VIETNAM NATIONAL UNIVERSITY – HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRONIC

REPORT MILESTONE 3
EE3203 – COMPUTER ARCHITECTURE
DESIGN OF PIPELINED RISC – V PROCESSOR

Supervisor: Assoc. Prof., PhD. Tran Hoang Linh

Group: L02 – Semester 251

No	Student ID	Full name
1	2313014	THAI TAI
2	2313225	TRAN DINH THIEN
3	2310434	LAM CHI DINH

Ho Chi Minh City, 12/2025

TASK ASSIGNMENT TABLE

No	Student ID	Full name	Level
1	2313014	Thai Tai	100%
2	2313225	Tran Dinh Thien	100%
3	2310434	Lam Chi Dinh	100%

Ho Chi Minh City, 12/2025

Contents

List of Tables

List of Figures

1. INTRODUCTION

Following the single-cycle processor design in Milestone 2, this milestone aims to implement a 5-stage Pipelined RISC-V Processor to enhance instruction throughput and overall performance. The execution flow is divided into five distinct stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

The project focuses on resolving pipeline hazard - a critical challenge in pipelined architectures. Two distinct models were developed and evaluated:

- Model 1 (Non – Forwading): Resolves hazards exclusively by stalling the pipeline (inserting bubbles) or flushing instructions.
- Model 2 (Forwarding): Implements a Forwarding Unit to bypass data from later stages (EX/MEM, MEM/WB) back to the execution stage, significantly reducing the number of stalls caused by data dependencies.

2. DESIGN STRATEGY

2.1. Overview

The processor is designed based on the RV32I ISA. We inserted pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB) between the logic stages to hold the intermediate data and control signals.

- IF Stage: Fetches instruction from Instruction Memory based on PC.
- ID Stage: Decodes instruction, reads Register File, and generates control signals.
- EX Stage: Performs ALU operations and calculates branch targets.
- MEM Stage: Accesses Data Memory (Load/Store).
- WB Stage: Writes results back to the Register File.

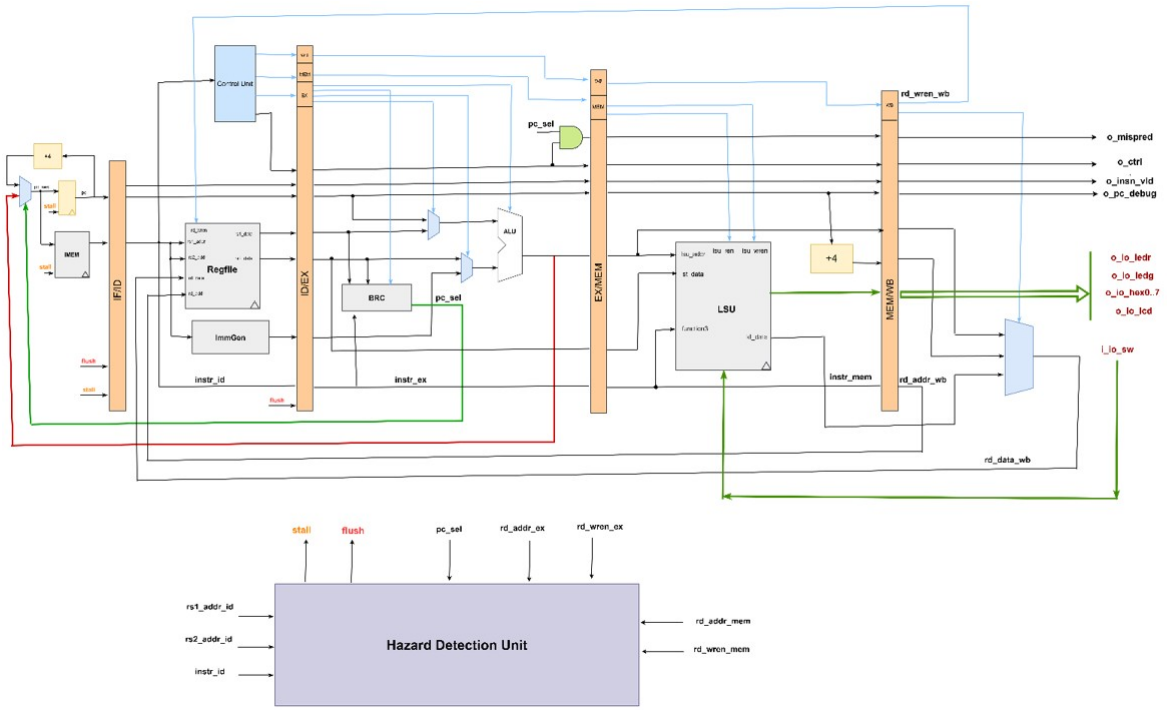

Figure 1: Datapath of Model 1

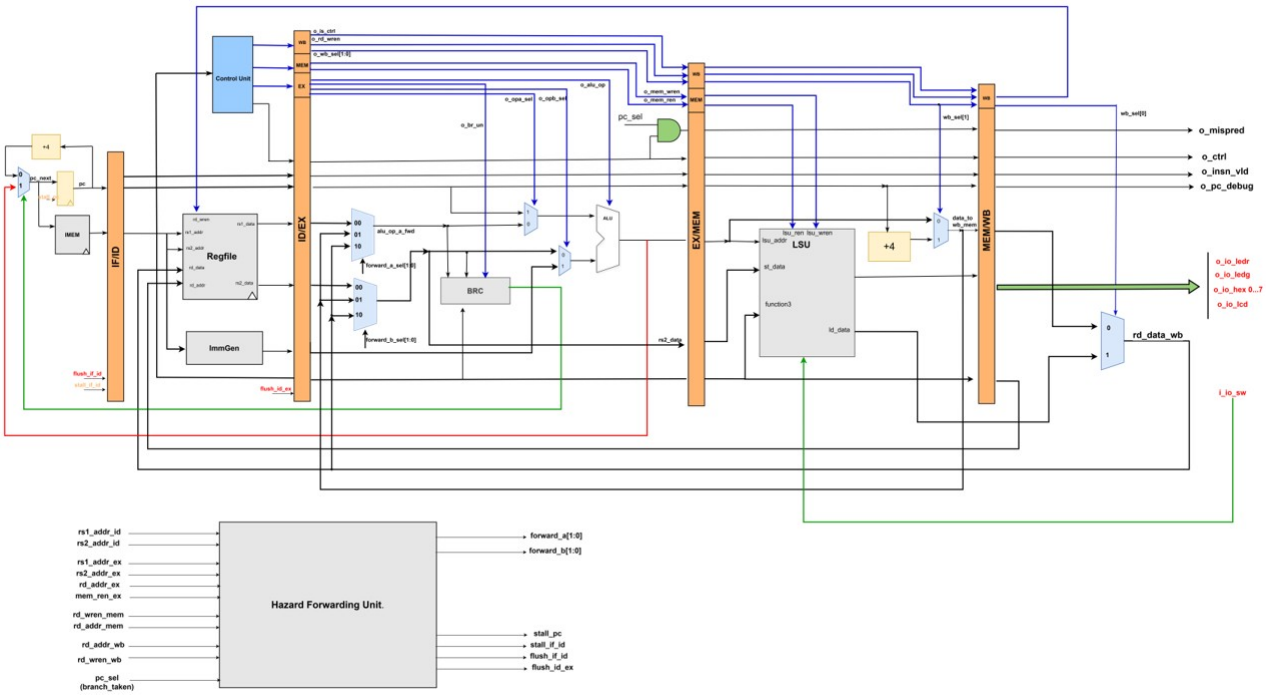

Figure 2: Datapath of Model 2

2.2. Hazard Handling Strategy

One of the main tasks in this milestone is handling Data Hazards and Control Hazards.

Data Hazards: Occur when an instruction depends on the result of a previous instruction that has not yet been written back.

- Strategy for Model 1 (Non - Forwarding): The Hazard Detection Unit detects the dependency in the ID stage. If a hazard is found, it stalls the PC and IF/ID register and inserts a "bubble" (NOP) into the ID/EX register until the dependency is resolved.
- Strategy for Model 2 (Forwarding): A Forwarding Unit is implemented. It compares the source registers (rs1, rs2) of the current instruction in EX with the destination registers (rd) of instructions in MEM and WB. If a match is found, the result is forwarded directly to the ALU input, avoiding stalls (except for the Load-Use case).

Control Hazards: Occur when the pipeline makes a decision based on an instruction that has not finished execution (e.g., BEQ, BNE).

- Strategy: We implemented a Static Branch Prediction scheme: "Always Not Taken".
- The pipeline assumes the branch is not taken and continues fetching PC + 4.
- If the branch condition is actually met (calculated in the EX stage), the pipeline flushes the instructions in the IF/ID and ID/EX registers and updates the PC to the correct branch target.

2.3. Hazard Detection Unit

In the Non-Forwarding architecture (Model 1), data hazards cannot be resolved by bypassing results from the pipeline registers. Instead, the processor must handle Read-After-Write (RAW) dependencies by strictly stalling the pipeline until the required data is written back to the Register File. The Hazard Detection Unit is responsible for monitoring instruction dependencies and control flow changes, thereby generating appropriate stall and flush signals to control the pipeline registers.

