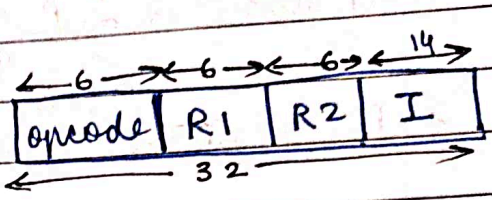
no of bits for immediate = 32 − (6 + 10)

= 32 − 16

= 16 bits → Ans.

$\therefore$ No of bits available for immediate operand field is 16 bits.

2] given: 1) instructn length = 32 bits
2) no of instructns = 45
3) no of registers = 64



no of bits for opcode can be found out by analyzing no of instructions; $2^? = 45 \rightarrow 2^6 = 64 > 45$

$\therefore$ 6 bits

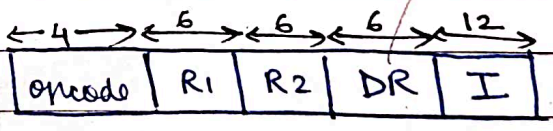
no of bits for ~~opcode~~ registers can be found out by analyzing no of registers; $2^? = 64 \rightarrow 2^6 = 64$

$\therefore$ 6 bits

no of bits for immediate = 32 − (6 + 12)
= 14 bits

max^m value of unsigned immediate operand = $2^{14} - 1$ $\{2^n - 1\}$
= 16384 − 1
= 16383 → Ans.

3] number of registers = 64
Number of bits to address register = 6
no of instructions = 12
no of bits for opcode : $2^? = 12 \rightarrow 2^4 = 16 > 12$ $\therefore$ 4 bits
no of bits for registers: $2^? = 64 \rightarrow 2^6 = 64$ $\therefore$ 6 bits
no of bits for immediate = 12 bits



Total bits = 4 + 6 + 6 + 6 + 12 = 34 bits

In terms of bytes = $\frac{34}{8}$ = 4.25 bytes

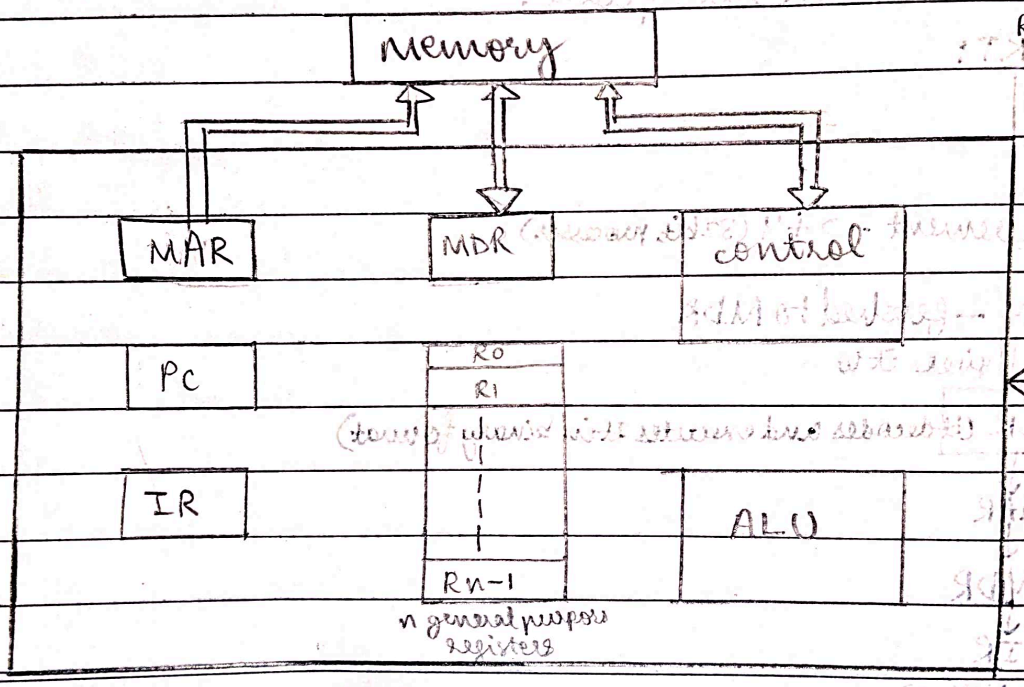
Here, given that each instructn must be stored in byte aligned manner
so we will take 5 bytes instead of 4.25 bytes
Total instructns = 100
Total size = 100 × 5 = 500 bytes → Ans.

