Faculty of Engineering & Technology

Electrical & Computer Engineering Department

Project No. 2 ENCS4370

Computer Architecture

The design and verification of a simple RISC processor in Verilog

Supervised By

Dr. Aziz Qaroush, Dr.Ayman Hroub

Prepared By

Raghad Afghani 1192423

Layan Saed 1180534

Ameena Jadallah 1171018

Section: 2, 3

2.7.2023

Abstract

In this report, we explore the process of designing and testing a basic Multi-Cycle RISC processor using Verilog HDL. We begin by constructing the necessary main components and then move on to implementing them. The control signals for these components are derived and the Control Unit is tested by creating a state diagram and deriving expressions for each signal. Each instruction is analyzed and the processor is designed accordingly. Finally, we conduct testing for each instruction and the Final Data Path.

Table of Contents

Table of Figures

2. Design Specification

2.1 Introduction

This project involves the design of a basic multi-cycle RISC processor that meets the following specifications:
• The instruction size is 32 bits.
• Four instruction types (R-type, I-type, J-type, and S-type).
• 32-bit general-purpose registers: from R0 to R31.
• The program counter (PC) is a special-purpose register that is 32 bits in size.
• The ALU of the processor generates a signal known as the "zero" signal, which is activated when the output of the previous ALU operation is zero..
• The processor has separate memories for data and instructions.
• The processor includes a control stack that stores the return addresses.
• The processor also contains a special-purpose register known as the stack pointer (SP), which points to the topmost element of the control stack. The value stored in SP represents the address of the empty element at the top of the stack. To simplify the design, it is assumed that a separate on-chip memory is used for the stack, and the initial value of SP is zero.

2.2 Instruction Types (Format)

The ISA has four instruction formats: R-type, I-type, J-type, and S-type.

2.2.1 R-type formula

Function^5	Rs1^5	Rd^5	Rs2^5	Unused^9	Type^2	Stop^1

Where:

- Rs1: 5-bit field representing the first source register.
- Rd: 5-bit field representing the destination register.
- Rs2: 5-bit field representing the second source register.
- Function: 9-bit unused field.
- 9-bit unused field.

2.2.2 I-type formula

Function^5	Rs1^5	Rd^5	Immediate^{14}	Type^2	Stop^1

Where:

- Rs1: 5-bit field representing the first source register.
- Rd: 5-bit field representing the destination register.
- For logic instructions, the immediate is unsigned and uses 14 bits. For other instructions, the immediate is signed.

2.2.3 J-type formula

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

Where:

- Signed Immediate: The jump offset uses a 24-bit immediate that is signed.

2.2.4 S-type formula

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

Where:

- Rs1: 5-bit field representing the first source register.
- Rd: 5-bit field representing the destination register.
- Rs2: 5-bit field representing the second source register. If the shift amount is variable and calculated at runtime, it will be stored in this register.
- 5-bit SA: The constant shift amount.
- 4-bit new field.

2.3 Instructions Encoding

Table 1 displays a list of instructions that are supported by this particular instruction set, along with their corresponding meanings and function value .

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
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & Immediate^{14}	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + Immediate^{14}	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
J-Type Instructions			
10	J	PC = PC + Immediate^{24}	00000
11	JAL	PC = PC + Immediate^{24} Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	Reg(Rd) = Reg(Rs1) << SA^{5}	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^{5}	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

Table 1: Instructions Encoding

3. Components

Upon examining the set of instructions, we have determined the necessary components and successfully constructed them.

3.1 PC 32-bit Register

Figure 1 shows the PC 32-bit component that was built using Verilog HDL, which takes the PC address and outputs it when the PC on the bus is enabled.

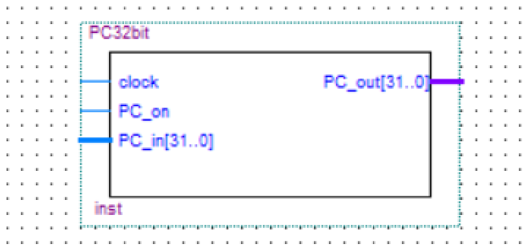


Figure 1: PC 32-Bit Register

3.2. Instruction Memory

Figure 2 shows the instruction memory component that was built using a Verilog HDL. It takes 32-bit PC address to give an instruction based on the pc address:

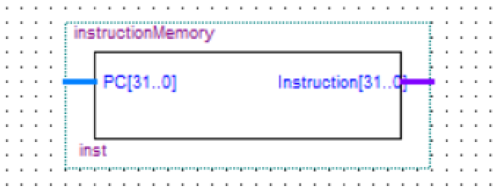


Figure 2: Instruction Memory

3.3. Data Memory

Figure 3 shows a part of a computer program that lets you read and write data to memory. This is useful for instructions that need to get data from memory, like LW, and for instructions that need to store data in memory, like SW. The WriteData bus is used to tell the computer what data to write to memory, while the MemData bus is used to load the data from memory to the register file.

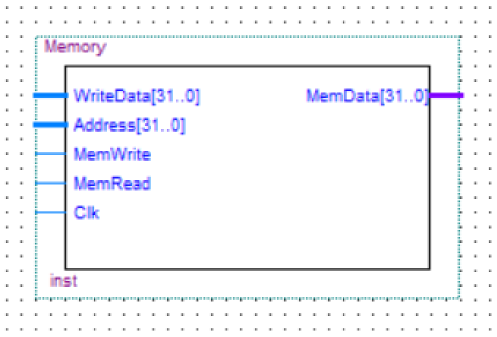


Figure 3: Data Memory

3.4. Register File

Figure 4 shows a part of a computer program that reads and writes data to registers. Registers are like small storage spaces inside the computer's processor. This component is used to get data from two registers, RA and RB, and send the output data to two different buses, data_1 and data_2. It can also write data to a register called RW using a bus called write_data. The RegWrite signal allows the program to write to the register when it needs to.

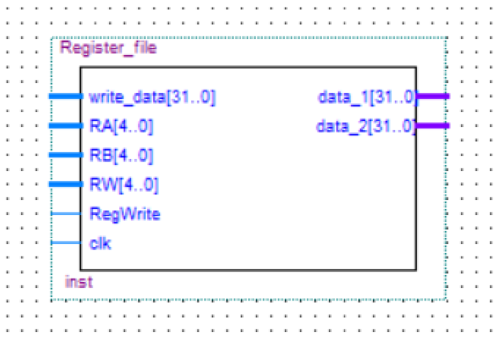


Figure 4: Register File

3.5. Extender

To enhance its functionality, the extender has been subdivided into three separate extenders. One of these extenders serves to extend a 5-bit immediate, while the other two are responsible for extending 24-bit and 14-bit immediate, respectively.
In Figure 5, an extender is depicted that takes a 24-bit immediate and the type of extension (signed or unsigned) as input. It then outputs a 32-bit extended immediate.

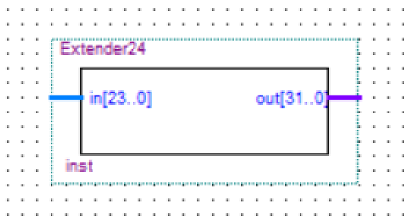


Figure 5: Extender 24 Bit

Figure 6 shows an extender that takes the 14-bit immediate to extend, and the extending type (signed or unsigned) and outputs a 32-bit extended immediate.

