Department of Electrical and Computer Engineering

ENCS4370

Computer Architecture – Project No. 2

Fall Semester 2025/2026

Design and Verification of a Simplified Predicated RISC Processor using Verilog

Students info:

Student 1: Heba Dababat ID: 1230151 SEC: 1

Student 2: Mayar Etawi ID: 1230501 SEC: 1

Student 3: Rahaf Salahat ID: 1231584 SEC: 2

Date: 7/1/2026

Abstract

This report documents the design and implementation of a 32-bit pipelined predicated RISC processor. We explored RISC architecture principles by designing a processor that supports predicated execution, where instructions execute conditionally based on predicate register values. Our design includes 32 general-purpose registers, separate instruction and data memories, and implements 14 instructions covering arithmetic, logical, memory, and control operations across three instruction formats (R-Type, I-Type, and J-Type). We implemented the complete processor in Verilog using a five-stage pipeline architecture consisting of instruction fetch, decode, execute, memory access, and write-back stages, which allows overlapping execution of multiple instructions to enhance performance. Verification was performed through simulation-based testing using custom testbenches and assembly programs that validated all processor instructions and pipelined operation. This project provided practical experience in pipelined processor design, digital logic implementation, and hardware description languages, connecting theoretical concepts with real-world hardware design challenges.

Table of Contents

Table of Figure

List of tables

Design and Implementation

Processor Specifications

In this processor we chose to implement a pipeline processer . According to these information :

- The instruction size and the word size is 32 bits
- 32 32-bit general-purpose registers from R0 to R31
- R0 is hardwired to zero
- R30 is hardwired to be the PC (Program Counter)
- R31 is hardwired to the return address register.
- The processor has two separate memories, namely, instruction and data memory.
- Word addressable memory
- The processor supports predicated execution, where an instruction executes only if the selected predicate register Rp contains a non-zero value. If $Rp = 0$, the instruction is not executed. Unconditional execution is achieved by using register $R0$as the predicate.

Instruction types format

We designed our processer according to the following instruction format :

- Register type (R – type)

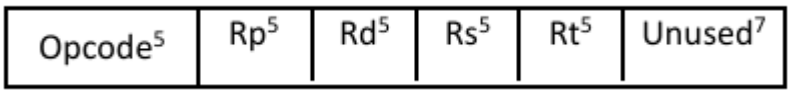


Figure 1: R type Format

➔ Opcode : 5 bit opcode .

➔ Rp : 5-bit Register (predicate register) .

➔ Rs : 5-bit Register .

➔ Rt : 5-bit Register .

➔ Unused 7-bits .

- Immediate type (I -Type)

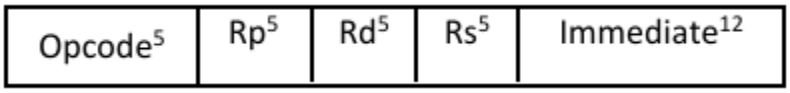


Figure 2: I type Format

➔ Opcode : 5 bit opcode .

➔ Rp : 5-bit Register (predicate register) .

➔ Rs : 5-bit Register .

➔ Rt : 5-bit Register .

➔ 12-bit Immediate , The immediate value is zero-extended for logical instructions, and sign-extended for all other instruction types.

- Jump type (J – Type)

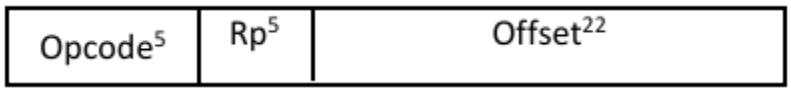


Figure 3: J type Format

➔ Opcode : 5 bit opcode .

➔ Rp : 5-bit Register (predicate register) .

➔ 22-bits Offset to jump and call instructions .

Instruction Encoding

Table 1: Instruction set

Instruction	Meaning	Opcode
R – Type		
ADD Rd, Rs, Rt, Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] + Reg[Rt]	0
SUB Rd, Rs, Rt, Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] - Reg[Rt]	1
OR Rd, Rs, Rt, Rp	if (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] | Reg[Rt]	2
NOR Rd, Rs, Rt, Rp	if (Reg[Rp] ≠ 0) Reg[Rd] = ~(Reg[Rs] | Reg[Rt])	3
AND Rd, Rs, Rt, Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] & Reg[Rt]	4
JR, Rs, Rp	If (Reg[Rp] ≠ 0) , jump to the address in register Rs	14
I – Type		
ADDI Rd, Rs, Imm, Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] + Imm	5
ORI Rd, Rs, Imm, Rp	if (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] | Imm	6
NORI Rd, Rs, Imm, Rp	if (Reg[Rp] ≠ 0) Reg[Rd] = ~(Reg[Rs] | Imm)	7
ANDI Rd, Rs, Imm, Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Reg[Rs] & Imm	9
LW Rd, Imm(Rs), Rp	If (Reg[Rp] ≠ 0) Reg[Rd] = Mem(Reg[Rs] + Imm)	10
SW Rd, Imm(Rs), Rp	If (Reg[Rp] ≠ 0) Mem(Reg[Rs] + Imm) = Reg[Rd]	11
J – Type		
J Label, Rp	If (Reg[Rp] ≠ 0), jump to the target address	12
CALL Label, Rp	If (Reg[Rp] ≠ 0) , jump to the function, store the return address in register R31	13

RTL Design

Note in all these if Rp == 0 it will not execute

R-Type Instructions:

1) Fetch Stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : input1 <= Reg[Rs] and input2 <= Reg [Rt] ,check if Rp is 0 or not

- If Rp equal 0 it will not complete execute

3) ALU (execute) stage : ALUresult <= input1(operation)input2
4) Memory stage : No memory access
5) Write back register stage : Reg[Rd] <= ALUresult

Load (LW) instruction:

1) Fetch stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : input1 <= Reg[Rs] , input2 <= Sign_extend(imm) , check if Rp is 0 or not

- If Rp equal 0 it will not complete execute

3) ALU stage : Result <= input1 + input2 .
4) Memory stage : MEM_DATA <= RAM[Result]
5) Write back register stage : Reg[Rd] <= MEM_DATA

Store (SW) instruction:

1) Fetch stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : input1 <= Reg[Rs] , input2 <= Sign_extend(imm)

- If Rp equal 0 it will not complete execute

3) ALU stage : Result <= input1+input2
4) Memory stage : RAM[Result] <= Reg[Rd]
5) Write back register stage : No write back to register

Jump (J) instruction:

1) Fetch stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : check if (Rp==0) if not : JumpTarget <= PC+Sign_extend(Offset)

- If Rp equal 0 it will not complete execute

3) ALU stage : Next_PC <= JumpTarget
4) Memory stage : No operation
5) Write back register stage : No operation

Call instruction:

1) Fetch stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : check if(Rp==0) if not ReturnAddr <= PC

- If Rp equal 0 it will not complete execute

3) ALU stage : PC<= TargetAddress
4) Memory stage : No operation
5) Write back register stage : Reg[31] <= ReturnAddr

JR instruction:

1) Fetch stage : IR <= MEM[PC] and PC <=PC+1
2) Decode stage : check iff (Rp==0)

- If Rp equal 0 it will not complete execute

3) ALU stage : PC<=Reg [Rs]
4) Memory stage : No operation
5) Write back register stage : No operation

Data path Design

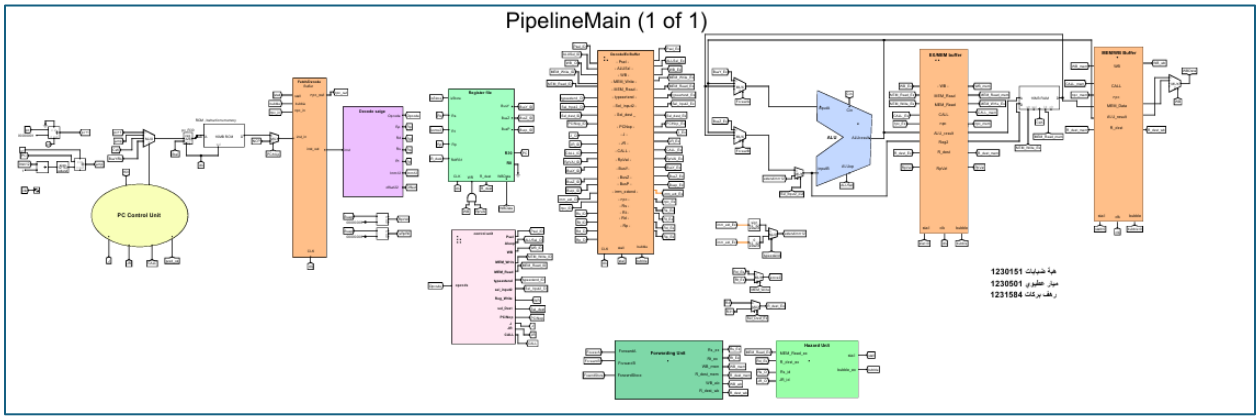


Figure 4: Data path Design

Control Unit Block

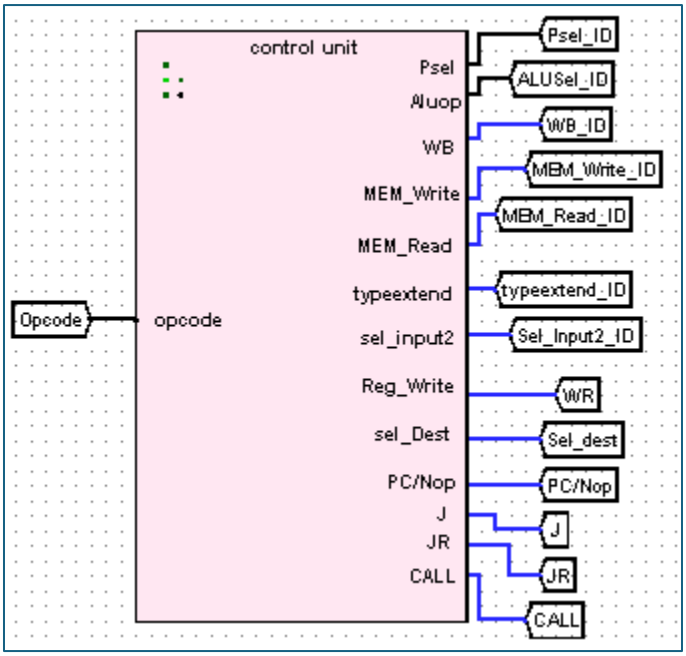


Figure 5: Control Unit

- CODE

```
// Control Unit
// This module generates all control signals based on the instruction opcode.
// A ROM-based control approach is used, where each opcode maps to a fixed
// set of control signals loaded from an external file.
// ------------------------------------------------------------------------
`timescale 1ns/1ps
module control_unit (
    input  [4:0] opcode,   // Instruction opcode field: inst[31:27]
    output reg [1:0] Psel,         // PC source selection
    output reg [2:0] Aluop,        // ALU operation control
    output reg       WB,           // Write-back enable
    output reg       MEM_Write,    // Memory write enable
    output reg       MEM_Read,     // Memory read enable
    output reg       typextend,    // Immediate type extension control
    output reg       Sel_input2,   // ALU second operand selection
    output reg       WR,           // Register file write enable
    output reg       Sel_Dest,      // Destination register selection
    output reg       PCNop,         // PC no-operation / control hazard handling
    output reg       J,             // Jump instruction
    output reg       JR,            // Jump register instruction
    output reg       CALL           // Call instruction
);
    // Control ROM --> Each entry contains the control signals for one opcode
    reg [15:0] ROM [0:31];
    // Load control signals from external file at simulation start
    initial begin
        $readmemh("SignalsFile.txt", ROM);
    end
    // Decode control signals from ROM based on opcode
    always @(*) begin
        WB          = ROM[opcode][0];
        MEM_Write   = ROM[opcode][1];
        MEM_Read    = ROM[opcode][2];
        typextend   = ROM[opcode][3];
        WR          = ROM[opcode][4];
        Sel_input2  = ROM[opcode][5];
        Sel_Dest    = ROM[opcode][6];
        PCNop       = ROM[opcode][7];
        J           = ROM[opcode][8];
        JR          = ROM[opcode][9];
        CALL        = ROM[opcode][10];
        Aluop       = ROM[opcode][13:11];
        Psel        = ROM[opcode][15:14];
    end
endmodule
```

Figure 6: Code Control Unit

The **Control Unit** serves as the central orchestration module of the processor, utilizing a microprogrammed approach to generate necessary control signals. It receives a 5-bit **opcode** which acts as an address to index a 32-entry **ROM**, where pre-defined control words are stored and loaded from an external file. Each bit in the retrieved 16-bit ROM word corresponds to a specific functional signal—such as **Aluop** for mathematical operations, **WB** for register write-back, and **MEM_Read/Write** for data memory access—ensuring that every component in the datapath is correctly configured for the specific instruction being executed.

Control Unit Signals

Table 2: Main Control Units Signal

Signals													
Inst	**Psel**	**ALUOP**	**Call**	**J**	**JR**	**PCNOP**	**Sel_ Dest**	**Sel_input2**	**WR**	**typextend**	**MEM_Read**	**MEM_Write**	**WB**
ADD	00	100	0	0	0	0	0	0	1	0	0	0	1
SUB	00	011	0	0	0	0	0	0	1	0	0	0	1
OR	00	010	0	0	0	0	0	0	1	0	0	0	1
NOR	00	000	0	0	0	0	0	0	1	0	0	0	1
AND	00	001	0	0	0	0	0	0	1	0	0	0	1
ADDI	00	100	0	0	0	0	0	1	1	1	0	0	1
ORI	00	011	0	0	0	0	0	1	1	0	0	0	1
NORI	00	000	0	0	0	0	0	1	1	0	0	0	1
ANDI	00	001	0	0	0	0	0	1	1	0	0	0	1
LW	00	100	0	0	0	0	0	1	1	1	1	0	1
SW	00	100	0	0	0	0	0	1	0	1	0	1	0
J	01	x	0	0	0	0	0	X	0	X	0	0	X
CALL	10	x	1	0	0	0	0	X	1	X	0	0	1
JR	11	x	0	1	1	0	1	X	0	X	0	0	X

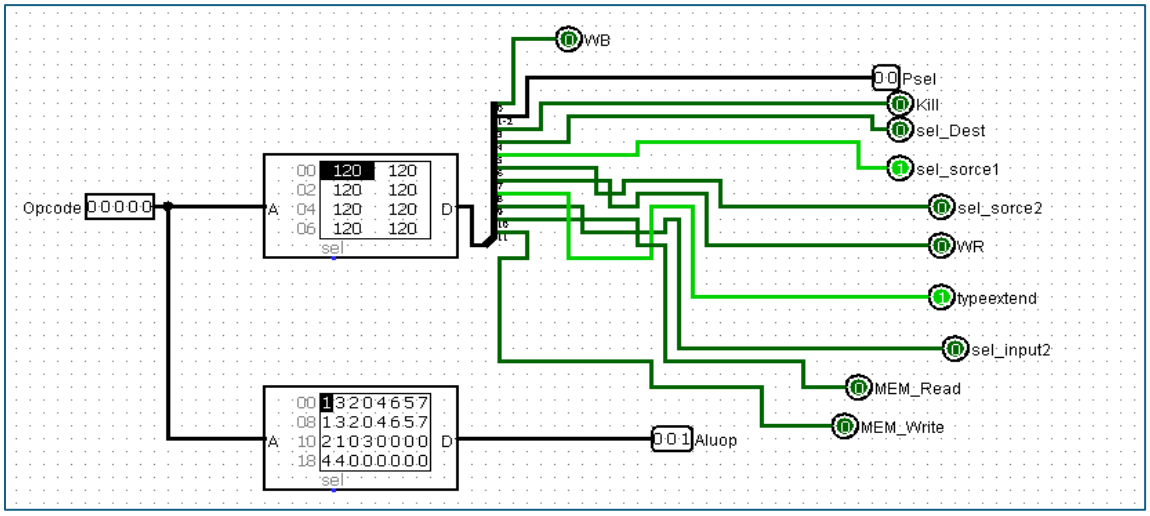


Figure 7: Control Unit

Table 3: Control Unit Truth Table

Signal	Equation
WB	ADD || SUB || OR || NOR || AND || ADDI || ORI || NORI || ANDI || LW || CALL
CALL	CALL
J	J
JR	JR
Sel_Dest	CALL
Sel_input2	ADDI || ORI || NORI || ANDI || SUBI||LW||SW
WR	ADD || SUB || OR || NOR || AND || ADDI || ORI || NORI || ANDI || LW || CALL
Type extend	ORI || NORI || ANDI || LW || SW
ALU op[0]	SUB || AND || ANDI
ALU op[1]	SUB || OR || ORI
ALU op[2]	ADD || ADDI || LW || SW
MEM_Read	LW
MEM_Write	SW
Psel[0]	J || JR
Psel[1]	CALL || JR

Based on their specific Boolean equations. These signals are mapped to each instruction's **Opcode**, as illustrated in the control unit schematic. The control unit functions by using the Opcode as an address to look up the corresponding control word in a ROM-based structure. This ensures that for every instruction—such as **LW, SW, or J**—the correct signals (like **WB, MEM_Read, and Psel**) are asserted to guide the datapath execution, implementing an efficient microprogrammed control approach."

Components

Instruction Memory and Data Memory

Following an evaluation of the instruction set, it was determined that distinct memory components are required. The architecture utilizes separate units for instructions and data to prevent resource contention. This separation ensures that the system can fetch a new instruction while simultaneously performing data load or store operations, thereby adhering to the principle of isolation and maintaining hardware efficiency.

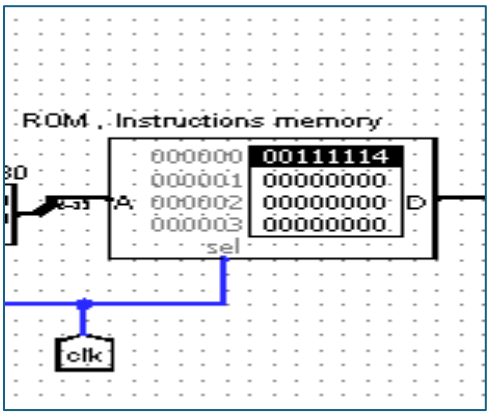


Figure 8: Instruction Memory Module

Code of Instruction memory

```
// Instruction Memory Module
// This module implements a simple instruction memory (ROM).
// It stores 32-bit instructions and outputs one instruction based on the
// provided address.
// The memory is initialized from an external hexadecimal file (InstFile.txt).
`timescale 1ns/1ps
module Inst_MEM (
    input  [31:0] address,   // 32-bit address input from (PC)
    output [31:0] inst       // 32-bit instruction output
);
    // 256 memory locations, each 32 bits wide
    reg [31:0] ROM [0:255];

    // Loads instruction data from "InstFile.txt" into the ROM at simulation start
    // The file must contain hexadecimal values
    initial begin
        $readmemh("InstFile.txt", ROM);
    end
    // Instruction fetch:
    // Uses the lower 8 bits of the address to index the ROM
    // This allows addressing 256 instruction locations
    assign inst = ROM[address[7:0]];
endmodule
```

Figure 9: Instruction Memory Code

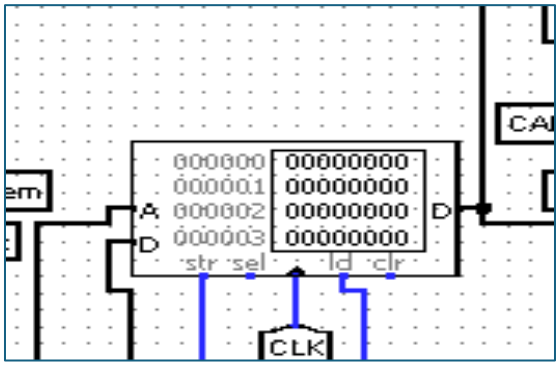


Figure 10: Data Memory Module

Code of Data memory

```
// Memory Stage
// ------------------------------------------------------------------------
// This module implements the Memory (MEM) stage.
// It performs data memory read and write operations.
// ------------------------------------------------------------------------

`timescale 1ns/1ps

module memory_stage (
    input clk,

    input  MEM_Read,
    input  MEM_Write,
    input [31:0] ALU_result,
    input [31:0] Reg2,          // Data to be written (store)

    output reg [31:0] MEM_Data // Data read from memory
);

    // Data memory: 256 words, each 32 bits wide
    reg [31:0] DATA_MEM [0:255];

    // Synchronous write operation
    always @(posedge clk) begin
        if (MEM_Write) begin
            DATA_MEM[ALU_result[7:0]] <= Reg2;
        end
    end

    // Combinational read operation
    always @(*) begin
        if (MEM_Read)
            MEM_Data = DATA_MEM[ALU_result[7:0]];
        else
            MEM_Data = 32'b0;
    end
endmodule
```

Figure 11: Data Memory Code

Register File

The register file serves as a fundamental element of the CPU, facilitating rapid data storage and immediate register accessibility for a range of tasks. Its optimized architecture and strategic placement near execution units are vital for enhancing the total performance and operational capability of the computer.

