ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA ĐIỆN – ĐIỆN TỬ

----------oOo----------

BÁO CÁO THỰC TẬP NGOÀI TRƯỜNG

DESIGN AND IMPLEMENTATION OF A PIPELINED RISC-V PROCESSOR WITH RV32IM ISA

GVHD: TS. Trần Hoàng Linh

Ký tên:

STT	HỌ VÀ TÊN	MSSV
1	NGUYỄN TRỊNH THÀNH TRUNG	2213703

Tp. Hồ Chí Minh, ngày 11 tháng 8 năm 2025

MỤC LỤC

Design of a Single Cycle RISC-V Processor 4

1.Introduction 4

1.1.Review of RISC-V Processor 4

1.2.Understading The RISC-V Intrucstion Set Architecture 4

1.3.Design and Stimulation. 5

1.3.1.R-Format Instructions Group: 6

1.3.2. I-Format Instruction Group 8

1.3.3.L-Format Instruction Group 8

1.3.4. S-Format Instruction Group 9

1.3.5. B-Format Instruction Group 10

1.3.6. U-Format Instruction Group 11

1.3.7. J-Format Instruction Group 12

1.3.8. IM-Format Instruction Group 12

2.Analyses 14

2.1.Arithmetic Logic Unit (ALU) 14

2.2.Branch Comparison Unit (BRC) 16

2.3.Regfile 17

2.4.Load Store Unit (LSU) 18

2.5. Control Unit 20

2.6.Immidiate Generation (Immgen) 21

2.7.Single-Cycle Processor 21

3.Results 23

3.1.Verification via Output Functionality Testing 23

3.1.1.Testing result on server 23

Design of a Pipeline RISC-V Processor 26

1.Develop a Pipelined Processor from the previously designed Single-Cycle Processor. 26

2.Analyses 27

2.1. Non - Forwarding 27

2.2.Forwarding 30

2.3.Two – Bit Prediction: 31

3.Design 34

3.1.Non – Forwarding Pipeline Processor 34

3.1.1.Module: hazard_unit 35

3.1.2.Module: IF_ID 36

3.1.3.Module: ID_EX 36

3.1.4.Module : EX_MEM 37

3.1.5.Module: MEM_WB 37

3.2.Forwarding Pipeline Processor 38

3.2.1. Module: forward_unit 39

3.3.Two – Bit Pipeline Processor 40

4.Results 46

4.1.Functional Verification Methodology (Verification plan) 46

4.2.Verification via Output Functionality Testing 46

4.2.1 Non – Forwarding Pipeline Processor 46

4.2.2. Forwarding Pipeline Processor 49

4.2.3. Two – Bit Pipeline Processor 51

6.FPGA Implementation 52

6.1. Testing result on server 52

6.2. Testing result on FPGA 53

Design of a Single Cycle RISC-V Processor

1. Introduction

1.1. Review of RISC-V Processor

RISC (Reduced Instruction Set Computer) refers to computer architectures designed to execute simple, individual instructions efficiently. Unlike CISC (Complex Instruction Set Computer), which handles complex, multi-task instructions, RISC uses uniform, single-task instructions that improve performance and simplify hardware design.

RISC-V is a prominent example of RISC architecture. Developed in 2010 by the University of California, Berkeley, RISC-V is an open-source, royalty-free instruction set architecture. Unlike proprietary architectures like ARM, RISC-V is freely available for anyone to use.

It follows a load-store model with fixed 32-bit instructions and supports variable-length extensions. RISC-V scales across a wide range of systems—from embedded devices to supercomputers—and offers 32-bit, 64-bit, and 128-bit address spaces.

Thanks to its flexibility and open nature, RISC-V has become a popular choice for companies looking to innovate and reduce development costs.

1.2. Understading The RISC-V Intrucstion Set Architecture.

The instruction set of RV32I (RISC-V 32-bit Integer Base Instruction Set) is the most basic set of instructions in the RISC-V architecture, with "32" indicating the size of each word (32 bits) and "I" representing "Integer." This signifies that this is the 32-bit instruction set designed for integer operations. Each instruction consists of an opcode, source register addresses, and a destination register address, allowing for the execution of arithmetic, logical, control flow, and memory access operations.

RV32I includes the following main types of instructions:

- Arithmetic and Logic Instructions: Instructions like ADD, SUB, AND, OR, and XOR perform operations on integers and bits.

- Control Flow Instructions: Instructions such as BEQ, BNE, JAL, and JR control the flow of program execution.
- Memory Access Instructions: Instructions like LW and SW perform reading and writing of data to and from memory.

1.3. Design and Stimulation.

Based on the RV32I instruction set, we can develop a comprehensive architectural model consisting of key components including the ALU, Register File, Control Unit, ImmGen, Branch Comparison, Load Store Unit, and several other elements. Each component is designed to meet specific requirements.

- **ALU (Arithmetic Logic Unit)** is a crucial component of the CPU, responsible for performing arithmetic, comparison, and logical operations. It is an essential part of any microprocessor architecture. In our design, the ALU supports all the basic arithmetic and logic instructions of RV32I.
- **Register File** serves the purpose of temporarily storing the data and addresses necessary for executing operations.
- **Control Unit** analyzes the opcode of each instruction and generates the control signals required to route data and execute the instruction.
- **Load Store Unit (LSU)** is responsible for managing memory access operations, specifically the load and store operations.
- **ImmGen** is tasked with generating immediate values from the encoded instructions in the microprocessor.
- **Branch Comparison** is an essential part that helps control the program's execution flow based on specific conditions.

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 1.1: Instruction Set

The instruction set of RISC-V is also called the load-store instruction set, which means that data in memory must first be loaded into the register file before it can be executed. After computation, the data is then stored back into memory.

The registers in the register bank are 32 bits wide, and there are 32 registers, so 5 bits are required to specify the address of the registers in the register bank.

Note: The register x0 always has a value of 0x00000000 and its value does not change.

1.3.1. R-Format Instructions Group:

This instruction group includes instructions with the structure shown in the diagram below:

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 1.2: R-Format Instruction Group

The opcode for this instruction group is [6:0] = 0110011.

This instruction group performs the operation of taking two values stored in registers rs1 and rs2, passing them into the ALU for computation, and then storing the result in register rd.

The R-type instructions in the RISC-V architecture are designed to perform arithmetic and logical operations on input values. The input and output parameters are stored in registers. Below is the list of R-type instructions in the RISC-V architecture:

ADD: Adds two numbers and stores the result in the destination register.

SUB: Subtracts two numbers and stores the result in the destination register.

AND: Performs a logical AND operation between two numbers and stores the result in the destination register.

OR: Performs a logical OR operation between two numbers and stores the result in the destination register.

XOR: Performs a logical XOR operation between two numbers and stores the result in the destination register.

SLL: Performs a left shift on an integer and stores the result in the destination register.

SRL: Performs a right shift on an integer, without sign extension, and stores the result in the destination register.

SRA: Performs a right shift on an integer, with sign extension, and stores the result in the destination register

SLT: Compares two signed numbers and stores the result in the destination register.

SLTU: Compares two unsigned numbers and stores the result in the destination register.

1.3.2. I-Format Instruction Group

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

Figure 1.3: I-Format Instruction

This instruction group has the opcode [6:0] = 0010011.

This group of instructions (except for the three instructions SRAI, SRLI, and SLLI) takes the value stored in register rs1 and the value stored in imm[11:0] (which is sign-extended), then sends them to the ALU for computation. The result is stored in the rd register

1.3.2. L-Format Instruction Group

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

Figure 1.4: L-Format Instruction

The L instruction group is used to load data from memory into a register. This group of instructions has a format with two operands, where the first operand is the destination register where the value will be loaded, and the second operand is the memory address.

This instruction group has the opcode [6:0] = 0000011.

It takes the two values stored in register rs1 and the value stored in imm[11:0] (which is sign-extended), computes the sum of rs1 + ext(imm[11:0]), and then loads the value from DMEM at the address rs1 + ext(imm[11:0]) into the rd register.

The L group instructions in the RISC-V architecture include:

- **LB**: Load one byte from memory.
- **LH**: Load half-word from memory.
- **LW**: Load word from memory.
- **LD**: Load double-word from memory (only used with a 64-bit architecture).
- **LBU**: Load unsigned byte from memory.
- **LHU**: Load unsigned half-word from memory.
- **LWU**: Load unsigned word from memory (only used with a 64-bit architecture).

1.3.4. S-Format Instruction Group

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

Figure 1.5: S-Format Instruction

This instruction group has the opcode [6:0] = 0100011.

This group of instructions performs the following steps: it takes the values stored in register rs1 and the values stored in imm[11:5] and imm[4:0] (which are concatenated and sign-extended) to compute the sum of rs1 + ext(imm[11:5]imm[4:0]). Then, it stores the value from register rs2 into DMEM at

the address rs1 + ext(imm[11:5]imm[4:0]).

The S group instructions in the RISC-V architecture include:

SB: Store one byte into memory.

SH: Store half-word into memory.

SW: Store word into memory.

SD: Store double-word into memory (only used with a 64-bit architecture).

Note that the S instruction group consists of instructions that store data at the destination address (the memory address is referenced by rs1, and the value to be stored is referenced by rs2).

1.3.5. B-Format Instruction Group

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Figure 1.6: B-Format Instruction

This instruction group has the opcode [6:0] = 1100011, and func3 is used to determine the type of jump condition, with the following values:

000: Jump if equal (=).

001: Jump if not equal (<>).

100: Jump if less than (<).

101: Jump if less than or equal to (<=).

110: Jump if greater than or equal to (>=).

111: Jump if greater than (>).

This instruction group will transfer the value of the PC register to the value stored in the imm field, and the values stored in rs1 and rs2 will satisfy the condition specified by the instruction (equal, not equal, greater than or equal to, etc.).

When retrieving the value from the imm field, we must concatenate the bits in the correct order and sign-extend. The LSB bit is always 0.

The B-type instructions in the RISC-V architecture include:

BEQ: Jump if the two registers are equal.

BNE: Jump if the two registers are not equal.

BLT: Jump if the first register is less than the second register.

BGE: Jump if the first register is greater than or equal to the second register.

BLTU: Jump if the first register is less than the second register, unsigned.

BGEU: Jump if the first register is greater than or equal to the second register, unsigned.

1.3.6. U-Format Instruction Group

imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC

Figure 1.7: U-Format Instruction

The U-type instructions are used to initialize values for registers with static, globally known immediate values. These instructions perform a calculation where the immediate value is added to the current value of the PC (Program Counter) to compute the destination register value.

The instructions in the U-type group include:

LUI (Load Upper Immediate): This instruction is used to initialize the highest 20 bits of the destination register with a global value taken from the lower 20 bits of the instruction.

Example: LUI x1, 0xFFFFF – This instruction will load the value 0xFFFFF into the highest 20 bits of register x1.

AUIPC (Add Upper Immediate to PC): This instruction is used to initialize the highest 20 bits of the destination register with a global value taken from the lower 20 bits of the instruction, added to the current value of the PC (Program Counter).

Example: AUIPC x1, 0x10000 – This instruction will load the value (0x10000 + PC) into the highest 20 bits of register x1.

Using the U-type instructions allows for quick and convenient initialization of register values. In particular, the **AUIPC** instruction is commonly used to compute absolute memory addresses for load or store instructions because it adds the current value of the PC (the address of the current instruction) to the immediate value.

1.3.7. J-Format Instruction Group

imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR

Figure 1.8: J-Format Instruction

The J-type instructions are a group of unconditional jump instructions used to jump to a specific instruction location within a program.

The instructions in the J-type group include:

JAL (Jump and Link): This is an unconditional jump instruction that also updates the value of the destination register rd to store the return address, so that it can return to the next instruction after the function call. The JAL instruction has a 20-bit jump target address field, which is stored in the lower 12 bits and the highest 8 bits of the destination register rd.

JALR (Jump and Link Register): This is another unconditional jump instruction that updates the return address value. JALR jumps to the address stored in the source register rs1, along with a specified 12-bit offset, and then updates the return address into the destination register rd.

Unconditional jump instructions like JAL and JALR are often used in specific scenarios such as function calls, loops with many iterations, or switch-case statements in programming languages.

1.3.8. IM-Format Instruction Group

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

The IM instruction group is used to perform integer multiplication and division

operations between two registers in the RISC-V architecture. These instructions use the standard R-type format, where the first operand is the destination register (rd) that will store the result, and the other two operands (rs1 and rs2) are the source registers.

This instruction group has the opcode [6:0] = 0110011 and a fixed funct7 field of 0000001 to indicate that the instruction belongs to the M-extension. The specific operation (multiply, divide, or remainder) is selected by the funct3 field.
The processor reads the values from registers rs1 and rs2, then performs the selected arithmetic operation (multiplication, division, or remainder) and writes the result to the rd register. For multiplication-high variants (MULH, MULHSU, MULHU), the high 32 bits of the 64-bit product are stored in rd.
The IM group instructions in the RV32IM architecture include:
MUL: Multiply two signed integers, store the lower 32 bits of the result.
MULH: Multiply two signed integers, store the upper 32 bits of the result.
MULHSU: Multiply a signed integer and an unsigned integer, store the upper 32 bits.
MULHU: Multiply two unsigned integers, store the upper 32 bits.
DIV: Divide two signed integers, store the quotient.
DIVU: Divide two unsigned integers, store the quotient.
REM: Divide two signed integers, store the remainder.
REMU: Divide two unsigned integers, store the remainder.

