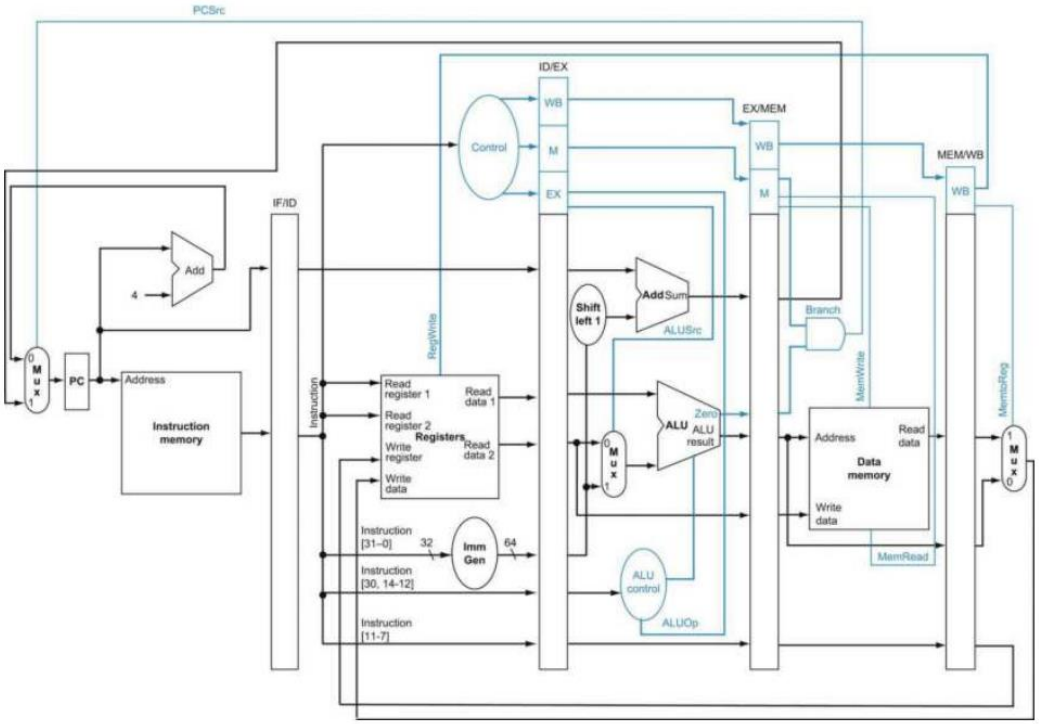
IPA Project (Spring 2025)

Processor Architecture Design based on RISC-V ISA



Team 14

Bhuvi Bhuvi-2023112015 bhuvi.bhuvi@research.iiit.ac.in

Vidvathama-2024122002 vidvathama.babu@research.iiit.ac.in

Dhruv Bansal-2023102048 dhruv.bansal@students.iiit.ac.in

1. Introduction

RISC-V is a modular and extensible instruction set architecture (ISA) that has gained widespread adoption due to its open-source nature and design flexibility. The processor can be implemented using the following approaches:

- **Sequential Processing**
- **Pipelined Processing**

Sequential Processing

- **Execution Method:** Instructions are executed sequentially, one at a time.
- **Speed:** Slower execution since only a single instruction is processed per cycle.
- **Throughput:** Low throughput as each instruction must complete before the next begins.
- **Resource Utilization:** Hardware resources remain idle at times, leading to lower efficiency.
- **Latency:** Higher latency when executing a sequence of instructions.
- **Complexity:** Simplified control logic, making it easier to design and implement.
- **Use Case:** Suitable for basic applications that do not require high-speed execution.

Pipelined Processing

- **Execution Method:** Multiple instructions are processed concurrently across different pipeline stages.
- **Speed:** Faster execution by overlapping the processing of instructions.
- **Throughput:** Higher throughput as new instructions enter the pipeline before previous ones finish.
- **Resource Utilization:** More efficient use of hardware by keeping all pipeline stages active.

- **Latency:** Lower latency per instruction once the pipeline is fully loaded.
- **Complexity:** More complex design due to the need for hazard management and stage synchronization.
- **Use Case:** Well-suited for high-performance computing and modern processors requiring efficient execution.

This project follows a systematic approach, starting with the design of a sequential execution processor and later transitioning to a pipelined architecture. The design is validated through testing and verification using simulation tools such as ModelSim or GTKWave. The processor architecture is implemented using **Verilog**.

This report outlines the design methodology, implementation challenges, verification strategies, and project results. The following sections provide a detailed discussion of each phase of development.

2.Sequential Processor Design Overview

Before implementing pipelining, the processor is built as a sequential architecture where instructions are fetched, decoded, executed, and written back one at a time. This design is simpler to verify, and it forms the foundation on which the pipelined version is built.

- **Modules Developed:**
    - **Instruction Fetch (IF):** Retrieves the instruction from memory using the Program Counter (PC).
    - **Instruction Decode (ID):** Decodes the fetched instruction, reads the register file, and extracts immediate values.
    - **Execution (EX):** Performs arithmetic and logical operations using the ALU.
    - **Memory Access (MEM):** Handles load and store operations by accessing the data memory.
    - **Write Back (WB):** Writes the computation results back to the register file.

This sequential approach allows us to test each module independently with module-specific test inputs, ensuring that individual functionalities work as intended.

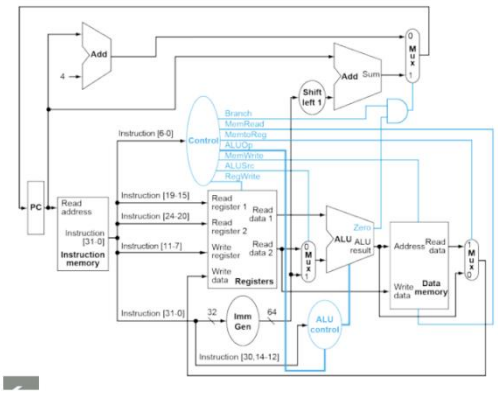


Fig. 1: Sequential Datapath with controls

4.Pipelined Processor Architecture

The pipelined processor is designed with a classic 5-stage pipeline. This structure significantly improves throughput by overlapping the execution of multiple instructions. However, this requires careful handling of hazards to maintain correct program execution.

5-Stage Pipeline Overview

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

Each stage is implemented as a separate module, and pipeline registers are inserted between the stages to store intermediate values and control signals.

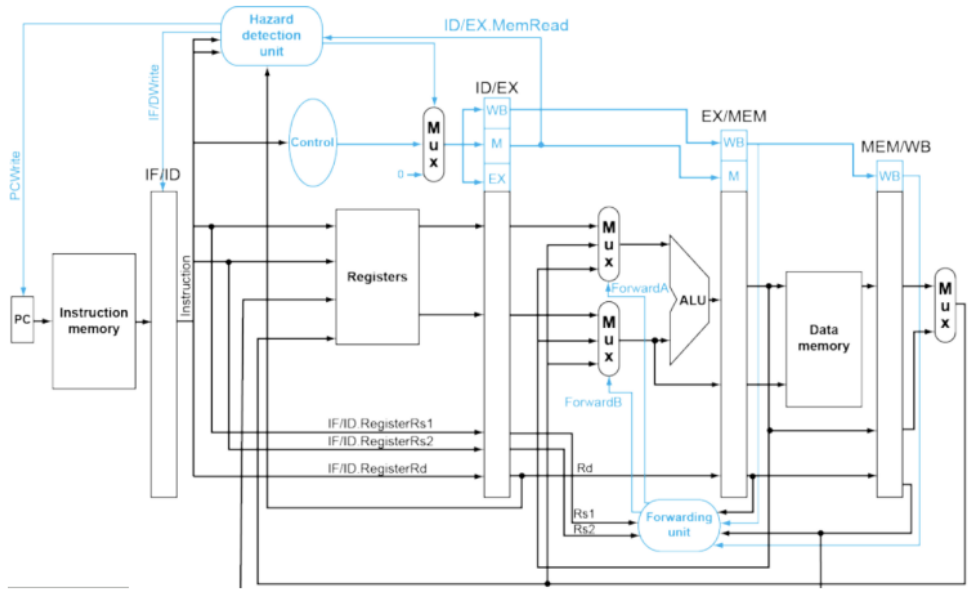


Fig. 2: Pipelined Datapath with Hazard Detection

5.Detailed Explanation of Each Pipeline Stage

The processor design must meet the following specifications:

• A fundamental implementation of the processor architecture using a sequential design.

• An advanced implementation featuring a 5-stage pipelined processor architecture, incorporating mechanisms to handle pipeline hazards effectively. Both implementations must support the execution of the following instructions from the RISC-V ISA: add, sub, and, or, ld, sd, and beq.

A. Architecture Overview The sequential RISC-V processor follows a single-cycle execution model, where each instruction is fetched, decoded, executed, and written back within one clock cycle. This approach simplifies control logic but results in longer cycle times due to the complexity of executing all operations in one step.

The processor is composed of several key components that work together to fetch, decode, execute, and store instructions. These include:

• Arithmetic Logic Unit (ALU): Performs arithmetic and logical operations based on control signals.

• Control Unit: Generates control signals required to execute instructions correctly.

