IPA Project Report
Sequential and Pipelined Processor

Roshan Kondabattini
2023102061

Madhan Sai Krishna
2023102030

Kaamya Dasika
2023102034

March 16, 2025

This processor is developed based on the book ***Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition,*** where a sequential processor is developed first. Then, pipeline registers are added to make it a pipelined processor. After that, a forwarding unit is implemented, followed by the deployment of a hazard detection unit, which eliminates data hazards and load-use data hazards.

Sequential RISC V Processor

This project involves the development of a RISC-V processor capable of executing a subset of instructions: ADD, SUB, AND, OR, LD, SD, and BEQ. The processor follows a five-stage pipeline design, which includes:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write-Back (WB)

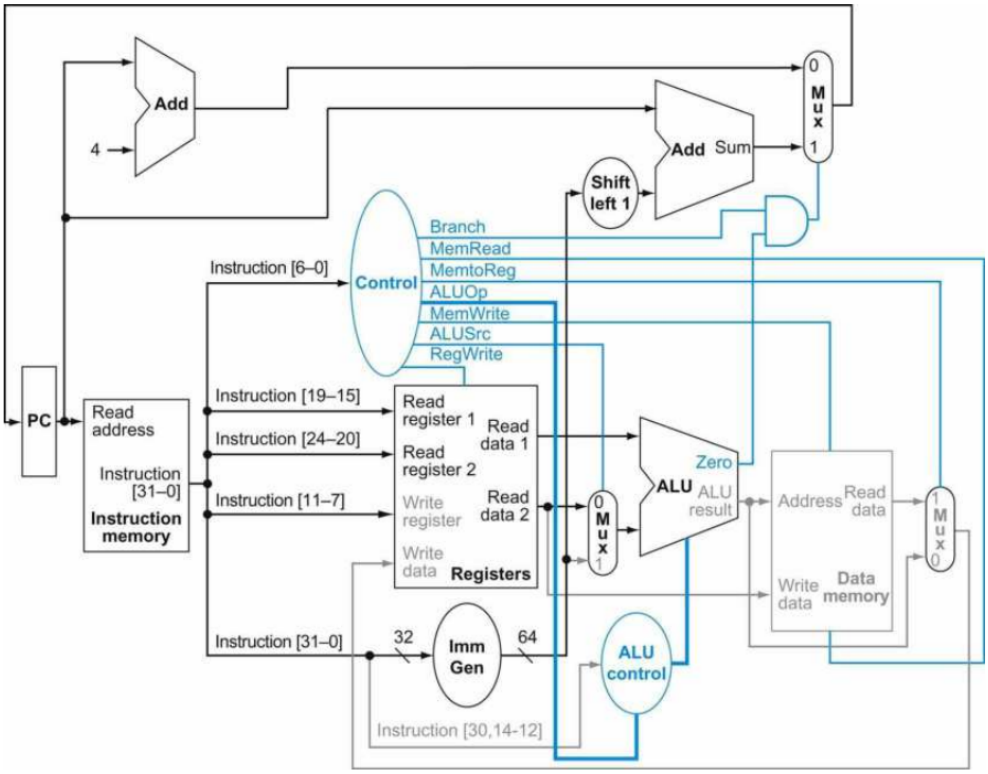

Figure 1: Full sequential figure without breaking into stages

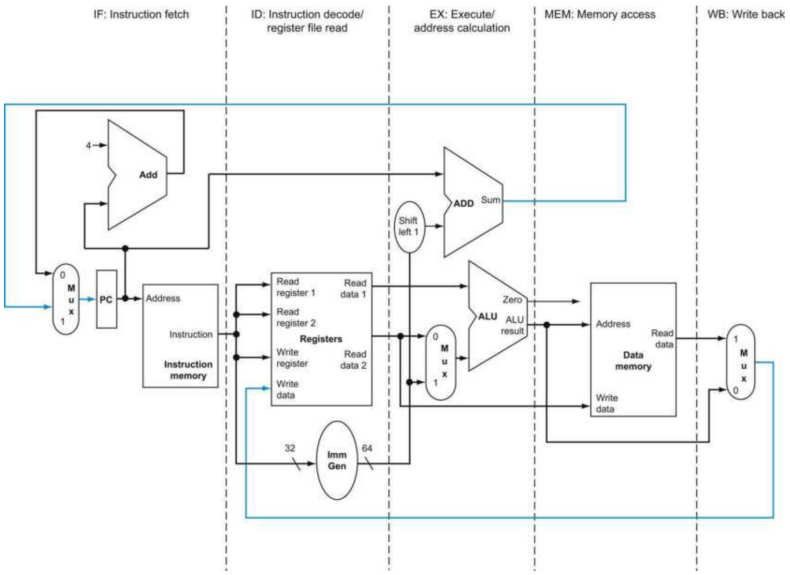

Figure 2: Sequential Figure

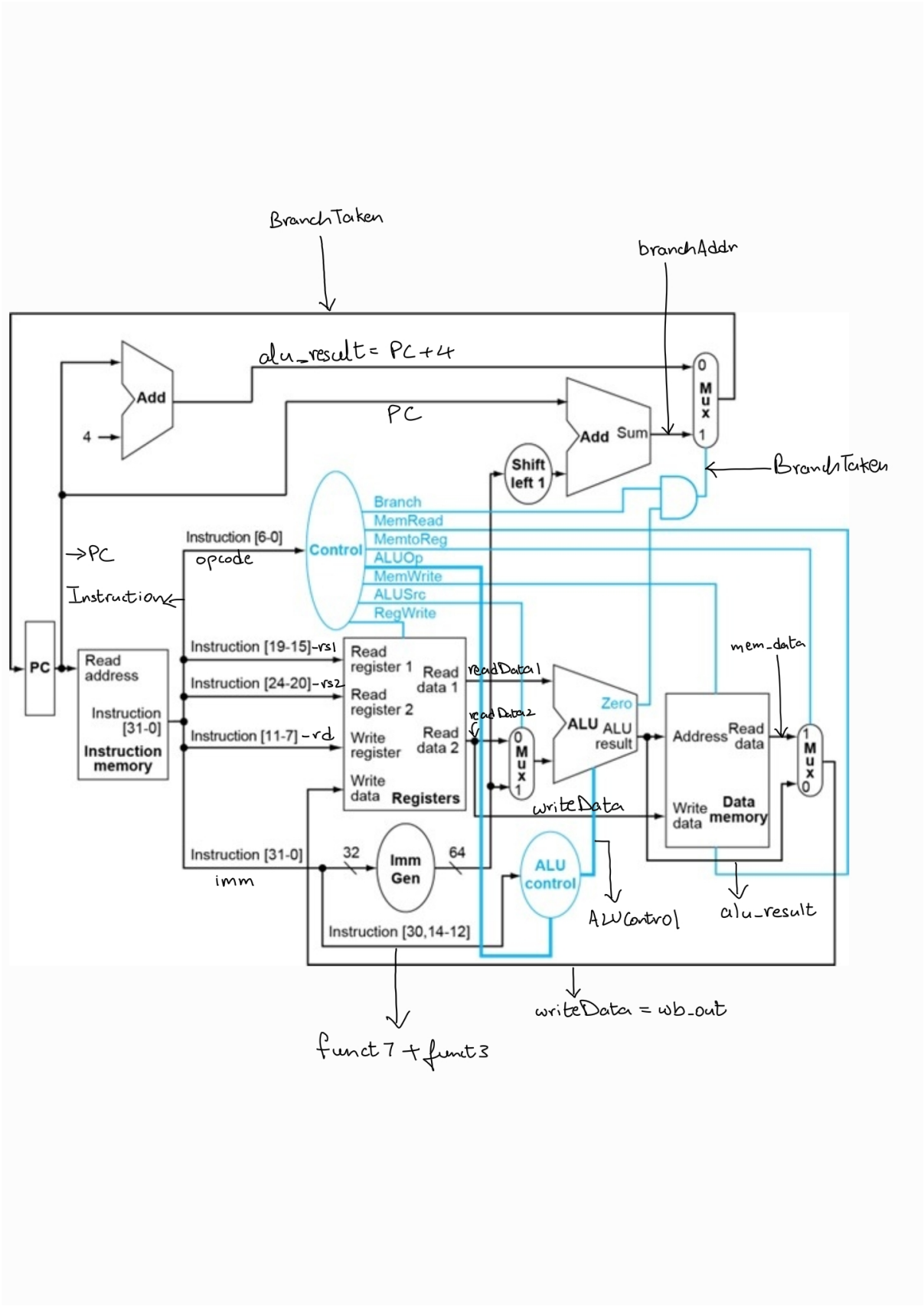

Figure 3: Wiring reference that matches our code

Instruction Fetch

1. The Instruction Fetch (IF) stage includes an instruction memory that can store up to 16 instructions at maximum.
2. An ALU is used to add 4 to the program counter (PC) to fetch the next instruction.
3. Based on the BranchTaken signal, the PC is updated with either PC + 4 (next sequential instruction) or the target branch address (if a branch is taken)
4. **Note** The Program Counter (PC) is always a multiple of 4, ensuring proper instruction alignment in memory.

