Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

Computer Architecture

ENCS4370

Project #2

Multi-Cycle RISC Processor Design

Prepared by:

- Ali Shaikh Qasem ID: 1212171
- Abdalrahman Juber ID: 1211769

Instructors: Dr. Ayman Hroub, Dr. Aziz Qaroush.

Sections: 1, 2.

Date: 20/6/2024

Birzeit University

2023-2024

Abstract

This project aims to design and implement a multi-cycle five-stage RISC processor. Specifically, to analyze the given instruction set and design a complete data path covers all of instructions, then, to build and test each individual component required for the processor such as ALU, data memory, etc. After that, connect all individual components to construct the data path. Additionally, to design a control unit that manages the flow of instructions through the data path and connect it to the data path to build the processor, furthermore, to apply a test bench to verify the correctness of the constructed processor.

Contents

Table of figures

List of tables

1. Design and Implementation

1.1. Instruction Memory

The memory was divided into two parts instruction memory and data memory.

In the below figure we design an instruction memory with byte addressable, so our instruction will need to access two addresses to grab its data. And the next instruction will be stored after two bytes.

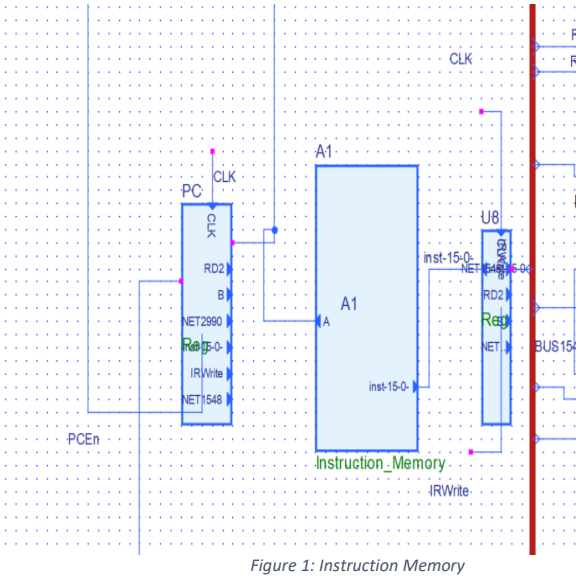

Figure 1: Instruction Memory

1.2. Data Memory

In this part, we designed a byte addressable data memory that takes multiple inputs as address bus, input data, memory read and memory write signals, as well as two additional inputs called CtrlBW and CtrlM, the signal CtrlBW controls the loaded data to be a byte or word, this is to support the LBu and LBs instructions, the CtrlM signal manages the extension of the loaded byte (sign extension or zero extension).

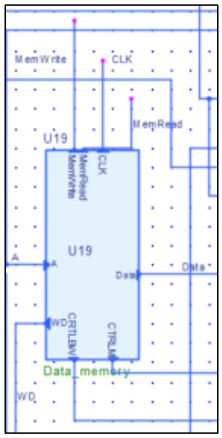

Figure 2: data memory

1.3. Extender

For the sign extension it adds new bits to the left of the data to extend it or to sign it.

It has an input signal as shown below that comes from the control unit.

This signal will determine if the input is a zero extension or sign extension.

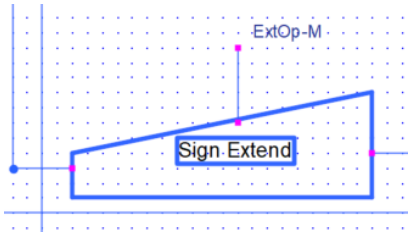

Figure 3 : Extender

1.4. ALU

In this part, we designed a simple ALU that performs only three operation which are ADD, SUB and AND. It has three inputs A, B and ALUcontrol which chose the operation. Values of A and B are chosen among multiple choices using two muxes (mux 2-1 and mux 5-1 as shown below) , this is to support all instruction in the ISA. In addition, it provides three outputs: result, negative flag and zero flag. These flags play an important part in handling branch instructions.

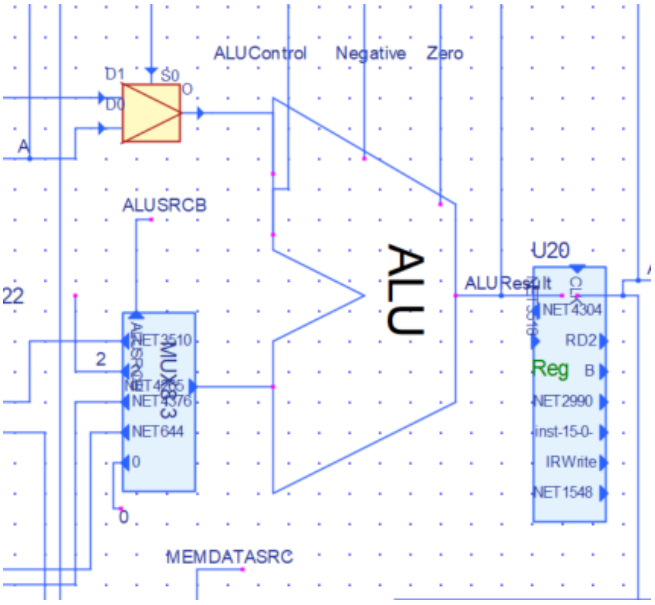

Figure 4: ALU

The following table displays the assigned code for each operation:

Operation	AluOP
ADD	00
SUB	01
AND	10

Table 1: ALU operation codes

1.5. Register File

For this project the register file will have 8 registers so we will have 3 bits for accessing the registers, also it has 4 inputs.

The First one is (RS1) it will access the register number [9:7] for R-Type, [7:5] for I-Type and [9:7] for S-Type.

The second access register will be (RS2) it will access the register number [5:3] for R-Type and [11:8] for I-Type.

Third line input will determine the destination register which will be RD or R7(For CALL inst.).

The last wire is for writing back from ALU result (For CALL inst.), ALU buffer or from Data memory.

The register file has one signal that comes from the control unit which is RegWrite as shown in figure below. This signal will determine if the WB register can be written to the register file or not.

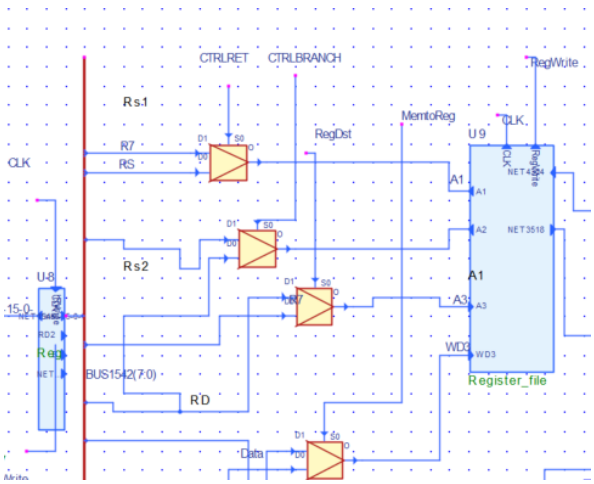

Figure 5: Register File

1.6. Branch Control region

To control the behavior of branch instructions, we added logic gates to the data path as shown below. The negative flag and zero flag are connected to these logic gates to determine if the branch is taken or not based on the subtraction operation between registers Rd and Rs1.

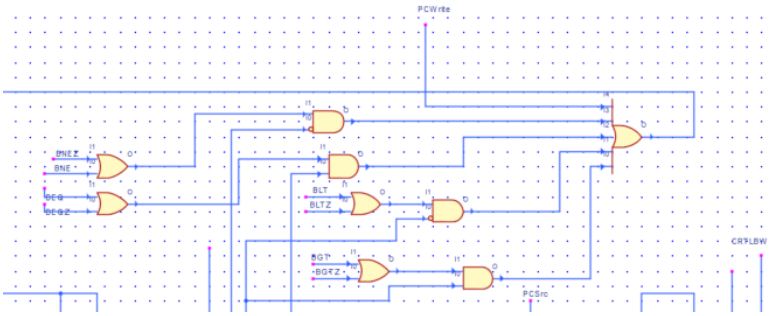

Figure 6: branch control region

1.7. Data Path

In this part, we used all components mentioned above and we connected them to form the complete data path as shown below. The following data path consists of five stages: fetch, decode, execute, memory and write back. As the figure shows, we used buffers to store that stores the result of each stage, this is to get a balance clock cycle length and store the results for next stages, so that the data path woks in multi-cycle design.

