USE YOUR IMAGINATION™

Blue Book

EXAMINATION BOOK

BOX NO. _____

NAME RAMAN ICR JHA

SUBJECT CSA

CLASS N13866145

SECTION A

INSTRUCTOR AZEEZ B

DATE 12/17/2025

11" x 8.5" 8 LEAVES 16 PAGES

ROARING SPRING® PAPER PRODUCTS
ROARING SPRING, PA 16673

① The RISC-V assembly program to add the contents of 2 registers x3, and x4. In 2's complement arithmetic, an overflow happens in basically two conditions —

(i) Adding two positive numbers yields a negative number.

(ii) Adding two negative numbers yields a positive number.

overflow cannot occur if we add a positive and a negative number in the addition operation.

RISC-V being a reduced instruction set does not have a special overflow flag to check in the addition operation.
In this case, we must manually check this overflow using a sequence of instructions while performing the add operation in the RISC-V processor.

Program —

This program explicitly follows the logic if the operand have same sign, check if the result have different sign

Registers used —

X3: first operand
X4: second operand
X5, ~~X6~~: Stores the result (X3+X4)
X6, X7: Temporary registers
X10: overflow flag (1 = overflow, 0 = No overflow)

Start

```
add  X5, X3, X4      # Perform the addition.
XOR  X6, X3, X4      # check if the
                       operand have different
                       signs.
blt  X6, zero, NO_overflow # if (X3 ^ X4) < 0
                       signs are different. branch.
XOR  X7, X3, X5      # Check if the sum
                       and one of the operand
                       have different sign.
blt  X7, zero, overflow # if (X3 ^ X5) < 0
NO_overflow          Signs are different, overflow!
addi X10, zero, 0    # Set overflow
                       flag to 0.
```

overflow:
addi x10, zero, 1 # set overflow flag to 1.

Done,
x5: It contains the sum which will be incorrect if the overflow occurred.
x10: contains the status of the overflow.

(2) This questions explores the byte-addressable and the concept of endianess of a memory —

Key informations —

* word size : 32 bits : 4 bytes

* Memory: byte addressable.

* Word-to-byte-mapping: 0th word is at address 0, and first word is at address 4.

* Pattern: The byte address of an nth word is at address n x 4.

(2) (a) The pattern address = n×4

→ n = 44 (since we start counting from 0, the 44th word is at index 44)

→ Byte address = 44×4 = 176

in hexadecimal, $176_{10} = B0_{16}$

Ans: The byte address of the 44th word is 176 (or 0XB0)

(b) A 32-bit (4 byte) word starting at address 176 will occupy that address and the next three consecutive byte address.

→ 1st byte : 176 address
→ 2nd byte : 177 address
→ 3rd byte : 178 address
→ 4th byte : 179 address

Ans: The word spans byte addresses 176, 177, 178, 179 (or 0XB0, 0XB1, 0XB2, 0XB3)

(2) (c) Number: 0XFF223344

→ Bytes (from most to least significant)

→ Most significant bytes (MSB): FF

Byte 2: 22
Byte 3: 33

→ Least significant bytes (LSB) = 44
Memory addresses: 176, 177, 178, 179

Big - Endian →

Memory address	Data byte
176 (lowest address)	FF (MSB)
177	22
178	33
179 (Highest address)	44 (LSB)

Little - Endian →

Memory address	Data byte
176 (lowest address)	44 (LSB)
177	33
178	22
179 (Highest address)	FF (MSB)

(3) The program provided in the question, is a simple loop that calculates the sum of the number from 0 to 9.

→ The loop body executes 10 times (for i = 0, 1, ... 9)

(i) Assumptions

↑ We use the standard cycle count for a multicycle datapath.

