IPA Project Report

Harshit Goyal (2023102054)

Harshit.goyal@students.iiit.ac.in

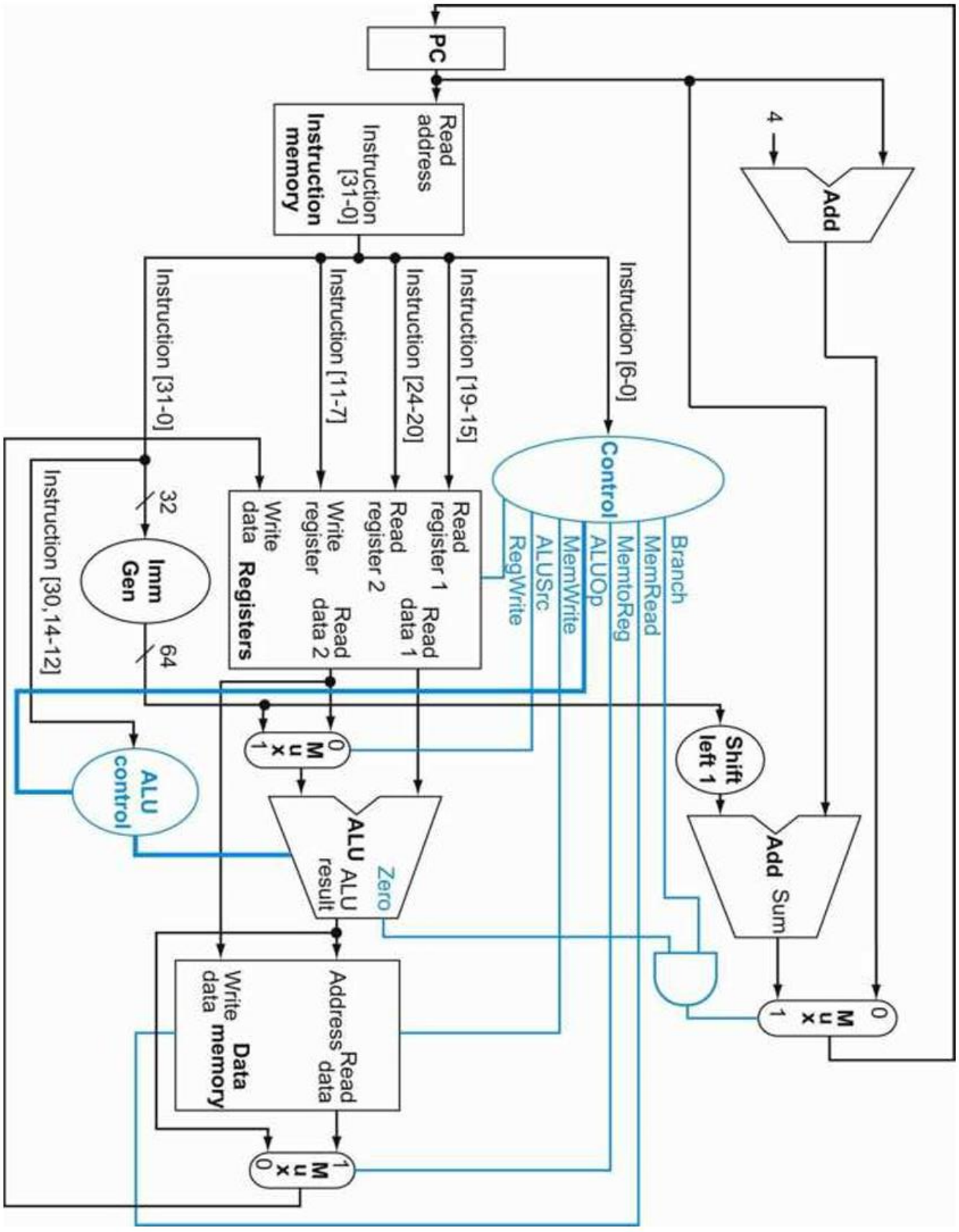
Manikya Pant (2023112024)

Manikya.pant@research.iiit.ac.in

Nitin Rama Sai Grandhi (2023112026)

grandhi.sai@research.iiit.ac.in

Task -1 Sequential Implemenmtation



This is the hardware design for the sequential implementation of the processor based on the 32 bit risc-v architecture which can perform the basic operations like add, and, sub, or, ld, sd, beq.

RISC-V Instruction Table

Instruction	Description
add x3, x4, x7	Adds the values in registers x4 and x7, storing the result in x3.
sub x3, x4, x7	Subtracts the value in x7 from x4 and stores the result in x3.
or x3, x4, x7	Performs a bitwise OR operation between x4 and x7, storing the result in x3.
and x3, x4, x7	Performs a bitwise AND operation between x4 and x7, storing the result in x3.
ld x3, 0(x4)	Loads a 64-bit value from memory at address x4 + 0 into register x3.
sd x4, 0(x29)	Stores a 64-bit value from register x4 into memory at address x29 + 0.
beq x2, x8, Target	Branches to the label Target if the values in x2 and x8 are equal.

Here, our processor executes one instruction per clock cycle, which means next instruction will only execute when the last instruction gets completed.

This simple processor design is divided into following blocks :

1. Instruction Memory (32-bit)

- **Function:** Stores the program instructions in the form of binary(Machine code).

---

2. Register File (64-bit)

- **Function:** Stores register values and allows reading/writing registers.
- **Inputs:**
  - **RegWrite (1 bit):** Enables writing to the register.
  - **Read Register 1 (5 bits) (rs1):** Specifies the first register to read.
  - **Read Register 2 (5 bits) (rs2):** Specifies the second register to read.
  - **Write Register (5 bits) (rd):** Specifies the register to be written to.
  - **Write Data (64 bits):** Data to be written into the specified register.
- **Outputs:**
  - **Read Data 1 (64 bits):** Value stored in register rs1.
  - **Read Data 2 (64 bits):** Value stored in register rs2.

---

3. ALU (Arithmetic Logic Unit)

- **Function:** Performs arithmetic and logical operations.
- **Inputs:**
  - **Operand1 (64 bits):** Data from register rs1.
  - **Operand2 (64 bits):** Data from register rs2 or an immediate value.
  - **ALU Control (4 bits):** Specifies the operation (add, sub, and, or, etc.).
- **Outputs:**
  - **Result (64 bits):** The computed result.

- **Zero (1 bit):** Signals if the result is zero (used for branch instructions).

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

---

4. ALU Control

- **Function:** Decodes ALU operations based on instruction type.

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

FIGURE 4.13 **The truth table for the 4 ALU control bits (called Operation).**

---

5. Immediate Generator (ImmGen)

- **Function:** Extracts and sign-extends the immediate value from an instruction from 12 to 64 bits.

---

6. Data Memory

- **Function:** Stores and retrieves data from memory.
- **Inputs:**
  - **MemRead (1 bit):** Enables reading from memory.

- **MemWrite (1 bit):** Enables writing to memory.
- **MemWriteData (64 bits):** Data to be written into memory.
- **MemAddress (64 bits):** Address of memory location.

- **Output:**
  - **ReadData (64 bits):** Data retrieved from memory.

---

7. Program Counter (PC)

- **Function:** Stores the current instruction address and updates it.
- **Inputs:**
  - **Mux Output (64 bits):** Decides between PC+4 or a branch/jump address.
- **Output:**
  - **PC (64 bits):** The updated instruction address.

---

8. Control Unit

- **Function:** Generates control signals based on the opcode.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

---

9. PC + 4 (Adder)

- **Function:** Calculates the address of the next sequential instruction.

- **Inputs:**
  - **PC (64 bits):** Current instruction address.
- **Output:**
  - **PC + 4 (64 bits):** Address of the next instruction.

---

10. PC + 2*Immediate Address

- **Function:** Calculates the branch/jump target address.
- **Inputs:**
  - **PC (64 bits):** Current instruction address.
  - **Immediate Value (12 bits, sign-extended to 64 bits):** The offset to be added.
- **Output:**
  - **PC + Address (64 bits):** The new instruction address.

---

12. Multiplexer (MUX)

- **Function:** Selects between two inputs based on control signals.
- **Usage Cases:**
  - **Selecting PC + 4 or branch address.**
  - **Selecting between register data or immediate value for ALU.**
  - **Selecting between ALU result or memory data for register write-back.**

Workflow of the Sequential RISC-V Processor (Based on Instruction Formats)

RISC-V instructions are categorized into different formats:

- **R-format** (Register-Register operations like ADD, SUB, AND, OR, etc.)
- **I-format** (Immediate operations like ld etc.)
- **S-format** (Store operations like Sd)
- **SB-format** (Branch operations like BEQ)

Each format follows a different path through the processor's blocks.

---

1. R-format (e.g., ADD, SUB, AND, OR, etc.)

Example Instruction: ADD x3, x1, x2 (i.e., x3 = x1 + x2)

Step-by-Step Execution

1. **Instruction Fetch**
    - PC → Address sent to **Instruction Memory**
    - **Instruction Memory** fetches **32-bit instruction**
2. **Instruction Decode**
    - **Control Unit** decodes the instruction (R-format).
    - **Register File** reads:
        - rs1 (x1) → Operand 1
        - rs2 (x2) → Operand 2
3. **Execution**
    - **ALU Control Unit** generates the appropriate **ALU control signal** using funct7 and funct3.
    - **ALU** performs the operation (ADD, SUB, AND, etc.).

- **Result** stored in **ALU Output**.

4. **Write Back**
    - **Register File** writes the result into rd (x3).

Key Blocks Used in R-format:

PC → Instruction Memory → Register File → ALU Control → ALU → Register File

---

2. I-format (e.g. Ld - Load double Word)

Example Instruction: ld x2 0(x3)

Step-by-Step Execution

1. **Instruction Fetch**
    - PC fetches **instruction from memory**.
2. **Instruction Decode**
    - **Control Unit** decodes the instruction as an **I-format** operation.
    - **Register File** reads rs1 (x3).
    - **Immediate Generator** extracts and sign-extends the **12-bit immediate (12 → 64-bit)**.
3. **Execution**
    - **ALU** adds rs1 and the **immediate value** (x3 + 0).
4. **Data Memory**
    - Reads the address stored in x3 in data memory.
5. **Write Back**
    - **Register File** stores the result in rd (x2).

Key Blocks Used in I-format:

PC → Instruction Memory → Register File → ImmGen → ALU → Data Memory → Register File

---

3. S-format (e.g., SD - Store DoubleWord)

Example Instruction: SD x3, 10(x1) (i.e., store x3 at x1 + 10)

Step-by-Step Execution

1. **Instruction Fetch**
    - PC fetches **instruction from memory**.
2. **Instruction Decode**
    - **Control Unit** decodes the instruction as an **S-format** operation.
    - **Register File** reads:
        - rs1 (x1) → Base address
        - rs2 (x3) → Data to store
    - **Immediate Generator** extracts and sign-extends **12-bit immediate (12 → 64-bit)**.
3. **Execution**
    - **ALU** computes memory address (x1 + 10).
4. **Memory Write**
    - **Data Memory** stores x3 at the computed address.

Key Blocks Used in S-format:

PC → Instruction Memory → Register File → ImmGen → ALU → Data Memory

---

4. SB-format (e.g., BEQ - Branch Equal)

Example Instruction: BEQ x1, x2, label (i.e., if x1 == x2, branch to label)

Step-by-Step Execution

1. **Instruction Fetch**
    - PC fetches **instruction from memory**.
2. **Instruction Decode**
    - **Control Unit** decodes the instruction as a **SB-format** operation.
    - **Register File** reads:
        - rs1 (x1)
        - rs2 (x2)
    - **Immediate Generator** extracts and sign-extends **12-bit branch offset**.
3. **Execution**
    - **ALU** compares rs1 and rs2.
    - If x1 == x2, **ALU outputs zero**.
4. **Branch Decision**
    - **Shift Left by 1** processes immediate offset.
    - **PC + Address Adder** computes **new PC** (PC + offset).
    - **MUX selects branch address if ALU outputs zero**.