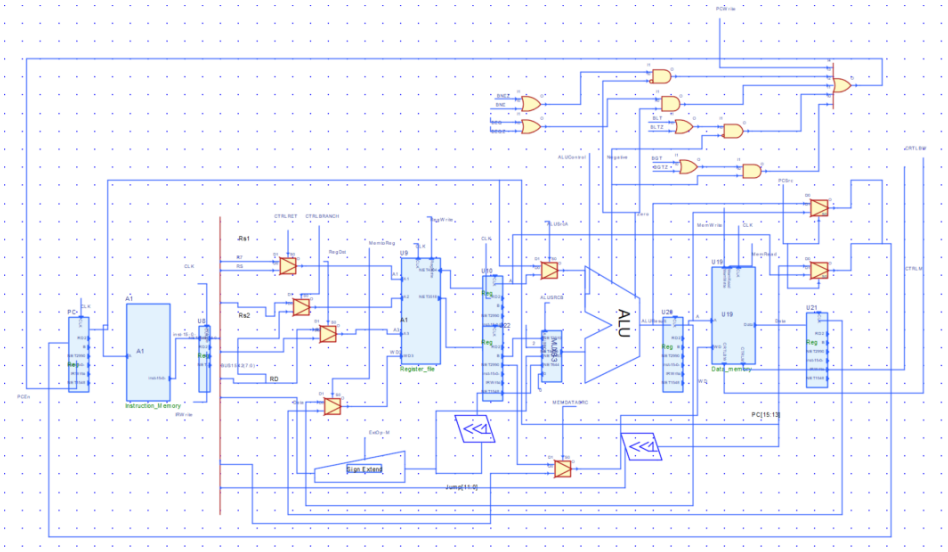

Figure 7: Data Path

1.8. Control Unit

We built the control unit to satisfy all the data path needs.

The control unit takes two inputs opcode and m bit to determine the outputs for each stage for the running instruction as shown in figure 8. Their description is in the table below.

Table 2: Description of the control signals.

PCWrite	is used in fetch to get the new instruction for instruction memory by increasing the PC + 2.
MemRead	determines if the data can be read from the data memory.
MemWrite	determines if the data can be written on the data memory or not.
IRwrite	used in the fetch stage it replaces the old instruction with the new instruction inside the IR register.
RegDest	it determines the register address will be written inside the register file.
AluSrcA	determines the operand for the ALU.
AluSrcB	determines the operand for the ALU.
AluOp	determines the operation for ALU.
PcSoruce	determines the input for the PC register.
CtrlRET	a new signal for CALL instruction and it will replace the RS1 with R7 inside register file.
CtrilBranch	a new signal for branches instructions will replace RS2 with RD inside register file.
ExtOp	is used as input for Extender to determine the zero extend from sign extend.
MemDataSrc	is a new signal to determine whom will write in the data memory SW or SV instructions.
CtrlBW	is a new signal to differentiates between normal load and the load byte inside the data memory.
CtrlM	is a new signal used to load byte with zero extension or to load byte with sign extension.

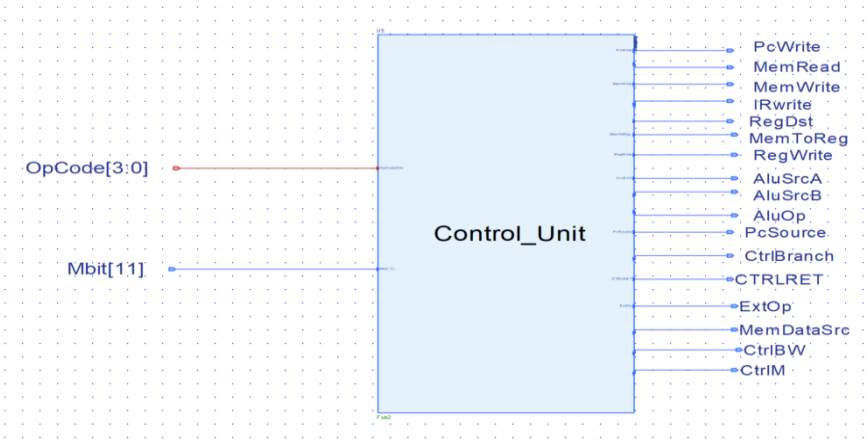

Figure 8: Control Unit

1.9. Truth Table for Control Unit

In the below table we add the truth table for each instruction and with each stage of that instruction.

Each instruction is colored with different color.

Current State	Input (Op)	Next State	Output (Control signals)																
			PC Write	CTRLRET	MemRead	MemWrite	IRWrite	**RegDst**	MemToReg	RegWrite	ALUSrcA	ALUSrcB	ALUOp	PCSource	CTRLBRANCH	ExtOp-M	MEMDATASRC	CtrlBW=1 CtrlM=1(Lbu) 0(LBs)
Instr Fetch	X	Reg Fetch	1	0	0	0	1	X	X	0	0	001	00	0	0	X	X	X
Reg Fetch	Branch	Branch Execute	0	0	0	0	0	X	X	0	0	011	00	X	1	1	X	X
Branch Execute	Branch	Instr fetch	0	0	0	0	0	X	X	0	1	00	01	01	X	X	X	X
Reg Fetch	R-type	R-type execute	0	0	0	0	0	X	X	0	0	000	X	X	0	X	X	X
R-type execute	R-type	ALU write Back	0	0	0	0	0	X	X	0	1	000	R-type	X	X	X	X	X
ALU write Back	R-type	Instr fetch	0	0	0	0	0	0	0	1	X	X	X	X	X	X	X	X
Reg Fetch	LW/SW LBu LBs SV	Computes address	0	0	0	0	0	X	X	0	0	010	X	X	0	X	X	X
Computes address	LW, LBu, LBs	MemRead	0	0	0	0	0	X	X	0	1	010	00	X	X	1	X	X

MemRead	LW, LBu, LBs	Mem Write back	0	0	1	0	0	X	X	0	X	X	X	X	X	X	X	CtrlW =1 CtrlM = 1(Lbu) 0(LBs)
Mem Write back	LW, LBu, LBs	Instr fetch	0	0	0	0	0	0	1	1	X	X	X	X	X	X	X	X
Reg Fetch	I-type	I-type execute	0	0	0	0	0	X	X	0	0	X	X	X	0	X	X	X
I-type execute	I-type	ALU Write back	0	0	0	0	0	X	X	0	1	010	I-type	X	X	M	X	X
ALU Write back	I-type	Instr fetch	0	0	0	0	0	0	0	1	X	X	X	X	X	X	X	X
Reg Fetch	**JMP**	Jump to PC	1	0	0	0	0	X	X	0	X	X	X	10	X	1	X	X
Reg Fetch	CALL	Computes address	0	0	0	0	0	X	X	0	1	001	X	X	X	X	X	X
Execute	CALL	WB to REG R7	0	0	0	0	0	X	X	0	1	001	00	X	X	X	X	X
WB to REG R7	CALL	JUMP	0	0	0	0	0	1	10	1	X	X	X	X	X	X	X	X
JUMP	CALL		1	0	0	0	0	X	X	0	X	X	X	10	X	1	X	X

Reg Fetch	RET	PC = R7	0	1	0	0	0	X	X	0	X	X	X	X	X	X	X	X
PC = R7	RET	Instr fetch	1	0	0	0	0	X	X	0	X	X	X	11	X	X	X	X
Reg Fetch	LW/SW LBu LBs SV	Computes address	0	0	0	0	0	X	X	0	0	010	X	X	0	X	X	X
Computes address	SW	MemWrite	0	0	0	0	0	X	X	0	1	010	00	X	X	1	X	X
Mem Write	SW	Instr fetch	0	0	0	1	0	X	X	0	X	X	X	X	X	X	0	X
Reg Fetch	LW/SW LBu LBs SV	Computes address	0	0	0	0	0	X	X	0	0	010	X	X	0	X	X	X
Computes address	SV	MemWrite	0	0	0	0	0	X	X	0	1	100	00	X	X	1	X	X
Mem Write	SV	Instr fetch	0	0	0	1	0	X	X	0	X	X	X	X	X	X	1	X

Table 3 : Truth Table for the Control signal

1.10. Boolean equation for the control signal

From the previous truth table, we can calculate the boolean equation for each signal.

N == 1 means fetch stage.

N == 2 means decode stage.

N == 3 means execution stage.

N == 4 means memory or write back stage.

N == 5 means write on the memory stage.

Table 4: Boolean equations table

Signal	Equation
PcWrite	(N==1) + ((N==2).(JUMP)) + ((N==3).(RET)) + ((N==5).(CALL))
MemRead	(N==4) . (LW + LBu + LBs)
MemWrite	(N==4) . (SW + SV)
MemWrite	(Stage==4) . (SW + SV)
IRwrite	(N==1)
RegDst	(CALL) . (N==4)
MemToReg	(N==5) . (LW + LBu + LBs)
MemToReg[1]	(N==4) . (CALL)
RegWrite	(N==4) . ((AND + ADD + SUB + ADDI + ANDI + CALL)) + ((N==5).(LW + LBu + LBs))
AluSrcA	(((N==2) + (N==3)) . (CALL)) + (N==3).((BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ) + (AND + ADD + SUB) + (LW + LBu + LBs + SW + ADDI + ANDI + SV));
AluSrcB[0]	(N==1) + (((N==2) + (N==3)) . (CALL)) + ((N==2) . (BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ))
AluSrcB[1]	(N==2) . (LW + LBu + LBs + SV + SW + ADDI + ANDI + BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ) + ((N==3).(LW + LBu + LBs + SW + ADDI + ANDI))
AluSrcB[2]	(N==3) . (SV)
AluOp[0]	(N==3) . (SUB + BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ)
AluOp[1]	(N==3) . (AND + ANDI)
PcSource[0]]	(N==3) . (RET + BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ)
PcSource[1]	((N==2) . (JUMP)) + ((N==5) . (CALL)) + ((N==3).(RET))
CtrlBranch	((N==2) . (BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ)