500 bytes of memory consumed by the program text.

MOV L, A
HLT
END

UNIT-2

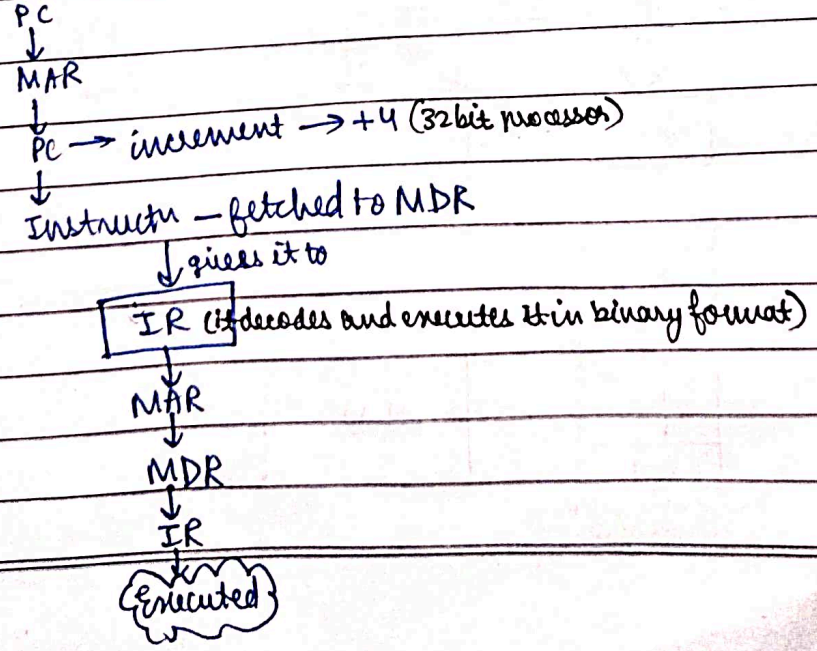
Q-1]



IR: instruction register
PC: program counter
R0-Rn-1: General purpose registers
MAR: memory address register
MDR: memory data register

- The PC (Program counter) contains the memory address of the instruction to be executed. During execution, the contents of the PC are updated to point to the next instruction. Every time that an instruction is to be executed, the program counter releases its contents to the internal bus and sends it to the memory address register.
- The MAR (Memory Address Register) holds the address of the location to or from which data are to be transferred. As can be seen from the figure above, the connection of the MAR to the main memory is one-way or unidirectional.
- The MDR (memory Data Register) contains the data to be written or read out of the addressed location.
- During the fetch operation, the MDR contains the instruction to be executed or data needed during execution. In write operation MDR the data to be written into the main memory.
- The IR (Instruction Register) contains the instruction that is being executed. Before the IR executes the instruction it needs to be decoded first. As soon as the content of MDR is transferred to IR, the decoding process commences. After decoding, execution of the instruction will take place.

FLOWCHART:



Explanatn → 2) Example:

#1) Assume that the instructn is stored in memory locatn 1000, the initial value of R1 is 50, and LOCA is 5000.

2) Before the instructn is executed, PC contains 1000.

3) Content of PC is transferred to MAR

4) ~~Read~~. READ request is issued to memory unit.

5] The instructn is fetched to MDR.

6] Content of MDR is transferred to IR

7] PC is incremented to point to the next instructn

8] The instructn is decoded by the control unit.