Key Blocks Used in SB-format:

PC → Instruction Memory → Register File → ImmGen → Shift Left 1 → ALU → PC + Address Adder → MUX → PC

Final Overview (Comparison of Formats & Blocks Used)

Format	Registers Used	Immediate Used?	ALU Used?	Memory Access?	PC Update?
R-format (ADD, SUB, AND)	rs1, rs2, rd	No	Yes	No	PC+4
I-format (ld)	rs1, rd	Yes	Yes	Read (LD)	PC+4
S-format (Sd)	rs1, rs2	Yes	Yes (Address Calculation)	Write (SD)	PC+4
SB-format (BEQ)	rs1, rs2	Yes	Yes (Comparison)	No	PC+offset

Test Cases

1. Basic operation we have implemented to show the wiorking seperately

   Assembly code

```
ld x5, 0(x2)
ld x6, 1(x2)
add x10, x5, x6
sub x11, x5, x6
and x12, x5, x6
or x13, x5, x6
sd x10 2(x2)
sd x11 3(x2)
sd x12 4(x2)
sd x13 5(x2)
```

   Machine code

```
00000000000000010000001010000011
00000000000100010000001100000011
00000000010100110000010100110011
01000000010100110000010110110011
00000000010100110111011000110011
00000000010100110110011010110011
00000000101000010000000100100011
00000000101100010000000110100011
```

```
00000000110000010000001000100011
00000000110100010000001010100011
```

Result snapshot

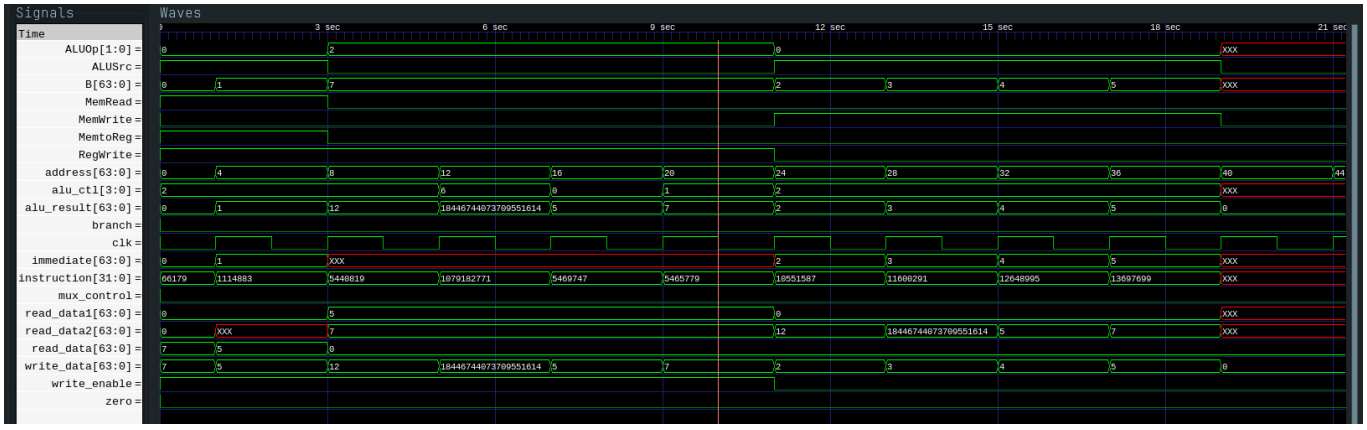
```
20
21  initial begin
22      registers[2]=64'd0;
23  end
```

X2 contains the address of memory where the number is stored on which operations are to be performed.

initial_data.txt

```
1  0000000000000000000000000000000000000000000000000000000000000111
2  0000000000000000000000000000000000000000000000000000000000000101
3
```

This shows that 7 and 5 are stored in mem[0] and mem[1].



This shows the updated memory,

Mem[0]=7

Mem[1]=5

Mem[2]=c(12) add 7 and 5

Mem[3]=-2 in hex sub 5 and 7

Mem[4]=5 (7 and 5)

Mem[5]=7 (7 or 5)

```
memory_dump.hex
 1    // 0x00000000
 2    0000000000000007
 3    0000000000000005
 4    000000000000000c
 5    fffffffffffffffe
 6    0000000000000005
 7    0000000000000007
 8    xxxxxxxxxxxxxxxx
 9    xxxxxxxxxxxxxxxx
10    xxxxxxxxxxxxxxxx
11    xxxxxxxxxxxxxxxx
12    xxxxxxxxxxxxxxxx
13    xxxxxxxxxxxxxxxx
14    xxxxxxxxxxxxxxxx
15    xxxxxxxxxxxxxxxx
16    xxxxxxxxxxxxxxxx
17    xxxxxxxxxxxxxxxx
18    // 0x00000010
```

2. Finding sum of all the elements in an array

Assembly code

```
0         add x27 x0 x0        // i=0
4         add x25 x0 x0        // sum=0
8         add x28 x0 x0        // j=0
12         loop:   beq x27 x24 End     // if i==n then go to end
16                 add x26 x28 x23     // tempaddress = baseaddress + j
20                 ld x26 0(x26)       // loading the value at tempaddress
24                 add x25 x25 x26     // sum+=a[i]
28                 add x28 x28 x22     // j++
32                 add x27 x27 x22     // i++
36                 beq x0 x0 loop      // go to loop
40       End :     sd x25 0(x5)       // storing the sum in the address of x5
```

Machine code

```
00000000000000000000110110110011
00000000000000000000110010110011
00000000000000000000111000110011
```

```
00000001101111000000111001100011
00000001110010111000110100110011
00000000000011010000110100000011
00000001101011001000110010110011
00000001011011110000111000110011
00000001011011011000110110110011
11111110000000000000010011100011
00000001100100101000000000100011
```

Result snapshot

```
21    initial begin
22        registers[5]=64'd10;
23        registers[23]=64'd0;
24        registers[24]=64'd5;
25        registers[22]=64'd1;
26    end
27
```

X5 contains the memory location where the ans needs to be stored.

X23 contains the base address of the array

X24 contains the length of the aray

X22 contains 1 for increment the array index

initial_data.txt

```
1   00000000000000000000000000000000000000000000000000000000000000111
2   00000000000000000000000000000000000000000000000000000000000000101
3   00000000000000000000000000000000000000000000000000000000000000001
4   00000000000000000000000000000000000000000000000000000000000000010
5   00000000000000000000000000000000000000000000000000000000000000011
```

Initial array of length 5 stored in the memory [7,5,1,2,3]

memory_dump.hex

```
1   // 0x00000000
2   0000000000000007
3   0000000000000005
4   0000000000000001
5   0000000000000002
6   0000000000000003
7   xxxxxxxxxxxxxxxx
8   xxxxxxxxxxxxxxxx
9   xxxxxxxxxxxxxxxx
10  xxxxxxxxxxxxxxxx
11  xxxxxxxxxxxxxxxx
12  0000000000000012
13  xxxxxxxxxxxxxxxx
14  xxxxxxxxxxxxxxxx
15  xxxxxxxxxxxxxxxx
16  xxxxxxxxxxxxxxxx
17  xxxxxxxxxxxxxxxx
18  // 0x00000010
```

Updated memory, mem[10] contains the ans i.e. sum of all elements of the array in this case (18) in decimal.

3. Finding odd number of elements in an array

Assembly code

```
0               add x27 x0 x0       // i=0
4               add x25 x0 x0       // sum=0
8               add x28 x0 x0       // j=0
```

```
12              beq x0 x0 loop      // go to loop
16    check:    add x25 x25 x22     // ans++
20              beq x0 x0 cont      // go to cont
24    loop:     beq x27 x24 End     // if i==n then go to end
28              add x26 x28 x23     // tempaddress = baseaddress + j
32              ld x26 0(x26)       // loading the value at tempaddress
36              or x30 x26 x22       // a[i]|1
40              beq x30 x26 check
44    cont:     add x28 x28 x22     // j++
48              add x27 x27 x22     // i++
52              beq x0 x0 loop      // go to loop
56    End:      sd x25 0(x22)       // storing the ans in the address of x22
```

Machine code

```
00000000000000000000110110110011
00000000000000000000110010110011
00000000000000000000111000110011
00000000000000000000011001100011
00000001011011001000110010110011
00000000000000000000110001100011
00000011100011011000000001100011
00000001011111100000110100110011
00000000000011010000110100000011
00000001011011010110111100110011
11111111101011110000010011100011
00000001011011100000111000110011
00000001011011011000110110110011
11111110000000000000001011100011
00000001100100011000000000100011
```

Result snapshot

```
initial begin
    registers[3]=64'd10;
    registers[23]=64'd0;
    registers[24]=64'd7;
    registers[22]=64'd1;
end
```

X3 contains the memory location where the ans needs to be stored.

X23 contains the base address of the array

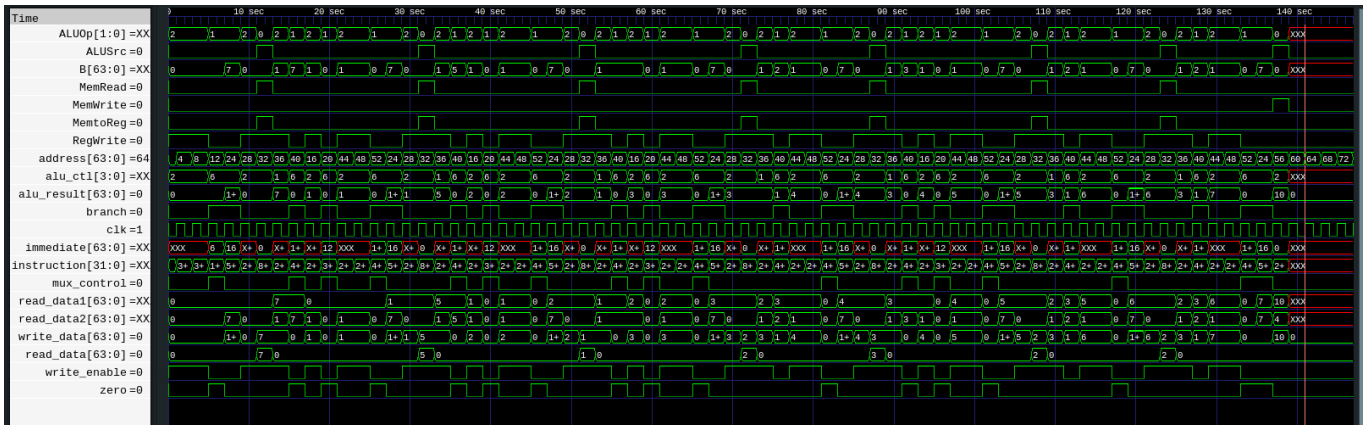
X24 contains the length of the aray

X22 contains 1 for increment the array index

initial_data.txt

```
1   00000000000000000000000000000000000000000000000000000000000000111
2   00000000000000000000000000000000000000000000000000000000000000101
3   00000000000000000000000000000000000000000000000000000000000000001
4   00000000000000000000000000000000000000000000000000000000000000010
5   00000000000000000000000000000000000000000000000000000000000000011
6   00000000000000000000000000000000000000000000000000000000000000010
7   00000000000000000000000000000000000000000000000000000000000000010
```

Initial memory , array of length 7 is stored in this.



memory_dump.hex

```
1   // 0x00000000
2   0000000000000007
3   0000000000000005
4   0000000000000001
5   0000000000000002
6   0000000000000003
7   0000000000000002
8   0000000000000002
9   XXXXXXXXXXXXXXXX
10  XXXXXXXXXXXXXXXX
11  XXXXXXXXXXXXXXXX
12  0000000000000004
13  XXXXXXXXXXXXXXXX
14  XXXXXXXXXXXXXXXX
15  XXXXXXXXXXXXXXXX
16  XXXXXXXXXXXXXXXX
17  XXXXXXXXXXXXXXXX
18  // 0x00000010
```

