" بسم الله الرحمن الرحيم "


Faculty of Engineering and Technology.

Electrical and Computer Engineering Department.

COMPUTER ARCHITECTURE-ENCS4370.

First Semester – 2023|2024.

# Project#2: RISC processor.

**********

**Team members (prepared by):** Razan Abdalrahman – 1200531.

Yasmeen Kamal – 1201146.

Maisam Alaa – 1200650.

**Instructor name:** Dr.Aziz Qaroush.

**Section:** 2.

**Date:** January | 2024.

## Abstract.

This project aims to design and validate a multicycle Reduced Instruction Set Computing (RISC) processor using Verilog. The processor is specified with a 32-bit instruction and word size, featuring 16 general-purpose registers (R0 to R15), a 32-bit program counter (PC), and a 32-bit stack pointer (SP). The memory layout includes static data, code, and a stack segment. The stack operates as a Last in First out (LIFO) data structure, with explicit push/pop instructions for stack manipulation. The processor employs separate physical memories for instructions and data, both stored in word-addressable memory. It supports four instruction types (R-type, I-type, J-type, and S-type) and incorporates an Arithmetic Logic Unit (ALU) to generate signals for condition branch outcomes, such as zero, carry, overflow, etc. This project combines hardware description language (Verilog) and multicycle techniques to create a functional and efficient RISC processor architecture.

# Table of contents.

**Abstract.** ..... 1

**1. Introduction.** ..... 4

**1.1 RISC Machines.** ..... 4

**1.2 Multi Cycle Processors.** ..... 4

**1.3 Differences between single cycle, multi cycle and pipelined processers.** ..... 4

**2. Design and Implementation.** ..... 5

**2.1 Processor specifications.** ..... 5

2.1.1 Processor properties. ..... 5

2.1.2 Instruction types and formats. ..... 6

2.1.3 Instruction Set. ..... 7

2.1.4 Finite state machine (FSM). ..... 8

**2.2 Detailed description of the data path.** ..... 10

2.2.1 Data path diagram. ..... 10

2.2.2 RTL description. ..... 11

2.2.3 Components of the data path. ..... 14

2.3.1 Description of the control signals used. ..... 26

**3. Simulation and Testing.** ..... 35

**4. Teamwork.** ..... 36

**5. Conclusion.** ..... 37

**6. References.** ..... 38

# List of figures.

**Figure 2-1 R-Type format.** ..... 6
**Figure 2-2 I-Type format.** ..... 6
**Figure 2-3 J-Type format.** ..... 6
**Figure 2-4 S-Type format.** ..... 7
**Figure 2-5 Data path diagram.** ..... 10
**Figure 2-6 PC-Register code.** ..... 14
**Figure 2-7 PC-Register test.** ..... 15
**Figure 2-8 Instruction memory code.** ..... 16
**Figure 2-9 Instruction memory test.** ..... 17
**Figure 2-10 Register file code.** ..... 18
**Figure 2-11 Register file test.** ..... 19
**Figure 2-12 ALU code.** ..... 20
**Figure 2-13 ALU test.** ..... 21
**Figure 2-14 Data memory code.** ..... 22
**Figure 2-15 Data memory test.** ..... 23
**Figure 2-16 Extender code.** ..... 24
***Figure 2-17 Extender test.*** ..... 25
**Figure 2-18 Main control code.** ..... 30
**Figure 2-19 PC control code.** ..... 34

# List of tables.

**Table 2-1 Instrction set.** ..... 7
**Table 2-2 Main control signals**. ..... 27
**Table 2-3 Main Control Truth Table.** ..... 28
**Table 2-4 PC control signals**. ..... 32
**Table 2-5 PC Control Truth Table**. ..... 33

# 1. Introduction.

## 1.1 RISC Machines.

A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions rather than the highly-specialized set of instructions typically found in other architectures. RISC is an alternative to the Complex Instruction Set Computing (CISC) architecture and is often considered the most efficient CPU architecture technology available today [1].

## 1.2 Multi Cycle Processors.

A multi-cycle processor is a type of processor design where each instruction is divided into multiple stages, and each stage is completed in a separate clock cycle. This allows for more efficient use of hardware resources and can improve overall performance [2].

## 1.3 Differences between single cycle, multi cycle and pipelined processers.

In a single-processor system, only one process can be executed at a time, chosen from the ready queue, limiting concurrent execution even if multiple applications need processing. On the other hand, multiprocessor systems, comprising two or more processors, allow simultaneous execution of multiple applications. Symmetric or asymmetric multiprocessing, the two main types, enable multiple processors to handle different tasks concurrently, enhancing overall system efficiency [3].

In a Multiple Cycle Data path, instructions have varying clock cycles and are executed one at a time, requiring extra registers for result transfer. Performance is moderately faster than single cycle. In a Pipeline Data path, instructions have a fixed cycle count, multiple instructions can be processed simultaneously with duplicate hardware, and extra registers facilitate inter-stage data transfer, resulting in significantly faster performance than single cycle [4].

# 2. Design and Implementation.

## 2.1 Processor specifications.

### 2.1.1 Processor properties.

- The instruction size and the words size is 32 bits.
- 16 32-bit general-purpose registers: from R0 to R15.
- 32-bit special purpose register for the program counter (PC)
- 32-bit special purpose register for the stack pointer (SP), which points to the topmost empty element of the stack.
- The program memory layout comprises the following three segments:
    - Static data segment
    - Code segment
    - Stack segment. It is a LIFO (Last in First out) data structure. This machine has explicit instructions that enables the programmer to push/pop elements on/from the stack. The stack stores the return address, registers' values upon function calls.
- The processor has two separate physical memories, one for instructions and the other one for data. The data memory stores both the static data segment and the stack segment.
- Four instruction types (R-type, I-type, J-type, and S-type).
- Separate data and instructions memories.
- Word-addressable memory.
- You need to generate the required signals from the ALU to calculate the condition branch outcome (taken/ not taken). These signals might include zero, carry, and overflow.

### 2.1.2 Instruction types and formats.

As mentioned above, this ISA has four instruction formats, namely, R-type, I-type, J-type, and S-type. These four types have a common 6-bit opcode field, which determines the specific operation of the instruction.

- ❖ R-Type (Register Type).
  - 4-bit Rd: destination register.
  - 4-bit Rs1: first source register.
  - 4-bit Rs2: second source register.
  - 14-bit unused.

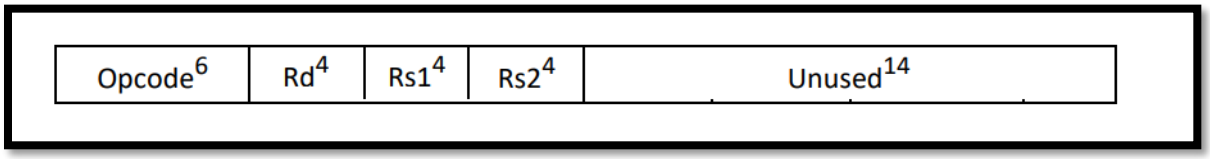



*Figure 2-1 R-Type format.*

- ❖ I-Type (Immediate Type).
  - 4-bit Rd: destination register.
  - 4-bit Rs1: first source register.
  - 16-bit immediate: unsigned for logic instructions, and signed otherwise.
  - 2-bit mode: this is used with load/store instructions only.

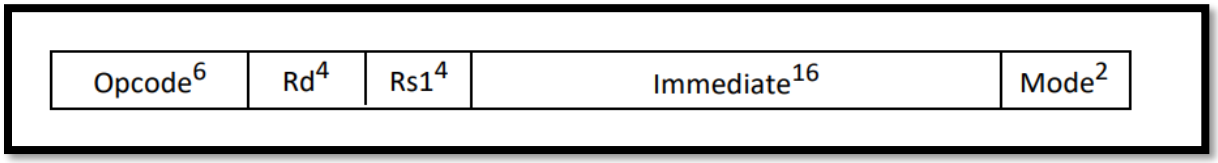



*Figure 2-2 I-Type format.*

- ❖ J-Type (Jump Type).
  - 24-bit: jump offset.