→ R-type (add) : 4 cycles
→ I-type (addi) : 4 cycles
→ load (lw) : 5 cycles
→ store (sw) : 4 cycles
→ Branch (beq) : 3 cycles (whether taken or not taken)
→ jump (j) : 3 cycles

Cycle calculation:

(a) Setup (outside loop) —

→ addi s0, zero, 0 : 4 cycles
→ addi s1, zero, 0 : 4 cycles
→ addi t0, zero, 10 : 4 cycles

Setup total = 12 cycles

(b) loop (10 iterations, from i=0 ... 9)

→ beq (Not taken) : 3 cycles
→ add : 4 cycles
→ addi : 4 cycles
→ j : 3 cycles
→ cycles per iteration: 14 cycles
→ loop total : 14 × 10 = 140 cycles

(c) Final loop check (i=10)
→ beq : 3 cycles
The loop then exits to L2

Total cycles = 12 + 140 + 3 = 155 cycles

CPI calculation —
⇒ total instructions
setup : 3
loop : 10 × 4 = 40
Final check : 1

total instructions = 3 + 40 + 1 = 44

$$CPI = \frac{\text{Total cycles}}{\text{Total instructions}} = \frac{155}{44} = 3.523$$

(3)(b) Pipelined RISC-V processor —

Stall calculation —

Total cycles = Total instructions + Total stall cycles + (Pipeline latency - 1)

(a) Total instruction, as calculated before, I = 44

(b) Total stall cycles

Control hazard (jumps)
= 10 × 1 = 10 stalls

Control hazard (branches)
= 1 (10 times not taken, 1 time taken)

Data hazard = 0
(These are all resolved by standard forwarding.)

Total cycle stalls = 10 + 1 = 11

(c) Pipeline latency = 5 - 1 = 4

③ (b) Total cycles = 44 + 11 + 4 = 59

CPI Calculation = $\frac{\text{total cycles}}{\text{total instructions}}$

$= \frac{59}{44} = 1.341$

④ In this question, we must map the instructions to their pipeline stages for each cycles.—

The key here is to identify the stalls first. The pipeline's state without stalls, which is what exists during cycle 5.

Clock cycle	1	2	3	4	5
addi S1 ...	IF	ID	EX	MEM	WB
addi S0		IF	ID	EX	MEM
lw S3 ...			IF	ID	EX
sw S3 ...				IF	ID
xor S2 ...					IF

Based on the diagram, here is the activity in cycle 5:

→ addi S1, zero, S2 (WB stage)
writing: The value S2 to register S1.

→ addi S0, S1, -4 (MEM stage)
This is an ALU instruction, it is idle during the MEM stage. No register read or writes.

→ lw S3, 16(S0) (Ex stage):
Reading: The value of register S0 to calculate the memory address (S0+16)

→ sw $3, 20(S0) (ID stage):
Reading: The value of register S0 (for the base address) and register S3 (the data to be stored)

→ xor S2, S0, S3 (IF stage)
The information is being fetched from the memory.
No register reads or writes.

Summary for cycle 5:

Registers being written: S1
Registers being READ: S0 (read by two different instructions) and S3

(5) Hazard Analysis

1. Addi S0, S1, -4 (reads S1 from addi S1)
→ addi S1 (EX) → addi S0 (ID)
→ Handled by EX → EX forwading, No stall.

2. lw S3, 16(S0) (reads S0 from addi S0)
→ addi S0 (EX) → lw S3 (ID)
→ Handled by EX → EX forwarding, No Stall.

3. sw S3, 20(S0) (reads S3, from lw S3)
→ lw S3 (MEM) → sw S3 (ID)
→ This is a load use 1 dazard. The sw instruction reads S3 in its ID stage. But the lw instruction only has the data available after in MEM stage.
→ The pipeline must stall for 1 cycle. The sw instruction waits

→ in cycle 7, sw (in D) gets S3 value from the lw's MEM/WB register, → EX forwarding path.

4. xor S2, S0, S3 (reads S3 from lw S3):
→ This instruction is stalled by the sw in front of it.
→ when it reaches its ID stage, the lw instruction now has already completed its WB stage (cycle 7). The value of S3 is read normally from the register file. no stall.

5. or S2, S2, S3 (reads S2 from xor S2)
→ xor S2 (EX) → or S2 (ID)
→ Handled by EX → EX forwarding. No Stall.

Pipeline diagram —

Instruction	1	2	3	4	5
1. addi S1, zero, 52	IF	ID	EX	MEM	WB
2. addi S0, S1, -4		IF	ID	EX	MEM
3. lw S3, 16(S0)			IF	ID	EX
4. sw S3, 20(S0)				IF	ID
5. xor S2, S0, S3					IF
6. or S2, S2, S3					