   When PC = 0, it fetches the first instruction. When PC = 4, it fetches the second instruction, and so on.

   This ensures that each instruction is correctly aligned in memory, as RISC-V instructions are 32-bit (4 bytes) wide.

Instruction Decoder

The Instruction Decode (ID) stage consists of several key components:

1. A decoder that takes in the instruction and, based on the opcode, categorizes it into various R-type, I-type, and S-type instructions.
2. A register file containing 32 registers, each 64 bits wide, used for operations (fetching and writing back registers).
3. An immediate generator (ImmGen) that extracts the immediate values from the instruction and sign-extends them to 64 bits.
4. A control unit, which generates 7 control signals based on the opcode to guide instruction execution.

Execute Stage

1. An adder that adds the PC and the shifted immediate value.

   (Note: The immediate value must be a multiple of 2 to maintain proper alignment.)
2. A multiplexer (MUX) that determines the ALU's second input:

   It selects either the immediate value (for memory addressing) or another register value.
3. The main ALU, which performs the primary operations, including:

   Executing R-type instructions (e.g., ADD, SUB, AND, OR).
4. Computing memory addresses for LD and SD.
5. A secondary ALU, which performs a shift left by 1 (SLL1) operation.
6. The main ALU generates a zero signal, which is used for branch condition checking:

   This is achieved using an AND gate that combines the Branch signal and the Zero signal.
7. An ALU Control Unit, which maps instructions to their corresponding ALU operations (as shown in the reference figure).

Memory Stage

1. The data memory consists of 32 registers (custom-defined) that are byte-addressed.
2. To access mem[i], the ALU output must be $8 \times i$ (ensuring aligned memory access).
3. In LD rd, imm(r1), both imm and the value in r1 must be multiples of 8.
4. A memory read operation is performed, based on the control signals.

Write Back

Writeback happens in negative edge (to resolve structural hazard)

Working

Refer to this LINKin order to see terminal outputs with hex files and gtkwave.

Pipelining

An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

RISC-V instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

To convert the sequential stages into a pipelined version, we add pipeline registers at IF/ID, TD/EX, EX/MEM, MEM/WB. They store the following :

1. IF/ID:
   (a) **IF_ID_PC**: The program counter value at the instruction fetch stage.
   (b) **IF_ID_instruction**: The fetched instruction.
2. ID/EX:
   (a) **ID_EX_RegWrite**: Control signal for register write.
   (b) **ID_EX_MemtoReg**: Control signal indicating if data comes from memory.
   (c) **ID_EX_Branch**: Control signal for branch instruction.
   (d) **ID_EX_MemRead**: Control signal for memory read operation.
   (e) **ID_EX_MemWrite**: Control signal for memory write operation.
   (f) **ID_EX_ALUSrc**: Control signal selecting ALU source operand.
   (g) **ID_EX_ALUOp**: ALU operation control bits.
   (h) **ID_EX_PC**: Program counter value passed to the next stage.
   (i) **ID_EX_readData1**: First source register value.
   (j) **ID_EX_readData2**: Second source register value.
   (k) **ID_EX_imm**: Immediate value extracted from the instruction.
   (l) **ID_EX_funct7**: Funct7 field from the instruction.
   (m) **ID_EX_funct3**: Funct3 field from the instruction.
   (n) **ID_EX_rd_addr**: Destination register address.
3. EX/MEM:
   (a) **EX_MEM_RegWrite**: Control signal for register write.
   (b) **EX_MEM_MemtoReg**: Control signal for memory to register operation.
   (c) **EX_MEM_Branch**: Control signal for branch instruction.
   (d) **EX_MEM_MemRead**: Control signal for memory read.
   (e) **EX_MEM_MemWrite**: Control signal for memory write.

(f) **EX_MEM_PCPlusImmShifted**: Computed branch address.

(g) **EX_MEM_Zero**: Zero flag from ALU.

(h) **EX_MEM_ALUResult**: Result computed by the ALU.

(i) **EX_MEM_readData2**: Second source register value, used for memory writes.

(j) **EX_MEM_rd_addr**: Destination register address.

4. MEM/WB:

(a) **MEM_WB_RegWrite**: Control signal for register write.

(b) **MEM_WB_MemtoReg**: Control signal for selecting memory or ALU output.

(c) **MEM_WB_mem_readData**: Data read from memory.

(d) **MEM_WB_ALUResult**: Result computed by ALU.

(e) **MEM_WB_rd_addr**: Destination register address.

Note that , to the subsequent stage, pipelined values are passed. The main advantage of using a pipelined register is that it can run 5 instructions at once.

