Vietnam National University Ho Chi Minh City
Ho Chi Minh City University of Technology
Department of Electronics

EE3043: Computer Architecture

Milestone 3 Report

Student
Pham Quang Anh — 2051034
Phan Quang Minh — 2051052
Vo Viet Hung — 2051076

Supervisor
Dr. Linh Tran Hoang

8th December 2023

Table of Contents

Design of Pipelined Processor

Milestone 3

Design of Pipelined Processors

1. Introduction.

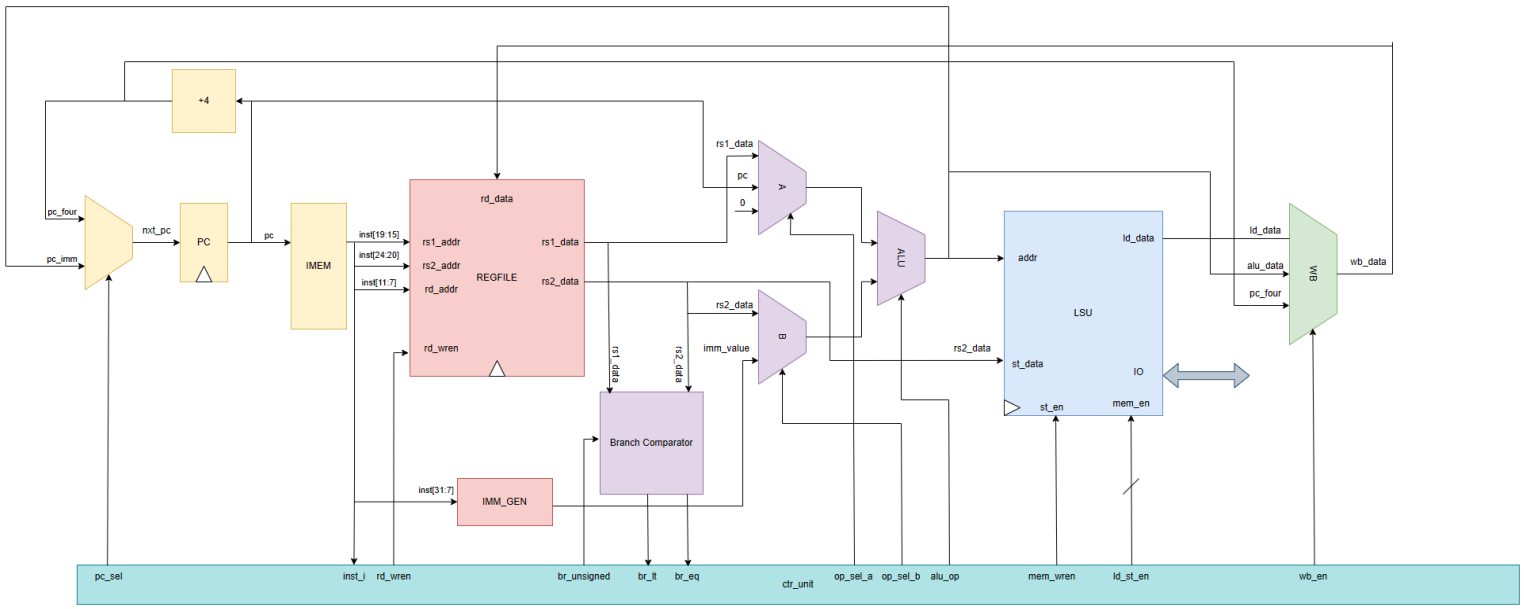
In Milestone 3, the purpose of this project is to create the 5-Stage Pipelined Processor RV32I based on the architecture of Single Cycle done in milestone 2. Additionally, in this project, we will use some sort of technique to resolve the hazard (Contain the Control and Data Hazard) caused while pipelining the RISC-V processor.

In this report, our team confirmed that we have checked most of the criteria that are required throughout the project, especially the construction of each block and waveform's performance. By applying different techniques for pipelining, we design 3 types of designs that can resolve the hazard and each design can run correctly with different efficiency based on the technique applied.

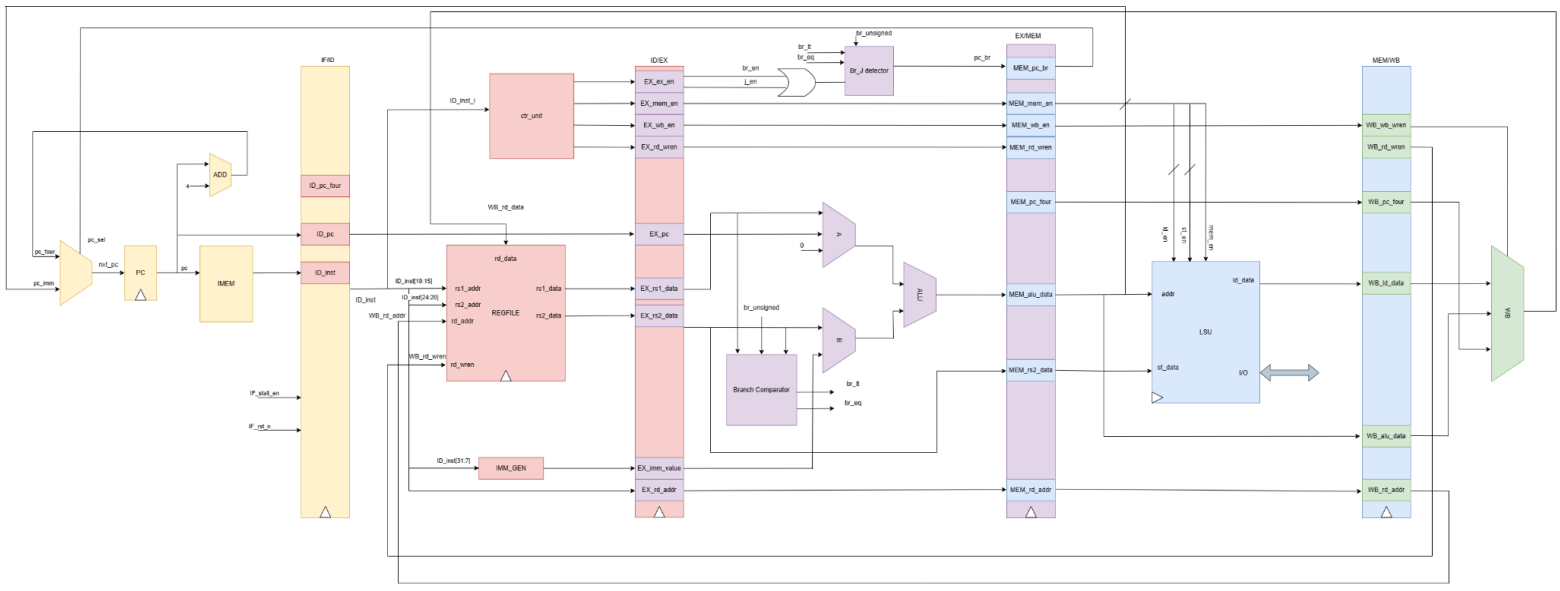
2. Design Strategy.

2.1. Design Strategy.

From the architecture of Single Cycle Processor we've done in Milestone 2, we decided to create a 5-Stage Pipelined Processor:



The 5-Stage Pipelined Processor is designed as below:



As the name suggests, there are five stages of the pipeline: (1) Instruction Fetch, (2) Instruction Decode, (3) Execute, (4) Memory and (5) Write-Back.

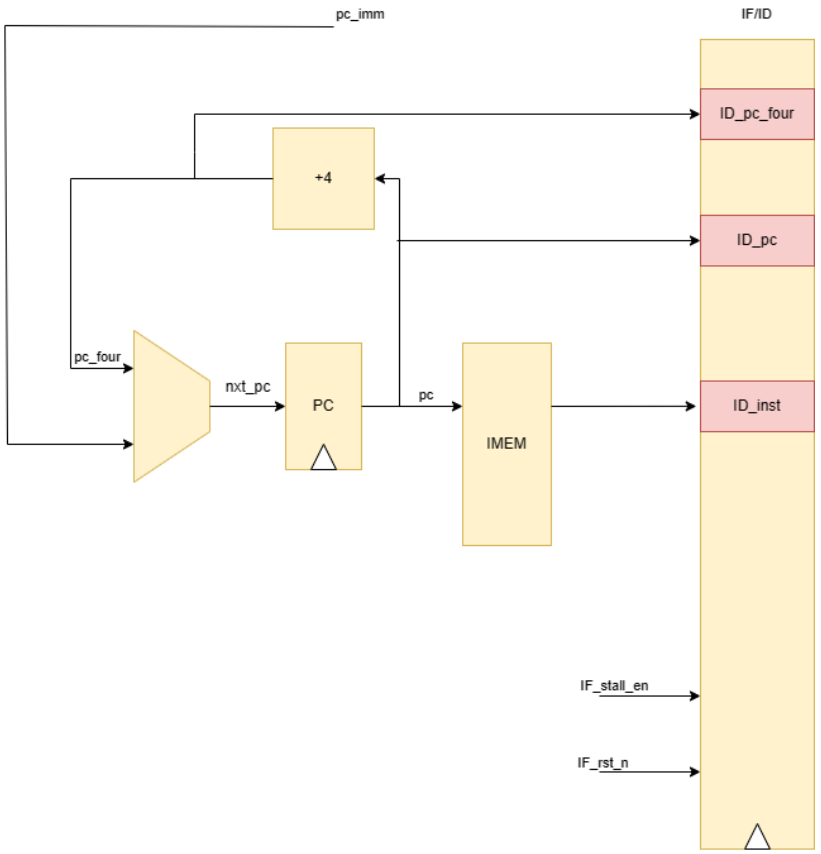
Stage	Abbreviation	Description
Instruction Fetch	IF	Queue the memory controller to fetch the instruction at the program counter's address.
Instruction Decode	ID	Register file is queried for any source registers or the immediate value is sign or zero extended into a full size (byte, halfword, word, or doubleword).
Execute	EX	The functional unit performs the operation, such as the ALU.
Memory	MEM	The memory controller is queried for load and store operations.
Write-Back	WB	The destination register has the “wren” (write enable) pin set and the destination register records the value provided by the functional unit (ALU or memory)