Here we have our ans stored in x10 i.e. 4 (odd numbers in the array)

4. Sorting an array of length n

Assembly code

```
0                   add x26 x0 x0  // i=0
4                   sub x31 x1 x6  //  x31 contains n-2 SAY(P)
8    Loop1:         beq x26 x31 Exit
12                  add x27 x0 x0 // j=0
16                  sub x30 x1 x26  // n-i
20                  sub x30 x30 x6 // n-i-1
24    Loop2:        beq x27 x30 Temp
28                  add x20 x2 x27  // address of arr[j]
32                  add x22 x27 x6  // calcualted j+1
36                  add x21 x2 x22   // address of arr[j+1]
40                  ld  x18 0(x20)   // x18= arr[j]
44                  ld x19 0(x21)    // x19=arr[j+1]
48                  blt x19 x18 swap  // swap arr[j] & arr[j+1]
52    intermediate: add x27 x27 x6   // j=j+1
56                  beq x0 x0 Loop2  // go back to beginnig
60    Temp:         add x26 x26 x6     // i=i+1
64                  beq x0 x0 Loop1
68    Swap:         sd x18 0(x21)
72                  sd x19 0(x20)
76                  beq x0 x0 intermediate
80    Exit:
```

Machine code

```
00000000000000000000110100110011
01000000011000001000111110110011
00000101111111010000010001100011
00000000000000000000110110110011
01000001101000001000111100110011
01000000011011110000111100110011
00000011111011011000001001100011
00000001101100010000101000110011
00000000011011011000101100110011
00000001011000010000101010110011
00000000000010100011100100000011
00000000000010101011100110000011
00000001001010011100101001100011
00000000011011011000110110110011
11111110000000000000000011100011
00000000011011010000110100110011
11111100000000000000010011100011
00000001001010101011000000100011
00000001001110100011000000100011
11111110000000000000010011100011
```

Result snapshot

```
initial begin
    registers[2]=64'd0;
    registers[1]=64'd3;
    registers[6]=64'd1;
    registers[7]=64'd2;
end
```

X2 contains the base address of the array

X1 contains the length of the array

X6 and x7 contains 1 and 2 for increment.

initial_data.txt

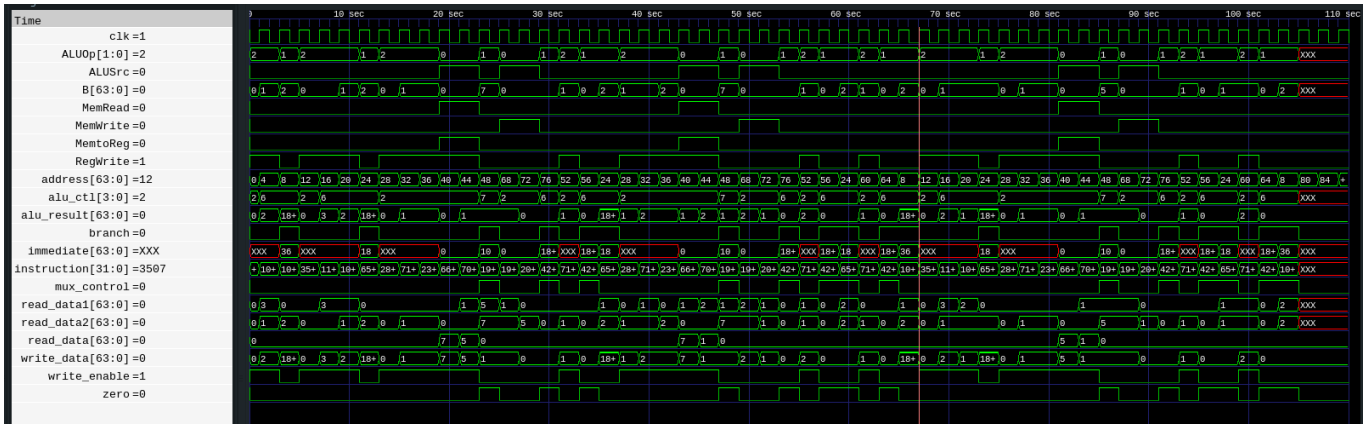
```
1  00000000000000000000000000000000000000000000000000000000000000111
2  00000000000000000000000000000000000000000000000000000000000000101
3  00000000000000000000000000000000000000000000000000000000000000001
```

Initial memory where the array is stored of length 3.

memory_dump.hex

```
1   // 0x00000000
2   0000000000000001
3   0000000000000005
4   0000000000000007
5   xxxxxxxxxxxxxxxx
6   xxxxxxxxxxxxxxxx
7   xxxxxxxxxxxxxxxx
8   xxxxxxxxxxxxxxxx
9   xxxxxxxxxxxxxxxx
10  xxxxxxxxxxxxxxxx
11  xxxxxxxxxxxxxxxx
12  xxxxxxxxxxxxxxxx
13  xxxxxxxxxxxxxxxx
14  xxxxxxxxxxxxxxxx
15  xxxxxxxxxxxxxxxx
16  xxxxxxxxxxxxxxxx
17  xxxxxxxxxxxxxxxx
18  // 0x00000010
```

updated memory with sorted array.



Challenges Faced

1. Instruction Fetch (IF) Challenges

Challenge: Handling PC updates correctly

- In normal execution, PC = PC + 4.
- For branch instructions (BEQ), **PC must update conditionally**, requiring proper control logic.
- Incorrect MUX control may lead to **wrong instruction fetch**, causing misalignment or incorrect branching.

2. Instruction Decode (ID) Challenges

Challenge: Control Unit

- The control unit must generate signals for ALU operations, memory access, register write-back, and Complexity branching.
- A wrong control signal (e.g., treating SD as an ADD) **can corrupt memory or registers**.

3. Execution (EX) Challenges

Challenge: ALU Control Complexity

- The ALU needs **different operations** based on funct7, funct3, and the instruction format.

- Errors in ALU control logic can result in **wrong computations**, affecting all instructions.

Challenge: Branch Comparison Logic

- **BEQ (if x1 == x2 then branch)** relies on ALU output (Zero flag).

4. Overall Challenges in Sequential Execution

Challenge: No Parallel Execution

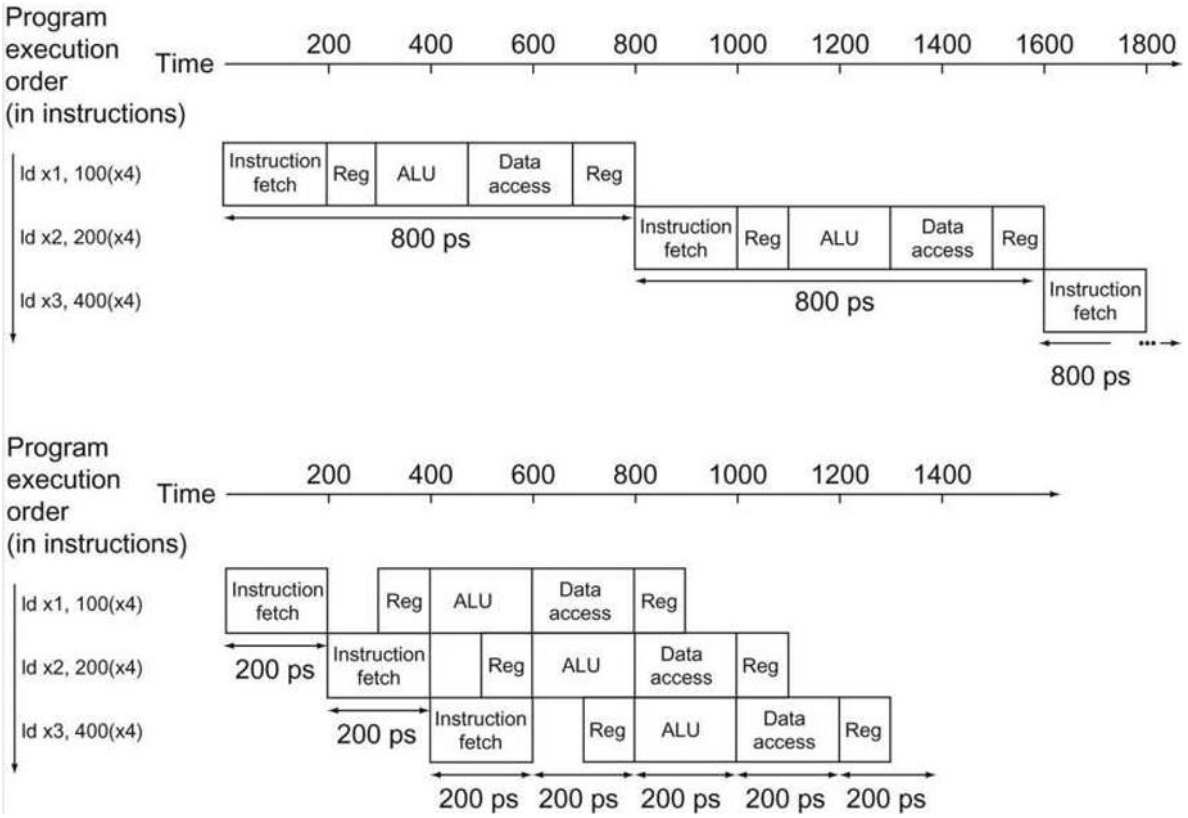
- Since the processor follows a strict **fetch → decode → execute → memory → write-back** cycle, **no two instructions can execute simultaneously**.
- Leads to **low performance compared to pipelined processors**.

Challenge: Debugging Incorrect Execution

- Errors in **PC update, ALU control, immediate decoding, or memory access** can result in completely incorrect execution.
- Debugging these requires careful observation of register/memory values **after each instruction**.

Task -2 Pipeline Implementation

A key feature of pipelining is that it increases the throughput of the system (i.e., the number of instructions per unit time), but it may also slightly increase the latency (i.e., the time required to complete an entire instruction)



Comparison

Time period of each stage= 200ps, assuimg 0 delay of pipeline registers, we have assumed a balance load for this scenario.

