IPA PROJECT REPORT

Team no.: 10
GARIMA MITTAL (2023102069)
HARSHITA KUMARI (2023102073)
SAMPATH RAVIKANTI (2023102033)

Abstract

The rapid evolution of computer architectures has led to the increasing adoption of Reduced Instruction Set Computing (RISC) principles, with RISC-V emerging as a prominent open-source instruction set architecture (ISA). This project focuses on the sequential and pipelined implementation of a RISC-V processor using Verilog, a widely used hardware description language (HDL) for digital circuit design.

The processor is designed to implement the instructions **ADD, SUB, AND, OR, LOAD, STORE** and **BRANCH IF EQUAL**. The processor's datapath is constructed using essential components such as the instruction fetch unit, instruction decode unit, register file, arithmetic logic unit (ALU), data memory, and control unit. Moreover, the pipelined implementation of the RISC - V processor contains the necessary hazard control techniques.

This report presents a comprehensive overview of the processor architecture, detailing the design methodology, implementation in Verilog, and performance evaluation based on GTKWave plots.

Contents

Chapter 1

Introduction

A processor, also known as a central processing unit (CPU), is the fundamental computing unit responsible for executing instructions and performing arithmetic and logical operations.

1.1 Components of a Processor

A processor consists of several key components that work together to execute instructions efficiently:

- **Program Counter (PC):** A 64-bit memory unit that stores the address of the location of instruction being executed, in the instruction memory.
- **Instruction Memory:** An array of 32-bit memory units, where each unit stores a single machine code instruction. Each instruction is stored at a unique memory location, indexed sequentially. In our code, we have used an instruction memory of size 1024.
- **Control Unit (CU):** Directs the operation of the processor by interpreting instructions and coordinating data flow. It determines the operation mode of the processor based on opcode and function fields in instructions.
- **Register Memory:** A set of 32 general-purpose registers (x0 to x31) where each register stores 64-bit. The register x0 is hardwired to zero. Registers are used for fast access to operands during instruction execution.
- **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations on data. The ALU operates on data from the register file and produces results stored back into registers or memory.
- **Immediate Generator:** A unit that extracts immediate values from instruction fields, extending them to the appropriate bit-width (sign-extension or zero-extension) based on instruction type (I-type, S-type, B-type, etc.).
- **Main Data Memory:** A huge array of 64 bit memory units that holds data required by the processor. This memory is typically implemented as RAM and may include a memory hierarchy (cache, main memory, etc.).

- **Hazard Control Unit:** Primarily used for detecting and resolving load - use data hazard in the pipelined implementation, by sending the necessary signal for stalling.
- **Forwarding Unit:** Primarily used for forwarding data after the ALU output in the case of data hazard with the help of appropriate signals.

1.2 Stages of Instruction Execution

Every instruction executed by a processor goes through multiple stages to ensure correct operation and efficient processing. The main stages are:

- **Instruction Fetch (IF):** The processor retrieves the instruction from memory based on the program counter (PC). The instruction is then stored in the instruction register for decoding.
- **Instruction Decode (ID):** The fetched instruction is decoded to determine the operation to be performed and the required operands. Register values are read from the register file if necessary.
- **Execute (EX):** The ALU performs the required computation, such as arithmetic or logical operations, based on the decoded instruction.
- **Memory Access (MEM):** If the instruction involves memory operations (e.g., load or store), the processor accesses the data memory to either read from or write to memory.
- **Write Back (WB):** The final result is written back to the destination register, updating the register file with the new computed value. Write-back is always synchronized with the positive edge of the clock.

This project focuses on both the sequential (single-cycle) and pipelined implementation of a RISC-V processor using Verilog, where the sequential design completes each instruction in a single clock cycle, while the pipelined design improves performance by overlapping instruction execution stages.

1.3 Instruction Set Overview

- **R-type instruction** In this type of instruction, both the ALU inputs are read from the register memory after decoding the read addresses in the decode stage. The necessary operation is performed by the ALU, and the ALU output is stored in the register memory using the write address that was obtained in the decode stage. AND, OR, ADD, and SUB are R-type instructions.
- **I-type instruction** In this type of instruction, one ALU input is read from the register memory, while the second operand is an immediate value that is extracted and sign-extended from the instruction. After performing the required ALU operation, the result is stored in the destination register. I-type instructions include LOAD, where the immediate value is used for memory addressing.

- **S-type instruction** In this type of instruction, one register provides the base address, and another register provides the data to be stored in memory. The immediate value from the instruction is sign-extended and added to the base address to generate the effective memory address. The data is then stored at this computed address in the data memory. Examples of S-type instructions include STORE which write data from a register to memory.

- **B-type instruction** In this type of instruction, two registers are compared, and based on the result, the program counter (PC) is updated to a new address computed using an immediate value extracted from the instruction. The immediate value is sign-extended and added to the PC to determine the branch target. If the branch condition is met, execution jumps to this address; otherwise, execution continues sequentially. B-type instructions include BEQ .

Chapter 2

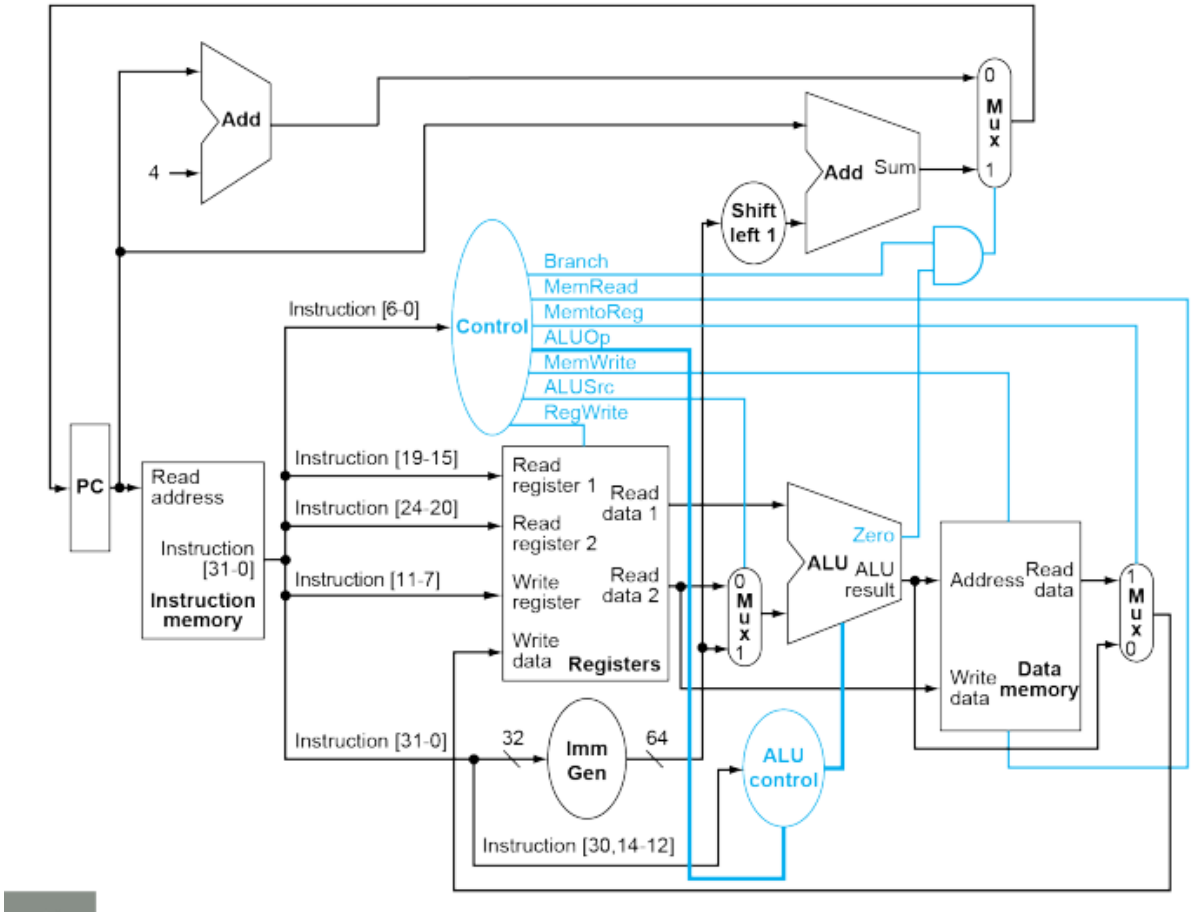
Sequential Implementation

2.1 Introduction to Sequential Implementation

A sequential (single-cycle) RISC-V processor executes each instruction in a single clock cycle, completing all five stages—fetch, decode, execute, memory access, and write-back—within one cycle. Its performance is limited as the clock cycle must accommodate the slowest instruction.

The sequential implementation of the RISC V processor, contains a module for each stage (except write back) and a wrapper datapath module in which all the other modules are instantiated. The step by step working of each and every module is provided in the following sections.

2.2 Understanding the Diagram: Stage by Stage Breakdown



Sequential Processor.

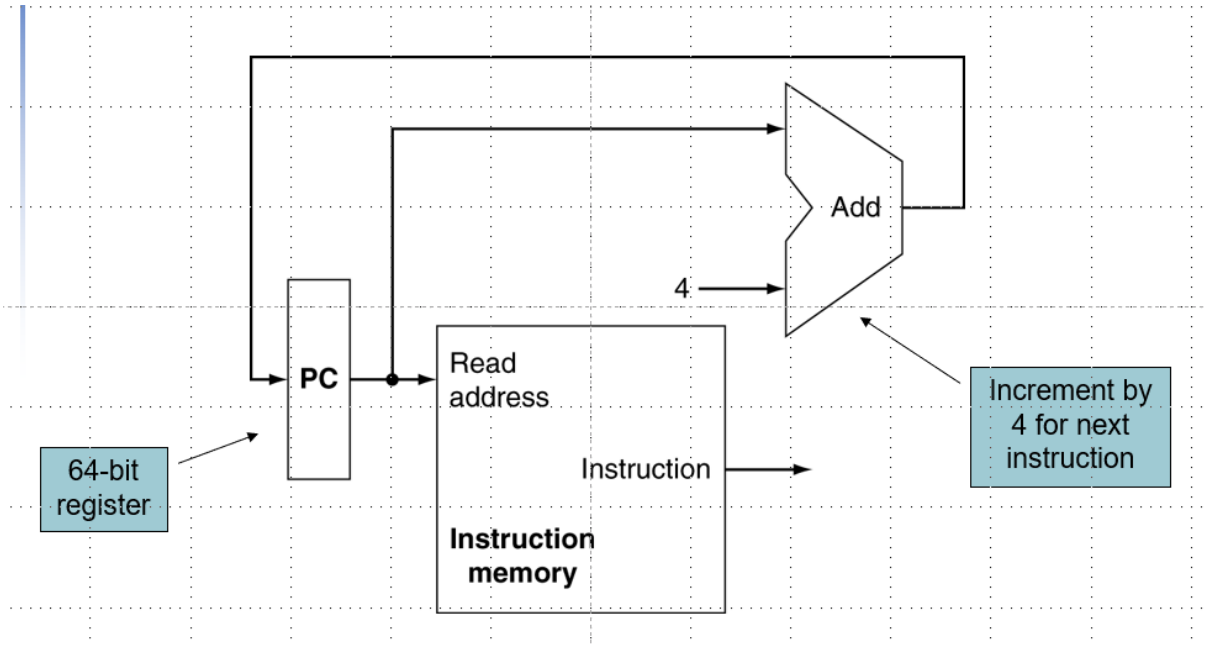
The implementation of a sequential processor progresses in 5 stages.

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

2.2.1 Instruction Fetch (IF Stage) Module

- The **Program Counter (PC)** is sent as an input and the **instruction** and the flag **invAddr** is sent as the output.
- **PC** The holds the address of the instruction to be fetched. The instruction is fetched from **Instruction Memory**.

- The **invAddr** is set high if the **PC** points to any invalid address (PC ¿= 0 and PC ¡=) The **PC is incremented by 4** for the next instruction to be fetched (if not a branch) (since each instruction is 4 bytes in RISC-V).