2.3.1. Specification

Signal	Width	Direction	Description
i_rs1_addr	5	input	Address of the first source register
i_rs2_addr	5	input	Address of the second source register
i_rd_addr_ex	5	input	Address of the destination register in EX stage
i_rd_wren_ex	1	input	Write enable for the destination register in EX stage
i_rd_addr_mem	5	input	Address of the destination register in MEM stage
i_rd_wren_mem	1	input	Write enable for the destination register in MEM stage
i_branch_taken	1	input	From pc_sel of BRC
o_stall_pc	1	output	Stall (stop) PC register
o_stall_if_id	1	output	Stall (stop) IF/ID register
o_flush_if_id	1	output	Flush (clear) IF/ID register
o_flush_id_ex	1	output	Flush (clear) ID/EX register

Table 1: Hazarding Detection Unit Specification

2.3.2. Datapath

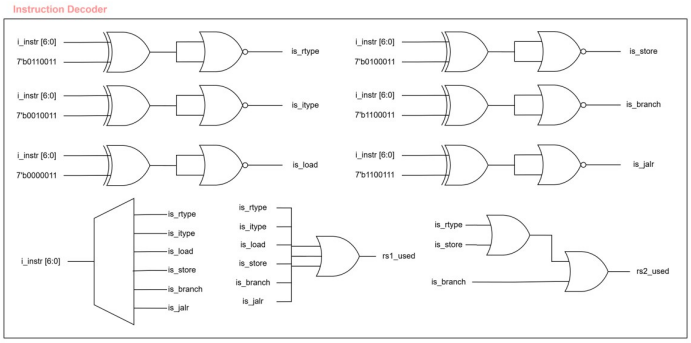

Figure 3: Datapath of Hazard Detection Unit

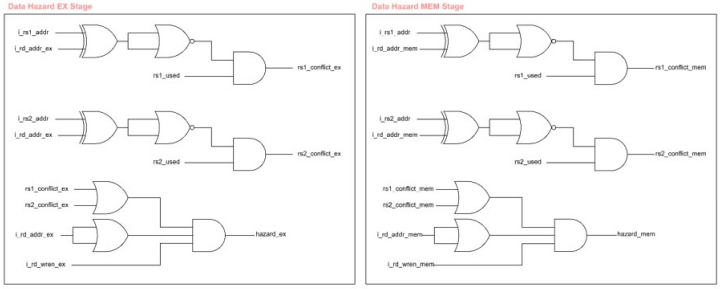

Figure 4: Datapath of Hazard Detection Unit

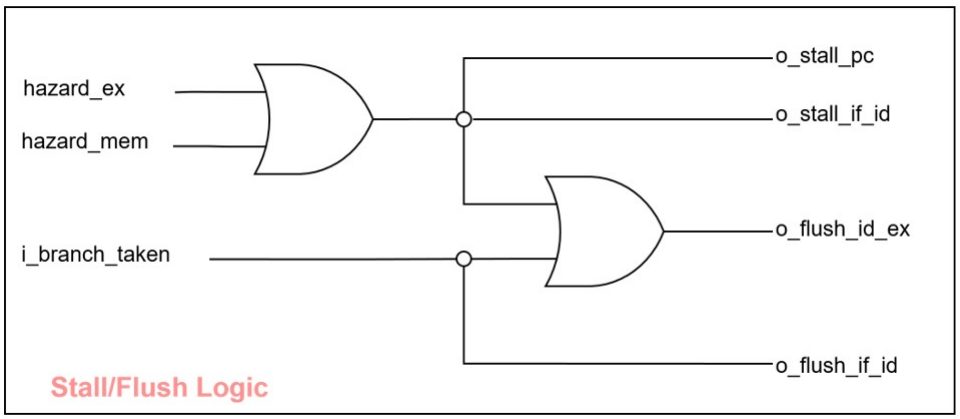

Figure 5: Datapath of Hazard Detection Unit

2.4.Forwarding Unit

The Forwarding Unit is the central control block of the Forwarding Model, responsible for maintaining data integrity and control flow within the RISC-V processor. This module integrates two primary functions:

- Hazard Detection Logic: Identifies hazards that cannot be resolved by forwarding, specifically Load-Use Hazards and Control Hazards (Branch Taken). Upon detection, it triggers stall signals to freeze the pipeline or flush signals to discard invalid instructions.
- Forwarding Logic: Compares register addresses across the EX, MEM, and WB stages to control the ALU input multiplexers. This mechanism enables data bypassing from previous instructions directly to the current execution stage, effectively resolving Read-After-Write (RAW) hazards without performance penalties.

2.4.1. Specification

Signal	Width	Direction	Description
i_rs1_addr_id	5	input	Source register 1 address (decoding at ID).
i_rs2_addr_id	5	input	Source register 2 address (decoding at ID).
i_instr_id	32	input	Current instruction at ID (to check register usage).
i_mem_ren_ex	1	input	Memory Read Enable (indicates Load instruction).
i_rd_addr_ex	5	input	Destination register address at EX.
i_rs1_addr_ex	5	input	Source register 1 address required at EX.
i_rs2_addr_ex	5	input	Source register 2 address required at EX.
i_rd_wren_mem	1	input	Write Enable signal at MEM stage.
i_rd_addr_mem	5	input	Destination register address at MEM.
i_rd_wren_wb	1	input	Write Enable signal at WB stage.
i_rd_addr_wb	5	input	Destination register address at WB.
i_branch_taken	1	input	Branch decision signal (1 = Taken).
o_stall_pc	1	output	Stalls the Program Counter.
o_stall_if_id	1	output	Stalls the IF/ID pipeline register.
o_flush_if_id	1	output	Flushes the IF/ID pipeline register.
o_flush_id_ex	1	output	Flushes the ID/EX pipeline register.
o_forward_a	2	output	ALU Operand A Mux Select.
o_forward_b	2	output	ALU Operand B Mux Select.

Table 2: Forwarding Unit Unit Specification

2.4.2. Datapath

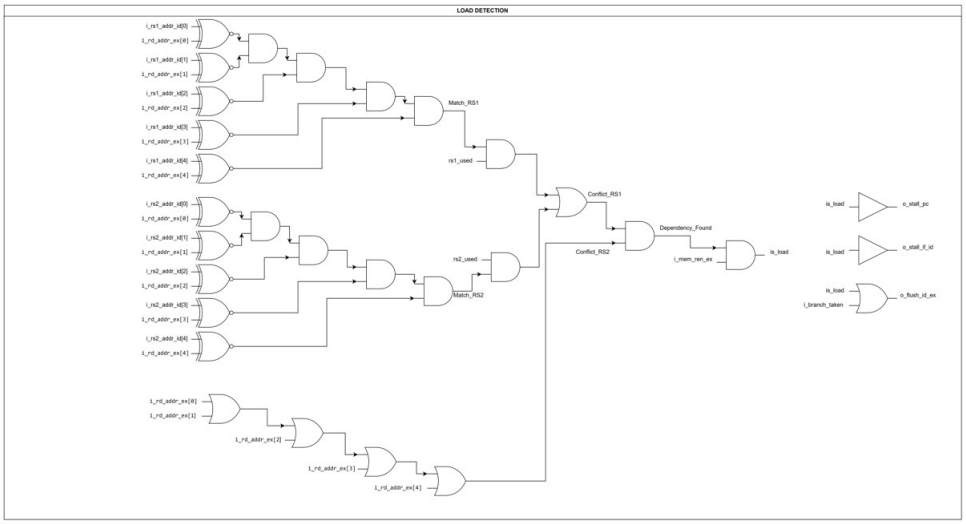

Figure 6: Datapath of Forwarding Unit

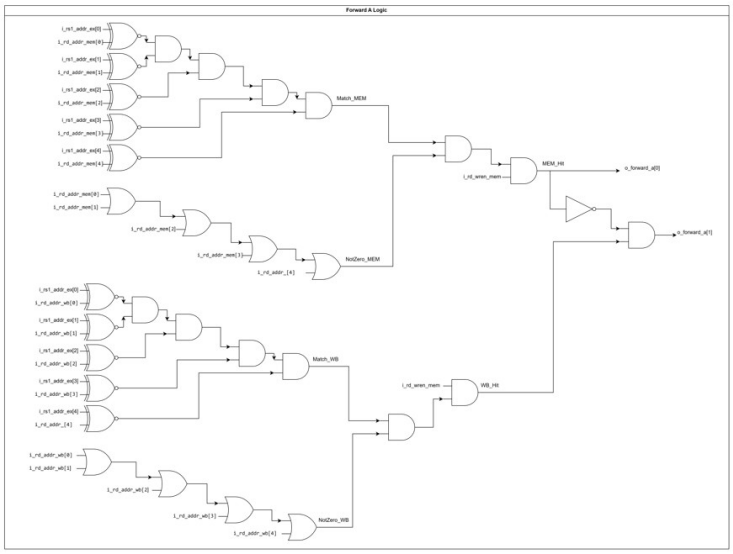

Figure 7: Datapath of Forwarding Unit

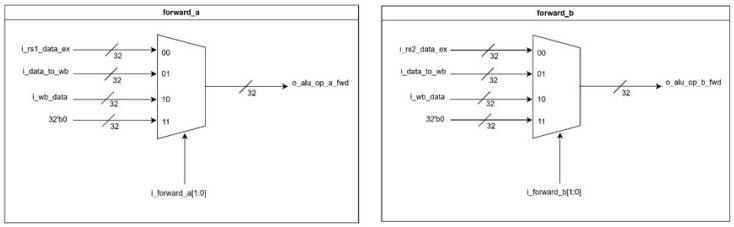

Figure 8: Datapath of Forwarding Unit

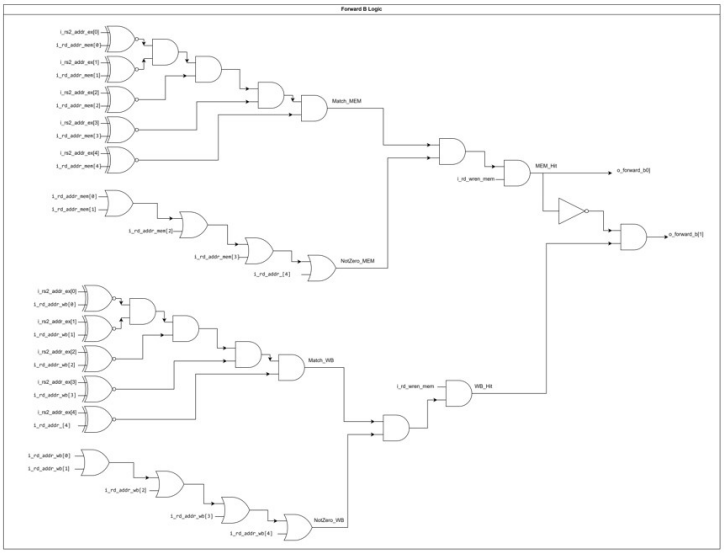

Figure 9: Datapath of Forwarding Unit

2.5. ALU

The ALU performs all arithmetic and logical operations defined by the RV32I instruction set, driven by the alu_op control signal generated by the Control Unit. It receives

operands from the register file, the immediate generator, or the program counter (PC), then executes the corresponding operation and outputs the result.