ExtOp	(N==3) . (LW + LBu + LBs + SW + SV + ADDI) + (N==2) . (JUMP + BGT + BGTZ + BLT + BEQ + BEQZ + BNE + BNEZ) + (N==5).(CALL)
MemDataSrc	(N==4) . (SV)
CTRLRET	(N==2).(RET)
CtrlM	(N==4) . mbit . (LBu)
CtrlBW	(N==4) . (LBu + LBs)

2. Simulation and Testing

2.1. R-type instructions program

We tested all R-type instructions. All the values of operands and registers all shown in the following figures with red square to illustrate.

```
/*

    // --------R-type-------------------//

      register[0] <= {2'b10, R1,3'b000};
        register[1] <= {AND,R3,1'b0};
        register[2] <= {2'b10, R3,3'b000};
        register[3] <= {ADD,R4,1'b0};
        register[4] <= {2'b10, R4,3'b000};
        register[5] <= {SUB,R5,1'b0};
        register[6] <= {3'b001, 5'b00001};
        register[7] <= {RET,1'b0, 3'b010 };
        register[8] <= {3'b001, 5'b00001};


/*
```

Figure 9 : R-type arithmetic logic program

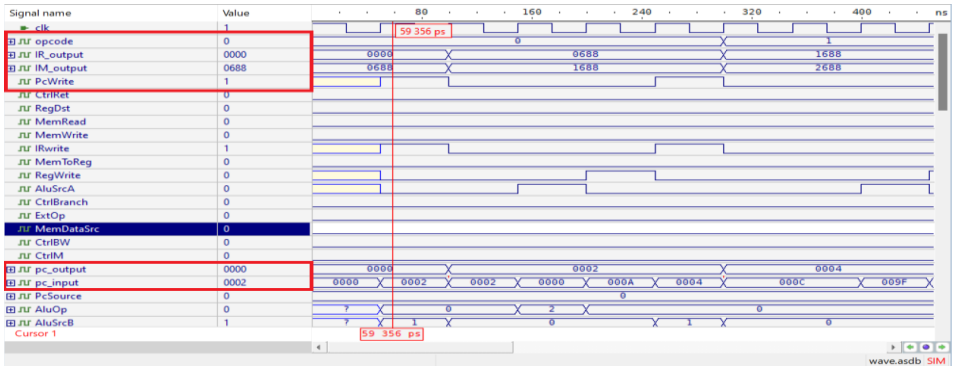

Figure 10: Fetching And instruction

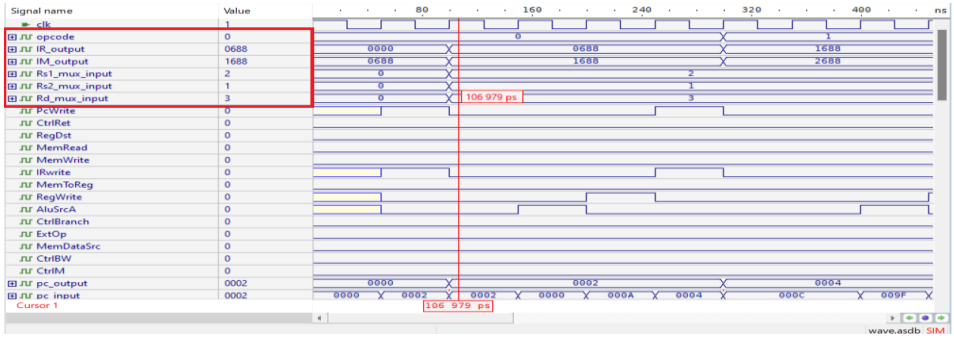

Figure 11: Decode Stage for And

Figure 12: Execution Stage for And

Figure 13 : Write back stage for And

Figure 14: Fetching Add instruction

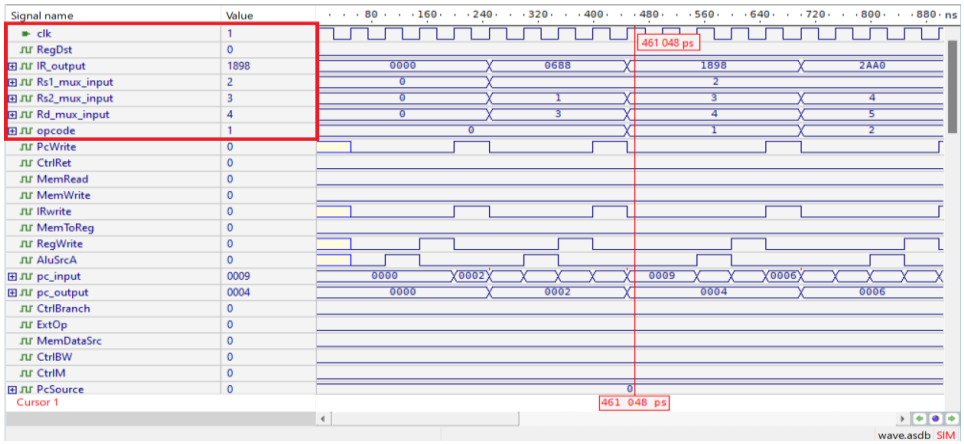

Figure 15: Decode Stage for ADD

As was expected Rs1,Rs2 and Rd were the same as was used in figure 9 for Add instruction which equals

Rs1 = 2 , Rs2 = 3 and Rd = 4.

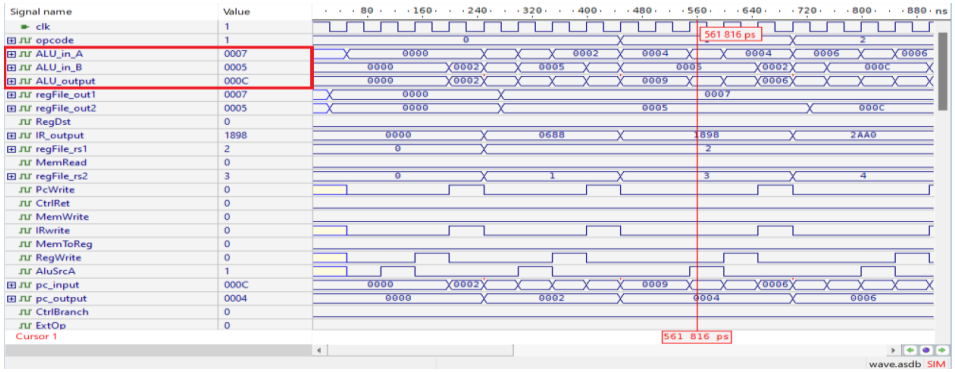

Figure 16: Execution Stage for Add

For and instruction the alu value was 5 saved on R3 register, So we use R3 as Rs2 to Add instruction so we make sure it was saved correctly.

Figure 17: Write back stage for Add

Figure 18: Fetching Sub instruction

Figure 19: Decode Stage for Sub

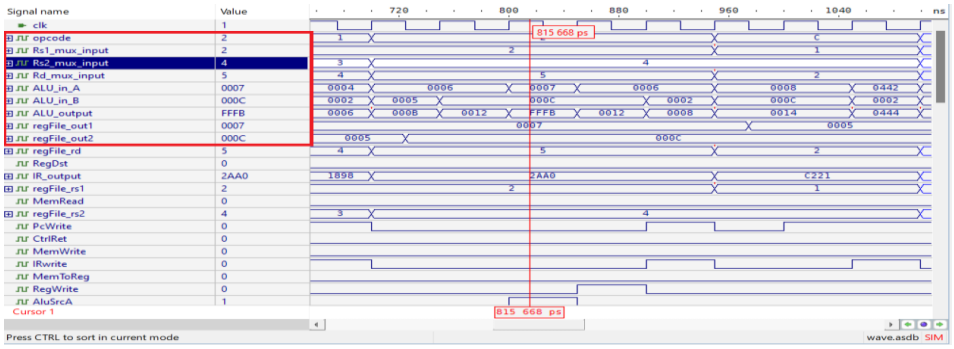

Figure 20: Execution Stage for Sub

Also for Sub instruction we used R4 as Rs2 to test the write back stage for Add and as shown in figure above it was saved correctly.

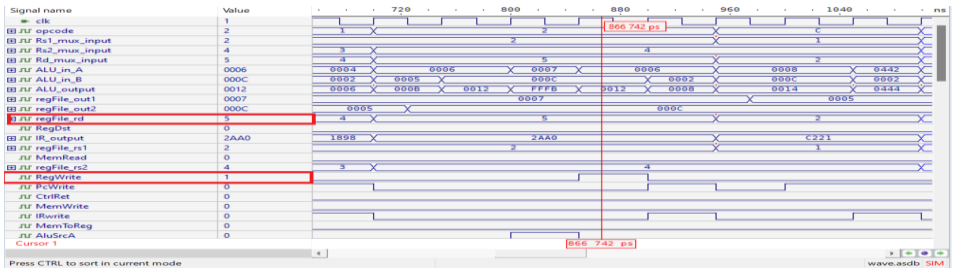

Figure 21: Write back stage for Add

All the instructions worked appropriately as shown in the previous figures with all stages.

2.2. I-type arithmetic and logic program

This simple program contains two instructions as shown below, to test the correctness of I-type logical and arithmetic instructions. Note that the register R1 is initialized with value = 3 before executing the program.