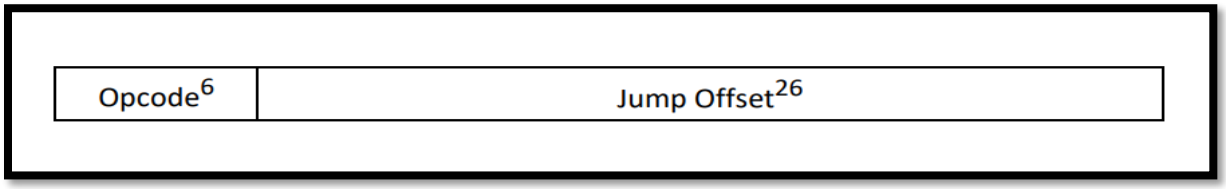



*Figure 2-3 J-Type format.*

- S-Type (Stack Type).
  - 4-bit Rd.
  - 22-bit unused.

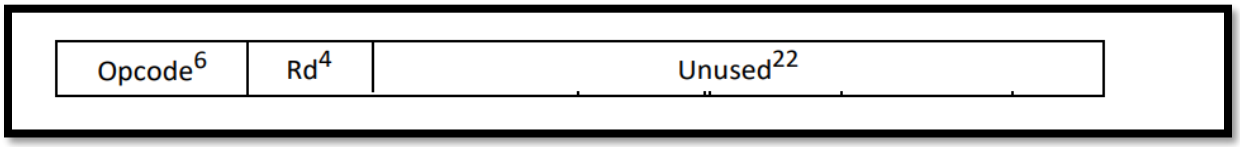



*Figure 2-4 S-Type format.*

### 2.1.3 Instruction Set.

Table 2-1 shows the instructions supported by this instruction set, with their meaning and decoding.

| No. | Instr | Meaning | Opcode Value |
|---|---|---|---|
| | | **R-Type Instructions** | |
| 1 | AND | Reg(Rd) = Reg(Rs1) & Reg(Rs2) | 000000 |
| 2 | ADD | Reg(Rd) = Reg(Rs1) + Reg(Rs2) | 000001 |
| 3 | SUB | Reg(Rd) = Reg(Rs1) - Reg(Rs2) | 000010 |
| | | **I-Type Instructions** | |
| 4 | ANDI | Reg(Rd) = Reg(Rs1) & $Imm^{16}$ | 000011 |
| 5 | ADDI | Reg(Rd) = Reg(Rs1) + $Imm^{16}$ | 000100 |
| 6 | LW | Reg(Rd) = Mem(Reg(Rs1) + $Imm^{16}$) | 000101 |
| 7 | LW.POI | Reg(Rd) = Mem(Reg(Rs1) + $Imm^{16}$)<br>Reg[Rs1] = Reg[Rs1] + 1 | 000110 |
| 8 | SW | Mem(Reg(Rs1) + $Imm^{16}$) = Reg(Rd) | 000111 |
| 9 | BGT | if (Reg(Rd) > Reg(Rs1))<br>Next PC = PC + sign_extended ($Imm^{16}$)<br>else PC = PC + 1 | 001000 |
| 10 | BLT | if (Reg(Rd) < Reg(Rs1))<br>Next PC = PC + sign_extended ($Imm^{16}$)<br>else PC = PC + 1 | 001001 |
| 11 | BEQ | if (Reg(Rd) == Reg(Rs1))<br>Next PC = PC + sign_extended ($Imm^{16}$)<br>else PC = PC + 1 | 001010 |
| 12 | BNE | if (Reg(Rd) != Reg(Rs1))<br>Next PC = PC + sign_extended ($Imm^{16}$)<br>else PC = PC + 1 | 001011 |
| | | **J-Type Instructions** | |
| 13 | JMP | Next PC = {PC[31:26], $Immediate^{26}$ } | 001100 |
| 14 | CALL | Next PC = {PC[31:26], $Immediate^{26}$ }<br>PC + 1 is pushed on the stack | 001101 |
| 15 | RET | Next PC = top of the stack | 001110 |
| | | **S-Type Instructions** | |
| 16 | PUSH | Rd is pushed on the top of the stack | 001111 |
| 17 | POP | The top element of the stack is popped, and it is stored in the Rd register | 010000 |

*Table 2-5 Instrction set.*

## 2.1.4 Finite state machine (FSM).

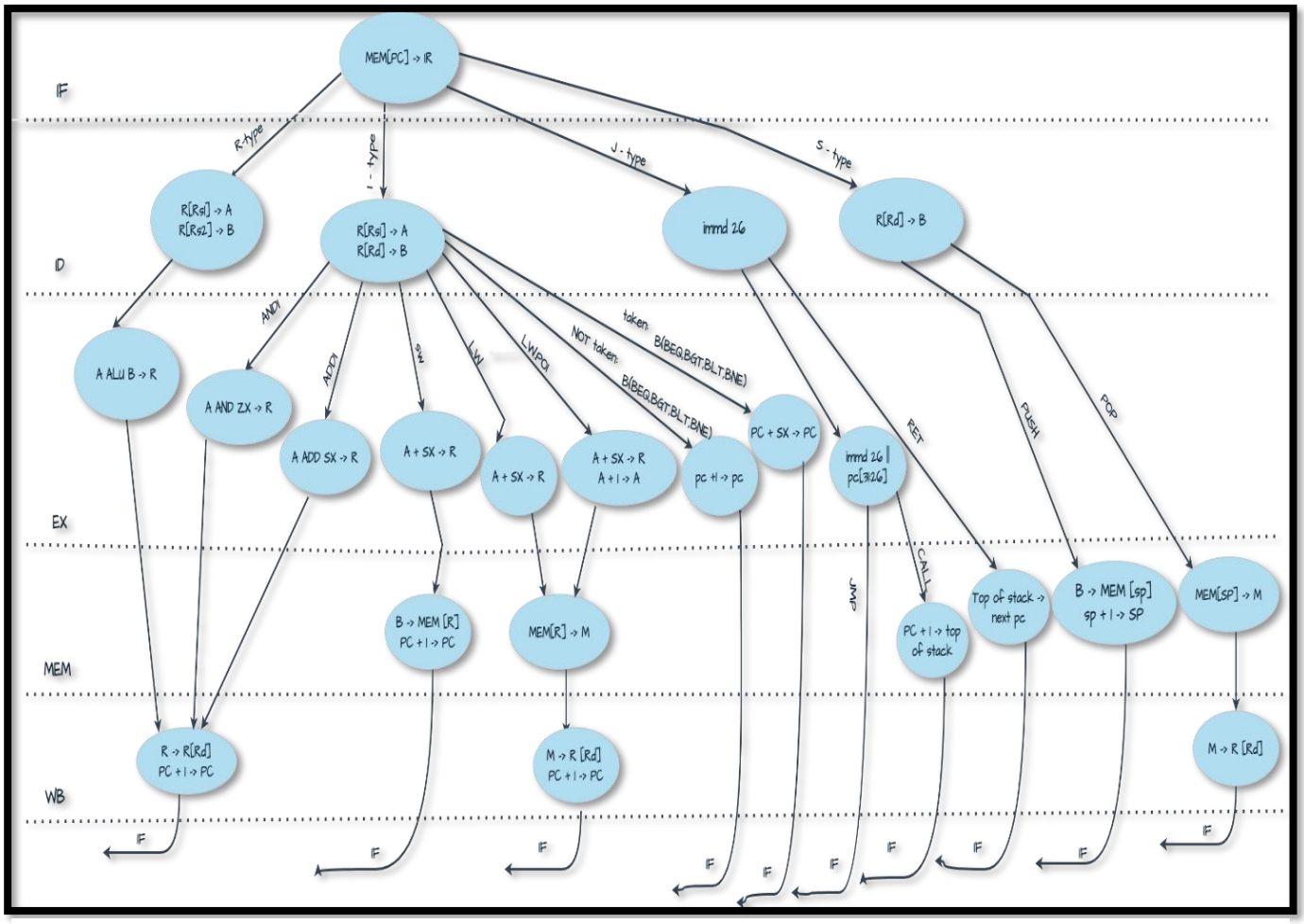



*Figure 2-6 FSM.*

The instruction cycle, also known as the fetch-execute cycle, is the fundamental process by which a computer executes instructions. It typically consists of the following main stages: IF, ID, EX, MEM, and WB.

IF: OR Instruction fetch, in the fetch stage, the processor retrieves the next instruction from memory. The program counter (PC) holds the address of the next instruction to be fetched. The instruction is then placed into the instruction register (IR) for decoding and execution.