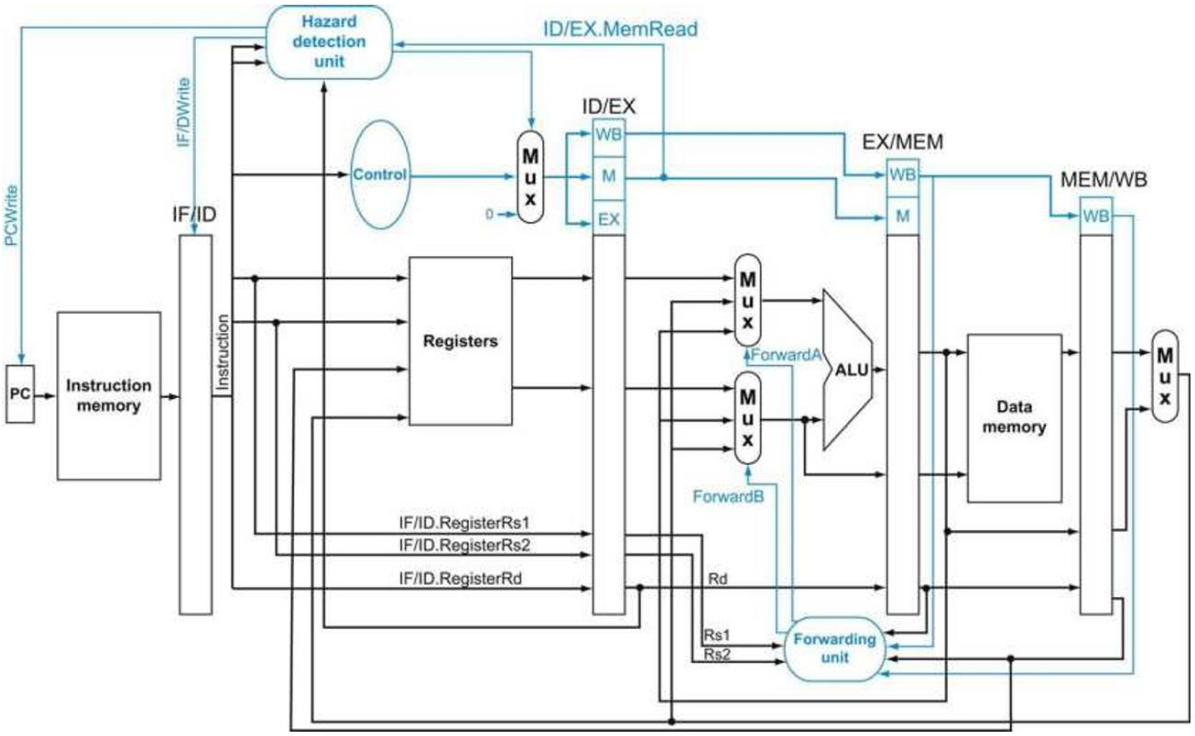
Throughput = 1/200ps

For sequential the throughput = 1/(200*5)ps

Latency remains the same if we ignore the register delay

PIPELINED RISC-V IMPLEMENTATION

Hardware implementation

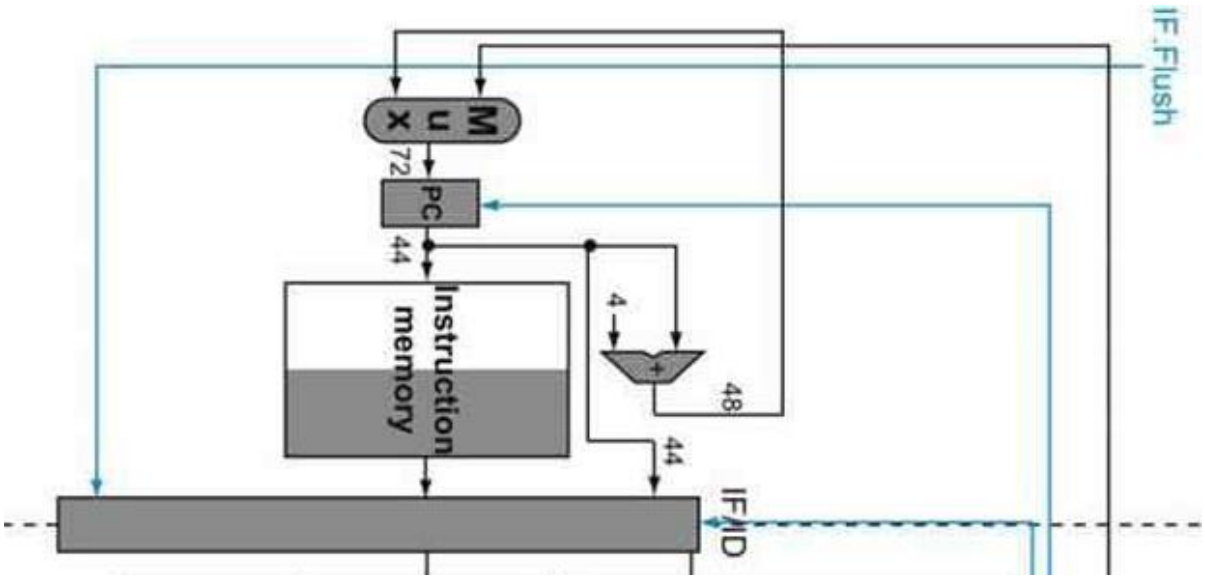


Example Execution in a Pipeline

Clock Cycle	Fetch (IF)	Decode (ID)	Execute (EX)	Memory (MEM)	Write Back (WB)
1	Instr 1	-	-	-	-
2	Instr 2	Instr 1	-	-	-
3	Instr 3	Instr 2	Instr 1	-	-
4	Instr 4	Instr 3	Instr 2	Instr 1	-
5	Instr 5	Instr 4	Instr 3	Instr 2	Instr 1

Here's a detailed breakdown of each stage of the pipeline with blocks, how they function, and how they handle hazards:

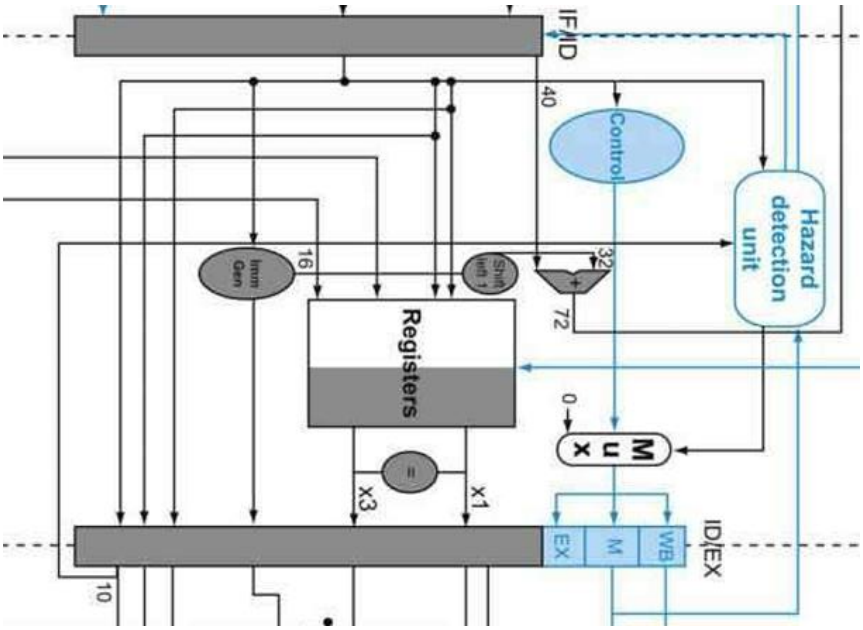
Instruction Fetch (IF) Stage



This stage retrieves the instruction from memory.

1. **Instruction Memory**
    - **Function:** Fetches the instruction from memory.
2. **PC Register**
    - **Function:** Holds the address of the next instruction to fetch.
3. **Adder (PC + 4)**
    - **Function:** Increments PC for sequential instruction fetching.
4. **MUX**
    - **Function:** Determines whether to continue sequential execution or jump to a branch.
5. **IF_FLUSH**
    - **Function:** Clears the IF/ID pipeline register when a control hazard (branch) occurs.Ensures that wrong instructions fetched after a mispredicted branch are discarded.

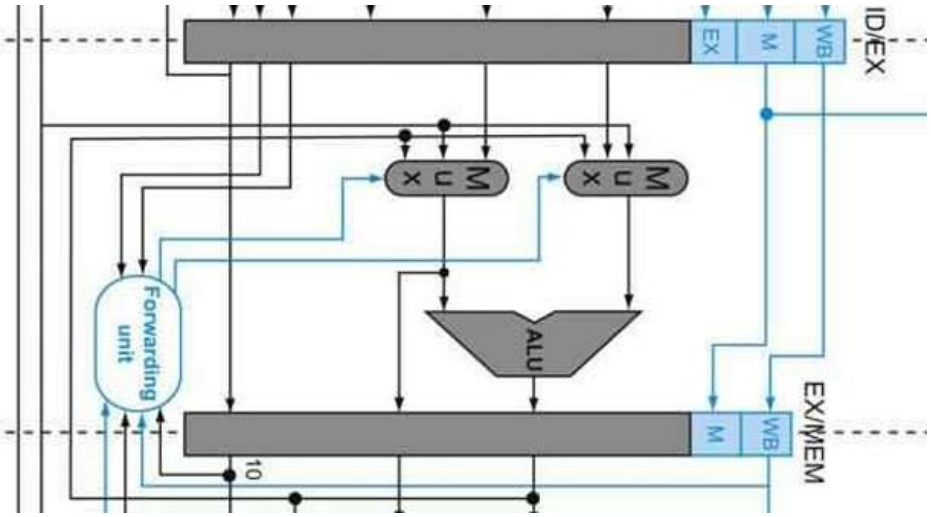
Instruction Decode (ID) Stage



This stage decodes the instruction and retrieves register values.

1. **Control Unit**
    - **Function:** Determines what type of instruction is being executed.
2. **Register File**
    - **Function:** Reads or writes register data.
3. **Immediate Generator**
    - **Function:** Extends immediate values for arithmetic/branching.
4. **Hazard Detection Unit**
    - **Input:** rs1, rs2, rd
    - **Output:** Stall (1-bit), Flush (1-bit)
    - **Function:** Detects data hazards and stalls pipeline when needed.
5. **Adder**
    - **Function:** Computes the new PC for branching.

Execute (EX) Stage

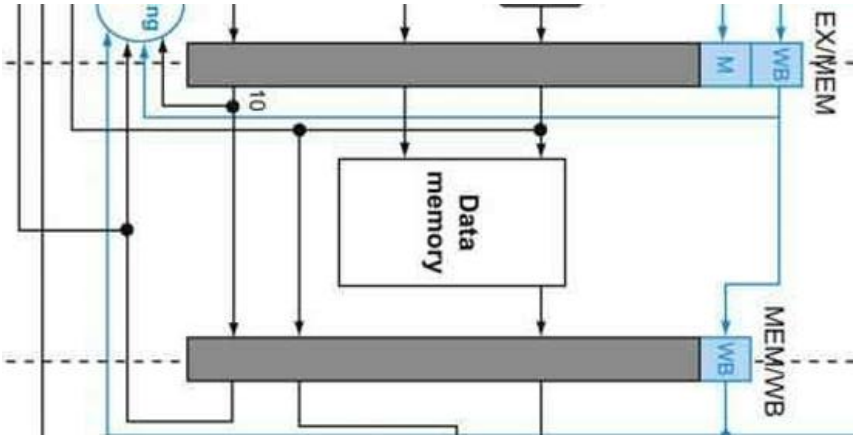


This stage performs arithmetic/logic operations and determines branch decisions.

1. **ALU Control**
    - **Function:** Determines ALU operation type.
2. **ALU**
    - **Function:** Executes arithmetic/logic instructions.
3. **Forwarding Unit**
    - **Inputs:** rs1, rs2, rd
    - **Outputs:** ForwardA (2-bit), ForwardB (2-bit)
    - **Function:** Resolves data hazards by forwarding values from later stages.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

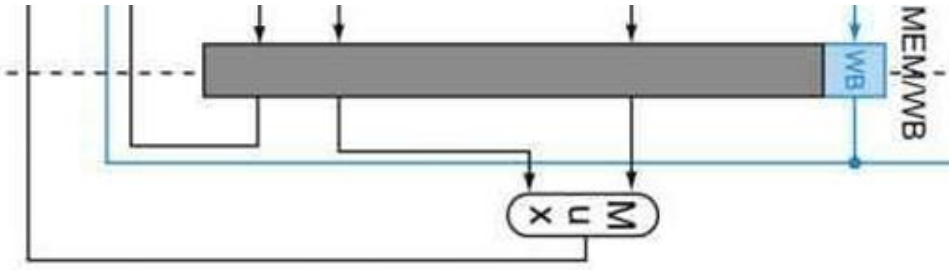
Memory Access (MEM) Stage



This stage handles memory operations.

1. **Data Memory**
    - **Function:** Loads/stores data from memory.

Write Back (WB) Stage



This stage writes data back to the register file.

1. **MUX**
    - **Inputs:**
        - ALU result
        - Memory Read Data

- **Control Signal:** MemToReg (1-bit)
- **Output:** Data to be written back to registers.
- **Function:** Chooses whether to write back memory or ALU output.