2. Analyses

2.1. Arithmetic Logic Unit (ALU)

1. Requirement:

The ALU must be capable of executing a variety of arithmetic and logical operations as defined by the RV32I instruction set.

alu_op	Description (R-type)	Description (I-type)
ADD	rd ← rs1 + rs2	rd ← rs1 + imm
SUB	rd ← rs1 − rs2	n/a
SLT	rd ← (rs1 < rs2)? 1 : 0	rd ← (rs1 < imm)? 1 : 0
SLTU	rd ← (rs1 < rs2)? 1 : 0	rd ← (rs1 < imm)? 1 : 0
XOR	rd ← rs1 ⊕ rs2	rd ← rs1 ⊕ imm
OR	rd ← rs1 ∨ rs2	rd ← $rs1 \vee imm$
AND	$rd \leftarrow rs1 \wedge rs2$	rd ← $rs1 \wedge imm$
SLL	rd ← rs1 << rs2[4:0]	rd ← rs1 << imm[4:0]
SRL	rd ← rs1 >> rs2[4:0]	rd ← rs1 >> imm[4:0]
SRA	rd ← rs1 >>> rs2[4:0]	rd ← rs1 >>> imm[4:0]
MUL	rd = (rs1 * rs2)[31:0]	n/a
MULH	rd = (rs1 * rs2)[63:32]	n/a
MULHSU	rd = signed(rs1) * unsigned(rs2) >> 32	n/a
MULHU	rd = (rs1 * rs2)[63:32]	n/a
DIV	rd = rs1 / rs2	n/a
DIVU	rd = unsigned(rs1) / unsigned(rs2)	n/a
REM	rd = rs1 % rs2	n/a
REMU	rd = unsigned(rs1) % unsigned(rs2)	n/a

Note: Do not use built-in SystemVerilog operators for subtraction (−), comparison (<, >), or shifting (<<, >>, and >>>).

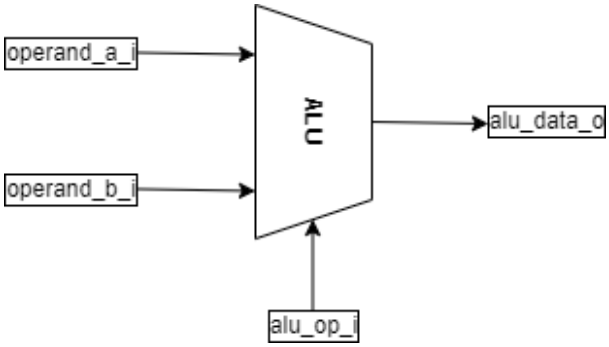
2. **Module name:** alu.sv

3. I/O ports:

Signal Name	Width	Direction	Description
i_operand_a	32	Input	First operand for ALU operations.
i_operand_b	32	Input	Second operand for ALU operations.
i_alu_op	6	Input	The operation to be performed.
o_alu_data	32	Output	Result of the ALU operation.

4. Design

The ALU design will be based on the following block diagram.



SLL

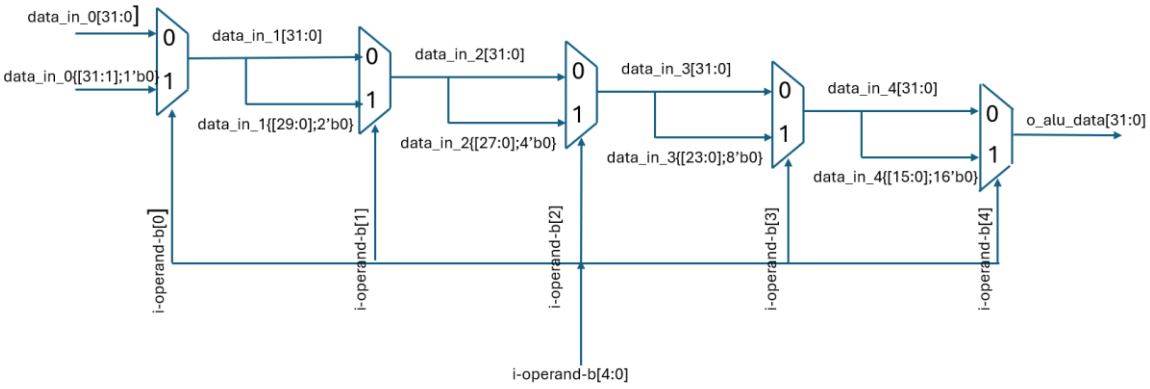


Figure 1.9: SLL

SRL

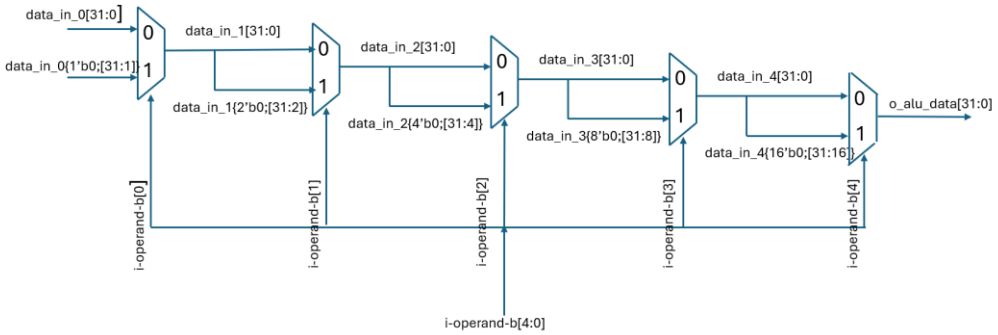


Figure 1.10: SRL

SRA

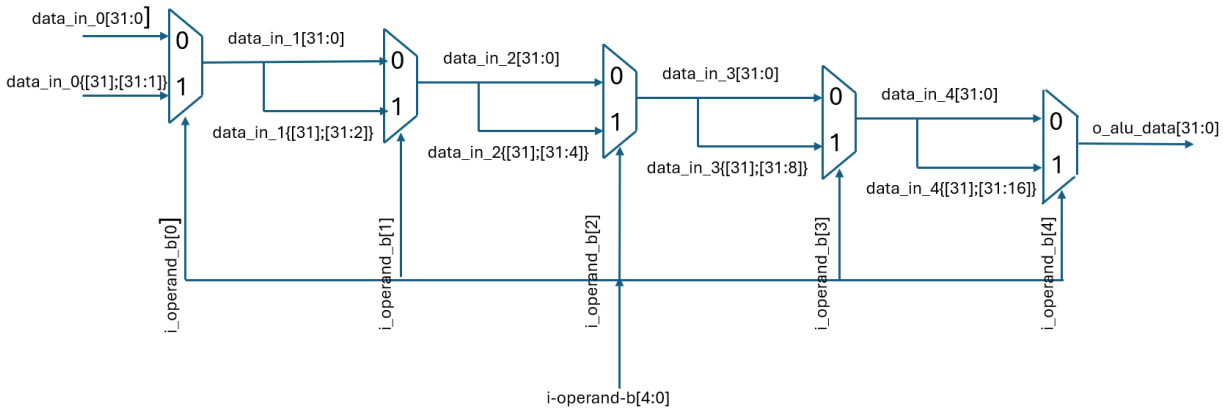


Figure 2.1: SRA

2.2. Branch Comparison Unit (BRC)

1. Requirement:

This unit is responsible for comparing two registers to determine the outcome of branch instructions. The unit should be capable of handling both signed and unsigned comparisons.

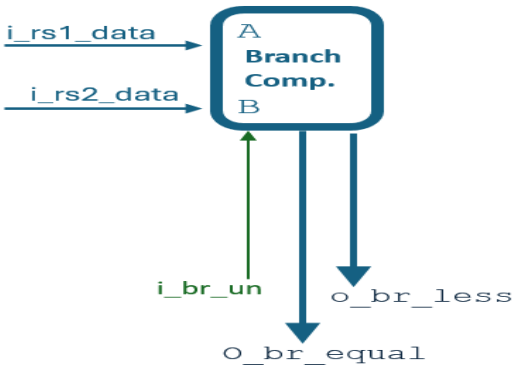
Note: Do not use built-in SystemVerilog operators for subtraction (−), comparison (<, >), or shifting (<<, >>, and >>>).

2. **Module name:** brc.sv

3. I/O ports:

Signal Name	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 unsigned, 0 if signed).
o_br_less	1	Output	Output is 1 if rs1 < rs2.
o_br_equal	1	Output	Output is 1 if rs1 = rs2.

4. Design



a. Input :

Two data busses **A** and **B**

i_br_un ("Branch Unsigned") control bit

b. Output:

o_br_equal: 1 if A == B

o_br_less:1 if A < B.

- Unsigned comparison if i_br_un=1, signed otherwise.

2.3. Regfile

1. Requirement:

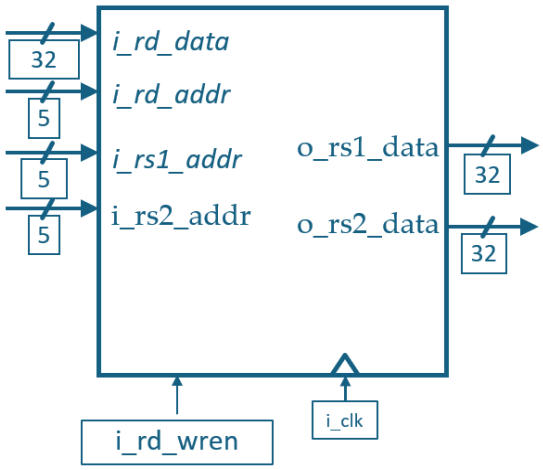
Implement a register file with 32 registers, each 32-bit wide. The register file must have two read ports and one write port, with register 0 always reading as zero.

2. **Module name:** Regfile.sv

3. I/O ports:

Signal Name	Width	Direction	Description
i_clk	1	Input	Global clock.
i_rst	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.

4. Design



- Registers are accessed via their 5-bit register numbers:
- i_rs1_addr selects register to put on_o_rs1_data bus out.
- i_rs2_addr selects register to put on_o_rs2_data bus out.
- i_rd_addr selects register to be written via i_rd_data when i_rd_wren = 1.

2.4. Load Store Unit (LSU)

In real-world applications, a processor interfaces with peripheral devices to either transmit data or receive data through the implementation of an Input/Output (I/O) system. Common peripheral devices include LEDs, LCDs, and switches, among others. These peripherals essentially function as a form of "memory" or "registers."

For instance, when a 32-bit register is linked to a set of 32 LEDs, depositing data into that register results in manipulating the state of the LED array. Memory mapping is a strategic method used to organize the layout of memory, with different memory regions serving specific functions. Design a Load Store Unit (LSU) divided into areas including: data, input peripherals, and output peripherals

1. Requirements

Implement a Load-Store Unit (LSU) to manage memory-mapped in Table 1.

Boundary Address Mapping:

Address Range	Description
0x7820 -- 0xFFFF	Reserved
0x7810 -- 0x781F	Buttons
0x7800 -- 0x780F	Switches (required)
0x7040 -- 0x70FF	Reserved
0x7030 -- 0x703F	LCD Control Registers
0x7020 -- 0x7027	Seven-segment LEDs
0x7010 -- 0x701F	Green LEDs (required)
0x7000 -- 0x700F	Red LEDs (required)
0x4000 -- 0x6FFF	Reserved
0x2000 -- 0x3FFF	Data Memory (8KiB using SDRAM) (required)
0x0000 -- 0x1FFF	Instruction Memory (8KiB) (required)

Table 1: LSU mapping

2. **Module name**: lsu
3. **I/O ports**:

Signal Name	Width	Direction	Description
i_clk	1	Input	Global clock, active on the rising edge.
i_rst	1	Input	Global active reset.
i_lsu_addr	32	Input	Address for data read/write.
i_st_data	32	Input	Data to be stored.
i_lsu_wren	1	Input	Write enable signal (1 if writing).
o_ld_data	32	Input	Data read from memory.
o_io_ledr	32	Input	Output for red LEDs.
o_io_ledg	32	Input	Output for green LEDs.
o_io_hex0..7	7	Input	Output for 7-segment displays.
o_io_lcd	32	Input	Output for the LCD register.
i_io_sw	32	Input	Input for switches.
i_io_btn	4	Input	Input for buttons.