A key design constraint is that the ALU must avoid using any non-synthesizable operators such as subtraction ($-$), comparison ($<$, $>$), shifting ($\ll$, $\gg$, $\ggg$), multiplication ($*$), division ($/$), or modulo ($\%$).

alu_op	R-Type	I-Type
ADD	rd $\leftarrow rs1 + rs2$	rd $\leftarrow rs1 + imm$
SUB	rd $\leftarrow rs1 - rs2$	n/a
SLT	rd $\leftarrow (rs1 < rs2)?1 : 0$	rd $\leftarrow (rs1 < imm)?1 : 0$
SLTU	rd $\leftarrow (rs1 < rs2)?1 : 0$	rd $\leftarrow (rs1 < imm)?1 : 0$
XOR	rd $\leftarrow rs1 \oplus rs2$	rd $\leftarrow rs1 \oplus imm$
OR	rd $\leftarrow rs1 \vee rs2$	rd $\leftarrow rs1 \vee imm$
AND	rd $\leftarrow rs1 \wedge rs2$	rd $\leftarrow rs1 \wedge imm$
SLL	rd $\leftarrow rs1 \ll rs2[4:0]$	rd $\leftarrow rs1 \ll immrs2[4:0]$
SRL	rd $\leftarrow rs1 \gg rs2[4:0]$	rd $\leftarrow rs1 \gg imm[4:0]$
SRA	rd $\leftarrow rs1 \ggg rs2[4:0]$	rd $\leftarrow rs1[4:0]$

2.5.1. Specification

Signal	Width	Direction	Description
i_op_a	32	input	First operand for ALU operations
i_op_b	32	input	Second operand for ALU operations
i_alu_op	4	input	The operation to be performed
o_alu_data	32	output	Result of the ALU operation

Table 3: ALU Specification

2.5.2. Datapath

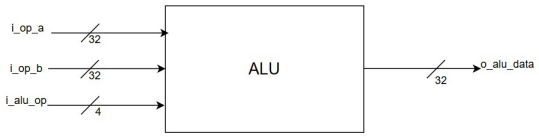

Figure 10: Overview of ALU

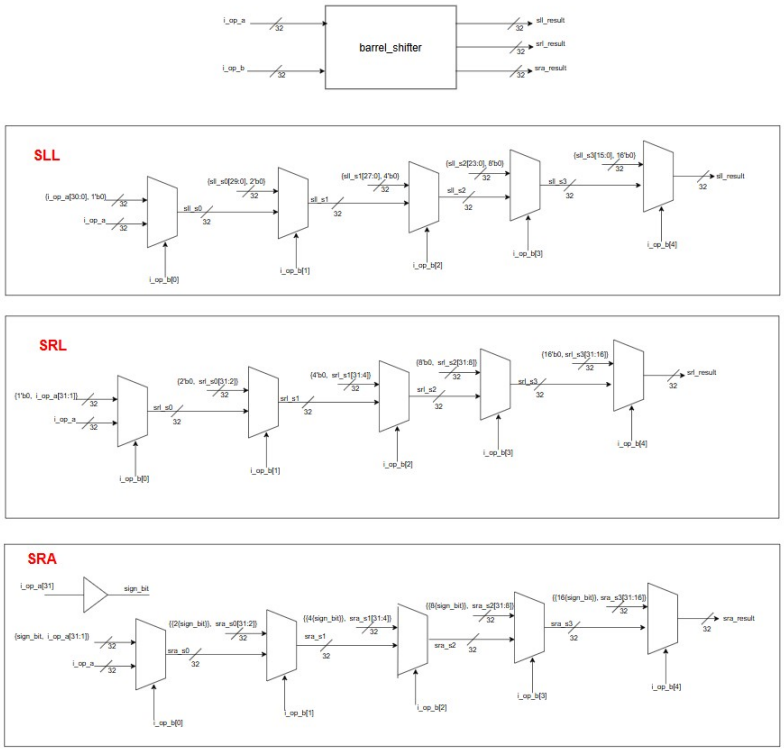

Figure 11: Barrel Shifter

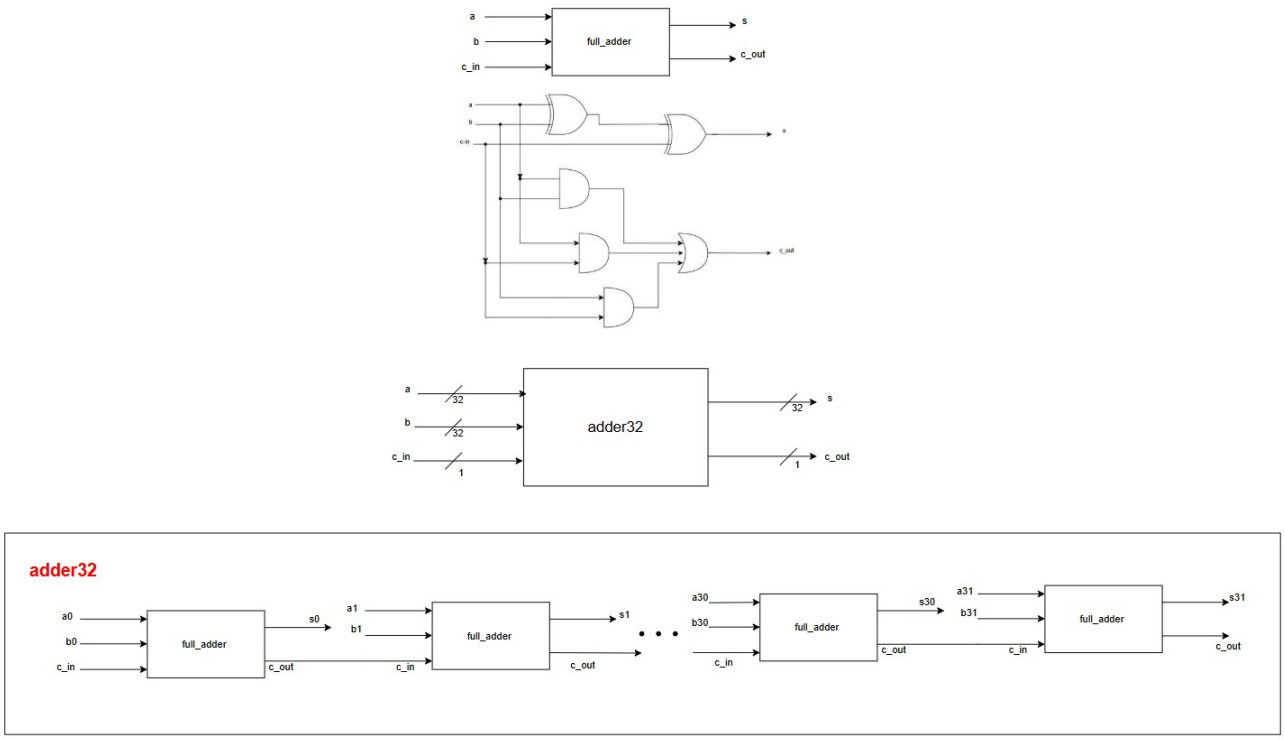

Figure 12: Adder 32-bit

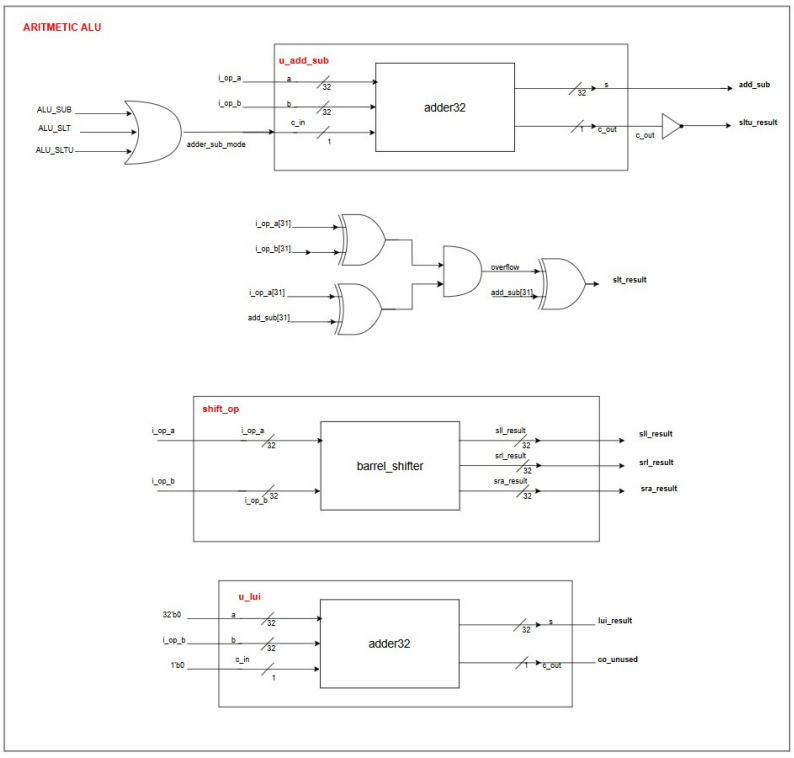

Figure 13: Arithmetic operations

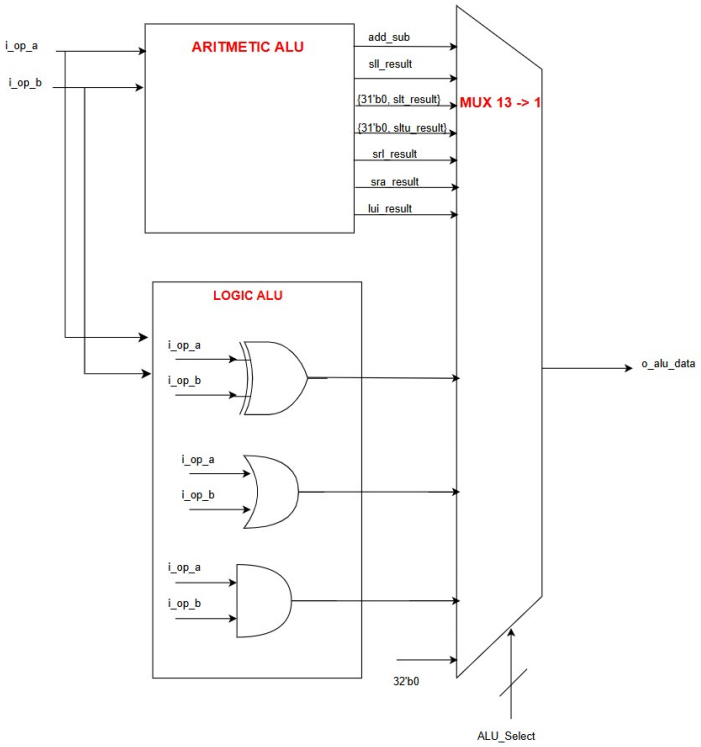

Figure 14: Overall datapath of ALU

2.6. BRC (Branch Comparison Unit)

Located in the Execute (EX) stage, the BRC unit compares two input operands based on the instruction's opcode and funct3 fields to handle conditional branch instructions and identify Jump instructions. Unlike the Milestone 2 design, the primary output of this module is the pc_sel signal, which determines whether to update the PC to the target address (Taken) or proceed with the next sequential instruction (PC + 4).