IF Stage

2.2.2 Instruction Decode (ID) Stage

- The fetched instruction is decoded, and the **register file** reads values from the source registers.
- The 32-bit instruction is decoded as following(for different type of instructions):
  - R-type instruction decoder (and, or, add, sub):

func7	**rs2**	**rs1**	**func3**	**rd**	**opcode**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

  - I-type instruction decoder(load):

immediate	**rs1**	**func3**	**rd**	**opcode**
12 bits	5 bits	3 bits	5 bits	7 bits

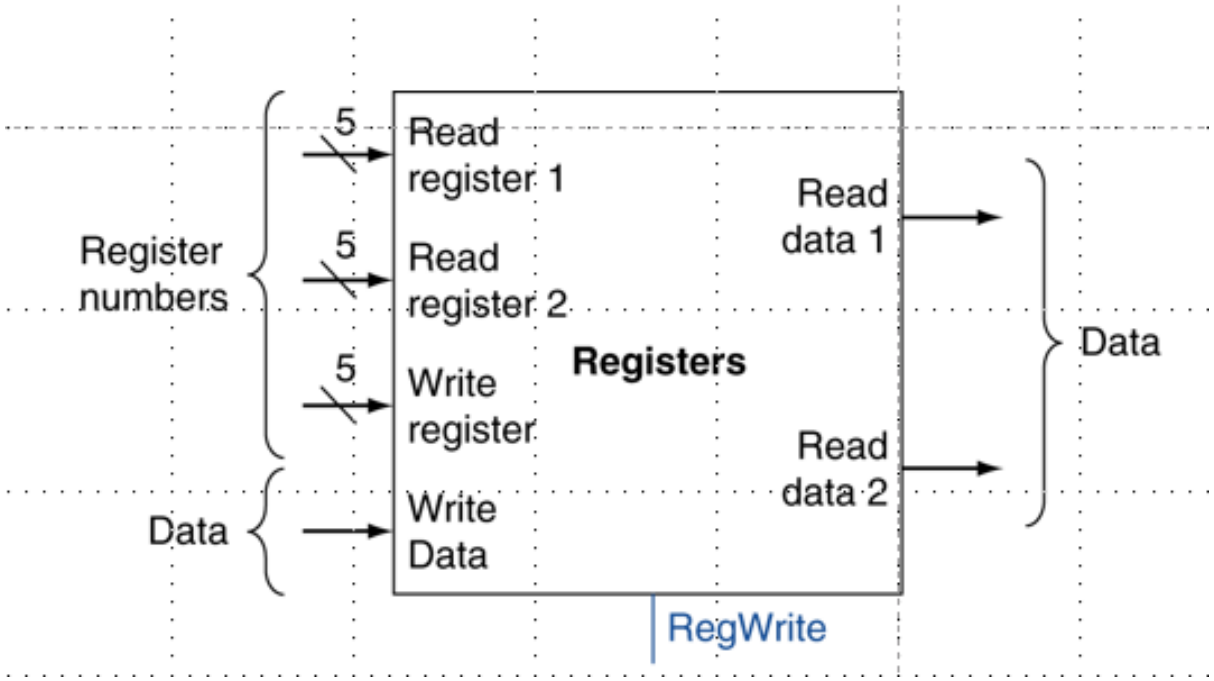
  - S-type instruction decoder(store):

imm[11:5]	**rs2**	**rs1**	**func3**	**imm[4:0]**	**opcode**
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

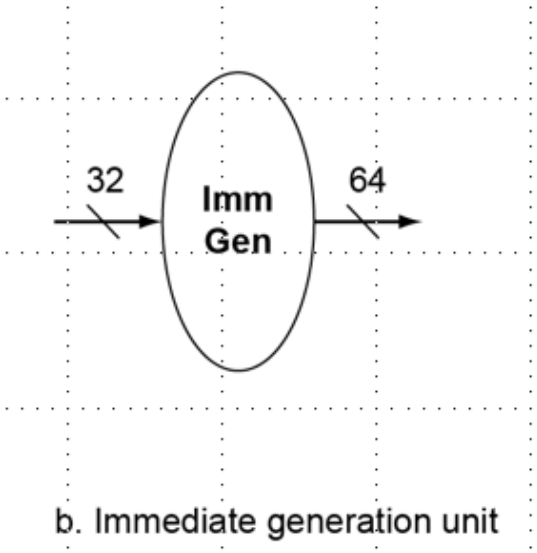
  - SB-type instruction decoder (branch):

imm[12]	**imm[10:5]**	**rs2**	**rs1**	**imm[4:1], imm[11]**		**opcode**
				4 bits	1 bit	
1 bit	6 bits	5 bits	5 bits	5 bits		7 bits

- Immediate values are generated using the **Immediate Generator (Imm Gen)** for load/store/branch instructions. Immediate generator gets the 32-bit instruction and according to the instruction type it calculates the 12-bit immediate and the sign-extends to 64-bit.
- The **control unit** generates control signals required for execution which are ALUOp, Memread, Memwrite, ALUSrc, Branch, MemtoReg, Regwrite.
- ALUOp is a 2-bit control signal given to the ALU-Control which tells it what type of instruction is to be implemented.
- Memread, Memwrite given to Data-memory based on load/store.
- ALUSrc decides whether the input to the read_data_2 will be from register(R-type instruction) or from immediate(I-type/S-type/B-type)
- Branch signal is given to the and gate which decides whether the instruction is branch or not.
- MemtoReg is for load instruction and Regwrite is for load and R-type where data needs to be stored back into the write register.
- Following images show how instruction is decoded, immediage is sign-extended and how Control produces the signals.

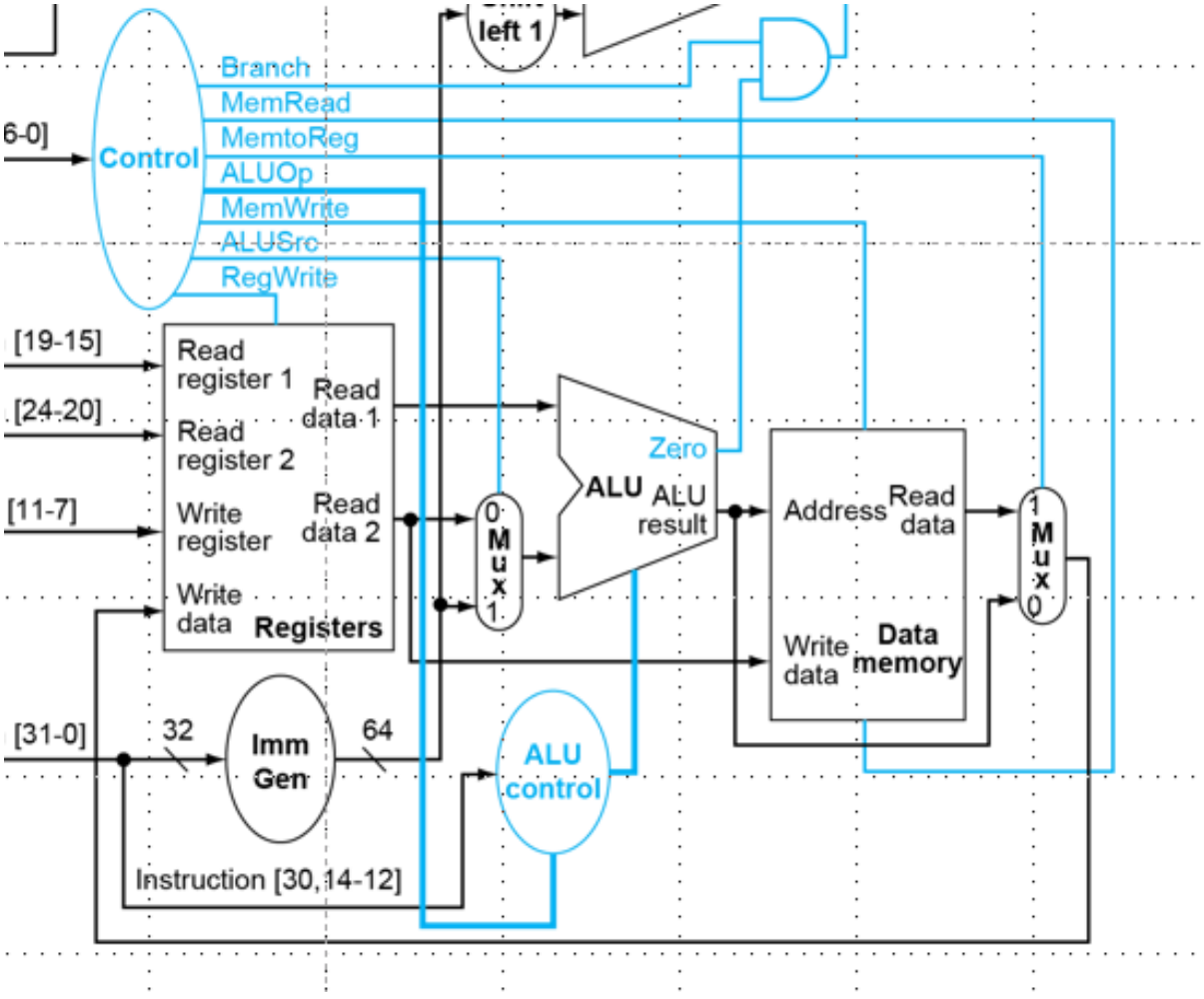


Instruction Decode.



b. Immediate generation unit

Immediate Generator.

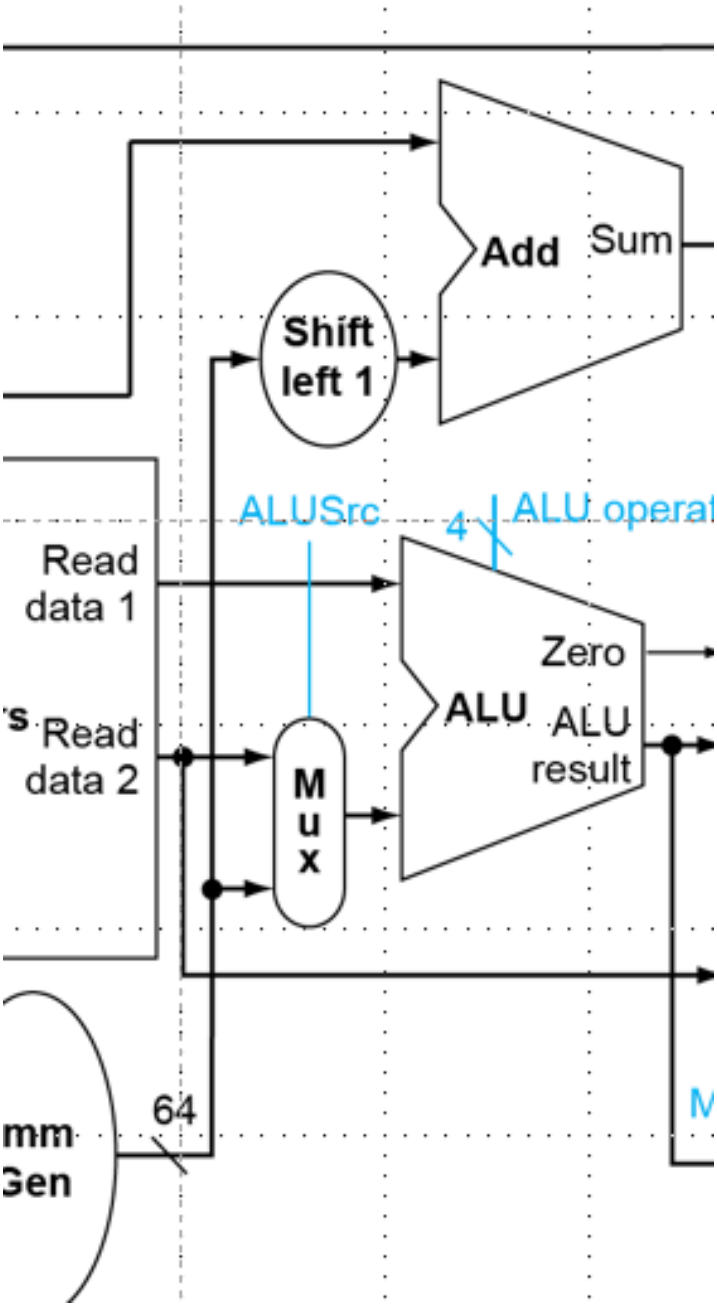


Control Unit

2.2.3 Execution (EX) Stage

- This stage executes ALU operations based on control signals.
- The **ALU (Arithmetic Logic Unit)** performs arithmetic or logical operations and reads data which was decoded in ID stage.

- **ALU control** determines the operation based on instruction type and generates a 4-bit ALU operation based on which ALU compute the result.
- A **MUX (Multiplexer)** decides whether the second operand comes from a register or the immediate value (ALUSrc).
- In this processor also computes the branch address in which the immediate is shifted left and then added to the PC. This address goes to another MUX which decides whether the next PC will be branch or not.

