Department of Electrical and Computer Engineering, Gina Cody School of Engineering and Computer Science,Concordia University, Montreal, Canada

COEN 6741 – Computer Architecture and Design

Pipelined Mini MIPS Processor Design

Group (2):

Ragul Nivash Rangasamy Sekar	**40169564**
Thenmozhie Rajan	**40192527**

Instructor: Prof. Sofiène Tahar

April 12, 2023

Table of Contents

Table of Figures

Abstract:

The mini-MIPS (Microprocessor without Interlocked Pipeline Stages / Multiple Instructions Per Second) processor design is presented in this report. The aim of this project is to create a fully functional processor with a reduced instruction set architecture (ISA) using VHDL (VHSIC Hardware Description Language). The processor has five stages: instruction fetch, instruction decode, execution, memory access, and write-back. It is made to run a portion of the MIPS 32 instruction set architecture. The instructions executed in this processor are as follows, NORI, SRL, XOR, NANDI, SUBU, ADDI, BEQ, LH, SW, JR and J. The design includes data path and control unit for decoding, pipeline, and hazard control.

In this project we used ModelSim for compilation and simulation, precision RTL for synthesis. The simulation findings and synthesized design are included.

1 Introduction

Any electronic device that uses diverse functionality must have processors to control and operate it in accordance with the user's needs. The MIPS architecture is famous for being straightforward, effective, adaptable, and it is still a popular option for a variety of computing applications. MIPS is a reduced instruction set architecture (RISC), with a simple, streamlined instruction set that enables quick and effective processing. The MIPS architecture is made with great focus on performance. To minimize the latency for memory access, Load/store architecture is used, where memory access is only necessary when fetching load and store instructions [1].

The MIPS-based RISC processor is built on a pipeline layout that utilizes five separate hardware stages, such as Instruction fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). A group of registers referred to as pipeline registers separates these stages. Pipelining is used to boost overall throughput for instructions.

In this project, we are designing a mini-MIPS pipelined processor which is 32-bit architecture using VHDL. Mimi-MIPS has fixed length instruction with three instruction formats, such as, R, I and J-types to execute NORI, SRL, XOR, NANDI, SUBU, ADDI, BEQ, LH, SW, JR, and J instruction set. Each of instruction will be assigned to an instruction format.

In the pipelined processor, there are possibilities of hazards, such as structural, data and control hazard. Structural hazards are caused because of the use of the same hardware for different operations at the same time. Data hazards are caused when one instruction in a pipeline depends on the outcome of an earlier instruction that is currently running but not entirely executed. Control hazards happen in control path during wrong prediction on jump and branch instructions.

2 Instruction Set Architecture Design and Encoding

Instruction set architecture is the group of instruction used by the software to interface with the hardware. The ISA also specifies how instructions are represented in binary form, including their format, and encoding.

2.1 MIPS Instruction

A processor performs operations, including arithmetic and logical operations, memory, and control instructions.

Arithmetic and logical Instructions perform basic arithmetic and logical operations such as bitwise AND, OR, and NOT operations, as well as addition, subtraction, multiplication, and division. Usually, these actions are carried out on data that has been placed in memory or registers.

Memory instruction/load and store instruction allows the processor to read or write data from or to memory. Examples include instructions for loading and storing. Whereas store instructions transport data from registers into memory, load instructions are used to move data from memory into registers. Instruction sets such as LW, SW are some of the load/store instructions.

Control instructions control the program execution. Conditional and unconditional branch instructions are examples. When a specific condition is met, conditional branch instructions force to jump to a different position in memory, whereas unconditional branch instructions cause the program to go to a new location without condition. Instruction sets like, BEQ, JR, J are some of the control instructions.

2.2 Instruction Format

Instruction format specifies the arrangement of the instructions including size and location of operand fields. It is the structure and arrangement of a machine instruction's binary representation in a computer's instruction set architecture (ISA). The opcode in the MIPS instruction format is a fixed-length, 6-bit code. All the instructions are aligned and have a 32-bit instruction length (MIPS32).
There are three different instruction formats:

- R (Register)
- I (Immediate)
- J (Jump)

Instruction R-Type:

The instruction type (ALU instruction, ALU instant, load, store, branch, or jump) is specified by the opcode (6-bits). For arithmetic and logical operations on registers, R-format instructions are utilized. The three register fields that follow the opcode in the R-format each specify a register that is used as an operand for the instruction.

The rs and rt are source registers and rd is the destination register. Shamt is used for shift operations, it specifies the amount to be shifted. Func is used as control bits to differentiate various instructions. The R-format is predetermined in length and normally comprises 32 bits. The add instruction, which adds the contents of two registers and saves the outcome in a third register, is an illustration of an R-format instruction.

31 26	25 21	20 16	15 11	10 6	5 0
Opcode	rs	rt	rd	Shamt (h-Shift)	Func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction I-Type:

For arithmetic and logical operations that use a constant or a memory location, I-format instructions are used. Opcode specifies the operation to be performed. Rs (5-bits) is the source register. Rd (5-bits) is the destination register where the data should be stored. The immediate field (16-bit), which would be a constant or a memory location offset, is the second operand for performing operation. The load instruction, which transfers data from a memory address into a register, is an illustration of an I-format instruction.

31 26	25 21	20 16	15 0
Opcode	rs	rd	Immediate
6 bits	5 bits	5 bits	16 bits

Instruction J-Type:

Jump instructions are written in J-format. The destination address of the jump is specified in the 26-bit address field that follows the opcode in the J-format. Opcode (6-bit) represents the operation code to perform jump operation. Target (26-bit) is the destination address to execute different set of instruction.

31 26	25 0
Opcode	Target
6 bits	26 bits

2.3 MIPS Pipeline

A five-stage pipeline is used by the mini-MIPS processor to carry out instructions. Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write-Back (WB) are the five steps that make up the pipeline.

Instruction Fetch:
In the instruction fetch stage, the processor reads the instruction's address from the memory and then increases the PC by 4 to address the subsequent instruction. Instruction register is used to hold the instruction that will be required on upcoming clock cycles.

IR ← MEM [PC] ;
NextPC ← PC+4 ;

Instruction Decode:
In the decode stage, the processor decodes the instruction at this point by determining the opcode and operands from IF-ID register. Also, the processor examines the registers mentioned in the instruction before sending the operands to the following step.

A←Reg [rs] ;
B ← Reg [rt] ;
Immediate ←sign-extended immediate field of IR

Execution Stage:
With the help of the ID-EX register, it retrieves the instruction to carry out an ALU operation or memory address computation (for load, store, and branch) depending on the kind of instruction, or we compute a branch condition.

- Register-Register ALU instruction: ALUOutput ← A func B;
- Register-Immediate ALU instruction: ALUOutput ← A op Imm;
- Memory instruction: ALUOutput ← A + Imm;
- Branch instruction: ALUOutput ← NPC + Imm;
  Cond ← (A op 0);

Memory Stage:
The processor accesses the memory in this stage to read or write data. When a CPU receives a load instruction, it reads information data from memory. When a CPU receives a store instruction, it stores the data in memory. Except when a branch is taken, update the PC with the branch destination for all instructions with NPC.

- Memory reference: LoadMemData ← Mem[ALUOutput] or
  Mem[ALUOutput] ← B;
- Branch: if (Cond) PC ← ALUOutput
- The PC is updated for all instructions: PC ← NextPC;

Writeback Stage:
The processor now writes the instruction's outcome back to the register file. This stage is applicable for load operation only.

- Register-register or Register-immediate ALU instruction: Regs[rd] 🡐 ALUOutput.
- Load instruction: Regs[rd] 🡐 LoadMemData;

2.4 Encoding of MIPS instruction:

The encoding table use for the give 11 instructions are as follows,

Instruction	Operation between	Type	Opcode	Func	Example Instruction	Encoding
SRL	srl rd, rt, shamt	R	000000	000010	SRL R4,R0,#2	000000 00000 00000 00100 00010 000010
XOR	xor rd, rs, rt	R	000000	100110	XOR R5,R4,R6	000000 00100 00110 00101 00000 100110
SUBU	SUBU rd, rs, rt	R	000000	100011	SUBU R2,R5,R6	000000 00101 00110 00010 00000 100011
JR	jr rs	R	000000	001000	JR R31	000000 11111 00000 00000 00000 001000
NORI	nori rt, rs, imm	I	001101	-	NORI R3,R6,#7	001101 00110 00011 0000000000000111
NANDI	NANDI rt,rs, imm	I	001100	-	NANDI R16,R2,#3	001100 00010 10000 0000000000000011
ADDI	ADDI rt,rs, imm	I	001000	-	ADDI R17,R8, #9	001000 01000 10001 0000000000001001
BEQ	beq rs, rt, offset	I	000100	-	BEQ R16,R17,#4	000100 10000 10001 0000000000000100
LH	lh rt, offset(rs)	I	100001	-	LH R10,0(R16)	100001 10000 01010 0000000000000000
SW	sw rt, offset(rs)	I	101011	-	SW R18, 0(R17)	101011 10001 10010 0000000000000000
J	j target_address	J	000010	-	J R0	000010 00000000000000000000000000

3 Data path design:

The hardware components that are responsible for the execution of instruction represent the data path. The data path is made to handle memory, arithmetic and logical operations on the information kept in the registers. The description of each component in all the five stages are as follows.

3.1 Instruction Fetch

The components in instruction fetch are Program Counter, Multiplexer, Adder, Instruction Memory, and IF/ID Buffer (For pipelining). In this stage, the processor fetches the instruction from the memory.

Program Counter: PC holds the address of the next instruction that needs to be executed. Module name – pc_32
Instruction Memory: Current instructions are stored in the instruction memory. Module name- inst_mem
Multiplexer: Mux (Multiple input single output) is used for selecting input (Memory Address) based on the selection line (Branch AND gate output). Module name- mux1
Adder: Adder is used for incrementing by 4 after the completion of each instruction. Module name- add1
RTL synthesis of Instruction fetch stage is shown below,

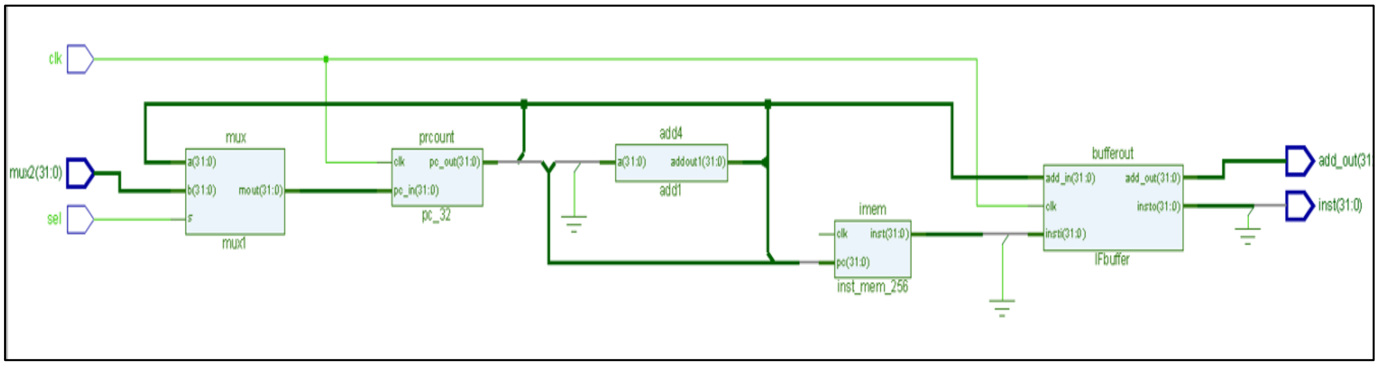


Figure 1:RTL Synthesis for Instruction Fetch Stage

3.2 Instruction Decode:

In the instruction decode stage, instructions in the memory are decoded, that is, the values are decoded in the register file. Control unit decides which operation need to be performed. IF/ID buffer is used for pipelining (Module name- IDbuffer). The component description are as follows,