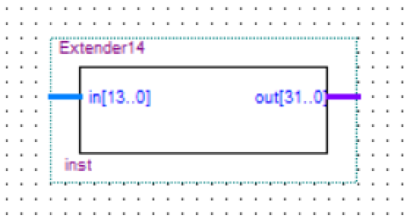


Figure 6: Extender 14 Bit

Figure 7 shows an extender that takes the 5-bit Shift Amount to extend, and the extending type (signed or unsigned) and outputs a 32-bit extended immediate.

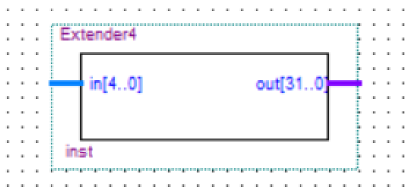


Figure 7: Extender 5 Bit

3.6. ALU

Figure 8 shows a part of a computer program called the ALU (Arithmetic Logic Unit). This component takes two pieces of data, does some math operations on them, and gives the result as output. It also sets a flag called the "zero flag" if the result is zero.
The computer program decides which math operation to do based on a signal called aluSel, which comes from another part of the program called aluControl. The result of the operation is sent to another part of the program called the "result bus."
The "Branch" signal is used to decide whether the program should go to a different part of the program or not, based on whether the "zero flag" is set or not.

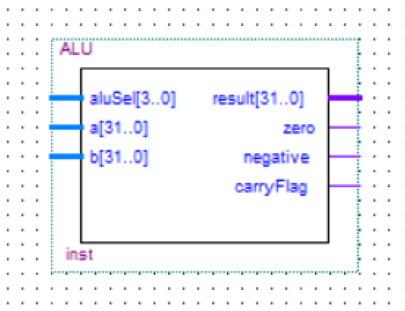


Figure 8: ALU

3.7. ALU Control

In Figure 9, we see an ALU control component constructed using Verilog HDL. This component takes two inputs, "function" and "type" and produces aluSel as output. The aluSel signal is responsible for detecting the operation that needs to be performed.

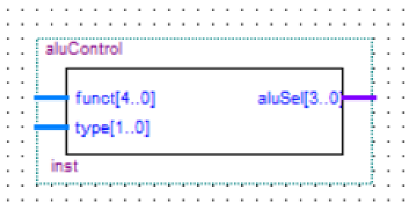


Figure 9: ALU Control

3.8. Control

Figure 10 shows a part of a computer program called the control component that is built using Verilog HDL. The control unit interprets instructions and generates control signals to coordinate the operation of the processor's components, ensuring proper execution of the program. It controls the flow of data between the memory, ALU, and registers, enabling instruction fetching, decoding, execution, and memory operations.

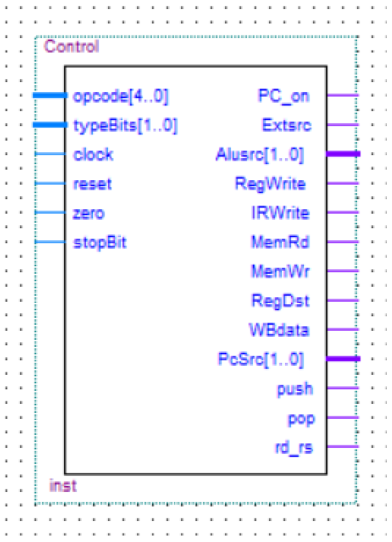


Figure 10: Control

– Other components were used in the built datapath like instruction_register, A_register, B_register, and ALUOut_register, which helps to get a balanced clock cycle length and save any results needed for the remaining cycles.

3.9. Stack pointer (SP)

The unit implements a stack using Verilog HDL with input signals such as "clock," "reset," "push," "pop," and "data_in," and an output called "data_out." The module contains an 8-bit memory array and an 8-bit stack pointer. The "always" block either performs a push or pop operation based on the inputs, and the output data_out reflects the most recently stored data in the stack.

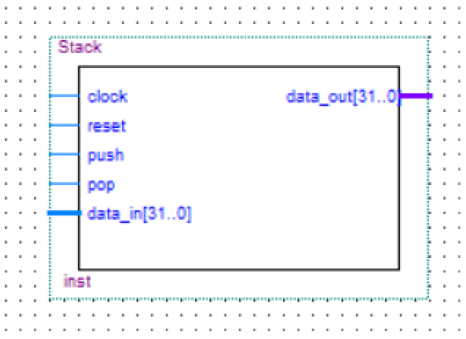


Figure 11: Stack Pointer

3.10. Load Multiplexer

This component implements a multiplexer (mux) that selects between two 5-bit input signals,a(Rs2) and b(Rd), based on the value of the sel(rd_rs) input signal which is detect by the branch result [if (Reg(Rs1) == Reg(Rd))], and the stores value of a register into a memory address obtained by adding the value of another register and an immediate value.[SW Mem(Reg(Rs1) + Imm14) = Reg(Rd)]. If "sel" is 1, the output signal "y" is set to the value of b. Otherwise, if sel is 0, the output signal "y" is set to the value of "a".

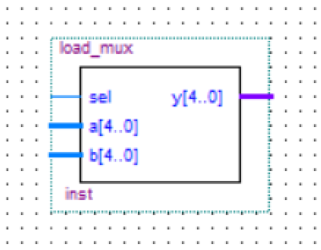


Figure 12: Load Multiplexer

3.9. Control Unit

3.9.1 Truth Table

After building the data path, the control signals were derived as shown in Table

ins/sig	PcSrc	ExtSrc	AluSrc	RegWrite	WBdata	MemR	MemW	Rd_Rs	Push
AND	10	X	00	1	1	0	0	0	0
ADD	10	X	00	1	1	0	0	0	0
SUB	10	X	00	1	1	0	0	0	0
CMP	10	X	00	0	x	0	0	0	0
ANDI	10	1	01	1	1	0	0	0	0
ADDI	10	1	01	1	1	0	0	0	0
LW	10	1	01	1	0	1	0	0	0
SW	10	1	01	0	x	0	1	1	0
BEQ	00	1	01	0	x	0	0	1	0
J	01	0	X	0	x	0	0	0	0
JAL	01	0	X	0	x	0	0	0	1
SLL	10	X	11	1	1	0	0	0	0

SLR	10	X	11	1	1	0	0	0	0
SLLV	10	X	00	1	1	0	0	0	0
SLRV	10	X	00	1	1	0	0	0	0

3.9.2 Boolean Expression

- if (stopBit) $\rightarrow$ PcSrc = 11
  else if (BEQ) $\rightarrow$ PcSrc = 00
  else if (J + JAL) $\rightarrow$ PcSrc = 01
  else $\rightarrow$ PcSrc = 10
- if (SLL + SLR) AluSrc = 10
  else if ($LW + SW + ADDI + ANDI$ + BEQ) $\rightarrow$ AluSrc = 01
  else if (CMP + $ADD + AND$ + SUB) $\rightarrow$ AluSrc = 00
- ExtSrc = ($\overline{J} + \overline{JAL}$)
- RegWrite = ($\overline{CMP} + \overline{BEQ} + \overline{SW} + \overline{JAL} + \overline{J}$)
- WBdata = $\overline{LW}$
- MemR = LW
- MemW = SW
- Rd_Rs = (SW + BEQ)
- Push = JAL
- Pop = StopBit