Summary of Pipeline Hazards and Solutions

Hazard Type	Description	Solution
Data Hazard	Instruction depends on previous instruction's result	Forwarding Unit, Stalls
Control Hazard	Incorrect instruction fetched due to branch	Branch Prediction, Flushing
Structural Hazard	Multiple instructions need the same resource	Use separate hardware units

---

HAZARDS

Structural Hazard in Pipeline Processors

A **structural hazard** occurs when multiple pipeline stages try to access the same hardware resource at the same time, leading to conflicts. The most common structural hazard in a basic processor pipeline happens when both the **Instruction Fetch (IF) stage** and **Memory (MEM) stage** need to access memory simultaneously.

Example of Structural Hazard

In a **single memory architecture**, both instruction and data are stored in the same memory. This means:

- The **Fetch stage (IF)** needs to read the next instruction from memory.
- The **Memory stage (MEM)** needs to read/write data from/to memory.

- If both stages try to access memory in the same cycle, a conflict occurs, stalling the pipeline.

Solution: Separate Instruction and Data Memory

To eliminate structural hazards,we use separate instruction memory and data memory.

How It Works

1. **Instruction Memory**
    - Dedicated memory unit for storing instructions.
    - Accessed only by the **Fetch (IF) stage**.
2. **Data Memory**
    - Separate memory unit for storing data.
    - Accessed only by the **Memory (MEM) stage**.

Since both units are separate, the processor can **fetch instructions and read/write data in the same cycle without conflicts**.

Illustration with Pipeline Stages

Without Separate Memories (Structural Hazard)

Cycle	IF (Fetch)	ID (Decode)	EX (Execute)	MEM (Memory)	WB (Write Back)
1	Inst 1	-	-	-	-
2	Inst 2	Inst 1	-	-	-
3	Inst 3	Inst 2	Inst 1	-	-
4	**Stall**	Inst 3	Inst 2	Inst 1 (MEM)	-
5	Inst 4	**Stall**	Inst 3	Inst 2 (MEM)	Inst 1 (WB)

- The pipeline stalls in **Cycle 4** because both **Fetch (IF)** and **Memory (MEM)** stages need memory access at the same time.

With Separate Memories (No Structural Hazard)

Cycle	IF (Fetch)	ID (Decode)	EX (Execute)	MEM (Memory)	WB (Write Back)
1	Inst 1	-	-	-	-
2	Inst 2	Inst 1	-	-	-
3	Inst 3	Inst 2	Inst 1	-	-
4	Inst 4	Inst 3	Inst 2	Inst 1 (MEM)	-
5	Inst 5	Inst 4	Inst 3	Inst 2 (MEM)	Inst 1 (WB)

- Since the **Instruction Memory (Fetch)** and **Data Memory (MEM stage)** are separate, there is **no conflict**.

Conclusion

By using **separate memory for instructions and data**, the processor can perform **simultaneous instruction fetching and memory accesses**, **eliminating structural hazards** and improving pipeline efficiency.

Data Hazards and Their Management

A **data hazard** occurs when an instruction depends on the result of a previous instruction that has not yet completed its execution. This can happen when instructions execute in a pipeline and require operands that are still being computed or written back.

Case 1: RAW - Read After Write

Example:

```
sub x2, x1, x3      // Register x2 set by sub
and x12, x2, x5     // 1st operand (x2) set by sub
or x13, x6, x2      // 2nd operand (x2) set by sub
add x14, x2, x2     // 1st (x2) & 2nd (x2) set by sub
sd x15, 100(x2)     // Index (x2) set by sub
```

- and x12, x2, x5: Needs x2 from sub (not yet available).
- or x13, x6, x2: Needs x2 from sub (not yet available).
- add x14, x2, x2: Needs x2 from sub (not yet available).
- sd x15, 100(x2): Needs x2 from sub (not yet available).

Since x2 is written by sub, all subsequent instructions using x2 create hazards.

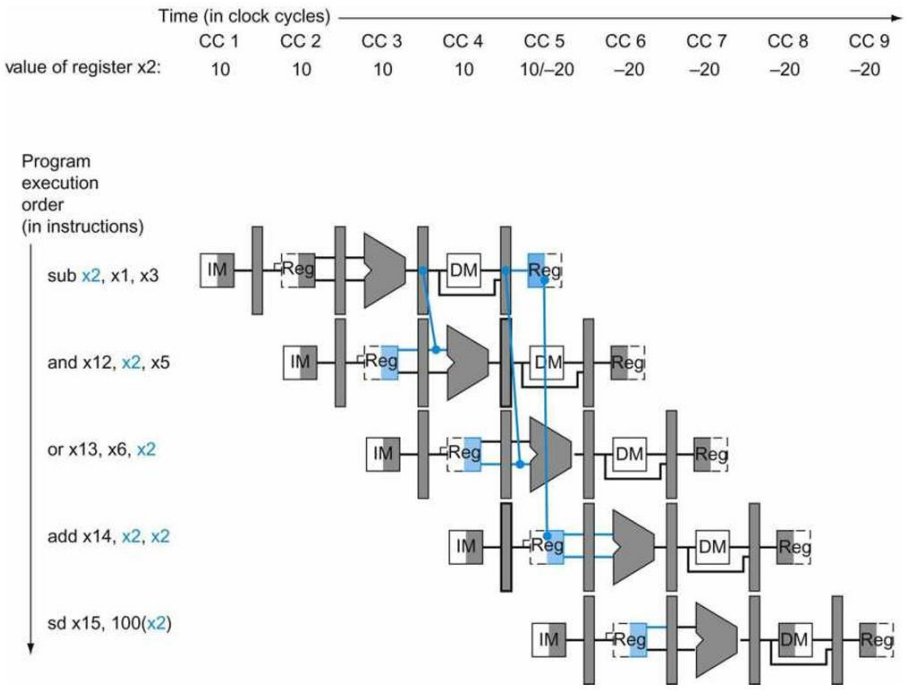
Solution: Forwarding (Bypassing)

- The **ALU forwarding unit** detects that x2 is being computed.
- Instead of waiting for x2 to be written to the register file, the result is directly forwarded from the ALU output of the first instruction to the ALU input of the second instruction and wherever needed by Forwarding Unit

How Forwarding Works:

1. First sub computes x2 in the **EX stage**.
2. Before x2 is written back, the **forwarding unit** sends the result to the and instruction in the next cycle.

Pipeline with Forwarding:

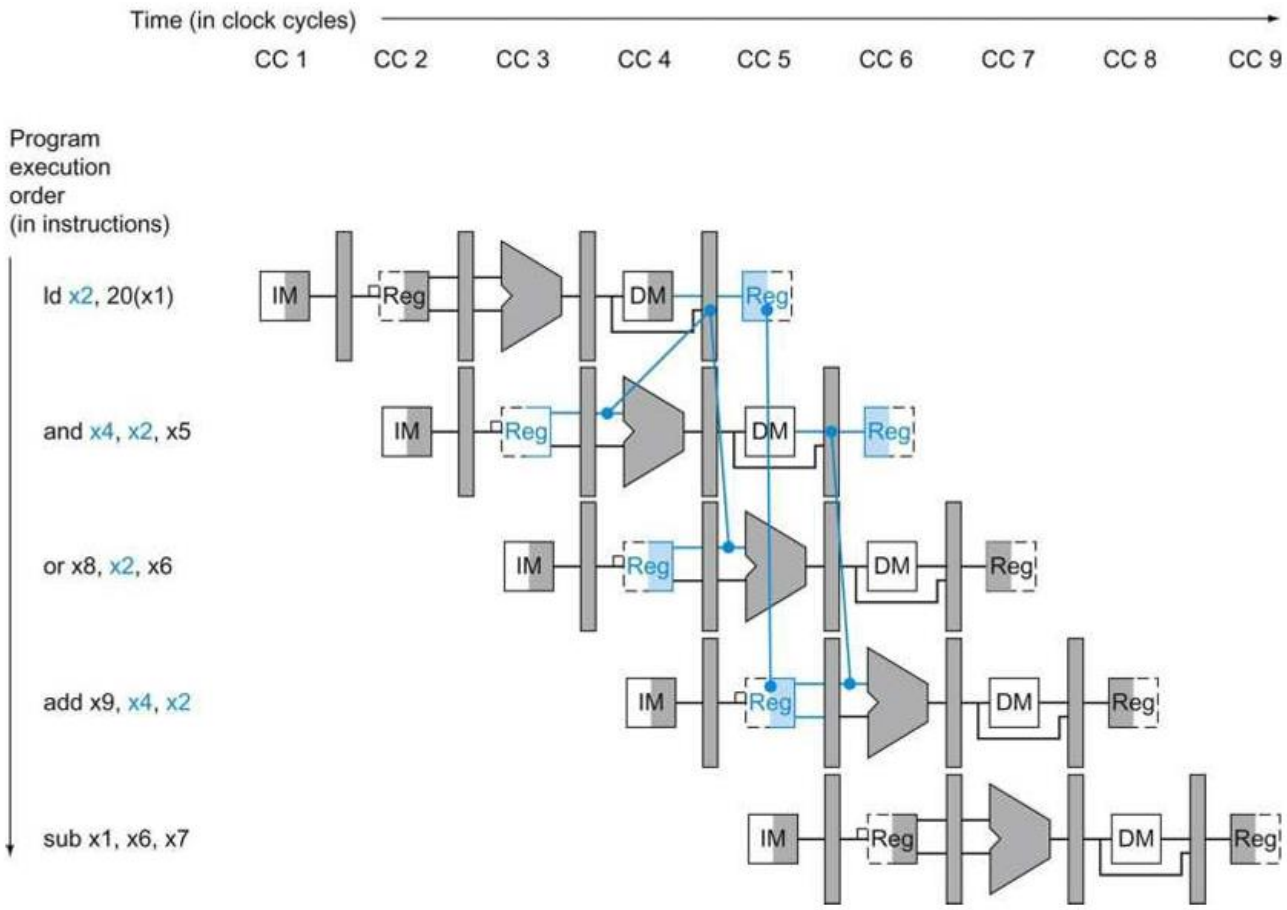


Case 2: LD-ADD Hazard (Load-Use Data Hazard)

Example:

```
ld x2, 20(x1)
and x4, x2,x5
or x8,x2, x5
add x9, x4, x2
sub x1, x6, x7
```

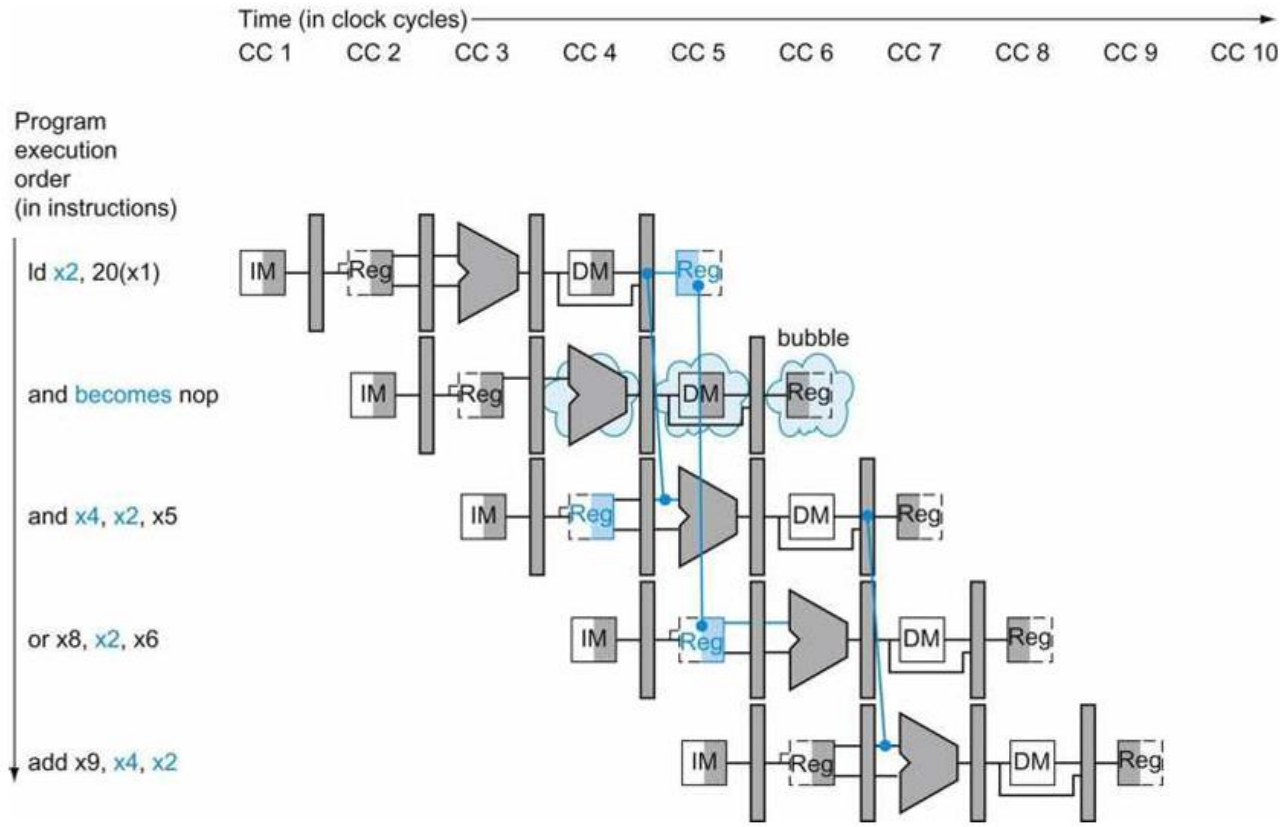
- The add instruction needs x2, but **x2 is not available until the MEM stage** of ld.
- The pipeline cannot use **forwarding** here, because the value is only available after memory access.