Register File: It is a 32-bit register, where instruction (25:21) – **rs** represents data1, instruction (20:16)- **rt** represents data2, instruction (15:11) – **rd** represents destination register which will vary based on the instruction type. For R type instruction, rd is the destination register. For I type instruction, rt is the destination register.
Sign Extended: Immediate values are sign extended to 32-bit and used as one of the inputs for performing ALU operation.
Multiplexer: Mux(2:1) is used for selection between data1 and shamt value (shift value). For SRL instruction, shamt value (shifting number) is used for doing shift operation.
Control Unit: Control unit in decode stage helps in deciding which operation need to be performed in execution stage. All the selection line for mux in execution stage comes from control unit.
RTL synthesis of Instruction decode stage is shown below,

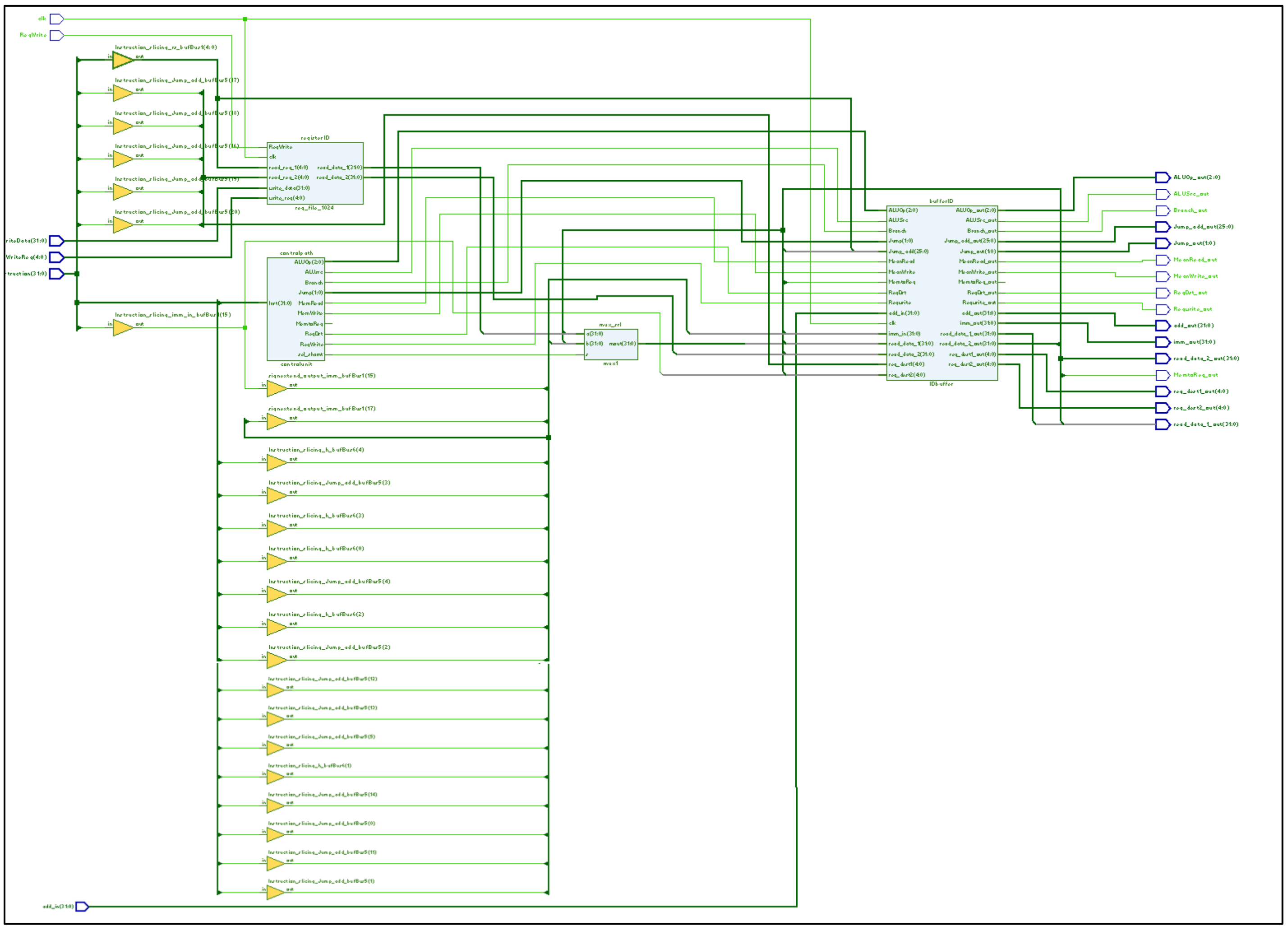


Figure 2:RTL Synthesis for Instruction Decode Stage

3.3 Execution stage:

The decoded instruction from decode stage is sent to execution stage where required operation takes place. The following components are present in this stage. EX/MEM buffer is used (Modeule name - ex_mem_pr)

ALU: ALU unit performs arithmetic and logical operation. There are two output from ALU unit, one is result, which is the actual result after performing operation and other is zero flag. Zero flag is used for identifying branch instruction (i.e) zero flag will be 1 for branch instruction and 0 for other instruction.

Adder with Shift left: This module performs add and shift by 2 operations.

Multiplexer: Two **mux (2:1)** is used. One of them is used for selection between rt and immediate, and other is used for selecting destination register (i.e.) selection between rd and rt. One **mux (3:1)** is used. This is utilized for selecting one of the Branch, JR, J instructions.

Adder: Module name - adder_jr used for Jump register instruction. It performs addition of next PC and (rs) data1 for JR operation.

RTL synthesis of execution stage is shown below,

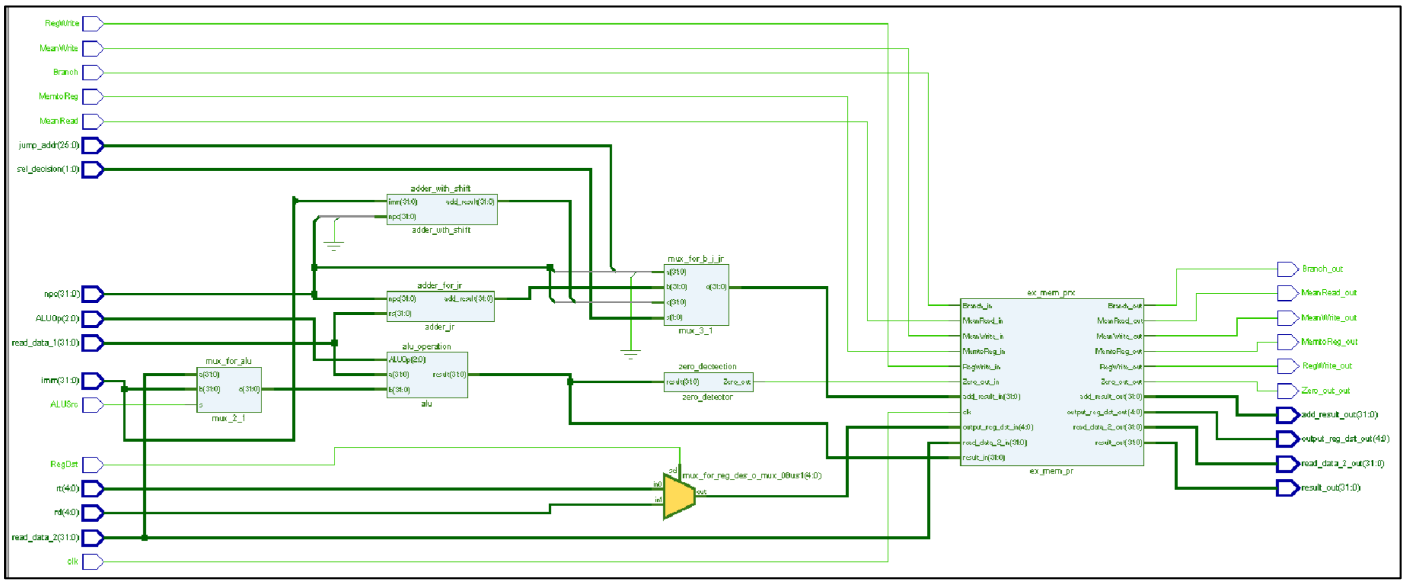


Figure 3:RTL Synthesis for Execution Stage

3.4 Memory Stage:

Load and store instructions are executed in the memory stage. The only component present in this stage is data memory. MeanRead and MeanWrite are the control signals used for controlling between load and store instructions respectively. Whenever MeanRead is 1, Load operation will be performed and this the output from memory stage. Wherever MeanWrite is 1, store operation will be executed where data will be stored in the memory.

RTL synthesis of Memory stage is shown below,

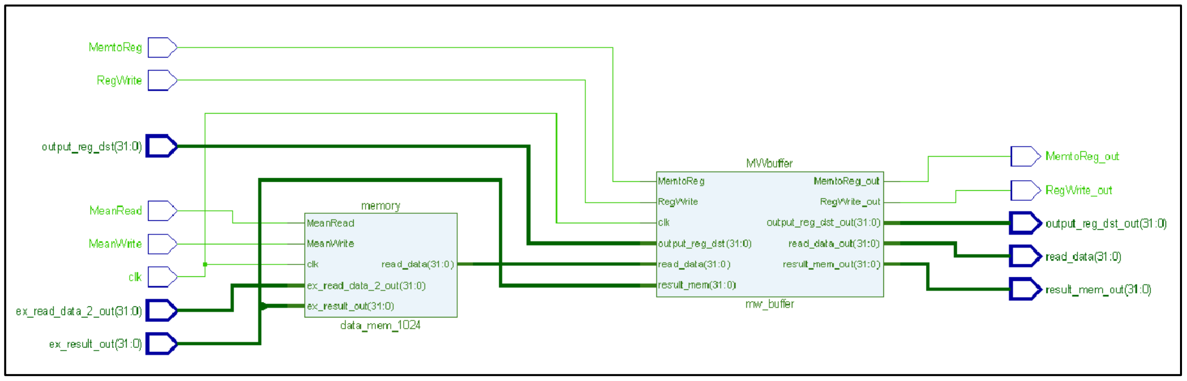


Figure 4: RTL Synthesis for Memory Stage

3.5 WriteBack Stage:

This stage uses (2:1) MUX. The executing result out from execution stage and read data out from memory for load operation are the two inputs (2:1) mux. This stage uses MemtoReg selection line for selecting between loading ALU operation result and Load operation result to the respective destination register (rd or rt).

