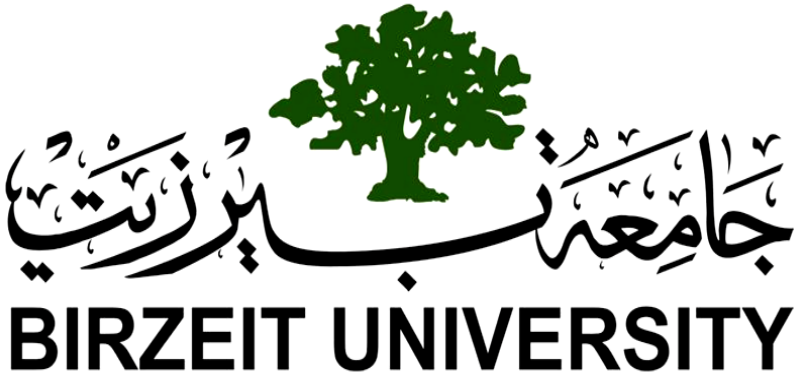


Faculty of Engineering & Technology
Electrical & Computer Engineering Department

COMPUTER ARCHITECTURE – ENCS4370

Project #2

Simple Multi-Cycle RISC Processor

Prepared by:

Hatem Hussein	1200894
Ali Faqy	1200048
Nafe Abubaker	1200047

Instructor: Dr. Aziz Qaroush

Section: 2

Date: 29-01-2024

Abstract/Objectives

In simpler terms, this project is about creating and implementing a multi-cycle MIPS RISC processor as part of a Computer Architecture course. The processor is designed based on the MIPS Reduced Instruction Set Computer (RISC) architecture, using a subset of MIPS instructions. The processing is divided into five cycles: instruction fetch, decode, execute, memory access, and writeback, each completed in one clock cycle. The Register-Transfer Level (RTL) description details how different instruction types (R-Type, I-Type, J-Type, and S-Type) are represented. The control unit is essential for generating control signals, ensuring proper coordination for efficient instruction execution. In summary, the project aims to create an effective multi-cycle processor supporting various instruction types and functions.

Table of Contents

List of figures

Design and Implementation

The design required multiple main components to design the full multi-cycle RISC processor considering all datapath and controls requirements:

1. **Program Counter (PC)**

The program counter component has an input address with 32bit long. Moreover, it has two input signals which are clk and EN.

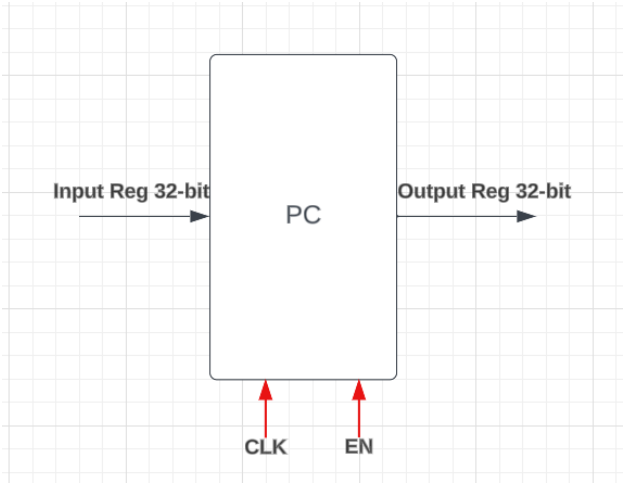


Figure 1: PC module

2. **Instruction Memory (IM)**

The IM component is a main component in our complete design. The IM plays the role of the memory that saves the instructions as it has an input called Address of 32bit long with two outputs of 32 bits called Instruction and outputPC.

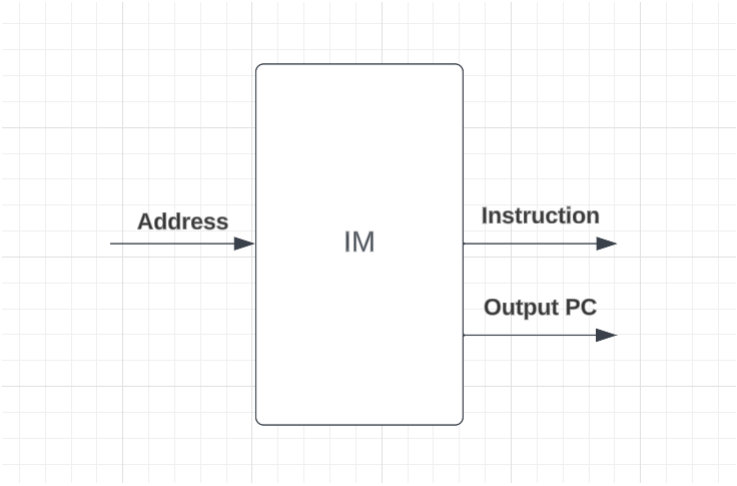


Figure 2: Instruction Memory module

3. Instruction Register (IR)

This Verilog module, named IR, represents an Instruction Register in our design. It takes input signals such as the clock (clk), a control signal for writing to the register (IRWrite), and a 32-bit input register (inputRegister). The module outputs various fields of the instruction, including a 6-bit opcode (opcode), two 4-bit registers (Rs1 and Rd), a 16-bit immediate value (imm), and a 2-bit mode indicator (modeBits). The always @* block triggers when there's a write request (IRWrite is high) and updates the output registers based on specific bit ranges of the input register. In essence, this module extracts and stores different components of an instruction for further processing in the full design.

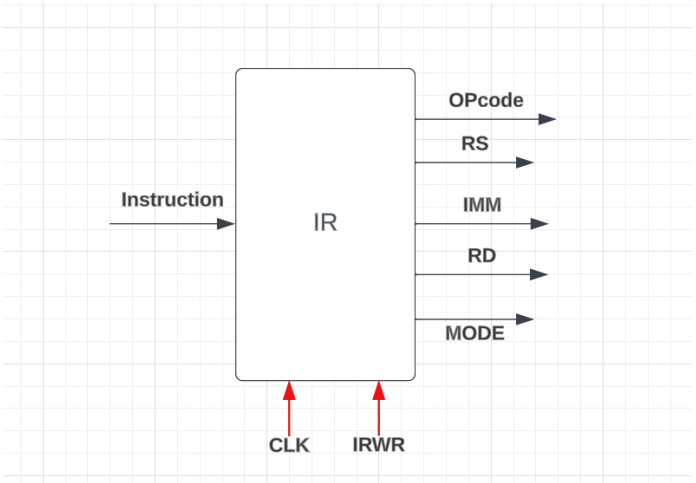


Figure 3: IR module

4. Register File (RF)

The RegisterFile module simulates a register file in the full design. It takes inputs such as clock (clk), signals for register write (regWR), read register addresses (readRegisterOne and readRegisterTwo), destination register address (destRegister), and data to be written (writeData). Additionally, there's a mode flag (modeFlag) that, when activated, increments the value read from the first register. The module features a 32-bit register file (registers) with 16 registers initialized to zero. The initial block ensures the registers are set to zero during initialization. The always block continuously assigns values from the register file to the output registers (readDataOne and readDataTwo) based on the specified read register addresses. The always block triggered by the rising edge of the clock updates the specified destination register with the provided data if the register write signal is active (regWR is high). If the mode flag is active, it increments the value read from the first register.