Solution: Hazard Detection Unit + Stall

- The **hazard detection unit** detects that x3 is not ready.
- It **inserts a stall (NOP)** for one cycle to wait for x3 to be available.

One cycle stall is introduced to ensure correct execution.



Control Hazards and Their Management

A **control hazard** (also called a **branch hazard**) occurs when the pipeline **does not know** the outcome of a branch instruction, and it may **fetch incorrect instructions** due to branch delay.

Example of Control Hazard

```
beq x1, x2, LABEL   # Branch if x1 == x2
add x3, x4, x5      # Next instruction (may be incorrect)
```

- The processor does **not know** whether to take the branch or not until the **EX stage**, but the next instruction (add x3, x4, x5) is already fetched.
- If the branch is **taken**, the fetched instruction is **wrong**, causing a hazard.

Techniques Used to Handle Control Hazards

1) Flushing

Flushing (also called pipeline clearing) is a technique where we remove incorrect or unnecessary instructions from the pipeline when a control hazard (such as a branch) occurs.

How It Works

- When a branch is taken, all instructions that were fetched after the branch (but before it was resolved) are invalid.
- We replace these instructions with **NOPs (no operation)** or reset the pipeline registers in those stages.
- This ensures that incorrect instructions don't affect execution.

Implementation in the Pipeline

- If a branch is taken, the **IF/ID** and **ID/EX** pipeline registers are cleared.
- The control unit issues a flush signal that invalidates the fetched instructions.

Advantages

✔ Ensures correctness by preventing wrong instructions from executing.
✔ Simple to implement.

TEST CASES

1) Basic operation we have implemented to show the working seperately

Assembly code

```
0           beq x0 x0 24
4           add x1 x1 x1
8           add x2 x2 x2
12          add x3 x3 x3
16          add x4 x4 x4
20          add x5 x5 x5
```

```
24  Target: add x7 x7 x7
```

Machine code

```
00000000000000000000110001100011
00000000001000010000000110110011
00000000001000010000000100110011
00000000001100011000000110110011
00000000010000100000001000110011
00000000010100101000001010110011
00000000011100111000001110110011
```

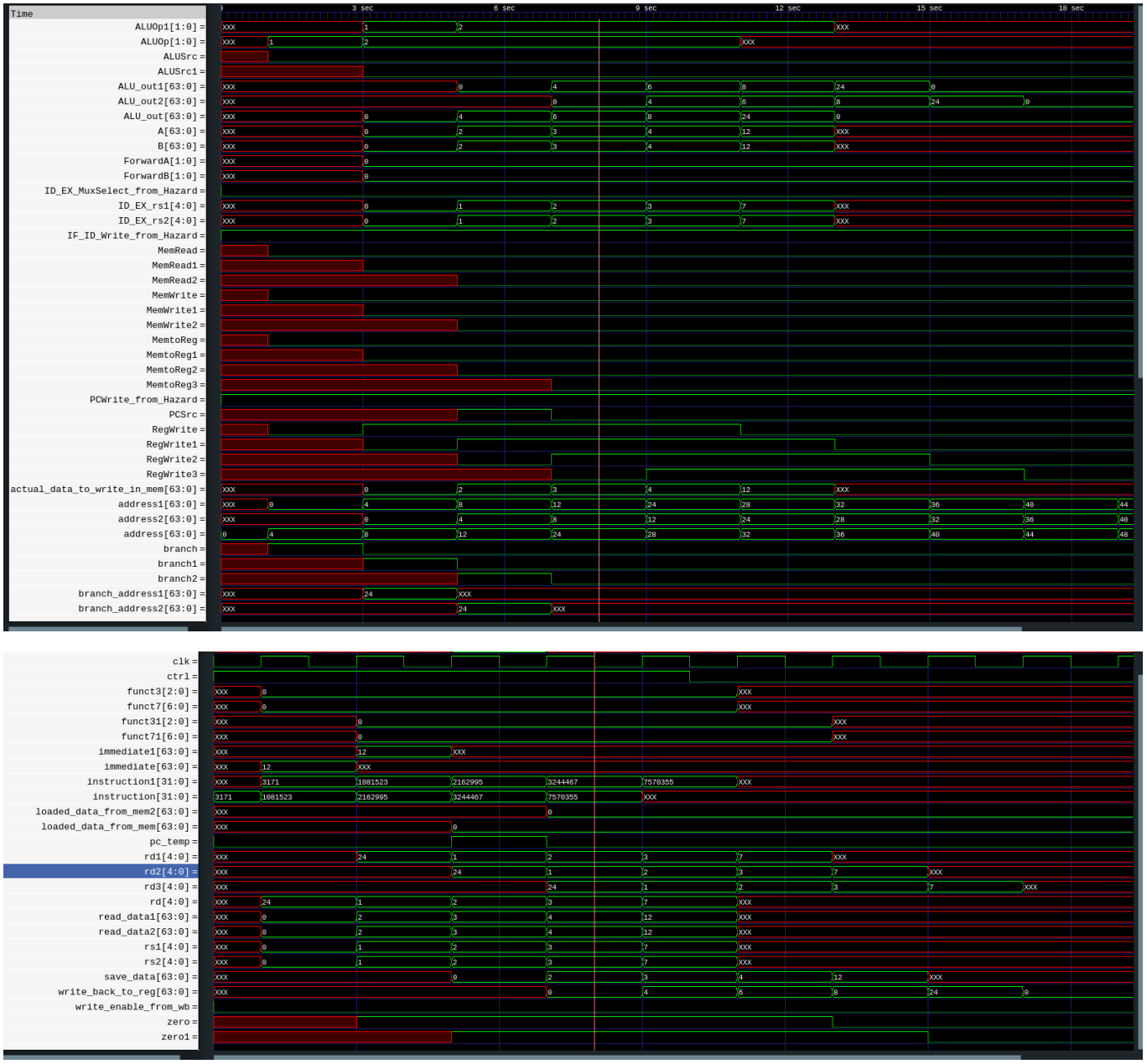
Result snapshot

```
21  initial begin
22      registers[1]=64'd2;
23      registers[2]=64'd3;
24      registers[3]=64'd4;
25      registers[4]=64'd6;
26      registers[5]=64'd10;
27      registers[7]=64'd12;
28
29  end
```

X1 x2 x3 x4 x5 x7 contains the element

```
1   // 0x00000000
2   XXXXXXXXXXXXXXXX
3   0000000000000004
4   0000000000000006
5   0000000000000008
6   0000000000000006
7   000000000000000a
8   XXXXXXXXXXXXXXXX
9   0000000000000018
10  XXXXXXXXXXXXXXXX
11  XXXXXXXXXXXXXXXX
12  XXXXXXXXXXXXXXXX
13  XXXXXXXXXXXXXXXX
14  XXXXXXXXXXXXXXXX
15  XXXXXXXXXXXXXXXX
16  XXXXXXXXXXXXXXXX
17  XXXXXXXXXXXXXXXX
18  // 0x00000010
```

updated memory. Beq is working fine perfectly.



2) Finding sum of all the elements in an array

Assembly code

```
ld x5, 0(x2)
ld x6, 1(x2)
ld x7, 2(x2)
ld x8, 3(x2)
ld x9, 4(x2)
```

```
add x12, x5, x6
add x11, x7, x8
// double data hazard
add x4, x12, x11
add x4, x4, x9
sd x4, 0(x3)
ld x10, 0(x3)
add x10, x10, x10
sd x10, 0(x3)
```

Machine code

```
000000000000_00010_011_00101_0000011
000000000001_00010_011_00110_0000011
000000000010_00010_011_00111_0000011
000000000011_00010_011_01000_0000011
000000000100_00010_011_01001_0000011
0000000_00110_00101_000_01100_0110011
0000000_01000_00111_000_01011_0110011
0000000_01011_01100_000_00100_0110011
0000000_01001_00100_000_00100_0110011
0000000_00100_00011_011_00000_0100011
000000000000_00011_011_01010_0000011
0000000_01010_01010_000_01010_0110011
0000000_01010_00011_011_00000_0100011
```

Result snapshot

```
21    initial begin
22       registers[3]=64'd7;
23       registers[2]=64'd0;
24
25    end
```

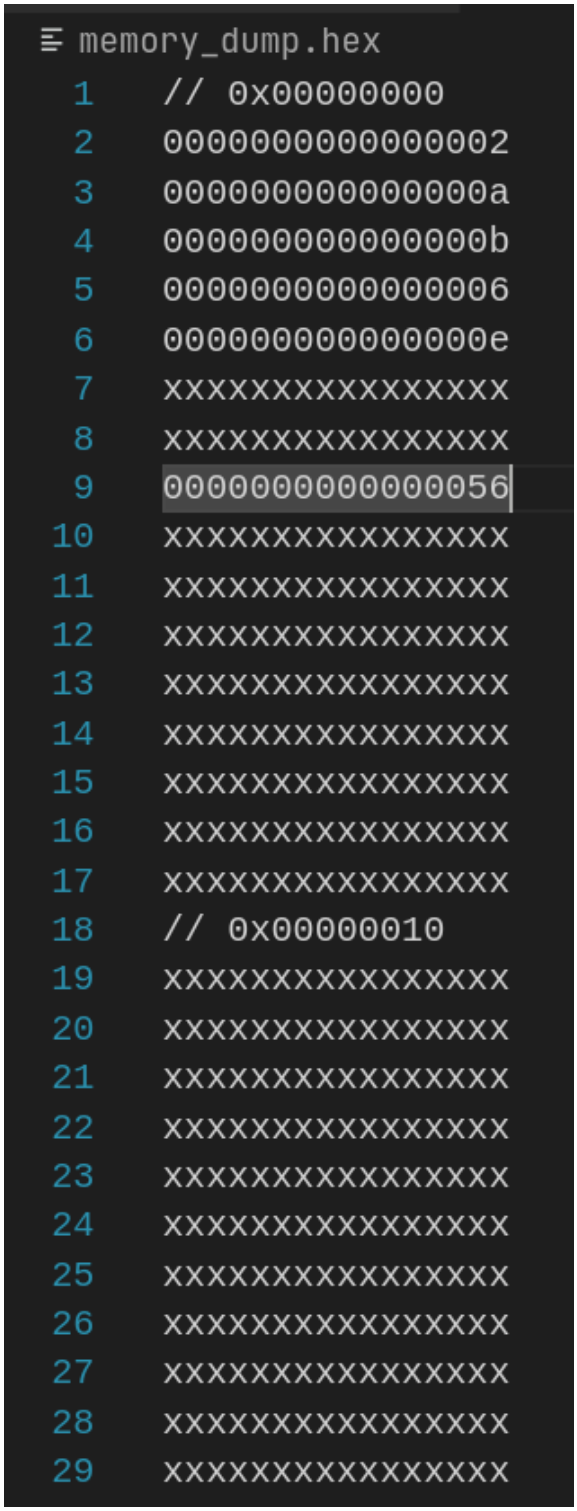
X2 contains the base address of the array