ID: OR Instruction decode, during the decode stage, the processor interprets the fetched instruction to determine the operation to be performed and the operands involved. The opcode portion of the instruction is typically used to identify the operation, while additional fields specify operands or other information.

EX: OR Execute, in the execute stage, the processor carries out the operation specified by the instruction. This may involve arithmetic or logical operations, data transfers between registers or memory, or control transfers such as branching or subroutine calls.

MEM: OR Memory, in this stage, the processor interacts with memory. For instructions that involve accessing data in memory (e.g., load or store instructions), the necessary data is read from or written to memory. For instructions that don't involve memory access, this stage may be a no-operation (NOP).

WB: OR Write back, finally, in the write back stage, the results of the execution are stored back into registers or memory, depending on the nature of the instruction. For example, the result of an arithmetic operation might be stored in a register, while the outcome of a memory load operation might be placed into a register for subsequent use.

## 2.2 Detailed description of the data path.

### 2.2.1 Data path diagram.

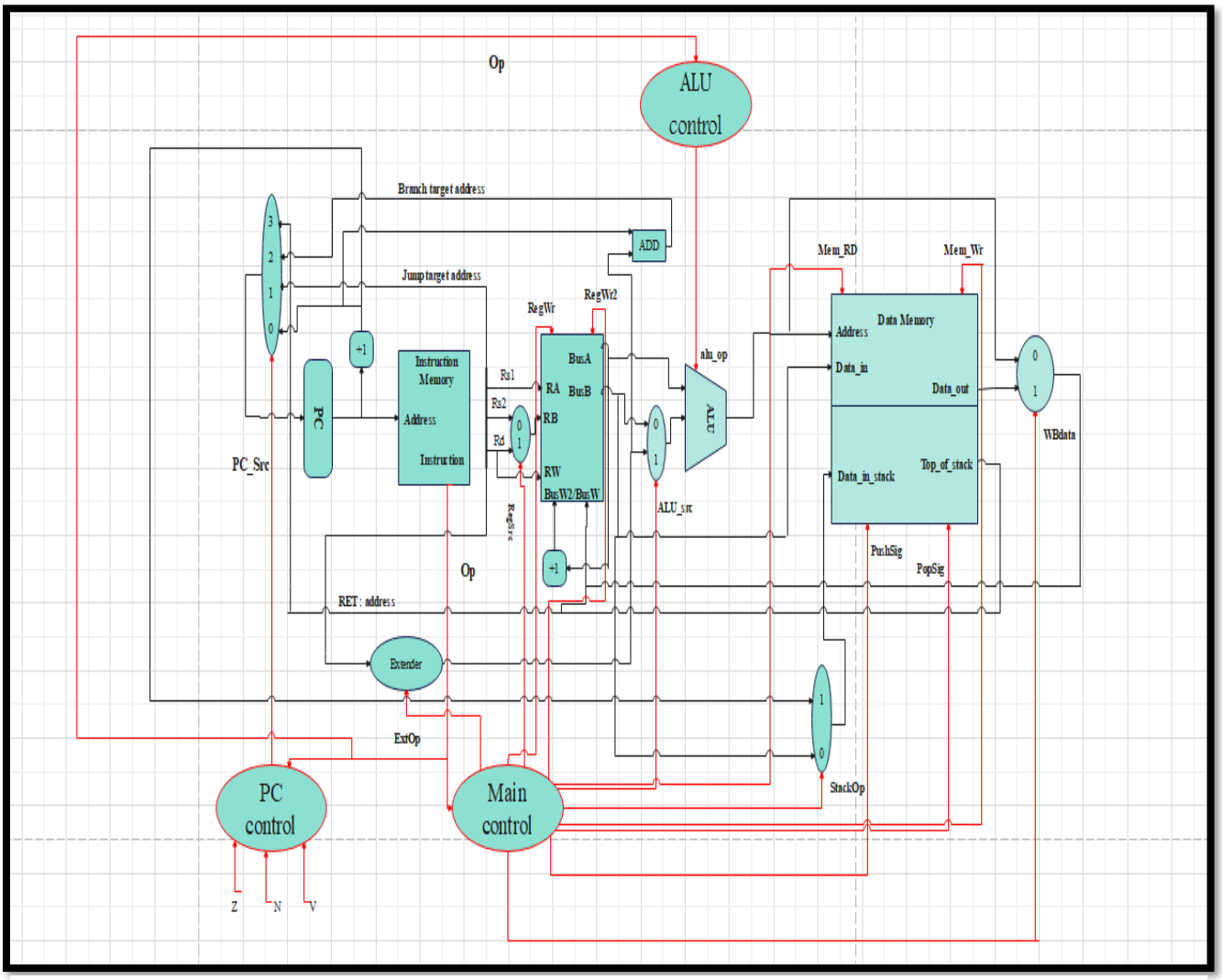



*Figure 2-7 Data path diagram.*

### 2.2.2 RTL description.

* R-type:

Fetch instruction: Mem[PC] → Instruction

Fetch operands: data1→ Rs1, data2 → Rs2

Execute operation: op(data1,data2) → ALU_result

Write ALU result: ALU_result → reg(Rd)

Next PC address: PC=PC+1

* I-type:(ADDI, ANDI):

Fetch instruction: Mem[PC] → Instruction

Fetch opearnds: data1→ Rs1, data2 → Imm16

Fetch (Extended): if (ADDI) immd(16) → signed_Extended(immd 16)

Execute operation: op(data1, data2) → ALU_result.

Write ALU_result: ALU_result → reg(Rd)

Next PC address: PC=PC+1

* I-type:(LW, LW.POI):

Fetch instrcution: Mem[PC] → Instruction

Fetch register: Reg(Rs1) → base

Calculate address: base + sign_extended(Imm16) → address

Read memory: MEM[address] → data

Write register: data → Reg(Rd)

if (LW.POI) Reg(Rs1) + 1 → Reg (Rs1)

Next PC address: PC=PC+1

* I-type:(SW):

Fetch instruction: Mem[PC] → Instruction

Fetch register: Reg(Rs1) → base, Reg(Rd) → data

Calculate address: base + sign_extended(Imm16) → address

Write memory: data → Mem[address]

Next PC address: PC = PC + 1

* I_type:(BEQ, BNQ, BGT, BLT):

Fetch instruction: Mem[PC] → Instruction

Fetch operands: Reg(Rs1) → data1, Reg(Rd) → data2

Result: Subtract (data1,data2) → ZERO, OVER_FLOW, NEGATIVE

Branch: if (zero): PC + sign_ext(offset16) → PC

else : PC + 1 → PC

* J_type: (JMP, CALL, RET):

Fetch instruction: Mem[PC] → Instruction

Target PC address: if (JUMP || CALL) : PC[31:26] || address26 → Target

else: TOP OF STACK → Target

Jump:Target → PC

Push on stack: if(CALL) : PC + 1 → TOP OF STACK

*S_type: (PUSH):

Fetch instruction: Mem[PC] → Instruction

Decrement SP: SP - 1 → SP

Update stack: Reg (Rd) → Top of stack.

*S_type: (POP):

Fetch instruction: Mem[PC] → Instruction

Update stack: Top of stack → Reg (Rd)

Increamnet SP: SP + 1 → SP

## 2.2.3 Components of the data path.

### PC Register.

The Program Counter (PC) register, alternatively referred to as the instruction pointer, serves as a specialized register within a CPU (Central Processing Unit). Its primary function is to store the memory address of the upcoming instruction that the processor needs to fetch and execute. During the execution of an instruction, the PC register plays a crucial role in determining the address of the next instruction to be fetched. This sequential process enables the processor to fetch and execute instructions in the correct order, facilitating the orderly progression of program execution.