3.10 Implementing Final Data Path

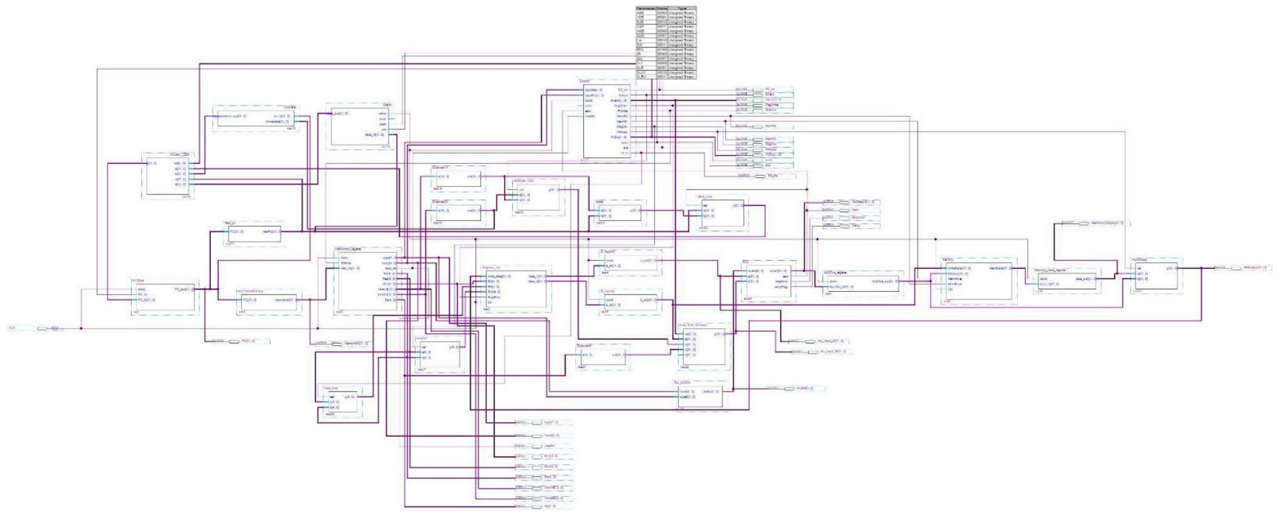


Figure 13: Data Path

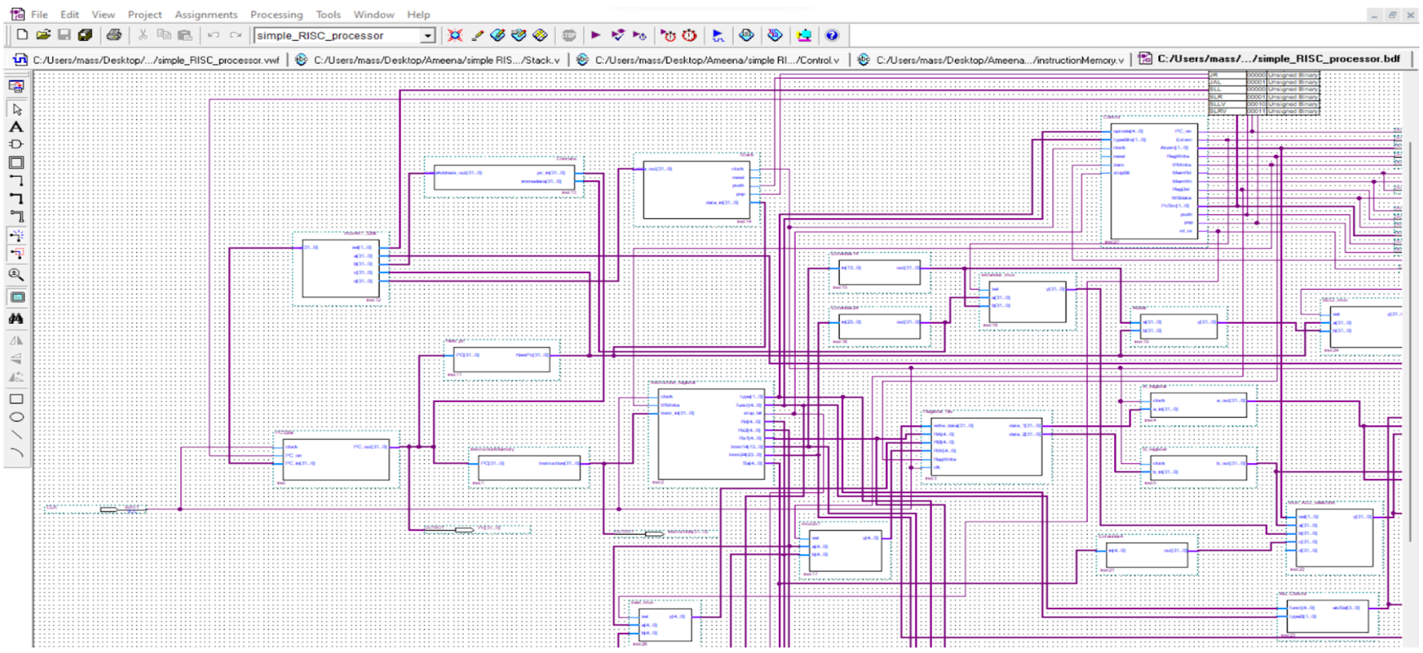

File
Edit
View
Project
Assignments
Processing
Tools
Window
Help
simple_RISC_processor
C:/Users/mass/Desktop/.../simple_RISC_processor.vwf
C:/Users/mass/Desktop/Ameena/simple RIS.../Stack.v
C:/Users/mass/Desktop/Ameena/simple RI.../Control.v
C:/Users/mass/Desktop/Ameena.../instructionMemory.v
C:/Users/mass/.../simple_RISC_processor.bdf