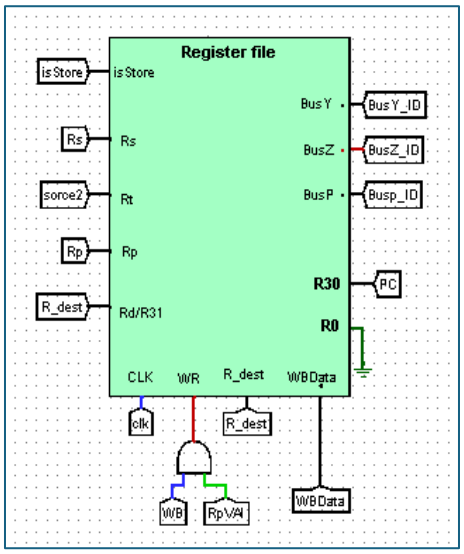


Figure 12: Register File Module

```
1   //  Register File ------------------------------------------
2       `timescale 1ns/1ps
3   module reg_file (
4       input         clk,
5       input  [31:0] pc_value,
6       input  [4:0]  Rs,
7       input  [4:0]  Rt,
8       input  [4:0]  Rd,
9       input         isStore,
10      input  [4:0]  Rp,
11      input  [4:0]  R_dest,
12      input  [31:0] WBData,
13      input         WR,
14      output [31:0] BusY,
15      output [31:0] BusZ,
16      output [31:0] Busp
17  );
18      reg [31:0] R [0:31];
19      wire [4:0] read2 = (isStore) ? Rd : Rt;
20      always @(posedge clk) begin
21          // hardwire R0=0
22          R[0]  <= 32'b0;
23          // hardwire R30 = PC
24          R[30] <= pc_value;
25          // normal write (but not to R0 or R30)
26          if (WR && (R_dest != 5'd0) && (R_dest != 5'd30))
27              R[R_dest] <= WBData;
28      end
29      // combinational reads
30      assign BusY = (Rs == 5'd0) ? 32'b0 : R[Rs];
31      assign BusZ = (read2 == 5'd0) ? 32'b0 : R[read2];
32      assign Busp = (Rp == 5'd0) ? 32'b0 : R[Rp];
33  endmodule
```

Figure 13: Register File Code

ALU

The Arithmetic Logic Unit (ALU) The ALU is a specialized digital circuit inside the processor responsible for performing logical and mathematical calculations on binary information. It functions by processing two input operands based on a specific command to generate an output. Its capabilities span a broad spectrum, including standard math (addition, subtraction, multiplication, and division), bitwise logic (AND, OR, XOR), and relational comparisons like identifying greater or equal values.

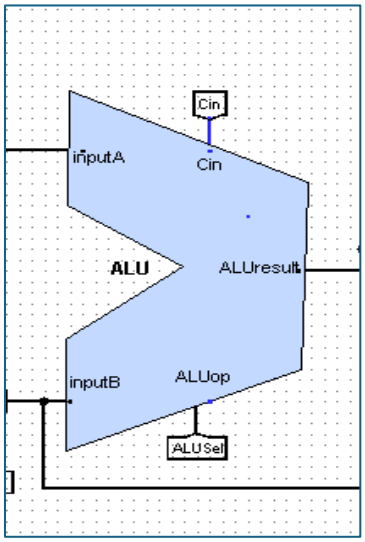


Figure 14: ALU Block

```
// ALU -----------------------------------------------------
module ALU_module (
    input  [31:0] a,
    input  [31:0] b,
    input  [2:0]  op,
    output reg [31:0] out
);
    always @(*) begin
        case (op)
            3'b000: out = ~(a | b);   // NOR / NORI
            3'b001: out =  a & b;     // AND / ANDI
            3'b010: out =  a | b;     // OR  / ORI
            3'b011: out =  a - b;     // SUB
            3'b100: out =  a + b;     // ADD / ADDI
            default: out = 32'b0;
        endcase
    end
endmodule
```

Figure 15: ALU Code

The **ALU Module** serves as the computational core of the processor, responsible for executing a variety of arithmetic and logical operations on 32-bit data. It receives two 32-bit operands (**a** and **b**) and a 3-bit operation code (**op**) that determines the specific function to be performed. Based on the value of the opcode, the unit can perform logical **NOR** (3'b000), bitwise **AND** (3'b001), bitwise **OR** (3'b010), as well as arithmetic **subtraction** (3'b011) and **addition** (3'b100). This module is designed to handle both register-to-register operations and immediate-based instructions (like **ADDI** or **ORI**), providing a single, 32-bit result that is then passed through the pipeline stages for further processing or write-back.

Kill and Stall Detection Unit

Stall Unit is a critical component in pipelined processor architecture designed to manage data hazards and ensure synchronized execution. As shown in the diagram, this unit monitors various control signals, such as ForwardA, ForwardB, and EX.MEMRead, to detect situations where a necessary data value is not yet available for the next instruction. When a hazard is identified—particularly when a previous instruction (like a memory read) has not finished retrieving data needed by a subsequent operation—the unit generates a Stall signal. This signal effectively pauses the earlier stages of the pipeline, preventing the processor from executing incorrect data and maintaining the overall integrity of the computer system's operations.

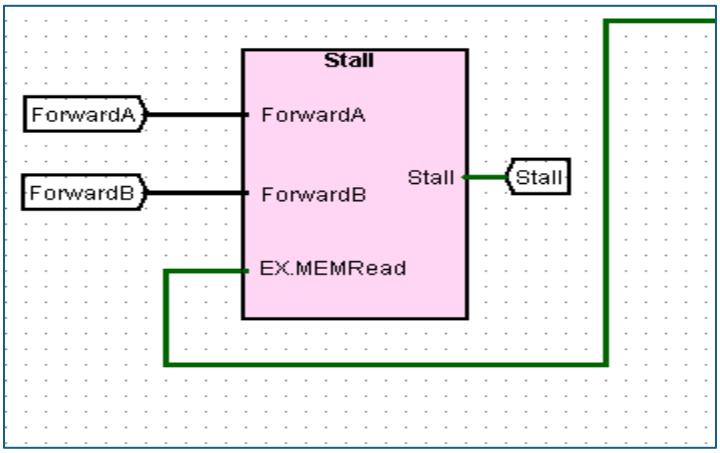


Figure 16: Stall Block

Multiplexers

4x1 Multiplexers in the Processor Design

The implementation of this multicycle processor utilizes two 4x1 multiplexers to manage critical data paths and control signals. These components are essential for selecting the correct input from multiple sources based on specific control lines.

- **A 4x1 Mux**

is employed to establish the subsequent value of the Program Counter. This selection is driven by the **PC source signal**, which acts as the control line. As illustrated in the hardware schematic, the multiplexer evaluates inputs such as **pc+1**, **jump** (calculated via sign extension and addition), **Call**, and **BusZ** to determine the next instruction address.

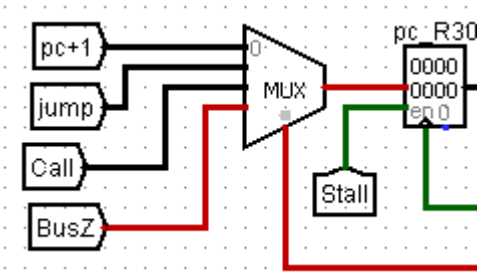


Figure 17: (PC) Source Selection Logic

- **A 2x1 Mux**

This component is a 2x1 Multiplexer designed to handle the sign extension of immediate values. It selects between two different methods of transforming a 12-bit immediate value (Imm12) into a full 32-bit format.

The Mux receives two processed versions of the same 12-bit input:

- **Sign Extension:** The first input is the **Imm12** value passed through a "sign extend" block, which preserves the mathematical sign of the number.

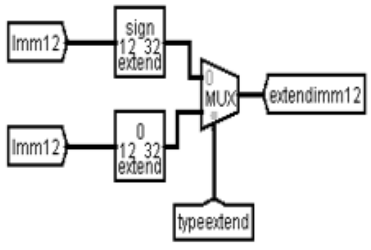


Figure 18: Immediate Extension .

- **Zero Extension:** The second input is the **Imm12** value passed through a "0 extend" block, which simply fills the remaining bits with zeros.

Pc control unit

We implemented PC control to decide what is the next pc (the next instruction to execute) the following figure shows the data path of the PC control unit :

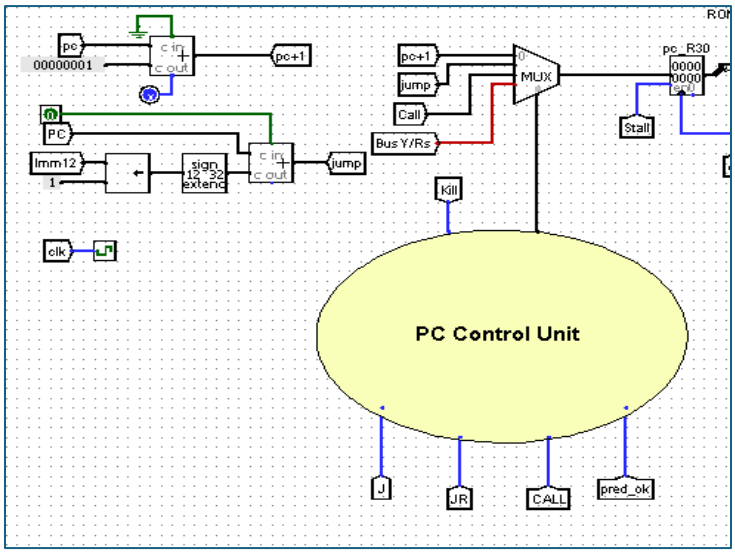


Figure 19: PC control Module

- Code

```
// PC Control Unit
// This module determines the next PC source for the fetch stage based on jump,
// jump-register, and call instructions. It also generates a Kill signal to
// invalidate the instruction in the pipeline if a branch or jump is taken.
`timescale 1ns/1ps
module pc_control_unit (
    input  J,
    input  JR,
    input  CALL,
    input  pred_ok,// Predicate/condition evaluation (true if instruction should execute)
    output reg [1:0] pcSrc,  // Select next PC source: 00=PC+1, 01=PC+offset, 10=JR
    output reg  Kill
);
    always @(*) begin
        // Default: increment PC by 1, no instruction kill
        pcSrc = 2'b00; // Next PC = PC + 1
        Kill  = 1'b0;

        if (pred_ok) begin
            if (JR) begin
                // Jump to address in register
                pcSrc = 2'b10; // Select JR as next PC
                Kill  = 1'b1;  // Flush instruction in pipeline (invalidate)
            end
            else if (J || CALL) begin
                // Conditional jump or call instruction
                pcSrc = 2'b01; // Select PC + offset as next PC
                Kill  = 1'b1;  // Flush instruction in pipeline (invalidate)
            end
        end
    end
endmodule
```

Figure 20: PC control Code

The **PC Control Unit** module coordinates instruction flow by managing the **pcSrc** and **Kill** signals based on the instruction type and the **pred_ok** condition. By default, the unit selects the next sequential instruction (**PC + 1**) and keeps the pipeline active. However, when the predicate condition is met, the unit evaluates if the instruction is a **Jump Register (JR)**, which sets **pcSrc** to 2'b10, or a **Jump/Call**, which sets **pcSrc** to 2'b01. In both cases, the **Kill** signal is asserted to flush the pipeline, ensuring that instructions from the incorrect path are discarded after a successful control-flow change.

Forwarding Control Unit

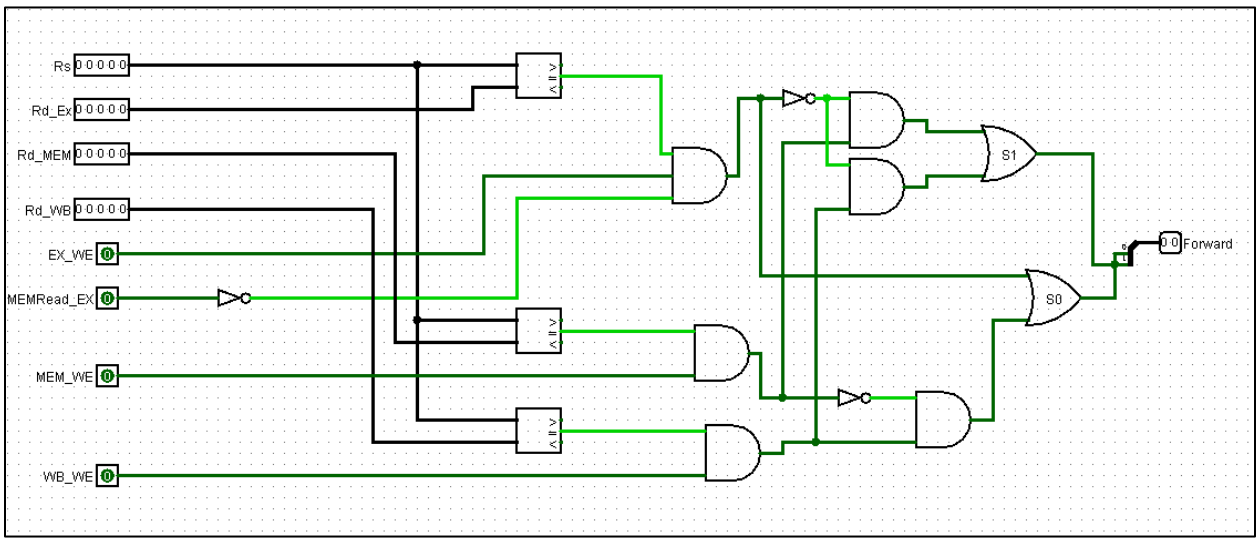


Figure 21: Forwarding CU

The forwarding unit detects data hazards by comparing source and destination registers across pipeline stages and selects the appropriate forwarded value based on instruction priority

In Boolean:

Match_EX = (R_source_Decode == Rd_EX) && WE_EX

Match_MEM = (R_source_Decode == Rd_MEM) && WE_MEM

Match_WB = (R_source_Decode == Rd_WB) && WE_WB

Forward[1] = Match_MEM || Match_WB

Forward[0] = Match_EX || Match_WB

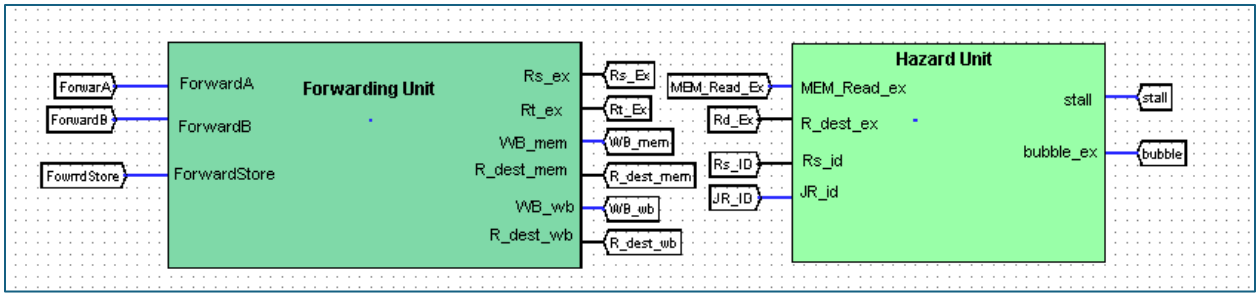


Figure 22: Forwarding CU Block

```
// Forwarding Unit
// This module detects data hazards in the pipeline and selects the correct
// forwarding paths to resolve them without stalling.
// Forwarding is applied from MEM and WB stages to EX stage operands and store data.
`timescale 1ns/1ps
module forward_unit (
    input  [4:0] Rs_ex,
    input  [4:0] Rt_ex,
    input        WB_mem,
    input  [4:0]  R_dest_mem,
    input        WB_wb,
    input  [4:0]  R_dest_wb,
    output reg [1:0] ForwardA,      // Forwarding control for ALU input A
    output reg [1:0] ForwardB,      // Forwarding control for ALU input B
    output reg [1:0] ForwardStore   // Forwarding control for store operand
);
    always @(*) begin
        // Default: no forwarding
        ForwardA     = 2'b00;
        ForwardB     = 2'b00;
        ForwardStore = 2'b00;
        // Forward from MEM stage to EX stage
        if (WB_mem && (R_dest_mem != 5'd0) && (R_dest_mem == Rs_ex))
            ForwardA = 2'b10;
        if (WB_mem && (R_dest_mem != 5'd0) && (R_dest_mem == Rt_ex))
            ForwardB = 2'b10;
        // Forward from WB stage to EX stage (only if MEM stage is not forwarding)
        if (WB_wb && (R_dest_wb != 5'd0) &&
            !(WB_mem && (R_dest_mem != 5'd0) && (R_dest_mem == Rs_ex)) &&
            (R_dest_wb == Rs_ex))
            ForwardA = 2'b01;

        if (WB_wb && (R_dest_wb != 5'd0) &&
            !(WB_mem && (R_dest_mem != 5'd0) && (R_dest_mem == Rt_ex)) &&
            (R_dest_wb == Rt_ex))
            ForwardB = 2'b01;

        // Forward for store instructions
        if (WB_mem && (R_dest_mem != 5'd0) && (R_dest_mem == Rt_ex))
            ForwardStore = 2'b10;
        else if (WB_wb && (R_dest_wb != 5'd0) && (R_dest_wb == Rt_ex))
            ForwardStore = 2'b01;
    end
endmodule
```

Figure 23: Forwarding CU Code

The Forward Unit is a critical hardware module designed to resolve data hazards in the pipelined processor by implementing a forwarding (or bypassing) mechanism. It continuously monitors the destination registers from the Memory (R_dest_mem) and Write-Back (R_dest_wb) stages to check for dependencies with the source registers in the Execution stage (Rs_ex and Rt_ex). When a match is detected and the corresponding Write-Back signal (WB_mem or WB_wb) is active, the unit generates selection signals like ForwardA and ForwardB to redirect the most recent data directly to the ALU inputs. This allows the processor to use results from previous instructions before they are officially written back to the register file, effectively eliminating unnecessary stalls and maintaining

IMM_EXTEND

```
1   // Immediate Extend
2   // ------------------------------------------------------------------------------
3   // This module extends a 12-bit immediate to 32 bits.
4   // It supports both sign-extension and zero-extension based on 'typextend'.
5   // ------------------------------------------------------------------------------
6
7   `timescale 1ns/1ps
8
9   module imm_extend (
10      input  [11:0] imm12,      // 12-bit immediate from instruction
11      input         typextend, // 1 = sign-extend, 0 = zero-extend
12      output [31:0] imm_ext    // 32-bit extended immediate
13  );
14
15      // Extend immediate to 32 bits
16      assign imm_ext = (typextend) ? {{20{imm12[11]}}, imm12}  // Sign-extension
17                                   : {20'b0, imm12};           // Zero-extension
18
19  endmodule
```

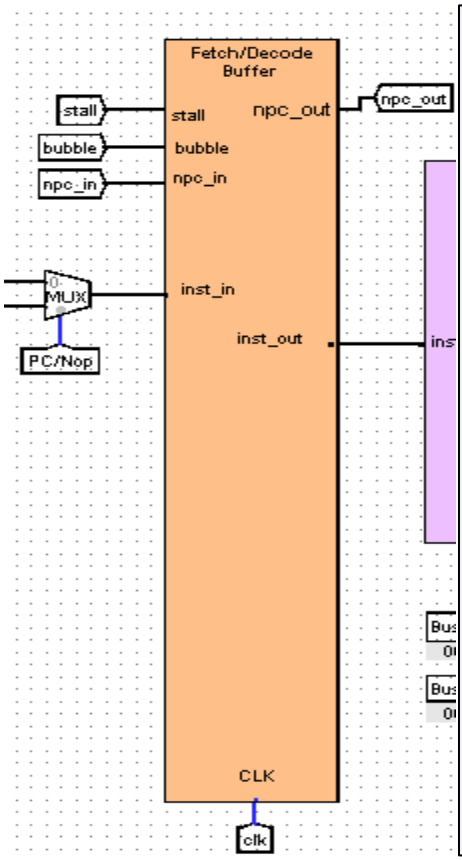
Figure 24: Imm Extend Code

The Immediate Extend (imm_extend) module is a critical component in the decode stage that converts 12-bit immediate values into a full 32-bit format required for ALU operations. It operates based on a typeextend control signal: if the signal is active, the module performs sign-extension by replicating the most significant bit (bit 11) across the upper 20 bits. If the signal is inactive, it performs zero-extension by filling the upper 20 bits with zeros. This flexibility allows the processor to correctly handle both signed arithmetic and unsigned logical operations.