The Input and Output of each stage:

Module name: fetch_cycle

Signal	Type	Size	Description
pc_sel	Input	[1:0]	Select signal for the next PC
pc_imm	Input	[31:0]	Immediate value for the PC
pc_four	Input	[31:0]	PC incremented by 4
IF_stall_en	Input	[1:0]	Enable stall in IF stage
IF_rst_n	Input	[1:0]	Active-low reset for the IF stage
ID_pc_four	Output	[31:0]	Output of the PC incremented by 4
ID_pc	Output	[31:0]	Output of the current PC
ID_pc_inst	Output	[31:0]	Output of the instruction fetch

Block diagram of the Instruction Fetch stage:

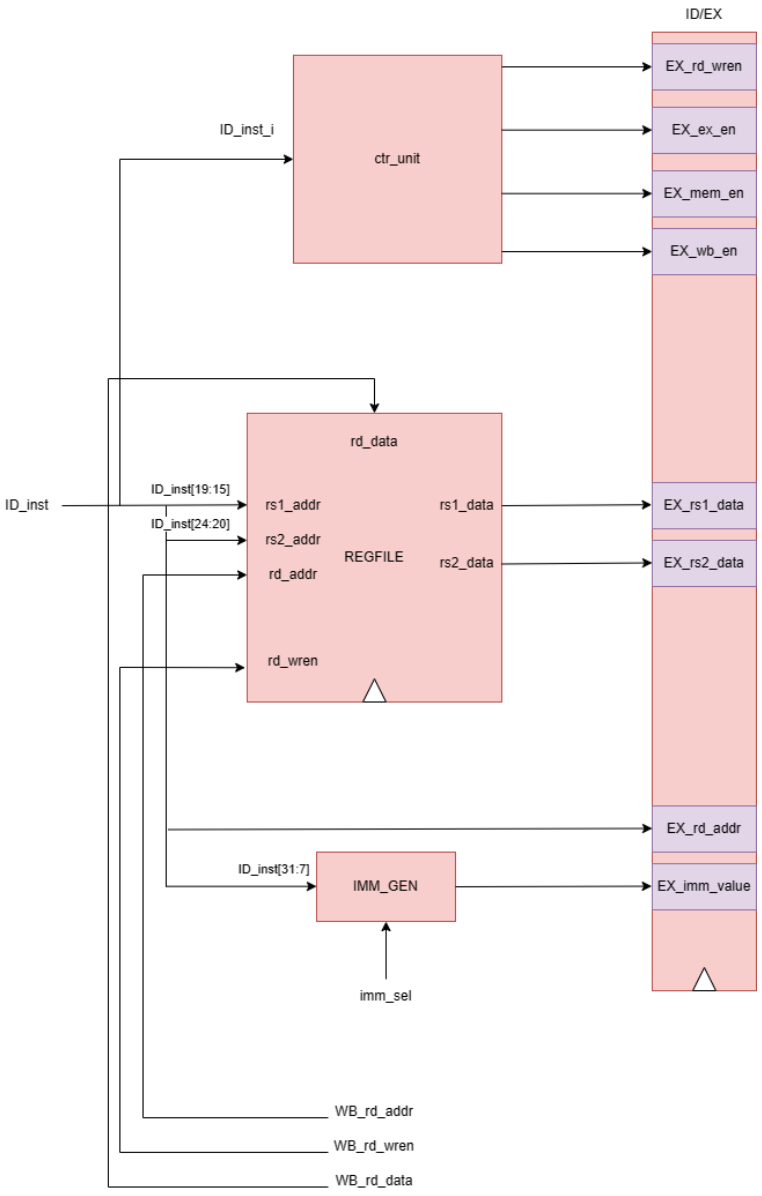


Module name: decode_cycle

Signal	Type	Size	Description
ID_inst	Input	[31:0]	Input the instructions
WB_rd_addr	Input	[4:0]	Address for the register file written back
WB_rd_wren	Input	[1:0]	Write enable signal for the register file
WB_rd_data	Input	[31:0]	Data to write back to the register file
EX_ex_en	Output	[15:0]	Control signals for the EX stage
EX_mem_en	Output	[8:0]	Control signals for the MEM stage
EX_wb_en	Output	[1:0]	Control signals for the WB stage
EX_rd_en	Output	[1:0]	Enable signal for the register file
EX_pc	Output	[31:0]	Output of the PC for the EX stage
EX_rs1_data	Output	[31:0]	Output data rs1

EX_rs2_data	Output	[31:0]	Output data rs2
EX_imm_value	Output	[31:0]	Output of immediate value
EX_rd_addr	Output	[4:0]	Address for the destination register for the EX stage

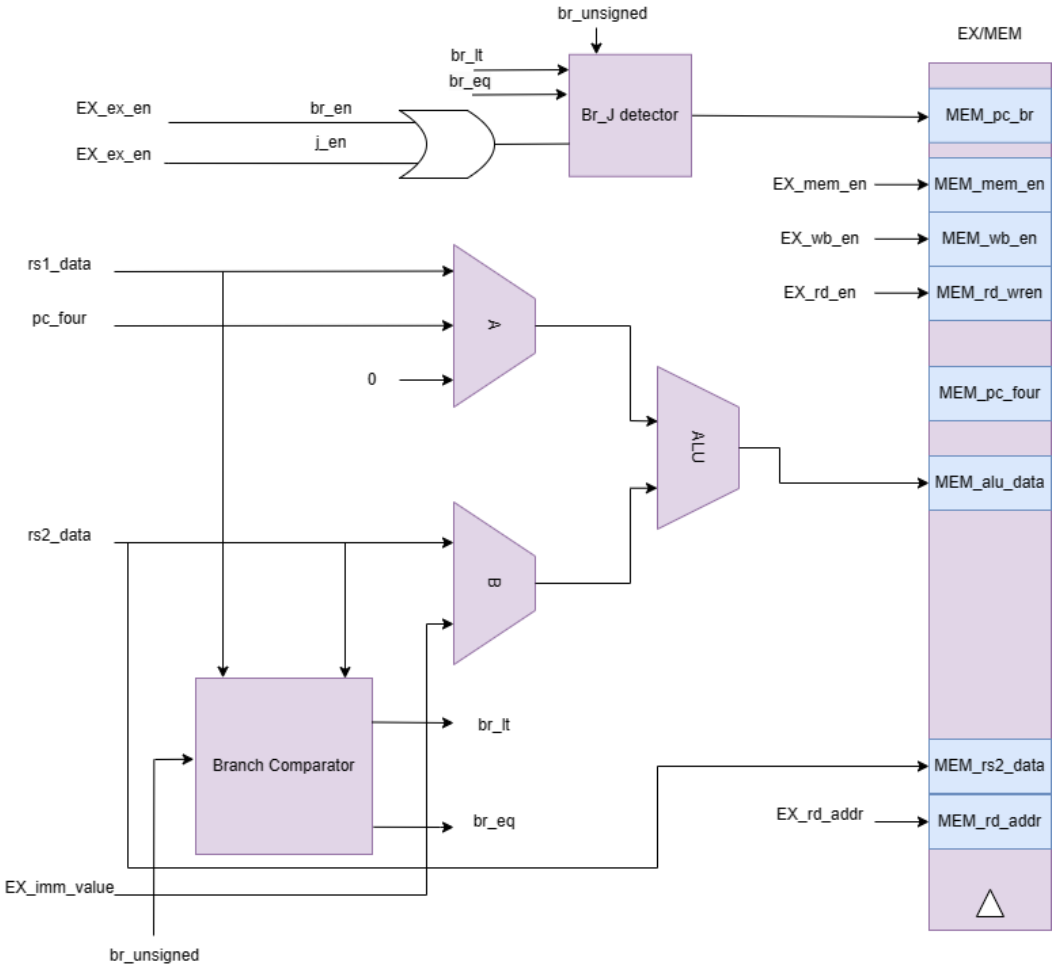
Block diagram of the Instruction Decode stage:



Module name: execute_cycle

Signal	Type	Size	Description
EX_rs1_data	Input	[31:0]	Input data rs1
EX_rs2_data	Input	[31:0]	Input data rs2
EX_pc	Input	[31:0]	Input of the PC
EX_imm_value	Input	[31:0]	Input immediate value
EX_rd_addr	Input	[4:0]	Address for the destination register for forwarding
MEM_pc_br	Output	[1:0]	Choose the PC address
MEM_mem_en	Output	[8:0]	Control signals for the MEM stage
MEM_wb_en	Output	[1:0]	Control signals for the WB stage
MEM_rd_en	Output	[1:0]	Enable signal for the register file
MEM_pc_four	Output	[31:0]	PC incremented by 4 for the next stage
MEM_alu_data	Output	[31:0]	The result of the operation
MEM_rs2_data	Output	[31:0]	Data rs2 for the store instructions
MEM_rd_addr	Output	[4:0]	Address for the destination register for the MEM stage

Block diagram of the Execute stage:

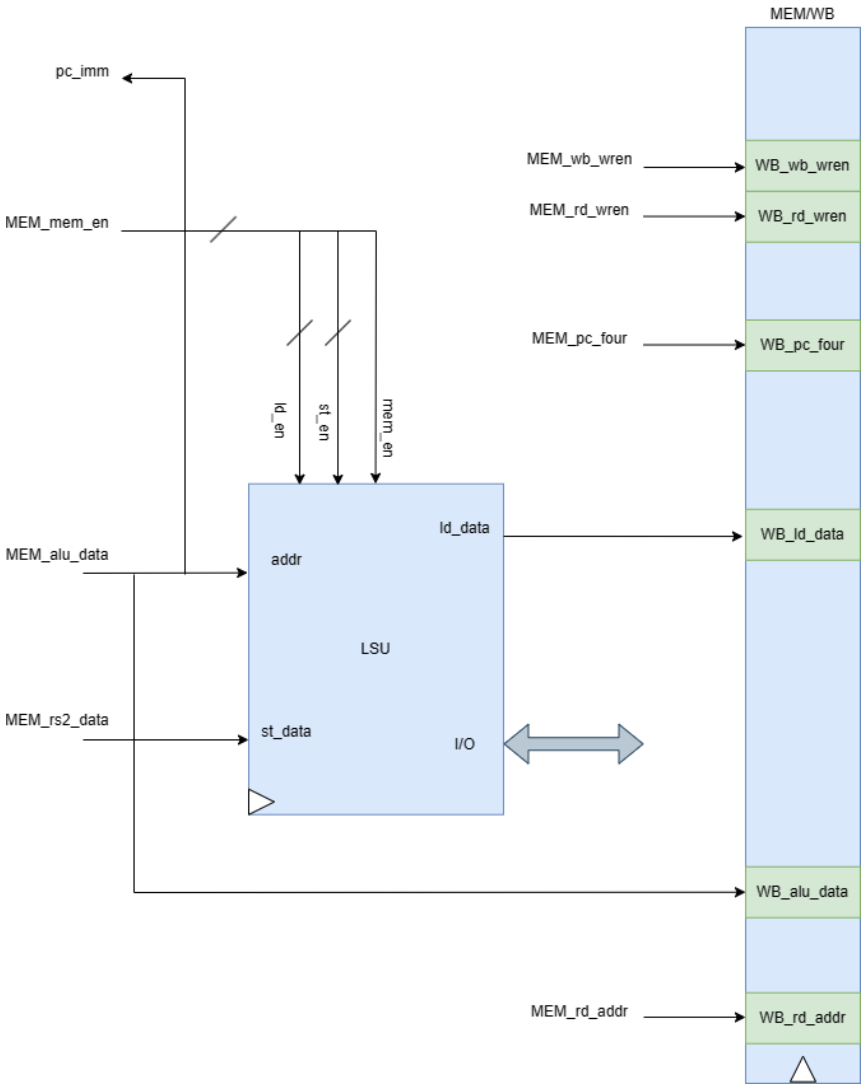


Module name: mem_cycle

Signal	Type	Size	Description
MEM_mem_en	Input	[8:0]	Control signals for the MEM stage
MEM_wb_en	Input	[1:0]	Control signals for the WB stage
MEM_pc_four	Input	[31:0]	The PC incremented by 4
MEM_alu_data	Input	[31:0]	Input of the ALU operation
MEM_rs2_data	Input	[31:0]	Input of data rs2
MEM_rd_addr	Input	[4:0]	Address for the destination register
WB_wb_en	Output	[1:0]	Control signals for the WB stage
WB_rd_en	Output	[1:0]	Enable signal for the register file
WB_pc_four	Output	[31:0]	The PC incremented by 4 for the next stage

WB_ld_data	Output	[31:0]	Data loaded from memory
WB_alu_data	Output	[31:0]	The result of the ALU operation
WB_rd_addr	Output	[4:0]	Address for the destination register for the WB stage
Io_hex [0:7]	Output	[31:0]	Output LEDs from LSU

Block diagram of the Memory stage:

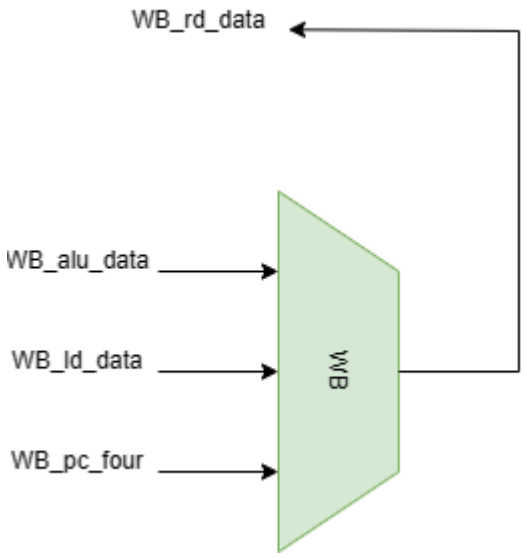


Module name: wb_cycle

Signal	Type	Size	Description
WB_ld_data	Input	[31:0]	Data loaded from memory

WB_alu_data	Input	[31:0]	The result from the ALU operation
WB_pc_four	Input	[31:0]	The PC incremented by 4
WB_data	Output	[31:0]	The selected data to be written to the register file

Block diagram of the Write Back stage:



2.2. Design Models

2.2.1. Non-Forwarding Model

Description: This model of the processor tends to function as adding no-operation (nop) between each stage if the next instructions used the previous instructions data that have not been done through the Write-Back stage yet in order to prevent Data hazard happening during operating.

Design Strategy: By adding an additional block to control Data hazard happens when the instruction in the EX stage has to use the data that is still in the MEM stage or WB stage. So the instruction in the ID stage can not load into the EX stage for processing.

EX-MEM Data hazard:

This hazard appears when the instruction in the EX stage needs the data that is still in the MEM stage.

EX-WB Data hazard:

This hazard appears when the instruction in the EX stage needs the data that is still in the WB stage.

ID-WB Data hazard:

This hazard appears when the instruction in the ID stage needs the data that still needs 1 clock cycle to write the data into the REG FILE.

Here is the examples:

Instruction	IF	ID	EX	MEM	WB
i0: add x5, x3, x2	i0				
i1: xor x6, x5, x1	i1	i0			
i2: sub x9, x3, x5	i2	i1	i0		
i3: or x2, x7, x5	i2	i1	nop	i0	
i4: sll x4, x5, x5	i2	i1	nop	nop	i0
	i2	i1	nop	nop	nop
	i3	i2	i1	nop	nop

EX-WB Data hazard appears when the instruction i1 in the EX stage needs data of x5 which is in instruction i0 have not done yet.

Instruction	IF	ID	EX	MEM	WB
i0: add x4, x3, x2	i0				
i1: lw x5, 0x40 (x1)	i1	i0			
i2: sub x9, x5, x1	i2	i1	i0		
i3: or x2, x7, x5	i3	i2	i1	i0	
i4: sll x4, x5, x1	i3	i2	nop	i1	i0
	i3	i2	nop	nop	i1
	i3	i2	nop	nop	nop
	i4	i3	i2	nop	nop

EX-MEM Data hazard: appears when the instruction i2 in EX stage needs data of x5 which is in WB stage of instruction i1.

Instruction	IF	ID	EX	MEM	WB
i0: add x4, x3, x2	i0				
i1: beq x5, x6, _L0	i1	i0			
i2: sub x9, x5, x1	i2	i1	i0		
… i8: _L0 sll x4, x5, x1	i3	i2	i1	i0	
	i8	nop	nop	i1	i0
i9: xor x6, x8, x2	i9	i8	nop	nop	i1
		i9	i8	nop	nop

ID-EX Data hazard: appears when the instruction i1 in EX stage is a branch or jump that occurs, which needs to load instruction i8 into IF stage for next operation.

Solution:

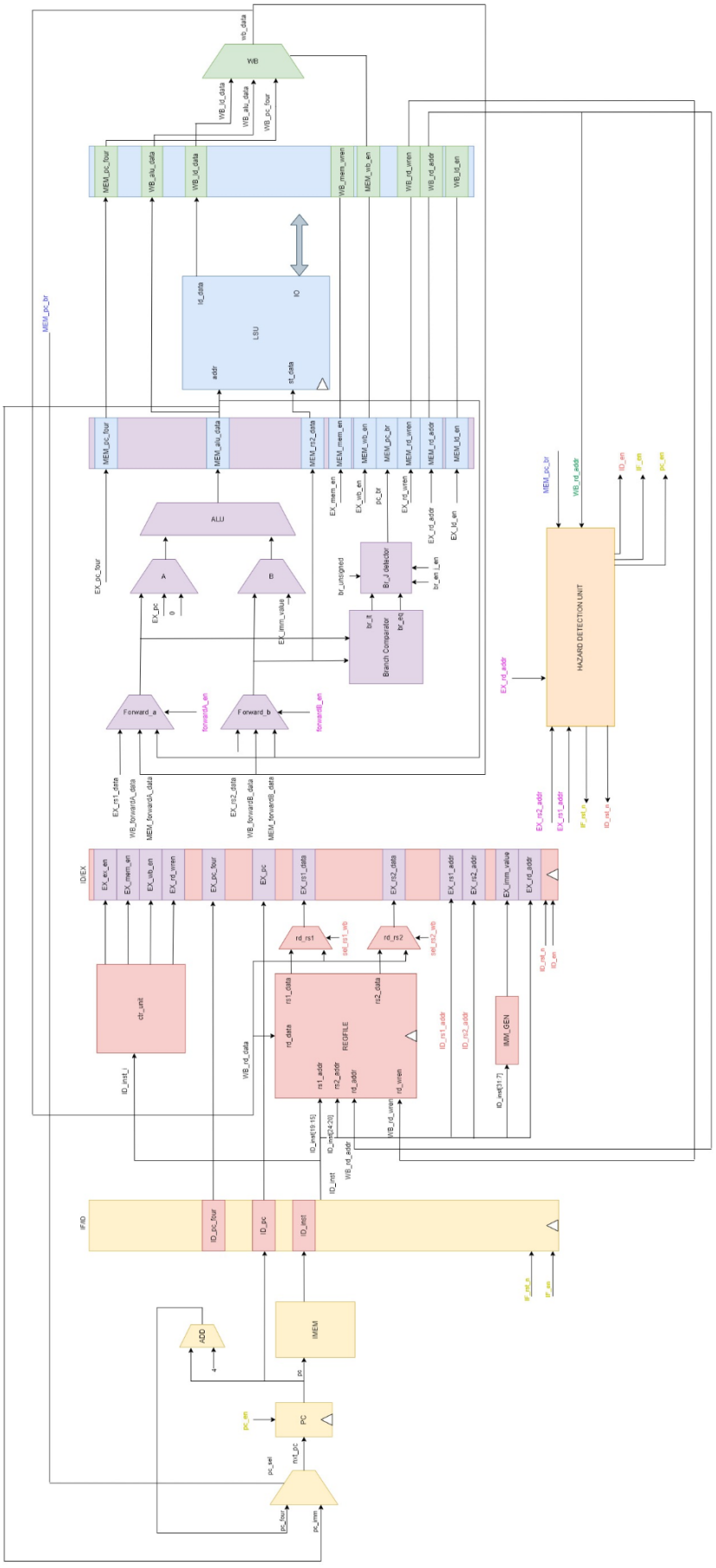
For EX-MEM Data hazard and EX-WB Data hazard:

Non - Forward the data in the MEM stage and WB if the instruction in EX stage is needed.