Figure 14: Data Path

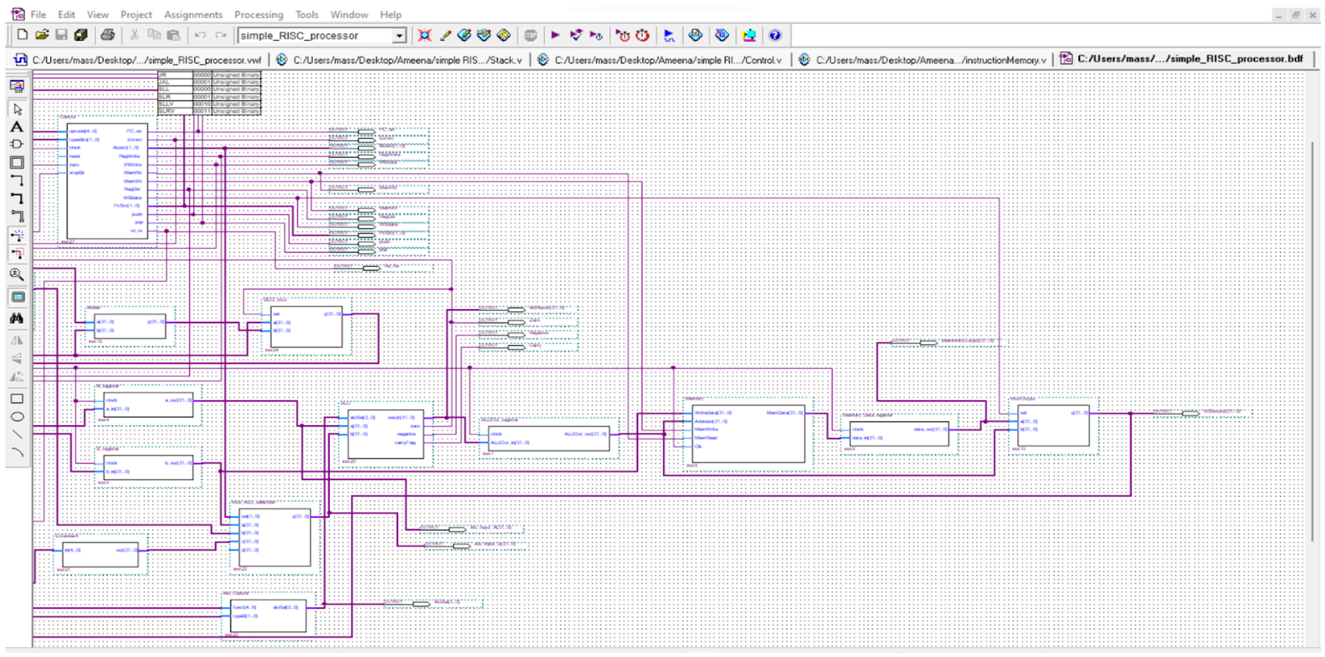


Figure 15: Data Path

- Here is the file that contains a high-quality picture of the datapath: [datapath - Google Drive]

The data path shown in the figure is made up of five different stages:

3.10.1 Instruction Fetch

The instruction is obtained by fetching it from the instruction memory using the Program Counter (PC) address. The PC address can be updated to four different values depending on the instruction type:

1) PC = Branch target address.
2) PC = Jump address.
3) PC = PC+1.

3.10.2 Instruction Decode

During this stage, the instruction is decoded into its respective components, including Function, Type, Rs, Rt, Rd, Immediate14, and Immediate24. Following this, the registers are read from the Register File based on the control signals. Once the registers and Immediate values are obtained, they are passed on to the Execution stage, concluding this stage of the program.

3.10.3 Execute

During this stage of the program, the ALU unit performs any necessary calculations or operations required by the instruction. Once the operations are completed, the resulting values are passed on to the Memory stage.

3.10.4 Memory

Based on the control signals obtained from the previous stages, the memory can either perform a read operation, a write operation, or no operation at all.

3.10.5 Write Back

The write back stage is responsible for writing data back to the Register File. The data to be written back can originate from the ALU, Memory, or an Immediate.

4. Simulation and Testing

In this section, we present the testbench comprising individual tests for every required command and operation in this project. Alongside these tests, we offer screenshots of the simulation that capture each vital operation. Furthermore, we thoroughly analyze the resulting outcomes to gain insights and understanding.

```
0:  Instruction = 32'b00001_00001_00001_00000000000011_10_0; // ADDI R1,R1, 3
1:  Instruction = 32'b00001_00010_00010_00000000000010_10_0; // ADDI R2,R2, 2
2:  Instruction = 32'b00001_00010_00100_00001_000000000_00_0; // ADD R4,R2,R1
3:  Instruction = 32'b00000_00100_00100_00000_00001_0000_11_0; // SLL R4,R4,1
4:  Instruction = 32'b00001_00100_00100_00000_00001_0000_11_0; // SLR R4,R4,1
5:  Instruction = 32'b00010_00100_00100_00010_00001_0000_11_0; // SLLV R4,R4,R2
6:  Instruction = 32'b00001_00101_00101_00000000000011_10_0; // ADDI R5,R5, 3
7:  Instruction = 32'b00011_00001_00000_00101_00000_0000_00_0; // CMP R1,R5
8:  Instruction = 32'b00011_00001_00101_00000000000011_10_0; // SW R5,imm(R1)
9:  Instruction = 32'b00010_00001_00110_00000000000011_10_0; // LW R6,imm(R1)
10: Instruction = 32'b00000_00110_00110_00000000000001_10_0; // ANDI R6,R6, 1
11: Instruction = 32'b00000_000000000000000000001101_01_0; // J pc = 13
12: Instruction = 32'b00000_000000000000000000000001_01_0; // J pc = 1
13: Instruction = 32'b00001_00001_00001_00000000000111_10_0; // ADDI R1,R1, 7
14: Instruction = 32'b00100_00001_00001_00000000000001_10_0; // BEQ R1,R1,16
15: Instruction = 32'b00000_000000000000000000000001_01_0; // J pc = 1 ,
16: Instruction = 32'b00010_00001_00001_00100_000000000_00_0; //SUB R1, R1, R4
17: Instruction = 32'b00001_000000000000000000010100_01_0; // JAL PC => 20
18: Instruction = 32'b00000_00010_00100_00010_000000000_00_0; // AND R4,R2,R2
19: Instruction = 32'b00011_00100_00100_00010_00001_0000_11_0; //SLRV R4,R4,R2
20: Instruction = 32'b00001_00000_00010_00000000000010_10_1; // ADDI R2,R0, 2
```

4.1 ADDI simulation analysis

Referring to Figure (16), we observe that the instruction "ADDI R1, R1, 3" positioned at "pc = 0" requires approximately four cycles to complete. During the execution, the instruction performs an Add operation in the Arithmetic Logic Unit (ALU), with the second ALU input being equal to the extended 14-bit immediate value. Notably, the execution stage concludes by the third cycle, and the resulting value from the ALU “3” is written back into the register file during the fourth cycle.

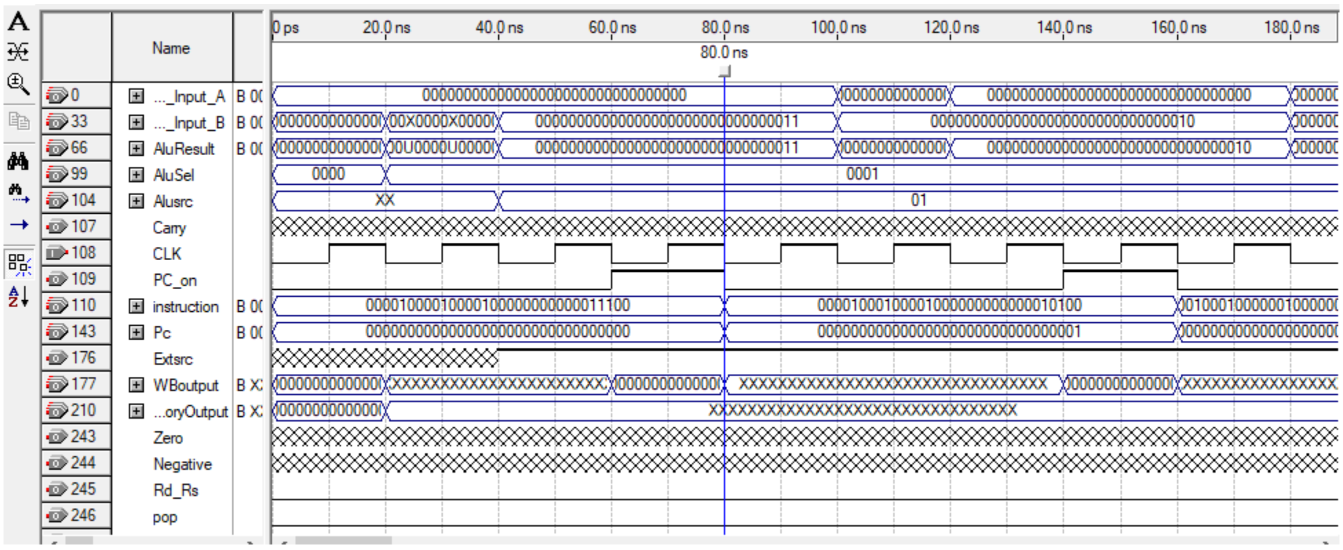


Figure 16: Add Immediate instruction simulation “I-type”