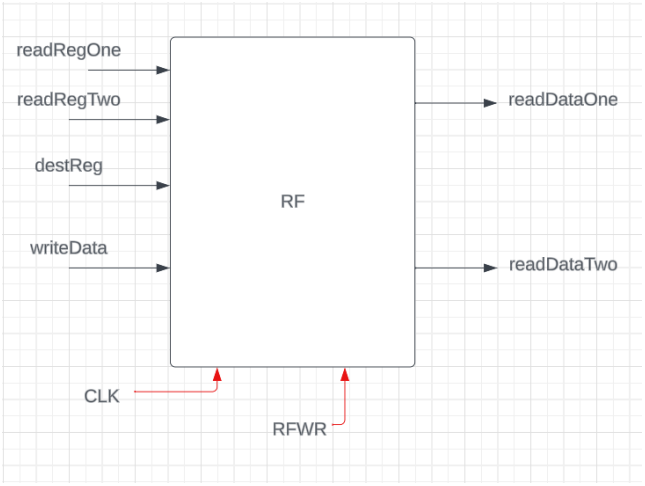


Figure 4: RF module

5. ALU

The ALU module represents a digital circuit that performs arithmetic and logical operations on two operands. It takes inputs operandOne and operandTwo, as well as an opcode that specifies the operation to be executed. The module has outputs including the result (ALUOut) and various flags such as zeroFlag, carryFlag, negativeFlag, greaterThan, and lessThan.

The always block uses a case statement to determine the operation based on the opcode. For R-Type operations, it performs bitwise AND, addition with carry flag detection, and subtraction. For I-Type operations, it includes bitwise AND, addition with carry flag, addition with carry flag, addition with carry flag, comparison for greater than, comparison for less than, and subtraction. For J-Type operations, it performs addition with carry flag detection.

Following the operation, the block checks if ALUOut is equal to zero and sets the zeroFlag accordingly. It also checks if ALUOut is negative and sets the negativeFlag. The module simulates the behavior of an ALU in a digital system, supporting various arithmetic and logical operations with associated flags for result interpretation.

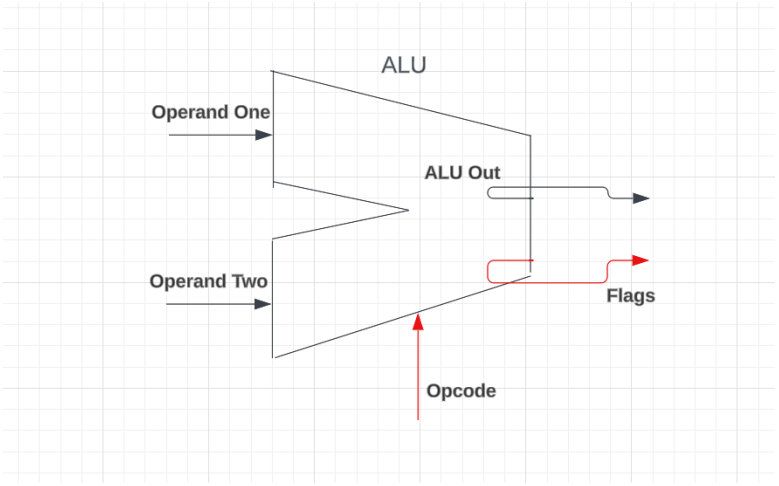


Figure 5: ALU module

6. Data Memory (DM)

The DataMemo module simulates a memory unit with stack functionality. It includes inputs such as clock (clk), address, data input (dataIn), read and write control signals (memoR and memoWR), as well as control signals for stack operations (pop and push). Outputs include the data output (dataOut) and various flags indicating the status of the stack and memory.

The module maintains two sets of memory arrays, 'staticMemo' and 'stackMemo,' each with ***512*** 32-bit elements. It also includes a stack pointer (sp) of 9 bits, tracking the current position in the stack. Flags like emptyStackFlag, fullStackFlag, emptyStaticFlag, and fullStaticFlag are used to indicate the status of the stack and static memory.

In the always @(posedge clk) block, the module updates the stack pointer based on push and pop operations, as well as writes and reads to the static memory. Additionally, it determines the status of the stack and static memory flags.

The subsequent always blocks manage the flags based on the stack pointer and empty stack conditions. The module properly updates flags like emptyStack, fullStack, emptyStatic, and fullStatic based on the stack pointer and stack conditions.

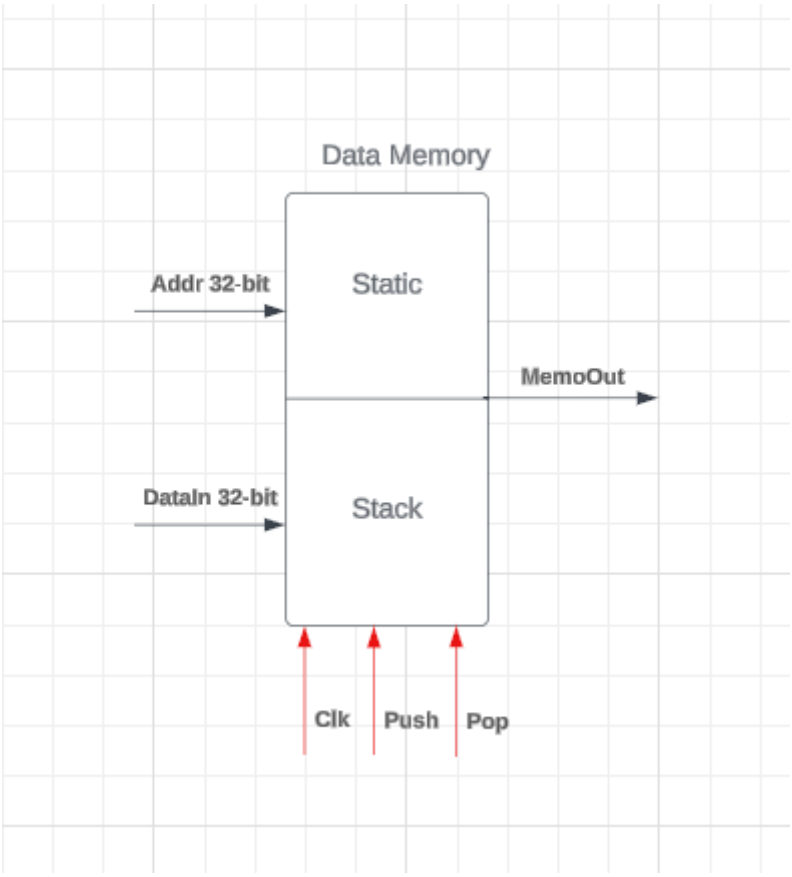


Figure 6: Data memory module

7. Control Unit

The module "ControlUnit" is a control unit for a multi-cycle microprocessor. It takes various control signals as inputs and produces control signals for different functional units of the processor. The control signals include opcode, modeBits, flags (e.g., isStackEmpty, isStackFull, etc.), and other control signals related to the execution of instructions.