For ID-EX Data hazard:

Non - Forward the data in the ID and EX stage then load the destination instruction into IF stage for continuing.

Overview of the Forwarding RISC-V design:



2.2.2. Forwarding Model

Description: To increase the speed of the processor, compared to the Non-Forwarding Model, we use the forwarding technique to skip the waiting time for the data calculation between each stage.

Design Strategy:

This Data hazard happens when the instruction in the EX stage has to use the data that is still in the MEM stage or WB stage. So the instruction in the EX stage cannot load the data for processing.

Same with Non - Forwarding, we have to detect the data hazard that can appear. There are 3 main data hazards that we have to detect.

EX-MEM Data hazard:

This hazard appears when the instruction in the EX stage needs the data that is still in the MEM stage.

EX-WB Data hazard:

This hazard appears when the instruction in the EX stage needs the data that is still in the WB stage.

ID-WB Data hazard:

This hazard appears when the instruction in the ID stage needs the data that still needs 1 clock cycle to write the data into the REG FILE.

Here is the example:

Instruction	IF	ID	EX	MEM	WB
i1: add x3 x2 x1	i1				
i2: sub x4 x2 x3	i2	i1			
i3: and x5 x4 x3	i3	i2	i1		
i4: sll x6 x4 x3	i4	i3	i2	i1	
		i4	i3	i2	i1
			i4	i3	i2

EX-MEM Data hazard appears when the instruction i2 in EX stage need data of x3 which is in MEM stage of instruction i1

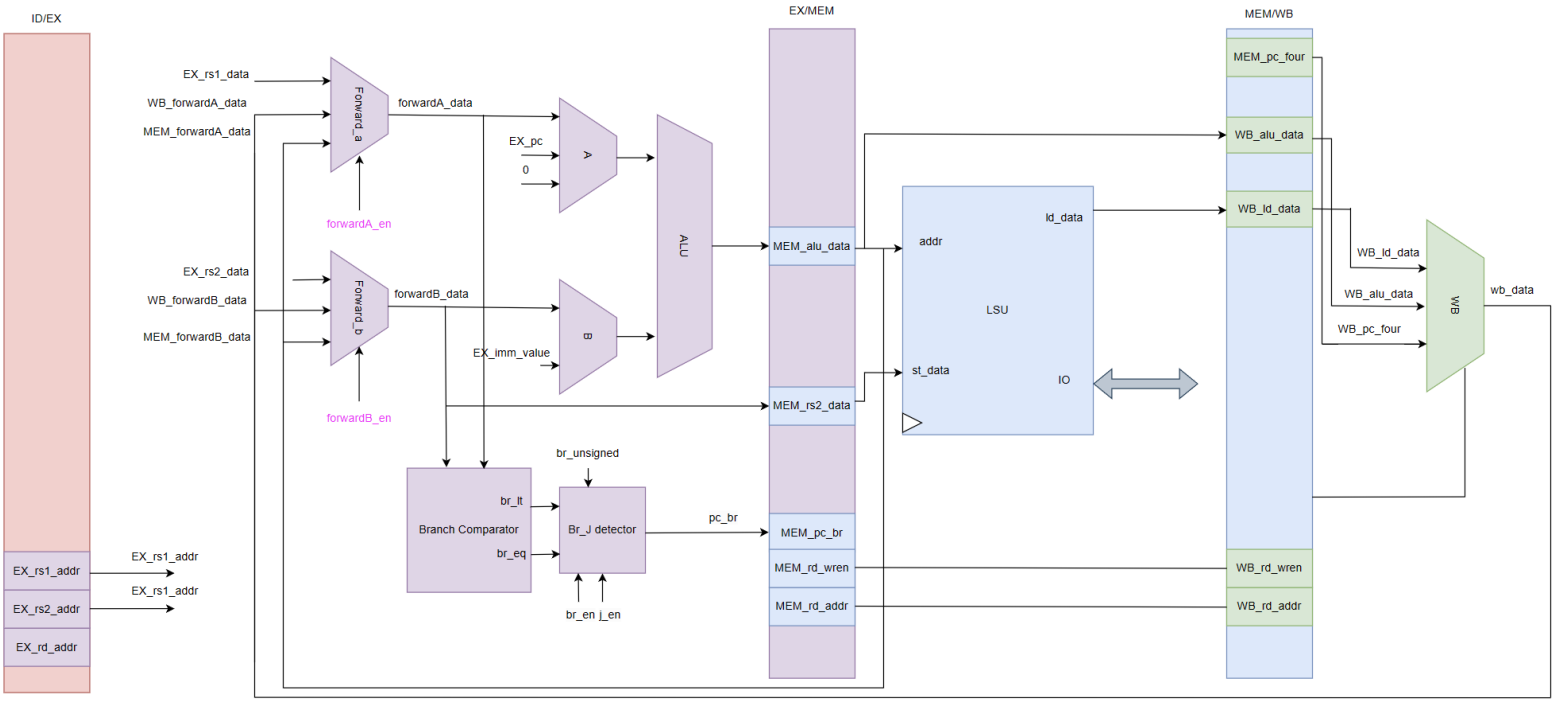
EX-WB Data hazard: appears when the instruction i4 in EX stage needs data of x4 which is in WB stage of instruction i2

ID-WB Data hazard: appears when the instruction i4 in ID stage need data of x4 which is in WB stage of instruction i1 and need 1 cycle to write the data into x4 of the REG FILE

Solution: For EX-MEM Data hazard and EX-WB Data hazard:

Forward the data in the MEM stage and WB if the instruction in EX stage is needed

Block Diagram:



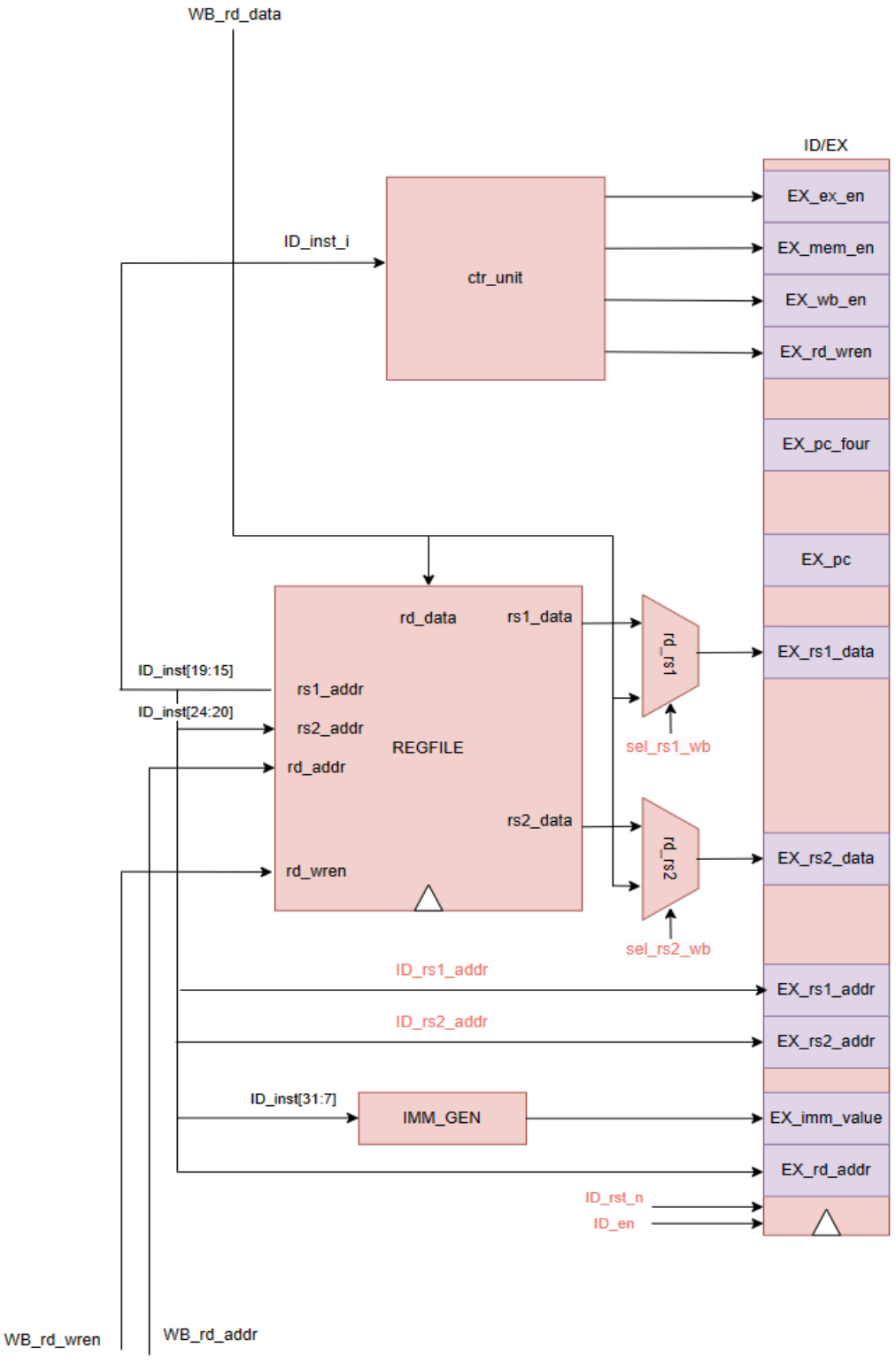
Description: The forward data from MEM and WB will be selected by the forwardA_en and forwardB_en. These select signals will be based on the comparison between EX_rs1_addr , EX_rs2_addr with the MEM_rd_addr and WB_rd_addr.

```
If (MEM_rd_wren  &&  (MEM_rd_addr != 0) && (MEM_rd_addr == EX_rs1_addr))
    forwardA_en = 2'b10;                // Forward form MEM
else if (WB_rd_wren  &&  (WB_rd_addr != 0) && (WB_rd_addr == EX_rs1_addr))
    forwardA_en = 2'b01;                // Forward form WB
else
    forwardA_en = 2'b00;                // No Forward
```

Similar to forwardB_en with the comparison of rd_addr in WB, MEM stages and rs1_addr in EX stage

Solution for ID-WB Data hazard: We will make the select component in the ID and select the data that need 1 cycle to write into the REG FILE if the ID instruction needs that data.