Hazard Type	Cause	Example	Solution
Load-Use	Instruction needs a value from memory before it's available	`LW R2, 0(R1)` → `ADD R3, R2, R4`	Pipeline stall (or forwarding if possible)
Structural	Two instructions compete for the same hardware resource	`LW` and `SW` both need memory	Separate instruction & data memory
Forwarding	Data dependency on a previous instruction	`ADD R2, R1, R3` → `SUB R4, R2, R5`	Forward data from EX/MEM pipeline register
Branch (BEQ)	Uncertainty about branch outcome	`BEQ R1, R2, LABEL` → next instruction may be wrong	Branch prediction, delay slots

Figure 4: Hazards,Example and Solution

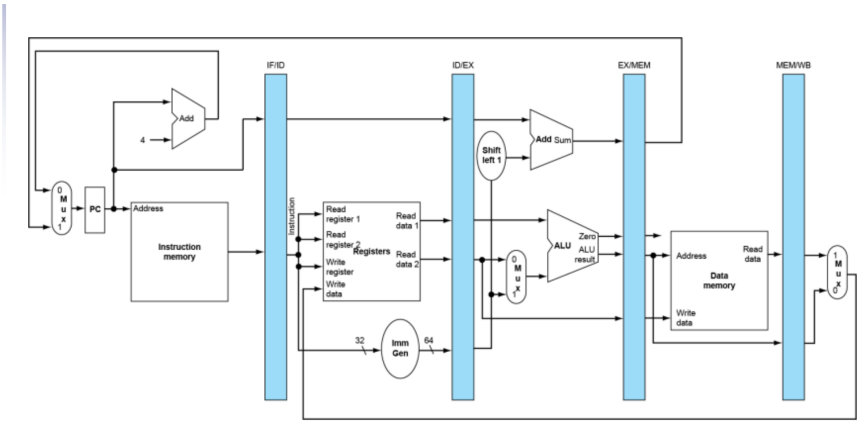

Figure 5: circuit with pipelined registers.