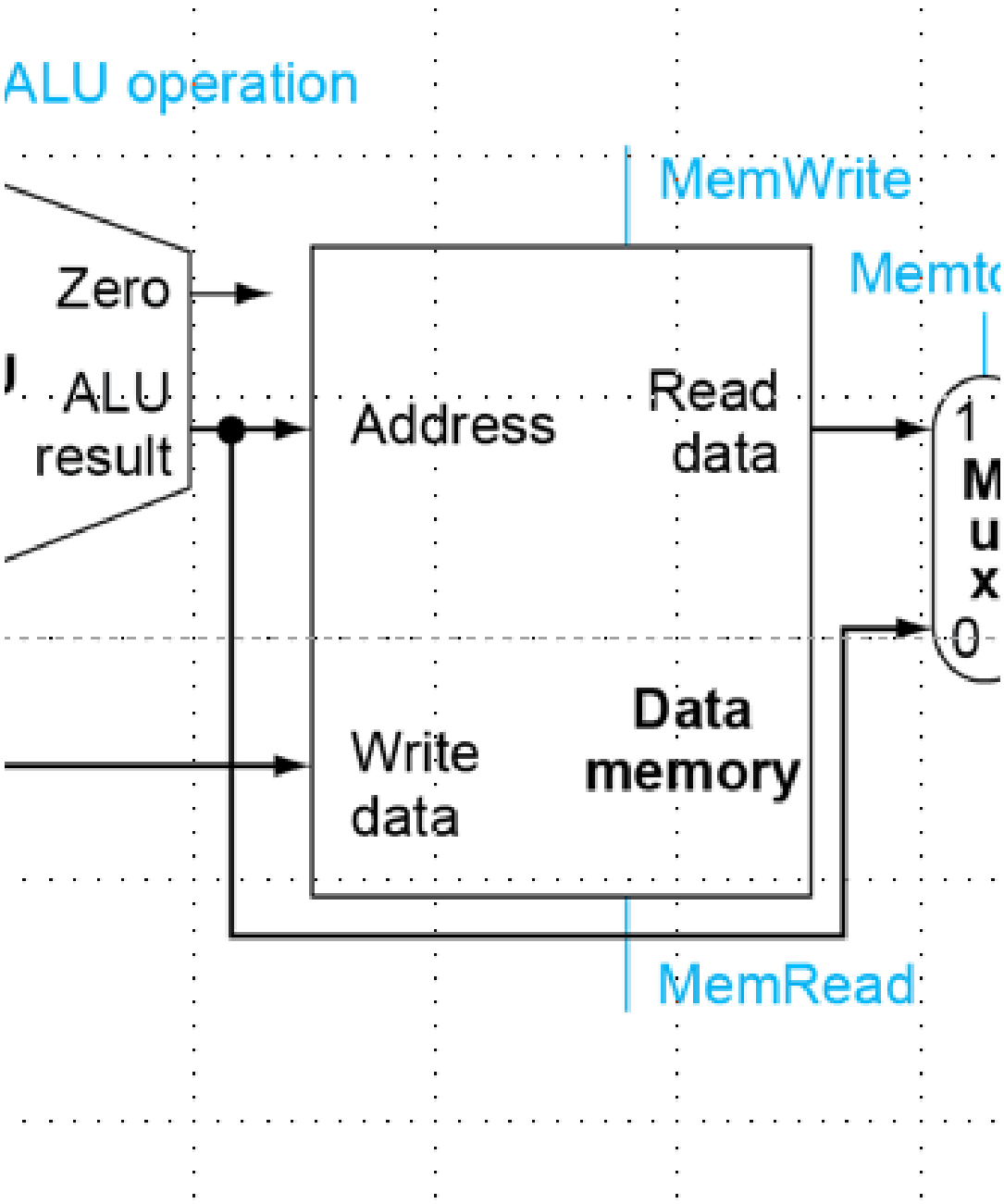


Execute Stage and Branch PC calculation.

2.2.4 Memory Access (MEM) Stage

- If the instruction is a **load/store**, the computed memory address is used to access **Data Memory**.

- For a **load instruction**, data is read from memory at the address computed by the ALU.
- For a **store instruction**, data is written to memory at the address computed by the ALU.
- This stage is controlled by Memwrite, Memread. Data memory is accessed only if one of these control signals is 1. If memwrite is 1 then the data is stored in the memory and if memread is 1 then data is read from the memory and then given to read_data, which gives it further to write back stage.



Memory access stage.

2.2.5 Write-back (WB) Stage

- The final stage writes the result back to the **register file**.
- A **MUX (Multiplexer)** decides whether the data comes from the **ALU result** (R-type instruction) or **memory load result**.
- The result is written into the destination register which was earlier decoded in the ID stage.

Chapter 3

Pipelined Implementation

3.1 Introduction to Pipelined Implementation

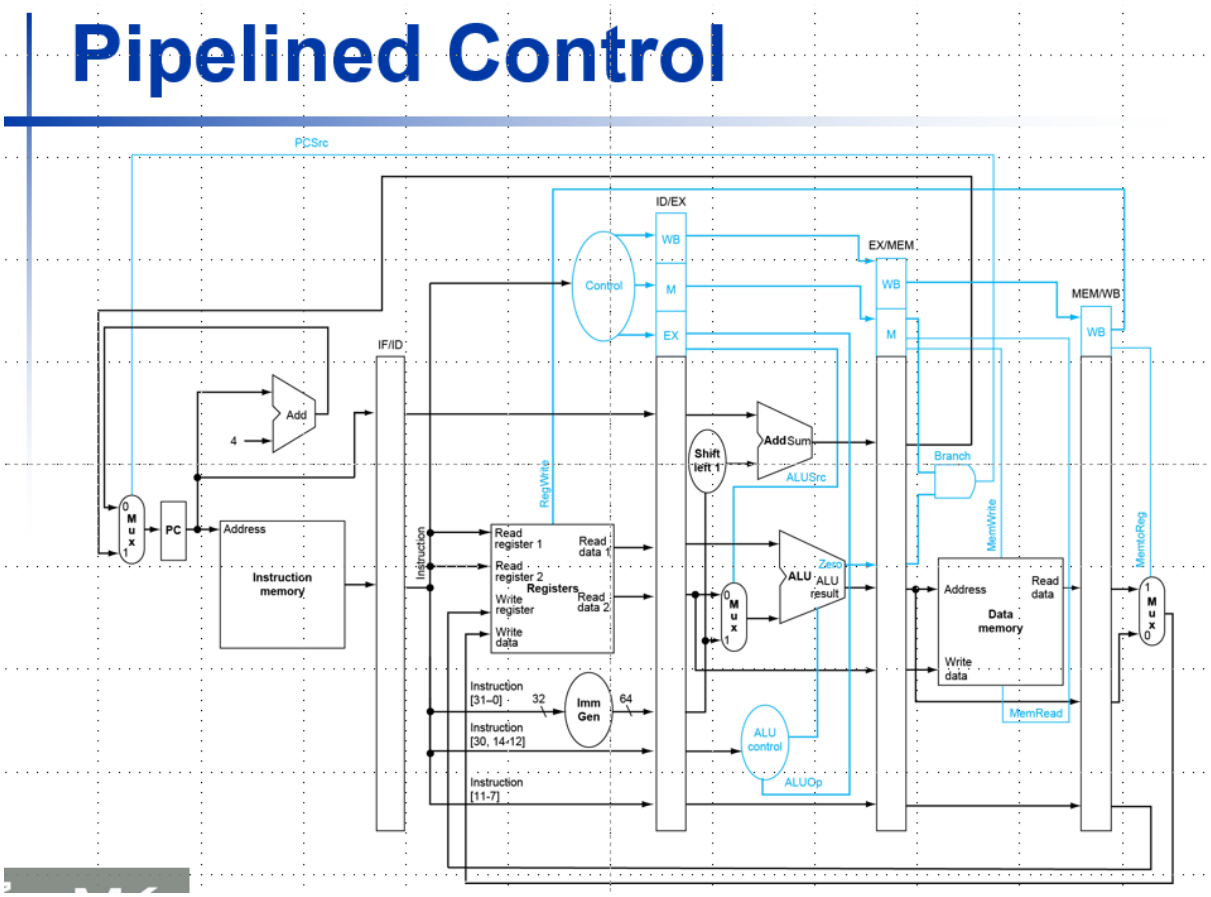
The pipelined implementation of the **RISC-V processor** follows a systematic approach where the execution of instructions is broken down into multiple stages, allowing multiple instructions to be executed simultaneously at different stages. This increases instruction throughput and enhances performance compared to a sequential (single-cycle) processor.

A key feature of pipelining is the use of **registers between each stage**, ensuring that each stage operates independently and passes required data to the next stage. These **pipeline registers** are clocked, meaning that at every clock cycle, the intermediate values from one stage are stored and forwarded to the next stage.

The primary goal of pipelining is to **retain the same logical structure** as a sequential processor but introduce registers at each stage to separate different instruction executions. This modification allows parallel execution while ensuring correct instruction flow.

3.2 Understanding the Diagram: Stage by Stage Breakdown

The following diagram represents a basic **5-stage pipelined implementation** of the RISC-V processor without hazard correction.



Pipelined processor

Each instruction progresses through the following five pipeline stages (all of these are explained above):

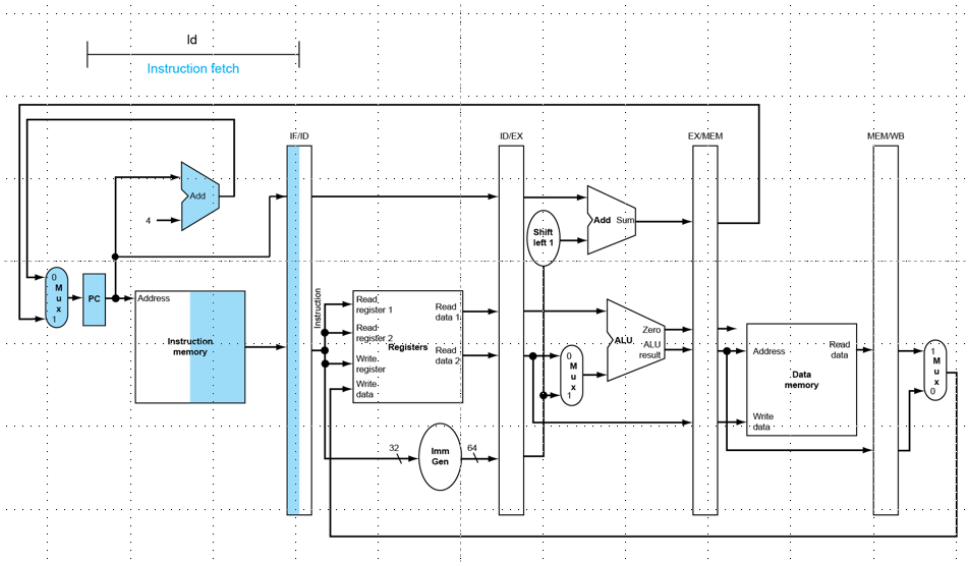
1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID)**
3. **Execution (EX)**
4. **Memory Access (MEM)**
5. **Write-back (WB)**

Between each stage, **registers** store intermediate results and pass them to the next stage, maintaining synchronization with the clock.

3.2.1 IF/ID Register

- This register stores the current PC and instruction fetched from that PC on 1st clock cycle.
- In the next clock cycle, it outputs the PC value and instruction to the next stage.
- In the next cycle it as the next instruction is fetched from updated PC, new values are inputed to it which will be latched to the output on the next clock cycle.

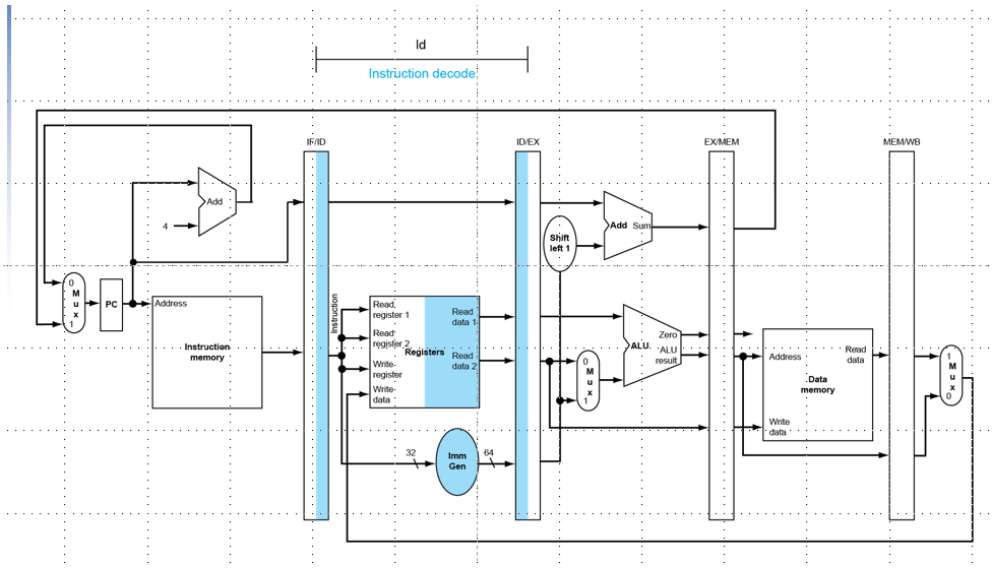
- Following diagram shows the IF stage and IF/ID register.



IF stage with IF/ID Register.

3.2.2 ID/EX Register

- The fetched instruction which was outputted from IF/ID register is decoded and ID stage is carried out.
- Now when the instructions are decoded the values like read_data-1 and read_data-2, immediate, PC (from IF/ID for branch calculation), write_register and controls are given as input to ID/EX register.
- In the next clock cycle the inputed vales get latched to the output which are used by next stage and now ID stage decodes the next instruction now outputted from the IF/ID register.

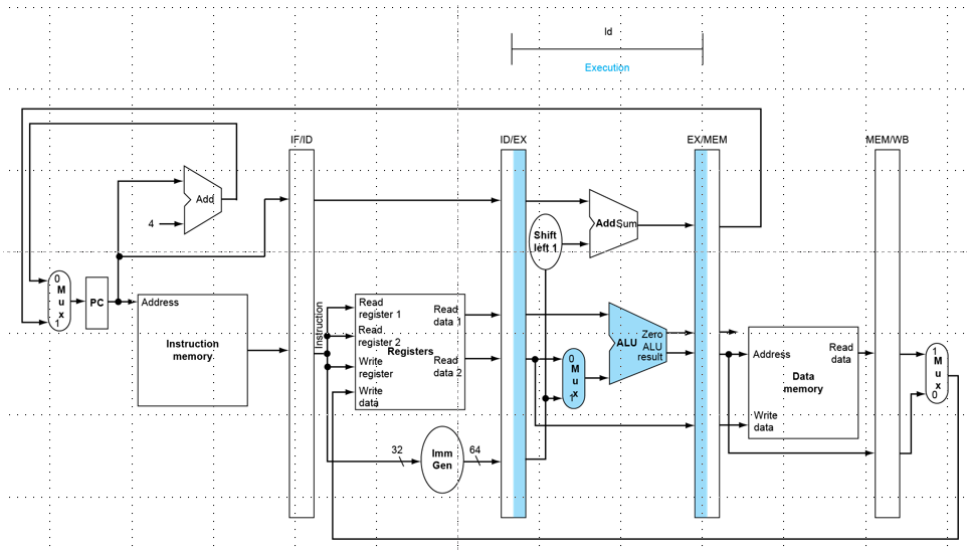


ID stage and ID/EX stage.