2.6.1. Specification

Signal	Width	Direction	Description
i_rs1_data	32	input	Data from the first register
i_rs2_data	32	input	Data from the second register
i_br_un	1	input	Comparison mode (1 if signed, 0 if unsigned)
i_funct3	3	input	Function 3 (from instruction)
i_is_branch	1	input	If the instruction is branch (from control unit)
i_is_jump	1	input	If he instruction is jump (from control unit)
o_pc_sel	1	output	PCMux select signal

Table 4: BRC Specification

2.6.2. Datapath

Figure 15: Overview of BRC

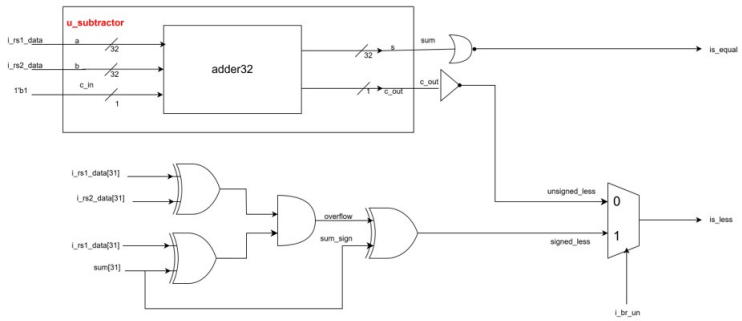

Figure 16: Detailed internal datapath of the BRC

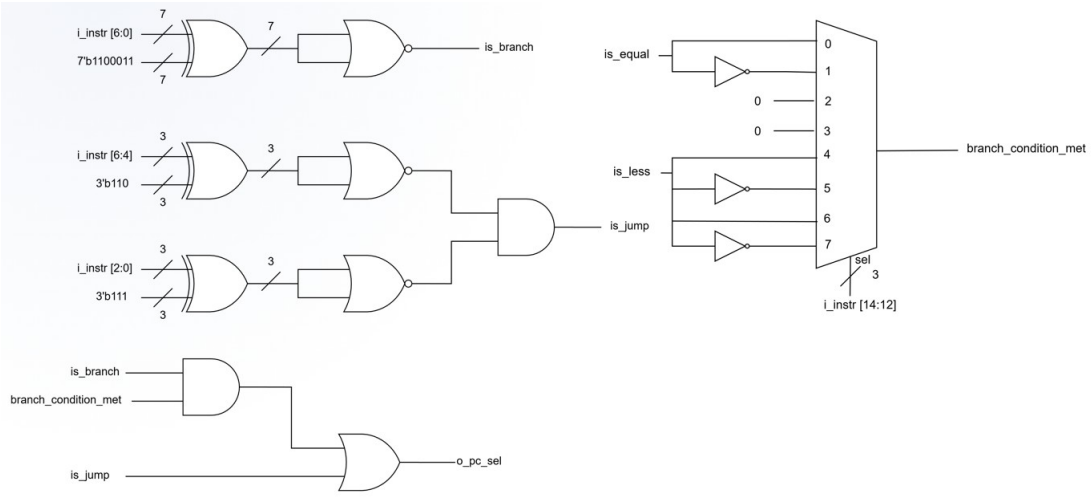

Figure 17: Determine pc_sel

2.7. ImmGen (Immediate Generator)

The ImmGen module is responsible for extracting and sign-extending immediate values defined in the RV32I instruction set. Since RV32I employs multiple immediate encoding formats (I, S, B, U, and J), ImmGen must correctly identify the instruction type through its opcode and reconstruct the immediate field by re-arranging the corresponding instruction bits according to each format's specification.

The handling of each immediate format is summarized in the following table:

Type	Immediate
I-Type	20instr[31], instr[31:20]
S-Type	20instr[31], instr[31:25], instr[11:7]
B-Type	19instr[31], instr[31], instr[7], instr[30:25], instr[11:8], 1'b0
J-Type	11instr[31], instr[31], instr[19:12], instr[20], instr[30:21], 1'b0
U-Type	instr[31:12], 12'b0

Table 5: Immediate Format

2.7.1. Specification

Signal	Width	Direction	Description
i_instr	32	input	instruction 32-bit
o_immExt	32	output	immediate value

Table 6: ImmGen Specification

2.4.2. Datapath

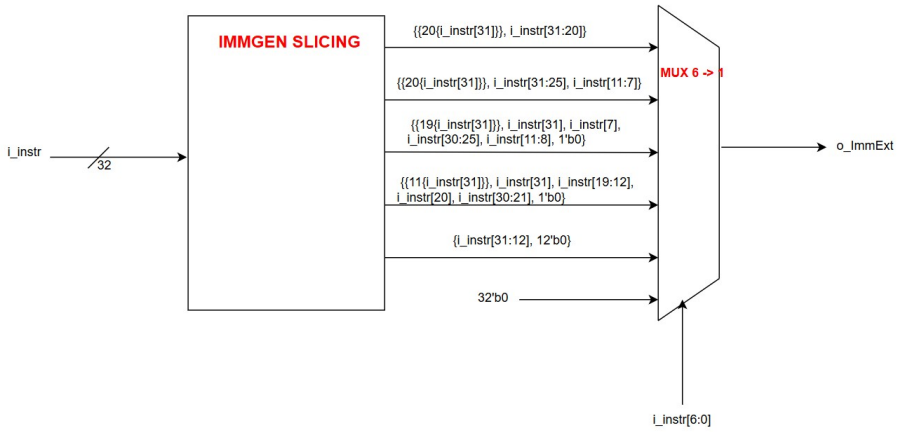

Figure 18: Overview of ImmGen

2.8. Regfile

Located in the ID stage, the Register File consists of 32 RISC-V 32-bit registers (x0–x31) configured with two asynchronous read ports and one synchronous write port. In Milestone 3, this module incorporates a "Read-After-Write" mechanism (Internal Forwarding), allowing the ID stage to instantly retrieve the value currently being written by the WB stage within the same clock cycle, effectively resolving structural hazards.

2.8.1. Specification

Signal	Width	Direction	Description
i_clk	1	input	Global clock
i_reset	1	input	Global active reset
i_rs1_addr	5	input	Address of the first source register
i_rs2_addr	5	input	Address of the second source register
o_rs1_data	32	output	Data from the first source register
o_rs2_data	32	output	Data from the second source register
i_rd_addr	5	input	Address of the destination register
i_rd_data	32	input	Data to write to the destination register
i_rd_wren	1	input	Write enable for the destination register

Table 7: Regfile Specification

2.8.2. Datapath

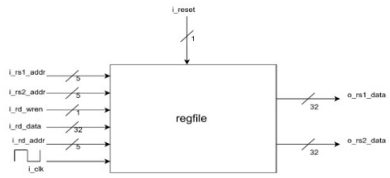

Figure 19: Overview of Regfile

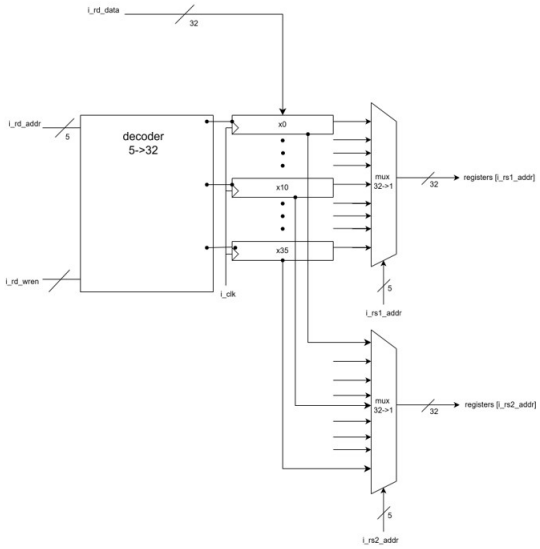

Figure 20: Detailed internal datapath of Regfile

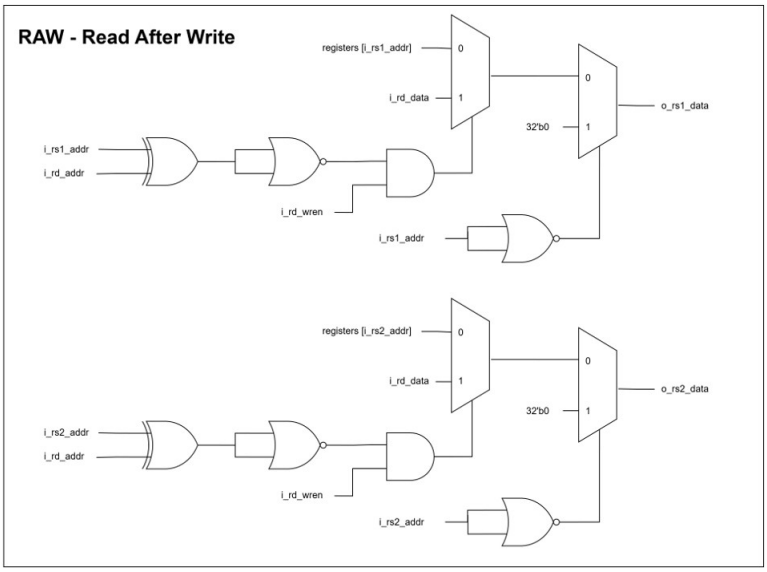

Figure 21: Detailed internal datapath of Regfile

2.9. IMEM

The Instruction Memory stores the machine code of the program to be executed. In Milestone 3, the design has been upgraded from an asynchronous read model (used in the Single-cycle processor) to a synchronous read model. This change aligns with the pipelined architecture, where the instruction fetch occurs at the rising edge of the clock in the IF stage. Additionally, the module now supports a stall mechanism to handle pipeline hazards.