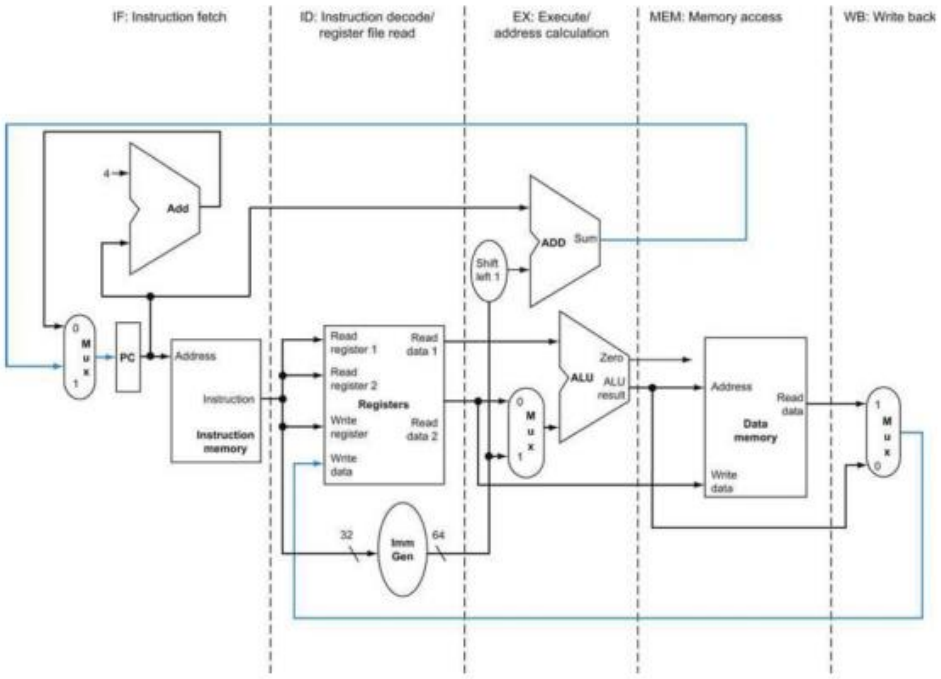
• Register File: Holds a set of registers for temporary data storage.

• Immediate Generator: Extracts and sign-extends immediate values from instruction encoding.

• Instruction Memory: Stores the program instructions.

• Data Memory: Holds data values that are loaded and stored during execution.

• Program Counter (PC): Maintains the address of the next instruction to be executed. The overall architecture can be seen in the block diagram below:



Supported Instruction Set

This processor supports a basic subset of the RISC-V instruction set, covering arithmetic, logical, memory, and control operations. The following instructions are implemented:

• R-type Instructions: add, sub, and, or, xor, sll, srl, slt

• I-type Instructions: addi, andi, ori, xori, slli, srli, slti, lw

• S-type Instructions: sw

• B-type Instructions: beq, bne, blt, bge

The instruction encoding follows the standard RISC-V format, with fields for opcode, register addresses, function codes, and immediates.

General Explanation of Execution Flow

The processor operates in a step-by-step manner, completing each instruction within a single cycle before moving to the next. The execution process follows these stages:

- The Program Counter (PC) retrieves an instruction.
- The instruction is decoded, and the necessary control signals are generated.
- If register operands are needed, they are read from the Register File.
- The Arithmetic Logic Unit (ALU) executes computations, or memory is accessed if required.
- The computed result is stored in the Register File or written to memory.
- The PC is updated to fetch the next instruction.

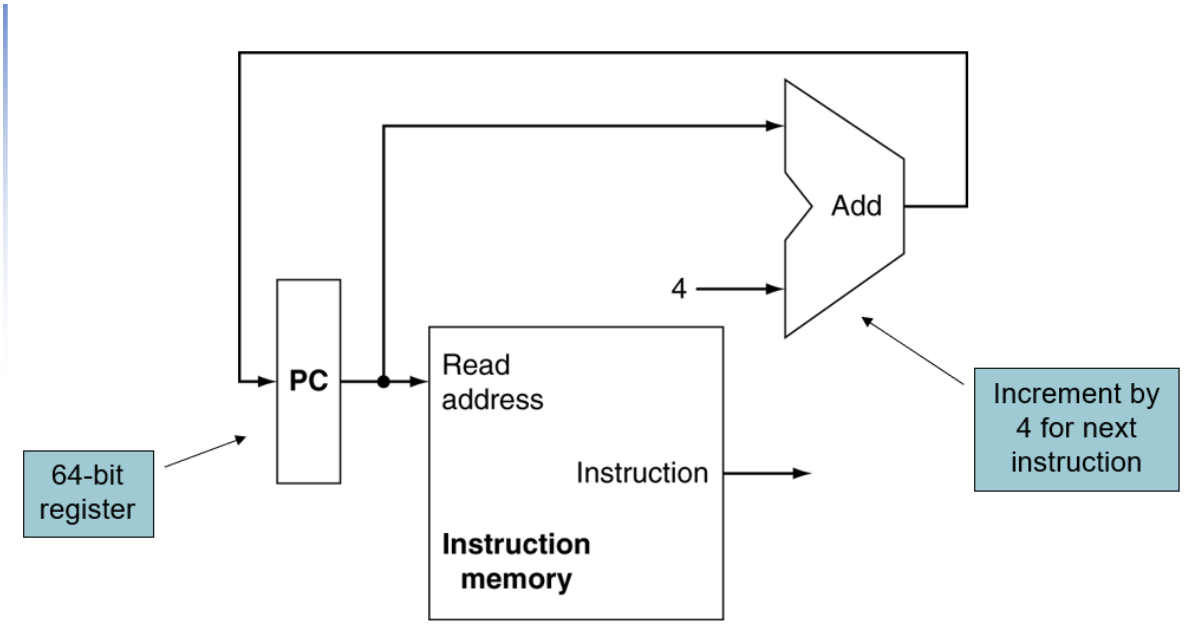
This cycle continues indefinitely until a termination condition, such as an exit instruction, is encountered.

6. Microarchitecture Details

A. Instruction Fetch (IF) Stage

This stage is done by the ifblock.v . The instruction_memory module is a simple read-only memory (ROM) that stores machine code instructions for a processor. Given a 64-bit program counter (PC), it outputs a 32-bit instruction stored at the corresponding memory location. This module is commonly used in processor designs to fetch instructions during execution.

- The module is named instruction_memory.
- It has one **input**, PC (Program Counter), which is a **64-bit address** used to fetch instructions.
- It has one **output**, instruction, which is a **32-bit instruction** retrieved from memory.
- Declares an array memory of **64** elements, each storing a **32-bit instruction**.
- This models a small instruction memory, capable of storing **64 instructions** (assuming each instruction is 4 bytes).
- Memory Initialization:
- The initial block executes only once at the start of the simulation.
- The $readmemb system task loads binary instructions from a file named "instructions.txt" into the memory array.
- This allows pre-loading the instruction memory with machine code before the simulation starts.
- assign instruction = memory[PC[63:0]]; is the **instruction fetch** operation.
- It uses the PC value to index into the memory array.
- It also retrieves the **32-bit instruction** stored at the address given by PC.
- **Function:**
    - Retrieves the instruction from the instruction memory using the current value of the Program Counter (PC).
    - Increments the PC to point to the next instruction.
    - For branch instructions (e.g., beq), the PC update may be modified based on branch decisions (address calculated in later stages).
- **Module Considerations:**
    - A simple memory read interface.
    - Incorporates PC updating logic, which may later include branch target selection and prediction logic for further enhancements.



B.Instruction Decode (ID) Stage

The instruction_decoder module stores in the file instdecoder.v is responsible for decoding a 32-bit instruction in the RISC-V instruction set. It extracts various fields from the instruction, including register addresses, immediate values, opcode, and function codes. This is a crucial part of a processor's instruction execution pipeline.

Inputs and Outputs

- **Inputs:**
  - instruction (32-bit): The instruction to be decoded.
- **Outputs:**
  - read_reg1 (5-bit): Address of the first source register.
  - read_reg2 (5-bit): Address of the second source register (for instructions requiring two operands).
  - write_reg (5-bit): Address of the destination register.
  - imm (32-bit): Immediate value extracted from the instruction (for instructions that use immediate operands).
  - opcode (7-bit): Specifies the instruction type (e.g., R-type, I-type, S-type, etc.).

- funct3 (3-bit): Specifies additional functionality within a given opcode.
- funct7 (7-bit): Used for further classification of some R-type instructions.

Register and Immediate Variables

- rs1 (5-bit): First source register.
- rs2 (5-bit): Second source register.
- rd (5-bit): Destination register.
- parth (5-bit): Temporary register (not necessary for functionality).
- imm_val (32-bit): Stores the immediate value for instructions that require it.

Decoding Based on opcode

The **opcode** determines the instruction format and behavior.

R-type Instructions (Register-Register ALU Operations)

- R-type instructions use two source registers (rs1, rs2) and a destination register (rd).
- Examples: ADD, SUB, AND, OR, XOR, SLL, SRL.

I-type Instructions (Load Instructions Only)

- I-type instructions use one source register (rs1) and a destination register (rd).
- The immediate value (imm_val) is sign-extended from bits [31:20].
- The variable parth is assigned 5'b1, but it is unused and unnecessary.

S-type Instructions (Store Instructions)

S-type instructions (store instructions) use:

- rs1: Base register.
- rs2: Register containing the value to store.
- imm_val: Immediate value, which is formed by concatenating instruction[31:25] and instruction[11:7].

B-type Instructions (Branch Instructions)

- **B-type instructions (branching instructions)** use:
- rs1, rs2: Registers whose values are compared.
- imm_val: Sign-extended branch offset formed by **concatenating** several fields, with 1'b0 appended for proper alignment.

The final values of rs1, rs2, rd, and imm_val are assigned to the corresponding module outputs.

