Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370

COMPUTER ARCHITECTURE

Design and Verification of a Multi-Cycle RISC Processor in Verilog

---

Partners: Aseel Deek **ID:** 1190587

Lojain Abdalrazaq **ID:** 1190707

Fayez Backleh **ID:** 1190216.

Instructors Names: Dr. Aziz Qaroush.

Dr. Ayman Hroub.

Sections: 1, 3

July , 10 2023

Table of Contents

Abstract 3

Processor Stages: Design and Testing 4

1. **Instruction Fetch Stage (IF)** 4
2. **Instruction Decode Stage (ID)** 6
   - ***2.1 Decoder Stage*** 6
   - ***2.2 Register File*** 9
   - ***2.3 Extender*** 12
3. **Execution Stage** 14
4. **Memory Access** 17
5. **Write Back** 19
6. **Buffer Register** 20
7. **Control Stack** 21
8. **Control Unit** 22

The Data path Control Signal: 23

Control Truth Table: 23

Logic equation of the control signals: 24

Finite State Machine: 25

Conclusion 30

Appendix 31

Appendix A: Instruction Fetch (IF): 31

Appendix B: Instruction Decode (ID): 33

Appendix C: Extender: 36

Appendix D: Register File (RF): 37

Appendix E: Register Buffer: 39

Appendix F: Memory (M): 39

Appendix G: Write Back (WB): 41

Appendix H: Control Stack: 42

Appendix J: Control Unit (CU): 44

Abstract

This project aims to design and validate a basic multi-cycle RISC processor using Verilog. The processor follows a 32-bit instruction format and incorporates 32 general-purpose registers (R0 to R31) along with a program counter (PC). Additionally, it includes a control stack to store return addresses, managed by a stack pointer (SP) that points to the top of the stack. The stack is assumed to be implemented using on-chip memory, and the initial value of SP is zero. The processor supports four instruction types: R-type, I-type, J-type, and S-type. It features an ALU with a "zero" output signal that indicates when the result of the previous ALU operation is zero. The design also includes separate memories for data and instructions**.**

Processor Stages: Design and Testing

In this particular section of the report, we will extensively discuss and explore the design and testing processes for each stage, namely **fetch, decode, execution, memory access,** and **write back**, in the multiprocessor data path. These stages will be implemented and tested using Verilog within the Quartus software environment.

1. Instruction Fetch Stage (IF)

Instruction fetch considered to be the first stage in the data path. The fetching stage in a processor is responsible for retrieving the next instruction from instruction memory block and preparing it for further processing in subsequent stages. In addition, the PC is updated in the fetching stage to prepare for the retrieval of the next instruction.

The instruction memory module is responsible for storing our **15** instructions divided into the 4 instruction types (R**-type, I-type, J-type, and S-type**) of the RISC processor and providing the next instruction to the fetching stage.

Firstly, the instruction memory module consist of the following inputs:

- **reset**: A synchronous reset signal.
- **clk**: The clock signal.
- **PCSrc**: A 2-bit input representing the PC (program counter) source.
- **JA**: The jump address input, a 32-bit value.
- **BTA**: The branch target address input, a 32-bit value.

While the outputs of the fetching block are as the following:

- **nextpc**: A 32-bit output representing the next program counter value.
- **instruction**: A 32-bit output representing the fetched instruction.
- **valid**: A 1-bit output indicating whether or not the instruction was being fetched.

Initially, a memory (**mem)** array is initialized with a set of 15 instructions. Each instruction is represented as a 32-bit binary value and stored in the corresponding memory location.

```
reg [31:0] mem [14:0];
initial begin
    mem[0] = 32'b00000100110000000011100000000000; // AND
    mem[1] = 32'b00001100110000000011100000000000; // ADD
    mem[2] = 32'b00010100110000000011100000000000; // SUB
    mem[3] = 32'b00011001010000001010000000000000; // CAM
    mem[4] = 32'b00000000111000000000000000100100; // ANDI
    mem[5] = 32'b00001001110000000000000000100100; // ADDI
    mem[6] = 32'b00010001110000000000000000110100; // LW
    mem[7] = 32'b00011001110000000000000000110100; // SW
    mem[8] = 32'b00100001110011100000000010000100; // BEQ
    mem[9] = 32'b00000000000000000000000000001000010; // J //
    mem[10]= 32'b00001000000000000000011111010000010; // JAL
    mem[11]= 32'b00000000111001010000000000010000110; // SLL
    mem[12]= 32'b00001001110010101000000000010000110; // SLR
    mem[13]= 32'b00010001110010101000100000000000110; // SLLV
    mem[14]= 32'b00011001110010101000100000000000110; // SLRV
end
```

Figure 2 Instructions saved in the instruction memory array.

During running time of the fetching stage, and since the saved value in the **Mem**[0] is the value of AND the first instruction that will be out of IM is the **ADD instruction (R-type)**, and the second one will be Mem[1] = ADD based on the following instruction encoding:

No.	Instr	Meaning	Function Value
		R-Type Instructions	
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001

Figure 1 ADD R-Type instruction encoding.

And the simulation is as the following:

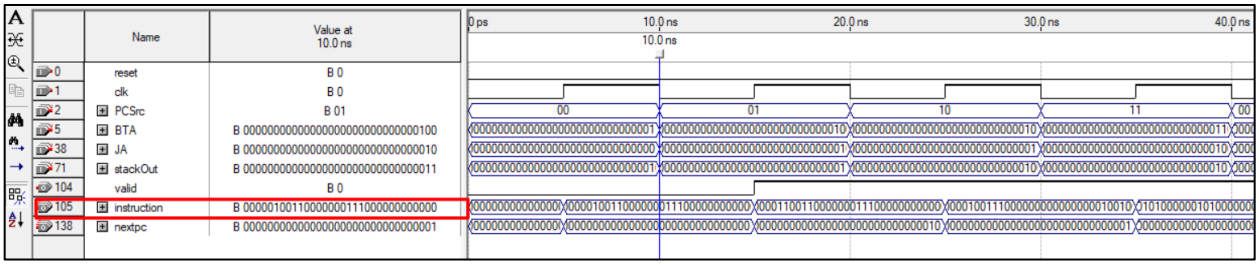

Figure 4 Instruction memory simulation.

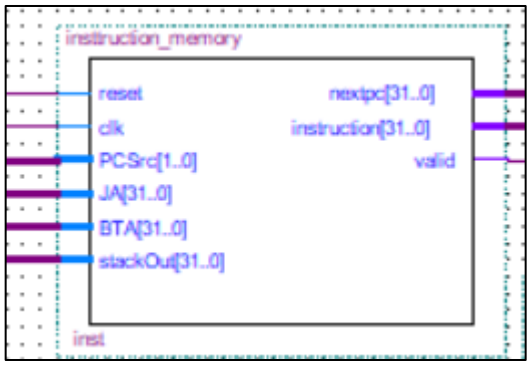

Figure 3 Instruction Memory Block Diagram.

2. Instruction Decode Stage (ID)

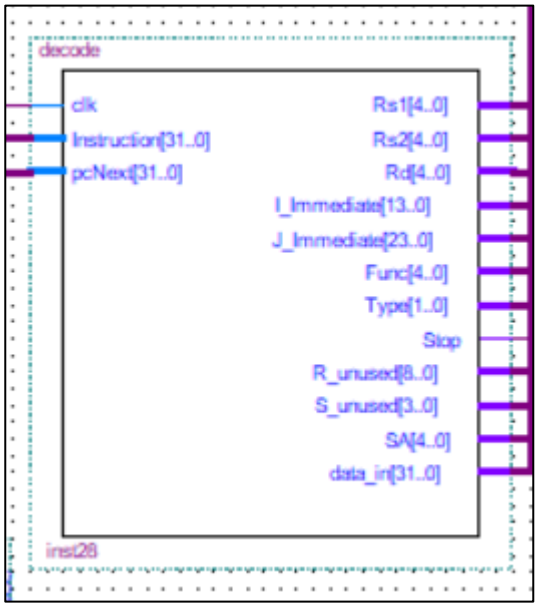

Figure 5: Decode Stage Diagram

The second stage of the data path is the instruction decode stage (ID) that plays a crucial role in preparing the necessary information and operands for the execution stage by extracting fields from the fetched instruction, extend the immediate values using the extender module, and obtaining register values from the register file.

2.1 Decoder Stage

In this stage, the **decoder** module takes the fetched instruction as input and performs decoding operations to determine the type of the instruction (**R-type, J-type, I-type, or S-type**). It extracts specific fields from the instruction, such as register addresses, immediate values, function codes, and other relevant information based on the instruction type. The following figures shows the instruction format for each type:

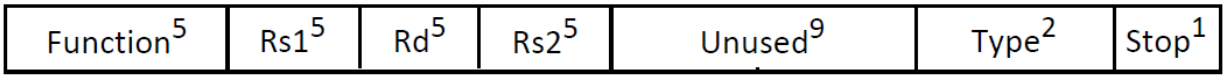

Figure 6 R-Type Format.

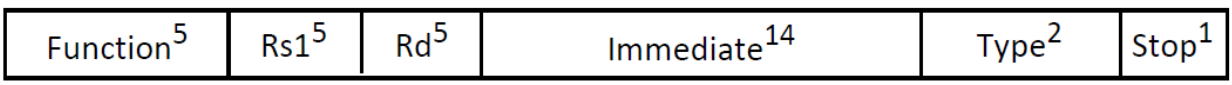

Figure 7 I-Type Format.

Function^5	Signed Immediate^{24}	Type^2	Stop^1

Figure 9 J-Type Format.

Function^5	Rs1^5	Rd^5	Rs2^5	SA^5	Unused^4	Type^2	Stop^1

Figure 8 S-Type Format.

The decoding operation for the **R-Type** instruction (AND) is as the following simulation:

It is noticed that based on the type in the 32-bit input instruction which is (00), the output fields following:

- ✓ “00000”: Function type.
- ✓ “00”: Instruction Type (R-Type).
- ✓ “00000”: Destination register (Rd) address.
- ✓ “10011“: First source register.
- ✓ “00111”: Second source register.
- ✓ “0”: Stop bit.
- ✓ “000000000”: Unused field.

While other fields are not assigned sine, they will not be used such as I or J type immediate.

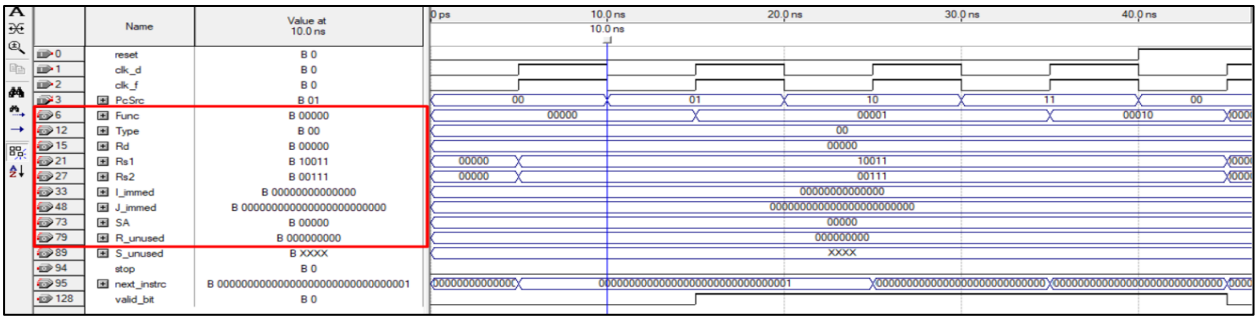

Figure 10 Decoding operation of the AND instruction.