RTL synthesis of write back stage is shown below,

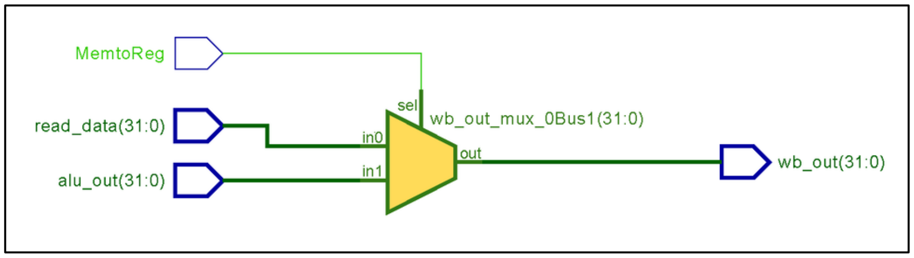


Figure 5: RTL Synthesis for Write Back Stage

4 Control Unit Design:

The Control Unit of the mini-MIPS processor produces the control signals that regulate how the data path operates. After receiving the instruction code from the instruction decoder, the control unit creates the relevant signals to choose the operands, carry out the required operations, and store the outcomes. The following signals are generated in the control unit.

RegWrite (RW) – Indicates whether the operation will write the outcome into a register.
MemtoReg (MTR)– Used for selecting between ALU result and load data to the respective destination register
MemRead(MR) – Used for performing load operation in memory stage.
MemWrite(MW) – Used for store operation in memory stage.
Branch(B) – Used for finding if branch instruction is taken or not.
RegDst – Used for selecting destination register (rd or rd)
ALUOp – Used for controlling ALU operation. Based on ALUOp, ALU unit performs the respective operation.
ALUSrc – Used for selection between immediate and data2 (rt) in execution stage**.**
PCSrc – Used for contolling branch or J or JR instruction.
Sel_JR – Used for selecting JR instruction.
sshamt –sshamt is 1 for SRL instruction execution.
The following table shows the value used for differentiating each instruction.

Instruction	**Operation between**	**Control Unit for EXE stage**			**sshamt**	**Sel_JR**	**B|MR|MW**	**RW |MTR**
		ALUSrc	**ALUOp**	**RegDst**				
NORI	nori rt, rs, immediate	1	110	0	0	XX	0|0 |0	1|1
SRL	srl rd, rt, shamt	0	010	1	1	XX	0|0 |0	1|1
XOR	xor rd, rs, rt	0	100	1	0	XX	0|0 |0	1|1
NANDI	NANDI rt,rs, imm	1	111	0	0	XX	0|0 |0	1|1
SUBU	SUBU rd, rs, rt	0	011	1	0	XX	0|0 |0	1|1
ADDI	ADDI rt,rs, imm	1	000	0	0	XX	0|0 |0	1|1
BEQ	beq rs, rt, offset	0	001	X	0	00	1|0 |0	0|X
LH	lh rt, offset(rs)	1	101	0	0	XX	0|1 |0	1|0
SW	sw rt, offset(rs)	1	101	0	0	XX	0|0 |1	0|X
JR	jr rs	X	X	X	0	01	0| X|X	0|X
J	j target_address	0	X	X	0	10	0|0 |0	0|X

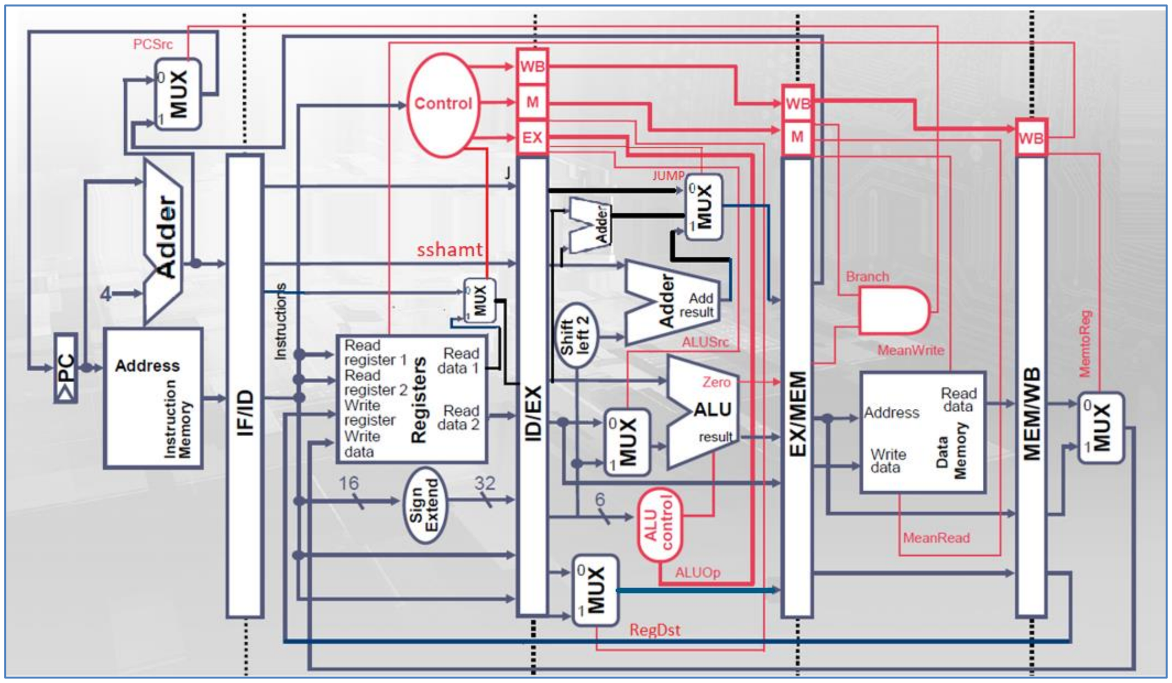


Figure 6: Mini MIPS Pipelined Processor Architecture

5 Hazard Analysis:

Pipelining helps in effective use of processors as it helps in execution of larger number of instructions. There are a few problems that slow down the processor or result in incorrect execution of processor, such as data hazard, control hazard and structural hazard.

i. Structural Hazard:

This is related to hardware usage. Structural hazard happens due to the attempt of using same hardware to do differ operation at the same time. Solutions to structural hazards can be 'writing' in first half of clock period and 'reading' in second half of the clock period (or) Different memory for registers and data memory. Structural hazards are handled by implementing both the above-mentioned solutions in our project.

ii. Data Hazard:

Data Hazards are caused if the instruction is dependent on the outcome of previous instructions that are still in the pipeline. Solutions for data hazards are data forwarding and stalling. In forwarding or bypassing method, the required data is forwarded to the instruction that depends on it. In the stalling method, stalling occurs when the dependent instruction is "pushed back" for one or more clock cycles. Stalling may also be thought of as the execution of a no operation for one or more cycles.

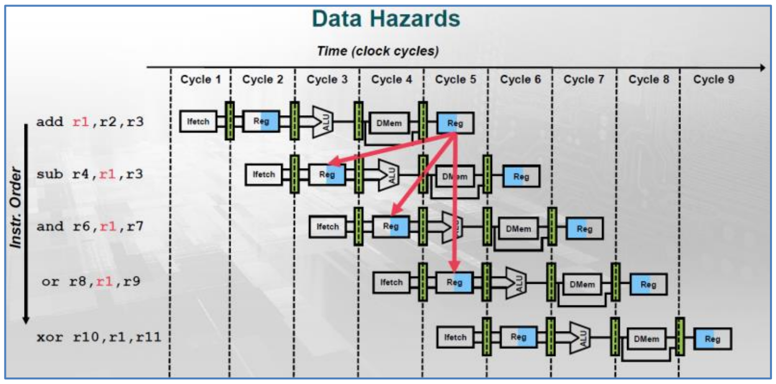


Figure 7: Data Hazard

There are three types of data hazards, Write After Read (WAR), Write After Write (WAW), Read After Write (RAW). WAR, WAW are not in MIPS pipelined structure. For RAW hazard, we have to do both data forwarding (ALU to ALU and memory to memory) and stalling (load hazard).

iii. Control hazard

It arises because of a delay that normally occurs between decisions regarding changes in the flow (branches and leaps) and taking directions. There are many solutions to overcome control or branch hazard. In our project, we will use stalling.

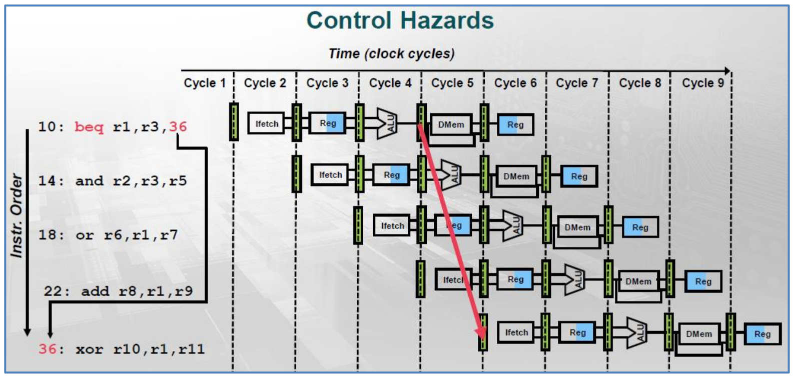


Figure 8: Control Hazard

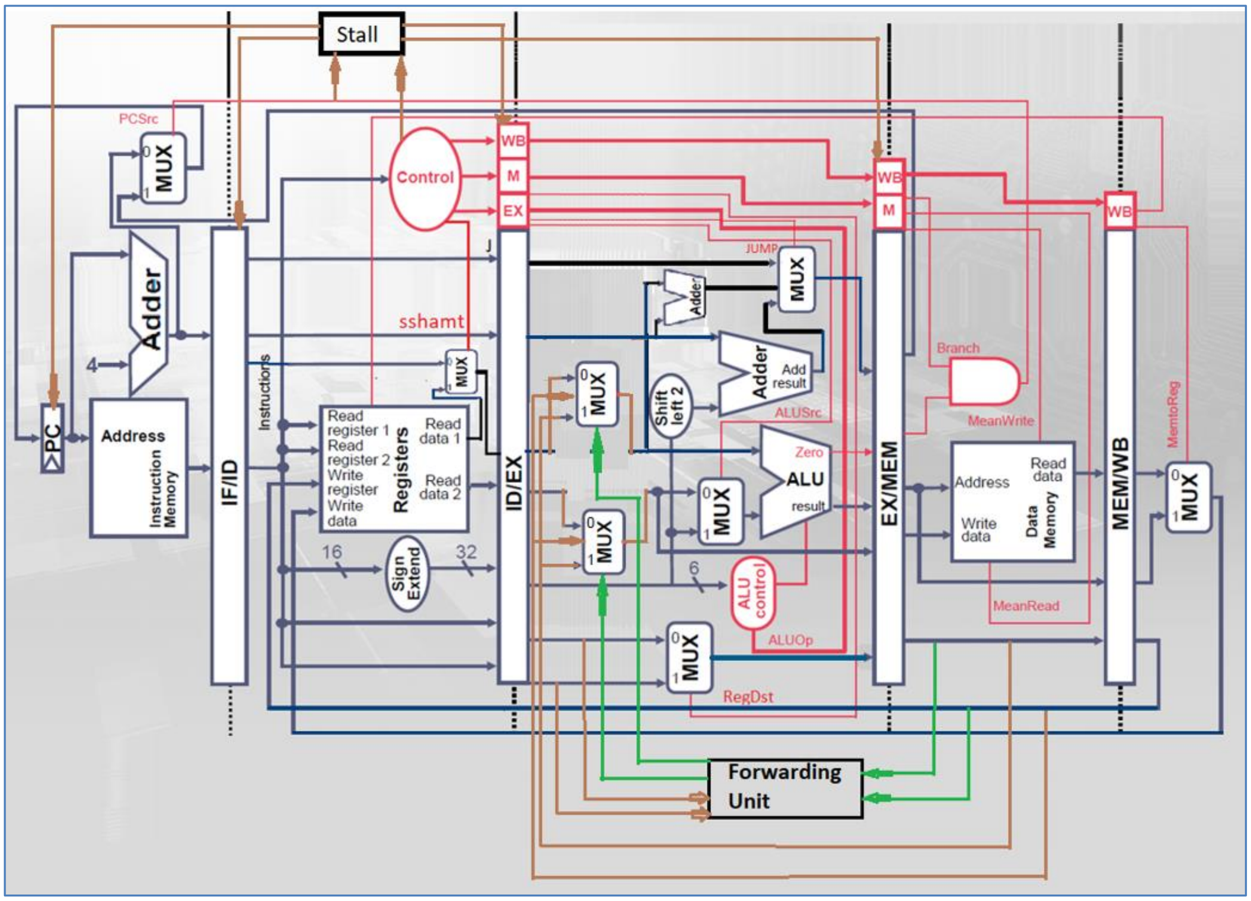


Figure 9: MINI MIPS Processor Architecture with stalling and forwarding unit.

