VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY (HCMUT)
FACULTY OF COMPUTER SCIENCE & ENGINEERING

Introduction to SoC Assignment

REPORT

Pipeline Datapath

MAJOR: COMPUTER ENGINEERING

Supervisors:

Assoc. Prof. Dr. Hoang Trong Thuc - UEC
Assoc. Prof. Dr. Tran Ngoc Thinh - HCMUT
Assoc. Inst. Pham Kieu Nhat Anh - HCMUT

Authors:

Pham Nguyen Minh Hieu - 2252215

HO CHI MINH CITY - December-2025

CONTENTS

1

THEORETICAL BASIS

1.1. Pipelined Datapath

1.1.1. Overview

Pipelined Datapath is a computer architecture design technique aimed at enhancing processor performance by exploiting **stage-level parallelism**. Unlike the Single-cycle architecture, which completes an entire instruction in one long clock cycle, Pipelining decomposes the instruction execution process into a sequence of independent, successive stages.

A standard implementation typically involves a **5-stage pipeline**: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). *Pipeline Registers* are placed between these stages to separate data and synchronize control signals across clock cycles. Pipeline registers named by stages they begin: PC, D, X, M and W.

The core advantage of Pipelining is the significant improvement in **instruction throughput**—the number of instructions completed per unit of time. Although this technique does not reduce the **latency** of an individual instruction (and may slightly increase it due to register overhead), allowing multiple instructions to overlap in execution drastically increases the overall program execution speed. This principle operates similarly to an assembly line in a factory, where items effectively enter and leave the process at a much faster rate.

inst0.fetch	inst0.dec	inst0.exec	
	inst1.fetch	inst1.dec	inst1.exec

Break instruction execution into stages

- When inst. advances from stage 1 to 2, next inst. enters at stage 1
- Exploits “stage-level parallelism”
- Maintains illusion of sequential fetch/execute loop

- Individual instruction takes the same number of stages
- **But instructions enter and leave at a much faster rate**

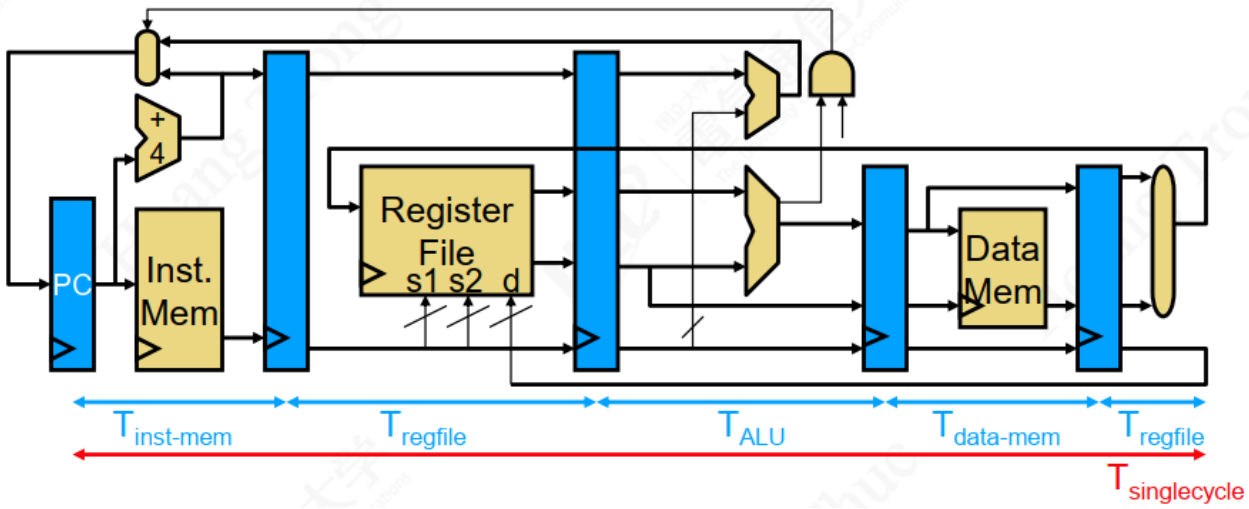

Figure 1.1: 5-stage pipelined

1.2. Dependencies

Dependencies are properties of the program logic rather than the underlying hardware. They dictate the order in which instructions must be executed to ensure the correctness of the program. There are two primary types of dependencies:

- **Data Dependencies:** These occur when an instruction refers to the data produced by a preceding instruction. For example:

  ```
  add x1, x2, x3   (Produces x1)
  sub x4, x1, x5   (Consumes x1)
  ```

  In this sequence, the `sub` instruction depends on the result of the `add` instruction. The second instruction cannot correctly execute until the first has updated the register `x1`.

- **Control Dependencies:** These determine the ordering of an instruction with respect to a branch instruction. An instruction inside an `if` block is control-dependent on the condition evaluation of the `if` statement.

1.3. Pipeline Hazards

Hazards are situations in hardware that prevent the next instruction in the instruction stream from executing during the following clock cycle. Hazards are often the hardware manifestation of dependencies.

1. **Structural Hazards:** arise when the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A common example is when a processor has a single memory unit for both instructions and data; a hazard occurs if the processor attempts to fetch an instruction and access data memory simultaneously.

2. **Data Hazards:** occur when an instruction depends on the result of a previous instruction that is still in the pipeline and has not yet been committed to the register file. The most common type is *Read After Write (RAW)*.

3. **Control Hazards (Branch Hazards):** arise from the pipelining of branches and other instructions that change the Program Counter (PC). The pipeline may stall because it does not yet know the address of the next instruction to fetch while the branch condition is being evaluated.

1.4. Handling Hazards: Stalling

Stalling is a method used to resolve hazards by suspending the execution of instructions.

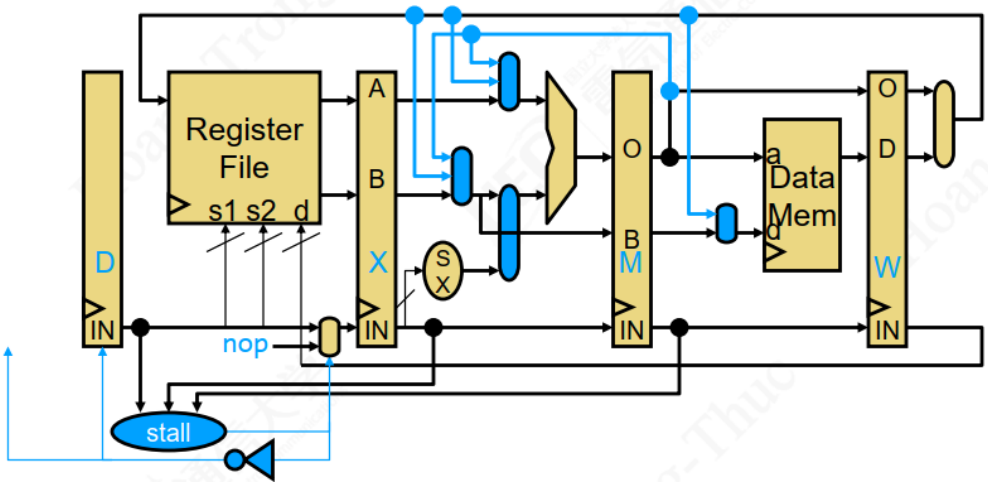

Figure 1.2: 5-stage pipelined stall

- **Mechanism:** When a hazard is detected, the control unit inserts a "bubble" or **NOP** (No Operation) into the pipeline. This effectively pauses the instruction fetch or execution in the dependent stages.
- **Impact:** While stalling ensures correctness, it negatively impacts performance by increasing the Cycles Per Instruction (CPI). During a stall, no productive work is performed, reducing the overall throughput of the processor.

1.5. Data Forwarding (Bypassing)

To minimize the performance penalty caused by stalling, modern processors employ a technique called **Forwarding** (also known as **Bypassing**).

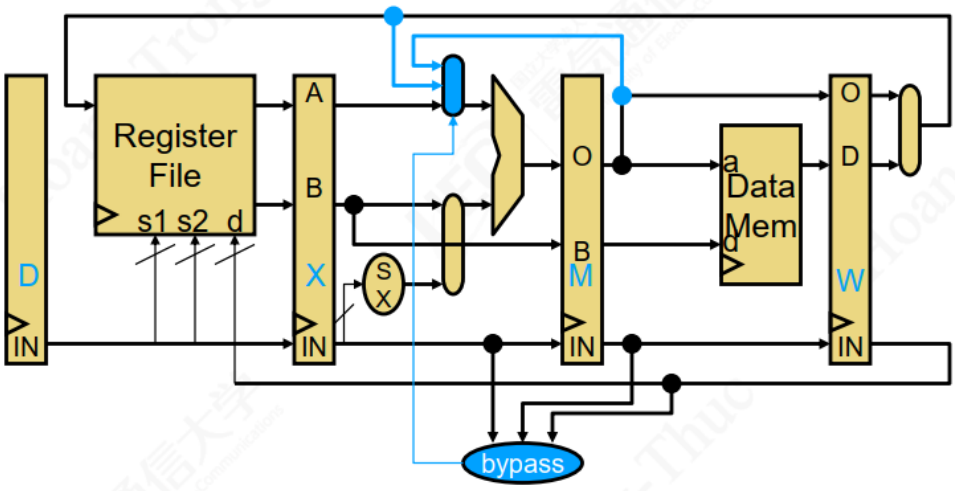

Figure 1.3: 5-stage pipelined bypass

1.5.1. Concept

Forwarding is a hardware optimization that resolves data hazards by retrieving the missing data directly from internal pipeline buffers rather than waiting for the data to be written back to the Register File.

The key insight is that the result of an instruction is often available in the pipeline registers (e.g., `EX/MEM` or `MEM/WB`) long before the instruction completes the Write Back (WB) stage. By adding multiplexers (MUX) to the inputs of the ALU, the datapath can select valid data from these intermediate stages to supply the current instruction.

1.5.2. Forwarding Mechanisms

In a standard 5-stage RISC-V pipeline, the ALU requires operands at the beginning of the Execute (EX) stage. Depending on where the dependency originates, different forwarding paths are utilized. These are often denoted by the stage supplying the data (Source) to the stage consuming the data (Destination, usually EX).

- **MX Bypassing (Memory-to-Execute):**
  - *Scenario:* A Data Hazard between adjacent instructions (Distance = 1).
  - *Description:* The result of the instruction immediately preceding the current one is currently in the **MEM stage** (sitting in the `EX/MEM` pipeline register).
  - *Mechanism:* The Forwarding Unit detects that the destination register (`Rd`) of the instruction in the MEM stage matches one of the source registers (`Rs1` or `Rs2`) of the instruction in the EX stage. It effectively "bypasses" the Register File by routing the data from the `EX/MEM` register directly back to the ALU input.
  - *Example:*

```
ADD x1, x2, x3
SUB x4, x1, x5    (Uses result of ADD immediately)
```

- **WX Bypassing (Writeback-to-Execute):**
  - *Scenario:* A Data Hazard between instructions separated by one instruction (Distance = 2).
  - *Description:* The result of an instruction executed two cycles ago is currently in the **WB stage** (sitting in the `MEM/WB` pipeline register).
  - *Mechanism:* The Forwarding Unit detects that the destination register (`Rd`) of the instruction in the WB stage matches a source register in the EX stage. The data is routed from the `MEM/WB` register to the ALU input.
  - *Example:*

```
ADD x1, x2, x3
OR x6, x7, x8
SUB x4, x1, x5    (Uses result of ADD after 1 cycle delay)
```

- **WM Bypassing** (or *Writeback-to-Memory* forwarding) is a mechanism used to resolve hazards where a value currently in the Writeback (WB) stage is needed by a Store instruction currently in the Memory (MEM) stage.
  - **Scenario:**

`ADD x1, x2, x3` (Produces x1, currently in WB stage)
`SW x1, 0(x4)` (Stores x1, currently in MEM stage)

- **Mechanism:** Without this bypass, the Store instruction might store an outdated value of `x1`. The WM Bypass path forwards the result from the end of the pipeline (the `MEM/WB` pipeline register) directly to the **Data Input** of the Data Memory unit in the MEM stage.

- **The "Load-Store" Dependency** A special optimization case arises in the interaction between Load and Store instructions, often referred to as the *Load-Use* case for Stores.

  Consider the following sequence where a value is loaded from memory and immediately stored to a different location:

  `LW x1, 0(x2)` (Loads memory to x1)
  `SW x1, 0(x3)` (Stores x1 to memory)

1.6. Pipelined Divider Unit

In complex arithmetic operations such as division, the combinational logic required is significantly deeper than standard ALU operations (like ADD or SUB). Consequently, executing division in a single cycle is often impossible without drastically reducing the clock frequency. To address this, the architecture employs a **Pipelined Divider**.

1.6.1. Architecture Overview

As illustrated in the system diagram, the Divider Unit is decoupled from the main execution stage. It operates in parallel with the standard 5-stage pipeline.

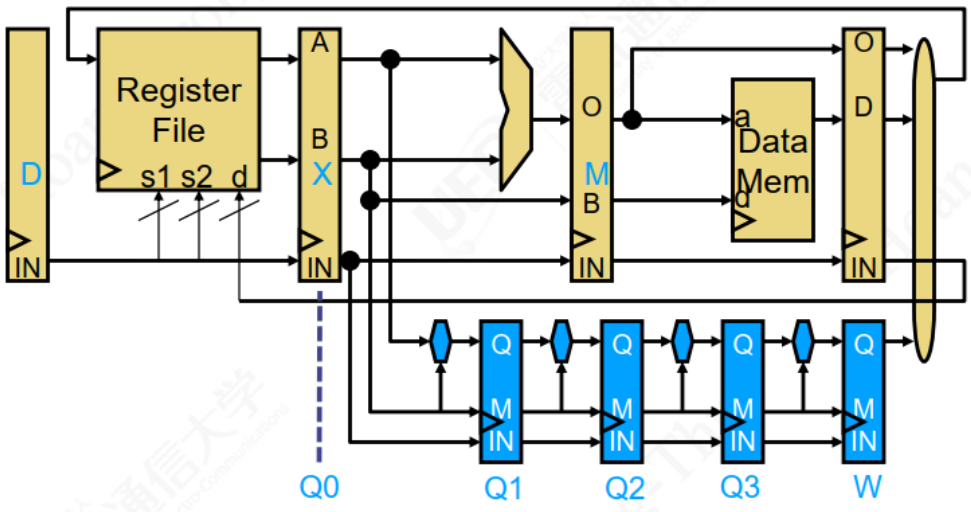

Figure 1.4: 5-stage pipelined divider

- **Parallel Execution:** When a divide instruction is decoded, operands are routed to the divider unit (stages labeled $Q0$ to W). Meanwhile, the main pipeline can effectively continue processing subsequent independent instructions (unless there is a dependency).
- **Multistage Design:** The division logic is decomposed into discrete stages (depicted as $Q0, Q1, Q2, Q3$). Pipeline registers separate these stages, holding partial results (partial remainders and quotients) between clock cycles.

1.6.2. Structural Implications

The integration of a pipelined divider introduces specific structural considerations:

- **Writeback Contention:** Since the divider takes 4 cycles and standard ALU operations take 1 cycle, it is possible for a divide instruction and a younger ALU instruction to reach the Write-back (WB) stage in the same clock cycle. The control logic must handle this structural hazard, typically by prioritizing one unit or stalling the pipeline.

2

PIPELINED DATAPATH IMPLEMENTATION

2.1. Overview

The main task in this assignment is to convert from multi-cycle datapath to 5-Stage Pipelined Datapath RISC-V processor. The five stages include: Fetch (IF) $\rightarrow$ Decode (ID) $\rightarrow$ Execute (EX) $\rightarrow$ Memory (MEM) $\rightarrow$ Writeback (WB).

The main task of the assignment is to apply the learned techniques (stall, flush, bypass) to resolve conflicts (Hazards) during instruction processing to ensure that the data is always correct.

In addition, we need to integrate the 4-stage divider implemented in lab 4 into the pipelined datapath to handle divisions, increasing the performance of the 5-stage RISC-V processor when encountering division instructions (which are very resource-consuming).