Both instructions are added in the instruction memory as shown below:

```
// ---------------------------- I-type logic arithmetic programm ---------------------------------//

// first instruction (ADDI) R3 = R1 + 5 = 3 + 5 = 8
register[0] <= { R0, 5'b00101 } ;
register[1] <= {ADDI, 1'b1, R3} ;
// second instruction (ANDI) R3 = R1 + 5 = 3 & 5 = 1
register[2] <= { R0, 5'b00101 }  ;
register[3] <= {ANDI, 1'b1, R3} ;
```

Figure 22: I-type arithmetic logic program

The table below describe both programs and illustrate their expected behavior:

Instruction number	Instruction	Behavior
1	ADDI R3, R0, #5	R3 = R0 + 5 = 3 + 5 = 8
2	ANDI R3, R0, #5	R3 = R0 & 5 = (011) & (101) = 001

Table 5: I-type arithmetic logic program

Program test and simulation:

ADDI instruction:

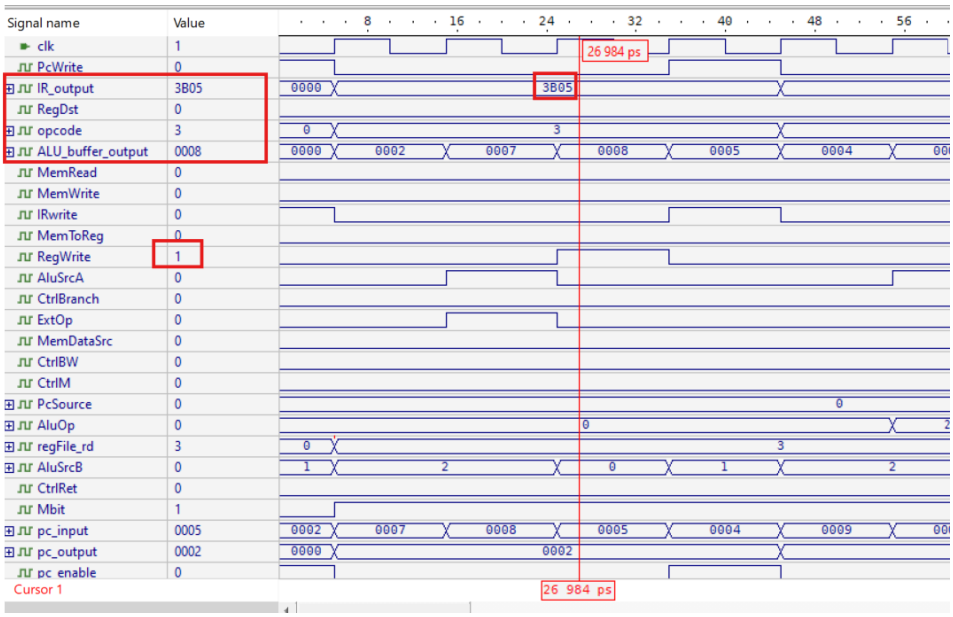

Figure 23: ADDI instruction simulation

From the figure above, we notice that the result of addition in the ALU buffer is 8 which is the correct result, additionally we see that the destination register is 0 and the RegWrite control signal is high, thus, the value will be written in register file at the correct place, hence the instruction works well.

ANDI instruction:

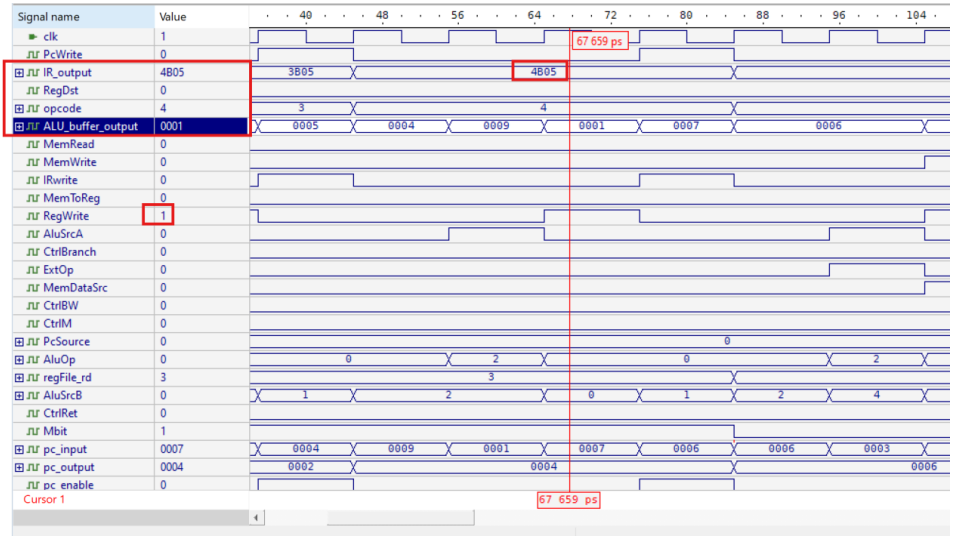

Figure 24: ANDI instruction simulation

From results above, we notice that the result of bitwise AND is 1 which is the right result, and it will be stored at the right place at register file, thus, this instruction also works well.

2.3. LOAD and STORE program

This program contains four different instructions SW, LW, LBu, and LBs. to verify the correctness of load store instructions. Note that the following registers and memory slots are initialized with the corresponding values:

R0 = 5, R1 = 0, Mem[6] = -3.

Instructions are added in the instruction memory as shown:

```
`include "parameters.v"
module instructionMemory(Address,Instruction);
    input [15:0]Address;
    output  [15:0] Instruction;
    reg [7:0] register[63:0];     //Byte addressable memory


    assign Instruction = {register[Address+1],register[Address]};

    initial
        begin

            /*
            // ----------------------------- LOAD STORE programm ---------------------------------//

            // first instruction (STORE WORD) --> mem[2] = R0 =  5
        register[0] <= { R1, 5'b00010 } ;
        register[1] <= {SW, 1'b0, R0} ;
            // second instruction (LOAD WORD) --> R2 = mem[2] = 5
        register[2] <= {R1, 5'b00010  } ;
        register[3] <= { LW, 1'b0, R2} ;
           // THIRD instruction (LBu) --> R3 = mem[6] = -3 (unsigned extended)
        register[4] <= { R1, 5'b00110 } ;
        register[5] <= {LBu, 1'b0, R3} ;
           // THIRD instruction (LBs) --> R3 = mem[6] = -3 (signed extended)
        register[6] <= { R1, 5'b00110 } ;
        register[7] <= {LBs, 1'b1, R3} ;

        // ------------------------------------------------------------------------------- //
          */
```

Figure 25: load store program

The table below describe both programs and illustrate their expected behavior:

Instruction number	Instruction	Behavior
1	SW R0, 2(R1)	Mem[R1 + 2] = R0 → Mem[2] = 5
2	LW R2, 2(R1)	R2 = Mem[R1 + 2] = 5
3	LBu R3, 6(R1)	R3 = zero extended(Mem [6]) = -3 = 0x00FD
3	LBs R3, 6(R1)	R3 = sign extended(Mem [6]) = -3 = 0xFFFD

Table 6: LOAD STORE program

Program test and simulation:

SW instruction:

Formatted: Font: (Default) +Body (Calibri), 11 pt, Complex Script Font: +Body CS (Arial), 11 pt

Figure 26: SW instruction simulation

From the figure above, we notice that the value of register R0 (which is 5) is stored at address 2 of data memory, and the memory write signal is high, thus, the instruction works well and the value will be stored correctly.

LW instruction:

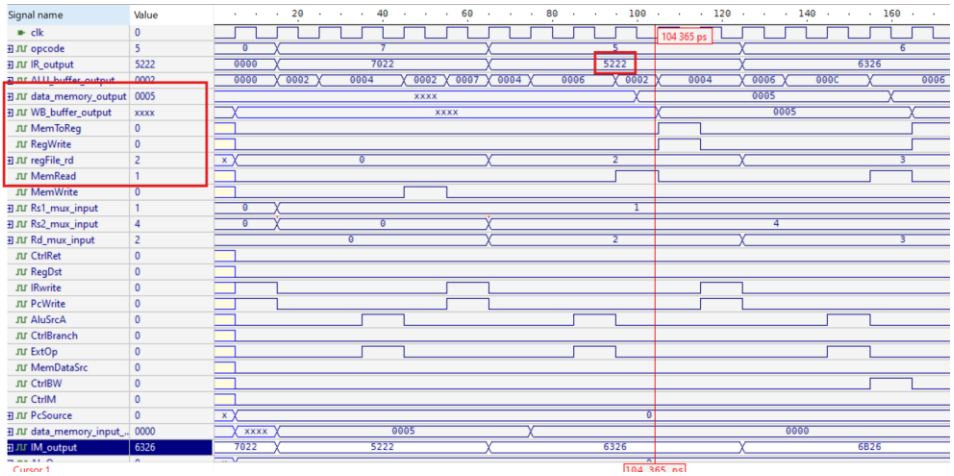

Figure 27: LW instruction simulation

From the figure above, we notice that the value 5 will be stored at the register R2, thus, the instruction works fine.