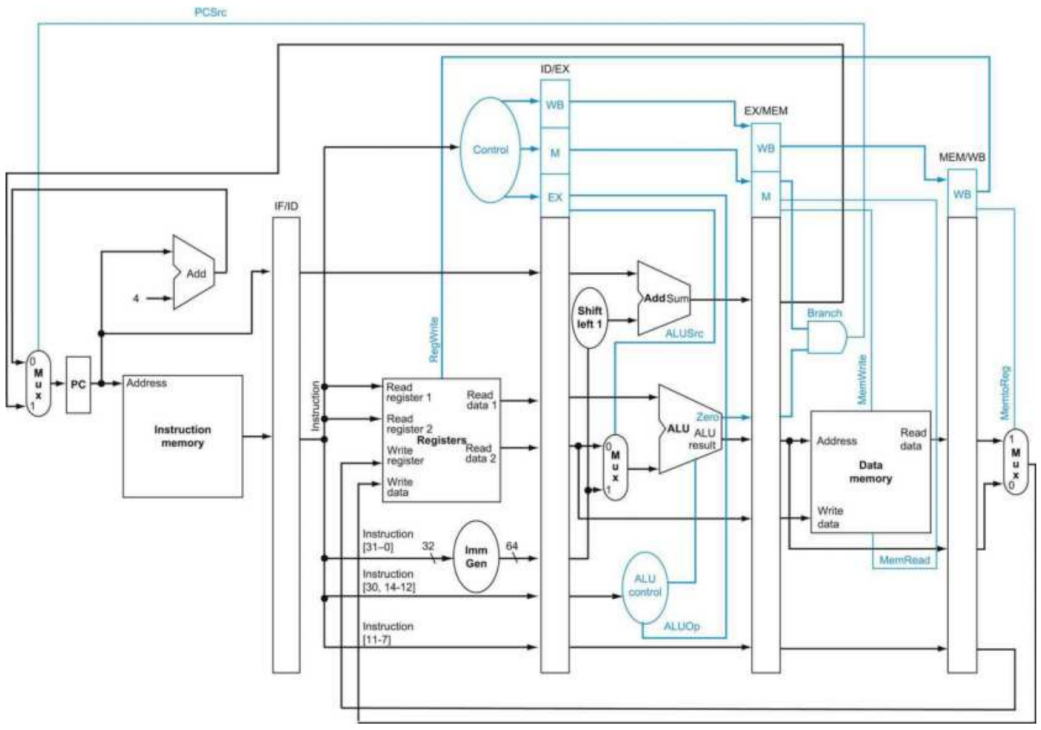

Figure 6: Pipelined datapath and contorl without hazard detection

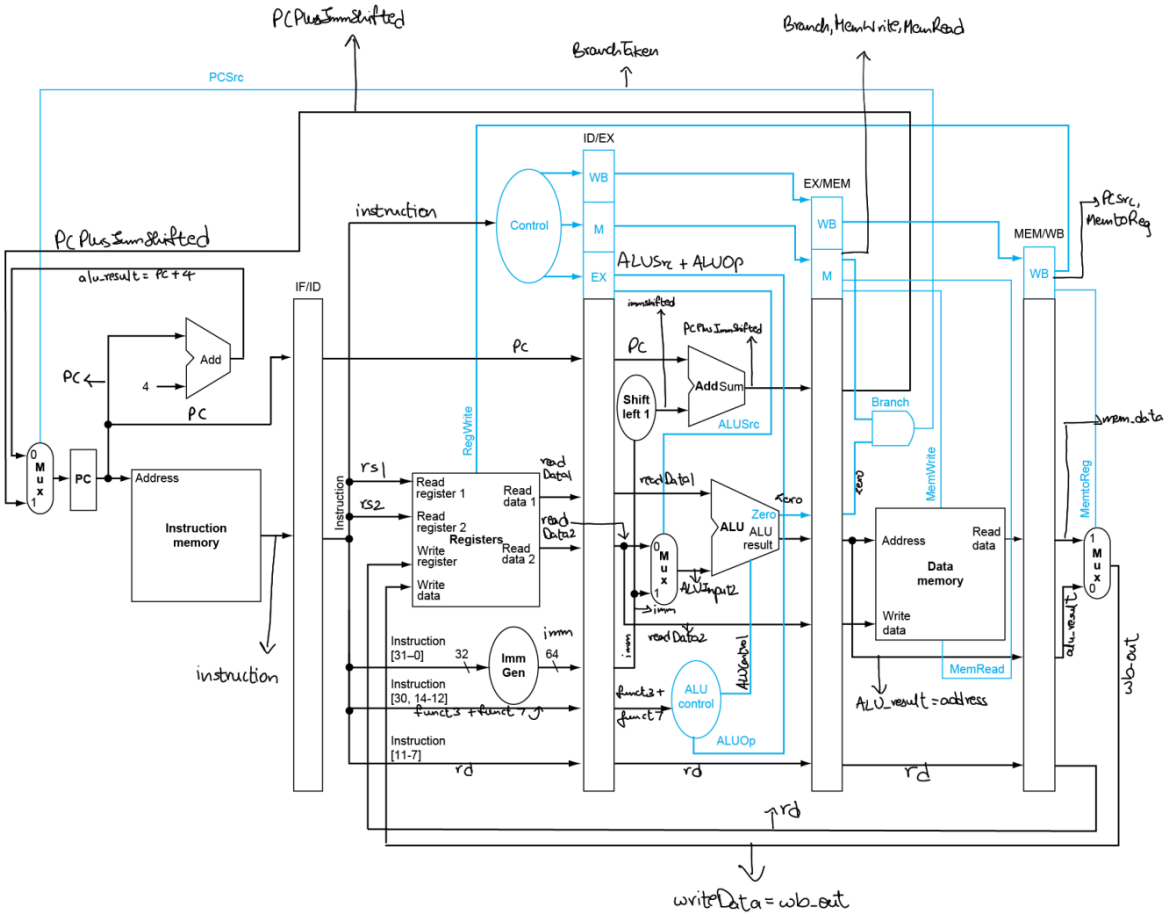

Figure 7: Labelling wires as per verilog code

Note the following:

- The mux that selects new PC has been shifted to stage 1.
- The And gate that does branch taken is shifted to stage 4.
- For writeback , the destination register address has been forwarded till last stage so value gets loaded from memory/alu result to right register.
- refer to the figure for which contro signals to be sent(as a new program is fetched, control signals are changed and hence it is necessary to keep the track of old signals)

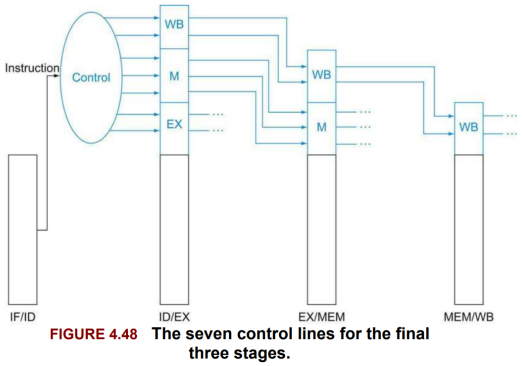

FIGURE 4.48 The seven control lines for the final three stages.