In addition, the simulation results of running R-Type instruction (**ADDI Function**) are as the following figure. It is noticed that when the decoder determined the R-Type function, the 14-bits immediate was e

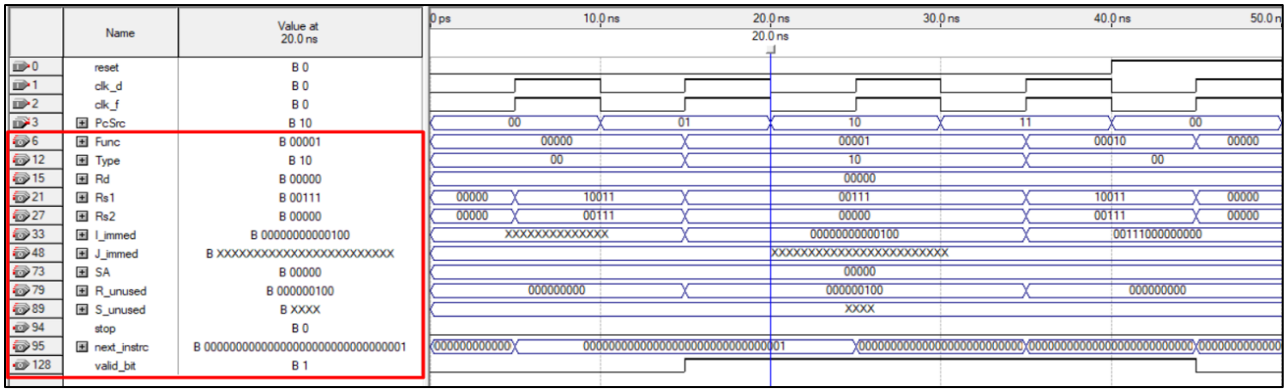

Figure 11 Decoding operation of the ANDI instruction.

- Example of decoding for the JAL instruction:

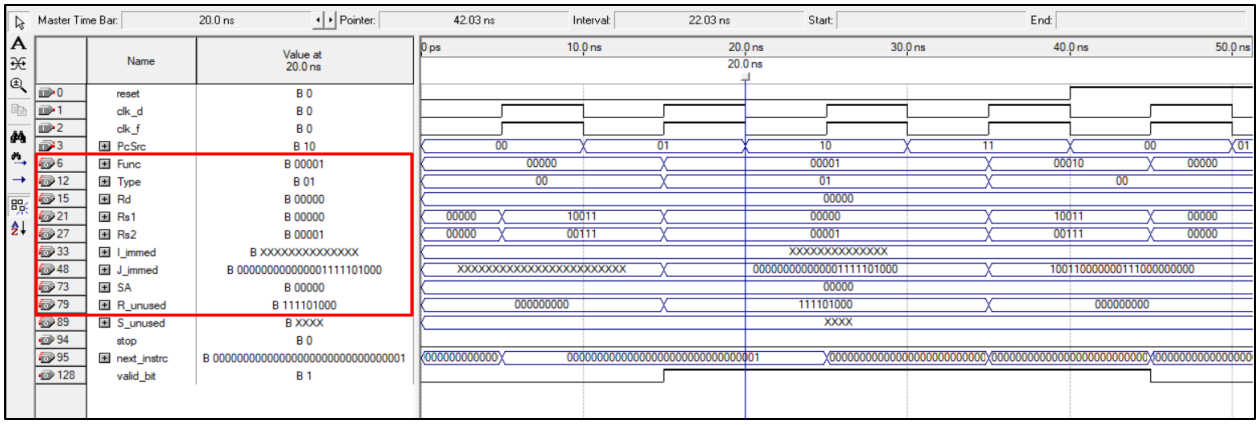

Figure 12: Decoding operation of the Jal Instruction

- Example of decoding for SLL instruction

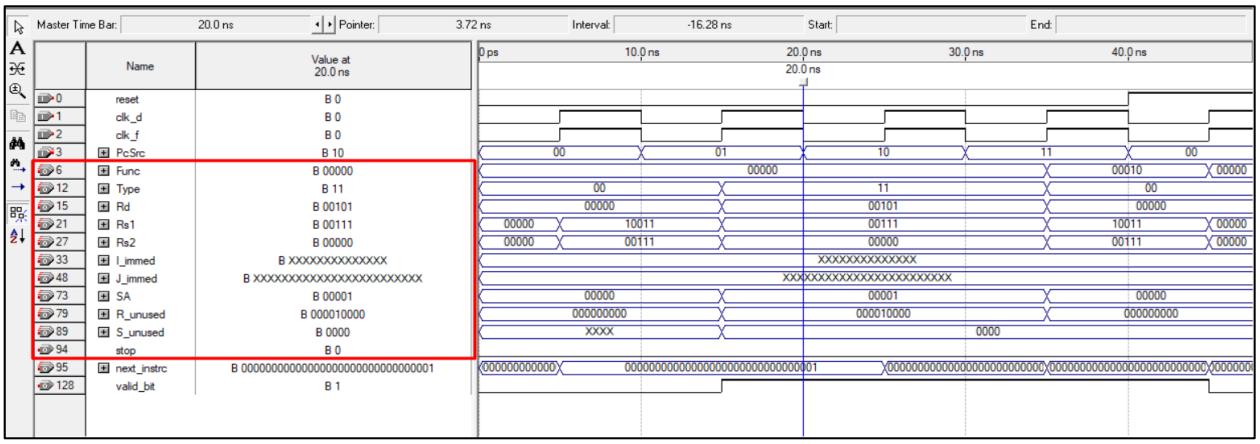

Figure 13: Decoding operation of the S-type instruction

2.2 Register File

The Register File module is a key component in multiprocessor systems that facilitates register-based data storage and retrieval. It consists of a 32-register array, where each register is 32 bits wide. This module enables read and write operations on registers based on input addresses and control signals which are the WriteEnable signal that is controlled by the control unit, and the input clock.

The Register File consists of the following inputs:

- **clk:** The clock signal for synchronous operations.
- **RA:** The address of the first register (Rs1) for reading.
- **RB:** The address of the second register (Rs2) for reading.
- **RW:** The address of the destination register (Rd) for writing.
- **WriteEnable:** A control signal that enables writing to the register file.
- **secOperant:** A control signal that determines the second operand source (Rs2 or Rd) for reading.
- **BusW:** The data value to be written to the destination register.

While the outputs of the Register File are as the following:

- **BusA:** The data content of the first register (Rs1).
- **BusB:** The data content of the second register (Rs2 or Rd) based on the secOperant control signal.

Subsequently, during runtime, the Register File module operates on the rising edge of the clock signal. It checks the WriteEnable signal to determine if a write operation is requested. When WriteEnable is active, the value present in the BusW input is written to the register specified by the RW address. The module also continuously outputs the contents of three registers: RA, RB, and RW. The contents of the RA register are provided as the BusA output, representing the first operand in the instruction. Depending on the secOperant signal, the module selects either the contents of the RB register or the RW register as the BusB output. This flexibility allows the module to handle cases where the destination register (Rd) might be used as the second operand in an instruction rather than just the destination register.

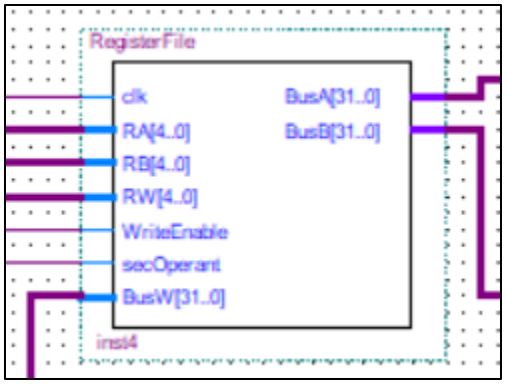

Figure 15 Register File Block Diagram.

Firstly, when the input clock is not active (low), the output buses that contains the values of the registers will be equal 0 in hex.

When the input clock is active (positive edge) and the WriteEnable signal is low, no write operation

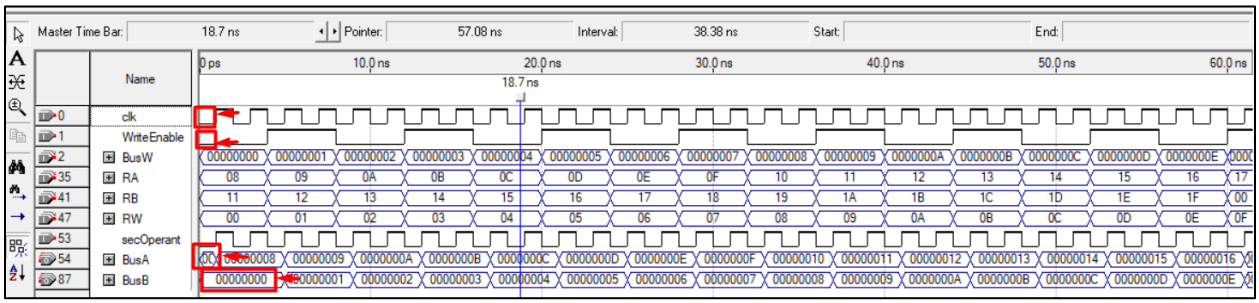

Figure 14 Register File simulation No.1

will occur on the destination register, and since the secOperand signal equals 1, which means that the second operand is Rd, the output buses (RA and RB) will retain the values stored in the register file at RA(First operand Rs1), and RW(Rd) input addresses.

It is worth mentioning that during the initialization phase, each index of the register file array is assigned an initial value equal to its index. For example, registers[8] will have an initial value of 8, registers[6] will have an initial value of 6, and so on. This initialization ensures that each register initially holds a unique value.

From the provided information and the given figure, it is shown that the output values of **RA, RB** are equal to **0A**, and **02**, respectively, since the clock is high, and the WriteEnable is OFF, and the **secoOperant** is 1, which means that the BusB contains the address of the destination register Rd wich is "**02**", and BusA first operand (Rs1 = "**0A**"). In other words, our results are correctly generated.

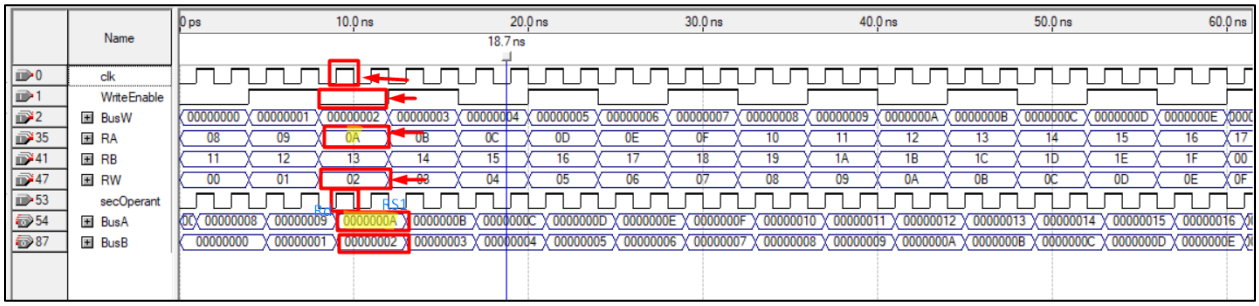

Figure 16 Register File simulation No.2

While in the last case, when both input signals (clock and WriteEnable) are high, the value of the secOperant is 1 (Second operant will be Rd) , then the destination register (Rd) will be updated to the value in the **BusW input.** As shown in the following simulation, the value of the Rd register is updated from the "**00000005**" to "**010D00E5**", which is equals to the value of the write bus input **BusW.**

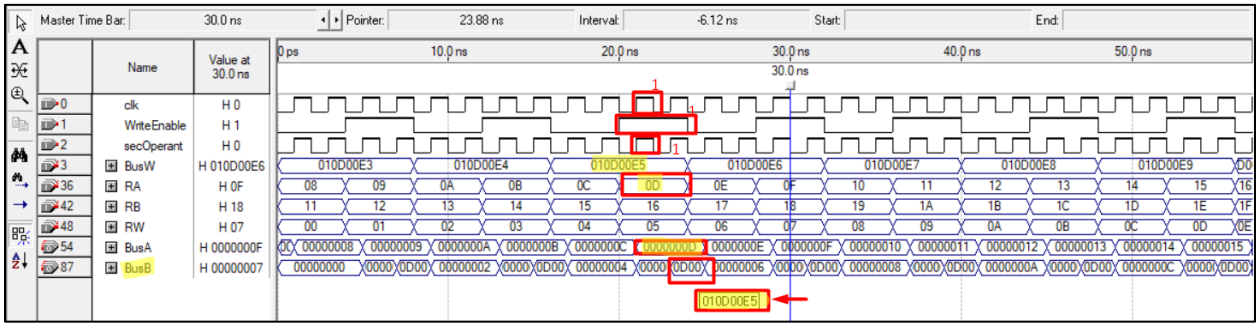

Figure 17 Register File simulation No.3

From the previous results and observations, it is evident that the register file block is functioning correctly and providing the expected output values. The initialization phase ensures that each register is assigned the appropriate initial value, and subsequent read operations correctly retrieve the contents of the specified registers (RA, RB, or Rd). The write operation, triggered by the positive edge of the clock and an active WriteEnable signal, updates the value in the destination register as expected. Overall, the register file block demonstrates proper functionality and meets the requirements for data storage and retrieval within the multiprocessor system.

