Faculty of Engineering and Technology
Electrical and Computer Engineering Department
Computer Architecture ENCS4370

Project #2 – Pipelined Processor Design

Prepared by
Osama Zeidan
ID: 1210601, Sec.: 3
Heba Mustafa
ID: 1221916, Sec.:3

Instructor
Dr. Ayman Hroub

Birzeit
12/1/2024

Table of Contents

Table of Figures

List of Tables

Abstract

This report presents the design and implementation of a pipelined 16-bit RISC processor capable of executing a subset of RISC instructions. The processor is divided into five stages: Instruction Fetch, Decode, Execute, Memory Access, and Write Back. Key components such as the ALU, instruction and data memory, and control unit are integrated to ensure efficient operation.

Control signals are generated dynamically to manage the flow of data and resolve hazards, utilizing techniques like stalling, forwarding, and flushing. Performance-monitoring registers are included to track execution metrics such as the total number of instructions, clock cycles, and stalls. The processor's functionality was validated through Verilog simulations, demonstrating correctness and efficiency in handling various instruction types.

This report provides a detailed explanation of the design, including the Datapath, control path, and hazard management mechanisms, along with the results from the simulation and testing phase.

Introduction

In this project, we aim to design a pipelined processor, an efficient implementation of instruction execution that divides the processing task into five stages: Instruction Fetch, Decode, Execute, Memory Access (Read/Write), and Write Back. The pipelining approach ensures that multiple instructions are processed simultaneously in different stages, allowing the processor to achieve a clock cycles per instruction (CPI) of 1, once the pipeline is filled. This significantly improves the speed and utilization of the processor.

Pipelining plays a crucial role in modern processor design as it provides a substantial performance advantage over single-cycle and multi-cycle processors. By overlapping the execution of multiple instructions, pipelined processors maximize resource utilization and minimize idle time. The goal of this project is to implement a processor that demonstrates these advantages by handling data and control hazards effectively using techniques like stalling, forwarding, and flushing.

In addition to improving performance, this project emphasizes correctness and completeness, ensuring all implemented instructions conform to the defined instruction set architecture (ISA). The processor will be simulated and tested using **Verilog** to validate its functionality and efficiency.

Objectives

- To design and implement a pipelined processor.
- To handle hazards using forwarding and stalling.
- To verify functionality through simulation.

Theory

Overview of Pipelining

Pipelining is a technique used to execute multiple instructions simultaneously by dividing the processor into several stages. Each instruction passes through all the stages sequentially. While the first instruction takes multiple clock cycles to move through all the stages, subsequent instructions enter the pipeline, filling it up. Once the pipeline is full, the processor achieves a Clock Cycles Per Instruction (CPI) of 1. This significantly improves efficiency and performance, making pipelined processors faster and more resource-efficient compared to single-cycle or multi-cycle designs.

Implementing pipelining involves defining distinct stages, assigning specific tasks to each stage, and handling hazards such as data dependencies and control conflicts. Effective hazard detection and resolution mechanisms, such as stalling and forwarding, are critical to ensuring smooth execution in a pipelined processor.

In our design, the processor is divided into five stages:

- Instruction Fetch (IF):
  This stage fetches the next instruction from the instruction memory using the program counter (PC). The fetched instruction is then passed to the next stage.

- Instruction Decode (ID):
  The decode stage accesses the register file to read the source registers and determines the destination register. In this stage, the control unit generates all necessary control signals based on the opcode and other parameters, such as the source and destination registers.

- Execute (EX):
  This stage includes the Arithmetic and Logical Unit (ALU), which performs arithmetic and logical operations. Additionally, the stage contains adders and other components for handling branch instructions, calculating target addresses, and determining execution results.

- Memory Access (MEM):
  In this stage, instructions that require memory access (e.g., load and store instructions) interact with the data memory. Read and write operations are controlled by memory control signals generated in the decode stage.

- Write Back (WB):

This final stage writes the result produced by the ALU or data retrieved from memory back to the register file. The destination register is determined by the decode stage, ensuring the correct result is stored.

By dividing the processor into these stages, pipelining allows overlapping execution of instructions, significantly enhancing performance. The design's efficiency depends on correctly managing hazards and ensuring seamless data flow between stages.

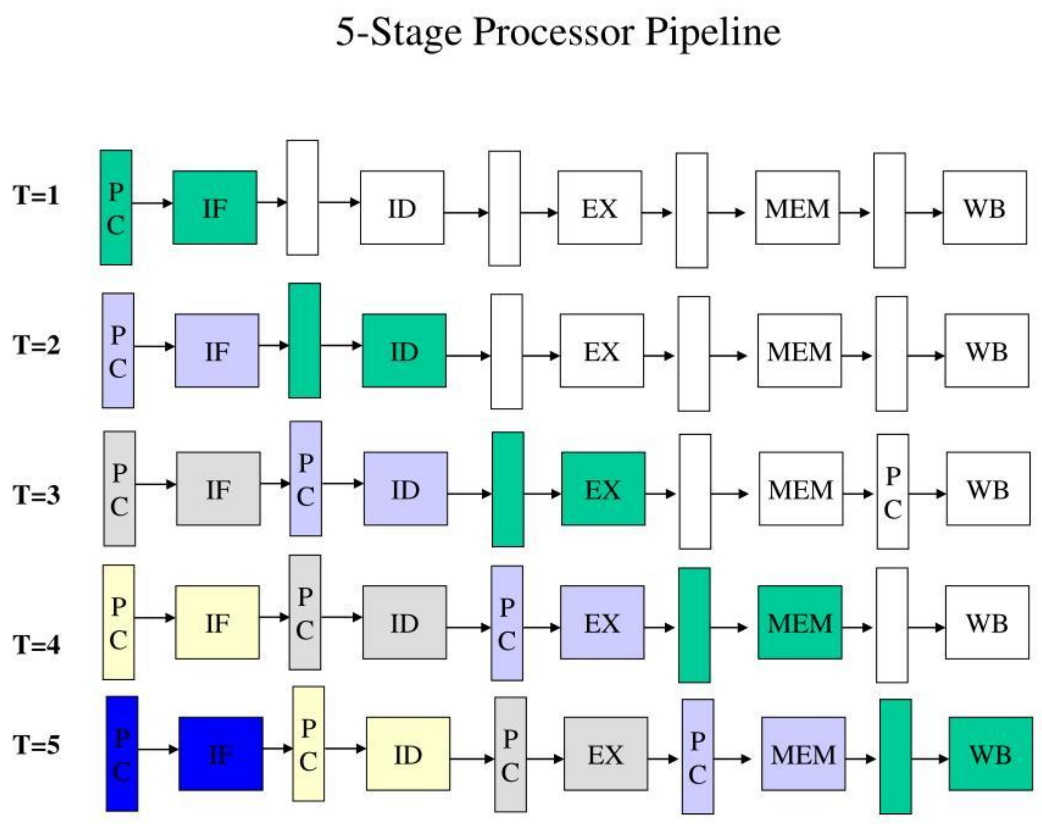


Figure 1: 5-Stage Pipelined Processor

Why Pipelining?

Pipelining significantly improves processor performance by achieving a CPI (Clock Cycles Per Instruction) of 1 in the ideal case, as mentioned earlier. It allows multiple instructions to be executed simultaneously by dividing the execution process into stages, enabling faster and more efficient instruction handling.

When comparing pipelining with other processor designs, the advantages become evident:

- Single-cycle Processors: Each instruction is executed in one clock cycle, requiring the clock cycle to be long enough to accommodate the slowest instruction. This leads to inefficiency and underutilization of hardware resources.
- Multi-cycle Processors: Each instruction is executed over multiple shorter clock cycles, reducing the cycle time and each instruction will go through the required stages only. However, instructions must still be executed sequentially, limiting throughput.
- Pipelined Processors: By overlapping instruction execution across multiple stages, pipelining combines the benefits of both designs. It reduces the execution time per instruction while improving overall throughput and resource utilization.

While pipelining offers clear performance benefits, its efficiency depends on effectively handling hazards, such as data dependencies, control conflicts, and structural resource sharing. By resolving these challenges with techniques like stalling, forwarding, and flushing, pipelined processors ensure smooth and fast execution.

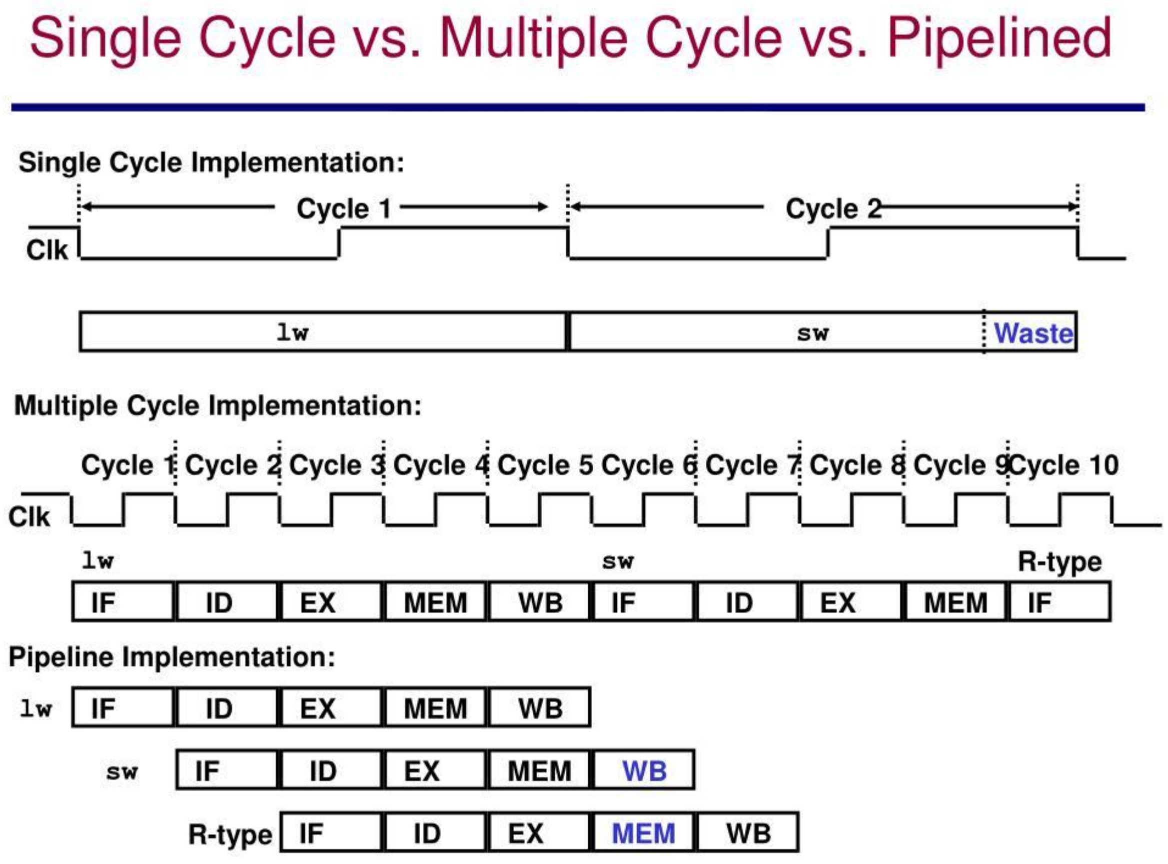


Figure 2: Single Cycle Vs. Multi-Cycle Vs. Pipelined Processor

Challenging in Pipelining

- Data Hazards:

When two dependent instructions are executed, where the first instruction writes to a register and the second instruction reads from the same register, a Read After Write (RAW) Hazard occurs. This hazard arises because the second instruction requires a value that has not yet been written by the first instruction.

To resolve this and ensure the correct value is used, stall signals must be generated to temporarily pause the pipeline. This allows the first instruction to complete writing the new value to the register before the second instruction proceeds. Once the correct value is available, the stalled pipeline can resume execution seamlessly.

- Control Hazards:
  Using control flow instructions, such as Jump or Branch, alters the normal flow of instructions in the pipeline, creating a control hazard. This happens because the pipeline must wait to determine the correct instruction flow. To handle this issue, the processor can flush the pipeline registers in the affected stages, clearing incorrect or speculative instructions.

  To further improve performance and reduce the impact of control hazards, a branch prediction logic can be implemented. This predicts the outcome of branches in advance, allowing the pipeline to continue execution with minimal interruptions. If the prediction is correct, performance is maintained; otherwise, the pipeline is flushed, and the correct instruction flow is restored.

- Structural Hazards:
  If multiple instructions attempt to access the same component simultaneously, such as reading or writing to memory, a structural hazard occurs. This conflict can be resolved by either separating instruction and data memory or by providing multiple access ports to the shared component.

  In our case, this challenge is avoided as we use separate instruction and data memories, allowing simultaneous access without any conflicts. This design choice ensures smooth and efficient execution of instructions in the pipeline.

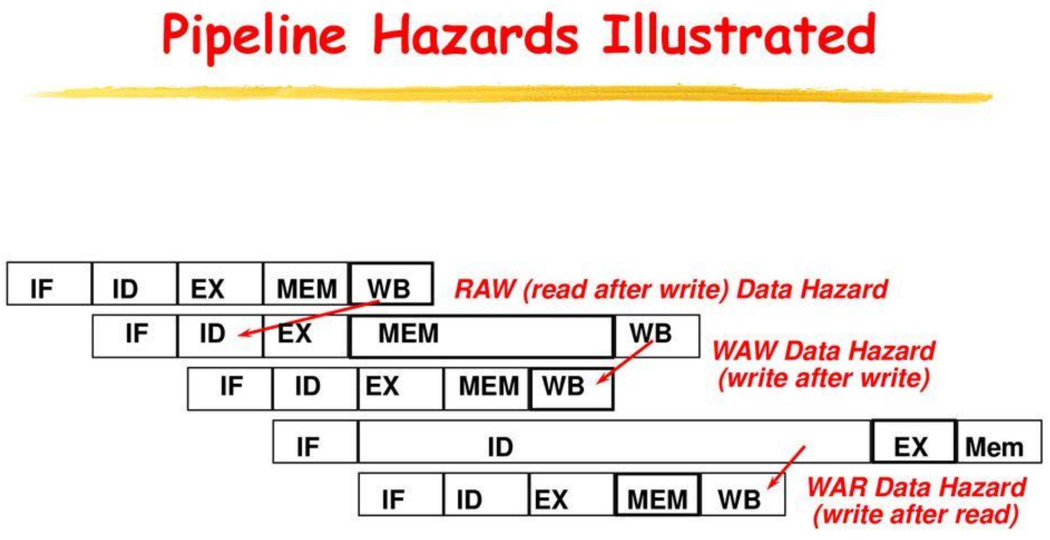

Pipeline Hazards Illustrated
IF
ID
EX
MEM
WB
RAW (read after write) Data Hazard
WAW Data Hazard (write after write)
Mem
WAR Data Hazard (write after read)

Figure 3: Pipeline Hazards

Supported ISA

In our processor design, the supported Instruction Set Architecture (ISA) includes three types of instructions: Immediate Type (I-Type), Jump Type (J-Type), and Register Type (R-Type). This classification simplifies the design by grouping similar instructions under the same format, allowing them to be controlled using a common set of control signals.

The use of these instruction types ensures flexibility and consistency in the processor's operation, enabling it to handle various functionalities efficiently. The following figure illustrates the structure and functionality of the ISA in greater detail.

R-Type (Register Type)

Opcode^4	Rd^3	Rs^3	Rt^3	Function^3

- **4-bit opcode:** opcode.
- The opcode is zero for all R-Type instructions.
- **3-bit Rd**: destination register
- **3-bit Rs:** first source register
- **3-bit Rt**: second source register
- **3-bit:** Function. This field determines the specific operation of the instruction.

