RISC-V PIPELINED PROCESSOR
IPA PROJECT-2025

Team Number – 3

Aikya Oruganti (2023102062)

Indira C Reddy (2023102070)

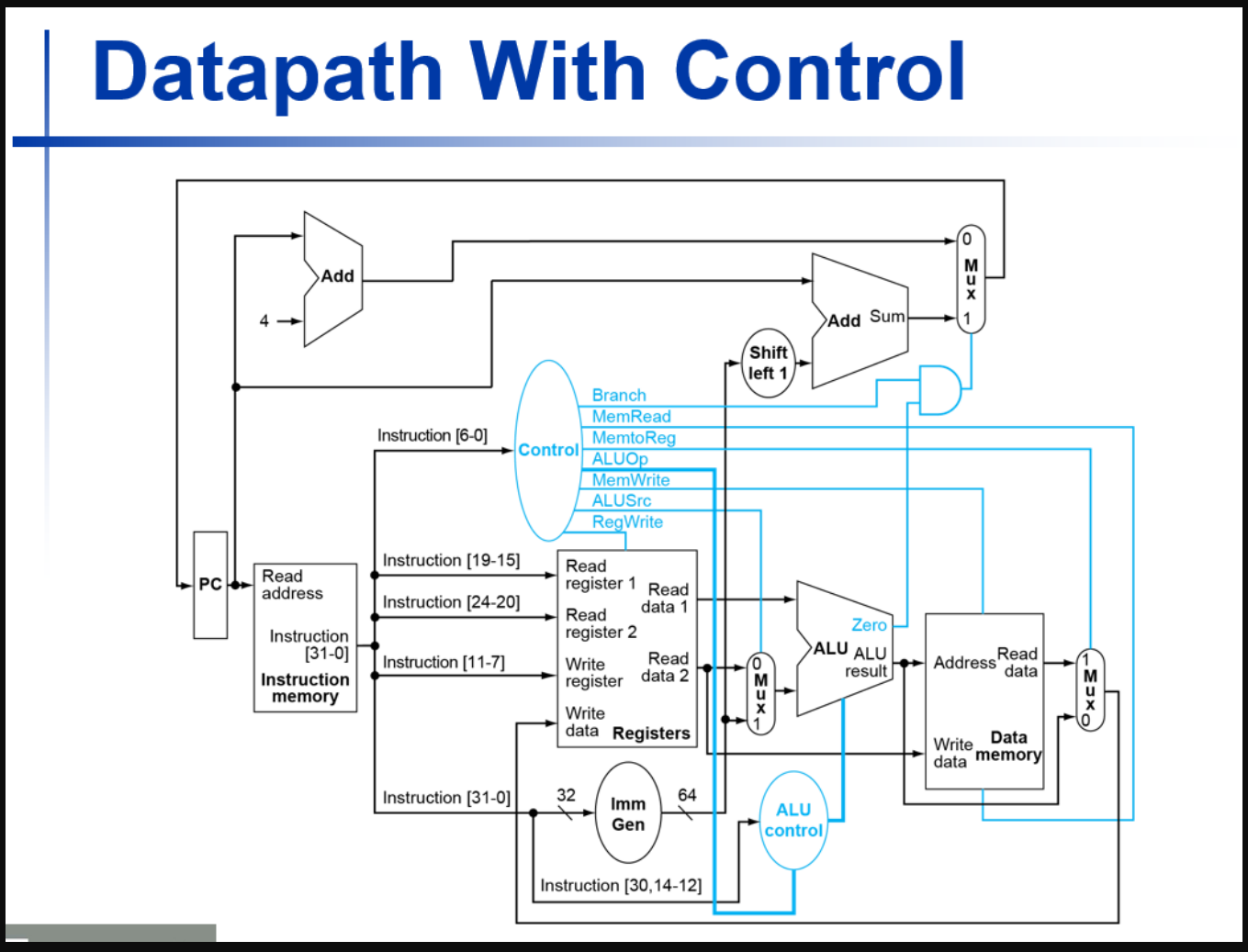
Pragnya Tatiparthi (2023102067)

ABSTRACT

This project uses a 5-stage pipelined processor using the RISC-V instruction set architecture (ISA). The design is different from a sequential processor in that it uses parallelism to enhance throughput by running several instructions in parallel across different pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Writeback (WB). The report explains both sequential and pipelined datapaths, with emphasis on their components, operation, and hazard handling mechanisms. Important modules like fetch_cycle, decode_cycle, execute_cycle, and auxiliary units like Register_File, Instruction_Memory, and ALU are studied. Simulation output and waveform traces confirm the correctness of the design

SEQUENTIAL DATAPATH

A sequential processor executes instructions in a step-by-step manner by completing one instruction fully before moving to the next.



PROGRAM COUNTER

It maintains the address of the current instruction.
Increments the PC value by 4 bytes in case there are no branches.

Else it goes to the branch target.

Implementation: A register updated each cycle, with a multiplexer selecting between PC+4 and branch target.

INSTRUCTION MEMORY

Its main purpose is to store and retrieve instructions based on the PC.

Decodes the instruction based on the opcode and divides the instruction into opcode (decides what operation it is), input registers, output register, funct3, funct7 values. This decides the type of operation that needs to be executed. Also helps in generating ALUOP which decides what operation the ALU performs.

Name (Bit position)	Fields					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

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

opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

The Type of operation the ALU performs-

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

For **BEQ**-

ALU performs subtraction. It subtracts both the register values and gives out a zero flag based on its value.

For **LOAD and STORE** -
ALU always performs addition to find the target address.

For **R type**-

Depends on what the opcode is. ADD, SUB, AND, OR are available operations.

REGISTER FILE

Its main purpose is to store 32 64-bit registers (x0 to x31)

The addresses from the instruction memory (rs1 , rs2 , rd) goes to the register file to get the data contained in these registers. This data goes on into the ALU to get operations performed.

IMMEDIATE GENERATOR

The entire instruction goes into immediate gen in case of branch and load instructions to generate the target register addresses. It sign extends the 32 bit instruction to a 64 bit one(extends the MSB for 32 more bits)

CONTROL UNIT

The opcode goes into the control unit which generates various control signals to activate different blocks.

These are the signals generated-

- Register Write (RegWrite) : enables register write.
- Immediate Source (ImmSrc) : Selects immediate format (I, S, B types).
- ALU Source (ALUSrc): Chooses between register (RD2 that is data from rs2) and immediate for ALU.
- Memory Write (MemWrite): Enables memory write(sd).
- Result Source (ResultSrc): Selects ALU rseult or memory data for writeback.
- Branch (Branch): Indicates a branch instruction(beq).
- ALU Operation (ALUOp): Specifies ALU operation.

EXECUTE BLOCK

It performs arithmetic and logical operations with the help of ALU.

Theres a mux between the ALU and decode stage. It decides if the input is numerical data (for R type instructions) or if the inputs are addresses for ld ,sd and branches.

Includes the ALU which performs operations depending on the AluOp given to it.

The Alu returns zero flag if the subtraction value in branch opns is zero.

Else it outputs the result of the operation and it gives into the next stage.

ALU

We're implementing an ALU with the operations ADD SUB AND and OR. The design utilizes a **64-bit Ripple Carry Adder** to handle addition and subtraction.

- Addition is performed using a **64-bit Ripple Carry Adder**, which consists of a series of full adders connected in sequence.
- Subtraction is carried out using the same 64-bit Ripple Carry Adder, where the second operand is converted to its 2's complement (by inverting all bits and adding 1) before being passed to the adder, effectively transforming subtraction into an addition problem: A−B=A+(2's complement of B).

- A bitwise AND operation is applied to corresponding bits of both operands, setting a bit to 1 only if both corresponding bits are 1, while a bitwise OR operation sets a bit to 1 if at least one of the corresponding operand bits is 1.

SHIFT LEFT BLOCK

Contains the Shift left operation for when branch target needs to be found.

Branch target=PC+ immediate*2(essentially shift left by 1).