2.3 Extender

The aim of using an extender in the processor is to ensure consistent and proper data handling within the ALU stage. In this specific processor, the extender is responsible for extending shorter bit-length values to a 32-bit length, which is the standard size used in subsequent operations.

There are two types of immediate values in this processor: the **14-bit I-immediate** and the **24-bit J-immediate.** These immediate values need to be extended to a 32-bit length before being used in the ALU stage. The extender performs this extension by adding additional bits to the immediate value while preserving its original value.

Furthermore, the extender can perform two types of extensions: **signed extension** and **unsigned extension.** The choice of extension type depends on the instruction being executed. For the I-Type instructions we have to specify the type, while in J-Instructions, only signed extension will be performed.

In this module, we have the following inputs:

- **Immd14**: represents I-Type 14-bit immediate value that needs to be extended.
- **Immd24**: represents J-Type 24-bit immediate value that needs to be extended.
- **PC**: represents the Program Counter value, to determine the next instruction to be fetched when the instruction type is **Jump type**.
- **JorI**: determines whether the extension is for a J-type instruction or an I-type instruction.
- **Extop**: This input determines the type of extension to be performed, whether signed or unsigned.

While the output will be 32-bit extended immediate for I-Type, while next PC for J-type.

Figure 18 Extender Block Diagram.

The testing simulation for J-Type instruction is as the following figure. When the input control signal **jori is 0,**then input immediate will be J-type (24-bits immediate), to be extended 32-bit length, and to added later to the 32-bit input PC value to determine the next instruction value to be fetched.As a result, when the 24-bit immediate is **01101,** and PC value with **00011,** the output 32-bit immediate that represent the next PC is **10000**, which is noticed as shown:

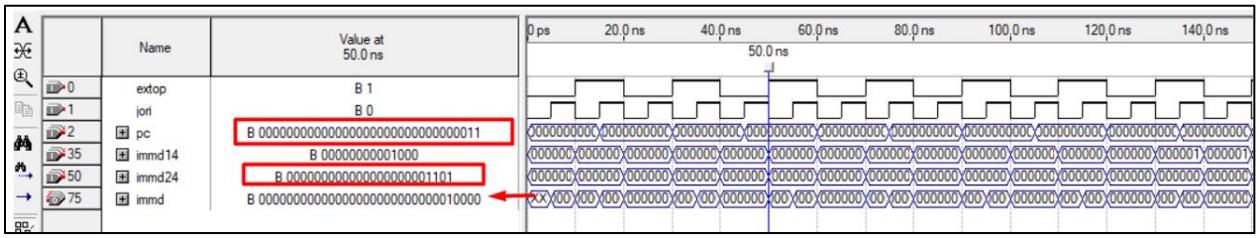

Figure 19 Extender module simulation.

In this approach, we utilized the extender module to calculate the jump address (JA). To verify the functionality of the extender, we connected the IM (Instruction Memory), ID (Instruction Decode), and Extender modules. In order to simulate a jump instruction, we rearranged the sequence of instructions in the IM module to indicate the presence of a jump instruction. The J-Immediate value was set to 8 (10000), and the PC value was 5 (000101).

According to the data path design, for executing a jump instruction, we needed to perform a signed extension on the J-Immediate value. The extended value would then be added to the original PC value to calculate the jump address (JA). The following explanation provides further details on this process and illustrates the resulting value of JA.

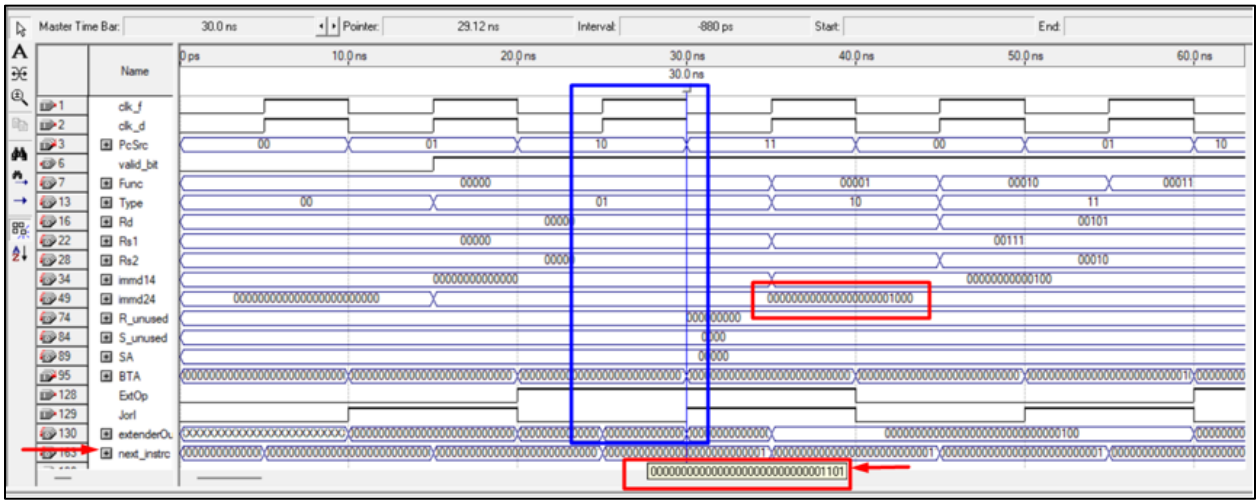

Figure 20 JA Extender approach.

Note: ExtOp & JorR are control signals that control what immediate to do extend of and should it be signed extended of zero extended.

For the figure above, the value of nextPC = J-immd + pc = 5 (0101)+8(1000)= 13(1101).

3. Execution Stage

The execution stage in the data path is responsible for executing the instruction based on the decoded operation and operands. It performs arithmetic and logical operations on the data, using the ALU, and produces the desired result. The ALU (Arithmetic Logic Unit) module performs arithmetic and logical operations based on the inputs and control signals.

The ALU module have the following inputs:

- **clk:** Clock signal for synchronization.
- **rs1:** First operand (BusB).
- **rs2:** Second operand (BusA).
- **rd:** Value of Rd register.
- **func:** Function code for specifying the operation.
- **typ:** Type code for specifying the instruction type.
- **immd14:** 14-bit immediate value.
- **SA:** Shift amount.

And the following outputs:

- **zero_flag:** Flag indicating if the ALU result is zero.
- **carry_flag:** Flag indicating if there is a carry-out.
- **neg_flag:** Flag indicating if the ALU result is negative.
- **ALU_result:** Result of the ALU operation.

The second always block uses a case statement to select the appropriate operation based on the func and typ values controlled by the control unit.

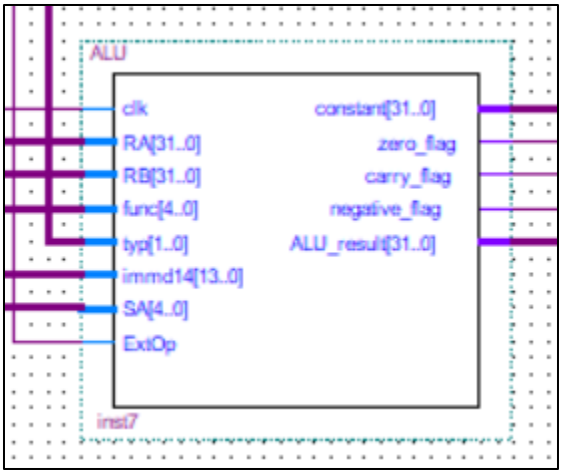

Figure 21 ALU Block Diagram.

Moreover, the following cases represent the ALU simulation:

- **Testing the CMP instruction:**

Since the value of Rs1 is 1 and the value of Rs2 is 12, then the condition of CAM is true and that means the flags will be to 1. The negative flag will be one because if we perform subtraction the value will be negative, also the borrow flag will be since when the result is negative It indicates that the subtraction operation resulted in a negative value or required a borrow.

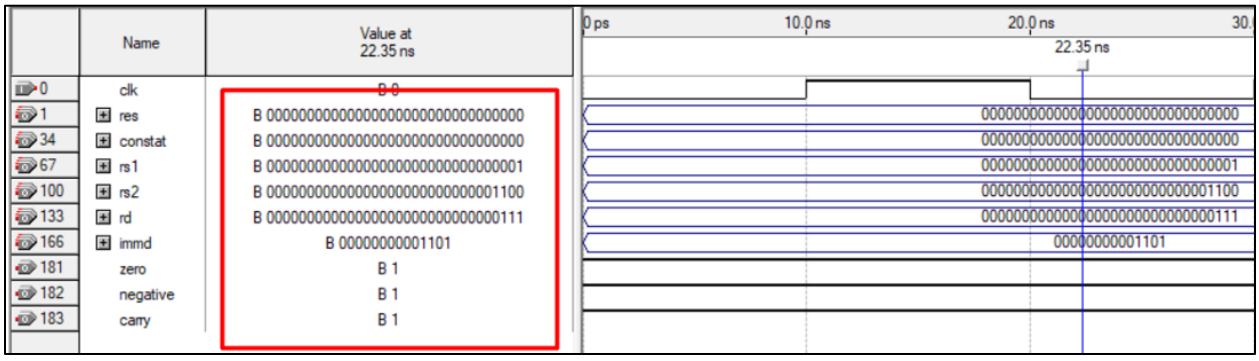

Figure 22 ALU CMP simulation.

- **Testing ADDI instruction:**

Since the value of Rs1 is **7** and the value of immediate is **13**, and the function is ADDI, then the expected output result is **20** which obtained as shown in the following figure:

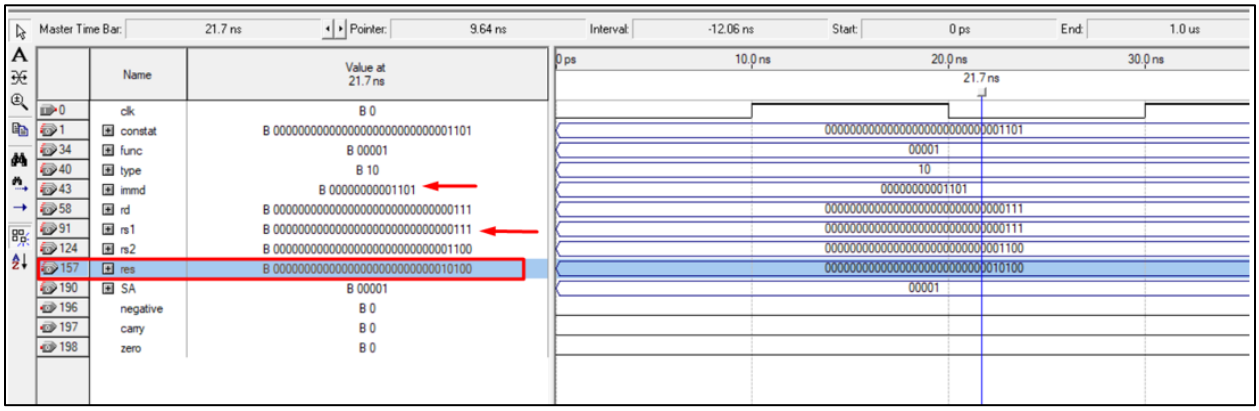

Figure 23 ALU ADDI simulation.

- **Testing SLLV instruction:**

In terms of shifting operation, the value of rs1 was 7 and the shifting amount was 12, the output was shifted to left by 12 bit.

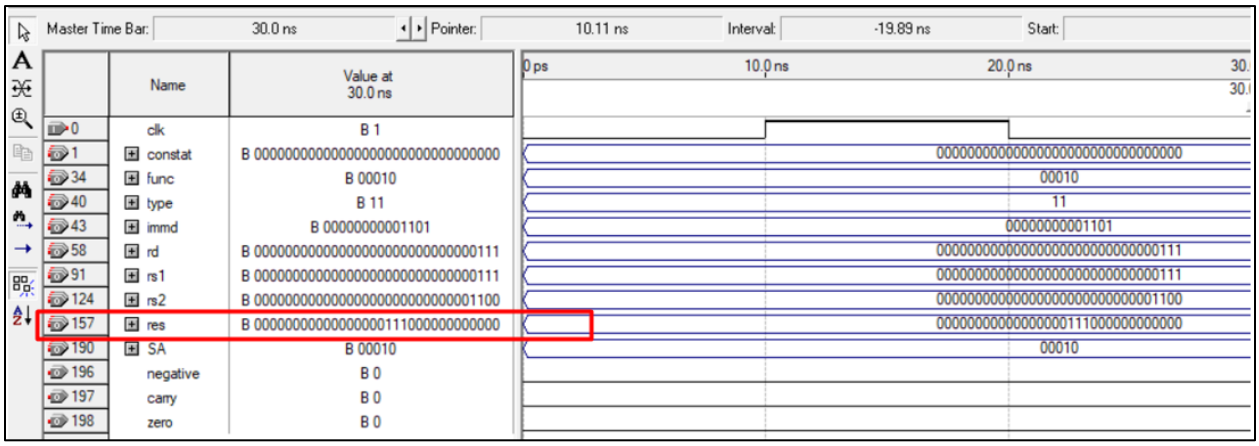

Figure 24 ALU Shifting (l SLLV) simulation.