LBu instruction:

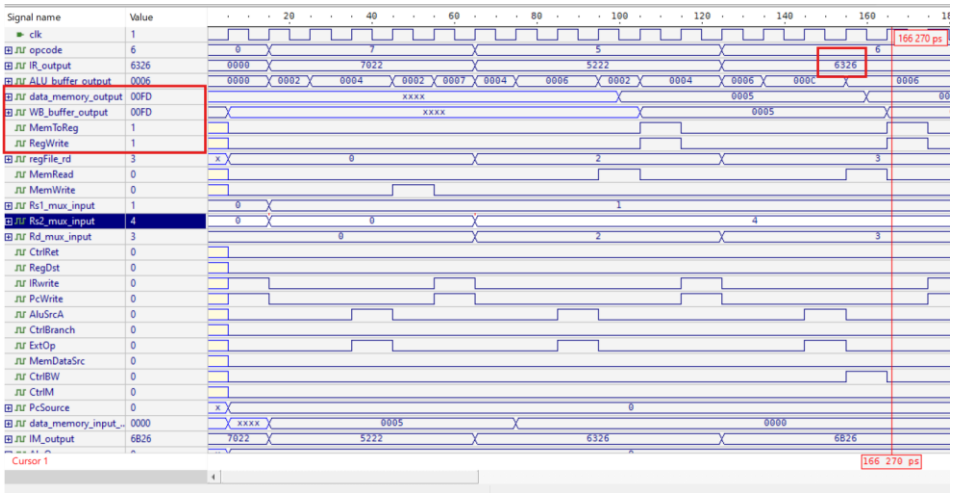

Figure 28: LBu instruciton simulation

Formatted: Normal

From the figure above, we notice that the value 00FD (which is the zero extended -3) will be stored at the register R3, thus, the instruction works fine

LBs instruction:

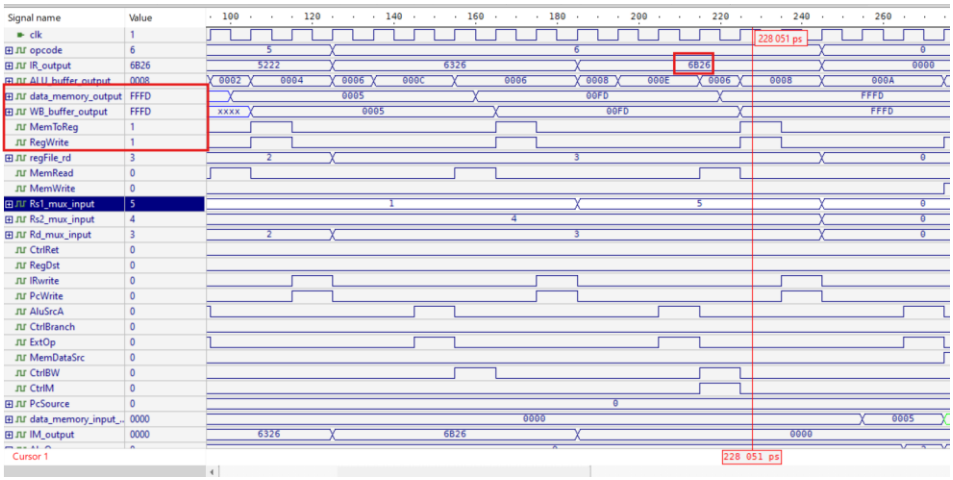

Figure 29: LBs instruction simulation

From the figure above, we notice that the value FFFD (which is the sign extended -3) will be stored at the register R3, thus, the instruction works fine.

2.4. Branches Instructions program

For brevity we just used BLT taken and BLT non taken and BGTZ taken and BGTZ non taken as shown in the following pictures.

```
assign Instruction = {register[Address+1],register[Address]};

initial
    begin
     register[2] <= {R3,5'b01000};  // Not taken
      register[3] <= {BLT,1'b0,R2};
      register[4] <= {R5,5'b01000};
      register[5] <= {BLT,1'b0,R4};    // taken
      register[7] <= {SUB,R5,1'b0};
      register[8] <= {3'b001, 5'b00001};
      register[9] <= {RET,1'b0, 3'b010 };
      register[10] <= {3'b001, 5'b00001};
      register[11] <= {RET,1'b0, 3'b001 };
      register[12] <=  {3'b001, 5'b00001};
      register[13] <=  {BGT,1'b1, 3'b010 };
      register[14] <= {3'b001, 5'b00001};
      register[15] <=  {SW,1'b0, 3'b010 };
      register[16] <=  {3'b001, 5'b00001};
      register[17] <=    {BGT,1'b0, 3'b001 };
      register[18] <=    {3'b001, 5'b00001};
      register[19] <=  {BGT,1'b0, 3'b001 };
      register[20] <= 8'h64;
    end
```

```
initial begin
    Registers[0] <= 16'h0000;
    Registers[1] <= 16'h0005;
    Registers[2] <= 16'h0009;
    Registers[3] <= 16'h0007;
    Registers[4] <= 16'h0001;
    Registers[5] <= 16'h0006;
    Registers[6] <= 16'h0010;
    Registers[7] <= 16'h0027;
end
```

Figure 30: BLT (taken and non-taken) instructions with Register file values

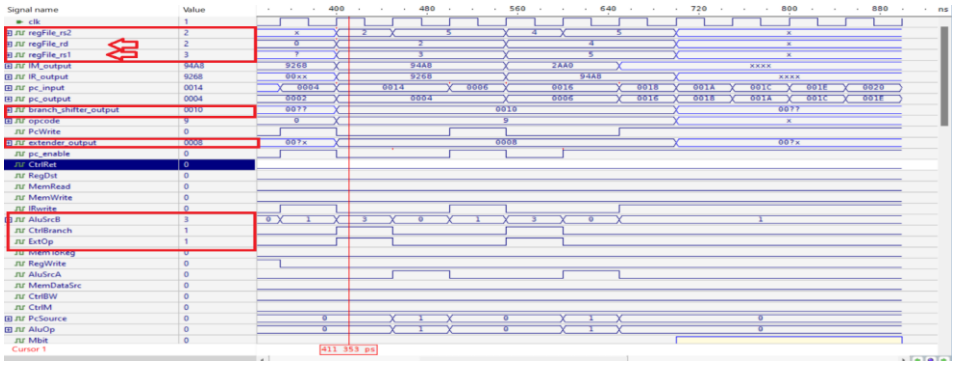

Figure 31: Decode Stage For BLT

First we used BLT non taken. For this instruction when it reaches decode stage it computes the register to process them and at the same time it computes the value of the new pc if the instruction was taken.

So, at this stage we the new PC will saved on the ALU buffer waiting for the execution to be done to take the appropriate action.

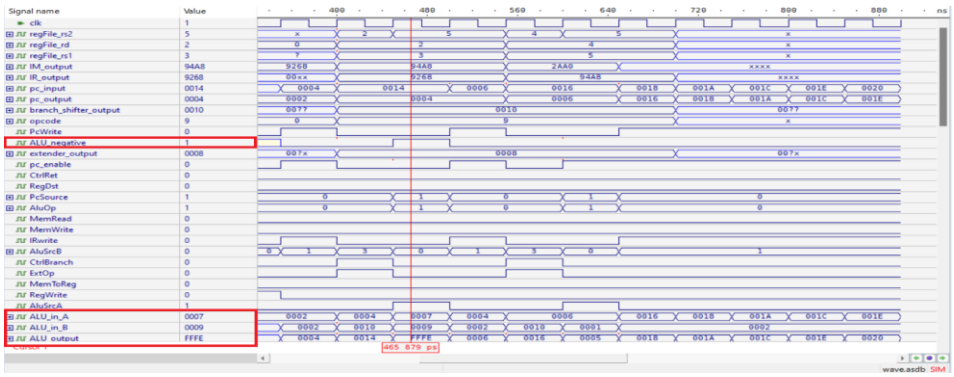

Figure 32: Execution Stage for BLT (non-taken)

For this stage we subtract Rs1 from Rd and if the result equals negative then its non-taken.

For our instruction Rd equals 9 and Rs1 equals 7 so by subtracting 7 – 9 we got a negative flag.