2.2. Architecture

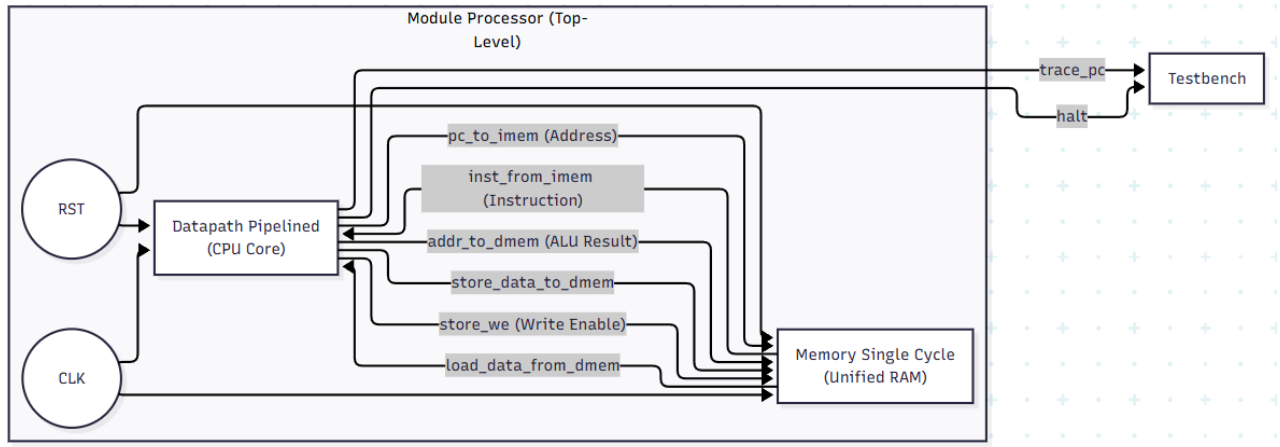

Figure 2.1: Top module architecture

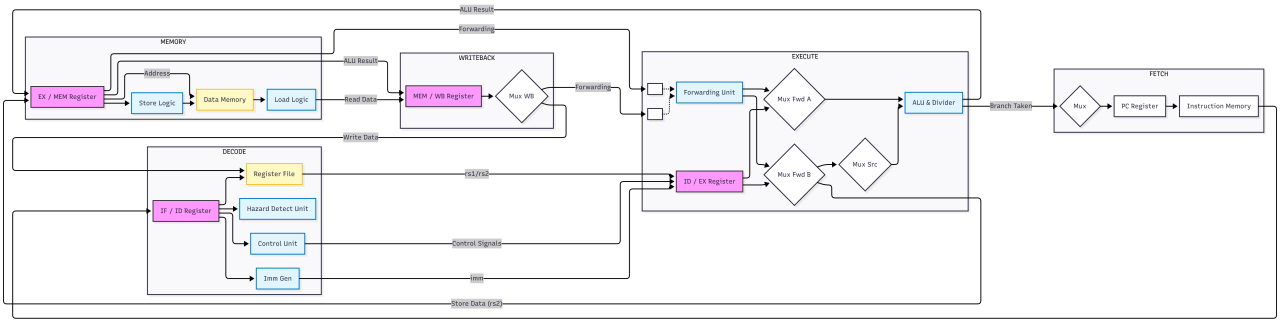

Figure 2.2: 5 Stage Pipelined architecture

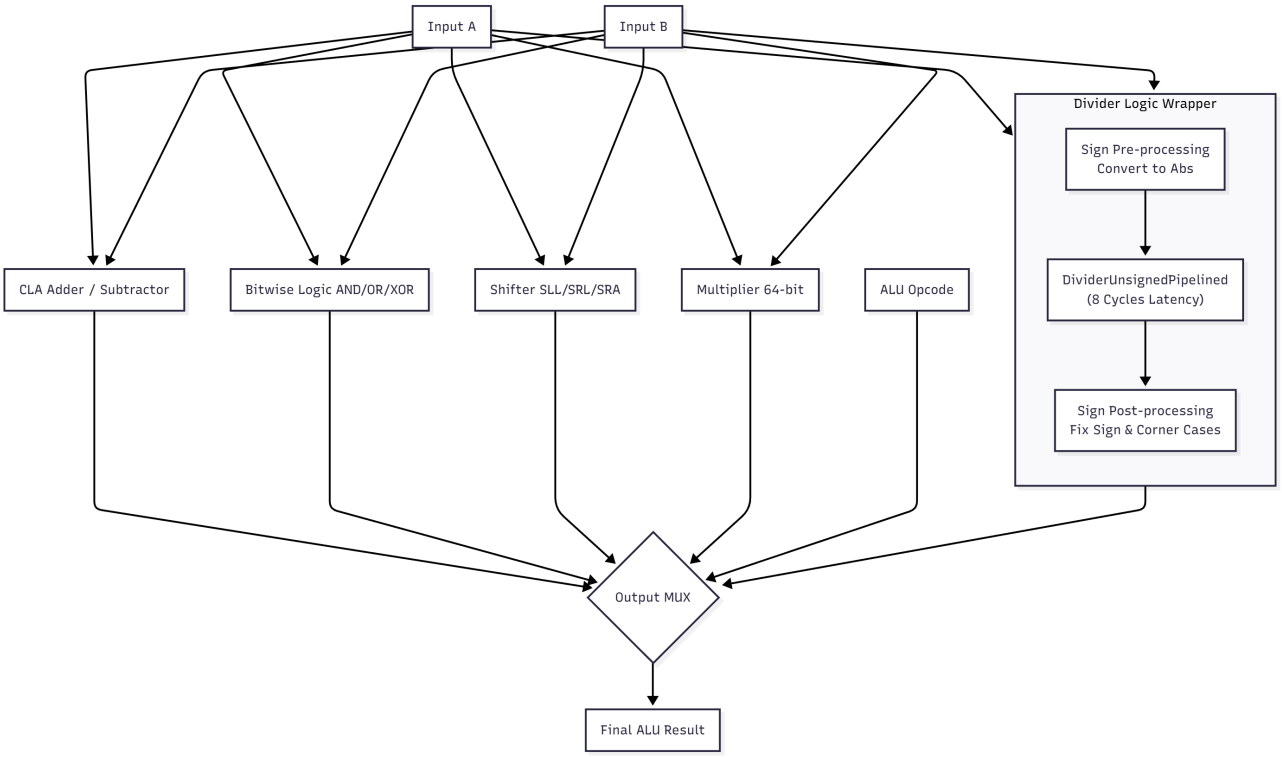

Figure 2.3: ALU Logic operations architecture

2.3. Detail Code Implementation

Here, for ease of management and debugging, we will divide the code into small parts that play different roles. Each small part will be called a small block, marked by commenting the block name above.

2.3.1. Module RegFile

```
module RegFile (
  input       [         4:0] rd,
  input       [`REG_SIZE:0] rd_data,
  input       [         4:0] rs1,
  output reg  [`REG_SIZE:0] rs1_data,
  input       [         4:0] rs2,
  output reg  [`REG_SIZE:0] rs2_data,
  input                     clk,
  input                     we,
  input                     rst
);
  localparam NumRegs = 32;
  reg [`REG_SIZE:0] regs[0:NumRegs-1];

  // TODO: your code here


  integer i;
  always @(posedge clk) begin
    if (rst) begin
      for (i = 0; i < NumRegs; i = i + 1) begin
        regs[i] <= 0;
      end
    end else if (we && rd != 0) begin
      regs[rd] <= rd_data;
    end
  end

  // WD Bypass: Forwarding logic for reading registers in the same
   cycle they are written
  always @(*) begin
```

```
31      if (rd == rs1 && we && rd != 0) rs1_data = rd_data;
32      else rs1_data = regs[rs1];
33
34      if (rd == rs2 && we && rd != 0) rs2_data = rd_data;
35      else rs2_data = regs[rs2];
36    end
37
38  endmodule
```

Functional

The RegFile module simulates the Architectural Register File of a RISC-V microprocessor. It contains 32 32-bit registers (from x0 to x31). The module supports simultaneous access:

- 2 Read Ports: rs1 and rs2 (Asynchronous).
- 1 Write Port: rd (Synchronous).

Realistic details

a. Write and Reset Mechanism (Sequential Logic)

The code uses the always @(posedge clk) block to ensure that data writing occurs synchronously with the rising edge of the clock.

- Reset: When the rst signal is positive, all registers from 0 to 31 are cleared to 0 using the for loop.
- Write data (Write Operation): Writing data to the destination register rd only occurs when:
  - The write enable signal we (Write Enable) is turned on.
  - The destination register rd is different from 0 (rd != 0). This complies with the RISC-V architecture standard, in which the x0 register is hard-coded to 0 and cannot be overwritten.

b. WD Bypass Mechanism (Internal Forwarding)

Problem: In the pipeline architecture, the Writeback phase (writing data to RegFile) occurs at the end of the cycle, while the Decode phase (reading RegFile) of the next instruction also occurs in the same cycle. Without special handling, the read instruction will get the old value before the write instruction has time to update.

Solution (WD Bypass): The module implements Internal Forwarding technique right inside the Register File.

The always @(*) combinational logic block checks whether the address of the register being read (rs1 or rs2) is the same as the address being written (rd) at the same time.

```
// WD Bypass: Forwarding logic for reading registers in the same
 cycle they are written
always @(*) begin
  if (rd == rs1 && we && rd != 0) rs1_data = rd_data;
  else rs1_data = regs[rs1];

  if (rd == rs2 && we && rd != 0) rs2_data = rd_data;
  else rs2_data = regs[rs2];
end
```

Explanation: If a duplicate address is detected (and there is a write signal), the module will bypass the value stored in the regs array and send the input data `rd_data` directly to the `rs1_data` read port. This ensures that the following command always receives the latest data even if the data has not yet been stably written to the memory array. The same goes for `rs2_data`.

2.3.2. Module Datapath Pipelined

Now, we will come to the most important part in implementing the 5-Stage Pipelined Datapath RISC-V processor, which is implementing the DatapathPipelined module. I will go into detail on each part to have the easiest possible understanding of the algorithm.

```
module DatapathPipelined (
  input                     clk,
  input                     rst,
  output      [`REG_SIZE:0] pc_to_imem,
  input       [`INST_SIZE:0] inst_from_imem,
  // dmem is read/write
  output reg  [`REG_SIZE:0] addr_to_dmem,
  input       [`REG_SIZE:0] load_data_from_dmem,
  output reg  [`REG_SIZE:0] store_data_to_dmem,
  output reg  [         3:0] store_we_to_dmem,
  output reg                halt,
  // The PC of the inst currently in Writeback. 0 if not a valid inst.
  output reg  [`REG_SIZE:0] trace_writeback_pc,
  // The bits of the inst currently in Writeback. 0 if not a valid
   inst.
  output reg [`INST_SIZE:0] trace_writeback_inst
);
```

The DatapathPipelined module is the top-level module that contains all the processing logic of the 5-layer RISC-V processor (Fetch, Decode, Execute, Memory, Writeback). This module does not contain actual memory but communicates with external memory modules through Input/Output ports.

Port Description

a. System Control Signals

- input clk: Synchronous clock signal for the entire system.
- input rst: Reset signal. When active, it returns the processor to the initial state (clears PC to 0, clears pipeline registers) to start execution from the beginning.

b. Communication with Instruction Memory Interface

These are the signals serving the Fetch stage:

- output `pc_to_imem`: Address of the instruction to be fetched (Program Counter). The processor sends this address to the instruction memory.
- input `inst_from_imem`: 32-bit instruction data is returned from the memory corresponding to the address `pc_to_imem`.

c. Communication with Data Memory Interface (Memory Stage)

These are the signals serving the Load/Store commands at the Memory layer:

- output `addr_to_dmem`: Data memory address to be retrieved (calculated from the ALU at the Execute layer).
- input `load_data_from_dmem`: Data read from memory (used for Load commands - LW, LH, LB,...).
- output `store_data_to_dmem`: Data to be written to memory (used for Store commands - SW, SH, SB).
- output `store_we_to_dmem`: Write Enable signal in 4-bit vector form. Each bit corresponds to allowing writing to 1 byte in a 32-bit word.

d. Debug & Trace Interface Signals

- output halt: Signal that the processor stops running (usually triggered when a special Environment Call or test termination condition is encountered).
- output `trace_writeback_pc` & `trace_writeback_inst`: Outputs the PC value and machine code (Instruction) of the instruction that is completing at the Writeback stage in the current cycle. This helps Testbench compare the actual CPU behavior with the sample trace-*.json file for scoring.

Listing 2.1: Opcode definition

```
// opcodes - see section 19 of RiscV spec
localparam [`OPCODE_SIZE:0] OpcodeLoad    = 7'b00_000_11;
localparam [`OPCODE_SIZE:0] OpcodeStore   = 7'b01_000_11;
localparam [`OPCODE_SIZE:0] OpcodeBranch  = 7'b11_000_11;
localparam [`OPCODE_SIZE:0] OpcodeJalr    = 7'b11_001_11;
localparam [`OPCODE_SIZE:0] OpcodeMiscMem = 7'b00_011_11;
localparam [`OPCODE_SIZE:0] OpcodeJal     = 7'b11_011_11;

localparam [`OPCODE_SIZE:0] OpcodeRegImm  = 7'b00_100_11;
localparam [`OPCODE_SIZE:0] OpcodeRegReg  = 7'b01_100_11;
localparam [`OPCODE_SIZE:0] OpcodeEnviron = 7'b11_100_11;

localparam [`OPCODE_SIZE:0] OpcodeAuipc   = 7'b00_101_11;
localparam [`OPCODE_SIZE:0] OpcodeLui     = 7'b01_101_11;
```

This is the definition of the logic fields based on the 7-bit opcode of the instruction.

Listing 2.2: Cycle counter

```
reg [`REG_SIZE:0] cycles_current;
always @(posedge clk) begin
  if (rst) begin
    cycles_current <= 0;
  end else begin
    cycles_current <= cycles_current + 1;
  end
end
```

Here is cycle counter

2.3.2.1 Hazard signals declaration

Listing 2.3: Hazard signals

```
wire stall_load_use; // 1 if find Load-Use Hazard
wire stall_div;      // 1 if waiting for divive op
wire flush_branch;   // 1 if wrong branch prediction (Taken Branch/
 Jump)
wire stall_all;
```

- `stall_load_use`: A flag signal that detects a case where the previous instruction is loading data from memory (Load), and the instruction immediately following needs to use that data. The pipeline must be stopped to wait for the data to come back from memory.
- `stall_div`: A signal that the divider is running. Because the multi-cycle divider (8 cycles) is slower than the main pipeline, the previous layers must be stopped to wait for the division result to be completed.
- `flush_branch`: A signal triggered when a branch or jump instruction is executed (Taken). Instructions that have been loaded into the pipeline via the wrong path (PC+4) must be flushed (Flush) and reloaded from the correct destination address.
- `stall_all`: A signal that combines all stall conditions. When this signal is active, the entire pipeline freezes (PC and all pipeline registers do not update).

2.3.2.2 Pipeline Registers declaration

Listing 2.4: Pipeline Registers

```
// IF/ID Registers
```

```
reg [`REG_SIZE:0] d_pc_current;
reg [`INST_SIZE:0] d_inst;

// ID/EX Registers
reg [`REG_SIZE:0] e_pc_current;
reg [`INST_SIZE:0] e_inst;
reg [`REG_SIZE:0] e_rs1_data, e_rs2_data, e_imm;
reg [4:0]         e_rs1, e_rs2, e_rd;
reg [2:0]         e_funct3;

// ID/EX Control Signals
reg       e_alu_src1; // 0: rs1, 1: pc (for AUIPC, JAL)
reg       e_alu_src2; // 0: rs2, 1: imm
reg [3:0] e_alu_op;
reg       e_mem_read, e_mem_write;
reg       e_reg_write, e_mem_to_reg; // 0: ALU, 1: MEM, 2: PC+4
reg       e_halt;
reg       e_is_branch, e_is_jump; // Use for jump - branch


// EX/MEM Registers
reg [`REG_SIZE:0] m_pc_current;
reg [`INST_SIZE:0] m_inst;
reg [`REG_SIZE:0] m_alu_result;
reg [`REG_SIZE:0] m_rs2_data;
reg [4:0]         m_rd;
reg [2:0]         m_funct3;
reg m_mem_read, m_mem_write, m_reg_write, m_mem_to_reg, m_halt;

// MEM/WB Registers
reg [`REG_SIZE:0] w_pc_current;
reg [`INST_SIZE:0] w_inst;
reg [`REG_SIZE:0] w_alu_result;
reg [`REG_SIZE:0] w_mem_data;
reg [4:0]         w_rd;
reg w_reg_write, w_mem_to_reg, w_halt;

// Forwarding Wires
wire [`REG_SIZE:0] wb_rd_data;
wire wb_we;
```

```
42  wire [4:0] wb_rd;
```

This is a set of status registers located between processing levels (Inter-stage Buffers). They act as "baffles" that store all the necessary information of an Instruction before passing it to the next level. This allows each level to process a separate instruction in parallel without mixing up data.

To manage the complexity of the pipeline design, the code follows a strict prefix naming convention corresponding to each processing stage, making it easy to track the instruction flow:

- d_* (Decode): IF/ID registers (Input to the Decode layer).
- e_* (Execute): ID/EX registers (Input to the Execute layer).
- m_* (Memory): EX/MEM registers (Input to the Memory layer).
- w_* (Writeback): MEM/WB registers (Input to the Writeback layer).

The `wb_*` wires are the Feedback wires from the Writeback layer back to the Decode/Execute layer.

2.3.2.3 Divider Tracking

Listing 2.5: Divider Tracking

```
1  // Shift register follow: [Rd, Valid, Opcode(Div/Rem), Signed, NegA,
       NegB]
2  reg [4:0] div_rd_pipe    [0:7];
3  reg       div_valid_pipe [0:7];
4  reg [2:0] div_info_pipe  [0:7]; // Bit 2: is_rem, Bit 1: a_neg, Bit
       0: b_neg
5  integer k;
6
7  // Logic Stall cho Divider
8  wire div_busy = div_valid_pipe[0] | div_valid_pipe[1] |
       div_valid_pipe[2] | div_valid_pipe[3] |
9                  div_valid_pipe[4] | div_valid_pipe[5] |
       div_valid_pipe[6] | div_valid_pipe[7];
```

This section implements an 8-stage pipeline tracker for division operations:

- `div_rd_pipe`: An array of 8 registers that store the destination register (Rd) for each division instruction currently in the divider pipeline.
- `div_valid_pipe`: An array of 8 flags indicating whether each stage in the divider pipeline is currently processing a valid division instruction.