4. **Design**

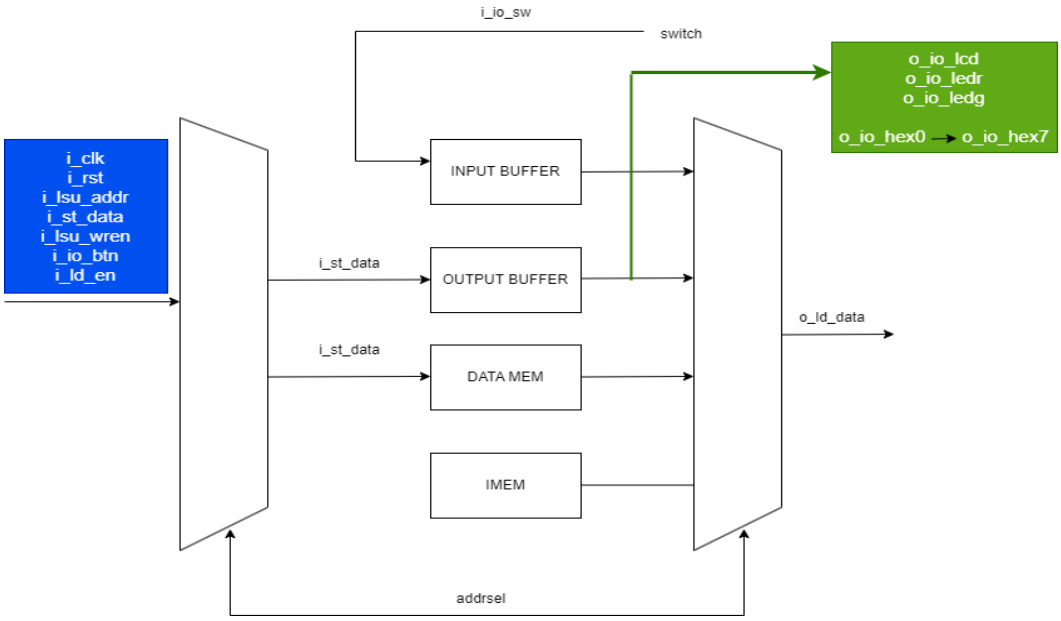


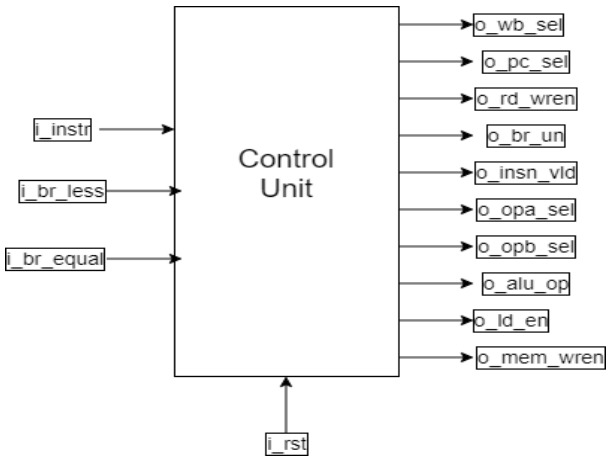
Figure 2.4.1. The block diagram of the Load Store Unit

2.5. Control Unit

1. Module name: ctrl_unit.sv.

Signal	Source	Size	Description
i_instr	Input	32	Instruction with a size of 32 bits
i_br_less	Input	1	1 if rs1 < rs2
i_br_equal	Input	1	1 if rs1 = rs2
o_pc_sel	Output	1	0 for PC + 4, 1 for alu_data
o_br_un	Output	1	1 if both operands are unsigned
o_rd_wren	Output	1	Write enable for the destination register
o_op_a_sel	Output	1	0 for rs1, 1 for PC
o_op_b_sel	Output	1	0 for rs2, 1 for imm
o_mem_wren	Output	1	1 if data is written to LSU
o_alu_op	Output	4	Operation code for ALU execution
o_wb_sel	Output	4	0: alu_data, 1: ld_data, 2 or 3: pc_four
o_ld_en	Output	3	represent the load-enable signal based on the funct3
o_insn_vld	Output	1	1 means the instruction is valid, 0 is invalid

2.Design



Block diagram of Control Unit

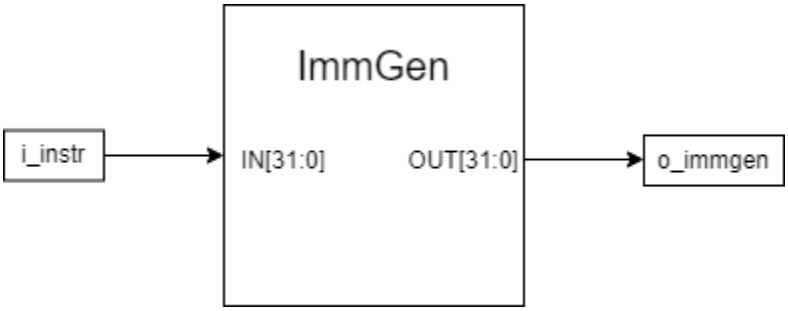
2.6. Immidiate Generation (Immgen)

The module immgen is responsible for generating the immediate value based on the instruction provided.

1. **Module name:** immgen.sv
2. **I/O ports:**

Signal Name	Width	Direction	Description
i_instr	32	Input	The instruction from which the immediate value is extracted.
o_immgen	32	output	The generated immediate value based on the instruction.

3. **Design**



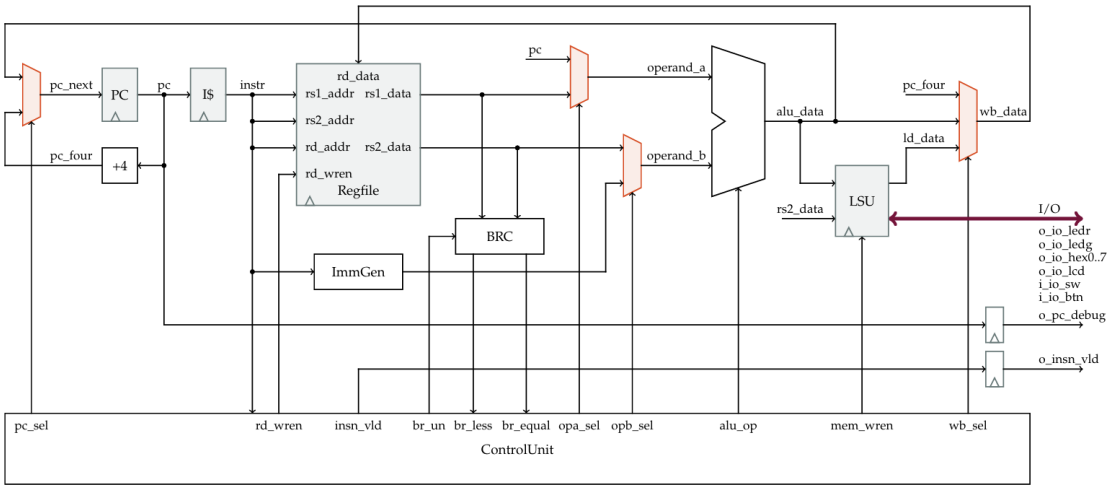
Block diagram of Immediate Generator

2.7. Single-Cycle Processor

The input and output signals in the design of the Single-cycle processor are presented in the following table:

Signal	Source	Size	Description
i_clk	input	1	Clock signal
i_rst_n	input	1	Reset signal
i_io_btn	input	4	a value of 1 could represent a pressed state, and 0 could represent a released state.
i_io_sw	input	32	Communicate with switch
o_pc_debug	output	32	Provides the current value of the Program Counter (PC)
o_insn_vld	output	1	1 means the instruction is valid, 0 is invalid
o_io_ledr	output	32	Bit for displaying on external red LEDs.
o_io_ledg	output	32	Bit for displaying on external green LEDs.
o_io_hex0	output	7	Value displayed on segment 0 of the 7-segment LED display.
o_io_hex1	output	7	Value displayed on segment 1 of the 7-segment LED display.
o_io_hex2	output	7	Value displayed on segment 2 of the 7-segment LED display.
o_io_hex3	output	7	Value displayed on segment 3 of the 7-segment LED display.
o_io_hex4	output	7	Value displayed on segment 4 of the 7-segment LED display.
o_io_hex5	output	7	Value displayed on segment 5 of the 7-segment LED display.
o_io_hex6	output	7	Value displayed on segment 6 of the 7-segment LED display.
o_io_hex7	output	7	Value displayed on segment 7 of the 7-segment LED display.
o_io_lcd	output	32	Value displayed on LCD

The input and output signals in the design of the Load Store Unit



Block diagram of Single-Cycle Processor

3. Results

3.1. Verification via Output Functionality Testing

3.1.1. Testing result on server

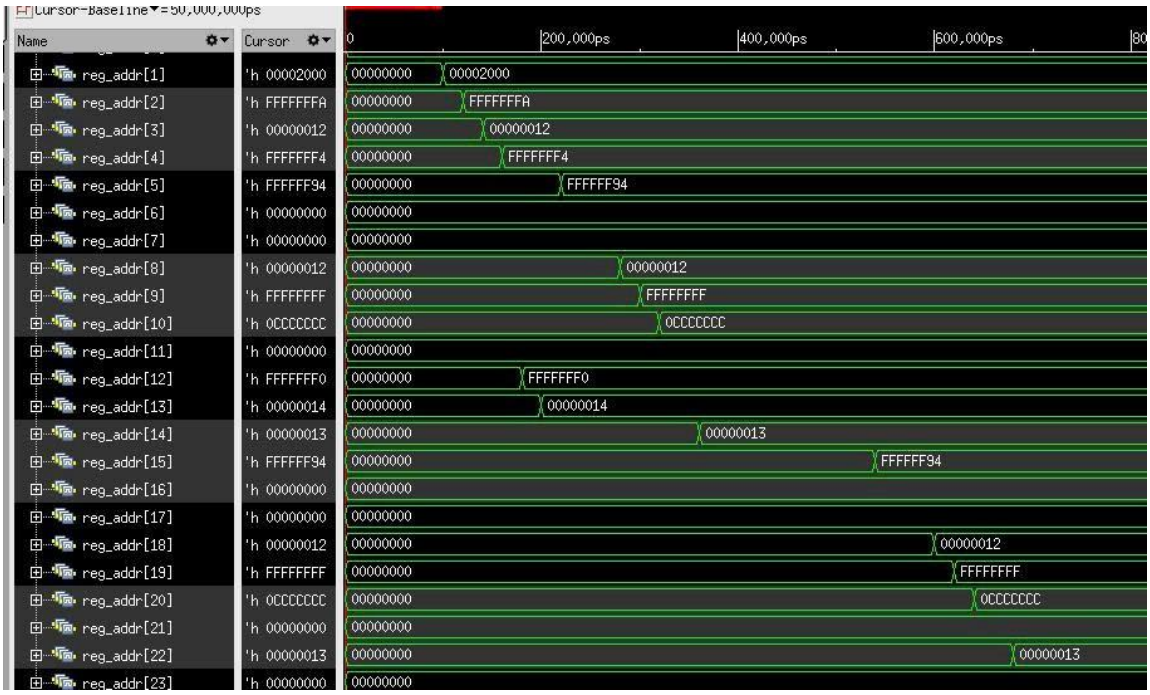
```
module scoreboard(
  input  logic [31:0]       i_io_sw  ,
  input  logic [31:0]       o_io_lcd ,
  input  logic [31:0]       o_io_ledg,
  input  logic [31:0]       o_io_ledr,
  input  logic [ 6:0]       o_io_hex0,
  input  logic [ 6:0]       o_io_hex1,
  input  logic [ 6:0]       o_io_hex2,
  input  logic [ 6:0]       o_io_hex3,
  input  logic [ 6:0]       o_io_hex4,
  input  logic [ 6:0]       o_io_hex5,
  input  logic [ 6:0]       o_io_hex6,
  input  logic [ 6:0]       o_io_hex7,
  input  logic [31:0]       o_pc_debug,
  input  logic              o_insn_vld,
  input  logic i_clk,
  input  logic i_reset
);

  logic [31:0] pc_debug;

  assign pc_debug = o_insn_vld ? o_pc_debug : 32'h0;

  logic insn_vld;

  always_comb begin
    case (o_pc_debug)
      32'h0004: begin
        $display("TEST for SINGLECYCLE");
        insn_vld = 1;
      end

  32'h0020:                                                                 $write("[%9t]:: 1::ADD.............", $time);
  32'h0108: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 2::SUB.............", $time); end
  32'h0200: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 3::XOR.............", $time); end
  32'h02E4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 4::OR..............", $time); end
  32'h03CC: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 5::AND.............", $time); end
  32'h04B4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 6::SLL.............", $time); end
  32'h057C: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 7::SRL.............", $time); end
  32'h0660: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 8::SRA.............", $time); end
  32'h073C: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]:: 9::SLT.............", $time); end
  32'h0808: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::10::SLTU............", $time); end
  32'h08D4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::11::ADDI............", $time); end
  32'h0964: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::12::XORI............", $time); end
  32'h09F4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::13::ORI.............", $time); end
  32'h0A80: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::14::ANDI............", $time); end
  32'h0B04: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::15::SLLI............", $time); end
  32'h0B94: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::16::SRLI............", $time); end
  32'h0C34: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::17::SRAI............", $time); end
  32'h0CD8: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::18::SLTI............", $time); end
  32'h0D50: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::19::SLTIU...........", $time); end
  32'h0DC8: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::20::LUI.............", $time); end
  32'h0E34: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::21::AUIPC...........", $time); end
  32'h0E80: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::22::LW............", $time); end
  32'h0F34: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::23::LH............", $time); end
  32'h0FB4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::24::LB............", $time); end
  32'h1024: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::25::LHU...........", $time); end
  32'h10A8: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::26::LBU...........", $time); end
  32'h1118: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::27::SW............", $time); end
  32'h1180: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::28::SH............", $time); end
  32'h1210: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::29::SB............", $time); end
  32'h1298: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::30::misaligned.....", $time); end
  32'h12F0: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::31::BEQ...........", $time); end
  32'h1328: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::32::BNE...........", $time); end
  32'h1360: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::33::BLT...........", $time); end
  32'h13C8: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::34::BGE...........", $time); end
  32'h1434: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::35::BLTU..........", $time); end
  32'h149C: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::36::BGEU..........", $time); end
  32'h1508: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::37::JAL...........", $time); end
  32'h15C4: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::38::JALR..........", $time); end
  32'h1664: begin if (o_io_ledr == 32'h1) $write("PASSED\n"); else $write("FAILED\n"); $write("[%9t]::39::illegal_insn....", $time); end
```

```
xmsim: *W,DSEM2009: This SystemVerilog design
xcelium> source /opt/cadence/XCELIUM2009/tools/
xcelium> run
TEST for SINGLECYCLE
[     1800]:: 1::ADD.............PASSED
[    12600]:: 2::SUB.............PASSED
[    24200]:: 3::XOR.............PASSED
[    34800]:: 4::OR..............PASSED
[    45600]:: 5::AND.............PASSED
[    56400]:: 6::SLL.............PASSED
[    65600]:: 7::SRL.............PASSED
[    76200]:: 8::SRA.............PASSED
[    86400]:: 9::SLT.............PASSED
[    95800]::10::SLTU............PASSED
[   105200]::11::ADDI............PASSED
[   111600]::12::XORI............PASSED
[   118000]::13::ORI.............PASSED
[   124200]::14::ANDI............PASSED
[   130000]::15::SLLI............PASSED
[   136400]::16::SRLI............PASSED
[   143600]::17::SRAI............PASSED
[   151000]::18::SLTI............PASSED
[   156200]::19::SLTIU...........PASSED
[   161400]::20::LUI.............PASSED
[   166000]::21::AUIPC...........PASSED
[   169000]::22::LW..............PASSED
[   177200]::23::LH..............PASSED
[   182800]::24::LB..............PASSED
[   187600]::25::LHU.............PASSED
[   193400]::26::LBU.............PASSED
[   198200]::27::SW..............PASSED
[   202600]::28::SH..............PASSED
[   209000]::29::SB..............PASSED
[   215000]::30::misaligned.....PASSED
[   218600]::31::BEQ.............PASSED
[   220800]::32::BNE.............PASSED
[   223000]::33::BLT.............PASSED
[   227000]::34::BGE.............PASSED
[   231000]::35::BLTU............PASSED
[   235000]::36::BGEU............PASSED
[   239000]::37::JAL.............PASSED
[   242000]::38::JALR............PASSED
[   245000]::39::illegal_insn....PASSED

Timeout...

DUT is considered       P A S S E D
```