Block Diagram:



Description: The ID_rs1_addr and ID_rs2_addr will compare to the WB_rd_addr to detect if the instruction in ID stage needs to use the rd data.

```
if (WB_rd_wren && (WB_rd_addr != 0) && (WB_rd_addr == ID_rs1_addr))
        sel_rs1_wb = 1;
```

```
    else     sel_rs1_wb = 0;
if (WB_rd_wren && (WB_rd_addr != 0) && ((WB_rd_addr == ID_rs2_addr))
            sel_rs2_wb = 1;
    else     sel_rs2_wb = 0;
```

New Problem:

When dealing with the load instruction, the load instruction can only output the data when going through the DMEM, which means it cannot forward from the MEM_alu_data.

We have to make the IF, ID and EX stage wait for the MEM stage to load the data and go to WB stage for forwarding and reset the MEM stage for the previous instruction.

Example:

Instruction	IF	ID	EX	MEM	WB
i1: lw x3 0x02 (x1)	i1				
i2: sub x4 x2 x3	i2	i1			
i3: and x5 x4 x3	i3	i2	i1		
i4: sll x6 x4 x3	i4	i3	i2	i1	
	i4	i3	i2	nop	i1
		i4	i3	i2	nop

In this example, the instruction i2 in EX stage needs data x3 which is taken from DMEM in the instruction in MEM stage. The i1 instruction at this time cannot have the data for forwarding and has to wait until go into the WB stage.

Solution: Detect the load instruction in MEM stage and disable the pc, IF, ID EX stages anh reset the MEM stage

```
    if ((!MEM_mem_wren & MEM_ld_en) && (MEM_rd_addr != 0) && (MEM_rd_addr ==
EX_rs1_addr)) begin
        IF_ID_en        = 0;
        ID_EX_en        = 0;
        EX_MEM_en       = 0;
```

```
        pc_en               = 0;
        EX_MEM_rst_n  = 0;
    end
    else if ((!MEM_mem_wren & MEM_ld_en) && (MEM_rd_addr != 0) && ((MEM_rd_addr == EX_rs2_addr) begin
        IF_ID_en          = 0;
        ID_EX_en          = 0;
        EX_MEM_en      = 0;
        pc_en               = 0;
        EX_MEM_rst_n  = 0;
    end
    else begin
        IF_ID_en          = 1;
        ID_EX_en          = 1;
        EX_MEM_en      = 1;
        pc_en               = 1;
        EX_MEM_rst_n  = 1;
    end
```

In some cases, some instructions such as I-type, L-type and jalr, which do not use the data rs2 but this type of design compares 2 addresses rs1 and rs2 at the same time. There will be the possibility that the immediate value will overlap the rs2 address that needs to check forwarding.

Solution: Create the signal that can detect the I-type, L-type and jalr instruction in EX stage and ID stage (2 stages that need forwarding value) and check with the comparison of rd_addr and rs2_addr between EX and MEM, EX and WB, ID and WB

```
// Forward from MEM
if (MEM_rd_wren && (MEM_rd_addr != 0) && ((MEM_rd_addr == EX_rs2_addr) & !EX_rd_rs2_en))
// Forward from WB
else if (WB_rd_wren && (WB_rd_addr != 0) && ((WB_rd_addr == EX_rs2_addr) & !EX_rd_rs2_en) )
```

// Check load instruction

else if ((!MEM_mem_wren & MEM_ld_en) && (MEM_rd_addr != 0) && ((MEM_rd_addr == EX_rs2_addr) & **!EX_rd_rs2_en**))

// Hazard regfile

if (WB_rd_wren && (WB_rd_addr != 0) && ((WB_rd_addr == ID_rs2_addr) & **!ID_rd_rs2_en**))

For the Branch and Jump instruction:

If there is no pc jump, the will be ok like in this example,

Instruction	IF	ID	EX	MEM	WB
i1: beq x3 x2 LOOP1	i1				
i2: sub x4 x2 x3	i2	i1			
i3: and x5 x4 x3	i3	i2	i1		
…	i4	i3	i2	i1	
LOOP1 i7: sll x6 x4 x3		i4	i3	i2	i1
			i4	i3	i2

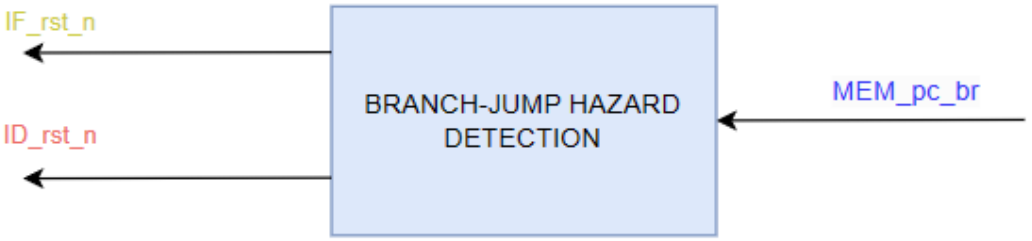
If the branch instruction i1 is jump after move to the MEM stage, the instruction i2 and i3 in stage EX and ID need to be clear and the i4 is replace by the instruction i7

Instruction	IF	ID	EX	MEM	WB
i1: beq x3 x2 LOOP1	i1				
i2: sub x4 x2 x3	i2	i1			
i3: and x5 x4 x3	i3	i2	i1		
…	i7	nop	nop	i1	

LOOP1 i7: sll x6 x4 x3	…	i7	nop	nop	i1
	…	…	i7	nop	nop

Solution: Create the component to detect the branch and reset stage if the branch is correct. We have the **Branch and Jump Hazard Detector.**

Block Diagram:



Specification:

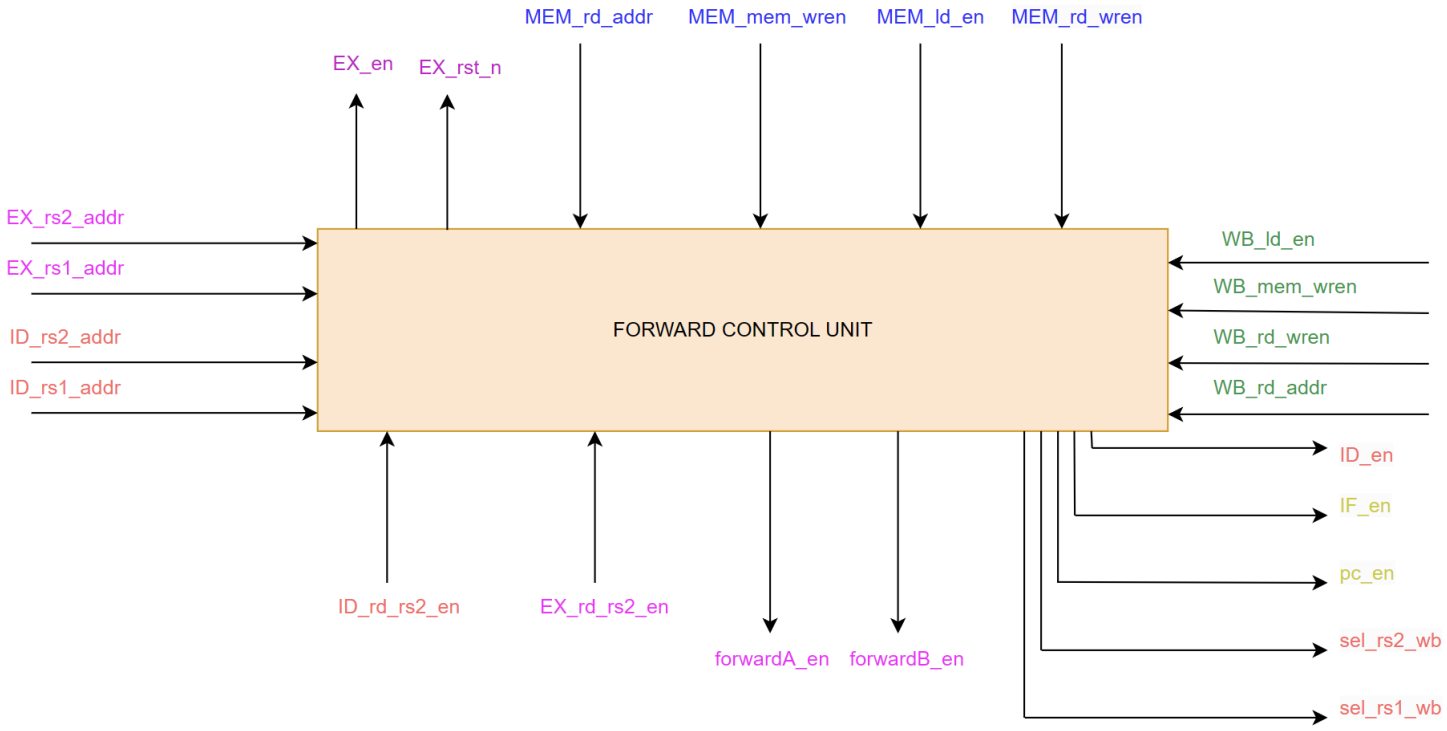
Signals	Type	Description
MEM_pc_br	**Input**	Signal for detecting if the branch instruction is correct or not.
IF_rst_n **ID_rst_n**	**Output**	Reset signal for IF/ID Register Reset signal for ID/EX Register

Description: This component will reset the data in ID and EX stage (Active low the **IF_rst_n** and **ID_rst_n**) if the **MEM_pc_br** signal (active high if the branch comparison is correct).

