Question 2 [CLO1]

Given a decimal number 773.4375.

(a). Write down the equivalent binary of this number. [5]

(b). Write down this binary number into normalized form. [5]

(c). Convert the number in part (b) into 32-bit IEEE-754 format (single-precision). [5]

(d). Convert the 32-bit number of part (c) to hexadecimal. [5]

Solution: Equivalent binary number is

$$773.4375_{10} = 1100000101.0111_2$$

In normalized form the number will be written as

$$1.1000001010111 \times 2^9$$

In IEEE-754 format (single precision)

$$\text{Sign} = 0, \quad \text{Exponent} = 10001000, \quad \text{Mantissa} = 10000010101110000000000$$

(Note that Exponent was originally 9, but we will add 127 into it to make it 136 which is 10001000_2).

So, 32-bit number is 0100 0100 0100 0001 0101 1100 0000 0000

which in hexadecimal is 44415C00

Question 3 [CLO2]

Consider the following RISC-V assembly code sequence: [10]

```
        addi x6, x0, 10
        addi x1, x0,19
        slli x7, x6, 2
        sub x8, x7, x6
        and x9, x7, x8
        or x10, x7, x8
```

What are the final values of registers x6, x7, x8, x9 and x10 (in decimal) after executing the above code?

The instruction **`addi x6, x0, 10`** initializes the register x6 to 10.

x6 ← 10

The instruction **`addi x1, x0, 10`** initializes the register x1 to 19.

x1 ← 19

The instruction **`slli x7, x6, 2`** shifts left the contents of x6 towards left twice and saves the results into x7 (which is equivalent to multiplication by 4).

x7 ← x6 * 4

The instruction **`sub x8, x7, x6`** subtracts the value of x6 from x7 and saves the result in register x8.

x8 ← x7 – x6

The instruction **`and x9, x7, x8`** ANDs the values of x7 and x8 and saves the result in register x9.

x9 ← x7 & x8

The instruction **`or x9, x7, x8`** ORs the value of x7 and x8 and saves the result in register x10.

x10← x7 | x8

So the final values of the registers are **x6 = 10, x7 = 40, x8 = 30, x9 = 8, x10 = 62**

Question 4 [CLO2]

Assume that register x11 contains the value 400 when the following RISC-V assembly code is executed: [10]

```
        flw f0, 0(x11)
        flw f1, 4(x11)
        fadd.s f2, f0, f1
        fmul.s f4, f0, f1
        fsw f2, 0(x11)
        fsw f4, 4(x11)
```

The memory contents from address 400 to 407 are initially as follows:

Memory Address	Byte Value
400	00
401	00
402	BC
403	C1
404	00
405	00
406	40
407	40

What will be the memory contents at these addresses after the program is executed? Fill the following table to show the updated contents.

Instruction	Meaning	Action
`flw f0, 0(x11)`	`f0 ← M[x11]` Load a floating-point value (4-bytes) from memory address x11+0 to FP register f0.	`f0` = `C1BC0000`
`flw f1, 4(x11)`	`f1 ← M[x11+4]` Load a 4-byte floating-point value from memory address x11+4 to FP register f1.	`f1` = `40400000`
`fadd.s f2, f0, f1`	`f2 ← f0 + f1` Add the contents of registers f0 and f1 and save the result in register f2.	`f2` = `C1A40000`
`fmul.s f4, f0, f1`	`f4 ← f0 + f1` Multiply the contents of registers f0 and f1 and save the result in register f2.	`f4` = `C28D0000`
`fsw f2, 0(x11)`	`M[x11] ← f2` Save the contents of f2 into memory at address M[x11]	`M[x11]= M[address 400 to 403]=` **C1A40000**
`fsw f2, 0(x11)`	`M[x11+4] ← f4` Save the contents of f4 into memory at address M[x11+4]	`M[x11+4]= M[address 404 to 407]=` **C28D0000**

Explanation and final answers are as follows:

Step 1: Load `f0` from Memory (`flw f0, 0(x11)`)

- **Base Address:** `x11 = 400`
- **Offset:** `0`
- **Load 4 bytes from `400-403` (little-endian):**

$$\text{Bytes: } 00\ 00\ BC\ C1$$

$$\text{Word (hex): } C1BC0000$$

- **Interpret as IEEE-754 Single-Precision Float:**
  - **Sign:** `1` (negative)
  - **Exponent:** `10000011` (131 - 127 = 4)
  - **Mantissa:** `01111000000000000000000`
  - **Value:**

$$-1.01111 \times 2^4 = -10111.1_2 = -23.5_{10}$$

$$f0 = -23.5$$

Step 2: Load `f1` from Memory (`flw f1, 4(x11)`)

- **Base Address:** `x11 = 400`
- **Offset:** `4` → Load from `404-407`:

$$\text{Bytes: } 00\ 00\ 40\ 40$$

$$\text{Word (hex): } 40400000$$

- **Interpret as IEEE-754 Single-Precision Float:**
  - **Sign:** `0` (positive)
  - **Exponent:** `10000000` (128 - 127 = 1)
  - **Mantissa:** `10000000000000000000000`
  - **Value:**

$$1.1 \times 2^1 = 11.0_2 = 3.0_{10}$$

$$f1 = 3.0$$

Step 3: Floating-Point Addition (`fadd.s f2, f0, f1`)

$$f2 = f0 + f1 = -23.5 + 3.0 = -20.5$$

- **Convert `-20.5` to IEEE-754:**
  - Binary: `-10100.1` → `-1.01001 × 2^4`
  - **Sign:** `1`
  - **Exponent:** `4 + 127 = 131` → `10000011`
  - **Mantissa:** `01001000000000000000000`
  - **IEEE-754 Hex:** `C1A40000`