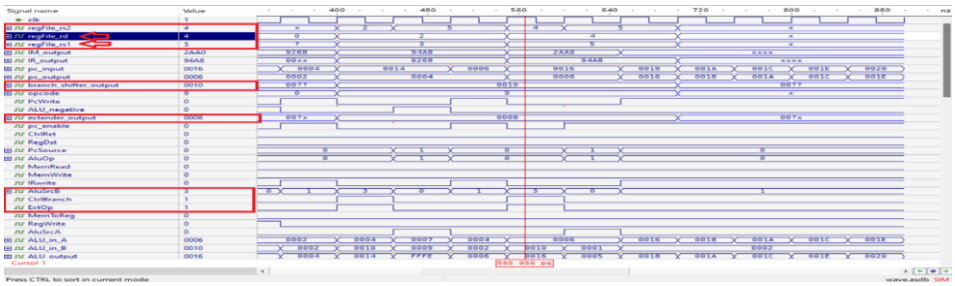

Figure 33: Decode Stage For BLT(Taken)

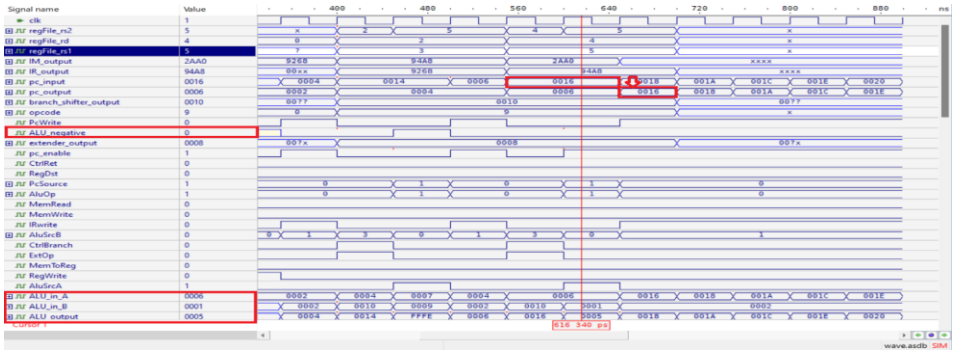

Figure 34: Execution Stage for BLT (Taken)

For the BLT taken instruction as shown in the execution stage in figure 34 the negative flag was zero because we used Rd equals R4 and Rs1 equals R5 which equals 6 - 1 = 5 as shown above.

In this case and because the negative flag equals zero we branch to the new PC value. And it was computes at decode stage.

Our immediate equals 01000 in binary with shift by one to the left it becomes 10000 if we notice that we have an output names branch shifter output will do that operation and it equals 0010 in Hexadecimal after that we add that value to PC value and becomes 0006 + 0010 = 0016 and as shown in figure 34 the PC will jump to that value.

For BGTZ (Non taken) and BGTZ (Taken) we did use the following instruction and register file values.

```
module RegisterFile(ReadRegister1, ReadRegister2, WriteReg

    input [2:0] ReadRegister1,ReadRegister2,WriteRegister;
    input [15:0] WriteData;
    input RegWrite,Clk;

    output reg [15:0] ReadData1,ReadData2;


    reg [15:0] Registers [0:7];

    initial begin
        Registers[0] <= 16'h0000;
        Registers[1] <= 16'h0005;
        Registers[2] <= 16'hF000;
        Registers[3] <= 16'h0007;
        Registers[4] <= 16'h0001;
        Registers[5] <= 16'h0006;
        Registers[6] <= 16'h0016;
        Registers[7] <= 16'h0027;
    end
```

```
assign Instruction = {register[Address+1],register[Address]};

initial
    begin
    register[1] <= 0;
    register[2] <= {R3,5'b01000};  // Not taken
    register[3] <= {BGTZ,1'b1,R2};
    register[4] <= {R5,5'b01000};
    register[5] <= {BGTZ,1'b1,R4};  // taken
```

Figure 35: BGTZ (taken and non-taken) instructions with Register file values

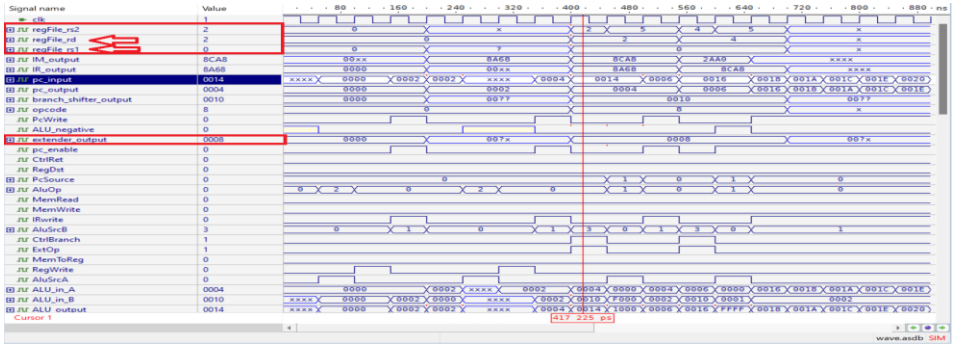

Figure 36: Decode Stage for BGTZ (non-Taken)

When we counter this type of instruction, we set the RS1 to the register R0.

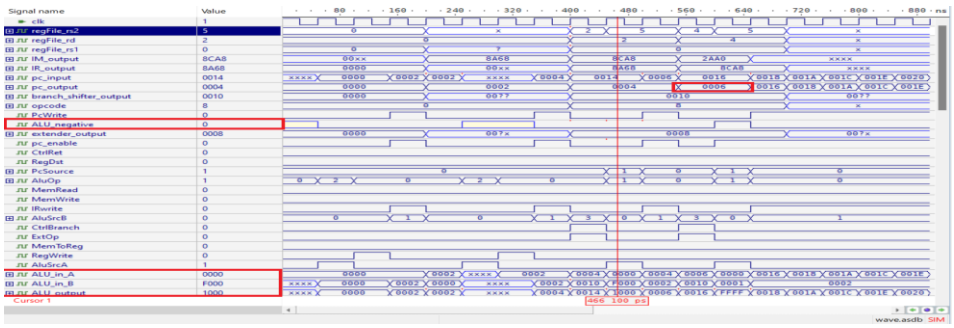

Figure 37: Execution Stage for BGTZ (non-Taken)

In the above figure we set 0xF000 value to Rd so when subtracting it with zero the flag negative will remain zero because we subtract 0 with negative value, So in this case we have non taken case.

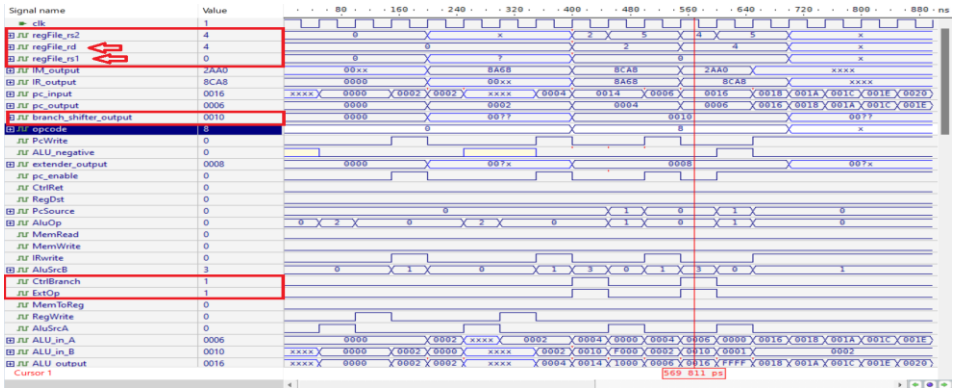

Figure 38: Decode Stage for BGTZ (Taken)

For this instruction we used R4 which equals 1 and it greater than 0 so it must be a taken.

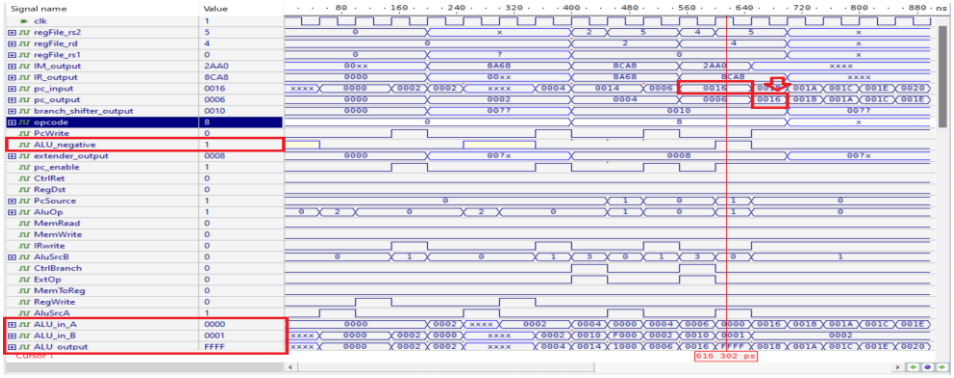

Figure 39: Execution Stage for BGTZ (Taken)

As shown above when we subtract 0 – 1 we got a negative flag, and the PC will branch to the new PC value and its value like the value that was used in BLT instruction.

2.5. CALL and RET Instructions program

```
register[0] <= 0;
register[1] <= 0;
register[2] <= {3'b000,5'b00110};
register[3] <= {CALL,1'b0,3'b000};
register[4] <= {R5,3'b01000};
register[5] <= {BGTZ,1'b1,R4};
register[6] <= {2'b10, R4,3'b000};
register[7] <= {SUB,R5,1'b0};
register[8] <= {3'b001, 5'b00001};
register[9] <= {RET,1'b0, 3'b010 };
register[10] <= {3'b001, 5'b00001};
register[11] <= {RET,1'b0, 3'b001 };
register[12] <=  {2'b10, R4,3'b000};;
register[13] <=  {SUB,R5,1'b0};
register[14] <= {3'b001, 5'b00001};
register[15] <=  {RET,1'b0, 3'b001 };
register[16] <=  {3'b001, 5'b00001};
register[17] <=  {RET,1'b0, 3'b001 };
register[18] <=  {3'b001, 5'b00001};
register[19] <=  {BGT,1'b0, 3'b001 };
register[20] <= 8'h64;
end
```

Figure 40: CALL and RET instructions

For this program the PC will start with CALL instruction then it will jump to 12 address which will perform the SUB then the next instruction will be RET instruction and this instruction will jump the saved address of R7.