3.2.3 Execution (EX) Stage

- In the **Execute (EX) stage**, the values latched from the ID/EX register are used for computations. The ALU performs operations based on the decoded control signals, utilizing operands from registers or immediate values. The computed result, along with control signals and necessary data, is stored in the EX/MEM register for the next stage.

- In the next clock cycle, these values are latched as outputs from EX/MEM and passed to the MEM stage, while the ID/EX register latches the next instruction's decoded values from the ID stage. Thus, execution continues in parallel for multiple instructions.

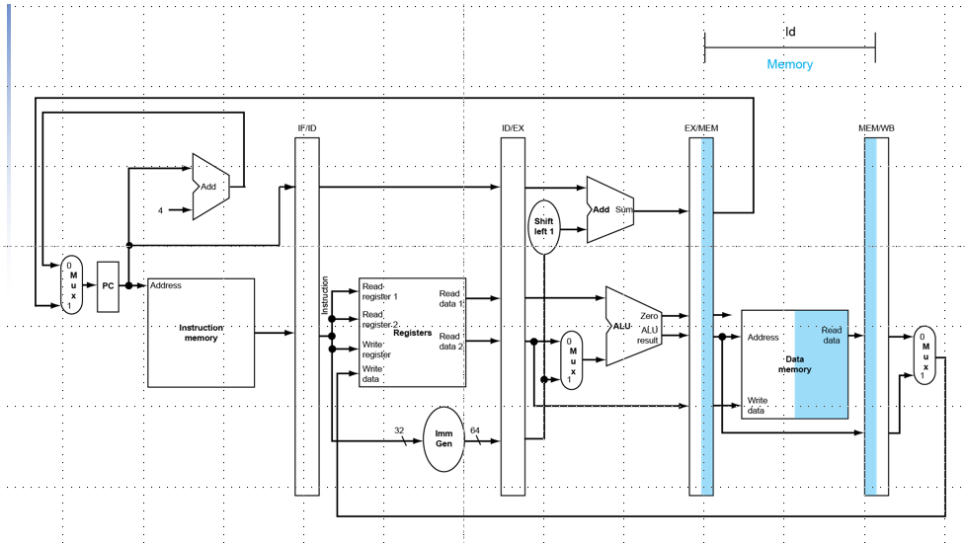


Execure stage and EX/MEM stage

3.2.4 Memory Access (MEM) Stage

- In the **Memory (MEM) stage**, the EX/MEM register outputs are used to access memory if required. Load instructions fetch data from memory, while store instructions write values to the computed memory address. The control signals determine whether memory access is performed, and the relevant data is stored in the MEM/WB register for the final stage.

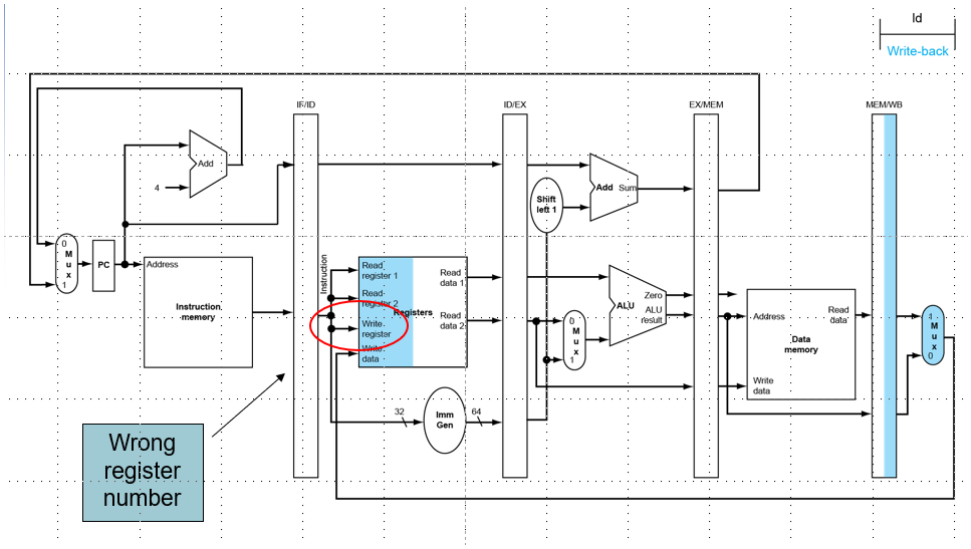
- In the next clock cycle, these values are latched as outputs from MEM/WB and passed to the WB stage, while EX/MEM stores the next instruction's computed results. This ensures that memory operations for one instruction occur while the next instruction is already executing.



Memory stage and MEM/WB register.

3.2.5 Write-back (WB) Stage

- In the **Write Back (WB) stage**, values latched from the MEM/WB register are written back to the destination registers. This includes either the ALU result or memory-loaded data, depending on the instruction type. The register file updates occur, completing the instruction execution cycle while the pipeline continues to process subsequent instructions in parallel.
- In the whole pipelining we also keep storing the write_register forward in every clock cycle because when the data is ready for write back then in that cycle the register in the decode will be of instruction received by it in that cycle.
- In the next clock cycle, a new instruction reaches the WB stage, while the MEM/WB register latches the memory stage output for another instruction. This ensures that while one instruction completes, multiple other instructions are still in the pipeline, keeping the CPU busy and improving instruction throughput.



Writing back into register.

No further pipeline registers are needed after this stage.

3.3 Key Modifications from Sequential to Pipelined Implementation

1. **Register Addition**: Each stage has an additional **pipeline register** at the end to hold intermediate values.
2. **Datapath Modification**:
   - The **output of each stage** is linked to **pipeline registers**.
   - The **output of registers** serves as input for the next stage.
3. **Parallel Execution**: Since instructions are at different stages simultaneously, the processor achieves **higher instruction throughput**.

3.4 HAZARDS

Pipelining increases instruction throughput by allowing multiple instructions to execute simultaneously at different stages. However, it introduces challenges called **hazards**, which occur when an instruction in the pipeline cannot proceed due to dependencies or conflicts.

There are three types of hazards in pipelining:

1. **Structural Hazards**
2. **Data Hazards**
3. **Control Hazards**

Each hazard disrupts the pipeline's smooth operation, but various techniques can be employed to mitigate their effects.

3.5 Structural Hazards

A **structural hazard** occurs when hardware resources required by multiple instructions overlap, leading to conflicts. This happens when the processor does not have enough functional units to execute multiple instructions simultaneously.

3.5.1 Example of Structural Hazard

Consider a scenario where both instruction fetch (IF) and memory access (MEM) stages need access to memory, but the processor has only a single memory unit. This leads to resource contention.

3.5.2 Solution to Structural Hazards

Structural hazards can be mitigated by:

- **Hardware Duplication:** Using separate instruction and data memory (**Harvard Architecture**) eliminates conflicts between fetching and memory operations.
- **Pipeline Stalling:** Temporarily pausing instruction execution when resource contention occurs.

3.6 Data Hazards

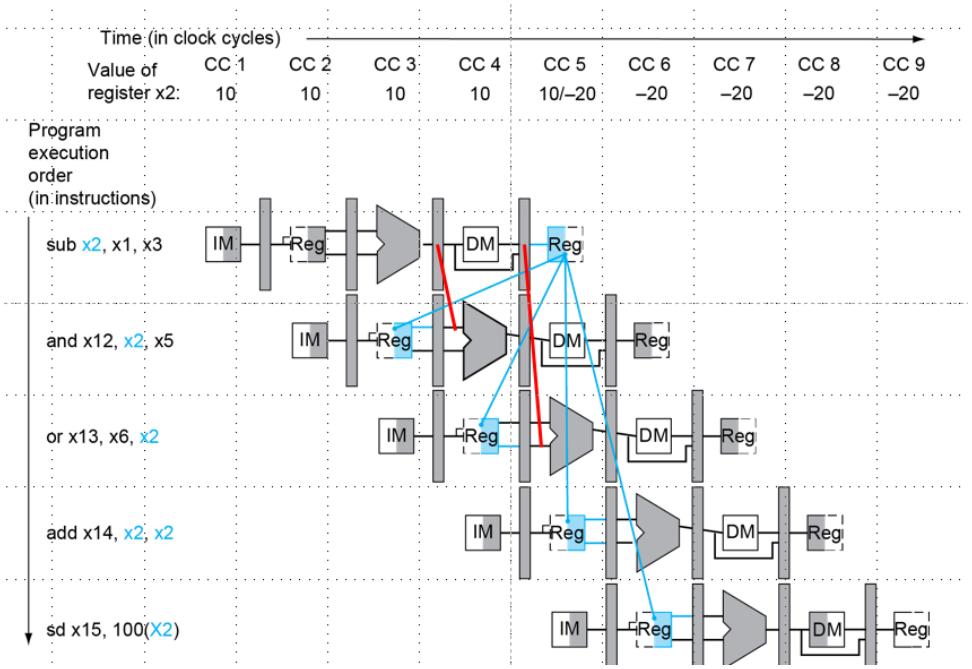
A **data hazard** occurs when an instruction depends on the result of a previous instruction that has not yet completed its execution.

3.6.1 Data Hazards

- **Read After Write (RAW) Hazard:** Occurs when an instruction tries to read a register before a previous instruction has written to it. The following assembly code computes values for registers $x5$, $x6$, and $x7$ with data dependencies between instructions:

```
add x5, x1, x2      % computes x5 here
add x6, x5, x5      % x6 depends on x5
sub x7, x6, x5      % x7 depends on x6
```

In the above given set of instructions RAW hazard can happen beacuse in pipelined implementation, simulaneous execution of different instructions is happening. In the following picture another example of RAW hazard is explained.



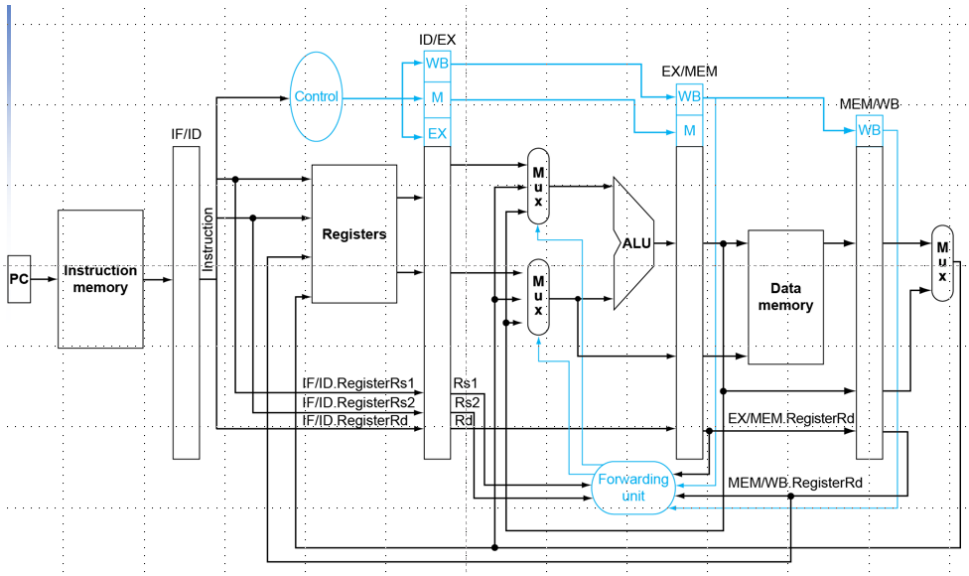
Data Hazard

As we can see above, in normal stage value of x2 will be written back to the register in CC 5, but in pipelined structure the next instruction requires the value of x2 at the start of CC4 and if it reads directly from the register, it will read the value wrong. So to overcome this hazard we use the method of stalling which is explained later.

3.6.2 Solution to Data Hazards

Data hazards can be handled using the following techniques:

- **Forwarding (Data Forwarding or Bypassing):**
  - ALU results are forwarded directly to dependent instructions instead of waiting for them to be written back.
  - This reduces delays by allowing the pipeline to use results as soon as they are computed.
  - This can be seen in above example where red lines show where there is the need to forward the output of previous instruction directly because the value required has been computed at the end of Execute stage.
  - Following diagram tells how forwarding works:



Solution to Data hazard- Forwarding Unit.

This picture shows which input to chose from either from the execute output or from the output of previous stage.

- **Instruction Scheduling (Reordering):**
  - The compiler reorders instructions to avoid conflicts.
  - Independent instructions are executed first to allow enough time for the dependent instructions to execute.
  - The reordering is required in the case where stalling is required i.e. the output hasn't been computed yet.