Buffers

Pipeline buffers synchronize data transfer between processor stages on each clock cycle. They isolate stages to prevent data overlap and manage hazards by implementing Stalls to hold data or Bubbles to flush instructions.

Fetch/Decode Buffer




```
1  // IF/ID Pipeline Buffer
2  // ------------------------------------------------------------------------
3  // This module represents the pipeline register between the Instruction Fetch
4  // (IF) stage and the Instruction Decode (ID) stage.
5  // It handles normal operation, stalls, and bubbles (flushes).
6  // ------------------------------------------------------------------------
7  `timescale 1ns/1ps
8
9  module fetch_decode_buffer (
10     input clk,
11     input reset,
12     input stall,     // Stall signal from Hazard Detection Unit
13     input bubble,    // Bubble/Flush signal from PC Control Unit
14     input [31:0] inst_in,   // Instruction from IF stage
15     input [31:0] npc_in,    // Next PC value (PC + 1)
16
17     output reg [31:0] inst_out, // Instruction to ID stage
18     output reg [31:0] npc_out   // NPC forwarded to ID stage
19 );
20
21     // Pipeline register update on clock edge
22     always @(posedge clk or posedge reset) begin
23         if (reset) begin
24             // On reset, insert NOP and clear NPC
25             inst_out <= 32'b0;
26             npc_out  <= 32'b0;
27         end
28         else if (stall) begin
29             // On stall, hold current values (freeze pipeline)
30             inst_out <= inst_out;
31             npc_out  <= npc_out;
32         end
33         else if (bubble) begin
34             // On bubble, flush instruction by inserting NOP
35             inst_out <= 32'b0;
36             npc_out  <= npc_in;
37         end
38         else begin
39             // Normal pipeline operation
40             inst_out <= inst_in;
41             npc_out  <= npc_in;
42         end
43     end
44 endmodule
```

Figure 25: Fetch/Decode Buffer Block **Figure 26: Fetch/Decode Buffer Code**

The Fetch/Decode Buffer acts as a critical synchronization point between the instruction fetch and decode stages, ensuring the stable transfer of the instruction (INST) and the next program counter (NPC). Beyond simple data storage, this buffer plays a vital role in Hazard Management through its control inputs. When a Stall is detected, the buffer freezes its current state to prevent data loss while the pipeline waits. Conversely, when a Bubble (or Kill) signal is received from the control logic, the buffer clears the current instruction and injects a NOP (No Operation) into the pipeline, effectively flushing the stage to handle control hazards like jumps or calls.

Decode/EXC Buffer

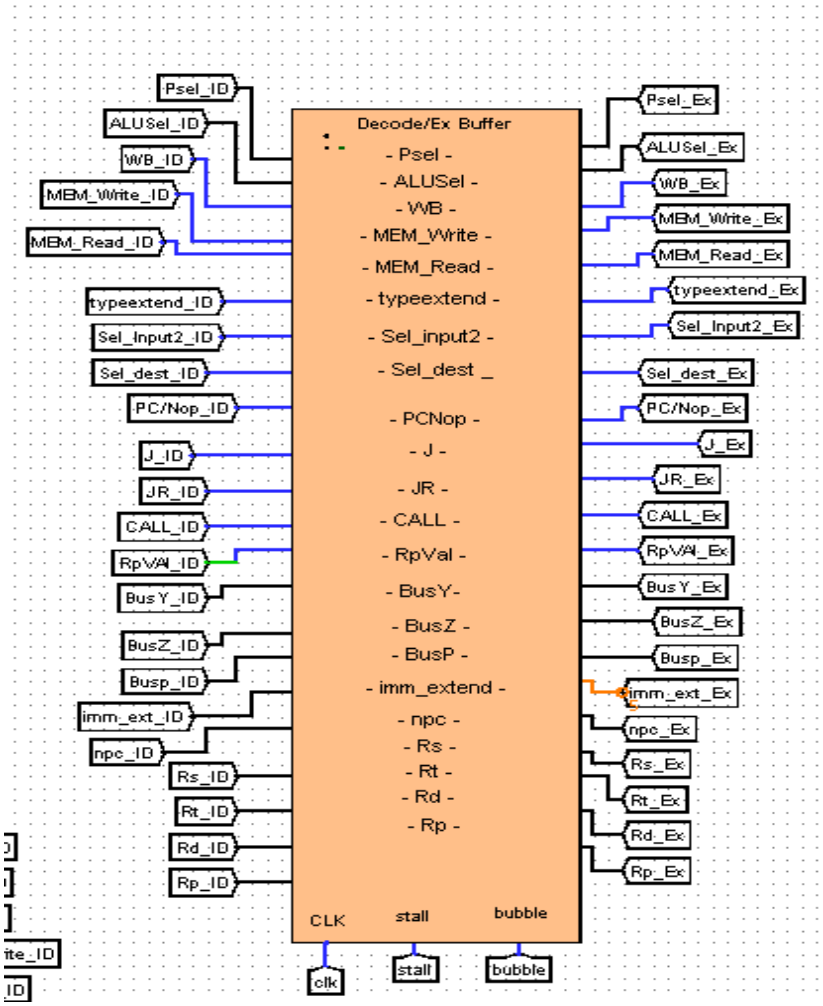


Figure 27: Decode/EXC Buffer block

- **CODE**

```
// Decode Stage
// This module implements the Instruction Decode (ID) stage.
// It extracts instruction fields, generates control signals,
// evaluates the predicate condition, and reads operands from the register file.
`timescale 1ns/1ps

module decode_stage (
    input         clk,
    input  [31:0] pc_value,      // Current PC value (used by register file)
    input  [31:0] inst_ID,       // Instruction from IF/ID buffer
    input  [4:0]  R_dest_WB,
    input  [31:0] WBData,
    input         WR_WB,
    output [1:0]  Psel,
    output [2:0]  AluSel,
    output        WB,            // Write-back control
    output        MEM_Write,     // Memory write control
    output        MEM_Read,      // Memory read control
    output        typextend,     // Immediate extension control
    output        Sel_input2,
    output        Sel_Dest,
    output        PCNop,         // PC no-operation control
    output        J,
    output        JR,
    output        CALL,
    output        WR_final,      // Final write enable after predicate check
    output [31:0] BusY,          // Source operand 1
    output [31:0] BusZ,          // Source operand 2
    output [31:0] Busp,
    output [4:0]  Rs, Rt, Rd, Rp,
    output [11:0] imm12,
    output [4:0]  opcode,
    output        RpVal          // Predicate evaluation result
);
    wire [2:0] Aluop;
    wire WR_raw;
    // Instruction field extraction
    assign opcode = inst_ID[31:27];
    assign Rp     = inst_ID[26:22];
    assign Rd     = inst_ID[21:17];
    assign Rs     = inst_ID[16:12];
    assign Rt     = inst_ID[11:7];
    assign imm12  = inst_ID[11:0];
```

```
    // Control unit: generates control signals from opcode
    control_unit CU (
        .opcode(opcode),
        .Psel(Psel),
        .Aluop(Aluop),
        .WB(WB),
        .MEM_Write(MEM_Write),
        .MEM_Read(MEM_Read),
        .typextend(typextend),
        .Sel_input2(Sel_input2),
        .WR(WR_raw),
        .Sel_Dest(Sel_Dest),
        .PCNop(PCNop),
        .J(J),
        .JR(JR),
        .CALL(CALL)
    );
    // ALU control forwarding
    assign AluSel = Aluop;
    // Predicate evaluation:
    // Instruction executes if Rp == R0 OR Reg[Rp] != 0
    assign RpVal = (Rp == 5'd0) || (Busp != 32'b0);
    // Final write control considering predicate result
    assign WR_final = WB & RpVal;
    // Register file:
    // Reads source registers, predicate register, and supports write-back
    reg_file RF (
        .clk(clk),
        .pc_value(pc_value),
        .Rs(Rs),
        .Rt(Rt),
        .Rd(Rd),
        .isStore(MEM_Write),
        .Rp(Rp),
        .R_dest(R_dest_WB),
        .WBData(WBData),
        .WR(WR_WB),
        .BusY(BusY),
        .BusZ(BusZ),
        .Busp(Busp)
    );
endmodule
```

Figure 28: Decode/EXC Buffer Code

The Decode/Execute (ID/EX) Buffer serves as a vital pipeline register that carries all decoded control signals and register data from the decode stage to the execution unit. This buffer stores a wide range of signals, including the ALU operation (AluSel), memory read/write enables, and immediate values, ensuring they are synchronized with the correct instruction as it moves through the pipeline. It also plays a key role in Hazard Management; when a stall is triggered, the buffer maintains its current values to pause the instruction, and when a bubble is detected, it clears the control signals (resetting them to zero) to effectively insert a NOP and prevent any unintended write-back or memory operations.

EXC/MEM Buffer

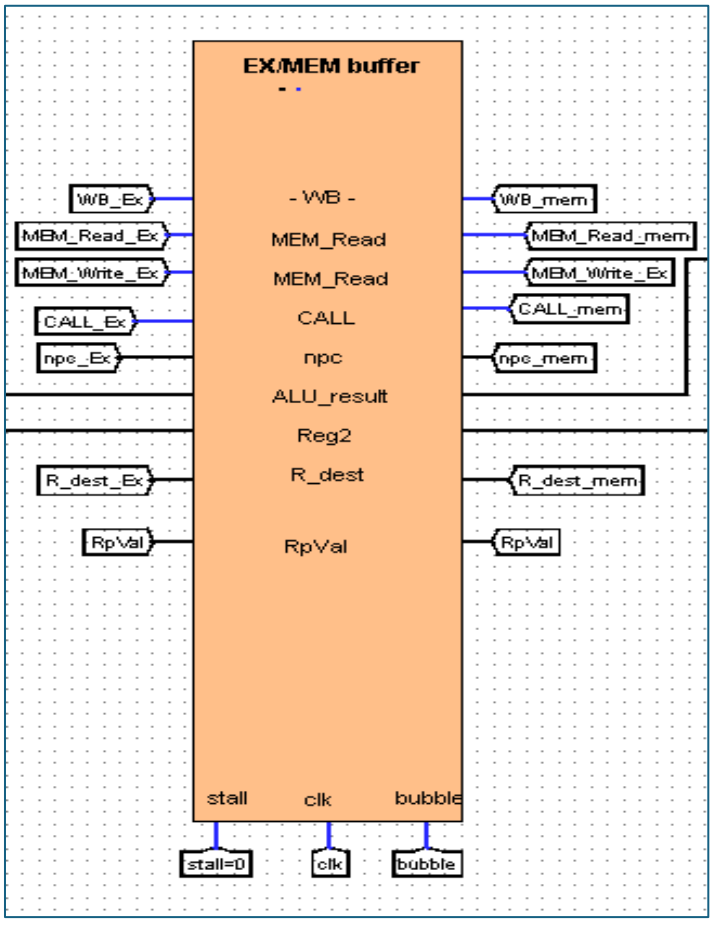


Figure 29:EXC/ MEM Buffer Block

```
// EX/MEM Pipeline Buffer
// This module represents the pipeline register between the Execute (EX)
// stage and the Memory (MEM) stage.
// It supports stall and bubble (flush) control.
`timescale 1ns/1ps
module execute_memory_buffer (
    input clk,stall,bubble,WB_in, // Write-back control from EX stage
    input MEM_Read_in,
    input MEM_Write_in,
    input CALL_in,          // Call instruction control
    input [31:0] npc_in,
    input [31:0] ALU_result_in, // ALU result from EX stage
    input [31:0] Reg2_in,       // Store data
    input [4:0]  R_dest_in,
    output reg WB,
    output reg MEM_Read,
    output reg MEM_Write,
    output reg CALL,
    output reg [31:0] npc,  // NPC forwarded
    output reg [31:0] ALU_result, // ALU result to MEM stage
    output reg [31:0] Reg2,       // Store data to MEM stage
    output reg [4:0]  R_dest      // Destination register
);   // Pipeline register update
    always @(posedge clk) begin
        if (stall) begin
            // On stall: hold current values (no update)
        end
        else if (bubble) begin
            // On bubble: flush control and data signals
            WB         <= 1'b0;
            MEM_Read   <= 1'b0;
            MEM_Write  <= 1'b0;
            CALL       <= 1'b0;
            npc        <= 32'b0;
            ALU_result <= 32'b0;
            Reg2       <= 32'b0;
            R_dest     <= 5'b0;
        end
        else begin
            // Normal pipeline transfer
            WB         <= WB_in;
            MEM_Read   <= MEM_Read_in;
            MEM_Write  <= MEM_Write_in;
            CALL       <= CALL_in;
            npc        <= npc_in;
            ALU_result <= ALU_result_in;
            Reg2       <= Reg2_in;
            R_dest     <= R_dest_in;
        end
    end
endmodule
```

Figure 30: EXC/ MEM Buffer Code

MEM/WB Buffer

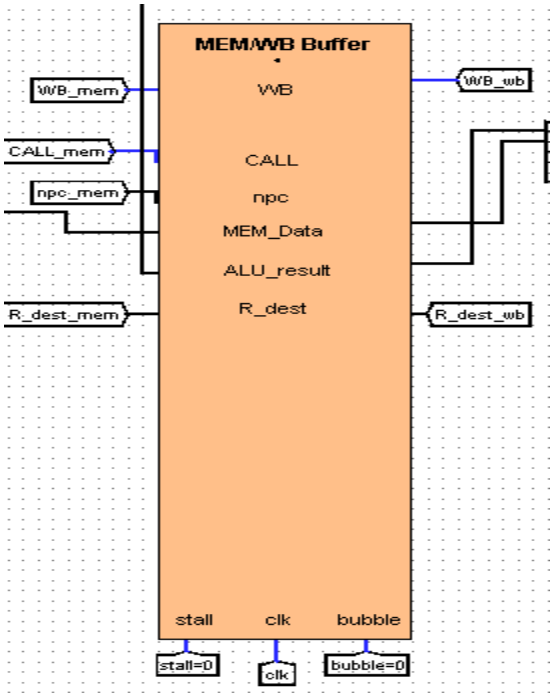


Figure 31: MEM/WB Buffer Block

```
1   // MEM/WB Pipeline Buffer
2   // ---------------------------------------------------------------------------------
3   // This module represents the pipeline register between the Memory (MEM)
4   // stage and the Write-Back (WB) stage.
5   // It forwards control and data signals and supports stall and bubble control.
6   `timescale 1ns/1ps
7   module memory_writeback_buffer (
8       input clk, stall,bubble,WB_in,MEM_Read_in,CALL_in,
9       input [31:0] npc_in,
10      input [31:0] MEM_Data_in,
11      input [31:0] ALU_result_in,
12      input [4:0]  R_dest_in,
13      output reg         WB,          // Write-back enable to WB stage
14      output reg         MEM_Read,
15      output reg         CALL,
16      output reg [31:0] npc,          // NPC forwarded
17      output reg [31:0] MEM_Data,
18      output reg [31:0] ALU_result,
19      output reg [4:0]  R_dest        // Destination register to WB stage
20  );
21      // Pipeline register update
22      always @(posedge clk) begin
23          if (stall) begin
24              // On stall: hold current values
25          end
26          else if (bubble) begin
27              // On bubble: flush control and data signals
28              WB         <= 1'b0;
29              MEM_Read   <= 1'b0;
30              CALL       <= 1'b0;
31              npc        <= 32'b0;
32              MEM_Data   <= 32'b0;
33              ALU_result <= 32'b0;
34              R_dest     <= 5'b0;
35          end
36          else begin
37              // Normal pipeline transfer
38              WB         <= WB_in;
39              MEM_Read   <= MEM_Read_in;
40              CALL       <= CALL_in;
41              npc        <= npc_in;
42              MEM_Data   <= MEM_Data_in;
43              ALU_result <= ALU_result_in;
44              R_dest     <= R_dest_in;
45          end
46      end
47  endmodule
```

Figure 32: MEM/WB Buffer Code

Stages

Decode Stage

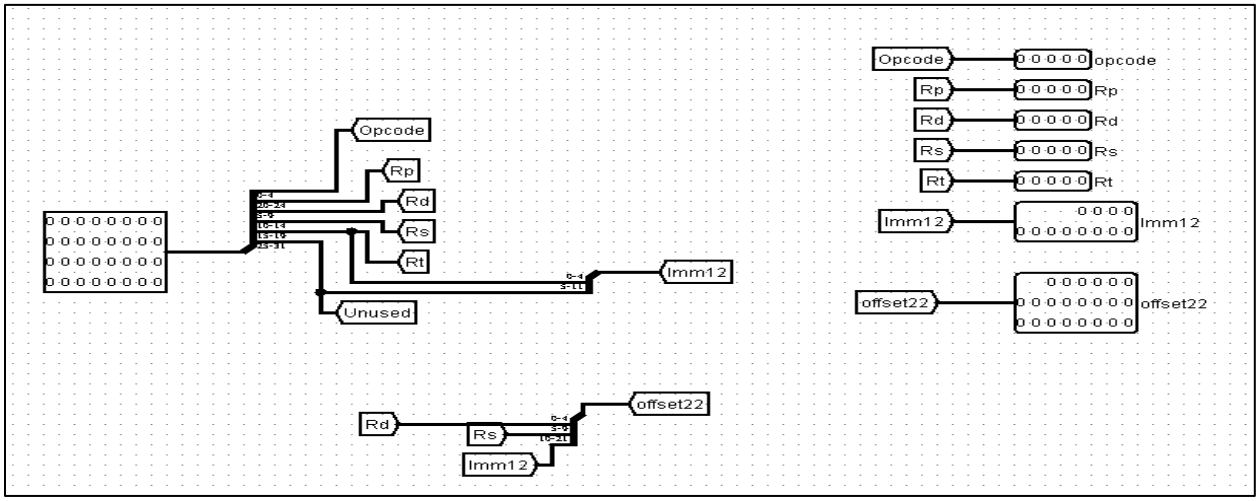


Figure 33: Decode Stage

CODE

```
    control_unit CU (
         .opcode(opcode),
         .Psel(Psel),
         .Aluop(Aluop),
         .WB(WB),
         .MEM_Write(MEM_Write),
         .MEM_Read(MEM_Read),
         .typextend(typextend),
         .Sel_input2(Sel_input2),
         .WR(WR_raw),
         .Sel_source2(Sel_source2),
         .Sel_Dest(Sel_Dest),
         .PCNop(PCNop),
         .J(J),
         .JR(JR),
         .CALL(CALL)
    );

    assign AluSel = Aluop;

    // predicate check: execute if Rp == R0 OR Rp register value != 0
    assign RpVal = (Rp == 5'd0) || (Busp != 32'b0);

    // "execute enable" for this instruction (you pipeline this!)
    assign WR_final = WB & RpVal;

    // ===== Register file write must come from WB stage enable =====
    reg_file RF (
         .clk(clk),
         .Rs(Rs),
         .Rt(Rt),
         .Rp(Rp),
         .R_dest(R_dest_WB),
         .WBData(WBData),
         .WR(WR_WB),
         .BusY(BusY),
         .BusZ(BusZ),
         .Busp(Busp)
    );
endmodule
```

```
    wire [2:0] Aluop;
    wire WR_raw;
    // Instruction field extraction
    assign opcode = inst_ID[31:27];
    assign Rp     = inst_ID[26:22];
    assign Rd     = inst_ID[21:17];
    assign Rs     = inst_ID[16:12];
    assign Rt     = inst_ID[11:7];
    assign imm12  = inst_ID[11:0];
    // Control unit: generates control signals from opcode
    control_unit CU (
        .opcode(opcode),
        .Psel(Psel),
        .Aluop(Aluop),
        .WB(WB),
        .MEM_Write(MEM_Write),
        .MEM_Read(MEM_Read),
        .typextend(typextend),
        .Sel_input2(Sel_input2),
        .WR(WR_raw),
        .Sel_Dest(Sel_Dest),
        .PCNop(PCNop),
        .J(J),
        .JR(JR),
        .CALL(CALL)
    );
    // ALU control forwarding
    assign AluSel = Aluop;
    // Predicate evaluation:
    // Instruction executes if Rp == R0 OR Reg[Rp] != 0
    assign RpVal = (Rp == 5'd0) || (Busp != 32'b0);
    // Final write control considering predicate result
    assign WR_final = WB & RpVal;
    // Register file:
    // Reads source registers, predicate register, and supports write-back
    reg_file RF (
        .clk(clk),
        .pc_value(pc_value),
        .Rs(Rs),
        .Rt(Rt),
        .Rd(Rd),
        .isStore(MEM_Write),
        .Rp(Rp),
        .R_dest(R_dest_WB),
        .WBData(WBData),
        .WR(WR_WB),
        .BusY(BusY),
        .BusZ(BusZ),
        .Busp(Busp)
    );
```

Figure 34: Decode Stage Code

The Decode Stage functions as the primary instruction processing hub, where the 32-bit machine code is disassembled into its constituent fields, such as the opcode, source registers (Rs, Rt, Rp), and destination register (Rd). During this phase, the Control Unit translates the opcode into a set of functional signals—including AluSel for computation and WB for register updates—while the immediate values and offsets are extracted for use in branching or memory calculations. These extracted values and control signals are then packaged and passed to the next pipeline stage through the ID/EX Buffer, ensuring that the execution unit receives all the necessary parameters to perform the operation correctly.