Figure 8: Control signals to be forwarded

Working

Refer to this LINKin order to see terminal outputs with hex files and gtkwave.

Data hazard

We use **forwarding unit** to stop data hazard.

- This data hazard will happen in a case like

```
sub x2, x1, x3 // Register z2 written by sub
and x12, x2, x5 // 1st operand(x2) depends on sub
or x13, x6, x2 // 2nd operand(x2) depends on sub
add x14, x2, x2 // 1st(x2) & 2nd(x2) depend on sub
sd x15, 100(x2) // Base (x2) depends on sub
```

- The final condition to stop data hazards are as follows
  - The destination register shoulsnot be written to x0.
  - he source registers of the instruction in the ID/EX stage match the destination register of a later instruction:
    ID/EX.rs1==EX/MEM.rdID/EX.rs1==EX/MEM.rd OR ID/EX.rs2==EX/MEM.rdID/EX.rs2==EX/MEM.rd

    ID/EX.rs1==MEM/WB.rdID/EX.rs1==MEM/WB.rd OR ID/EX.rs2==MEM/WB.rdID/EX.rs2==MEM/WB.rd
  - RegWrite==1
  - If EX/MEM contains a more recent result than MEM/WB, forward from EX/MEM.
- There are two MUXes , whose logic is given as

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.

Figure 9: MUX control

- The problem of double data hazard is also solved by passing the newer value from ALU result.

```
sub x2, x1, x3 // Register z2 written by sub
and x12, x2, x5 // 1st operand(x2) depends on sub
or x13, x6, x2 // 2nd operand(x2) depends on sub
add x14, x2, x2 // 1st(x2) & 2nd(x2) depend on sub
sd x15, 100(x2) // Base (x2) depends on sub
```

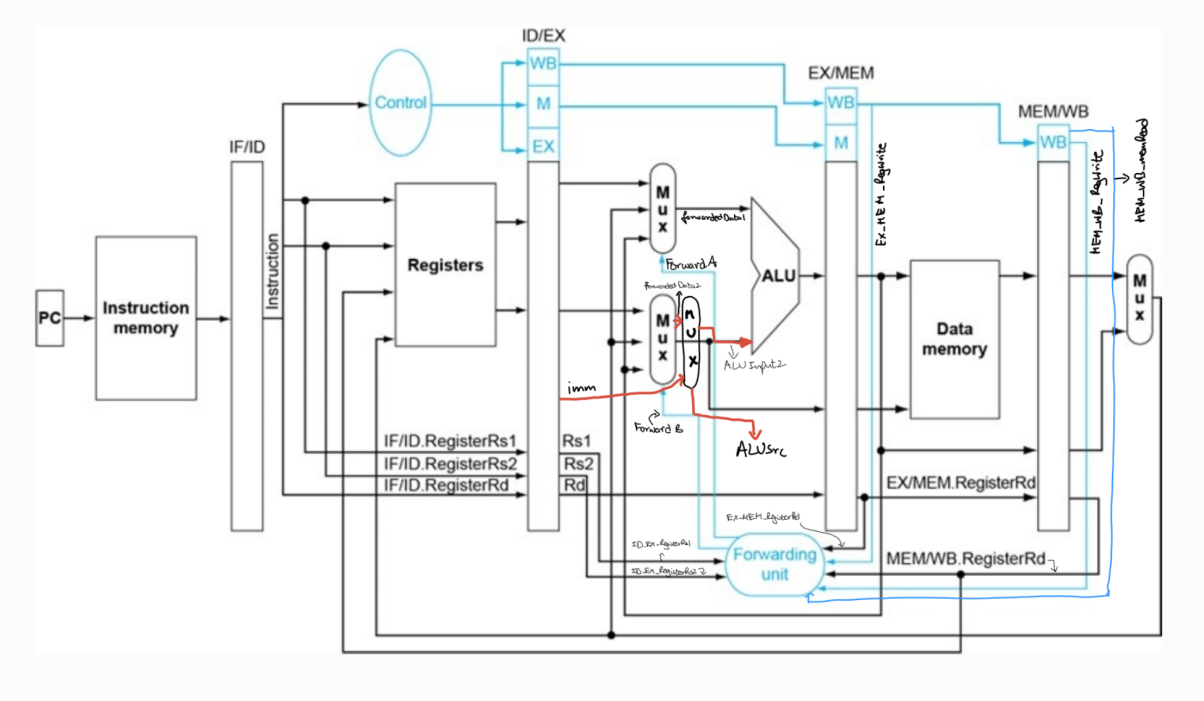

Figure 10: Wire labels

- Working Refer to this LINKin order to see terminal outputs with hex files and gtkwave.