MAR ← PC
MDR ← Mem [MAR]
IR ← MDR
PC ← PC + 4

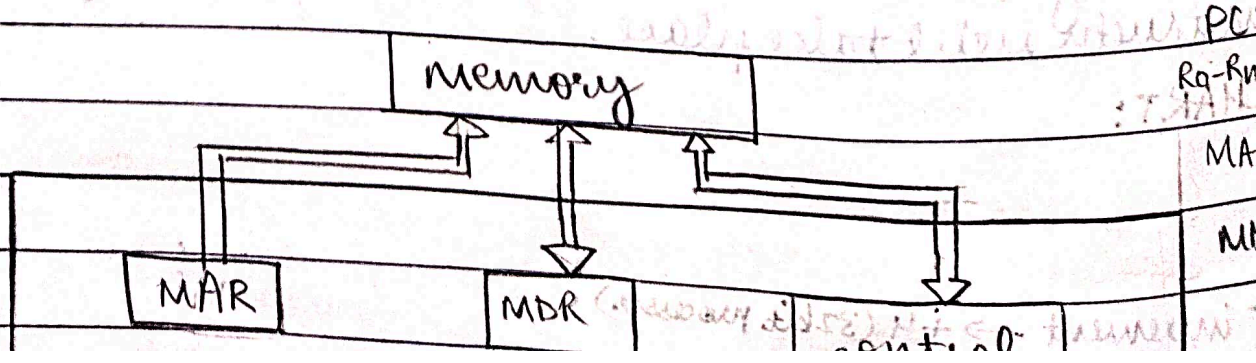
ADD R1	5000

Q-2]

```
LXI  H, 1500H
MVI  B, 00
MOV  C, B
MOV  A, 00H
LOOP:
     LOA X  H
     ADD  A
     INX  H
     DCR  B
     JNX  LOOP
     MOV B, C
     CPI  00H
     JZ  DONE
     XCHG
     DIV  B
     MOV  E, A
     MVI A, 00H
     MOV C, A
     DAD  H
     MOV  L, A
     HLT
     END
```

UNIT-2

Q1]



Q-3] Design techniques for control units in CPU's include:

1] Hardwired control:
Fixed combinational logic for simple instruction sets. The control signals are generated using a network of logic gates and flip-flops.

2] MICROPROGRAMMING:
Microinstructions for complex instruction sets, providing flexibility and modifiability. Each microinstructn corresponds to a single or small grp of related control signals.

3] CONTROL STORE:
Microinstructions stored in a control memory, common in microprogrammed control units

4] FINITE STATE MACHINE (FSM):
State based control for simple architecture.

5] PIPELINE CONTROL:
Managing instructn flow in pipelined cpu's.

6] SUPERSCALAR CONTROL:
Handling parallel execution in superscalar architecture

7] BRANCH PREDICTION:
Predicting conditional branches to reduce stalls.

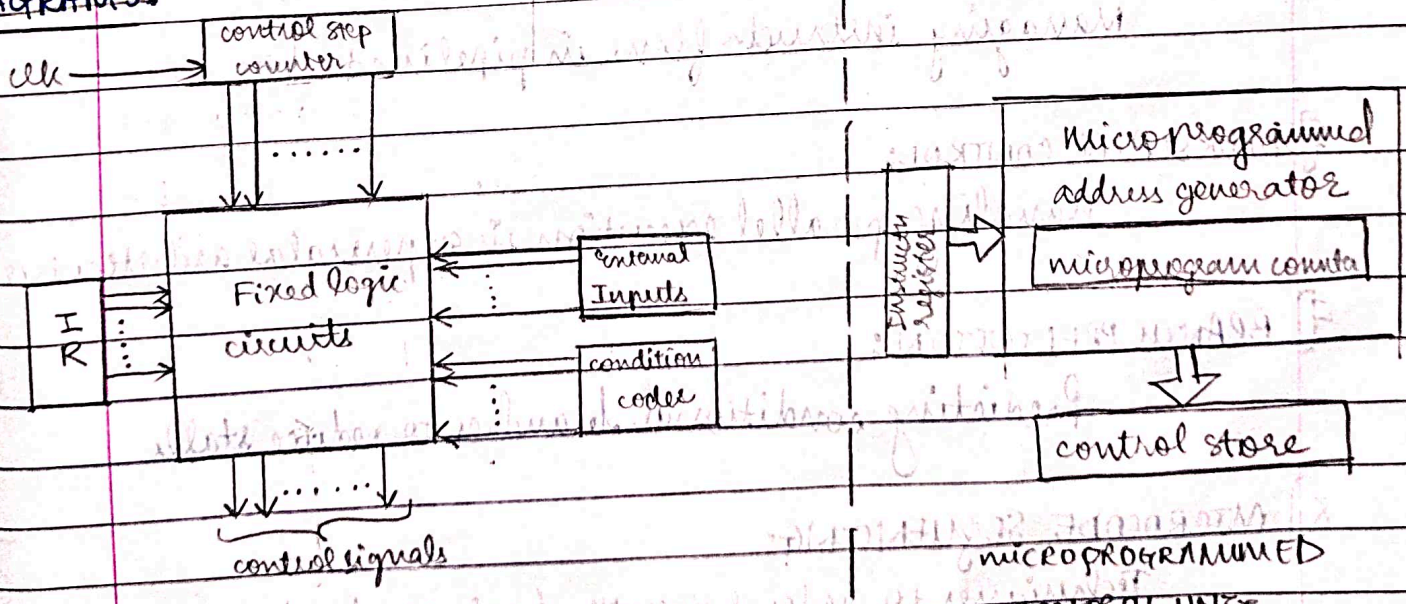
8] MICROCODE SEQUENCING:
Techniques to order execustn of microinstructns in microprogrammed control units.