Control Unit

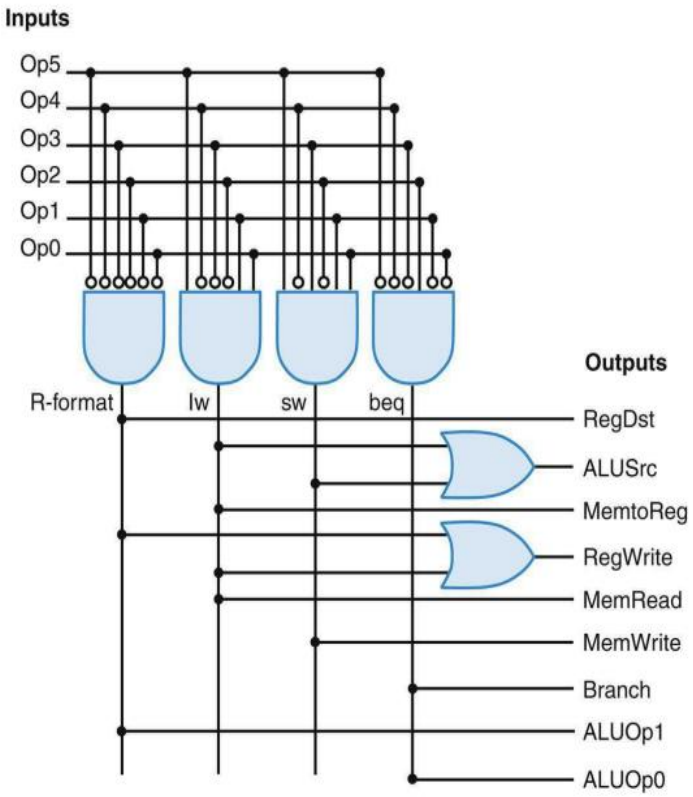


FIGURE C.2.5 **The structured implementation of the control function as described by the truth table in**

Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

The ControlUnit module is a combinational circuit that generates control signals based on the instruction's opcode. It is part of a single-cycle processor design, where it determines how the processor should execute a given instruction.

This module supports the following instruction types:

- R-Type (Register operations)
- LW (Load Word)
- SW (Store Word)
- BEQ (Branch Equal)

The control signals generated by this module are used to control different parts of the datapath, such as the register file, memory, ALU, and multiplexer selection.

Inputs and Outputs

- **Inputs:**
  - Op (6-bit): The **opcode** field from the instruction. It determines the instruction type.
- **Outputs:**
  - RegWrite (1-bit): Enables writing to the register file.
  - MemtoReg (1-bit): Selects data from memory instead of ALU output for register writes.
  - MemRead (1-bit): Enables memory reading.
  - MemWrite (1-bit): Enables memory writing.
  - Branch (1-bit): Indicates a **branch instruction**.
  - ALUSrc (1-bit): Selects **immediate value** instead of register operand for ALU.
  - ALUOp (2-bit): Determines ALU operation mode.

Identifying Instruction Types Using and Gates

Each instruction type is identified by checking the **bit pattern** of Op:

R-Type (000000)

- Op = 000000
- Represents R-Type instructions (e.g., ADD, SUB, AND, OR, etc.).
- Uses register operands and does not use memory.

LW (100011) - Load Word

- Op = 100011
- Loads a word from memory into a register.
- Requires memory read and writes to a register

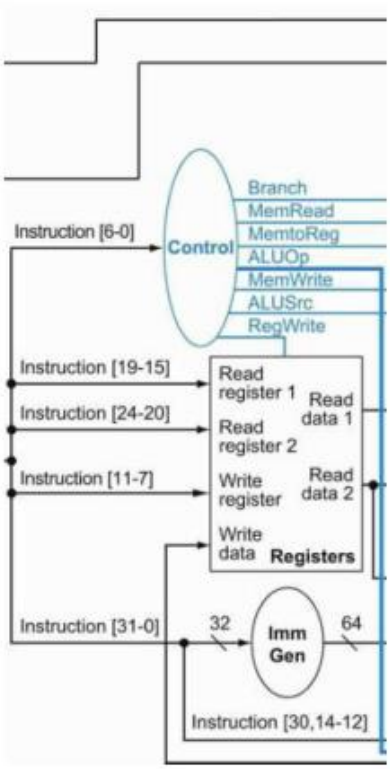
SW (101011) - Store Word

- Op = 101011
- Stores a word from a register into memory.
- Requires memory write but does not write to a register.

BEQ (000100) - Branch if Equal

- Op = 000100
- Compares two registers and branches if they are equal.
- Requires branch control signals.

ID: Instruction Decode Stage



- **Function:**

    - Decodes the fetched instruction to determine the operation, source registers, destination register, and immediate values.
    - Reads the register file to get operands for the ALU.
    - Generates control signals for subsequent stages.
- **Module Considerations:**
    - Immediate extraction and sign-extension for instructions like ld and sd.
    - A Hazard Detection Unit may be introduced in this stage to detect dependencies and initiate stalls if required.

C.Execute (EX) Stage

The ALU must support various operations such as:

- **Arithmetic operations:**
    - Addition (add)
    - Subtraction (sub)
- **Logical operations:**
    - Bitwise AND (and)
    - Bitwise OR (or)

In the RISC-V ISA subset for this project, these operations form the basis for most instruction execution in the EX (Execute) stage. The ALU takes two operands as inputs along with an operation code (opcode) to determine which operation to execute.

The provided Verilog code in the files, implements a 64-bit ALU (Arithmetic Logic Unit) along with supporting modules for arithmetic and bitwise operations.

1. **Full Adder Module**

```
module full_adder(input A, B, cin, output sum, cout);
  assign sum = A ^ B ^ cin;
  assign cout = (A & B) | (B & cin) | (A & cin);
endmodule
```

- Purpose:
  The full_adder module performs the addition of two 1-bit binary numbers along with a carry-in (cin).
- Operation Details:
  - Sum Calculation:
    The sum is calculated using the XOR operation (^), which effectively implements bit addition without carry propagation.
  - Carry-Out Calculation:
    The carry-out (cout) is generated using a combination of AND (&) and OR (|) operations. It ensures that a carry is produced when any two of the three inputs (A, B, or cin) are high.
- Usage:
  This module is instantiated multiple times in a generate loop to build a 64-bit ripple-carry adder.

2. **Bitwise NOT Module**

```
module bitwise_not(input [63:0] A, output [63:0] Y);
  assign Y = ~A;
endmodule
```

- **Purpose:**
  The bitwise_not module performs a bitwise NOT operation on a 64-bit input vector.
- **Operation Details:**
  - It inverts each bit of the input A using the unary NOT operator (~), resulting in a 64-bit output Y.
- **Usage:**
  This module is used in the subtraction module to create the two's complement of a number.

3. **64-bit Adder Module**

```
module adder(input signed [63:0] A, B, input cin, output [63:0] sum, output cout);
  wire [63:0] carry;
  genvar i;
  generate
    for (i = 0; i < 64; i = i + 1) begin: FA
      if (i == 0)
        full_adder FA (A[i], B[i], cin, sum[i], carry[i]);
      else
        full_adder FA (A[i], B[i], carry[i-1], sum[i], carry[i]);
    end
  endgenerate
  assign cout = carry[63];
endmodule
```

- **Purpose:**
  The adder module performs 64-bit addition on two signed numbers.
- **Operation Details:**
  - **Generate Loop:**
    A generate loop instantiates 64 instances of the full_adder module, one for each bit.
  - **Carry Propagation:**
    - For the least significant bit (LSB), the external carry input (cin) is used.
    - For each subsequent bit, the carry output from the previous full adder is used as the carry-in.
  - **Final Carry:**
    The final carry-out from the most significant bit (MSB) is assigned to the module output cout.
- **Usage:**
  This module is used to calculate the sum for arithmetic operations in the ALU.

4. 64-bit Subtractor Module

```
module subtractor_64bit(input signed [63:0] A, B, output [63:0] diff, output cout);
  wire [63:0] B_neg;
  bitwise_not B_INV (B, B_neg);
  adder ADD (A, B_neg, 1'b1, diff, cout);
endmodule
```

- **Purpose:**
  The subtractor_64bit module calculates the difference between two 64-bit signed numbers.

- **Operation Details:**
  - **Two's Complement Subtraction:**
    - It first inverts the bits of B using the bitwise_not module to obtain B_neg.
    - Then, it adds A and B_neg with an initial carry-in of 1 using the adder module.
    - This effectively computes A - B using the two's complement method.

- **Usage:**
  This module provides subtraction functionality to the ALU.

4. Bitwise AND Module

```
module bitwise_and(input [63:0] A, B, output [63:0] Y);
  genvar i;
  generate
    for (i = 0; i < 64; i = i + 1) begin: gen_and_gates
      and a1 (Y[i], A[i], B[i]);
    end
  endgenerate
endmodule
```

- **Purpose:**
  The bitwise_and module computes the bitwise AND of two 64-bit input vectors.

- **Operation Details:**
  - A generate loop instantiates 64 individual AND gates.
  - Each bit of the output Y is the result of AND-ing the corresponding bits of A and B.

- **Usage:**
  This module provides bitwise AND functionality to the ALU.

6. Bitwise OR Module

```
module bitwise_or(input [63:0] A, B, output [63:0] Y);
  genvar i;
  generate
    for (i = 0; i < 64; i = i + 1) begin: gen_or_gates
      or o1 (Y[i], A[i], B[i]);
    end
  endgenerate
endmodule
```

- **Purpose:**
  The bitwise_or module computes the bitwise OR of two 64-bit input vectors.

- **Operation Details:**
  - Similar to the AND module, a generate loop creates 64 OR gates.
  - Each output bit Y[i] is the result of OR-ing the corresponding bits of A and B.
- **Usage:**
  This module provides bitwise OR functionality to the ALU.