According to the previous testing results, it is noticed that the ALU unit works correctly, and is giving the predicted values.

4. Memory Access

The Data Memory block is the fourth stage of the multiprocessor data path which is responsible for storing and retrieving data during program execution. It consists of a memory array with **256** elements, each element being **32** bits wide.

This module takes the following inputs:

- **clock**: The clock input is the clock signal used to synchronize the operations of the Data Memory module.
- **address [31:0]:** The address input specifies the memory location from which data should be read (in the case of a load operation) or to which data should be written (in the case of a store operation).
- **data_in [31:0]:** The data_in input contains the data to be stored in the memory during a store operation.
- **RW [1:0]:** The RW input is a control signal that determines the operation to be performed on the data memory. It can take three values:
  1. **'00' or '11':** No operation (no read or write).
  2. **'01'**: Store operation (write data_in to the specified memory location).
  3. **'10'**: Load operation (read data from the specified memory location).

And the outputs of the module are:

- **result [31:0]:** The result output represents the data read from the memory during a load operation. It holds the contents of the memory location specified by the address input. In case of rest operations, the output will be equals to zero.

At the beginning, the memory array is initialized with initial values. The initialization loop assigns initial values to each memory location using a linear feedback shift register (LFSR) algorithm. The LFSR algorithm ensures pseudo-random initial values for each memory location.

The following figure shows the block diagram of the data memory stage with its inputs and outputs:

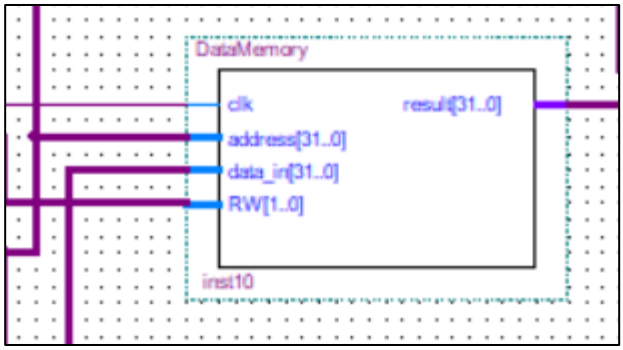

Figure 25 The block diagram of the data memory stage.

Firstly, when the control signal is equals to 0, this means that there is no operation, and the output result will be zero.

While when the input RW control signal is 1, the operation performed is writing the data in the

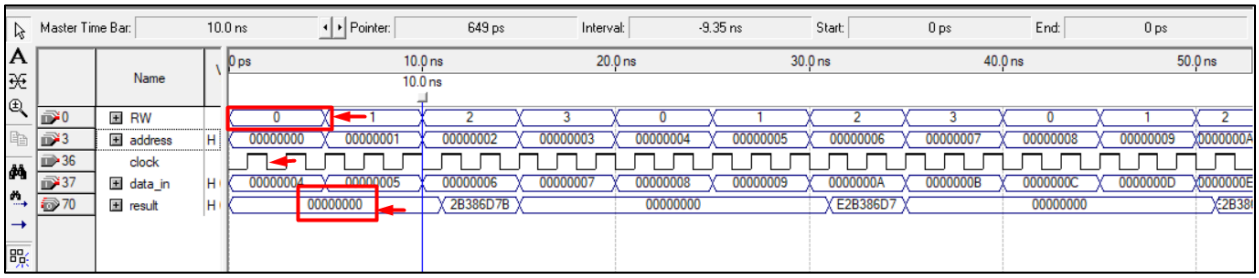

Figure 26 data memory simulation No.1

input (data_in) at the address specified in the input, and the output will be still equals to Zero value.

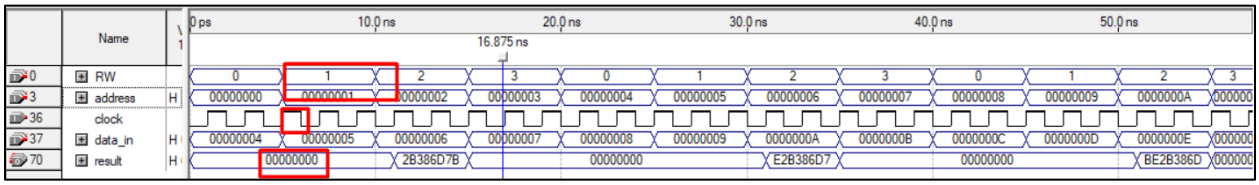

Figure 27 data memory simulation No.2

Finally, when RW control signal is 2, the reading operation is performed at the specified address. The readed content from address 2 is **2B386D7B** as shown in the following figure:

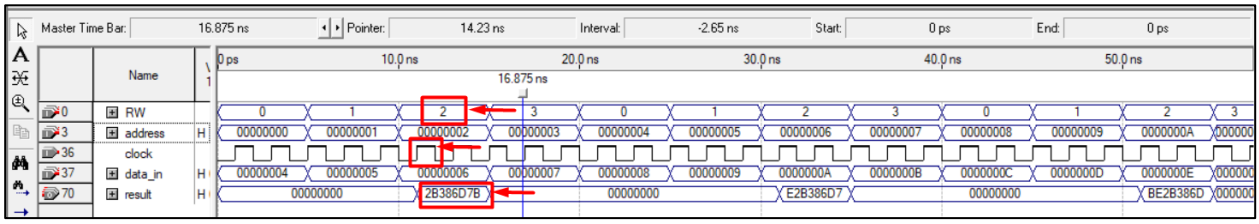

Figure 28 data memory simulation No.3

5. Write Back

The write back module is responsible for selecting the appropriate value to be written back into the destination register (Rd) based on the control signal WBdata.

The inputs to the module are as follows:

- **clk**: Clock signal for synchronous operation.
- **ALU_out**: The output value from the ALU.
- **Data_out**: The data output value from memory (e.g., in case of a load operation).
- **WBdata**: Control signal indicating the source of the value to be written back:
  1. **0**: Indicates that the value from the **ALU** (ALU_out) should be written back.
  2. **1**: Indicates that the value from **memory** (Data_out) should be written back.

The output of the module is Rd_val, which represents the value to be written back into the destination register.

Inside the always block, the write-back operation is performed on the rising edge of the clock (posedge clk). The condition flags are checked to determine the appropriate value to be written back. If WBdata is equal to 1, indicating that the value from memory (Data_out) should be written back, Rd_val is assigned the value of Data_out. Otherwise, if WBdata is not equal to 1 (i.e., 0 or x), indicating that the value from the ALU (ALU_out) should be written back, Rd_val is assigned the value of ALU_out.

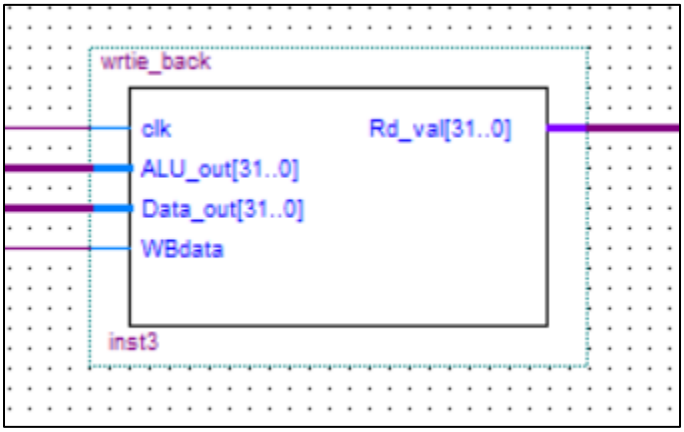

Figure 29 Write back block diagram.

As shown in the following simulation, when the input control signal **WBdata equals 1,** the output data will be the same as Data_out which is the putput data form the memory.

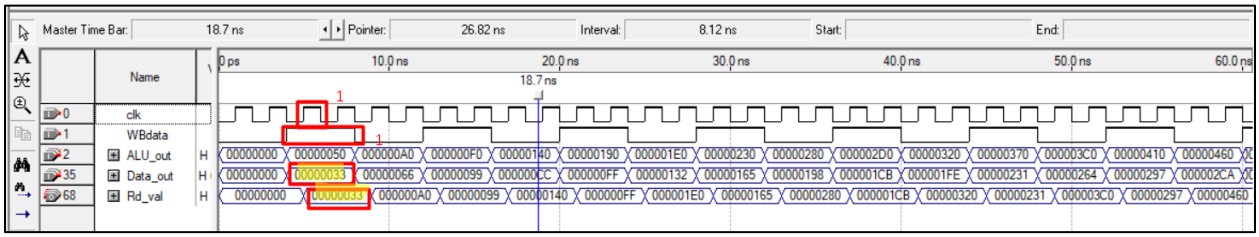

Figure 30 Write back simulation No.1

While, when the control changes to 0 value, the output changed to be the same ALU output value as shown:

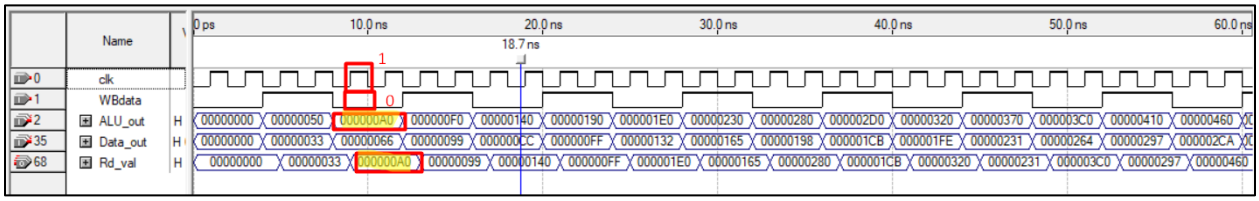

Figure 31 Write back simulation No.2

6. Buffer Register

In a multi-cycle approach, registers play a crucial role in facilitating the orderly execution of instructions by storing intermediate results, maintaining timing, resolving data dependencies, and managing control signals. They provide the necessary buffering and synchronization mechanism between stages, ensuring correct and efficient operation of the processor.

The buffer register module takes a 32-bit input signal, inreg, and outputs a registered version of that signal, outreg.

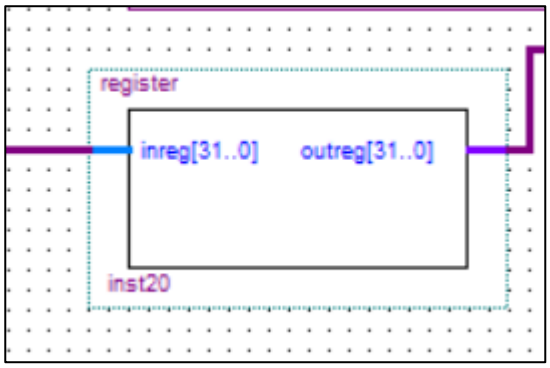

Figure 32 Buffer Register Block Diagram.

7. Control Stack

The stack component is specially referred as the "control stack". We will save or store the addresses we want to return to after executing the call function or the interrupt service routine.

The processes of the stack are a push (storing address), and pop (getting address) during the program execution.

Now, the inputs of the code will be as the following:

- **Clk**:
- **[1:0] pushOrPop**: Control signal for push or pop operation.
    1. **00**: no pop or push operation
    2. **01**: pop operation when an instruction last bit (stop bit) is one.
    3. **10**: push operation only when there is a **JAL** instruction.
- **[31:0] data_In:** The data that we want to push into the stack, in this situation, the data is the return address that we want to save in order to return to when finishing the function call or ISA.

The output of the module is **the [31:0] data_out → which is the address that we return into.**

Firstly, the stack was initialized as an array with 16 elements, each element is 32 bits and a stack pointer (SP) with 4-bits since we have 16 empty locations to fill in the control stack.