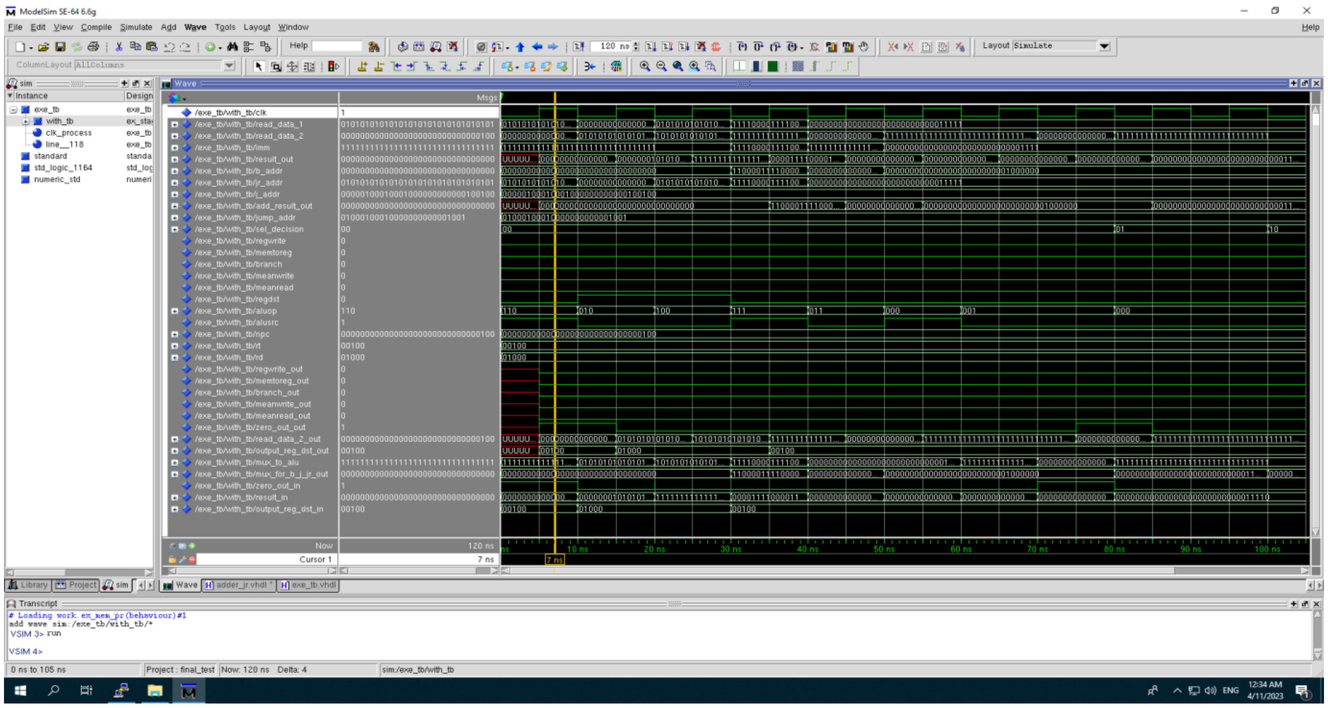
The following table shows the order of instruction and hazard analysis,

Instruction Location	Instruction	Hazards	Solution
0	NORI R3,R6,#7		
4	NANDI R16,R2,#3		
8	XOR R4,R16,R0	**RAW hazard**	DATA FORWARDING FROM ALU TO ALU
12	LH R10,0(R5)		
16	SW R10,0(R7)	**RAW hazard**	DATA FORWARDING FROM MEM TO MEM
20	ADDI R17,R8,#9		
24	BEQ R9,R10,36	**Control hazard [AND JUMPS TO INSTRUCTION LOCATION 172 (36 <<2+ npc = 172)]**	STALL
…	CAN BE ANY INSTRUCTION. ANYWAY, IT DOESN'T TAKE PLACE, INSTEAD STALLING TAKES PLACE		
172	SUBU R11,R12,R13		
176	LH R15,0(R2)		
180	ADDI R18,R15,R0	**Load hazard**	STALL
184	J R14	**Control hazard**	NEXT INSTRUCTION DOESN'T TAKE, STALL TAKES PLACE UNTIL EFFECTIVE ADDRESS IS FOUND.
186	J R0	**Control hazard**	STALL

The corresponding encoding of instruction table is already included.

6 Simulation and Testing Results:

The mini-MIPS processor is designed using VHDL and simulated using ModelSim. The simulated result for each instruction is as follows,

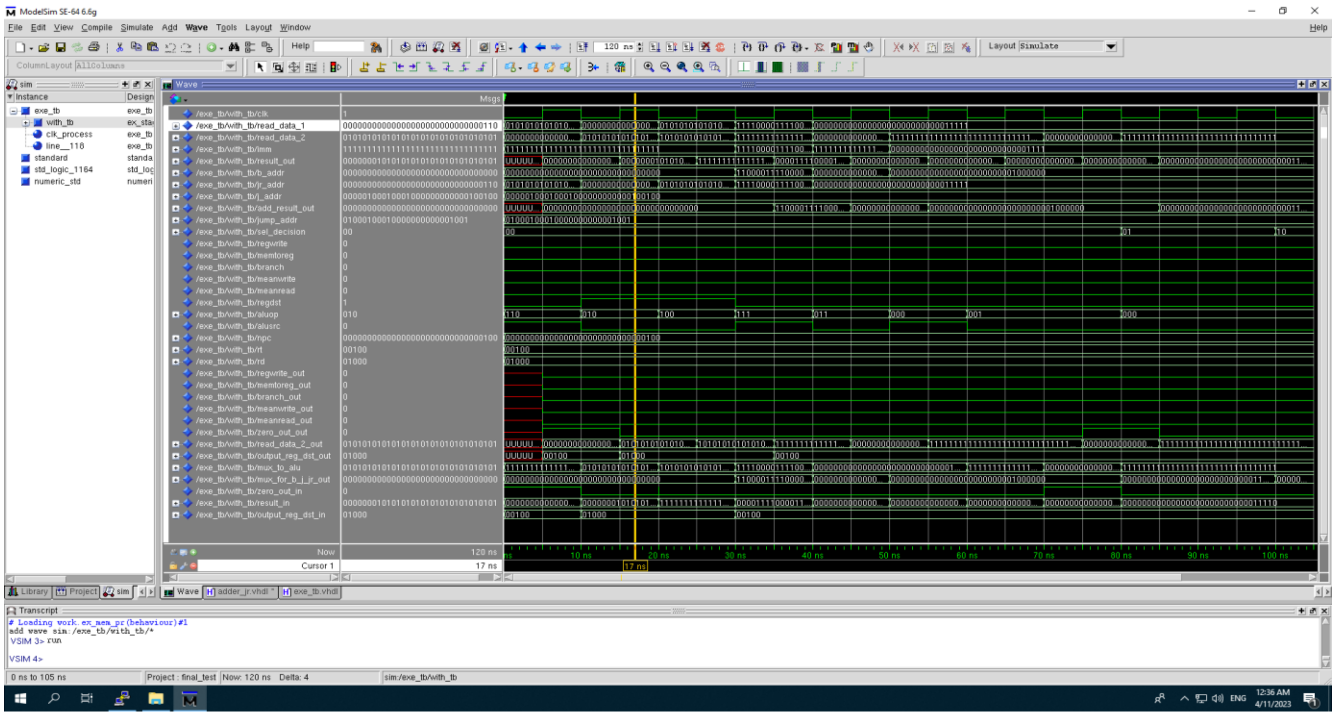


```
/exe_tb/with_tb/clk                   1
/exe_tb/with_tb/read_data_1           01010101010101010101010101010101
/exe_tb/with_tb/read_data_2           00000000000000000000000000000100
/exe_tb/with_tb/imm                   11111111111111111111111111111111
/exe_tb/with_tb/result_out            00000000000000000000000000000000
/exe_tb/with_tb/b_addr                00000000000000000000000000000000
/exe_tb/with_tb/jr_addr               01010101010101010101010101010101
/exe_tb/with_tb/j_addr                00000100010001000000000000100100
/exe_tb/with_tb/add_result_out        00000000000000000000000000000000
/exe_tb/with_tb/jump_addr             01000100010000000000001001
/exe_tb/with_tb/sel_decision          00
/exe_tb/with_tb/regwrite              0
/exe_tb/with_tb/memtoreg              0
/exe_tb/with_tb/branch                0
/exe_tb/with_tb/meanwrite             0
/exe_tb/with_tb/meanread              0
/exe_tb/with_tb/regdst                0
/exe_tb/with_tb/aluop                 110
/exe_tb/with_tb/alusrc                1
/exe_tb/with_tb/npc                   00000000000000000000000000000100
/exe_tb/with_tb/rt                    00100
/exe_tb/with_tb/rd                    01000
/exe_tb/with_tb/regwrite_out          0
/exe_tb/with_tb/memtoreg_out          0
/exe_tb/with_tb/branch_out            0
/exe_tb/with_tb/meanwrite_out         0
/exe_tb/with_tb/meanread_out          0
/exe_tb/with_tb/zero_out_out          1
/exe_tb/with_tb/read_data_2_out       00000000000000000000000000000100
/exe_tb/with_tb/output_reg_dst_out    00100
/exe_tb/with_tb/mux_to_alu            11111111111111111111111111111111
/exe_tb/with_tb/mux_for_b_j_jr_out    00000000000000000000000000000000
/exe_tb/with_tb/zero_out_in           1
/exe_tb/with_tb/result_in             00000000000000000000000000000000
/exe_tb/with_tb/output_reg_dst_in     00100
Now       120 ns
Cursor 1  7 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 10: Simulated result of NORI Instruction

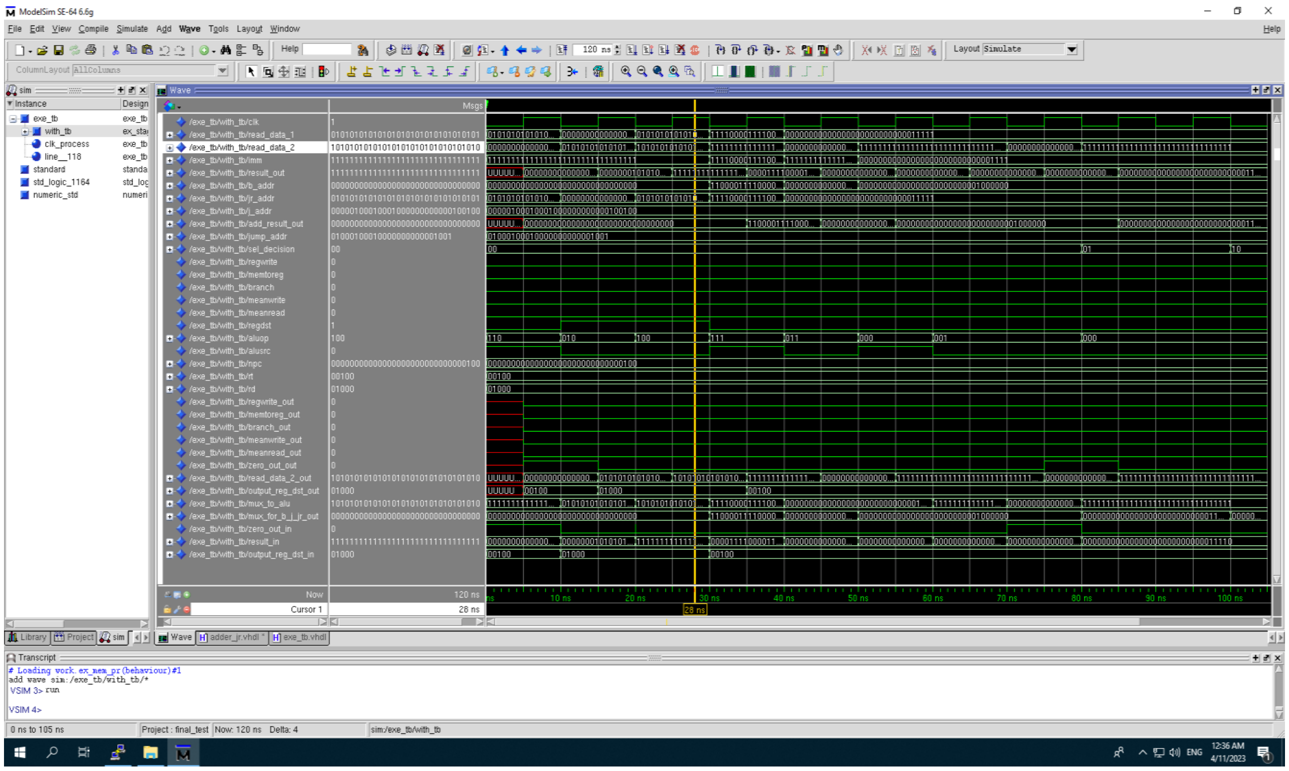