- `div_info_pipe`: An array of 8 registers that store additional information about each division instruction, such as whether it is a division or remainder operation and the sign of the operands.

The `div_busy` wire checks if any stage in the divider pipeline is active, indicating that the divider is currently processing an instruction.

Listing 2.6: Instruction Decode

```
  // --- Inst Decode ---
wire [6:0] inst_opcode = d_inst[6:0];
wire [4:0] inst_rd     = d_inst[11:7];
wire [2:0] inst_funct3 = d_inst[14:12];
wire [4:0] inst_rs1    = d_inst[19:15];
wire [4:0] inst_rs2    = d_inst[24:20];
wire [6:0] inst_funct7 = d_inst[31:25];
```

This block extracts the main fields from the 32-bit instruction currently in the Decode layer (`d_inst`). These fields are essential for understanding the operation to be performed and for generating control signals for subsequent stages.

```
wire is_id_div = (inst_opcode == OpcodeRegReg) && (inst_funct7 == 7'
 b0000001) && (inst_funct3[2] == 1'b0);

reg div_raw_hazard;
always @(*) begin
    div_raw_hazard = 0;
    for(k=0; k<8; k=k+1) begin
        if (div_valid_pipe[k] && div_rd_pipe[k] != 0 &&
           (div_rd_pipe[k] == inst_rs1 || div_rd_pipe[k] == inst_rs2
 ))
           div_raw_hazard = 1;
    end
end

assign stall_div = div_raw_hazard;
assign stall_all = stall_load_use || stall_div || div_valid_pipe[6];
```

This block detects Read After Write (RAW) hazards related to division operations. It checks if the current instruction in the Decode layer is trying to read registers that are still being processed by the divider pipeline. If such a hazard is detected, it sets the `stall_div` signal to indicate that the pipeline should be stalled to wait for the division operation to complete.

2.3.2.4 Fetch Stage

Listing 2.7: Fetch Stage

```
reg  [`REG_SIZE:0] f_pc_current;
wire [`REG_SIZE:0] f_inst;

// signal Feedback from EX (Branch/Jump Target)
wire [`REG_SIZE:0] pc_target;
wire               pc_cond; // 1: Jump/Branch Taken, 0: PC+4


// program counter
// PC Update Logic
always @(posedge clk) begin
  if (rst) begin
    f_pc_current <= 32'd0;
  end
  else if (!stall_all) begin // just update if not stall
    if (pc_cond)
      f_pc_current <= pc_target; // jump
    else
      f_pc_current <= f_pc_current + 4; // default
  end
  // if Stall: keep current PC
end


// send PC to imem
assign pc_to_imem = f_pc_current;
assign f_inst = inst_from_imem;
```

This stage is responsible for managing the program counter (PC) and communicating with the Instruction Memory to retrieve the instructions to be executed for the next cycles. The always @(posedge clk) block describes the behavior of the PC register:

- Reset (rst): When the system restarts, the PC is reset to address 0.
- Stall mechanism (`!stall_all`): This is the key point to handle Hazard.
  - If the `stall_all` signal is active (due to Load-Use Hazard or Divider running), the PC update instruction block will be skipped.
  - Result: PC keeps the old value (Freezing the PC). This ensures that the current instruction is not lost and the processor does not load a new instruction until the Hazard is resolved.

- Select the next address (Next PC Logic):
  - Branch/Jump (if `pc_cond`): If the `pc_cond` signal (calculated from the Execute layer) is 1, the PC will update the new value to `pc_target` (target address of the jump instruction). This is the mechanism to change the execution flow.
  - Sequential (else): By default, the PC will be incremented by 4 (`f_pc_current + 4`) to point to the next 32-bit instruction in memory.

2.3.2.5 Decode Stage

Listing 2.8: Decode Stage

```
  // --- Update Reg IF/ID ---
always @(posedge clk) begin
  if (rst || flush_branch) begin // Flush when Jump
    d_pc_current <= 32'd0;
    d_inst       <= 32'd0; // NOP (0x00000013 is proper NOP but 0 works if handled)
  end
  else if (!stall_all) begin // Stall when Load-Use or Div, No stall take the next value
    d_pc_current <= f_pc_current;
    d_inst       <= f_inst;
  end
  // if Stall: Keep current decoding inst
end
```

This logic block manages the transfer of data from the Fetch layer to the Decode layer. It acts as a buffer (Latch), storing the newly fetched instruction (`f_inst`) and the address of that instruction (`f_pc_current`) on the rising edge of the clock, so that the Decode layer has stable data to process in the next cycle.

Listing 2.9: Instruction Decode

```
  // --- Inst Decode ---
wire [6:0] inst_opcode = d_inst[6:0];
wire [4:0] inst_rd     = d_inst[11:7];
wire [2:0] inst_funct3 = d_inst[14:12];
wire [4:0] inst_rs1    = d_inst[19:15];
wire [4:0] inst_rs2    = d_inst[24:20];
wire [6:0] inst_funct7 = d_inst[31:25];

```

```
// Imm Gen
wire [11:0] imm_i = d_inst[31:20];
wire [11:0] imm_s = {inst_funct7, inst_rd};
wire [12:0] imm_b = {inst_funct7[6], inst_rd[0], inst_funct7[5:0],
 inst_rd[4:1], 1'b0};
wire [20:0] imm_j = {d_inst[31], d_inst[19:12], d_inst[20], d_inst
 [30:21], 1'b0};

wire [`REG_SIZE:0] imm_i_sext = {{20{imm_i[11]}}, imm_i};
wire [`REG_SIZE:0] imm_s_sext = {{20{imm_s[11]}}, imm_s};
wire [`REG_SIZE:0] imm_b_sext = {{19{imm_b[12]}}, imm_b};
wire [`REG_SIZE:0] imm_j_sext = {{11{imm_j[20]}}, imm_j};
wire [`REG_SIZE:0] imm_u_sext = {d_inst[31:12], 12'b0}; // LUI,
 AUIPC
```

This instruction block performs the "dissection" of a 32-bit instruction string (`d_inst`) into its basic components based on the standard format of the RISC-V architecture. The `*_sext` variables perform sign bit (MSB) extension to the full 32 bits.

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Figure 2.4: RV32IM Instruction set

Listing 2.10: RegFile connection

```
// RegFile
wire [`REG_SIZE:0] rs1_data_raw, rs2_data_raw;
RegFile rf (
  .clk(clk),
  .rst(rst),
  .we(wb_we),
  .rd(wb_rd),
  .rd_data(wb_rd_data), // Write from WB
  .rs1(inst_rs1),
```

```
    .rs2(inst_rs2),                 // Read at ID
    .rs1_data(rs1_data_raw),
    .rs2_data(rs2_data_raw)
  );
```

Connect the RegFile module to the processor's data path. This is where the intersection of the two stages of the pipeline: Decode and Writeback occurs.

Connection details:

- Read Ports: Controlled by `inst_rs1` and `inst_rs2` from the currently decoded instruction. The read-out data (`rs1_data_raw`, (`rs2_data_raw`) will be sent to the pipeline registers to prepare for the Execute stage.
- Write Port: Controlled by signals from the Writeback stage (`wb_we`, `wb_rd`, `wb_rd_data`).

Listing 2.11: Control Unit & Hazard Detection

```
  // --- Control Unit & Hazard Detection ---
  reg        c_alu_src1; // 0: rs1, 1: pc
  reg        c_alu_src2; // 0: rs2, 1: imm
  reg [3:0]  c_alu_op;
  reg        c_mem_read, c_mem_write, c_reg_write, c_mem_to_reg, c_halt
   ;
  reg        c_is_branch, c_is_jump;
  reg [`REG_SIZE:0] c_imm;
```

Control Signals Declaration:

The reg ... c_* variables (such as `c_alu_op`, `c_mem_read`...) are intermediate variables containing control signals decoded from Opcode. They determine the behavior of the instruction at the following levels:

- `c_alu_*`: Controls computation at the Execute level.
- `c_mem_*`: Controls memory access at the Memory level.
- `c_reg_*`: Controls writeback at the Writeback level.

Listing 2.12: Hazard Detection Logic

```

// Hazard Detection Logic (Load-Use)
  assign stall_load_use = (e_mem_read) && ((e_rd == inst_rs1) || (e_rd
    == inst_rs2)) && (e_rd != 0);
```

Problem (Load-Use Hazard): Occurs when a Load instruction (read from memory) is at the Execute stage, and the instruction immediately following it (at the Decode stage) needs to use the result of this Load instruction as an input operand. Since the Load instruction must wait until the Memory stage to have data, the usual Forwarding technique (from EX to EX) cannot be applied because the data does not exist in the CPU yet.

- `stall_load_use`: This signal will be activated (level 1) when 3 conditions are satisfied at the same time:
- `e_mem_read`: The instruction preceding (at EX) is a memory read instruction (Load).
- `(e_rd == inst_rs1) || (e_rd == inst_rs2)`: The destination register of the Load instruction coincides with one of the two source registers of the instruction being decoded.
- `e_rd != 0`: Do not consider the case of writing to the x0 register

When `stall_load_use = 1`:

- Stall: This signal will activate `stall_all`, keeping the PC and the IF/ID register still so that the decoding instruction "stands still" waiting.
- Bubble: At the same time, a NOP (bubble) instruction will be inserted into the EX stage in the next cycle (because the ID/EX register update logic will notice the Stall and clear the control signals).

Listing 2.13: Control Unit Logic

```
always @(*) begin
  // Defaults (NOP)
  c_alu_src1 = 0;
  c_alu_src2 = 0;
  c_alu_op = 4'b0000;
  c_mem_read = 0;
  c_mem_write = 0;
  c_reg_write = 0;
  c_mem_to_reg = 0;
  c_halt = 0;
  c_is_branch = 0;
  c_is_jump = 0;
  c_imm = 32'd0;
```

The ... `c_*` intermediate variables will be assigned the default value of 0, and depending on the OpCode, will be assigned signals appropriate to that OpCode type, making it more convenient to

recognize and handle the behavior. To be safe, if the Opcode is invalid or unsupported, the CPU will not perform unauthorized register writes or memory writes. Now, we will go on with each OpCode cases.

Before talking about the code, why these signal have that value, we need to look back about the rv32im instructions set.

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Figure 2.5: RV32IM Instruction set

Listing 2.14: Control Unit Logic - R-Type Instructions

```
1    case (inst_opcode)
2      OpcodeRegReg: begin // R-Type
3        c_reg_write = 1;
```

```
4         if (inst_funct7 == 7'b0000001) c_alu_op = (inst_funct3[2]) ?
  4'b1011 : 4'b1010; // DIV / MUL
5         else begin // Standard ALU
6           case(inst_funct3)
7             3'b000: c_alu_op = (inst_funct7[5]) ? 4'b0001 : 4'b0000;
  // SUB/ADD
8             3'b001: c_alu_op = 4'b0010; // SLL
9             3'b010: c_alu_op = 4'b0011; // SLT
10            3'b011: c_alu_op = 4'b0100; // SLTU
11            3'b100: c_alu_op = 4'b0101; // XOR
12            3'b101: c_alu_op = (inst_funct7[5]) ? 4'b0111 : 4'b0110;
  // SRA/SRL
13            3'b110: c_alu_op = 4'b1000; // OR
14            3'b111: c_alu_op = 4'b1001; // AND
15          endcase
16        end
17      end
```

As we can see, the code will check the `inst_opcode` value to classify which command group it belongs to. For example: if opcode is 0110011, it will be in OpcodeRegReg group.

Go detail in each group, it will continue checking the `inst_func3` to to find out what command is being executed. Each command will have it's personal `inst_func3` based on the rv32im instruction set. For example: sll command is 3'b001 in `inst_func3`.

However, there will be some cases `inst_func3` of 2 or more commands are the same so we need to check the value `inst_funct7[5]` to modify it. For example: sra and srl have the same `inst_func3` but the different lied in the value of `inst_funct7`

The same technique to identify each command group, each command code is performed for other command groups, I will present the code below, I have noted a few points in the code so that you can understand them better.

Listing 2.15: Control Unit Logic - Other Instructions

```
1       OpcodeRegImm: begin // I-Type (ADDI...)
2         c_reg_write = 1;
3         c_alu_src2 = 1;
4         c_imm = imm_i_sext;
5         case(inst_funct3)
6             3'b000: c_alu_op = 4'b0000; // ADDI
7             3'b010: c_alu_op = 4'b0011; // SLTI
8             3'b011: c_alu_op = 4'b0100; // SLTIU
9             3'b100: c_alu_op = 4'b0101; // XORI
```

```
            3'b110: c_alu_op = 4'b1000; // ORI
            3'b111: c_alu_op = 4'b1001; // ANDI
            3'b001: begin
              c_alu_op = 4'b0010;
              c_imm = {27'd0, d_inst[24:20]};
            end // SLLI
            3'b101: begin
              c_alu_op = (inst_funct7[5]) ? 4'b0111 : 4'b0110;
              c_imm = {27'd0, d_inst[24:20]};
            end // SRAI/SRLI
        endcase
    end
    OpcodeLoad: begin
        c_reg_write=1;
        c_mem_read=1;
        c_mem_to_reg=1;
        c_alu_src2=1;
        c_imm=imm_i_sext;
    end
    OpcodeStore: begin
        c_mem_write=1;
        c_alu_src2=1;
        c_imm=imm_s_sext;
    end
    OpcodeBranch: begin
      c_is_branch = 1;
      c_alu_op = 4'b0001;
      c_imm = imm_b_sext; // ALU SUB for compare
      // Funct3 will be processed EX to define the jump
    end
    OpcodeJal: begin
      c_is_jump = 1;
      c_reg_write = 1;
      c_imm = imm_j_sext;
      c_alu_src1 = 1; // PC
      c_alu_src2 = 1; // Imm
      // JAL need: PC_next = PC + Imm. Rd = PC + 4.
      // Here we use ALU to calculate PC+Imm (for jump).
      // Write PC+4 to Rd will be process speacialy or use cla to
  sum.
```

```
        // Simplest: ALU calculate PC+Imm. Rd take PC+4.
      end
      OpcodeJalr: begin
        c_is_jump = 1;
        c_reg_write = 1;
        c_imm = imm_i_sext;
        c_alu_src2 = 1;
        c_alu_op = 4'b0000; // ADD (rs1 + imm)
      end
      OpcodeLui: begin
        c_reg_write = 1;
        c_imm = imm_u_sext;
        // LUI: Rd = Imm. Can use  ALU: A=0, B=Imm, Op=ADD.
        // Here we defined LUI is handled by direct assignment .
        c_alu_src2 = 1;
        c_alu_op   = 4'b1100;
      end
      OpcodeAuipc: begin
        c_reg_write = 1;
        c_alu_src1 = 1;
        c_alu_src2 = 1;
        c_alu_op = 4'b0000;
        c_imm = imm_u_sext;
      end
      OpcodeEnviron: if(d_inst[31:7] == 0) c_halt = 1;
    endcase
  end
```

2.3.2.6 Mux Inputs for ALU

Listing 2.16: Mux Inputs for ALU

```
    // --- MUX Inputs cho ALU ---
  reg [`REG_SIZE:0] alu_in_a_val, alu_in_b_val_fwd;
```

The code declares two registers (actually wires in the combinational circuit):

- `alu_in_a_val`: Actual value of operand A (source rs1) after going through the Forwarding selector.
- `alu_in_b_val_fwd`: Actual value of operand B (source rs2) after going through the Forwarding selector.

In the Pipeline architecture, the value read from the ID/EX register (`e_rs1_data`, `e_rs2_data`) may be outdated (stale data) because the previous instructions have not yet written the results to the Register File. These two variables act as the output of the Forwarding MUXes. They will contain the "freshest" value, selected from 3 sources:

- Old value from ID/EX (In case of no Hazard).
- Value transferred from the Memory layer (Forwarding from EX/MEM).
- Value transferred from the Writeback layer (Forwarding from MEM/WB).

2.3.2.7 Update Register ID/EX

This sequential logic block is responsible for transferring the entire instruction context (Context Transfer) from the Decode stage to the Execute stage on the rising edge of the clock. It decides whether the Execute stage will receive a new valid instruction, receive a null instruction (NOP), or keep the old state.