Execute Stage

```
1  // Execute Stage
2  // ------------------------------------------------------------------------------
3  // This module implements the Execute (EX) stage.
4  // It selects the ALU operands, performs the ALU operation,
5  // and determines the destination register.
6  // ------------------------------------------------------------------------------
7
8  `timescale 1ns/1ps
9
10 module execute_stage (
11     input [31:0] BusY,      // First ALU operand from register file
12     input [31:0] BusZ,      // Second register operand
13     input [31:0] imm_ext,
14
15     input        Sel_input2,
16     input [2:0]  AluSel,
17
18     input [4:0]  Rd,
19     input [4:0]  Rt,
20     input        Sel_Dest,
21
22     output [31:0] ALU_result,
23     output [31:0] Reg2_out,   // Forwarded second operand (for store)
24     output [4:0]  R_dest_exec // Selected destination register
25 );
26
27     // Select second ALU input: register or immediate
28     wire [31:0] ALU_in2 = (Sel_input2) ? imm_ext : BusZ;
29
30     // ALU operation
31     ALU_module ALU (
32         .a(BusY),
33         .b(ALU_in2),
34         .op(AluSel),
35         .out(ALU_result)
36     );
37
38     // Forward register value for store instructions
39     assign Reg2_out = BusZ;
40
41     // Select destination register
42     assign R_dest_exec = (Sel_Dest) ? Rt : Rd;
43
44 endmodule
45
```

Figure 35: Execute Stage

The Execute Stage is the core computational phase of the pipeline where arithmetic and logical operations are physically performed on the instruction operands. It utilizes an ALU (Arithmetic Logic Unit) that takes two primary inputs: ALU_in1, which is typically connected to the first register bus (BusY), and ALU_in2, which is dynamically selected via a multiplexer controlled by Sel_input2 to be either the second register bus (BusZ) or a sign-extended immediate value (imm_ext). The specific operation performed (such as addition, subtraction, or logical bitwise functions) is determined by the 3-bit AluSel signal. Additionally, this stage resolves the final destination register (R_dest_exec) by choosing between the Rt and Rd fields based on the Sel_Dest signal, ensuring that the computed results and data are correctly routed for memory access or register write-back.

Memory Stage

```
1  // Memory Stage
2  // ------------------------------------------------------------------
3  // This module implements the Memory (MEM) stage.
4  // It performs data memory read and write operations.
5  // ------------------------------------------------------------------
6
7  `timescale 1ns/1ps
8
9  module memory_stage (
10     input clk,
11
12     input  MEM_Read,
13     input  MEM_Write,
14     input [31:0] ALU_result,
15     input [31:0] Reg2,          // Data to be written (store)
16
17     output reg [31:0] MEM_Data // Data read from memory
18 );
19
20     // Data memory: 256 words, each 32 bits wide
21     reg [31:0] DATA_MEM [0:255];
22
23     // Synchronous write operation
24     always @(posedge clk) begin
25         if (MEM_Write) begin
26             DATA_MEM[ALU_result[7:0]] <= Reg2;
27         end
28     end
29
30     // Combinational read operation
31     always @(*) begin
32         if (MEM_Read)
33             MEM_Data = DATA_MEM[ALU_result[7:0]];
34         else
35             MEM_Data = 32'b0;
36     end
37 endmodule
38
```

Figure 36: Memory Stage Code

The Memory Stage is the fourth phase of the pipeline, responsible for direct interaction with the Data Memory for both reading and writing operations. It operates based on control signals received from the control unit; when MEM_Write is enabled, the processor stores the data from the register (Reg2) into the memory address specified by the ALU_result. Conversely, when MEM_Read is active, data is retrieved from the calculated address and output as MEM_Data to be passed onto the final Write-Back stage. This design ensures a clear separation between instruction and data memory, which is a key characteristic of the Harvard architecture used in this processor.

WB Stage

```
1  // Write Back Stage
2  // ------------------------------------------------------------------------------
3  // This module implements the Write-Back (WB) stage.
4  // It selects the data to be written to the register file and
5  // forwards the write enable signal.
6  // ------------------------------------------------------------------------------
7
8  `timescale 1ns/1ps
9
10 module write_back_stage (
11     input         WBSel,        // Write-back select: 0 = ALU result, 1 = Memory data
12     input         WR,           // Register write enable
13     input [31:0] ALU_result,
14     input [31:0] MEM_Data,
15
16     output [31:0] WBData,       // Data written back to register file
17     output        WR_out        // Forwarded write enable
18 );
19
20     // Select write-back data source
21     assign WBData = (WBSel) ? MEM_Data : ALU_result;
22
23     // Forward write enable signal
24     assign WR_out = WR;
25
26 endmodule
```

Figure 37: WB Stage Code

The Write-Back (WB) Stage is the final phase of the pipeline, where the results of an instruction are officially committed to the Register File. A multiplexer, controlled by the WBSel signal, selects the appropriate data to be written back: it chooses the ALU_result (for arithmetic/logic instructions) or the MEM_Data (for load instructions). This selected value (WBData), along with the write-enable signal (WR_out), is sent back to the decode stage to update the destination register, effectively completing the instruction's lifecycle.

Hazard Management and Control Flow

In this pipelined processor, the **Hazard Control Unit** manages data and control hazards to ensure system integrity. A **Stall** is triggered based on the boolean condition:

Stall = (ForwardA == 1 || ForwardB == 1) && EX_MEMRead. This signal pauses the **Program Counter (PC)** and earlier pipeline stages when a load-use dependency is detected.

If **pc_sel = 0 the** next instruction is PC+1 . else If **pc_sel = 1** this case for jump instruction . And **when pc_sel = 2** this case for the jump register instruction .

Regarding control flow, a **Kill** operation occurs when a control-flow instruction **such as : (J, CALL, or JR)** is executed and its predicate condition is met. To mitigate this control hazard, a **2x1 Multiplexer** is used to flush the pipeline by inserting a **NOP (No Operation)** signal, effectively invalidating the fetched instructions that followed the jump. The next PC value is then selected via a **4x1 Multiplexer** using the **Psel** signal, choosing between sequential execution (**pc+1**), jump targets (**jump/Call**), or register addresses (**BusZ**).

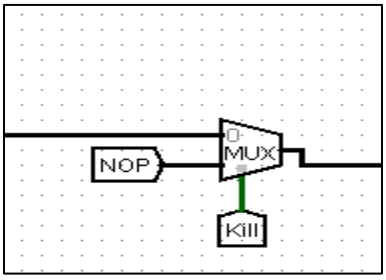


Figure 38: (Kill Signal Mux).

In Boolean: Kill = J || CALL || JR

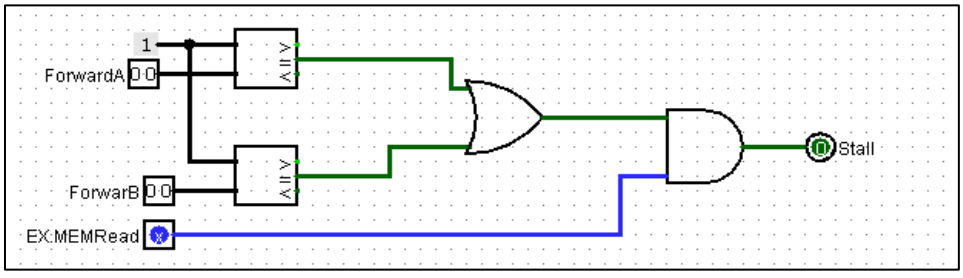


Figure 39: Hazard Control Logic

- CODE

```
// Hazard Detection Unit
// ------------------------------------------------------------------------------
// This module detects data hazards in the pipeline, particularly load-use hazards.
// It generates stall and bubble signals to prevent incorrect instruction execution.
// ------------------------------------------------------------------------------

`timescale 1ns/1ps

module hazard_unit (
    input        MEM_Read_ex,
    input  [4:0] R_dest_ex,
    input  [4:0] Rs_id,
    input  [4:0] Rt_id,
    input        JR_id,

    output reg   stall,       // Stall signal for pipeline
    output reg   bubble_ex    // Bubble signal for EX stage
);

    always @(*) begin
        // Default: no stall or bubble
        stall     = 1'b0;
        bubble_ex = 1'b0;

        // Load-use hazard: if EX stage reads memory and ID uses same register
        if (MEM_Read_ex && (R_dest_ex != 5'd0) &&
            ((R_dest_ex == Rs_id) || (R_dest_ex == Rt_id))) begin
            stall     = 1'b1;
            bubble_ex = 1'b1;
        end

        // Special case: JR instruction depends on EX load
        if (JR_id && MEM_Read_ex && (R_dest_ex != 5'd0) &&
            (R_dest_ex == Rs_id)) begin
            stall     = 1'b1;
            bubble_ex = 1'b1;
        end
    end
endmodule
```

Figure 40: Hazard Unit Code

The Hazard Unit is a critical protection module designed to detect and resolve data dependencies that could lead to incorrect execution. It specifically monitors for "load-use" hazards by checking if an instruction in the Execute (EX) stage is a memory load (MEM_Read_ex) and if its destination register matches the source registers (Rs_id or Rt_id) of the next instruction in the Decode (ID) stage. When such a conflict is detected, the unit asserts both stall and bubble_ex signals; the stall freezes the Program Counter and the current instruction, while the bubble_ex flushes the pipeline by inserting a NOP (No Operation). This temporary pause allows the required data to be retrieved from memory before the dependent instruction proceeds to the execution phase.

State Diagram

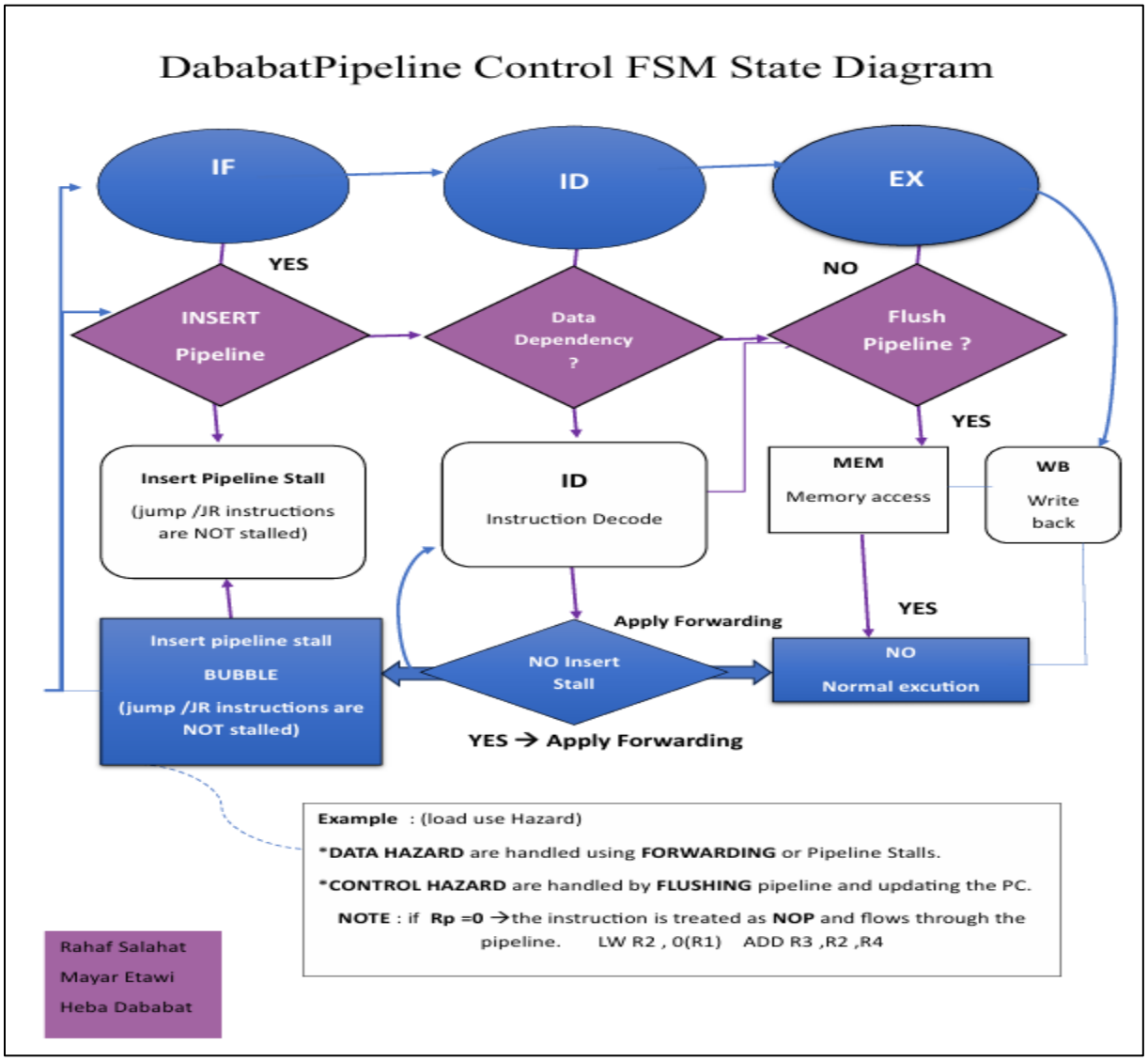


Figure 41: Pipeline Control State Diagram Analysis

This diagram illustrates the control logic required to manage instruction flow within a high-performance CPU pipeline. The primary goal of this finite state machine is to maximize instruction throughput while identifying and resolving "Hazards" that could lead to incorrect data processing or system instability. The control logic operates based on several key mechanisms:

- **Data Hazard Management:** Hazards are detected when there is a logical dependency between instructions (Data Dependency). The system resolves these by either using Forwarding (passing data directly to the next stage) or by Inserting a Pipeline Stall (Bubble) to delay the execution of the dependent instruction.

- **Control Hazard Handling:** These occur during jump or branch instructions. The controller manages this by Flushing the pipeline stages and updating the Program Counter (PC) to the correct target address.
- **Flow Stages**: The diagram tracks the instruction as it transitions through the standard five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write-Back (WB).

Illustrative Example: Load-Use Data Hazard

To better understand the control flow, consider the following instruction sequence:

1. **LW R2, 0(R1)** (Loads a value from memory into register R2)
2. **ADD R3, R2, R4 (**Adds the value in R2 to R4 and stores it in R3)

Analysis according to the Diagram:

- **Detection:** During the ID (Decode) stage of the ADD instruction, the controller detects a Data Dependency because the ADD operation requires the value in R2 which is still being processed by the LW instruction.
- **Conflict:** In this specific case (Load-Use), Forwarding alone cannot resolve the hazard because the data is not available until the end of the MEM stage.
- **Action:** Following the decision paths in the diagram, the controller will Insert a Pipeline Stall (Bubble). This stalls the ADD instruction for one clock cycle, allowing the LW instruction to complete its memory access, thereby ensuring that the ADD operation uses the correct, updated data.

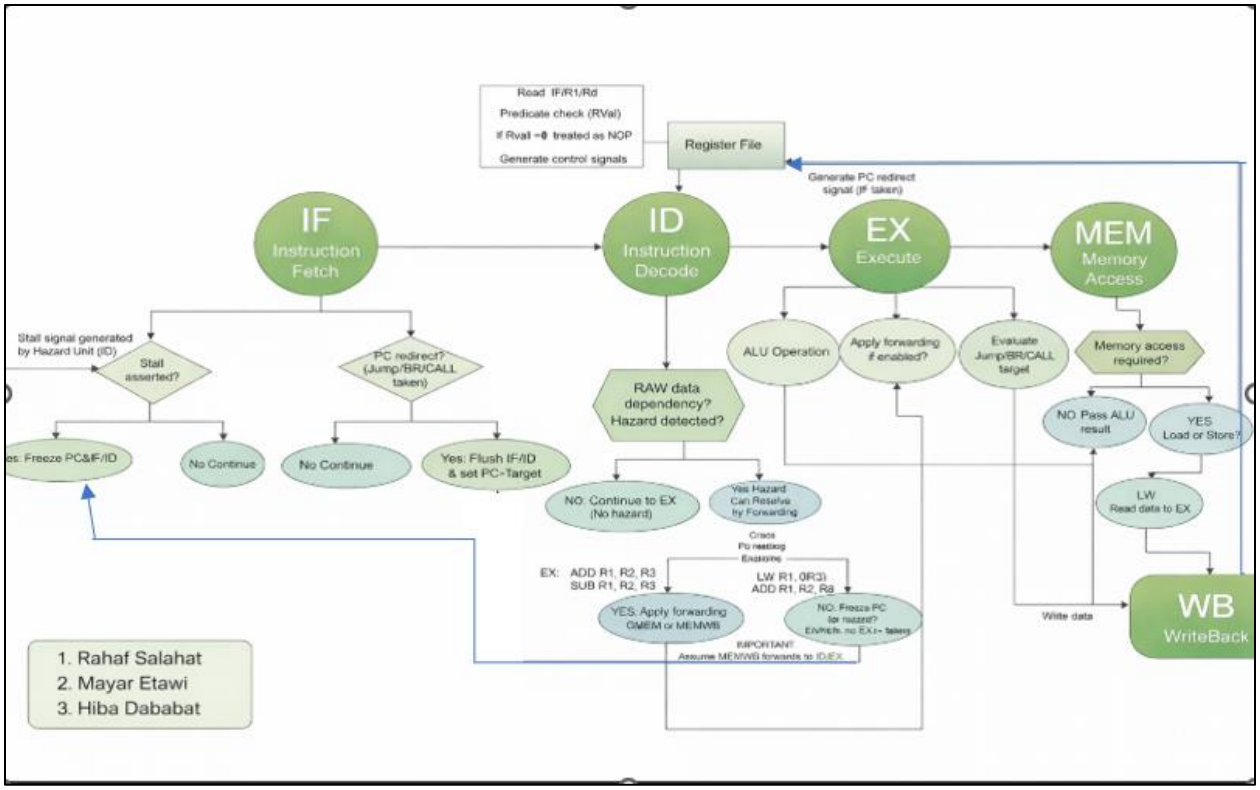


Figure 42: Five-Stage Pipelined Processor Data path with Hazard and FW

This figure illustrates the complete data path of a five-stage pipelined processor, consisting of the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. The diagram highlights the flow of instructions and data between pipeline stages, along with the associated control logic.