Load Use Data Hazard

- This occurs when data is loaded from memory into register and then used.

```
Ld x2,16(x8)
Add x4,x5,x2
Sub x8,x2,x6
Add x9,x4,x2
```

- When a load instruction is decoded, it inserts a a stall. Inserting a stall means making control signals zero.
- Two wires come out of the hazard detection unit that control the IF ID and PC write.
- with introduction of hazard unit, forwarding unit gets updated, now data from memory should also be forwarded.Refer to the updated figure.

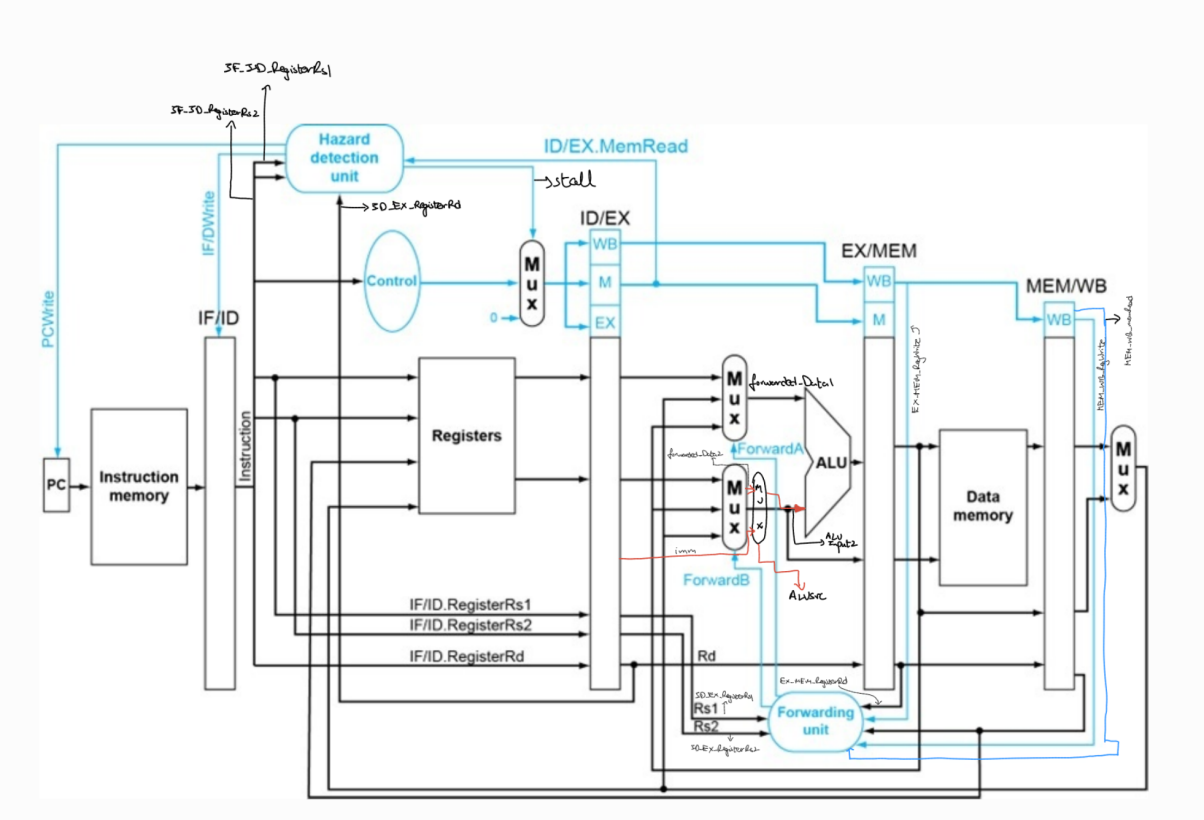

Figure 11: Hazard unit

- Working
  Refer to this LINKin order to see terminal outputs with hex files and gtkwave.

Control hazard

- Also called as branch hazard, it occurs in pipelined processors when the flow of instruction execution is altered by branch (control) instructions.
- The pipeline fetches instructions before knowing whether a branch will be taken or not, leading to potential misfetches and wasted cycles.
- The methods of resolving the hazard include flushing the unnecessary instructions when branch is taken and static/dynamic branch prediction.
- With the first method, we assume that the branch is untaken and continue with the execution of the instructions. When the branch result is available in the execute stage, we flush out IF/ID, ID/EX and EX/MEM pipeline registers(NOP).
- This ensures that unwanted instructions are stopped and discarded, but introduces a delay of a few clock cycles.
- However, if branch is not taken, then this method is fast and improves performance; otherwise, it will just lag for a few cycles only. In general, performance improves by 50-60% from the conventional method.
- Example-
  1. beq x0, x0,10
  2. add x25, x0, x8
  3. add x2, x0, x1