4.2 ADD simulation analysis

Referring to Figure (17), we observe that the instruction "ADD R4, R2, R1" positioned at "pc = 2" requires approximately four cycles to complete. During the execution, the instruction performs an Add operation in the Arithmetic Logic Unit (ALU), with the second ALU input being equal to the value of the “R1” register from the register file. Notably, the execution stage concludes by the third cycle, and the resulting value from the ALU “5” is written back into the register file during the fourth cycle into the “R4” register.

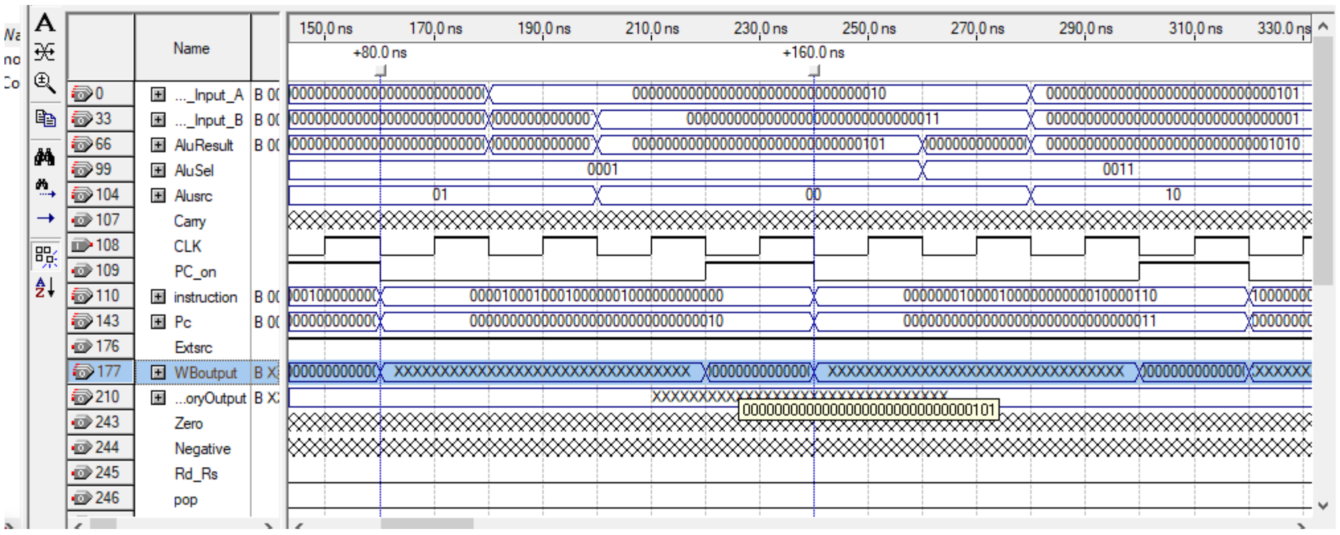


Figure 17: Add instruction simulation “R-type”

4.3 Shift-Left simulation analysis

Referring to Figure (18), we observe that the instruction "SLL R4, R4, 1" positioned at "pc = 3" requires approximately four cycles to complete. During the execution, the instruction performs a shift-left operation in the Arithmetic Logic Unit (ALU), with the second ALU input being equal to the extended 14-bit immediate value. Notably, the execution stage concludes by the third cycle, and the resulting value from the ALU is written back into the register file during the fourth cycle into the “R4” register, the ALU result is “32’b01010” which is the result of shifting “32’b00101” by 1 bit to the left.

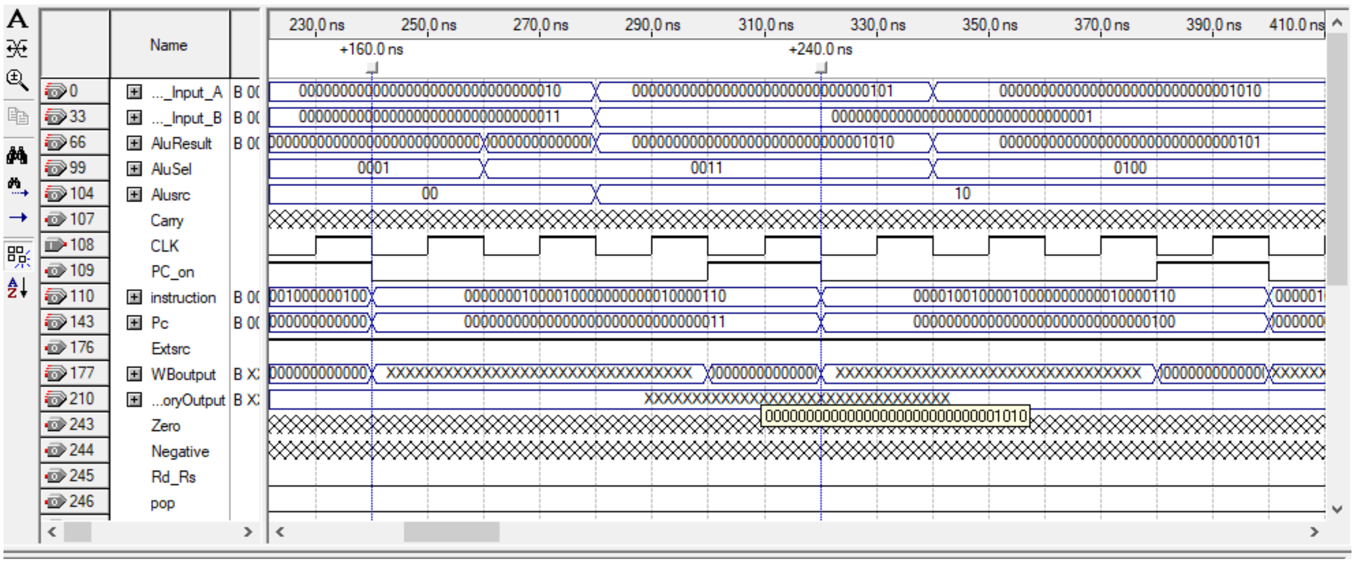


Figure 18: Shift Left instruction simulation “S-type”

4.4 CMP simulation analysis

Referring to Figure (19), we observe that the instruction "CMP R1, R5" positioned at "pc = 7" requires approximately three cycles to complete. During the execution, the instruction performs a subtraction operation in the Arithmetic Logic Unit (ALU). Notably, the execution stage concludes by the third cycle, Although the resulting value from the ALU is disregarded, it is important to note that the Zero and Carry flags are both set to 1, while the Negative flag is set to 0, indicating a non-negative value.