It demonstrates how pipeline hazards are managed through stall control, flushing mechanisms, and data forwarding paths to **resolve RAW** (Read After Write) dependencies. Branch, jump, and call instructions are handled via PC redirection logic to ensure correct program execution. Additionally, the interaction between the Write Back stage and the Register File is shown through the write-back data path, completing the instruction lifecycle.

Overall, the figure provides a clear and structured representation of how pipelining improves processor performance while maintaining correctness through hazard detection and control strategies.

Test Bench Verification

R-Type Instructions

This test bench verifies the correct functionality of R-Type instructions in the proposed CPU design.

Table 4: R-Type Instructions

R – Type Encoding		
Instruction	**opcode**	**In Hex Code**
ADD R1, R3, R2,R0	0	0x00023100
SUB R5, R1, R2,R0	1	0x080a1100
OR R6, R4, R6,R0	2	0x100c4300
NOR R6, R2, R5,R0	3	0x180c2280
AND R1, R1, R1,R0	4	0x20021080
JR ,R3,R0	13	0x68023300

Table 5: R-Type Control Unit Signals

R – Type Encoding		
Description	**Instruction**	**Control Hex**
ALU add, RegWrite enabled	ADD	0x2021
ALU subtract, RegWrite enabled	SUB	0x1811
ALU OR, RegWrite enabled	ORR	0x1011
ALU NOR, RegWrite enabled	NOR	0x0011
ALU AND, RegWrite enabled	AND	0x0811
Register jump only	JR	0Xc200

```
// ------------------------------

initial begin

  for (i = 1; i < 32; i = i + 1)
    cpu.IDSTG.RF.R[i] = 32'd0;
    cpu.IDSTG.RF.R[1] = 32'd10;
    cpu.IDSTG.RF.R[2] = 32'd1;
    cpu.IDSTG.RF.R[3] = 32'd5;
      cpu.IDSTG.RF.R[4] = 32'd10;
  cycle = 0;
end

//
```

Figure 43: Initial Values of Register File

Before instruction execution begins, the register file is initialized with predefined values to ensure deterministic behavior and to avoid undefined states. Registers **R1, R2, R3, and R4** are assigned known initial values, while the remaining registers are cleared. This initialization guarantees that all arithmetic and logical operations operate on valid data from the start of execution.
Furthermore, the **RP (Predicate Register)** field in the instructions is connected to **register R0**. Since **R0** always contains zero, the execution condition is automatically satisfied, allowing all instructions to execute without being blocked. Under this configuration, the processor executes all operations normally through the pipeline, and the control signals behave as expected.

```
◦ # KERNEL: ============================================================
◦ # KERNEL: CYCLE 1   TIME=6000 ns
◦ # KERNEL: ------------------------------------------------------------
◦ # KERNEL: IF : PC=0   instIF=00023100
◦ # KERNEL: ID : instID=00000000
◦ # KERNEL: HZ : stall=0   Kill=0
◦ # KERNEL: ------------------------------------------------------------
◦ # KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
◦ # KERNEL: ------------------------------------------------------------
◦ # KERNEL: EX : ALU=00000000   Rdest=0   WB=1 MR=0 MW=0
◦ # KERNEL: MEM: ALU=xxxxxxxx   MEMD=xxxxxxxx   Rdest=x   WB=x MR=x MW=x
◦ # KERNEL: WB : WR=x   Rdest=x   WBData=xxxxxxxx
◦ # KERNEL: ------------------------------------------------------------
◦ # KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000000 R8=00000000
◦ # KERNEL: ------------------------------------------------------------
◦ # KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
◦ # KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
◦ # KERNEL: ============================================================
```

Figure 44: Results for the testing

In the first clock cycle (Cycle 1), no visible computation results appear because the processor pipeline is still in its initialization phase. During this cycle, the instruction is only fetched from the instruction memory and placed in the **Instruction Fetch (IF)** stage, as indicated by the valid instIF value. However, the **Instruction Decode (ID)** stage still contains a null or default instruction (instID = 0) because the instruction has not yet propagated through the IF/ID pipeline register. Consequently, the **Execute (EX)**, **Memory (MEM)**, and **Write Back (WB)** stages do not process any valid instruction in this cycle. This explains the presence of undefined (X) values

in signals such as the ALU output, memory data, and destination register fields. These undefined values are expected behavior in early cycles since the pipeline stages are initially empty and their registers have not yet been loaded with valid data. Meaningful computation results only begin to appear after several cycles, once the pipeline becomes fully filled and instructions progress through all stages.

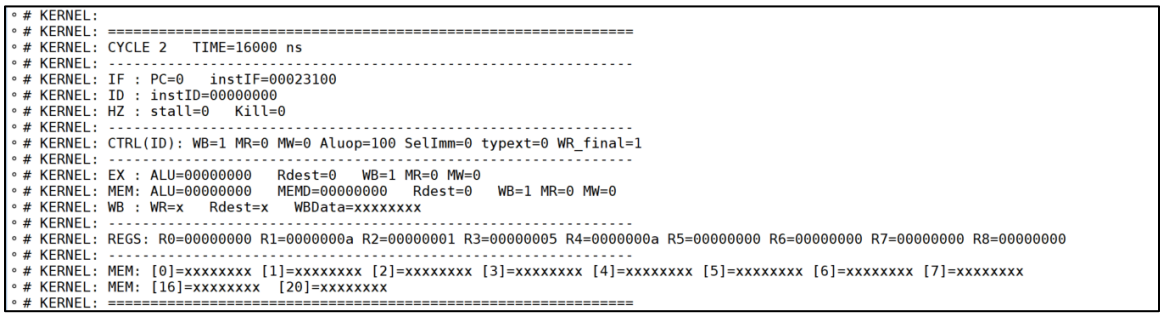


Figure 45: Results for the testing

In Cycle 2, the instruction has been successfully fetched and is now ready to begin its execution path through the pipeline. The instruction remains visible in the **Instruction Fetch (IF)** stage, while the **Decode (ID)** stage is preparing to process it. Control signals are already generated in this cycle, indicating that the processor has correctly recognized the instruction type and is setting up the required execution behavior. Although no final results are produced yet, this cycle marks the point where the instruction starts actively progressing through the pipeline toward execution in the subsequent stages.

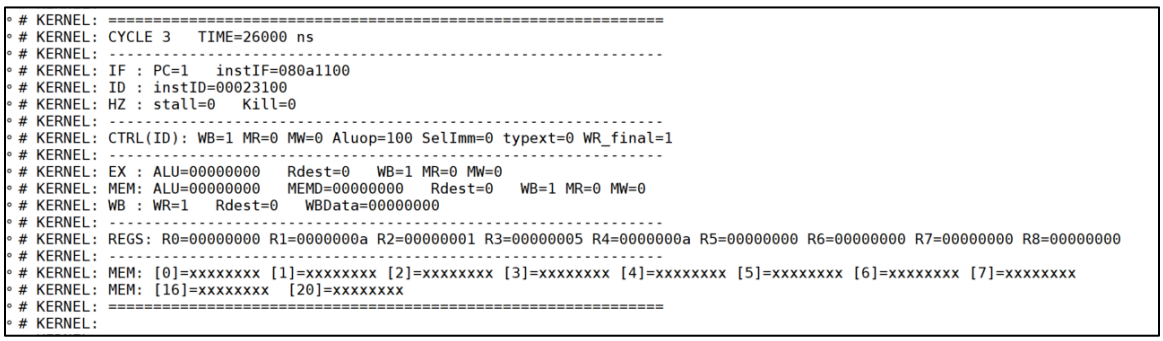


Figure 46: Results for the testing

In Cycle 3, the first instruction reaches the **Instruction Decode (ID)** stage, where it is decoded and the required control signals are generated. As shown, the **ALUop** signal is set to **100**, indicating an **ADD** operation, in preparation for executing the instruction that adds the contents of registers **R3** and **R2** and stores the result in register **R1**.
At the same time, due to the parallel nature of the **pipelined architecture**, the second instruction immediately enters the **Instruction Fetch (IF)** stage. This demonstrates how pipelining allows

multiple instructions to be processed concurrently, with the fetch of a new instruction occurring while a previous instruction is being decoded.

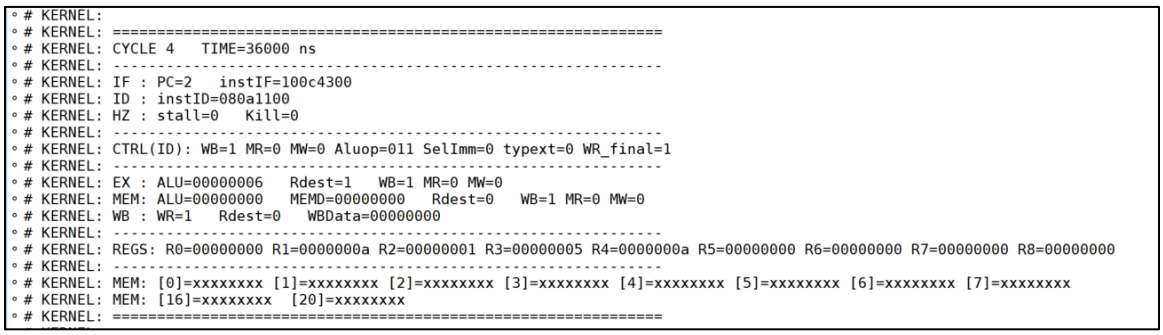


Figure 47: Results for the testing

In Cycle 4, the pipeline continues to progress in a parallel manner across multiple stages simultaneously. The **third instruction** enters the **Instruction Fetch (IF)** stage, where it is fetched from instruction memory. At the same time, the **second instruction** reaches the **Instruction Decode (ID)** stage, and its corresponding control signals are generated. In this stage, the **ALUop** value is updated to **011**, indicating that the required operation is a subtraction (**SUB**). Meanwhile, the **first instruction** advances to the **Execute (EX)** stage, where the actual arithmetic operation is performed by the ALU. As shown in the output, the result of this operation is clearly visible in the **ALU output** during the execute stage, confirming that the first instruction is being executed

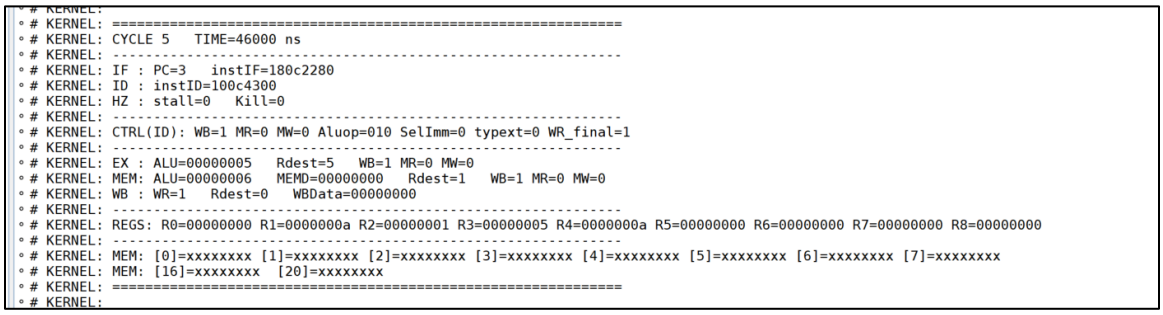


Figure 48: Results for the testing

In Cycle 5, the pipeline becomes fully utilized, with multiple instructions executing concurrently across different stages. During this cycle, a new instruction enters the **Instruction Fetch (IF)** stage, while the third instruction is located in the **Instruction Decode (ID)** stage, where its control signals are generated. As shown, the **ALUop** signal is set to **010**, indicating a logical OR operation, with write-back enabled for a later stage.

At the same time, the **second instruction** reaches the **Execute (EX)** stage, where the actual computation is performed by the ALU. It is important to note that this instruction depends on the

value of register **R1**. Although **R1** is being updated by a previous instruction and has not yet been written back to the register file, the ALU result clearly shows that the processor uses the **updated value directly**. This behavior demonstrates the effectiveness of the **forwarding mechanism**, which forwards the most recent result from later pipeline stages directly into the execution stage, eliminating the need to wait for the write-back phase and preventing data hazards.

Meanwhile, the **first instruction** has advanced to the **Memory (MEM)** stage. Since this instruction does not involve memory access, the MEM stage functions as a pass-through stage, forwarding the ALU result toward the **Write Back (WB)** stage.

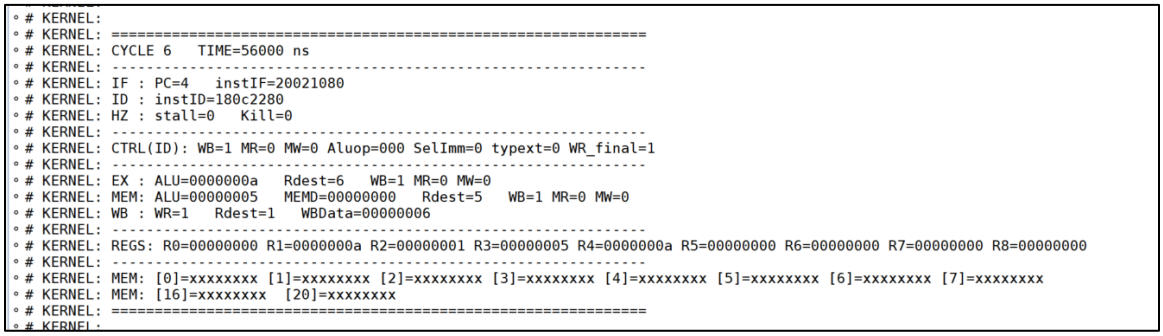


Figure 49: Results for the testing

In Cycle 6, the processor continues to operate in a pipelined manner, with instructions progressing concurrently through different stages. During this cycle, a new instruction enters the **Instruction Fetch (IF)** stage at **PC = 4**, while the preceding instruction is located in the **Instruction Decode (ID)** stage, where it is decoded and its control signals are generated. The control signals indicate **ALUop = 000**, which corresponds to a **NOR** operation, with the final write enable signal **WR_final = 1** asserted.

At the same time, another instruction reaches the **Execute (EX)** stage, where the logical operation is performed by the ALU. The result of this operation is clearly visible as **ALU = 0x0000000a**, with the destination register set to **Rdest = 6**, confirming correct execution using the appropriate source operands.

In parallel, an older instruction advances into the **Memory (MEM)** stage. The ALU result associated with this instruction is **0x00000005**, with **Rdest = 5**. Since this instruction does not involve any memory access, the MEM stage functions purely as a pass-through stage, forwarding the result toward write-back.

Finally, in the **Write Back (WB)** stage, the write enable signal (**WR = 1**) is asserted, and the value **0x00000006** is prepared to be written into the destination register **Rdest = 1**. However, this value does not immediately appear in the register file during the same cycle. This is because **register writes occur only on the positive clock edge**, meaning the actual update of the register

file becomes visible in the following cycle. This behavior is expected and ensures proper synchronization of data within the pipelined architecture.

```
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 7   TIME=66000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=5   instIF=xxxxxxxx
# KERNEL: ID : instID=20021080
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=001 SelImm=0 typext=0 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=fffffffa   Rdest=6   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=0000000a   MEMD=00000000   Rdest=6   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=5   WBData=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000000 R8=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
```

Figure 50: Results for the testing

In Cycle 7, the pipeline continues to advance smoothly, with each instruction moving into its next stage. During this cycle, the newest instruction is located in the **Instruction Decode (ID)** stage, where it is decoded and its control signals are generated. As shown, the **ALUop** signal is set to **001**, indicating an **AND** operation, with the final write enable signal **WR_final = 1** asserted, meaning this instruction will write its result back to a destination register in a later stage.

At the same time, a previous instruction reaches the **Execute (EX)** stage, where the arithmetic or logical operation is performed by the ALU. The result of this operation appears as **ALU = 0xFFFFFFFA**, with the destination register set to **Rdest = 6**, confirming that the instruction has been successfully executed.

In parallel, an older instruction progresses into the **Memory (MEM)** stage. The ALU result associated with this instruction is **0x0000000A**, with **Rdest = 6**. Since no memory access is required, the MEM stage functions as a pass-through stage, forwarding the result toward the write-back stage.

Finally, in the **Write Back (WB)** stage, the write enable signal (**WR = 1**) is asserted, and the value **0x00000005** is written into the destination register **Rdest = 5**. In this cycle, the updated register values are clearly visible in the register file, confirming that the results of previous instructions have now been fully committed. This cycle demonstrates the steady-state operation of the pipeline, where instruction completion, execution, and decoding occur concurrently without stalls.

```
# KERNEL: ============================================================
# KERNEL: CYCLE 8   TIME=76000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=6   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000006   Rdest=1   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=fffffffa   MEMD=00000000   Rdest=6   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=6   WBData=0000000a
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000005 R6=00000000 R7=00000000 R8=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
```

Figure 51: Results for the testing

In Cycle 8, the processor reaches the final stage of program execution within the pipeline. During this cycle, the **last valid instruction** is located in the **Execute (EX)** stage, where its arithmetic or logical operation is performed and the result is produced by the ALU. At the same time, the **Instruction Fetch (IF)** and **Instruction Decode (ID)** stages contain undefined (**X**) values. This behavior occurs because **no additional instructions are present in the instruction file** beyond the last loaded instruction. As a result, no new instruction is fetched or decoded, and the control signals in the ID stage appear as X since they are no longer meaningful.

Meanwhile, older instructions continue progressing through the remaining pipeline stages. One instruction is visible in the **Memory (MEM)** stage, while another completes execution in the **Write Back (WB)** stage, where the final result is written into the destination register. The register file contents confirm that all expected results have been successfully committed. The appearance of X values in this cycle does not indicate an error; rather, it reflects normal pipeline behavior as the instruction stream ends and the pipeline gradually drains.

```
# KERNEL: ============================================================
# KERNEL: CYCLE 9   TIME=86000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=7   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000006   MEMD=00000000   Rdest=1   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=6   WBData=fffffffa
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000005 R6=0000000a R7=00000000 R8=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
```

Figure 52: Results for the testing

In Cycle 9, the last instruction reaches the **Memory (MEM)** stage, where the ALU result is simply forwarded without performing any memory operation. This cycle represents the final progression of the instruction as the pipeline continues draining normally after all valid instructions have completed execution.

```
# KERNEL: ================================================================
# KERNEL: CYCLE 10   TIME=96000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=8   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: WB : WR=1   Rdest=1   WBData=00000006
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000005 R6=fffffffa R7=00000000 R8=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
```

Figure 53: Results for the testing

In Cycle 10, no new register write occurs, since the value in **register R1 (value 6)** was already written in the previous cycle. At this point, the pipeline is fully drained, with no valid instructions remaining, which explains the undefined (**X**) values observed in the **IF** and **ID** stages. Therefore, this cycle does not represent a new write-back operation, but rather the natural completion and idle state of the pipeline after all instructions have finished execution.

```
# KERNEL: ================================================================
# KERNEL: CYCLE 11   TIME=106000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=9   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: WB : WR=x   Rdest=x   WBData=0000000X
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000005 R6=fffffffa R7=00000000 R8=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
```

Figure 54: Results for the testing

In Cycle 11, a register write operation occurs, but no visible change is observed in the register file because **the written value is identical to the value already stored**. Since the register already contains the value **6** from previous cycles, rewriting the same value does not produce any observable update. At this point, the pipeline is fully idle after all valid instructions have completed, and undefined (**X**) values appear in the remaining stages as a normal result of pipeline draining.