I-Type (Immediate Type)

Opcode^4	Rs^3	Rt^3	Signed Imm^6

- **4-bit opcode:** opcode.
- **3-bit Rs:** first source register
- **3-bit Rt**: destination register
- The immediate value is zero-extended for logical instructions and sign-extended for all other instructions.
- In the case of BEQ and BNE instructions, which are I-type instructions, the branch target is calculated as follows:
  Branch target = Current PC + sign extended immediate

J-Type (Jump Type)

Opcode^4	9-bit offset	Function^3

- **4-bit opcode:** opcode.
- The opcode is 1 for all J-Type instructions.
- **3-bit:** Function. This field determines the specific operation of the instruction.
- The jump target address is calculated by concatenating these two fields: **PC[15:9], 9-bit offset**

Figure 4: Supported ISA

Design & Implementation

Datapath Design

To implement the processor, we designed a detailed Datapath to define how data flows through the various components of the processor and how it is synchronized across the pipeline stages. Each stage and component are controlled by specific signals, ensuring proper operation and coordination. In the following sections, we will provide a detailed description and explanation of the Datapath, its components, and the control signals that govern its functionality.

Description

This figure illustrates the Datapath of our pipelined processor, highlighting its key components and the critical control signals that govern its operation:

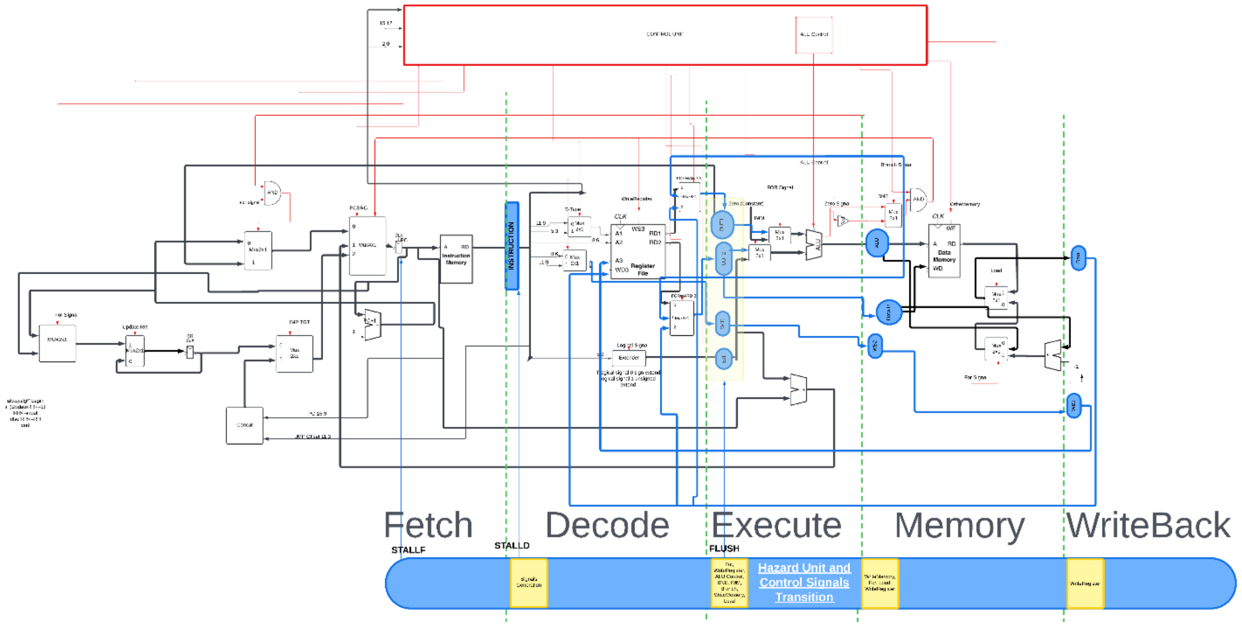


Figure 5: Full Datapath

The Datapath of our pipelined processor consists of several key components, including multiplexers, logic gates, instruction memory, data memory, an ALU, adders, extenders, and concatenators. As shown in the figure, each pipeline stage is separated by a register, which passes essential data and control signals to the next stage while maintaining synchronization.

Instruction Fetch (IF)

The process begins with the Program Counter (PC), which fetches the instruction from the instruction memory. The fetched instruction is then passed through the register between the Instruction Fetch and Decode stages. This register ensures that the instruction is preserved and ready for decoding.

Instruction Decode (ID)

In the Decode stage, the instruction is processed to:

- Read the required registers from the register file.
- Identify the destination register for storing the final result.
- Generate all necessary control signals based on the instruction's opcode and parameters.
- If the instruction is a jump, its target address is calculated in this stage, directly updating the PC to ensure correct execution flow.

Execute (EX)

The control signals and data from the Decode stage are passed to the Execute stage. Here, the processor performs:

- Arithmetic or logical operations using the ALU (e.g., ADD, SUB).
- Address calculation for memory instructions.
- Comparisons for branch instructions to determine the branch target.
- For branch instructions, the branching address is computed in this stage, and the PC is updated accordingly if the branch condition is met.

Memory Access (MEM)

In the Memory stage, the processor performs:

- Memory read operations for load instructions.
- Memory writes operations for store instructions. These operations are controlled by the signals generated in the Decode stage.

Write Back (WB)

Finally, in the Write Back stage, the processor writes the result to the destination register. The result can come either from the ALU (execution stage) or from the memory (memory stage), depending on the instruction.

Hazard Management

- Jump instructions: Resolved in the Decode stage, directly updating the PC.
- Branch instructions: Resolved in the Execute stage after condition evaluation and address calculation.

This structured Datapath ensures proper execution of instructions by maintaining synchronization and leveraging control signals for seamless transitions between stages. The separation of stages with registers enables smooth pipelining, improving performance and efficiency.

Control Path Design

The control unit is a crucial component of the processor, often referred to as the "brain" of the system. Its primary role is to interpret instructions and generate the necessary control signals to ensure that each stage of the processor operates correctly and in the proper sequence. These signals coordinate the flow of data and the operations performed by various components.

Control signals are generated during the Decode stage based on the instruction's opcode, function code, and other parameters. These signals dictate the actions required for each stage, such as selecting ALU operations, enabling memory access, or updating registers. In each pipeline stage, the relevant control signals are used to execute the instruction, while the remaining signals are passed along to subsequent stages to maintain synchronization.

Additionally, the control unit works closely with the hazard detection unit to handle data, control, and structural hazards. It generates stall or flush signals when necessary to ensure smooth execution and prevent errors. The following figure provides an overview of the control signals used in each stage and their interaction throughout the pipeline.

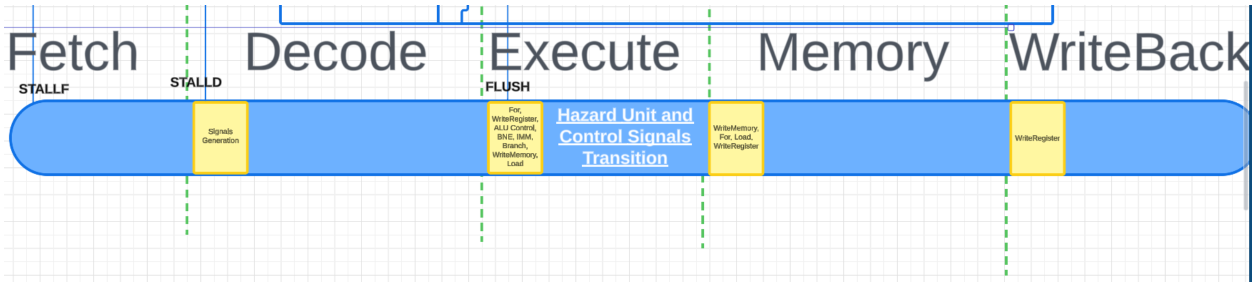
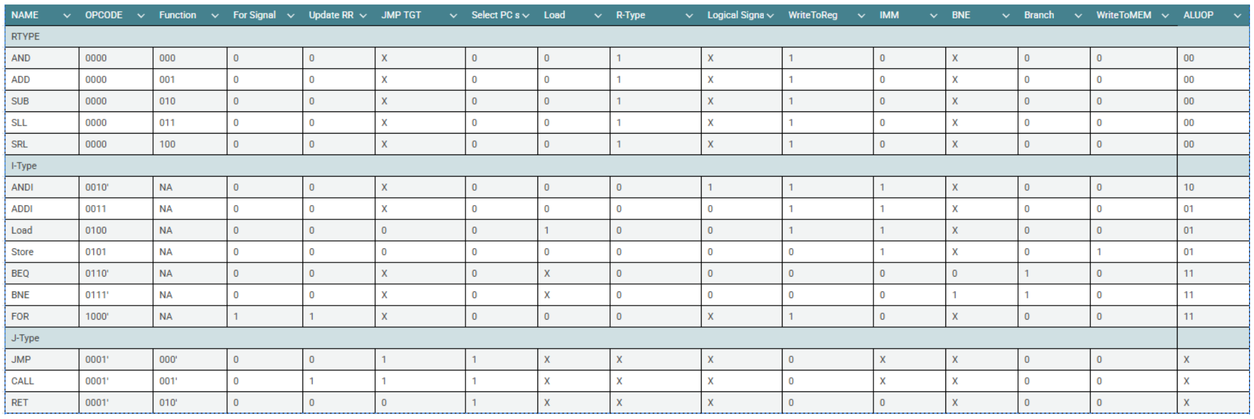

Fetch
Decode
Execute
Memory
WriteBack
STALLF
STALLD
FLUSH
Signals Generation
For, WriteRegister, ALU Control, BNE, IMM, Branch, WriteMemory, Load
Hazard Unit and Control Signals Transition
WriteMemory, For, Load, WriteRegister
WriteRegister

Figure 6: Control Unit

Now, we will discuss the control signals for each instruction type and opcode. This will be explained using the control signals' truth table, which provides a clear and systematic way to trace how each control signal is generated and used for different instructions. This table simplifies understanding and ensures the control signals are accurately aligned with the processor's operations.

Table 1: Control Signals Table



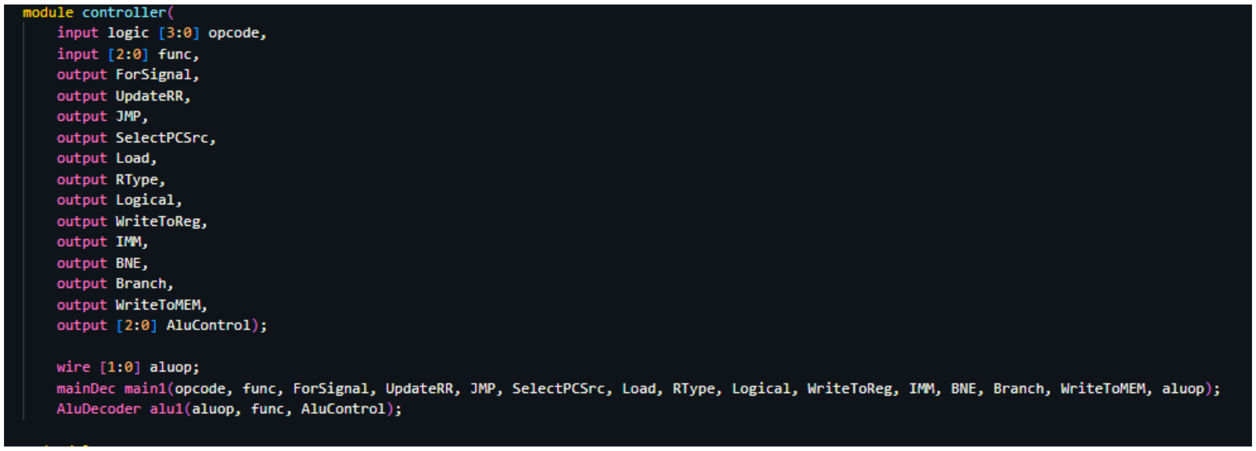
NAME	OPCODE	Function	For Signal	Update RR	JMP TGT	Select PC s	Load	R-Type	Logical Signa	WriteToReg	IMM	BNE	Branch	WriteToMEM	ALUOP
RTYPE															
AND	0000	000	0	0	X	0	0	1	X	1	0	X	0	0	00
ADD	0000	001	0	0	X	0	0	1	X	1	0	X	0	0	00
SUB	0000	010	0	0	X	0	0	1	X	1	0	X	0	0	00
SLL	0000	011	0	0	X	0	0	1	X	1	0	X	0	0	00
SRL	0000	100	0	0	X	0	0	1	X	1	0	X	0	0	00
I-Type															
ANDI	0010'	NA	0	0	X	0	0	0	1	1	1	X	0	0	10
ADDI	0011	NA	0	0	X	0	0	0	0	1	1	X	0	0	01
Load	0100	NA	0	0	0	0	1	0	0	1	1	X	0	0	01
Store	0101	NA	0	0	0	0	0	0	0	0	1	X	0	1	01
BEQ	0110'	NA	0	0	X	0	X	0	0	0	0	0	1	0	11
BNE	0111'	NA	0	0	X	0	X	0	0	0	0	1	1	0	11
FOR	1000'	NA	1	1	X	0	0	0	X	1	0	X	0	0	11
J-Type															
JMP	0001'	000'	0	0	1	1	X	X	X	0	X	X	0	0	X
CALL	0001'	001'	0	1	1	1	X	X	X	0	X	X	0	0	X
RET	0001'	010'	0	0	0	1	X	X	X	0	0	X	0	0	X

Here, we will control the ALU operations based on the specific requirements of each instruction. This is achieved using the ALU Decoder Unit, which determines the appropriate operation for the ALU according to the instruction's opcode and function field. The operations are defined in the following table for clarity.

Table 2: ALU Decoder Table

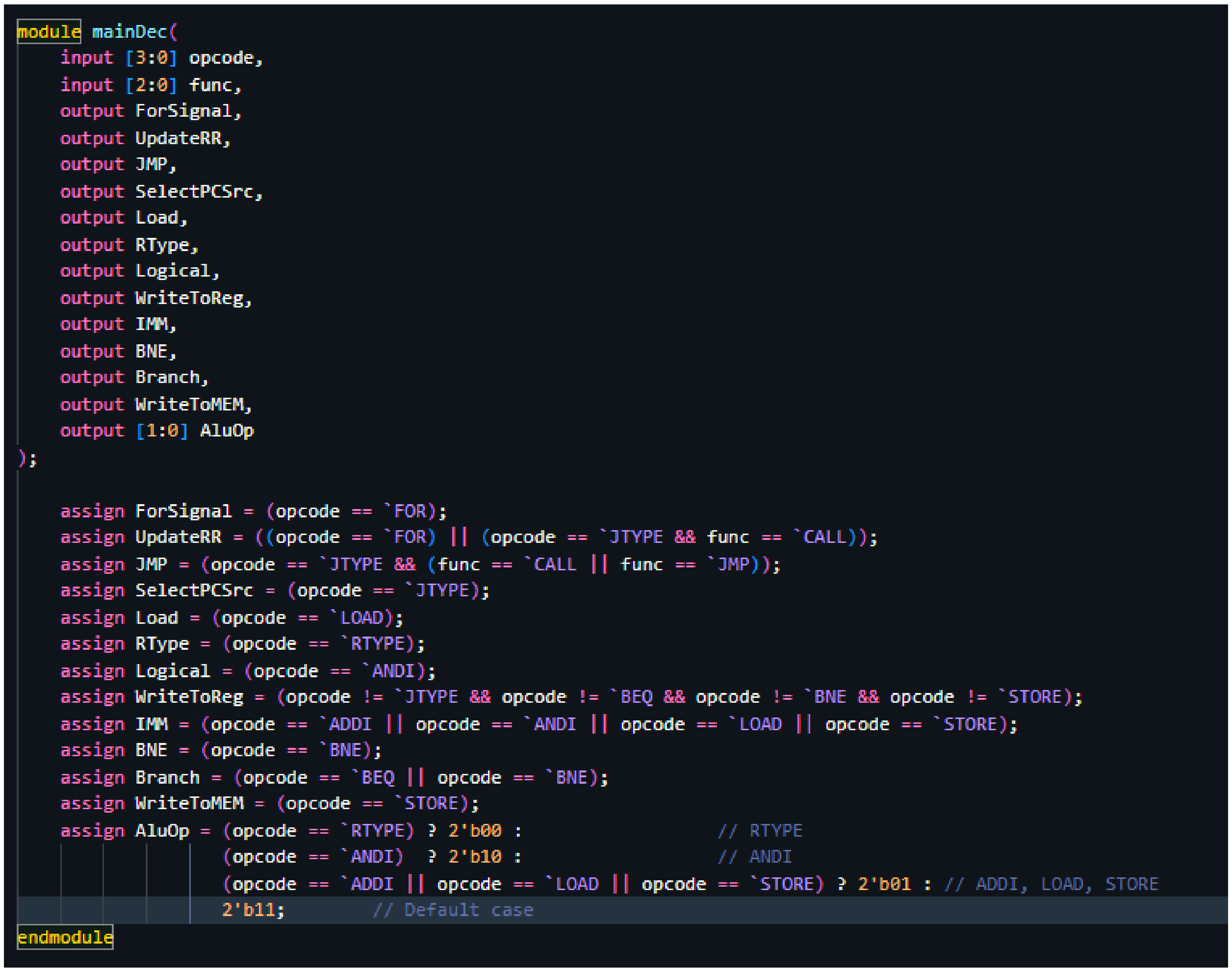
ALU OP	Function	ALU Control	Column 1
00	000	000	AND
00	001	001	ADD
00	010	010	SUB
00	011	011	SLL
00	100	100	SRL
10	XXX	000	AND
01	XXX	001	ADD
11	XXX	010	SUB

From the provided tables, we can derive the logical expressions or equations for each instruction and instruction type. These equations are implemented in the control unit of our processor to generate the necessary control signals. The following snapshots from our controller's Verilog code illustrate these expressions and their usage in detail.



