Birzeit University

Faculty of Engineering & Technology

Department of Electrical & Computer Engineering

ENCS4370 || Computer Architecture

Multicycle RISC Processor in Verilog

Prepared by:

Haneen Odeh 1210716

Rana Musa 1210007

Mays Ajaleen 1211242

Instructor:

Dr. Aziz Qaroush

June.2024

Abstract

This report presents the design, implementation, and testing of a 16-bit multicycle RISC processor in Verilog HDL. Supporting R-type, I-type, J-type, and S-type instructions, it features 8 registers and separate instruction and data memories. The multicycle Datapath includes fetch, decode, execute, memory access, and write-back stages. Key components are the ALU, register file, control unit, and multiplexers. The state machine-based control unit ensures precise signal generation. Testing confirms the processor's functionality and integration, demonstrating the efficiency of the design.

Contents

Table of Figures

List of Tables

Introduction

1.1 Theory

This project focuses on designing and testing a simple multicycle RISC processor using Verilog HDL. The processor features a 16-bit instruction and word size, ensuring compact and efficient instruction encoding and execution. The architecture includes 8 general-purpose 16-bit registers (R0 to R7), with R0 hardwired to zero to discard any write attempts. Additionally, a 16-bit program counter (PC) is used as a special-purpose register.

The processor supports four instruction types: R-type, I-type, J-type, and S-type, allowing a variety of operations from arithmetic to control flow. It has separate data and instruction memories, enhancing parallelism and efficiency during instruction fetch and data access cycles. Both memories are byte-addressable, enabling flexible data manipulation at the byte level, and use little endian byte ordering, where the least significant byte is stored at the lowest memory address.

The ALU generates the necessary signals to determine the condition branch outcome (taken/not taken), facilitating effective control flow management. The multicycle Datapath design breaks down each instruction into multiple cycles, enabling more complex and efficient instruction execution phases.

The project's aim is to develop a functional and efficient processor design, leveraging Verilog HDL for precise and modular hardware description and testing. Central to the processor's architecture is its multicycle Datapath, which consists of five stages: Instruction Fetch, Instruction Decode, Execution, Memory Access, and Write Back. Each stage is essential for systematic and efficient instruction processing.

A critical component of this project is developing a control unit using a state machine approach. This control unit is vital for managing the Datapath and generating control signals based on the instruction type and the Datapath’s current state.

This project offers practical experience in processor design and provides an in-depth understanding of instruction set architecture and multicycle processor design. It combines theoretical knowledge with hands-on application, aiming to master the complexities of designing a multicycle RISC processor using Verilog HDL.

1.2 Instruction Format

The processor's Instruction Set Architecture (ISA) includes four distinct instruction formats: R-type, I-type, J-type, and S-type, each with its own unique structure.

1.2.1 R-Type (Register Type)

- Opcode (4-bits)
- Rd (3-bits): Destination register
- Rs1 (3-bits): First Source register
- Rs2 (3-bits): Second Source register
- Unused (3-bits)

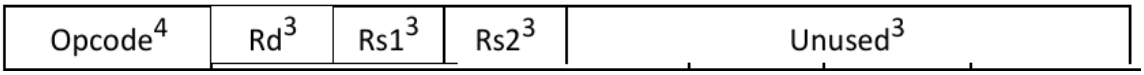
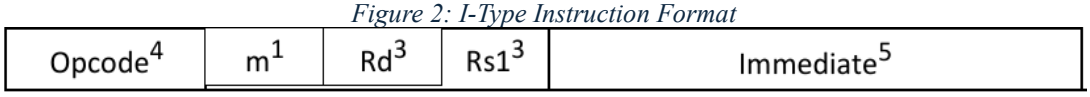


Figure 1: R-Type Instruction Format

1.2.2 I-Type (Immediate Type)

- Opcode (4-bits)
- Mode (1-bit) :
  - For Load :
    - 1 : LBs load byte with sign extension
    - 0 : LBu load byte with zero extension
  - For Branch :
    - 0 : compared Rd with Rs1
    - 1 : compared Rd with R0
- Rd (3bits): Destination register
- Rs1 (3-bits): First Source register
- Immediate (5 -bits)

Figure 2: I-Type Instruction Format



1.2.3 J-Type (Jump Type)

- Opcode (4-bits)
- Jump Offset (12-bits)
  - For JMP and Call instructions

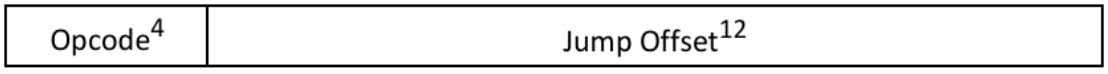


Figure 3: J-Type instruction Format

  - For RET instruction [return]

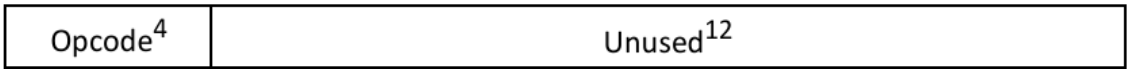


Figure 4: J-Type ret- Format

1.2.4 S-Type (Store Type)

- Opcode (4-bits)
- Rs (3-bits): Source register
- Immediate (9 -bits)

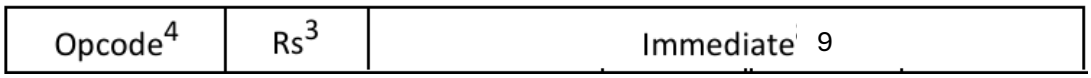


Figure 5:S-Type Format

1.3 Instruction Sets

The processor's instruction set is integral to its functionality, encompassing various categories designed to address specific computational needs. R-Type instructions handle arithmetic and logical operations, I-Type instructions manage operations involving immediate values, J-Type instructions facilitate unconditional branching, and S-Type instructions support a single operation for memory storage.

Each instruction type enhances the processor's operational versatility, ensuring efficient handling of a broad range of tasks. The table below details the processor's diverse instructions, demonstrating its capability to execute complex functions effectively.

NO.	Instruction	Meaning	Opcode Value	M
R-Type				
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	-
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	-
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	-
I-Type				
4	ADDI	Reg(Rd) = Reg(Rs1) + Imm	0011	-
5	ANDI	Reg(Rd) = Reg(Rs1) & Imm	0100	-
6	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	-
7	LBu	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
8	LBs	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
9	SW	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	-
10	BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	0
11	BGTZ	if (Reg(Rd) > Reg(0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1000	1
12	BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1001	0
13	BLTZ	if (Reg(Rd) < Reg(R0)) Next PC = PC + sign_extended (Imm)	1001	1

		else PC = PC + 2		
14	BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	0
15	BEQZ	if (Reg(Rd) == Reg(R0)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1010	1
16	BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	0
17	BNEZ	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm) else PC = PC + 2	1011	1
J-Type				
18	JMP	Next PC = {PC[15:10], Immediate}	1100	-
19	CALL	Next PC = {PC[15:10], Immediate} PC + 2 is saved on r7	1101	-
20	RET	Next PC = r7	1110	-
S-Type				
21	Sv	M[rs] = imm	1111	-

Table 1: Instructions Table

Components

2.1 Memory

In the context of a multicycle Datapath, memory is divided into two essential components: Instruction Memory and Data Memory.

2.1.1 Instruction Memory

The Instruction Memory is designed to store and fetch the 16-bit instructions needed by the processor. It works in sync with the clock signal, using a 16-bit Program Counter (PC) address to fetch instructions. Preloaded with a variety of instructions, including R-Type, I-Type, J-Type, and S-Type operations, it uses parameterized opcodes for efficient access and execution.

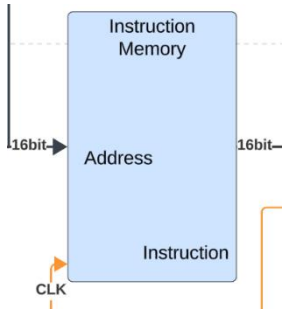


Figure 6: Instruction Memory Component

During the instruction fetch phase, the Instruction Memory provides the necessary instructions based on the PC. This separation from data operations ensures an optimized instruction execution process. The setup guarantees reliable and prompt retrieval of instructions, perfectly synchronized with the processor's clock cycles, highlighting the critical role of memory in efficient processing.

2.1.1 Data Memory

It is Essential for storing and accessing the data needed for the processor's operations. It is accessed during the memory access stage of the multicycle execution process, allowing the processor to read or write data based on specific instructions. This memory holds the operands for various calculations and stores the results generated by the processor. The key inputs, outputs and control units affecting Data Memory include:

- **Address**: Specifies the memory location to read from or write to.
- **Data_in**: The data to be written into the memory at the specified address.
- **Data_out**: The data read from the memory at the specified address.

- **Memory_Write**: A control signal that enables writing data to memory.
- **Memory_Read**: A control signal that enables reading data from memory.

These inputs and control signals ensure that the Data Memory can be accurately and efficiently accessed during the appropriate stage of the instruction cycle, facilitating smooth and effective data handling within the processor.

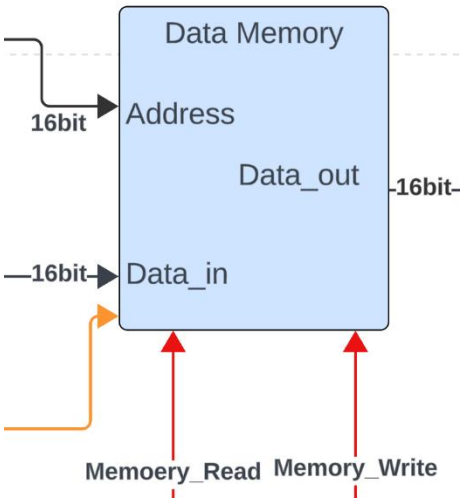


Figure 7:Data Memory Component

2.2 Register File

It is an essential component responsible for providing quick access to a set of general-purpose registers . The processor features 8 16-bit registers (R0 to R7), where R0 is hardwired to zero. These registers are used to store temporary data that the processor needs for various operations. The register file includes the following key elements:

- **Inputs**:
  - **Write_data**: The data to be written into a specified register.
  - **rs1**: The address of the first source register.
  - **rs2**: The address of the second source register.
  - **rd**: The address of the destination register where data is to be written.
- **Outputs**:
  - **reg_data1**: Data read from the register.
  - **reg_data2**: Data read from the other wanted register.
- **Control Signals**:
  - **reg_write**: A control signal that enables writing data to the specified register.

The register file interacts with other components of the Datapath, such as the ALU and the data memory, to facilitate the execution of instructions. The appropriate registers are accessed to read the required operands or to store the results of computations. This interaction is governed by the control signals that ensure the correct registers are used and the necessary operations are performed efficiently.