When we add (push or store) an address to the stack, we add to the address stack[sp+1], and we update the value of the SP. While when read (pop or load) the address, we load the address (data) that the SP point to, and then we decrement the value of the SP.

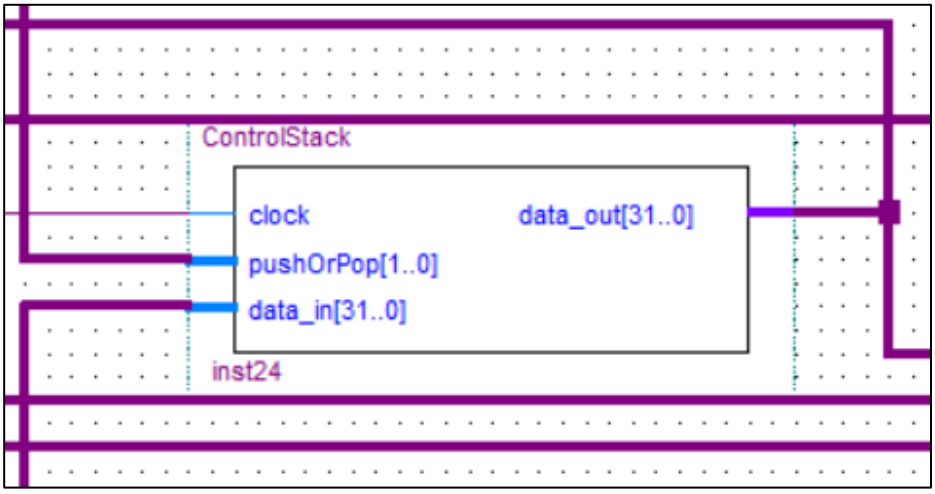

Figure 33:Control Stack Block Diagram

When the control stack module will work only when the input clock is high, and the value of the input signal **pushOrPop** is equals to:

1. **2b'10':** the operation will be **pushing** (storing) the value of the input 32-bit data in the stack. During this operation, the SP will be updated (incremented by 1).
2. **2b'01'**: the operation will be **pop** (reading) the value pointed by the SP. In addition, the SP value will be updated (decremented by 1).

Any other values will not make any update or change on the control stack.

As shown in the following simulation, when the **pushOrPop** control signal equals to 2 (push operation), the control stack saved the input address in it. And then when the clock was high and the control signal was changed to 1 (pop operation), the output result will be "**00000001**" which equals the input value that was pushed previously.

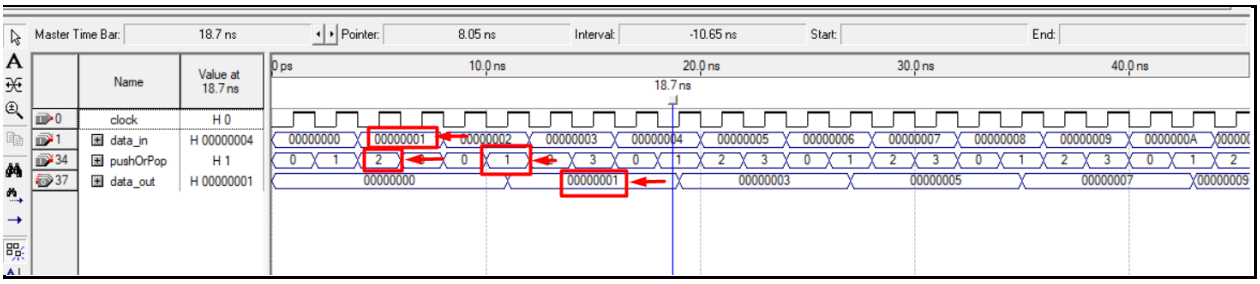

Figure 34: Control Stack simulation

8. Control Unit

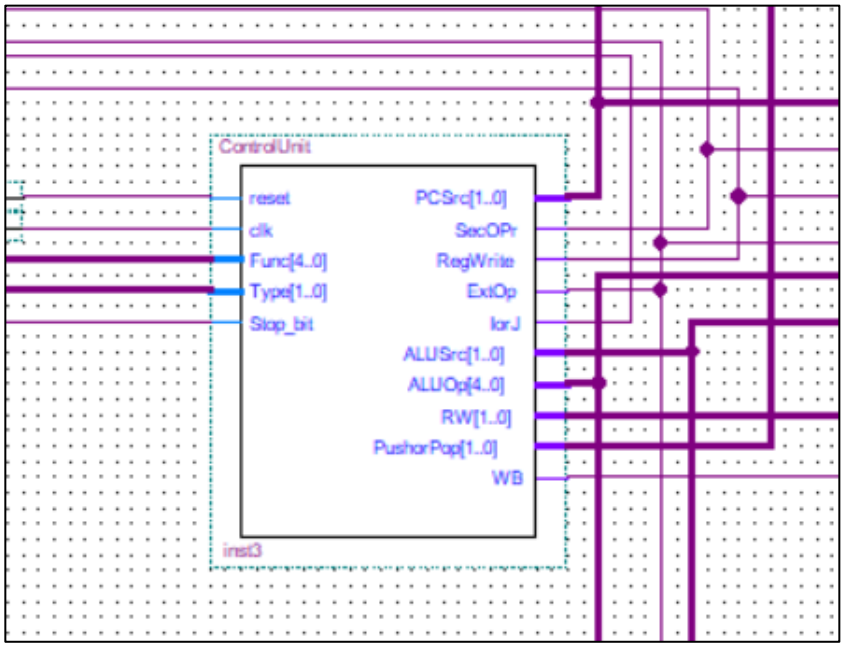

Figure 35: Control Unit Block Diagram

The Data path Control Signal:

Control signal name	Explanation
Pcrsc[1:0]	for PC control: 0 for pc+4, 1 for a j-type and 2 for a taken branch
ALUOp	is the ALU operation as defined in the funct field for all types
ALUsrc	is the instruction type that differentiate the instructions
RegWr	is used to enable the writing of the ALU result
ExtOp	is used to control the extension of the 14-bit & 24-bit Immediate
IorJ	is used to control what immediate enter the extension operation
SecOpt	is used to select the second register as either Rs2 or Rd
RW[1..0]	is used to select the operations that can access the memory
WBdata	is used to select data on BusW as ALU result or Memory Data_out
PushOrPop	Is used to control the instructions that can push or pop on control stack

Control Truth Table:

	PCSrc	SecOpr	RegWr	ExOp	IorJ	ALUSrc	ALUOp	RW	WBdata	PushorPop
AND	00(0)	0 rs2	1 rd	X	X	00	00000	00	0	00
ADD	00(0)	0 rs2	1 rd	X	X	00	00001	00	0	00
SUB	00(0)	0 rs2	1 rd	X	X	00	00010	00	0	00
CAM	00(0)	0 rs2	0	X	X	00	00011	00	X	00
ANDI	00(0)	X	1 rd	0	1	10	00000	00	0	00
ADDI	00(0)	X	1 rd	0	1	10	00001	00	0	00
LW	00(0)	X	1 rd	1	1	10	00010	10	1	00
SW	00(0)	1 rd	0	1	1	10	00011	**01**	X	00
BEQ	**01**(1)	1 rd	**0**	1	1	10	00100	00	X	00
J	10(2)	X	0	1	0	01	00000	00	X	00
JAL	10(2)	X	0	1	0	01	00001	00	X	10
SLL	00(0)	X	1 rd	X	X	11	00000	00	0	00
SLR	00(0)	X	1 rd	X	X	11	00001	00	0	00
SLLV	00(0)	0 rs2	1 rd	X	X	11	00010	00	0	00
SLRV	00(0)	0 rs2	1 rd	X	X	11	00011	00	0	00

1. **PCsrc :**
   When 0 -> Pc = Pc +4

   When 1 -> Pc = Rs

   When 2 -> Pc = Extend(imm_J) + PC

   When 3 -> Pc = popped address from the control stack
2. **SecondOp:** when this signal is zero, the second operant will be read is Rs2, else Rd will be read.
3. **RegWr**: when this signal is 0 then writing on register will be performed else not.
4. **ExOp:** when ExOp is 1, then it means Signed extend, else is unsigned extended.

5. **IorJ:** this signal is used to choose between Immediate and J immediate.
6. **AlUsrc** is the type.
7. **ALUOp** is the function.
8. **RW** 00 is no read no write, 01 is read and 10 is write.
9. **WBData**: when 0 then the ALU operations will write back, 1 the value of **DATA_out** form memory is written.
10. **PushOrPop:** when

    00: No operation,

    10: Then push the address to control stack and increment SP. When JAL instruction

    01: Then pop the address from control stack and decrement that the SP. When the stop bit is one in an instruction.

Logic equation of the control signals:

SecondOp = SW OR BEQ

RegWr = $\overrightarrow{(CAM + SW + BEQ + J + JAL)}$

ExOp = $\overrightarrow{(ANDI + ADDI)}$

IorJ = $\overrightarrow{(J + JAL)}$

WBdata = LW

PushOrPop[0] = (any instruction except(JAL OR J) AND stop bit =1)

PushOrPop[1] = JAL

RW [0] = SW

RW [1] = LW

PCsrc[0] = BEQ OR (any instruction **[**except**(**JAL OR J**)]** AND stop bit =1)

PCsrc[1] = JAL OR J OR (any instruction **[**except(JAL OR J**)]** AND stop bit =1)

Finite State Machine:

1. Instruction Fetch: The processor retrieves the instruction from memory and loads it into the instruction register for further processing.

2. Instruction Decode: The processor analyzes the fetched instruction, determines its type, and extracts relevant information, such as operands and operation codes, to prepare for execution.

3. Execution: The processor carries out the necessary operations specified by the instruction, which may involve multiple clock cycles, to perform calculations or logical operations.

4. Memory Access: If the instruction involves accessing memory, the processor reads or writes data from or to memory locations to perform data operations, such as loading or storing values.

5. Write Back: The processor updates the appropriate internal registers or memory locations with the result of the executed instruction, ensuring that the output of the instruction is properly stored for future use.

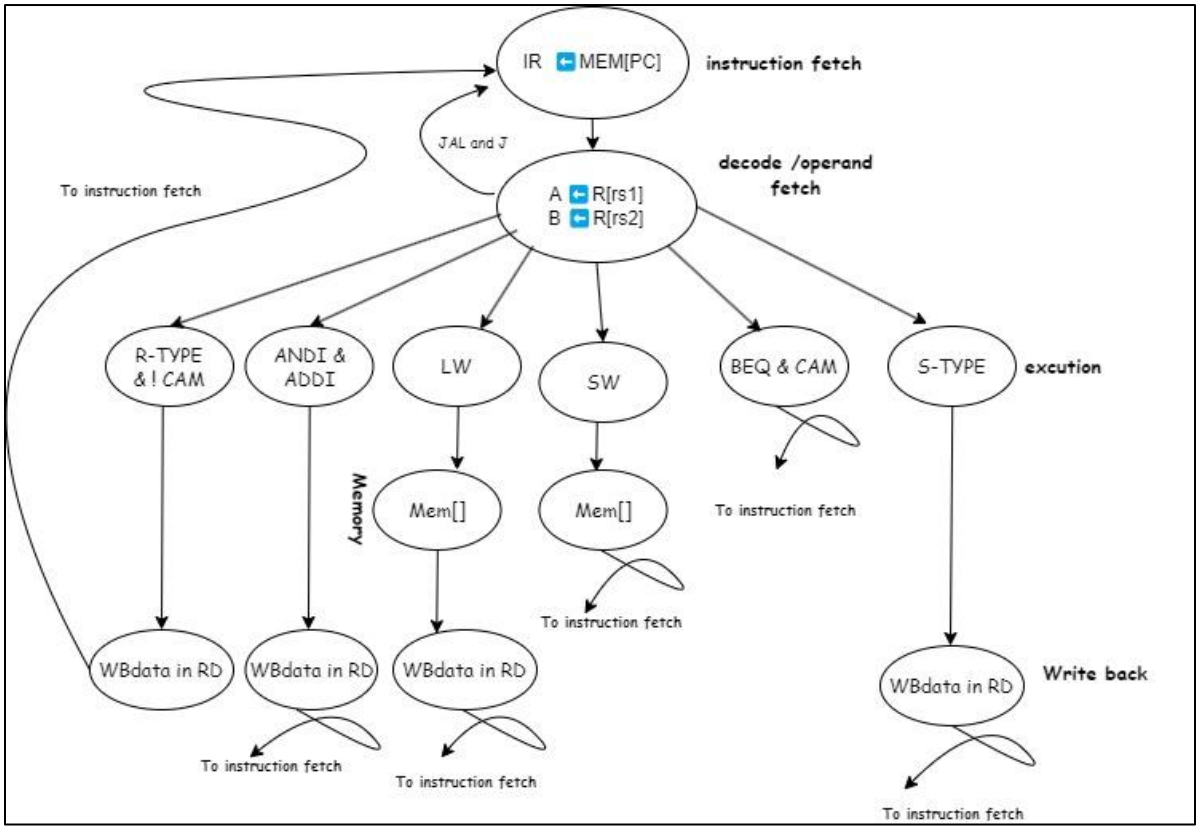

Figure 36: state machine for multicycle Datapath

The finite state machine diagram presented below illustrates the sequential progression between stages, ranging from the fetch stage to the write back stage. Additionally, it depicts the specific behavior associated with each instruction during the execution process.