-Branch target computation

- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

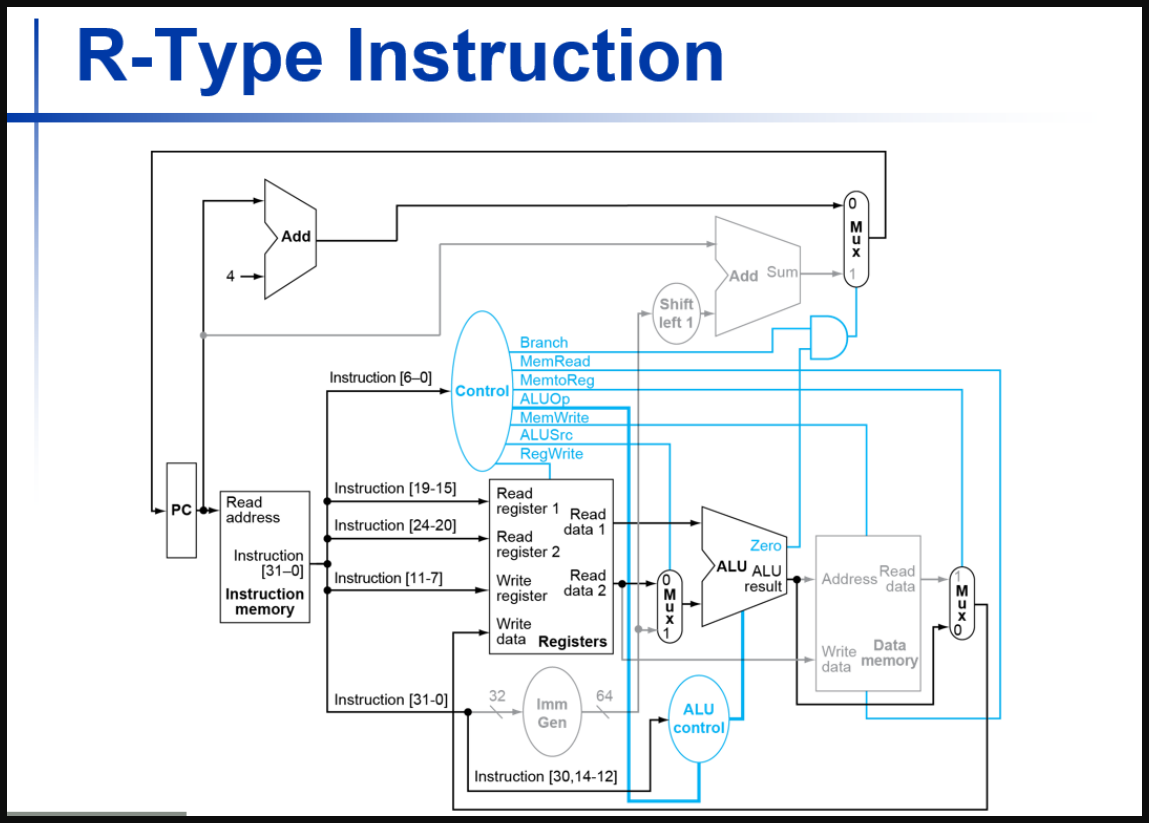
MEMORY STAGE

It has access to the memory from which data is either loaded (ld) or stored into(sd). It has control signals (MemWrite and MemRead) which are activated depending on what operation is being performed.

WRITEBACK

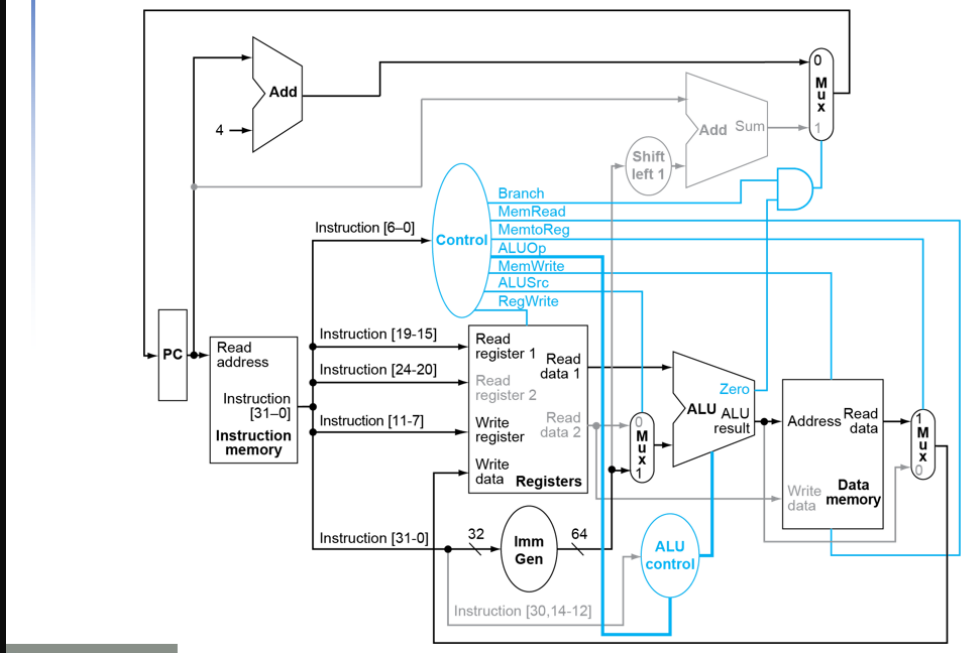
Finally, there's writeback which writes the output data back into the register file. There's a 2x1 Mux which decides if the ALU result must be written back (R type instructions) or if address must be written back(for load) into the rd register.

DATAPATH FOR DIFFERENT TYPES OF INSTRUCTIONS



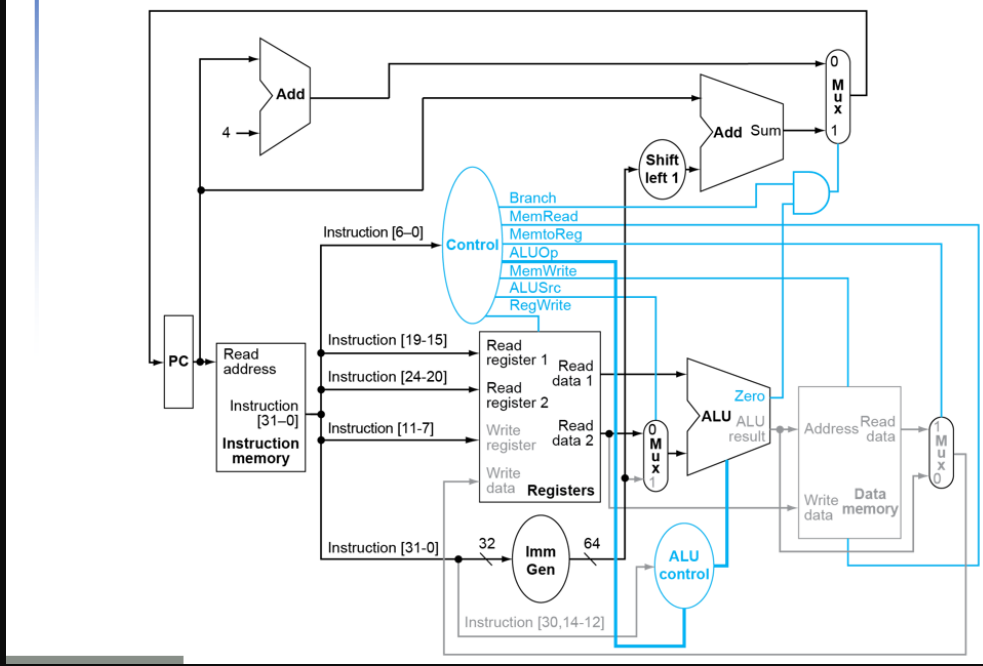
R type instructions have 4 stages except memory stage. PC incremented by 4. Instruction is divided into various parts. Depending on the opcode ALU computes the value. The result from ALU can directly be written into rd with writeback stage.

Load Instruction



L type has all the stages. PC is incremented by 4. Instruction is decoded. Value of address is calculated by the ALU after sign extending. Then the value from that address in the memory is loaded into the register and that value is written back into rd.

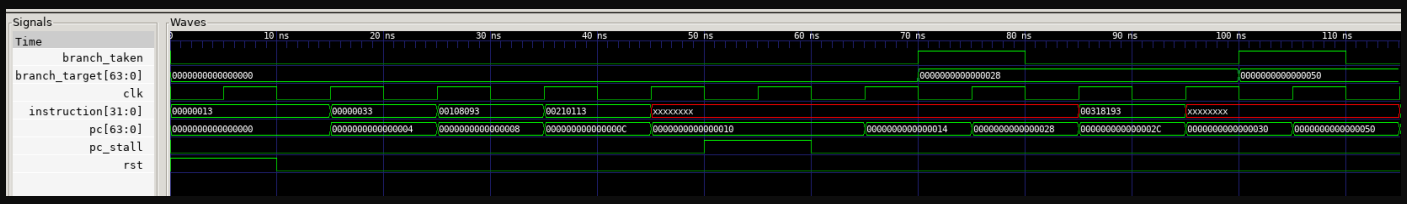
BEQ Instruction



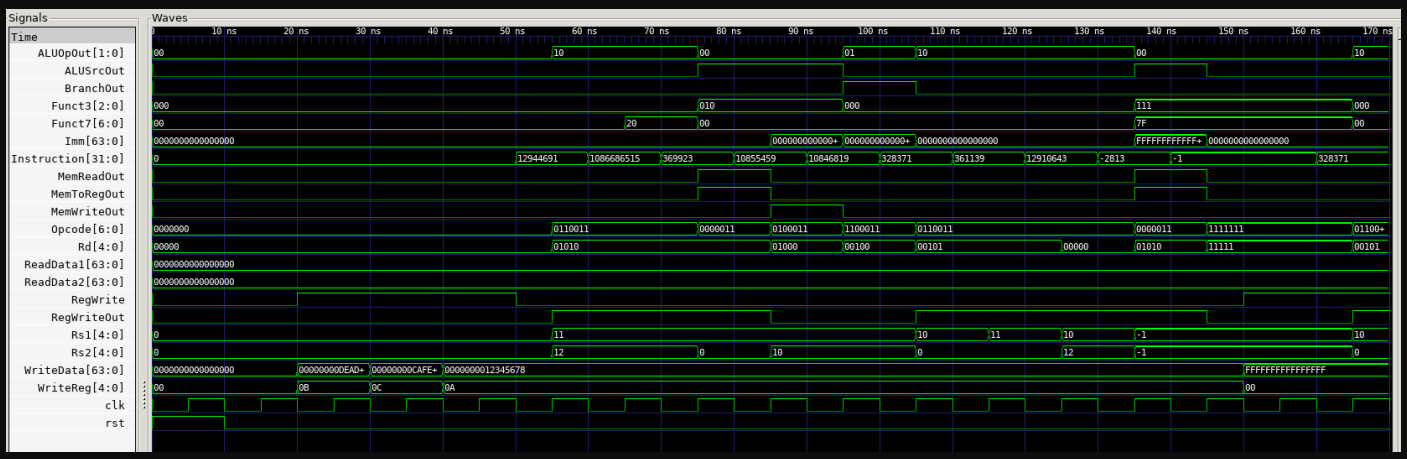
Only the first 3 stages are part of it. The PC value takes the value of branch target. The ALU gives the zero flag based on the subtraction value. An AND gate takes this zero flag and the branch control value and decides if the branch address must be taken or if +4 must be done.