- **Pipeline Stalling (Interlocking):**
  - If forwarding is not possible, the pipeline is stalled until the required data is available.
  - This introduces a **NOP (No Operation)** instruction to delay execution. This is used in load-use hazard which is explained next.

3.7 Load-Use Hazards

A **load-use hazard** is a type of *data hazard* that occurs in a pipelined processor. It happens when an instruction that immediately follows a load instruction attempts to use the data being loaded.

Example:

```
lw   x5, 0(x6)   % Load data from memory into x5
add  x7, x5, x8  % Use the loaded value from x5 immediately
```

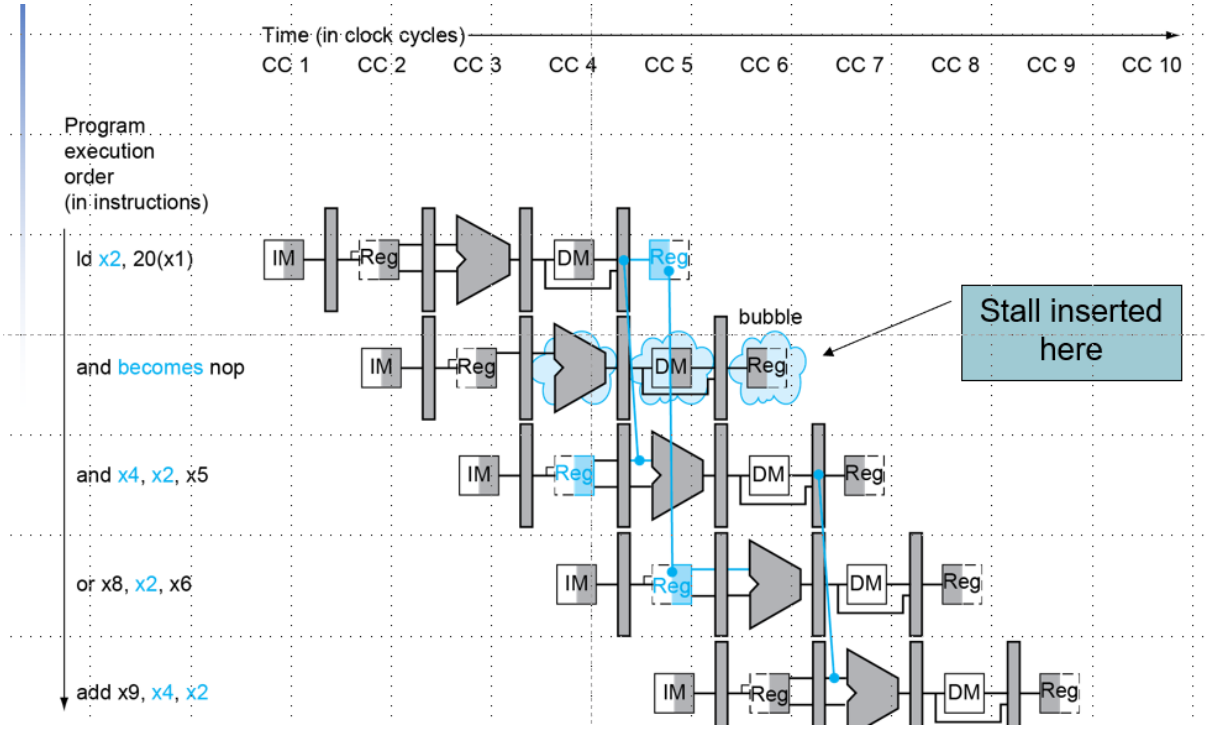
In this example, the `add` instruction requires the value from register `x5`, which is being loaded by the previous `lw` instruction. Since the loaded data is not available until the memory stage is completed, the subsequent instruction may not receive the correct operand on time. This situation forces the pipeline to stop and then employs techniques such as data forwarding to resolve the hazard.

Implications:

- **Stalling:** The pipeline can be stalled, inserting a bubble (NOP) until the data is available.
- **Forwarding:** After the data is computed, hardware forwarding paths can be used to provide the needed data directly from a later stage.

This hazards cannot be corrected just using forwarding as in this case the data required has not yet been computed in the previous cycle itself. 1-cycle stall allows MEM to read the data for ld and subsequently forward to the EX stage.

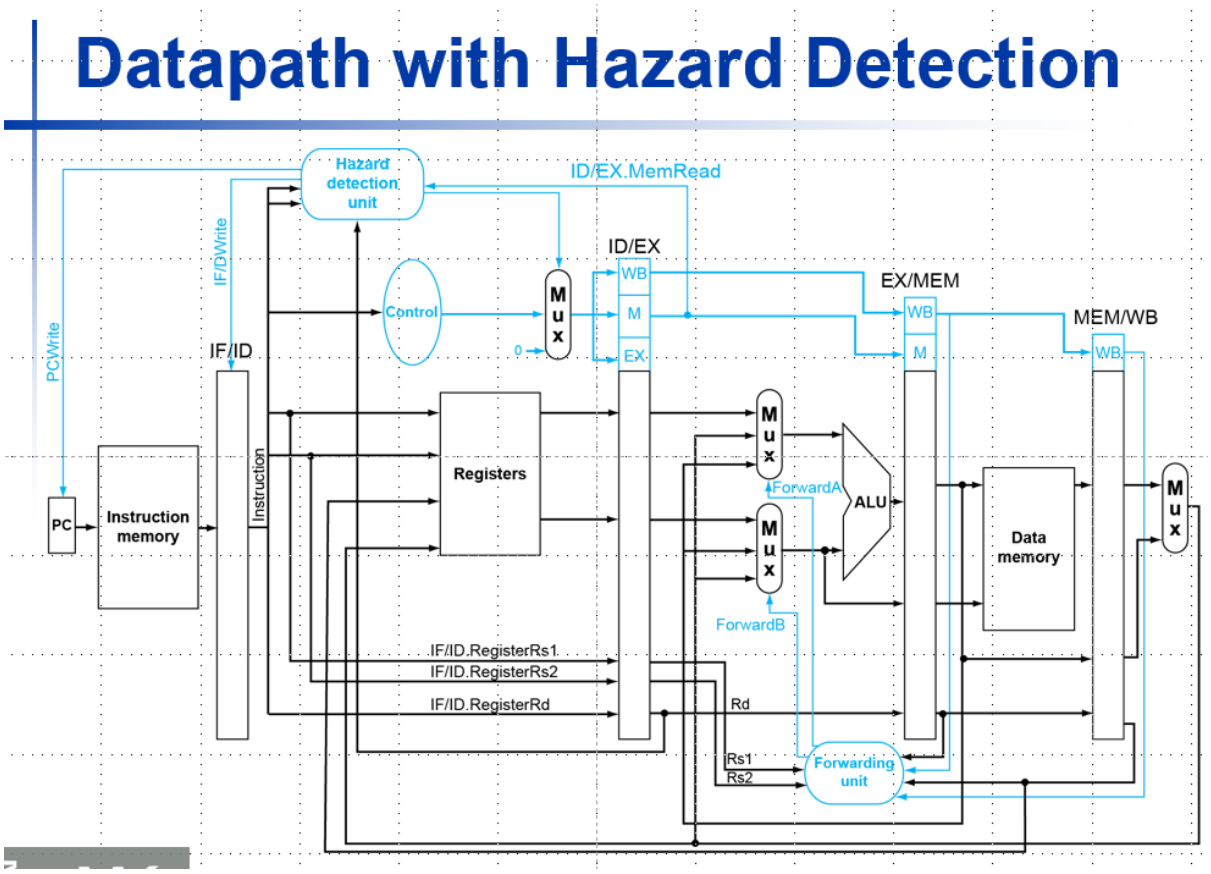
The following image can help better explain things.



Load-use hazard, can be solved by stalls.

If the previous instruction is a load and the next instruction requires the data stored in the register of the previous instruction, this hazard occurs. In load operations, data is

read from memory after the memory stage, so this cannot just be solved by forwarding. The data required at that stage is actually computed in the future, so we stall for one stage and then forward.

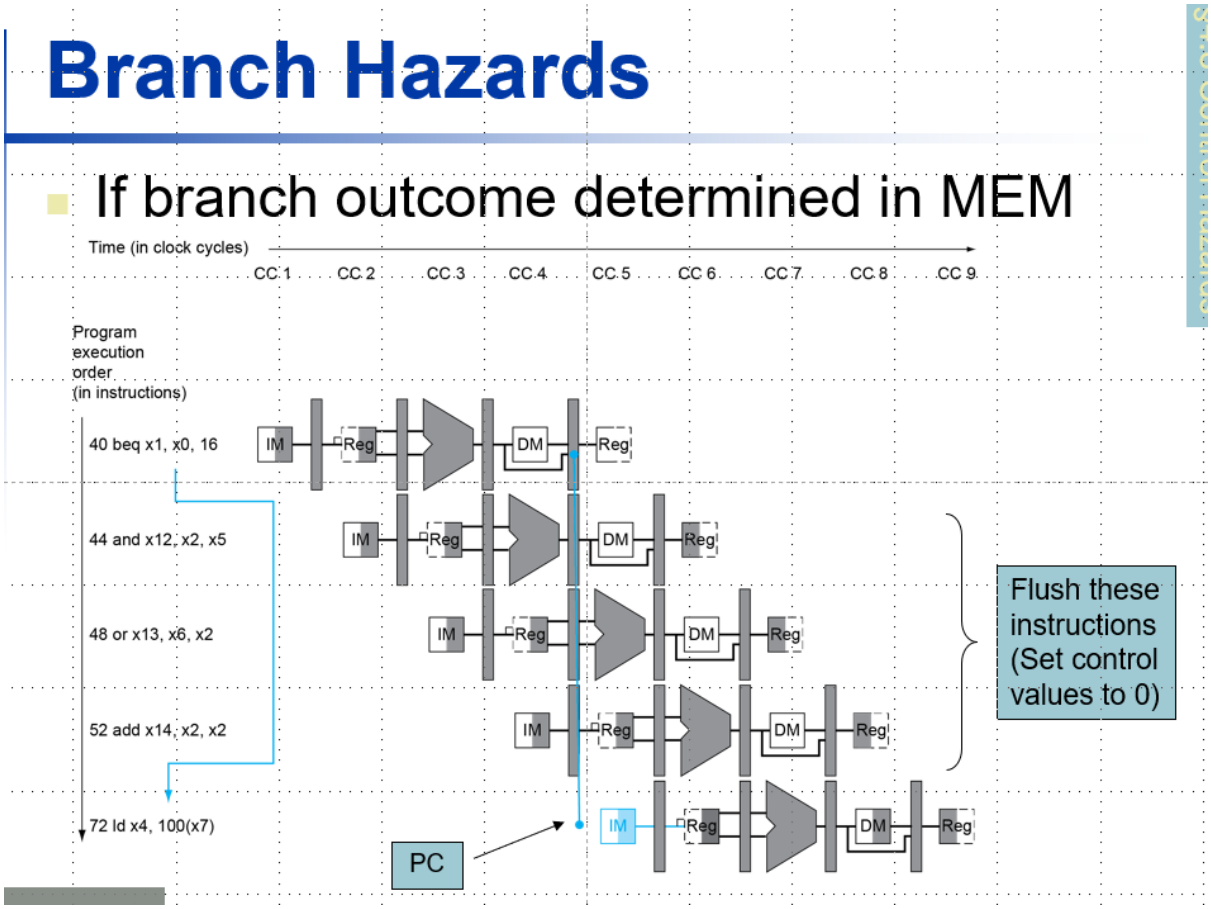


Stalling a cycle- solution of load-use hazard.

Thus, a load-use hazard is critical to address in order to maintain the efficiency and correctness of a pipelined architecture.

3.8 Control Hazards

A **control hazard** (or branch hazard) occurs when the pipeline does not know the next instruction to fetch due to a conditional branch.



Control hazard.