Listing 2.17: Update Register ID/EX

```
    // --- Update Reg ID/EX ---

  always @(posedge clk) begin
    if (rst || flush_branch || stall_load_use) begin // Flush or Stall
    Load-Use -> insert NOP to EX
      e_pc_current <= 0;
      e_inst <= 32'h13; // NOP
      e_imm <= 0;
      e_rd <= 0;
      e_rs1 <= 0;
      e_rs2 <= 0;
      e_funct3 <= 0;
      e_rs1_data <= 0;
      e_rs2_data <= 0;
      e_alu_src1 <= 0;
      e_alu_src2 <= 0;
      e_alu_op <= 0;

      e_mem_read <= 0;
      e_mem_write <= 0;
      e_reg_write <= 0;
      e_mem_to_reg <= 0;
```

```
22      e_halt <= 0;
23
24      e_is_branch <= 0;
25      e_is_jump <= 0;
26    end
27    else if (!stall_div) begin // Just update if it not stall by
    divide op
28      e_pc_current <= d_pc_current;
29      e_inst <= d_inst;
30      e_rs1_data   <= rs1_data_raw;
31      e_rs2_data   <= rs2_data_raw;
32      e_imm        <= c_imm;
33      e_rs1        <= inst_rs1;
34      e_rs2        <= inst_rs2;
35      e_rd         <= inst_rd;
36      e_funct3     <= inst_funct3;
37
38      e_alu_src1   <= c_alu_src1;
39      e_alu_src2   <= c_alu_src2;
40      e_alu_op     <= c_alu_op;
41      e_mem_read   <= c_mem_read;
42      e_mem_write  <= c_mem_write;
43      e_reg_write  <= c_reg_write;
44      e_mem_to_reg <= c_mem_to_reg;
45      e_halt       <= c_halt;
46      e_is_branch  <= c_is_branch;
47      e_is_jump    <= c_is_jump;
48    end
49    else begin
50      // when Stall, all commend before (WB/MEM) will run away.
51      // we need to capture the current value Forwarding into the reg
    for not being lost.
52      e_rs1_data <= alu_in_a_val;      // Updated by the value
    Forward
53      e_rs2_data <= alu_in_b_val_fwd;
54    end
55  end
56
```

The update logic is classified into 3 cases of decreasing priority:

Priority 1: Insert NOP (Bubble Insertion)
`if (rst || flush_branch || stall_load_use)`
Action: Clear all control registers and data to 0.

- rst: Reset the system.
- `flush_branch`: When there is a jump command (Branch Taken), the command in Decode is the wrong path command.
- `stall_load_use`: When there is a Load-Use conflict, the Decode layer is stopped. To ensure continuity, we must insert a "gap" (Bubble/NOP) into the Execute layer so that it does not execute the old command or execute a garbage command.

Priority 2: Normal Advance
`else if (!stall_div)`
Action: Copy values from Decode layer (d_*, c_*) to e_* registers.

- When Divider is not busy, pipeline operates normally. Control signals (`c_alu_op`, etc) and data (`rs1_data, imm`) are pushed to Execute layer for processing in the next cycle.

Priority 3: Divider Stall & Data Capture

- When encountering a division instruction, the Execute layer must stop (Stall) for 8 cycles to wait for the result.
- However, the following layers (Memory, Writeback) are not stopped but continue to run.
- If the division instruction is using data Forwarded from the MEM or WB layer, then in the next cycle, that data will disappear (because the instruction providing the data has finished running and left the pipeline). The input value of the divider will be incorrect in the next cycle.

Listing 2.18: Update Register ID/EX - Capture Forwarded Data

```
1
2     else begin
3         // when Stall, all commend before (WB/MEM) will run away.
4         // we need to capture the current value Forwarding into the reg
      for not being lost.
5         e_rs1_data <= alu_in_a_val;       // Updated by the value
     Forward
6         e_rs2_data <= alu_in_b_val_fwd;
7     end
8   end
9
```

This code implements a technique called “Capture” of data. Instead of keeping the old value in the register, it updates the register with the value that is currently appearing on the Forwarding wire.

This ensures that even if the data supply (previous instruction) has passed away, the data needed for the division has been safely stored in the ID/EX register for the divider to use consistently for 8 cycles.

2.3.2.8 Execute Stage

Listing 2.19: Execute Stage - Forwarding Unit

```
// ========================================
// 3. EXECUTE STAGE (EX)
// ========================================

// --- Forwarding Unit ---
reg [1:0] fwd_a, fwd_b;
always @(*) begin
  // Forward A
  if (m_reg_write && (m_rd != 0) && (m_rd == e_rs1)) fwd_a = 2'b10;
 // from MEM
  else if (w_reg_write && (w_rd != 0) && (w_rd == e_rs1)) fwd_a = 2'
 b01; // from WB
  else fwd_a = 2'b00;

  // Forward B
  if (m_reg_write && (m_rd != 0) && (m_rd == e_rs2)) fwd_b = 2'b10;
  else if (w_reg_write && (w_rd != 0) && (w_rd == e_rs2)) fwd_b = 2'
 b01;
  else fwd_b = 2'b00;
end

```

In a 5-stage pipeline architecture, the result of an instruction is only written to the Register File at the last stage (Writeback). However, subsequent instructions may need to use that result immediately at the Execute stage (i.e. 2 cycles earlier).

Without Forwarding, we have to wait (Stall) until the data is written. The Forwarding Unit solves this problem by taking ("stealing") data from the front pipeline registers (MEM or WB) and sending it back to the ALU input without waiting for the Register File to be written.

The test is based on 3 prerequisites for each layer (MEM or WB):

- `reg_write`: Does the instruction at that layer actually write data to the register? (For example,

BEQ or SW instructions do not write registers, so they are not forwarded).

- `rd != 0`: Is the destination register x0? (In RISC-V, x0 is always 0, never changed, so it is never forwarded from x0).
- `rd == e_rs1` (or `e_rs2`): Does the destination register of the previous instruction match the source register of the current instruction?

Why check MEM first (if), then WB later (else if)
Example scenario: Suppose we have the following code:

- `ADD x1, x2, x3` (Command 1 - Currently on WB) -> Write to x1
- `SUB x1, x4, x5` (Command 2 - Currently on MEM) -> Also write to x1
- `AND x6, x1, x7` (Command 3 - Currently on EX) -> Need to read x1

Analysis:

- The AND command needs the value of x1.
- Both Command 1 (in WB) and Command 2 (in MEM) write to x1.
- According to the sequential program logic, Command 3 should get the "latest" value from Command 2 (SUB), not the old value from Command 1 (ADD).

Listing 2.20: Execute Stage - Forwarding Logic

```
always @(*) begin
  case(fwd_a)
    2'b00: alu_in_a_val = e_rs1_data;
    2'b10: alu_in_a_val = m_alu_result;
    2'b01: alu_in_a_val = wb_rd_data;
    default: alu_in_a_val = e_rs1_data;
  endcase

  case(fwd_b)
    2'b00: alu_in_b_val_fwd = e_rs2_data;
    2'b10: alu_in_b_val_fwd = m_alu_result;
    2'b01: alu_in_b_val_fwd = wb_rd_data;
    default: alu_in_b_val_fwd = e_rs2_data;
  endcase
end
```

```
16
17  // Select final source (PC or rs1? Imm or rs2?)
18  wire [`REG_SIZE:0] alu_a = (e_alu_src1) ? e_pc_current :
     alu_in_a_val;
19  wire [`REG_SIZE:0] alu_b = (e_alu_src2) ? e_imm : alu_in_b_val_fwd;
20
21  // --- Instantiate ALU & Divider ---
22  wire [`REG_SIZE:0] alu_result_comb;
23
24
```

The code uses two case blocks to select the primary data source for operands A and B based on control signals from the Forwarding Unit.

- 2'b00 (Default): Select the original data from the ID/EX pipeline register (`e_rs1_data` / `e_rs2_data`). This is the case where there is no data conflict.
- 2'b10 (Forward from MEM): Select the result of the calculation just completed from the Memory layer (`m_alu_result`). This is the shortest bypass to handle data conflicts that occur with the previous instruction.
- 2'b01 (Forward from WB): Select data from the Writeback layer (`wb_rd_data`). This is the shortcut to handle conflicts with the instruction 2 cycles ago.

After Forwarding is resolved, the data continues through a second selection layer to determine the actual operand for the calculation, based on the type of instruction being executed (R-Type, I-Type, J-Type...).

- Operand a (alu_a):
  - If `e_alu_src1 = 1`: Select current PC. Used for instructions that need to calculate relative addresses such as AUIPC or JAL.
  - If `e_alu_src1 = 0`: Select rs1 register value (after Forwarding). Used for normal calculations.
- Operand b (alu_b):
  - If `e_alu_src2 = 1`: Select Immediate Value. Used for I-Type instructions (like ADDI), S-Type (calculate Store address), or Branch.
  - If `e_alu_src2 = 0`: Select rs2 Register Value (through Forwarding). Used for R-Type instructions (like ADD, SUB)

Finally, the code declares the wires to be ready to receive the results from the calculation modules:

- `alu_result_comb`: Results from the combinatorial ALU.

2.3.2.9 ALU for calculate

Listing 2.21: ALU Module Instantiation

```
// Module ALU
ALU alu_inst (
  .clk(clk),
  .rst(rst),
  .a_alu(alu_a),
  .b_alu(alu_b),
  .alu_op(e_alu_op),
  .funct3(e_funct3),
  .result(alu_result_comb));

wire [`REG_SIZE:0] final_ex_result = alu_result_comb;

```

The ALU code `alu_inst (...)` performs the embedding of the ALU module into the Execute layer's data path. This is where all arithmetic (Add, Subtract, Multiply, Divide) and logic (AND, OR, XOR, Shift) calculations take place.

- Inputs: The module receives two carefully prepared operands, `alu_a` and `alu_b` (after going through the Forwarding logic and source selection), along with the control signals `alu_op` and funct3 from the ID/EX register.
- Output: The calculation result is output through the result port and assigned to the output wire.

`final_ex_result` is the final result of the Execute stage. This `final_ex_result` signal will be the input to the EX/MEM pipeline register to transfer to the Memory stage in the next cycle (or used for Forwarding back for the following instructions).

2.3.2.10 Logic Divider Stall

Listing 2.22: Divider Stall Logic

```
 // --- DIVIDER LOGIC (MOVED TO DATAPATH) ---
wire is_div_op_ex = (e_alu_op == 4'b1011);
wire is_signed_ex = is_div_op_ex && (~e_funct3[0]); // Check bit 0
 funct3
wire is_rem_ex    = is_div_op_ex && (e_funct3[1]);  // Check bit 1
 funct3 (REM/REMU)

```

```
// Calculate ABS values for inputs
wire a_neg_ex = is_signed_ex && alu_in_a_val[31];
wire b_neg_ex = is_signed_ex && alu_in_b_val_fwd[31];
wire [31:0] div_in_a = a_neg_ex ? (~alu_in_a_val + 1) : alu_in_a_val
 ;
wire [31:0] div_in_b = b_neg_ex ? (~alu_in_b_val_fwd + 1) :
 alu_in_b_val_fwd;
wire [31:0] div_quot_out, div_rem_out;

```

This code section implements the input preprocessing logic for a pipelined hardware divider in the Execute stage. Since the divider module only operates on unsigned integers, this logic handles the conversion of signed operands to their absolute values while carefully tracking sign information needed to restore correct results later.

- `is_div_op_ex`: Detects if the current instruction is a division operation based on the ALU operation code.
- `is_signed_ex`: Determines if the division is signed by checking the funct3 bits.
- `is_rem_ex`: Determines if the operation is a remainder calculation (REM/REMU) based on funct3.
- `a_neg_ex` and `b_neg_ex`: Identify if the input operands are negative (for signed division).
- `div_in_a` and `div_in_b`: Compute the absolute values of the input operands for the divider module.

Listing 2.23: Divider Module Instantiation

```
DividerUnsignedPipelined divider(
  .clk(clk),
  .rst(rst),
  .stall(1'b0),
  .i_dividend(div_in_a),
  .i_divisor(div_in_b),
  .o_quotient(div_quot_out),
  .o_remainder(div_rem_out)
);

```

The code instantiates a pipelined hardware divider module within the Execute stage of the CPU pipeline. This module performs division and remainder calculations on unsigned integers.

- Inputs:
  - `clk` and `rst`: Standard clock and reset signals for synchronous operation.
  - `i_dividend` and `i_divisor`: The dividend and divisor inputs for the division operation, which are the absolute values of the original operands.
  - `stall`: A control signal to pause the divider operation if needed (set to 0 here, meaning no stall).
- Outputs:
  - `o_quotient`: The result of the division (quotient).
  - `o_remainder`: The result of the modulus operation (remainder).

Listing 2.24: Update Shift Register for Divider

```
// Update Shift Register for Divider
always @(posedge clk) begin
    if (rst) begin
        for(k=0; k<8; k=k+1) begin
            div_rd_pipe[k] <= 0;
            div_valid_pipe[k] <= 0;
            div_info_pipe[k] <= 0;
            end
    end else begin
        for(k=7; k>0; k=k-1) begin
            div_rd_pipe[k] <= div_rd_pipe[k-1];
            div_valid_pipe[k] <= div_valid_pipe[k-1];
            div_info_pipe[k] <= div_info_pipe[k-1];
        end
        if (!stall_div && is_div_op_ex) begin
            div_rd_pipe[0] <= e_rd;
            div_valid_pipe[0] <= 1'b1;
            div_info_pipe[0] <= {is_rem_ex, a_neg_ex, b_neg_ex}; // Store sign info
        end else begin
            div_rd_pipe[0] <= 0; div_valid_pipe[0] <= 0; div_info_pipe[0] <= 0;
        end
    end
end

```

This sequential logic block implements an 8-stage shift register to track the state of division operations in a pipelined CPU architecture. Since division is a multi-cycle operation, this mechanism helps manage the timing and data flow associated with division instructions.

Listing 2.25: Logic Prepare Div Result (Sign Restore)

```
  // Logic Prepare Div Result (Sign Restore)
wire is_rem_wb = div_info_pipe[7][2];
wire a_neg_wb  = div_info_pipe[7][1];
wire b_neg_wb  = div_info_pipe[7][0];

wire [31:0] quot_correct = (a_neg_wb ^ b_neg_wb) ? (~div_quot_out + 1) : div_quot_out;
wire [31:0] rem_correct  = (a_neg_wb) ? (~div_rem_out + 1) : div_rem_out;
wire [31:0] div_final_res = is_rem_wb ? rem_correct : quot_correct;

```

This combinational logic block handles the sign restoration of division results in a pipelined CPU architecture. Since the divider module operates on absolute values, this logic ensures that the final quotient and remainder results have the correct signs based on the original operands.

- `is_rem_wb`, `a_neg_wb`, `b_neg_wb`: Extract control information from the shift register to determine if the operation is a remainder calculation and the signs of the original operands.
- `quot_correct`: Restores the sign of the quotient based on the signs of the original operands. If the signs differ, the quotient is negated.
- `rem_correct`: Restores the sign of the remainder based on the sign of the dividend. If the dividend was negative, the remainder is negated.
- `div_final_res`: Selects either the corrected quotient or remainder as the final result based on whether the operation was a division or remainder calculation.

2.3.2.11 Branch Logic

Listing 2.26: Branch Logic

```
// --- Branch Logic ---
// Check jump condition based on ALU result (eg: SUB output 0 -> Equal)
wire is_equal = (alu_result_comb == 0); // for BEQ
wire is_less_signed  = alu_result_comb[31];  // for BLT (negative result)
```

```
5  wire is_less_unsigned = (alu_a[31] != alu_b[31]) ?
6                          (!alu_a[31]) :         // different sign: A
    =0(small), B=1(high)
7                          alu_result_comb[31];   // same sign:
    negative result -> A<B correct
8
9  reg branch_taken;
10 always @(*) begin
11    branch_taken = 0;
12    if (e_is_branch) begin
13       case(e_funct3)
14           3'b000: branch_taken = is_equal;          // BEQ
15           3'b001: branch_taken = !is_equal;         // BNE
16           3'b100: branch_taken = is_less_signed;    // BLT
17           3'b101: branch_taken = !is_less_signed;   // BGE
18           3'b110: branch_taken = is_less_unsigned;  // BLTU
19           3'b111: branch_taken = !is_less_unsigned; // BGEU
20           default: branch_taken = 0;
21       endcase
22    end
23 end
24
25 assign pc_cond = e_is_jump || branch_taken;
26 assign pc_target = (e_is_jump && !e_alu_src1) ? (alu_a + e_imm) & ~1
     : // JALR (rs1+imm)
27                   (e_pc_current + e_imm); // JAL/Branch (PC+Imm)
28
29 assign flush_branch = pc_cond; // if jump -> delete 2 command go in
    wrong
30
```

Branch Resolution and Target Address Calculation logic block. This is the part that determines the program's execution flow (Control Flow), allowing the program to execute for/while loops or if/else statements.

To decide whether to perform a Branch Taken or not, the processor relies on the result of the subtraction (SUB) from the ALU in the current cycle.

- `is_equal` (Zero Flag): If the result of subtracting A - B is 0, then A = B. Used for the BEQ (Branch if Equal) instruction.
- `is_less_signed` (Sign Flag): Get the sign bit of the subtraction result. If this bit is 1 (negative

number), then A < B (in the context of signed numbers), used for BLT (Less Than). On the contrary, if it is 0, then A > B, used for BGE.

- `is_less_unsigned` (Unsigned Comparison): Handling unsigned comparisons is more complicated because we cannot rely only on the normal subtraction result if the two numbers have different sign bits.
  - If `alu_a` and `alu_b` have different sign bits (`alu_a[31] != alu_b[31]`): Any number with sign bit 0 (positive number) is smaller than the number with sign bit 1 (in unsigned comparison, a larger number is considered a very large positive number).
  - If same sign bit: Based on subtraction result (`alu_result_comb[31]`), same as Sign flag case.

Branch Decision Unit: The always @(*) block acts as a condition decoder: Based on the rv32im instruction and funct3 code of the Branch instruction (stored in `e_funct3`), it selects the corresponding state flag to assign to the `branch_taken` signal.