GTK WAVE PLOTS

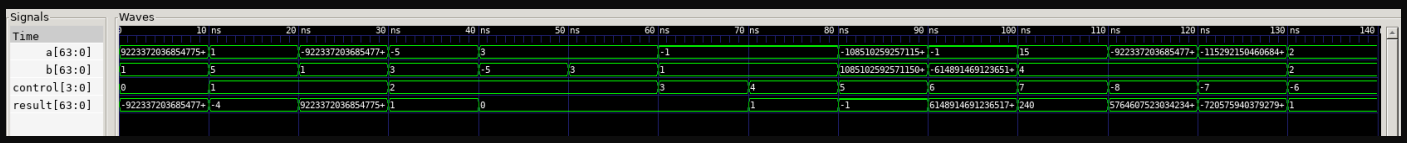
FETCH:



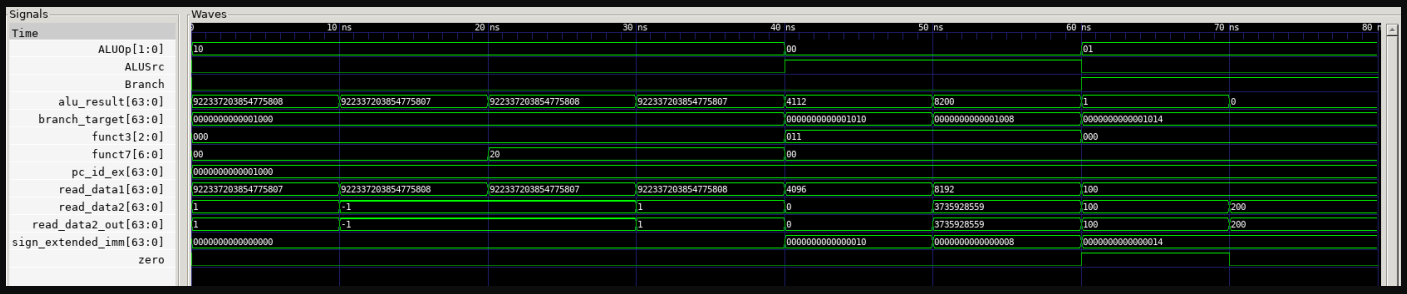
DECODE:



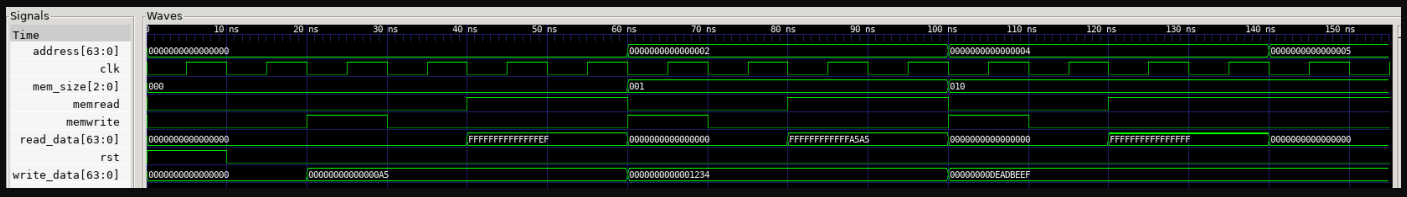
ALU:



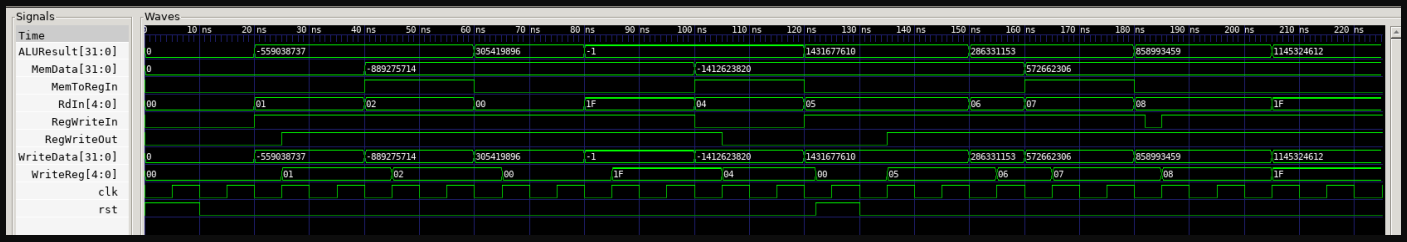
EXECUTE:



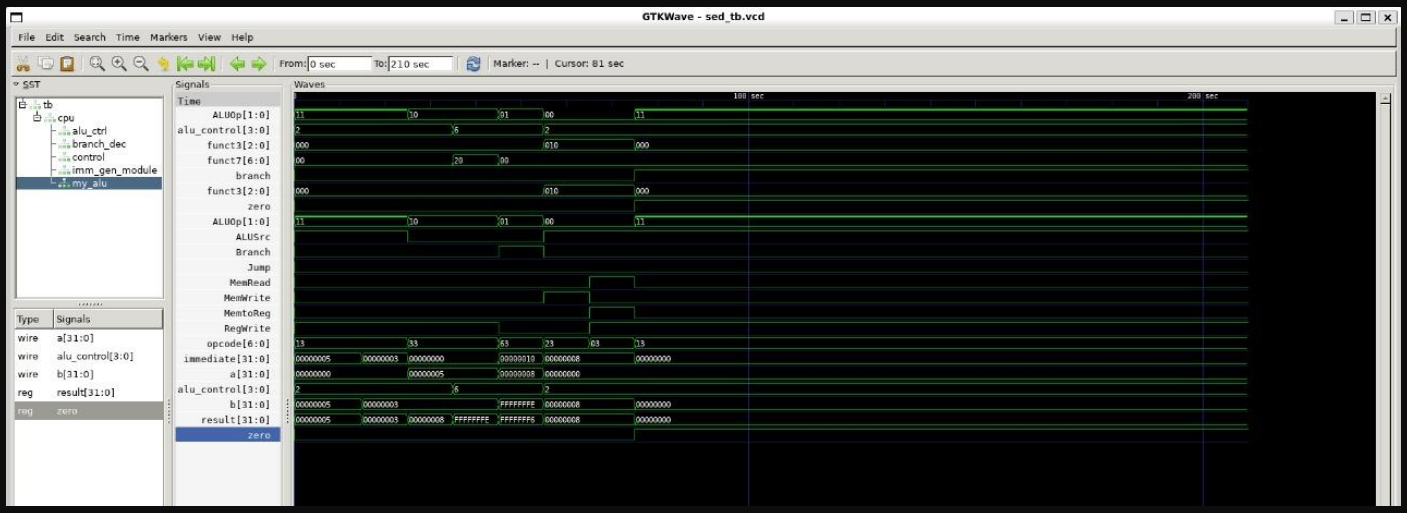
MEMORY:



WRITEBACK:



TOTAL SEQUENTIAL:

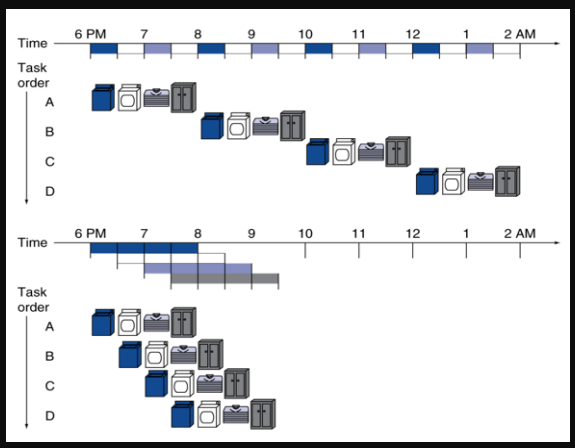


PERFORMANCE

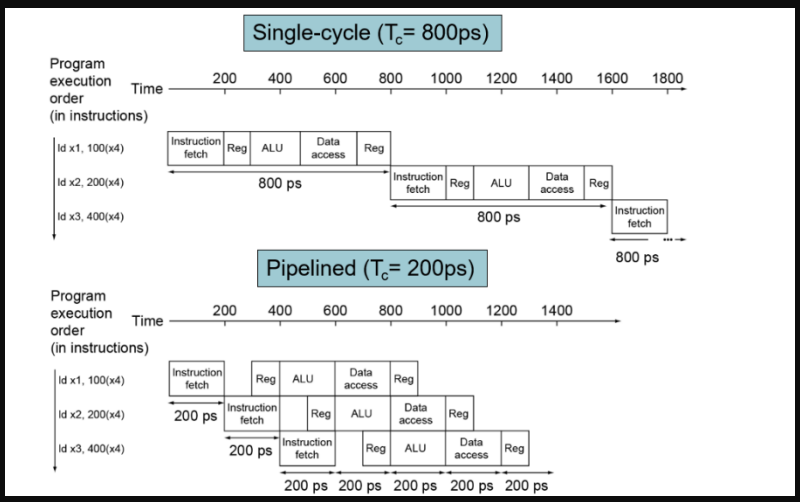
- Latency: 3-5 cycles per instruction
- Throughput: One instruction every 3-5 cycles.
- Clock Period: Short, limited by the slowest stage.