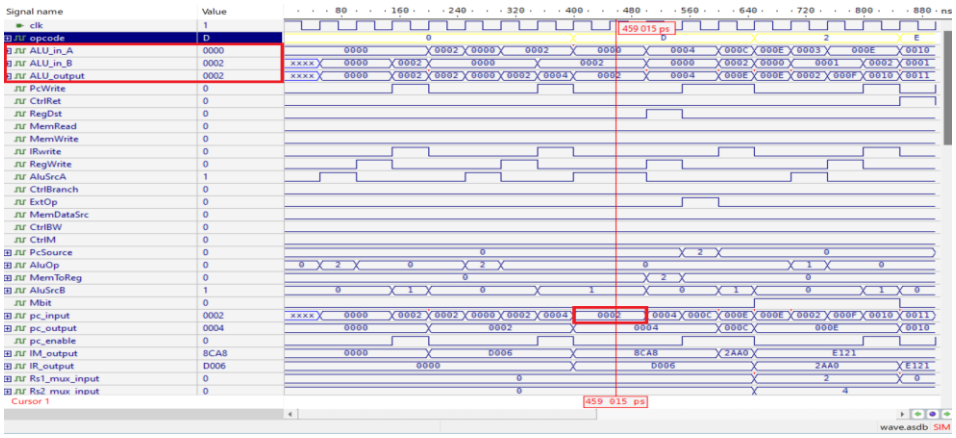

Figure 41: Call instruction with stage number 3

For this stage the program will compute PC + 2 and save it to R7 register on the next stage as shown above.

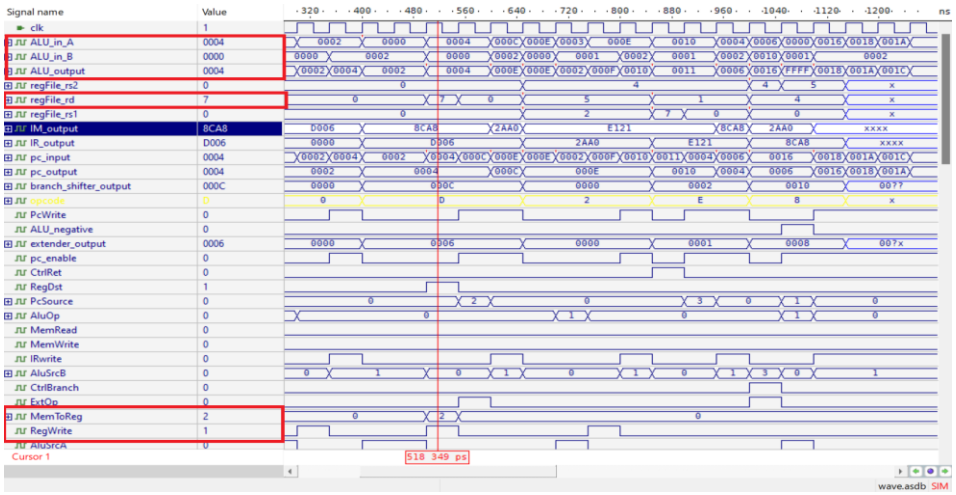

Figure 42: Call instruction with stage number 4

For this stage the program will write the ALU buffer output to R7.

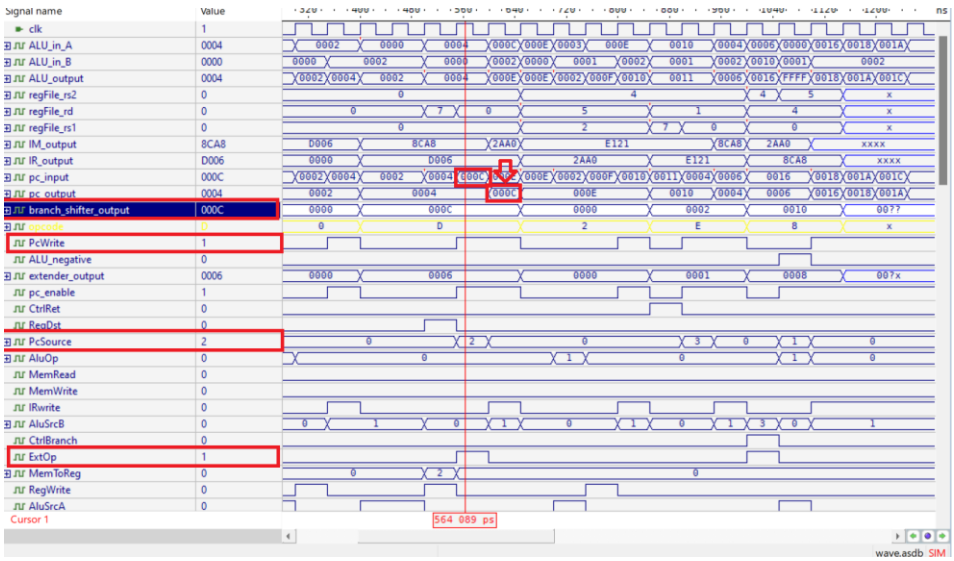

Figure 43: Call instruction with stage number 5

This stage is for jumping to the new PC address. And it equals to Next PC = {PC [15:13], Immediate}.

Our immediate equals 00110 with shift to the left it becomes 01100 and PC [15:13] equals 000 so our new PC will be 000C as seen in figure above.

As shown in figure above we jump to SUB address, and it address equals 12.

After we execute the Sub instruction, we will run the RET instruction as shown below.

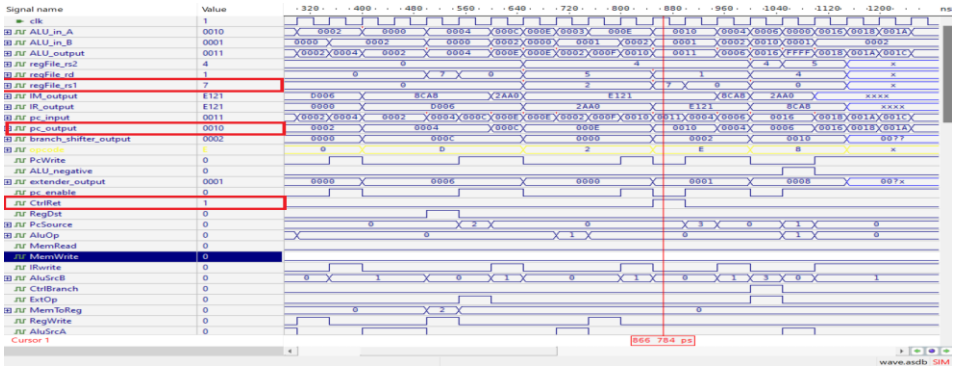

Figure 44: Stage 2 for RET instruction

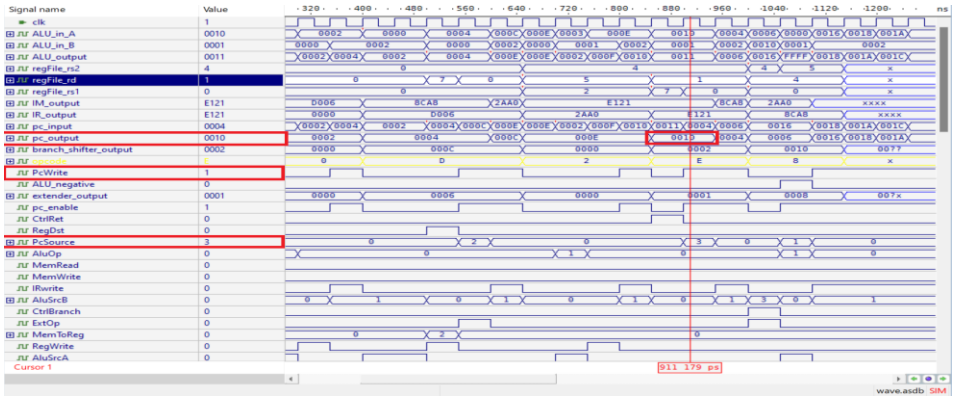

Figure 45: Stage 3 for RET instruction

As was illustrated before the RET instruction used for getting back from the CALL instruction plus 2.

As shown in figure 44 we compute the value of R7 to PC register by directing it to the mux of PCSource as shown in figure 45.

Then we jump to the new address which will be PC old plus 2 and it will be equal 0010 and we can see that the next instruction is BGTZ as shown in figure 40.

2.6. Jump and SV program

```
// ----------------------------  JUMP   programm --------------------------------//
  /*
  register[0] <= { 4'h2, 4'h1 } ;
  register[1] <= {JUMP, 4'h2} ;
  */

// ----------------------------  SV  programm --------------------------------//
  /*
// first instruction (ADD) R1 = R2 + R3 = 3 + 5 = 8
register[0] <= { 8'b10011000 } ;
register[1] <= {ADD, R1, 1'b0} ;
// second instruction (SV) MEM[R1] = MEM[8] = 17
register[2] <= { 8'b00100010 }  ;
register[3] <= {SV, R1, 1'b0} ;
```

Figure 46: jump and sv program

For the jump program as shown above, the target address must be {pc[15:13] = 000, 0x221 << 1}, thus, the next pc value must be 0x0441.