For example:

- 3'b000 (BEQ): Select `is_equal`.
- 3'b001 (BNE): Select `!is_equal` .
- 3'b101 (BGE): Select `!is_less_signed` (Not less than → Greater than or equal).

Listing 2.27: Branch Logic

```
// --- Branch Logic ---
assign pc_cond = e_is_jump || branch_taken;
assign pc_target = (e_is_jump && !e_alu_src1) ? (alu_a + e_imm) & ~1
  : // JALR (rs1+imm)
                   (e_pc_current + e_imm); // JAL/Branch (PC+Imm)

assign flush_branch = pc_cond; // if jump -> delete 2 command go in
 wrong

```

If there is a jump or branch instruction, `pc_cond` will be updated to 1, this is a signal that a jump or branch instruction is being executed.

`pc_target` is address that PC needs to jump to. There are two calculation modes:

- JALR:
  - `e_is_jump`: Is a jump instruction (JAL or JALR).

  - `!e_alu_src1`: Source 1 is not pc. (In Decode: JAL uses PC, JALR uses rs1). So this is JALR.
  - Formula: `(alu_a + e_imm) & ~1`.
  - `& ~1`: An important specification of RISC-V is that the least significant bit (LSB) of the JALR destination address must always be cleared to 0 to ensure even address alignment.

- JAL & Branch
  - Formula: `e_pc_current + e_imm`.
  - Destination address is equal to current PC plus offset (Offset/Immediate).

When performing a jump at the Execute stage, the next instructions that were accidentally loaded into the Fetch and Decode stages were incorrect (because they were PC+4 instructions). This signal `flush_branch` is used to Flush (clear to 0) the IF/ID and ID/EX registers, removing those incorrect instructions

2.3.2.12 Update Register EX/MEM

Listing 2.28: Update Register EX/MEM

```
// --- MEMORY STAGE ---
always @(posedge clk) begin
  if (rst) begin
     m_pc_current <= 0;
     m_inst <= 0;
     m_alu_result <= 0;
     m_rs2_data <= 0;
     m_rd <= 0;
     m_funct3 <= 0;
     m_mem_read <= 0;
     m_mem_write <= 0;
     m_reg_write <= 0;
     m_mem_to_reg <= 0;
     m_halt <= 0;
  end
  else if (div_valid_pipe[7]) begin // DIV COMPLETE
     m_pc_current <= 0; m_inst <= 0; // Don't trace div here
     m_alu_result <= div_final_res;
     m_rd <= div_rd_pipe[7];
     m_reg_write <= 1;
```

```
        m_mem_read <= 0;
        m_mem_write <= 0;
        m_mem_to_reg <= 0;
        m_halt <= 0;
        m_rs2_data <= 0;
        m_funct3 <= 0;
    end
    else if (stall_all) begin // STALL
        m_mem_read <= 0;
        m_mem_write <= 0;
        m_reg_write <= 0;
        m_halt <= 0;
        m_pc_current <= 0;
        m_inst <= 32'h13;
        m_rd <= 0;
    end
    else begin // NORMAL
        m_pc_current <= e_pc_current; m_inst <= e_inst;
        m_alu_result <= (e_is_jump) ? e_pc_current + 4 :
  final_ex_result;
        m_rs2_data <= alu_in_b_val_fwd;
        m_rd <= e_rd;
        m_funct3 <= e_funct3;
        m_mem_read <= e_mem_read;
        m_mem_write <= e_mem_write;
        m_reg_write <= e_reg_write;
        m_mem_to_reg <= e_mem_to_reg;
        m_halt <= e_halt;
    end
  end

```

This sequential logic block updates the EX/MEM pipeline registers at the Memory stage of the CPU pipeline. It handles three main scenarios: Reset, Division Completion, and Normal Operation (with Stall handling). The update logic is classified into 4 cases of decreasing priority:

- *Reset Case*: When the reset signal is active, all pipeline registers are cleared to 0 to ensure a known initial state.
- *Division Completion Case*: Indicated by `div_valid_pipe[7]`, when a division operation completes, the pipeline registers are updated with the division result and relevant control signals.

The instruction and PC are set to 0 to avoid tracing the division operation in the pipeline.

- *Stall Case*: If a stall condition is detected (`stall_all`), the pipeline registers are set to neutral values to prevent any unintended operations during the stall.
- *Normal Operation Case*: Under normal conditions, the pipeline registers are updated with values

Listing 2.29: Update Register EX/MEM - Normal Operation

```
        // JAL/JALR need to write PC+4 to Rd.
        // Logic simpl: If Jump, write PC+4. If not, write ALU Result.
        m_alu_result <= (e_is_jump) ? e_pc_current + 4 :
    final_ex_result;

        m_rs2_data   <= alu_in_b_val_fwd; // Data Store (forwarded)

      end
    end

```

- For normal arithmetic/logic instructions: The result written to the destination register (rd) is the result from the ALU (`final_ex_result`).
- For Jump instructions (JAL/JALR): According to the RISC-V standard, the destination register rd needs to store the return address, i.e. PC + 4 (address of the next instruction if not jumping).
- With the Store (SW, SH, SB) command, we need to write the value of the rs2 register into memory.
- This rs2 value must be taken from the variable `alu_in_b_val_fwd` (which has gone through Forwarding at the EX layer).

Why is it not necessary to save `rs1_data` to the Memory layer?

Register rs1 (Source 1): Only used as an input operand for the ALU at the Execute layer (to add, subtract, or calculate the Base + Offset memory address). After the ALU has finished calculating and given the result (`alu_result`), the original value of rs1 has completed its task and is no longer needed for the following layers.

Register rs2 (Source 2): With ALU instructions (like ADD): Used as an operand, after EX it is no longer needed. But with the STORE instruction (SW, SH, SB): This instruction requires 2 elements:

- Address: Calculated by ALU (rs1 + imm).
- Data to be written: Exactly the value of rs2.

2.3.2.13 MEMORY STAGE (MEM)

Listing 2.30: Memory Stage - Store Logic

```
always @(*) begin
  // Default
  store_we_to_dmem   = 4'b0000;
  store_data_to_dmem = 32'd0;

  // The access address is always the result calculated from the ALU
  .
  addr_to_dmem = m_alu_result;

  // Only process if current command requires Memory Write
  // (m_mem_write == 1)
  if (m_mem_write) begin
    case (m_funct3)
      // SB (Store Byte) - write 8 bit
      3'b000: begin
         store_data_to_dmem = {4{m_rs2_data[7:0]}}; // Up to 32 bits
         store_we_to_dmem   = 4'b0001 << m_alu_result[1:0];
      end

      // SH (Store Halfword) - write 16 bit
      3'b001: begin
         store_data_to_dmem = {2{m_rs2_data[15:0]}};
         store_we_to_dmem   = 4'b0011 << {m_alu_result[1], 1'b0};
      end

      // SW (Store Word) - write 32 bit
      3'b010: begin
         store_data_to_dmem = m_rs2_data; // take all 32 bit
         store_we_to_dmem   = 4'b1111;    // write all 4 byte
      end

      default: begin
         store_we_to_dmem = 4'b0000;
      end
    endcase
  end
end
```

```
37
```

This combinational logic block is responsible for preparing the signal to write data to Data Memory. It converts RISC-V Store instructions (SB, SH, SW) into physical signals that the memory understands, including: address, aligned data, and write mask.

- Access address: Obtained directly from the ALU calculation result at the Execute layer (Base Address + Offset).
- The data memory uses this address to determine which 32-bit memory cell (Word) is being accessed.
- Since memory is organized by Word (32-bit), but RISC-V supports writing by Byte (8-bit) or Halfword (16-bit), we need the `store_we_to_dmem` (4-bit) mechanism to specify exactly which byte in that word will be changed.
  - Go detail in Store Byte (Sb):

    if `m_alu_result[1:0]` is 01, it will create a mask `store_we_to_dmem = 0010`, which mean store into the byte 1

    if `m_alu_result[1:0]` is 11, it will create a mask `store_we_to_dmem = 1000`, which mean store into the byte 3
  - Go detail in Store Byte (SH):

    if `m_alu_result[1]` is 1, it will create a mask `store_we_to_dmem = 1100`, which mean store into the byte 2 and 3.

    if `m_alu_result[1]` is 0, it will create a mask `store_we_to_dmem = 0011`, which mean store into the byte 0 and 1.
  - Go detail in Store Byte (SW): It will store without mask due to it have to write all 4 bytes

2.3.2.14 LOGIC LOAD

Listing 2.31: Load Logic

```
1  reg [`REG_SIZE:0] mem_data_fmt;
2
3  always @(*) begin
4    // Default to get whole word (for LW or commands that don't need
   to load)
5    mem_data_fmt = load_data_from_dmem;
6
7    case (m_funct3)
```

```
8       // LB (Load Byte Signed) - extend sign
9       3'b000: begin
10        case (m_alu_result[1:0])
11          2'b00: mem_data_fmt = {{24{load_data_from_dmem[7]}},
    load_data_from_dmem[7:0]};
12          2'b01: mem_data_fmt = {{24{load_data_from_dmem[15]}},
    load_data_from_dmem[15:8]};
13          2'b10: mem_data_fmt = {{24{load_data_from_dmem[23]}},
    load_data_from_dmem[23:16]};
14          2'b11: mem_data_fmt = {{24{load_data_from_dmem[31]}},
    load_data_from_dmem[31:24]};
15        endcase
16      end
17
18      // LH (Load Halfword Signed) - extend sign
19      3'b001: begin
20        case (m_alu_result[1])
21          1'b0: mem_data_fmt = {{16{load_data_from_dmem[15]}},
    load_data_from_dmem[15:0]};
22          1'b1: mem_data_fmt = {{16{load_data_from_dmem[31]}},
    load_data_from_dmem[31:16]};
23        endcase
24      end
25
26      // LW (Load Word) - take all 32 bit
27      3'b010: mem_data_fmt = load_data_from_dmem;
28
29      // LBU (Load Byte Unsigned) - extend number 0
30      3'b100: begin
31        case (m_alu_result[1:0])
32          2'b00: mem_data_fmt = {24'b0, load_data_from_dmem[7:0]};
33          2'b01: mem_data_fmt = {24'b0, load_data_from_dmem[15:8]};
34          2'b10: mem_data_fmt = {24'b0, load_data_from_dmem[23:16]};
35          2'b11: mem_data_fmt = {24'b0, load_data_from_dmem[31:24]};
36        endcase
37      end
38
39      // LHU (Load Halfword Unsigned) - extend number 0
40      3'b101: begin
41        case (m_alu_result[1])
```

```
            1'b0: mem_data_fmt = {16'b0, load_data_from_dmem[15:0]};
            1'b1: mem_data_fmt = {16'b0, load_data_from_dmem[31:16]};
          endcase
        end

        default: mem_data_fmt = load_data_from_dmem;
      endcase
    end


```

Although memory returns data in 32-bit words (Words), the Load instructions in RISC-V (LB, LH, LW, LBU, LHU) require extraction of smaller parts (Bytes or Halfwords) and different sign handling. This logic block performs the extraction and extending of the raw `load_data_from_dmem` data into the standard 32-bit `mem_data_fmt` format for writing to the registers.

Since memory is accessed by word address (4 byte aligned), the returned data is always a 32-bit block containing the requested address. To get the correct smaller piece of data, we rely on the low bits of the address (`m_alu_result`). The location access rules are similar to the store command described above.

- `m_alu_result[1:0]`: Used to determine the location of the Byte (0, 1, 2, or 3) in the word.
- `m_alu_result[1]`: Used to determine the location of the Halfword (Low Half or High Half

Instructions Processing Details:

- Signed Extension (LB, LH)

  For these instructions, the most significant bit (MSB) of the extracted data is considered the sign bit and needs to be "stretched" to the full 32 bits to preserve the arithmetic value (in 2's complement code).

- Zero Extension (LBU, LHU)

  For these instructions, the data is considered a primitive positive number. The extension simply fills in zeros.

2.3.2.15 Update Register MEM/WB

Listing 2.32: Update Register MEM/WB

```
// --- Update Register MEM/WB ---
always @(posedge clk) begin
```

```
    if (rst) begin
        w_pc_current <= 0;
        w_inst <= 0;
        w_alu_result <= 0;
        w_mem_data <= 0;
        w_rd <= 0;
        w_reg_write <= 0;
        w_mem_to_reg <= 0;
        w_halt <= 0;
    end else begin
        w_pc_current <= m_pc_current;
        w_alu_result <= m_alu_result;
        w_mem_data   <= mem_data_fmt;
        w_rd         <= m_rd;
        w_reg_write  <= m_reg_write;
        w_mem_to_reg <= m_mem_to_reg;
        w_halt       <= m_halt;
    end
  end

```

This sequential logic block is responsible for latching all the results that have been processed at the Memory layer to transfer to the Writeback layer. Here, the data has reached its "purest" form: the calculation results are finished, the memory data has been formatted correctly (Byte/Halfword, Signed/Unsigned).

- `w_mem_data <= mem_data_fmt`:

  This register does not store raw data from memory (`load_data_from_dmem`) but stores formatted data (`mem_data_fmt`) at the combinational logic of the previous Memory layer.

  This ensures that the Writeback layer only needs to write this value to the Register File without having to worry about sign extension or bit truncation.

- `w_alu_result <= m_alu_result`: Transmit the calculation result from the ALU (for R-Type, I-Type instructions) to continue.

- `w_rd <= m_rd`: Transmit the destination register address so that the WB layer knows where to write the data (x1, x2, ...).

The data in these `w_*` registers is also the data source for the farthest Forwarding path (Forwarding from the WB layer to the EX layer) that we mentioned in the previous sections.

Writeback Stage

Listing 2.33: Writeback Stage

```
assign wb_rd_data = (w_mem_to_reg) ? w_mem_data : w_alu_result;
assign wb_rd      = w_rd;
assign wb_we      = w_reg_write;

always @(*) begin
    trace_writeback_pc   = w_pc_current;
    trace_writeback_inst = 32'd0; // No trace inst
    halt                 = w_halt;
end

```

This `wb_rd_data` signal wire is the feedback loop that returns to the `rd_data` port of the RegFile module that was initialized at the beginning of the DatapathPipelined.v file. At the same time, it is also the data source for the Forwarding logic from the WB layer.

Control signal `w_mem_to_reg`:

- Level 1 (True): Select `w_mem_data`. This is the data read from memory (for Load instructions: LW, LH, LB...).
- Level 0 (False): Select `w_alu_result`. This is the calculation result from ALU (for R-Type, I-Type) or return address (for JAL/JALR).
- `wb_rd`: Address of the destination register to be written.
- `wb_we`: Write Enable signal.

  These two signals are connected directly to the control port of RegFile. At the next rising edge of the clk clock, if `wb_we` == 1 and `wb_rd` != 0, the `wb_rd_data` value will be officially saved to the register, completing the execution cycle of an instruction.

Listing 2.34: Writeback Stage - Trace and Halt Signals

```
always @(*) begin
    trace_writeback_pc   = w_pc_current;
    trace_writeback_inst = 32'd0; // No trace inst
    halt                 = w_halt;
end

```

These output ports help the external Testbench to monitor the CPU's operating status cycle-by-cycle tracing.

2.3.3. Module MemorySingleCycle

Listing 2.35: Module MemorySingleCycle

```
module MemorySingleCycle #(
    parameter NUM_WORDS = 512
) (
    input                    rst,                 // rst for both imem
    and dmem
    input                    clk,                 // clock for both
   imem and dmem
                                                 // The memory reads/
   writes on @(negedge clk)
    input      [`REG_SIZE:0] pc_to_imem,          // must always be
   aligned to a 4B boundary
    output reg [`REG_SIZE:0] inst_from_imem,      // the value at
   memory location pc_to_imem
    input      [`REG_SIZE:0] addr_to_dmem,        // must always be
   aligned to a 4B boundary
    output reg [`REG_SIZE:0] load_data_from_dmem, // the value at
   memory location addr_to_dmem
    input      [`REG_SIZE:0] store_data_to_dmem,  // the value to be
   written to addr_to_dmem, controlled by store_we_to_dmem
    // Each bit determines whether to write the corresponding byte of
   store_data_to_dmem to memory location addr_to_dmem.
    // E.g., 4'b1111 will write 4 bytes. 4'b0001 will write only the
   least-significant byte.
    input      [        3:0] store_we_to_dmem
);

  // memory is arranged as an array of 4B words
  reg [`REG_SIZE:0] mem_array[0:NUM_WORDS-1];

  // preload instructions to mem_array
  initial begin
    $readmemh("mem_initial_contents.hex", mem_array);
  end

  localparam AddrMsb = $clog2(NUM_WORDS) + 1;
  localparam AddrLsb = 2;
```

```
27
28    always @(negedge clk) begin
29      inst_from_imem <= mem_array[{pc_to_imem[AddrMsb:AddrLsb]}];
30    end
31
32    always @(negedge clk) begin
33      if (store_we_to_dmem[0]) begin
34        mem_array[addr_to_dmem[AddrMsb:AddrLsb]][7:0] <=
     store_data_to_dmem[7:0];
35      end
36      if (store_we_to_dmem[1]) begin
37        mem_array[addr_to_dmem[AddrMsb:AddrLsb]][15:8] <=
     store_data_to_dmem[15:8];
38      end
39      if (store_we_to_dmem[2]) begin
40        mem_array[addr_to_dmem[AddrMsb:AddrLsb]][23:16] <=
     store_data_to_dmem[23:16];
41      end
42      if (store_we_to_dmem[3]) begin
43        mem_array[addr_to_dmem[AddrMsb:AddrLsb]][31:24] <=
     store_data_to_dmem[31:24];
44      end
45      // dmem is "read-first": read returns value before the write
46      load_data_from_dmem <= mem_array[{addr_to_dmem[AddrMsb:AddrLsb]}];
47    end
48  endmodule
49
```

This is a module that simulates the system's Main Memory. In this architecture, it acts as a Unified Memory in terms of storage (only one mem array) but provides a Dual-Port interface (2 access ports).