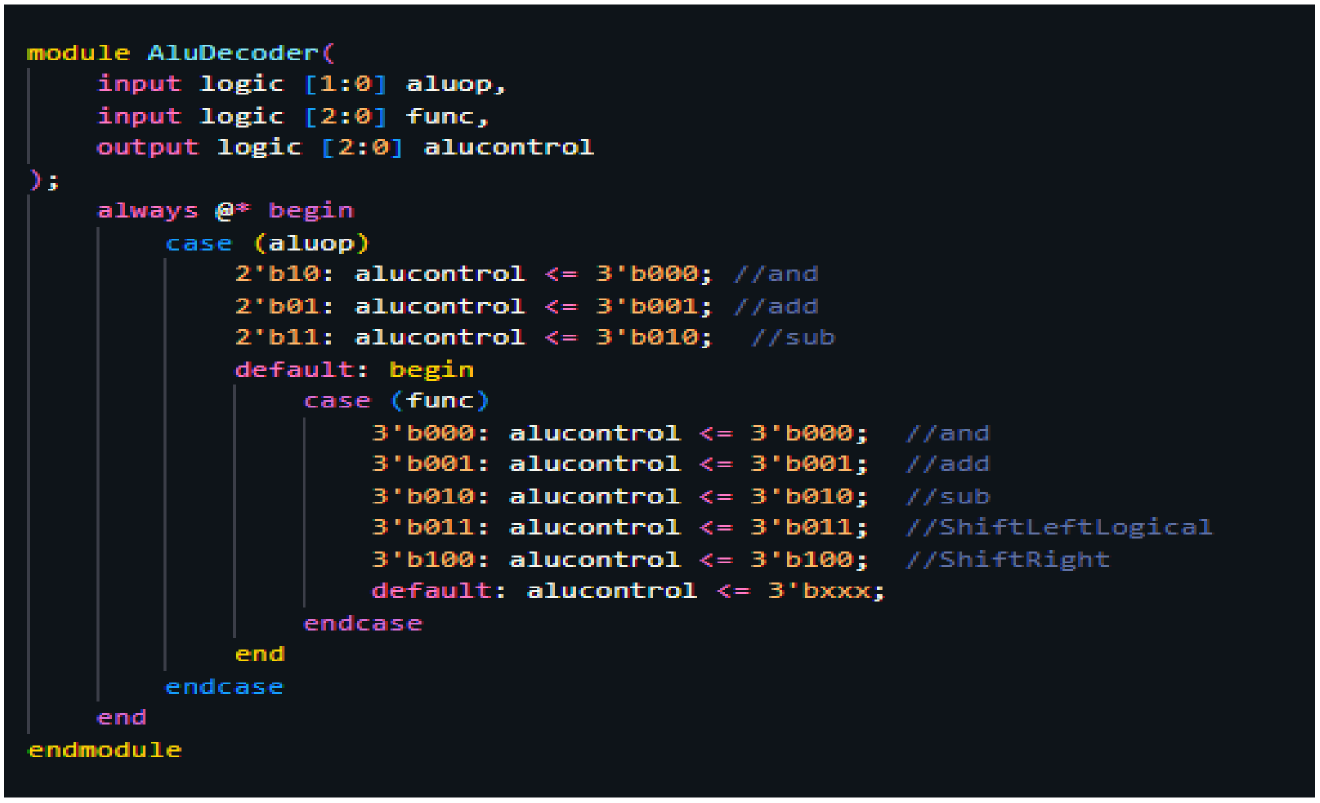
```
module controller(
    input logic [3:0] opcode,
    input [2:0] func,
    output ForSignal,
    output UpdateRR,
    output JMP,
    output SelectPCSrc,
    output Load,
    output RType,
    output Logical,
    output WriteToReg,
    output IMM,
    output BNE,
    output Branch,
    output WriteToMEM,
    output [2:0] AluControl);

    wire [1:0] aluop;
    mainDec main1(opcode, func, ForSignal, UpdateRR, JMP, SelectPCSrc, Load, RType, Logical, WriteToReg, IMM, BNE, Branch, WriteToMEM, aluop);
    AluDecoder alu1(aluop, func, AluControl);
```

Figure 7: Controller Module



```
module mainDec(
    input [3:0] opcode,
    input [2:0] func,
    output ForSignal,
    output UpdateRR,
    output JMP,
    output SelectPCSrc,
    output Load,
    output RType,
    output Logical,
    output WriteToReg,
    output IMM,
    output BNE,
    output Branch,
    output WriteToMEM,
    output [1:0] AluOp
);

    assign ForSignal = (opcode == `FOR);
    assign UpdateRR = ((opcode == `FOR) || (opcode == `JTYPE && func == `CALL));
    assign JMP = (opcode == `JTYPE && (func == `CALL || func == `JMP));
    assign SelectPCSrc = (opcode == `JTYPE);
    assign Load = (opcode == `LOAD);
    assign RType = (opcode == `RTYPE);
    assign Logical = (opcode == `ANDI);
    assign WriteToReg = (opcode != `JTYPE && opcode != `BEQ && opcode != `BNE && opcode != `STORE);
    assign IMM = (opcode == `ADDI || opcode == `ANDI || opcode == `LOAD || opcode == `STORE);
    assign BNE = (opcode == `BNE);
    assign Branch = (opcode == `BEQ || opcode == `BNE);
    assign WriteToMEM = (opcode == `STORE);
    assign AluOp = (opcode == `RTYPE) ? 2'b00 :                    // RTYPE
                   (opcode == `ANDI)  ? 2'b10 :                    // ANDI
                   (opcode == `ADDI || opcode == `LOAD || opcode == `STORE) ? 2'b01 : // ADDI, LOAD, STORE
                   2'b11;          // Default case
endmodule
```

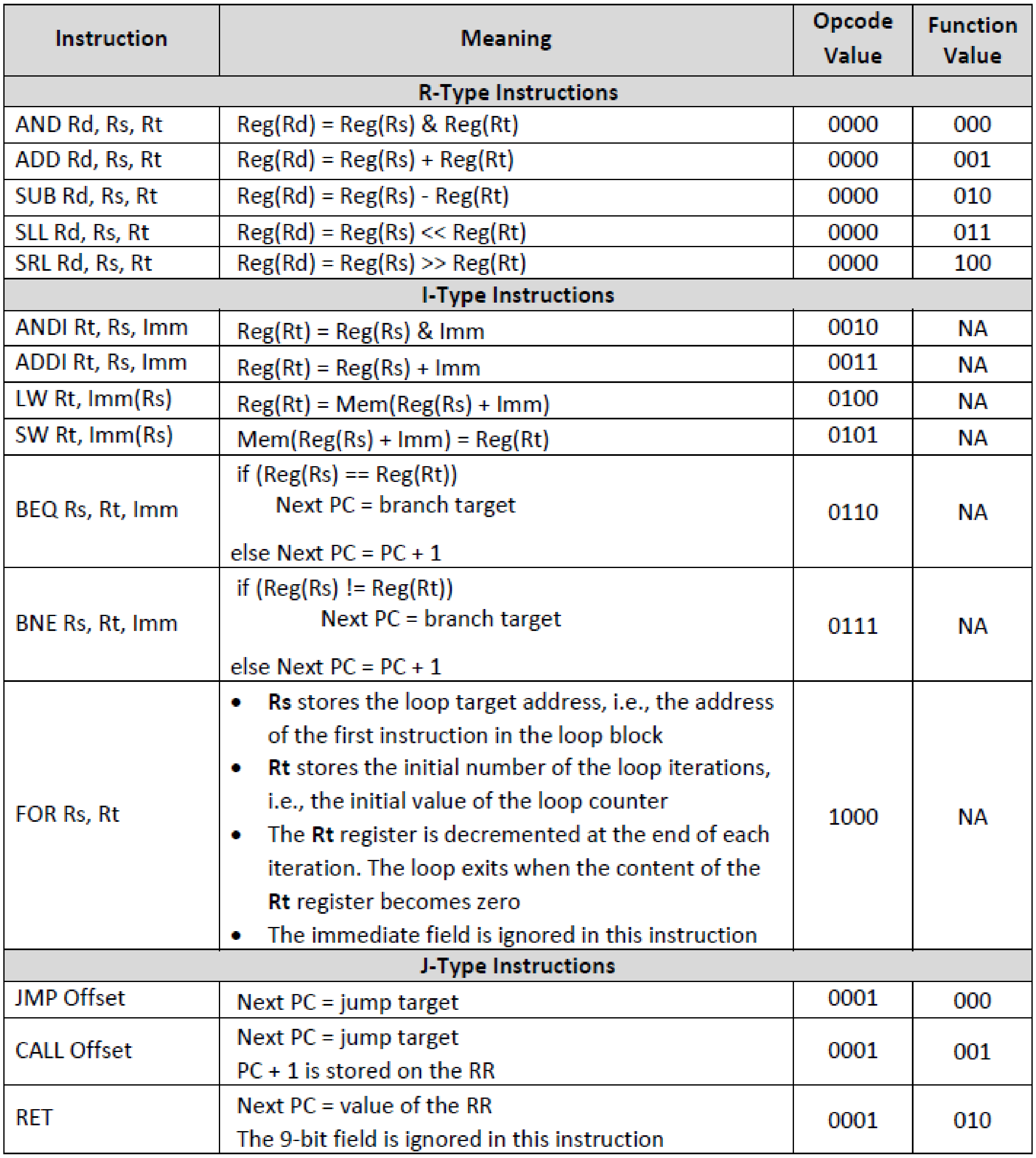
Figure 8: Main Decode Module



```
module AluDecoder(
    input logic [1:0] aluop,
    input logic [2:0] func,
    output logic [2:0] alucontrol
);
    always @* begin
        case (aluop)
            2'b10: alucontrol <= 3'b000; //and
            2'b01: alucontrol <= 3'b001; //add
            2'b11: alucontrol <= 3'b010;  //sub
            default: begin
                case (func)
                    3'b000: alucontrol <= 3'b000;  //and
                    3'b001: alucontrol <= 3'b001;  //add
                    3'b010: alucontrol <= 3'b010;  //sub
                    3'b011: alucontrol <= 3'b011;  //ShiftLeftLogical
                    3'b100: alucontrol <= 3'b100;  //ShiftRight
                    default: alucontrol <= 3'bxxx;
                endcase
            end
        endcase
    end
endmodule
```

Figure 9: ALU Decoder Module

In the previous figures, we have shown the expressions for generating each control signal and how they are derived. These signals are crucial for handling the supported instructions in our processor. Now, we will discuss the supported instructions and highlight the most important control signals required for each one.



Instruction	Meaning	Opcode Value	Function Value
R-Type Instructions			
AND Rd, Rs, Rt	Reg(Rd) = Reg(Rs) & Reg(Rt)	0000	000
ADD Rd, Rs, Rt	Reg(Rd) = Reg(Rs) + Reg(Rt)	0000	001
SUB Rd, Rs, Rt	Reg(Rd) = Reg(Rs) - Reg(Rt)	0000	010
SLL Rd, Rs, Rt	Reg(Rd) = Reg(Rs) << Reg(Rt)	0000	011
SRL Rd, Rs, Rt	Reg(Rd) = Reg(Rs) >> Reg(Rt)	0000	100
I-Type Instructions			
ANDI Rt, Rs, Imm	Reg(Rt) = Reg(Rs) & Imm	0010	NA
ADDI Rt, Rs, Imm	Reg(Rt) = Reg(Rs) + Imm	0011	NA
LW Rt, Imm(Rs)	Reg(Rt) = Mem(Reg(Rs) + Imm)	0100	NA
SW Rt, Imm(Rs)	Mem(Reg(Rs) + Imm) = Reg(Rt)	0101	NA

BEQ Rs, Rt, Imm	if (Reg(Rs) == Reg(Rt))
	Next PC = branch target
	else Next PC = PC + 1
		0110	NA

BNE Rs, Rt, Imm	if (Reg(Rs) != Reg(Rt))
	Next PC = branch target
	else Next PC = PC + 1
		0111	NA

FOR Rs, Rt	• **Rs** stores the loop target address, i.e., the address of the first instruction in the loop block • **Rt** stores the initial number of the loop iterations, i.e., the initial value of the loop counter • The **Rt** register is decremented at the end of each iteration. The loop exits when the content of the **Rt** register becomes zero • The immediate field is ignored in this instruction	1000	NA
J-Type Instructions			
JMP Offset	Next PC = jump target	0001	000
CALL Offset	Next PC = jump target PC + 1 is stored on the RR	0001	001
RET	Next PC = value of the RR The 9-bit field is ignored in this instruction	0001	010

Figure 10: Supported Instructions

In **R-Type** instructions, the key control signals include **RegWrite** (to enable writing to a register), **R-Type** (to indicate the instruction type), and the Function field (to specify the ALU operation for the instruction). These signals ensure correct handling of arithmetic and logical operations.

For **I-Type** instructions, all control signals are important, as each serves a specific purpose for different instructions:

- The **ForSignal** is used for the **FOR** instruction to manage loop operations.
- The **UpdateRR** signal updates the **Return Register (RR)** after completing loop iterations.
- The **MemWrite** signal enables memory writes for **SW (Store Word)** instructions.
- The **Branch** signal controls branch instructions by determining the **Program Counter (PC)** value.
- The **Immediate** signal, used in most **I-Type** instructions, is essential for operations involving immediate values. It is worth noting that the Function field is not utilized in **I-Type** instructions.

For **J-Type** instructions, the control logic involves fewer signals:

- The **UpdateRR** signal is used for **CALL** instructions to save the return address in the **Return Register**.
- The **PCSrc** signal controls the Program Counter (PC) source, enabling jumps to the correct address.
- The **JmpTgt** signal determines the jump target address, which is critical for handling jump instructions effectively.

Each instruction type uses a unique combination of control signals tailored to its specific requirements, ensuring efficient execution in the processor.

Implementation Procedure

Divide Pipeline into stages

The implementation process began by constructing a **single-cycle Datapath** processor as a foundation. This approach simplifies the development of the pipelined Datapath processor by dividing the single-cycle Datapath into five distinct stages: **Fetch, Decode, Execute, Memory, and Write Back**. To facilitate the transition between stages, pipeline registers were added to transfer data and control signals from one stage to the next. The following figure illustrates the single-cycle Datapath, which serves as the starting point for the pipelined implementation.