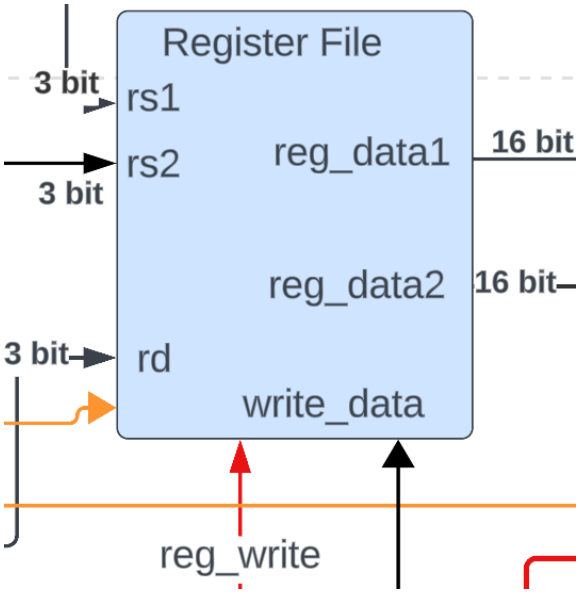


Figure 8: Register File Component

2.3 Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU) in the multicycle datapath is a pivotal component responsible for executing a wide range of arithmetic and logical operations. It operates under the control of the `ALUOp` signal, which specifies the exact operation to perform, such as addition, subtraction, and bitwise operations (AND) in our case. The ALU interacts closely with the register file, fetching operands and storing results, and plays a crucial role in branch decisions by generating condition codes like `Zero`.

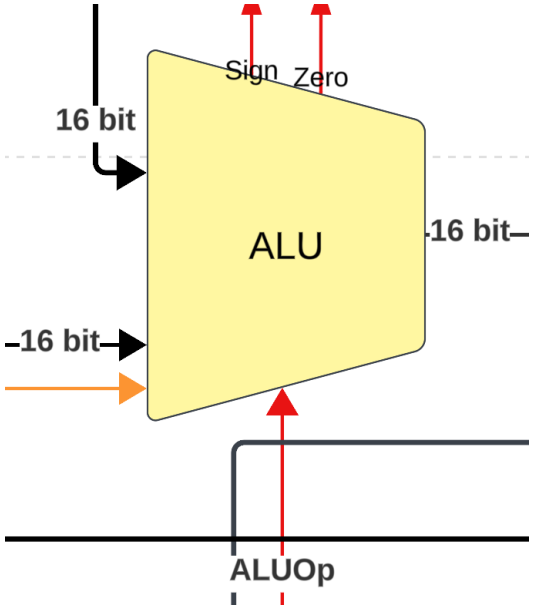


Figure 9: ALU Component

Additionally, the `ALUSrc` signal determines whether the second operand is sourced from a register or an exten. immediate value. This design not only optimizes resource utilization but also ensures that complex instructions are executed correctly over several cycles, enhancing the processor's flexibility and functionality.

2.4 Program counter (PC)

In the multicycle Datapath, the Program Counter (PC) is a special-purpose register that holds the address of the next instruction to be executed. The PC is crucial for instruction sequencing, ensuring that the processor fetches instructions in the correct order. During each cycle, the PC is updated to point to the next instruction, which could be sequential or determined by branch and jump instructions. The PC update mechanism includes adding a

constant value (typically 2, given the word size) to move to the next instruction or using a computed branch target address for control flow changes.

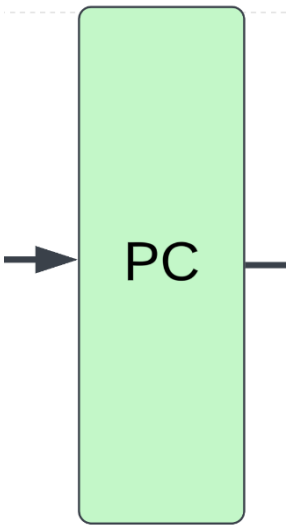


Figure 10: PC Component

2.5 Extender

In the our Datapath, the extender takes shorter bit-length immediate values from the instruction and extends them to match the processor's word size (16 bits). This is crucial for operations that involve immediate values, such as arithmetic operations with constants or addressing calculations. The extender can perform zero-extension or sign-extension, depending on the instruction type, ensuring that the immediate values are correctly interpreted and used in subsequent operations. This allows the processor to handle a variety of instruction formats and data types efficiently.

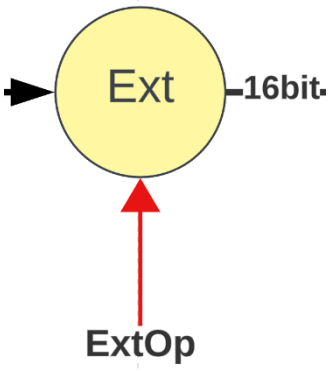


Figure 11: Ext Component

2.6 Control Unit

The control units, coloured in red in the Datapath, are responsible for generating the control signals that guide the operation of various components within the processor. These signals determine the actions of the ALU, memory, and registers during each cycle.

The main control unit interprets the opcode of the current instruction to produce signals such as' ALUOp', 'RegWrite'," MemRead', 'MemWrite', and 'PCSrc'. Additionally, there are specific control units for branching and returning from function calls, ensuring that the correct execution flow is maintained. These units coordinate closely to manage the transitions between different states of instruction execution, thereby maintaining precise control over the Datapath.

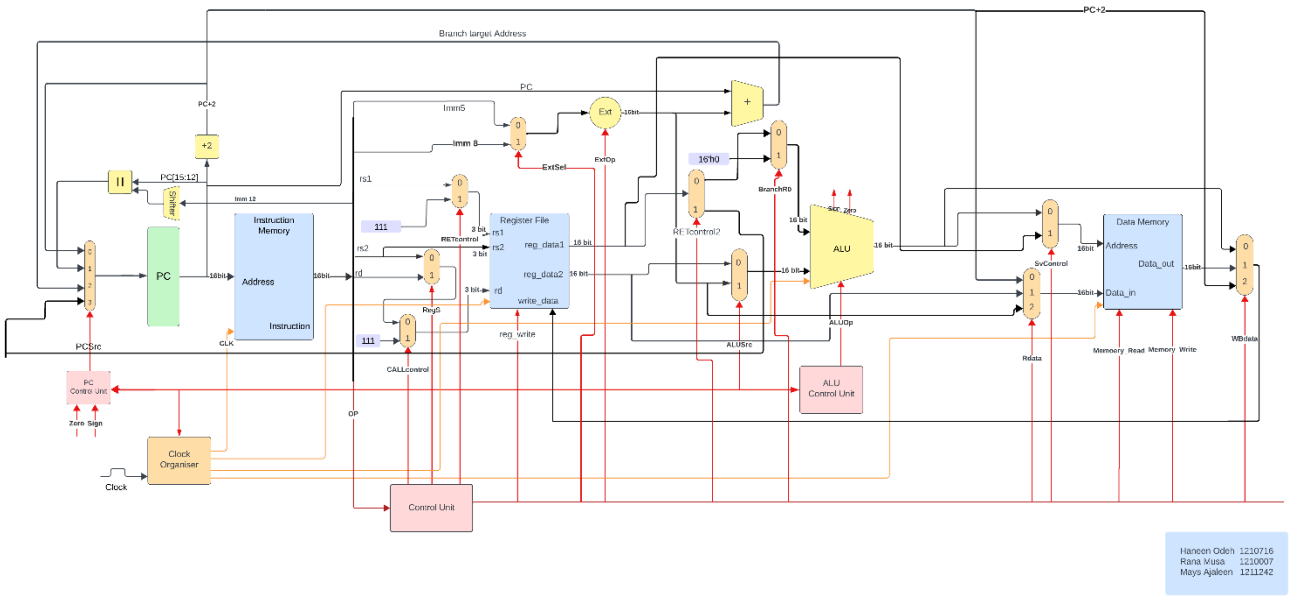


Figure 12:Multicycle Datapath Architecture

2.7 Clock Organiser

The clock organiser in the Datapath is essential for synchronizing processor operations, ensuring all components perform tasks in a coordinated manner within each cycle's timing constraints. It generates clock pulses that drive sequential logic elements like registers and memory. By regulating these pulses, the clock organiser ensures that instructions are executed correctly over multiple cycles, maintaining orderly data transfer and processing.
In our implementation, the clock organiser uses a finite state machine (FSM) to control the stages of instruction execution: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (M), and Write Back (WB).

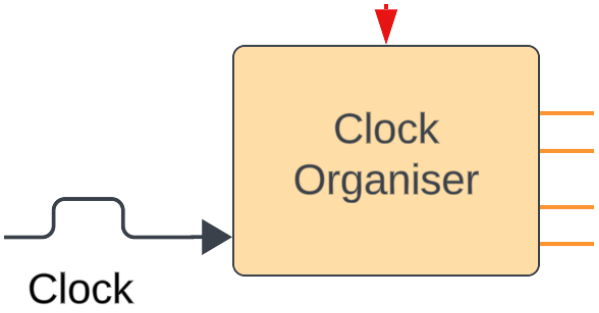


Figure 13: clk organizer Component

2.8 Multiplexer

2.8.1 Multiplexer 2to1

A 2-to-1 multiplexer selects one of two inputs to pass through to the output based on a single select signal. In our Datapath, a 2-to-1 MUX is used, for example, to select between the immediate value and the second register value as the second operand for the ALU. The select signal (ALUSrc) determines which input (register data or immediate) is forwarded to the ALU.

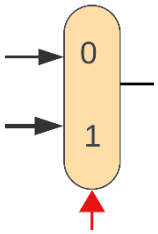


Figure 14:Mux2to1 Component

2.8.2 Multiplexer 4to1

A 4-to-1 multiplexer selects one of four inputs based on two select signals. This MUX is useful for more complex decisions, such as determining the source of the Program Counter (PC) update. For example, it can choose between the next sequential PC, a branch target address, a jump address, or a return address from a function call. This allows the processor to handle different types of control flow instructions efficiently. The select signals determine which of these inputs will be forwarded to the PC, ensuring the correct instruction is fetched next.

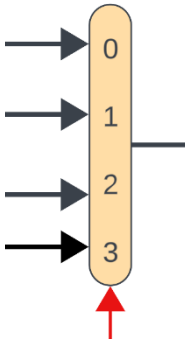


Figure 15:Mux4to1 Component

2.8.3 Multiplexer 3to1

A 3-to-1 multiplexer chooses one of three inputs to pass to the output based on two select signals. This type of MUX can be used to determine the source of data to write back to the register file. For example, it can select between the ALU results, data read from memory, or an immediate value. By using the appropriate select signals, the processor ensures that the correct data is written back to the registers, supporting various instruction types and enhancing the flexibility of the Datapath.

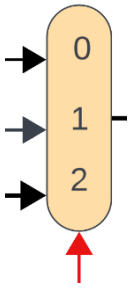


Figure 16:Mux3to1 Component

2.9 Concatenation

It involves joining multiple binary values into a single value. This technique is used, for example, in forming the target address for jump instructions. By concatenating a portion of the PC with the offset specified in the instruction, the processor can generate a full address for the jump.