- Port 1 (Instruction Port): Read-only, serves the Fetch layer to load instructions.
- Port 2 (Data Port): Read/Write, serves the Memory layer (Load/Store instructions).

2.3.4. Module Processor

Listing 2.36: Module Processor

```
/* This design has just one clock for both processor and memory. */
module Processor (
    input                  clk,
    input                  rst,
    output                 halt,
    output [ `REG_SIZE:0] trace_writeback_pc,
    output [`INST_SIZE:0] trace_writeback_inst
);

  wire [`INST_SIZE:0] inst_from_imem;
  wire [ `REG_SIZE:0] pc_to_imem, mem_data_addr, mem_data_loaded_value
   , mem_data_to_write;
  wire [         3:0] mem_data_we;

  // This wire is set by cocotb to the name of the currently-running
   test, to make it easier
  // to see what is going on in the waveforms.
  wire [(8*32)-1:0] test_case;

  MemorySingleCycle #(
      .NUM_WORDS(8192)
  ) memory (
    .rst                 (rst),
    .clk                 (clk),
    // imem is read-only
    .pc_to_imem          (pc_to_imem),
    .inst_from_imem      (inst_from_imem),
    // dmem is read-write
    .addr_to_dmem        (mem_data_addr),
    .load_data_from_dmem (mem_data_loaded_value),
    .store_data_to_dmem  (mem_data_to_write),
    .store_we_to_dmem    (mem_data_we)
  );

  DatapathPipelined datapath (
    .clk                   (clk),
```

```
35      .rst                    (rst),
36      .pc_to_imem             (pc_to_imem),
37      .inst_from_imem         (inst_from_imem),
38      .addr_to_dmem           (mem_data_addr),
39      .store_data_to_dmem     (mem_data_to_write),
40      .store_we_to_dmem       (mem_data_we),
41      .load_data_from_dmem    (mem_data_loaded_value),
42      .halt                   (halt),
43      .trace_writeback_pc     (trace_writeback_pc),
44      .trace_writeback_inst   (trace_writeback_inst)
45    );
46
47  endmodule
48
```

The Processor Module acts as the top-level module of the hardware design. It acts as a virtual motherboard, connecting the "Datapath" with the "Memory" to form a complete computer system.

2.3.5. Module ALU

Listing 2.37: Module ALU

```
module ALU (
  input                clk, rst,
  input [`REG_SIZE:0] a_alu,
  input [`REG_SIZE:0] b_alu,
  input [3:0]          alu_op,
  input [2:0]          funct3,
  output reg [`REG_SIZE:0] result
);

```

The ALU module acts as the "computational brain" of the microprocessor. It receives control signals (`alu_op`, `funct3`) and two 32-bit operands (`_alu`, `b_alu`), performs arithmetic/logic operations, and returns a single result. This design fully supports the RV32IM (Integer + Multiplication/Division) instruction set.

2.3.5.1 Basic Arithmetic & Logic block

Listing 2.38: Basic Arithmetic & Logic block

```
  // --- 1. Basic ALU Logic (ADD/SUB/LOGIC) ---
  wire [`REG_SIZE:0] sum;
  wire is_sub = (alu_op == 4'b0001) || (alu_op == 4'b0011) || (alu_op
   == 4'b0100);
  wire cin = (is_sub) ? 1'b1 : 1'b0;
  wire [`REG_SIZE:0] b_alu_inverted = (is_sub) ? ~b_alu : b_alu;

  cla cla_inst (
    .a      (a_alu),
    .b      (b_alu_inverted),
    .cin    (cin),
    .sum    (sum)
  );

    // Logic compare (SLT/SLTU)
  wire check_equal = (a_alu == b_alu);
  wire check_less_signed = ($signed(a_alu) < $signed(b_alu));
  wire check_less_unsigned = (a_alu < b_alu);

```

1. Addition and Subtraction (ADD/SUB)

- Instead of using two separate adders and subtractors, the module uses a single CLA (Carry Lookahead Adder) to optimize the area.
- Subtraction mechanism: Subtraction A - B is performed by adding 2's complement: `A + (~B) + 1`.
- The `is_sub` signal will invert the input bit B (`~b_alu`) and put bit 1 into cin (Carry In).

2. Comparison (Set Less Than - SLT/SLTU)

Principle: The comparison result is derived from the flags of the subtraction that I presented above.

Distinguishing Signs:

- SLT (Signed): Comparison based on signed values ($signed).
- SLTU (Unsigned): Comparison based on absolute values.

2.3.5.2 Multiplication Logic block

```
reg [63:0] multiply;
always @(*) begin
    case (funct3)
        3'b000: multiply = a_alu * b_alu;
            // MUL
        3'b001: multiply = ($signed(a_alu) * $signed(b_alu));
            // MULH
        3'b010: multiply = ($signed(a_alu) * $signed({1'b0, b_alu}))
 ;          // MULHSU
        3'b011: multiply = (a_alu * b_alu);
            // MULHU
        default: multiply = 64'd0;
    endcase
end

```

Multiplication modes (based on funct3):

- MUL (000): Take the low 32-bit of the result.
- MULH (001): Take the high 32-bit of the signed $\times$ signed.
- MULHSU (010): Take the high 32-bit of the unsigned $\times$ unsigned.
- MULHU (011): Take the high 32-bit of the unsigned $\times$ unsigned.

2.3.5.3 Output Mux

Finally, a large MUX (case `alu_op`) will select one of the results (Sum, Product, Quotient, Remainder, or Logic result) to output to the result gate.

Listing 2.39: Output Mux

```

// --- 4. Final Output MUX ---
always @(*) begin
  case (alu_op)
    4'b0000: result = sum;                       // ADD
    4'b0001: result = sum;                       // SUB
    4'b0010: result = a_alu << b_alu[4:0];       // SLL
    4'b0011: result = {31'd0, check_less_signed};   // SLT
    4'b0100: result = {31'd0, check_less_unsigned}; // SLTU
    4'b0101: result = a_alu ^ b_alu;             // XOR
    4'b0110: result = a_alu >> b_alu[4:0];       // SRL
    4'b0111: result = $signed(a_alu) >>> b_alu[4:0]; // SRA
    4'b1000: result = a_alu | b_alu;             // OR
    4'b1001: result = a_alu & b_alu;             // AND

    4'b1010: begin // MUL
        case (funct3)
           3'b000: result = multiply[31:0];       // MUL
           3'b001: result = multiply[63:32];      // MULH
           3'b010: result = multiply[63:32];      // MULHSU
           3'b011: result = multiply[63:32];      // MULHU
           default: result = 0;
        endcase
    end

    4'b1011: result = final_div_res;             // DIV/REM

    4'b1100: result = b_alu;                     // LUI/COPY_B
    default: result = 32'd0;
  endcase
end


```

2.3.6. Module DividerUnsignedPipelined

This module was implemented in lab 1 and lab 4, however I will reiterate the explanation here.

Base on the division algorithm presented in the software model (C code), we can implement a hardware module that performs unsigned integer division using a pipelined architecture. The division process is broken down into multiple stages, each responsible for one iteration of the division algorithm.

```c
int divide(int dividend, int divisor) {
    int quotient = 0;
    int remainder = 0;

    for (int i = 0; i < 32; i++) {
        remainder = (remainder << 1) | ((dividend >> 31) & 0x1);
        if (remainder < divisor) {
            quotient = (quotient << 1);
        } else {
            quotient = (quotient << 1) | 0x1;
            remainder = remainder - divisor;
        }
        dividend = dividend << 1;
    }

    return quotient;
}
```

Figure 2.6: Division Algorithm Overview

2.3.6.1 Module divu_1iter

Listing 2.40: Module divu_1iter

```
1 // ============================================================
2 // MODULE divu_1iter
3 // ============================================================
4 module divu_1iter (
5     input       [31:0] i_dividend,
6     input       [31:0] i_divisor,
```

```
    input       [31:0] i_remainder,
    input       [31:0] i_quotient,
    output      [31:0] o_dividend,
    output      [31:0] o_remainder,
    output      [31:0] o_quotient
);
    reg [31:0] remainder_r;
    reg [31:0] quotient_r;
    reg [31:0] dividend_r;

    always @(*) begin
        // Shift remainder left by 1 bit, bring MSB of dividend into LSB of remainder
        remainder_r = (i_remainder << 1) | ((i_dividend >> 31) & 1'b1);

        // compare remainder with divisor
        if (remainder_r < i_divisor) begin
            quotient_r = i_quotient << 1;       // Cannot subtract, shift 0 into quotient
        end else begin
            quotient_r = (i_quotient << 1) | 1'b1; // Can subtract, shift 1 into quotient
            remainder_r = remainder_r + (~i_divisor + 1'b1);; // Update remainder
        end

        dividend_r = i_dividend << 1; // Shift dividend preparing for next iteration
    end

    assign o_dividend  = dividend_r;
    assign o_remainder = remainder_r;
    assign o_quotient  = quotient_r;

endmodule
```

The `divu_1iter` module acts as the fundamental processing element. It implements exactly one step of the division algorithm described in the software model (C code).

Signal Interface

The module accepts the current state of the `dividend`, `divisor`, `remainder`, and `quotient` as inputs and produces updated values for the next stage.

Logic Description

The combinational logic within the module performs three specific operations corresponding to the standard long division steps:

1. **Shift and Append:** The temporary remainder is calculated by shifting the input remainder to the left by 1 bit and appending the Most Significant Bit (MSB) of the current dividend:

$$R_{temp} = (R_{in} \ll 1) \vee (D_{in}[31]) \tag{2.1}$$

2. **Compare and Update:** The module compares the temporary remainder (R_{temp}) with the divisor (D_{ivr}):
   - **Case $R_{temp} < D_{ivr}$:** The divisor cannot be subtracted. The quotient is shifted left (appending 0), and the remainder remains unchanged .
   - **Case $R_{temp} \geq D_{ivr}$:** The subtraction is possible. The quotient is shifted left and the LSB is set to 1. The new remainder is updated as $R_{temp} - D_{ivr}$.
3. **Prepare Next Dividend:** The dividend is shifted left by 1 bit to expose the next MSB for the subsequent iteration.

2.3.6.2 Module DividerUnsignedPipelined

Listing 2.41: Module DividerUnsignedPipelined

```
module DividerUnsignedPipelined (
    input              clk, rst, stall,
    input       [31:0] i_dividend,
    input       [31:0] i_divisor,
    output reg  [31:0] o_remainder,
    output reg  [31:0] o_quotient
);

    wire [31:0] w_dividend  [0:7][0:4];
    wire [31:0] w_remainder [0:7][0:4];
    wire [31:0] w_quotient  [0:7][0:4];

    // Pipeline registers store results between stages
    // There are 8 register stages (0 to 7)
```

```
    reg [31:0] r_dividend  [0:7];
    reg [31:0] r_remainder [0:7];
    reg [31:0] r_quotient  [0:7];
    reg [31:0] r_divisor   [0:7];

    genvar s, i; // s = stage (0-7), i = iteration (0-3)

    generate
        for (s = 0; s < 8; s = s + 1) begin : gen_stages

            // First, connect inputs to the stage
            if (s == 0) begin
                // First stage takes input from module inputs
                assign w_dividend[s][0]  = i_dividend;
                assign w_remainder[s][0] = 32'b0; // Initial remainder is 0
                assign w_quotient[s][0]  = 32'b0; // Initial quotient is 0
            end else begin
                // Subsequent stages take data from the previous stage's registers
                assign w_dividend[s][0]  = r_dividend[s-1];
                assign w_remainder[s][0] = r_remainder[s-1];
                assign w_quotient[s][0]  = r_quotient[s-1];
            end

            // 2. Instantiate 4 "divu_1iter" instances in series
            for (i = 0; i < 4; i = i + 1) begin : gen_iters
                divu_1iter u_iter (
                    .i_dividend  (w_dividend[s][i]),
                    .i_divisor   ((s == 0) ? i_divisor : r_divisor[s-1]),
                    .i_remainder (w_remainder[s][i]),
                    .i_quotient  (w_quotient[s][i]),

                    .o_dividend  (w_dividend[s][i+1]),
                    .o_remainder (w_remainder[s][i+1]),
                    .o_quotient  (w_quotient[s][i+1])
                );
            end
```

```
51
52              // 3. PIPELINE REGISTER LOGIC (STORE RESULTS AFTER 4
    ITERATIONS)
53              always @(posedge clk) begin
54                  if (rst) begin
55                      r_dividend[s]  <= 0;
56                      r_remainder[s] <= 0;
57                      r_quotient[s]  <= 0;
58                      r_divisor[s]   <= 0;
59                  end else begin
60                      // Store the final output (position 4) of the
    logic chain into registers
61                      r_dividend[s]  <= w_dividend[s][4];
62                      r_remainder[s] <= w_remainder[s][4];
63                      r_quotient[s]  <= w_quotient[s][4];
64
65                      // Pass divisor to the register of this stage
66                      if (s == 0)
67                          r_divisor[s] <= i_divisor;
68                      else
69                          r_divisor[s] <= r_divisor[s-1];
70                  end
71              end
72          end
73      endgenerate
74
75      // 4. OUTPUT OF MODULE IS OUTPUT OF FINAL STAGE REGISTER (Stage 7)
76      always @(*) begin
77          o_quotient  = r_quotient[7];
78          o_remainder = r_remainder[7];
79      end
80
81  endmodule
82
```

The `DividerUnsignedPipelined` module implements an 8-stage pipelined unsigned integer division unit. Each stage consists of 4 iterations of the division algorithm, allowing the module to process one division operation every clock cycle after the initial latency.

The Pipeline Registers

Listing 2.42: The Pipeline Registers

```
    wire [31:0] w_dividend  [0:7][0:4];
    wire [31:0] w_remainder [0:7][0:4];
    wire [31:0] w_quotient  [0:7][0:4];

    // Pipeline registers store results between stages
    // There are 8 register stages (0 to 7)
    reg [31:0] r_dividend  [0:7];
    reg [31:0] r_remainder [0:7];
    reg [31:0] r_quotient  [0:7];
    reg [31:0] r_divisor   [0:7];
```

To transform the division algorithm into an 8-stage pipeline, it is necessary to store the intermediate state of the computation between clock cycles. Arrays of 32-bit registers (reg) are declared for the dividend, remainder, and quotient corresponding to stages 0 through 7.

Crucially, a register array for the divisor (r_divisor) is also implemented. Although the divisor value remains constant for a single operation, it must propagate through the pipeline stages alongside the data. This ensures that at any given stage *N*, the logic has access to the correct divisor associated with the specific instruction currently being processed in that stage.

Combinational Logic Structure

Listing 2.43: Combinational Logic Structure

```
wire [31:0] w_dividend  [0:7][0:4]; // internal wire
    // ...
    genvar s, i;
    generate
        for (s = 0; s < 8; s = s + 1) begin : gen_stages
            // ... (see the after code)

            // run 4 times (iterations) connect sequential
            for (i = 0; i < 4; i = i + 1) begin : gen_iters
                divu_1iter u_iter (
                    .i_dividend  (w_dividend[s][i]),
                    // ... c
                    .o_dividend  (w_dividend[s][i+1]), // connect to
   the input of the next loop
                    // ...
                );
            end
        end
```

```
    endgenerate
```

The core arithmetic logic is constructed using a nested generate loop. To balance the trade-off between latency (total cycles) and throughput (frequency), the 32 required division iterations are distributed across 8 stages, with each stage performing 4 iterations combinatorially. Inside the gen_iters loop, four instances of the divu_1iter module are instantiated in a daisy-chain configuration. The output wires of iteration i are connected directly to the input wires of iteration $i+1$. This creates a short combinational path that executes 4 subtraction/shift steps within a single clock cycle.

Data Multiplexing (Input Selection)

Listing 2.44: Data Multiplexing (Input Selection)

```
  // First, connect inputs to the stage
if (s == 0) begin
                // take input from external
                assign w_dividend[s][0]  = i_dividend;
                assign w_remainder[s][0] = 32'b0;
                assign w_quotient[s][0]  = 32'b0;
            end else begin
                // take from the previous registers
                assign w_dividend[s][0]  = r_dividend[s-1];
                assign w_remainder[s][0] = r_remainder[s-1];
                assign w_quotient[s][0]  = r_quotient[s-1];
            end
```

This logic controls the data source for the beginning of each pipeline stage:

- Stage 0 (Initialization): For the first stage, the inputs are taken directly from the module's external ports (i_dividend, i_divisor). The remainder and quotient are explicitly initialized to zero.
- Stages 1-7 (Propagation): For all subsequent stages, the inputs are decoupled from external signals. Instead, they retrieve data from the pipeline registers of the preceding stage (r_dividend[s-1], etc.). This isolates the stages and ensures data flows sequentially through the pipeline.