3.8.1 Example of Control Hazard

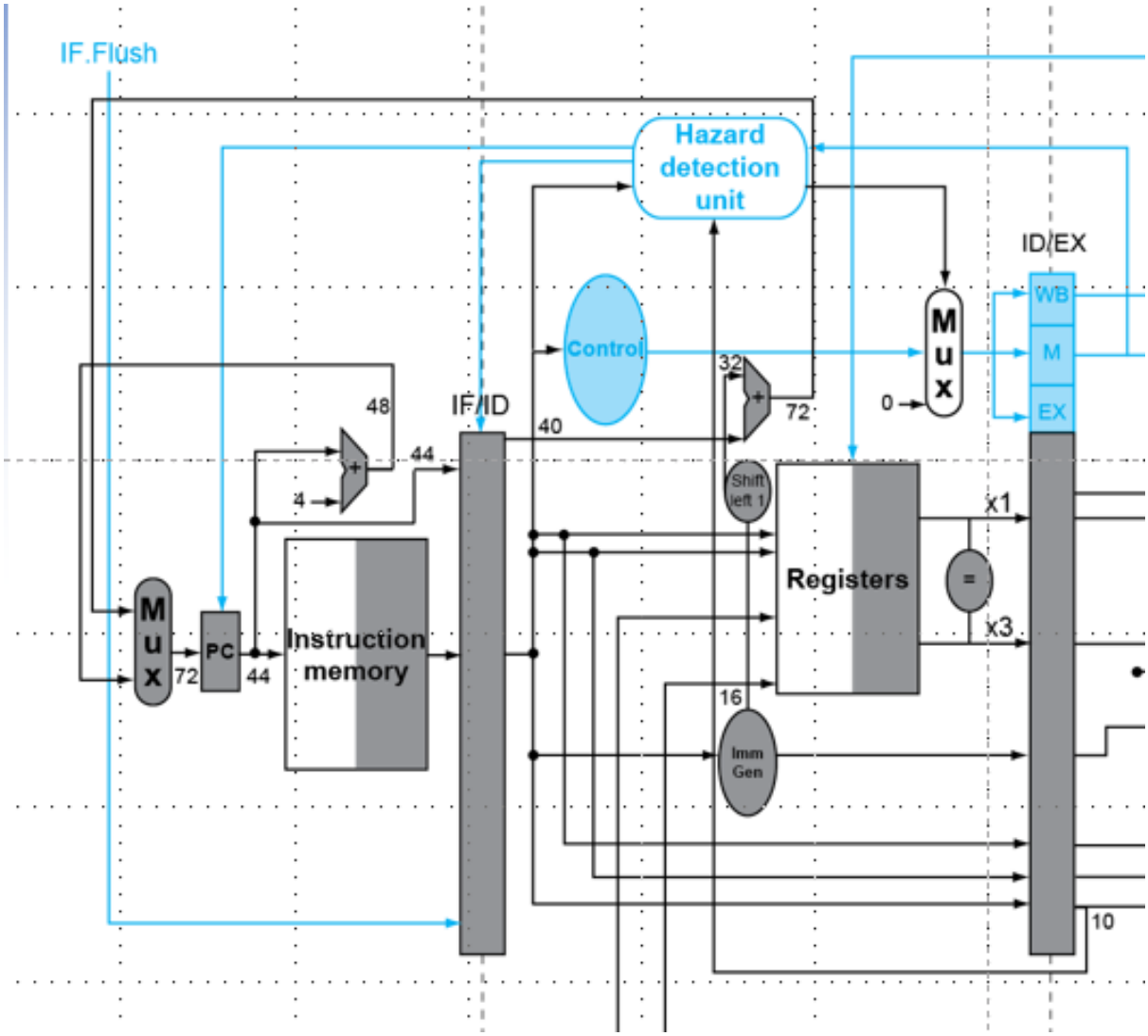
Consider a branch instruction that may or may not change the program counter (PC). Until the branch decision is made in the **EX** stage, the next instruction cannot be determined.

3.8.2 Solution to Control Hazards

- **Branch Prediction:**
  - Predict the branch outcome and continue fetching instructions based on the prediction.
  - If the prediction is correct, execution continues smoothly; if incorrect, the incorrect instructions are discarded, and the correct path is taken.
- **Pipeline Flushing:**
  - If a branch is mispredicted, the pipeline is flushed (cleared), and execution restarts from the correct address.
  - This introduces performance penalties due to wasted execution cycles.
- **Stall**

– We can implement the computation of the branch address in the decode stage itself (instead of in Execute and deciding the branch in Memory stage)so we can compute if branch needs to be taken or not at this stage and can stall this stage and in next stage give the correct PC value for computation. IF.flush is the signal which flushes the wrong instruction which was going to be processed in this cycle and by the next cycle correct PC has been calculated.

– We have implemented by finding the branch address in the decode stage and the branch signal is given to the IF/ID register and if it is 1, it flushes the current instruction and in next cycle correct instruction is fed to it.



Shifting the computation of branch address in decode..

3.9 Conclusion

While hazards in pipelining can cause delays and reduce performance, techniques like **hardware duplication, forwarding, instruction scheduling, branch prediction, and stalling** help mitigate their effects. By efficiently handling hazards, the pipeline achieves better instruction throughput and improved overall processor performance.

Chapter 4

Results and Discussion

Analyze and interpret the results of your project.

4.1 Sequential

```
0000000_00101_00000_000_00101_0110011  # add x5, x0, x5
0000000_00101_00101_000_00101_0110011  # add x5, x5, x5
0000000_00101_00101_000_00110_0110011  # add x6, x5, x5
0100000_00101_00110_000_00111_0110011  # sub x7, x6, x5
0000000_00000_00000_000_01000_0110011  # add x8, x0, x0
0000000_00101_01000_000_01000_0110011  # add x8, x8, x5
0000000_00111_01000_111_01010_0110011  # and x10, x8, x7
0000000_00000_00000_000_10000_1100011  # beq x0, x0, 8
0000000_00101_00110_110_01011_0110011  # or x11, x6, x5
0000000_00101_01001_000_01100_0110011  # add x12, x9, x5
0100000_00101_01111_000_01010_0110011  # sub x10, x15, x5
0000000_01010_00000_000_00000_1100011  # beq x10, x0, 0
0000000_00101_00111_000_01011_0110011  # add x11, x7, x5
0000000_00101_01011_000_01100_0110011  # add x12, x11, x5
0000000_00000_00000_000_00000_0110011  # add x0, x0, x0
0000000_00000_01110_011_01101_0000011  # ld x13, 0(x14)
0000000_01111_01110_011_00000_0100011  # sd x15, 0(x14)
0000000_00000_10001_011_10000_0000011  # ld x16, 0(x17)
0000000_10000_10011_000_10010_0110011  # add x18, x16, x19
1111111_11111_11111_111_11111_1111111  # halt instruction
```

Figure 4.1: Instruction testcase used for testing.

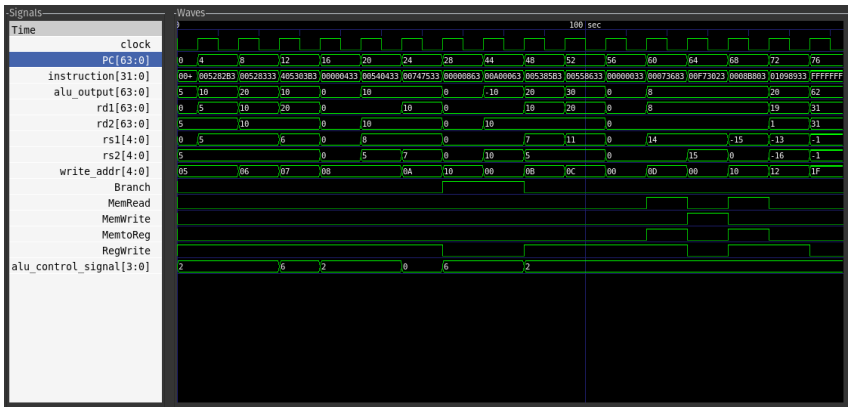


Figure 4.2: GTKWave of the complete sequential RISC-V processor.

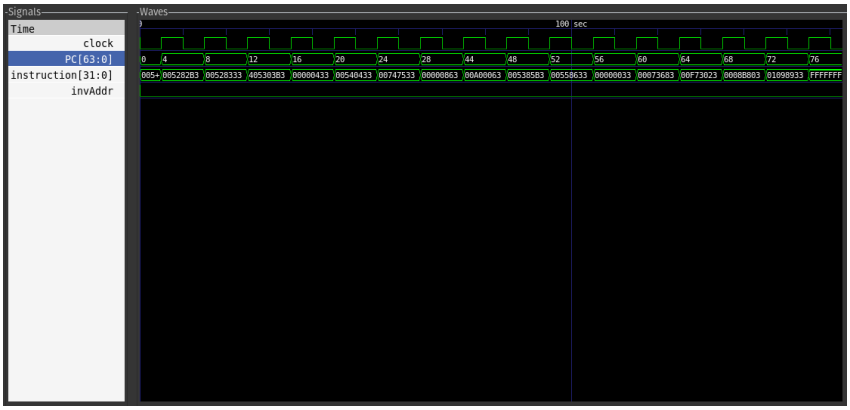


Figure 4.3: GTKWave of IF stage.

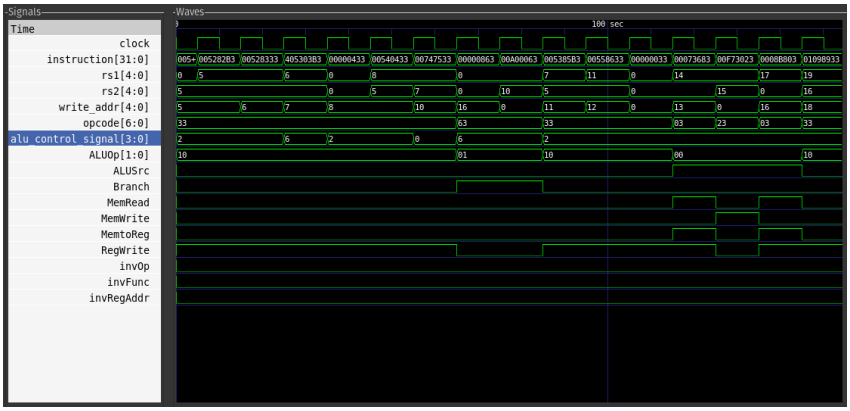


Figure 4.4: GTKWave of ID stage.

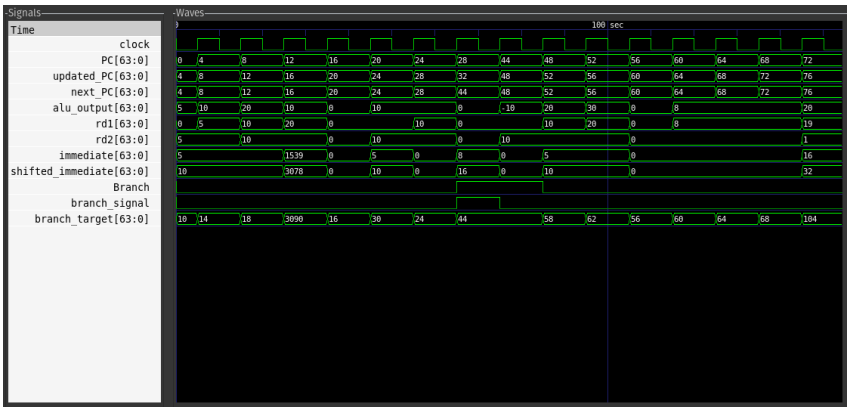


Figure 4.5: GTKWave of EX stage.

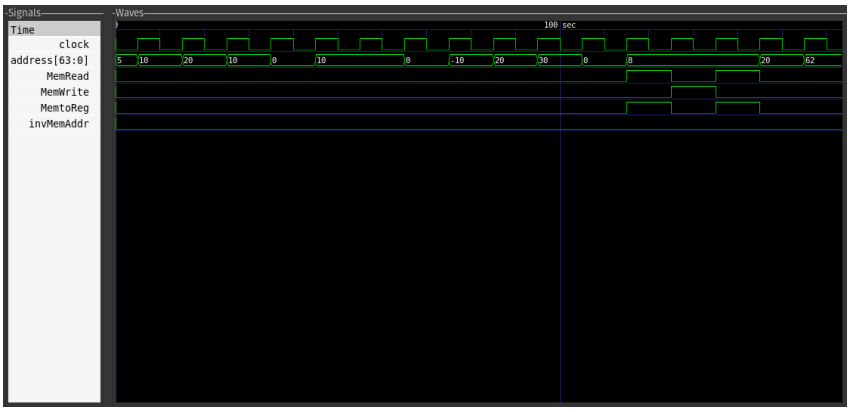


Figure 4.6: GTKWave of MEM stage.

PC	rs1	rs2	rd1	rd2	ALU	Write Data	Reg_Addr	Mem_Addr	Reg_Val	Mem_Val
0	0	5	0	5	5	5	5	0	5	x
4	5	5	5	5	10	10	5	1	5	31
8	5	5	10	10	20	20	6	2	6	x
12	6	5	20	10	10	10	7	1	7	31
16	0	0	0	0	0	0	8	0	8	x
20	8	5	0	10	10	10	8	1	0	31
24	8	7	10	10	10	10	10	1	10	31
28	0	0	0	0	0	0	16	0	16	x
44	0	10	0	10	-10	-10	0	-1	0	x
48	7	5	10	10	20	20	11	2	11	x
52	11	5	20	10	30	30	12	3	12	x
56	0	0	0	0	0	0	0	0	0	x

WARNING: Attempt to write in X0.

60	14	0	8	0	8	31	13	1	13	31
64	14	15	8	0	8	8	0	1	0	31
68	17	0	8	0	8	1	16	1	16	1
72	19	16	19	1	20	20	18	2	18	x
76	31	31	31	31	62	62	31	7	31	x

Program Execution completed.

Figure 4.7: Terminal output obtained

```
register[0]  = 64'd0;
register[1]  = 64'd1;
register[2]  = 64'd2;
register[3]  = 64'd3;
register[4]  = 64'd4;
register[5]  = 64'd5;
register[6]  = 64'd6;
register[7]  = 64'd7;
register[8]  = 64'd8;
register[9]  = 64'd9;
register[10] = 64'd10;
register[11] = 64'd11;
register[12] = 64'd12;
register[13] = 64'd13;
register[14] = 64'd8;
register[15] = 64'd15;
register[16] = 64'd16;
register[17] = 64'd8;
register[18] = 64'd18;
register[19] = 64'd19;
register[20] = 64'd20;
```

Figure 4.8: Register memory values obtained.

```
Register[0] =  0
Register[1] =  1
Register[2] =  2
Register[3] =  3
Register[4] =  4
Register[5] = 10
Register[6] = 20
Register[7] = 10
Register[8] = 10
Register[9] =  9
Register[10] = 10
Register[11] = 20
Register[12] = 30
Register[13] = 31
Register[14] =  8
Register[15] = 15
Register[16] =  1
Register[17] =  8
Register[18] = 20
Register[19] = 19
Register[20] = 20
```

Figure 4.9: Terminal output obtained.