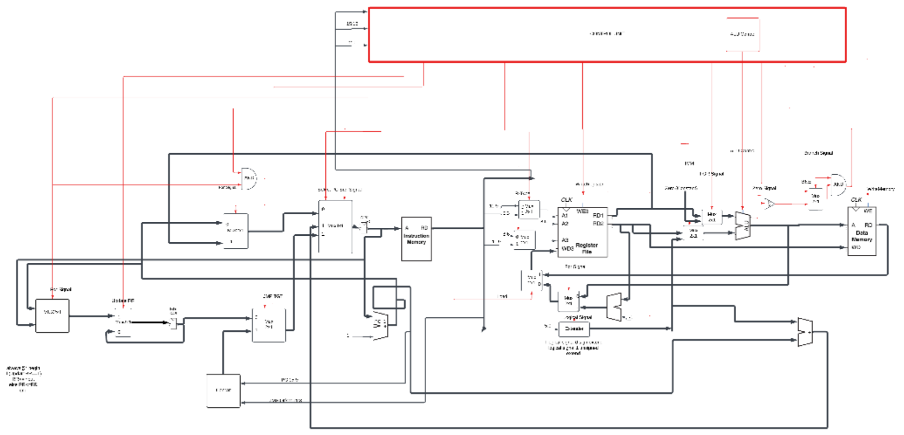


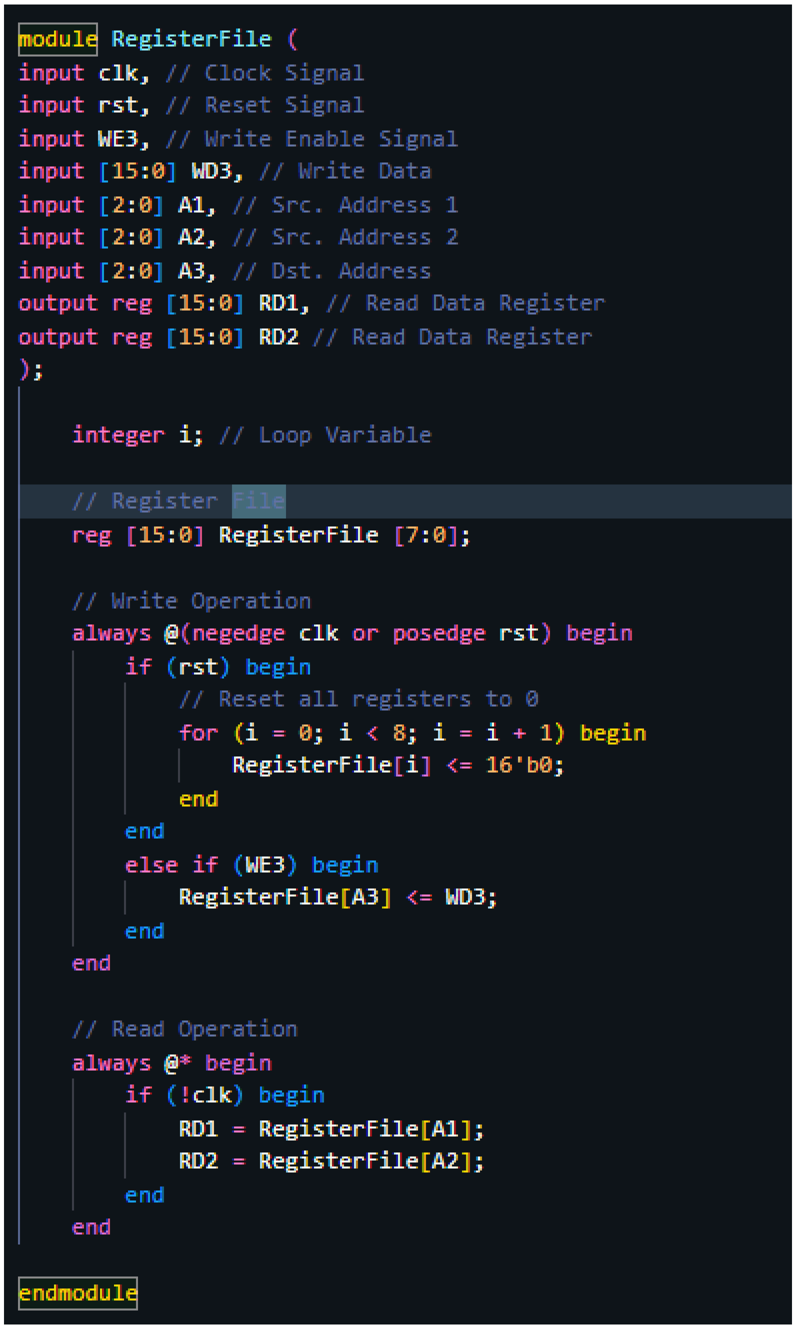
Figure 11: Single Cycle Datapath

Implementation of Main Components

After designing the Datapath, the required components for the processor were identified and implemented to support the logic of the instructions. Key components include:

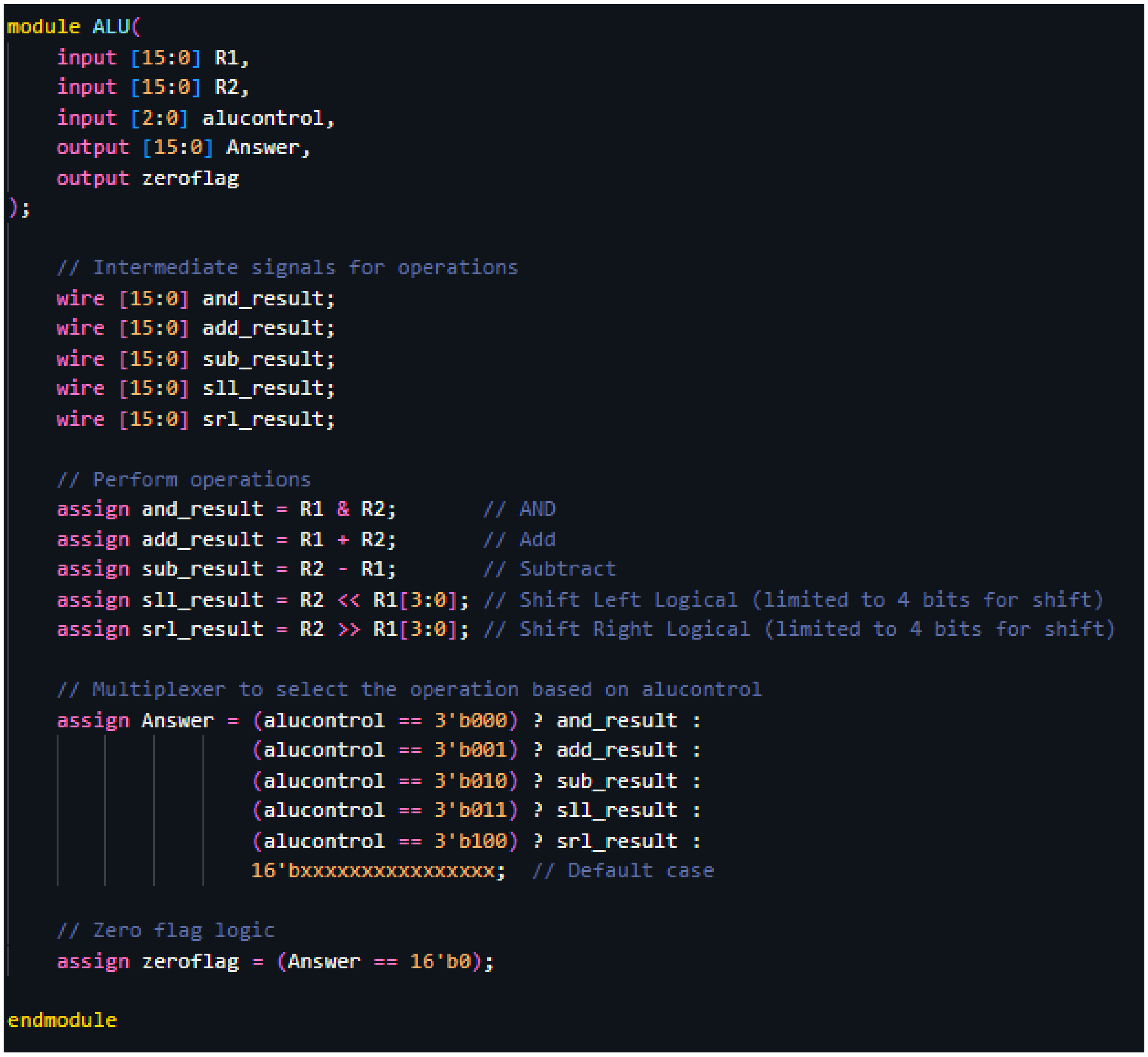
- ALU (Arithmetic and Logical Unit): Handles operations such as addition, subtraction, left and right shifts, and logical AND operations.
- Register File: Accepts three address inputs two for reading registers and one for writing back results.
- Adders: Two dedicated adders are included:
  1. One for calculating target addresses for branch instructions.
  2. Another for determining the number of iterations in the context of for loop instructions.
- Extender: Extends the immediate value, which is used in branch instructions to calculate the branch target based on the offset.
- Concatenation Block: Used in jump instructions (JMP) to concatenate the first 7 bits of the program counter (PC) with a 9-bit offset to compute the jump target address.
- Multiplexers and Logical Gates: Common components that route and control data flow within the processor.
- Instruction and Data Memories: These are kept separate to avoid structural hazards, enabling simultaneous access to instruction and data memories.

These components are designed to ensure compatibility with the processor's instruction logic while enabling efficient execution and effective hazard handling. The following figures showcase the Verilog implementation of the key components.



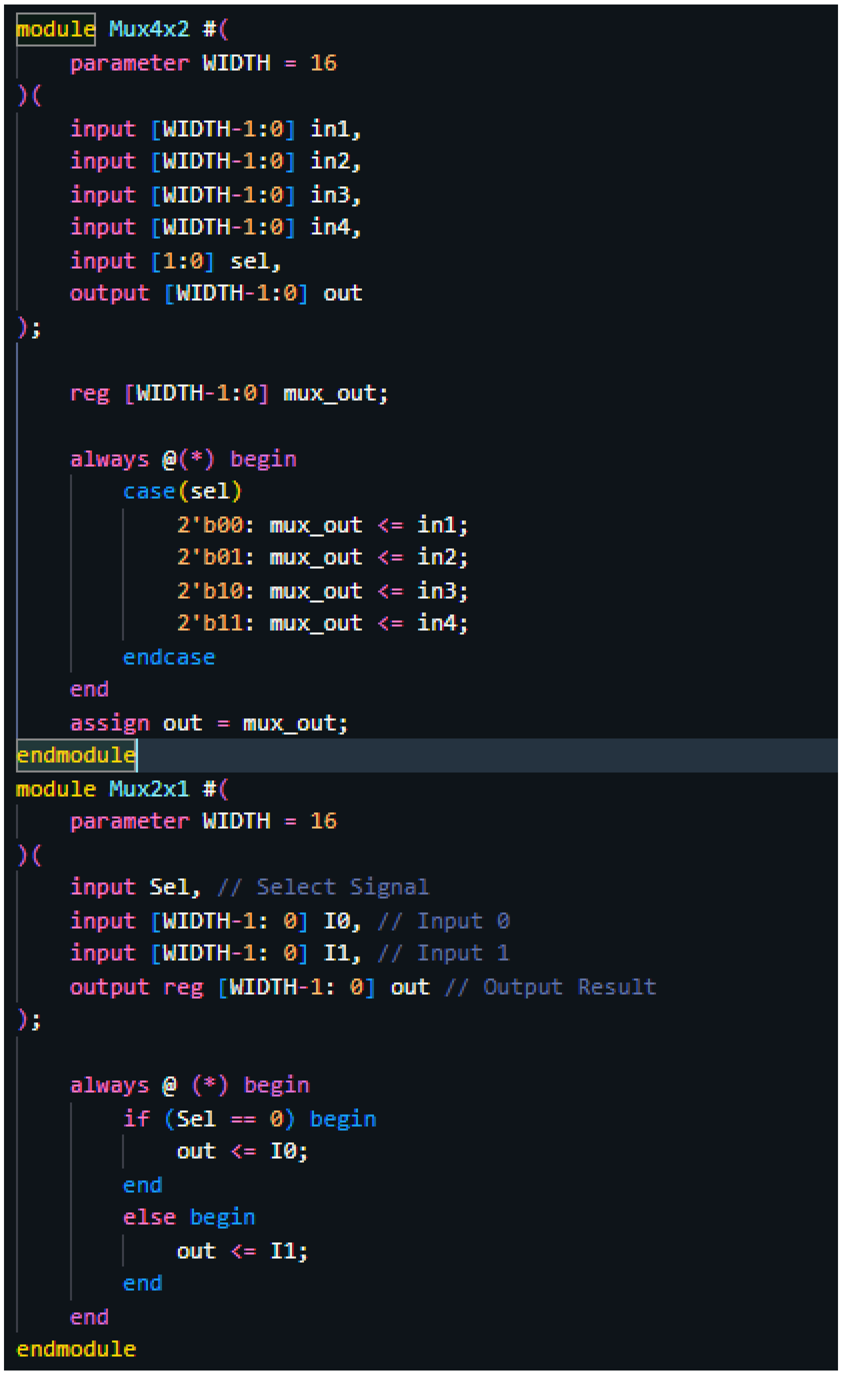
```
module RegisterFile (
input clk, // Clock Signal
input rst, // Reset Signal
input WE3, // Write Enable Signal
input [15:0] WD3, // Write Data
input [2:0] A1, // Src. Address 1
input [2:0] A2, // Src. Address 2
input [2:0] A3, // Dst. Address
output reg [15:0] RD1, // Read Data Register
output reg [15:0] RD2 // Read Data Register
);

    integer i; // Loop Variable

    // Register File
    reg [15:0] RegisterFile [7:0];

    // Write Operation
    always @(negedge clk or posedge rst) begin
        if (rst) begin
            // Reset all registers to 0
            for (i = 0; i < 8; i = i + 1) begin
                RegisterFile[i] <= 16'b0;
            end
        end
        else if (WE3) begin
            RegisterFile[A3] <= WD3;
        end
    end

    // Read Operation
    always @* begin
        if (!clk) begin
            RD1 = RegisterFile[A1];
            RD2 = RegisterFile[A2];
        end
    end

endmodule
```