The purpose of this is when updating the new PC calculated by the ALU and given into the IMEM from MEM stage if the branch comparison is correct, the instruction after the branch is in the ID and EX stage, we have to remove that data.

Forwarding Control Unit:

Block diagram:

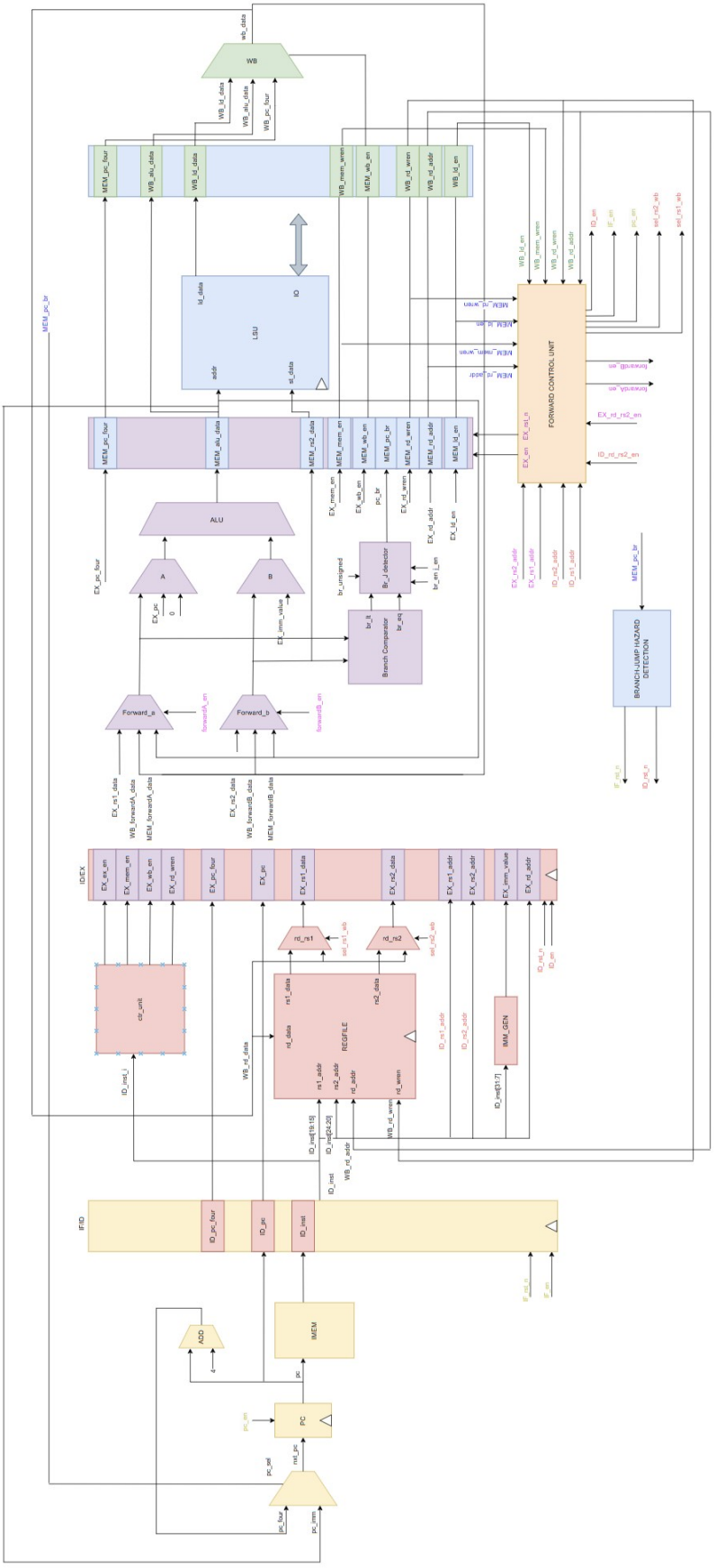


Specification:

Signal	Type	Description
EX_rs2_addr	Input	Address of rs2 for the instruction in EX stage.
EX_rs1_addr		Address of rs1 for the instruction in EX stage.
EX_rd_rs2_en		The enable signal for detecting the instruction in EX stage using rs2 data.
ID_rs2_addr		Address of rs2 for the instruction in ID stage.
ID_rs1_addr		Address of rs1 for the instruction in ID stage.
ID_rd_rs2_en		The enable signal for detecting the instruction in ID stage using rs2 data.
MEM_rd_addr		The RD address of the instruction that needs to write the data back to REG FILE in MEM stage.
MEM_rd_wren		The Write enable signal for write back to REG FILE of the instruction in MEM stage.
MEM_mem_wren		The Write enable signal for load instruction to DATA

		MEMORY of the instruction in MEM stage.
MEM_ld_en		The enable signal for detecting if the instruction in MEM stage is load instruction or not.
WB_ld_en		The enable signal for detecting if the instruction in WB stage is load instruction or not
WB_mem_wren		The Write enable signal for load instruction to DATA MEMORY of the instruction in WB stage.
WB_rd_wren		The RD address of the instruction that needs to write the data back to REG FILE in WB stage.
WB_rd_addr		The RD address of the instruction that needs to write the data back to REG FILE in WB stage.
forwardA_en	Output	A 2-bit select signal for forwarding data of rs1 (00: No Forward; 01: Forward from WB; 10, 11: Forward from MEM)
forwardB_en		A 2-bit select signal for forwarding data of rs2 (00: No Forward; 01: Forward from WB; 10, 11: Forward from MEM)
sel_rs2_wb		Select signal for forward the rd data to rs1 that need one clock cycle to write back to REG FILE
sel_rs1_wb		Select signal for forward the rd data to rs2 that need one clock cycle to write back to REG FILE
IF_en		Enable signal for IF/ID Register
pc_en		Enable signal for PC
ID_en		Enable signal for ID/EX Register

Overview of the Forwarding RISC-V design:



2.2.3. Static Branch Prediction Model

Description: To increase the correctness of the processor, compared to the Forwarding Model, we use the static branch prediction technique to predict whether the program is looping or not for reloading the address of the previous instructions.

Design Strategy:

This branch prediction happens when the instructions are Branch or Jump type, which in the first cycle the subroutine programs have created the PC address before and looping for times. Therefore, by applying this technique we can reduce the time-consuming for recalculating the next PC address in each subroutine.

First, we have to detect whether the addresses can be taken. There are 2 main PC's address that we have to detect.

MEM PC address:

This address happens when the instruction in the MEM stage needs the address for branching (in always-taken mode).

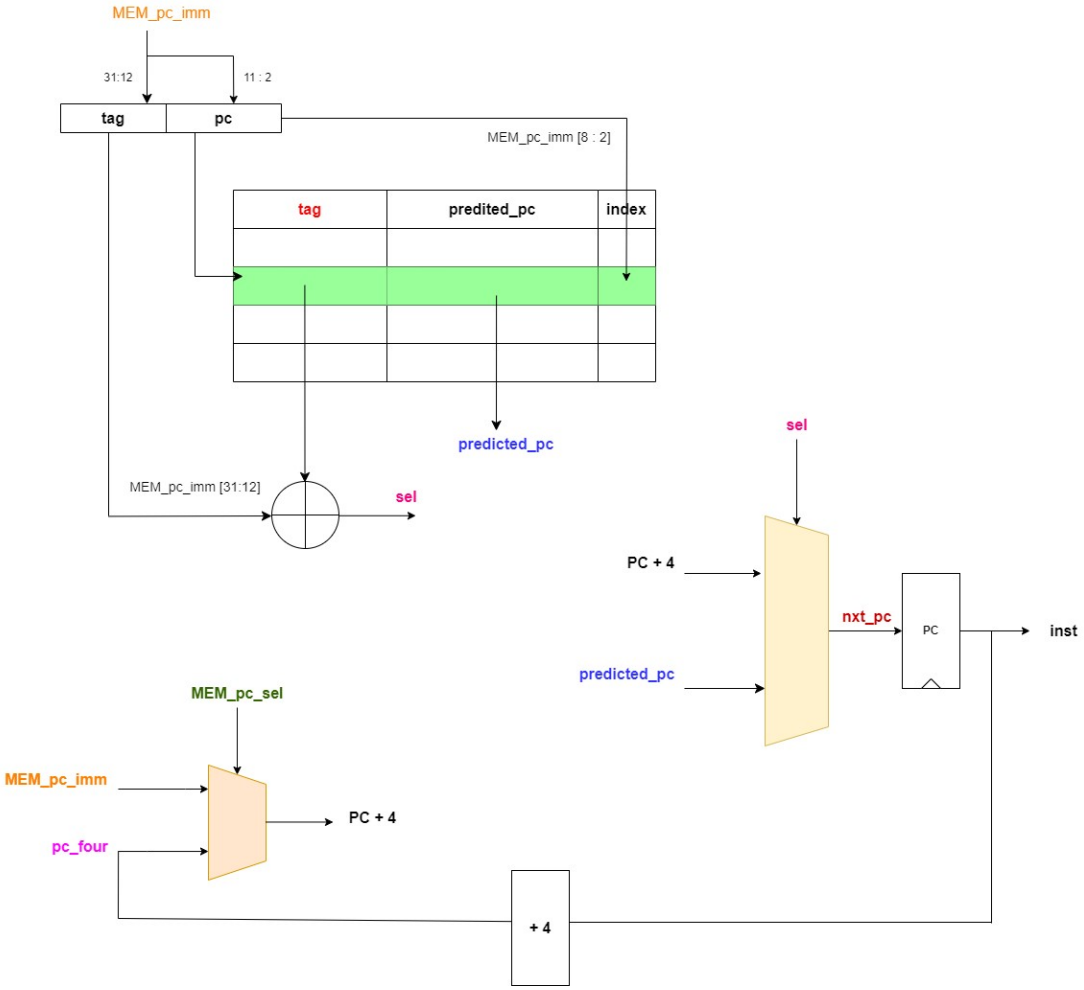
IF PC address:

This address happens when the instruction is normally functioning.