Step 4: Floating-Point Multiplication (`fmul.s f4, f0, f1`)

$$f4 = f0 \times f1 = -23.5 \times 3.0 = -70.5$$

- **Convert `-70.5` to IEEE-754:**
  - Binary: `-1000110.1` → `-1.0001101 × 2^6`
  - **Sign:** `1`
  - **Exponent:** `6 + 127 = 133` → `10000101`
  - **Mantissa:** `00011010000000000000000`
  - **IEEE-754 Hex:** `C28D0000`

Step 5: Store `f2` Back to Memory (`fsw f2, 0(x11)`)

- **Value:** `-20.5` → `C1A40000` (little-endian):

$$00\ 00\ A4\ C1 \quad \text{(stored at addresses 400-403)}$$

Step 6: Store `f4` Back to Memory (`fsw f4, 4(x11)`)

- **Value:** `-70.5` → `C28D0000` (little-endian):

$$00\ 00\ 8D\ C2 \quad \text{(stored at addresses 404-407)}$$

Final Memory State (After Execution):

Memory Address	Byte Value
400	00
401	00
402	A4
403	C1
404	00
405	00
406	8D
407	C2

Question 5 [CLO2]

Consider the following assembly language code:

I_1	`ADDI x1, x0, 5`
I_2	`ADDI x2, x0, 10`
I_3	`ADD x3, x1, x2`
I_4	`SUB x4, x3, x1`
I_5	`SW x4, 0(x0)`
I_6	`LW x5, 4(x0)`
I_7	`ADD x6, x5, x3`

(a). Identify any data hazards in the above assembly language code if it is run on a 5-stage RISC-V pipeline. Specify the instructions involved the pipeline stages where the hazard occurs. [4]

Pipeline Hazard Analysis

(a). Data Hazards Identification:

The following data hazards occur in the 5-stage pipeline (IF-ID-EX-MEM-WB):

- `I2 (ADDI x2)` $\rightarrow$ `I3 (ADD x3)`:
  - Hazard between EX stage of I2 and ID stage of I3
- `I3 (ADD x3)` $\rightarrow$ `I4 (SUB x4)`:
  - Hazard between EX stage of I3 and ID stage of I4
- `I4 (SUB x4)` $\rightarrow$ `I5 (SW x4)`:
  - Hazard between EX stage of I4 and ID stage of I5
- `I6 (LW x5)` $\rightarrow$ `I7 (ADD x6)`:
  - Hazard between MEM stage of I6 and ID stage of I7 (load-use hazard)

(b). How many stall cycles will have to be inserted to have the smooth operation if the processor has no forwarding hardware? Show the stall cycles on the pipeline space-timing diagram. (Note that a register can be written during the 1st half of clock cycle and read later during the same cycle). [4]

Stall Cycles Without Forwarding:

Total stalls needed: 2 cycles x 4 = 8 stall cycles

(c). Now let us assume that the processor has the forwarding hardware. Which of the data hazard(s) can not be resolved by forwarding? Now how many stall cycles are needed? [4]

Cannot be resolved by forwarding:

- I6 $\rightarrow$ I7 (load-use hazard, needs 1 stall cycle)

(d). Compare the performance impact of forwarding versus no-forwarding.

Performance Comparison:

	No Forwarding	With Forwarding
Total cycles	19	12
Stalls	8	1
Speedup	1.0x	1.58x

(e). Is it possible to eliminate all the data hazards by reordering the code? If yes, write down the updated code. If not, why not? [4]

Code Reordering:

Yes, we can eliminate all hazards by reordering:

```
ADDI x1, x0, 5        ; I1
ADDI x2, x0, 10       ; I2
ADD  x3, x1, x2       ; I3
SUB  x4, x3, x1       ; I4
LW   x5, 4(x0)        ; I6 moved up
SW   x4, 0(x0)        ; I5
ADD  x6, x5, x3       ; I7
```

This reordering:

- Moves the load (I6) earlier to avoid load-use hazard
- Maintains correct dependencies
- Requires no stalls with forwarding

Question 6 [CLO3]

Write down RISC-V assembly language program that reverses a floating-point array in place. i.e., the element at index 0 should occupy the last index, at index 1 should be placed at second last index and so on. [10]

(Assume the array is stored in memory having all elements of type `float` and its starting address is already in register x9. The number of elements are stored in register x10).

```
        # x9: starting address of the float array
        # x10: number of elements in the array (n)
        # Initialize pointers
addi x11, x9, x0            # x11 = left pointer (starts at beginning of array)
slli x12, x10, 2            # Calculate n * 4 ( since each float is 4 bytes)
add x12, x9, x12            # x12 = right pointer ( starts at end of array )
addi x12, x12, -4           # Adjust to point to last element
        # Calculate loop count (n/2)
srli x13, x10, 1            # x13 = n / 2 ( number of swaps needed )
loop:
beq x13, x0, end            # If x13 == 0, we 're done
        # Load left and right elements
flw f0, 0(x11)              # f0 = array [ left ]
flw f1, 0(x12)              # f1 = array [ right ]
        # Swap the elements
fsw f1, 0(x11)              # array [ left ] = f1
fsw f0, 0(x12)              # array [ right ] = f0
        # Update pointers and counter
addi x11, x11, 4            # left pointer += 4 ( move right )
addi x12, x12, -4           # right pointer -= 4 ( move left )
addi x13, x13, -1           # decrement swap counter
j loop                      # repeat
end:
```