Date
Page

Difference betⁿ Hardwired & Microprogrammed control unit

Attributes	Hardwired cu	Microprogrammed cu
1] speed	speed is fast	speed is slow
2] cost of implementatⁿ	more costly	cheaper
3] flexibility	not flexible to accomodate new sys specificatⁿ or new instⁿ redesign is reqd.	more flexible to accomodate new sys specificatⁿ or new instⁿ sets.
4] ability to handle complex instructn	difficult to handle complex IS.	Easier to handle complex IS.
5] decoding	complex decoding and sequencing logic.	Easier to decoding & sequencing logic
6] Applications	RISC microprocessor	CISC microprocessor.
7] Instructn set of size.	small	large
8] control memory	Absent	Present
9] occurence	occurence of error is more	occurence of error is less.

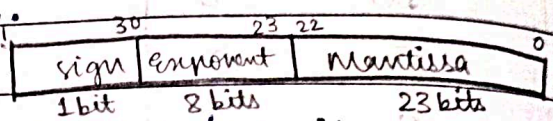
DIAGRAMS:



Unit-3

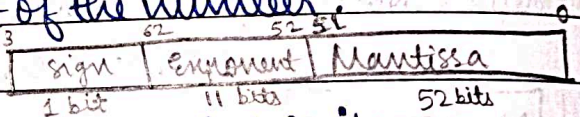
Q-1] Floating point numbers are used to present real numbers in a binary format in computers. The IEEE 754 standard is widely used for representing floating point numbers in single and double precision.

SINGLE PRECISION: [32 bit]



→ a floating pt no. is represented using 32 bits in it.
→ It has 3 parts: sign bit [1 bit], exponent [8 bits], mantissa [23 bits]
→ Sign indicates the sign of the number (+ve/-ve)
→ exponent part represents exponent of no in a biased form.
→ mantissa holds the fractional part of the number.

DOUBLE PRECISION: [64 bit]



→ a floating pt no is represented using 64 bits in it.
→ It has 3 parts: sign bit [1 bit], exponent [11 bits], mantissa [52 bits]
→ similar to single precision, sign bit represents sign of no, exponent part expresses biased exponent.
→ the mantissa is larger, allowing for greater precision.

NORMALIZED VALUE:

→ it is the value in floating pt representation in which the most significant bit [leftmost bit] of mantissa is always set to 1. This bit is not explicitly stored in representation but is implied. It ensures that floating pt representation of a no is as accurate as possible for given no of bits.

NEED OF NORMALIZATION:

1) Increased precision: normalization maximizes precision of representation by ensuring most significant bit is always 1.
2) consistency: It provides consistency to representation.
3) Efficiency: Normalized value leads to more efficient arithmetic operations.