Test for IM-Format Instruction Group

```
lui   x1, 2                # x1 = 0x2000 (base address for memory ops)

    # Test dữ liệu signed
    addi  x2, x0, -6             # x2 = -6  (signed)
    addi  x3, x0, 18             # x3 = 18  (signed)
    addi  x4, x0, -12            # x4 = -12 (signed)

    # Test dữ liệu unsigned (lấy giá trị lớn để test overflow)
    li    x12, 0xFFFFFFF0        # x12 = 4294967280 (unsigned ~ -16 signed)
    li    x13, 20                # x13 = 20 (unsigned)

    # ---- Signed MUL / DIV ----
    mul   x5,  x3, x2            # MUL signed × signed (18 * -6)
    mulh  x6,  x4, x2            # MULH signed × signed (-12 * -6)
    div   x7,  x3, x2            # DIV signed ÷ signed (18 / -6)
    rem   x8,  x3, x2            # REM signed % signed (18 % -6)

    # ---- Mixed signed/unsigned ----
    mulhsu x9,  x4, x13          # MULHSU signed × unsigned (-12 * 20)
    divu   x10, x12, x13         # DIVU unsigned ÷ unsigned (0xFFFFFFF0 / 20)
    remu   x11, x12, x13         # REMU unsigned % unsigned (0xFFFFFFF0 % 20)

    # ---- Unsigned MUL ----
    mulhu  x14, x12, x13         # MULHU unsigned × unsigned

    # ---- Lưu kết quả ra bộ nhớ ----
    sw    x5,   0(x1)
    sw    x6,   4(x1)
    sw    x7,   8(x1)
    sw    x8,  12(x1)
    sw    x9,  16(x1)
    sw    x10, 20(x1)
    sw    x11, 24(x1)
    sw    x14, 28(x1)
```

```
# ---- Đọc lại để test ----
lw    x15,  0(x1)    # MUL
lw    x16,  4(x1)    # MULH
lw    x17,  8(x1)    # DIV
lw    x18, 12(x1)    # REM
lw    x19, 16(x1)    # MULHSU
lw    x20, 20(x1)    # DIVU
lw    x21, 24(x1)    # REMU
lw    x22, 28(x1)    # MULHU

nop
nop
```


Cursor-Baseline=50,000,000ps
Name
Cursor
0
200,000ps
400,000ps
600,000ps
reg_addr[1] 'h 00002000 00000000 00002000
reg_addr[2] 'h FFFFFFFA 00000000 FFFFFFFA
reg_addr[3] 'h 00000012 00000000 00000012
reg_addr[4] 'h FFFFFFF4 00000000 FFFFFFF4
reg_addr[5] 'h FFFFFF94 00000000 FFFFFF94
reg_addr[6] 'h 00000000 00000000
reg_addr[7] 'h 00000000 00000000
reg_addr[8] 'h 00000012 00000000 00000012
reg_addr[9] 'h FFFFFFFF 00000000 FFFFFFFF
reg_addr[10] 'h 0CCCCCCC 00000000 0CCCCCCC
reg_addr[11] 'h 00000000 00000000
reg_addr[12] 'h FFFFFFF0 00000000 FFFFFFF0
reg_addr[13] 'h 00000014 00000000 00000014
reg_addr[14] 'h 00000013 00000000 00000013
reg_addr[15] 'h FFFFFF94 00000000 FFFFFF94
reg_addr[16] 'h 00000000 00000000
reg_addr[17] 'h 00000000 00000000
reg_addr[18] 'h 00000012 00000000 00000012
reg_addr[19] 'h FFFFFFFF 00000000 FFFFFFFF
reg_addr[20] 'h 0CCCCCCC 00000000 0CCCCCCC
reg_addr[21] 'h 00000000 00000000
reg_addr[22] 'h 00000013 00000000 00000013
reg_addr[23] 'h 00000000 00000000

Design of a Pipeline RISC-V Processor

1. Develop a Pipelined Processor from the previously designed Single-Cycle Processor.

A pipelined processor is an advanced processor developed as an improvement over the single-cycle processor design. A single-cycle processor executes each instruction in a single clock cycle, which can limit performance due to extended instruction delays and inefficient use of hardware resources.

In contrast, a pipelined processor divides the instruction execution process into a series of stages, allowing multiple instructions to be processed simultaneously. Each stage of the pipeline performs a specific operation on an instruction, such as instruction fetch, decode, execute, memory access, and write-back. By splitting the instruction execution into multiple stages, the overlapping execution of instructions is enabled, significantly enhancing the processor's overall performance.

The stages in a pipelined processor operate synchronously, with each stage passing intermediate results to the next stage. This enables a continuous flow of instructions through the pipeline, with different instructions being processed simultaneously at different stages of the pipeline. As a result, the processor can achieve higher instruction throughput and improved performance compared to a single-cycle processor.

A pipelined processor also enables optimized utilization of hardware resources. While a single-cycle processor dedicates all hardware resources to executing a single instruction at any given time, a pipelined processor distributes the hardware resources across different stages. This allows for more efficient use of resources, potentially resulting in higher overall processor performance.

However, implementing pipelining introduces new challenges such as data hazards, control hazards, and structural hazards. Data hazards occur when instructions depend on the results of previous instructions still being processed in earlier pipeline stages. Control hazards arise from branch instructions that alter the program flow, leading to pipeline flushing and wasted cycles. Structural hazards occur when

multiple instructions compete for the same hardware resource at the same time.

To address these challenges, techniques such as forwarding, branch prediction, and hazard detection logic are applied in pipelined processors. These techniques help mitigate the impact of hazards and ensure smooth and continuous instruction execution while maintaining the benefits of pipelining.

In summary, a pipelined processor builds upon the single-cycle processor design by dividing the instruction execution process into multiple stages and enabling simultaneous processing of instructions. This leads to improved performance, higher instruction throughput, and better utilization of hardware resources. However, it also introduces challenges related to hazards, which must be addressed through corresponding techniques. Overall, pipelining is a significant advancement in processor design that has contributed substantially to the performance improvements in modern processors.

2. Analyses

2.1. Non - Forwarding

The initial model will NOT use forwarding or a branch predictor. In other words, the processor will always fetch the next instruction located at the address PC + 4 into the pipeline. We will design a hazard detection unit to determine when to stall the pipeline to prevent Read-After-Write (RAW) hazards. The processor will flush incorrect instructions or stall the pipeline when a branch instruction is "executed" or detected. We will design a hazard detection unit along with additional modules to handle these cases.

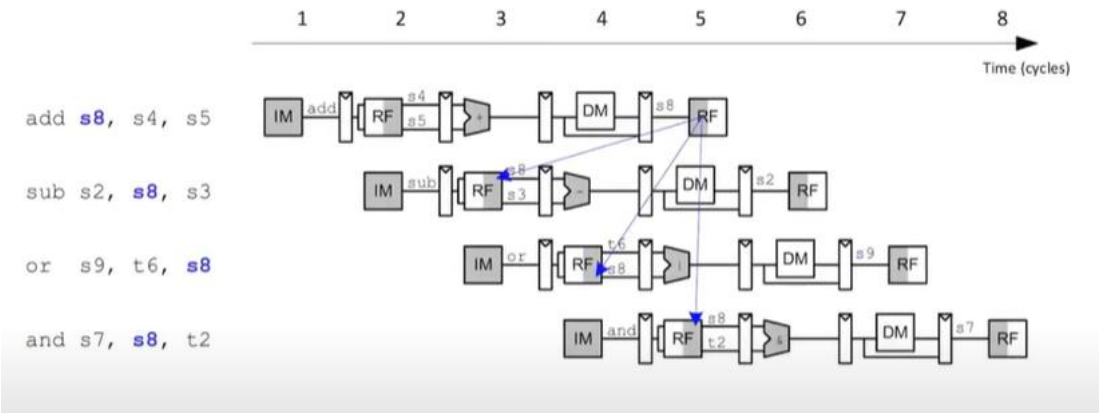
Design a five-stage pipelined processor consisting of:

1. **IF (Instruction Fetch)**: Fetching the instruction from memory.
2. **ID (Instruction Decode)**: Reading registers and decoding the instruction.
3. **EX (Execute)**: Performing operations or calculating addresses.
4. **MEM (Memory Access)**: Accessing operands in the data memory.
5. **WB (Write-back)**: Writing results back to the registers.

This is a basic pipelined design, easy to implement and partitioned based on the previously designed single-cycle processor. There are various hazards that we will

handle, such as Data Hazards, Load-Store Hazards, and Branch Hazards.

Regarding **Data Hazards**, the issue arises when the following instructions require the register content that is being processed by the previous instruction in the Execute or MEM stage. In other words, one or two consecutive instructions use the same source register address as the destination register address of the current instruction.



Handles 3 different types of hazards commonly found in a pipeline:

- Hazard 1: The instruction in the ID stage depends on data that the instruction in the WB stage has not yet written back.
- Hazard 2: The ID stage is requesting data that the instruction in the MEM stage is still processing.
- Hazard 3: the instruction in the ID stage needs to use data from a register that the instruction in the EX stage has not yet finished computing and has not written back.

Hazard handling using bubble insertion

- flush_EX = 0: insert a NOP into the EX stage (remove the instruction waiting on the hazard).
- stall_ID = 0: stall the pipeline at the IF/ID stage.
- pc_enable: if i_pc_sel = 0, keep the current PC; if i_pc_sel = 1 (branch), allow the PC to update to the branch target.

Load-Store Hazard:

Cycle 1: lw x5, 0(x1) // Load value into x5 (in ID stage)

Cycle 2: sw x5, 4(x2) // Store value from x5 to memory (in ID stage now)

When the Store is in the ID stage, the Load is in the EX stage and has not yet fetched the value for x5.

The Store needs that value immediately for its data path, but it won't be ready until the Load finishes reading from memory in the MEM stage.

The hazard detection logic spots this dependency and stalls the pipeline until the Load completes, then forwards the value to the Store.

For branch instructions, a flush_ID flag is required because a wrong branch instruction may have been fetched and passed through the ID stage. This flush ensures that any incorrect instruction in the pipeline is cleared before proceeding further. The branch instruction will be handled when the i_pc_sel signal is set to 1, starting from the EX stage. This signal indicates that the program counter should be updated to the target address of the branch, effectively redirecting the flow of execution and flushing the pipeline of incorrect instructions.

