Computer Architecture (CS-213)

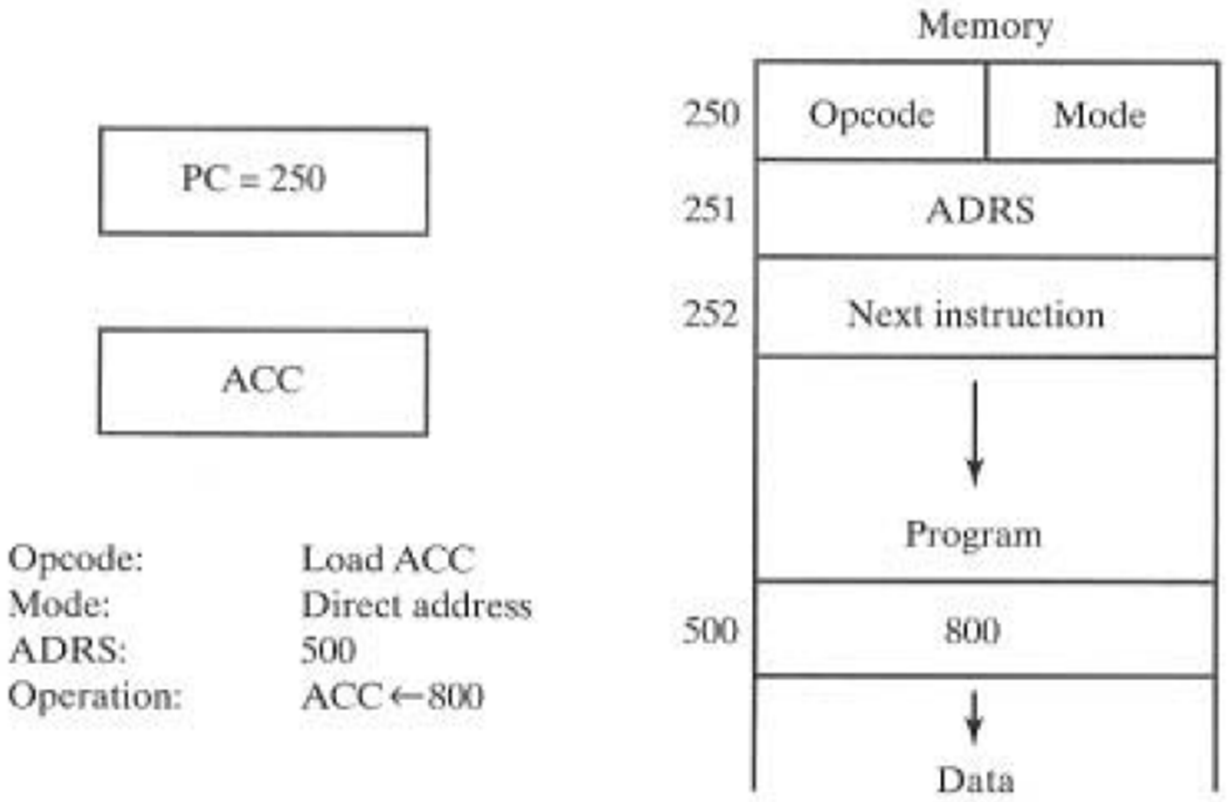
Lecture -7

Outline

- Addressing Modes
  - Indirect addressing mode
  - Relative addressing mode
  - Indexed addressing mode
  - Summary of Addressing modes
- CISC vs RISC architecture
- Data Transfer instructions
- Data Manipulation Instructions
  - Floating Point Computations