Figure 12: Register File Implementation



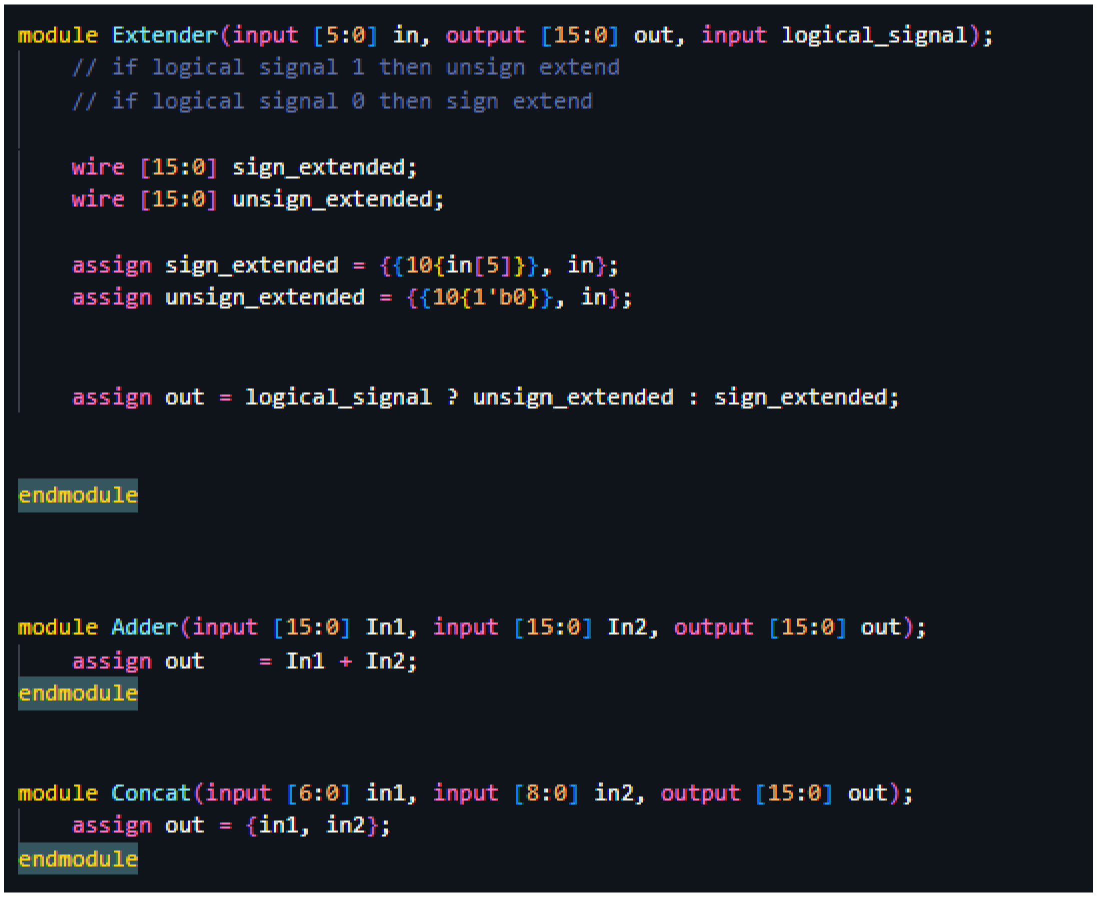
```verilog
module ALU(
    input [15:0] R1,
    input [15:0] R2,
    input [2:0] alucontrol,
    output [15:0] Answer,
    output zeroflag
);

    // Intermediate signals for operations
    wire [15:0] and_result;
    wire [15:0] add_result;
    wire [15:0] sub_result;
    wire [15:0] sll_result;
    wire [15:0] srl_result;

    // Perform operations
    assign and_result = R1 & R2;       // AND
    assign add_result = R1 + R2;       // Add
    assign sub_result = R2 - R1;       // Subtract
    assign sll_result = R2 << R1[3:0]; // Shift Left Logical (limited to 4 bits for shift)
    assign srl_result = R2 >> R1[3:0]; // Shift Right Logical (limited to 4 bits for shift)

    // Multiplexer to select the operation based on alucontrol
    assign Answer = (alucontrol == 3'b000) ? and_result :
                    (alucontrol == 3'b001) ? add_result :
                    (alucontrol == 3'b010) ? sub_result :
                    (alucontrol == 3'b011) ? sll_result :
                    (alucontrol == 3'b100) ? srl_result :
                    16'bxxxxxxxxxxxxxxxx;  // Default case

    // Zero flag logic
    assign zeroflag = (Answer == 16'b0);

endmodule
```

Figure 13: ALU Implementation



```verilog
module Mux4x2 #(
    parameter WIDTH = 16
)(
    input [WIDTH-1:0] in1,
    input [WIDTH-1:0] in2,
    input [WIDTH-1:0] in3,
    input [WIDTH-1:0] in4,
    input [1:0] sel,
    output [WIDTH-1:0] out
);

    reg [WIDTH-1:0] mux_out;

    always @(*) begin
        case(sel)
            2'b00: mux_out <= in1;
            2'b01: mux_out <= in2;
            2'b10: mux_out <= in3;
            2'b11: mux_out <= in4;
        endcase
    end
    assign out = mux_out;
endmodule
module Mux2x1 #(
    parameter WIDTH = 16
)(
    input Sel, // Select Signal
    input [WIDTH-1: 0] I0, // Input 0
    input [WIDTH-1: 0] I1, // Input 1
    output reg [WIDTH-1: 0] out // Output Result
);

    always @ (*) begin
        if (Sel == 0) begin
            out <= I0;
        end
        else begin
            out <= I1;
        end
    end
endmodule
```

Figure 14: Multiplexers Implementation



```
module Extender(input [5:0] in, output [15:0] out, input logical_signal);
    // if logical signal 1 then unsign extend
    // if logical signal 0 then sign extend

    wire [15:0] sign_extended;
    wire [15:0] unsign_extended;

    assign sign_extended = {{10{in[5]}}, in};
    assign unsign_extended = {{10{1'b0}}, in};


    assign out = logical_signal ? unsign_extended : sign_extended;


endmodule



module Adder(input [15:0] In1, input [15:0] In2, output [15:0] out);
    assign out    = In1 + In2;
endmodule


module Concat(input [6:0] in1, input [8:0] in2, output [15:0] out);
    assign out = {in1, in2};
endmodule
```

Figure 15: Extender, Adder, Concater Implementation

Design Hazard Detection

To ensure the proper functionality of the pipelined processor, a mechanism is required to detect and handle hazards effectively. This is achieved through the implementation of a Hazard Unit, a critical component of the processor. The Hazard Unit detects potential hazards based on specific conditions and utilizes **forwarding** to resolve data **dependencies**, ensuring smooth execution.

The Hazard Unit generates expressions to detect and handle different types of hazards, such as data and control hazards, and takes the necessary control signals to manage **stalls** and resolve conflicts at the right time. Additionally, it manages data **forwarding** between pipeline stages, improving efficiency by minimizing stalls whenever possible.

In the following sections, we will discuss the **hazard** detection expressions, how **stalls** are generated when necessary, and the role of the Hazard Unit in facilitating data forwarding to enhance the overall performance of the processor.

Complete Processor Datapath

After implementing the hazard detection and handling mechanisms, as well as designing the required components, the next step is to integrate all components into a cohesive system. This integration ensures that data and control signals flow seamlessly from one stage to the next across the pipeline. The full Datapath for the pipelined processor, which includes all stages and their interconnections, is illustrated in Figure 7: Pipelined Processor Complete Datapath. This figure demonstrates how the components work together to execute instructions efficiently while managing hazards effectively.

Block Diagrams

In this section, we will present the block diagrams with a closer and more detailed view (zoomed in) to highlight specific components and their interconnections.

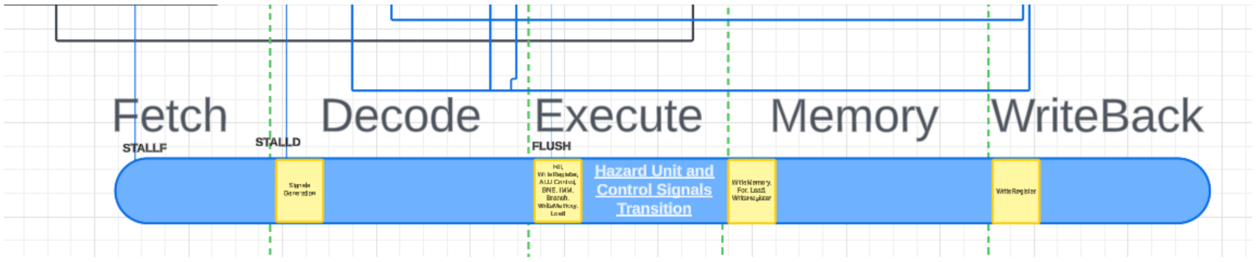


Figure 16: Hazards Unit

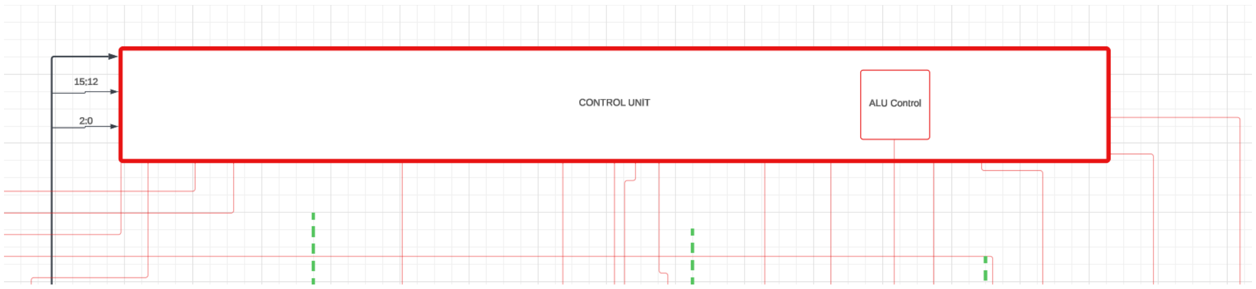


Figure 17: Control Unit

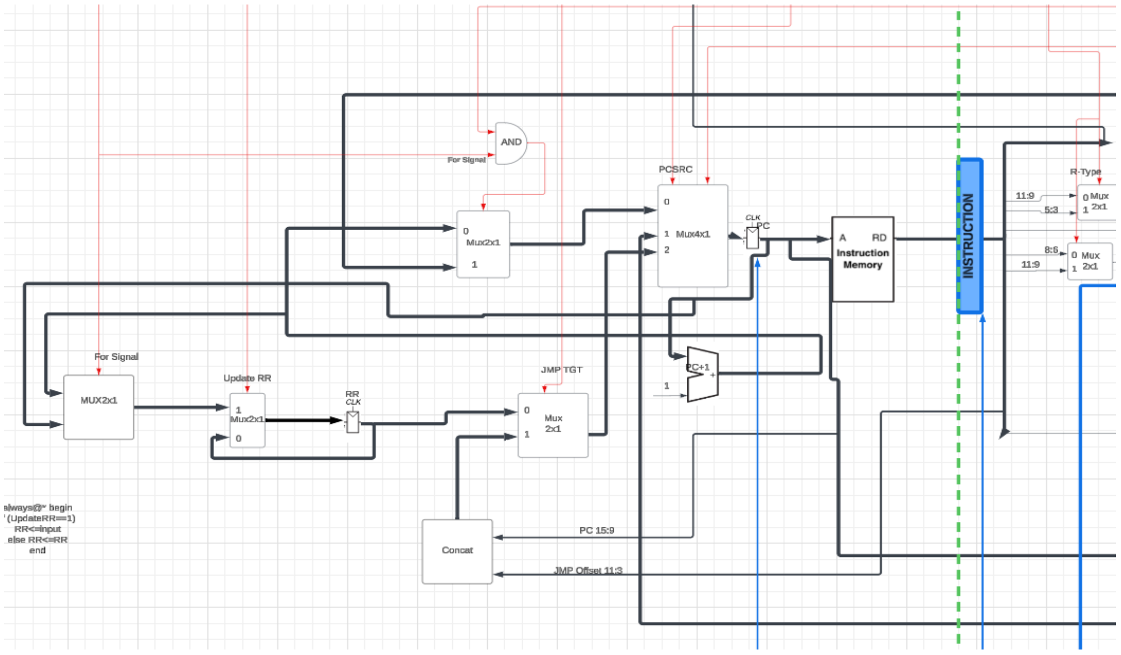


Figure 18: PC Control Logic

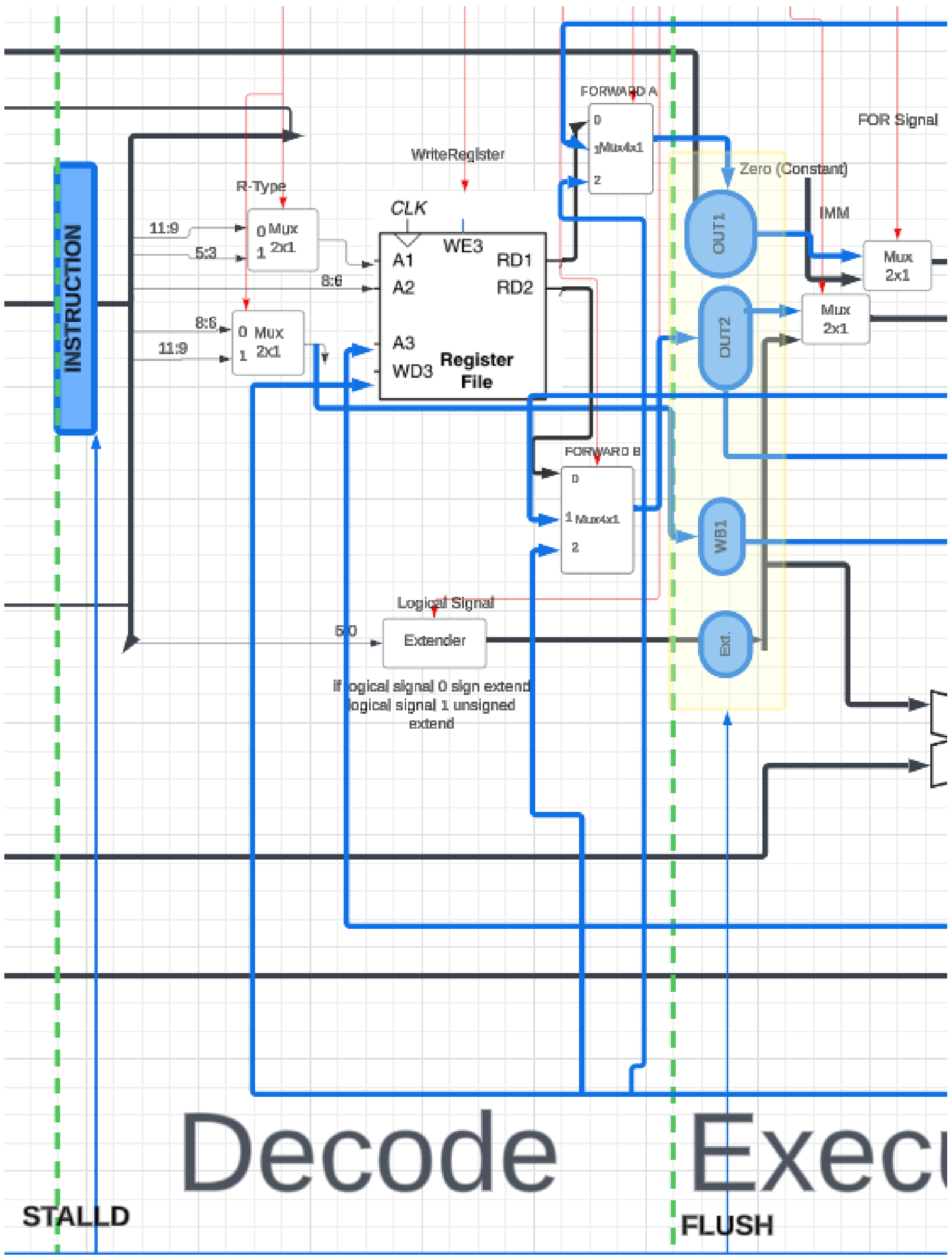
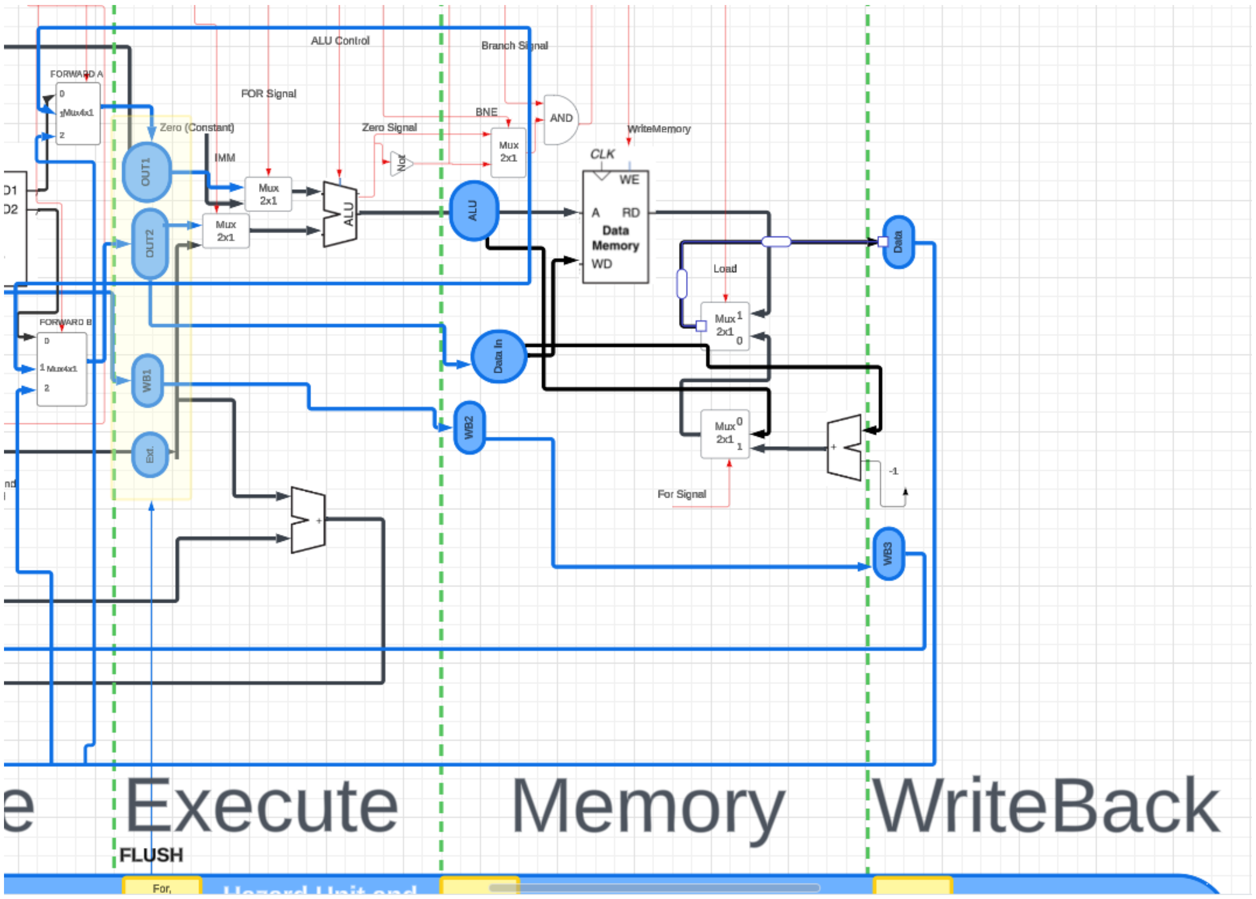