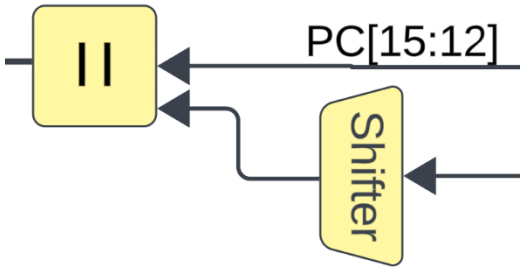


Figure 17:Concatenation Component

2.10 Adder

The adder is a fundamental component used for performing arithmetic addition. In the Datapath, an adder is used to increment the PC by a fixed value (2) to point to the next instruction.

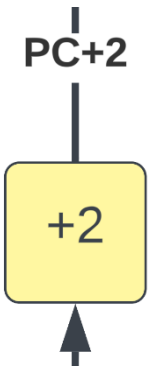


Figure 18:Adder Component

Another adder is used to compute branch target addresses by adding the sign-extended immediate value to the current PC.

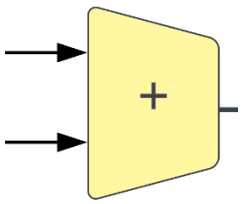


Figure 19: Two Input Adder Component

Design and Implementation

3.1. Full Data Path

The Multi-Cycle Processor with Memory's flawless integration of all of its parts and connections culminates in the fully connected Datapath. This thorough integration includes designs for the control signal as well as the data path. At first, in the initial stage that's it is the fetch stage, we have added the suitable components for it, such as the program counter "PC", as well as an incrementing adder to generate the next instruction address, that adder adds 2 to the value of the PC, also there is the instruction memory, to fetch and obtain the wanted instruction, its input is the value of the PC added to 2, and the output is the instruction, the next stage is the decode stage, that contains the register file component, its inputs are the outputs from the instruction memory, the register file outputs then are seamlessly forwarded to the ALU stage or as its called the execution stage, where we perform the different operations such as and, add or others , each operations has its own opcode. The resulting data from the execution stage goes to the memory stage, where we read data from it, or write data to the memory by its address. The final stage is the write back data stage, where we store the needed data to the register file in the destination register. The figure below shows our Datapath, with all control units, components needed to build it fully and correctly.

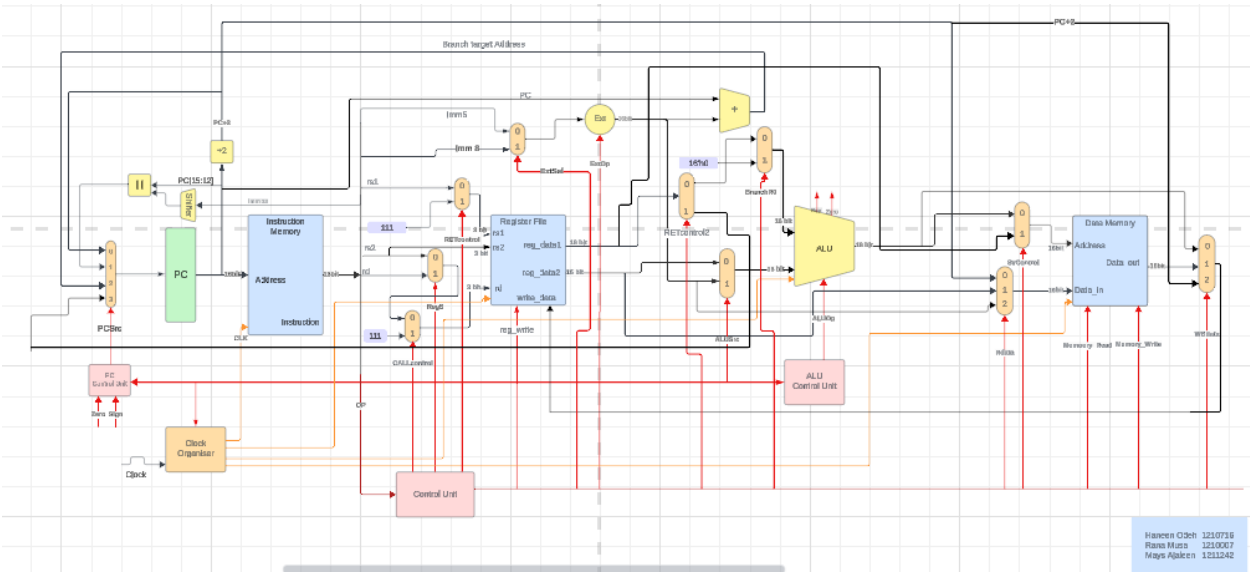


Figure 20:Full Datapath Diagram

3.2. System's Control Signals

3.2.1. Signals Outputs Meanings

In our system, we have sixteen control signals, each one of them has its own meaning and purpose, those are determined by factors such as the instruction type, instruction function and others. The below signals table are shown with their respective meanings

Table 2:Description of Signals and Effects

Signal	Effect when "0"	Effect when "1"	Effect when "2"	Effect when "3"
PCSrc	The next pc=pc+2	The next PC = {PC[15:10], Immediate}	The next PC = PC + sign_extended (Imm)	The next pc=R7
AluOp	The operation is AND	The operation is ADD	The operation is SUB	----------
ExtOp	Zero immediate extension	Sign immediate extension	----------	----------
RegS	Input to mux call control=Rs2	Unput to mux call control= Rd	----------	----------
AluSrc	Input to Alu=BusB	Input to Alu=result of the extender	----------	----------
MemRead	Data memory is not read	Data memory is read	----------	----------
MemWrite	Data memory is not written	Data memory is written	----------	----------
RegWrite	Destination register Rd is not written	Destination register Rd is written	----------	----------
SVcontrol	Input to address memory=ALU result	Input to address memory=BusA	----------	----------
RETcontrol	Input to Rs1=Rs1	Input to Rs1=3'b111	----------	----------
RETcontrol2	Input to muxBranchR0=BusA	Input to muxPCSrc=BusA	----------	----------
CALLcontrol	Input to Rd= Rs2||rd	Input to Rd=3'b111	----------	----------
ExtSel	Input to the Extender=Imm5	Input to the Extender=Imm8	----------	----------
BranchR0	Input to ALU=BusA	Unput to ALU=16'h0	----------	----------
WBdata	Write back data to register=ALU result	Write back data to register= Data out from memory	Write back data to register= PC+2	----------
Rdata	Data into memory = pc+2	Data into memory = BusB	Data into memory= immediate	----------

3.2.2. Main Control Unit

The values of all the control signals for each instruction are shown in the table below; some of the signals are denoted with a 'x' to indicate that their exact value is not critical to the functioning of the instruction or has little effect.

OP	RegS	RegW	ExtOp	AluSrc	MemR	MemW	SVcontrol	RETcontrol	RET control2	CALL control	EXT Sel
AND	1=Rd	1	X	0	0	0	0	0	0	0	X
ADD	1=Rd	1	X	0	0	0	0	0	0	0	X
SUB	1=Rd	1	X	0	0	0	0	0	0	0	X
ADDI	0	1	0	1	0	0	0	0	0	0	0=Imm5
ANDI	0	1	1	1	0	0	0	0	0	0	0=Imm5
LW	0	1	1	1	1	0	0	0	0	0	0=Imm5
LBu	0	1	1	1	1	0	0	0	0	0	0=Imm5
LBs	0	1	1	1	1	0	0	0	0	0	0=Imm5
SW	0	0	1	1	0	1	0	0	0	0	0
BGT	0	0	1	0	0	0	0	0	0	0	0
BGTZ	0	0	1	0	0	0	0	0	0	0	0
BLT	0	0	1	0	0	0	0	0	0	0	0
BLTZ	0	0	1	0	0	0	0	0	0	0	0
BEQ	0	0	1	0	0	0	0	0	0	0	0
BEQZ	0	0	1	0	0	0	0	0	0	0	0
BNE	0	0	1	0	0	0	0	0	0	0	0
BNEZ	0	0	1	0	0	0	0	0	0	0	0
JMP	0	0	0	0	0	0	0	0	0	0	0
CALL	0		0	0	0	0	0	0	0	1	0
RET	0	0	0	0	0	0	0	1	1	0	0
SV	0	0	1	0	0	0	1	0	0	0	1

Table 3: Main Control Unit Signals

OP	BranchR0	WBdata	Rdata
AND	0	00	00
ADD	0	00	00
SUB	0	00	00
ADDI	0	00	00
ANDI	0	00	00
LW	0	01	00
LBu	0	01	00
LBs	0	01	00
SW	0	xx	01
BGT	0	xx	00
BGTZ	1	xx	00
BLT	0	xx	00
BLTZ	1	xx	00
BEQ	0	xx	00
BEQZ	1	xx	00
BNE	0	xx	00
BNEZ	1	xx	00
JMP	0	xx	00
CALL	0	2	00
RET	0	xx	00
SV	0	xx	10

Table 4: Main Control Unit Signals Cont.

Each signal is assigned a Boolean value based on its corresponding Boolean expression, determined by instruction type, function.

- RegWrite= ADD || AND || SUB || ADDI || ANDI || LW || LBu || LBs || CALL;
- RegS= AND || ADD || SUB;
- ExtOp= SW || LW || ADDI || BEQ || BNEZ || BGT || BLT || BGTZ || BLTZ || ANDI || LBu || LBs || BEQZ || BNE || Sv;
- ALUSrc= ADDI || ANDI || LW || SW || LBu || LBs;
- MemRead= LW || LBu || LBs;
- MemWrite= SW;
- RETcontrol= RET;
- RETcontrol2= RET;
- CALLcontrol=CALL;
- ExtSel= Sv;
- BranchR0= BGTZ || BLTZ || BEQZ || BNEZ ;
- SvControl = Sv;

3.2.3. ALU Control Unit

The value of the "ALUop" signal determines its consequences. The AND operation is carried out by the ALU when it is '0'. It starts the ADD process when set to '1'. It tells the ALU to carry out the SUB operation when it is set to "2". On the other hand, when the value is '3', it remains inactive or undefined in this context because no impact is indicated.

OP	AluOp	2-bit coding in alu
AND	AND	00
ADD	ADD	01
SUB	SUB	10
ADDI	ADD	01
ANDI	AND	00
LW	ADD	01
LBu	ADD	01
LBs	ADD	01
SW	ADD	01
BGT	SUB	10
BGTZ	SUB	10
BLT	SUB	10
BLTZ	SUB	10
BEQ	SUB	10
BEQZ	SUB	10
BNE	SUB	10
BNEZ	SUB	10
JMP	X	x
CALL	X	x
RET	X	x
SV	x	x

Table 5:Opcode and AluOp

3.2.4. PC Control Unit

The "PCSrc" signal dictates the program counter's (PC) behavior: '0' increments PC by 2, '1' combines bits 15 to 12 with Immediate12, '2' increments PC by sign-extended Imm5, and '3' sets PC to the same value in register 7 for the RET instruction.