```
/exe_tb/with_tb/clk                   1
/exe_tb/with_tb/read_data_1           00000000000000000000000000000110
/exe_tb/with_tb/read_data_2           01010101010101010101010101010101
/exe_tb/with_tb/imm                   11111111111111111111111111111111
/exe_tb/with_tb/result_out            00000001010101010101010101010101
/exe_tb/with_tb/b_addr                00000000000000000000000000000000
/exe_tb/with_tb/jr_addr               00000000000000000000000000000110
/exe_tb/with_tb/j_addr                00000100010001000000000000100100
/exe_tb/with_tb/add_result_out        00000000000000000000000000000000
/exe_tb/with_tb/jump_addr             01000100010000000000001001
/exe_tb/with_tb/sel_decision          00
/exe_tb/with_tb/regwrite              0
/exe_tb/with_tb/memtoreg              0
/exe_tb/with_tb/branch                0
/exe_tb/with_tb/meanwrite             0
/exe_tb/with_tb/meanread              0
/exe_tb/with_tb/regdst                1
/exe_tb/with_tb/aluop                 010
/exe_tb/with_tb/alusrc                0
/exe_tb/with_tb/npc                   00000000000000000000000000000100
/exe_tb/with_tb/rt                    00100
/exe_tb/with_tb/rd                    01000
/exe_tb/with_tb/regwrite_out          0
/exe_tb/with_tb/memtoreg_out          0
/exe_tb/with_tb/branch_out            0
/exe_tb/with_tb/meanwrite_out         0
/exe_tb/with_tb/meanread_out          0
/exe_tb/with_tb/zero_out_out          0
/exe_tb/with_tb/read_data_2_out       01010101010101010101010101010101
/exe_tb/with_tb/output_reg_dst_out    01000
/exe_tb/with_tb/mux_to_alu            01010101010101010101010101010101
/exe_tb/with_tb/mux_for_b_j_jr_out    00000000000000000000000000000000
/exe_tb/with_tb/zero_out_in           0
/exe_tb/with_tb/result_in             00000001010101010101010101010101
/exe_tb/with_tb/output_reg_dst_in     01000
Now       120 ns
Cursor 1  17 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 11: Simulated result of SRL Instruction



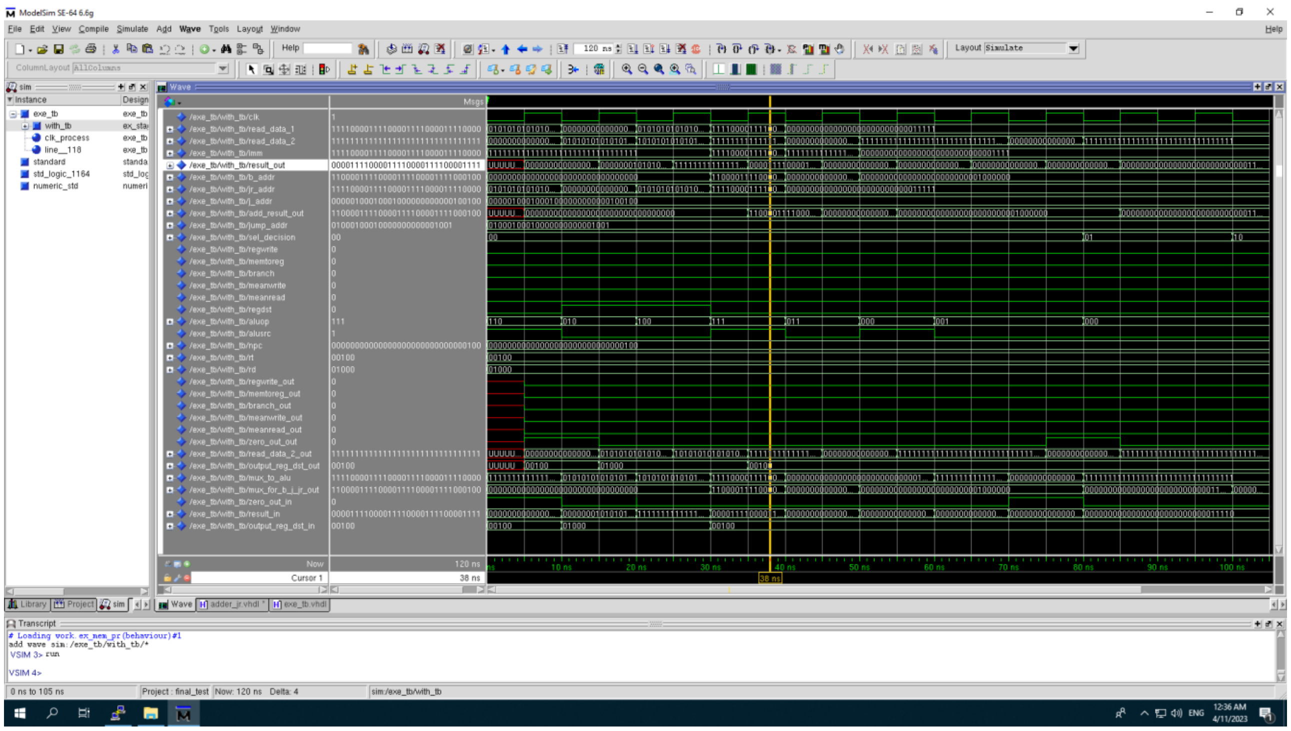
```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          01010101010101010101010101010101
/exe_tb/with_tb/read_data_2          10101010101010101010101010101010
/exe_tb/with_tb/imm                  11111111111111111111111111111111
/exe_tb/with_tb/result_out           11111111111111111111111111111111
/exe_tb/with_tb/b_addr               00000000000000000000000000000000
/exe_tb/with_tb/jr_addr              01010101010101010101010101010101
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       00000000000000000000000000000000
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         00
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               1
/exe_tb/with_tb/aluop                100
/exe_tb/with_tb/alusrc               0
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      10101010101010101010101010101010
/exe_tb/with_tb/output_reg_dst_out   01000
/exe_tb/with_tb/mux_to_alu           10101010101010101010101010101010
/exe_tb/with_tb/mux_for_b_j_jr_out   00000000000000000000000000000000
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_in    01000
Now        120 ns
Cursor 1   28 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

```
0 ns to 105 ns    Project : final_test    Now: 120 ns    Delta: 4    sim:/exe_tb/with_tb
```

Figure 12: Simulated result of XOR Instruction



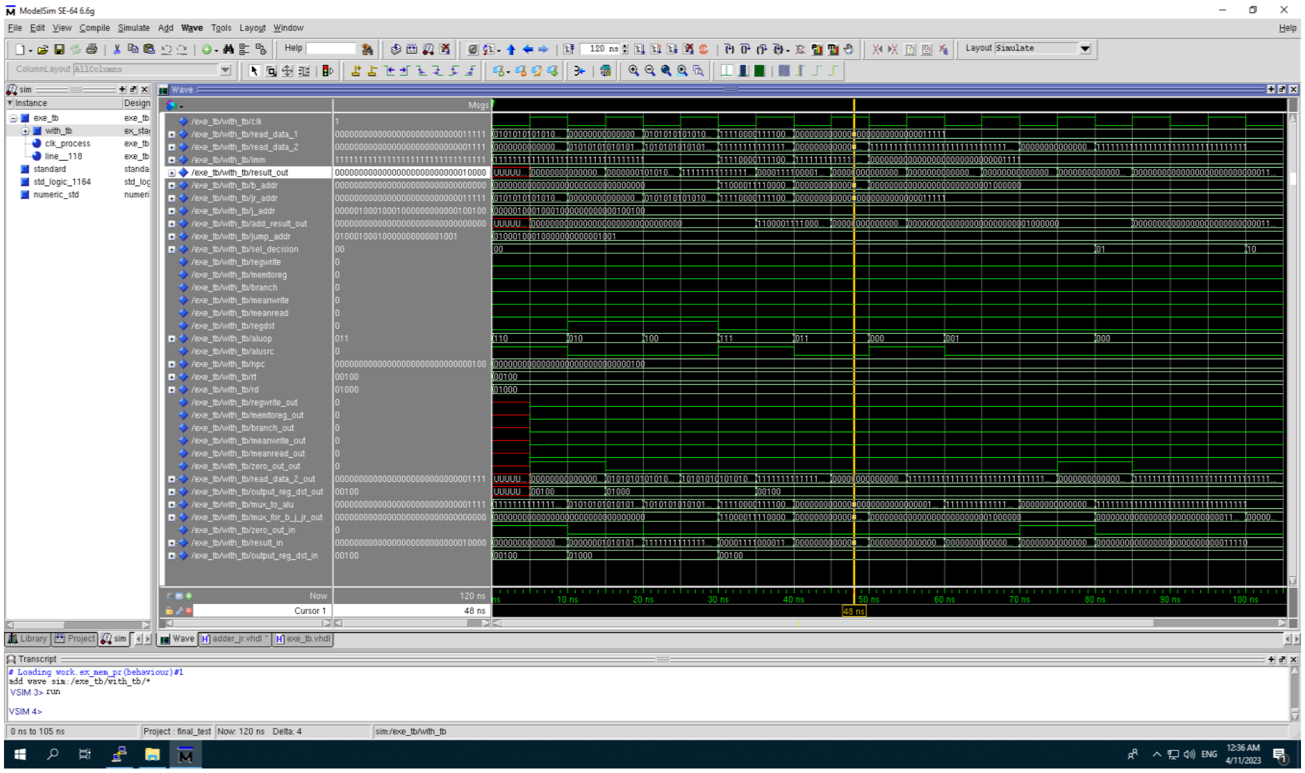
```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          11110000111100001111000011110000
/exe_tb/with_tb/read_data_2          11111111111111111111111111111111
/exe_tb/with_tb/imm                  11110000111100001111000011110000
/exe_tb/with_tb/result_out           00001111000011110000111100001111
/exe_tb/with_tb/b_addr               11000011110000111100001111000100
/exe_tb/with_tb/jr_addr              11110000111100001111000011110000
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       11000011110000111100001111000100
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         00
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               0
/exe_tb/with_tb/aluop                111
/exe_tb/with_tb/alusrc               1
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out   00100
/exe_tb/with_tb/mux_to_alu           11110000111100001111000011110000
/exe_tb/with_tb/mux_for_b_j_jr_out   11000011110000111100001111000100
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            00001111000011110000111100001111
/exe_tb/with_tb/output_reg_dst_in    00100
Now        120 ns
Cursor 1   38 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

```
0 ns to 105 ns    Project : final_test    Now: 120 ns    Delta: 4    sim:/exe_tb/with_tb
```