When the branch instruction is in the Decode (ID) stage:

- flush_ID = 0: insert a NOP into the ID stage.
- pc_enable = 0: stop incrementing the PC to prepare for redirection (branch target).

Case where the branch is evaluated at EX:

- If i_pc_sel = 1 ⇒ the branch is taken, meaning the PC needs to jump → we flush the instruction in ID (do not execute it).
- If i_pc_sel = 0 ⇒ the branch is not taken → keep the instruction in ID, no flush.

This combination of stall, flush_EX, and flush_ID flags allows the processor to handle both data hazards and branch hazards, maintaining pipeline integrity and ensuring correct instruction execution.

2.2. Forwarding

This model will use **forwarding** but will not include a **branch predictor**. We will design a **forwarding unit** to handle **Read-After-Write (RAW) hazards**, in addition to the approach taken in the first model. Split into a hazard unit and a forward unit because some instructions, such as lw, need to retrieve data from the WB stage and cannot use forwarding.

The forwarding unit will allow data to be forwarded directly from the **EX** or **MEM** stages to the **ID** or **EX** stages, based on the pipeline stages, to resolve data dependencies between instructions. This mechanism helps prevent stalls caused by RAW hazards when an instruction needs data that is still being written by a previous instruction.

For each instruction, we need to determine whether the destination register of the previous instructions is used in the next instruction, where this register has become a source register (either rs1 or rs2) in the new instruction. Specifically, we need to compare the destination register of the previous instruction in the pipeline with the source registers in the current instruction during the **decode** stage, while ignoring the case where the register being written to is x0 (since x0 is a constant zero register and does not affect data dependencies).

The **Forwarding Unit** plays a crucial role in resolving **Data Hazards** in the pipeline. Here's how it works and how the various hazards are managed:

Data Hazard Handling in Forwarding:

R-Format Instructions: Data hazards are resolved by forwarding the results that are completed in later pipeline stages to earlier stages. This allows the pipeline to continue running without stalls.

- **Example**: If the destination register (rd) in the **MEM stage** is needed by the **EX stage**, the forwarding unit will forward the result from the **MEM stage** to be used as a source register in the **EX stage**.

Store Instructions:

- Store instructions use rs2 to write data to memory. When there is a data hazard involving store instructions, the forwarding unit requires a separate output signal, forward_store_sel, to control a multiplexer (mux) that selects the correct data to perform the store operation.

Data Memory Access (I-Format and S-Format Instructions):

- **Load/Store Data Hazards**: Hazards during data access (e.g., loading data into memory) are handled by forwarding. However, the **use after load hazard** occurs when a load instruction writes data into a register, and the next instruction (which uses that register) needs that data. This cannot be resolved by forwarding because the data from memory is not yet available at the time the next instruction executes.
- To handle this, the pipeline stall for one cycle to ensure that the correct data is available. This is managed by a **stall_check** unit, which detects this hazard and issues a **NOP** (no operation) instruction for that cycle.

Branch Hazard:

- Branch hazards are handled by checking the branch signal at the **EX stage**. When the i_pc_sel signal is active (set to 1), it indicates that a branch is being executed. The multiplexer (mux) in the **IF** and **ID** stages will use this signal to flush the previous instructions and adjust the pipeline flow accordingly.
    - For **B-type instructions** (branch instructions), although there may be data hazards in the rs1 and rs2 registers, these hazards are handled earlier in the **ID stage**, so there is no problem by the time the instruction reaches the **EX stage**.
    - There will not be a separate module for branch prediction. Instead, two input muxes will use the branch signal (via pc_sel at the **EX stage**) to discard the previous instructions and handle the branch correctly, turning data into NOP if necessary.

2.3. Two – Bit Prediction:

The two-bit prediction algorithm is a method used in pipelined processors to improve the accuracy of branch prediction. In a pipelined processor, when a branch instruction is encountered, predicting whether the branch will be taken or not taken is important to maintain the continuity of the instruction execution. If the branch prediction is correct, the processor can continue executing the next instruction without waiting for the result of the branch instruction. On the other hand, if the prediction is incorrect, the processor must perform operations to correct the prediction and wait for the actual result of the branch instruction to be determined. The two-bit prediction algorithm uses a two-bit counter to track the history of the branch. Each branch is associated with a two-bit counter, where each bit represents a prediction state, including:

1. STRONG_TAKEN (11): The branch is predicted to be taken at level 2 (if the prediction is wrong next time, the BTB remains at 0 and the state is updated).

2. WEAK_TAKEN (10): The branch is predicted to be taken at level 1 (if the prediction is wrong next time, the BTB is updated to state 0 for not taken and level 1; if the prediction is correct, the BTB increases to level 2).

3. STRONG_NOT_TAKEN (00): The branch is predicted not to be taken at level 2 (if the prediction is wrong next time, the BTB remains at 0 and the state is updated).

4. WEAK_NOT_TAKEN (01): The branch is predicted not to be taken at level 1 (if the prediction is wrong next time, the BTB is updated to state 1 for taken and level 1; if the prediction is correct, the BTB increases to level 2 for not taken).

Initially, all the counters are initialized to the Strongly Not Taken (NT) state. When a branch instruction appears, the corresponding two-bit counter for that branch is checked. If the predicted state is TT or WT, the processor predicts that the branch will be taken and continues executing the next instruction. On the other hand, if the predicted state is WN or NT, the processor predicts that the branch will not be taken and will wait for the result of the branch instruction before continuing execution.

When the branch instruction completes, its result is used to update the two-bit counter. If the branch is taken, the counter is incremented to a higher prediction state (TT or WT). If the branch is not taken, the counter is decremented to a lower

prediction state (WN or NT). This helps improve future branch prediction accuracy based on the previous execution history.

The two-bit prediction algorithm provides a balance between accuracy and performance. By using a two-bit counter, this algorithm can adapt to different branch prediction patterns and minimize the frequency of incorrect predictions. However, it also poses some issues, such as when a branch is encountered for the first time or when there is a rapid transition between prediction states.

In summary, the two-bit prediction algorithm is a branch prediction method used in pipelined processors. It employs a two-bit counter to track prediction history and offers a balance between accuracy and performance. By utilizing information from previous predictions, this algorithm helps improve branch prediction accuracy and optimize the instruction processing in a multi-stage processor.

3. Design

3.1. Non – Forwarding Pipeline Processor

The input and output signals in the Non - Forwarding design will be presented in this section:

To handle data hazards and load-store data, the Hazard Detection Unit is responsible for checking and activating the stall signals (logic high). For branch instructions, the implementation is similar to the Single Cycle design, relying on the pc_sel signal (from EX) being activated when a branch is taken. These signals will be configured into multiplexers in the two preceding stages to ensure no errors occur, as discussed in the Analysis section.

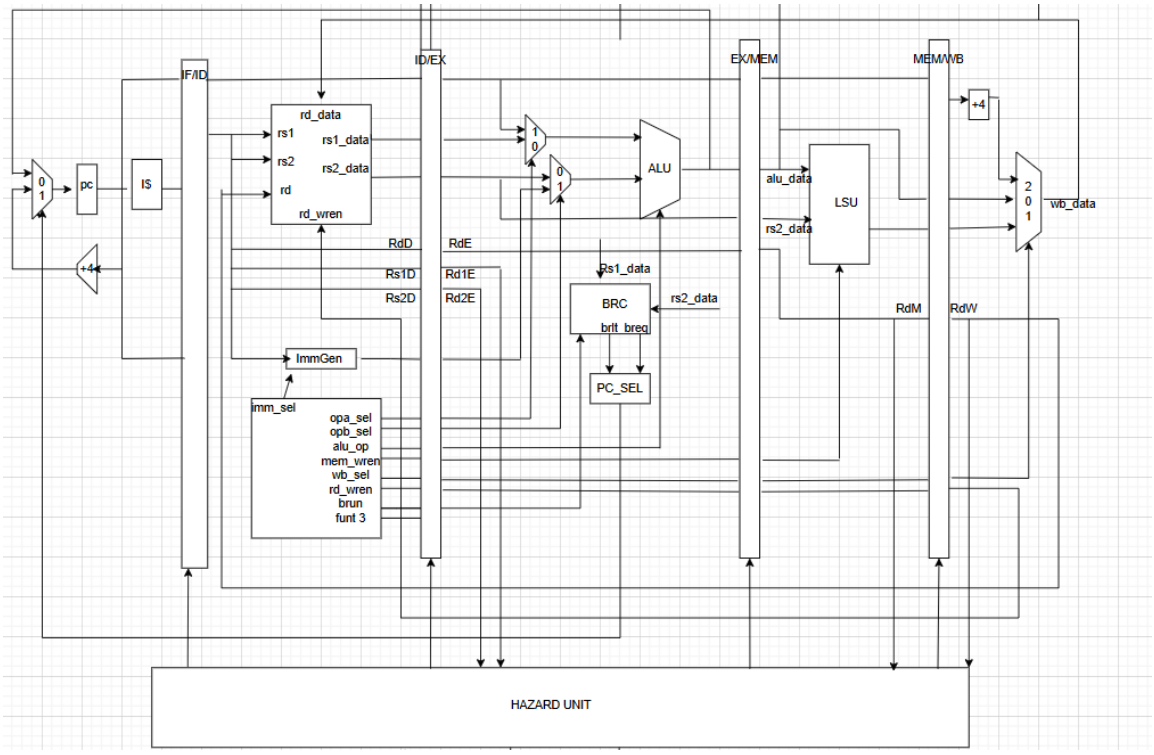


Figure 1. Block Diagram of a Non-Forwarding Pipelined Processor

Thus, the design only adds components such as flip-flop registers to implement pipelining. There are four such flip-flop units, with their inputs and outputs depending on the stage of execution and the data required by the instruction.

This section focuses on specifying the signals for the Hazard Detection Unit and the register blocks.

3.1.1. Module: hazard_unit

Signal	Type	Size	Description
i_pc_sel	Input	1	Indicates branch condition: 1 for ALU data, 0 for PC+4.
ex_rd_wren	Input	1	Write enable signal for the register file in the EX stage.
mem_rd_wren	Input	1	Write enable signal for the memory (LSU) in the MEM stage.
wb_rd_wren	Input	1	Write enable signal for the register file in the WB stage.
ex_rd_addr	Input	5	Destination register address in the EX stage.
mem_rd_addr	Input	5	Destination register address in the MEM stage.
wb_rd_addr	Input	5	Destination register address in the WB stage.
id_rs1_addr	Input	5	Source register 1 address in the ID stage.
id_rs2_addr	Input	5	Source register 2 address in the ID stage.
id_opcode	Input	7	Opcode of the instruction in the ID stage.
ex_opcode	Input	7	Opcode of the instruction in the EX stage.
mem_opcode	Input	7	Opcode of the instruction in the MEM stage.
stall_ID	Output	1	Signal to stall the IF/ID pipeline stage.
stall_EX	Output	1	Signal to stall the ID/EX pipeline stage.
stall_MEM	Output	1	Signal to stall the EX/MEM pipeline stage.
stall_WB	Output	1	Signal to stall the MEM/WB pipeline stage.
pc_enable	Output	1	Enables the PC update: 0 for stall, 1 for normal operation.
flush_ID	Output	1	Flushes the IF/ID pipeline stage.
flush_EX	Output	1	Flushes the ID/EX pipeline stage.
flush_MEM	Output	1	Flushes the EX/MEM pipeline stage.
flush_WB	Output	1	Flushes the MEM/WB pipeline stage.

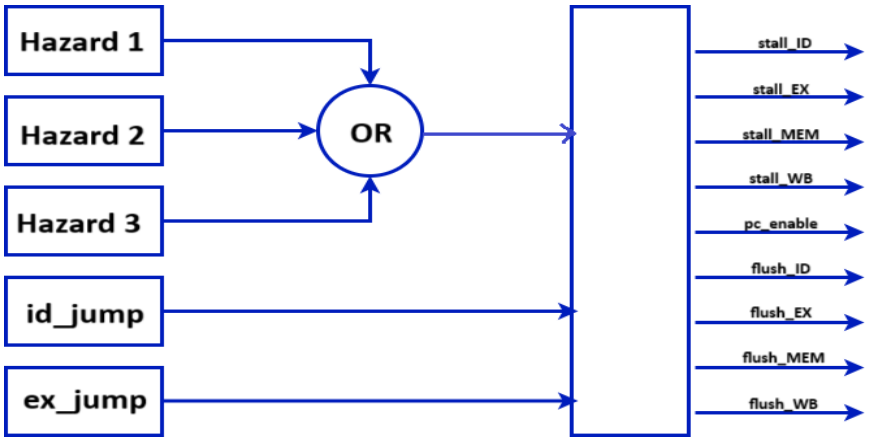


Figure 2. Block Diagram of the Hazard Detection Unit in Non-Forwarding Pipelined Processor

3.1.2. Module: IF_ID

Signal	Type	Size	Description
i_clk	Input	1	Clock signal to trigger the pipeline register updates.
i_rst_n	Input	1	Active-low reset signal to initialize the pipeline registers.
enable	Input	1	Enable signal to allow updating of the pipeline registers.
pc_IF	Input	32	Program counter value from the IF stage.
instr_IF	Input	32	Instruction fetched from the IF stage.
pc_ID	Output	32	Program counter value passed to the ID stage.
instr_ID	Output	32	Instruction passed to the ID stage.