2.9.1. Specification

Signal	Width	Direction	Description
i_clk	1	input	System clock signal.
i_stall	1	Input	Stall signal from the Hazard Detection Unit.
i_pc	32	input	The instruction address is provided to the IMEM
o_instr	32	output	The corresponding 32-bit instruction is executed

Table 8: IMEM Specification

2.9.2. Datapath

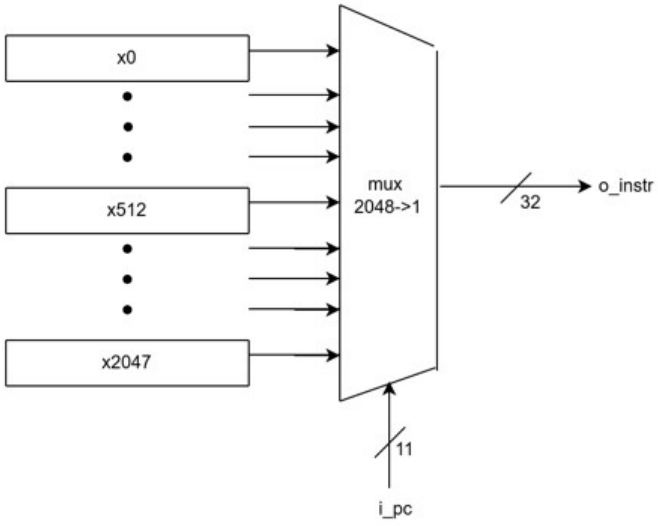

Figure 22: Overview of IMEM

2.10. LSU

Located in the Memory Access (MEM) stage, the Load-Store Unit (LSU) is responsible for accessing data memory and interfacing with peripheral devices. It executes all load instructions (lw, lh, lb, lhu, lbu) and store instructions (sw, sh, sb).

Unlike the asynchronous design in Milestone 2, the LSU in Milestone 3 adopts a synchronous read model to align with the pipeline timing and ensure compatibility with FPGA Block RAM resources.

To standardize interaction, the LSU employs a Memory-Mapped I/O architecture. An address decoder monitors the incoming address from the ALU and routes the transaction to the appropriate target:

- Data Memory (DMEM): Modeled as a 64 KiB array (16,384 words).
- Peripherals: Includes Red LEDs, Green LEDs, 7-segment displays, LCDs, and Switches. Dedicated registers function as buffers for output devices, while switches are read directly.

Base address	Top address	Mapping
0x1001_1000	0xFFFF_FFFF	(Reserved)
0x1001_0000	0x1001_0FFF	Switches
0x1000_5000	0x1000_FFFF	(Reserved)
0x1000_4000	0x1000_4FFF	LCD Control Registers
0x1000_3000	0x1000_3FFF	7-segment LEDs 7-4
0x1000_2000	0x1000_2FFF	7-segment LEDs 3-0
0x1000_1000	0x1000_1FFF	Green LEDs
0x1000_0000	0x1000_0FFF	Red LEDs
0x0001_0000	0x0FFF_FFFF	(Reserved)
0x0000_0000	0x0000_FFFF	Memory (64KiB)

Based on the analysis above, the overall structure of the LSU can be summarized as follows:

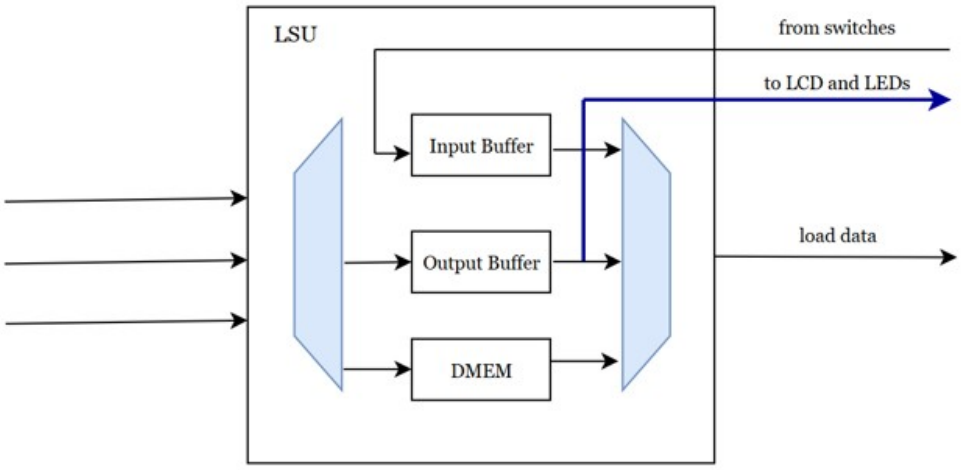

Figure 23: Overview of LSU

2.10.1. Specification

Signal	Width	Direction	Description
i_clk	1	input	Global clock, active on the rising edge
i_reset	1	input	Global active reset
i_lsu_addr	32	input	Address for data read/write
i_funct3	3	input	Select read - write size
i_st_data	32	input	Data to be stored
i_lsu_wren	1	input	Write enable signal (1 if writing)
i_lsu_ren	1	input	Read enable signal (1 if reading).
o_ld_data	32	output	Data read from memory
o_io_ledr	32	output	Output for red LEDs
o_io_leg	32	output	Output for green LEDs
o_io_hex0..7	7	output	Output for 7-segment displays
o_io_lcd	32	output	Output for the LCD register
i_io_sw	32	input	Input for switches