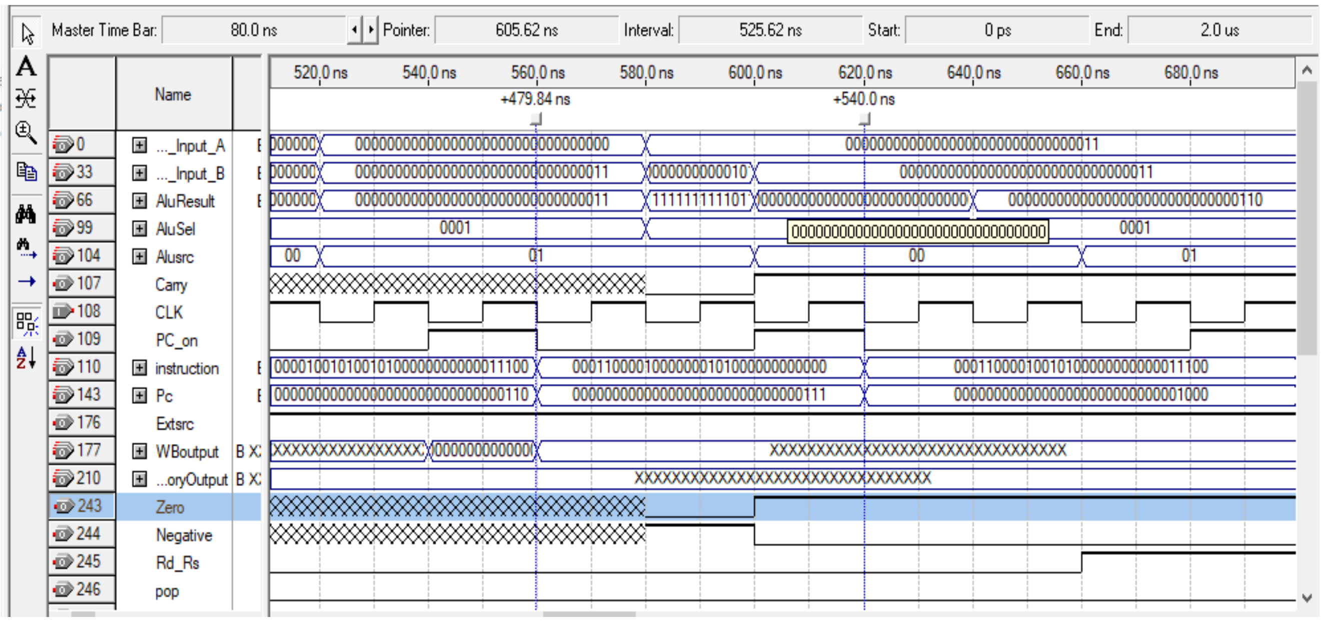


Figure 19: CMP instruction simulation "R-type"

4.5 SW simulation analysis

Referring to Figure (20), we observe that the instruction "SW R5, imm(R1)" positioned at "pc = 8" requires approximately four cycles to complete. During the execution, the instruction performs an Add operation in the Arithmetic Logic Unit (ALU), with the second ALU input being equal to the extended 14-bit immediate value. Notably, the execution stage concludes by the third cycle, and the resulting value from the ALU "32'b0110" is the calculated address that will be sent to the memory. With both the MemWr signal and the rd_rs flag being activated, the rd_rs signal serves as input for the multiplexer, determining that Rd is chosen to be transmitted into the register file as the second input. Subsequently, the resulting value from the register is written into the memory, specifically at the calculated address, ensuring its completion by the end of the fourth cycle.

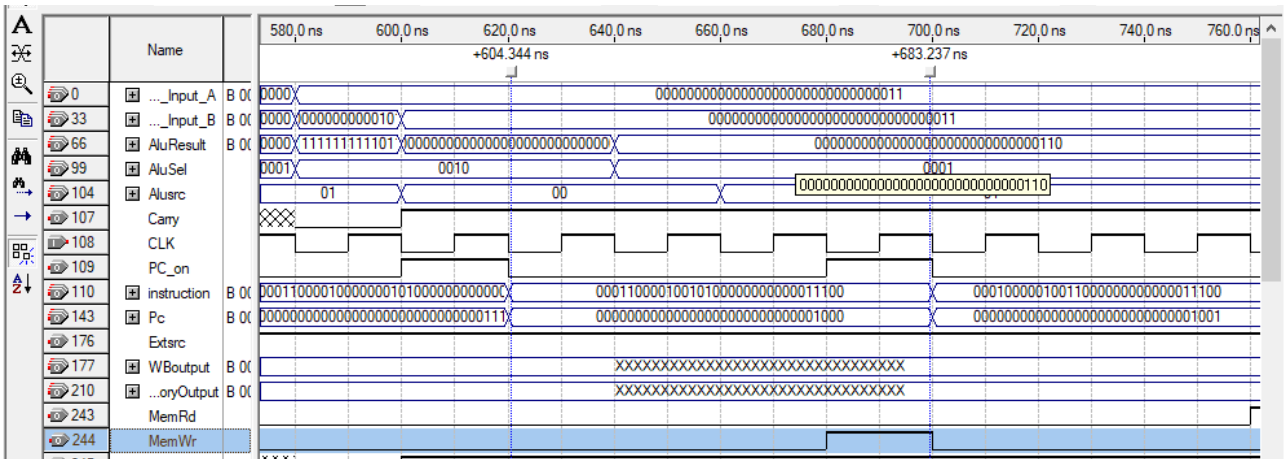
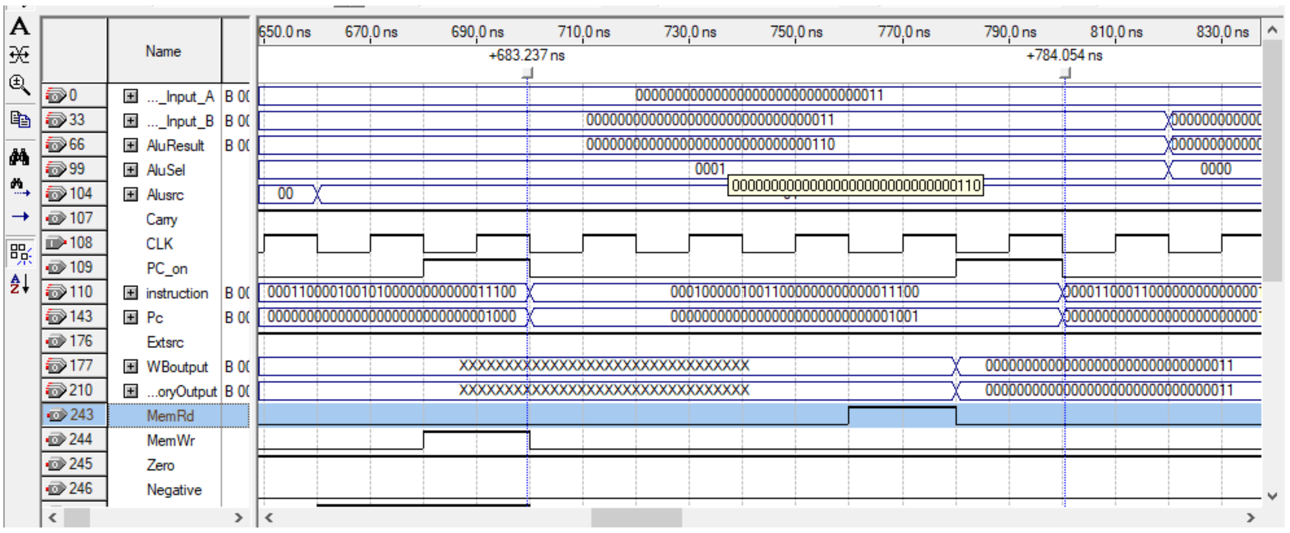