3.1.3. Module: ID_EX

4. Signal	Type	Size	Description
i_clk	Input	1	Clock signal to trigger the pipeline register updates.
i_rst_n	Input	1	Active-low reset signal to initialize the pipeline registers.
enable	Input	1	Enable signal to allow updating of the pipeline registers.
i_insn_vld	Input	1	Instruction valid signal to indicate if instruction is valid.
i_pc	Input	32	Program counter value from the ID stage.
i_instr	Input	32	Instruction passed to the ID stage.
i_rd_wren	Input	1	Register write enable signal for the ID stage.
i_ld_en	Input	3	Load enable signal for the ID stage.
i_opa_sel	Input	1	Operand A selection signal for the ALU.
i_opb_sel	Input	1	Operand B selection signal for the ALU.
i_lsu_wren	Input	1	Load/store unit write enable signal.
i_alu_op	Input	4	ALU operation selection signal.
i_br_un	Input	1	Branch uncondtional signal for branch prediction.
i_wb_sel	Input	2	Write-back selection signal for the write-back stage.
i_rs1_data	Input	32	Data from register Rs1.
i_rs2_data	Input	32	Data from register Rs2.
o_insn_vld	Output	1	Output signal for instruction validity.
o_pc	Output	32	Program counter output for the EX stage.
o_instr	Output	32	Instruction passed to the EX stage.
o_rd_wren	Output	1	Register write enable signal for the EX stage.
o_ld_en	Output	3	Load enable signal for the EX stage.
o_opa_sel	Output	1	Operand A selection signal for the ALU in the EX stage.

o_opb_sel	Output	1	Operand B selection signal for the ALU in the EX stage.
o_lsu_wren	Output	1	Load/store unit write enable signal in the EX stage.
o_alu_op	Output	4	ALU operation selection signal in the EX stage.
o_br_un	Output	1	Branch uncondtional signal for branch prediction in the EX stage.
o_wb_sel	Output	2	Write-back selection signal in the EX stage.

3.1.4. Module : EX_MEM

Signal	Type	Size	Description
i_clk	Input	1	Clock signal for synchronization
i_rst_n	Input	1	Active low reset signal
enable	Input	1	Enable signal for operation
i_insn_vld	Input	1	Instruction valid flag
i_pc	Input	32	Program counter value at the EX stage
i_rs2_data	Input	32	Data from rs2 operand at EX stage
i_instr	Input	32	Instruction at EX stage
i_ld_en	Input	3	Load enable signal
i_lsu_wren	Input	1	LSU write enable signal
i_rd_wren	Input	1	Register write enable signal
i_wb_sel	Input	2	Write-back selection (which data to write back)
i_alu_data	Input	32	ALU result at EX stage
o_insn_vld	Output	1	Instruction valid flag
o_pc	Output	32	Program counter value at MEM stage
o_rs2_data	Output	32	Data from rs2 operand at MEM stage
o_instr	Output	32	Instruction at MEM stage
o_ld_en	Output	3	Load enable signal
o_rd_wren	Output	1	Register write enable signal
o_lsu_wren	Output	1	LSU write enable signal
o_wb_sel	Output	2	Write-back selection (which data to write back)

3.1.5. Module: MEM_WB

Signal	Type	Size	Description
i_clk	Input	1	Clock signal for synchronization
i_rst_n	Input	1	Active low reset signal
enable	Input	1	Enable signal for operation
i_pc	Input	32	Program counter value at the MEM stage
i_instr	Input	32	Instruction at MEM stage
i_insn_vld	Input	1	Instruction valid flag
i_rd_wren	Input	1	Register write enable signal
i_ld_data	Input	32	Load data at MEM stage
i_wb_sel	Input	2	Write-back selection (which data to write back)
i_alu_data	Input	32	ALU result at MEM stage
o_pc	Output	32	Program counter value at WB stage
o_instr	Output	32	Instruction at WB stage
o_insn_vld	Output	1	Instruction valid flag
o_rd_wren	Output	1	Register write enable signal

o_ld_data	Output	32	Load data at WB stage
o_wb_sel	Output	2	Write-back selection (which data to write back)

3.2. Forwarding Pipeline Processor

Resolving data hazards and load-store data will utilize the Forwarding unit, which is responsible for checking the destination and source registers of consecutive adjacent instructions. It then provides the correct value of the register for the instruction being executed to ensure time efficiency in the processing flow.

For branch instructions, the design is similar to the Single Cycle design, relying on the pc_sel signal being asserted when a branch is taken. This signal is configured into the multiplexers in the two preceding stages to ensure no errors occur, as discussed in the Analysis section. The design only adds flip-flop registers to implement pipelining.

There are four such flip-flops, and their inputs and outputs depend on the stage executing the instruction and the data required at that stage.

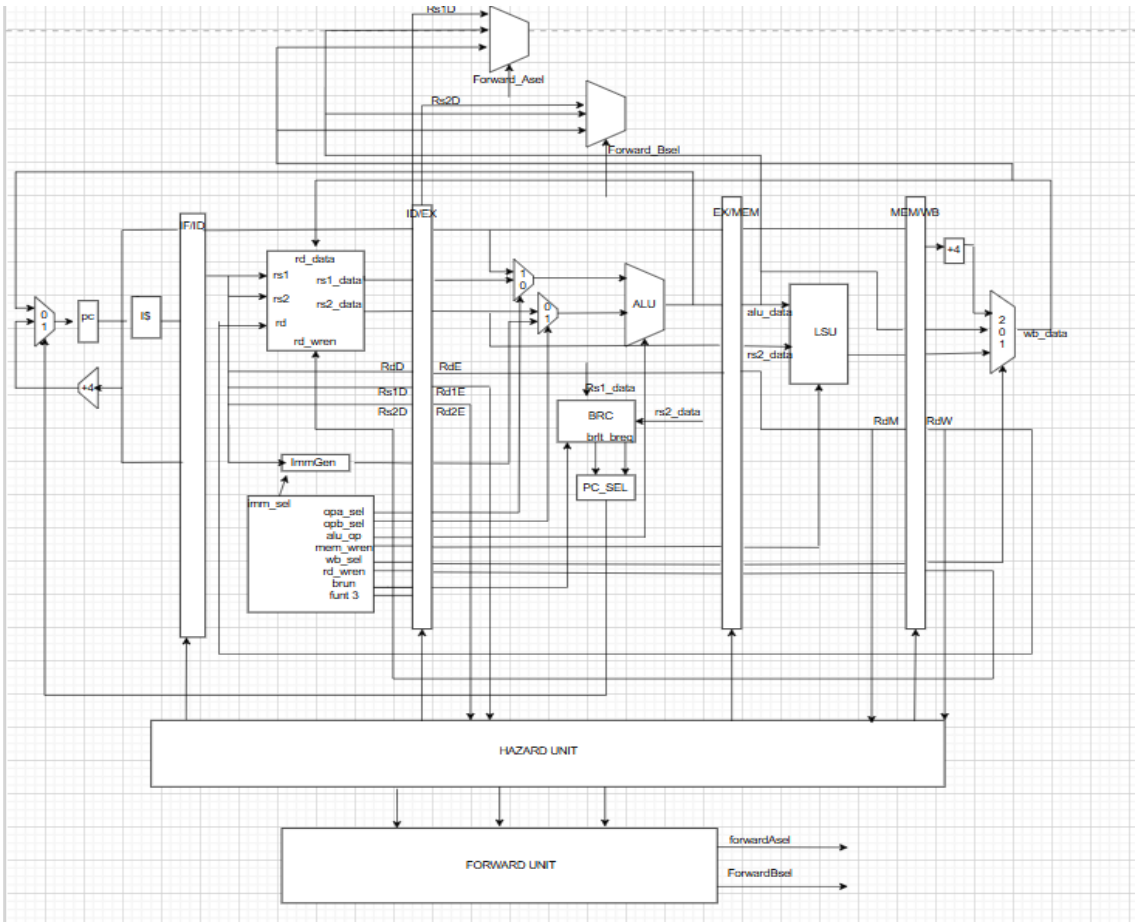


Figure 3. Block Diagram of a Forwarding Pipelined Processor

3.2.1. Module: forward_unit

Signal	Size	Type	Description
instr_MEM	32	Input	Instruction in the MEM stage, containing destination register information.
instr_WB	32	Input	Instruction in the WB stage, containing destination register information.
instr_EX	32	Input	Instruction in the EX stage, containing source register information currently being processed.
rd_wren_MEM	1	Input	Write-enable signal for the destination register in the MEM stage.
rd_wren_WB	1	Input	Write-enable signal for the destination register in the WB stage.
forward_ASel	2	Output	Selection signal for forwarding to the source operand A in the EX stage.
forward_BSel	2	Output	Selection signal for forwarding to the source operand B in the EX stage.

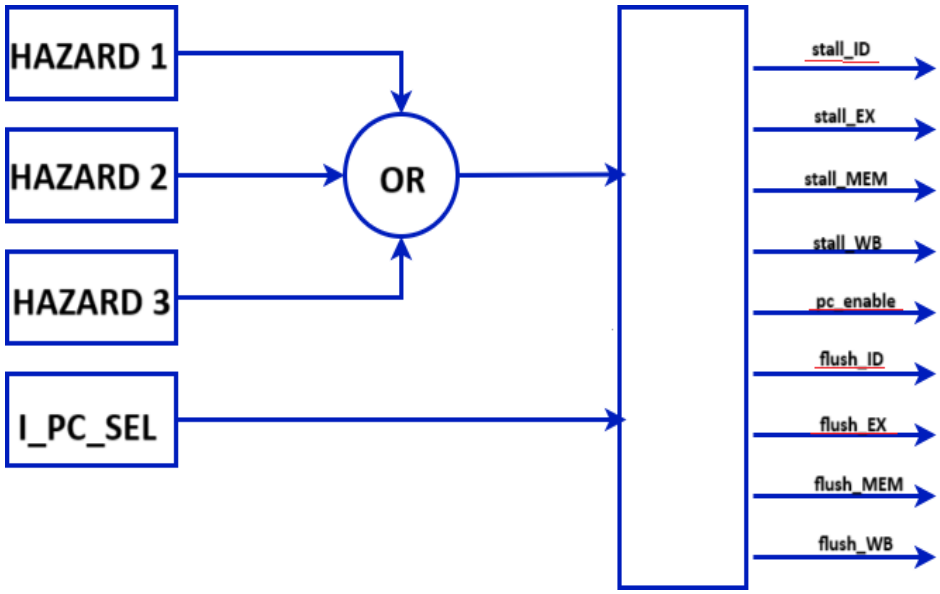


Figure 4. Block Diagram of the Hazard Detection Unit in Forwarding Pipelined Processor

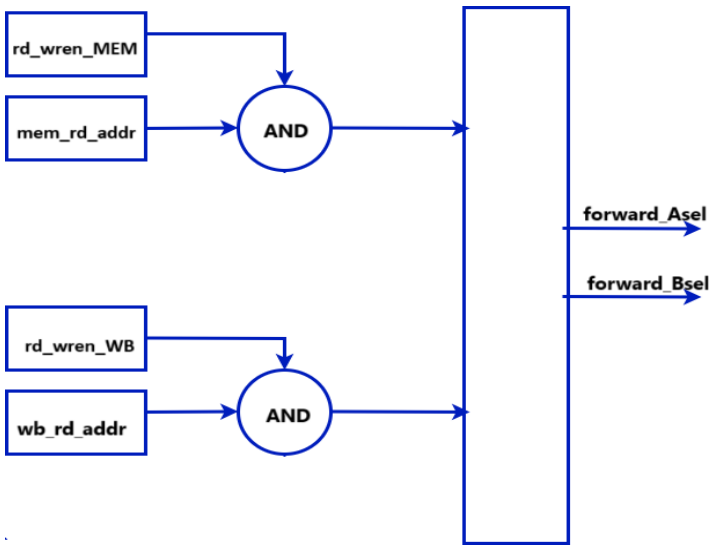


Figure 5. Block Diagram of the Forwarding Unit

3.3. Two – Bit Pipeline Processor

In pipeline design, the branch prediction unit is used to predict the outcomes of branches in the instruction code to optimize processor performance. The goal of branch prediction is to determine whether a branch will be executed and, if so, to which instruction address the branch will jump.

The branch prediction unit is often designed with a small cache, referred to as the branch prediction buffer or branch history table. This memory stores information about previous branch outcomes during program execution. Each entry in the branch prediction buffer contains details about the branch address and the predicted outcome (taken or not taken).

When a new branch is encountered during execution, the branch prediction unit consults the branch prediction buffer to check if previous branches similar to the current one exist. Based on this information, the branch prediction unit makes a prediction about the outcome of the current branch.

The input and output signals in the design of the Two-bit Pipelined Processor will be presented in this section:

Below is an explanation of the block diagram for the two-bit prediction algorithm in a pipelined processor:

1. **Fetch Stage:**

• In this stage, an instruction is fetched from the instruction memory based on the current program counter (PC).

• This instruction may include a branch instruction or the next sequential instruction potentially affected by the branch.

2. **Decode Stage:**

• In this stage, the instruction is decoded, and key fields are extracted.

• This includes identifying whether the instruction is a branch or not.