Table 9: LSU Specification

2.10.2. Datapath

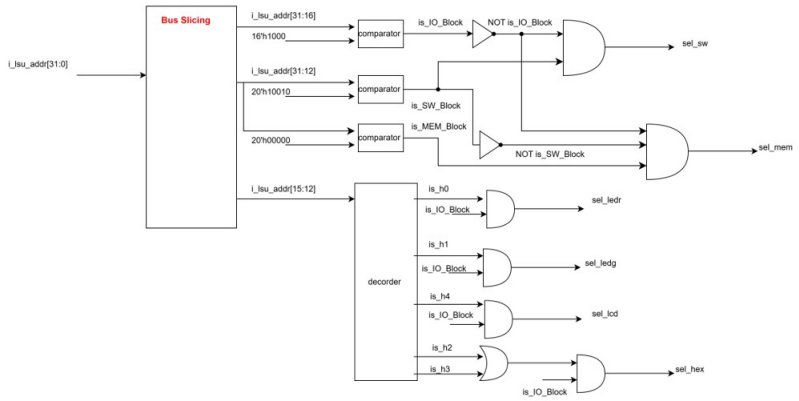

Figure 24: Address Decoder

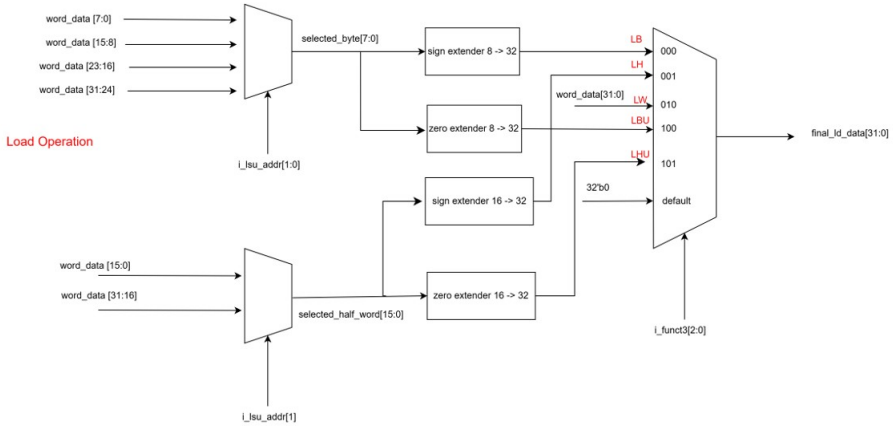

Figure 25: Memory Read

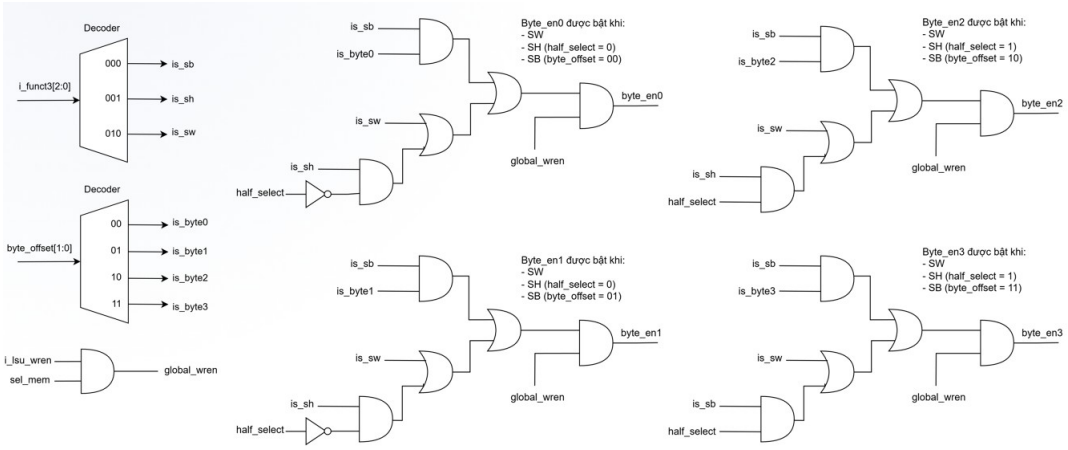

Figure 26: Write enable signals

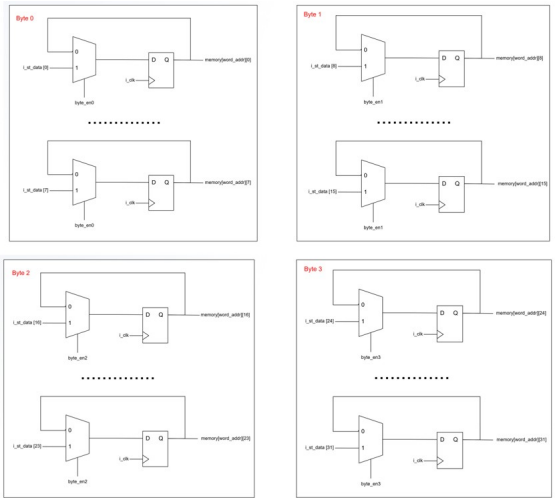

Figure 27: Writing Data to Individual Bytes in a Word

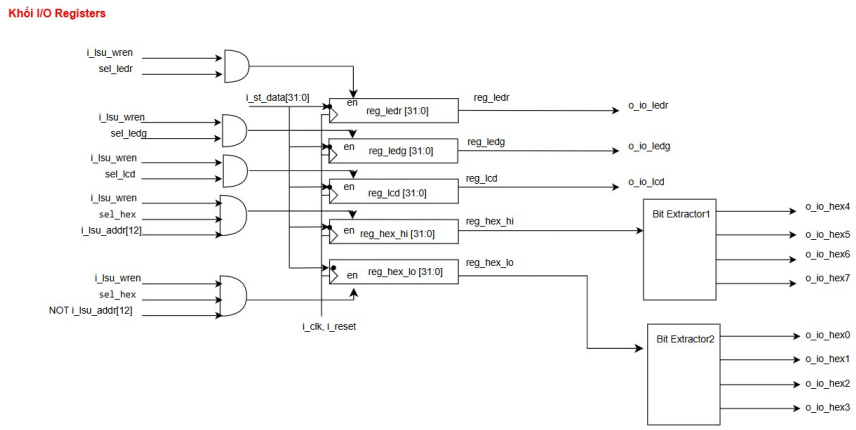

Figure 28: I/O peripherals

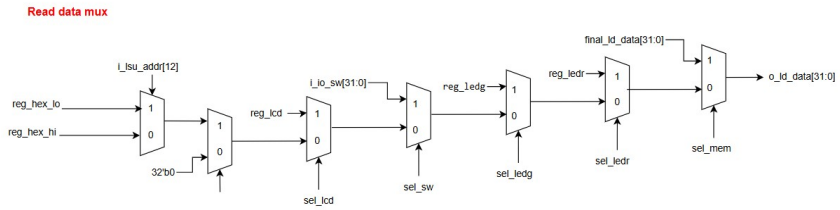

Figure 29: Data read MUX

2.11. Control Unit

The Control Unit is the ”brain” of the processor. It accepts the 7-bit Opcode (i_instr[6:0]) from the current instruction and generates all necessary control signals to orchestrate the datapath operations.

In this pipelined design, unlike the single-cycle architecture where signals are used immediately, the Control Unit generates signals for all future stages (EX, MEM, WB) simultaneously. These signals are then bundled and passed into the pipeline registers (ID/EX, EX/MEM, MEM/WB) to ensure they arrive at the corresponding stage in synchronization with the instruction execution.

2.11.1. Specification

Signal	Width	Direction	Description
i_instr	32	input	Instruction code fetched from IMEM
o_rd_wren	1	output	Write enable signal for destination register
o_br_un	1	output	Comparison mode selection
o_opa_sel	1	output	Operand A selection signal for ALU
o_opb_sel	1	output	Operand B selection signal for ALU
o_alu_op	4	output	ALU operation selection signal
o_mem_wren	1	output	Write enable signal for LSU
o_mem_ren	1	output	Read enable signal for LSU
o_wb_sel	2	output	Data selection signal for writing into rd
o_is_ctrl	1	output	To record control transfer instructions

Table 10: Control Unit Specification

2.11.2. Datapath

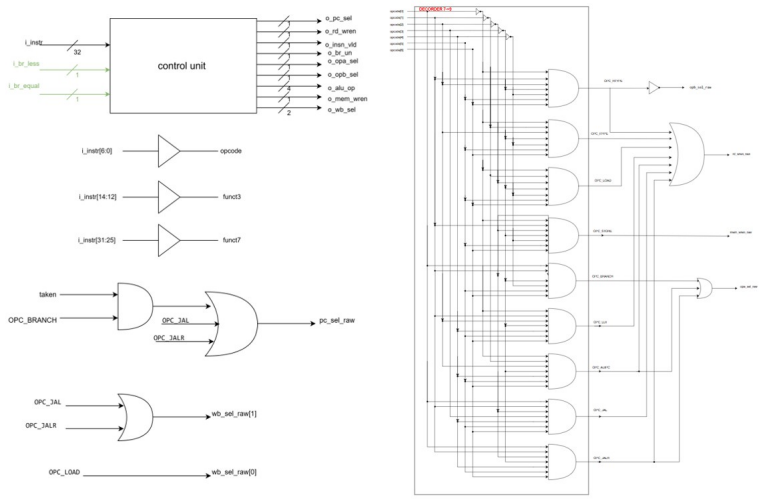

Figure 30: Detailed internal datapath of Control Unit

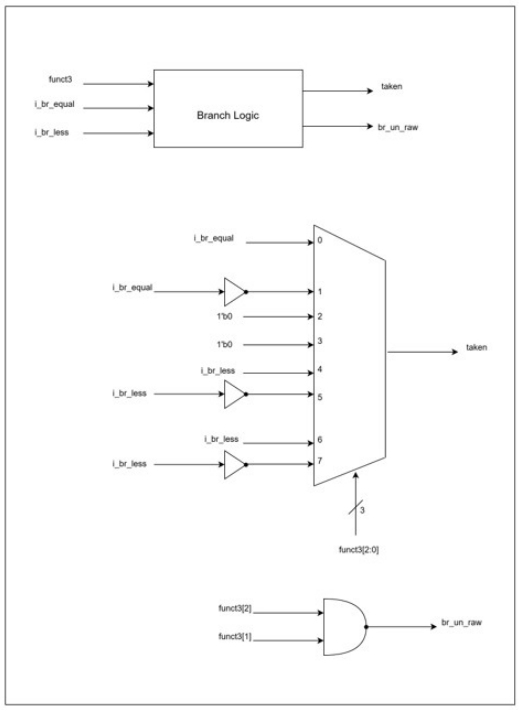

Figure 31: Detailed internal datapath of Control Unit

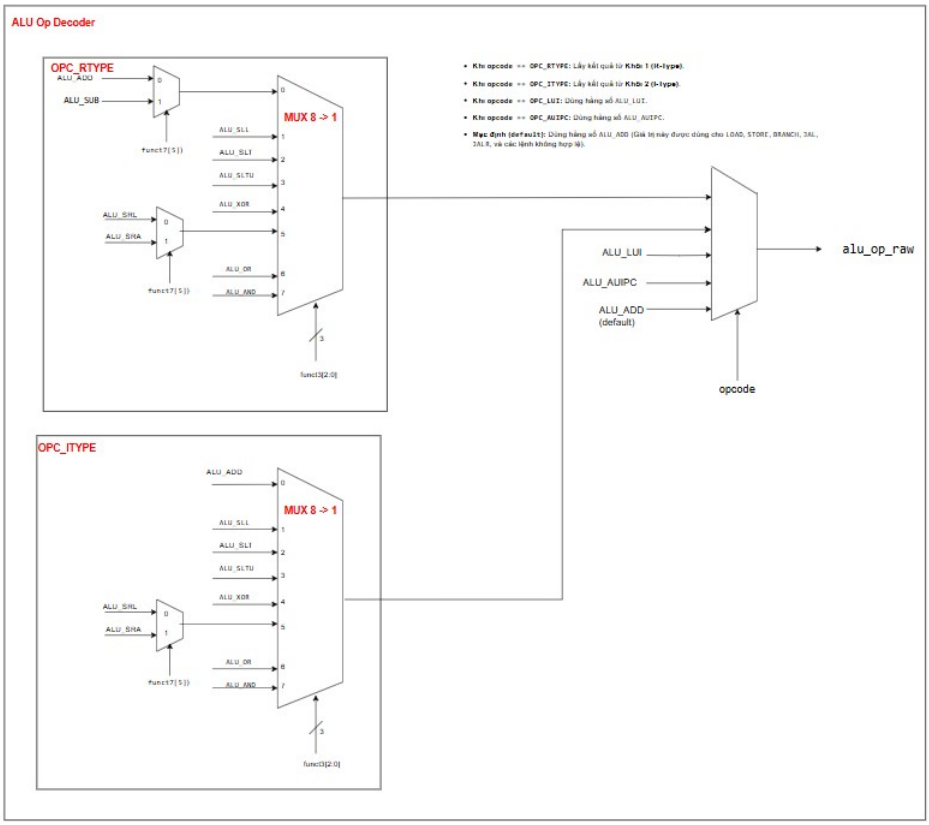

Figure 32: Detailed internal datapath of Control Unit

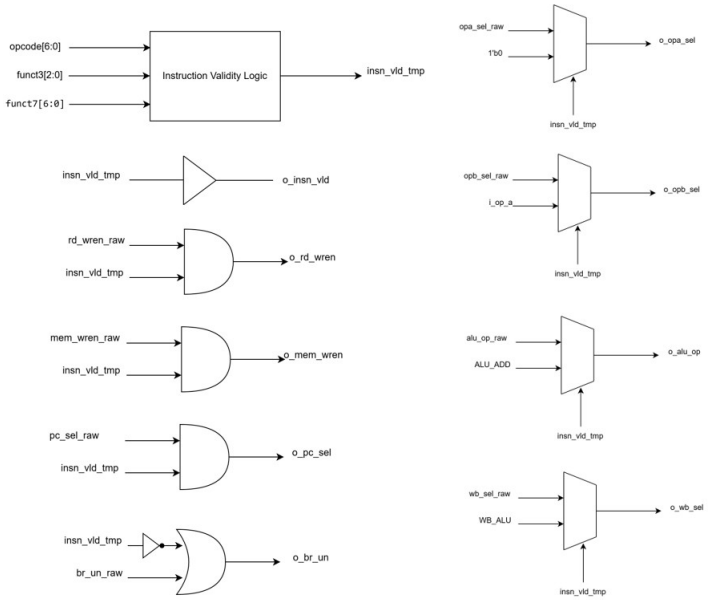

Figure 33: Detailed internal datapath of Control Unit

3. VERIFIATION STRATEGY

3.1. Objective

The objective of the verification process is to ensure that the single-cycle RV32I processor is correctly designed to function properly with all instruction groups in the RV32I instruction set, including:

- R-type: Arithmetic operations between two registers (ADD, AND, XOR, SLT, SLL, ...)
- I-type: Operations between a register and immediate values (ADDI, XORI, ...)
- Load/Store: Memory access operations (LB, LH, LW, SB, SH, SW, ...)
- B-type: Branch operations (BEQ, BNE, BLT, BGE, BLTU, BGEU)
- J-type: Jump operations (JAL, JALR)
- U-type: Address constant generation (LUI, AUIPC)

The verification process uses the Grand Testbench provided by the TA to test the entire CPU block, including the Datapath, control logic, regfile, ALU, BRC, LSU, and jump/branch designs.