Figure 19: Decode Stage



Design Justification

As detailed in the previous sections, we now discuss the rationale behind our design and implementation choices, focusing on decisions made to improve efficiency and minimize hazards.

1. Forwarding Mechanism
   To handle dependencies between instructions efficiently, we implemented a **forwarding** mechanism between **pipeline stages**. This choice minimizes the need for **stall** cycles, which would otherwise delay the processor by waiting for the previous instruction to complete. By **forwarding** data directly from later stages (e.g., Execute or Memory) to earlier stages (e.g., Decode), we reduce stalls and improve overall **performance**.

2. Handling **RAW** (Read After Write) Hazards
   Instead of stalling the pipeline to address RAW hazards, we adopted a more efficient approach:

   - The write operation is performed in the **first half of the clock cycle**.
   - The read operation is performed in the **second half of the clock cycle**. This ensures that the required data is available for the dependent instruction within the

same cycle, eliminating unnecessary delays and improving processor speed and efficiency.

3. For Loop Instruction
   We designed the **FOR** instruction to streamline loop execution:
   - The processor jumps to the target address if the iteration count is non-zero.
   - Once the iterations are complete, it returns to the base address using the **RET** (Return) instruction.

   This design allows programmers to use loops flexibly, jumping to any address and efficiently tracking iterations using the return register (**RR**). The use of **RET** makes the iteration process straightforward and programmer-friendly.
4. Pipeline Registers Between Stages
   Between each pipeline stage, we included **pipeline registers** to store all essential control signals and data:
   - These registers ensure that each stage has the required information to execute its operation without conflict.
   - The design also facilitates the implementation of **flush** and **stall** signals, as these inputs can easily be added to the pipeline registers to control data flow during hazards or branch mispredictions.

Simulation & Testing

Simulation Setup

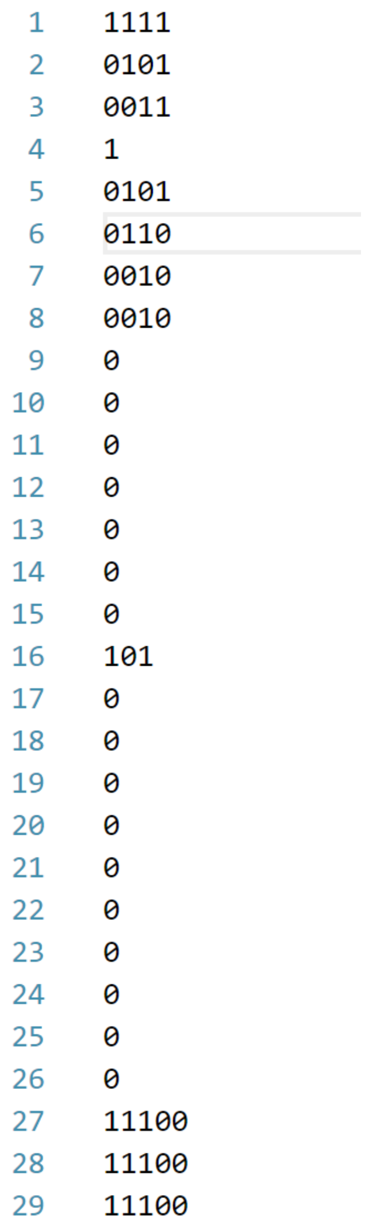
To simulate and test our processor, we utilized a professional and efficient toolchain. **The Icarus Verilog (iverilog)** compiler was used to compile the **Verilog** code, and **GTKWave** was employed to visualize the waveform outputs for detailed analysis. The entire setup was implemented on a **WSL (Windows Subsystem for Linux)** environment, providing a robust and Unix-like platform for development. Additionally, **Visual Studio Code (VS Code)** served as the primary IDE, offering an intuitive and feature-rich workspace for writing, simulating, and debugging Verilog code.

This setup ensured a streamlined workflow for simulation and testing, providing a reliable and effective environment for designing and verifying our processor. The combination of tools allowed us to perform precise debugging, analyze signals across pipeline stages, and validate the correctness of the processor's functionality under various scenarios.

Test Programs

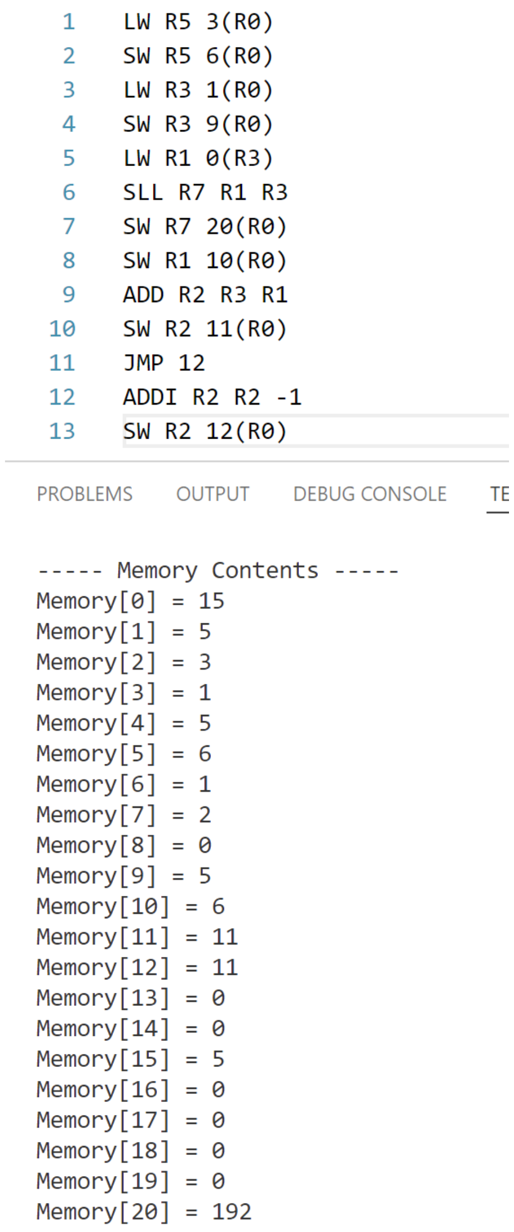
In this section, we present test programs written in **assembly language** to evaluate the functionality of our processor. The results are validated by observing and printing the values of memory cells at various stages of execution. Additionally, the **assembly** instructions are converted into **binary** format using a custom **Python script**. This automated conversion process is seamlessly integrated into our workflow, providing a professional and efficient approach to **testing** and **implementation**.

The test programs are designed to cover a wide range of scenarios, including arithmetic operations, memory access, branching, and loops, ensuring comprehensive verification of the processor's capabilities. The combination of assembly-to-binary conversion and memory state analysis ensures accurate validation of the processor's functionality.



```
1111
0101
0011
1
0101
0110
0010
0010
0
0
0
0
0
0
0
101
0
0
0
0
0
0
0
0
0
0
11100
11100
11100
```

Figure 20: Initial Memory Data



```
LW R5 3(R0)
SW R5 6(R0)
LW R3 1(R0)
SW R3 9(R0)
LW R1 0(R3)
SLL R7 R1 R3
SW R7 20(R0)
SW R1 10(R0)
ADD R2 R3 R1
SW R2 11(R0)
JMP 12
ADDI R2 R2 -1
SW R2 12(R0)
```

```
----- Memory Contents -----
Memory[0] = 15
Memory[1] = 5
Memory[2] = 3
Memory[3] = 1
Memory[4] = 5
Memory[5] = 6
Memory[6] = 1
Memory[7] = 2
Memory[8] = 0
Memory[9] = 5
Memory[10] = 6
Memory[11] = 11
Memory[12] = 11
Memory[13] = 0
Memory[14] = 0
Memory[15] = 5
Memory[16] = 0
Memory[17] = 0
Memory[18] = 0
Memory[19] = 0
Memory[20] = 192
```

Figure 21: Assembly Code

After executing the given instructions, the memory state is as follows:

Load R5 with 1 and store it into memory location 6:

R5 = 1, and after execution, memory location 6 also holds the value 1.
Load R3 with the value at memory location 6 (which is 1) and store it into memory location 9:

R3 = 5, and memory location 9 now contains the value 5.
Load R1 with the value at the location indicated by R3 (memory location 6, which holds 5):

R1 = 6.
Shift Left R1 by the value in R3 and store the result in R7:

Shifting the value in R1 (6) left by 3 positions gives R7 = 192.
Add R3 and R1, storing the result in R2:

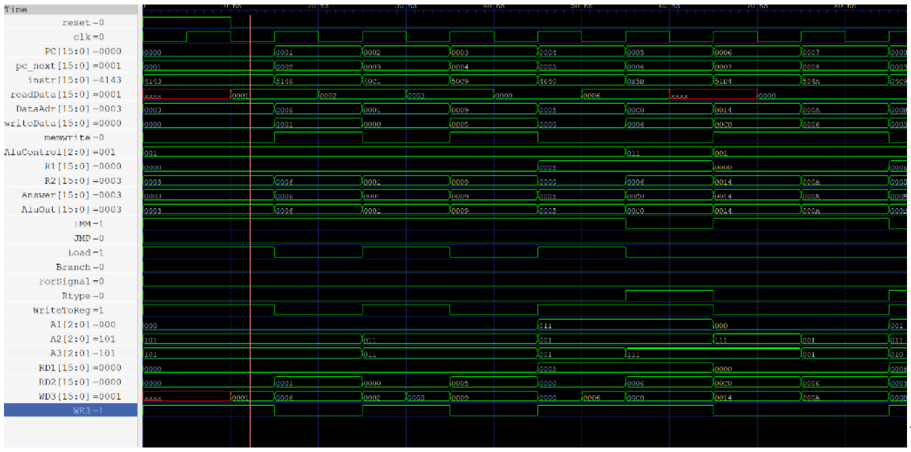
R2 = R3 + R1 = 5 + 6 = 11.
This value can be observed in memory location 11.
Jump to address 12:

The jump skips the instruction at address 12. This happens because the memory starts counting from 1 instead of 0, so the expected ADDI instruction at address 12 is skipped.
Store the value of R2 into memory location 12:

R2 = 11, and this value is stored in memory location 12, as the R2 value was not modified further.

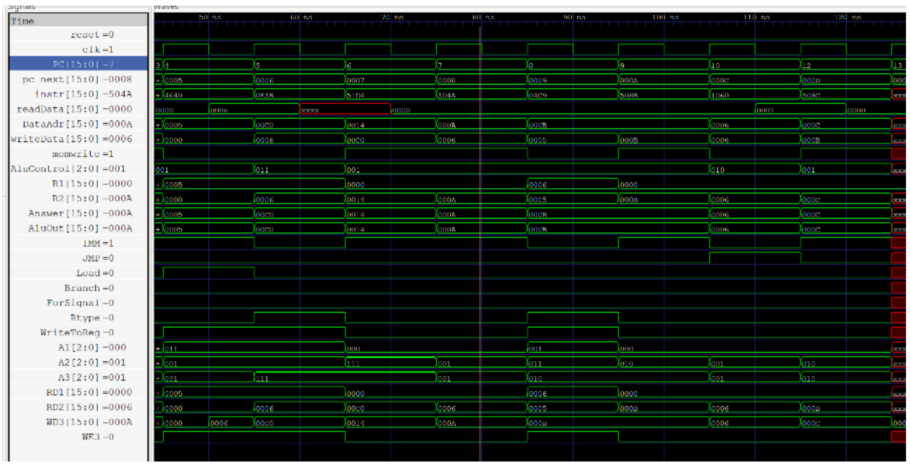
Single Cycle Waves

Each Instruction takes a whole clock!



```
Time
reset =0
clk =0
PC[15:0] =0000
pc_next[15:0] =0001
instr[15:0] =4143
readData[15:0] =0001
DataAdr[15:0] =0003
writeData[15:0] =0000
memwrite =0
AluControl[2:0] =001
R1[15:0] =0000
R2[15:0] =0003
Answer[15:0] =0003
AluOut[15:0] =0003
IMM =1
JMP =0
Load =1
Branch =0
ForSignal =0
Rtype =0
WriteToReg =1
A1[2:0] =000
A2[2:0] =101
A3[2:0] =101
RD1[15:0] =0000
RD2[15:0] =0000
WD3[15:0] =0001
WE3 =1
```