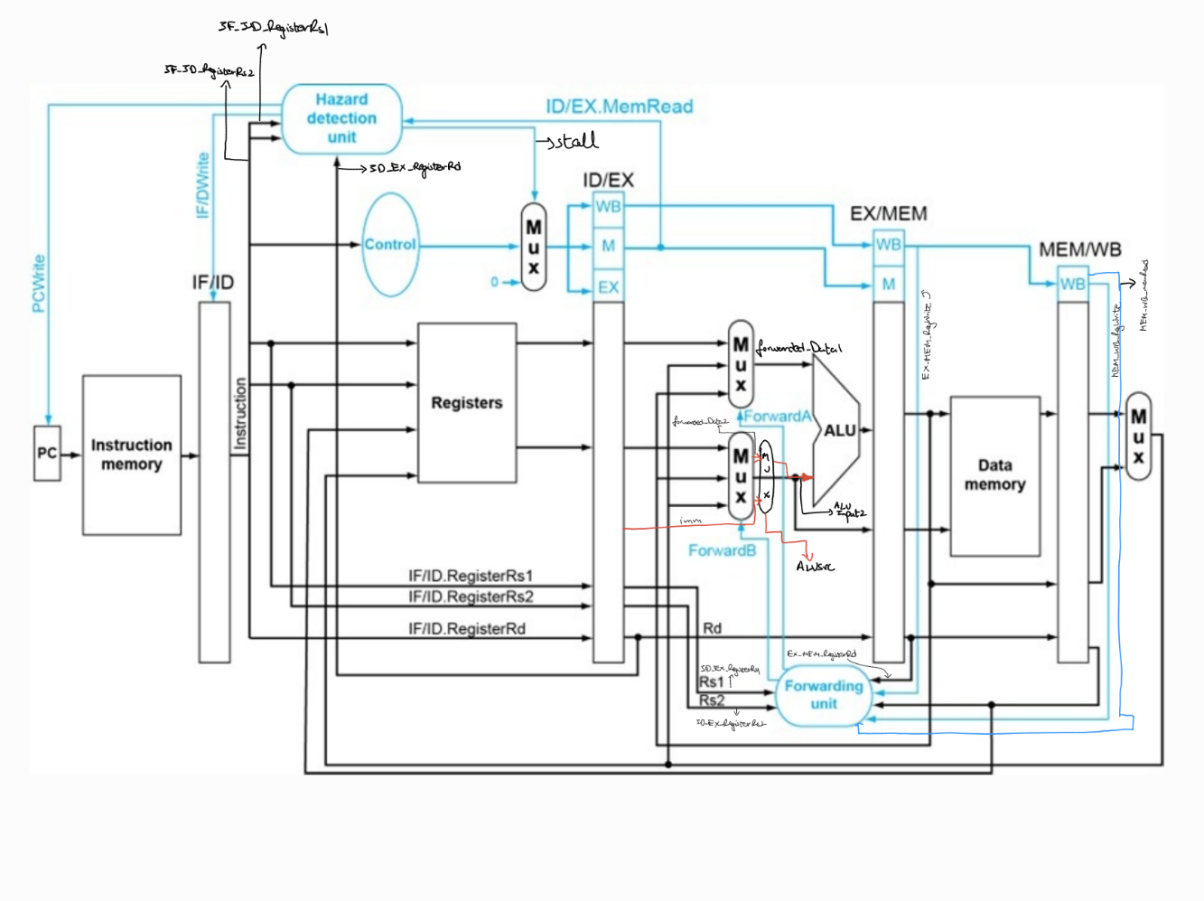

Figure 12: Hazard Unit

4. add x10, x0, x1
5. addi x6, x0 , 4
6. addi x5, x0, 4

- In this example, we observe that the beq result will be available during the 3rd clock cycle. By then, unnecessary instructions are executed(2 add operations), so after branchTaken is triggered, we flush all 3 registers and update PC too. In the next clock cycle we observe proper branching and passing of all 0(NOP) in following stages.

Contribution

We believe in teamwork rather than independent contributions. Spending all-nighters was fun, and generative AI helped us a lot. ,

- Kaamya: Independent blocks in sequential circuit, Control hazards,report and figures.
- Roshan: Sequential circuit integration, Pipelined circuit integration, forwarding unit,hazard detection unit.
- Madhan Sai Krishna : Sequential circuit integration, Pipelined circuit integration, forwarding unit,hazard detection unit.