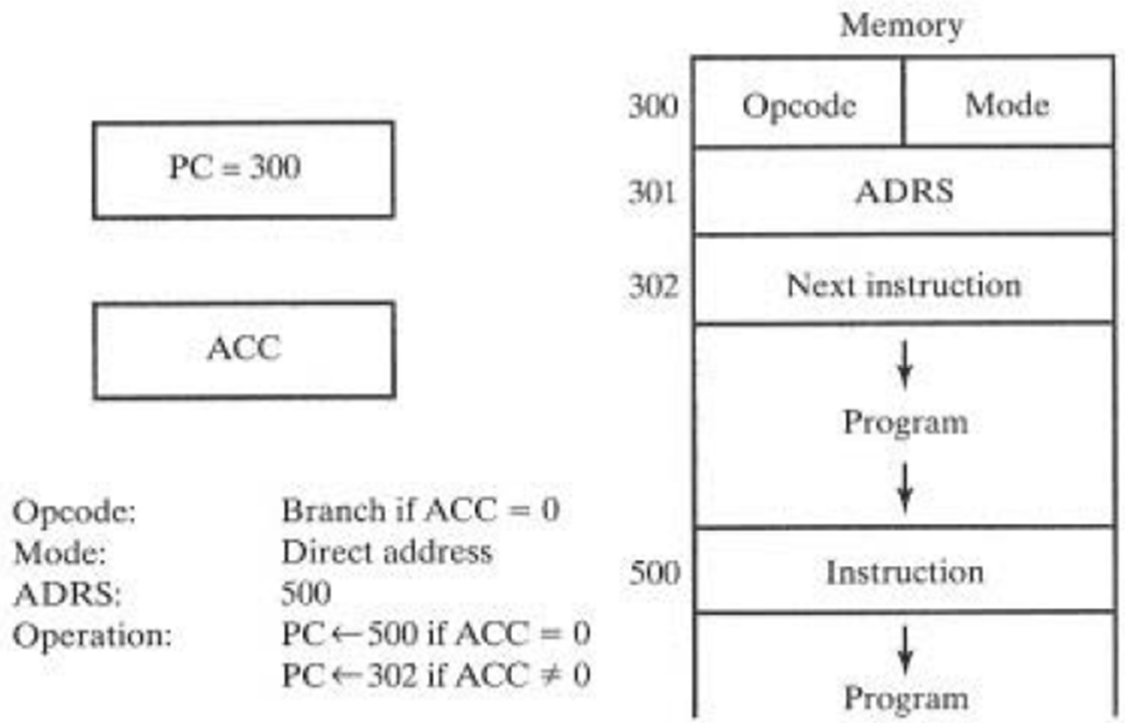
Direct Addressing Mode

- In direct addressing mode the address field directly specify the address of the memory where operand is stored.