- **Testing Result for Control Unit:**

Testing for the control unit when there is a **jump instruction**

Figure 37: Control Unit Test No.1

Notice from the above figure, the value of the PC is 2'b10 and that means according to the table (number of table) the instruction is a jump instruct,

This is for testing the **SLR instruction**

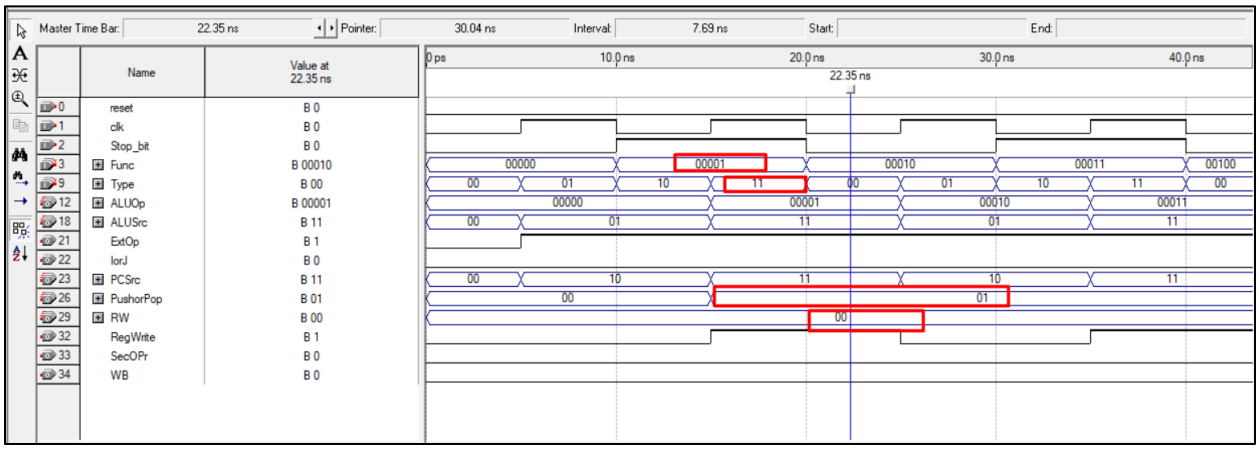

Figure 38: Control Unit Test No.2

From the above figure, the value of the PC is now 2'b11 not 2'b00, because the stop bit is one and that means this instruction is the last instruction in a function, so do a pop for the saved address in the control stack to return to the right position.

- Project Final Datapath:

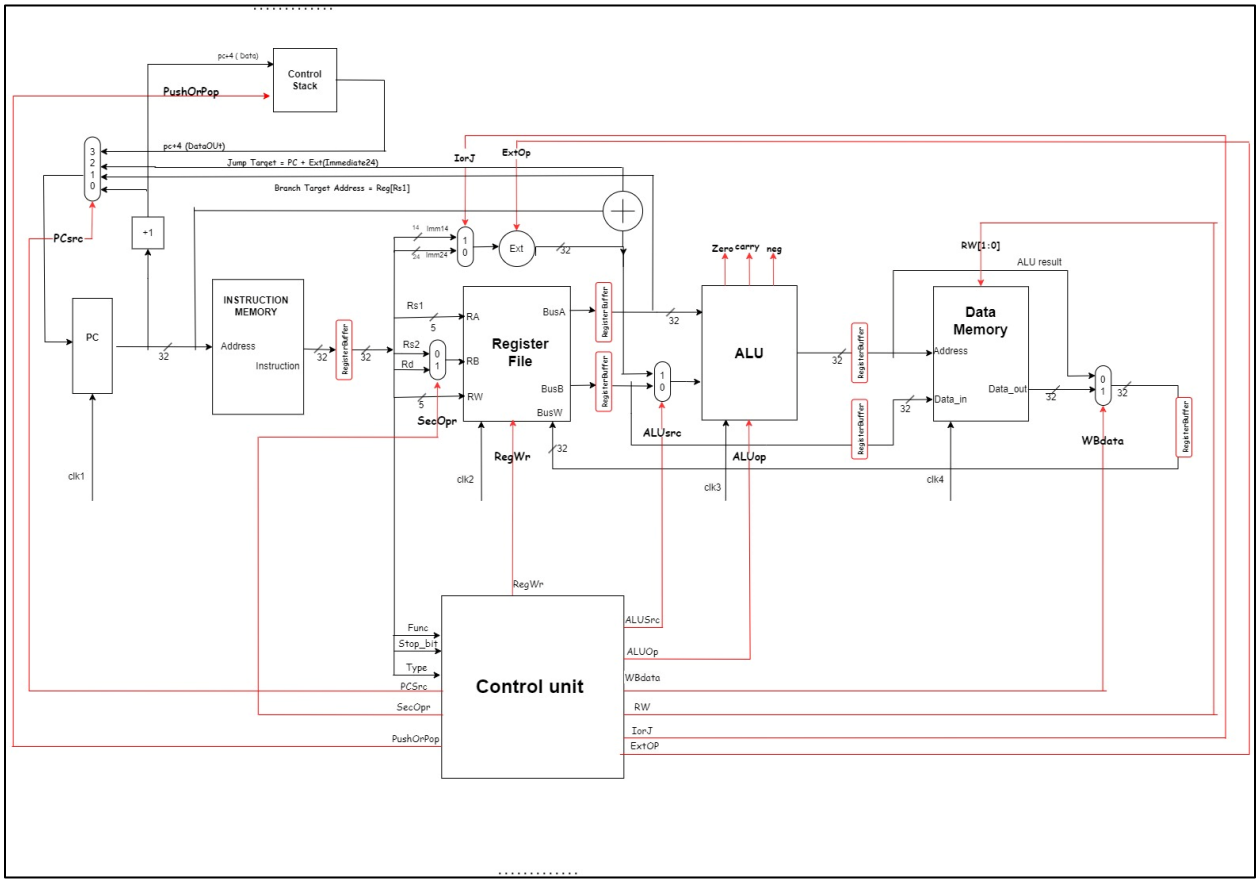

Figure 39: Final Datapath

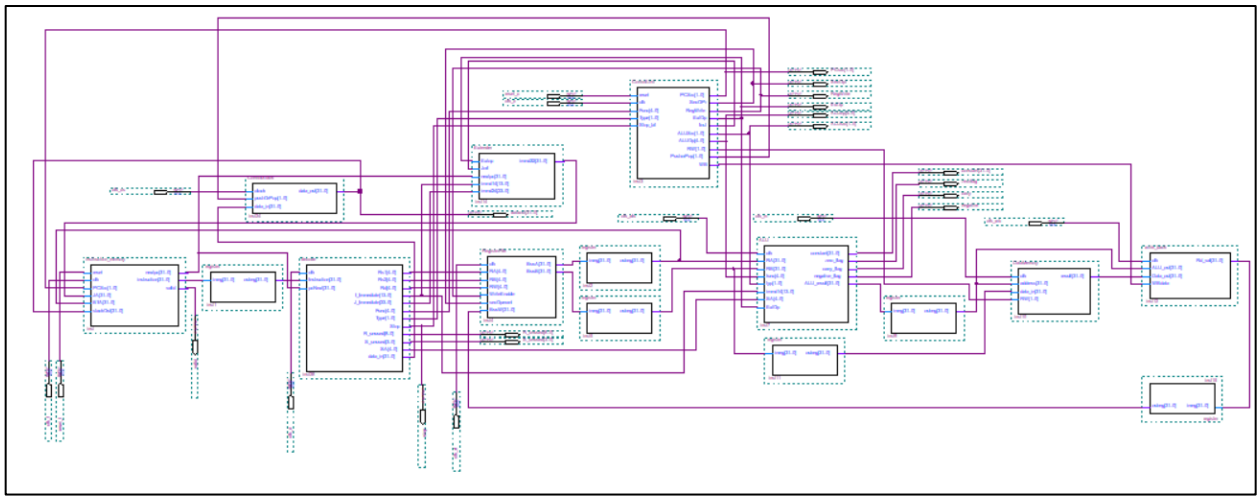

Figure 40: Final Datapath block diagram

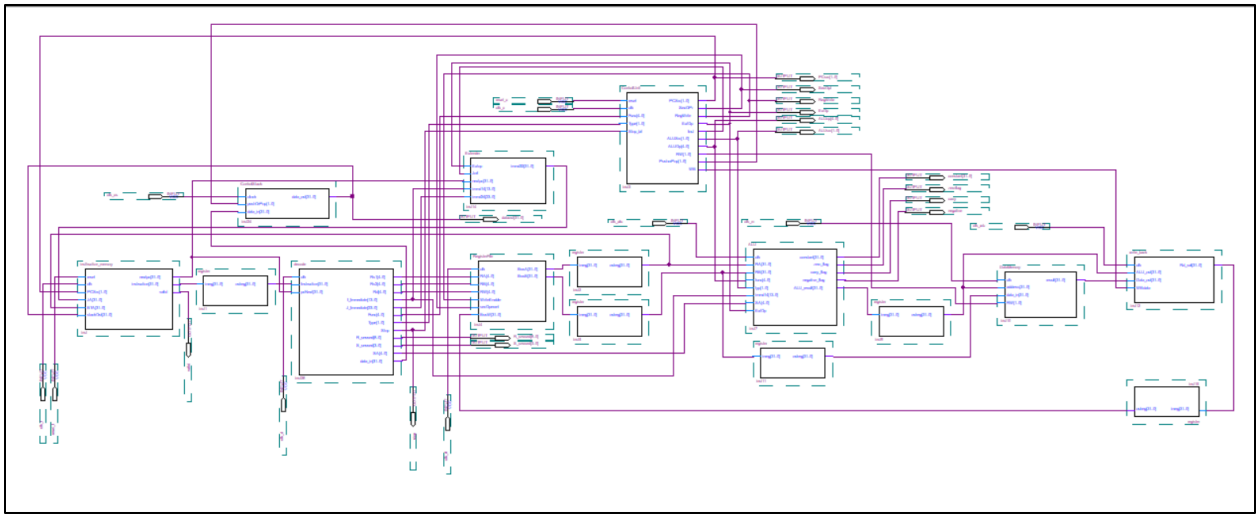

Figure 41: final diagram

- **The Testing results for the Final Diagram are**

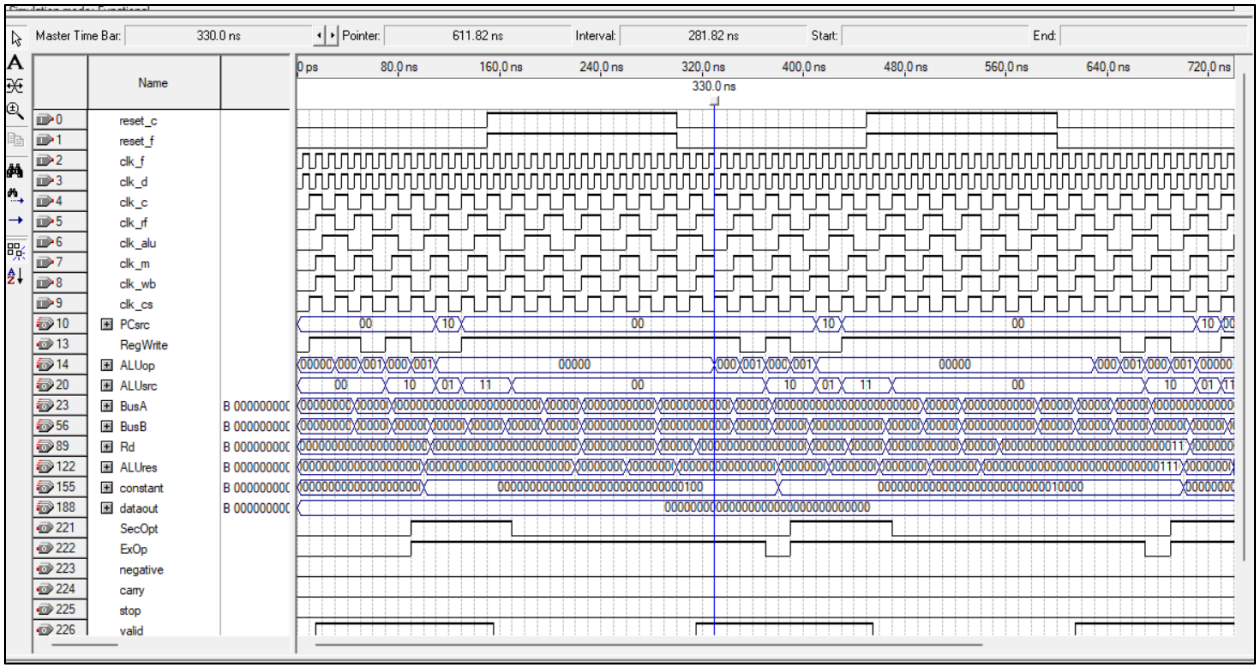

Figure 42: final path test No.1

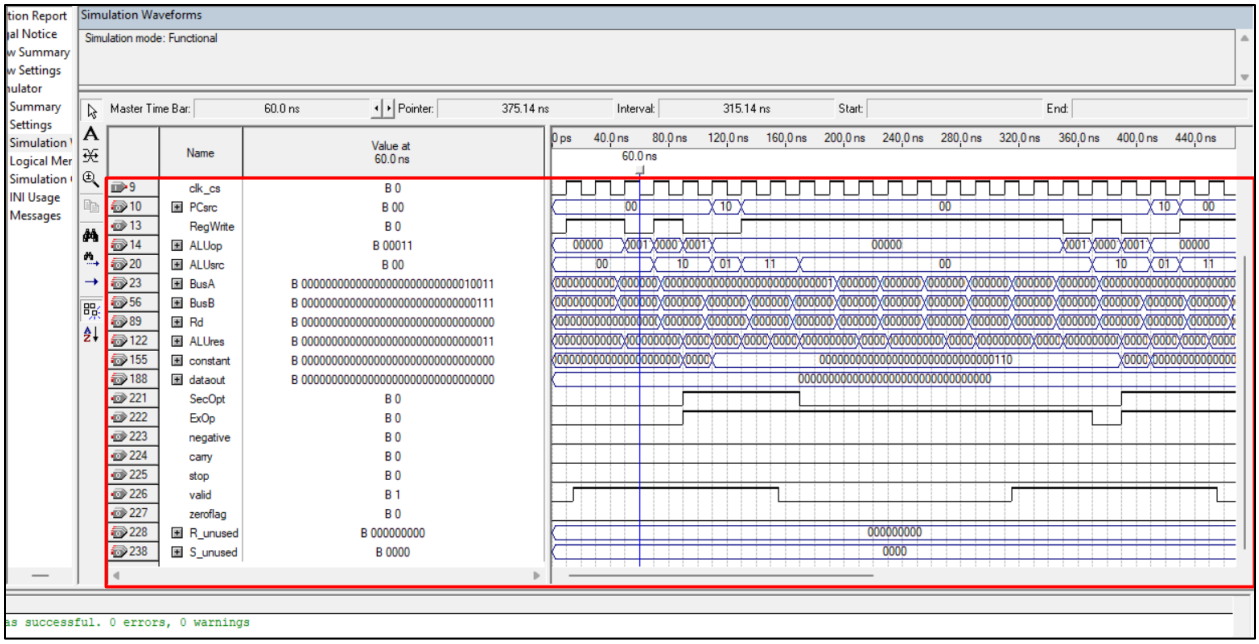

Figure 43: final path test No.2

Note that the First Instruction pass though the Datapath and the output of the ALU is correct, and the value of the control signals are correct.

Conclusion

In summary, this project designed and verified a multi-cycle RISC processor using Verilog. The processor has 32 general-purpose registers, a program counter, and a control stack for storing return addresses. It supports four instruction types and features an ALU with a "zero" output signal. Separate memories are used for data and instructions. Overall, this project contributes to processor architecture advancement and sets the groundwork for future developments in the field.

Appendix

Appendix A: Instruction Fetch (IF):

```
module insttruction_memory(
        input reset,
        input clk,
        input [1:0] PCSrc,
        input [31:0] JA, // jumb address
        input [31:0] BTA,// branch address
        input [31:0] stackOut,  // poped address from the control stack
        output reg [31:0] nextpc,
        output reg [31:0] instruction,
        output reg valid
);


        reg [31:0] mem [14:0];
        initial begin
                mem[0] = 32'b00000000110000000111000000000000;  // AND
32'b00000100110000000111000000000000; // to make the extender work but this first
32'b00000000000000000000000001000010;
                mem[1] = 32'b00001100110000000111000000000000;   // ADD
32'b00001100110000000111000000000000;
                mem[2] = 32'b00010100110000000111000000000000; // SUB
                mem[3] = 32'b00011001010000001010000000000000; // CAM // ************  Finish
the R-Type ********************/
                mem[4] = 32'b00000001110000000000000000100100; // ANDI
                mem[5] = 32'b00001001110000000000000000100100; // ADDI // ************  Finish
the I-Type (ALU_type) ********************/
                mem[6] = 32'b00010001110000000000000000110100; // LW
                mem[7] = 32'b00011001110000000000000000110100; // SW // ************  Finish
the I-Type (Mem_type) ********************/
```

```
                mem[8] = 32'b00100001110011100000000010000100; // BEQ // ************  Finish the I-Type (Br_type) ********************/

                mem[9] = 32'b00000000000000000000000001000010; // J // immd = 8

                mem[10]= 32'b00001000000000000000011111101000010; // JAL / ************  Finish the J-Type  ********************/

                mem[11]= 32'b00000001110010100000000010000110; // SLL by 1 shift amount.

                mem[12]= 32'b00001001110010100000000010000110; // SLR by 1 shift amount.

                mem[13]= 32'b00010001110010100010000000000110; // SLLV by 2 shift amount.

                mem[14]= 32'b00011001110010100010000000000110; // SLRV by 2 shift amount./ ************  Finish the S-Type  ********************/

        end


        reg fetchValidReg;


        // Register for updating the next PC value
        always @(posedge clk) begin
                if (reset) begin
                        fetchValidReg <= 1'b0;
                        nextpc <= 32'd0;
                end
                else begin
                        fetchValidReg <= 1'b1;
                        case (PCSrc)
                                2'b00: nextpc <= nextpc + 2'd1; // PC = PC + 4
                                2'b01: nextpc <= nextpc + BTA; // PC = PC + Jumb_address
                                2'b10: nextpc <= JA;  // PC = imm17 + PC
                                2'b11: nextpc <= stackOut; // PC = stackOut
                                default: nextpc <= nextpc + 2'd1; // Default case for unknown PCSrc value
                        endcase
                end
```

```
        end

        // Register for updating the instruction and valid signal
        always @(posedge clk) begin
                if (reset) begin
                        valid <= 1'b0;
                        instruction <= 32'd0;
                end
                else begin
                        valid <= fetchValidReg;
                        instruction <= mem[nextpc];
                end
        end

endmodule
```

Appendix B: Instruction Decode (ID):

```
module decode(
        input clk,
        input [31:0] Instruction,
        input [31:0] pcNext,                    // value of the next pc
        output reg [4:0] Rs1,
        output reg [4:0] Rs2,
        output reg [4:0] Rd,
        output reg [13:0] I_Immediate,
        output reg [23:0] J_Immediate,
        output reg [4:0] Func,
        output reg [1:0] Type,
        output reg Stop,
        output reg [8:0] R_unused,
```

```
        output reg [3:0] S_unused,
        output reg [4:0] SA,
        output reg [31:0] data_in // this output is the input for the control stack
);

        reg [1:0] TypeBits;
        reg [4:0] Jfunc;

        // if there is an extend then calculate the jumb address
        //always @(posedge clk) begin
                //if( ExtOp == 1)begin
                        //imm32 = {{9{J_Immediate[23]}}, J_Immediate};
                        //pc_JA <= imm32;
                //end
        //end

        always @(posedge clk) begin
                TypeBits = Instruction[2:1]; // save type bits
                Jfunc    = Instruction[31:27];

                case (TypeBits)
                   /* when R-type */
                        2'b00: begin
                        Stop     <= Instruction[0];
                        Type     <= Instruction[2:1];
                        R_unused <= Instruction[11:3];
                        Rs2      <= Instruction[16:12];
                        Rd       <= Instruction[21:17];
                        Rs1      <= Instruction[26:22];
                        Func     <= Instruction[31:27];
```

```
SA          <= 5'b00000;
end
/* when J-type */
2'b01: begin
Func     <= Instruction[31:27];
// when there is a JAL instruction
if ( Jfunc == 5'b00001) begin
            data_in <= pcNext;  // Stack.Push (PC + 4)
end
Stop      <= Instruction[0];
Type                <= Instruction[2:1];
J_Immediate <= Instruction[26:3];

end
/* when I-type */
2'b10: begin
Stop      <= Instruction[0];
Type                <= Instruction[2:1];
I_Immediate <= Instruction[16:3];
Rd        <= Instruction[21:17];
Rs1                 <= Instruction[26:22];
Func      <= Instruction[31:27];
end
/* when S-type */
2'b11: begin
Stop     <= Instruction[0];
Type      <= Instruction[2:1];
S_unused <= Instruction[6:3];
SA       <= Instruction[11:7];
Rs2      <= Instruction[16:12];
```

```
                Rd      <= Instruction[21:17];
                Rs1     <= Instruction[26:22];
                Func    <= Instruction[31:27];
                end
        endcase
    end
endmodule
```

Appendix C: Extender:

```
module Extender(
    input Extop,
    input JorI,
    input [31:0] nextpc,
    input [13:0]immd14,
    input [23:0]immd24,
    output reg [31:0] immd32
);



always @(Extop, JorI) begin
if (Extop ==0 && JorI == 1) begin   // I - logic Types
    immd32 <= {{18{immd14[13]}}, immd14};
end

if (Extop == 1 && JorI == 1) begin   // I - non logic Types
    immd32 <= {{18{immd14[13]}}, immd14};
end
if (Extop == 1 && JorI == 0) begin   // J-Types
    immd32 <= nextpc +{{9{immd24[23]}}, immd24};
```

```
end
end
endmodule
```

Appendix D: Register File (RF):

```
module RegisterFile(
        input clk,                  // clock bit
        input [4:0] RA,                         // the first address register
        input [4:0] RB,                         // the second address register
        input [4:0] RW,                         // the distation register address when writing mode
        input WriteEnable,              // enable write bit, will be 1 when we want to write on reg (CS)
        input secOperant,         // 0: when second operant is Rs2, 1: second operant is Rd (CS)
        input [31:0] BusW,                      // the data that we want to save on the destination
register
        output reg [31:0] BusA,         // Rs content -> [Rs]
        output reg [31:0] BusB          // Rt content -> [Rt]

);

//----------------------------------------------------------------
//>>>>ntialzing the array of the register, the array has 32 registers
//each register is 32-bit size
reg [31:0] registers [31:0]; // intializing the array of the registers

initial begin
   integer i; // counter
  // loop fir assigning values to each register
 for (i = 0; i < 32; i = i + 1) begin
                registers[i] = i;
    end
 end
```

```
// checking if the its the positive edge clock
// if true, then we check if the write enable is equals to 1
// it true, then we  write the value of the >>BusW>> into the address of the destination register
always @(posedge clk) begin
   if (WriteEnable) begin
                    // assigning the BusW value to the dist register
                    registers[RW] <= BusW;
   end

   // the output is the content value of the Register A
   BusA <= registers[RA];
   // according to the instrction we chose the the second operant
   if ( secOperant == 1 )
                    begin
                    BusB <= registers[RW]; // chose Rd
                    end
          else
                    begin
                    BusB <= registers[RB]; // chose Rs2
                    end
 end
endmodule
```