Figure 13: Simulated result of NANDI Instruction

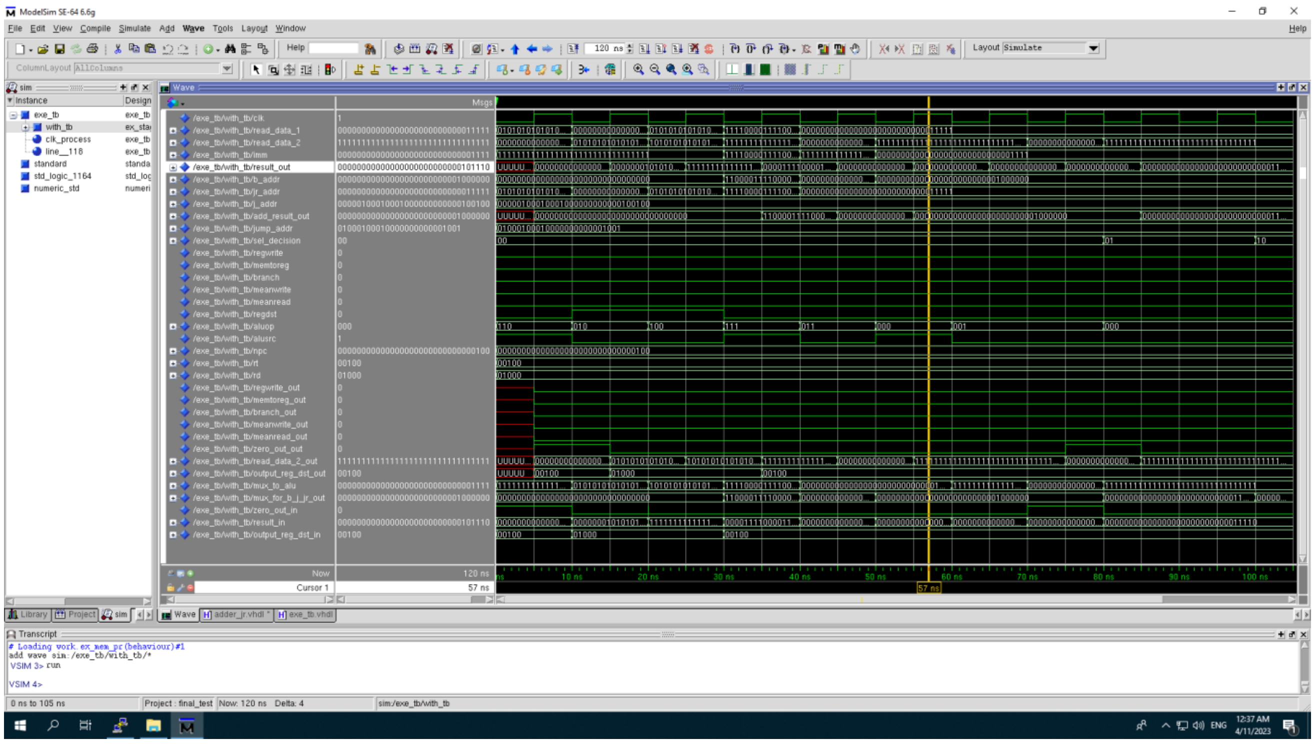


```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          00000000000000000000000000011111
/exe_tb/with_tb/read_data_2          00000000000000000000000000001111
/exe_tb/with_tb/imm                  11111111111111111111111111111111
/exe_tb/with_tb/result_out           00000000000000000000000000010000
/exe_tb/with_tb/b_addr               00000000000000000000000000000000
/exe_tb/with_tb/jr_addr              00000000000000000000000000011111
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       00000000000000000000000000000000
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         00
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               0
/exe_tb/with_tb/aluop                011
/exe_tb/with_tb/alusrc               0
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      00000000000000000000000000001111
/exe_tb/with_tb/output_reg_dst_out   00100
/exe_tb/with_tb/mux_to_alu           00000000000000000000000000001111
/exe_tb/with_tb/mux_for_b_j_jr_out   00000000000000000000000000000000
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            00000000000000000000000000010000
/exe_tb/with_tb/output_reg_dst_in    00100
Now        120 ns
Cursor 1   48 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 14: Simulated result of SUBU Instruction

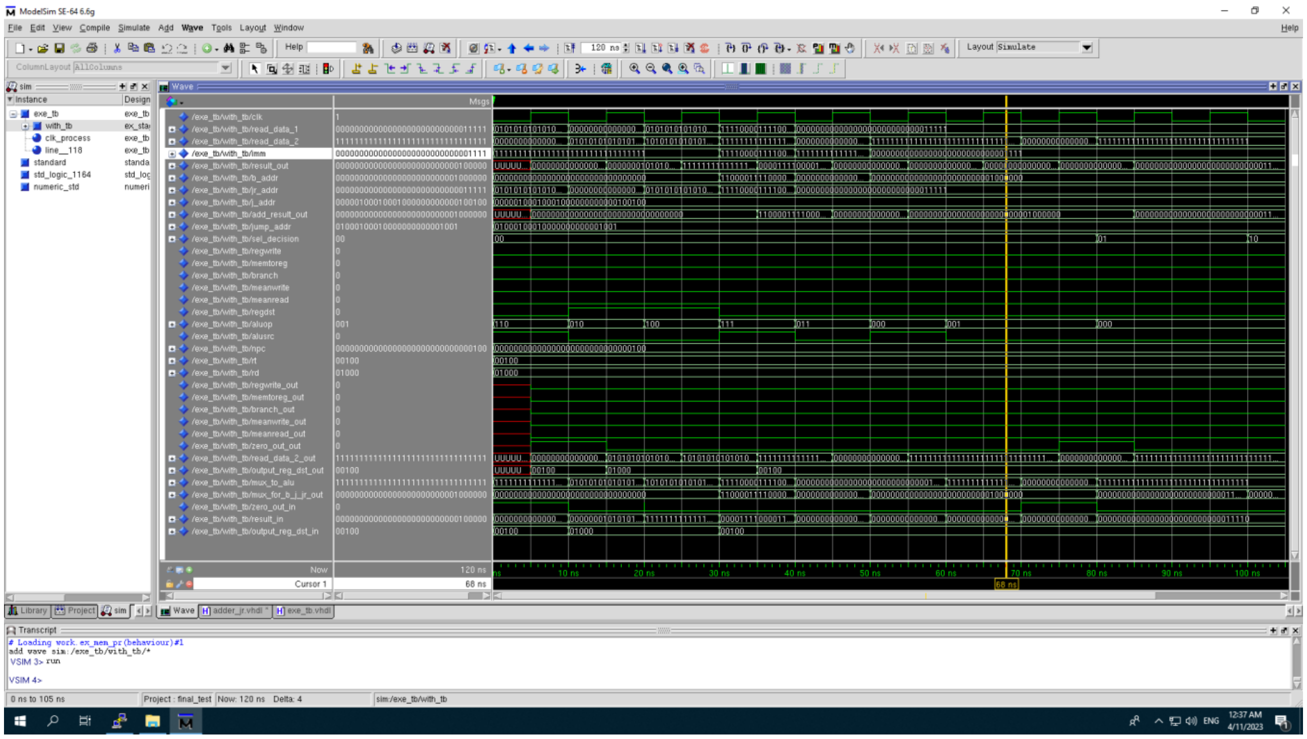


```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          00000000000000000000000000011111
/exe_tb/with_tb/read_data_2          11111111111111111111111111111111
/exe_tb/with_tb/imm                  00000000000000000000000000001111
/exe_tb/with_tb/result_out           00000000000000000000000000101110
/exe_tb/with_tb/b_addr               00000000000000000000000001000000
/exe_tb/with_tb/jr_addr              00000000000000000000000000011111
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       00000000000000000000000001000000
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         00
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               0
/exe_tb/with_tb/aluop                000
/exe_tb/with_tb/alusrc               1
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out   00100
/exe_tb/with_tb/mux_to_alu           00000000000000000000000000001111
/exe_tb/with_tb/mux_for_b_j_jr_out   00000000000000000000000001000000
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            00000000000000000000000000101110
/exe_tb/with_tb/output_reg_dst_in    00100
Now        120 ns
Cursor 1   57 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 15: Simulated result of ADDI Instruction