3.2. Verification environment

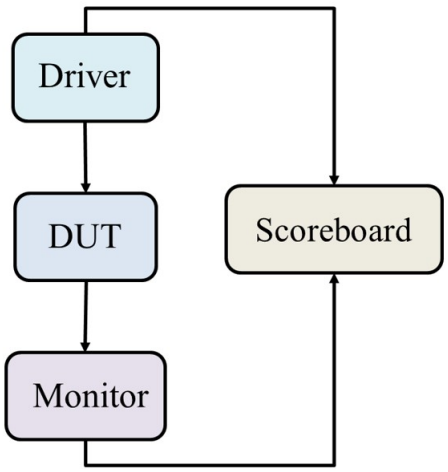

Figure 34: Overall Verification Model for the Single-Cycle RV32I Processor

The testbench includes the following main components:

Component	Description
Driver	Provides instruction sequences (in HEX format) read from the file isa_4b.hex
DUT (Device Under Test)	Single-cycle RV32I processor
Monitor	Observes signals o_pc_debug, o_insn_vld, and memory-mapped I/O
Scoreboard	Compares actual results with expected results and determines whether to display PASS / ERROR for each test case

Table 11: Testbench

3.3. Simulation procedure

- Collect all code files (.sv) into a single folder ./../00_src.
- Set the paths to the design files and compile them into a file list (flist) located at ./../03_sim/flist.
- Load the test program (isa_4b.hex) into the instruction memory (IMEM) using the $readmemh() command.

- Run the simulation commands as instructed on the server.
- Monitor the verification process via the o_pc_debug signal and observe the PASS/ERROR results printed for each subprogram.

3.4. Test results

The Pipelined RISC-V processor designed by our team successfully passed all functional verification tests across various instruction groups. The verification results are summarized in the scoreboard below:

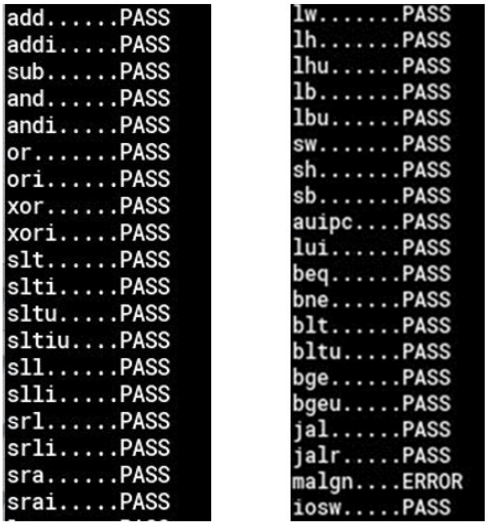

Figure 35: Functional test results on the server

The displayed results align perfectly with the expected outcomes of the Grand Test, confirming the absence of logic errors within the Datapath or Control Unit.

3.4.1. Model 1: Non-Forwarding

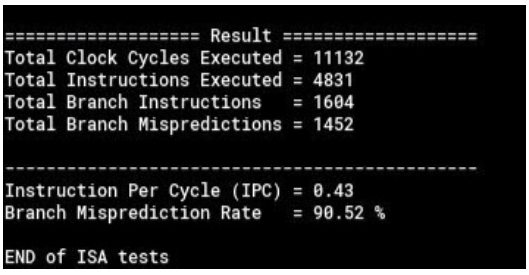

Figure 36: Instruction processing statistics for Model 1

The processor executed a total of 4,831 instructions, including 1,604 branch instructions. For Model 1, the system required 11,132 clock cycles to complete this instruction set.

Model 1 exhibits a relatively low IPC (Instructions Per Cycle) of 0.43. The Branch Misprediction Rate is high (90.52%) as this represents a baseline implementation without advanced branch prediction mechanisms.

3.4.2. Model 2: Forwarding

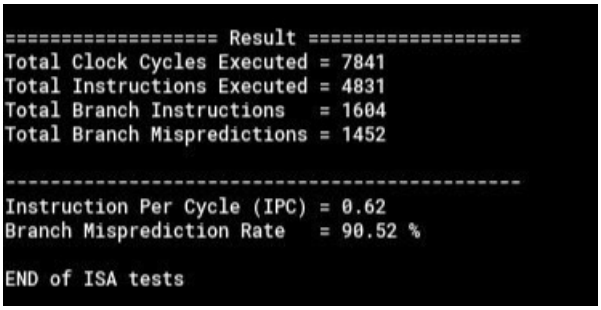

Figure 37: Instruction processing statistics for Model 2

Compared to Model 1, Model 2 requires significantly fewer clock cycles to process the testbench (7,841 vs 11,132). This improvement is attributed to the Data Forwarding mechanism, which bypasses data from previous stages to the current instruction, thereby significantly reducing the frequency of pipeline Stalls and Flushes. The total number of executed instructions and branch instructions remains unchanged.

Consequently, the IPC of Model 2 improved to 0.62 (compared to 0.43 in Model 1). However, the Branch Misprediction Rate remains at 90.52%, as this model still relies on the static prediction scheme (Always Not Taken) and has not yet implemented dynamic Branch Prediction or an Always Taken strategy.

4. APPLICATION

4.1. Objective

The system continuously receives input values from the switches and then:

- Displays the decimal form on the seven-segment LEDs HEX0...HEX3.

- Displays the hexadecimal form on the seven-segment LEDs HEX4...HEX7.
- Allows turning on/off each display section using SW[14] (for hexadecimal) and SW[13] (for decimal).

4.2. Program Operation

Decimal conversion: Uses repeated division by 10 to extract each digit. Each digit is then mapped to the corresponding seven-segment LED code via map_hex_digit.

Hexadecimal conversion: Extracts 4-bit groups (nibbles) from the original input value (provided by the switches). The same map_hex_digit table used for the decimal part is applied here.