Sequential Logic - Flip Flops

Listing 2.45: Sequential Logic - Flip Flops

```
if (s == 0) begin
```

```
            always @(posedge clk) begin
                if (rst) begin
                    r_dividend[s]  <= 0;
                    r_remainder[s] <= 0;
                    r_quotient[s]  <= 0;
                    r_divisor[s]   <= 0;
                end else begin
                    // Store the final output (position 4) of the
    logic chain into registers
                    r_dividend[s]  <= w_dividend[s][4];
                    r_remainder[s] <= w_remainder[s][4];
                    r_quotient[s]  <= w_quotient[s][4];

                    // Pass divisor to the register of this stage
                    if (s == 0)
                        r_divisor[s] <= i_divisor;
                    else
                        r_divisor[s] <= r_divisor[s-1];
                end
            end
```

This block represents the temporal boundary of the pipeline. At every positive edge of the clock, the computed results from the combinational logic (specifically, the output of the 4th iteration in the chain) are captured into the pipeline registers.

- Critical Path Reduction: By inserting these registers, the long critical path of a 32-bit divider is broken into 8 smaller segments. The timing constraint is now determined only by the delay of 4 iterations plus register setup time, allowing the system to achieve a significantly higher maximum clock frequency (F_{max}).
- Non-blocking Assignments: The use of non-blocking assignments (<=) ensures that all pipeline stages update synchronously without race conditions.

Output Logic

Listing 2.46: Output Logic

```
    always @(*) begin
        o_quotient  = r_quotient[7];
        o_remainder = r_remainder[7];
    end
```

The final outputs of the module, o_quotient and o_remainder, are driven directly by the registers of the final stage (Stage 7). This ensures that the valid result is presented to the datapath exactly 8 clock cycles after the input was sampled, consistent with the pipeline depth.

2.3.7. Module CLA

This module was implemented in lab 2, however I will reiterate the explanation here.

The gp4 module processes 4-bit chunks of data. Instead of waiting for the carry to ripple through each bit, this module calculates the internal carries (c_1, c_2, c_3) simultaneously using Look-ahead logic equations.

Reasoning: To speed up calculation, we compute the internal carries (c_1, c_2, c_3) and the group-level signals (G_{out}, P_{out}) simultaneously using "flattened" logic equations, rather than waiting for a ripple effect.

Listing 2.47: module module gp4

```
module gp4(input wire [3:0] gin, pin,
           input wire cin,
           output wire gout, pout,
           output wire [2:0] cout,
           output wire [3:0] sum
           );

   wire [3:0] g, p;

   genvar i;

   generate
      for (i = 0; i < 4; i = i + 1)
      begin
         gp1 gp(
            .a(gin[i]),
            .b(pin[i]),
            .g(g[i]),
            .p(p[i])
         );
      end
   endgenerate

   assign cout[0] = g[0] | p[0] & cin;
   assign cout[1] = g[1] | p[1] & cout[0];
   assign cout[2] = g[2] | p[2] & cout[1];

   assign gout = g[3]
               | p[3] & g[2]
```

```
                | p[3] & p[2] & g[1]
                | p[3] & p[2] & p[1] & g[0];
  assign pout = &p;

  assign sum[0] = p[0] ^ cin;
  assign sum[1] = p[1] ^ cout[0];
  assign sum[2] = p[2] ^ cout[1];
  assign sum[3] = p[3] ^ cout[2];

endmodule
```

Interpretation:

1. **Instantiation:** A loop instantiates four gp1 modules to calculate bit-wise g and p signals from inputs gin (Input A) and pin (Input B).
2. **Internal Carry Calculation:** The carries cout[0..2] are calculated using flattened logic. For example, cout[1] depends on g_1 OR (p_1 AND the logic for c_0).
3. **Group Signals:**
   - gout: Represents the Look-ahead generation for the whole 4-bit block. It logic is: $G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$.
   - pout: Represents the group propagation ($P_3 \cdot P_2 \cdot P_1 \cdot P_0$).
4. **Sum Calculation:** The final sum is computed as $S_i = P_i \oplus C_i$.

2.3.7.1 8-bit Look-ahead Group (gp8)

The gp8 module is an 8-bit adder that uses Carry Look-ahead (CLA) technology. Instead of performing calculations on each bit directly, it combines two gp4 (4-bit CLA) submodules together. This is a typical example of Hierarchical Design.

Listing 2.48: module module gp8

```
module gp8(input wire [7:0] gin, pin,
           input wire cin,
           output wire gout, pout,
           output wire [6:0] cout,
           output wire [7:0] sum
           );

  wire gout_lower, pout_lower;
```

```
    wire gout_upper, pout_upper;

    gp4 lower_gp4(
        .gin(gin[3:0]),
        .pin(pin[3:0]),
        .cin(cin),
        .gout(gout_lower),
        .pout(pout_lower),
        .cout(cout[2:0]), // c1, c2, c3
        .sum(sum[3:0])
    );

    wire c4;
    assign c4 = gout_lower | (pout_lower & cin);
    assign cout[3] = c4;

    gp4 upper_gp4(
        .gin(gin[7:4]),
        .pin(pin[7:4]),
        .cin(c4),
        .gout(gout_upper),
        .pout(pout_upper),
        .cout(cout[6:4]), // c5, c6, c7
        .sum(sum[7:4])
    );

    assign gout = gout_upper | (pout_upper & gout_lower);
    assign pout = pout_upper & pout_lower;

endmodule
```

Interpretation:

Listing 2.49: module module gp8

```
gp4 lower_gp4(
        .gin(gin[3:0]), .pin(pin[3:0]), // process 4 low bit (0-3)
        .cin(cin),                      // take external carry
        .gout(gout_lower),              // Gen G lower
        .pout(pout_lower),              // Gen P lower
        .cout(cout[2:0]),               // Calculate c1, c2, c3
        .sum(sum[3:0])                  // Sum of 4 lower bit
```

```
8     );
```

Explanation: This block processes the first 4 bits. The gut_low and pout_low signals indicate whether this entire 4-bit block should generate memory (Create) or propagate memory (Propagate).

Listing 2.50: module module gp8

```
1 wire c4;
2     assign c4 = gout_lower | (pout_lower & cin);
3     assign cout[3] = c4;
```

Explanation: This is the key point connecting the two 4-bit blocks. Instead of waiting for c_3 to propagate, we immediately calculate c_4 based on the group signal.

Formula: $C_4 = G_{lower} + (P_{lower} \cdot C_{in})$.

- If the low group generates a carry ($G_{lower} = 1$) $\rightarrow$ There is a carry to the high group.
- If the low group transmits a carry ($P_{lower} = 1$) AND has an input carry ($C_{in} = 1$) $\rightarrow$ There is a carry to the high group.

Listing 2.51: module module gp8

```
1 gp4 upper_gp4(
2         .gin(gin[7:4]), .pin(pin[7:4]),
3         .cin(c4),                          // Input Carry is c4 calculate
    above
4         .gout(gout_upper),
5         .pout(pout_upper),
6         .cout(cout[6:4]),                  // Calculate c5, c6, c7
7         .sum(sum[7:4])
8     );
```

Explanation: This block processes the next 4 bits. The important point is that its input .cin is concatenated with c4. Because c4 is look-ahead, this block can start computing earlier than the Ripple Carry type.

Listing 2.52: module module gp8

```
1 assign gout = gout_upper | (pout_upper & gout_lower);
2     assign pout = pout_upper & pout_lower;
```

Purpose: So that higher level modules (e.g. 32-bit cla modules) can treat this gp8 block as a "giant bit". It needs to know whether this 8-bit block generates a carry or transmits a carry.

Explanation of Propagate 8-bit logic:

- To propagate a carry from input C_0 through all 8 bits out, BOTH the lower block (pout_lower) AND the higher block (pout_upper) must allow propagation.

- Formula: $P_{8bit} = P_{upper} \cdot P_{lower}$.

Explanation of Generate 8-bit logic: An 8-bit block will generate a carry out if:

- The higher block itself generates a carry (G_{upper}).
- OR: The lower block generates a carry (G_{lower}) and the higher block allows propagation of that carry (P_{upper}).
- Formula: $G_{8bit} = G_{upper} + (P_{upper} \cdot G_{lower})$.

2.3.7.2 Top-level 32-bit Adder (`cla`)

The final `cla` module integrates four `gp8` blocks to process 32-bit operands.

Design Improvement: In the final implementation, we strictly avoided the ripple-carry approach between the 8-bit blocks. Instead of daisy-chaining the carries, we implemented a Look-ahead unit logic at the top level to calculate boundary carries (C_8, C_{16}, C_{24}) in parallel.

Listing 2.53: module module cla

```
module cla
  (input wire [31:0]  a, b,
   input wire         cin,
   output wire [31:0] sum);

   // DO YOUR CODE HERE
   // Declare sinals Generate/Propagate for 4 blocks 8-bit
   wire gout0, pout0; // from bits [7:0]
   wire gout1, pout1; // from bits [15:8]
   wire gout2, pout2; // from bits [23:16]
   wire gout3, pout3; // from bits [31:24]

   // Declare signals Carry Look-ahead
   wire c8, c16, c24;

   // Internal cout wires from gp8 (not necessarily used at top-level,
    but needed to connect ports)
   wire [6:0] cout_dummy0, cout_dummy1, cout_dummy2, cout_dummy3;

   // --- Block 0 (Bits 0-7) ---
   gp8 gp8_0(
      .gin(a[7:0]),
      .pin(b[7:0]),
```

```
        .cin(cin),
        .gout(gout0),
        .pout(pout0),
        .cout(cout_dummy0),
        .sum(sum[7:0])
    );

    // --- Calculate C8 (Carry Look-ahead) ---
    // Formula: C1 = G0 | (P0 & C0)
    assign c8 = gout0 | (pout0 & cin);

    // --- Block 1 (Bits 8-15) ---
    gp8 gp8_1(
        .gin(a[15:8]),
        .pin(b[15:8]),
        .cin(c8),
        .gout(gout1),
        .pout(pout1),
        .cout(cout_dummy1),
        .sum(sum[15:8])
    );

    // --- Calculate C16 (Carry Look-ahead) ---
    // Formula: C2 = G1 | (P1 & G0) | (P1 & P0 & C0)
    // This carry is calculated immediately WITHOUT waiting for c8 from
     gp8_1 block
    assign c16 = gout1 | (pout1 & gout0) | (pout1 & pout0 & cin);

    // --- Block 2 (Bits 16-23) ---
    gp8 gp8_2(
        .gin(a[23:16]),
        .pin(b[23:16]),
        .cin(c16),
        .gout(gout2),
        .pout(pout2),
        .cout(cout_dummy2),
        .sum(sum[23:16])
    );

    // --- Calculate C24 (Carry Look-ahead) ---
```

```
62    // Formula: C3 = G2 | (P2 & G1) | ...
63    assign c24 = gout2 | (pout2 & gout1) | (pout2 & pout1 & gout0) | (
      pout2 & pout1 & pout0 & cin);
64
65    // --- Block 3 (Bits 24-31) ---
66    gp8 gp8_3(
67        .gin(a[31:24]),
68        .pin(b[31:24]),
69        .cin(c24),
70        .gout(gout3),
71        .pout(pout3),
72        .cout(cout_dummy3),
73        .sum(sum[31:24])
74    );
75
76 endmodule
```

Interpretation: Instead of connecting the output carry of gp8_0 to the input of gp8_1, we calculate the boundary carries (c_8, c_{16}, c_{24}) in parallel. This utilizes the "Windowed Generate/Propagate" abstraction:

- $C_8 = G_0 + P_0 C_{in}$
- $C_{16} = G_1 + P_1 G_0 + P_1 P_0 C_{in}$
- $C_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$

This ensures that the Most Significant Byte (bits 24-31) can begin calculation almost as quickly as the Least Significant Byte, as it does not wait for a ripple chain.

3

SIMULATION

3.1. Simulation

The test environment was built to verify the correctness of the 5-stage Pipelined Processor design (Fetch, Decode, Execute, Memory, Writeback) with an integrated unsigned/signed integer divider (Multi-cycle Pipelined Divider).

- Input: The program file `mem_initial_contents.hex` contains the machine code for the RISC-V instructions.
- Simulation Components:
  - Clock/Reset: The system operates synchronously with a clock signal with a cycle of 10ns (or 20ns depending on the settings; in this case, it's the rising edge at even intervals).
  - Memory: Single-cycle memory model (Instruction and data memory share the same address space).
  - Processor: DatapathPipelined module connected to DividerUnsignedPipelined.

Observe important internal signals (PC, ALU Result, Register File Writeback, Stall signals) on a waveform and compare them with theoretical results.

3.1.1. Test Scenarios

This test program is designed to activate the most complex control logic in the pipeline:

- Forwarding Logic (Bypassing): Tests the ability to push data from the MEM or WB layer to the EX layer to resolve data hazards without inserting a NOP instruction.
- Pipelined Divider Operation: Tests the operation of the 8-stage divider (8-stage pipeline).

- Verifies the stability of the input data buffering.
- Ensures the accuracy of the division algorithm over 32 iterations.
- Hazard Detection & Stall: Tests the collision detection system to stop (stall) the pipeline when the next instruction needs to use the result from the divider before completion (RAW Hazard).
- Tests the correctness of branch and jump instructions.

3.1.2. Waveform Simulation

Based on the hex file used to run the waveform simulation, I obtained the following results.