7. ALU Module

```
module ALU(
    input signed [63:0] A, B,
    input [3:0] control,
    output reg signed [63:0] result,
    output reg Zero
);
  wire [63:0] sum, diff, and_out, or_out;
  wire cout;

  adder ADD (A, B, 1'b0, sum, cout);
  subtractor_64bit SUB (A, B, diff, cout);
  bitwise_and AND_ (A, B, and_out);
  bitwise_or OR_ (A, B, or_out);

  always @(*) begin
    case (control)
      4'b0010: result = sum;
      4'b0110: result = diff;
      4'b0000: result = and_out;
      4'b0001: result = or_out;
      default: result = 64'b0;
    endcase

    Zero = (result == 64'b0) ? 1'b1 : 1'b0;
  end
endmodule
```

- **Purpose:**
  The ALU module performs arithmetic and logical operations based on a 4-bit control signal.

- **Inputs and Outputs:**
  - **Inputs:**
    - Two 64-bit signed operands A and B.
    - A 4-bit control signal that selects the operation.
  - **Outputs:**
    - result: A 64-bit signed output representing the outcome of the selected operation.
    - Zero: A flag set to 1 if the result is zero; otherwise, it is 0.

- **Internal Signals:**
  - Intermediate wires (sum, diff, and_out, or_out) hold the outputs from the adder, subtractor, bitwise AND, and bitwise OR modules.
  - A wire cout is used to capture the carry-out from the arithmetic operations.

- **Module Integration:**
  - **Arithmetic Operations:**
    - The adder module is used for addition, and the subtractor_64bit module is used for subtraction.
  - **Logical Operations:**
    - The bitwise_and and bitwise_or modules are used to perform the respective bitwise operations.

- **Operation Selection (Case Statement):**
  - The always block is combinational and selects the correct operation based on the control signal:
    - 4'b0010: Executes addition (result = sum).
    - 4'b0110: Executes subtraction (result = diff).

- 4'b0000: Executes bitwise AND (result = and_out).
- 4'b0001: Executes bitwise OR (result = or_out).
- **Default:**
  If none of the cases match, the result defaults to 64'b0.

- **Zero Flag Computation:**
  After the operation is performed, the Zero flag is assigned based on whether the result is zero. This is useful for branch decisions in the processor.

ALU CONTROL

```
module alu_control(
    input  [1:0] ALUOp,
    input  [2:0] func3,
    input  [6:0] func7,
    output reg [3:0] ALUControl
);
    wire func7_5 = func7[5];

    always @(*) begin
        case (ALUOp)
            2'b00: ALUControl = 4'b0010;
            2'b01: ALUControl = 4'b0110;
            2'b10: begin
                case (func3)
                    3'b000: begin
                        if (func7_5 == 1'b1)
                            ALUControl = 4'b0110;
                        else
                            ALUControl = 4'b0010;
                    end
                    3'b111: ALUControl = 4'b0000;
                    3'b110: ALUControl = 4'b0001;
                    default: ALUControl = 4'b1111;
                endcase
            end
            default: ALUControl = 4'b1111;
        endcase
    end
endmodule
```

D. Module Purpose

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE C.2.1 The truth table for the four ALU control bits (called Operation) as a function of the ALUOp and function code field.

The alu_control module maps high-level ALU operation signals along with specific instruction function bits to a precise 4-bit control signal (ALUControl) used by the ALU. This control signal selects the correct arithmetic or logical operation (addition, subtraction, bitwise AND, bitwise OR, etc.).

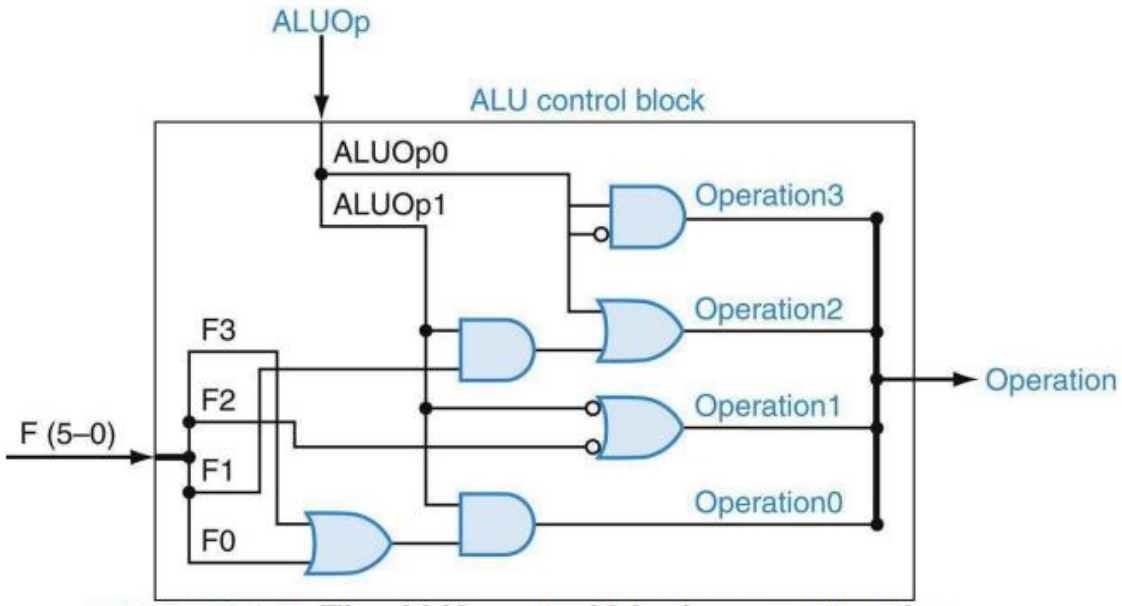


FIGURE C.2.3 The ALU control block generates the four ALU control bits, based on the function code and ALUOp bits.

E. Inputs and Outputs

Inputs

- ALUOp (2 bits):
  This signal is provided by the main control unit and represents the overall ALU operation type for a given instruction.
    - For example, 2'b00 might correspond to an addition operation (often used for load and store instructions).
    - 2'b01 could denote a subtraction operation (commonly used for branch comparisons).
    - 2'b10 indicates that the instruction is an R-type instruction, where the operation is further determined by the function fields.
- func3 (3 bits):
  This field comes directly from the instruction and specifies the operation type for R-type instructions. It helps differentiate between operations like add, subtract, and, or, etc.
- func7 (7 bits):
  Another field from the instruction, where one specific bit (bit 5) is used in conjunction with func3 to determine whether an operation should be an addition or subtraction.
    - In many RISC-V instructions, the difference between add and sub is indicated by this bit.

Output

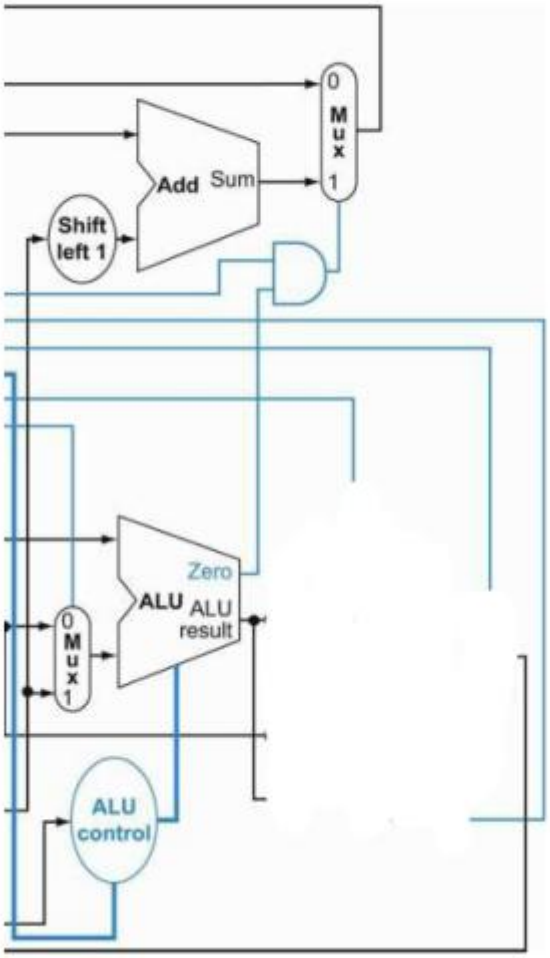
- ALUControl (4 bits):
  This is the control signal that is sent to the ALU. It precisely selects the arithmetic or logical operation:
    - For example, 4'b0010 might represent addition, 4'b0110 for subtraction, 4'b0000 for AND, and 4'b0001 for OR.
    - The default value 4'b1111 is used as an error or undefined operation code.

- **Function:**
    - Performs the actual computation using the Arithmetic Logic Unit (ALU).
    - Calculates addresses for load/store operations.

- Evaluates branch conditions (e.g., for beq) and computes branch target addresses.

- **Module Considerations:**
  - The ALU supports arithmetic (e.g., add, sub) and logical operations (e.g., and, or).
  - Integration of forwarding logic to reduce data hazards, ensuring that results from recent operations are available to subsequent instructions without waiting for the entire pipeline to clear.

EX: Execute Stage



F.Memory Access (MEM) Stage

The data_memory module in the memblock.v file represents the data memory component in a processor’s datapath. It is responsible for:

- Reading from memory (when memRead is enabled).
- Writing to memory (when memWrite is enabled).
- Resetting memory (when reset is asserted).
- Synchronous write operations (write on the rising edge of clk).
- Asynchronous read operations (data is read whenever memRead is high).

This module models a simple RAM (Random Access Memory) where:

- Memory is addressed in 64-bit words.
- There are 1024 memory locations (64-bit wide).
- The memory is initialized to zero on reset.