Figure 20: SW instruction simulation "I-type"

4.6 LW simulation analysis

Referring to Figure (21), we observe that the instruction "LW R6,imm(R1)" positioned at "pc = 9" requires approximately five cycles to complete. During the execution, the instruction performs an Add operation in the Arithmetic Logic Unit (ALU), with the second ALU input being equal to the extended 14-bit immediate value. Notably, the execution stage concludes by the third cycle, and the resulting value from the ALU "32'b0110" is the calculated address that will be sent to the memory. The MemRd flag is set and by the end of the fourth cycle the data from the memory is ready, in the fifth cycle, the WBdata flag is set and the data is written back to the register file, notice the value in Memory_output or WBoutput by the end fifth cycle.



Name
650.0 ns
670.0 ns
690.0 ns
710.0 ns
730.0 ns
750.0 ns
770.0 ns
790.0 ns
810.0 ns
830.0 ns
+683.237 ns
+784.054 ns
0 ..._Input_A B 00 00000000000000000000000000000011
33 ..._Input_B B 00 00000000000000000000000000000011 000000000000
66 AluResult B 00 00000000000000000000000000000110 000000000000
99 AluSel 0001 0000
104 Alusrc 00
00000000000000000000000000000110
107 Carry
108 CLK
109 PC_on
110 instruction B 00 00011000010010100000000000011100 00010000010011000000000000011100 0000110001100000000000000
143 Pc B 00 00000000000000000000000000001000 00000000000000000000000000001001 0000000000000000000000000
176 Extsrc
177 WBoutput B 00 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 00000000000000000000000000000011
210 ...oryOutput B 00 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX 00000000000000000000000000000011
243 MemRd
244 MemWr
245 Zero
246 Negative

Figure 21: LW instruction simulation "I-type"

4.7 J simulation analysis

Referring to Figure (22), we observe that the instruction "J pc = 13" is positioned at "pc = 11". The pc_src signal is set to "2b'01", its responsible for selecting the program counter (pc) calculated by the "concate" block, which determines the jump address target. Upon analyzing the simulation, it becomes evident that the pc transitions from "11" to "13".

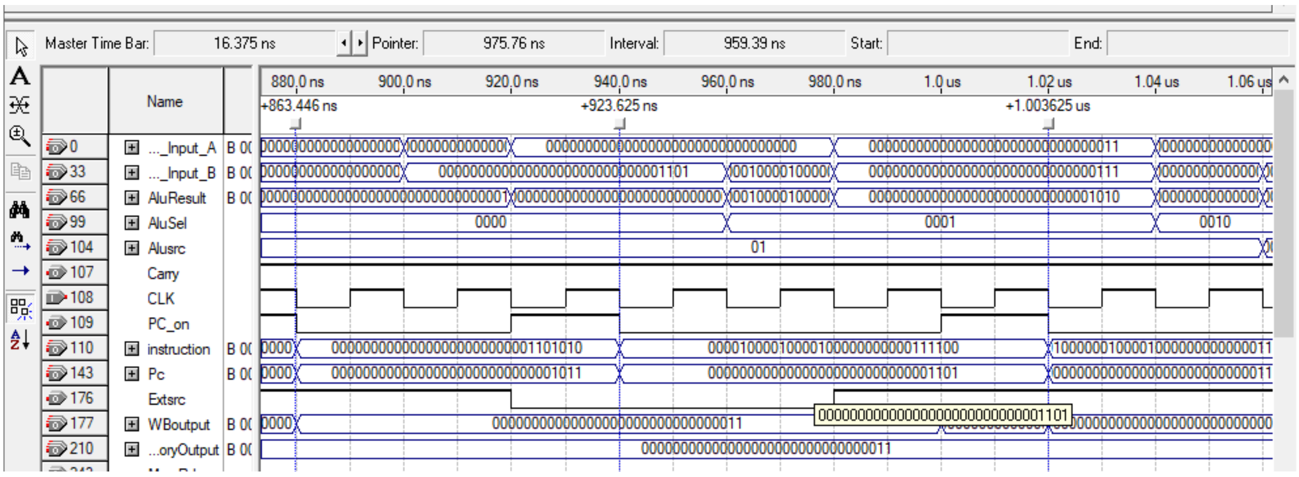


Figure 22: J instruction simulation "J-type"

4.8 BEQ simulation analysis

Referring to Figure (23), we observe that the instruction "BEQ R1, R1, (offset + new PC)" is positioned at "pc = 14" and requires approximately four cycles to complete. During the execution, the instruction performs a subtraction operation in the Arithmetic Logic Unit (ALU). Notably, the execution stage concludes by the third cycle, Although the resulting value from the ALU is disregarded, it is important to note that the Zero and Carry flags are both set to 1, indicating that the registers are equal, while the Negative flag is set to 0, indicating a non-negative value. The zero flag plays a crucial role in enabling the transmission of the newly calculated pc address to the pc value mux. Once the pc_src signal is set to "2b'10", it selects the necessary pc value. Upon examining the simulation, it is evident that the pc transitions from "14" to "16".