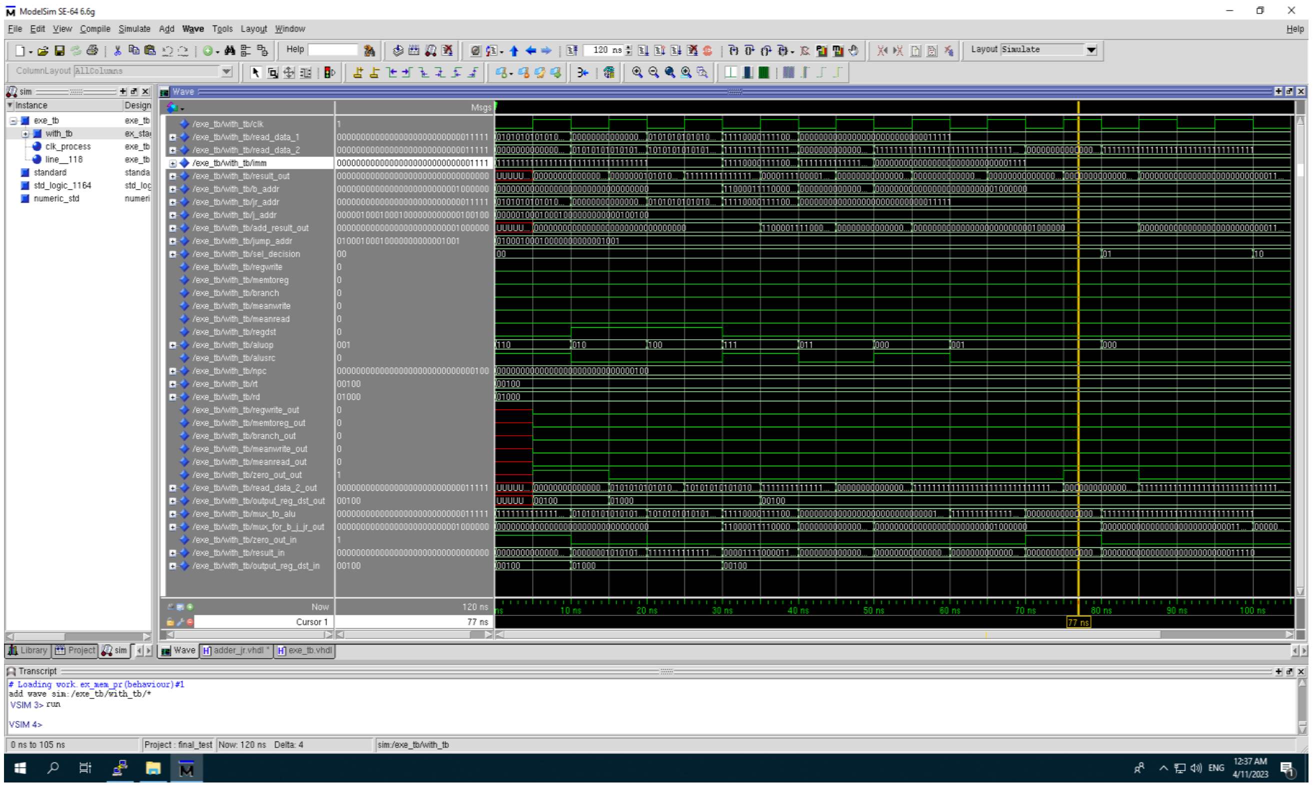


```
/exe_tb/with_tb/clk                 1
/exe_tb/with_tb/read_data_1         00000000000000000000000000011111
/exe_tb/with_tb/read_data_2         11111111111111111111111111111111
/exe_tb/with_tb/imm                 00000000000000000000000000001111
/exe_tb/with_tb/result_out          00000000000000000000000000100000
/exe_tb/with_tb/b_addr              00000000000000000000000001000000
/exe_tb/with_tb/jr_addr             00000000000000000000000000011111
/exe_tb/with_tb/j_addr              00000100010001000000000000100100
/exe_tb/with_tb/add_result_out      00000000000000000000000001000000
/exe_tb/with_tb/jump_addr           0100010001000000000000001001
/exe_tb/with_tb/sel_decision        00
/exe_tb/with_tb/regwrite            0
/exe_tb/with_tb/memtoreg            0
/exe_tb/with_tb/branch              0
/exe_tb/with_tb/meanwrite           0
/exe_tb/with_tb/meanread            0
/exe_tb/with_tb/regdst              0
/exe_tb/with_tb/aluop               001
/exe_tb/with_tb/alusrc              0
/exe_tb/with_tb/npc                 00000000000000000000000000000100
/exe_tb/with_tb/rt                  00100
/exe_tb/with_tb/rd                  01000
/exe_tb/with_tb/regwrite_out        0
/exe_tb/with_tb/memtoreg_out        0
/exe_tb/with_tb/branch_out          0
/exe_tb/with_tb/meanwrite_out       0
/exe_tb/with_tb/meanread_out        0
/exe_tb/with_tb/zero_out_out        0
/exe_tb/with_tb/read_data_2_out     11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out  00100
/exe_tb/with_tb/mux_to_alu          11111111111111111111111111111111
/exe_tb/with_tb/mux_for_b_j_jr_out  00000000000000000000000001000000
/exe_tb/with_tb/zero_out_in         0
/exe_tb/with_tb/result_in           00000000000000000000000000100000
/exe_tb/with_tb/output_reg_dst_in   00100
Now        120 ns
Cursor 1   68 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 16: Simulated result of BEQ Instruction

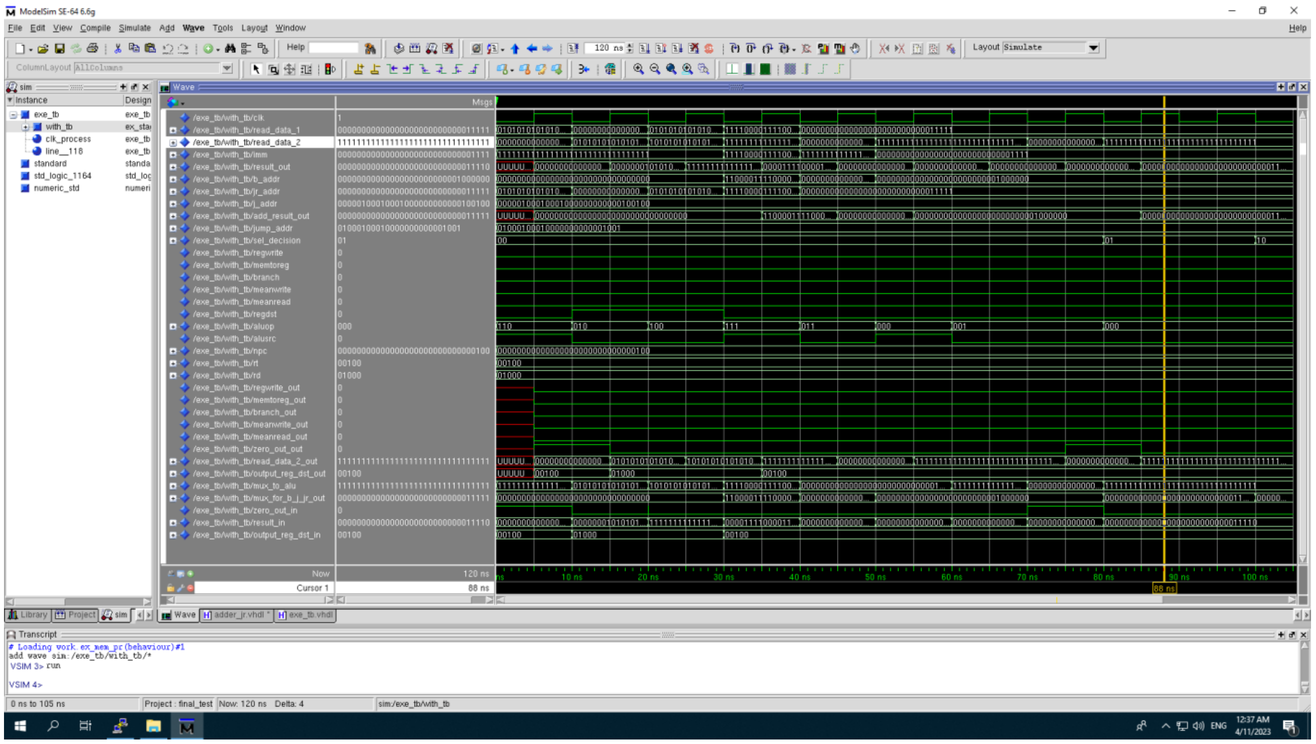


```
/exe_tb/with_tb/clk                 1
/exe_tb/with_tb/read_data_1         00000000000000000000000000011111
/exe_tb/with_tb/read_data_2         00000000000000000000000000011111
/exe_tb/with_tb/imm                 00000000000000000000000000001111
/exe_tb/with_tb/result_out          00000000000000000000000000000000
/exe_tb/with_tb/b_addr              00000000000000000000000001000000
/exe_tb/with_tb/jr_addr             00000000000000000000000000011111
/exe_tb/with_tb/j_addr              00000100010001000000000000100100
/exe_tb/with_tb/add_result_out      00000000000000000000000001000000
/exe_tb/with_tb/jump_addr           0100010001000000000000001001
/exe_tb/with_tb/sel_decision        00
/exe_tb/with_tb/regwrite            0
/exe_tb/with_tb/memtoreg            0
/exe_tb/with_tb/branch              0
/exe_tb/with_tb/meanwrite           0
/exe_tb/with_tb/meanread            0
/exe_tb/with_tb/regdst              0
/exe_tb/with_tb/aluop               001
/exe_tb/with_tb/alusrc              0
/exe_tb/with_tb/npc                 00000000000000000000000000000100
/exe_tb/with_tb/rt                  00100
/exe_tb/with_tb/rd                  01000
/exe_tb/with_tb/regwrite_out        0
/exe_tb/with_tb/memtoreg_out        0
/exe_tb/with_tb/branch_out          0
/exe_tb/with_tb/meanwrite_out       0
/exe_tb/with_tb/meanread_out        0
/exe_tb/with_tb/zero_out_out        1
/exe_tb/with_tb/read_data_2_out     00000000000000000000000000011111
/exe_tb/with_tb/output_reg_dst_out  00100
/exe_tb/with_tb/mux_to_alu          00000000000000000000000000011111
/exe_tb/with_tb/mux_for_b_j_jr_out  00000000000000000000000001000000
/exe_tb/with_tb/zero_out_in         1
/exe_tb/with_tb/result_in           00000000000000000000000000000000
/exe_tb/with_tb/output_reg_dst_in   00100
Now        120 ns
Cursor 1   77 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