Inputs and Outputs

- **Inputs:**
  - clk (Clock Signal): Synchronizes write operations.
  - memWrite (Memory Write Enable): If **1**, writes write_data to memory[address] on the **positive edge** of clk.
  - memRead (Memory Read Enable): If **1**, reads from memory[address] into read_data.
  - reset (Reset Signal): If **1**, initializes all memory locations to 0.
  - address (64-bit): Specifies the memory location for **reading or writing**.
  - write_data (64-bit): The data to be written when memWrite is **enabled**.
- **Outputs:**
  - read_data (64-bit): The **data read** from memory when memRead is high.

G. Read Operation (Combinational Logic)

Functionality

- This block executes **whenever memRead or address changes** (@(*) means it is **combinational**).
- If memRead is **enabled**, the data at memory[address] is read into read_data.
- If memRead is **disabled**, read_data is set to 0.

Why use = (blocking assignment)?

- Since **reading is a combinational operation**, blocking assignment ensures that the read value is updated **immediately**.

G. Write Operation (Sequential Logic)

Functionality

- This block executes on the rising edge of clk.
- If memWrite is enabled, write_data is stored in memory[address].
- Writes happen only on clock edges, making them synchronous.

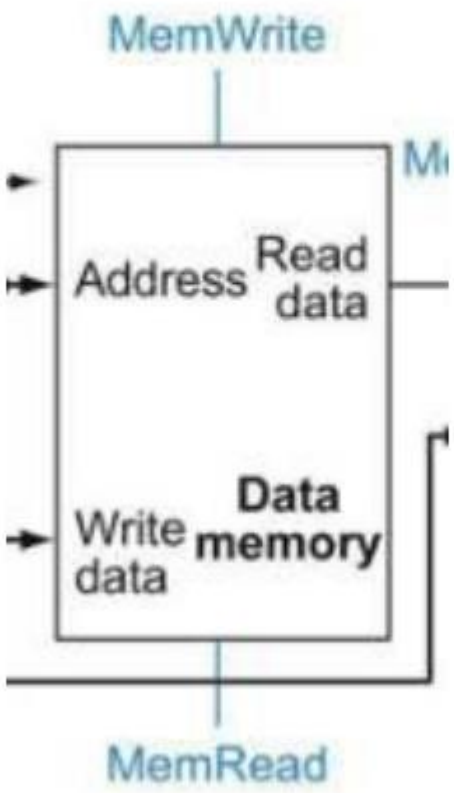
Why use <= (non-blocking assignment)?

- Ensures correct order of operations when multiple registers are updated in a clock cycle.

Summary of the entire module

- **Function:**
  - Accesses the data memory to read or write data for load (ld) and store (sd) instructions.
  - For load instructions, the data retrieved from memory is passed along to the write-back stage.
- **Module Considerations:**
  - A separate memory module (or dual-port memory) to handle simultaneous access if needed.
  - Memory control signals are generated based on the instruction type.

MEM: Memory Stage



H.Write Back (WB) Stage

1. Register File Module

The register_file module in the writeback.v file implements a small storage unit for the processor registers. It is designed to support both synchronous write and asynchronous read operations.

1.1 Inputs and Outputs

- Clock and Reset:
    - clk: The clock signal used to synchronize write operations.
    - reset: When asserted, this signal resets all the registers to a known state.

- Write Control:
    - regWrite: A control signal that enables writing to the register file.
    - write_reg: A 5-bit signal indicating which register to write into (register addresses 0 to 31).
    - write_data: A 64-bit data input that is written into the register specified by write_reg when regWrite is active.
- Read Ports:
    - read_reg1 and read_reg2: Two 5-bit signals used to select which registers to read.
    - read_data1 and read_data2: The corresponding 64-bit outputs from the registers specified by read_reg1 and read_reg2.

1.2. Internal Structure

- Register Array:
  The register file is implemented as an array of 32 registers, each 64 bits wide:
- **Synchronous Reset and Write Block:**
  A clocked always block handles both the reset and the write operations:
- **Reset:**
  On the rising edge of clk, if reset is high, all 32 registers are cleared (set to 0) using a for loop. Additionally, specific registers are pre-initialized:
- **Write Operation:**
  If the reset is not active and regWrite is enabled, the module writes the write_data into the register indexed by write_reg, provided the target register is not register 0. This protects register 0 from being overwritten:
- **Combinational Read Blocks:**
  Two separate combinational always @(*) blocks handle the asynchronous reading of data:
- For read_data1, if read_reg1 is 0, it returns 0; otherwise, it outputs the value stored in registers[read_reg1].
- Similarly for read_data2 using read_reg2.

- **1.3. Summary of the Register File**
- **Synchronous Write/Reset:**
  Writes to the registers occur only on the positive edge of clk. The module clears all registers when reset is active.
- **Asynchronous Read:**
  The outputs read_data1 and read_data2 are continuously updated based on the selected register addresses.
- **Protection of Register 0:**
  Register 0 is always maintained at 0, a common convention in many processor architectures.

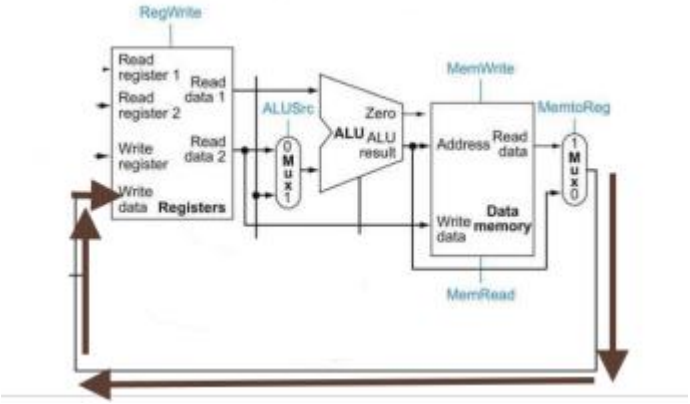
2. Multiplexer (Mux) Module

- The mux module is a simple 2-to-1 multiplexer that selects one of two 64-bit inputs based on a control signal.
- **2.1. Inputs and Outputs**
- **Inputs:**
- in0: The first 64-bit input.
- in1: The second 64-bit input.
- select: A control signal that determines which input is forwarded to the output.
- **Output:**
- out: A 64-bit output that is assigned either in0 or in1 based on the value of select.
- **2.2. Operation**
- The multiplexer uses a continuous assignment to decide the output:
- If select is 1, out is assigned the value of in1.
- If select is 0, out is assigned the value of in0.
- **2.3. Summary of the Mux**

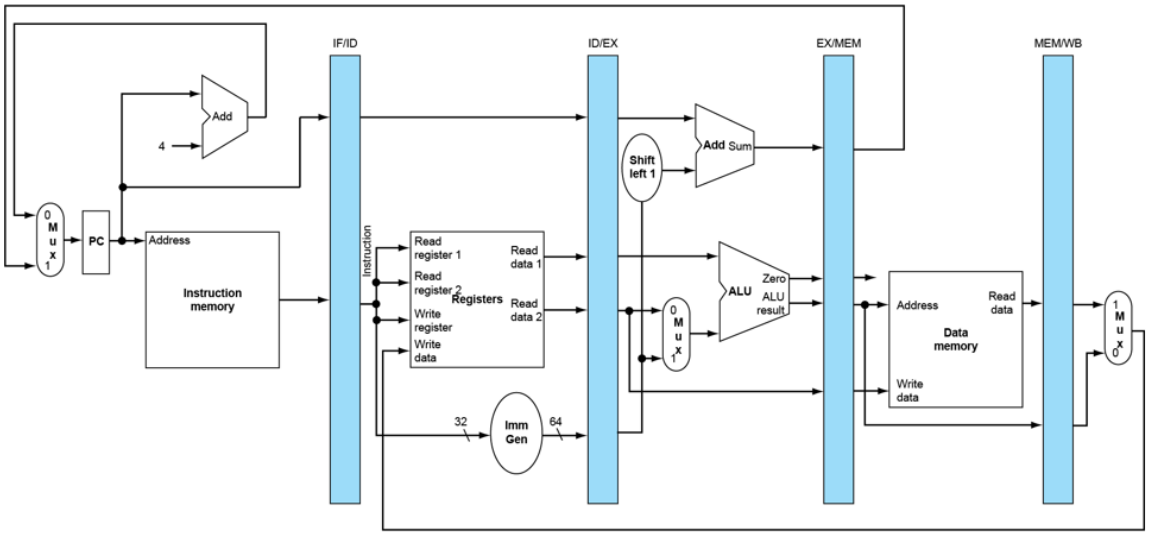
- **Simple and Efficient:**
  The mux is a fundamental combinational logic block that is widely used in hardware designs for selecting between alternative data sources.
- **Control Signal Based:**
  Its operation is determined solely by the select signal, making it very flexible for various routing tasks within the processor design.

Basic Functional Summary

- **Function:**
  - Writes the result from the execution or memory stage back into the register file.
  - Ensures that the register file is updated correctly to reflect the outcomes of instructions.
- **Module Considerations:**
  - Priority handling if multiple instructions attempt to write to the same register (typically managed via the forwarding and hazard detection logic).
  - Correct timing to update registers at the end of the instruction cycle.



I. All Pipelined Registers



1. Immediate Extension Module

Purpose

The immediate extension module in processortop_pipeline.v converts a 32-bit immediate value from the instruction into a 64-bit signed value. This is necessary because the processor operates on 64-bit data, and sign extension preserves the signed nature of the immediate.

Explanation

- **Sign Extension:**
  The code uses Verilog's replication operator ({32{imm_in[31]}}) to replicate the sign bit (bit 31) 32 times. This, concatenated with the 32-bit immediate, produces a 64-bit value.
- **Combinational Logic:**
  The always @* block ensures that the output is updated immediately when the input changes.