- ***PC-Register module.***

```
module PC (clk, PC_src, J_instruction, I_instruction, S_instruction, instruction_address);
    input clk;
    input [1:0] PC_src;
    input [31:0] J_instruction, I_instruction, S_instruction;

    output reg [31:0] instruction_address;

    wire [31:0] pc_plus_one, jump_target_address, branch_target_address;

    assign pc_plus_one = instruction_address + 32'd1;
    assign jump_target_address = instruction_address [31:26] + J_instruction [25:0];
    assign branch_target_address = instruction_address + I_instruction;

    initial begin
        instruction_address <= 32'd0;
    end
    always @(posedge clk)
        begin
            case (PC_src)
            2'd0:begin
                instruction_address = pc_plus_one;
            end
            2'd1:begin
                instruction_address = jump_target_address;
            end
            2'd2:begin
                instruction_address = branch_target_address;
            end
            2'd3:begin
                instruction_address = S_instruction;
            end

            endcase
        end
endmodule
```

*Figure 2-8 PC-Register code.*

- ***PC-Register test.***

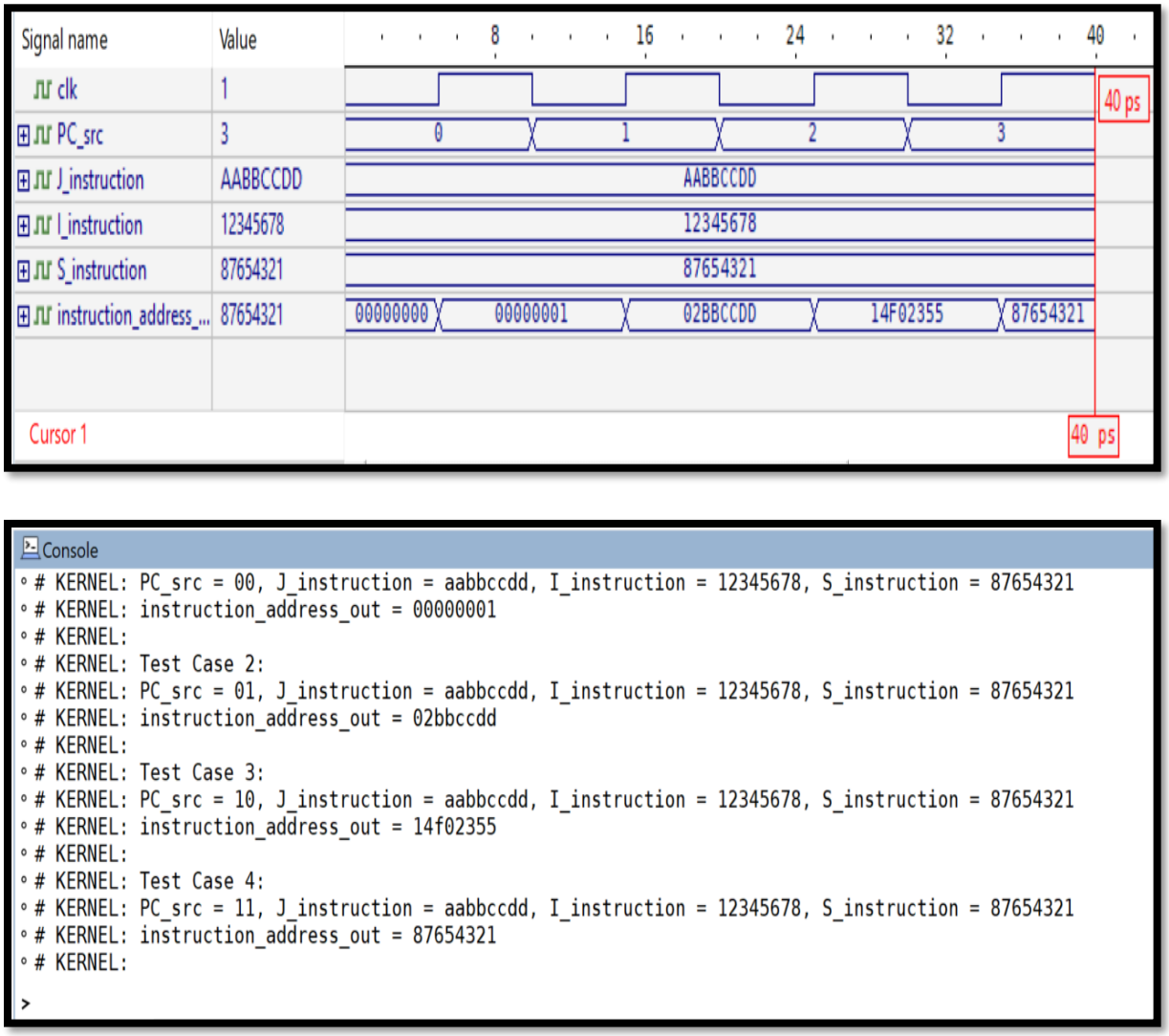



***Figure 2-9 PC-Register test.***

## Instruction memory.

The implementation features a division of memories into two distinct parts: instruction memory and data memory.

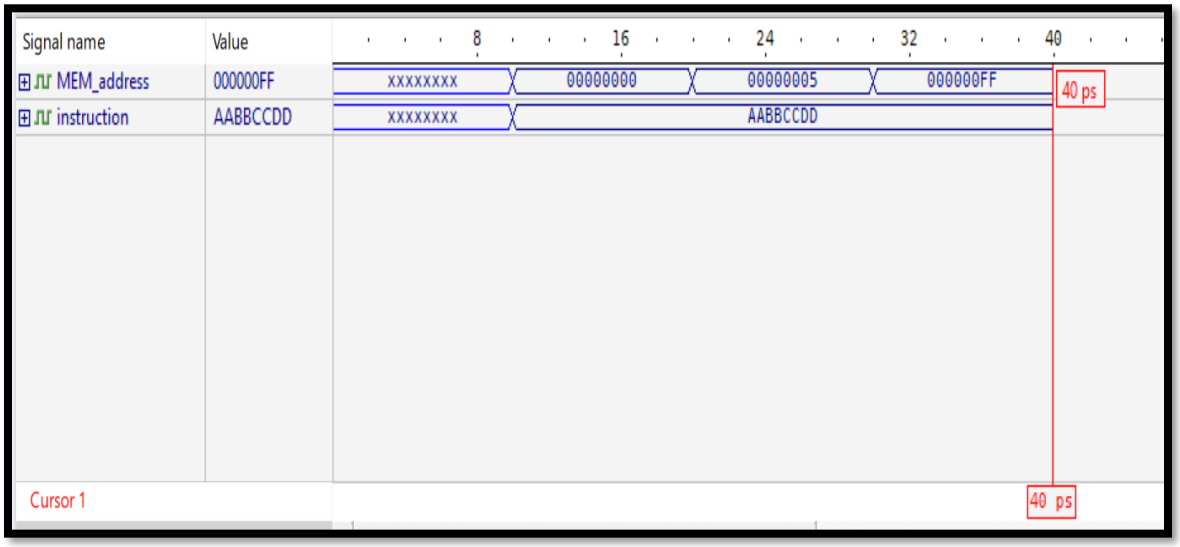
The instruction memory is a crucial component in the data path. It is a dedicated storage unit responsible for holding the program instructions that the processor needs to execute. In the context of a CPU, the instruction memory stores the binary representations of machine instructions. The Program Counter (PC) determines the address in the instruction memory from which the next instruction is fetched. The fetched instruction is then passed to the processor for decoding and execution.

- ***Instruction memory module.***

```
1  module Instruction_MEM (MEM_address, instruction);
2
3      input [31:0] MEM_address;
4      output reg [31:0] instruction;
5
6      reg [31:0] instruction_MEM [0:255];
7
8
9      always @ (MEM_address)
10         begin
11             instruction=instruction_MEM[MEM_address];
12         end
13
14     initial
15         begin
16             instruction_MEM[0] = 32'haabbccdd;
17             instruction_MEM[5] = 32'haabbccdd;
18             instruction_MEM[255] = 32'haabbccdd;
19         end
20
21
22 endmodule
23
```

*Figure 2-10 Instruction memory code.*

- ***Instruction memory test.***




```
# KERNEL: Time=0 MEM_address=xxxxxxxx instruction=xxxxxxxx
# KERNEL: Time=10 MEM_address=00000000 instruction=aabbccdd
# KERNEL: Time=20 MEM_address=00000005 instruction=aabbccdd
# KERNEL: Time=30 MEM_address=000000ff instruction=aabbccdd
# RUNTIME: Info: RUNTIME_0068 tb_Instruction_MEM.v (23): $finish called.
```

*Figure 2-11 Instruction memory test.*

## Register file.

The register file stands as a vital component within the CPU, offering swift storage and rapid access to registers for diverse operations. Its streamlined design, coupled with its proximity to the execution units, plays a pivotal role in enhancing the overall performance and functionality of the computer system.