3. **Branch Prediction Stage:**

• In this stage, the two-bit prediction scheme is applied to predict the outcome of the branch instruction. The block diagram includes the following main components:

a. **Branch History Table (BHT):**

• The BHT is a table storing the prediction history of branch instructions.

• Each row in the BHT corresponds to a branch instruction and includes a two-bit counter.

b. **Branch Predictor:**

• The Branch Predictor is a logic or circuitry component used to predict the branch instruction outcome based on information from the BHT.

• The prediction states are:

▪ Strongly Taken

▪ Weakly Taken

▪ Weakly Not Taken

▪ Strongly Not Taken

c. **Branch Target Buffer (BTB):**

• The BTB is a cache used to store information about the target of branch instructions (e.g., target addresses).

• This helps reduce memory access time when executing branch instructions.

4. **Execute Stage:**

• In this stage, the instruction is executed based on the prediction results from the Branch Prediction stage.

• If the prediction is correct:

The next instruction is fetched from the sequential address (in case the branch is not taken).

• If the prediction is incorrect:

- The next instruction is fetched from the new address determined by the branch instruction.
- The block diagram of the two-bit prediction algorithm in a multi-stage processor allows for predicting the outcome of branch instructions before they are actually executed. This helps optimize processor performance by reducing waiting time caused by branch instructions.

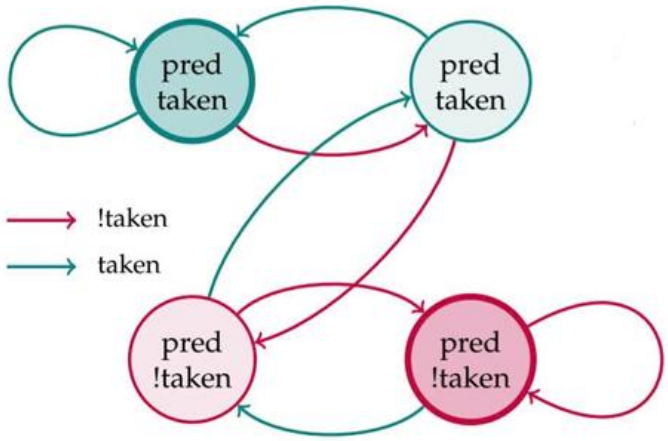


Figure 4.1. Scheme of a Two-Bit Prediction Pipelined Processor

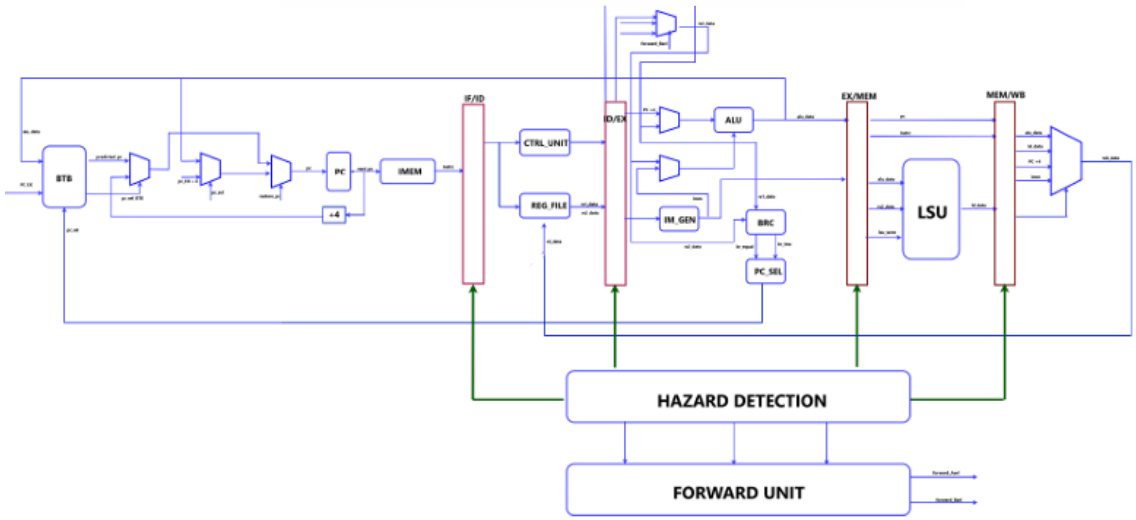


Figure 4.2. Block Diagram of the Two-Bit Prediction Pipelined Processor

1. Module: branch_predictor

Signal	Type	Size	Description
i_clk	Input	1	Clock signal to synchronize the operation of the branch predictor.
i_rst_n	Input	1	Active-low reset signal to initialize the predictor.
i_alu_data	Input	32	Data from the ALU used for branch calculations.
instr_IF	Input	32	Instruction fetched in the instruction fetch (IF) stage.
instr_EX	Input	32	Instruction in the execution (EX) stage.
pc_IF	Input	32	Program counter (PC) value from the instruction fetch (IF) stage.
pc_EX	Input	32	Program counter (PC) value from the execution (EX) stage.
i_taken	Input	1	Signal indicating whether the branch was taken or not.
o_pc	Output	32	Predicted program counter (PC) value based on the branch prediction algorithm.
o_pc_sel_BTB	Output	1	Signal indicating whether the PC should be selected from the Branch Target Buffer (BTB).
o_mispred	Output	1	Signal indicating a misprediction occurred.

Here, we will analyze the Branch Prediction module to understand the specific operation of this design:

- The memory for predicted PC is 128 memory slots, each with a size of 32 bits.

- The memory for tags is 128 memory slots, with each memory slot being 23 bits.

- In a 32-bit PC, the first 2 bits are ignored due to PC + 4, so the first 2 bits are always 00. Bits 8 to 2 are used for the predicted PC memory address and tag address.

- The predict_taken memory holds the prediction state according to the scheme below (00, 01, 10, 11).

- We have a predicted bit component used to help predict whether the next value will be PC + 4 or the predicted PC value.

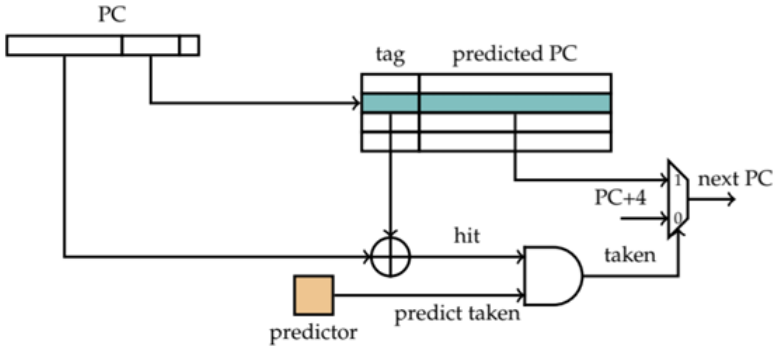


Figure 4.3. Operation Diagram of the Taken Block

We will look into the RTL code of the Branch Prediction module to further analyze the main operation of this module:

- According to the diagram, we need to create a predicted_pc memory with a size of 128 entries, each with a size of 32 bits, to store the predicted PC values for where the branch will jump. The tag register consists of 128 entries, each 23 bits in size, and is used to compute the hit bit level of the processor.

- The index_W and index_R variables are temporary variables to store the address values of PC during the execution of instructions like EX and IF, with an address range of 7 bits from bit 8 to bit 2. The pc_EX[8:2] variable is taken from the execute phase and is used as the address for predicted_pc, and the pc_IF[8:2] variable is taken from the fetch phase and is used as the address for predicted_pc.

- For the predicted_pc to make accurate predictions, it needs to learn from its mistakes. Therefore, writing and reading branch info is essential in the design of this module. Here, we will first analyze the code for the Write the branch info section.

- We use 7 bits [8:2] as the address for the predicted_pc section, leaving the remaining bits for the tag, which is 23 bits. This corresponds to the line of code: tag[index_W] <= pc_EX_i[31:9]. The write address for predicted_pc is calculated using the value of alu_data_i (computed by the ALU module and used as the PC value).

- Divides it into two main state cases: when predict_taken equals 1, the pred_taken state indicates a branch; otherwise, when predict_taken equals 0, the !pred_taken state indicates no branch.

- Next, our team will analyze the process of reading from the predicted_pc memory (Read the branch info in the buffer) in the buffer

+ Initially, our team will reset the values of pc_o and br_sel_BTB_o to 0. At the beginning, the predicted_pc memory will be empty and contain no values, so we cannot read when and whether a branch should be taken or not, or to which address it should jump. At this point, we have to accept that the pc will take the value of PC + 4, and there will be an error during this process.

+ First, since this is a read operation, we will fetch the instruction at the Fetch phase, unlike a write operation. We need to compare the pc address in the Fetch phase with the predicted and stored tag address. If they match, combined with the condition that the current state is in the taken state (predtaken), and both conditions are true, the value of pc_o will be assigned the address read from predicted_pc[index_R]. At this point, the variable br_sel_BTB_o (equal to 1 to return the pc_o value, or 0 for PC+4) will be set to 1, corresponding to the PC taking the computed value in predicted_pc. Conversely, pc_o = 0 and br_sel_BTB_o = 0, in which case the PC will take the value of PC+4.

- Next, revisiting the earlier issue, if the system is initially reset and the predicted_pc memory has no values, two scenarios may occur:

1. The actual result differs from the predicted value (e.g., the "actual" = no branch, while the "prediction" = branch, or vice versa).

2. Both the prediction and the actual result are the same (e.g., "actual" = branch and "prediction" = branch), but the predicted address is incorrect (e.g., the calculated branch address alu_data_i != pc_BTB – the predicted value, leading to an incorrect branch outcome.

-> To address the above two cases, the team incorporates them into the hazard module for resolution (represented as hazard_4 and hazard_5). If either of these cases occurs, meaning hazard_4 or hazard_5 is triggered, we will reset the values in the decode and execute stages. At this point, the restore_pc variable is set to 1, signaling the CPU that "the prediction was incorrect" and requesting it to "restore the address prior to the branch." The system will then store the ALU's data output from the EX stage at the current moment into the PC. Once it is known that the prediction is wrong, the restore_pc value will be raised to 1, and the next_pc value will be recalculated based on the comparison module. If the comparison is correct, the ALU data from the execute phase will be used for the PC; otherwise, the value PC = PC + 4 will be used.

4. Results

4.1. Functional Verification Methodology (Verification plan)

The goal of the functional verification methodology is to ensure that the vending machine operates correctly according to the specified requirements and that there are no errors during operation. To achieve this, we will use a series of testing and simulation techniques to verify each part of the design.

Steps for Verification:

1. Description of the Design Under Test (DUT): What inputs and outputs are involved? What is the desired relationship between inputs and outputs (assertions can be used to compare the actual and expected outputs)?
2. Test Plan**:** Establish a test plan for the design, including what components (DUT, testbench, etc.) are involved and how they relate to each other.
3. Perform Testing**:** Run the design on basic simulation software such as Quartus, execute test scripts in the established environment, and record the results. If errors are encountered in Step 2, return to Step 1 and review the design until Step 2 shows no more errors.
4. Set up a testbench to check all cases: List all test cases based on random input signals.
5. Analyze the results**:** Evaluate the test results to determine whether the design functions correctly. If all functional tests pass without errors, we can open the waveform to check whether the intermediate signals are correct. If not, we return to Step 1 and Step 2 to recheck the design.

4.2. Verification via Output Functionality Testing

4.2.1 Non – Forwarding Pipeline Processor

To verify the functionality of the Non-Forwarding design, it is necessary to address the two cases of hazards occurring in the assembly code by checking the enable and reset signals of the registers. When no hazard is encountered, all register enable signals remain active, allowing instructions to pass through the stages, while the register reset signals remain inactive.

- Test 1:

addi x3,x0,36

addi x2, x0, 64 (1)

addi x1, x0, 19

addi x7, x0, 12

add x5, x3, x2 (2)

xor x6, x5, x1 (5)

sub x9, x3, x5

or x2, x7, x5

sll x4, x5, x5

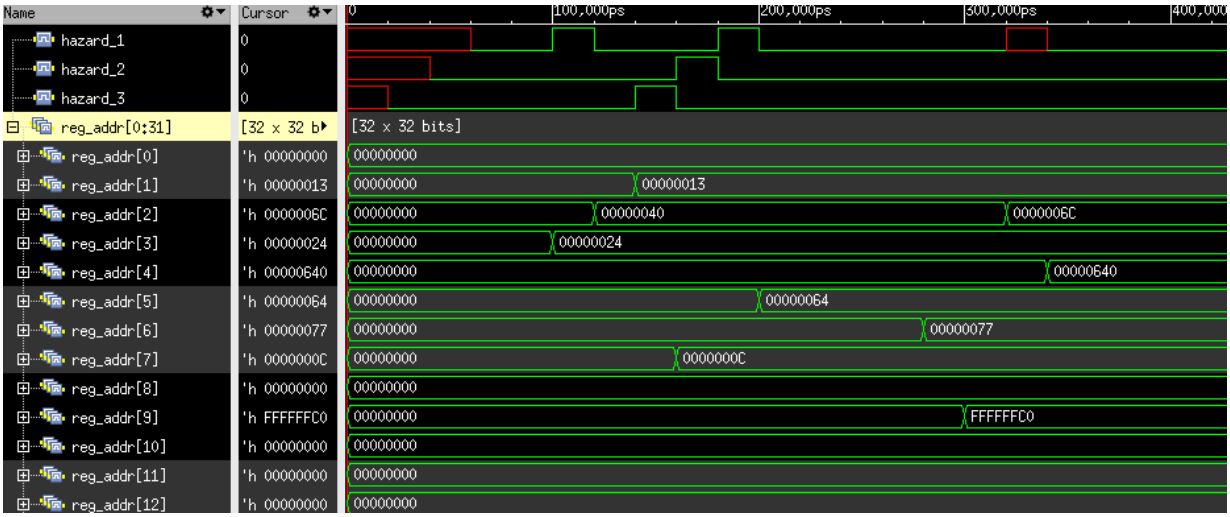
nop

nop

-> For instructions (1) and (2), when a data hazard (hazard 1 = 1) is detected due to the x2 register not yet having its value written back, the instruction in the EX_stage will become a NOP in the following cycle as the flush_EX flag is set to 1.