2. Pipeline Registers

The design uses several pipeline registers to hold intermediate signals between stages. Each register module is sensitive to the clock and reset, ensuring that data is properly synchronized.

2.1. IF/ID Pipeline Register

Purpose

Captures the Program Counter (PC) and instruction fetched in the Instruction Fetch (IF) stage and passes them to the Instruction Decode (ID) stage.

Explanation

- **Synchronous Operation:**
  On each positive edge of the clock, the IF-stage outputs (PC and instruction) are captured.
- **Reset Condition:**
  When reset is asserted, both the PC and instruction outputs are cleared.
- **Data Bundling:**
  The use of concatenation ({IF_pc, IF_instruction}) allows multiple signals to be transferred together.

2.2. ID/EX Pipeline Register

Purpose

Transfers control signals and data (e.g., PC, register operands, immediate value, destination register, and function fields) from the Instruction Decode (ID) stage to the Execute (EX) stage.

Explanation

- **Control Signals Transfer:**
  Control signals such as RegWrite, MemtoReg, and ALU settings are passed from the ID to the EX stage.
- **Data Path Signals:**
  Data values including the PC, register contents, immediate, and function fields are transferred.
- **Reset Behavior:**
  On reset, all outputs are cleared to prevent erroneous operations.

2.3. EX/MEM Pipeline Register

Purpose

Carries ALU results, second register operand, destination register, and additional control signals from the Execute (EX) stage to the Memory (MEM) stage.

Explanation

- **ALU Result and Second Operand:**
  These are forwarded to the memory stage for operations like memory store.
- **Control Signal Propagation:**
  Signals determining write-back or memory access are passed along.
- **Zero Flag:**
  The Zero flag, used for branch decisions, is also carried forward.

2.4. MEM/WB Pipeline Register

Purpose

Passes the final data (either from memory or ALU computation) and control signals from the Memory (MEM) stage to the Write-Back (WB) stage.

Explanation

- **Data Selection:**
  The write-back stage later chooses between the memory read data and the ALU result based on the MemtoReg signal.
- **Control Signals:**
  Only the signals required for writing back data are transferred.
- **Reset Mechanism:**
  Ensures that on reset, the write-back stage does not process stale or invalid data.

---

3. Top-Level Processor Pipeline Module

Purpose

The processor_pipeline module instantiates all the sub-modules and connects them to implement a complete pipelined processor. It demonstrates how instructions flow through the pipeline stages (IF, ID, EX, MEM, WB) while handling hazards and forwarding.

Explanation

- **PC Update Logic:**
  The Program Counter (PC) is updated every clock cycle. It uses branch logic (if EX_Branch is asserted and the ALU's zero flag is high) to calculate the branch target (EX_pc + EX_imm) or simply increments by one. The PC update is stalled if a hazard is detected.
- **Hazard Detection:**
  The hazard_detection_unit monitors the pipeline for load-use hazards. It takes the memory read signal, current instruction, and destination register from the EX stage to generate a stall signal if needed.
- **Forwarding Unit:**
  The ForwardingUnit inspects the source registers (rs1, rs2) and compares them with destination registers in later stages (MEM and WB). It generates control signals (forwardA, forwardB) to resolve data hazards by forwarding data when required.
- **Pipeline Stages:**
  - **IF Stage:**
    The instruction memory module (instruction_memory) fetches the instruction based on the PC.
  - **IF/ID Register:**
    Captures the PC and fetched instruction for use in the decode stage.
  - **ID Stage:**
    The instruction_decoder breaks the instruction into its fields (opcode, funct3, funct7, immediate, and register specifiers). The immediate value is then sign-extended by the immediate_extension module. The ControlUnit decodes the opcode to produce control signals.
  - **Register File:**
    The register file module reads operands (reg1 and reg2) based on the source register addresses.
  - **ID/EX Pipeline Register:**
    Transfers control and data signals (e.g., register operands, immediate, function fields) from the decode stage to the execution stage.
  - **(EX, MEM, WB Stages):**
    Though only partially shown here, subsequent pipeline registers (EX/MEM and MEM/WB) handle the results from the ALU, memory accesses, and control signals, eventually selecting the final write-back data.

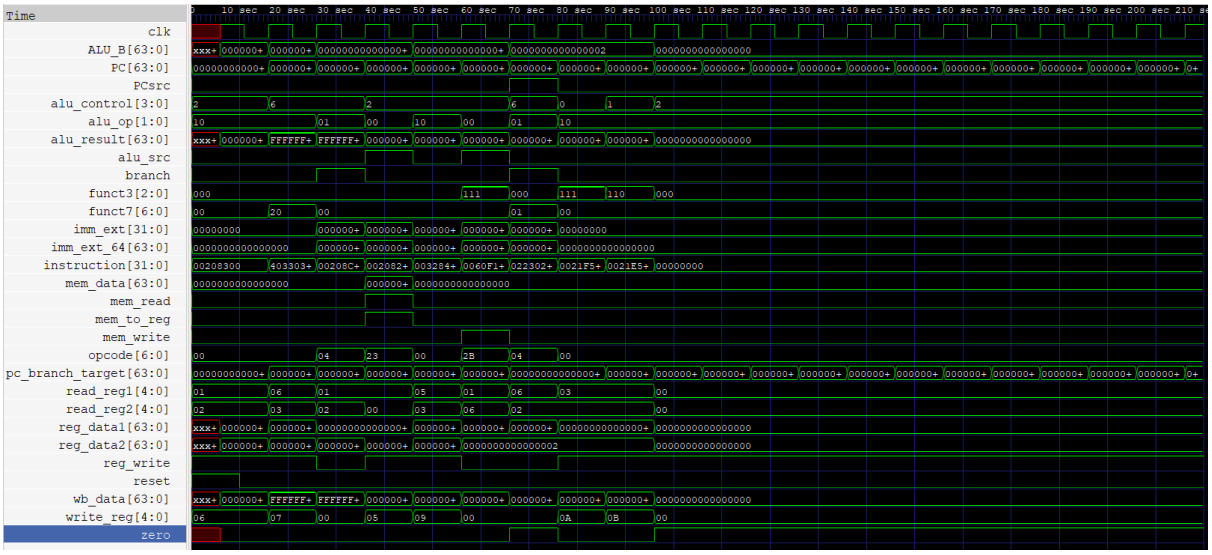
- **Write-Back Logic:**
  In the WB stage, a multiplexer selects between memory data and the ALU result based on the MemtoReg control signal. The selected data (wb_data) is then written back into the register file.

We have another pipeline+hazard detection and solving unit which will be explained in the next module

Instruction set:

```
ADD  x6, x1, x2      # x6  = x1 + x2  = 10 + 5 = 15
SUB  x7, x3, x4      # x7  = x3 - x4  = 1765 - 281 = 1484
BEQ  x5, x6, 5       # If x5 == x6, branch taken (PC += 5 instructions) (branch taken)
NOP                  # No operation (placeholder for 00)
NOP
NOP
NOP
AND  x8, x3, x4      # x8  = x3 & x4  = 1
OR   x9, x3, x4      # x9  = x3 | x4  = 2045
SD   x6, 10(x0)      # Store x5 at memory[10] (mem[10] = 15)
SD   x9, 1(x5)       # Store x9 at memory[x5 + 1] (mem[16] = 2045)
BEQ  x1, x2, 10      # If x1 == x2, branch taken (not taken in this case)
LD   x10, 10(x1)      # Load x10 = mem[10] = 15
LD   x11, 16(x0)     # Load x11 = mem[16] = 2045
ADD  x12, x10, x11   # x12 = x10 + x11 = 15 + 2045 = 2060
ADD  x0, x1, x2      # x0 should not change
```

Gtkwave:



These are the results for a non hazard handling pipeline which gives us inaccurate results and doesnot take delays into account which need to be taken in especially for cases when stalling is required for hazard cases like control hazards,data hazards and structure hazards

6. Hazard Detection and Resolution

1. Data Hazards

1.1. What Are Data Hazards?

Data hazards occur when instructions that are close together in the pipeline depend on each other's results. They typically manifest when one instruction (the consumer) requires data that a previous instruction (the producer) has not yet written back to the register file.

Types of Data Hazards:

- **RAW (Read After Write):** The most common hazard, where an instruction tries to read a source operand before the previous instruction writes it.
- **WAR (Write After Read) and WAW (Write After Write):** These are less common in a simple in-order pipeline, as instructions are processed in order.

1.2. How to Detect Data Hazards

Step-by-Step Detection:

1. **Instruction Decoding:**
    - In the Instruction Decode (ID) stage, check if the source registers of the current instruction match the destination registers of instructions that are still in the pipeline (for example, those in the Execute or Memory stages).
2. **Hazard Detection Unit (HDU):**
    - Implement an HDU that monitors the pipeline registers. When a hazard is detected (e.g., the next instruction's source is still being computed), the unit signals the control logic.

Hazard Detection unit (hazard_detection_unit.v)

1.In pipelined processor architectures, hazards arise when different instructions interfere with each other, leading to incorrect execution or performance degradation. The Hazard Detection Unit (HDU) is responsible for identifying such hazards and taking appropriate actions, such as stalling or flushing the pipeline, to ensure correct program execution.

This module specifically detects three types of hazards:

- Data Hazards (specifically load-use hazards)
- Structural Hazards (resource conflicts)
- Control Hazards (branch-related issues)

The HDU takes multiple inputs from different pipeline stages and generates control signals to stall or flush the pipeline as necessary.

Module Functionality

The hazard_detection_unit module is implemented in Verilog and monitors pipeline dependencies and conflicts.