X3 contains the address where the ans to be stopped.

initial_data.txt

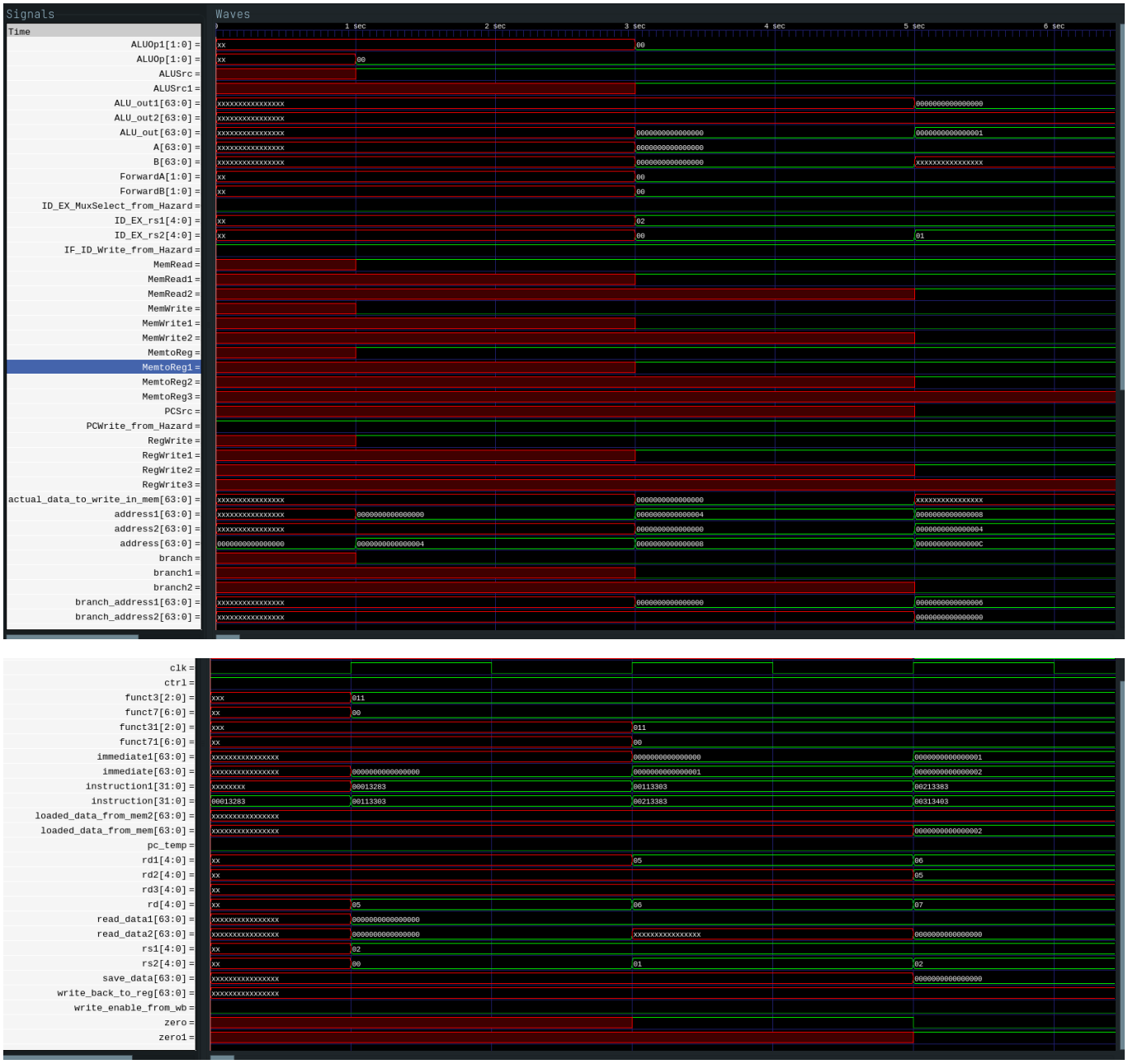
```
1  0000000000000000000000000000000000000000000000000000000000000010
2  0000000000000000000000000000000000000000000000000000000000001010
3  0000000000000000000000000000000000000000000000000000000000001011
4  0000000000000000000000000000000000000000000000000000000000000110
5  0000000000000000000000000000000000000000000000000000000000001110
6
```

Initial memory with the input array.



Updated memory.

X9 contains the ans (Sum of all elements of an array)



3) Finding sum of all elements of an array (Load hazard)

Assembly code

```
// data hazard with load instruction
ld x5, 0(x2)
ld x6, 1(x2)
add x4, x5, x6
ld x7, 2(x2)
add x4, x4, x7
ld x8, 3(x2)
add x4, x4, x8
ld x9, 4(x2)
add x4, x4, x9
sd x4, 0(x3)
```

```
ld x10, 0(x3)
add x10, x10, x10
sd x10, 0(x3)
```

Machine code

```
00000000000000010011001010000011
00000000000100010011001100000011
00000000011000101000001000110011
00000000000100001001110010000011
00000000011100100000001000110011
00000000000110001001101000000011
00000000100000100000001000110011
00000000010000010011010010000011
00000000100100100000001000110011
00000000010000011011000000100011
00000000000000001101101010000011
00000000101001010000010100110011
00000000101000011011000000100011
```

Result snapshot

```
20
21    initial begin
22        registers[2]=64'd0;
23        registers[3]=64'd15;
24
25    end
26
```

X2 contains the base address of the array in memory.

initial_data.txt

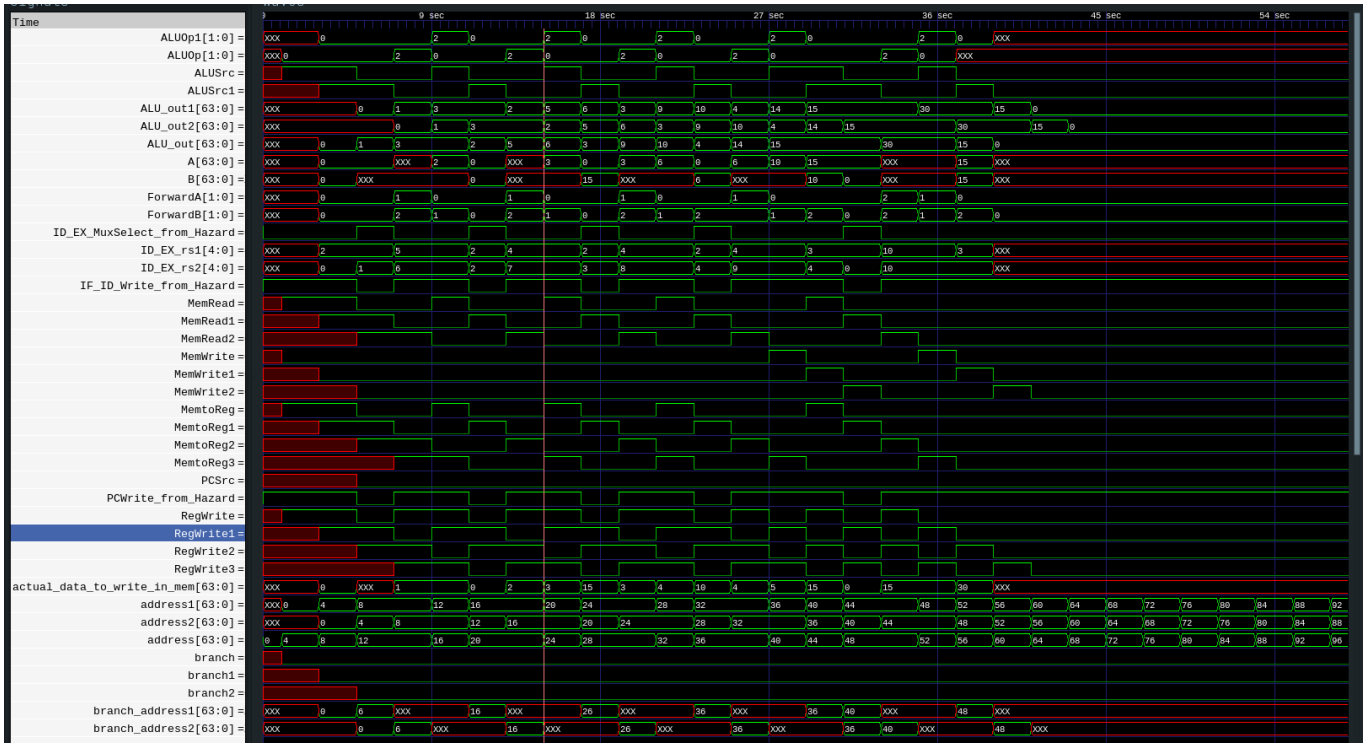
```
1   00000000000000000000000000000000000000000000000000000000000000010
2   00000000000000000000000000000000000000000000000000000000000000001
3   00000000000000000000000000000000000000000000000000000000000000011
4   00000000000000000000000000000000000000000000000000000000000000100
5   00000000000000000000000000000000000000000000000000000000000000101
```

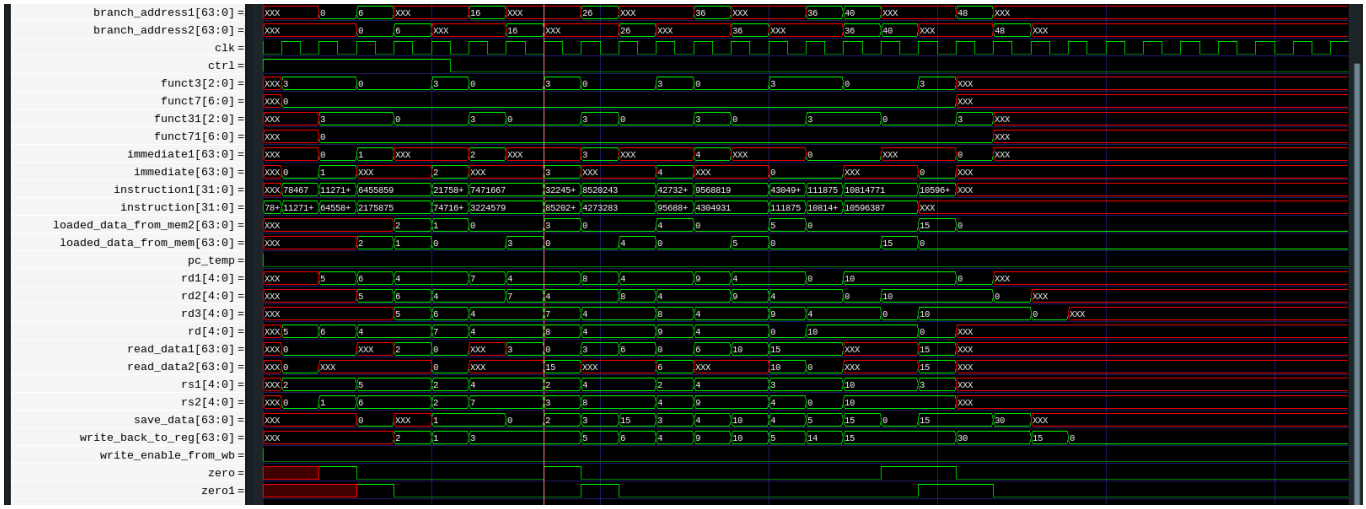
Initial memory with input array

memory_dump.hex

```
1   // 0x00000000
2   0000000000000002
3   0000000000000001
4   0000000000000003
5   0000000000000004
6   0000000000000005
7   xxxxxxxxxxxxxxxx
8   xxxxxxxxxxxxxxxx
9   xxxxxxxxxxxxxxxx
10  xxxxxxxxxxxxxxxx
11  xxxxxxxxxxxxxxxx
12  xxxxxxxxxxxxxxxx
13  xxxxxxxxxxxxxxxx
14  xxxxxxxxxxxxxxxx
15  xxxxxxxxxxxxxxxx
16  xxxxxxxxxxxxxxxx
17  000000000000001e
18  // 0x00000010
19  xxxxxxxxxxxxxxxx
20  xxxxxxxxxxxxxxxx
```