4.2 Pipeline

Instructions performed:

```
1   0000000_00101_00000_000_00101_0110011  # add x5, x0, x5
2   0100000_10000_10011_000_10010_0110011  # sub x18, x16, x19
3   0000000_00110_01000_111_01000_0110011  # and x8, x8, x6
4   0100000_10001_00110_110_00111_0110011  # or x7, x6, x17
5   0000000_01000_01110_011_01101_0000011  # ld x13, 8(x14)
6   0000000_01111_01110_011_00000_0100011  # sd x15, 0(x14)
7   0000000_00000_00000_000_10000_1100011  # beq x0, x0, 8
8   0000000_00000_10001_011_10000_0000011  # ld x16, 0(x17)
9   0000000_00101_00000_000_01011_0110011  # add x11, x0, x5
10  0000000_10011_00101_110_00110_0110011  # or x6, x5, x19
11  0000000_10011_01110_000_10001_0110011  # add x17, x14, x19
```

Figure 4.10: instructions

```
Cycle 0:
PC:                  0
IF:  Instruction = 005002b3
------------------------------------------------------
Cycle                1:
PC:                  4
IF:  Instruction = 01000001000010011000100100110011  (41098933)
ID:  Instruction = 005002b3, Rs1 =  0, Rs2 =  5, Rd =  5, imm =                    5
EX:  ALU Control = 00000000, Alu_in1 = 0000000000000000, Alu_in2 = xxxxxxxxxxxxxxxx, Alu_output = xxxxxxxxxxxxxxxx
MEM: Address = 0000000000000000, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------
Cycle                2:
PC:                  8
IF:  Instruction = 00000000011001000111010000110011  (00647433)
ID:  Instruction = 41098933, Rs1 = 19, Rs2 = 16, Rd = 18, imm =                 1040
EX:  ALU Control = 005002b3, Alu_in1 = 0000000000000000, Alu_in2 = 0000000000000005, Alu_output = 0000000000000005
MEM: Address = xxxxxxxxxxxxxxxx, MemRead = 0, MemWrite = 0, Data = xxxxxxxxxxxxxxxx
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------
Cycle                3:
PC:                 12
IF:  Instruction = 01000001000100110110001110110011  (411363b3)
ID:  Instruction = 00647433, Rs1 =  8, Rs2 =  6, Rd =  8, imm =                    6
EX:  ALU Control = 41098933, Alu_in1 = 0000000000000013, Alu_in2 = 0000000000000010, Alu_output = 0000000000000003
MEM: Address = 0000000000000005, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 0, WriteReg =  0, WriteData = xxxxxxxxxxxxxxxx
------------------------------------------------------
Cycle                4:
PC:                 16
IF:  Instruction = 00000000100001110011011010000011  (00873683)
ID:  Instruction = 411363b3, Rs1 =  6, Rs2 = 17, Rd =  7, imm =                 1041
EX:  ALU Control = 00647433, Alu_in1 = 0000000000000008, Alu_in2 = 0000000000000006, Alu_output = 0000000000000000
MEM: Address = 0000000000000003, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 1, WriteReg =  5, WriteData = 0000000000000005
------------------------------------------------------
Cycle                5:
PC:                 20
IF:  Instruction = 00000000111101110011000000100011  (00f73023)
ID:  Instruction = 00873683, Rs1 = 14, Rs2 =  x, Rd = 13, imm =                    8
EX:  ALU Control = 411363b3, Alu_in1 = 0000000000000006, Alu_in2 = 0000000000000008, Alu_output = 0000000000000000
MEM: Address = 0000000000000000, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 1, WriteReg = 18, WriteData = 0000000000000003
```

Figure 4.11: Pipeline terminal outputs

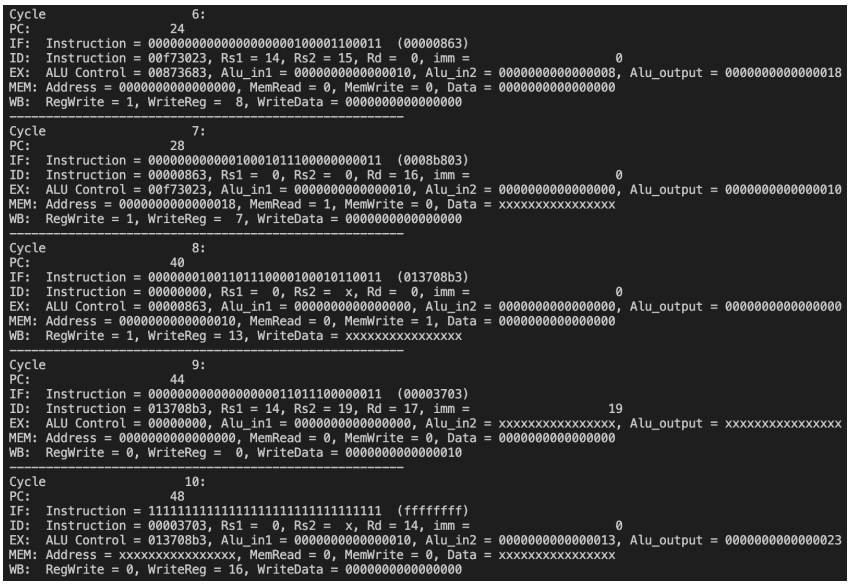


Figure 4.12: Pipeline terminal outputs

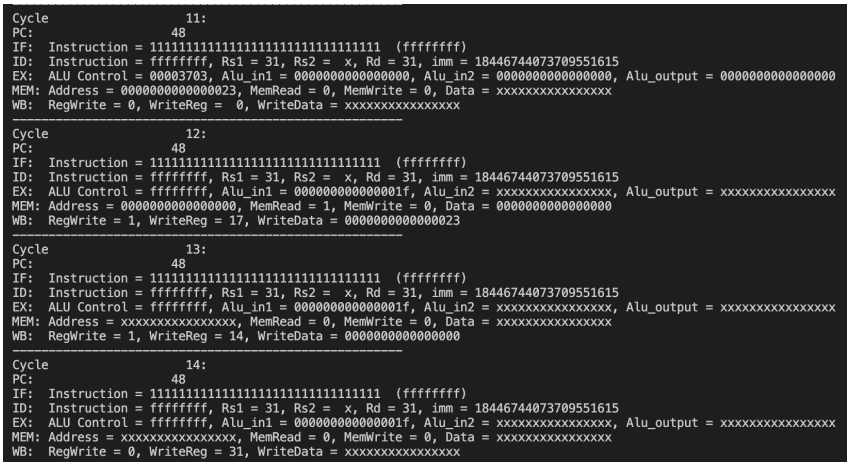


Figure 4.13: Pipeline terminal outputs

```
Register[0] =  0
Register[1] =  1
Register[2] =  2
Register[3] =  3
Register[4] =  4
Register[5] =  5
Register[6] =  6
Register[7] =  0
Register[8] =  0
Register[9] =  9
Register[10] = 10
Register[11] = 11
Register[12] = 12
Register[13] = 13
Register[14] = 16
Register[15] = 15
Register[16] = 16
Register[17] =  8
Register[18] = 18
Register[19] = 19
Register[20] = 20
```

Figure 4.14: Pipeline terminal outputs

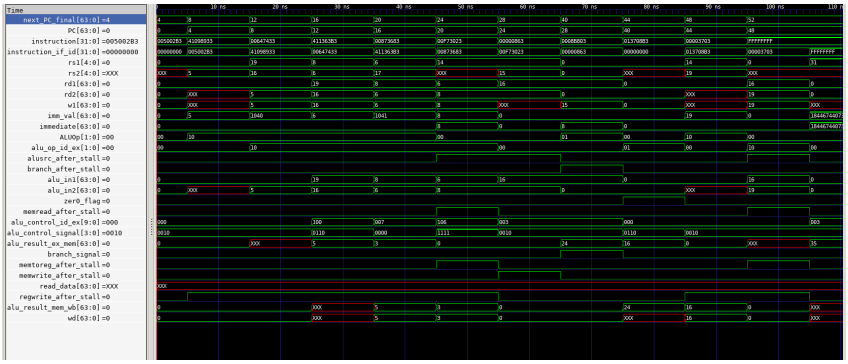


Figure 4.15: Pipeline GTKwave outputs

4.3 Handling Hazards

4.3.1 Data Hazard

```
0000000_00101_00000_000_00101_0110011  # add x5, x0, x5
0000000_00101_00101_000_00101_0110011  # add x5, x5, x5
0000000_00110_01000_111_01000_0110011  # and x8, x8, x6
0000000_10011_01110_000_10001_0110011  # add x17, x14, x19
```

Figure 4.16: Data hazard instruction

```
Cycle 0:
PC:                  0
IF:  Instruction = 005002b3
------------------------------------------------------
Cycle                1:
PC:                  4
IF:  Instruction = 00000000101001010000010110011
ID:  Instruction = 005002b3, Rs1 =  0, Rs2 =  5, Rd =  5, imm =                  5
EX:  ALU Control = 00000000, Alu_in1 = 0000000000000000, Alu_in2 = xxxxxxxxxxxxxxx, Alu_output = xxxxxxxxxxxxxxx Forward1 = 00, Forward2
 = 00
MEM: Address = 0000000000000000, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------
Cycle                2:
PC:                  8
IF:  Instruction = 00000000011001000111010000110011
ID:  Instruction = 005282b3, Rs1 =  5, Rs2 =  5, Rd =  5, imm =                  5
EX:  ALU Control = 005002b3, Alu_in1 = 0000000000000000, Alu_in2 = 0000000000000005, Alu_output = 0000000000000005 Forward1 = 00, Forward2
 = 00
MEM: Address = xxxxxxxxxxxxxxxx, MemRead = 0, MemWrite = 0, Data = xxxxxxxxxxxxxxx
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------
Cycle                3:
PC:                 12
IF:  Instruction = 00000001001101110000100010110011
ID:  Instruction = 00647433, Rs1 =  8, Rs2 =  6, Rd =  8, imm =                  6
EX:  ALU Control = 005282b3, Alu_in1 = 0000000000000005, Alu_in2 = 0000000000000005, Alu_output = 000000000000000a Forward1 = 10, Forward2
 = 10
MEM: Address = 0000000000000005, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 0, WriteReg =  0, WriteData = xxxxxxxxxxxxxxx
------------------------------------------------------
```

Figure 4.17: Handling Data hazard: Terminal Output

```
Cycle                4:
PC:                 16
IF:  Instruction = 11111111111111111111111111111111
ID:  Instruction = 013708b3, Rs1 = 14, Rs2 = 19, Rd = 17, imm =                 19
EX:  ALU Control = 00647433, Alu_in1 = 0000000000000008, Alu_in2 = 0000000000000006, Alu_output = 0000000000000000 Forward1 = 00, Forward2
 = 00
MEM: Address = 000000000000000a, MemRead = 0, MemWrite = 0, Data = 0000000000000100
WB:  RegWrite = 1, WriteReg =  5, WriteData = 0000000000000005
------------------------------------------------------
Cycle                5:
PC:                 16
IF:  Instruction = 11111111111111111111111111111111
ID:  Instruction = ffffffff, Rs1 = 31, Rs2 =  x, Rd = 31, imm = 18446744073709551615
EX:  ALU Control = 013708b3, Alu_in1 = 0000000000000010, Alu_in2 = 0000000000000013, Alu_output = 0000000000000023 Forward1 = 00, Forward2
 = 00
MEM: Address = 0000000000000000, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 1, WriteReg =  5, WriteData = 000000000000000a
------------------------------------------------------
Cycle                6:
PC:                 16
IF:  Instruction = 11111111111111111111111111111111
ID:  Instruction = ffffffff, Rs1 = 31, Rs2 =  x, Rd = 31, imm = 18446744073709551615
EX:  ALU Control = ffffffff, Alu_in1 = 000000000000001f, Alu_in2 = xxxxxxxxxxxxxxx, Alu_output = xxxxxxxxxxxxxxx Forward1 = 00, Forward2
 = 00
MEM: Address = 0000000000000023, MemRead = 0, MemWrite = 0, Data = xxxxxxxxxxxxxxx
WB:  RegWrite = 1, WriteReg =  8, WriteData = 0000000000000000
------------------------------------------------------
Cycle                7:
PC:                 16
IF:  Instruction = 11111111111111111111111111111111
ID:  Instruction = ffffffff, Rs1 = 31, Rs2 =  x, Rd = 31, imm = 18446744073709551615
```