JR_ Instructions

```
----------------------------------------------------------------
REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R8=00000000
----------------------------------------------------------------
```

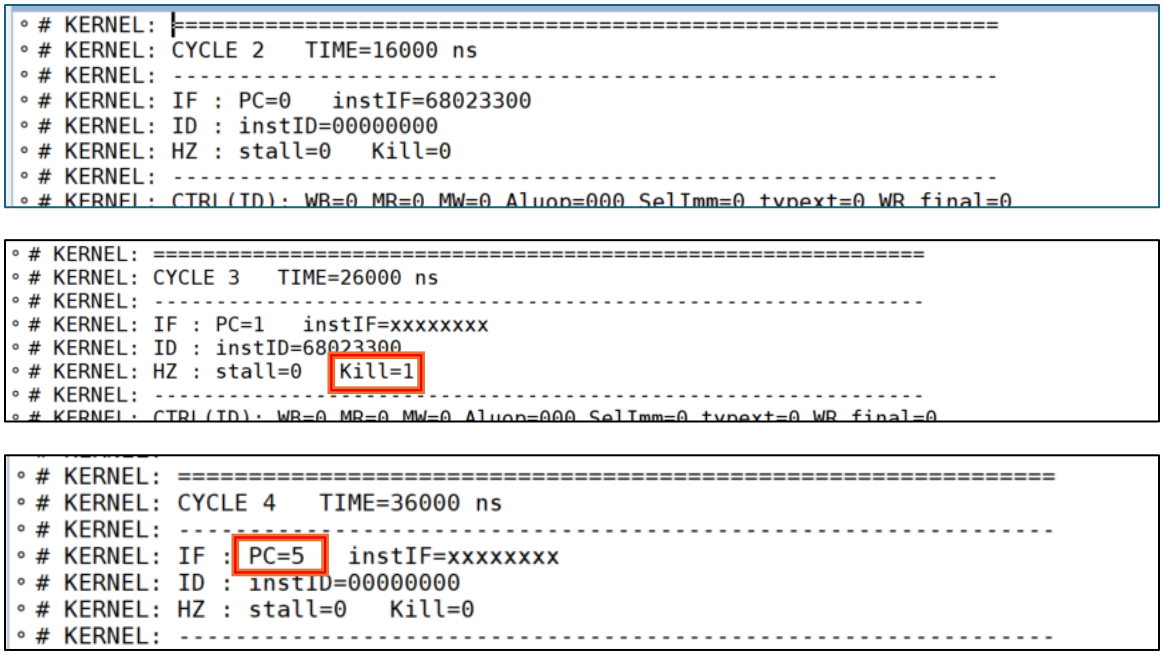


Figure 55: Results for the testing

In these cycles, a **Jump Register (JR)** instruction is executed, where the jump target address is taken directly from a register rather than an immediate value. The instruction is first fetched in the **Instruction Fetch (IF)** stage when the program counter is **PC = 0**, and then moves into the **Decode (ID)** stage in the following cycle. During the Decode stage, the control unit identifies the instruction as a Jump Register instruction and disables all register write and memory access signals, since no arithmetic or memory operation is required. The source register for this instruction is **R3**, which contains the value **5**. As a result, the jump target address is set to the content of **R3**. Once the jump decision is resolved, the pipeline asserts the **Kill** signal to flush the incorrectly fetched sequential instruction. In the next cycle, the effect of the jump becomes visible when the **Instruction Fetch (IF)** stage fetches the instruction located at **PC = 5**, confirming that execution has been redirected to the address stored in register **R3**.

I-Type Instructions

This test bench verifies the correct functionality of I-Type instructions in the proposed CPU design.

Table 6: I-Type Instructions

I – Type Encoding		
Instruction	**opcode**	**In Hex Code**
ADDI R1, R2, 6,R0	5	0x28022006
ORI R1, R1,6,R0	6	0x30021006
NORI R3,R17,9,R0	7	0x38072009
ANDI R4,R2,1,R0	8	0x40083001
LW R4, 1(R2),R0	9	0x48082002
SW R1,3 (R7),R0	10	0x50027003

Table 8: I-Type Control Unit Signals

I – Type Encoding		
Description	**Instruction**	**Control Hex**
ALU add immediate, RegWrite enabled	ADDI	0x 2039
ALU OR immediate, RegWrite enabled	ORI	0x 1031
ALU NOR immediate, RegWrite enabled	NORI	0x 0031
ALU AND immediate, RegWrite enabled	ANDI	0x 0831

Memory read, RegWrite enabled	LW	0x 203d
Memory write, RegWrite disabled	SW	0x 202a

```
for (i = 1; i < 32; i = i + 1)
  cpu.IDSTG.RF.R[i] = 32'd0;
  cpu.IDSTG.RF.R[1] = 32'd10;
  cpu.IDSTG.RF.R[2] = 32'd1;
  cpu.IDSTG.RF.R[3] = 32'd5;
  cpu.IDSTG.RF.R[4] = 32'd10;
  cpu.IDSTG.RF.R[7] = 32'd3;

   cpu.MEMSTG.DATA_MEM[3]=32'd15;
```

Figure 56: Initial Values of Register File

Before instruction execution begins, the register file is initialized with predefined values to ensure deterministic behavior and to avoid undefined states. Registers and memory **R1, R2, R3, R4, R7 and MEM[7]** are assigned known initial values, while the remaining registers are cleared. This initialization guarantees that all arithmetic and logical operations operate on valid data from the start of execution.
Furthermore, the **RP (Predicate Register)** field in the instructions is connected to **register R0**. Since **R0** always contains zero, the execution condition is automatically satisfied, allowing all instructions to execute without being blocked. Under this configuration, the processor executes all operations normally through the pipeline, and the control signals behave as expected.

```
# KERNEL: ================================================================
# KERNEL: CYCLE 2   TIME=16000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=0   instIF=28022006
# KERNEL: ID : instID=00000000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=0 MR=0 MW=0 Aluop=000 SelImm=0 typext=0 WR_final=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=ffffffff   Rdest=0   WB=0 MR=0 MW=0
# KERNEL: MEM: ALU=ffffffff   MEMD=00000000   Rdest=0   WB=0 MR=0 MW=0
# KERNEL: WB : WR=x   Rdest=x   WBData=xxxxxxxx
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000000 R8=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=xxxxxxxx [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
```

Figure 57: Results for the testing

In **Cycle 2**, the first instruction has reached the **Instruction Fetch (IF)** stage, where it is fetched from the instruction memory and placed in the IF register. Meanwhile, the **Decode (ID)** stage still contains a zero value, meaning that no instruction has been decoded yet. At this point, no actual computation is performed, and most control signals remain inactive. This behavior is expected at the beginning of pipeline execution, since the pipeline needs several cycles to be

filled before meaningful execution results appear. Consequently, no register values are updated in this cycle, and the ALU output does not yet represent a valid operation.

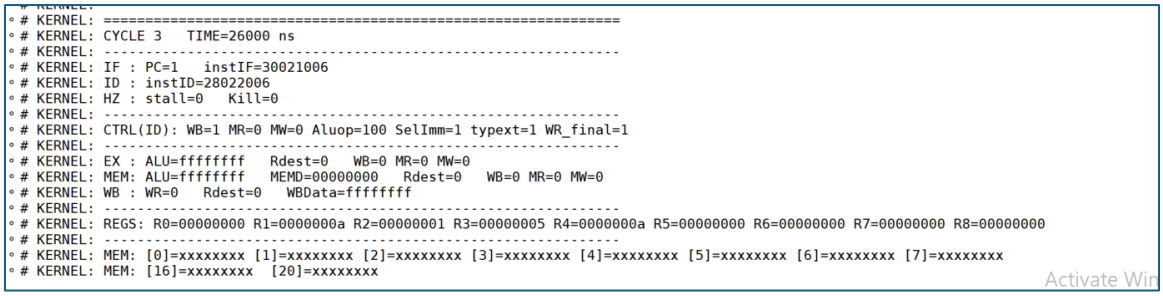


Figure 58: Results for the testing

In **Cycle 3**, the first instruction has moved into the **Decode (ID)** stage after being fetched in the previous cycle. This instruction is an arithmetic **ADD immediate (I-type)** instruction, so the control unit generates the appropriate control signals. The signal **Aluop = 100** is selected to perform an addition in the ALU, while **SelImm = 1** enables the use of the immediate value as the second ALU operand. In addition, **typext = 1** indicates that the immediate value is **sign-extended**, which is the correct behavior for arithmetic I-type instructions. At this stage, the instruction is decoded and control signals are prepared, while the actual execution will take place in the next cycle.

At the same time, due to the pipelined nature of the processor, the **second instruction enters the Instruction Fetch (IF) stage**, where it is fetched from instruction memory and stored in the IF register. This overlap demonstrates how the pipeline allows multiple instructions to be processed concurrently, with one instruction being decoded while another is being fetched.

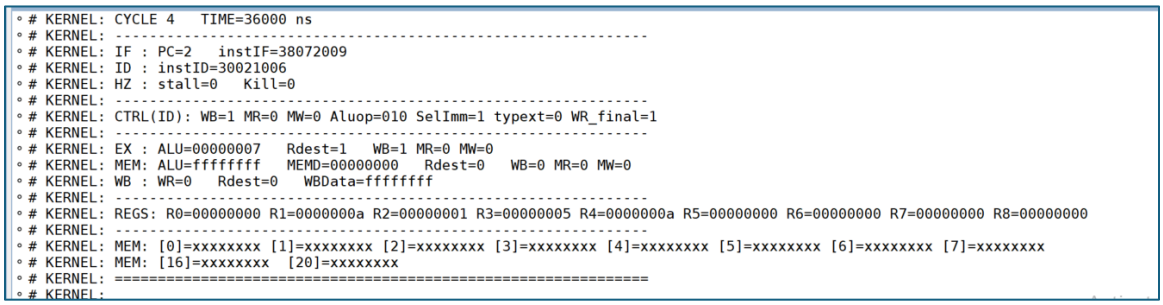


Figure 59: Results for the testing

In **Cycle 4**, the **first instruction** has reached the **Execute (EX)** stage. This instruction is an **arithmetic instruction**, specifically an **ADD Immediate** operation. During this stage, the ALU

performs an addition between the value stored in **register R2**, which is **1**, and the immediate value **6**, producing the correct result **7**, as shown on the ALU output. Since this is an arithmetic instruction, the immediate operand is handled using **sign extension**.

At the same time, the **second instruction** enters the **Decode (ID)** stage after being fetched in the previous cycle. This instruction is a **logic instruction**, with its **opcode corresponding to OR**. Accordingly, the control unit configures the ALU to perform a logical OR operation and applies **zero extension** to the immediate value, as logic instructions operate on unsigned data.

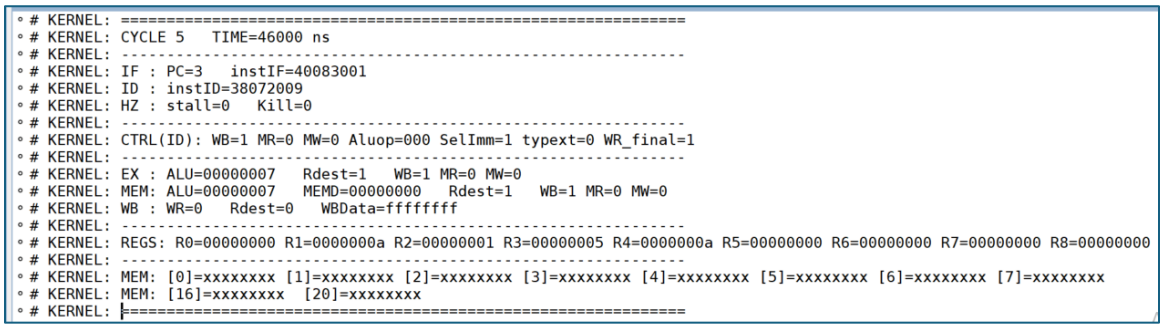


Figure 60: Results for the testing

In **Cycle 5**, the **first instruction** has reached the **Memory (MEM)** stage. Since this instruction is an arithmetic operation and does not involve memory access, no actual memory read or write is performed. The MEM stage simply forwards the ALU result to the next stage without any memory activity.

At the same time, the **second instruction** enters the **Execute (EX)** stage. This instruction is a **logical OR immediate instruction**, where the ALU performs an OR operation between **register R1** and the immediate value. Although register R1 is being updated by the first instruction, the second instruction correctly uses the **new updated value**, not the old one. This behavior clearly demonstrates that the **forwarding mechanism is functioning correctly**. The ALU output shows the correct result **7**, confirming successful execution.

Meanwhile, the **third instruction** has moved into the **Decode (ID)** stage. Since it is also a logic instruction, the control unit sets **type extension to zero extension** and assigns **AluOp = 000**, which corresponds to the **NOR operation** in the ALU design.

Finally, the **fourth instruction** enters the **Instruction Fetch (IF)** stage during this cycle, where it is fetched from instruction memory in preparation for subsequent pipeline stages. This overlapping execution highlights the effectiveness of the pipelined architecture, allowing multiple instructions to be processed simultaneously at different stages.

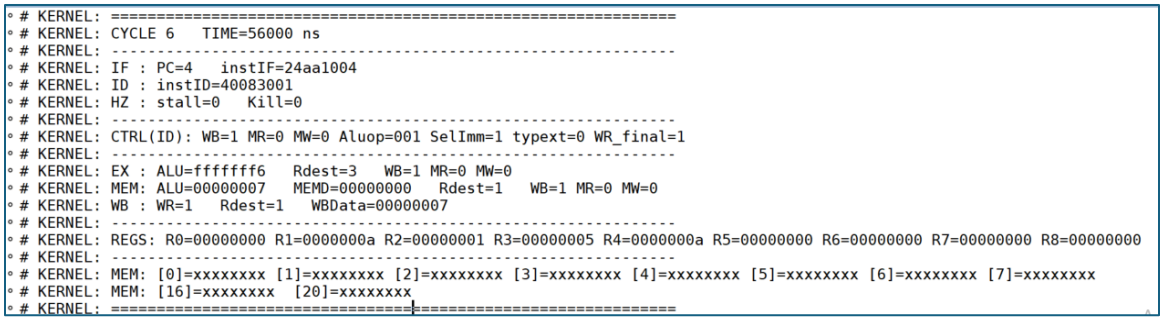


Figure 61: Results for the testing

In **Cycle 6**, the **first instruction** reaches the **Write Back (WB)** stage, where the result is prepared to be written to the register file. However, **the updated register value does not appear in this cycle**, not because the new value is the same as the old one, but because **register writes occur only on the next positive clock edge**.
As a result, the effect of the write-back operation becomes visible in the following cycle rather than immediately during the WB stage.

At the same time, the **second instruction** is in the **Memory (MEM)** stage. Since this instruction does not access memory (it is neither a load nor a store), the MEM stage performs no actual operation and simply forwards the ALU result to the next stage.

Meanwhile, the **third instruction** enters the **Execute (EX)** stage. This is a **logical NOR immediate instruction**, where the ALU performs a NOR operation between the register value and the immediate operand. Given that the register value is **0** and the immediate value is **9**, the OR result is **9**, and applying the NOT operation yields **~9 = 0xFFFFFFF6**, which is correctly shown on the ALU output.

In parallel, the **fourth instruction** is in the **Decode (ID)** stage. Since it is a **logical AND instruction**, the control unit sets **type extension to zero extension** and assigns **AluOp = 001**, corresponding to the AND operation in the ALU.

Finally, the **fifth instruction** enters the **Instruction Fetch (IF)** stage during this cycle, where it is fetched from instruction memory to be processed in the subsequent cycles. This cycle clearly demonstrates the concurrent execution of multiple instructions across different pipeline stages.

```
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 7   TIME=66000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=5   instIF=50027003
# KERNEL: ID : instID=48082002
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=1 MW=0 Aluop=100 SelImm=1 typext=1 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=4   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=fffffff6   MEMD=00000000   Rdest=3   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=1   WBData=00000007
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000007 R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R8=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
```

Figure 62: Results for the testing

In **Cycle 7**, the update to **register R1** becomes visible for the first time, where its value changes to **7**. This update is a direct result of the **first instruction**, which modified the register in earlier cycles. The change only appears now because register writes occur exclusively on the **positive clock edge**.

During this cycle, the **second instruction** is in the **Write Back (WB)** stage and is preparing to write its result to the register file. However, the actual write is not committed until the next positive clock edge, which explains why the effect may not be immediately visible in this cycle.

At the same time, the **third instruction** is in the **Memory (MEM)** stage. Since this instruction does not involve any memory access, no changes occur during this stage, and the ALU result is simply forwarded to the next stage.

Meanwhile, the **fourth instruction** is in the **Execute (EX)** stage and performs a **logical AND operation**. The ALU uses the **updated value of register R3**, which was modified by a previous instruction. The fact that the correct new value is used instead of the old one clearly demonstrates that the **forwarding mechanism is working correctly**. The AND operation with the value **1** produces the correct result, which is shown on the ALU output.

In parallel, the **fifth instruction** is in the **Decode (ID)** stage and is identified as a **Load instruction**. As a result, the control unit enables the **memory read signal** and applies **sign extension** to the immediate value, since this is not a logic instruction.

Finally, the **sixth instruction** enters the **Instruction Fetch (IF)** stage during this cycle, where it is fetched from instruction memory in preparation for execution in subsequent cycles. This cycle clearly highlights the concurrent progression of multiple instructions through different pipeline stages.

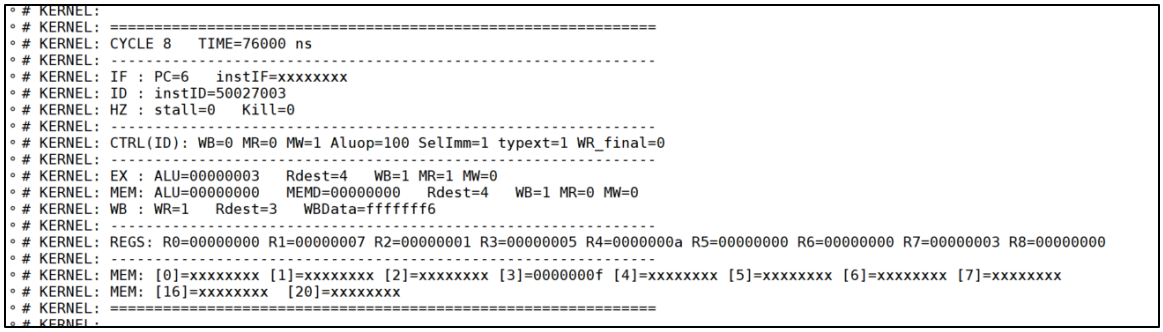


Figure 63: Results for the testing

In Cycle 8, the effect of the second instruction becomes visible in the register file. As a result of the write-back operation completed at the positive clock edge, the value of register **R3** is updated to **0xFFFFFFF6**. This update corresponds to the logical NOR immediate instruction that was executed in earlier cycles.

During this cycle, the third instruction is in the **Write Back (WB)** stage and is preparing to write its result to the register file. However, the actual register update will not be visible until the next positive clock edge, so the effect of this instruction does not yet appear in the register values during this cycle.

At the same time, the fourth instruction is in the **Memory (MEM)** stage. Since this instruction is an arithmetic operation and does not require any memory access, no memory read or write occurs, and the ALU result is simply forwarded to the next stage.

Meanwhile, the fifth instruction is in the **Execute (EX)** stage and is identified as a **Load** instruction. In this stage, the ALU calculates the effective memory address by adding the value stored in register **R2**, which is **1**, to the immediate value **2**, producing the address **3**. This address points to the memory location from which the data will be loaded in the next stage.

In parallel, the sixth instruction is in the **Decode (ID)** stage, where it is identified as a **Store** instruction. The control unit generates the appropriate control signals in preparation for a memory write operation, while the actual address calculation and data transfer will occur in the following stages.

Finally, the seventh instruction enters the **Instruction Fetch (IF)** stage during this cycle, where it is fetched from instruction memory to be processed in the upcoming cycles.

```
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================
# KERNEL: CYCLE 9   TIME=86000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=7   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000006   Rdest=1   WB=0 MR=0 MW=1
# KERNEL: MEM: ALU=00000003   MEMD=0000000f   Rdest=4   WB=1 MR=1 MW=0
# KERNEL: WB : WR=1   Rdest=4   WBData=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000007 R2=00000001 R3=fffffff6 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R8=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================
```

Figure 64: Results for the testing

In Cycle 9, the effect of the third instruction becomes visible in the register file. After completing the write-back operation at the positive clock edge, the value loaded from memory is written into **register R4**, which now holds the value **0x0000000F**.

During this cycle, the fourth instruction is in the **Write Back (WB)** stage and is preparing to write its result to the register file. As with previous cycles, this update will only become visible at the next positive clock edge.