PIPELINED DATAPATH

Pipelining is a subset of parallelism. It lets multiple instructions run at once to make it efficient.

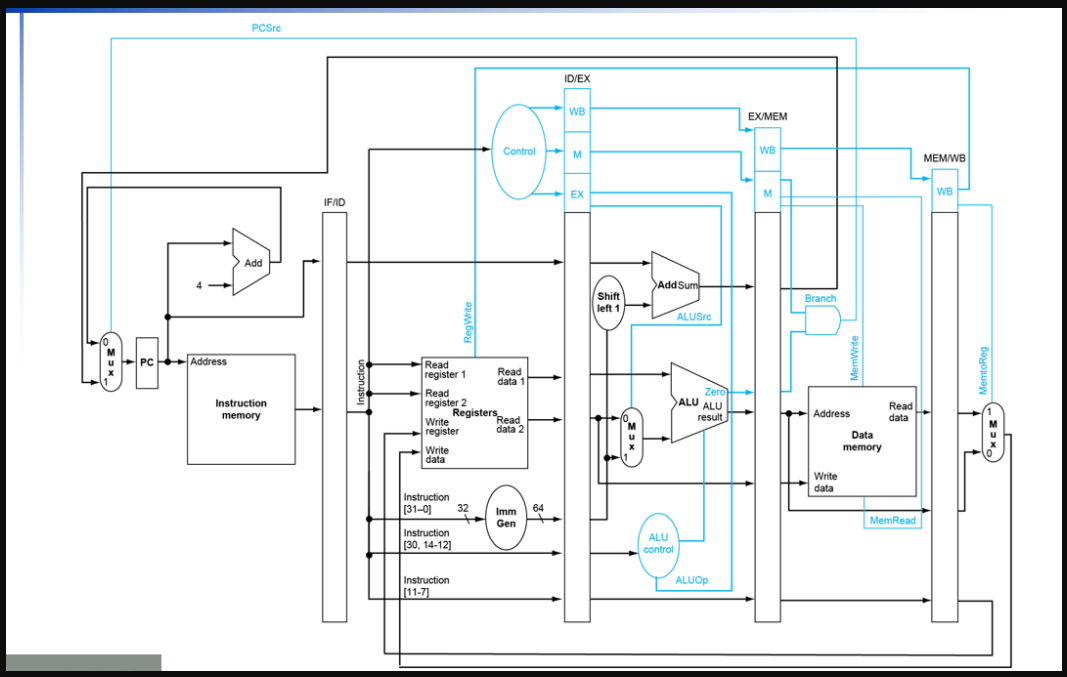


8hrs (non-pipelined) vs 3.5hrs (Pipelined)



800ps (non-pipelined) vs 200ps (Pipelined)

Essentially the datapath is divided into multiple stages and each stage has atleast one instruction running in each stage at once. It increases throughput but has no effect on latency.



It has 5 stages

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

These blocks are similar to that of the sequential blocks

The main additions made in Pipelining compared to Sequential -

- Registers between each stage and the next (To make it easy for forwarding and only required data can be stored)
- Writeback address is taken all the way to the end of the pipeline and given back.
- Hazard Detection (structural, data, and control)

Use of Register Files:

Registers **hold the intermediate results** of one stage before passing them to the next.

Without registers, the next stage might receive incomplete or incorrect data due to timing mismatches.

In a pipelined processor, each stage works on a different instruction **simultaneously**. Registers act as a **boundary**, ensuring that each stage completes its task before passing the data forward.

Without registers, signals would have to travel across multiple stages without any storage, leading to **signal degradation and delays**.

By isolating each stage with registers, the pipeline can work on **multiple instructions at once**.

This prevents **stalling**, where a delay in one stage would otherwise slow down the entire pipeline.

A major **drawback** of pipelining -

Pipelining introduces various hazards in the datapath.

HAZARDS

Situations that prevent starting the next instruction in the next cycle.

Structural Hazards - A required resource is busy. Could be due to 2 stages trying to access the same Memory at the same time.

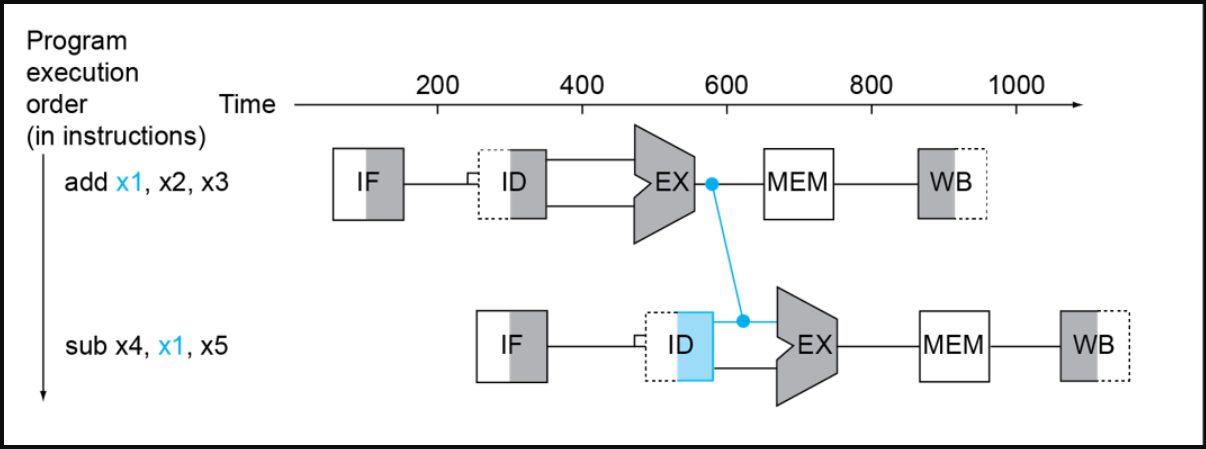
Data Hazards - Need to wait for previous instruction to complete its data read/write. Can be solved by Forwarding in most cases but Stalling is required for some.

Control Hazards - Deciding on control action depends on previous instruction. Whether Branch is taken or not is computed too late (Execute Stage) for the next instruction to start. Either extra hardware must be used in earlier stages or Branch Prediction must be implemented as a measure to take care of them.

HAZARDS HANDLING

FORWARDING (Handles Data Hazards)

Use result when it is computed without waiting for it to be stored in a register.It requires extra connections in the datapath.



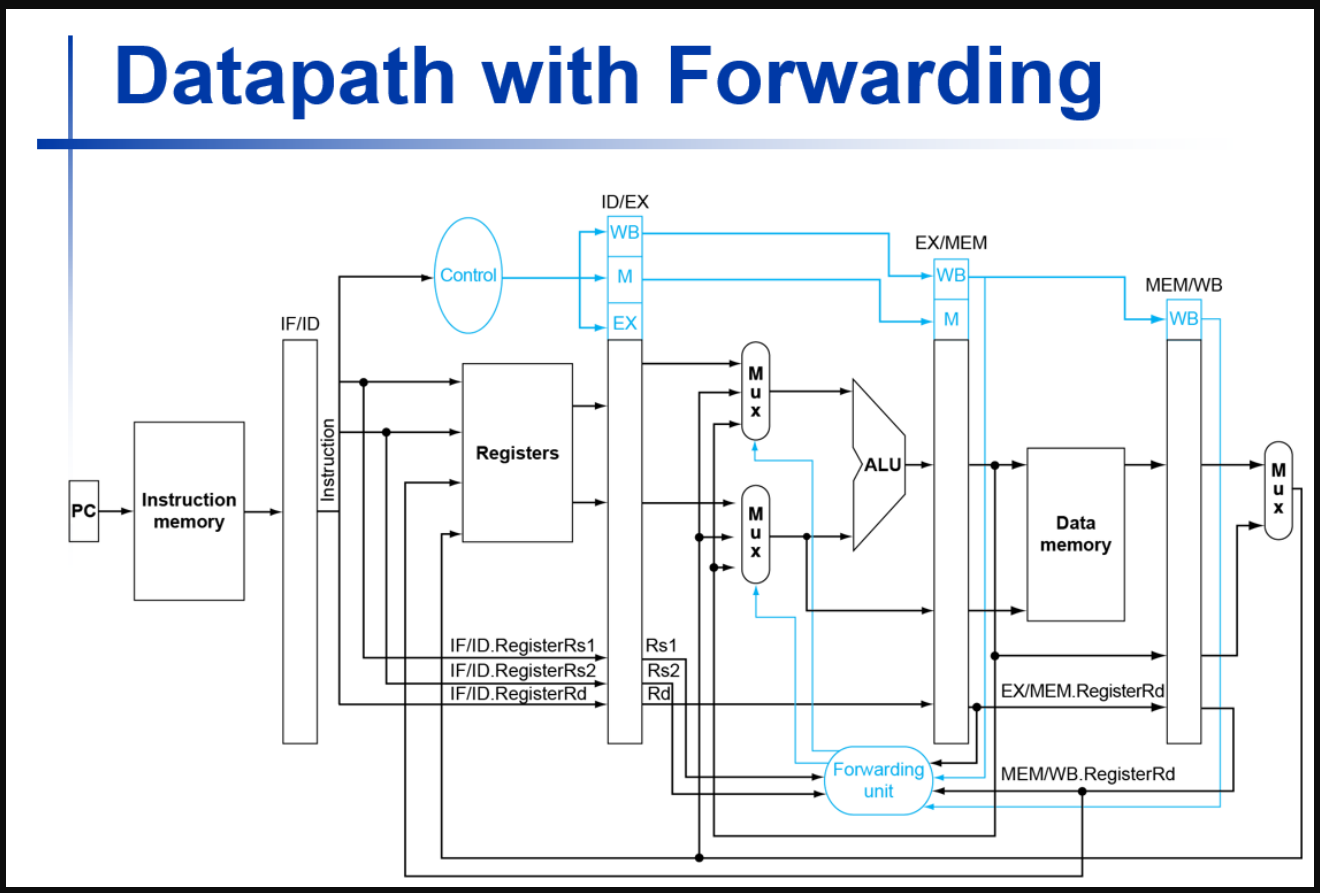
The cases when the data required for the next instruction is available after or before the execute stage can be resolved by forwarding.