-> For instructions (2) and (3), since the x5 register has not finished processing its data, flush_EX remains set to 1, and the NOP instruction will propagate through the MEM and WB stages. Meanwhile, the IF/ID stages do not fetch new instructions, indicating a program stall. This wait ensures that the x5 register is completely written before being used, demonstrating that the non-forwarding system works correctly.

-> Observing the final value of x2 = 0x6C, the result is correct as per the requirements of the code.



Test case 3:

addi x3,x0,36

addi x2,x0,64

addi x5,x0,100

addi x6,x0,100

addi x8,x0,21

add x4,x3,x2

beq x5,x6,_L0

sub x9,x5,x1

or x2,x7,x5

sll x4,x5,x5

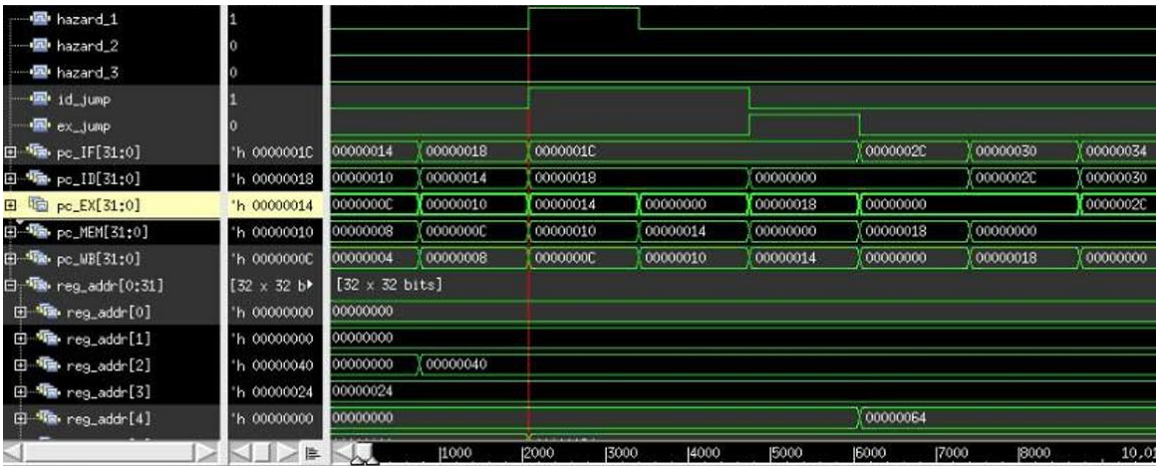
slt x3,x1,x4

_L0: sll x4,x5,x1

xor x6,x8,x2

nop

nop



According to wave form of test 2, both data and control hazards occur simultaneously, but data hazards are prioritized first. If control hazards are prioritized before data hazards, when the instruction jumps to a new one, we cannot guarantee that the old data will still be valid, leading to incorrect data. Therefore, it is necessary to handle the data hazard as a NOP in the EX stage and stall the program for 1 cycle before addressing the control hazard.

- According to the final waveform, the result of register x6 = 0x55 and x4 = 0x64 is correct as per the code requirements.

4.2.2 Forwarding Pipeline Processor

To test the functionality of the Forwarding design, we need to address the hazard case in the assembly code by checking the forwarding signals for rs1 and rs2. If these forwarding signals are correctly forwarded as expected, the hazard will be resolved.

- Test 1:

addi x3,x0,36

addi x2, x0, 64 (1)

addi x1, x0, 19

addi x7, x0, 12

add x5, x3, x2 (2)

xor x6, x5, x1 (5)

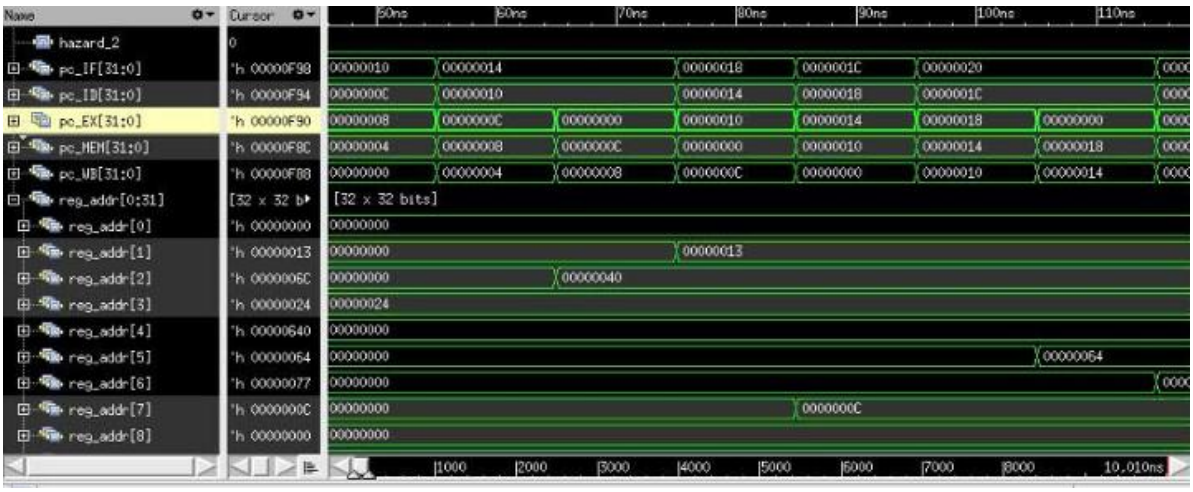
sub x9, x3, x5

or x2, x7, x5

sll x4, x5, x5

nop

nop



When looking at the instructions in test1 in forwarding mode, although there are data hazards, the ability to forward signals back to the previous stage reduces the number of NOP instructions created compared to non-forwarding.

Test 2:

lui x1,2

addi x5, x0, 2047

sh x5, 57(x1)

addi x3, x0, 36

addi x2, x0, 64

addi x7, x0, 12

add x4, x3, x2

lw x5, 64(x1)

sub x9, x5, x1

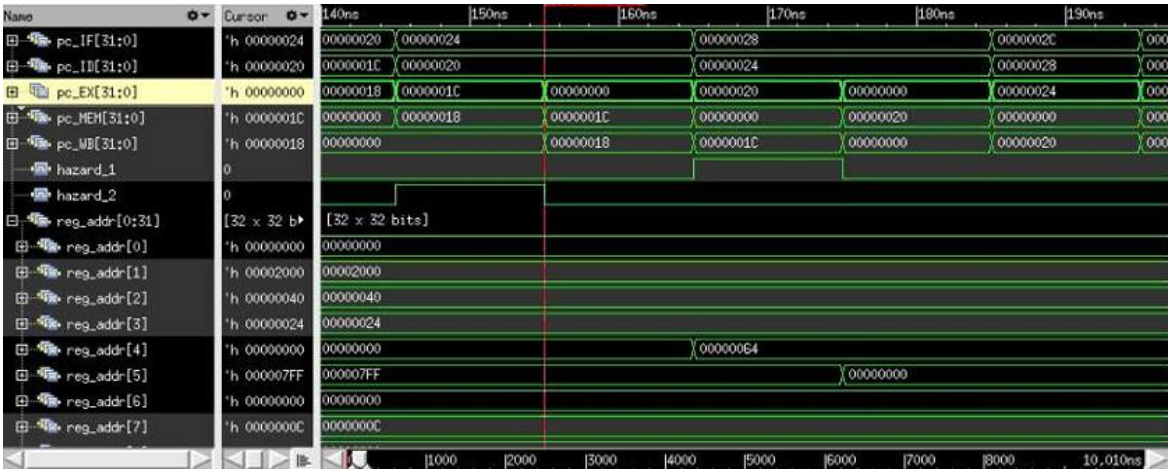
or x2, x7, x5

sll x4, x5, x1

nop

nop

When looking at the non - forwarding waveform here, we can see that for the lw instruction (1), this mode takes up to 4 cycles to resolve this hazard.



When looking at the forwarding waveform here, we can see that for the lw instruction (1), although this mode doesn't completely resolve the hazard, there is an improvement as it takes fewer cycles to handle the hazard caused by the branch instruction.

4.2.3 Two – Bit Pipeline Processor

- To check the functionality of the Two-Bit Prediction design, we need to evaluate whether the branch instructions in the assembly code perform better than with forwarding.

✓ **Test 3:**

addi x11,x0,30

addi x18,x0,1

add x13,x0,x0

COMPARE: and x12,x11,x18

beq x12,x0,ADD_EVEN (1)

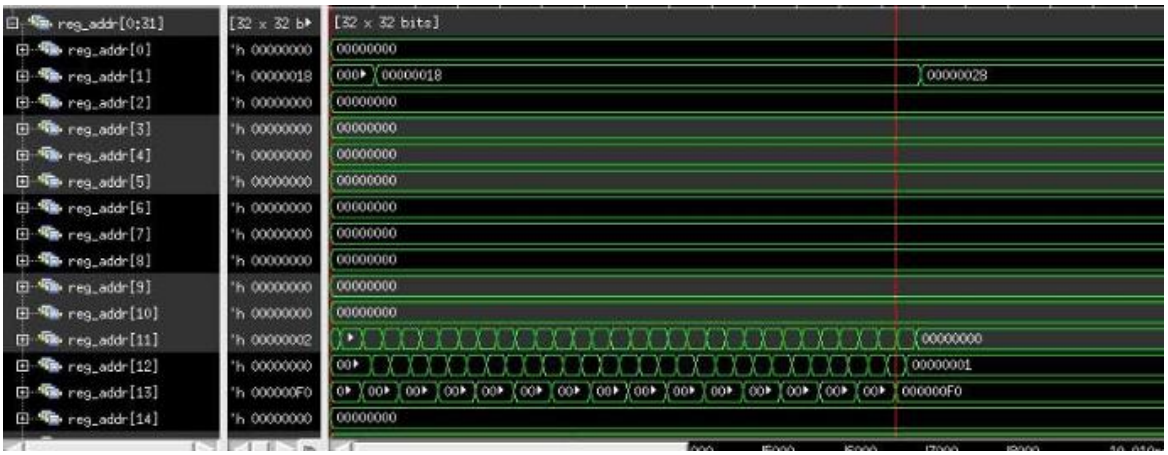
jal DECREASE

ADD_EVEN: add x13,x11,x13

DECREASE: sub x11,x11,x18

bne x11,x0,COMPARE (2)

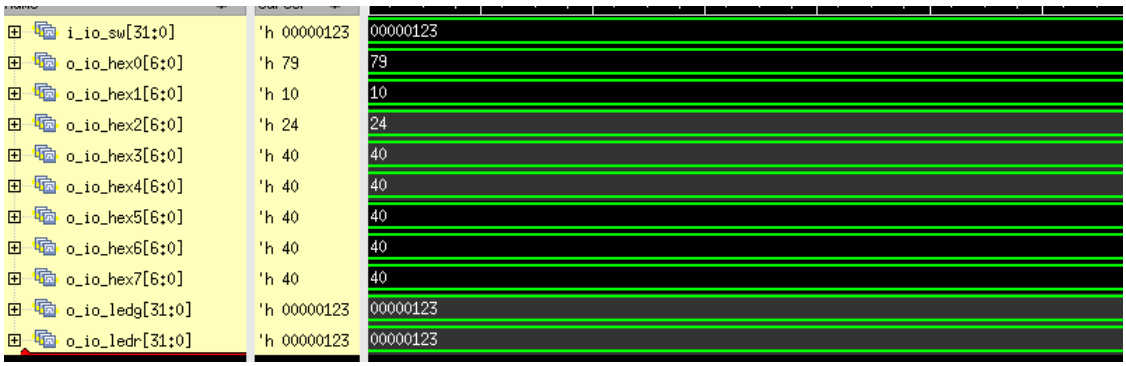
EXIT: jal EXIT



-> When observing the waveforms of the two modes, 2-bit prediction and forwarding, in the two figures below, we can see that with the forwarding mode, it takes 2665ns to complete branch instructions like instruction (1) in test3, while the 2-bit prediction mode only takes 1905ns. This shows that the processing speed of 2-bit prediction is much better.

6. FPGA Implementation

6.1. Testing result on server



6.2. Testing result on FPGA