At the same time, the fifth instruction is in the **Memory (MEM)** stage. This instruction is a **Load** instruction, and it performs a memory read operation. The data stored at memory address **3** is accessed and forwarded to the Write Back stage.

Meanwhile, the sixth instruction is in the **Execute (EX)** stage and is identified as a **Store** instruction. In this stage, the ALU calculates the effective memory address by adding the value in register **R2**, which is **1**, to the immediate value **2**, resulting in the address **3**. The data to be stored is taken from the content of **Rd** and will be written to memory in the next stage.

In parallel, the seventh instruction is in the **Decode (ID)** stage, where it is decoded and its control signals are generated.

Finally, the eighth instruction enters the **Instruction Fetch (IF)** stage with the program counter set to **PC = 7**, preparing it for execution in subsequent cycles.

```
# KERNEL: ================================================================
# KERNEL: CYCLE 10   TIME=96000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=8   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000006   MEMD=00000000   Rdest=1   WB=0 MR=0 MW=1
# KERNEL: WB : WR=1   Rdest=4   WBData=0000000f
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000007 R2=00000001 R3=fffffff6 R4=00000000 R5=00000000 R6=00000000 R7=00000003 R8=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
```

Figure 65: Results for the testing

In Cycle 10, the effect of the fourth instruction becomes visible in the register file. As a result of the write-back operation completed at the positive clock edge, the value **0x0000000F** is written into **register R4**. This value corresponds to the data that was loaded from memory address **3** by the previous load instruction.

During this cycle, the fifth instruction is in the **Memory (MEM)** stage. This instruction is a **Store** instruction, and the memory write signal is enabled. The value stored in **Rd** is written into memory at the calculated address **3**, completing the store operation.

At the same time, the sixth instruction is in the **Execute (EX)** stage. Since this instruction does not perform a meaningful operation, the ALU output is zero and no register or memory activity occurs in this stage.

Meanwhile, the seventh instruction is in the **Decode (ID)** stage. As the instruction value is undefined, no valid control signals are generated at this point.

Finally, the eighth instruction enters the **Instruction Fetch (IF)** stage with the program counter set to **PC = 8**, where it is fetched from instruction memory in preparation for execution in the following cycles.

```
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 11   TIME=106000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=9   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: WB : WR=0   Rdest=1   WBData=00000006
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000007 R2=00000001 R3=fffffff6 R4=0000000f R5=00000000 R6=00000000 R7=00000003 R8=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=xxxxxxxx [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=00000007 [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
```

Figure 66: Results for the testing

In Cycle 11, the effect of the fifth instruction becomes visible in memory. As shown in the memory dump, the value **0x0000000F** is now stored at memory address **3**, confirming that the store operation was successfully completed in the previous cycle.

During this cycle, the sixth instruction is in the **Write Back (WB)** stage. This instruction writes the value **0x00000006** into **register R1**, and this update becomes visible in the register file at the positive clock edge of this cycle.

At the same time, the seventh instruction is in the **Memory (MEM)** stage. Since this instruction does not perform any memory access, no read or write operation occurs, and the stage remains inactive.

Meanwhile, the eighth instruction is in the **Execute (EX)** stage. The ALU output is zero, and no effective computation or data transfer is performed in this stage.

In parallel, the ninth instruction is in the **Decode (ID)** stage, where it is decoded in preparation for execution.

Finally, the tenth instruction enters the **Instruction Fetch (IF)** stage with the program counter set to **PC = 9**, completing another cycle of overlapping pipeline execution.

Jump-Type Instructions

This test bench verifies the correct functionality of J-Type instructions in the proposed CPU design.

Table 7: j-Type Instructions

j – Type Encoding		
Instruction	**opcode**	**In Hex Code**
J 6,R0	11	0x58000006
CALL 7, R0	12	0x60000007

Table 8: j-Type Control Unit Signals

j – Type Encoding		
Description	**Instruction**	**Control Hex**
Jump control	J	0x 4100
PC update, R31 write	CALL	0x8411

```
# KERNEL: IF : PC=0   instIF=58000006
# KERNEL: ID : instID=00000000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=0 MR=0 MW=0 Aluop=000 SelImm=0 typext=0 WR_final=0
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=ffffffff   Rdest=0   WB=0 MR=0 MW=0
```

```
# KERNEL: ============================================================
# KERNEL: CYCLE 3   TIME=26000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=1   instIF=00000001
# KERNEL: ID : instID=58000006
# KERNEL: HZ : stall=0   Kill=1
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=0 MR=0 MW=0 Aluop=000 SelImm=0 typext=0 WR_final=0
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=ffffffff   Rdest=0   WB=0 MR=0 MW=0
```

```
# KERNEL: ============================================================
# KERNEL: CYCLE 4   TIME=36000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=6   instIF=00000006
# KERNEL: ID : instID=00000000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
```

Figure 67: Results for the testing

In these cycles, a **Jump instruction** is processed through the pipeline and causes a change in the control flow. Initially, the instruction is fetched in the **Instruction Fetch (IF)** stage while the Program Counter is **PC = 0**. At this point, no decoding or execution has occurred yet. In the following cycle, the instruction moves into the **Decode (ID)** stage, where it is identified as a jump instruction. Since jump instructions do not require any register write or memory access, the control unit disables all write and memory signals. Once the jump is resolved, the pipeline asserts a **Kill** signal to flush the incorrectly fetched sequential instruction. As a result, the Program Counter is updated to the jump target address. In the next cycle, the effect of the jump becomes visible when the **IF stage fetches the instruction at PC = 6**, which is the specified jump target. This sequence clearly shows how the jump instruction redirects execution from the sequential path to the target address **6**, while flushing intermediate instructions that were fetched speculatively.

CALL- Instructions

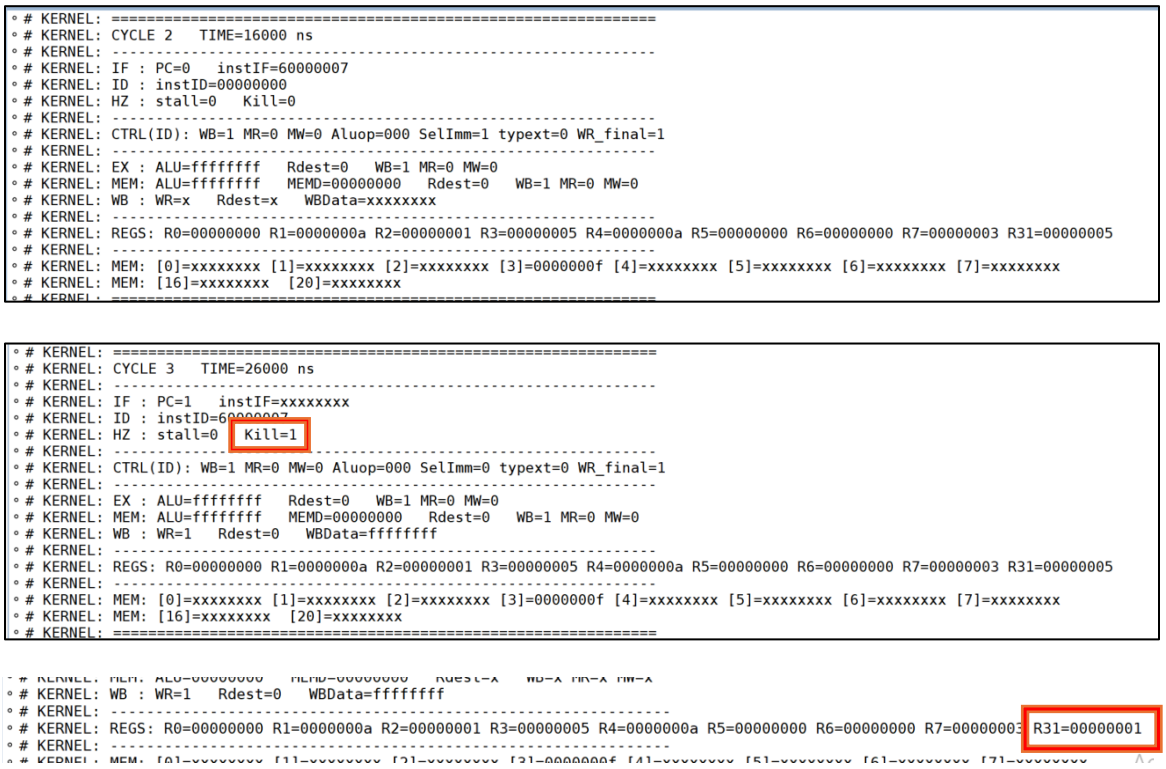


Figure 68: Results for the testing

The **CALL instruction** is a control-flow instruction that performs a function call by jumping to a target address specified by the **immediate field** of the instruction. When the CALL instruction is executed, the processor updates the **Program Counter (PC)** to the target address computed from the immediate value, allowing execution to continue at the called function. At the same time, the processor stores the **return address** (i.e., the address of the instruction following the CALL) into **register R31**, which is dedicated as the return address register. This mechanism ensures that, after completing the function, the processor can correctly return to the original execution point. The correct behavior of the CALL instruction confirms proper handling of both control transfer and return address storage in the processor design.

Pipeline Hazard and Control Test Cases

Stall

LW R1 ←MEM [[R2] +0] ,R2=1 ,MEM[1]=32’d4

ADD R3 ← R2+R1 (STALL)

In this case, the **LW** instruction loads a value into **R1**, but the loaded data is not available until the memory stage. The following **ADD** instruction tries to use **R1** immediately, which causes a **load-use hazard**, so a **stall** is inserted for one cycle to allow the load operation to complete before executing the addition.

```
# KERNEL: CYCLE 2   TIME=16000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=0   instIF=48022000
# KERNEL: ID : instID=00000000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=x   Rdest=x   WBData=xxxxxxxx
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 3   TIME=26000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=1   instIF=00061380
# KERNEL: ID : instID=48022000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=1 MW=0 Aluop=100 SelImm=1 typext=1 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
# KERNEL: ============================================================
```

```
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 4   TIME=36000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=2   instIF=xxxxxxxx
# KERNEL: ID : instID=00061380
# KERNEL: HZ : stall=1   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000001   Rdest=1   WB=1 MR=1 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 5   TIME=46000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=2   instIF=xxxxxxxx
# KERNEL: ID : instID=00061380
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=ffffffff   Rdest=0   WB=0 MR=0 MW=0
# KERNEL: MEM: ALU=00000001   MEMD=00000004   Rdest=1   WB=1 MR=1 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
```

```
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 6   TIME=56000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=3   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000007   Rdest=3   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=ffffffff   MEMD=00000000   Rdest=0   WB=0 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=1   WBData=00000004
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:
# KERNEL: ============================================================
# KERNEL: CYCLE 7   TIME=66000 ns
# KERNEL: ------------------------------------------------------------
# KERNEL: IF : PC=4   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=x   WB=x MR=x MW=x
# KERNEL: MEM: ALU=00000007   MEMD=00000000   Rdest=3   WB=1 MR=0 MW=0
# KERNEL: WB : WR=0   Rdest=0   WBData=ffffffff
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000004 R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ============================================================
# KERNEL:


# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000004 R2=00000001 R3=00000007 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
```

Figure 69: Results for the testing

Hazard

ADD R1 ← [R3]+ [R2],

[R1] ← 32'd5+ 32'd1

[R1]← 32'd6

SUB R5←[R1]-[R2]

[R5] ← 32'd6+ 32'd1

[R5]← 32'd5

In this case, sequence, the **ADD** instruction writes its result to register **R1**. The following **SUB** instruction immediately reads **R1** to use it as an operand. Since the read occurs before the write-back of the previous instruction is completed, a **Read After Write (RAW) hazard** occurs. Therefore, **forwarding** is required to ensure that the second instruction uses the correct updated value.

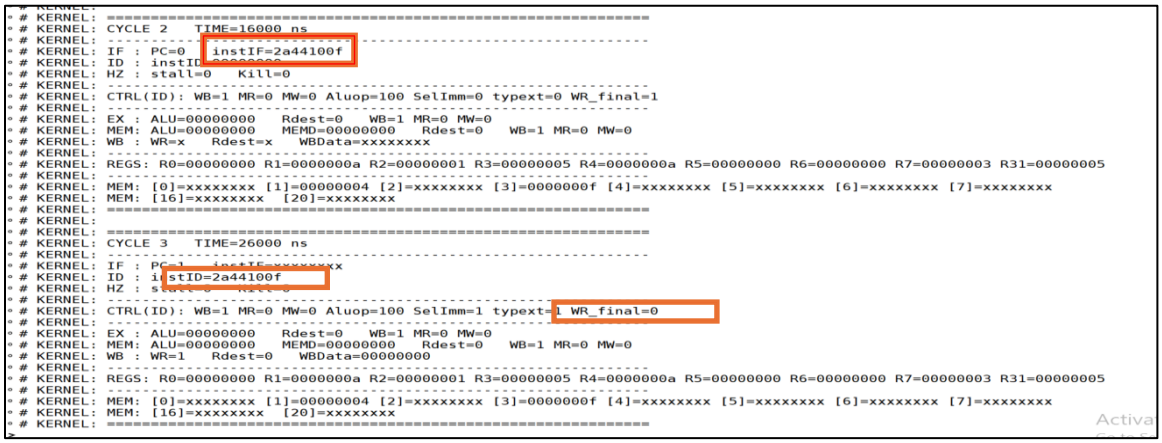
```
# KERNEL: CYCLE 2   TIME=16000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=0   instIF=00023100
# KERNEL: ID : instID=00000000
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=x   Rdest=x   WBData=xxxxxxxx
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================
# KERNEL: CYCLE 3   TIME=26000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=1   instIF=080a1100
# KERNEL: ID : instID=00023100
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=100 SelImm=0 typext=0 WR_final=1
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================
# KERNEL: CYCLE 4   TIME=36000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=2   instIF=xxxxxxxx
# KERNEL: ID : instID=080a1100
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=1 MR=0 MW=0 Aluop=011 SelImm=0 typext=0 WR_final=1
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000006   Rdest=1   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000000   MEMD=00000000   Rdest=0   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================
# KERNEL: CYCLE 5   TIME=46000 ns
# KERNEL: ----------------------------------------------------------------
# KERNEL: IF : PC=3   instIF=xxxxxxxx
# KERNEL: ID : instID=xxxxxxxx
# KERNEL: HZ : stall=0   Kill=0
# KERNEL: ----------------------------------------------------------------
# KERNEL: CTRL(ID): WB=x MR=x MW=x Aluop=xxx SelImm=x typext=x WR_final=x
# KERNEL: ----------------------------------------------------------------
# KERNEL: EX : ALU=00000005   Rdest=5   WB=1 MR=0 MW=0
# KERNEL: MEM: ALU=00000006   MEMD=00000000   Rdest=1   WB=1 MR=0 MW=0
# KERNEL: WB : WR=1   Rdest=0   WBData=00000000
# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000a R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ----------------------------------------------------------------
# KERNEL: MEM: [0]=xxxxxxxx [1]=00000004 [2]=xxxxxxxx [3]=0000000f [4]=xxxxxxxx [5]=xxxxxxxx [6]=xxxxxxxx [7]=xxxxxxxx
# KERNEL: MEM: [16]=xxxxxxxx  [20]=xxxxxxxx
# KERNEL: ================================================================
# KERNEL:
# KERNEL: ================================================================

# KERNEL: ----------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=00000006 R2=00000001 R3=00000005 R4=0000000a R5=00000005 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ----------------------------------------------------------------
```

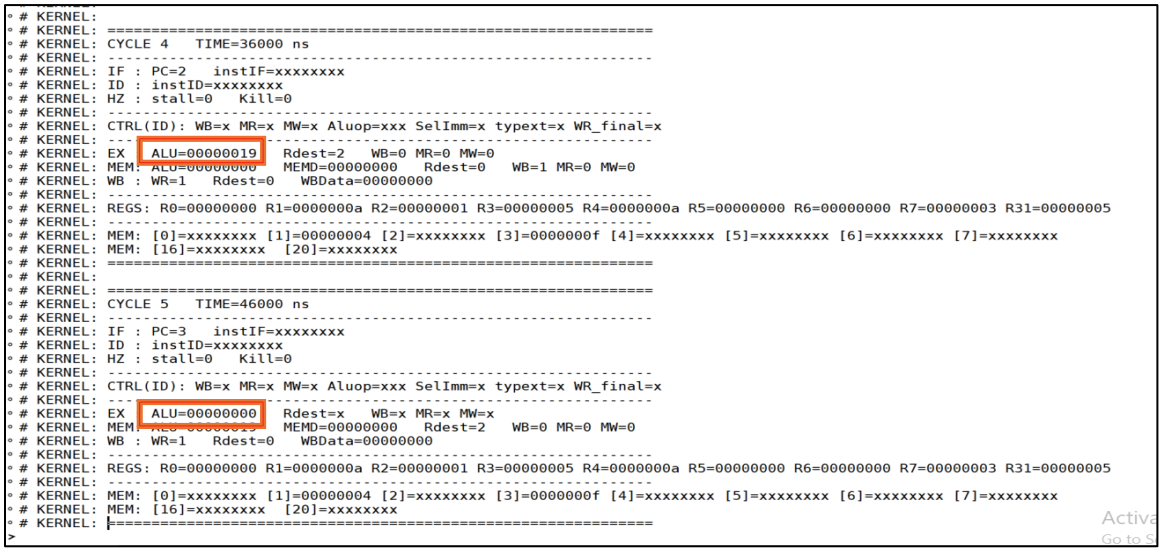
Figure 70: Results for the testing

Predicate Disabled Cases (Rp = 0)

ADDI R2 ← R1+ 32'Hf Rp=R9 ,[R9]=0

In this case, since the content of the predicate register **Rp** is zero, the instruction is allowed to proceed normally through the pipeline stages. However, when it reaches the **write-back stage**, the **WR** signal is deasserted (**WR = 0**), which prevents any update to the destination register. As a result, the instruction does not modify the register file, even though the earlier stages of execution are completed normally.





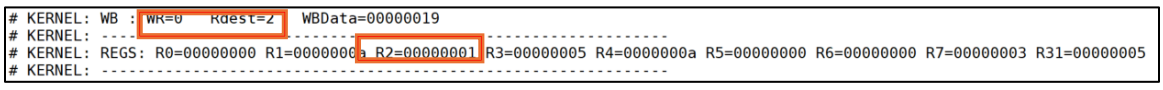


Figure 71: Results for the testing

Pipeline CPU Instruction Execution – Waveform Analysis

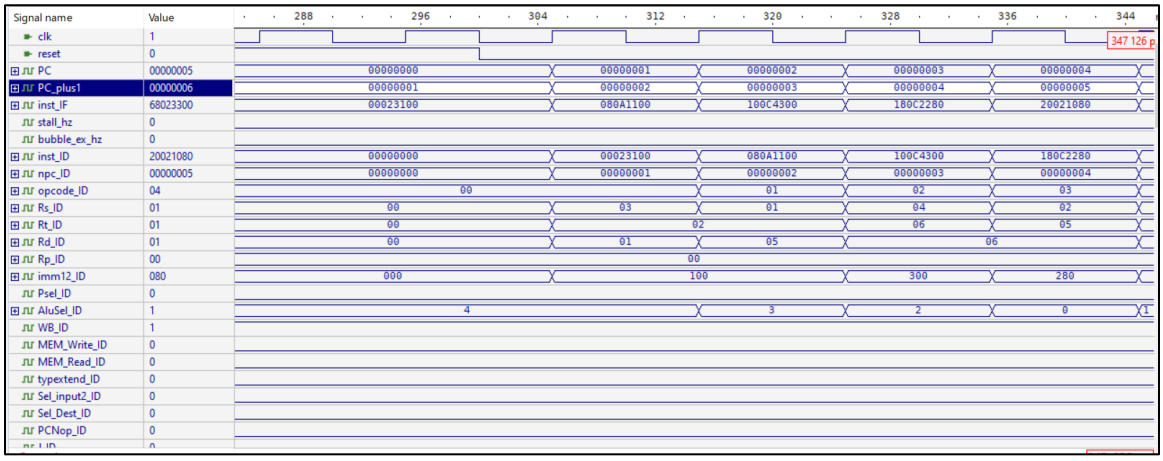


Figure 72: Waveform for Test

The waveform shows correct pipeline operation.
Instructions are fetched sequentially as indicated by the increasing PC values.
Each instruction is decoded correctly in the ID stage, with proper extraction of opcode, registers, and immediate values.
Control signals are generated correctly according to the instruction type, and no hazards or stalls are detected during this execution window.