Figure 4.18: Handling Data hazard: Terminal Output

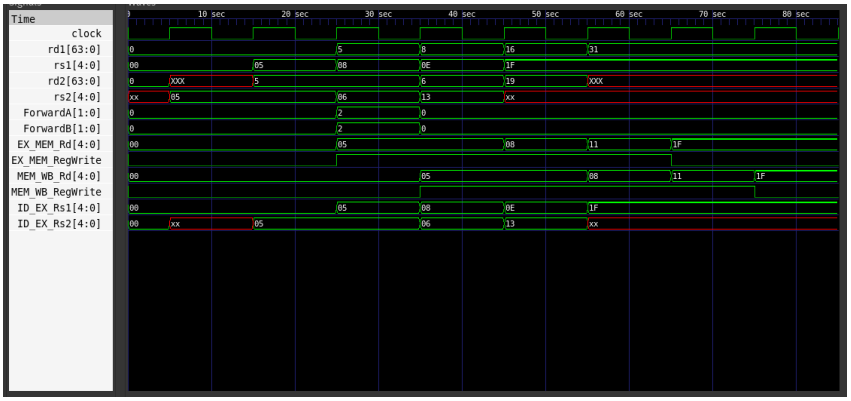


Figure 4.19: Handling Data hazard: GTKWave Output

4.3.2 Load - Use Hazard

```
0000000_00000_10001_011_10000_0000011  # ld x16, 0(x17)
0000000_10000_10011_000_10010_0110011  # add x18, x16, x19
0000000_00101_00110_110_01011_0110011  # or x11, x6, x5
```

Figure 4.20: Load - use hazard instruction

```
Cycle 0:
PC:                   0
IF:  Instruction = 0008b803
------------------------------------------------------------
Cycle                 1:
PC:                   4
IF:  Instruction = 00000001000010011000100110011
ID:  Instruction = 0008b803, Rs1 = 17, Rs2 =  x, Rd = 16, imm =  0, stall = 0, IF_ID_Write = 1, PCWrite = 1
EX:  ALU Control = 00000000, Alu_in1 = 0000000000000000, Alu_in2 = xxxxxxxxxxxxxxxx, Alu_output = xxxxxxxxxxxxxxxx
MEM: Address = 0000000000000000, MemRead = 0, MemWrite = 0, Data = 0000000000000000
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------------
Cycle                 2:
PC:                   8
IF:  Instruction = 00000000010100110110010110110011
ID:  Instruction = 01098933, Rs1 = 19, Rs2 = 16, Rd = 18, imm = 16, stall = 1, IF_ID_Write = 0, PCWrite = 0
EX:  ALU Control = 0008b803, Alu_in1 = 0000000000000008, Alu_in2 = 0000000000000000, Alu_output = 0000000000000008
MEM: Address = xxxxxxxxxxxxxxxx, MemRead = 0, MemWrite = 0, Data = xxxxxxxxxxxxxxxx
WB:  RegWrite = 0, WriteReg =  0, WriteData = 0000000000000000
------------------------------------------------------------
Cycle                 3:
PC:                   8
IF:  Instruction = 00000000010100110110010110110011
ID:  Instruction = 01098933, Rs1 = 19, Rs2 = 16, Rd = 18, imm = 16, stall = 0, IF_ID_Write = 1, PCWrite = 1
EX:  ALU Control = 01098933, Alu_in1 = 0000000000000013, Alu_in2 = 0000000000000008, Alu_output = 000000000000001b
MEM: Address = 0000000000000008, MemRead = 1, MemWrite = 0, Data = 0000000000000100
WB:  RegWrite = 0, WriteReg =  0, WriteData = xxxxxxxxxxxxxxxx
------------------------------------------------------------
Cycle                 4:
PC:                  12
IF:  Instruction = 11111111111111111111111111111111
```

Figure 4.21: Handling Load - use hazard: Terminal Output

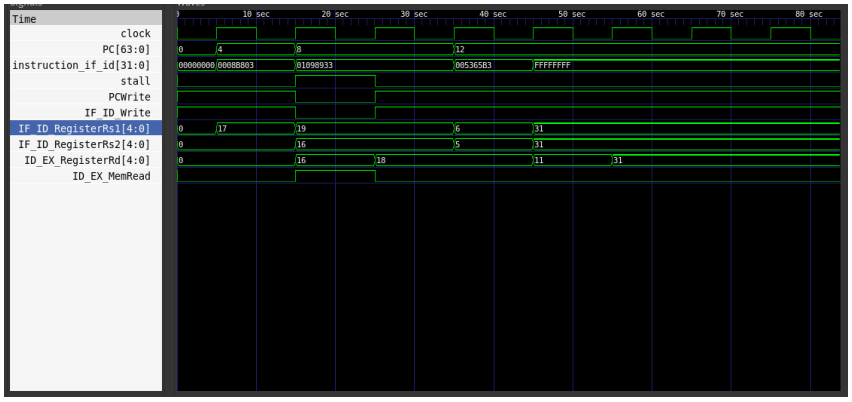


Figure 4.22: Handling Load - use hazard: GTKWave Output

4.3.3 Control Hazards

```
0000000_00000_00000_000_10000_1100011  # beq x0, x0, 8
0000000_00000_10001_011_10000_0000011  # ld x16, 0(x17)
0000000_00101_00000_000_01011_0110011  # add x11, x0, x5
0000000_10011_00101_110_00110_0110011  # or x6, x5, x19
0000000_10011_01110_000_10001_0110011  # add x17, x14, x19
0000000_00000_00000_011_01110_0000011  # ld x14, 0(x0)
```

Figure 4.23: Control hazard instruction

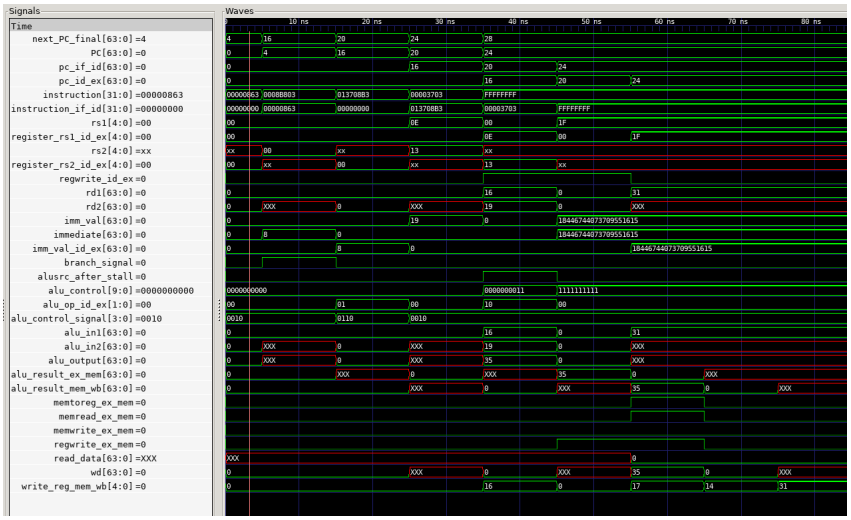


Figure 4.24: Control Hazard: GTKwave output

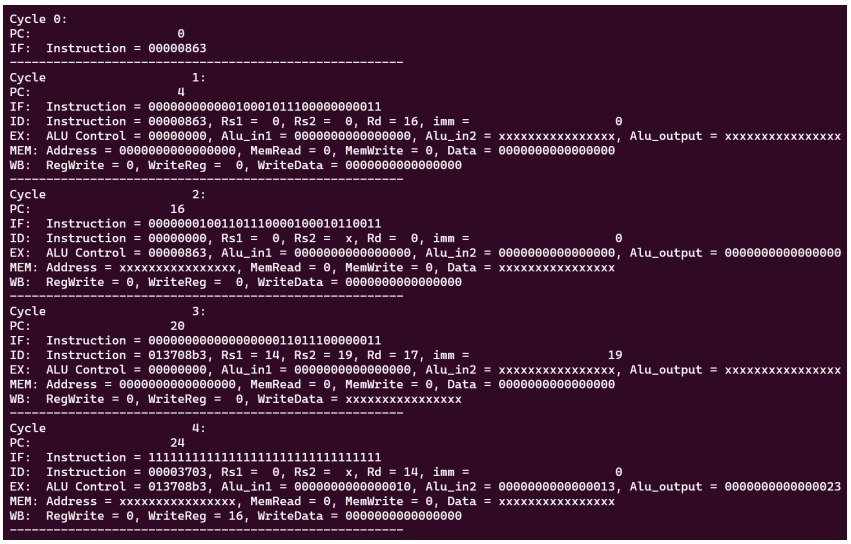


Figure 4.25: Control Hazard-terminal

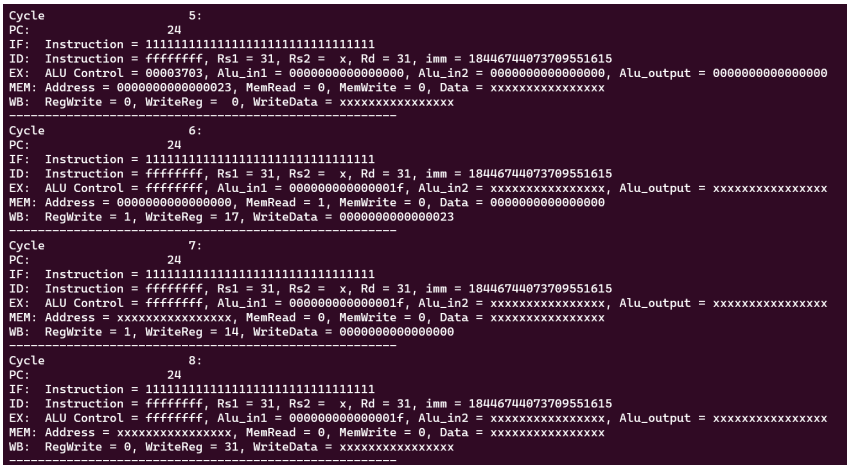


Figure 4.26: Control Hazard-terminal

Chapter 5

Difference Between Sequential and Pipelined Implementation

5.1 Sequential (Single-Cycle) Implementation

A sequential or single-cycle implementation executes one instruction completely in a single clock cycle. Each instruction goes through all the execution stages—fetch, decode, execute, memory access, and write-back—within one cycle.

5.1.1 Characteristics of Sequential Implementation

- Each instruction is completed in one clock cycle.
- Simple control logic due to lack of overlapping instructions.
- Requires a longer clock cycle to accommodate the slowest instruction.
- Inefficient use of hardware, as some components remain idle during certain operations.

5.2 Pipelined Implementation

A pipelined processor divides instruction execution into multiple stages and allows multiple instructions to be processed simultaneously. As each instruction progresses through the pipeline, new instructions can enter execution at different stages, improving overall throughput.

5.2.1 Characteristics of Pipelined Implementation

- Each instruction is broken down into smaller stages, allowing multiple instructions to execute concurrently.
- Reduces the average instruction execution time, increasing overall performance.
- Requires additional hardware, such as pipeline registers, to store intermediate results between stages.

- Increases complexity due to hazards such as data dependencies, control hazards, and structural hazards.

5.3 Comparison Between Sequential and Pipelined Implementation

Feature	**Sequential Implementation**	**Pipelined Implementation**
Execution Type	One instruction at a time	Multiple instructions in parallel
Clock Cycle Time	Longer (depends on slowest instruction)	Shorter (stages operate concurrently)
Hardware Complexity	Simple	More complex due to pipeline registers and hazard handling
Performance	Lower throughput	Higher throughput due to instruction overlap
Resource Utilization	Inefficient (some components idle)	Efficient (components used continuously)

Table 5.1: Comparison of Sequential and Pipelined Processor Implementations

5.4 Conclusion

While sequential implementation is simpler and easier to design, pipelined implementation significantly improves performance by allowing multiple instructions to be processed simultaneously. However, pipelining introduces additional complexity in the form of hardware requirements and hazard management. This project explores both implementations to understand their trade-offs and impact on processor performance.

Chapter 6

Conclusion and Contribution

A processor is a computing engine that executes a sequence of instructions stored in memory, performing operations such as arithmetic, logic, and data manipulation to achieve desired outcomes. In our implementation, we first developed a **sequential processor** where each instruction is fully executed one after the other, following a single-cycle design that emphasizes correctness and simplicity. Later, to enhance performance and improve throughput, we restructured the design into a **pipelined processor**. This pipelined approach divides the instruction execution into multiple stages (such as Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write-back) and introduces clocked pipeline registers between each stage. By linking the output of one stage directly to the input of the next, the pipelined design preserves the logical operations of the sequential model while allowing multiple instructions to be processed concurrently. This transformation significantly increases instruction throughput without compromising the processor's functionality.

To address the challenges introduced by pipelining, we implemented techniques to handle various hazards. Data hazards, including load-use hazards, were mitigated through methods such as data forwarding and pipeline stalling. For control hazards, we employed strategies like stalling and flushing the pipeline to maintain correct program execution. This comprehensive approach ensured that the processor operates efficiently while maintaining correctness across both sequential and pipelined implementations.

6.0.1 Contribution

- SAMPATH: Implemented all the different stages like fetch, decode, etc. Also handled the hazard detection unit and the control hazard.
- HARSHITA: Implemented the datapath for the sequential part, handled the data hazard forwarding unit and its integration.
- GARIMA: Implemented registers and datapath of the pipelining stage and integrated everything together in pipelining.

ALL tested the code for different sets of instructions and hazards and then debugged the code. Everyone also compiled the report together.

Chapter 7

References

- "Computer Organization and Design RISC-V Edition- The Hardware Software Interface" by David Patterson and John Hennessy.
- Class Slides
- https://www.youtube.com/playlist?list=PL-Mfq5QS-s8iUJpNzCOtQKRfpswCrPbiW