Figure 17: Simulated result of LH Instruction



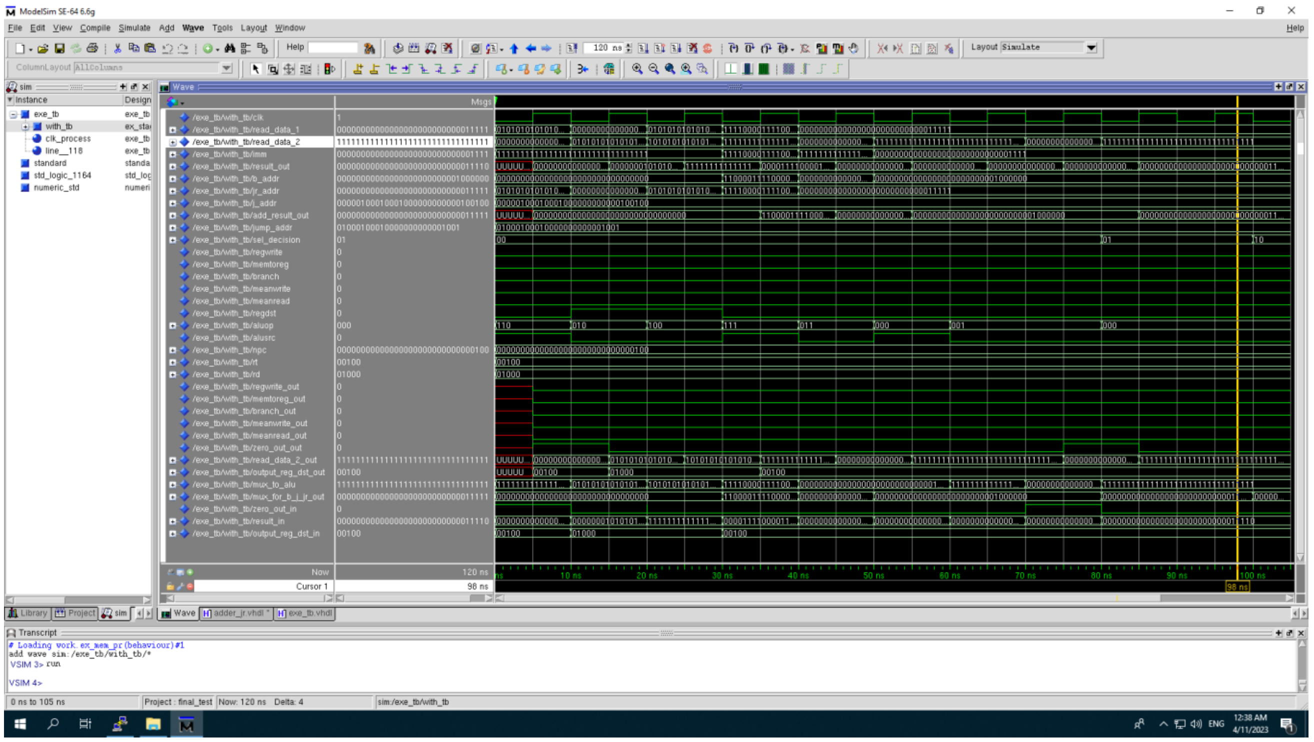
```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          00000000000000000000000000011111
/exe_tb/with_tb/read_data_2          11111111111111111111111111111111
/exe_tb/with_tb/imm                  00000000000000000000000000001111
/exe_tb/with_tb/result_out           00000000000000000000000000011110
/exe_tb/with_tb/b_addr               00000000000000000000000001000000
/exe_tb/with_tb/jr_addr              00000000000000000000000000011111
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       00000000000000000000000000011111
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         01
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               0
/exe_tb/with_tb/aluop                000
/exe_tb/with_tb/alusrc               0
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out   00100
/exe_tb/with_tb/mux_to_alu           11111111111111111111111111111111
/exe_tb/with_tb/mux_for_b_j_jr_out   00000000000000000000000000011111
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            00000000000000000000000000011110
/exe_tb/with_tb/output_reg_dst_in    00100
Now       120 ns
Cursor 1  88 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

```
0 ns to 105 ns    Project : final_test    Now: 120 ns  Delta: 4    sim:/exe_tb/with_tb
```

Figure 18: Simulated result of SW Instruction



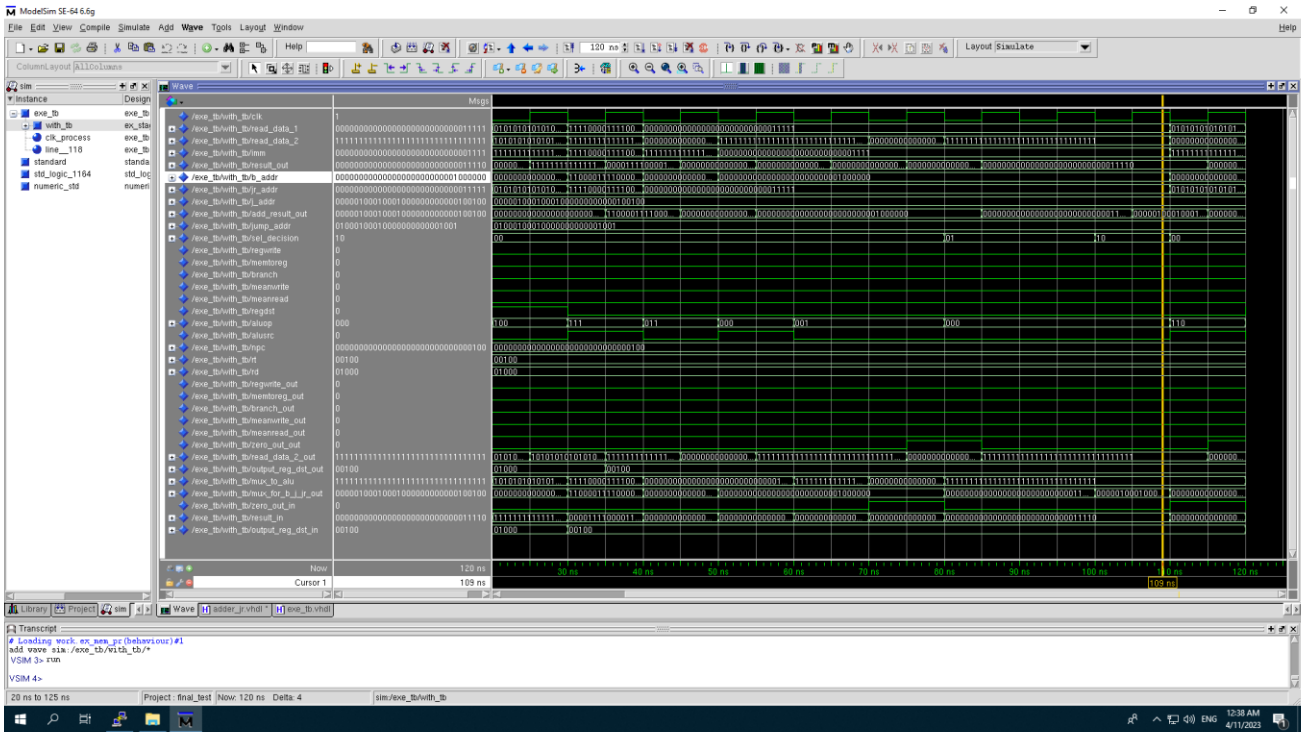
```
/exe_tb/with_tb/clk                  1
/exe_tb/with_tb/read_data_1          00000000000000000000000000011111
/exe_tb/with_tb/read_data_2          11111111111111111111111111111111
/exe_tb/with_tb/imm                  00000000000000000000000000001111
/exe_tb/with_tb/result_out           00000000000000000000000000011110
/exe_tb/with_tb/b_addr               00000000000000000000000001000000
/exe_tb/with_tb/jr_addr              00000000000000000000000000011111
/exe_tb/with_tb/j_addr               00000100010001000000000000100100
/exe_tb/with_tb/add_result_out       00000000000000000000000000011111
/exe_tb/with_tb/jump_addr            0100010001000000000000001001
/exe_tb/with_tb/sel_decision         01
/exe_tb/with_tb/regwrite             0
/exe_tb/with_tb/memtoreg             0
/exe_tb/with_tb/branch               0
/exe_tb/with_tb/meanwrite            0
/exe_tb/with_tb/meanread             0
/exe_tb/with_tb/regdst               0
/exe_tb/with_tb/aluop                000
/exe_tb/with_tb/alusrc               0
/exe_tb/with_tb/npc                  00000000000000000000000000000100
/exe_tb/with_tb/rt                   00100
/exe_tb/with_tb/rd                   01000
/exe_tb/with_tb/regwrite_out         0
/exe_tb/with_tb/memtoreg_out         0
/exe_tb/with_tb/branch_out           0
/exe_tb/with_tb/meanwrite_out        0
/exe_tb/with_tb/meanread_out         0
/exe_tb/with_tb/zero_out_out         0
/exe_tb/with_tb/read_data_2_out      11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out   00100
/exe_tb/with_tb/mux_to_alu           11111111111111111111111111111111
/exe_tb/with_tb/mux_for_b_j_jr_out   00000000000000000000000000011111
/exe_tb/with_tb/zero_out_in          0
/exe_tb/with_tb/result_in            00000000000000000000000000011110
/exe_tb/with_tb/output_reg_dst_in    00100
Now       120 ns
Cursor 1  98 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
VSIM 3> run

VSIM 4>
```

```
0 ns to 105 ns    Project : final_test    Now: 120 ns  Delta: 4    sim:/exe_tb/with_tb
```

Figure 19: Simulated result of JR Instruction



```
/exe_tb/with_tb/clk                 1
/exe_tb/with_tb/read_data_1         00000000000000000000000000011111
/exe_tb/with_tb/read_data_2         11111111111111111111111111111111
/exe_tb/with_tb/imm                 00000000000000000000000000001111
/exe_tb/with_tb/result_out          00000000000000000000000000011110
/exe_tb/with_tb/b_addr              00000000000000000000000001000000
/exe_tb/with_tb/jr_addr             00000000000000000000000000011111
/exe_tb/with_tb/j_addr              00000100010001000000000000100100
/exe_tb/with_tb/add_result_out      00000100010001000000000000100100
/exe_tb/with_tb/jump_addr           01000100010000000000001001
/exe_tb/with_tb/sel_decision        10
/exe_tb/with_tb/regwrite            0
/exe_tb/with_tb/memtoreg            0
/exe_tb/with_tb/branch              0
/exe_tb/with_tb/meanwrite           0
/exe_tb/with_tb/meanread            0
/exe_tb/with_tb/regdst              0
/exe_tb/with_tb/aluop               000
/exe_tb/with_tb/alusrc              0
/exe_tb/with_tb/npc                 00000000000000000000000000000100
/exe_tb/with_tb/rt                  00100
/exe_tb/with_tb/rd                  01000
/exe_tb/with_tb/regwrite_out        0
/exe_tb/with_tb/memtoreg_out        0
/exe_tb/with_tb/branch_out          0
/exe_tb/with_tb/meanwrite_out       0
/exe_tb/with_tb/meanread_out        0
/exe_tb/with_tb/zero_out_out        0
/exe_tb/with_tb/read_data_2_out     11111111111111111111111111111111
/exe_tb/with_tb/output_reg_dst_out  00100
/exe_tb/with_tb/mux_to_alu          11111111111111111111111111111111
/exe_tb/with_tb/mux_for_b_j_jr_out  00000100010001000000000000100100
/exe_tb/with_tb/zero_out_in         0
/exe_tb/with_tb/result_in           00000000000000000000000000011110
/exe_tb/with_tb/output_reg_dst_in   00100
Now                                 120 ns
Cursor 1                            109 ns
```

```
# Loading work.ex_mem_pr(behaviour)#1
add wave sim:/exe_tb/with_tb/*
 VSIM 3> run

VSIM 4>
```

```
20 ns to 125 ns    Project : final_test    Now: 120 ns  Delta: 4    sim:/exe_tb/with_tb
```

Figure 20: Simulated result of J Instruction

Challenges:

- Inclusion of necessary elements was challenging as it was changing throughout the design of the processor. Most of the components are included in decode and execution stages.
- When designing the entire processor, port mapping was challenging.
- Challenges in verification of data path and rectifying delays were interesting.

7 Conclusions

The pipelined mini-MIPS processor is successfully implemented for the given instruction NORI, SRL, XOR, NANDI, SUBU, ADDI, BEQ, LH, SW, JR and J. The simulation result of each instruction and RTL synthesis for each stage of the processor are included. Hazards are handled using forwarding and stalling techniques. The finished architecture can reliably execute all 11 instructions.

8 References

1. G. K. Dewangan, G. Prasad and B. C. Mandi, "Design and Implementation of 32 bit MIPS based RISC Processor," *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*, Noida, India, 2021, pp. 998-1002, doi: 10.1109/SPIN52536.2021.9566007.

2. "Computer Architecture: A Quantitative Approach" (6th Ed.) by John L. Hennessy and David A.Patterson, Elsevier, 2019. Cambridge, MA : Morgan Kaufmann Publishers, [2019]

3. https://www.youtube.com/watch?v=RlQYb3FZtQ8&list=PL59E5B57A04EAE09C&index=27