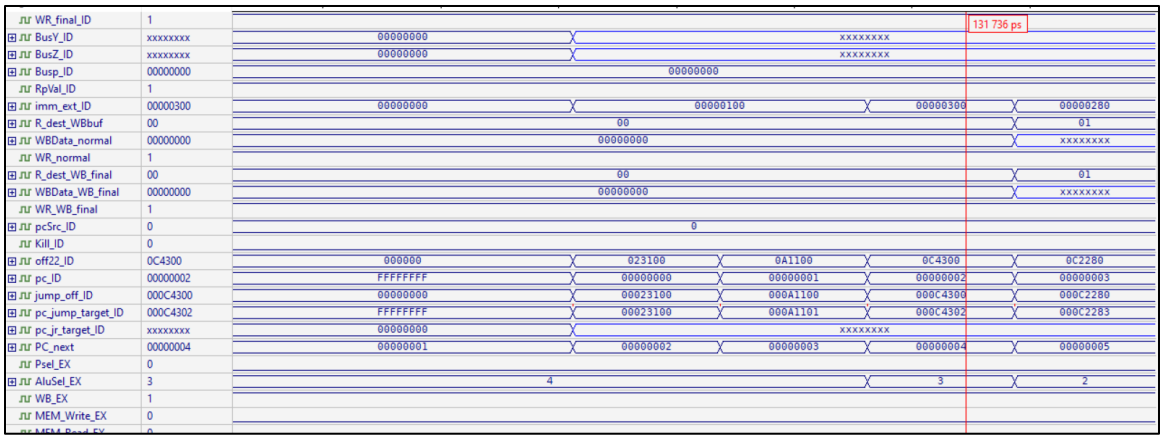


Figure 73: Waveform for Test

This waveform illustrates the execution and write-back stages of the pipelined CPU. During the execution (EX) stage, the arithmetic operation is performed according to the control signals generated by the control unit. The computed result is then forwarded to the write-back (WB) stage, where the destination register and the write data are selected. The assertion of the **WR_WB_final** signal confirms that the result is successfully written to the register file. In addition, the waveform demonstrates correct control of the program counter and proper instruction flow, indicating that the pipeline operates correctly through the execution and write-back stages.

Waveform Verification of ADD Instruction

ADD R1, R3, R2, R0

R1← R2+R3 , Rp=R0

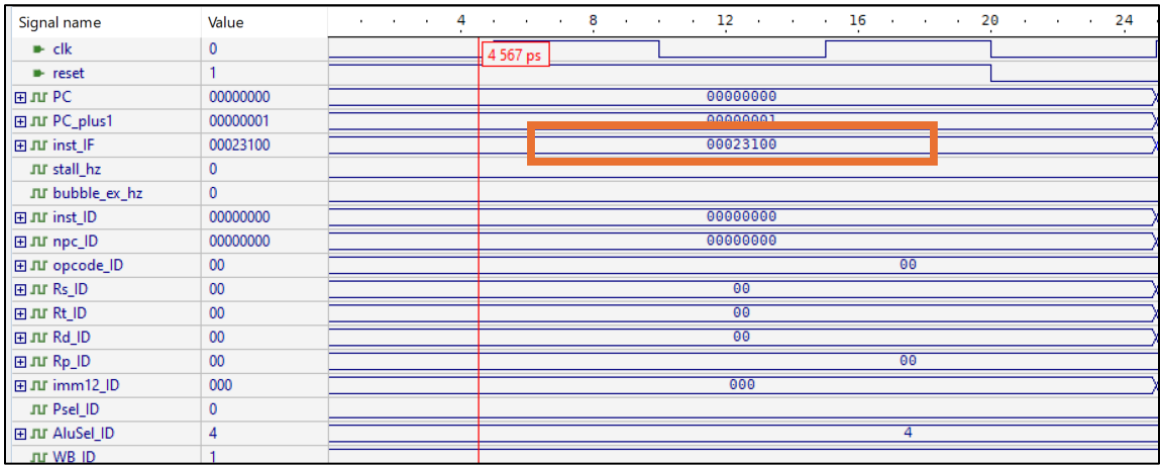


Figure 74: Instruction Fetch Stage Analysis of the ADD Instruction waveform.

Instruction Fetch Stage Analysis of the ADD Instruction

This waveform illustrates the arrival of the **ADD instruction** at the **Instruction Fetch (IF) stage)**. The Program Counter (**PC**) shows a correct value corresponding to the address of the current instruction, while the next instruction address is calculated using the **PC_plus1** signal. The instruction value is visible on the **inst_IF** signal, indicating that the instruction has been successfully fetched from the instruction memory. At this stage, the instruction has not yet moved to the **Instruction Decode (ID) stage)**, and therefore the decode-related signals remain at their default values. This behavior is expected at the beginning of instruction execution in a pipelined architecture. The waveform confirms that the fetch stage is functioning correctly and that the ADD instruction has been successfully loaded in preparation for the subsequent pipeline stages.

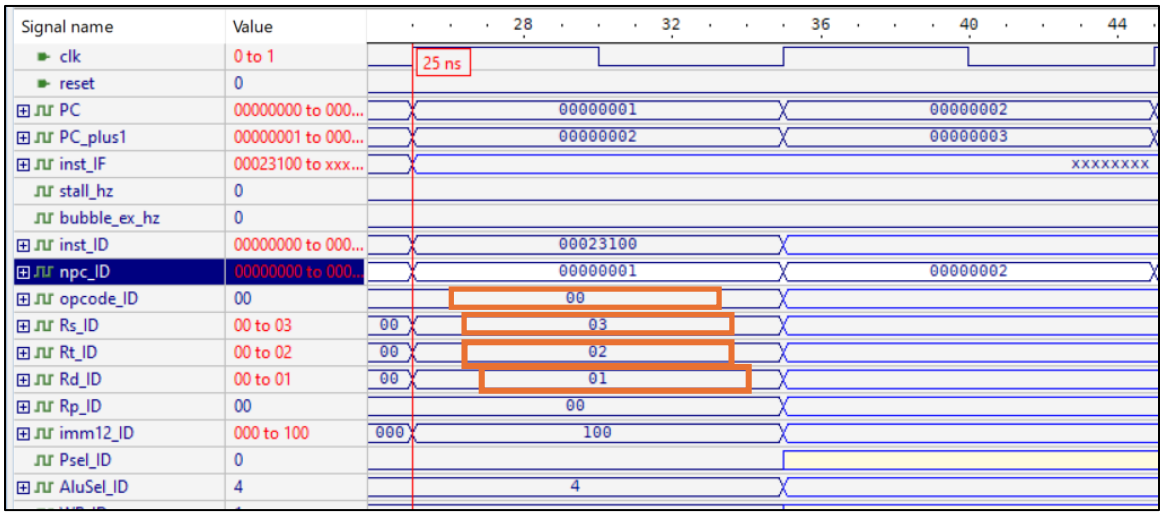


Figure 75: Instruction Decode Stage Analysis of the ADD Instruction waveform

Instruction Decode Stage Analysis of the ADD Instruction

This waveform illustrates the transition of the **ADD instruction** into the **Instruction Decode (ID) stage** after being fetched in the previous stage. At this stage, the instruction is clearly visible on the **inst_ID** signal, indicating that it is ready for decoding. The decode signals correctly identify the instruction fields, where the destination register is **Rd = R1**, the source registers are **Rs = R3** and **Rt = R2**, and the predicate register is **Rp = R0**, which means that the instruction is unconditional and will be executed normally. In addition, the signal **AluSel_ID = 4** selects the addition operation, which is consistent with the ADD instruction. The remaining control signals indicate that no hazards or stalls are present, confirming that the decode stage is functioning correctly and that the instruction is ready to proceed to the execution stage.

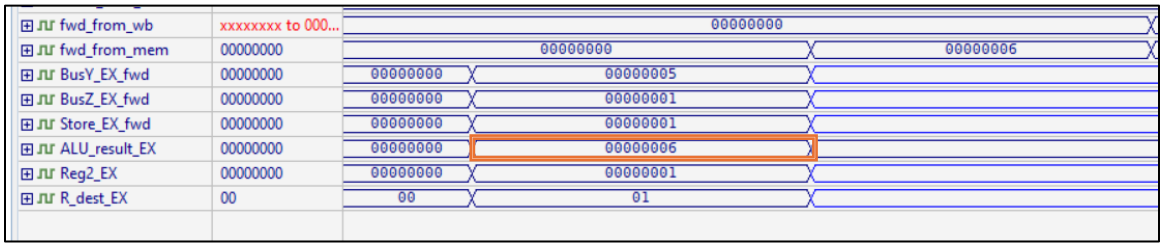


Figure 76: Execution Stage Analysis of the ADD Instruction waveform

Figure 77: Result in Reg file

Execution Stage Analysis of the ADD Instruction

This waveform shows the **ADD instruction** in the **Execution (EX) stage)**, where the arithmetic operation is performed by the Arithmetic Logic Unit (ALU). The input operands to the ALU are taken from the source registers, where **Rs = R3 holds the value 5** and **Rt = R2 holds the value 1**. Accordingly, the ALU correctly performs the addition operation, and the result appears as **ALU_result_EX = 6**, confirming that the computation **(5 + 1 = 6)** is executed correctly. This verifies the correct functionality of the execution stage and demonstrates that the ALU operates as expected.

Waveform Verification of ORI Instruction

ORI R1, R2,6 , R0

R1← R||1 6, Rp=R0

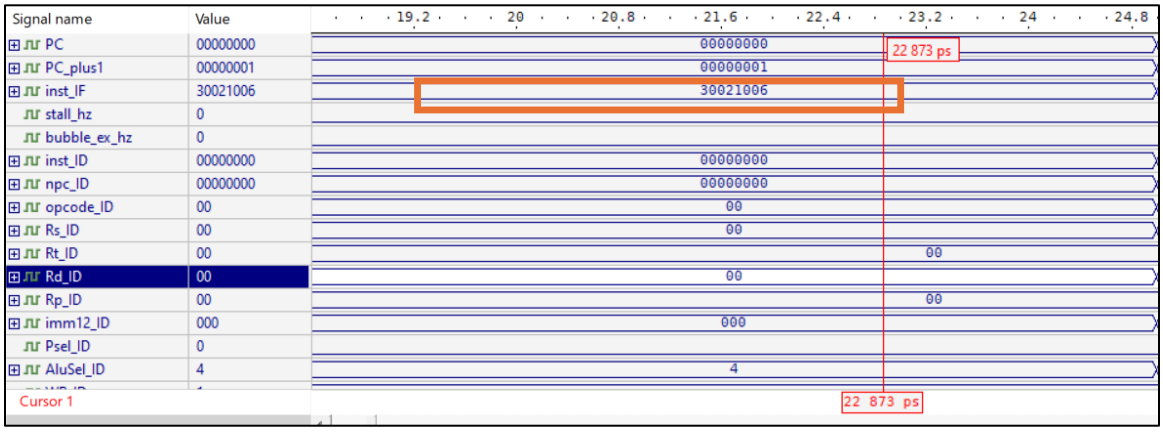


Figure 78: Instruction Fetch Stage Analysis of the OR Instruction Waveform

Instruction Fetch Stage Analysis of the OR Instruction

This waveform illustrates the **OR instruction** in the **Instruction Fetch (IF) stage** of the pipelined processor. The Program Counter (**PC**) holds the correct value corresponding to the address of the current instruction, while the next instruction address is calculated using the **PC_plus1** signal. The instruction value appears on the **inst_IF** signal, indicating that the OR instruction has been successfully fetched from the instruction memory. At this stage, the instruction has not yet advanced to the **Instruction Decode (ID) stage**, and therefore the decode-related signals remain at their default values. This behavior is expected at the beginning of instruction execution in a pipelined architecture and confirms that the fetch stage is operating correctly.

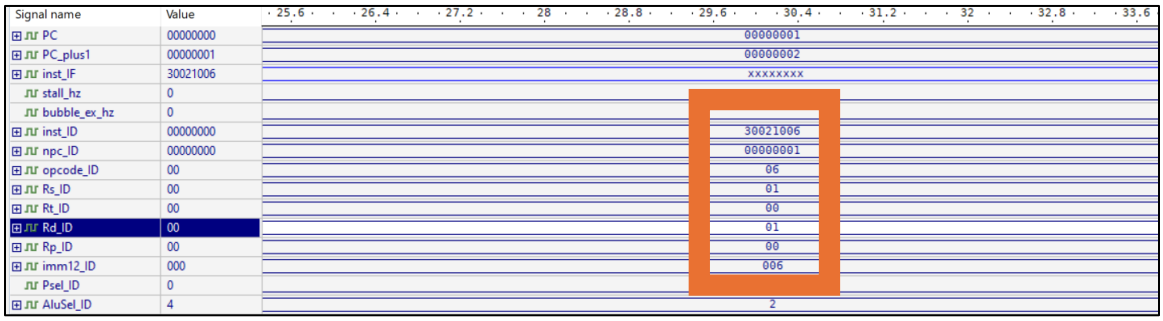


Figure 79: Instruction Decode Stage Analysis Waveform

Instruction Decode Stage Analysis

This waveform illustrates the **Instruction Decode (ID) stage**, where the fetched instruction is successfully decoded by the processor. The instruction value appears on the **inst_ID** signal, indicating that it has moved from the fetch stage into the decode stage. During this stage, the decoder correctly extracts the instruction fields, including the **opcode**, source registers (**Rs** and **Rt**), destination register (**Rd**), predicate register (**Rp**), and the immediate value (**imm12**). The control signals generated at this stage correctly reflect the instruction type, confirming that the processor has properly interpreted the instruction and prepared it for execution in the subsequent pipeline stages. This behavior verifies the correct operation of the decode stage within the pipelined architecture.

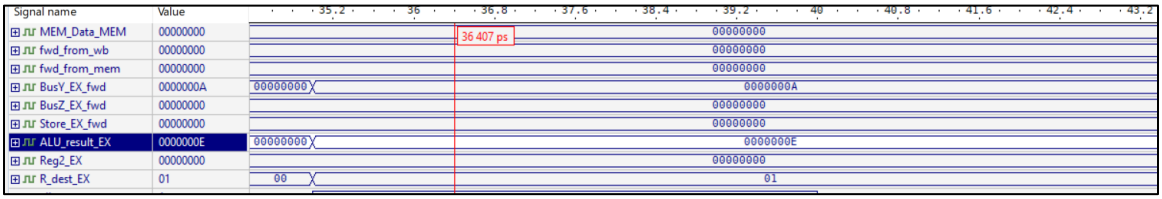


Figure 80: Instruction Execution Stage Analysis waveform

```
# KERNEL: ------------------------------------------------------------
# KERNEL: REGS: R0=00000000 R1=0000000e R2=00000001 R3=00000005 R4=0000000a R5=00000000 R6=00000000 R7=00000003 R31=00000005
# KERNEL: ------------------------------------------------------------
```

Figure 81: Result Reg File

Instruction Execution Stage Analysis

This waveform illustrates the operation of the instruction in the **Execution (EX) stage**, where the Arithmetic Logic Unit (ALU) performs the required computation using the source register values. The correct operands are applied to the ALU inputs, and the computed result is clearly visible on the **ALU_result_EX** signal.

Furthermore, the execution result matches the register values shown in the **Kernel Registers** output, confirming that the instruction has been executed correctly and that the execution stage functions as expected within the pipelined processor.

Waveform Verification of J Instruction

J ,6,R0

PC ← 6

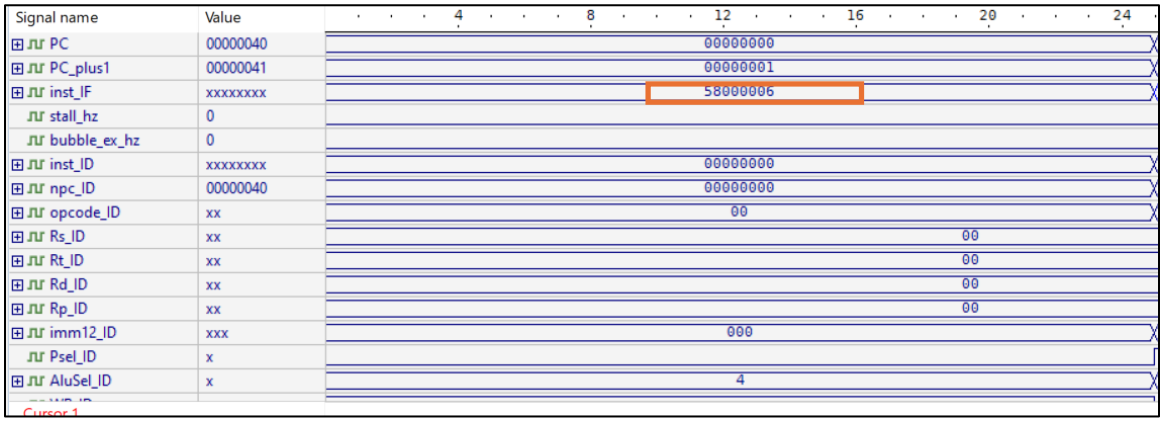


Figure 82: Instruction Fetch Stage Analysis of the Jump Instruction waveform

Instruction Fetch Stage Analysis of the Jump Instruction

This waveform illustrates the **Jump (J) instruction** in the **Instruction Fetch (IF) stage** of the pipelined processor. The Program Counter (**PC**) holds the correct value corresponding to the jump instruction address, and the next sequential address is initially calculated using the **PC_plus1** signal. The instruction value appears on the **inst_IF** signal, indicating that the jump instruction has been successfully fetched from the instruction memory. At this stage, the jump decision has not yet been resolved, and the decode and execution signals remain inactive, which is expected during the fetch stage. This waveform confirms that the fetch stage is operating correctly and that the jump instruction has been properly loaded for subsequent processing.

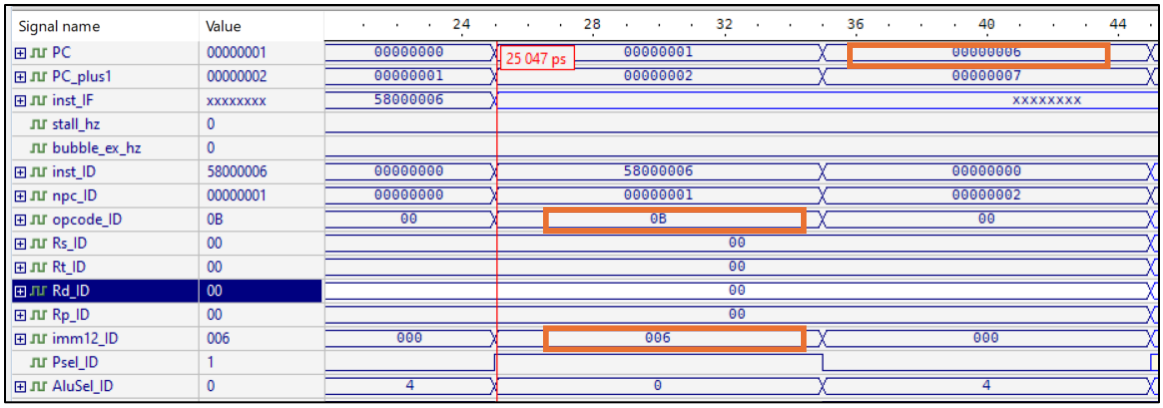


Figure 83: Decode Stage Analysis of the Jump Instruction and PC Redirection Waveform

Decode Stage Analysis of the Jump Instruction and PC Redirection

This waveform shows the **Jump instruction** in the **Instruction Decode (ID) stage**, where the instruction is visible on **inst_ID** and the decoded **opcode_ID = 0B**, confirming that the control unit identifies it as a jump operation. During decoding, the jump target/offset is extracted from **imm12_ID**, and the next PC selection is updated accordingly. As a result, the instruction flow is redirected and the **PC** changes from the normal sequential path to the jump target, which is clearly observed when **PC updates to value 8** (instead of continuing with PC+1). This verifies correct jump decoding and proper program counter update in the pipelined processor.

Teamwork