OP	Zero flag	Sign flag	PCSrc
AND	X	X	00
ADD	X	X	00
SUB	X	X	00
ADDI	X	X	00
ANDI	X	X	00
LW	X	X	00
LBu	X	X	00
LBs	X	X	00
SW & SV	X	X	00
BGT &BGTZ	X	0	00
BGT &BGTZ	1	X	00
BGT &BGTZ	0	1	10
BLT & BLTZ	X	1	00
BLT & BLTZ	1	x	00
BLT & BLTZ	0	0	10
BEQ & BEQZ	0	x	00
BEQ & BEQZ	1	x	10
BNE & BNEZ	1	x	00
BNE & BNEZ	0	x	10
JMP & CALL	X	x	01
RET	x	x	11

Table 6: PC Control Unit Signals

3.3 State Machine Diagram

In multi cycle processing, we use the state machine diagram to determine for each instruction what its next stage, either its IF,ID,Ex,Mem or WB. Having the state machine make it easily to understand the concept of the multi cycle processing , because we need to know for each instruction when it ends and what its last stage so the next instruction can fetch after it. The below figure is our state machine for this project.

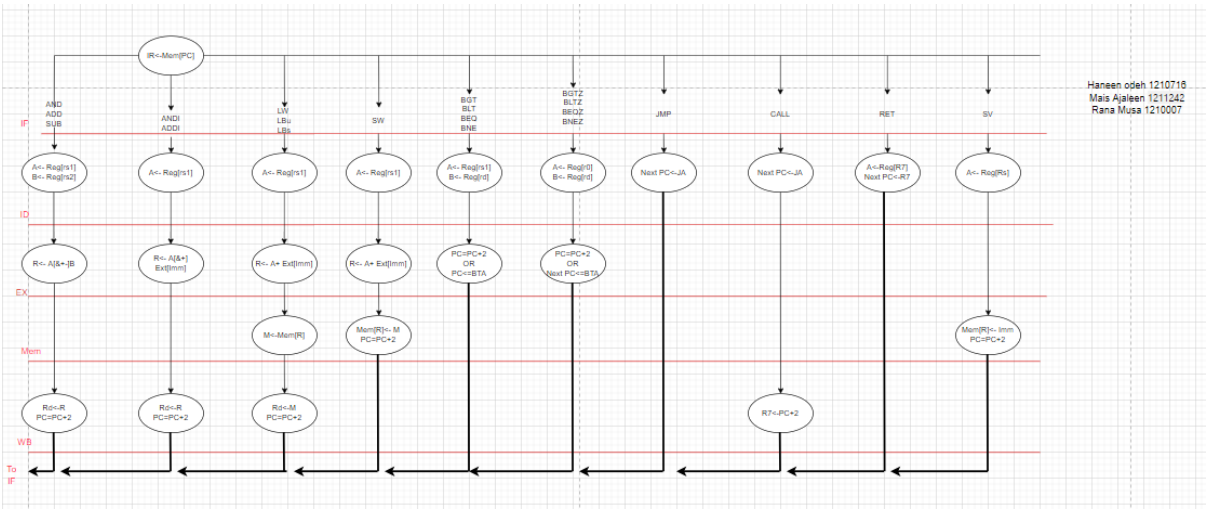


Figure 21: State Machine Diagram

Table 7: State Machine Table

Current Stage	State Code	Next Stage	Next Stage Code	Instruction Cases
Instruction Fetch	000	Instruction Decode	001	Always
Instruction Decode	001	Execute	010	R-type,I-type
		Instruction Fetch	000	JMP, RET
		Memory	011	SV
		Register Write	100	CALL
Execute	010	Register Write	100	R-type, ANDI,ADDI
		Memory	011	LW,LBu,LBs,SW
		Instruction Fetch	000	BGT,BGTZ,BLT,BLTZ,BEQ,BEQZ,BNE,BNEZ
Memory	011	Register Write	100	LW,LBu,LBs
		Instruction fetch	000	SW
Write Back	100	Instruction Fetch	000	Always

Testing and Validation

Overview

Testbenches provided a crucial platform for designers to evaluate modules in a controlled setting before incorporating them into larger systems, ensuring the modules functioned correctly. Understanding the module's requirements, input and output signals, and the conditions under which it was meant to operate was key to constructing effective Testbenches. By simulating various scenarios and stimuli, designers were able to verify that their modules performed as expected and identify and address errors early in the design phase. Thus, Testbenches played an essential role in validating the precision and reliability of Verilog designs. Test environments were created for each module to simulate a selection of possible scenarios.

4.1 Instruction Memory

This module represents an instruction memory, which stores instructions for a processor. It uses a clock signal and an address input to output the corresponding instruction.

```
module InstructionMemory(
    input wire clk,
    input [15:0] address,
    output reg [15:0] instruction
);

reg [15:0] MemoryInst [0:1023]; // Adjust the size to 1024 (0 to 1023) to fit 16-bit address

// Define Register Names (R0 to R7)
parameter R0 = 3'b000, R1 = 3'b001, R2 = 3'b010, R3 = 3'b011,
          R4 = 3'b100, R5 = 3'b101, R6 = 3'b110, R7 = 3'b111;

// R-Type Instructions Opcode Values
parameter AND = 4'b0000, ADD = 4'b0001, SUB = 4'b0010;

// I-Type Instructions Opcode Values
parameter ADDI = 4'b0011, ANDI = 4'b0100, LW = 4'b0101,
          LBu = 4'b0110, LBs = 4'b0110, BGTZ = 4'b1000,
          BGT = 4'b1000, BLT = 4'b1001, BLTZ = 4'b1001, SW=4'b0111,
          BEQ = 4'b1010, BEQZ = 4'b1010, BNEZ = 4'b1011, BNE = 4'b1011;

// J-Type Instructions Opcode Values
parameter JMP = 4'b1100, CALL = 4'b1101, RET = 4'b1110;

// S-Type Instructions Opcode Values
parameter Sv = 4'b1111;

// Initialize the instruction memory with a specific instruction
initial begin
    ///////////////  R Type ////////////
    MemoryInst[0] = {AND, R1/*rd*/, R2 /*rs1*/, R3 /*rs2*/, 3'b000};   // AND  R1(Rd) ,R2(Rs1), R3(Rs2)
    MemoryInst[1] = {ADD, R1, R2, R3, 3'b000}; //ADD
    MemoryInst[2] = {SUB, R1, R6, R5, 3'b000}; //SUB

    ////////////// / I Type //////////////

    MemoryInst[3] = {ADDI,1'b0, R1, R2, 5'b00001};  //ADDI  R1(Rd), R2(Rs1)
    MemoryInst[4] = {ANDI,1'b0, R1, R2, 5'b00011};
    MemoryInst[5] = {LW,1'b0, R1, R2, 5'b00010};
    MemoryInst[6] = {LBu,1'b0, R1, R2, 5'b10000};
    MemoryInst[7] = {LBs,1'b1, R1, R2, 5'b00001};
    MemoryInst[8] = {SW,1'b0, R6, R2, 5'b00001};
    MemoryInst[9] = {BGT,1'b0, R4, R2, 5'b00001};   //R1 keeps being 0000
    MemoryInst[10] = {BGTZ,1'b1, R2, R5, 5'b00011};
    MemoryInst[11] = {BLT,1'b0, R5, R2, 5'b01110};
     MemoryInst[12] = {BLTZ,1'b1, R3, R2, 5'b01010};
     MemoryInst[13] = {BEQ,1'b0, R1, R2, 5'b01100};
     MemoryInst[14] = {BEQZ,1'b1, R1, R2, 5'b00111};
```

```
48      MemoryInst[14] = {BEQZ,1'b1, R1, R2, 5'b00111};
49      MemoryInst[15] = {BNE,1'b0, R1, R2, 5'b10101};
50      MemoryInst[16] = {BNEZ,1'b1, R1, R2, 5'b00001};
51      ////////////// J Type //////////////
52     MemoryInst[17] = {JMP, 12'b000000000001};
53     MemoryInst[18] = {CALL,12'b101100101010};
54     MemoryInst[19] = {RET, 12'b000000000000};
55
56      /////////// S Type ///////////////
57
58       MemoryInst[20] = {Sv, R1, 9'b10101101};
59
60  end
61
62  // Read instruction from memory
63  always @(posedge clk) begin
64      instruction <= MemoryInst[address];
65  end
66
67  endmodule
```

The testbench validates the Instruction Memory module's ability to output the correct instruction based on the given address. By simulating different address inputs and observing the corresponding instruction outputs, the testbench confirms that the module reliably retrieves the correct instruction for each address, a critical operation for instruction execution

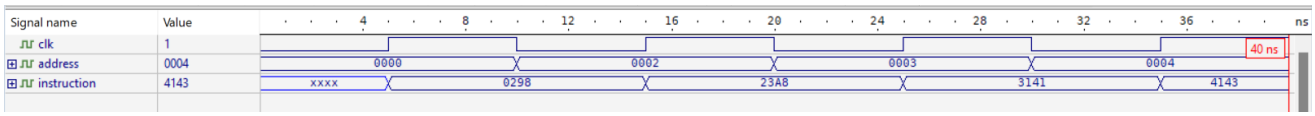


Figure 22: Instruction Memory Testbench

4.2 Data Memory

```
1   module Memory (
2       input wire clk,             // Clock signal
3        input wire [15:0] addr,     // Memory address
4       input wire mem_write,       // Memory write enable
5       input wire mem_read,        // Memory read enable
6       input wire [15:0] data_in,  // Data to be written to memory
7       output reg [15:0] data_out  // Data read from memory
8   );
9
10      // Memory array (256 words, each 16 bits, total 512 bytes)
11      reg [15:0] memory[0:65535]; // 64K memory locations
12
13       // Initialize memory with some values
14      initial begin
15          memory[0] = 16'h0001;
16          memory[1] = 16'h0002;
17          memory[2] = 16'h0003;
18          memory[3] = 16'h0004;
19          memory[4] = 16'h0005;
20           memory[5] = 16'h0006;
21          memory[6] = 16'h0007;
22          memory[7] = 16'h0008;
23          memory[8] = 16'h0009;
24          memory[9] = 16'h0010;
25            memory[10]=16'h0011;
26            memory[12]=16'h0011;
27            memory[16'hFFF6] = 16'ha2c4;
28            memory[16'h000e] = 16'haaaa;
29          // Add more initial values as needed
30      end
31
32      always @(posedge clk) begin
33          if (mem_write) begin
34              memory[addr] <= data_in; // Write data to memory (word-aligned)
35          end
36          if (mem_read) begin
37              data_out <= memory[addr]; // Read data from memory (word-aligned)
38          end
39      end
40
41  endmodule
42
```

The Data Memory module's testbench assesses its capability to handle read and write operations with respect to the given address, data input, and control signals. It tests various scenarios, including writing to and reading from different memory addresses, to ensure that the module correctly manages data storage and retrieval.