Detecting when to forward –

If the previous instruction's output register is the same as an input register as the next one.

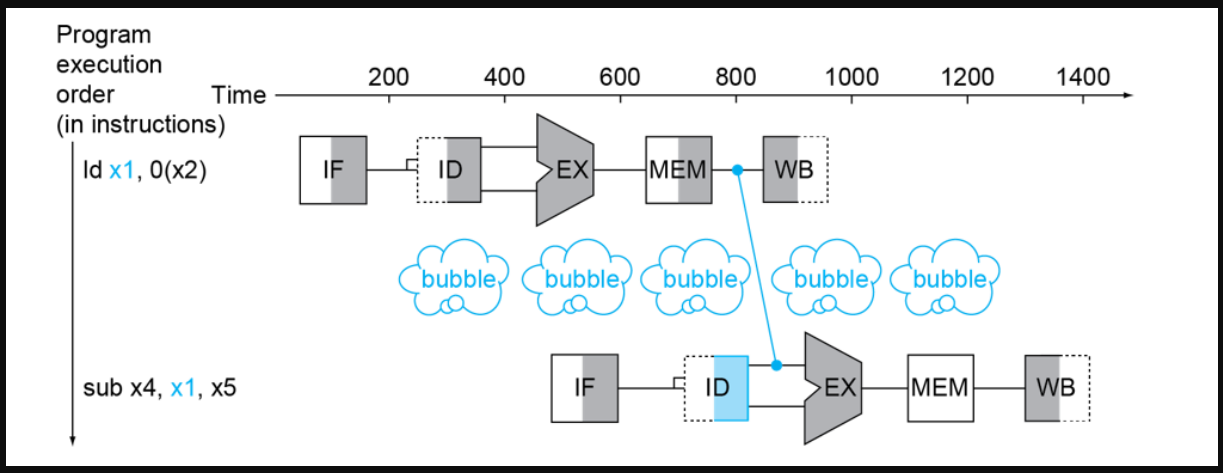
The data could be forwarded from the previous instruction's EX/MEM to ID/EX of next or MEM/WB to ID/EX.

The **Forward Block** uses this data to detect when to stall and when to bypass.



STALLING (Load Use Hazards)

In the case when a load instruction is followed by a R type(lets say) instruction requiring the same register,the data is available in the register only after the data memory stage. So the next instruction cannot start in the next clock cycle and there has to be a stall for one clock cycle.



BRANCH PREDICTION (Handles Control Hazards)

Static branch prediction

- Based on typical branch behavior
  - Predicts backward branches taken

    Predicts forward branches not taken

Dynamic branch prediction

- Hardware measures actual branch behavior
- Assume future behavior will continue the trend

When wrong, stall while re-fetching, and update history

HOW CODES WORK-

Decode_Stage.v Module

The decode_cycle extracts instruction fields, determines the required control signals, and fetches operand values from the register file. The decoded information is then stored in pipeline registers for use in the **execution stage**.

Functionality

1. **Instruction Decoding:** Extracts key instruction fields (opcode, funct3, funct7, register addresses).
2. **Control Signal Generation:** Determines the behavior of the ALU, memory, and register file via the control unit.
3. **Register Read:** Reads operand values from the register file based on instruction fields.

4. **Sign Extension:** Extends immediate values to 64-bit for ALU operations.
5. **Pipeline Registers:** Stores decoded control signals, operands, and immediate values for the next stage.

Module Components

1. Inputs

Signal	Size	Description
clk	1-bit	Clock signal
rst	1-bit	Active-low reset
InstrD	32-bit	Instruction from the instruction fetch stage
PCD	64-bit	Current program counter (PC)
PCPlus4D	64-bit	PC + 4 (next instruction address)
RegWriteW	1-bit	Register write enable from the write-back stage
RDW	5-bit	Destination register from write-back stage
ResultW	64-bit	Data to write back to register file

2. Outputs

Signal	Size	Description
RegWriteE	1-bit	Enables register write in the execution stage
ALUSrcE	1-bit	Selects between register and immediate for ALU
MemWriteE	1-bit	Enables memory write
ResultSrcE	1-bit	Determines if ALU result or memory data is written back
BranchE	1-bit	Indicates if instruction is a branch
ALUControlE	3-bit	ALU operation control signal
RD1_E, RD2_E	64-bit	Register values for ALU operation
Imm_Ext_E	64-bit	Sign-extended immediate value
RD_E	5-bit	Destination register for execution stage
PCE, PCPlus4E	64-bit	Pipeline PC values
RS1_E, RS2_E	5-bit	Register source indices

Instruction_Memory.v Module

This module represents the **Instruction Memory** in a processor. It is responsible for storing and providing instruction data based on the **Program Counter (PC) address**.

Functionality Overview

1. **Stores instructions in a memory array** (mem).
2. **Fetches the instruction** at the given memory address (A).
3. **Uses a reset signal (rst)** to determine whether to output valid instructions or reset the output.
4. **Loads memory contents from a file** (memfile.hex) during initialization.

It stores instructions in a **1024 x 32-bit** memory array.

Main_Decoder.v Module

Main Decoder module is a crucial part of a processor's **control unit**. It takes the **opcode (Op)** from an instruction and generates several control signals that direct the processor's operation in different instruction execution stages.

Functionality

Reads the 7-bit opcode (Op) from the instruction.
Generates control signals that dictate:

- Register Write (RegWrite)
- Immediate Source (ImmSrc)
- ALU Source (ALUSrc)
- Memory Write (MemWrite)
- Result Source (ResultSrc)
- Branch (Branch)
- ALU Operation (ALUOp)

Interprets the opcode (Op) to generate control signals.
Enables register write for Load, R-type, and I-type instructions.
Determines the immediate type for Store and Branch instructions.
Selects ALU operand source (register vs. immediate).
Controls memory writes for Store instructions.
Indicates branch instructions for conditional jumps.
Determines ALU operation mode for different instruction types.

Execute_cycle.v Module

This module implements the **Execute Cycle** in a pipelined processor, which is responsible for:

1. **Selecting ALU inputs** using multiplexers.
2. **Performing the ALU operation** to compute results.
3. **Computing the branch target address.**
4. **Determining if a branch should be taken (PCSrcE).**
5. **Passing values to the next pipeline stage (Memory stage).**

Functionality

- Takes **decoded instruction signals** from the **Decode Cycle** stage.
- Handles **ALU computation** using appropriate inputs.
- Supports **forwarding** to resolve data hazards.
- Computes the **branch target address** for conditional branching.
- Passes computed values to the next pipeline stage (Memory).

Handles forwarding to resolve data hazards.
Selects ALU inputs based on control signals.
Performs ALU operations and produces results.
Computes branch target addresses for control flow changes.
Passes values to the Memory stage using pipeline registers.

Fetch_cycle.v Module

The **Fetch Cycle** is the first stage of the pipelined processor. It is responsible for:

1. **Fetching the instruction** from memory.
2. **Updating the program counter (PC).**
3. **Passing the instruction and PC values to the Decode stage.**
4. **Handling branch instructions and updating the PC accordingly.**

Functionality

- Fetches an **instruction** from **instruction memory** using the **PC value**.
- Computes the **next PC value (PC+4)** for sequential execution.
- Supports **branching** by selecting either PC+4 or the branch target (PCTargetE).
- Stores the fetched instruction and PC values in **pipeline registers** for the **Decode stage**.
- **Fetches instruction** from memory using PC.
  Computes the next instruction address (PC + 4).
  Handles branching using a multiplexer.
  Updates PC every cycle.
  Passes instruction and PC values to the Decode stage.
- This module ensures smooth instruction fetching for the **pipelined processor**.

Data_Memory Module

Implements a **64-bit word-addressable memory** for a processor. It is typically used in the **memory stage** of a pipelined CPU to store and retrieve data. This memory allows **reading and writing of 64-bit data** at specific addresses.

Functionality

1. **Stores 64-bit Words**
   - The memory consists of **256 locations** (each 64-bit wide).
   - Uses **word addressing** (not byte addressing).
2. **Supports Read and Write Operations**
   - **Write (WE = 1)** → Stores a 64-bit value at the given address.
   - **Read (WE = 0)** → Reads the 64-bit value from memory.

3. **Word Addressing (A[9:2])**
    - The address A is **64 bits**, but only bits [9:2] are used.
    - This ensures data is stored at **64-bit word-aligned locations** (ignoring byte offsets).
4. **Memory Initialization**
    - At the start, all memory locations are set to **zero**.
5. **Reset (rst Signal)**
    - When rst = 0, the output RD is forced to 0.
    - The memory contents remain unchanged.