The module uses a finite-state machine (FSM) to manage the different states of the processor, transitioning between them based on the opcode and other conditions. The states include RS (Reset), IF (Instruction Fetch), ID (Instruction Decode), EX (Execution), MEM (Memory Access), and WB (Write Back).

The module defines parameters for different opcodes, including R-Type, I-Type, J-Type, and S-Type instructions. It also includes logic for different operations such as AND, ADD, SUB, ANDI, ADDI, LW, LWPOI, SW, BGT, BLT, BEQ, BNE, JMP, CALL, RET, PUSH, and POP.

In each state, the module sets various control signals such as EN (Enable), IRWR (Instruction Register Write), MemoR (Memory Read), MemoWR (Memory Write), pop, push, MemoToRF, ReadRegisterTwoSrc, BranchFlag, isStackAddress, PCSource, OpcodeOut,

ALUSrcTwo, ALUSrcOne, RFWR (Register File Write), ExtensionSrc, regWriteOne, regWriteTwo, callFlag, and modeFlag based on the current opcode and other conditions.

The module uses an always block triggered on the positive edge of the clock to update the state and control signals. The control unit determines the appropriate control signals for each instruction and manages the transitions between states accordingly.

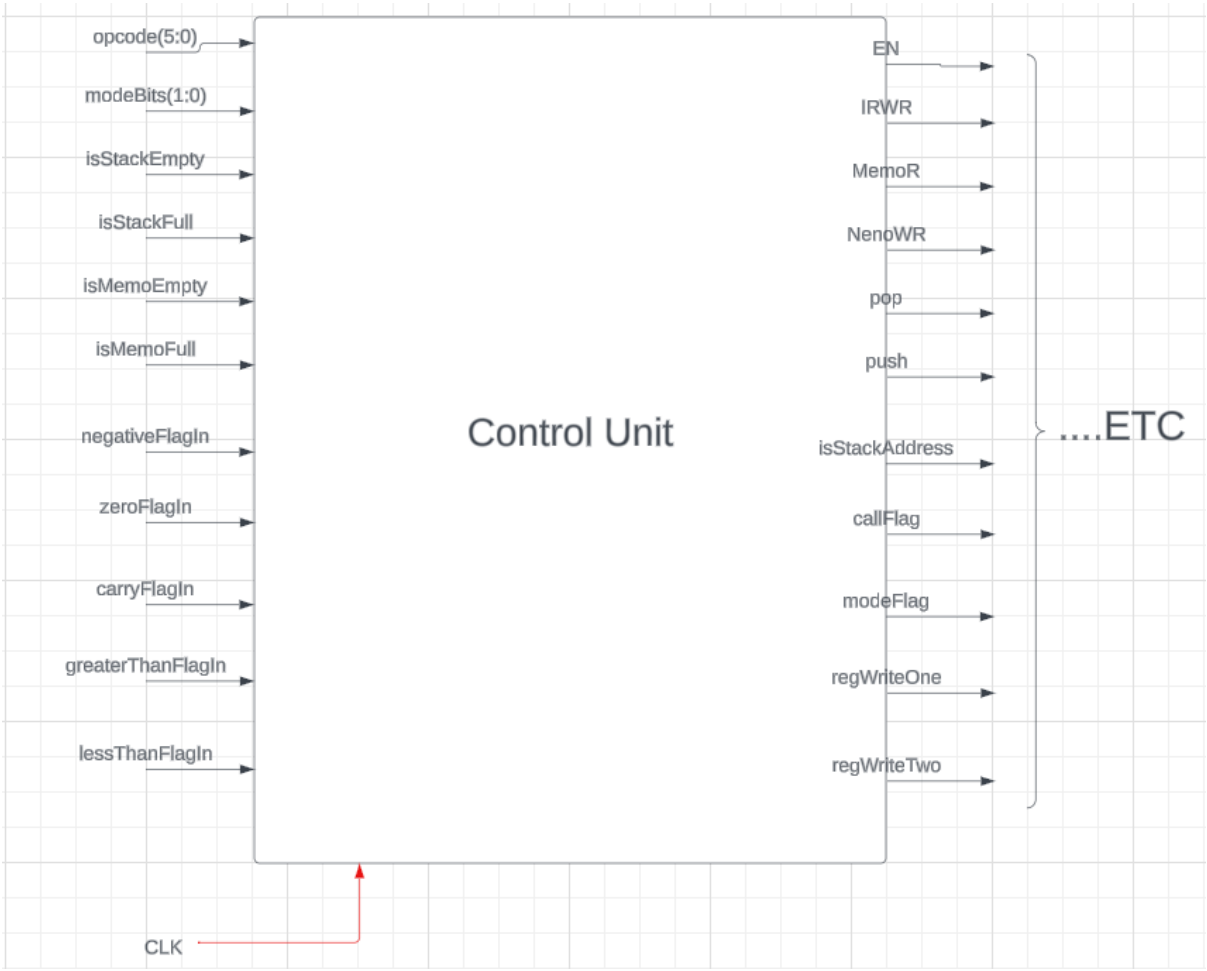


Figure 7: Control Unit module

8. immToRs2

The Verilog module named "immToRs2" receives a 16-bit immediate value (imm) as input and produces a 4-bit output value (RS2). In the always block, which activates upon any change in the input, the module assigns the upper 4 bits (imm[15:12]) of the immediate value to the output register RS2. This module essentially extracts the specified bits from the immediate value and outputs them as the RS2 value to be used in other operations.