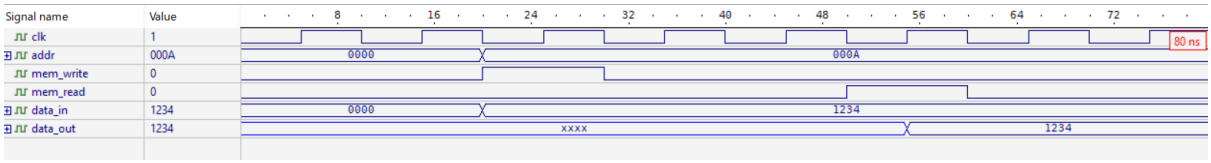


Figure 23: Data Memory Testbench

4.3 Register File

This module simulates a register file in a CPU, with read and write functionalities. It has two separate clock signals for reading and writing, input registers for read, write and buses for data transfer

```
module RegisterFile (
    input clk1,clk2,                // Clock input
    input [2:0] rs1, rs2, rd, // Read and write registers
    input [15:0] write_data, // Data to be written BusW
    input reg_write,          // Write enable
    output reg [15:0] reg_data1,  // Output data for rs1  Bus1
    output reg [15:0] reg_data2   // Output data for rs2      Bus2
);
    reg [15:0] registers [7:0];   // 8 registers, each 16 bits wide


 initial begin
     registers[0] = 16'h0000;
     registers[1] = 16'h0007;
     registers[2] = 16'h0001;
     registers[3] = 16'h0004;
     registers[4] = 16'h0002;
     registers[5] = 16'h0003;
     registers[6] = 16'h0008;
     registers[7] = 16'h0006;
     end

    // Read ports
    always @(posedge clk1) begin        //check for later
        reg_data1 = (rs1 != 3'b000) ? registers[rs1] : 16'b0;
        reg_data2 = (rs2 != 3'b000) ? registers[rs2] : 16'b0;
    end

    // Write port
    always @(posedge clk2) begin
        if (reg_write && (rd != 3'b000)) begin
            registers[rd] <= write_data;
        end
    end

endmodule

```

The testbench for the Registers File module ensures that the module accurately reads from and writes to the specified registers. It checks the module's ability to output the correct data from given register addresses and to update registers with provided data. This testbench is essential for verifying the register file's role in storing and providing CPU data.

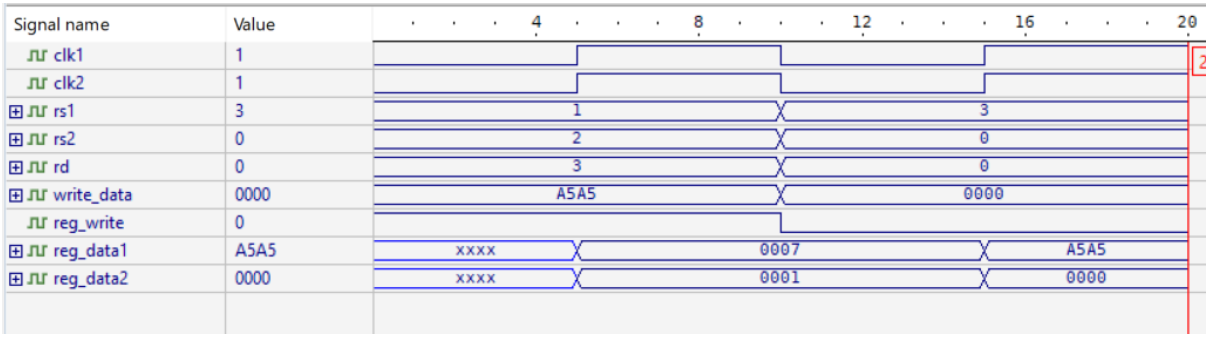


Figure 24: Register File Testbench

4.4 Arithmetic Logic Unit (ALU)

Performs arithmetic (addition, subtraction) and logical (AND) operations based on a 2-bit opcode. The ALU is the core of computational operations in a processor.

```
module ALU(
    input clk,
    input [15:0] A, B,        // 16-bit inputs
    input [1:0] Op,           // 4-bit operation code
    output reg [15:0] Result,// 16-bit result
    output reg Zero, Sign     // Zero and Sign flags
);

    // Define operation codes
    parameter AND_OP = 2'b00,
              ADD_OP = 2'b01,
              SUB_OP = 2'b10;


    // Perform the operation based on the operation code
    always @(posedge clk) begin
        case (Op)
            AND_OP: Result = A & B;
            ADD_OP: Result = A + B;
            SUB_OP: Result = A - B;

            default: Result = 16'b0;
        endcase
        Zero = (Result == 16'b0) ? 1'b1 : 1'b0;
        Sign = Result[15];
    end
endmodule
```

ALU_tb runs several scenarios, testing each operation (ADD, SUB, AND) and the correct setting of Zero and Sign flags, ensuring the ALU responds correctly to different inputs and operations

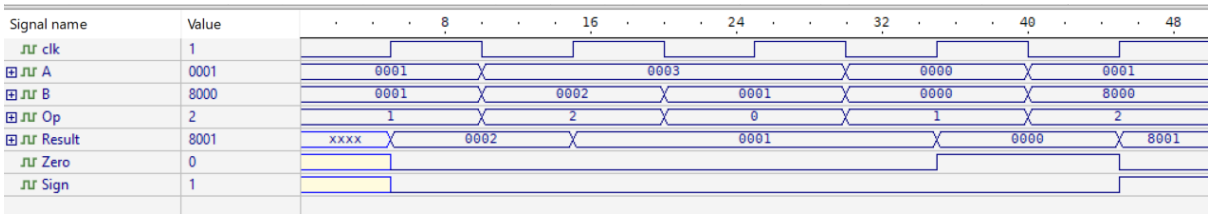


Figure 25: ALU Testbench

4.5 Control Unit

```
`include "opcodes.vh"

module control_unit (
    input m,
    input [3:0] Op,
    input zerof, signf,
    output reg ExtOp, RegS, ALUSrc, MemRead, MemWrite, RegWrite, SvControl, RETcontrol, CALLcontrol, RETcontrol2, ExtSel, BranchR0,
    output reg [1:0] ALUOp,
    output reg [1:0] PCSrc, WBdata, Rdata
);

// R-type
reg ADD, AND, SUB;
// I-type
reg ADDI, ANDI, LW, LBu, LBs, BGTZ, BGT, BLT, BLTZ, SW, BEQ, BEQZ, BNEZ, BNE;
// J-type
reg JMP, CALL, RET;
// S-type
reg Sv;

always @* begin
    // Default values
    ALUSrc = 0;
    MemRead = 0;
    MemWrite = 0;
    RegWrite = 0;
    PCSrc = 2'b00;
    ALUOp = 2'b00;
    RETcontrol = 0;
    RETcontrol2 = 0;
    CALLcontrol = 0;
    Rdata = 2'b00;
    ExtSel = 0;
    ExtOp = 0;
    BranchR0 = 0;

    ADD = 0; AND = 0; SUB = 0; ADDI = 0; ANDI = 0; LW = 0; LBu = 0; LBs = 0; BGTZ = 0; BGT = 0; BLT = 0; BLTZ = 0; SW = 0;
    BEQ = 0; BEQZ = 0; BNEZ = 0; BNE = 0; JMP = 0; CALL = 0; RET = 0; Sv = 0;

    case (Op)
        4'b0000: AND = 1;
        4'b0001: ADD = 1;
        4'b0010: SUB = 1;
        4'b0011: ADDI = 1;
        4'b0100: ANDI = 1;
        4'b0101: LW = 1;
        4'b0110: begin
            if (m == 0)
                LBu = 1;
            else
                LBs = 1;
        end
```

```
        4'b0111: SW = 1;
        4'b1000: begin
            if (m == 0)begin
                BGT = 1;
                BGTZ=0;
                end
            else begin
                BGTZ = 1;
                BGT=0;
                end
        end
        4'b1001: begin
            if (m == 0)
                BLT = 1;
            else
                BLTZ = 1;
        end
        4'b1010: begin
            if (m == 0)
                BEQ = 1;
            else
                BEQZ = 1;
        end
        4'b1011: begin
            if (m == 0)
                BNE = 1;
            else
                BNEZ = 1;
        end
        4'b1100: JMP = 1;
        4'b1101: CALL = 1;
        4'b1110: RET = 1;
        4'b1111: Sv = 1;

    endcase

RegWrite= ADD || AND || SUB || ADDI || ANDI || LW || LBu || LBs || CALL;
 RegS= AND || ADD || SUB;
 ExtOp= SW || LW || ADDI || BEQ || BNEZ || BGT || BLT || BGTZ || BLTZ || ANDI || LBu || LBs || BEQZ || BNE || Sv;
 ALUSrc= ADDI || ANDI || LW || SW || LBu || LBs;
 //Rdata= SW;
 MemRead= LW || LBu || LBs;
 MemWrite= SW;
 RETcontrol= RET;
 RETcontrol2= RET;
 CALLcontrol=CALL;
 ExtSel= Sv;
 BranchR0= BGTZ || BLTZ || BEQZ || BNEZ ;
 SvControl = Sv;
```

```
 SvControl = Sv;

  if ( ADD || ADDI || LW || LBu || LBs || SW ) begin
 ALUOp = 2'b01 ; // Assign ALUOp to 01
 end else if ( SUB || BGTZ|| BLTZ || BEQZ || BNEZ || BGT || BLT || BEQ || BNE ) begin
ALUOp = 2'b10 ; // Assign ALUOp to 10
end
else
    begin
    ALUOp= 2'b00;
    end


//////////////////////////
    case ( Op )
 // CALL
 4'b1101:begin
 WBdata = 2'b10 ;
 end

 // LW
 4'b0101:begin
 WBdata = 2'b01 ;
 end

 // LBu
 4'b0110:begin
 WBdata = 2'b01 ;
 end

 // LBs
4'b0110:begin
 WBdata = 2'b01 ;
 end


 // AND
4'b0000:begin
 WBdata = 2'b00 ;
end

//   ADD
4'b0001:begin
 WBdata = 2'b00 ;
end

// SUB
4'b0010:begin
 WBdata = 2'b00 ;
```

```
 WBdata = 2'b00 ;
end

//   ADD
4'b0001:begin
 WBdata = 2'b00 ;
end

// SUB
4'b0010:begin
 WBdata = 2'b00 ;
end

// ADDI
4'b0011:begin
 WBdata = 2'b00 ;
 end

 // ANDI
4'b0100:begin
 WBdata = 2'b00 ;
 end
      endcase
  ////////////////////////////////////////////
 case ( Op )
 // BGT || BGTZ
  4'b1000:
 if ( (zerof == 0 && signf == 1)) begin
 PCSrc = 2'b10 ;
 end
 else begin
 PCSrc = 2'b00 ;
 end

  // BLT || BLTZ
4'b1001 :
 if ( (zerof == 0 && signf == 0)) begin
 PCSrc = 2'b10 ;
 end
 else begin
 PCSrc = 2'b00 ;
 end

  // BEQ || BEQZ
4'b1010 :
 if ( zerof == 1) begin
 PCSrc = 2'b10 ;
 end
 else begin
 PCSrc = 2'b00 ;
 end

 // BNE || BNEZ
4'b1011 :
 if ( zerof == 0) begin
 PCSrc = 2'b10 ;
 end
 else begin
 PCSrc = 2'b00 ;
 end

 4'b1100 : PCSrc = 2'b01 ; // JMP
 4'b1101 : PCSrc = 2'b01 ; // CALL
 4'b1110 : PCSrc = 2'b11 ; // RET
 default :
 PCSrc = 2'b00 ;
 endcase



case (Op)
        4'b1111: Rdata = 2'b10; // Sv -> 2nd choice in the mux
        4'b0111: Rdata = 2'b01;   //SW
        default: Rdata = 2'b00;
    endcase
    end
endmodule

```

This testbench verifies the Control Unit module's functionality in generating the correct control signals based on various operation codes. It tests a range of instructions to ensure that the control unit appropriately sets the control signals for each type of operation, confirming its critical role in directing the processor's actions.