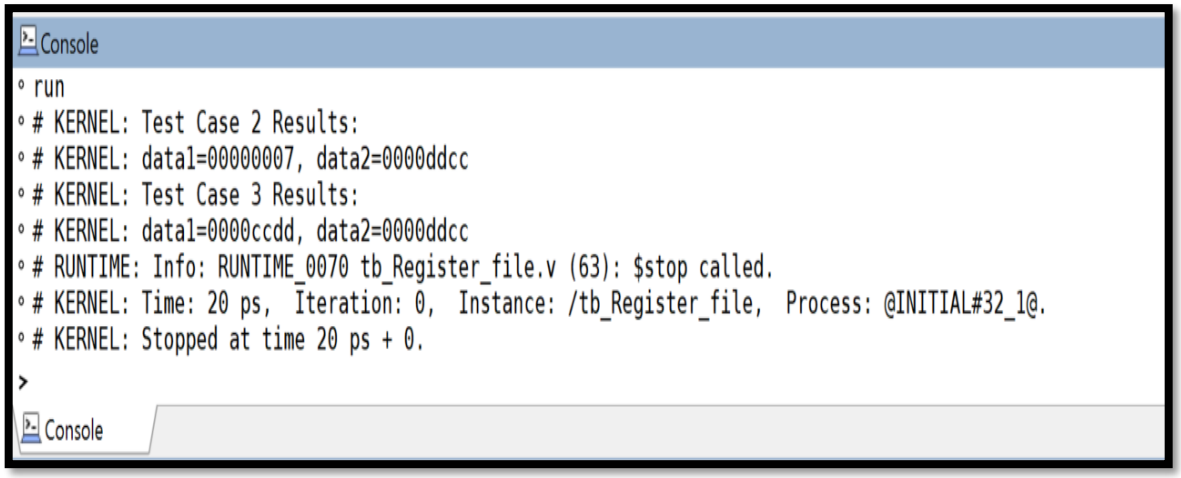
- ***Register file module.***

```
module Register_file(clk, first_src, second_src, destination, Sig_wr1, Sig_wr2, Sig_Src, Reg_wr1, Reg_wr2, data1, data2);

    input clk, Sig_wr1, Sig_wr2, Sig_Src;
    input wire [3:0] first_src;
    input [3:0] second_src, destination;
    input [31:0] Reg_wr1, Reg_wr2;

    output reg [31:0] data1, data2;

    reg [31:0] reg_array [0:15];

    always @ (Sig_Src)
        begin
            data1 <= reg_array[first_src];

            if(Sig_Src)
                data2 <= reg_array[destination];
            else
                data2 <= reg_array[second_src];

        end
```

```

    always @ (posedge clk)
        begin
            case({Sig_wr1,Sig_wr2})
                2'd2:begin
                    reg_array[destination] <= Reg_wr1;
                end
                2'd3:begin
                    reg_array[destination] <= Reg_wr1;
                    reg_array[first_src] = reg_array[first_src]+1 ;

                end
            endcase

        end
```

*Figure 2-12 Register file code.*

- ***Register file test.***



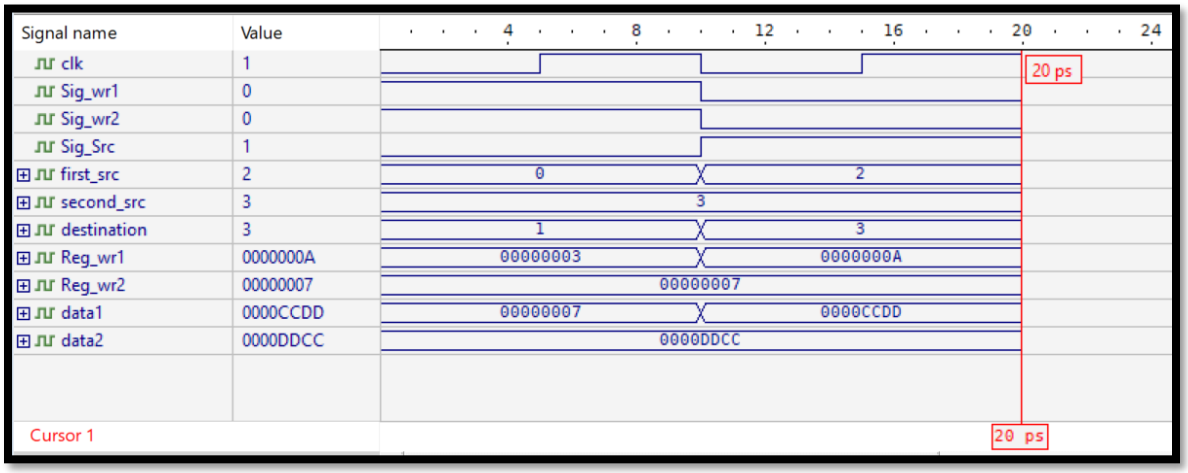




*Figure 2-13 Register file test.*

## Arithmetic Logical Unit.

The Arithmetic Logic Unit (ALU) is an integral digital circuit within the CPU responsible for executing arithmetic and logical operations on binary data. This crucial component takes two input operands, performs operations based on the specified instruction, and generates an output result. The ALU's capabilities encompass a broad spectrum of operations, spanning addition, subtraction, multiplication, division, bitwise logical operations (AND, OR, XOR), as well as comparisons like greater than, less than, and equal to.

- ***ALU module.***

```
1  module ALU (data1, data2, calculated_value, Sig_ALU);
2      input wire [31:0] data1, data2;
3      input [1:0] Sig_ALU;
4      output reg [31:0] calculated_value;
5
6
7      always @ (*)
8          begin
9              case (Sig_ALU)
10                 2'b00:begin
11                     calculated_value <= data1 & data2;
12                 end
13                 2'b01:begin
14                     calculated_value <= data1 + data2;
15                 end
16                 2'b10:begin
17                     calculated_value <= data1 - data2;
18                 end
19                 default:
20                     calculated_value <= 32'b0;
21             endcase
22         end
23
24 endmodule
```

*Figure 2-14 ALU code.*

- ***ALU test.***

```
Console
 'C:/Users/Support/Desktop/NS/Arch/Projects/SecondProject/finalProject/finalProject/finalProject/finalProject
run
# KERNEL: Time=0 data1=00000000 data2=ffffffff Sig_ALU=00 calculated_value=00000000
# KERNEL: Time=10000 data1=00000000 data2=ffffffff Sig_ALU=01 calculated_value=ffffffff
# KERNEL: Time=20000 data1=00000000 data2=ffffffff Sig_ALU=10 calculated_value=00000001
# RUNTIME: Info: RUNTIME_0070 tb_ALU.v (50): $stop called.
# KERNEL: Time: 30 ns,  Iteration: 0,  Instance: /ALU_tb,  Process: @INITIAL#25_2@.
# KERNEL: Stopped at time 30 ns + 0.
>
```

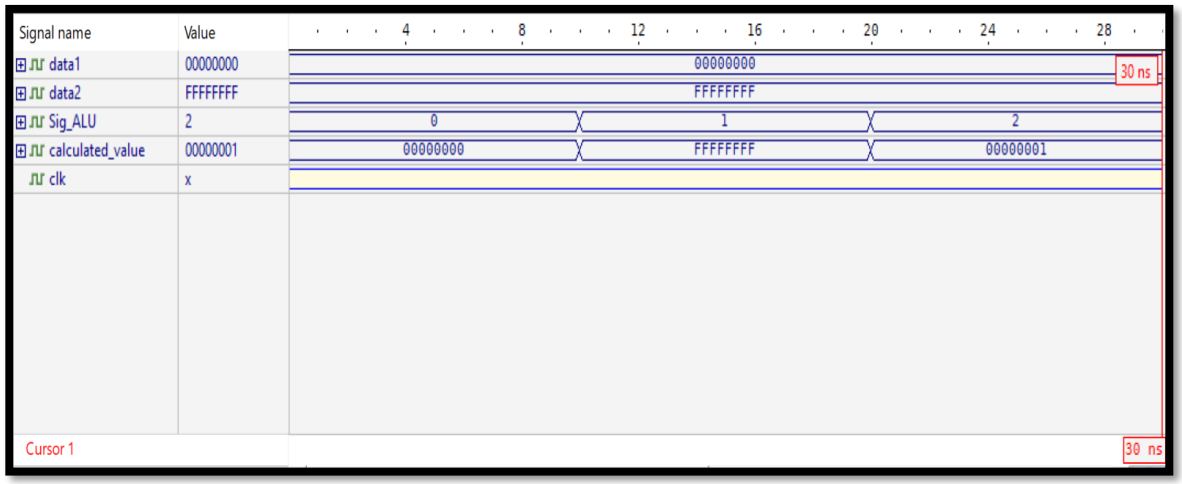



***Figure 2-15 ALU test.***

- Data memory.