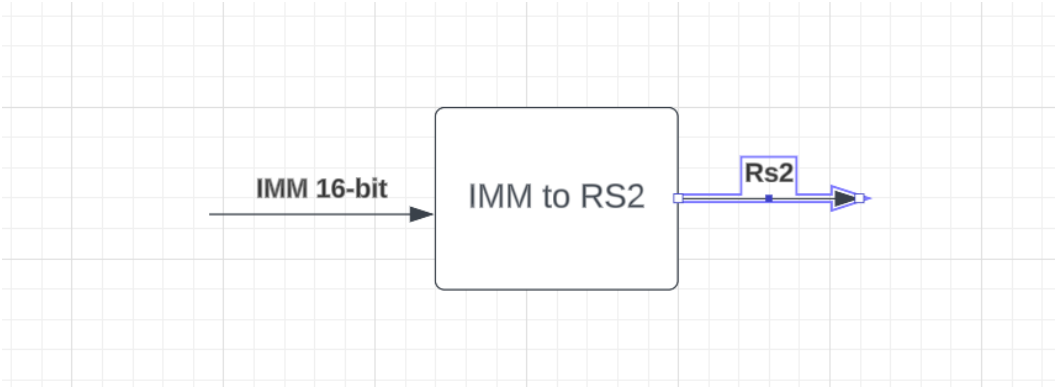


Figure 8: immToRs2 module

9. immTo26

The module "immTo26" takes three inputs: a 4-bit value for rs1, a 4-bit value for rd, and a 16-bit immediate value imm16. It produces a 32-bit output value immOut. The module outputs a 32-bit containing the 26-bit immediate value to be used in the jump instructions.

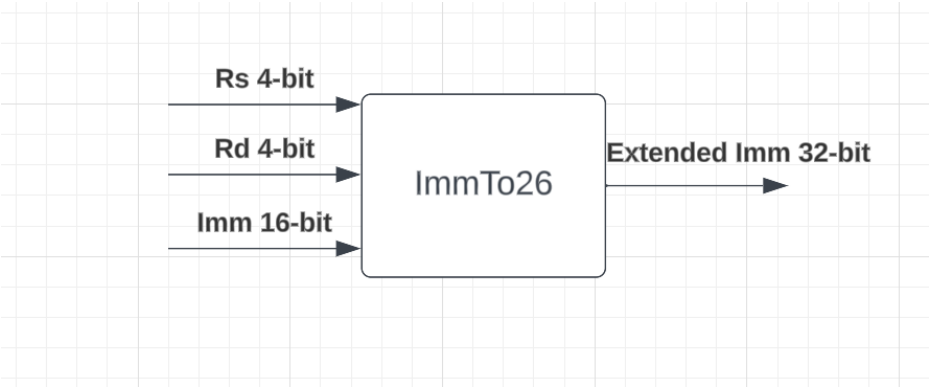


Figure 9: immTo26 module

10. SignExtender

The module "SignExtender" takes a 16-bit input value (extend) and outputs a 32-bit extended value (extended). This module effectively sign-extends the 16-bit input to a 32-bit value based on the sign bit.

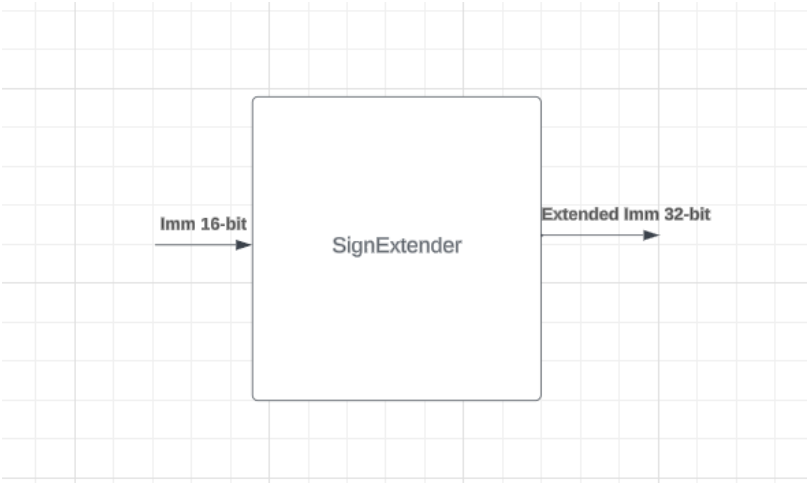


Figure 10: immSignExtender module

11. UnsignExtender

The module "UnsignExtender" takes a 16-bit input value (extend) and outputs a 32-bit extended value (extended). This module essentially extends an unsigned 16-bit value to a 32-bit value by zero-padding the upper 16 bits.

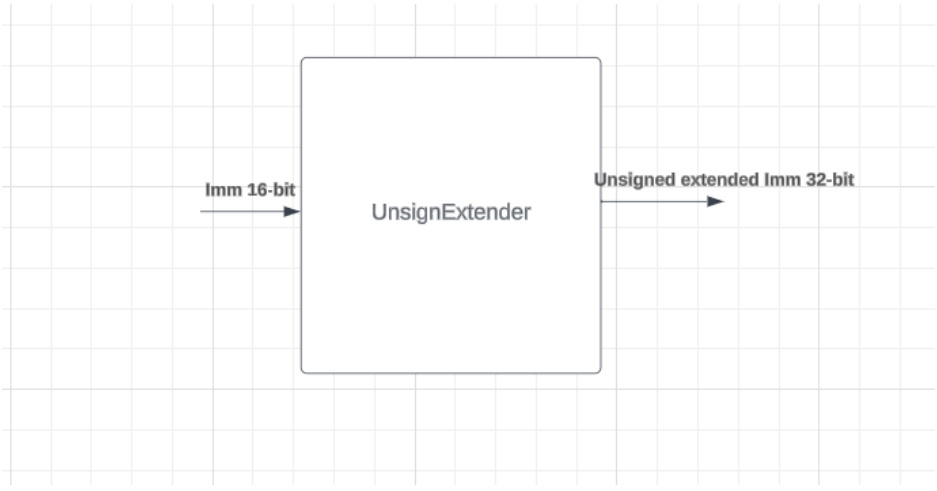


Figure 11: immUnsignExtender module

12. Adder

The module named "Adder" takes a 32-bit PC (Program Counter) input, a 16-bit immediate value (imm16), and outputs a 32-bit Branch Target Address (BTA). The module includes a wire named "extendedImm" to concatenate 16 zero bits to the most significant bits of imm16.

In the always block, which triggers on any change in the inputs, the module calculates the Branch Target Address (BTA) using non-blocking assignment. It adds the PC to the extended immediate value (extendedImm), resulting in a 32-bit sum stored in the output register BTA. This module essentially performs the addition of the PC and an extended immediate value to calculate the Branch Target Address.