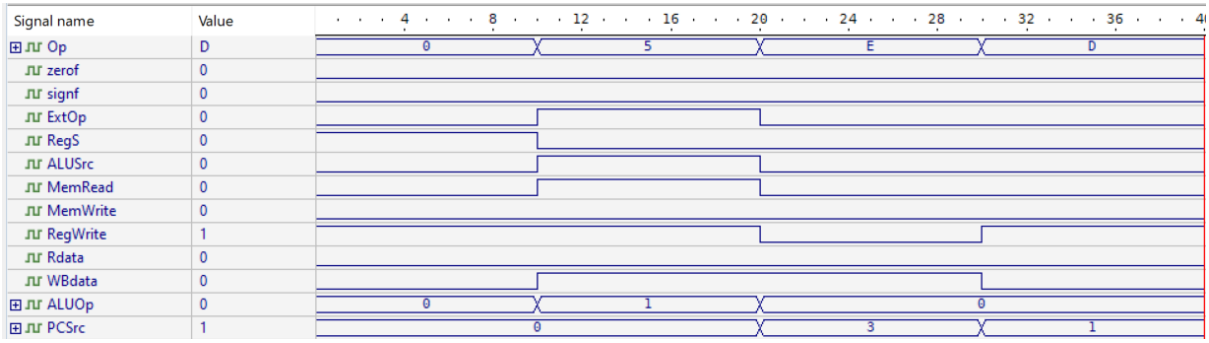


Figure 26: Control Unit Testbench

4.6 Clock Organizer

This module is responsible for organizing the clock signals for different stages of the processor's operation, such as instruction fetch, decode, execute, memory access, and write-back

```
module Clock_organizer(
    input clock,
    input [3:0] Op,
    output reg IF, ID, MEM, EX, WB, IFF
);

    // Opcode Parameters
    parameter AND = 4'b0000, ADD = 4'b0001, SUB = 4'b0010;
    parameter ADDI = 4'b0011, ANDI = 4'b0100, LW = 4'b0101,
              LBu = 4'b0110, LBs = 4'b0111, BGTZ = 4'b1000,
              BGT = 4'b1001, BLT = 4'b1010, BLTZ = 4'b1011, SW = 4'b0111,
              BEQ = 4'b1100, BEQZ = 4'b1101, BNEZ = 4'b1110, BNE = 4'b1111;
    parameter JMP = 4'b1100, CALL = 4'b1101, RET = 4'b1110;
    parameter Sv = 4'b1111;

    // Internal State
    reg [2:0] state;
    reg en = 0;

    // Initialization block
    initial begin
        state = 3'b000; // Initialize state
        IF = 1;         // Initialize all output signals
        ID = 0;
        MEM = 0;
        EX = 0;
        WB = 0;
        IFF = 0;
    end

    // State Machine
    always @(posedge clock) begin
        case (state)
            3'b000: begin
                state <= 3'b001;
                IF <= 0;
                ID <= 1;
                EX <= 0;
                MEM <= 0;
                WB <= 0;
                IFF <= 0;
            end

            3'b001: begin
```

```
43
44          3'b001: begin
45              if (Op == JMP  || Op == RET) begin
46                  state <= 3'b000;
47                  IFF <= 1;
48                  #2 IF <= 1;
49                  ID <= 0;
50                  EX <= 0;
51                  MEM <= 0;
52                  WB <= 0;
53              end else if (Op == Sv) begin
54                  state <= 3'b011;
55                  IF <= 0;
56                  ID <= 0;
57                  EX <= 0;
58                  MEM <= 1;
59                  WB <= 0;
60              end
61              else if (Op == CALL) begin
62                  state <= 3'b011;
63                  IF <= 0;
64                  ID <= 0;
65                  EX <= 0;
66                  MEM <= 0;
67                  WB <= 1;
68              end
69              else begin
70                  state <= 3'b010;
71                  IF <= 0;
72                  ID <= 0;
73                  EX <= 1;
74                  MEM <= 0;
75                  WB <= 0;
76              end
77          end
78
79          3'b010: begin
80              if (Op == BNE || Op == BEQ || Op == BGT || Op == BLT || Op == BGTZ || Op == BLTZ || Op == BEQZ || Op == BNEZ || Op== Sv || Op== CALL) begin
81                  state <= 3'b000;
82                  IFF <= 1;
83                  #2 IF <= 1;
84                  ID <= 0;
85                  EX <= 0;
86                  MEM <= 0;
```

```
                    state <= 3'b000;
                    IFF <= 1;
                    #2 IF <= 1;
                    ID <= 0;
                    EX <= 0;
                    MEM <= 0;
                    WB <= 0;
                end else if (Op == Lh || Op == LBu || Op == LBs || Op == Sh) begin
                    state <= 3'b011;
                    IF <= 0;
                    ID <= 0;
                    EX <= 0;
                    MEM <= 1;
                    WB <= 0;
                end else begin
                    state <= 3'b100;
                    IF <= 0;
                    ID <= 0;
                    EX <= 0;
                    MEM <= 0;
                    WB <= 1;
                end
            end

            3'b011: begin
                if (Op == Lh || Op == LBu || Op == LBs) begin
                    state <= 3'b100;
                    IF <= 0;
                    ID <= 0;
                    EX <= 0;
                    MEM <= 0;
                    WB <= 1;
                end else begin
                    state <= 3'b000;
                    IFF <= 1;
                    #2 IF <= 1;
                    ID <= 0;
                    EX <= 0;
                    MEM <= 0;
                    WB <= 0;
                end
            end

            3'b100: begin
                state <= 3'b000;
                IFF <= 1;
                #2 IF <= 1;
                ID <= 0;
                EX <= 0;
                MEM <= 0;
                WB <= 0;
            end
        endcase
    end
endmodule
```

The Clock Organizer module's testbench is focused on confirming the correct sequencing and timing of operations based on the operation codes. By simulating various instruction types, the testbench checks whether the module appropriately sets the clock signals for each stage of the instruction cycle, validating its role in synchronizing CPU operations.

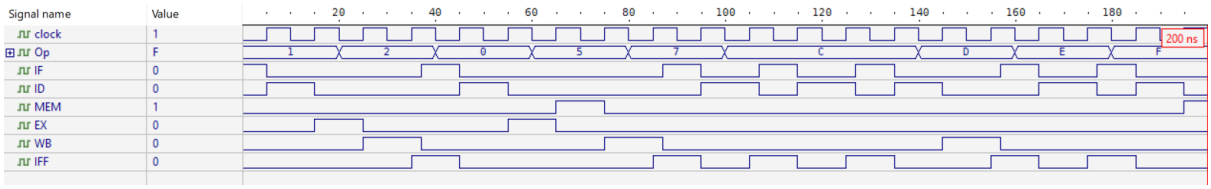


Figure 27: Clock Organizer Testbench

4.7 Program Counter (PC)

The PC module is designed to store and update a 16-bit address, typically used in a processor to keep track of the address of the next instruction to execute

```
module pc(
    input wire [15:0] input_data,
    output wire [15:0] output_data
);

    reg [15:0] register = 16'b0;

    always @(input_data) begin
        register <= input_data; // Load input data when it changes
    end

    assign output_data = register;

endmodule

//------------------------------

module PC_tb();

    reg [15:0] input_data;
    wire [15:0] output_data;

    PC pc (
        .input_data(input_data),
        .output_data(output_data)
    );

    initial begin
        input_data = 16'h1234; // Initial input value
        $display("Initial output_data = %h", output_data); // Display initial value
        #10;
        input_data = 16'hABCD;
        #10;
        input_data = 16'h8000;
        #10;
        $finish;
    end

    // Monitor to display output_data
    always @(output_data) begin
        $display("output_data = %h", output_data);
    end

endmodule
```

the testbench for the Program Counter (PC) module checks the module's ability to store and update its internal register based on the input data. It sequentially inputs different 16-bit values and verifies if the output matches the input after a clock cycle. This ensures that the PC correctly updates its value to point to the next instruction in a sequence

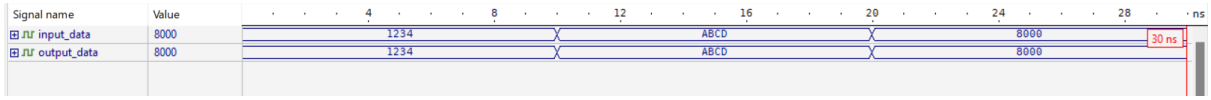


Figure 28: PC Testbench

4.8 Multiplexer 4x1

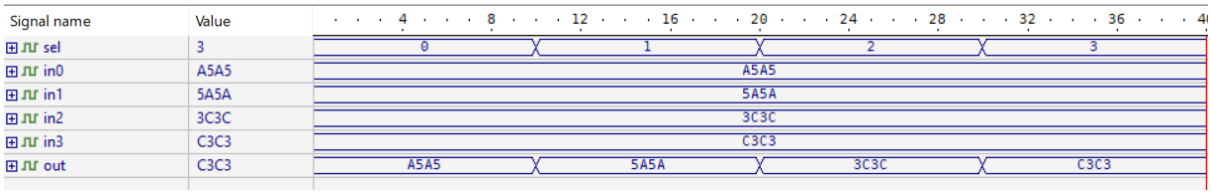
This module selects one of four 16-bit inputs based on a 2-bit select signal. It's more versatile for larger data selections.

```
module MUX4to1 (
    input wire [1:0] sel,
    input wire [15:0] in0,   // First 16-bit input
    input wire [15:0] in1,   // Second 16-bit input
    input wire [15:0] in2,   // Third 16-bit input
    input wire [15:0] in3,   // Fourth 16-bit input
    output reg [15:0] out    // 16-bit output
);

    always @(*) begin
        case (sel)
            2'b00: out = in0;
            2'b01: out = in1;
            2'b10: out = in2;
            2'b11: out = in3;
            default: out = 16'b0;
        endcase
    end

endmodule
```

Mux4x1_tb tests all possible select signal combinations to ensure the correct input is selected each time.