Appendix E: Register Buffer:

```
module register(
        input [31:0] inreg,
        output reg [31:0] outreg
);
        always @(*)
        begin
                outreg <= inreg;
        end
endmodule
```

Appendix F: Memory (M):

```
module DataMemory(
  input clk,          // input clock
  input [31:0] address, // the address of the data that we want to read (when the function is load)
  input [31:0] data_in, // the input data that we want to store (when the function is store)
  input [1:0] RW,
  output reg [31:0] result  // the resul is 32 - bit (when reading the data from memory
  );
//-----------------------------------------------------------------
// >>>>>>Creating the memory and assigning intial values to the elemets of the memory
reg [31:0] memory [0:255]; // Assuming a memory size of is 2^8
//declare a memory array in Verilog with 256 elements, each of size 32 bits.
reg [31:0] lfsr;
initial begin
        integer i;
        lfsr = 32'hACE1B5ED;       // assigning the "ACE1B5ED" to the lfsr
        // this will be used to assign intial values to each index of the memory
        // Loop through each element of the array
        for (i = 0; i < 256; i = i + 1) begin
```

```
                memory[i] = lfsr[31:0];
                lfsr = {lfsr[0]^lfsr[1]^lfsr[3]^lfsr[4], lfsr[31:1]};
        end
end
// 10 01 00 11
// >>>>>> Looping through the memory, and apply operations read/write/nothing
// When RW == 1 -->STORING OPERATION / WRITING
// When RW == 0 -->LOADING OPERATION / READING
// When RW == 2 OR 3  --> Output result will be zero
always @(posedge clk) begin
        if (RW == 2'b01) begin
   // writing case
   memory[address] <= data_in;
   end
   else if (RW == 2'b10) begin
   // reading case
   result <= memory[address];
   end
   else if (RW == 2'b00 | RW == 2'b11) begin
        // when there is no read or write on the data memory block
        // we adssign the result to the zero value
   result <= 32'h0000_0000;
        end
 end

endmodule
```

Appendix G: Write Back (WB):

```
module wrtie_back(
        input clk,
        input [31:0] ALU_out,
        input [31:0] Data_out,
        input WBdata,  // 0: for alu, 1: for lw, x: for others since RegWrite = 0
        output reg[31:0] Rd_val

);
        always @(posedge clk)
                begin
                // Check the condition flags
                        if (WBdata == 1)
                                begin // write back Data_out
                                Rd_val <= Data_out;
                           end
                        else
                                begin // always write back  ALU_out
                                Rd_val <= ALU_out;
                           end
                end
endmodule
```

Appendix H: Control Stack:

```
module ControlStack(
                                                    // we firstly have the follwoing
inputs
  input clock,                  // Clock signal
  input [1:0] pushOrPop,        // Control signal for push or pop operation
                                                    // the data is the address that we want to
save in the stack at address SP in the control stack.
  input [31:0] data_in,         // Data to be pushed onto the stack
                                                    // this when we finish calling the
function, or the routine function, and we want to return to our address
                                                    // we make the pop operation which is
reading the control stack
  output reg [31:0] data_out        // Data popped from the stack
);
                                                    // intializing the stack with 16
elements, each element is 32-bits
                                                    // these elements are the
addresses (PC+4) that we want to save in the stack
                                                    // the objective is to take these
addresses when finish the function call or the ISA.
  reg [31:0] stack [0:15];          // Stack memory with 16 elements
                                                    // the sp is 4 bits since we have
only 16 locations to fill in the control stack.
  reg [3:0] sp;                     // Stack pointer


  always @(posedge clock) begin
                                                    // the pushOrPop input
determine the operation that we want to do
                                                    // when pushOrPop == 2'b10' ->
push (store) the input 32-bit address
    case (pushOrPop)
      2'b10: begin                  // Push operation when pushOrPop is 10
```

```
                                                                    // we just insure that we have
a valid location in the stack control
        if (sp < 4'b1111) begin
                                                                    // we put the input vLue
on the top of the stack-> stack[sp+1]
                                                                    // we apdate the pointer
of the stack to point on the latest input address stored in the stack memory
          stack[sp + 1] <= data_in;    // Push data onto the stack
          sp <= sp + 1;               // Increment stack pointer


        end
      end
      2'b01: begin                    // Pop operation when pushOrPop is 01
                                                                    // when the input
pushOrPop == 2'b01'-> pop operation, reading the top address on the memory
                                                                    // and then apdating the
value of the pointer -> sp -1
        if (sp > 4'b0000) begin
          data_out <= stack[sp];       // Pop data from the stack
          sp <= sp - 1;               // Decrement stack pointer
        end
      end
    endcase
  end
endmodule
```

Appendix J: Control Unit (CU):

```
module ControlUnit(
        input reset,
        input clk,
        input [4:0]Func,
        input [1:0] Type,
        input Stop_bit,
        output reg [1:0] PCSrc,          // 0 -> PC = PC +3   // 1 -> PC = RS   // 2 -> PC = imm24 + PC
        output reg  SecOPr,              // 0->rs2, 1->rd
        output reg RegWrite,             // 0 -> Yes, 1 -> No`
        output reg ExtOp,                //0-> Zero extend    //1 -> Singed extend
        output reg IorJ,                 //0-> choose I      //1 -> choose J
        output reg [1:0] ALUSrc,         // TYPE
        output reg [4:0] ALUOp,          // func
        output reg [1:0] RW,             // 00 -> NO READ NO WRITE  // 01 -> READ  // 10 ->
WRITE
        output reg [1:0] PushorPop,     // 00 -> No push or pop, 10 -> push, 01 pop
        output reg  WB                   // 0 -> ALU rd    // 1 -> memory
);

reg STATE_FETCH;
reg STATE_DECODE;
reg STATE_EXECUTE;
reg STATE_MEMORY_ACCESS;
reg STATE_WRITE_BACK;

reg state_reg;
initial begin

state_reg = STATE_FETCH;
end
```

```
reg next_state;
reg current_state;

// If reset is asserted, the control unit is reset and the state_reg is set to STATE_FETCH.
// Otherwise, it is updated with the value of next_state.
always @(posedge clk) begin
        if (reset) begin
        state_reg <= STATE_FETCH;
        end
        else
        begin
        state_reg <= next_state;
        end
end

// implements the state transitions of the control unit
always @(posedge clk) begin
        case (state_reg)
        STATE_FETCH:
        begin
        next_state = STATE_DECODE;
        end
        STATE_DECODE:
        begin
        next_state = STATE_EXECUTE;
        end
        STATE_EXECUTE:
        begin
        next_state = STATE_MEMORY_ACCESS;
```

```
        end
        STATE_MEMORY_ACCESS:
        begin
        next_state = STATE_WRITE_BACK;
        end
        endcase
end

always @(posedge clk) begin
if ((Type == 2'b00) && (Func != 5'b00011)) begin //all R-type except CAM
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b00; // pc = pc+4
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
                SecOPr                  <= 1'b0;  // RB = Rs2
                RegWrite        <= 1'b1;  // Write on Rd
                ALUSrc          <= Type;  // ALusrc = 00
                ALUOp           <= Func;  // ALUOp = 00000 | 00001 |00010
                RW              <= 2'b00; // no accessing the memory
                WB              <= 1'b0;  // Rd = ALU data
                end


else if ((Type == 2'b00) && (Func == 5'b00011)) begin //CAM
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
```

```
                PushorPop       <= 2'b01;  // no push or pop in the control stack
            end
            else begin
                PCSrc           <= 2'b00; // pc = pc+4
                PushorPop       <= 2'b00;  // no push or pop in the control stack
                end
            SecOPr              <= 1'b0;  // RB = Rs2
            RegWrite        <= 1'b0;  // no writeback
            ALUSrc          <= Type;  // ALusrc = 00
            ALUOp           <= Func;  // ALUOp = 00011
            RW              <= 2'b00; // no accessing the memory

            end

else if ((Type == 2'b10) && (Func == 5'b00000 || Func == 5'b00001 )) begin //I-TYPE (ANDI & ADDI)
            if ( Stop_bit == 1 ) begin
                PCSrc           <= 2'b11; // pc = pc+4
                PushorPop       <= 2'b01;  // no push or pop in the control stack
            end
            else begin
                PCSrc           <= 2'b00; // pc = pc+4
                PushorPop       <= 2'b00;  // no push or pop in the control stack
                end
            RegWrite        <= 1'b1;  // Write on Rd
            ALUSrc          <= Type;  // ALusrc = 10
            ALUOp           <= Func;  // ALUOp = 00000 | 00001
            ExtOp           <= 1'b0;  // do unsigned extend
            IorJ    <= 1'b1;  // choose the immd14
            RW              <= 2'b00; // no accessing the memory
            WB              <= 1'b0;  // Rd = ALU data
```

```
                end

else if ((Type == 2'b10) && (Func == 5'b00010)) begin //I-TYPE (LW)
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b00; // pc = pc+4
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
                RegWrite        <= 1'b1;  // writeback on rd
                ALUSrc          <= Type;  // ALusrc = 10
                ALUOp           <= Func;  // ALUOp = 00010
                ExtOp           <= 1'b1;  // do signed extend
                IorJ    <= 1'b1;  // choose the immd14
                RW              <= 2'b10; // READ from memory
                WB              <= 1'b1;  // Rd = data out form memory

                end

else if ((Type == 2'b10) && (Func == 5'b00011)) begin //I-TYPE (SW)
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b00; // pc = pc+4
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
```

```
                SecOPr              <= 1'b1;  // RB = Rd
                RegWrite        <= 1'b0;  // no writeback
                ALUSrc          <= Type;  // ALusrc = 10
                ALUOp           <= Func;  // ALUOp = 00011
                ExtOp           <= 1'b1;  // do signed extend
                IorJ    <= 1'b1;  // choose the immd14
                RW              <= 2'b01; // write to memory

                end

else if ((Type == 2'b10) && (Func == 5'b00100)) begin //I-TYPE (BEQ)
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b01; // pc = Rs1
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
                SecOPr              <= 1'b1;  // RB = Rd
                RegWrite        <= 1'b0;  // no writeback
                ALUSrc          <= Type;  // ALusrc = 10
                ALUOp           <= Func;  // ALUOp = 00100
                ExtOp           <= 1'b1;  // do signed extend
                IorJ    <= 1'b1;  // choose the immd14
                RW              <= 2'b00; // no accessing the memory
                end

else if ((Type == 2'b01)) begin //J-TYPE (general)
```

```
                if (Func == 5'b00001) begin
                        PushorPop <= 2'b10;  // push the address in the control stack
                        end
                        PCSrc           <= 2'b10; // pc = immd24 + pc
                        RegWrite        <= 1'b0;  // no writeback
                        ALUSrc          <= Type;  // ALusrc = 10
                        ALUOp           <= Func;  // ALUOp = 00000 | 00001
                        ExtOp           <= 1'b1;  // do signed extend
                        IorJ      <= 1'b0;  // choose the immd24
                        RW              <= 2'b00; // no accessing the memory
                end

else if ((Type == 2'b11)&& (Func == 5'b00000 ||Func == 5'b00001 )) begin //S-TYPE (SLL, SLR)
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b00; // pc = pc + 4
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
                RegWrite        <= 1'b1;  // writeback on rd
                ALUSrc          <= Type;  // ALusrc = 11
                ALUOp           <= Func;  // ALUOp = 00000 | 00001
                RW              <= 2'b00; // no accessing the memory
                WB              <= 1'b0;  // Rd = ALU data
                end


else if ((Type == 2'b11) && (Func == 5'b00010 ||Func == 5'b00011 )) begin //S-TYPE (SLLV, SLRV)
```

```
                if ( Stop_bit == 1 ) begin
                        PCSrc           <= 2'b11; // pc = pc+4
                        PushorPop       <= 2'b01;  // no push or pop in the control stack
                end
                else begin
                        PCSrc           <= 2'b00; // pc = pc + 4
                        PushorPop       <= 2'b00;  // no push or pop in the control stack
                        end
                SecOPr                  <= 1'b0;  // RB = Rs2
                RegWrite        <= 1'b1;  // writeback on rd
                ALUSrc          <= Type;  // ALusrc = 11
                ALUOp           <= Func;  // ALUOp = 00010 | 00011
                RW              <= 2'b00; // no accessing the memory
                WB              <= 1'b0;  // Rd = ALU data
                end
end

// Define state register and initial state
endmodule
```