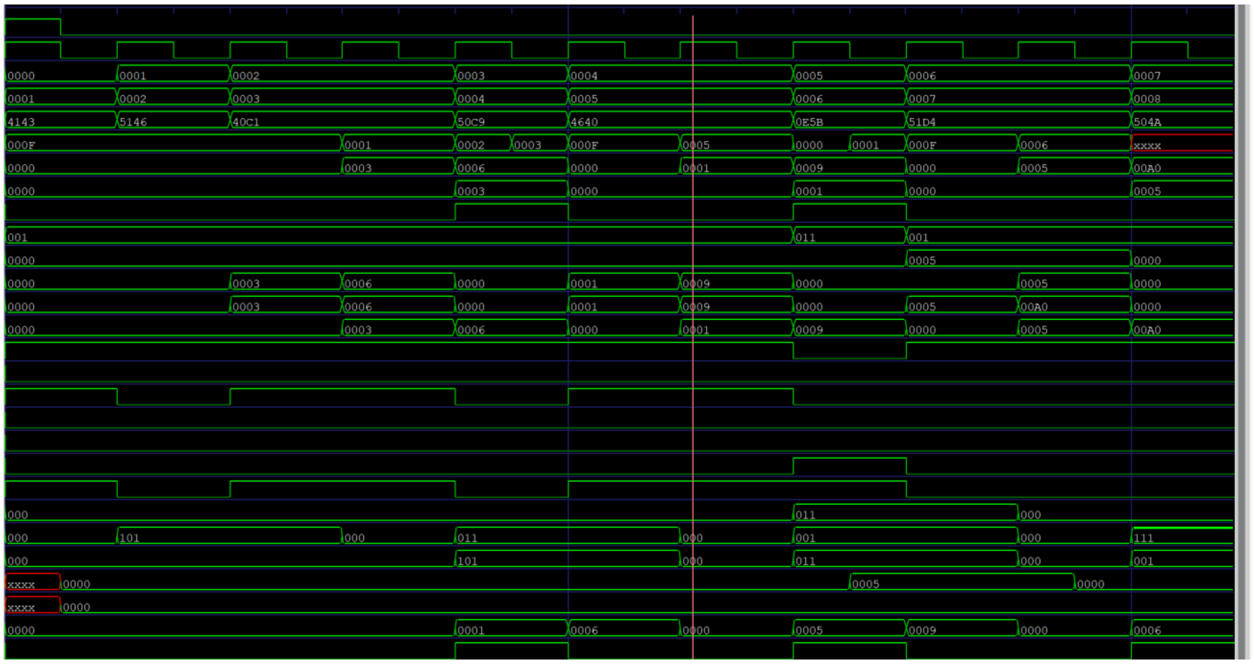
`

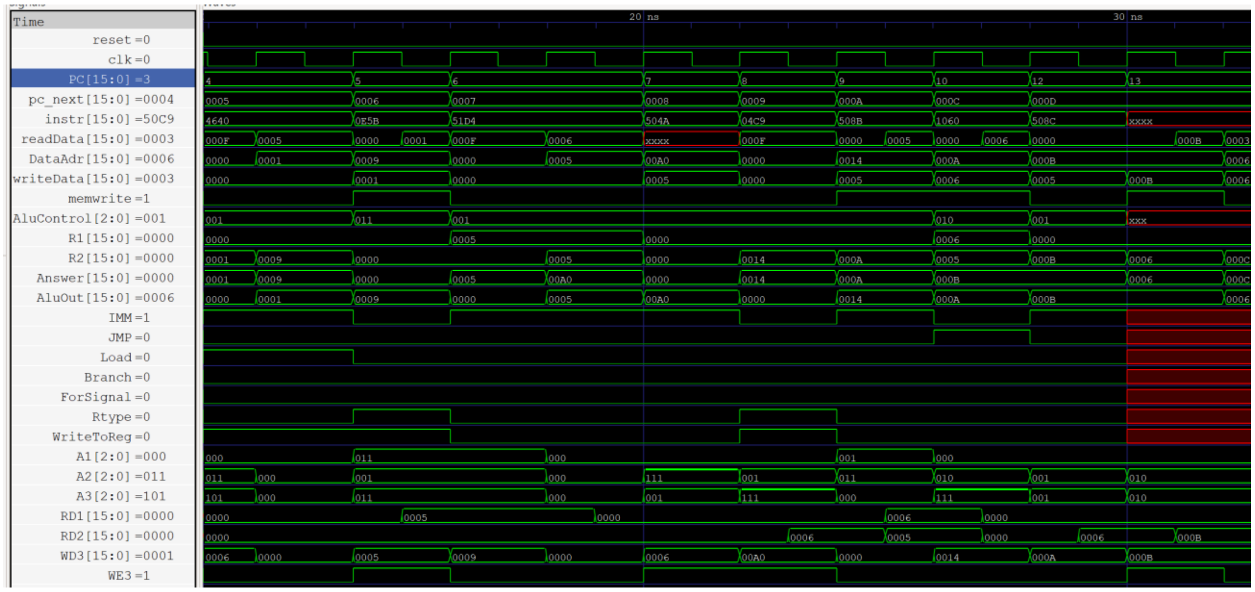


```
Signals
Time
reset =0
clk =1
PC[15:0] =7
pc_next[15:0] =0008
instr[15:0] =504A
readData[15:0] =0000
DataAdr[15:0] =000A
writeData[15:0] =0006
memwrite =1
AluControl[2:0] =001
R1[15:0] =0000
R2[15:0] =000A
Answer[15:0] =000A
AluOut[15:0] =000A
IMM =1
JMP =0
Load =0
Branch =0
ForSignal =0
Rtype =0
WriteToReg =0
A1[2:0] =000
A2[2:0] =001
A3[2:0] =001
RD1[15:0] =0000
RD2[15:0] =0006
WD3[15:0] =000A
WE3 =0
```

Pipelined Waves

In pipelining we observe the same answers but the wave looks different since instead of an instruction taking a whole cycle a stage is a cycle





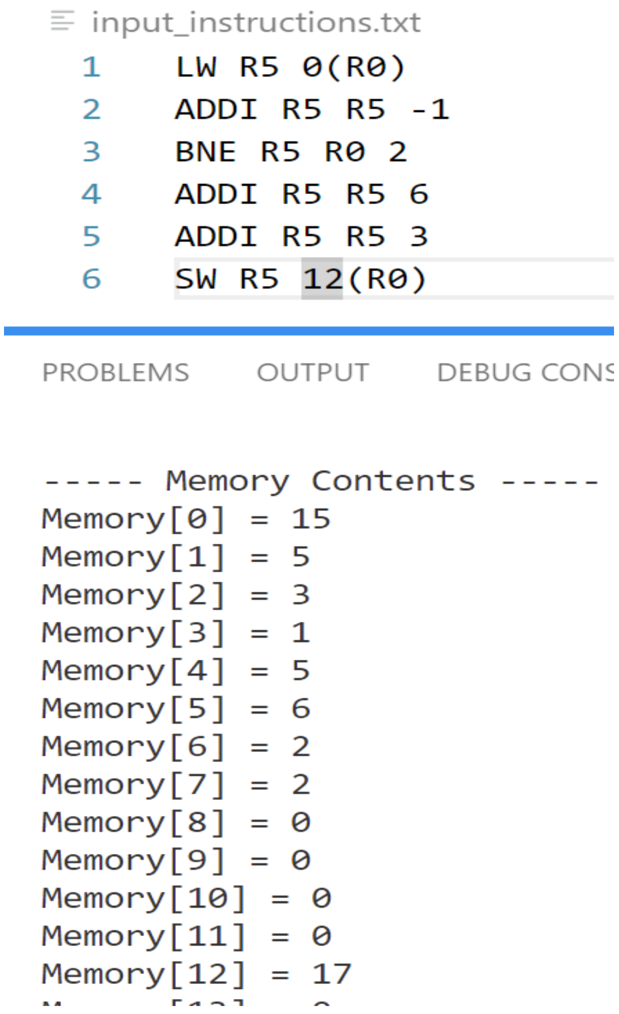
```
Time
reset =0
clk =0
PC[15:0] =3
pc_next[15:0] =0004
instr[15:0] =50C9
readData[15:0] =0003
DataAdr[15:0] =0006
writeData[15:0] =0003
memwrite =1
AluControl[2:0] =001
R1[15:0] =0000
R2[15:0] =0000
Answer[15:0] =0000
AluOut[15:0] =0006
IMM =1
JMP =0
Load =0
Branch =0
ForSignal =0
Rtype =0
WriteToReg =0
A1[2:0] =000
A2[2:0] =011
A3[2:0] =101
RD1[15:0] =0000
RD2[15:0] =0000
WD3[15:0] =0001
WE3 =1
```

Notice at the PC it goes from instruction 10 to 12 PC + 2

Notice the readData only happens on the second half of the cycle when the write is complete!

Notice at PC = 5 we have one stall cycle for the load

Testing Branch on Pipelined Processor



input_instructions.txt

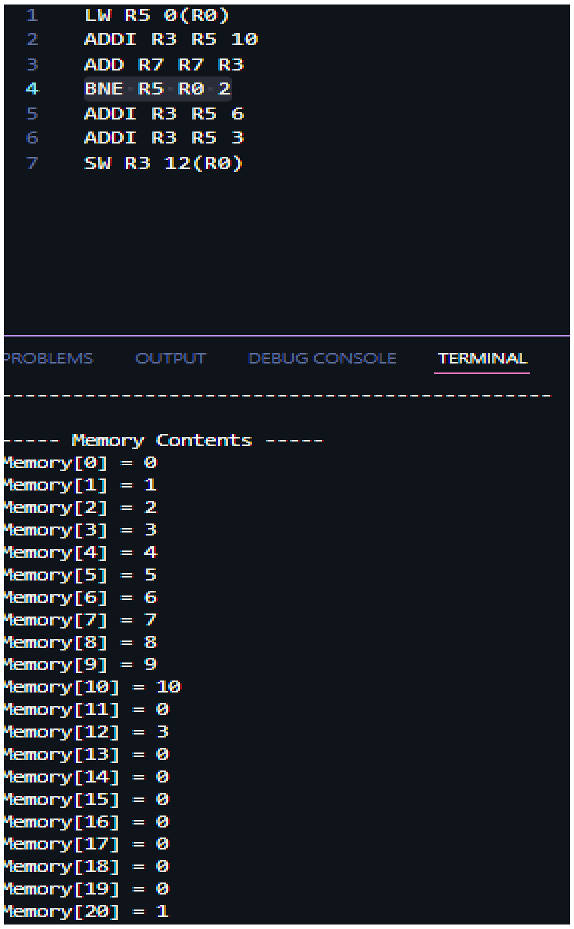
```
1    LW R5 0(R0)
2    ADDI R5 R5 -1
3    BNE R5 R0 2
4    ADDI R5 R5 6
5    ADDI R5 R5 3
6    SW R5 12(R0)
```

PROBLEMS OUTPUT DEBUG CONS

```
----- Memory Contents -----
Memory[0] = 15
Memory[1] = 5
Memory[2] = 3
Memory[3] = 1
Memory[4] = 5
Memory[5] = 6
Memory[6] = 2
Memory[7] = 2
Memory[8] = 0
Memory[9] = 0
Memory[10] = 0
Memory[11] = 0
Memory[12] = 17
```

Figure 22: Single Cycle Branch

1. R5 = 15
2. R5 –1 = 14
3. If R5 != 0 Branch offset = 2 PC + 2
4. Skip
5. R5 + 3 = 17
6. Store in 12 the value 7



```
LW R5 0(R0)
ADDI R3 R5 10
ADD R7 R7 R3
BNE R5 R0 2
ADDI R3 R5 6
ADDI R3 R5 3
SW R3 12(R0)
```

```
--------------------------------------------

----- Memory Contents -----
emory[0] = 0
emory[1] = 1
emory[2] = 2
emory[3] = 3
emory[4] = 4
emory[5] = 5
emory[6] = 6
emory[7] = 7
emory[8] = 8
emory[9] = 9
emory[10] = 10
emory[11] = 0
emory[12] = 3
emory[13] = 0
emory[14] = 0
emory[15] = 0
emory[16] = 0
emory[17] = 0
emory[18] = 0
emory[19] = 0
emory[20] = 1
```

Figure 23: Pipelined Branch

The same of previous test, but on new different data.

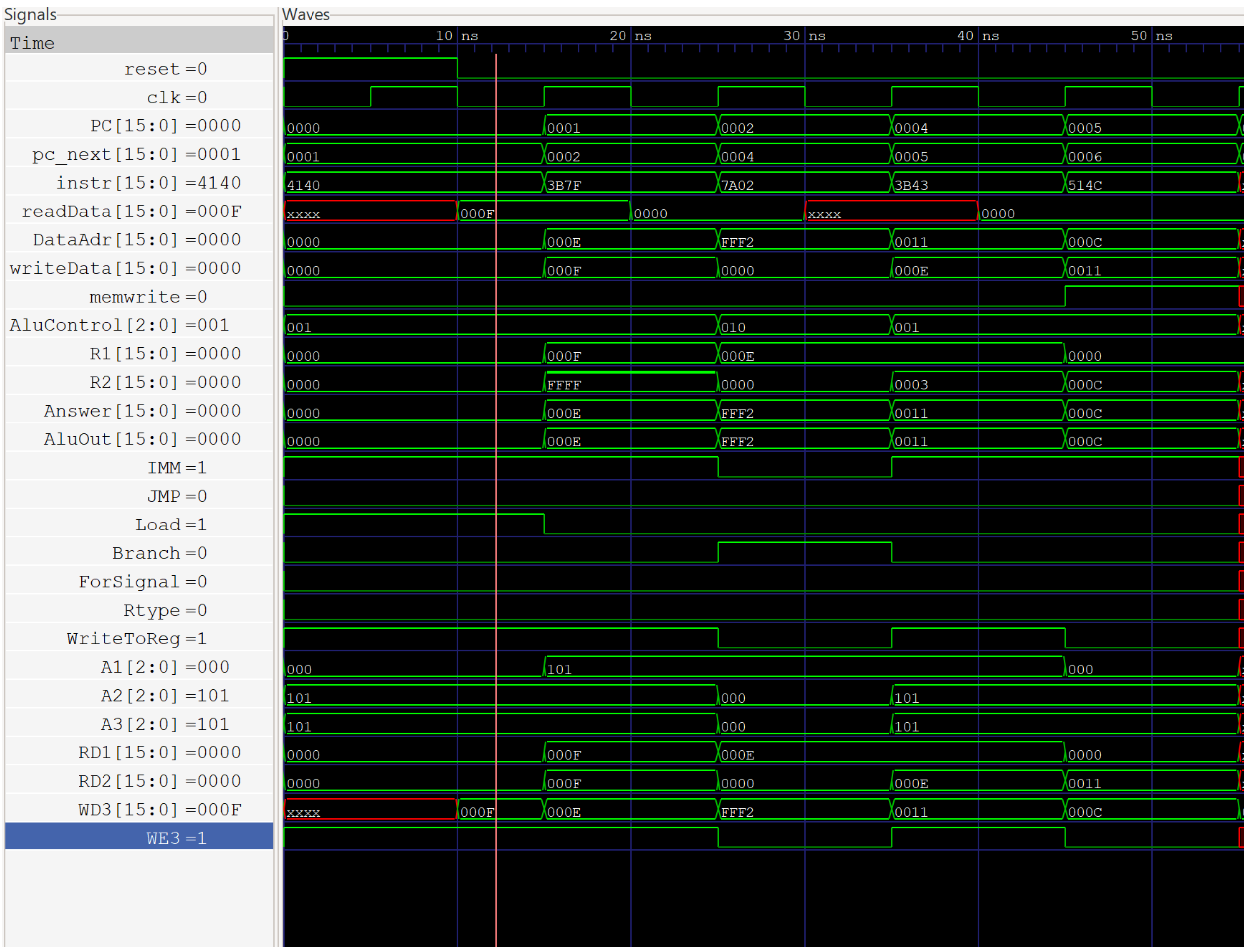
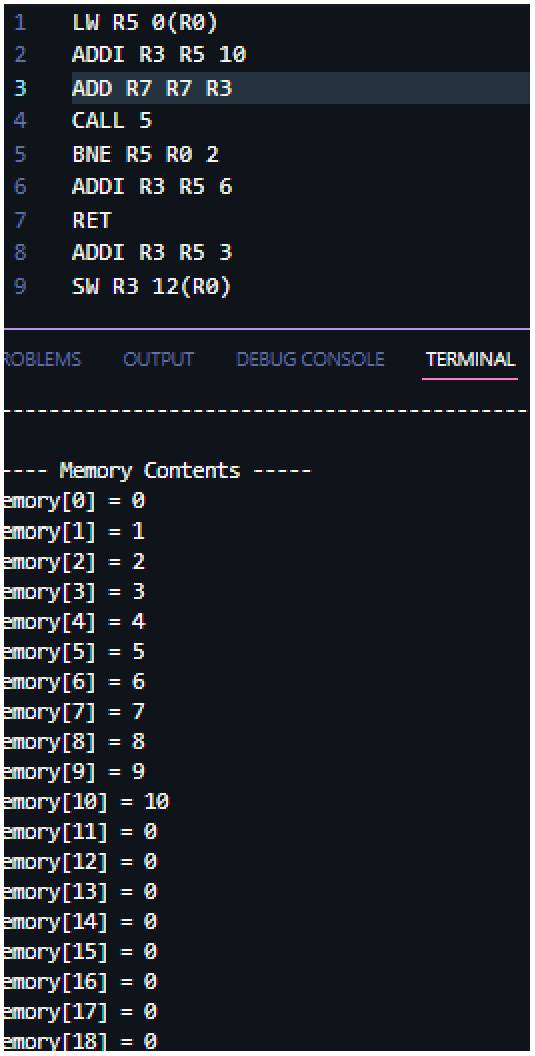