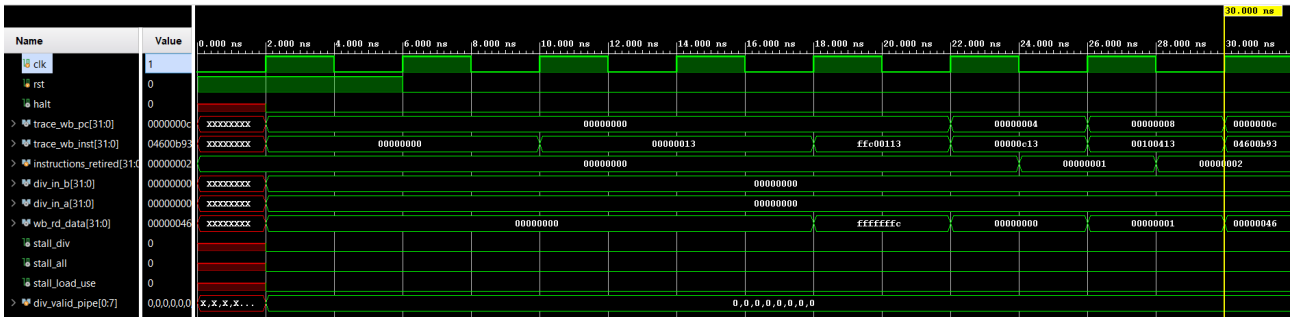

Figure 3.1: Waveform simulation from 0 - 30ns

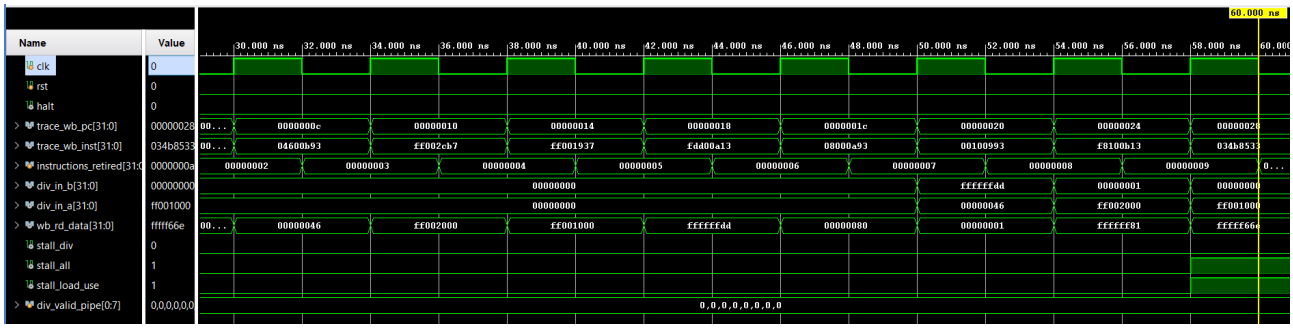

Figure 3.2: Waveform simulation from 30 - 60ns

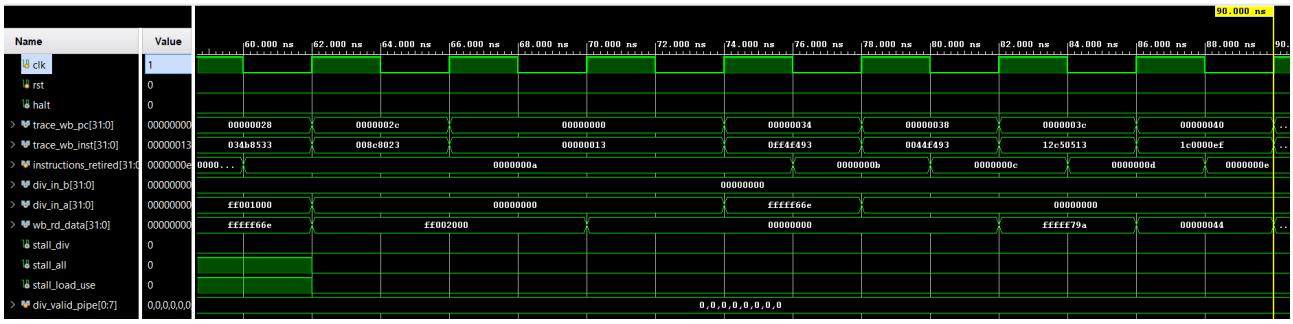

Figure 3.3: Waveform simulation from 60 - 90ns

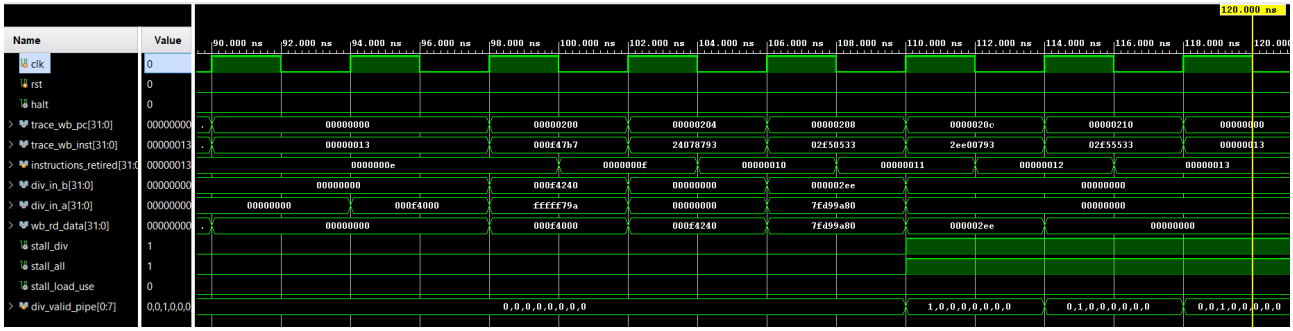

Figure 3.4: Waveform simulation from 90 - 120ns

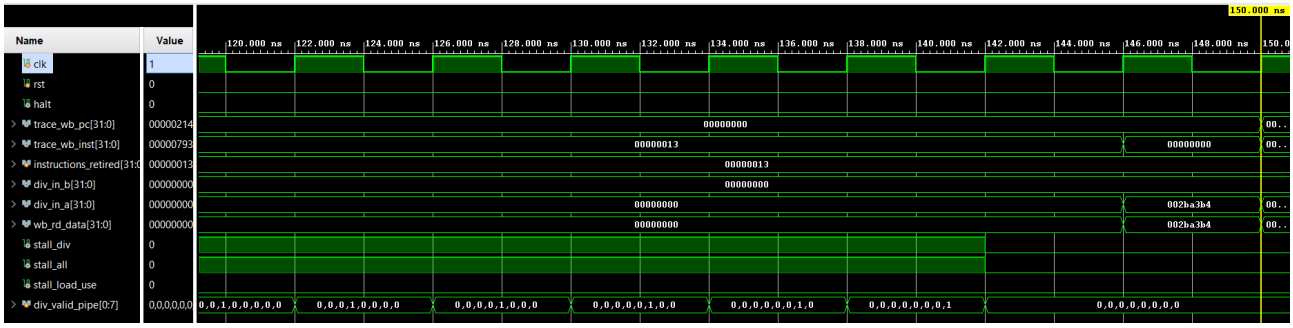

Figure 3.5: Waveform simulation from 120 - 150ns

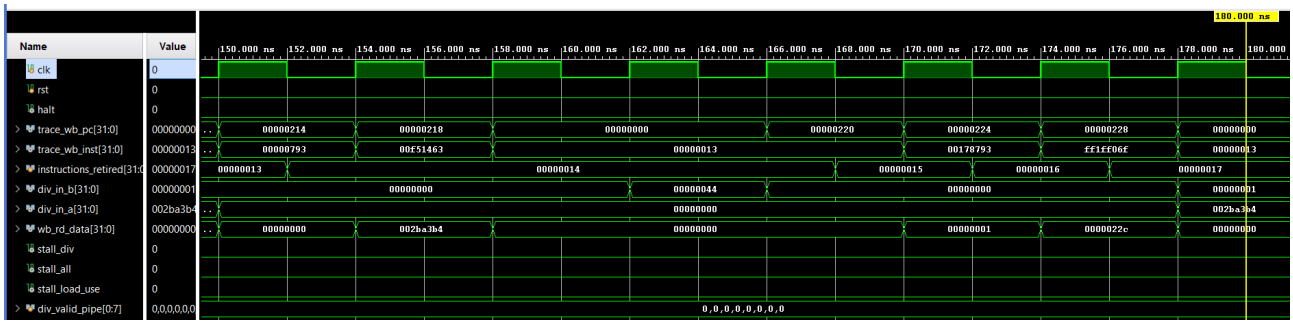

Figure 3.6: Waveform simulation from 150 - 180ns

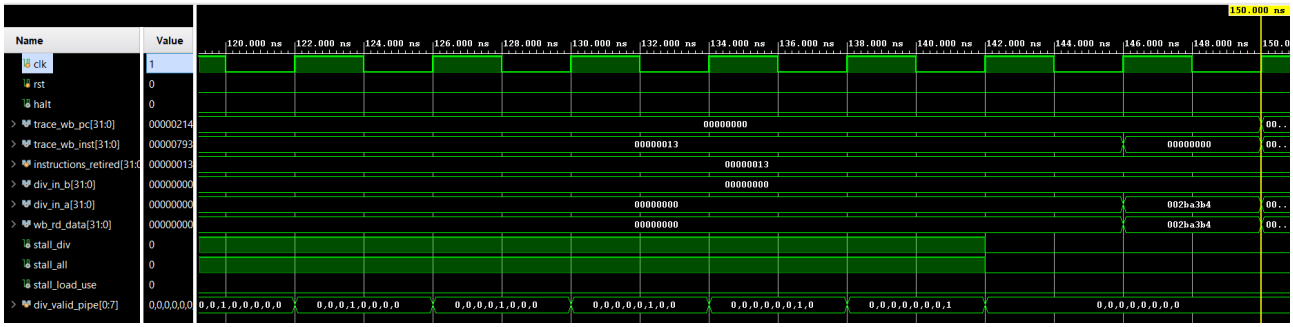

Figure 3.7: Waveform simulation from 180 - 210ns

The waveform diagram illustrates the execution process of a 5-stage Pipelined microprocessor through its main operating states: single-cycle instruction processing, multi-cycle instruction processing (division), and the stall mechanism to resolve data conflicts.

1. Normal Pipeline Operation Phase:

During the initial period, the microprocessor executes a series of basic arithmetic and logic instructions.

- The value of the `trace_wb_pc` register increases steadily with each clock cycle.
- Instructions pass through the 5 pipeline stages without collisions. The `stall_all` signal remains low (0). The resulting data (`wb_rd_data`) is continuously updated at each cycle, showing a maximum throughput of 1 instruction/cycle (CPI = 1).

For example:

- Time 0ns - 40ns: The addi instructions are executed to load values into sp, s0, s7, s4.
- Time 58ns - 60ns (Forwarding Check):

- The CPU executes the add a0, s7, s4 instruction (Adds s7 and s4).
- Status: s7 = 1, s4 = -35 (Hex: 0xFFFFFFDD).
- Result: The `wb_rd_data` signal records the value 0xFFFFFFDE (-34).

2. Multi-cycle Instruction Issue Phase:

When a complex calculation instruction (division) enters the Execute stage.

At the rising edge of the clock pulse, the input operands are immediately latched into the internal register of the divider. This ensures data stability even if the signal on the line (`div_in_b`) is noisy or changes state mid-clock cycle.

The `div_valid_pipe` signal is activated, marking the start of the instruction entering the divider's separate processing pipeline (Divider Pipeline). The microprocessor does not stop immediately but continues to load subsequent instructions, taking advantage of parallelism.

For example:

Time 110ns: The divide (div) instruction enters the Execute layer.

- Input: `div_in_a = 0x7fd99a80` (Dividend), `div_in_b = 0x000002ee` (Divisor).
- Event: `The div_valid_pipe[0]` signal goes to level 1 and then it count for 8 cycles.
- Result: `wb_rd_data` take value `0x2ba3b4` $\Rightarrow$ Right value

3. Hazard Detection & Stall Phase:

As soon as the controller detects the next instruction that needs to use the result of the ongoing division (Read-After-Write conflict):

- Stall Activated: The `stall_all` signal changes from low to high.
- The entire state of the microprocessor freezes. The program counter (PC) stops incrementing, and instructions in the pipeline remain in their current state to wait for the calculation result to complete. This is a mechanism to ensure data integrity.

For example:

- At 120ns - 136ns: The pipeline continues loading the next instructions. The `stall_all` signal remains at 0 (the CPU does not freeze immediately), taking advantage of the pipeline nature of the divider.
- At $\approx 136ns$ (Stall Activation): The CPU detects that the instruction at the Decode stage needs to use the result of the division (currently in the Divider Pipeline). Then the `stall_all` signal rises to 1. The entire PC and pipeline registers freeze to wait for the result.

4. Completion & Writeback Phase: After a fixed latency corresponding to the number of stages in the divider:

- The calculation is complete, and a valid result appears on the `wb_rd_data` data line.
- The `stall_all` signal returns to a low level. The microprocessor exits the waiting state, writes the result to the Register File, and immediately continues executing the next instructions.

4

STATIC TIMING ANALYSIS

4.1. Summary

Next, I proceeded to check the design timing and LUT usage of the pipelined datapath that I had implemented.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-10.357 ns	Worst Hold Slack (WHS):	0.100 ns	Worst Pulse Width Slack (WPWS):	3.020 ns
Total Negative Slack (TNS):	-6139.075 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	1075	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4105	Total Number of Endpoints:	4105	Total Number of Endpoints:	2243

Timing constraints are not met.

Figure 4.1: Design Timming summary at 125 MHz

The synthesis and analysis of timing revealed that the current design does not meet the timing constraints. However, the design functions logically correctly, as verified through Waveform simulation.

4.1.1. Detailed analysis of parameters

A. Setup Time - Fail This is the most important indicator determining the maximum operating frequency of the microprocessor.

- Worst Negative Slack (WNS): -10.357 ns. The longest signal path (Critical Path) in the circuit is running slower than required by more than 10 ns. This is a very large violation.

- Total Negative Slack (TNS): -6139.075 ns. The total violation time of all paths combined.
- Number of Failing Endpoints: 1075. There are 1075 signal paths that failed to stabilize before the next clock edge.

B. Hold Time - Pass

- Worst Hold Slack (WHS): 0.100 ns. Assessment: No Hold Time (Race condition) errors. This indicates that the clock tree is functioning correctly and there are no excessively short paths causing data chasing errors.

C. Pulse Width - Pass

- The clock pulse width meets the requirements of the Flip-Flops and Block RAM memory on the FPGA.

4.1.2. Root Cause Analysis

- To ensure the correctness of the division over 8 clock cycles, some Pipeline stages (Stage 1 to Stage 4) must perform up to 5 iterations of the division algorithm within a single clock cycle.
- Each iteration includes the following operations: Bit Shift, Compare, Subtract, and Multiplexing.
- The sequence of these 5 logic sets creates a very long critical path, preventing the signal from propagating within the current clock cycle (e.g., 10ns).
- We prioritized maintaining a low latency of 8 cycles. To achieve this, the computational load per cycle had to increase, resulting in a decrease in the maximum operating frequency (Fmax).

The actual time the circuit needs to complete one cycle is:

$$T_{actual\_time} = T_{req} - \text{WNS} = 8.00 - (-10.357) = 18.357\text{ns}$$

The maximum frequency (F_{max}) is:

$$F_{max} = \frac{1}{18.357 \times 10^{-9}} \approx 54.47 \text{ MHz}$$

4.1.3. Timing Constraint Adjustment

Based on the actual measured latency of 18,357 ns, we adjusted the system clock speed down to a safe level of 40 MHz (25 ns cycle). The results after rerunning the Implementation showed a positive WNS, with no more timing errors (Timing Closure achieved). This confirms that the processor operates completely stably and reliably at 40 MHz.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.229 ns	Worst Hold Slack (WHS):	0.114 ns	Worst Pulse Width Slack (WPWS):	3.020 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4068	Total Number of Endpoints:	4068	Total Number of Endpoints:	2225

All user specified timing constraints are met.

Figure 4.2: Design Timming summary at 40 MHz

Table 4.1: Performance Comparison at 125 MHz and 40 MHz

Parameter	**Case 1**	**Case 2**
Clock Period Setting	8.00 ns	20.00 ns
Frequency target	125 MHz	40 MHz
WNS (Worst Negative Slack)	-10.357 **ns (FAILED)**	$\approx +0.229$ **ns (PASSED)**
Status	Timing Violation	Met Constraints
Concluion	Not stable, wrong data.	Stable, correct.

4.1.4. LUT usage

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	4668	0	0	53200	8.77
LUT as Logic	4628	0	0	53200	8.70
LUT as Memory	40	0	0	17400	0.23
LUT as Distributed RAM	0	0			
LUT as Shift Register	40	0			
Slice Registers	2187	0	0	106400	2.06
Register as Flip Flop	2187	0	0	106400	2.06
Register as Latch	0	0	0	106400	0.00
F7 Muxes	259	0	0	26600	0.97
F8 Muxes	0	0	0	13300	0.00

Figure 4.3: LUT usage at 125 MHz

2. Slice Logic Distribution

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice	1664	0	0	13300	12.51
SLICEL	1013	0			
SLICEM	651	0			
LUT as Logic	4628	0	0	53200	8.70
using O5 output only	0				
using O6 output only	3773				
using O5 and O6	855				
LUT as Memory	40	0	0	17400	0.23
LUT as Distributed RAM	0	0			
using O5 output only	0				
using O6 output only	0				
using O5 and O6	0				
LUT as Shift Register	40	0			
using O5 output only	40				
using O6 output only	0				
using O5 and O6	0				
Slice Registers	2187	0	0	106400	2.06
Register driven from within the Slice	560				
Register driven from outside the Slice	1627				
LUT in front of the register is unused	823				
LUT in front of the register is used	804				
Unique Control Sets	42		0	13300	0.32

Figure 4.4: Design Timming summary at 125 MHz

Most LUT resources are used for combinational logic (LUT as Logic), with 4,628 units. Meanwhile, the amount of LUTs used for memory (LUT as Memory/Shift Register) is very low (only 40 units) and 2187 for register. The majority of resources are allocated to:

- Adders/subtractors in the ALU and address calculation block.
- Complex multiplexers in the Forwarding Unit and Hazard Detection Unit.
- Especially the dense combinational logic sequence in the Pipelined Divider (bit shift and arithmetic subtraction blocks).

This is the physical reason explaining the previous Timing analysis result (negative WNS at 8ns). Due to the large number of logic gates (LUTs) connected in series between the two registers (especially in the divider stages performing 5 loops/cycle), the signal takes a long time to propagate, making it difficult to achieve a high frequency (125 MHz).

However, perhaps the bright spot is its space efficiency, occupying less than 13% of the FPGA's total resources.

5

CONCLUSION

This project successfully designed, implemented, and verified a RISC-V pipelined processor with an integrated multi-cycle pipelined divider. The implementation process and results can be summarized in the following key points:

1. Functional Correctness: Behavior simulation confirms the system operates correctly according to the test script. Conflict handling mechanisms (Forwarding, Stalling) work effectively. In particular, Input Buffering techniques have been applied to completely overcome the instability of the splitter input signal, ensuring absolutely accurate calculation results in all cases.

2. Performance and Timing Analysis: Static Time Analysis (STA) reveals the physical limitations of the design at 125 MHz due to the long combinational logic sequence in the splitter (Setup Time Violation). A safe operating frequency has been identified and adjusted down to 40 MHz, allowing the system to achieve Timing Closure and ensuring reliability when deployed on hardware.

3. Hardware Resource Utilization: The design demonstrates optimal use of space on the FPGA (occupying approximately 12.5%of slices). The low resource utilization rate not only proves the efficiency of the architecture but also opens up significant room for integrating additional features in the future.

In summary, the project not only achieved the goal of building a fully functional microprocessor but also provided insightful analyses of the trade-offs between latency, area, and frequency in digital circuit design.

6

GITHUB REPOSITORY

The complete source code and related materials for this project can be found in the following GitHub repository

https://github.com/PhamHieu93/Introduction-to-SoC

REFERENCE

[1] T. T. Hoang, “Introduction to System-on-Chip: Single-Cycle Datapath,” Lecture slides, Department of Computer and Network Engineering, The University of Electro-Communications (UEC), 2025.

[2] T. T. Hoang, “Introduction to System-on-Chip: Multi-Cycle Datapath,” Lecture slides, Department of Computer and Network Engineering, The University of Electro-Communications (UEC), 2025.

[3] T. T. Hoang, “Introduction to System-on-Chip: Pipelined Datapath,” Lecture slides, Department of Computer and Network Engineering, The University of Electro-Communications (UEC), 2025.

[4] T. T. Hoang, “Introduction to System-on-Chip: Basic Arithmetic,” Lecture slides, Department of Computer and Network Engineering, The University of Electro-Communications (UEC), 2025.