In a RISC processor where the data memory is word-addressable, each memory location is associated with a specific word, which is typically the basic unit of data storage. This means that individual words, rather than bytes, are the smallest addressable units.

A portion of the data memory may be designated for use as a stack. The stack is a region of memory managed in a Last-In-First-Out (LIFO) fashion and is commonly employed for storing local variables, return addresses, and managing function calls.

- ***Data memory module.***

```
module Data_MEM(clk, address, data_in, Sig_RD, Sig_WR, data_in_stack, Sig_push, Sig_pop,sp
    ,data_out, top_Of_Stack);

    input clk, Sig_RD, Sig_WR, Sig_push, Sig_pop;
    input [31:0] address, data_in,sp, data_in_stack;
    output reg [31:0] data_out, top_Of_Stack;

    reg [31:0] reg_MEM [0:1023];


    always @ (*)
        begin
            if (Sig_RD) begin
                data_out <= reg_MEM[address];
            end

            else if (Sig_WR) begin
                reg_MEM[address] <= data_in;
            end
        end
```

```
        end

    always @ (*)
        begin
            if (Sig_push) begin

                reg_MEM[sp] <= data_in_stack;

            end

        else if (Sig_pop) begin

                top_Of_Stack <= reg_MEM[sp];

            end
        end
```

*Figure 2-16 Data memory code.*

- ***Data memory test.***

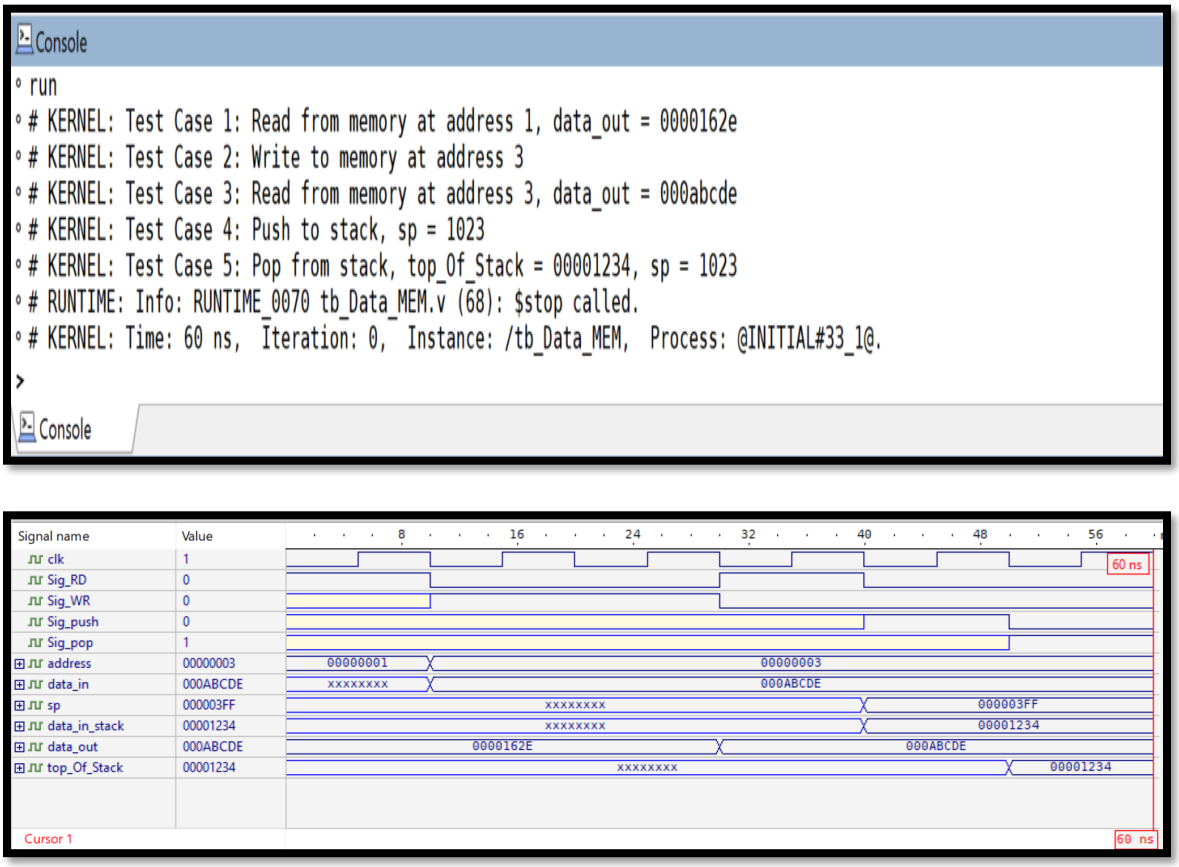



*Figure 2-17 Data memory test.*

## Extender.

In computer architecture, sign extension and zero extension are techniques used to expand the bit-width of binary numbers. These operations are particularly relevant when dealing with values of different sizes, ensuring proper representation and interpretation within a given system.

The extender is employed to expand a 26-bit immediate value to 32 bits, with the choice of either zero extension or sign extension. The extender ensures that the immediate value, originally represented in 26 bits, is appropriately extended to fill the 32-bit space while considering the specific requirements of the operation, whether it involves preserving the signed nature of the value (sign extension) or simply padding with zeros (zero extension).

### ***Extender module.***

```
1  module Extender(immediate, OpExt, extender_out);
2      input  [31:0] immediate;
3      input OpExt;
4      output reg [31:0] extender_out;
5
6      always @ (*)
7          begin
8              case (OpExt)
9                  1'b0:begin
10                     extender_out = {16'b0 , immediate [17:2]};
11                 end
12                 1'b1:begin
13                     extender_out = {{16{immediate[17]}},immediate [17:2]};
14                 end
15             endcase
16         end
17     endmodule
18
19
```

*Figure 2-18 Extender code.*

- ***Extender test***

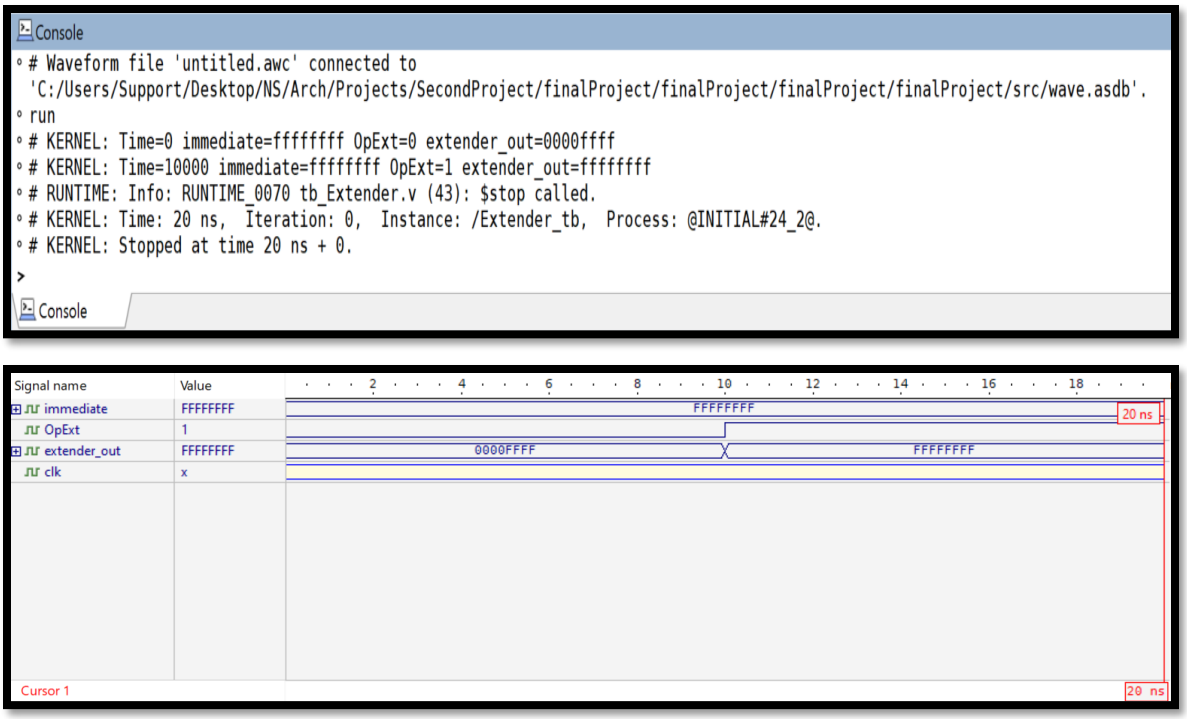



*Figure 2-19 Extender test.*

## 2.3 Detailed description of the control signals.

### 2.3.1 Description of the control signals used.

- Main control signals.