4.9 Multiplexer 2x1

This module selects one of two 16-bit inputs based on a single-bit select signal. Multiplexers are fundamental in digital circuits for routing signals.

```
        //Mux 2*1
module MUX2to1 (
    input wire sel,          // Select signal
    input wire [15:0] in0,   // First 16-bit input
    input wire [15:0] in1,   // Second 16-bit input
    output wire [15:0] out   // 16-bit output
);

    assign out = sel ? in1 : in0;

endmodule
```

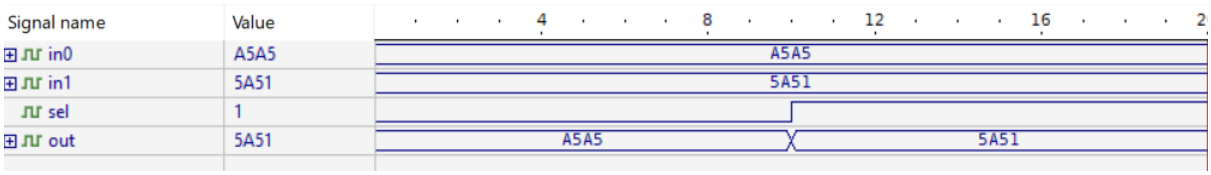


Figure 29: Mux2x1 Testbench

4.10 Adder

```
module Adder(
  input [15:0] A, B,
  output [15:0] out
);
  assign out = A + B; // Adds inputs A and B and assigns the result to out
endmodule

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

addition of different pairs of numbers, verifying the module correctly adds 16-bit values.

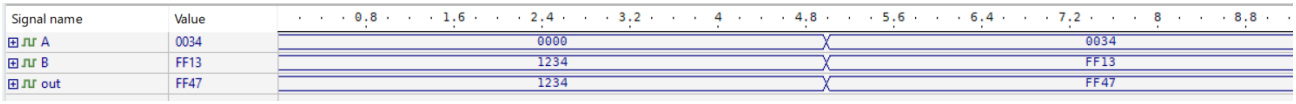


Figure 30: Two Input Adder Testbench

4.11 Concatenation

This module takes two inputs, A (12 bits) and B (4 bits), and concatenates them to form a 16-bit output. It's used in systems where bits from different sources need to be combined into a single word

```
module Concatenator(
  input [11:0] A,
  input [3:0] B,
  output [15:0] out //
);
  assign out = {B, A};
endmodule

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

The Concatenator_tb sets specific values to A and B, and initiates the simulation. It verifies that the concatenation of A and B produces the correct 32-bit output.

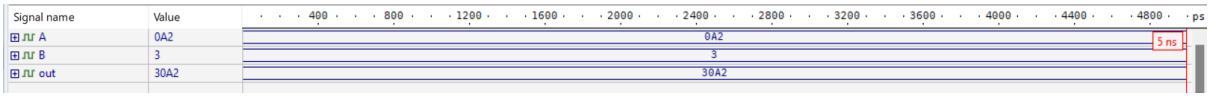


Figure 31: Concatenation Testbench

4.12 Complete Processor

In our data path we combined all the components together to work as multi-cycle processor, so we defined the important components such as control signals and wires to connect all components together sequentially, we started by the clock organizer to generate the stages for each instruction in our project and did the same for other components.

```
module datapath(input [3:0] Opc);
    reg CLK;
    reg [15:0] PCnext/*current pc */, PCnow, PCplus, PCplus4;
    reg [15:0] IR, BTA,BTAA, JA; //outCon;
    reg [2:0] Rs1, Rs2, Rd, RB,RB2,RB3,Rs,rs2;
    reg m;
    reg [4:0] Imm5;
    reg [11:0] Imm12;
    reg [11:0] Imm12S;
    reg [8:0] Imm8;
    reg [3:0] PCj,Op;
    reg [15:0] BusB, BusA, BusAA,BusBB, ExtImm, AluB, AluRes, Alu1,RETpc,AluA , MemAddre;
    reg [15:0] Data_in,Data_out, WB;
    reg clkIF, clkID, clkEX, clkMEM, clkWB, clkIFF;
    reg ExtOp,RegS, ALUSrc, MemRead, MemWrite, RegWrite,SvControl/*IMPORTANT*/, RETcontrol, CALLcontrol, RETcontrol2, ExtSel,BranchR0;
    reg [1:0] ALUOp, PCSrc, Rdata ,WBdata;
    reg zero, sign;
    parameter [2:0] r7_value = 3'b111;
    parameter [15:0] r0_value = 16'h0000;

    reg [15:0] PCnextt;  //This is the value of the pc for the next instruction
    wire [15:0] BusBt, BusAt;

    assign Op = IR[15:12];
    assign PCj = PCnow[15:12];


always @(*) begin
    case (Op)
        // R-type instructions
        4'b0000, 4'b0001, 4'b0010: begin
            Rd = IR[11:9];
            Rs1 = IR[8:6];
            Rs2 = IR[5:3];
        end

        // I-type instructions
        4'b0011, 4'b0100, 4'b0101,4'b0111,
        4'b1000, 4'b1001, 4'b1010, 4'b1011: begin
            m = IR[11]; // Example assignment

            Rd = IR[10:8];
            Rs1 = IR[7:5];
             Rs2 = IR[10:8];
            Imm5 = IR[4:0];

            $display("rs1= %h***", AluA);
            // Additional assignments specific to I-type if needed
        end

      4'b0110: begin
            m = IR[11]; // Example assignment

            Rd = IR[10:8];
            Rs1 = IR[7:5];
             Rs2 = IR[10:8];
            Imm5 = IR[4:0];


            $display("rs1= %h***", Rs1);
            // Additional assignments specific to I-type if needed
        end
```

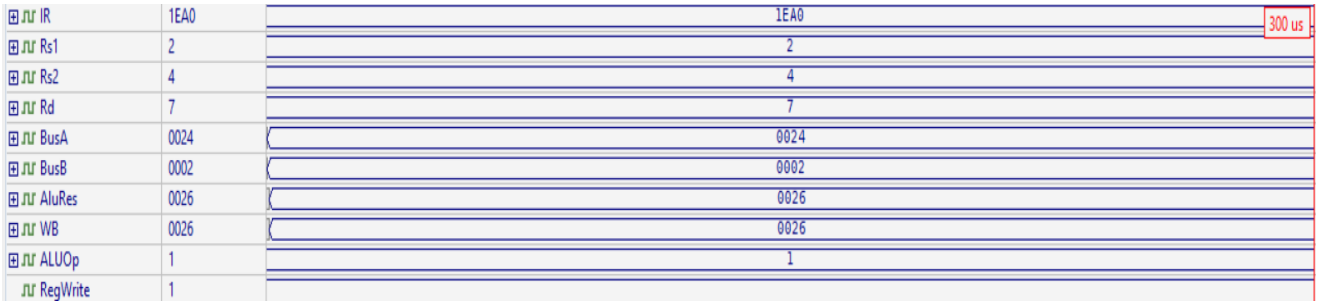
```
81
82      Clock_organizer clocorg(CLK, Op, clkIF, clkID, clkMEM, clkEX, clkWB,clkIFF);
83
84      pc get_pc(PCnext, PCnow);
85
86
87
88      InstructionMemory get_IR(clkIF, PCnext, IR);
89      Adder_2 adderPC(PCnow, PCplus);
90      ShiftLeftByOne shifter ( Imm12,Imm12S);
91      Concatenator cont(Imm12S, PCj, JA);
92
93      control_unit contuni(m,Op, zero, sign, ExtOp, RegS, ALUSrc, MemRead,  MemWrite, RegWrite,SvControl,RETcontrol, CALLcontrol, RETcontrol2, ExtSel, Br
94
95
96
97      MUX2to1 muxRET1(RETcontrol, Rs1, r7_value, RB2 /* will go to rs1*/);
98
99      MUX2to1 muxReg(RegS, Rs2, Rd, RB);
100     MUX2to1 muxRET2(CALLcontrol, RB, r7_value, RB3 /*will go to rd*/);
101
```

```
146
147     Adder adderBTA(ExtImm, PCnow, BTA);
148
149     MUX2to1 muxALU(ALUSrc, BusB, ExtImm, AluB);
150     Demux1to2 demuxALU(RETcontrol2, BusA, Alu1, RETpc);
151
152     MUX2to1 muxALU2(BranchR0, Alu1, r0_value, AluA);
153
154     ALU alu1(clkEX,AluA, AluB, ALUOp, AluRes, zero, sign);
155     MUX4to1 mux4PC(PCSrc, PCplus, JA, BTA, RETpc, PCnextt/* real input for pc*/);
156
157     MUX2to1 SvMux(SvControl,AluRes ,BusA ,MemAddre ) ; //SvControl mux
158     Mux3to1 RdataMux(Rdata, PCplus, BusB, ExtImm,Data_in);  //Rdata mux3to1
159
160     Memory datamem(clkMEM, MemAddre, MemWrite, MemRead, Data_in, Data_out);
161
162     Mux3to1 muxWB(WBdata, AluRes, Data_out, PCplus, WB);
163
```

Figure 32: data path code snippets

Test some instructions individually:

ADD:



This is the instruction:

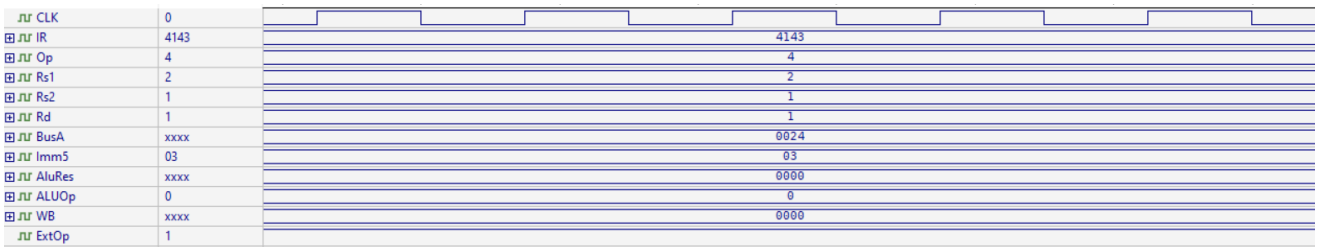
```
MemoryInst[1] = {ADD, R7, R2, R4, 3'b000};   //ADD
```

And this is the data memory:

```
initial begin
    registers[0] = 16'h0000;
    registers[1] = 16'h0012;
    registers[2] = 16'h0024;
    registers[3] = 16'h0033;
    registers[4] = 16'h0002;
    registers[5] = 16'h0000;
    registers[6] = 16'h0008;
    registers[7] = 16'h0006;
    end
```

The results obtained are correct for all R-Type instructions.

ANDI

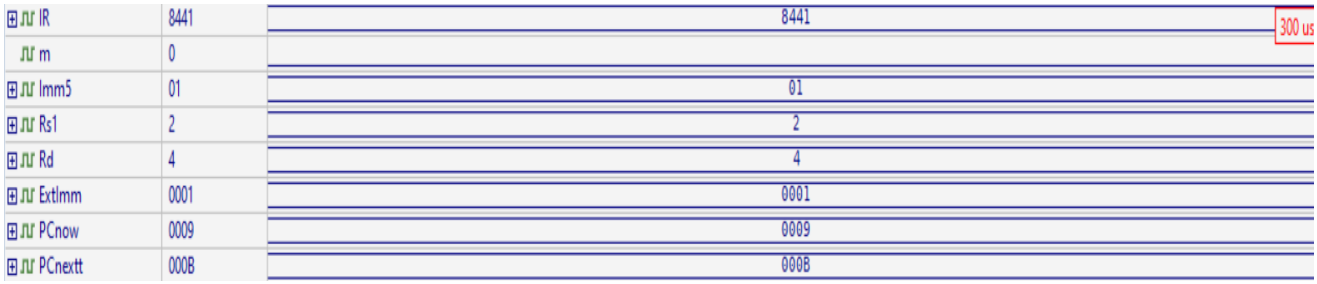


This is the instruction:

```
MemoryInst[4] = {ANDI,1'b0, R1, R2, 5'b00011};
```

The result obtained is correct.