4.3. Assembly

```
# ============================================================
# HEX/DEC Converter - Displays 12-bit value in decimal and hex
# SW[11:0]: Input value | SW[12]: Master enable
# SW[13]: DEC enable | SW[14]: HEX enable
# ============================================================
_start:
    # Initialize peripheral addresses
    lui     s11, 0x10010  # Switch base
    lui     t4, 0x10002   # HEX0-3 base
    lui     t5, 0x10003   # HEX4-7 base
    lui     s9, 0x10000   # Red LED base

main_loop:
    lw      s8, 0(s11)    # Read switches
    # Display SW[14:0] on red LEDs
    addi    s10, s8, 0
    slli    s10, s10, 17
    srli    s10, s10, 17
    sw      s10, 0(s9)
    # Check master enable (SW[12])
    srli    t6, s8, 12
    andi    t6, t6, 1
    beq     t6, x0, disabled
    # Extract 12-bit value (SW[11:0])
    addi    t1, s8, 0
    slli    t1, t1, 20
    srli    t1, t1, 20
    # Initialize all displays to blank
    addi    s0, x0, 0xFF
    addi    s1, x0, 0xFF
    addi    s2, x0, 0xFF
```

```
    addi    s3, x0, 0xFF
    addi    s4, x0, 0xFF
    addi    s5, x0, 0xFF
    addi    s6, x0, 0xFF
    addi    s7, x0, 0xFF

    # ========== DECIMAL CONVERSION (HEX0-3) ==========
    addi    t2, t1, 0
    beq     t2, x0, dec_is_zero
    addi    t3, x0, 4      # Max 4 digits
    addi    a3, x0, 0      # Digit counter

dec_loop:
    beq     t2, x0, done_dec
    beq     a3, t3, done_dec
    # Divide by 10 (repeated subtraction)
    addi    a0, t2, 0      # dividend
    addi    a1, x0, 0      # quotient
    addi    a2, x0, 10     # divisor
div10_dec:
    bltu    a0, a2, enddiv_dec
    addi    a0, a0, -10
    addi    a1, a1, 1
    jal     x0, div10_dec
enddiv_dec:
    # a0 = remainder, a1 = quotient
    jal     ra, map_hex_digit  # Map to 7-segment
    # Store to s0-s3 based on position
    addi    a4, x0, 0
    beq     a3, a4, st_dec0
    addi    a4, x0, 1
    beq     a3, a4, st_dec1
    addi    a4, x0, 2
    beq     a3, a4, st_dec2
    addi    a4, x0, 3
    beq     a3, a4, st_dec3
    jal     x0, after_store_dec

st_dec0:
    addi s0, a0, 0
    jal x0, after_store_dec
st_dec1:
    addi s1, a0, 0
    jal x0, after_store_dec
st_dec2:
    addi s2, a0, 0
    jal x0, after_store_dec
st_dec3:
    addi s3, a0, 0
    jal x0,  after_store_dec

after_store_dec:
    addi    a3, a3, 1
```

```
85      addi    t2, a1, 0
86      jal     x0, dec_loop
87
88  dec_is_zero:
89      addi    s0, x0, 0xC0  # Display '0'
90      addi    a3, x0, 1
91
92  done_dec:
93      # ========== HEX CONVERSION (HEX4-7) ==========
94      # Extract nibbles and convert to 7-segment
95      andi    a0, t1, 0xF
96      jal     ra, map_hex_digit
97      addi    s4, a0, 0
98
99      srli    t6, t1, 4
100     andi    a0, t6, 0xF
101     jal     ra, map_hex_digit
102     addi    s5, a0, 0
103
104     srli    t6, t1, 8
105     andi    a0, t6, 0xF
106     jal     ra, map_hex_digit
107     addi    s6, a0, 0
108
109     srli    t6, t1, 12
110     andi    a0, t6, 0xF
111     jal     ra, map_hex_digit
112     addi    s7, a0, 0
113
114 pack_and_write:
115     # Pack HEX0-3 into 32-bit word
116     slli    t1, s3, 24
117     slli    t2, s2, 16
118     or      t1, t1, t2
119     slli    t2, s1, 8
120     or      t1, t1, t2
121     or      t1, t1, s0
122     # Pack HEX4-7 into 32-bit word
123     slli    t2, s7, 24
124     slli    t3, s6, 16
125     or      t2, t2, t3
126     slli    t3, s5, 8
127     or      t2, t2, t3
128     or      t2, t2, s4
129     # Check SW[14] (HEX enable)
130     srli    t6, s8, 14
131     andi    t6, t6, 1
132     bne     t6, x0, write_hex
133     li      t2, 0xFFFFFFFF    # Blank if disabled
134
135 write_hex:
136     # Check SW[13] (DEC enable)
137     srli    t6, s8, 13
```

```
    andi    t6, t6, 1
    bne     t6, x0, write_dec
    li      t1, 0xFFFFFFFF    # Blank if disabled

write_dec:
    sw      t1, 0(t4)         # Write to HEX0-3
    sw      t2, 0(t5)         # Write to HEX4-7
    jal     x0, main_loop

disabled:
    # Clear all displays
    li      t1, 0xFFFFFFFF
    sw      t1, 0(t4)
    sw      t1, 0(t5)
    jal     x0, main_loop
# ============================================================
# map_hex_digit: Convert 0-F to 7-segment code (active-low)
# Input: a0 = digit (0-15) | Output: a0 = 7-seg code
# ============================================================
map_hex_digit:
    addi    t0, x0, 0
    beq     a0, t0, md0
    addi    t0, x0, 1
    beq     a0, t0, md1
    addi    t0, x0, 2
    beq     a0, t0, md2
    addi    t0, x0, 3
    beq     a0, t0, md3
    addi    t0, x0, 4
    beq     a0, t0, md4
    addi    t0, x0, 5
    beq     a0, t0, md5
    addi    t0, x0, 6
    beq     a0, t0, md6
    addi    t0, x0, 7
    beq     a0, t0, md7
    addi    t0, x0, 8
    beq     a0, t0, md8
    addi    t0, x0, 9
    beq     a0, t0, md9
    addi    t0, x0, 10
    beq     a0, t0, mdA
    addi    t0, x0, 11
    beq     a0, t0, mdB
    addi    t0, x0, 12
    beq     a0, t0, mdC
    addi    t0, x0, 13
    beq     a0, t0, mdD
    addi    t0, x0, 14
    beq     a0, t0, mdE
    addi    t0, x0, 15
    beq     a0, t0, mdF

```

```
191     addi    a0, x0, 0x00
192     jalr    x0, ra, 0
193
194 # 7-segment codes (gfedcba format, active-low)
195 md0:
196     addi a0, x0, 0xC0
197     jalr x0, ra, 0
198 md1:
199     addi a0, x0, 0xF9
200     jalr x0, ra, 0
201 md2:
202     addi a0, x0, 0xA4
203     jalr x0, ra, 0
204 md3:
205     addi a0, x0, 0xB0
206     jalr x0, ra, 0
207 md4:
208     addi a0, x0, 0x99
209     jalr x0, ra, 0
210 md5:
211     addi a0, x0, 0x92
212     jalr x0, ra, 0
213 md6:
214     addi a0, x0, 0x82
215     jalr x0, ra, 0
216 md7:
217     addi a0, x0, 0xF8
218     jalr x0, ra, 0
219 md8:
220     addi a0, x0, 0x80
221     jalr x0, ra, 0
222 md9:
223     addi a0, x0, 0x90
224     jalr x0, ra, 0
225 mdA:
226     addi a0, x0, 0x88
227     jalr x0, ra, 0
228 mdB:
229     addi a0, x0, 0x83
230     jalr x0, ra, 0
231 mdC:
232     addi a0, x0, 0xC6
233     jalr x0, ra, 0
234 mdD:
235     addi a0, x0, 0xA1
236     jalr x0, ra, 0
237 mdE:
238     addi a0, x0, 0x86
239     jalr x0, ra, 0
240 mdF:
241     addi a0, x0, 0x8E
242     jalr x0, ra, 0
```

5. EVALUATION

The evaluation strategy comprised two primary phases: Functional Verification and Hardware Synthesis Analysis.

First, the design was validated using the isa_4b.hex testbench to ensure RV32I compliance. Simulation waveforms confirmed that the Non-Forwarding model correctly handles hazards via stalls, while the Forwarding model successfully bypasses data to minimize delays. Additionally, a Hex-to-Decimal Converter application was deployed to benchmark I/O, branching, and arithmetic capabilities, validating the integration of the Datapath, Control Unit, and LSU in a real-world scenario.

Regarding performance, the Pipelined architecture (Milestone 3) was compared against the Single-Cycle design (Milestone 2). Theoretically, pipelining increases logic resource usage but significantly improves the maximum operating frequency (F_{max}) by reducing the critical path. Furthermore, the Forwarding model demonstrated a higher Instruction Per Cycle (IPC) compared to the Non-Forwarding model, proving that the added hardware complexity effectively maximizes instruction throughput.

6. RESULT

6.1. Quartus

Model 1: Non-Forwarding		
Maximum Frequency	57.02 MHz	
Logic Utilization/Total logic elements	4046	12%
Total registers	2455	
Model 2: Forwarding		
Maximum Frequency	48.26 MHz	
Logic Utilization/Total logic elements	4165	13%
Total registers	2438	

Table 12: Results in Quartus

6.2. Performance

6.2.1.IPC

Benchmark	Model 1	Model 2
Instruction	4831	4831
Cycle	11132	7841
IPC	0.43	0.62

Table 13: IPC Results

6.2.2.Branch Prediction

Benchmark	Model 1	Model 2
Branch Instructions	1604	1604
Branch Mispredictions	1452	1452
Branch Misprediction Rate	90.52%	90.52%

Table 14: Branch Prediction Results

6.3. Demo DE2

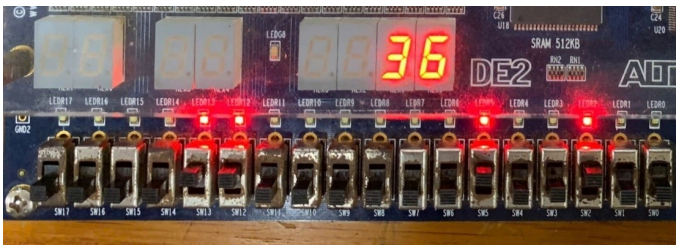

Figure 38: Turn on the Decimal LEDs – turn off the Hexadecimal LEDs

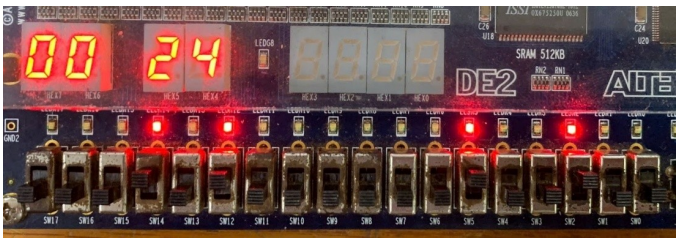

Figure 39: Turn on the Hexadecimal LEDs – turn off the Decimal LEDs

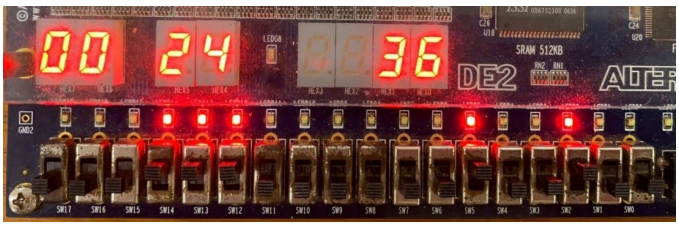

Figure 40: Turn on both the Decimal and Hexadecimal LEDs

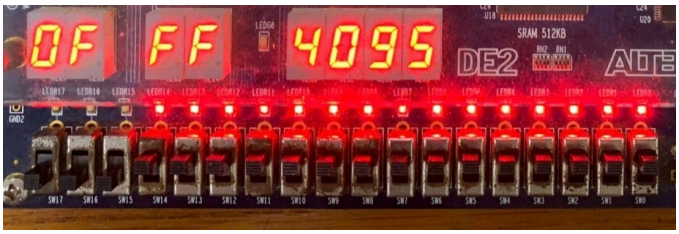

Figure 41: Turn on all Switches

Video demo trên FPGA: ***KTMT_L02_7_Application***

REFERENCES

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition*, Morgan Kaufmann, 2020.

[2] S. L. Harris and D. Harris, *Digital Design and Computer Architecture: RISC-V Edition*, Morgan Kaufmann, 2021.

[3] Dan Garcia, "Pipeline I", Great Ideas in Computer Architecture (Machine Structures), University of California, Berkeley, 2020.

[4] Dan Garcia, "Pipeline II: Control Hazards, Data Hazards I", Great Ideas in Computer Architecture (Machine Structures), University of California, Berkeley, 2020.

[5] Dan Garcia, "Pipeline III: More Hazards, Superscalar Processors", Great Ideas in Computer Architecture (Machine Structures), University of California, Berkeley, 2020.