**DESCRIPTION.**

- Reg_Src: This signal is utilized to identify the second source in the instruction, serving as either Rs2 for R-type instructions or Rd for I-type instructions.
- Reg_Wr: This signal is employed to prevent and manage the write operation on register Rd.
- Reg_Wr2: This signal is employed to prevent and manage the write operation on register Rs1.
- ExtOp: The signal is utilized to determine whether the extension operation will involve zero extension or sign extension.
- ALUSrc: This signal is employed to select the second input for the ALU, which has two inputs: BusA (the first input) and the second input with various possible values. The ALU_Src signal is used to decide between these options for the second ALU input, which include extended-immediate and BusB.
- Mem_Rd: This signal is exclusively utilized for the LW, LW.POI instruction, overseeing the read operation and preventing other instructions from accessing memory for reading.
- Mem_Wr: This signal is specifically designated for the SW instruction, overseeing the write operation and inhibiting other instructions from performing write operations to memory.
- WB_data: This signal is employed to select the appropriate data to be written into the register file, with two potential values: ALU_Result and Data_out of the memory.
- PushSig: This signal is activated to push data onto the top of the stack.
- PopSig: This signal is triggered to retrieve and remove data from the top of the stack.

**MAIN CONTROL SIGNALS.**

| Signal | Effect when 0 | Effect when 1 |
|---|---|---|
| **RegDs** | Destination register = Rs2 | Destination register = Rd |
| **RagWr** | No register is written | Destination register (Rt or Rd) is written with the data on BusW |
| **RagWr2** | No register is written | Destination register (Rs1+1) is written with the data on BusW2 |
| **ExtOp** | 16-bit immediate is zero-extended | 16-bit immediate is sign-extended |
| **ALUSrc** | Second ALU operand is the value of register Rs2 that appears on BusB | Second ALU operand is the value of the extended 16-bit immediate |
| **Mem_Wr** | Data Memory is NOT written Data | memory is written Memory[address] ← Data_in |
| **Mem_Rd** | Data memory is NOT read Data | memory is read Data_out ← Memory[address] |
| **WBdata** | BusW = ALU result | BusW = Data_out from Memory |
| **PushSig** | No data is pushed to Stack | Data(RD or pc+1) is pushed to Stack |
| **PopSig** | No data is popped from the stack | Data(RD of NextPC)is popped from the stack |
| **StackOp** | Takes data from BusB | Takes data from NextPC |

*Table 2-1 Main control signals.*

## MAIN CONTROL TRUTH TABLE.

| Opcode | RegSrc | RegWr | RegWr2 | ExtOp | ALUSrc | Mem_Rd | Mem_Wr | WB_data | PushSig | PopSig |
|---|---|---|---|---|---|---|---|---|---|---|
| **R-type** | 1 → Rd | 1 | 0 | X | 0 | 0 | 0 | 0 = ALU | X | X |
| **ANDI** | 0 → Rs2 | 1 | 0 | 0 → zero | 1 → imm | 0 | 0 | 0 = ALU | X | X |
| **ADDI** | 0 → Rs2 | 1 | 0 | 1 → sign | 1 → imm | 0 | 0 | 0 = ALU | X | X |
| **LW** | 0 → Rs2 | 1 | 0 | 1 → sign | 1 → imm | 1 | 0 | 1 = Mem | X | X |
| **LW.POI** | 0 → Rs2 | 1 | 1 | 1 → sign | 1 → imm | 1 | 0 | 1 = Mem | X | X |
| **SW** | X | 0 | 0 | 1 → sign | 1 → imm | 0 | 1 | X | X | X |
| **BGT** | X | 0 | 0 | 1 → sign | 0 → BusB | 0 | 0 | X | X | X |
| **BLT** | X | 0 | 0 | 1 → sign | 0 → BusB | 0 | 0 | X | X | X |
| **BEQ** | X | 1 | 0 | 1 → sign | 0 → BusB | 0 | 0 | X | x | X |
| **BNQ** | X | 1 | 0 | 1 → sign | 0 → BusB | 0 | 0 | X | X | X |
| **JMP** | X | X | 0 | X | X | 0 | 0 | X | x | x |
| **CALL** | X | X | 0 | X | X | 0 | 0 | X | 1 | 0 |
| **RET** | X | X | 0 | X | X | 0 | 0 | X | 0 | 1 |
| **PUSH** | X | 1 | 0 | X | X | 0 | 0 | X | 1 | 0 |
| **POP** | 1 | X | 0 | X | X | 0 | 0 | X | 0 | 1 |

*Table 2-2 Main Control Truth Table.*

**LOGIC EQUATIONS FOR MAIN CONTROL.**

RegSrc = R-type + POP

---

RegWr = SW + BGT + BLT

RegWr2 = LW.POI

---

ExtOp = ANDI

ALUSrc = ANDI + ADDI+ LW + LW.POI + SW

Mem_Rd = LW + LW.POI

Mem_Wr = SW

WB_data = LW + LW.POI

PushSig = CALL + PUSH

PopSig = RET + POP

---

## MAIN CONTROL MODULE.

```
module main_control(op_code,reg_des,reg_w1,reg_w2,ext_op,alu_src,mem_red,mem_wr,wb_data,sig_push,sig_pop,sp,next_sp,StackOp);
    input [5:0] op_code;
    input [31:0] sp;
    output reg  reg_des,reg_w1,reg_w2,ext_op,alu_src,mem_red,mem_wr,wb_data,sig_push,sig_pop,StackOp;
    output reg [31:0] next_sp;
    wire [5:0] BGT,BLT,BNE,BEQ,JMP,CALL,RET,AND,ADD,SUB,ANDI,ADDI,LW,LW_POI,SW,POP,PUSH;

    assign AND =  6'b000000;
    assign ADD =  6'b000001;
    assign SUB =  6'b000010;
    assign POP =  6'b010000;
    assign PUSH = 6'b001111;
    assign SW  =  6'b000111;
    assign RET =  6'b001110;
    assign LW  =  6'b000101;
    assign LW_POI=6'b000110;
    assign JMP  = 6'b001100;
    assign ANDI = 6'b000011;
    assign ADDI = 6'b000100;
    assign CALL = 6'b001101;
    assign BGT =  6'b001000;
    assign BLT =  6'b001001;
    assign BEQ =  6'b001010;
    assign BNE =  6'b001011;
    always @(*) begin
        reg_des = ((op_code == AND) ||(op_code == POP) ||(op_code == ADD) || (op_code==SUB));//R-type + POP
        reg_w1 = ((op_code != SW) ||(op_code != BGT) ||(op_code != BLT));   //  SW + BGT + BLT
        reg_w2 = (op_code == LW_POI)? 1'b1 : 1'b0;//LW.POI
        ext_op = (op_code == ANDI) ? 1'b0 : 1'b1; //ANDI
        alu_src = ((op_code == ANDI) ||(op_code == ADDI) ||(op_code == LW) || (op_code==LW_POI) || (op_code==SW)); //ANDI + ADDI+ LW + LW.POI + SW
        mem_red = ((op_code == LW) || (op_code ==LW_POI));//LW + LW.POI
        mem_wr = (op_code==SW);//SW
        wb_data = ((op_code == LW) || (op_code==LW_POI));//LW + LW.POI
        sig_push = ((op_code == CALL) || (op_code==PUSH));//CALL + PUSH
        sig_pop = ((op_code == RET) || (op_code==POP));//RET + POP
        StackOp = ((op_code == RET) || (op_code==CALL));//RET + CALL

        if(sig_push)
            begin
                next_sp = sp - 1;
            end
        else if (sig_pop)
            begin
                next_sp = sp + 1;
            end
        else
            begin
                next_sp = sp;
            end

    end

endmodule
```

*Figure 2-20 Main control code.*

- PC control signals.