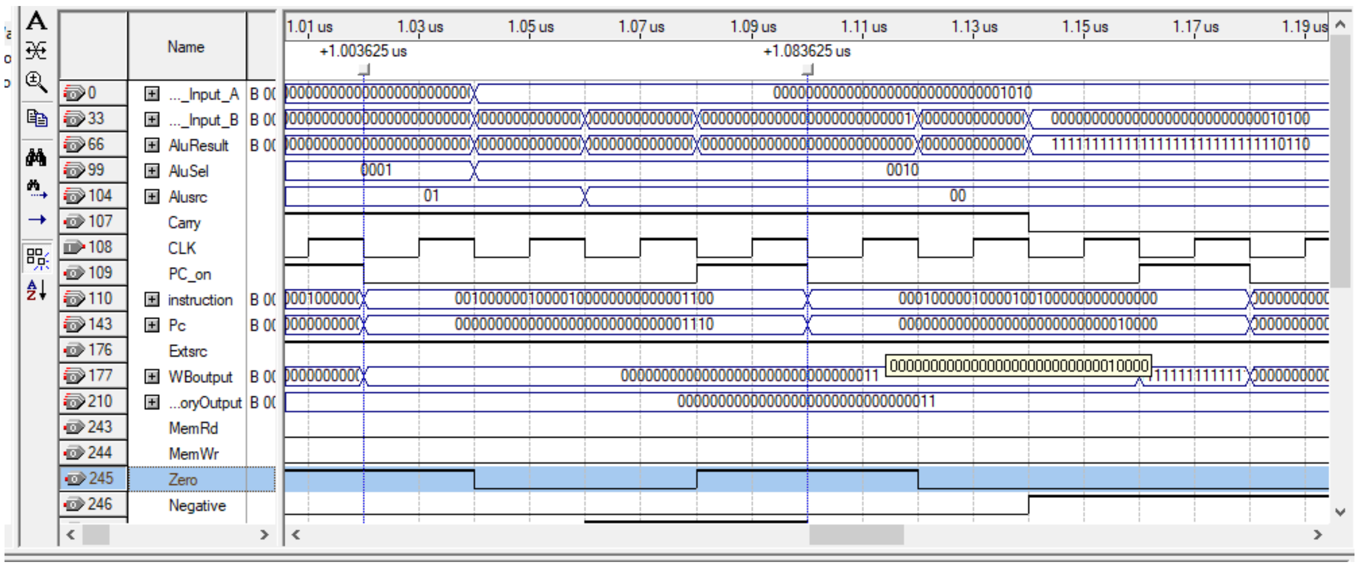


Figure 23: BEQ instruction simulation "R-type"

4.9 JAL simulation analysis

Referring to figure(24), we observe that the instruction "JAL PC => 20" is positioned at "pc = 17", The pc_src signal is set to "2b'01" is responsible for selecting the program counter (pc) calculated by the "concate" block, which determines the jump address target. Upon analyzing the simulation, it is evident that the pc transitions from "17" to "20", Furthermore, it is apparent that the push flag is set to one, indicating the instruction for the stack pointer to store the new pc value onto the stack.

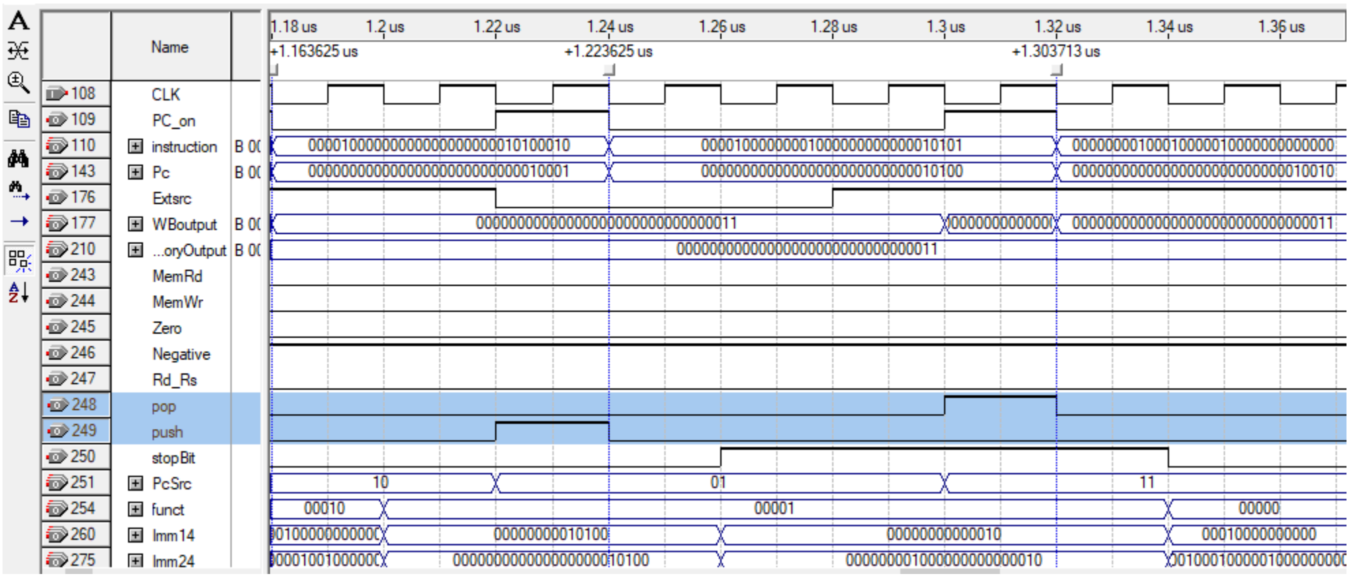


Figure 24: JAL instruction simulation "J-type" followed by an Instruction with a high stop bit

The instruction "ADDI R2, R0, 2" is found at "pc = 18" and is characterized by a high stop bit. Notably, the pop flag is set to one, signifying the instruction for the stack pointer to pop the stored pc value from the stack. The pc_src is set to 11, indicating to the multiplexer to choose to read the value from the stack pointer. After analyzing the simulation, it becomes evident that the pc transitions from "20" to "18".

5. TeamWork

From the initial stages of Design and Implementation, each member contributed their unique perspectives and skills, ensuring a well-rounded approach. We shared our ideas, brainstormed together, and seamlessly integrated our expertise to create a comprehensive datapath. During the Simulation and Testing phase, we divided responsibilities equitably, diligently executing simulations and conducting meticulous tests.

In a collaborative process, the testing of Instruction from 1 to 15 was efficiently managed by the team. Each team member took turns to test specific instructions, ensuring a fair distribution of responsibilities. Initially, one person tested instruction number 1, another tested instruction number 2, and a third team member tested instruction number 3. This rotation continued seamlessly as we progressed, with the first team member then testing instruction number 4, the second team member testing instruction number 5, and the last team member testing instruction number 6. This pattern continued until we reached instruction number 15, ensuring that each team member had an equal opportunity to contribute and participate actively in the testing process. By coordinating our efforts, we achieved comprehensive testing coverage and a shared understanding of the entire range of instructions.

By sharing our findings, analyzing results collectively, and adjusting strategies collaboratively, we demonstrated a true commitment to teamwork. We exemplified the essence of collaboration, fostering an environment where everyone's contributions were valued and honored, ultimately leading to outstanding outcomes.

6. Conclusion

In conclusion, the multi-cycle design approach plays a crucial role in creating an efficient processor. By breaking down the execution of instructions into multiple stages, each with its own specific task, we can achieve better utilization of hardware resources and improve overall performance.

Throughout the design process, accuracy and precision remain essential. Proper coordination between the different stages, such as instruction fetch, decode, execution, and memory access, is necessary to prevent conflicts and ensure seamless operation.

Thorough testing of each stage and its corresponding components is imperative to identify any potential issues and make necessary adjustments. This meticulous testing allows us to validate the functionality of the processor and guarantee its reliability.

The final testing phase, including a comprehensive evaluation of the Data Path, is the last crucial step. By subjecting the processor to various test cases and scenarios, we can assess its ability to handle different instructions and data types effectively.

Through diligent adherence to these practices, we can enhance the efficiency and performance of the processor while minimizing potential bottlenecks and conflicts. The multi-cycle design approach, combined with accurate testing methodologies, enables us to create processors that excel in executing complex instructions with speed and precision.