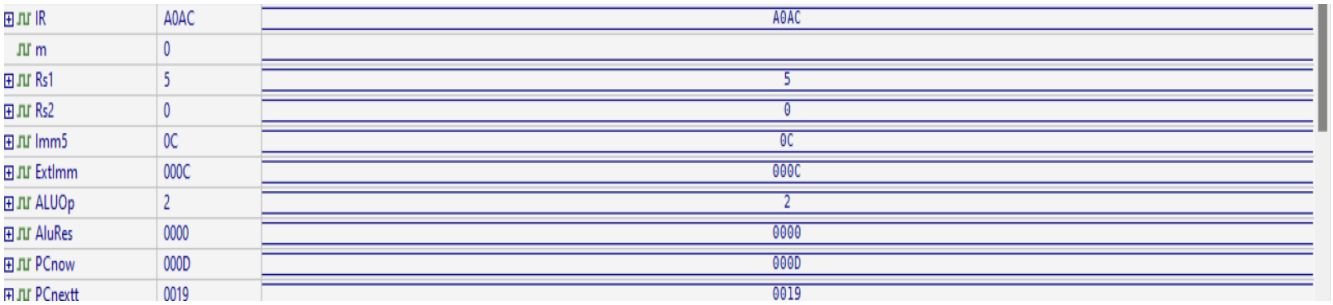
BGT




```
MemoryInst[9] = {BGT,1'b0, R4, R2, 5'b00001};
```

The result obtained is correct it made the next PC value equal to PC=PC+ 2

BEQ




```
MemoryInst[13] = {BEQ,1'b0, R0, R5, 5'b01100};
```

The result obtained is correct it made the next PC value equal to PC = PC + sign_extended (Imm)

All I-Type instructions worked correctly with no errors

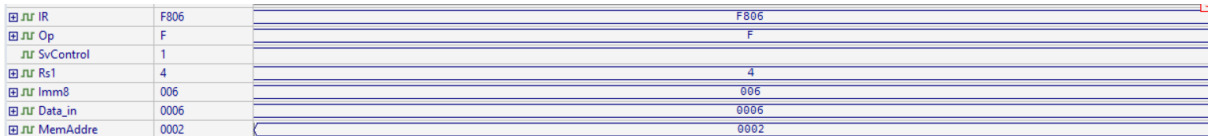
RET:


```
MemoryInst[19] = {RET, 12'b000000000001};
```

The result is correct, it made the next value of the PC equal to the value stored in R7

All J-type instructions work perfectly.

Sv:




```
MemoryInst[20] = {Sv, R4, 9'b000000110};
```

In this instruction we stored the immediate value in the memory address of Rs

Everything worked perfectly.

The whole System:

```
run
# KERNEL: Cycle: 1, PCnow: 0000, IR: 0770, RB2: 5, Rs2: 6, RB3: 3, BusA: 0000, BusB: 0008, AluRes: 0000, Data_out: xxxx, WB: 0000, RegWrite: 1, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0008, RETcontrol: 0, ExtImm: 0000, Rdata: 0, Data_out: xxxx, MemAddre: 0000, Data_in: 0002, m: x, MemWrite: 0, PCSrc: 0, PCnext: 0000, PCnextt: 0002, BranchR0: 0
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 2, PCnow: 0002, IR: 23a8, RB2: 6, Rs2: 5, RB3: 1, BusA: 0008, BusB: 0000, AluRes: 0008, Data_out: xxxx, WB: 0008, RegWrite: 1, WBdata: 0, ALUSrc: 0, AluA: 0008, AluB:
0000, RETcontrol: 0, ExtImm: 0000, Rdata: 0, Data_out: xxxx, MemAddre: 0008, Data_in: 0004, m: x, MemWrite: 0, PCSrc: 0, PCnext: 0002, PCnextt: 0004, BranchR0: 0
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 3, PCnow: 0004, IR: 4143, RB2: 2, Rs2: 1, RB3: 1, BusA: 0008, BusB: 0000, AluRes: 0008, Data_out: xxxx, WB: 0008, RegWrite: 1, WBdata: 0, ALUSrc: 1, AluA: 0008, AluB:
0003, RETcontrol: 0, ExtImm: 0003, Rdata: 0, Data_out: xxxx, MemAddre: 0008, Data_in: 0006, m: 0, MemWrite: 0, PCSrc: 0, PCnext: 0004, PCnextt: 0006, BranchR0: 0
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 4, PCnow: 0006, IR: 4143, RB2: 2, Rs2: 1, RB3: 1, BusA: 0024, BusB: 0008, AluRes: 0000, Data_out: xxxx, WB: 0000, RegWrite: 1, WBdata: 0, ALUSrc: 1, AluA: 0024, AluB:
0003, RETcontrol: 0, ExtImm: 0003, Rdata: 0, Data_out: xxxx, MemAddre: 0000, Data_in: 0008, m: 0, MemWrite: 0, PCSrc: 0, PCnext: 0006, PCnextt: 0008, BranchR0: 0
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 5, PCnow: 0008, IR: 7641, RB2: 2, Rs2: 6, RB3: 6, BusA: 0024, BusB: 0008, AluRes: 0025, Data_out: xxxx, WB: 0025, RegWrite: 0, WBdata: 0, ALUSrc: 1, AluA: 0024, AluB:
0001, RETcontrol: 0, ExtImm: 0001, Rdata: 1, Data_out: xxxx, MemAddre: 0025, Data_in: 0008, m: 0, MemWrite: 1, PCSrc: 0, PCnext: 0008, PCnextt: 000a, BranchR0: 0
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 6, PCnow: 000a, IR: 8aa3, RB2: 5, Rs2: 2, RB3: 2, BusA: 0024, BusB: 0008, AluRes: 0025, Data_out: xxxx, WB: 0025, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0008, RETcontrol: 0, ExtImm: 0003, Rdata: 0, Data_out: xxxx, MemAddre: 0025, Data_in: 000c, m: 1, MemWrite: 0, PCSrc: 0, PCnext: 000a, PCnextt: 000c, BranchR0: 1
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 7, PCnow: 000c, IR: 9d4a, RB2: 2, Rs2: 5, RB3: 5, BusA: 0000, BusB: 0024, AluRes: ffdc, Data_out: xxxx, WB: ffdc, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0024, RETcontrol: 0, ExtImm: 000a, Rdata: 0, Data_out: xxxx, MemAddre: ffdc, Data_in: 000e, m: 1, MemWrite: 0, PCSrc: 0, PCnext: 000c, PCnextt: 000e, BranchR0: 1
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 8, PCnow: 000e, IR: ab41, RB2: 2, Rs2: 3, RB3: 3, BusA: 0024, BusB: 0000, AluRes: 0000, Data_out: xxxx, WB: 0000, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0000, RETcontrol: 0, ExtImm: 0001, Rdata: 0, Data_out: xxxx, MemAddre: 0000, Data_in: 0010, m: 1, MemWrite: 0, PCSrc: 2, PCnext: 000e, PCnextt: 000f, BranchR0: 1
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 9, PCnow: 000f, IR: ab41, RB2: 2, Rs2: 3, RB3: 3, BusA: 0024, BusB: 0000, AluRes: 0000, Data_out: xxxx, WB: 0000, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0000, RETcontrol: 0, ExtImm: 0001, Rdata: 0, Data_out: xxxx, MemAddre: 0000, Data_in: 0011, m: 1, MemWrite: 0, PCSrc: 2, PCnext: 000f, PCnextt: 0010, BranchR0: 1
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 10, PCnow: 0010, IR: bc41, RB2: 2, Rs2: 4, RB3: 4, BusA: 0024, BusB: 0002, AluRes: fffe, Data_out: xxxx, WB: fffe, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0000, AluB:
0002, RETcontrol: 0, ExtImm: 0001, Rdata: 0, Data_out: xxxx, MemAddre: fffe, Data_in: 0012, m: 1, MemWrite: 0, PCSrc: 2, PCnext: 0010, PCnextt: 0011, BranchR0: 1
# KERNEL:
# KERNEL:
# KERNEL: Cycle: 11, PCnow: 0011, IR: c006, RB2: 2, Rs2: 4, RB3: 4, BusA: 0024, BusB: 0002, AluRes: fffe, Data_out: xxxx, WB: fffe, RegWrite: 0, WBdata: 0, ALUSrc: 0, AluA: 0024, AluB:
0002, RETcontrol: 0, ExtImm: 0000, Rdata: 0, Data_out: xxxx, MemAddre: fffe, Data_in: 0013, m: 1, MemWrite: 0, PCSrc: 1, PCnext: 0011, PCnextt: 000c, BranchR0: 0
```

Activate Windows

These are the results we obtained from running the whole system (multi cycle processor) and the results are correct.

Teamwork

Our team adopted a collaborative approach to design and build various components of our project. We began by dividing the tasks for writing the main components' code of the Datapath (Register File, Data Memory, Instruction Memory, PC) and developing the test benches among ourselves. This joint effort laid a solid foundation for our work.

Initially, Haneen was responsible for the Register File, Multiplexer, and Adder. Rana handled the Instruction Memory and ALU, while Mays took charge of the Data Memory and Program Counter (PC). Subsequently, we gathered online daily for extended sessions to refine our designs and work on the Control Unit and Clock Organizer, as well as concatenation. We ensured that these critical components received our collective attention, adding multiplexers as needed to achieve the highest precision in our Datapath.

In the final stage, we interconnected all the parts, culminating in a design and final code that reflected our united efforts. Additionally, we distributed the creation of the project report: Mays was responsible for the Introduction and Components Description sections, Haneen handled the Design and Implementation part, and Rana took charge of the Testing and Validation section. Throughout the process, we exemplified shared responsibility and unity, maintaining a strong team spirit from start to finish.

Conclusion

The Multi-Cycle Processor architecture represents a significant advancement in processing efficiency and flexibility. By dividing complex instructions into multiple cycles, it optimizes hardware utilization and speeds up instruction execution.

In our implementation of the multicycle processor, we utilized a finite state machine (FSM) to control the different stages of instruction execution, including Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (M), and Write Back (WB). This approach allowed us to handle each stage sequentially, ensuring precise control over the execution flow. Key components such as the ALU, multiplexers, and the extender were effectively coordinated by the control unit to execute instructions correctly over multiple cycles. Through this project, we have gained a deeper understanding of the MIPS CPU, particularly in terms of module connectivity and testing.