Figure 24: Pipelined Testing Waves

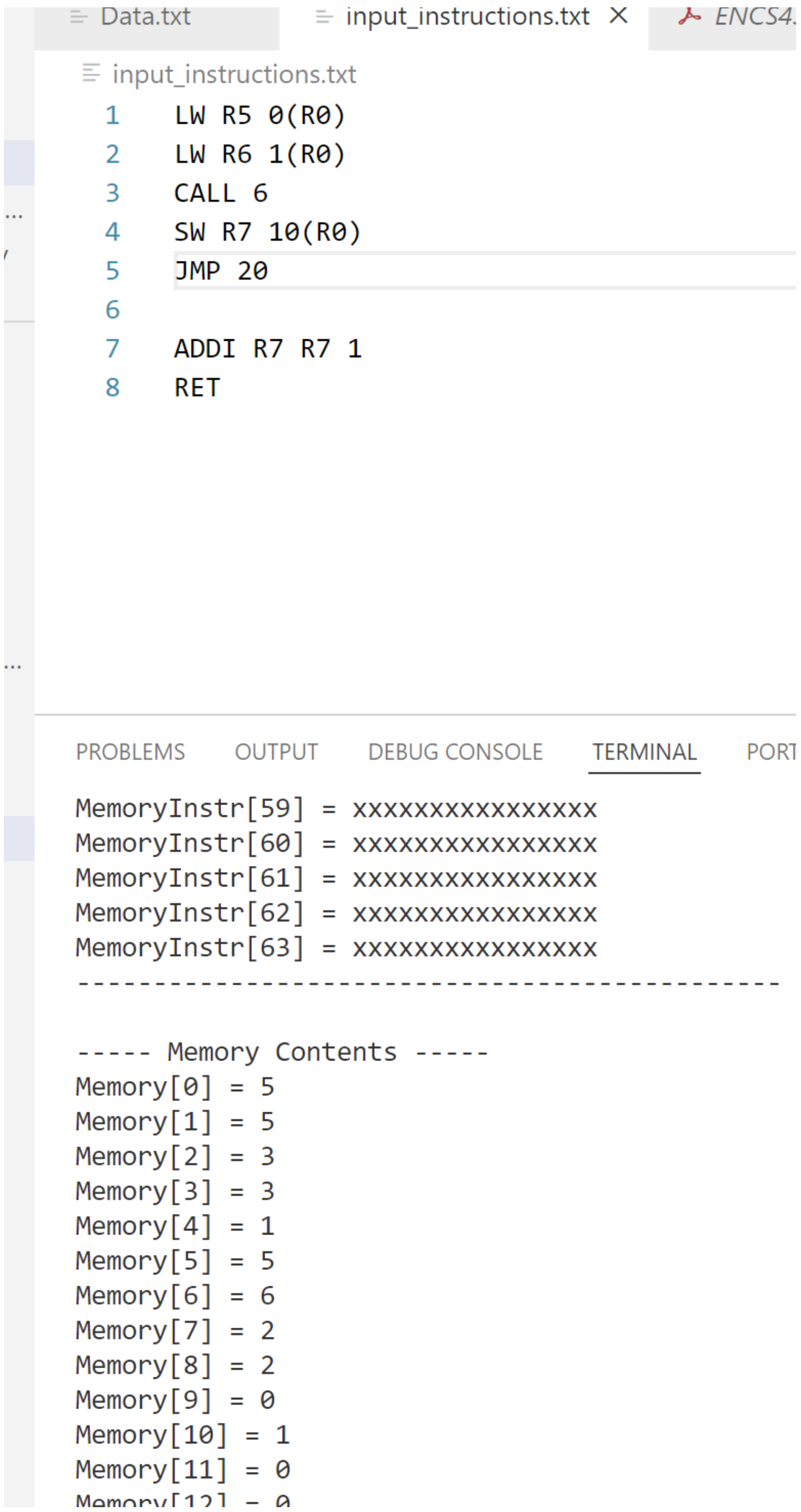
Testing for Call & Return



```
LW R5 0(R0)
ADDI R3 R5 10
ADD R7 R7 R3
CALL 5
BNE R5 R0 2
ADDI R3 R5 6
RET
ADDI R3 R5 3
SW R3 12(R0)
```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
-------------------------------------------

---- Memory Contents -----
emory[0] = 0
emory[1] = 1
emory[2] = 2
emory[3] = 3
emory[4] = 4
emory[5] = 5
emory[6] = 6
emory[7] = 7
emory[8] = 8
emory[9] = 9
emory[10] = 10
emory[11] = 0
emory[12] = 0
emory[13] = 0
emory[14] = 0
emory[15] = 0
emory[16] = 0
emory[17] = 0
emory[18] = 0
```

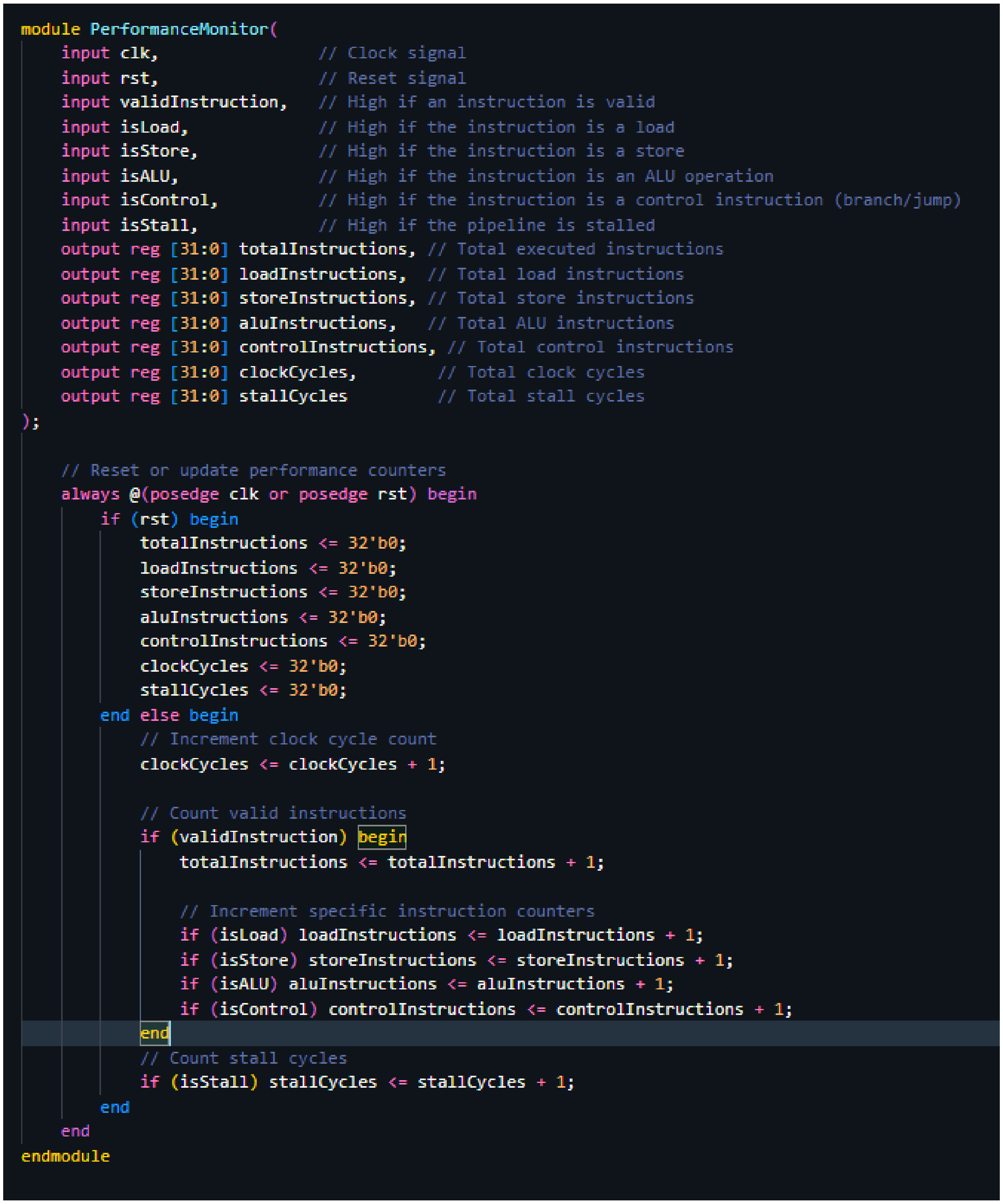


```
LW R5 0(R0)
LW R6 1(R0)
CALL 6
SW R7 10(R0)
JMP 20

ADDI R7 R7 1
RET
```

```
MemoryInstr[59] = xxxxxxxxxxxxxxxx
MemoryInstr[60] = xxxxxxxxxxxxxxxx
MemoryInstr[61] = xxxxxxxxxxxxxxxx
MemoryInstr[62] = xxxxxxxxxxxxxxxx
MemoryInstr[63] = xxxxxxxxxxxxxxxx
---------------------------------------------

----- Memory Contents -----
Memory[0] = 5
Memory[1] = 5
Memory[2] = 3
Memory[3] = 3
Memory[4] = 1
Memory[5] = 5
Memory[6] = 6
Memory[7] = 2
Memory[8] = 2
Memory[9] = 0
Memory[10] = 1
Memory[11] = 0
Memory[12] = 0
```

Figure 25: JMP, CALL, RET

Performance Registers



```
module PerformanceMonitor(
    input clk,                 // Clock signal
    input rst,                 // Reset signal
    input validInstruction,    // High if an instruction is valid
    input isLoad,              // High if the instruction is a load
    input isStore,             // High if the instruction is a store
    input isALU,               // High if the instruction is an ALU operation
    input isControl,           // High if the instruction is a control instruction (branch/jump)
    input isStall,             // High if the pipeline is stalled
    output reg [31:0] totalInstructions, // Total executed instructions
    output reg [31:0] loadInstructions,  // Total load instructions
    output reg [31:0] storeInstructions, // Total store instructions
    output reg [31:0] aluInstructions,   // Total ALU instructions
    output reg [31:0] controlInstructions, // Total control instructions
    output reg [31:0] clockCycles,        // Total clock cycles
    output reg [31:0] stallCycles         // Total stall cycles
);

    // Reset or update performance counters
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            totalInstructions <= 32'b0;
            loadInstructions <= 32'b0;
            storeInstructions <= 32'b0;
            aluInstructions <= 32'b0;
            controlInstructions <= 32'b0;
            clockCycles <= 32'b0;
            stallCycles <= 32'b0;
        end else begin
            // Increment clock cycle count
            clockCycles <= clockCycles + 1;

            // Count valid instructions
            if (validInstruction) begin
                totalInstructions <= totalInstructions + 1;

                // Increment specific instruction counters
                if (isLoad) loadInstructions <= loadInstructions + 1;
                if (isStore) storeInstructions <= storeInstructions + 1;
                if (isALU) aluInstructions <= aluInstructions + 1;
                if (isControl) controlInstructions <= controlInstructions + 1;
            end
            // Count stall cycles
            if (isStall) stallCycles <= stallCycles + 1;
        end
    end
endmodule
```

Figure 26: Performance Registers

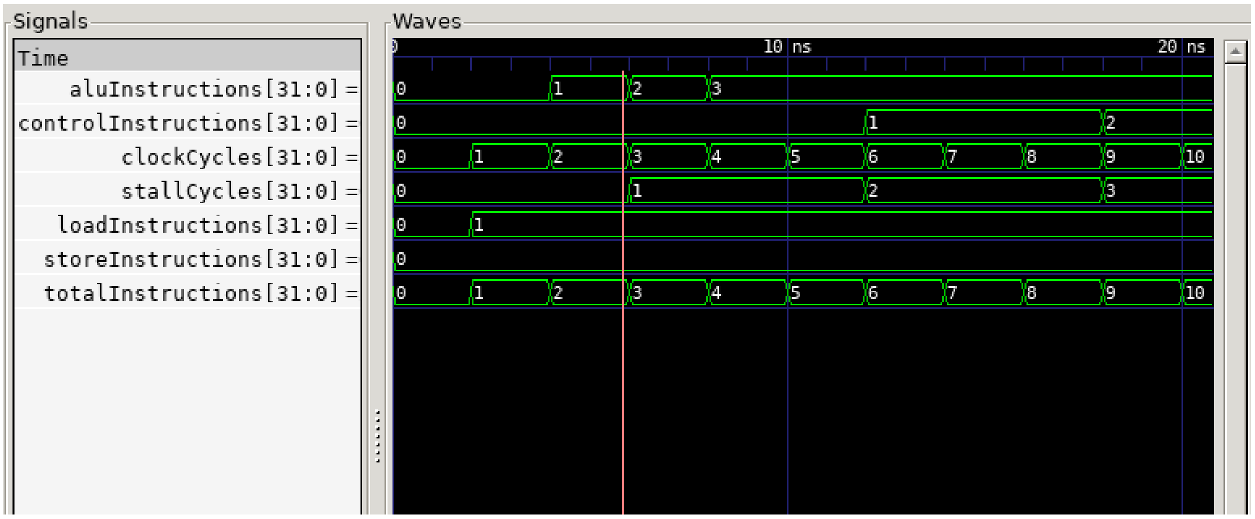


Figure 27: Performance Registers Results

Observations & Issues

All instructions in both the **single-cycle** and **pipelined** processors **function correctly** and as expected, demonstrating the effectiveness of the **Datapath** and **control** path design. This accuracy highlights the robustness of our implementation in handling various instruction types.

However, there are potential enhancements that could improve the processor's efficiency in the future. For example, adding a comparator in the Decode stage could enable **branch prediction** at an earlier stage, reducing delays caused by branch instructions.

It is also worth noting a minor issue identified with the **FOR-loop** instruction in the **pipelined** processor. While this **does not affect the overall functionality**, it can be addressed in future iterations to ensure flawless operation. With these refinements, the processor's performance and efficiency could be further optimized.

Conclusion

In this project, we successfully designed and implemented a **pipelined** processor, building on the foundational design of a **single-cycle** processor. Using a professional testing and implementation environment, combined with a detailed block diagram of the Datapath, we gained valuable insights into the inner workings of processor design. This project provided a deep understanding of pipelining, **single-cycle**, and **multi-cycle** processors, enabling us to compare their performance, efficiency, and trade-offs, as well as to defend our design choices effectively.

Our design handled key challenges such as **stalling**, **flushing**, and resolving **RAW** (Read After Write) hazards by optimizing the timing of write and read operations within a single clock cycle. While the project was a success, it is worth noting that testing and debugging required significant time due to the large number of control signals and the complexity of multiple stages. However, this experience has equipped us with the skills and expertise to approach similar projects more efficiently in the future.

Overall, the project not only strengthened our knowledge of processor architecture but also provided hands-on experience in **designing**, **testing**, and **refining** the pipelined processor.

References:

- Harris, D. M., & Harris, S. L. (2012). *Digital design and computer architecture* (2nd ed.). Morgan Kaufmann.
- Computer Architecture Slides – Dr. Aymen Hroub