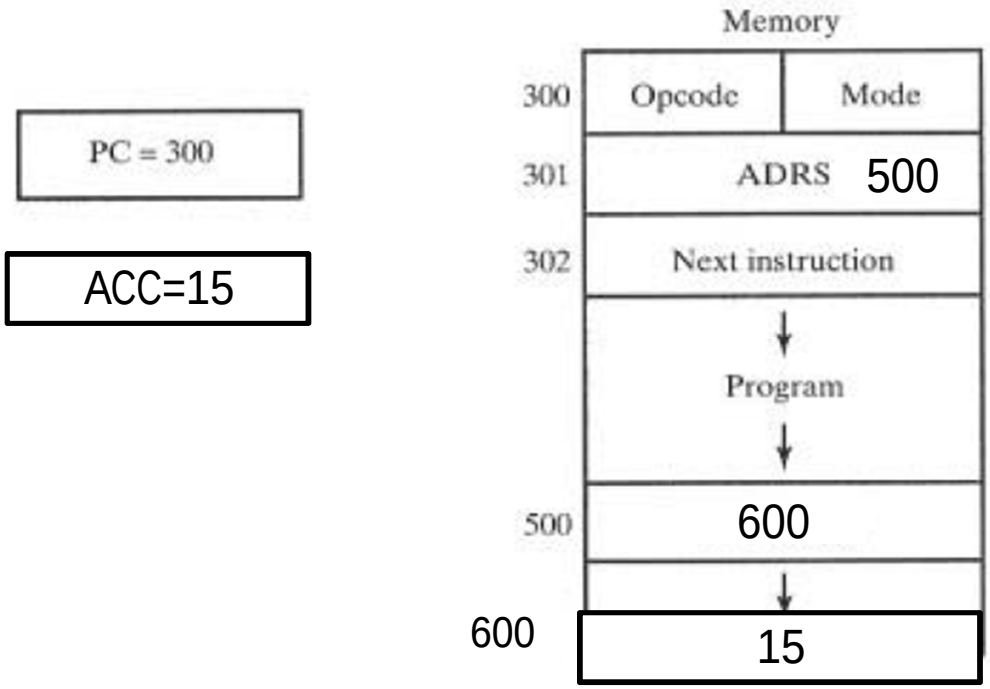
Direct Addressing Mode

- Following example illustrates a branch type instruction



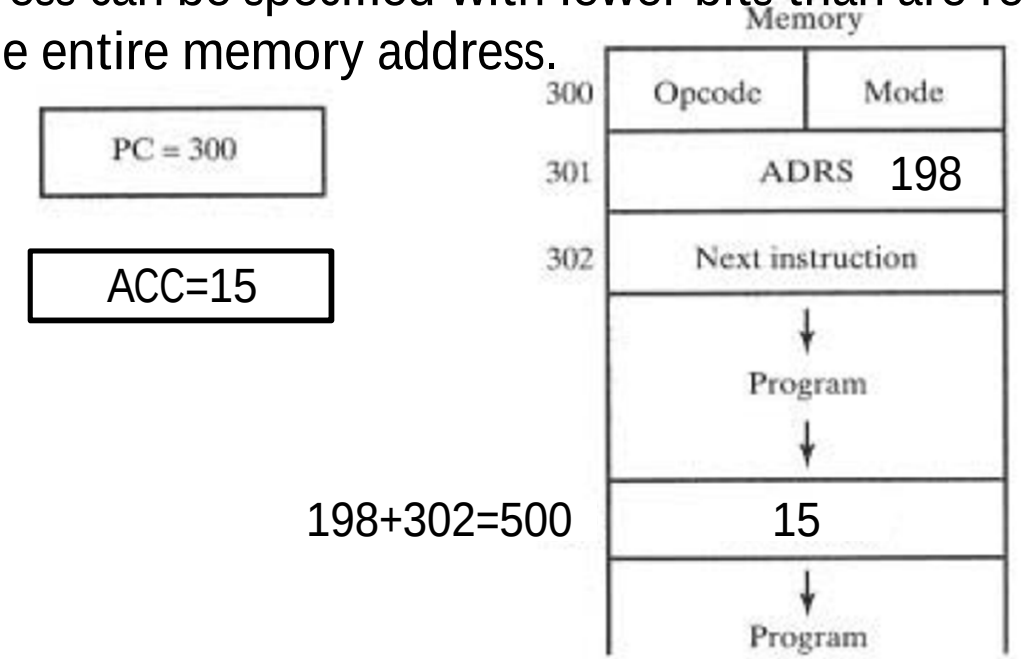
Indirect Addressing Mode

- In the indirect addressing mode, the address field of the instruction gives the address at which the effective address is stored in memory.
- The control unit fetches the instruction from memory and uses the address part to access memory again in order to read the effective address. Then accesses the operand placed at effective address.



Relative Addressing Mode

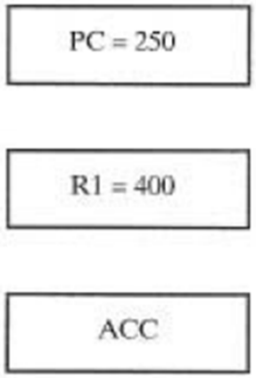
- In relative addressing mode, address is calculated as follows
- **Effective address = Address part of the instruction + Contents of *PC***
- Relative addressing is often used in branch-type instructions when the branch address is in a location close to the instruction word.
- Relative addressing produces, more compact instructions, since the relative address can be specified with fewer bits than are required to designate the entire memory address.