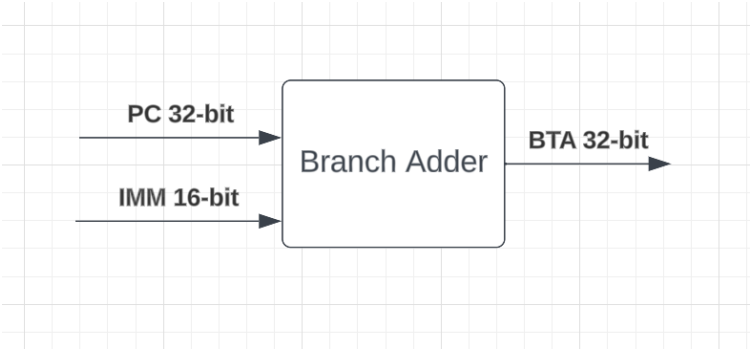


Figure 12: Branch Adder module

13. Buffer

The buffer module used multiple times named as "RFBufferOne", "RFBufferTwo", "ALUBuffer", and "MemoBuffer". The use of buffers came to simulate the logic of a multi-cycle processor in order to give the important data and signals from stage into another. It takes two inputs: a control signal for register writing (regWriteBuffOne) and a 32-bit input value (regIn). It outputs a 32-bit value (regOut). This module essentially acts as a buffer for register data, copying the input value to the output when the write signal is asserted.

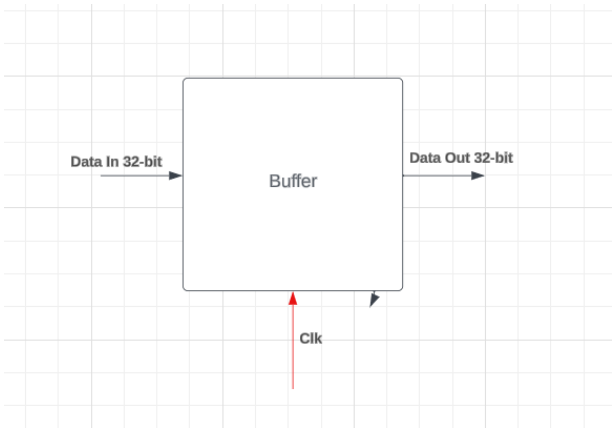


Figure 13: Buffer module

14. Mux2x1

The module "Mux2x1" is a 2-to-1 multiplexer. It has two 32-bit input wires (a and b), a select signal (sel), and a 32-bit output wire (out). This module essentially functions as a 2-to-1 multiplexer, allowing the selection of one of the two input values based on the control signal.

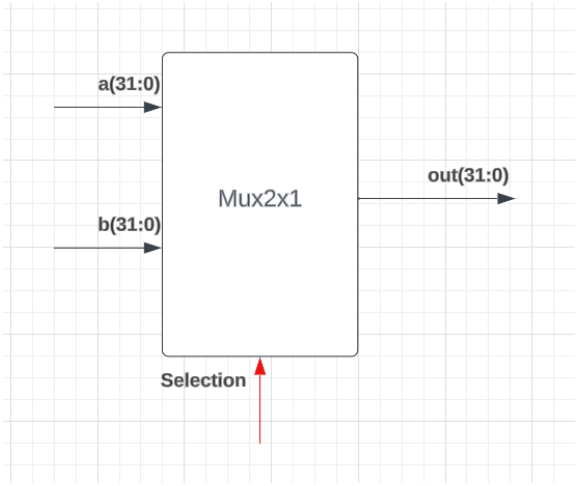


Figure 14: Mux2x1 module

15. Mux4x1

The module "Mux4x1" is a 4-to-1 multiplexer. It has four 32-bit input wires (a, b, c, and d), two select lines (sel) forming a 2-bit selection signal, and a 32-bit output wire (out). The assign statement utilizes a conditional operator to select one of the four inputs based on the binary value represented by the select lines.

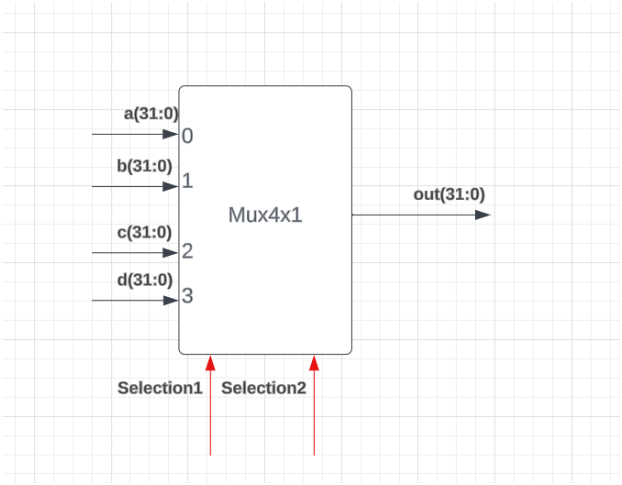


Figure 15: Mux4x1 module

FULL DESIGN

The below hand drawing represents the full design of the processor with taking into considerations each component illustrated previously:

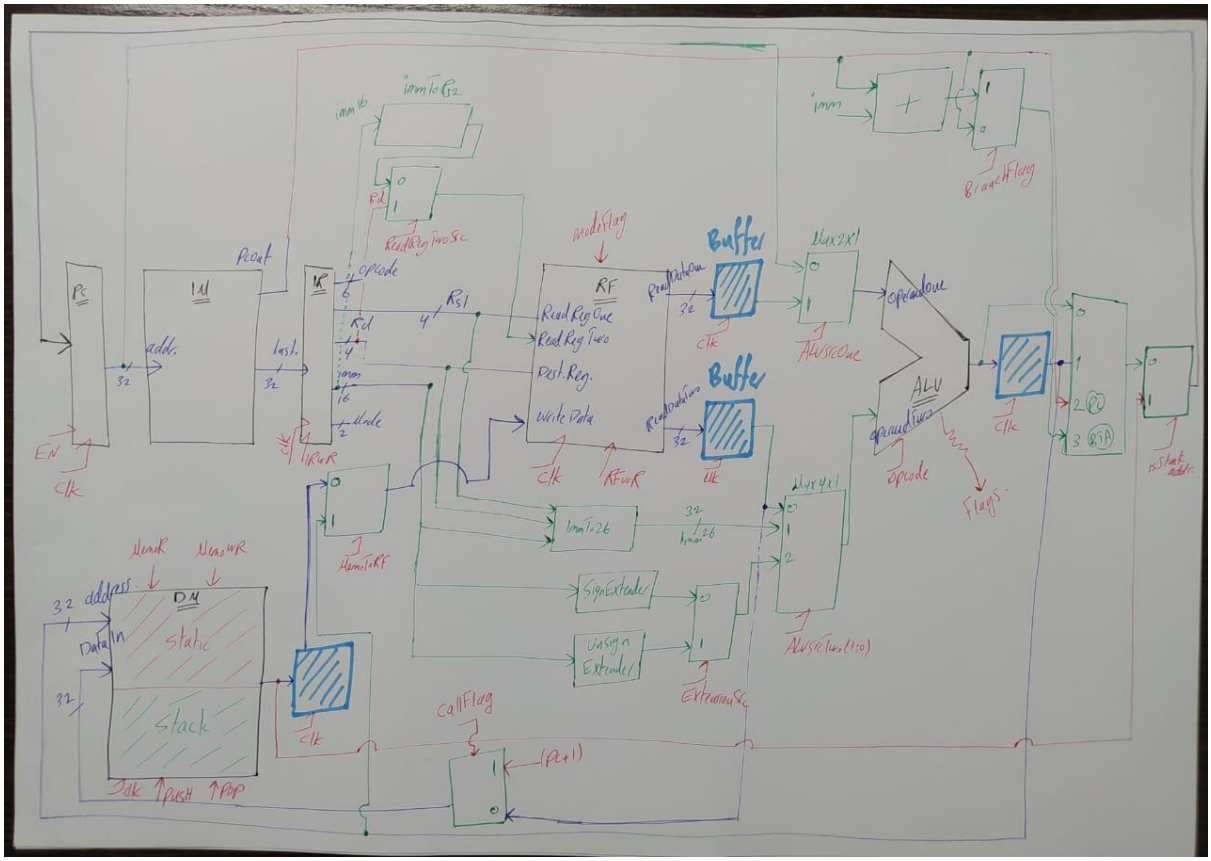


Figure 16: Full Design

NOTICE the below drive link for a clear image.

https://drive.google.com/file/d/1Pqhf4pMqfeIyg_bZzlwpvV9xObPqj-OK/view?usp=sharing

Control Signals

The Verilog module "ControlUnit" is designed to manage the control signals and states in a processor's datapath for various instructions. The state transitions and control signal assignments are determined based on the opcode and other conditions, reflecting the dynamic nature of the datapath as it processes different instructions. The tables help provide an overview of how the datapath behaves for each instruction, with the actual manipulation occurring during the states as the processor executes instructions.

1. AND

EN	**0**
IRWR	0
ReadRegTwoSrc	0
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	000000
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	X
MemoToRF	1
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

2. ADD

EN	**0**
IRWR	0
ReadRegTwoSrc	0
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	000001
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	X
MemoToRF	1
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

3. SUB

EN	**0**
IRWR	0
ReadRegTwoSrc	0
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	000010
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	X
MemoToRF	1
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

4. ANDI

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	10
Opcode(5:0)	000011
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	1
MemoToRF	1
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

5. ADDI

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	10
Opcode(5:0)	000100
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	0
MemoToRF	1
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

6. LW

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	10
Opcode(5:0)	000101
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	0
MemoToRF	0
MemoR	1
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

7. LWPOI

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	1
ALUSrcOne	1
ALUSrcTwo(1:0)	10
Opcode(5:0)	000110
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	0
MemoToRF	0
MemoR	1
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	1

8. SW

EN	**0**
IRWR	0
ReadRegTwoSrc	1
RFWR	0
ALUSrcOne	1
ALUSrcTwo(1:0)	10
Opcode(5:0)	000111
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	0
MemoToRF	X
MemoR	0
MemoWR	1
Pop	0
Push	0
callFlag	0
modeFlag	0

9. BGT

EN	**0**
IRWR	0
ReadRegTwoSrc	1
RFWR	0
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	001000
BranchFlag	1
PCSrc(1:0)	11
isStackAddress	0
ExtensionSrc	0
MemoToRF	X
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

10. BLT

EN	**0**
IRWR	0
ReadRegTwoSrc	1
RFWR	0
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	001001
BranchFlag	1
PCSrc(1:0)	11
isStackAddress	0
ExtensionSrc	0
MemoToRF	X
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

11. BEQ

EN	**0**
IRWR	0
ReadRegTwoSrc	1
RFWR	0
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	001010
BranchFlag	1
PCSrc(1:0)	11
isStackAddress	0
ExtensionSrc	0
MemoToRF	X
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

12. BNE

EN	**0**
IRWR	0
ReadRegTwoSrc	1
RFWR	0
ALUSrcOne	1
ALUSrcTwo(1:0)	00
Opcode(5:0)	001011
BranchFlag	1
PCSrc(1:0)	11
isStackAddress	0
ExtensionSrc	0
MemoToRF	X
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

13. JMP

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	0
ALUSrcOne	0
ALUSrcTwo(1:0)	01
Opcode(5:0)	001100
BranchFlag	X
PCSrc(1:0)	00
isStackAddress	0
ExtensionSrc	X
MemoToRF	0
MemoR	0
MemoWR	0
Pop	0
Push	0
callFlag	0
modeFlag	0

14. CALL

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	0
ALUSrcOne	0
ALUSrcTwo(1:0)	01
Opcode(5:0)	001101
BranchFlag	X
PCSrc(1:0)	00
isStackAddress	0
ExtensionSrc	X
MemoToRF	0
MemoR	0
MemoWR	0
Pop	0
Push	1
callFlag	1
modeFlag	0

15. RET

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	0
ALUSrcOne	0
ALUSrcTwo(1:0)	01
Opcode(5:0)	001110
BranchFlag	X
PCSrc(1:0)	00
isStackAddress	1
ExtensionSrc	X
MemoToRF	0
MemoR	0
MemoWR	0
Pop	1
Push	0
callFlag	0
modeFlag	0

16. PUSH

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	0
ALUSrcOne	X
ALUSrcTwo(1:0)	X
Opcode(5:0)	001111
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	X
MemoToRF	X
MemoR	0
MemoWR	0
Pop	0
Push	1
callFlag	0
modeFlag	0

17. POP

EN	**0**
IRWR	0
ReadRegTwoSrc	X
RFWR	1
ALUSrcOne	X
ALUSrcTwo(1:0)	X
Opcode(5:0)	010000
BranchFlag	X
PCSrc(1:0)	10
isStackAddress	0
ExtensionSrc	X
MemoToRF	1
MemoR	0
MemoWR	0
Pop	1
Push	0
callFlag	0
modeFlag	0

RTL Description

Instruction	RTL Description
AND	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rs2 EX: opOne && opTwo WB: Reg[Rd] = ALUout
ADD	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rs2 EX: opOne + opTwo WB: Reg[Rd] = ALUout
SUB	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rs2 EX: opOne - opTwo WB: Reg[Rd] = ALUout
ANDI	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = immExtended EX: opOne && opTwo WB: Reg[Rd] = ALUout
ADDI	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = immExtended

	EX: opOne + opTwo WB: Reg[Rd] = ALUout
LW	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = immExtended EX: opOne + opTwo MEM: Memo[ALUout] WB: Reg[Rd] = MemoRFBuffer
LWPOI	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = immExtended Rs1 = Rs1 +1 EX: opOne + opTwo MEM: Memo[ALUout] WB: Reg[Rd] = MemoRFBuffer
SW	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = immExtended EX: opOne + opTwo MEM: Memo[ALUout] = Rd
BGT	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rd EX: opOne > opTwo Then if greaterThanFlag → pc = pc +1 +result Else pc = pc +1