Working

Read Operation (Load)

- The processor requests data by setting an **address (A)**.
- If rst = 1, the module **reads** the **64-bit value** stored at mem[A[9:2]] and outputs it as RD.

Write Operation (Store)

- If WE = 1, the **64-bit value (WD)** is written into the memory at address A[9:2].
- If WE = 0, memory remains unchanged.

Reset (rst)

- If rst = 0, the **output (RD) is forced to zero**.
- However, the stored memory values **are not cleared** (except during initial setup).

Sign Extension Module Overview

The **Sign_Extend** module in **Verilog** extends a 32-bit immediate value (In) to a 64-bit value (Imm_Ext). It is commonly used in **RISC-V processors** to handle **immediate values** in different instruction formats.

Functionality

- Converts **32-bit immediate values** into **64-bit** values by **sign-extending** them.
- The **sign bit** (In[31]) is **replicated** across the upper bits to preserve the sign.
- Supports different **immediate types** (I-Type and S-Type) based on the **ImmSrc** selector.

Immediate Types Supported

Instruction Type	Immediate Extraction	Bits Used
I-Type (e.g., ADDI, LW)	{ In[31:20] }	12-bit
S-Type (e.g., SW)	{ In[31:25], In[11:7] }	12-bit

Working

I-Type (Load/Immediate Instructions)

For **I-Type** instructions like ADDI or LW, the immediate is stored in **bits [31:20]**.
Example (ADDI X1, X2, -5)

Instruction: 0xFFF20213 (Binary: 1111 1111 1111 0010 0000 0010 0001 0011)

Immediate: 0xFFF (-5 in decimal)

Sign Extend: 0xFFFFFFFFFFFFFFFB (-5 as 64-bit)

S-Type (Store Instructions)
For **S-Type** instructions like SW, the immediate is **split** into two parts:

- **Upper bits:** In[31:25]
- **Lower bits:** In[11:7] These are combined to form the **12-bit immediate**.

Example (SW X3, -8(X4))
Instruction: 0xFE342023

Immediate: { In[31:25], In[11:7] } = 0xFF8 (-8 in decimal)

Sign Extend: 0xFFFFFFFFFFFFFFF8 (-8 as 64-bit)

Default Case
If an unsupported ImmSrc value is provided, the output defaults to 64'h0000000000000000.

Writeback Cycle Module Overview

The **writeback_cycle** module is responsible for selecting the final result to be written back to the register file in a **RISC-V processor pipeline**.

Functionality

- Determines the value to be written into the **register file**.
- Uses a **multiplexer (Mux)** to choose between:
    1. **ALU result** (for arithmetic/logic operations).
    2. **Memory read data** (for load instructions).
- The selection is based on the **ResultSrcW** signal.

Input and Output Description

Signal Name	Direction	Size	Description
clk	**Input**	1-bit	Clock signal
rst	**Input**	1-bit	Reset signal
ResultSrcW	**Input**	1-bit	Selects between ALU result and memory read data
PCPlus4W	**Input**	64-bit	(Unused in this module) Address of the next instruction
ALU_ResultW	**Input**	64-bit	Result from ALU operations (e.g., ADD, SUB, AND, OR)
ReadDataW	**Input**	64-bit	Data read from memory (for load instructions)
ResultW	**Output**	64-bit	Final value written back to the register file

MUX Functionality

- The **Mux** module takes two inputs:
  - **ALU result** (ALU_ResultW) → Used for ALU instructions (e.g., ADD, SUB).
  - **Memory read data** (ReadDataW) → Used for Load instructions (e.g., LW).
- The **ResultSrcW** signal controls the Mux:
  - 0 → Selects ALU_ResultW.
  - 1 → Selects ReadDataW.
  -

Hazard Unit Module Overview

The **hazard_unit** module is responsible for **data forwarding** in a pipelined processor. It helps to **resolve data hazards** by selecting the correct values for operands when there are dependencies between pipeline stages.

Functionality

- Detects **data hazards** in the execution (EX) stage.
- Determines whether values need to be forwarded from:
  - The **memory (MEM) stage** (RegWriteM).
  - The **writeback (WB) stage** (RegWriteW).
- Generates control signals (ForwardAE, ForwardBE) to **bypass stalled data** and prevent incorrect computations.

Input and Output Description

Signal Name	Direction	Size	Description
rst	**Input**	1-bit	Reset signal
RegWriteM	**Input**	1-bit	Register write enable in the memory (MEM) stage
RegWriteW	**Input**	1-bit	Register write enable in the writeback (WB) stage
RD_M	**Input**	5-bit	Destination register in the memory (MEM) stage
RD_W	**Input**	5-bit	Destination register in the writeback (WB) stage
Rs1_E	**Input**	5-bit	Source register 1 in the execution (EX) stage
Rs2_E	**Input**	5-bit	Source register 2 in the execution (EX) stage
ForwardAE	**Output**	2-bit	Forwarding control signal for Rs1_E
ForwardBE	**Output**	2-bit	Forwarding control signal for Rs2_E

Forwarding Logic

The **hazard detection logic** prevents unnecessary stalls by forwarding data from later pipeline stages:

- **ForwardAE (for Rs1_E)**
  - **10** → Forward from MEM stage (RD_M == Rs1_E).
  - **01** → Forward from WB stage (RD_W == Rs1_E).
  - **00** → No forwarding (use value from register file).
- **ForwardBE (for Rs2_E)**
  - **10** → Forward from MEM stage (RD_M == Rs2_E).
  - **01** → Forward from WB stage (RD_W == Rs2_E).
  - **00** → No forwarding (use value from register file).
- **Reset condition (rst == 0)**
  - Both ForwardAE and ForwardBE are set to 00 (no forwarding).

Register File Module

Implements a 32-register file used in a processor. It allows reading and writing registers as part of instruction execution.

Functionality

Reads values from two registers (RD1, RD2).

Writes a value (WD3) to a register (A3) if write enable (WE3) is set.

Ensures register 0 remains zero (as per RISC-V convention).

Supports reset (rst) to initialize registers.

Input and Output Description

Signal	Name	Size	Description
clk	Input	1-bit	Clock signal
rst	Input	1-bit	Reset signal
WE3	Input	1-bit	Write enable signal
A1	Input	5-bit	Address of register to read (RD1)
A2	Input	5-bit	Address of register to read (RD2)
A3	Input	5-bit	Address of register to write (WD3)
WD3	Input	32-bit	Data to write into register A3
RD1	Output	32-bit	Read data from register A1
RD2	Output	32-bit	Read data from register A2

BEHAVIOR

Reading Registers

RD1 gets the value from Register[A1].

RD2 gets the value from Register[A2].

If reset (rst) is active, both RD1 and RD2 output 0.

Writing to a Register

Data WD3 is written to Register[A3] on the rising edge of clk if:

WE3 == 1 (write enable is active).

A3 != 0 (to prevent modifying register x0, which should always be 0).

Initialization

The register file contains 32 registers (Register [0] to Register [31]).

Register [0] is hardwired to 0.

Purpose	Implements a 32-register file for CPU execution

Read Mechanism	Two simultaneous register reads (RD1, RD2)

Write Mechanism	Writes on rising edge if WE3 == 1 and A3 ≠ 0

Reset Behaviour	Outputs 0 when rst == 0
Register x0	Hardwired to 0 (cannot be modified)

Memory Cycle Module Overview

The memory_cycle module represents the Memory Stage in a pipelined processor. It interacts with Data Memory to read/write data and stores results for the next pipeline stage (Writeback).

Functionality

- **Reads/Writes data to memory using Data Memory (Data_Memory) module.**
- **Passes computed values forward to the Writeback Stage.**

- Stores results in pipeline registers (RegWriteM_r, ResultSrcM_r, etc.).
- Supports reset (rst) to clear stored values.

Input and Output Description

Signal Name	Direction	Size	Description
clk	Input	1-bit	Clock signal
rst	Input	1-bit	Reset signal
RegWriteM	Input	1-bit	Write enable for registers (Memory stage)
MemWriteM	Input	1-bit	Write enable for memory
ResultSrcM	Input	1-bit	Selects memory data or ALU result for writeback
RD_M	Input	5-bit	Destination register address
PCPlus4M	Input	64-bit	PC + 4 value (for JAL instruction)
WriteDataM	Input	64-bit	Data to write into memory
ALU_ResultM	Input	64-bit	ALU result (memory address for load/store)
RegWriteW	Output	1-bit	Register write enable (Writeback stage)
ResultSrcW	Output	1-bit	Selects memory data or ALU result in Writeback
RD_W	Output	5-bit	Destination register address for Writeback
PCPlus4W	Output	64-bit	PC + 4 value for Writeback
ALU_ResultW	Output	64-bit	ALU result forwarded to Writeback
ReadDataW	Output	64-bit	Data read from memory

Behavior

Memory Access
- The Data_Memory module reads/writes data.
    - Load (LW): Reads memory into ReadDataM.
    - Store (SW): Writes WriteDataM into memory.