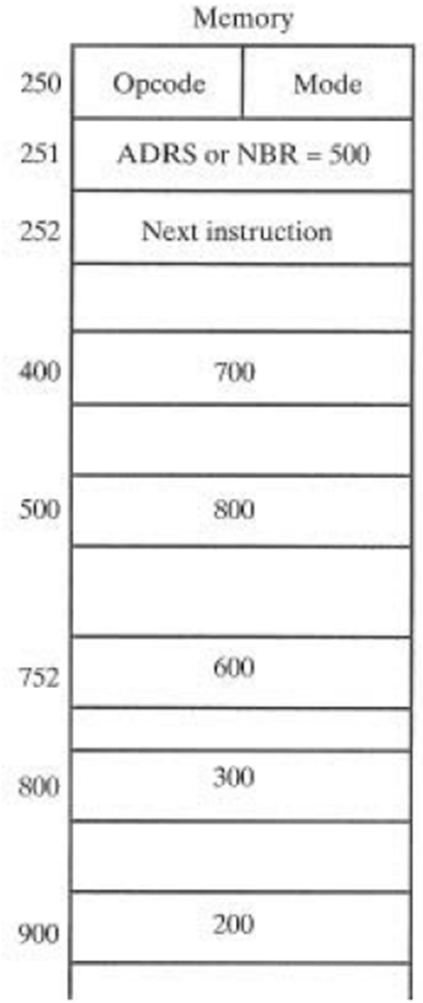


Indexed Addressing Mode

- In the indexed addressing mode, the contents of an index register are added to the address part of the instruction to obtain the effective address.
- It is often used to access consecutive locations of an array by using the same instruction and incrementing the index register after each instruction.
- A specialized variation of the index mode is the base-register mode. In this mode, the contents of a base register are added to the address part of the instruction to obtain the effective address.

Summary of Addressing Mode





Opcode: Load to ACC

Symbolic Convention for Addressing Modes

Addressing mode	Symbolic convention	Register transfer	Refers to Figure 11-6: Effective address	Refers to Figure 11-6: Contents of *ACC*
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

RISC VS CISC architectures

- RISC, or *Reduced Instruction Set Computer*, is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

- A RISC architecture has the following properties:

1. Memory accesses are restricted to load and store instructions, and data manipulation instructions are register-to-register.
2. Addressing modes are limited in number.
3. Instruction formats are all of the same length.
4. Instructions perform elementary operations.
5. RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC VS CISC architectures

- CISC is an acronym for Complex Instruction Set Computer and are chips that are easy to program and which make efficient use of memory.
- A CISC architecture has the following properties:

1. Memory access is directly available to most type of instructions.
2. Addressing modes are substantial in number.
3. Instruction formats are of different lengths.
4. Instructions perform both elementary and complex operations.

- The goal of the CISC architecture is to match more closely the operations used in programming languages and to provide instructions that facilitate compact programs and conserve memory.

Data Transfer Instructions

- Data transfer instructions move data from one place in the computer to another without changing the data.
- Typical transfers are between memory and processor registers, between processor registers and input and output registers, and among the processor registers themselves.
- Following is a list of data manipulation instructions

Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT

Stack Instructions

- The stack instructions push and pop transfer data between a memory stack and a processor register or memory. The *push* operation places an item onto the top of the stack (TOS).
- The *pop* operation removes one item from the TOS.
- Stack is basically part of the memory, and to provide the logic of pushing and popping a register is used which is called Stack Pointer (SP).
- Below is an example illustrating the operation
- For a push operation:

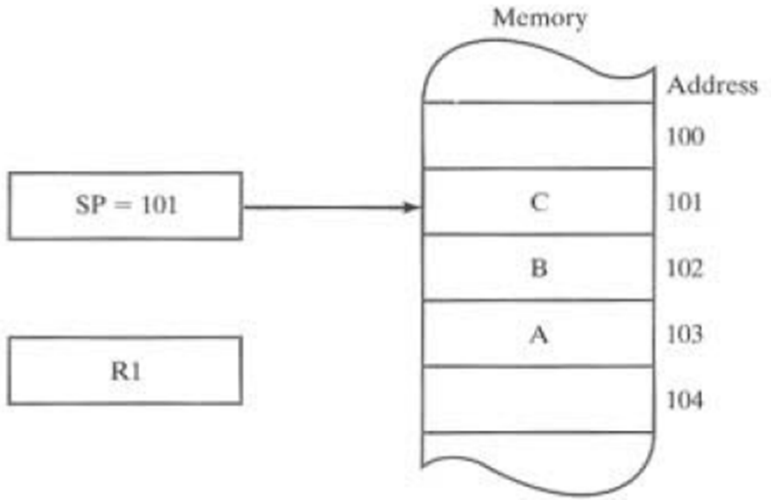
$$SP \leftarrow SP - 1$$

$$M[SP] \leftarrow R1$$

- For a pop operation:

$$R1 \leftarrow M[SP]$$

$$SP \leftarrow SP + 1$$



Independent vs Memory-Mapped I/O

- Input and output instructions transfer data between processor registers and input and output devices.
- These instructions are similar to load and store instructions except that the transfers are to and from external registers (I/O) instead of memory words.
- A *port* is typically a register with input and/or output lines connected to the device. And each port requires an address to be accessed by the computer.
- In the independent I/O the address ranges memory and I/O ports are independent from each other.
- In memory-mapped I/O the portion of memory address range is used to address the I/Os. So same type of instructions (e.g. load, store) can be used to access the both memory and I/Os.