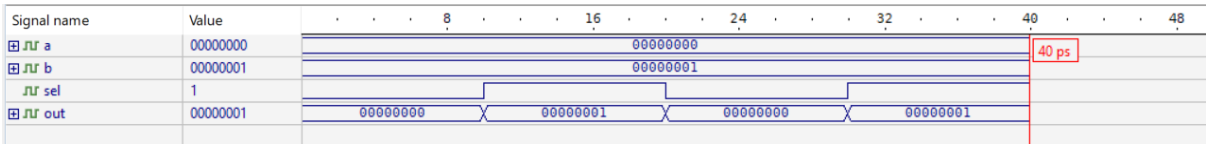
BLT	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rd EX: opOne < opTwo Then if lessThanFlag → pc = pc +1 +result Else pc = pc +1
BEQ	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rd EX: opOne - opTwo Then if zeroFlag → pc = pc +1 +result Else pc = pc +1
BNE	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = Rs1 , opTwo = Rd EX: opOne - opTwo Then if !zeroFlag → pc = pc +1 +result Else pc = pc +1
JMP	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = pc , opTwo = imm26 EX: opOne + opTwo Jumping into pc = target
CALL	IF: IR=Memo[pc] Pc = pc+1 ID: opOne = pc , opTwo = imm26

	EX: opOne + opTwo MEM: push into stack section (pc+1) Jumping into pc = target
RET	IF: IR=Memo[pc] Pc = pc+1 MEM: pop top element of stack section Nextpc = top of the stack
PUSH	IF: IR=Memo[pc] Pc = pc+1 ID: operand = Rd MEM: pushing Rd into stack Nextpc = pc+1
POP	IF: IR=Memo[pc] Pc = pc+1 ID: operand = Rd MEM: pop top element of the stack WB: store it id Rd Nextpc = pc+1

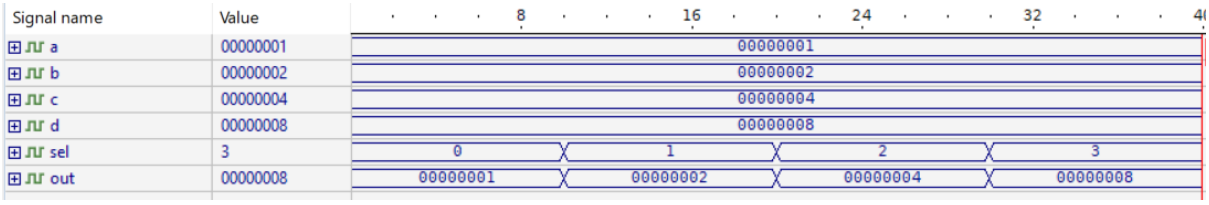
Testing Components

We wrote a testbench for every component to make sure that everything is flowing correctly.

Mux2x1



Mux4x1

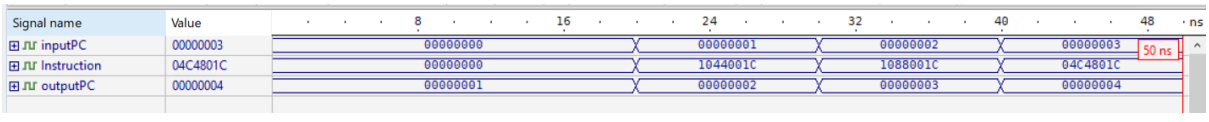


Instruction Memory

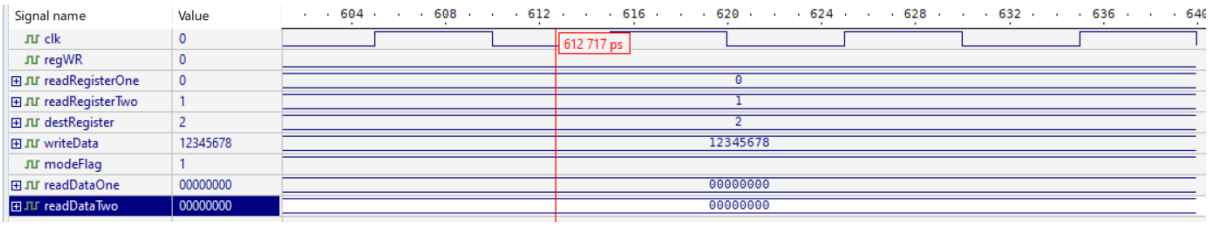
I loaded the below instructions into the memory:

```
reg [31:0] memo[2048:0]; // memoory is 32 bits wide for MIPS
// Example instructions:
initial begin
    memo[0] = 32'b00000000000000000000000000000000;// No operation
    memo[1] = 32'b00010000010001000000000000011100;//R1=R1+7
    memo[2] = 32'b00010000100010000000000000011100;//R2=R2+3
    memo[3] = 32'b00000100110001001000000000011100;//R3=R2+R1
```

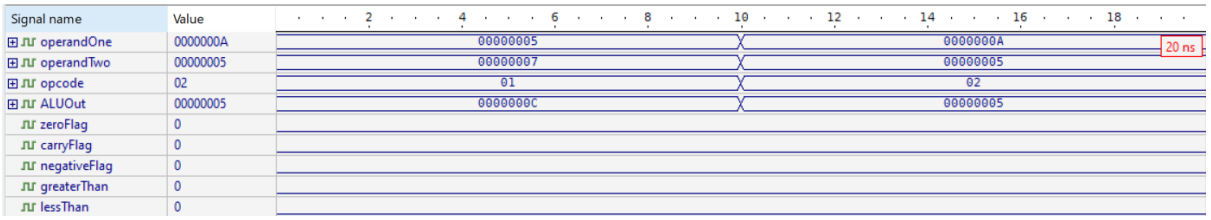
And the below waveform was the output after changing the value of pc:



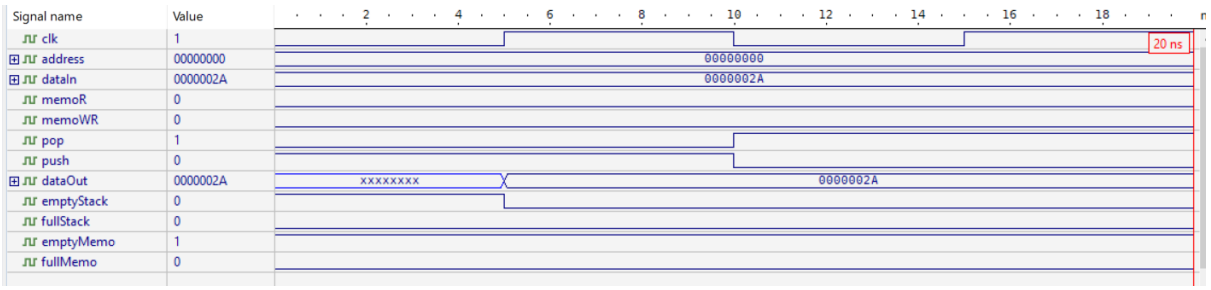
RF



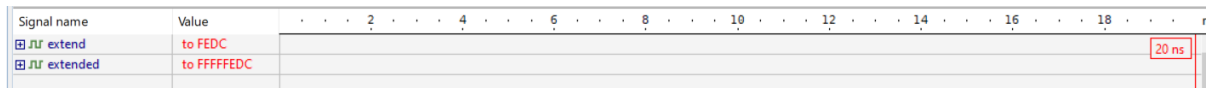
ALU



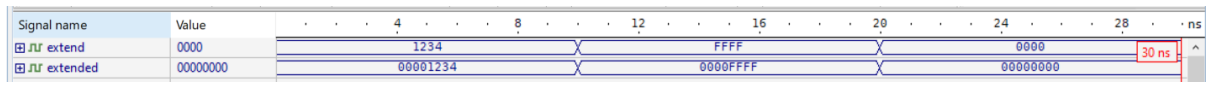
Data Memory (DM)



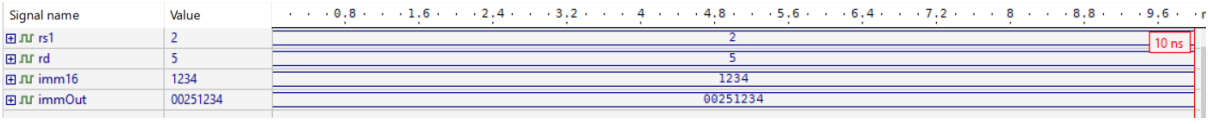
Sign Extender



Unsign Extender



ImmediateTo26



Design Verification

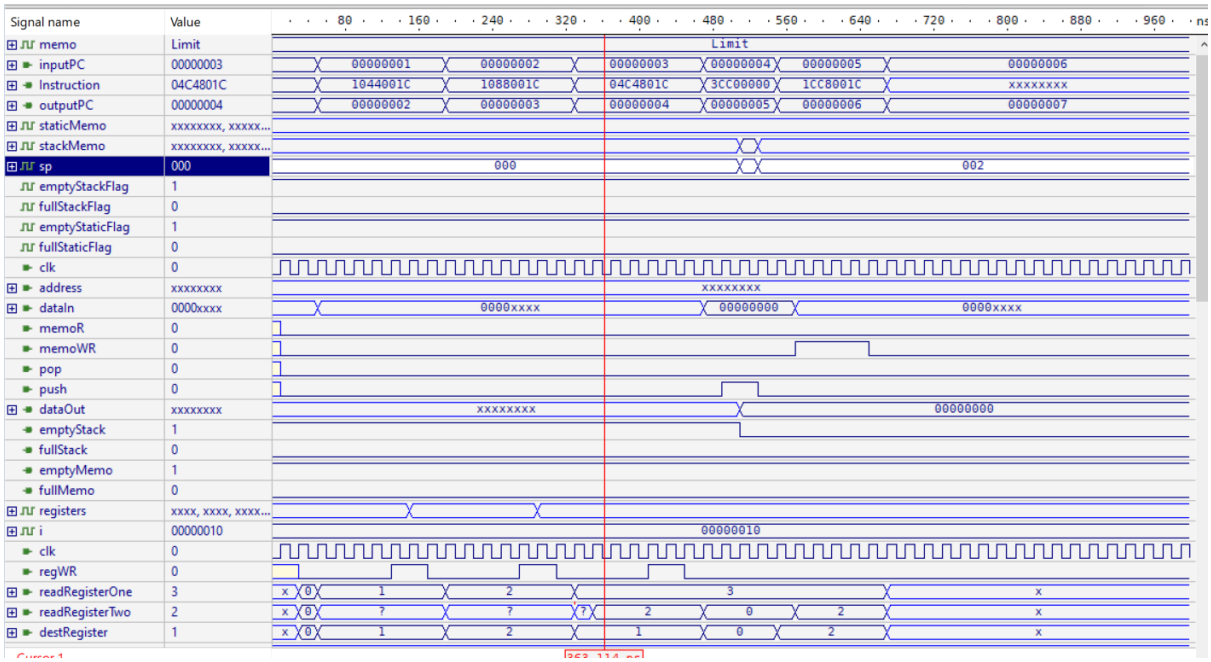
We wrote the below testbench in order to test the whole datapath of the processor:

```
`timescale 1ns / 1ps

module Datapath_tb;

    // Inputs
    reg clk;

    // Instantiate the Unit Under Test (UUT)
    Datapath uut (
    .clk(clk)
    );


    always
    #10 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;

        #5
        clk = 0;
        #2000
        $finish;
    end


endmodule
```

Moreover, some instructions have been loaded into the IM as shown below:

```
module InstructionMemo(
    input wire [31:0] inputPC,
    output reg [31:0] Instruction,
    output reg [31:0] outputPC

);

    reg [31:0] memo[2048:0]; // memoory is 32 bits wide for MIPS
    // Example instructions:
    initial begin
        memo[0] = 32'b00000000000000000000000000000000;// No operation
        memo[1] = 32'b00010000010001000000000000011100;//R1=R1+7
        memo[2] = 32'b00010000100010000000000000011100;//R2=R2+3
        memo[3] = 32'b00010011000100100000000000011100;//R3=R2+R1
        memo[4] = 32'b00111100110000000000000000000000;//push R3
        memo[5] = 32'b00011100110010000000000000011100;//sw R3, 7(R2)
        //memo[6] = 32'b00000100110001001000000000011100;//R3=R2+R1


    end
    always @* begin
        Instruction = memo[inputPC]; // I'm assuming inputPC is incremented externally
        outputPC <= inputPC + 1;
    end

endmodule
```

The testbench has been simulated and we got the below results:



As we can see from the previous snapshot that the instructions have been fetched correctly, decoded in the correct way, executed, and saved in the appropriate place.

Taking for example the instruction ***ADD*** after the two instructions before have been fetched already with loading the value 7 into R1 and 3 into R2, the instruction ADD has been executed and the output value of the addition operation was correctly 10.

Conclusion

In conclusion, this report showcases the successful development and implementation of a multi-cycle MIPS RISC processor as part of the ENCS4370 Computer Architecture project. Thorough testing and validation were conducted across all processor components, including the stack module, ALU module, control unit, register file, and data memory. The results demonstrated precise functionality with minimal errors. The provided RTL description, control state diagram, and testbench outcomes affirm the processor's efficient operation, capable of handling diverse instructions. The achievements of this project contribute significantly to the field of computer engineering, laying a robust foundation for future advancements in processor design and optimization.