Inputs

The HDU takes the following inputs:

- MemRead_EX (from the EX stage): Indicates whether the instruction in the EX stage is a load (lw) instruction. Load-use hazards arise when a subsequent instruction depends on the loaded data.
- Rd_EX (Destination register in EX stage): The destination register of the instruction in the EX stage. This is used to check if a subsequent instruction in the pipeline depends on it.
- structural_hazard (Structural hazard signal): Indicates if a resource conflict (such as memory port contention) occurs.
- branch (Branch control signal): Indicates if a branch instruction is taken. This can cause control hazards.
- IFID_inst (Instruction in the IF/ID stage): The instruction currently being fetched and decoded. The source registers from this instruction are checked for dependencies.

Outputs

The module generates two control signals to handle hazards:

- stall: When high (1), the pipeline is stalled to prevent hazards.
- flush: When high (1), the instruction in the IF/ID stage is flushed to avoid control hazards.

Strategies for Solving Data Hazards

A. Data Forwarding (Bypassing)

- **Concept:**
  - Instead of waiting for a value to be written back to the register file, the result is "forwarded" directly from a later pipeline stage (like Execute or Memory) back to the earlier stage that needs it.
- **Step-by-Step Implementation:**

1. **Identify Hazard Cases:**
   - Check if an instruction in the EX or MEM stage is writing to a register that the current instruction in the ID or EX stage needs.
2. **Add Forwarding Paths:**
   - Use multiplexers to select between the normally read operand from the register file and the forwarded value from a later stage.
3. **Control Logic:**
   - Design control logic to assert the appropriate multiplexer select signals when a hazard is detected.

B. Pipeline Stalling (Inserting Bubbles)

- **Concept:**
  - When forwarding is insufficient (for example, in a load-use hazard), the pipeline can be stalled—introducing a "bubble" to delay the dependent instruction until the data is available.
- **Step-by-Step Implementation:**

1. **Detect Load-Use Hazards:**
    - Identify cases where an instruction uses a load result immediately in the next cycle.
2. **Control Signal:**
    - Generate a stall signal that holds the pipeline registers in the ID stage and inserts a no-operation (NOP) into the EX stage.
3. **Resume Normal Operation:**
    - Once the data becomes available, release the stall signal to allow the pipeline to continue.

2. Structural Hazards

Structural hazards arise when the hardware structure of a processor is insufficient to support all possible simultaneous operations in the pipeline. These conflicts can lead to stalls in the pipeline because multiple stages are competing for the same resource. Examples include:

- **Memory Access Conflicts:**
  If the instruction memory and data memory share the same physical memory or access port, an instruction fetch and a data access (load/store) might conflict if they occur in the same cycle.
- **ALU or Functional Unit Conflicts:**
  If there is only one ALU or a specific functional unit, and multiple instructions require its usage simultaneously, a conflict will occur.
- **Register File Access:**
  If the register file cannot support simultaneous read and write operations or multiple reads from different instructions, then a structural hazard can appear.

Causes and Examples

Example: Unified Memory

In a processor with a unified memory architecture, both instructions and data share the same memory unit. In a pipelined design:

- **Cycle Conflict:**
  The Instruction Fetch (IF) stage and the Memory Access (MEM) stage may need to access the memory at the same time.
- **Result:**
  This leads to a structural hazard because the memory unit is occupied by one stage, forcing the other to wait.

Example: Single ALU

If a design includes only one ALU that is used by multiple pipeline stages (for example, both for arithmetic operations and address calculations), simultaneous demand by two different instructions will cause a conflict.

Strategies to Solve Structural Hazards

Pipeline Stall or Bubble Insertion

- **Stalling the Pipeline:**
  When a structural hazard is detected, the pipeline can be stalled, which means inserting bubbles (NOPs) until the required resource becomes available. This solution is often used as a last resort because it degrades performance.

- **Example:**
  In a unified memory design, if an instruction fetch and a data access conflict, one of the stages might be stalled for one cycle, effectively delaying the instruction fetch until the memory is free.

Hardware Reorganization

- **Implementing Dual-Ported Memories:**
  By designing memory modules with dual ports, you allow two simultaneous accesses—one for instruction fetch and one for data access. This reduces the chance of conflict.

- **Designing for Parallelism:**
  Adjust the architecture so that certain stages have their own dedicated hardware resources. For example, providing separate execution units for different types of instructions can prevent conflicts in the execution stage.

Forwarding Unit

The module named ForwardingUnit in the file (forwardingunit.v) is designed to resolve data hazards in a pipelined CPU by selecting the appropriate data source when an instruction depends on a result that has not yet been written back to the register file. In pipelined architectures, such hazards occur when instructions in different pipeline stages require access to the same registers. This unit generates control signals (Forward_A and Forward_B) that determine which data should be forwarded to the execution stage.

Module Interface

Inputs

1. **RS_1 (5 bits)**
    - Represents the first source register from the ID/EX stage (i.e., the register number that holds one of the operands needed for the ALU operation).
2. **RS_2 (5 bits)**
    - Represents the second source register from the ID/EX stage.
3. **rdMem (5 bits)**
    - This is the destination register of the instruction in the EX/MEM pipeline stage (i.e., the register where the result of an instruction is written in the memory stage).
4. **rdWb (5 bits)**
    - This is the destination register of the instruction in the MEM/WB pipeline stage (i.e., the register where the result of an instruction is written in the write-back stage).
5. **regWrite_Mem (1 bit)**

- A control signal from the EX/MEM stage that indicates if the instruction will write to a register. A nonzero value implies that the instruction intends to write its result back to the register file.

6. **regWrite_Wb (1 bit)**
    - A control signal from the MEM/WB stage indicating if the instruction will write to a register.

Outputs

1. **Forward_A (2 bits)**
    - This signal controls the source of the operand for the first ALU input. The values indicate:
        - 2'b10: Data should be forwarded from the EX/MEM stage.
        - 2'b01: Data should be forwarded from the MEM/WB stage.
        - 2'b00: No forwarding is required (i.e., the operand should be read from the register file).
2. **Forward_B (2 bits)**
    - This signal performs the same function as Forward_A but for the second ALU input.

The forwarding unit consists of two combinational logic blocks (implemented with always @(*)), each responsible for setting the appropriate forwarding control for one of the two source operands.

Forwarding for the First Operand (Forward_A)

1. **Check EX/MEM Stage (Highest Priority)**

    The first condition checks if the destination register from the EX/MEM stage (rdMem) matches the source register (RS_1)

Explanation:

rdMem == RS_1: Confirms that the register in the EX/MEM stage is the one that the current instruction is trying to read.

regWrite_Mem != 0: Ensures that the instruction in the EX/MEM stage will actually write to the register (i.e., it is not a no-op or a control instruction that does not modify registers).

rdMem != 0: Avoids forwarding if the destination register is the zero register (commonly used in RISC architectures as a constant zero value).

If this condition is met, Forward_A is set to 2'b10, meaning the operand should be taken (forwarded) from the EX/MEM stage.

Check MEM/WB Stage

If the first condition is not met, the next condition checks if the destination register from the MEM/WB stage (rdWb) matches the source register (RS_1):

- **Explanation:**
- **rdWb == RS_1**: Checks for a match between the MEM/WB destination register and the source register.
- **regWrite_Wb != 0**: Confirms that the MEM/WB stage instruction will write to the register.
- **rdWb != 0**: Again, avoids unnecessary forwarding from the zero register.
- If this condition is satisfied, Forward_A is set to 2'b01, selecting the operand from the MEM/WB stage.

Default Case

- If neither condition applies:
- else Forward_A = 2'b00;
- **Explanation:** The operand is read directly from the register file because no data hazard requiring forwarding exists.

Forwarding for the Second Operand (Forward_B)

The logic for Forward_B is analogous to that for Forward_A, but it uses the second source register (RS_2):

1. **Check EX/MEM Stage**
    - The first condition for Forward_B is:

- **Explanation:**
    - This checks if the destination register in the EX/MEM stage matches RS_2 and whether that instruction is writing to a register (while also ensuring it isn't the zero register). If true, the operand is forwarded from the EX/MEM stage.

2. **Check MEM/WB Stage**
    - If the above condition is false:
    - **Explanation:**
        - This condition checks if the MEM/WB stage should provide the forwarded data by matching RS_2 with rdWb and validating that writing is enabled and not targeting the zero register.
3. **Default Case**

If neither condition holds:

Explanation: No hazard is detected; therefore, the second operand is taken from the register file.

3. Control Hazards

3.1. What Are Control Hazards?

Control hazards (or branch hazards) arise from the need to make decisions based on branch instructions (like conditional branches and jumps). Since the outcome of a branch isn't known until a later pipeline stage, subsequent instructions might be fetched under the wrong assumption.

3.2. How to Detect Control Hazards

Step-by-Step Detection:

1. **Branch Instructions in Decode or Execute Stage:**
    - The control logic identifies branch instructions as soon as they are decoded.

2. **Delayed Decision Point:**
    - Since the branch outcome (taken or not taken) is determined in the Execute (or sometimes MEM) stage, any instructions fetched after the branch might be incorrect if the branch is taken.

3.3. Strategies for Solving Control Hazards

A. Branch Prediction

- **Concept:**
    - Predict whether a branch will be taken or not to decide which instruction to fetch next. If the prediction is correct, the pipeline continues without interruption.
- **Step-by-Step Implementation:**

1. **Predictor Design:**
    - Implement a simple predictor (e.g., a static predictor that always assumes “not taken” or a dynamic predictor using historical data).
2. **Speculative Fetching:**
    - Continue fetching instructions based on the prediction.
3. **Verification:**
    - Once the branch outcome is determined in the Execute stage, compare it with the prediction.
4. **Misprediction Handling:**
    - If the prediction was wrong, flush the incorrect instructions from the pipeline and fetch the correct ones.