P.T.O

- $(20.15)_{10}$

Step 1: Decimal to binary conversion:

$20 \div 2$

10	0
5	0
2	1
1	0
0	1

$0.15 \times 2 = 0.3$

$0.3 \times 2 = 0.6$

$0.6 \times 2 = 1.2$

$0.2 \times 2 = 0.4$

0 0 1 0

$(20.15)_{10} = (10100.0010)_2$

Step 2: Normalize the number

1.01000010×2^4 (sign bit)

$N = 01000010$

1) single precision format:

$e - 127 = 4$

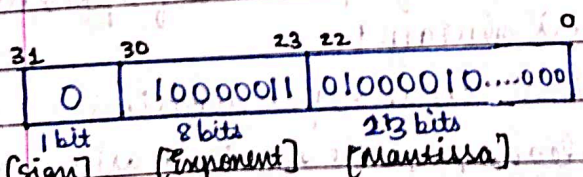
$e = 131$

Convert 131 in binary format:

$131 \div 2$

65	1
32	1
16	0
8	0
4	0
2	0
1	0
0	1

$(10000011)_2$



2) Double precision format:

$e - 1023 = 4$

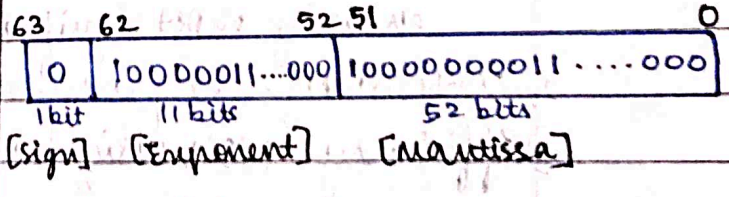
$e = 1027$

Convert 1027 in binary:

$1027 \div 2$

513	1
256	1
128	0
64	0
32	0
16	0
8	0
4	0
2	0
1	0
0	1

$(10000000011)_2$



Q-2](a) MANTISSA:

The mantissa, also known as the function, is a component of a floating point numbers in IEEE format 754. It represents fractional part of the number. In IEEE 754 std, the mantissa is normalized which means that the MSB is always assumed to be as 1 and not explicitly stored in representation.

ex: in single precession → mantissa is represented using 23 bits

in double precission → mantissa is represented using 52 bits to represent fractional part of no.

(b) Excess-127:

1) It is a bais or offset used in IEEE 754 floating pt representation.
2) It is used to represent exponent of floating pt no in a way that allows both +ve and -ve exponents to be stored.
3) The bais value of single precession IEEE 754 representation is 127.
4) And for ~~single~~ double precession it is 1023.

single: $[1.N] * 2^{e-127}$

Double: $[1.N] * 2^{e-1023}$

- $(-236.87)_{10}$

Step 1: convert in binary:

236 ÷ 2	
118	0
59	0
29	1
14	1
7	0
3	1
1	1
0	

$0.87 \times 2 = 1.74$	1
$0.74 \times 2 = 1.48$	1
$0.48 \times 2 = 0.96$	0
$0.96 \times 2 = 1.92$	1

$(236.87)_{10} = (11101100.1101)_2$

Step 2: Normalization

sign bit

$$1.11011001101 \times 2^{7}$$

$$N = 11011001101$$

1) Single Precession format:

$e - 127 = 7$

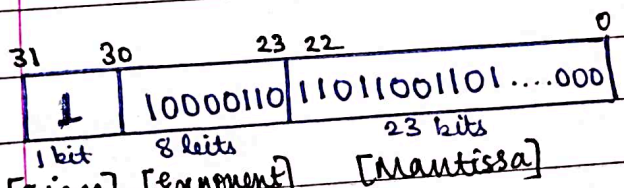
$e = 134$

convert 134 in binary:

$134 \div 2$

67	0
33	1
16	1
8	0
4	0
2	0
1	0
0	1

$(10000110)_2$



2) Double Precession format:

$e - 1023 = 7$

$e = 1030$

convert 1030 in binary:

$1030 \div 2$

515	0
257	1
128	1
64	0
32	0
16	0
8	0
4	0
2	0
1	0
0	1

$(10000000110)_2$