Updated memory with ans stored at location mem[15]





4) Basic operation on two registers

Assembly code

```
ld x5, 0(x2)
ld x6, 1(x2)
add x10, x5, x6
sub x11, x5, x6
and x12, x5, x6
or x13, x5, x6
sd x10 2(x2)
sd x11 3(x2)
sd x12 4(x2)
sd x13 5(x2)
```

Machine code

```
00000000000000001000001010000011
00000000000010001000001100000011
00000000010100110000010100110011
01000000010100110000010110110011
00000000010100110111011000110011
00000000010100110110011010110011
00000000101000010000000100100011
00000000101100010000000110100011
00000000110000010000001000100011
00000000110100010000001010100011
```

Result snapshot

```
21    initial begin
22        registers[2]=64'd0;
23
24    end
25
```

X2 contains the base address of the array in memory.

initial_data.txt

```
1  00000000000000000000000000000000000000000000000000000000000000010
2  00000000000000000000000000000000000000000000000000000000000001010
```

Initial memory with input array

memory_dump.hex

```
1   // 0x00000000
2   0000000000000002
3   000000000000000a
4   000000000000000c
5   0000000000000008
6   0000000000000002
7   000000000000000a
8   xxxxxxxxxxxxxxxx
9   xxxxxxxxxxxxxxxx
10  xxxxxxxxxxxxxxxx
11  xxxxxxxxxxxxxxxx
12  xxxxxxxxxxxxxxxx
13  xxxxxxxxxxxxxxxx
14  xxxxxxxxxxxxxxxx
15  xxxxxxxxxxxxxxxx
16  xxxxxxxxxxxxxxxx
17  xxxxxxxxxxxxxxxx
18  // 0x00000010
```

Updated memory.

Mem[0]=2

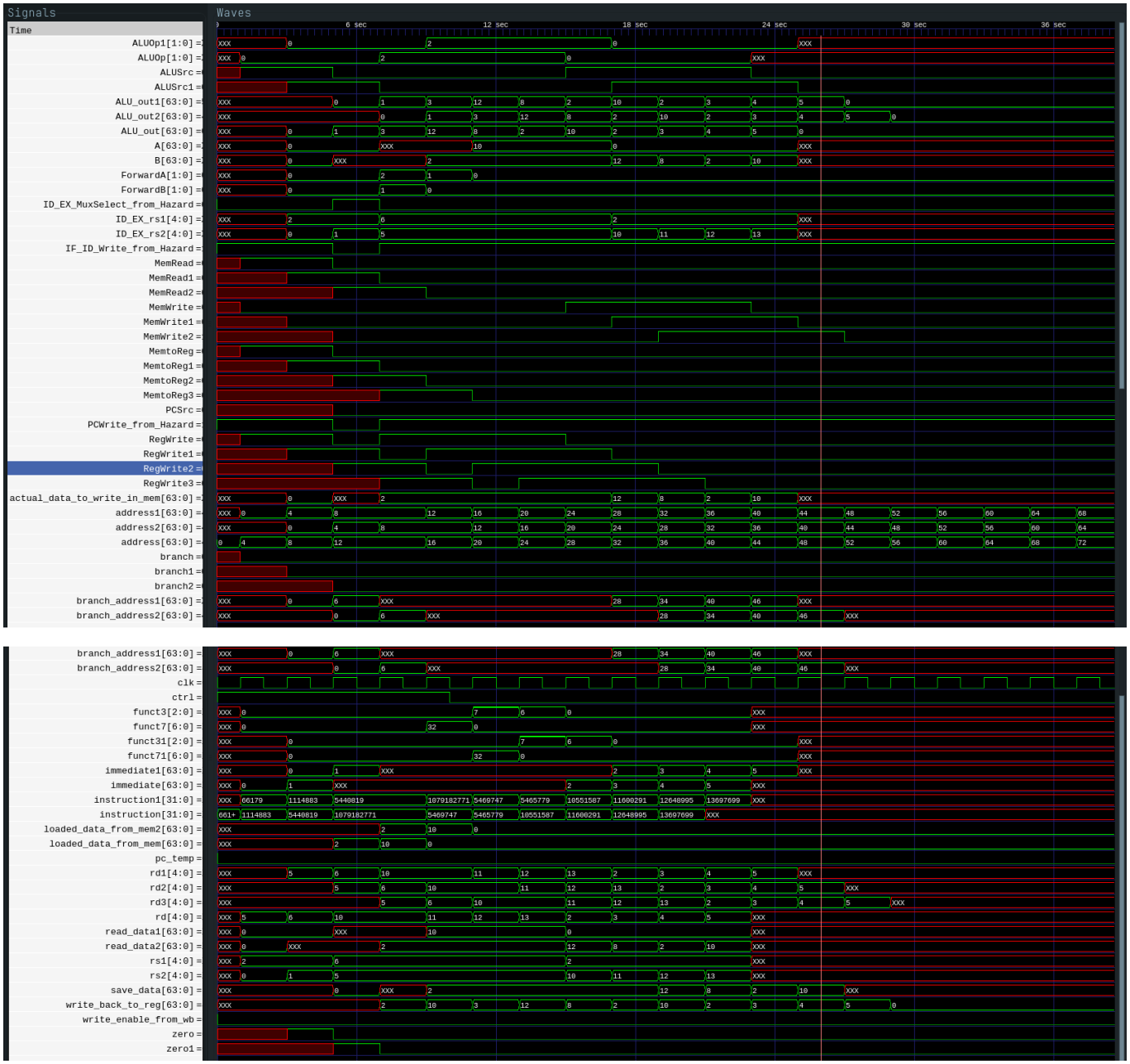
Mem[1]=10(a)_

Mem[2]=c(12) add 10 and 2

Mem[3]=8 in hex sub 10 and 2

Mem[4]=2 (2 and 10)

Mem[5]=10 (2 or 10)



Challenges FACED

1) Fetching the First Instruction & Handling Signal Dependencies

When fetching the first instruction, the required signals from later pipeline stages (such as Execute, Memory, and Writeback) might not yet be available. This creates a problem since some of these signals are necessary for correct execution, such as forwarding signals or hazard detection signals.

How we handled it:

- Introduce default values for control signals to prevent undefined behavior.
- Use registers/latches to store values and forward them when needed.
- Ensure proper initialization before pipeline execution starts.

2) Modularizing the Code by Introducing Separate Modules for Each Pipeline Stage

A well-structured pipeline processor should have individual modules for each stage (Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback).

3) Handling Control Signals for Hazard Detection for Each Type of Hazard

Pipeline hazards can be classified into:

- **Data Hazards:** Occur when an instruction depends on the result of a previous instruction that hasn't completed yet.
- **Control Hazards:** Arise from branch instructions where the next instruction’s execution depends on the branch outcome.
- **Structural Hazards:** Occur when hardware resources are not sufficient to handle multiple instructions simultaneously.

Steps taken to Resolve:

- **Forwarding (Bypassing):** Instead of waiting for the register writeback, the value is forwarded from the execution stage.
- **Stalling (NOP Insertion):** Delay execution until dependencies are resolved.

4) Challenges Faced While Setting Up Hazard Handling Mechanisms

- Determining the exact cycle when hazards occur.
- Implementing stall logic correctly without causing deadlocks.
- Optimizing the pipeline to reduce the performance impact of hazards.

- Handling complex scenarios like load-use hazards and control flow changes efficiently.

5) Limited Resources Available Online

Unlike standard topics in Verilog, pipeline implementations and hazard handling mechanisms are not as well documented online. The available examples are often simplified and don't cover real-world challenges like:

- Implementing forwarding paths in a realistic way.
- Handling control hazards with actual branch prediction.
- Writing a thorough testbench to validate all possible hazard cases.

6) Designing a Comprehensive Testbench for Various Operations & Hazards

A good testbench should:

- Cover normal execution scenarios.
- Include all types of hazards (data, control, and structural).
- Test edge cases like pipeline stalls, branch mispredictions, and forwarding logic.
- Use **$dumpfile and $dumpvars** to generate waveform files for debugging.

Key Features to Implement in the Testbench:

- Randomized instruction sequences.
- Manual test cases to verify each hazard.
- Checking if output matches expected results.

7) Debugging the Code & Analyzing Signals in GTKWave

- **Identifying Signal Mismatches:** Check if control signals are generated at the right time.
- **Monitoring Pipeline Register Contents:** Ensure each instruction progresses correctly through the stages.

- **Verifying Hazard Handling Logic:** Look for unexpected stalls or incorrect values being forwarded.
- **Comparing Expected vs. Actual Behavior:** Use assertions in the testbench to verify correctness.

Using **GTKWave**, focus on:

- Control signals like stall, flush, and forward.
- Pipeline register values at different stages.
- Branch resolution and misprediction corrections

8) Choosing the Pipeline Design Architecture

Selecting the right architecture for the pipeline was one of the key challenges. We had to decide:

- **Inter-stage registers**: What information to store in each pipeline register to ensure smooth data flow.
- **Forwarding logic placement**: Whether to handle forwarding in EX, MEM, or WB stages.

A poorly chosen architecture could lead to inefficient stalls or unnecessary complexity in hazard handling.

9) Deciding What Elements to Place in Each Pipeline Stage (IF, ID, EX, MEM, WB) to Handle Data Hazards

A major challenge was determining which signals and registers should be included in each stage to efficiently resolve data hazards. Incorrectly placing signals could lead to:

- Unnecessary stalls.
- Complex forwarding logic.
- Incorrect instruction execution due to timing mismatches.

Key considerations:

- **IF Stage**: Should fetch instructions and update PC without being affected by stalls in later stages.
- **ID Stage**: Should handle register read operations carefully and detect dependencies early.
- **EX Stage**: Should implement forwarding paths to avoid unnecessary stalls.
- **MEM Stage**: Should only introduce clock cycles when absolutely necessary, such as for memory accesses.
- **WB Stage**: Should write results back without causing write-after-read (WAR) hazards.

10) Choosing the Best Way to Handle Control Hazards for Running the Sorting Algorithm Efficiently

Sorting algorithms involve frequent comparisons and branches. We had to determine the best method to handle control hazards without slowing down execution. The main choices were:

- **Stalling** (simplest but slows performance).
- **Branch Prediction** (faster but increases complexity).
- **Flushing the pipeline** (wastes cycles but ensures correctness).

After testing, we aimed to optimize branch handling for better sorting performance, ensuring minimal stalls

Contribution of each team member

1) **Harshit goyal** : Project Report, ALU Unit, test_benches, forwarding unit, hazard detection unit, Test cases and Algorithm Generation and Testing. Assembly to Machine decoding, Making Python scripts for hex to binary and vice versa and Debugging.

2) **Nitin grandhi**: Sequential Control Unit ,ALU Control ,Immediate Generation ,Merging of module units into IF,ID,EX,MEM,WB and Wrapper for pipeline, modularity of the code, modification of sequential to pipeline,Handling control hazards

3) **Manikya Pant**: Basic functional units (such as memory, register file, Instruction fetch), wrapper for sequential, debugging each stage of both sequential and pipeline ,Handling control hazards,Debugging Structural Hazard which saved Data Hazard.

- *Thank You*!