B. Pipeline Flushing

- **Concept:**
    - When a branch is determined to be taken, flush the instructions that were fetched after the branch. This clears the pipeline of incorrect instructions.
- **Step-by-Step Implementation:**

1. **Detection:**
    - Detect the branch outcome as soon as it is available.
2. **Flush Signal:**
    - Generate a flush signal that clears the instructions in the IF and ID stages.
3. **Fetch Correct Instruction:**
    - Set the Program Counter (PC) to the correct branch target and resume fetching.

C. Delayed Branching

- **Concept:**
    - Rearrange instructions so that the instruction immediately following a branch is always executed, regardless of whether the branch is taken.
- **Step-by-Step Implementation:**

1. **Compiler Scheduling:**
    - The compiler reorders instructions to fill the branch delay slot with an instruction that is safe to execute regardless of the branch outcome.
2. **Hardware Support:**
    - Minimal hardware changes are required since the processor always executes the delay slot.

7.FINAL CODE IMPLEMENTING ALL THE HAZARD CONTROLS

1.The code implements a pipelined processor design. It is divided into several stages with corresponding pipeline registers:

- **IF (Instruction Fetch)**

- **ID (Instruction Decode)**
- **EX (Execute)**
- **MEM (Memory)**
- **WB (Write Back)**

The design includes modules for instruction memory, register file, ALU, data memory, and several pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB). In addition, there are specialized units for hazard detection and forwarding. These units ensure that the pipeline executes instructions correctly even when instructions have interdependencies or share hardware resources.

2. Handling Data Hazards

Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its write-back stage. This design uses **forwarding (bypassing)** and **stalling** mechanisms to mitigate these hazards:

2.1 Forwarding Unit

- **Purpose:**
  The forwarding unit is designed to resolve data hazards by redirecting the most recent data from later pipeline stages (EX/MEM or MEM/WB) back to the ALU inputs in the Execute stage. This avoids waiting for the data to be written back to the register file.
- **How It Works:**
  - It compares the source register addresses in the ID/EX stage (RS_1 and RS_2) with the destination register addresses from the EX/MEM (rdMem) and MEM/WB (rdWb) stages.
  - If there is a match and the corresponding stage is writing back (as indicated by regWrite_Mem or regWrite_Wb), the control signal (Forward_A or Forward_B) is set:
    - 2'b10 indicates that the data should be taken from the EX/MEM stage.

        - 2'b01 indicates that the data should be taken from the MEM/WB stage.
        - 2'b00 means no forwarding is needed, so the operand is read from the register file.
    - The ALU operands are selected using a multiplexer based on these control signals. For example, for operand A: EX_reg1;
    - This scheme ensures that even if an instruction is waiting for the result of a previous operation, the latest value is forwarded directly to the ALU without waiting for the entire pipeline to complete.

2.2 Hazard Detection Unit

- **Purpose:**
  While the forwarding unit can resolve many data hazards, some hazards—especially those caused by load instructions—require stalling. A load-use hazard occurs when an instruction attempts to use a value that is being loaded from memory but has not yet arrived.
- **How It Works:**
    - The hazard detection unit monitors the instruction in the ID stage (via IFID_inst) and the EX stage (via EX_MemRead and EX_rd).
    - If the instruction in the EX stage is a memory read and its destination register matches one of the source registers in the instruction in the ID stage, then a hazard is detected.
    - Upon detection, the unit asserts a hazard_stall signal.
    - **Pipeline registers and PC update logic**:
        - The IF/ID and ID/EX pipeline registers check the hazard_stall signal. For instance, in the IF/ID register:

If hazard_stall is high, the pipeline does not advance the instruction, effectively stalling the IF stage.

        - Similarly, the PC update logic is held (i.e., the PC is not incremented) when a hazard is detected.

- This stalling mechanism allows the pipeline to wait until the data from the memory read is available, thus preventing the execution of an instruction with incomplete operands.

---

3. Handling Structural Hazards

Structural hazards occur when hardware resources are insufficient to support all concurrent operations. In this design:

- **Assumption in the Code:**
  The hazard detection unit instantiation includes an input called structural_hazard, which is set to a constant 1'b0:

This indicates that the current design assumes that structural hazards do not occur (or are handled elsewhere). The processor has been designed such that separate instruction and data memories (or sufficient hardware resources) prevent conflicts.

- **Implications:**
  Since the structural hazard signal is hardwired to zero, the hazard detection unit will not stall the pipeline for resource conflicts. This simplifies the design under the assumption that the processor is implemented on hardware that can support parallel operations without contention (for example, using separate memories or sufficient ALUs).

4. Detailed Pipeline Register Behavior

The pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) play a key role in hazard management:

- **IF/ID Register:**
  - Updates the PC and instruction only when there is no stall (hazard_stall == 0).
  - Prevents new instructions from entering the pipeline when a hazard is detected.
- **ID/EX Register:**
  - Transfers control signals, register values, immediate values, and register addresses from the ID stage to the EX stage.

    - In case of a reset or hazard stall, it clears or holds the values to prevent erroneous execution.

- **EX/MEM and MEM/WB Registers:**
    - Propagate ALU results, control signals, and memory read data to subsequent stages.
    - Maintain the correct data flow needed by the forwarding unit and the write-back stage.

- **Summary and Conclusion**
- **Data Hazards:**
  The processor uses a combination of a forwarding unit and a hazard detection unit to manage data hazards:
    - **Forwarding Unit:**
      Directly supplies the most recent data from later pipeline stages to the ALU when there is an inter-instruction dependency.
    - **Hazard Detection Unit:**
      Stalls the pipeline (by asserting the hazard_stall signal) when a load-use hazard is detected, ensuring that dependent instructions do not proceed until the data is available.
- **Structural Hazards:**
  The design assumes that structural hazards are not an issue by setting the structural_hazard signal to 0. This implies that the hardware resources (instruction and data memory, ALU, register file) are sufficient to handle parallel operations without conflicts.

The processor pipeline code is structured to maintain smooth instruction flow while correctly handling interdependencies through data forwarding and stalling mechanisms. The clear separation into pipeline stages, along with the use of hazard detection and forwarding, demonstrates a robust approach to mitigating hazards in a pipelined architecture.

8.Testbench Outputs and GTKWAVE

INSTRUCTIONS :

```
00000000001000001000001100000000
01000000001100110000001110000000
00000000001000001000110000000100
00000000001000001000001010100011
00000000001100101000010010000000
00000000011000001111000110101011
00000010001000110000001000000100
00000000001100001000000100000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000001100001000000100000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000001000011111010100000000
00000000001000011110010110000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

add x6 x1 x2

sub x7 x6 x3

beq x2 x1 12

ld x5 2(x1)

add x9 x5 x3

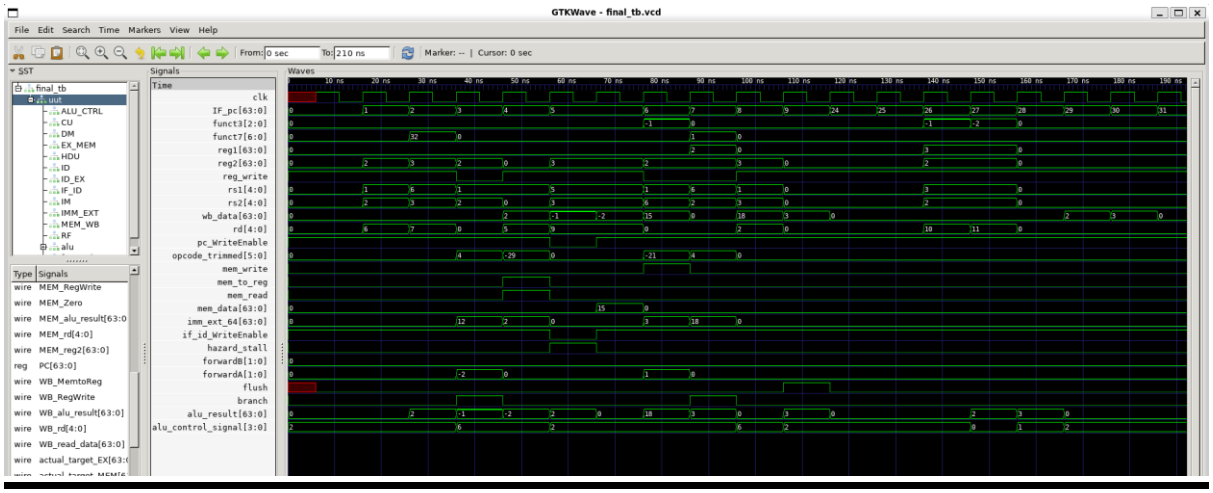
sd x6 3(x1)

beq x2 x6 18

add x2 x1 x3

and x10 x2 x3

or x11 x2 x3



Resources and Materials used :

- Computer Organization and Design The Hardware/Software Interface: RISC-V Edition
- Lecture Slides

Contribution Section:

Sequential Circuit Implementation

- Instruction Fetch , Memblock.v and writeback.v :Bhuvi
- Instdecoder.v block :Vidvathama R
- Control_unit block :Dhruv
- Compilation of sequential processor : Bhuvi and Dhruv

Pipelining Circuit Implementation

- Pipelined registers (IF/ID, ID/EX, EX/MEM, MEM/WB) : Vidvathama R
- Compilation of pipelined processor: Bhuvi
- Data Hazard : Bhuvi
- Control Hazard :Bhuvi, Dhruv
- Instructions.txt : Bhuvi
- Report :Vidvathama R