**DESCRIPTION.**

- PC_Src: Given the existence of multiple potential values for the next program counter (PC) value, a signal is required to designate the chosen value. The available options include PC+1, branch instructions (BGT, BLT, BEQ, BNE), jump instructions (JMP, CALL), and return (RET).

  - PC+1: Since the memory cell in our implementation is 32 bits wide, each instruction is stored in one memory cell. Consequently, after fetching it, the program counter (PC) value must be incremented by 1.

  - (BGT, BLT, BEQ, and BNE): It will serve as the next program counter (PC) value for the BRANCH instructions in the event that the branch is taken.

  - JMP, CALL: It will represent the subsequent program counter (PC) value in J-Type instructions.

  - RET: The PC source will be set to the value at the top of the stack.

- Flags: Zero, negative and Overflow.
  The status of zero, overflow, and negative flags in conditional branch instructions (BNE, BEQ, BGT, BLT) is typically influenced by the comparison or operation that precedes the branch

  - BNE (Branch if Not Equal):

    Zero Flag: Set to 0 when operands are not equal.
    Overflow Flag: Its status may not be directly affected.
    Negative Flag: Its status may not be directly affected.

  - BEQ (Branch if Equal):

    Zero Flag: Set to 1 if the operands are equal.
    Overflow Flag: Its status may not be directly affected.
    Negative Flag: Its status may not be directly affected.

    BGT (Branch if Greater Than):

    Zero Flag: Set to 0 if the result of the comparison is greater than zero.
    Overflow Flag: Set to 1.
    Negative Flag: Set to 0 if the result is positive.

- BLT (Branch if Less Than):

  Zero Flag: Set to 0 if the result of the comparison is less than zero.
  Overflow Flag: Set t0 0.
  Negative Flag: Set to 1 if the result is negative.

**PC CONTROL SIGNALS.**

| Signal | value | Effect |
|---|---|---|
| **PcSrc** | 00 | NextPC = PC +1 |
| | 01 | NextPC = pc[31:26] + offset26 |
| | 10 | NextPC = pc + signEXT(imm16) |
| | 11 | NextPC = Top of the stack |

*Table 2-3 PC control signals.*

**PC CONTROL TRUTH TABLE.**

| Opcode | Zero flag | Over flow flag | Negative flag | PC_Src |
|---|---|---|---|---|
| **R type** | X | x | X | 0 |
| **JMP** | X | X | x | 1 |
| **CALL** | X | x | X | 1 |
| **RET** | X | x | X | 3 |
| **BGT** | 1 | x | X | 0 |
| **BGT** | 0 | 1 | x | 2 |
| **BGT** | 0 | 0 | 1 | 0 |
| **BLT** | 1 | x | X | 0 |
| **BLT** | 0 | 1 | x | 0 |
| **BLT** | 0 | 0 | 1 | 2 |
| **BNE** | 0 | x | X | 2 |
| **BNE** | 1 | x | x | 0 |
| **BEQ** | 0 | x | X | 0 |

*Table 2-4 PC Control Truth Table.*

## LOGIC EQUATIONS FOR PC CONTROL.

if (Op == RET)

PCSrc = 3;

else if( (Op == JMP) ||(Op == CALL))

PCSrc = 1;

else if (Op == BGT && Over flow flag == 1 && Zero flag ==0)

|| (Op == BLT && Over flow flag == 0 && Zero flag ==0 && Negative flag ==1)

|| (Op == BNE &&Zero flag ==0)

||(Op == BEQ &&Zero flag ==1)

PCSrc = 2;

else PCSrc = 0;

## PC CONTROL CODE.

```
module pc_control(op_code,zero_flag,neg_flag,overflow_flag,pc_src);
    input [5:0] op_code;
    input zero_flag,neg_flag,overflow_flag;
    output reg [1:0] pc_src;
    wire [5:0] BGT,BLT,BNE,BEQ,JMP,CALL,RET;
    assign RET =  6'b001110;
    assign JMP  = 6'b001100;
    assign CALL = 6'b001101;
    assign BGT =  6'b001000;
    assign BLT =  6'b001001;
    assign BEQ =  6'b001010;
    assign BNE =  6'b001011;


    always @(*)begin
        if (op_code == RET )  begin
            pc_src =2'b11;
        end
    else if( (op_code == JMP) ||(op_code == CALL)) begin

        pc_src =2'b01;
    end
    else if( ((op_code == BGT) && (overflow_flag == 1) && (zero_flag ==0))
        ||((op_code == BLT) && (overflow_flag== 0) && (zero_flag ==0) && (neg_flag ==1))
        || ((op_code == BNE) &&(zero_flag ==0))
        ||((op_code == BEQ) &&(zero_flag ==1)))begin

        pc_src =2'b10;
        end
    else
        begin
        pc_src =2'b00;
        end

    end

endmodule

```

*Figure 2-21 PC control code.*

## 3. Simulation and Testing.

The test bench for the main module: RTL.

```
module tb_main_control;

  // Inputs
  reg [5:0] op_code;

  // Outputs
  reg reg_des, reg_w1, reg_w2, ext_op, alu_src, mem_red, mem_wr, wb_data, sig_push, sig_pop ,StackOp,next_sp,sp;

  // Instantiate the main_control module
  main_control uut (
    .op_code(op_code),
    .reg_des(reg_des),
    .reg_w1(reg_w1),
    .reg_w2(reg_w2),
    .ext_op(ext_op),
    .alu_src(alu_src),
    .mem_red(mem_red),
    .mem_wr(mem_wr),
    .wb_data(wb_data),
    .sig_push(sig_push),
    .sig_pop(sig_pop),
     .StackOp(StackOp),
     .next_sp(next_sp),
     .sp(sp)
  );

  // Clock generation
  initial begin
    // No clock is needed for this combinational module
  end
  // Test cases
  initial begin
    // Test Case 1
    op_code = 6'b000011; // Some value for AND
    #10;
    $display("Test Case 1 Results:");
    $display("reg_des=%b, reg_w1=%b, reg_w2=%b, ext_op=%b, alu_src=%b, mem_red=%b, mem_wr=%b, wb_data=%b, sig_push=%b, sig_pop=%b",
             reg_des, reg_w1, reg_w2, ext_op, alu_src, mem_red, mem_wr, wb_data, sig_push, sig_pop);
    // Test Case 2
    op_code = 6'b000111; // Some value for SW
    #10;
    $display("Test Case 2 Results:");
    $display("reg_des=%b, reg_w1=%b, reg_w2=%b, ext_op=%b, alu_src=%b, mem_red=%b, mem_wr=%b, wb_data=%b, sig_push=%b, sig_pop=%b",
             reg_des, reg_w1, reg_w2, ext_op, alu_src, mem_red, mem_wr, wb_data, sig_push, sig_pop);

    $stop;
  end
```

## 4. Teamwork.

Each team member contributed equally to all aspects of the project, collaborating on tasks spanning design, implementation, simulation, testing, report writing, and project presentation.

## 5. Conclusion.

Developing a multi-cycle system calls for a high degree of precision and accuracy, entailing the integration of a significant multitude of components.

# 6. References.


[1] [Online]: https://www.arm.com/glossary/risc [Accessed 25 1 2024].

[2] [Online]: https://en.wikipedia.org/wiki/Multi-cycle_processor [Accessed 25 1 2024].

[3] [Online]: https://stackoverflow.com/questions/56398029/single-processor-systems-vs-multi-processor-systems#:~:text=But%20multi%20processors%20contains%20two,processors%20can%20perform%20different%20tasks.&text=A%20Single%20processor%20contains%20only%20one%20proces [Accessed 25 1 2024].

[4] [Online]: https://www.geeksforgeeks.org/differences-between-multiple-cycle-datapath-and-pipeline-datapath/ [Accessed 25 1 2024].