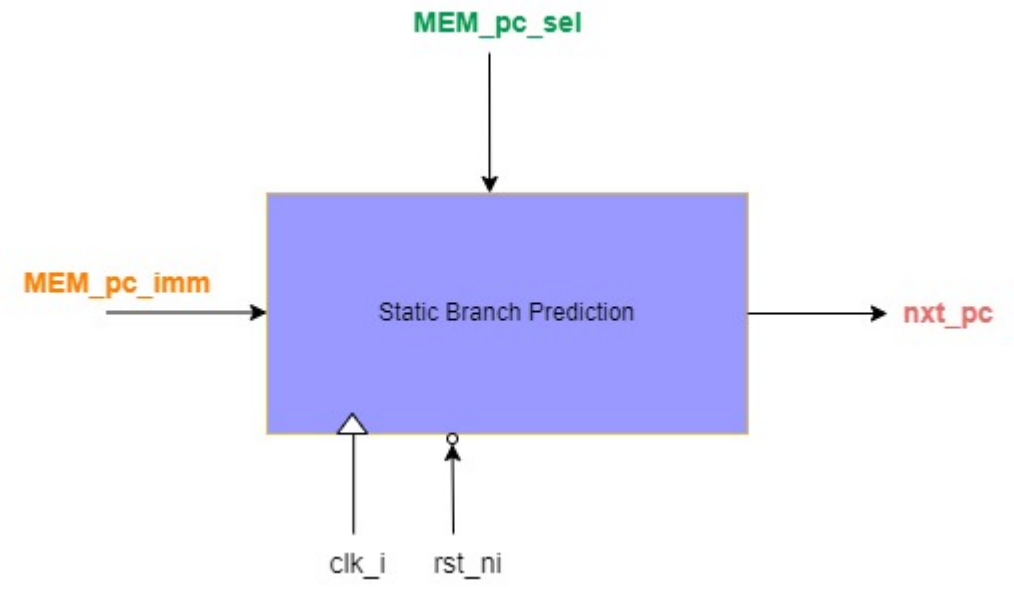
Second, we create a table for storing those addresses for use in the next subroutines program.

Name	Size	Description
index	[6 : 0]	The low bits of PC that are used for defining the index of PC to store the Tag and Predicted PC.
tag	[31 : 12]	The upper bits of the PC are stored in the Buffer which is used to define the label address for easy comparison with the next PC.
predicted_pc	[31 : 2]	The full address of the PC that stores in the Buffer for the next address that the subroutine will or is going to branch or jump to.

Block diagram detail:



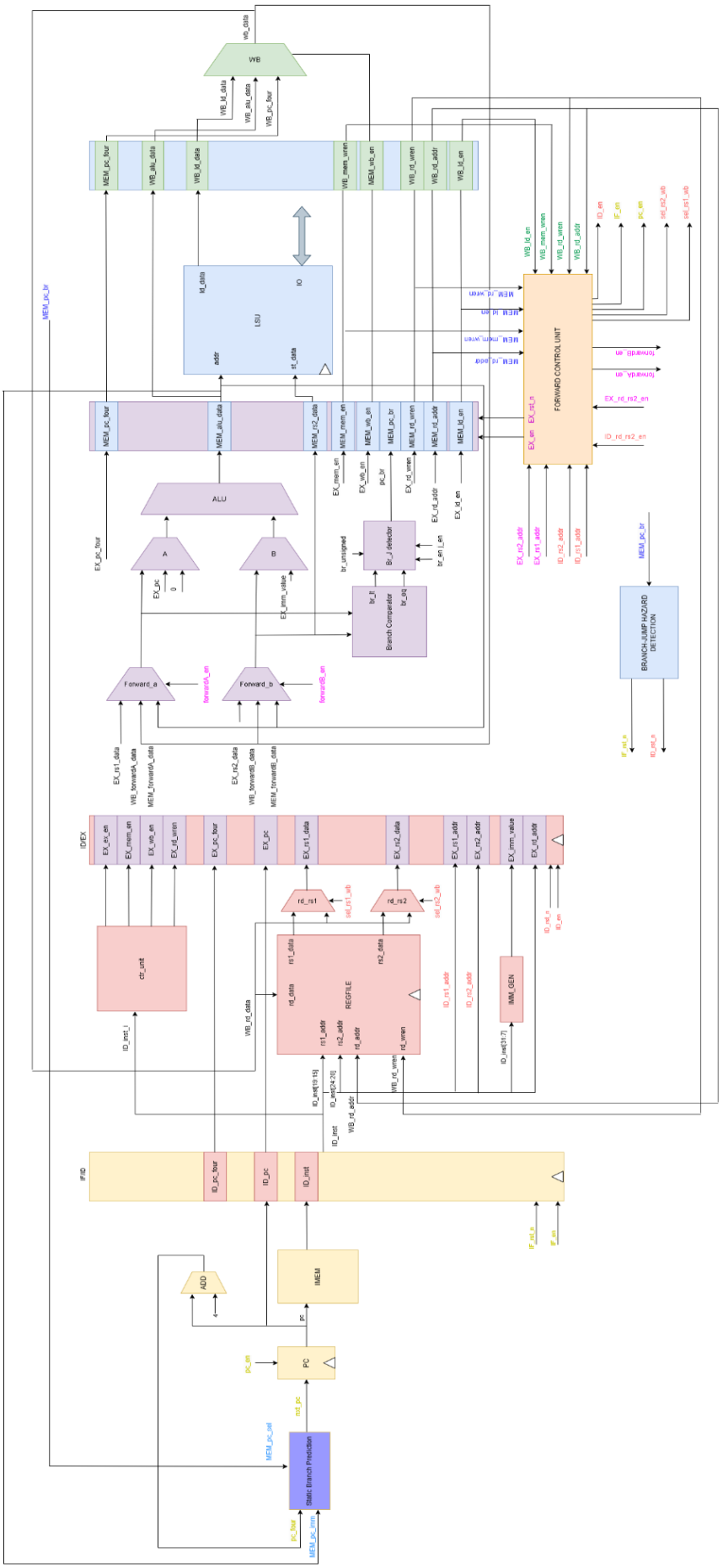
General Block diagram:



Specification:

Signal	Type	Description
MEM_pc_imm	Input	The PC that write-back from the MEM stage, calculated from Branch and Jump instructions for defining the next movement address.
MEM_pc_sel		Select signal to control the output nxt_pc signal in order to point address directly whether the instructions are Jump and Branch or others.
nxt_pc	Output	The next PC for the running program.

Overview of the Always-taken RISC-V design:


WB
LSU
FORWARD CONTROL UNIT
ALU
Branch Comparator
Br_J detector
Forward_a
Forward_b
BRANCH JUMP HAZARD DETECTION
REGFILE
IMM_GEN
ctr_unit
IMEM
ADD
PC
Static Branch Prediction

3. Verification Strategy.

3.1. Testbench - Test program

Strategy:

We will create a program that can represent all types of instructions and has all types of hazards that I have mentioned in the previous path.

We will get the input data from the input random value io_sw of the design that is programmed in the driven.cpp file, so that the data used in the testbench is much more reliable.

The testing program can detect the error by the branch instructions, we use these instructions as a checking condition, we will create new data based on the sw input and check these new values with suitable conditions in our plant.

We will have 3 main cases for this program:

```
#TESTBENCH FOR PROCESSOR

## LOAD DATA FOR TESTING
### USING X1 TO X10 AS VARIABLES
addi x31 x0 0x7ff

### USING X29 TO COUNT THE BRANCH THAT IS MISS WHEN NOT-TAKEN

lw x1 0x101(x31)
lw x2 0x101(x31)
lw x3 0x101(x31)
lw x4 0x101(x31)
lw x5 0x101(x31)
```

```
add x6 x0 x1
add x7 x0 x2
add x8 x0 x3
add x9 x0 x4
add x10 x0 x5
```

```
## TEST CASE 1: TEST WITH NO DATA HAZARD

                    beq x1 x6       CHECK_1
END_CHECK_1:
                    beq x2 x7       CHECK_2
END_CHECK_2:
                    beq x3 x8       CHECK_3
END_CHECK_3:
                    beq x3 x8       CHECK_4
CHECK_1:
                addi x11 x1 111                 # x11 = x1 + 111
                addi x16 x6 121                 # x16 = x6 + 121 = x11
                bne x11 x16 END                 # Not Jump
                 jal x29 END_CHECK_1
CHECK_2:
                or x12 x2 x7                    # x12 = x2 = x7
                bne x12 x2 END                  # Not Jump
                bne x12 x7 END                  # Not Jump
                xor x17 x12 x7                  # x17 = x12 = x7
                bne x17 x0 END                  # Not Jump
                 jal x29 END_CHECK_2
CHECK_3:
```

```
                    sub x13 x3 x8
                    bne x13 x0 END                  # x13 = 0 (x3 = x8)
                    and x18 x13 x8                  # x18 = x13 (= x8)
                     or x13 x3 x8                   # x13 = x3 (= x8)
                    bne x13 x18 END                 # Not Jump
                     jal x29 END_CHECK_3
CHECK_4:
                    sw x4 0x0(x0)
                     lw x14 0x0(x0)
                     lh x15 0x0(x0)
                     lb x16 0x0(x0)
                     lhu x17 0x0(x0)
                     lbu x18 0x0(x0)

                     or x19 x4 x9                       # x19 = x4 (= x9)
                     bne x19 x14 END                    # Not Jump
                     bltu x15 x17 END                   # Not Jump
                     bltu x16 x18 END                   # Not Jump

                     jal x29 CASE_2
```

```
    # TEST CASE 2: TEST WITH DATA HARZARD
    CASE_2:         auipc x29 0
    ## EX-MEM
            add x11 x1 x6                   # x11 = 2.x1
        add x16 x11 x6                      # x16 = 3.x1
        sub x17 x16 x1                      # x17 = 2.x1
        or x18 x1 x6                # x18 = x1
        sub x19 x18 x1              # x19 = 0
      # Check
```

```
        bne x19 x0 END          # Not Jump
        bne x17 x11 END         # Not Jump

## EX-MEM with load
    lw x11 0x0(x31)             # load from sw
    add x16 x0 x11              # x16 = x11
    lh x12 0x0(x31)             # load from sw
    or x17 x0 x12               # x17 = x12
    lb x13 0x0(x31)             # load from sw
    and x18 x13 x0              # x18 = 0

  # Check
        bne x18 x0 END          # Not Jump
    bne x17 x12 END             # Not Jump
    bne x16 x11 END             # Not Jump

## EX-MEM and EX-WB
  add x11 x2 x7                 # x11 = 2.x2
    add x16 x11 x6              # x16 = 3.x2
    sub x17 x16 x11             # x17 = x2
    or x18 x17 x6               # x18 = x2
    and x19 x18 x17             # x19 = x2
  # Check
        bne x19 x18 END

## ID-WB
        add x11 x3 x8           # x11 = 2.x3
    add x16 x11 x8              # x16 = 3.x3
    sub x17 x16 x11             # x17 = x3
    or x18 x17 x11              # x18 > x11 > x3 = x8 (Unsigned)
```

```
        # Check
            bne x17 x3 END                  # Not Jump
```

```
# TEST CASE 3: PROGRAM WITH LOOP AND HAZARD COMBINATION
addi x20 x20 0              # Reset the counter
                    addi x11 x1 20
                    addi x16 x6 30
CHECK_LOOP1:        bne x11 x16 LOOP1

                    addi x12 x1 30
                    addi x17 x6 40
CHECK_LOOP2:        bne x12 x17 LOOP2

                    addi x13 x1 40
                    addi x18 x6 50
CHECK_LOOP3:        bne x13 x18 LOOP3

                    jal x29 END_CHECK

LOOP1:              addi x16 x16 -1
                    addi x20 x20 1
                     jal x21 CHECK_LOOP1

LOOP2:              addi x17 x17 -1
                    addi x20 x20 1
                     jal x21 CHECK_LOOP2

LOOP3:              addi x18 x18 -1
                    addi x20 x20 1
```

```
                    jal x21 CHECK_LOOP3

END_CHECK:          addi x20 x20 -30        # Check if the number of counter is correct or not
                    bne x20 x0 END

                    auipc x29 0             # Instruction check if the program run completely
END:                auipc x30 0
```