Data Manipulation Instructions

- Data manipulation instructions perform operations on data and provide the computational capabilities of the Computer.
- There are three types of data manipulation instructions

1. Arithmetic instructions
2. Logical and bit manipulation instructions
3. Shift instructions

Arithmetic Instructions

- Four basic arithmetic instructions are addition, multiplication, subtraction and division.
- The subtract reverse instruction performs B-A instead of A-B.
- The negate instruction calculates the 2's complement of the number.

Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate	NEG

Logical and Bit-Manipulation Instructions

- Clear means making all the bit of a register ‘0’.
- Set means making all the bit of a register ‘1’.
- AND is sometimes referred to as *bit clear* instruction or *mask*
- OR is sometimes referred to as *bit set* instruction
- XOR is referred to as *bit complement* instruction

Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Set	SET
Complement	NOT
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC

Shift Instructions

- The table shows exemplary instructions of shifting. Below is a format of shift instruction which has following fields.
- OP is opcode field
- REG determines which register is to be shifted
- TYPE tells the type of shifting from the list given below
- RL tells whether to shift right or left.
- COUNT tells the number of places to be shifted

Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right with carry	RORC
Rotate left with carry	ROLC

Difference between Logical and Arithmetic Shift

- A logical shift is the shifting that we have been studying so far
- An arithmetic shift left is identical to a logical shift left, but an arithmetic shift right causes the most significant bit, the sign bit, to be propagated right. This action preserves the correct sign of a two's complement value.
- There is a chance of overflow in case of shift left. In case the sign is changed, the ALU indicates the V overflow flag. Following is an example illustrating shifting mechanism

	Initial Value	After First Shift
ASL	11101011	11010110
ASL	01111110	11111100
ASR	11101011	11110101
ASR	01111110	00111111
LSL	11101011	11010110
LSL	01111110	11111100
LSR	11101011	01110101
LSR	01111110	00111111

Floating Point Computations

- Floating point notation is used to represent long range numbers often found in scientific calculations.
- The floating-point number has two parts:
- Fraction (or mantissa) contains the sign and a fraction.
- Exponent designates position of the radix point in the number.

Fraction	Exponent
+.6132789	+04

- Decimal numbers are interpreted as representing the number in the form

$$F \times 10^E$$

- Only fraction and exponent are represented in the computer registers. 10 and decimal point are not stored explicitly.
- For example the number +1001.11 is represented with 8-bit fraction and 6-bit exponent as

Fraction	Exponent
01001110	000100

$$F \times 2^E = +(0.1001110)_2 \times 2^{+4}$$

Floating Point Computations

- A floating point number is said to be normalized if the most significant digit of the fraction is non zero. E.g., .350 is normalized but .0035 is not.
- Floating point representation increases the range of the numbers that can be accommodated in a given register.

Arithmetic Operations

- Adding and subtracting the two numbers requires that radix point must be aligned since exponent part must be equal.
- Alignment is done by shifting the one numbers radix point.
- Consider sum of following numbers

$$\begin{array}{r} .5372400 \times 10^{2} \\ +\ .1580000 \times 10^{-1} \end{array}$$

- We can either shift the first number three positions to the left or shift the second number three positions to the right.
- Shifting left is preferable cause shifting right causes loss of most significant digits.
- We shift the smaller exponent to the right by the number of places equal to the difference of exponent.
-

$$\begin{array}{r} .56780 \times 10^{5} \\ -\ .56430 \times 10^{5} \\ \hline .00350 \times 10^{5} \end{array} \qquad \begin{array}{r} .5372400 \times 10^{2} \\ +\ .0001580 \times 10^{2} \\ \hline .5373980 \times 10^{2} \end{array}$$

Arithmetic Operations

- In division we just divide the fraction and subtract the exponent.
- In multiplication we just multiply the fraction and add the exponent.