Pipeline Register Storage
- Stores values from Memory Stage for Writeback.
-
- Transfers:
    - RegWriteM → RegWriteW

- **ResultSrcM → ResultSrcW**
- **RD_M → RD_W**
- **PCPlus4M → PCPlus4W**
- **ALU_ResultM → ALU_ResultW**
- **ReadDataM → ReadDataW**

Reset Behavior

- **If rst == 0, clears all stored values.**

Terminal O/P for ALU Decode Stage

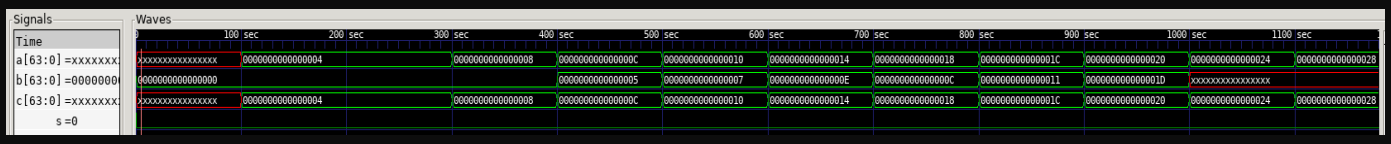
```
Test 1: ALUOp=00 | ALUControl=000 (Expected: 000)
Test 2: ALUOp=01 | ALUControl=001 (Expected: 001)
Test 3a: ALUOp=10, funct3=000, R-SUB | ALUControl=001 (Expected: 001)
Test 3b: ALUOp=10, funct3=000, R-ADD | ALUControl=000 (Expected: 000)
Test 3c: ALUOp=10, funct3=000, I-ADD | ALUControl=000 (Expected: 000)
Test 4: ALUOp=10, funct3=010 | ALUControl=101 (Expected: 101)
Test 5: ALUOp=10, funct3=110 | ALUControl=011 (Expected: 011)
Test 6: ALUOp=10, funct3=111 | ALUControl=010 (Expected: 010)
Test 7: ALUOp=10, funct3=001 | ALUControl=000 (Expected: 000)
Test 8: ALUOp=11 | ALUControl=000 (Expected: 000)
```

Terminal O/P for Hazard detection Unit

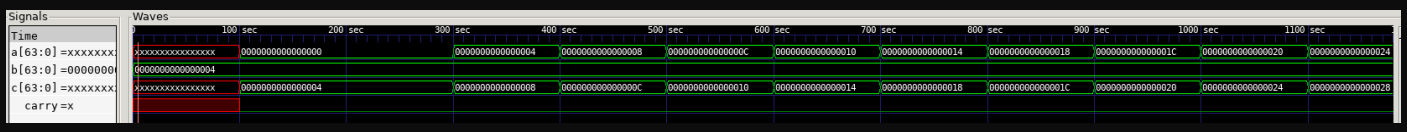
```
VCD info: dumpfile hazard_unit_tb.vcd opened for output.

Test Case 1: Reset
ForwardAE = 00, ForwardBE = 00

Test Case 2: No Hazard
ForwardAE = 00, ForwardBE = 00

Test Case 3: Forward from M to Rs1_E
ForwardAE = 10, ForwardBE = 00

Test Case 4: Forward from W to Rs1_E
ForwardAE = 01, ForwardBE = 00

Test Case 5: Priority to M over W
ForwardAE = 10, ForwardBE = 00

Test Case 6: Forward from M to Rs2_E
ForwardAE = 10, ForwardBE = 10

Test Case 7: Forward from W to Rs2_E
ForwardAE = 01, ForwardBE = 01

Test Case 8: RD_M is x0
ForwardAE = 00, ForwardBE = 01

Test Case 9: RD_W is x0
ForwardAE = 00, ForwardBE = 00

Test Case 10: Cross forwarding
ForwardAE = 01, ForwardBE = 10

All test cases completed.
```

GTK WAVE PLOTS

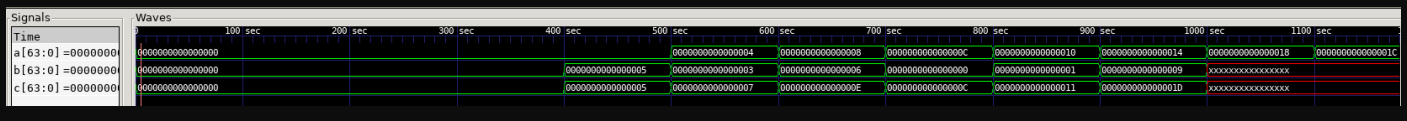
PC Mux:



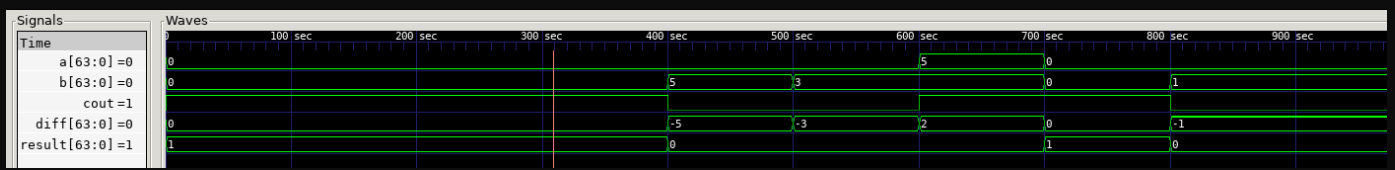
PC Adder:



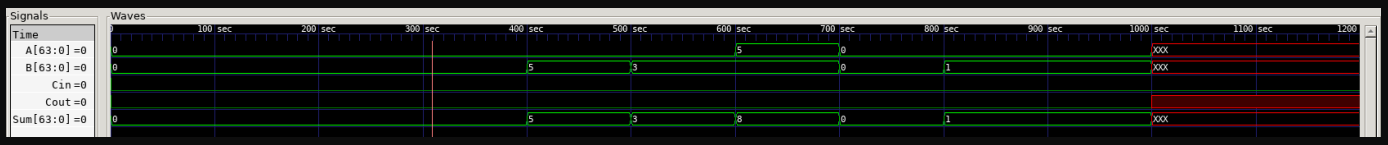
Branch Adder:



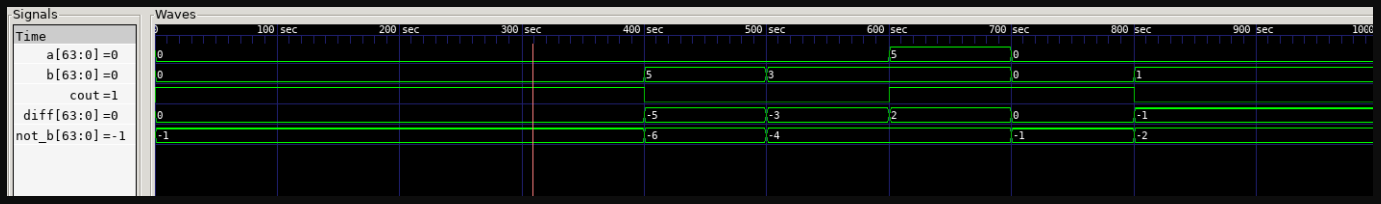
BEQ:



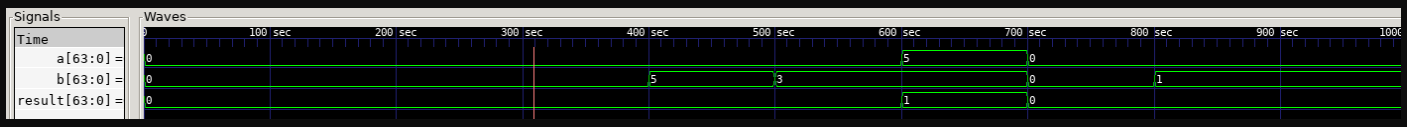
ADD:



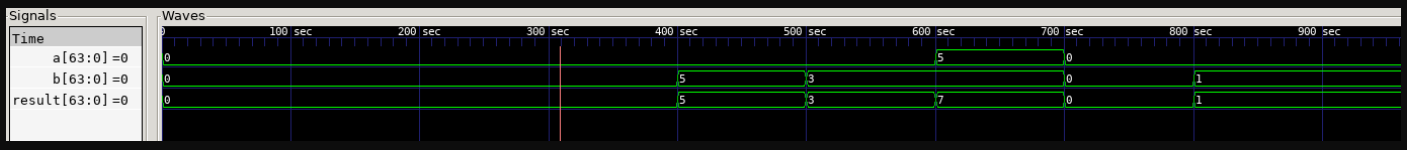
SUB:



AND:

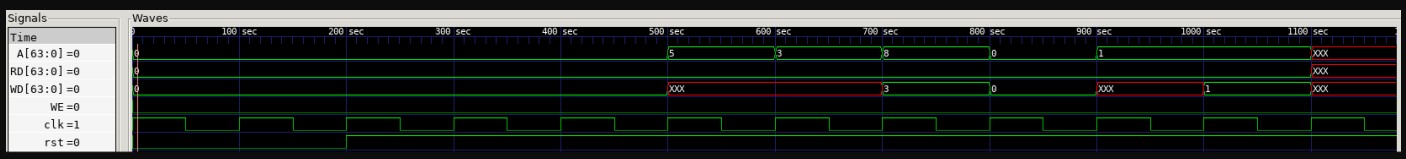


OR:

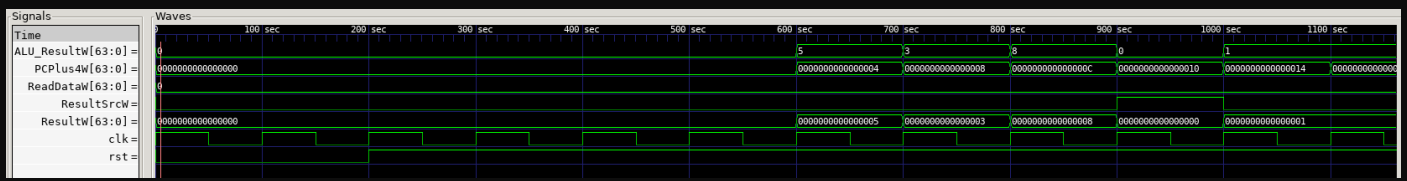


Forwarding:

Memory:



Writeback:



HEX CODE USED FOR THESE –

@00000000

00500293 # addi x5, x0, 5 | x5 = 5

00300313 # addi x6, x0, 3 | x6 = 3

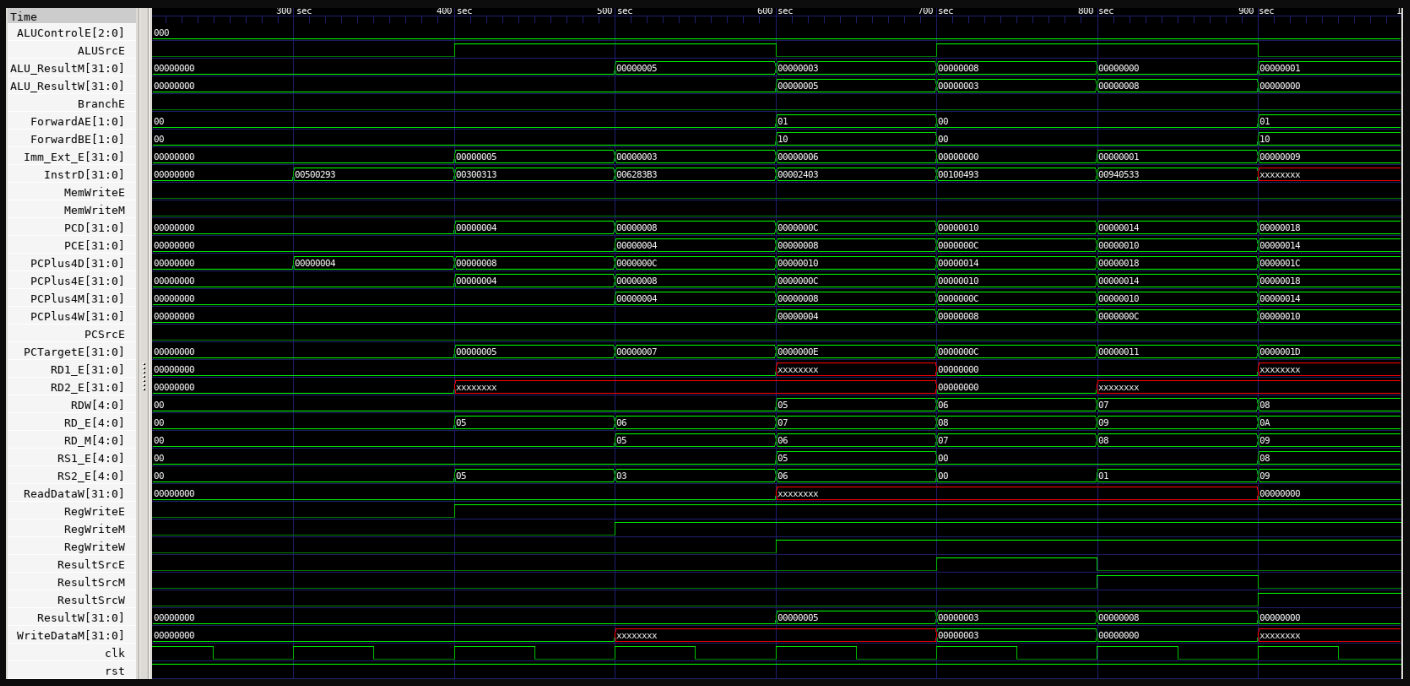
006283B3 # add x7, x5, x6 | x7 = x5 + x6 = 8

00003403 # ld x8, 0(x0) | x8 = M[0] (load doubleword from memory at address 0)

00100493 # addi x9, x0, 1 | x9 = 1

00940533 # add x10, x8, x9 | x10 = x8 + x9

00843023 # sd x10, 0(x7) | M[x7 + 0] = x10 (store doubleword into memory at address 8)



Instructions also tested –

00500293 # addi x5, x0, 5 | x5 = 5

00300313 # addi x6, x0, 3 | x6 = 3

006284B3 # add x9, x5, x6 | x9 = x5 + x6 = 8

0004A683 # ld x13, 0(x9) | x13 = M[x9 + 0] (load from memory at address 8)

406282B3 # sub x10, x5, x6 | x10 = x5 - x6 = 2

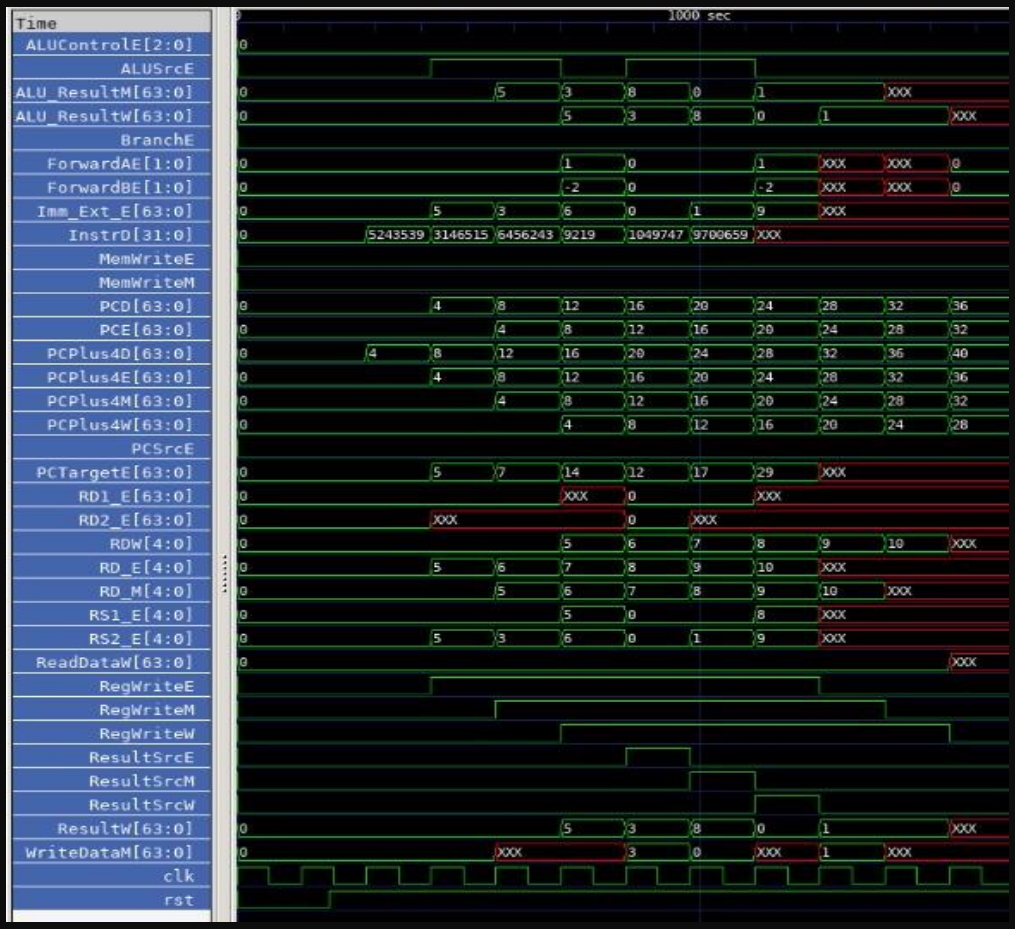
00C4A023 # sd x10, 0(x9) | M[x9 + 0] = x10 (store x10 into memory at address 8)

00529263 # beq x5, x5, target | If x5 == x5, jump to target (always branches)

00100793 # addi x15, x0, 1 | x15 = 1 (if beq not taken)

0096F5B3 # and x11, x13, x9 | x11 = x13 & x9

0096E633 # or x12, x13, x9 | x12 = x13 | x9



00500293 # addi x5, x0, 5 | x5 = 5

00300313 # addi x6, x0, 3 | x6 = 3

006284B3 # add x9, x5, x6 | x9 = x5 + x6 = 8

0004A683 # ld x13, 0(x9) | x13 = M[x9 + 0] (load from memory at address 8)

406282B3 # sub x10, x5, x6 | x10 = x5 - x6 = 2

00C4A023 # sd x10, 0(x9) | M[x9 + 0] = x10 (store x10 into memory at address 8)

00530463 # beq x5, x6, target | If x5 == x6, branch to target (branch not taken)

00100793 # addi x15, x0, 1 | x15 = 1 (if branch not taken)

0096F5B3 # and x11, x13, x9 | x11 = x13 & x9 (forwarding x13 from ld)

0096E633 # or x12, x13, x9 | x12 = x13 | x9 (forwarding x13 from ld)

Challenges Faced

Variables getting disconnected while Hazard Handling

Dealing with Branch Instructions

Handling Overflow in ALU

Clock Synchronization

Contributions

Aikya: Decode, Writeback, Pipeline synchronisation, GTKWave Analysis for Pipeline

Indira: ALU, Execute stage, sequential wrapper module, Pipeline memory handling, GTKWave Analysis for Sequential, Report

Pragnya: Fetch, Memory, Control block, Sequential Wrapper module, Pipeline synchronisation, Report

(We did the wrapper modules and hazard handling together since we were facing a lot of issues.)