4. Evaluation.

4.1. IPC Result

As our group has not finished the Static Branch Prediction, so we don't have the value of Pmiss (The percentage of Branch prediction that is miss). But tomorrow we will represent our ideal and maybe we will finish the module.

4.2. Application

Description:

We use the designed processor to calculate the SW value from binary into decimal and output it on 5 HEX 7-segment LEDs. As the number of SW in the DE2 kit is 17 (SW 17 for RST_ni signal), so the maximum number of input is just 0xFFFF = 65,535, which means we just use 5 HEX 7-segment LEDs.

Program:

```
# Application
    # x10 to store the address for sw register 0x900
addi x10 x0 0x7ff
addi x10 x10 0x101
    # x15 to store the address for HEX register start from 0x800
addi x16 x0 0x7ff
```

```
        # Load the sw value into the register x1
lw x1 0x0(x10)


addi x2 x0 10              # parameter HEX1
addi x3 x0 100                    # parameter HEX2
addi x4 x0 1000          # parameter HEX3
#store value 10000 into x5
addi x5 x0 0                                # Initial value
addi x6 x0 5                                # Counter
LOOP_HEX4:
beq x6 x0 END_LOOP4
   addi x5 x5 2000
   addi x6 x6 -1
jal x7 LOOP_HEX4


END_LOOP4:
addi x15 x0 0 # Clear register x14 (HEX3)
addi x14 x0 0 # Clear register x14 (HEX3)
addi x13 x0 0 # Clear register x13 (HEX2)
addi x12 x0 0 # Clear register x12 (HEX1)
addi x11 x0 0 # Clear register x11 (HEX0)


HEX4_LOOP:
blt x1 x5 HEX3_LOOP
sub x1 x1 x5
addi x15 x15 1 # Increase the thousands value HEX3 to 1 after each comparison
jal x7 HEX4_LOOP


HEX3_LOOP:
blt x1 x4 HEX2_LOOP
sub x1 x1 x4
addi x14 x14 1 # Increase the thousands value HEX3 to 1 after each comparison
```

```
jal x7 HEX3_LOOP


HEX2_LOOP:
blt x1 x3 HEX1_LOOP
sub x1 x1 x3
addi x13 x13 1 # Increase the hundreds value HEX3 to 1 after each comparison
jal x7 HEX2_LOOP


HEX1_LOOP:
blt x1 x2 HEX0_LOOP
sub x1 x1 x2
addi x12 x12 1 # Increase the tens value HEX3 to 1 after each comparison
jal x7 HEX1_LOOP


HEX0_LOOP:
add x11 x0 x1


# Set the display value for HEX0 // 0x800
sw x11, 0x01(x16)
# Set the display value for HEX1 // 0x810
sw x12, 0x11(x16)
# Set the display value for HEX2 // 0x820
sw x13, 0x21(x16)
# Set the display value for HEX3 // 0x830
sw x14, 0x31(x16)
# Set the display value for HEX4 // 0x840
sw x15, 0x41(x16)
```

Result:

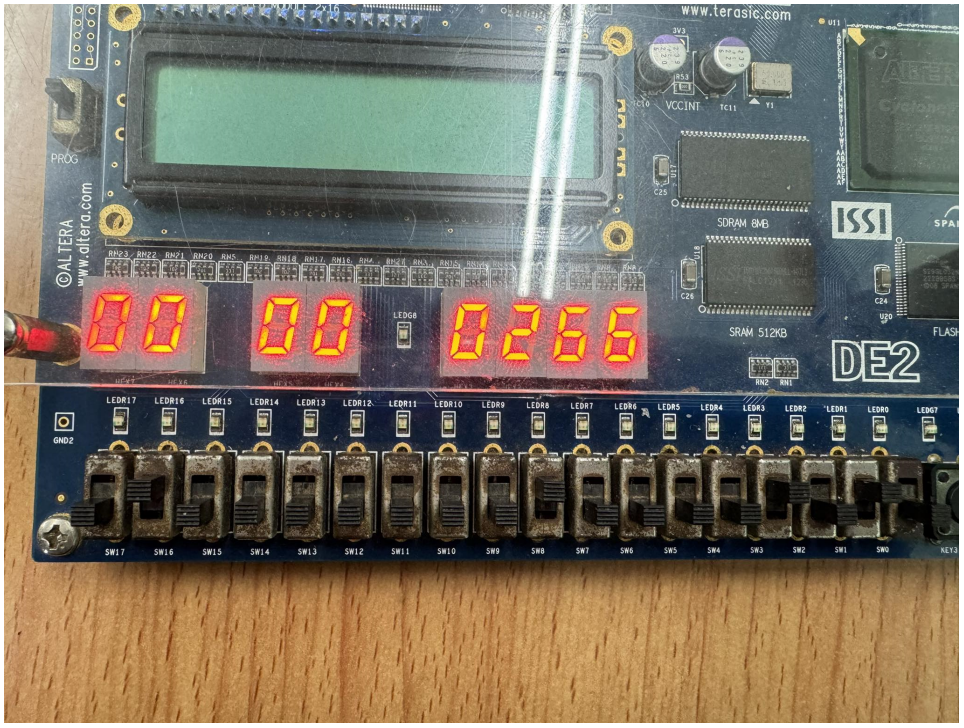


Figure 4.2a: Result with the input = 0001_0000_1010

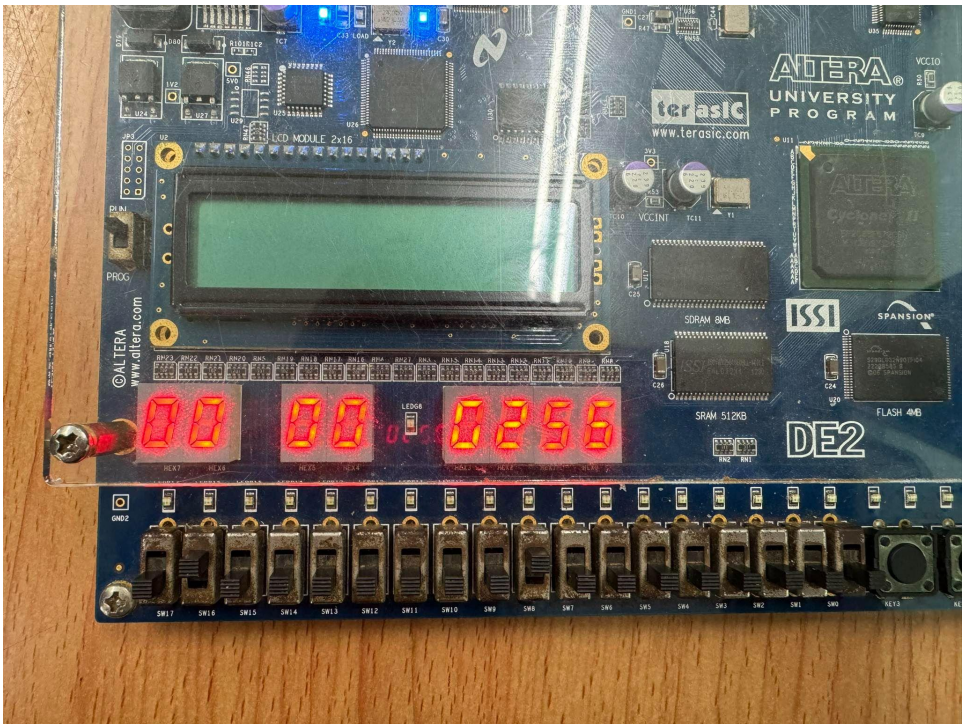


Figure 4.2b: Result with the input = 0001_0000_0000

5. Conclusion.

In conclusion, the development of the 5-Stages Pipelined Processor based on the construction of the Single Cycle Processor has been a comprehensive and enlightening project. The primary goal of this endeavor was to design a processor using 5 stage pipelined to reduce the time for the processor to process the instructions.

The successful completion of this project underscores the importance of a systematic approach to processor design, encompassing architecture, instruction set implementation, and rigorous testing methodologies. The project not only enhanced understanding of the pipeline but also provided valuable insights into the challenges and considerations involved in designing processors for real-world applications.