Biased Exponent

- Bias is number that is added to the exponent to make it positive.
- The biased number therefore all are positive and hence we do not need to have a sign bit for them.
- The bias is greater in magnitude than the magnitude of the exponent in order to avoid negative numbers.
- It is represented as

$$e = E + \text{number}$$

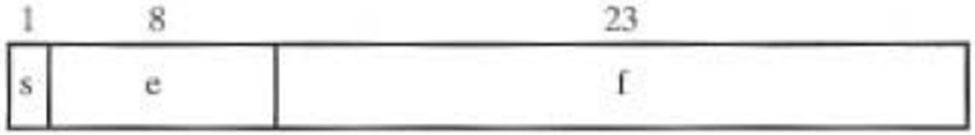
- E is actual exponent and e is biased exponent.
- It makes it easier to compare the exponent.

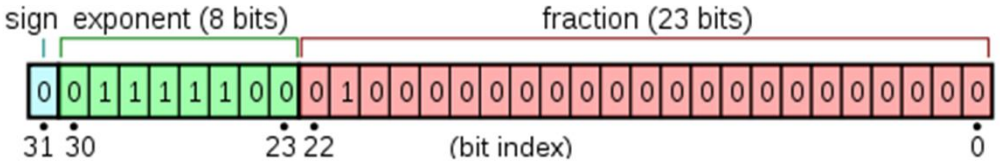
Standard Representation

- IEEE has defined two types of formats
- Single precision floating point number
- Double precision floating point number
- Single precision is 32-bit (float) while double precision is 64-bit(double).

IEEE standard format

- In IEEE format number is distributed in three fields. Sign, exponent and fraction field.





- Fraction part has 1 implicitly defined. While the number is represented in normalized form.
- have : $value = (-1)^{\text{sign}}(1 + \sum_{i=1} b_{23-i}2^{-i}) \times 2^{(e-127)}$

  In this example :
  - $sign = 0$
  - $1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 2^{-2} = 1.25$
  - $2^{(e-127)} = 2^{124-127} = 2^{-3}$

  thus :
  - $value = 1.25 \times 2^{-3} = 0.15625$

IEEE standard format

Note: This Example is taken from Wikipedia

Conversion of the fractional part:

consider 0.375, the fractional part of 12.375. To convert it into a binary fraction, multiply the fraction by 2, take the integer part and re-multiply new fraction by 2 until a fraction of zero is found or until the precision limit is reached which is 23 fraction digits for IEEE 754 binary32 format.

0.375 x 2 = 0.750 = 0 + 0.750 => $b_{-1} = 0$, the integer part represents the binary fraction digit. Re-multiply 0.750 by 2 to proceed

0.750 x 2 = 1.500 = 1 + 0.500 => $b_{-2} = 1$

0.500 x 2 = 1.000 = 1 + 0.000 => $b_{-3} = 1$, fraction = 0.000, terminate

We see that $(0.375)_{10}$ can be exactly represented in binary as $(0.011)_2$. Not all decimal fractions can be represented in a finite digit binary fraction. For example decimal 0.1 cannot be represented in binary exactly. So it is only approximated.

Therefore $(12.375)_{10} = (12)_{10} + (0.375)_{10} = (1100)_2 + (0.011)_2 = (1100.011)_2$

Also IEEE 754 binary32 format requires that you represent real values in $(1.x_1x_2...x_{23})_2 \times 2^e$ format, (see Normalized number, Denormalized number) so that 1100.011 is shifted to the right by 3 digits to become $(1.100011)_2 \times 2^3$

Finally we can see that: $(12.375)_{10} = (1.100011)_2 \times 2^3$

From which we deduce:

- The exponent is 3 (and in the biased form it is therefore 130 = 1000 0010)
- The fraction is 100011 (looking to the right of the binary point)

From these we can form the resulting 32 bit IEEE 754 binary32 format representation of 12.375 as: 0-10000010-10001100000000000000000 = 41460000_H

IEEE standard format

- Exponents with all 0's or all 1's, (decimal 255) are reserved for the following special conditions.

1) When e = 255 and f = 0, the number represents plus or minus infinity, The sign is determined from the sign bit 's'.
2) When e = 255 and f ≠ 0, the representation is considered to be not a number or NaN, regardless of the sign value. NaNs are used to signify invalid operations such as the multiplication of zero by infinity.
3) When e = 0 and f = 0, the number denotes plus or minus zero.
4) When e = 0 and f ≠ 0, the number is said to be denormalized. This is the name given to the numbers with a magnitude less than the minimum value that is represented in the normalized format.