For the SV program, the value of Mem[R1] = Mem[8] must equal to IMM = 17

Jump simulation:

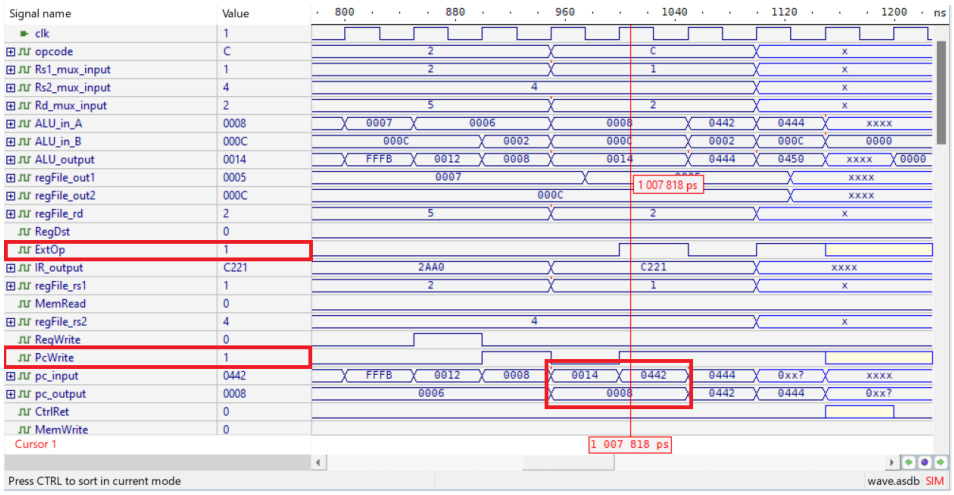

Figure 47: jump simulation part1

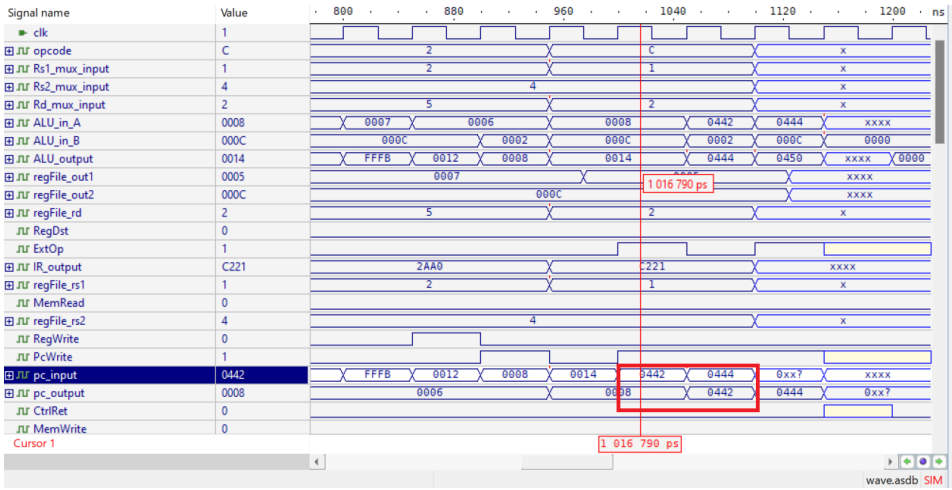

Figure 48: jump simulation part2

As we see above, the control signal pc write is high so that the pc value will be updated, in addition, the pc input value is 0x0442 which is the correct target address mentioned above, thus the program works well.

SV simulation:

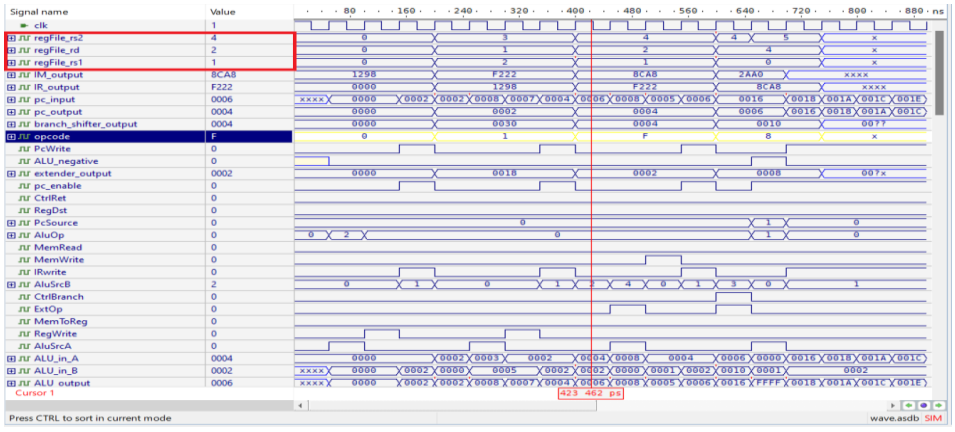

Figure 49: sv simulation part1

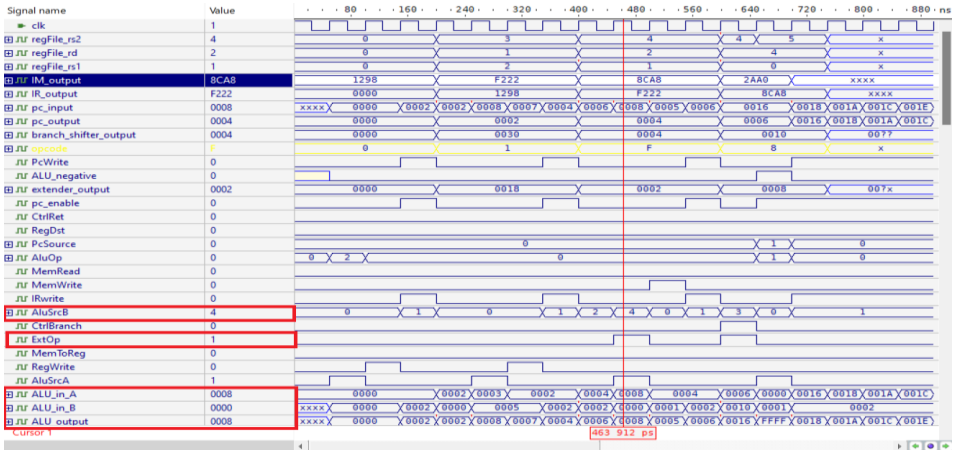

Figure 51: sv simulation part2

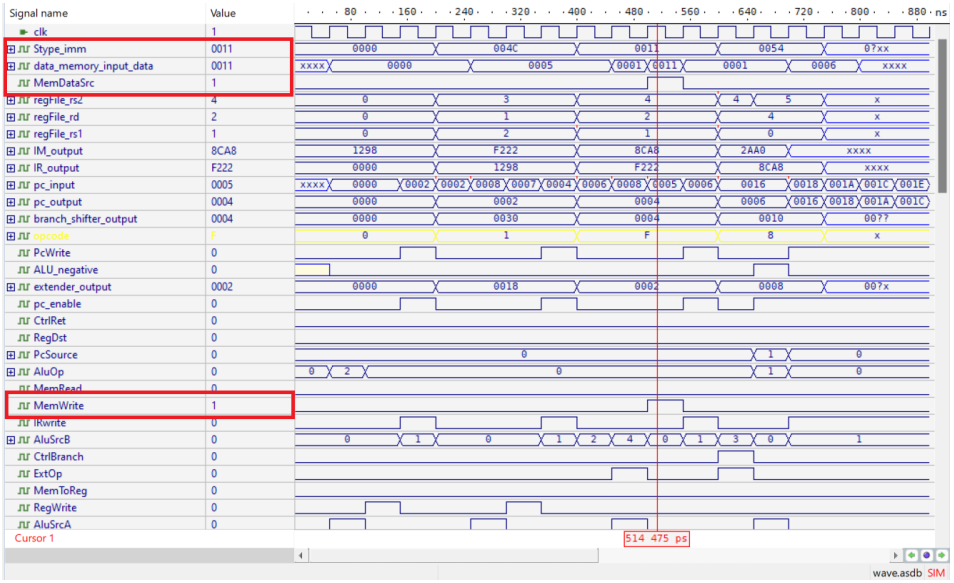

Figure 50: sv simulation part3

As we see above, the value of data memory input is 0x0011 which is 17 in decimal, and the ALU output is 0x0008 which represents the address of which the data is to be written, and the memory write signal is high. These results show that the program worked exactly as mentioned above, thus, it's correct.

3. Conclusion

In this project, we have successfully built a multi cycle five-stage RISC processor. The design process started by analyzing the given ISA and build a data path supports all instructions. We actually applied a modular design by building individual components and connect them together to form the data path. Additionally, we have built a control unit that represents a finite state machine which changes control signals every cycle to achieve a correct execution for each instruction, then we connected the control unit to the data path components to build the processor. Finally, we performed a design verification by writing programs contain all instructions inside the instruction memory and trace the processor waveform to check the correctness of our design. By performing RTL design, Verilog coding and design verification testing, this project improved our understanding in computer architecture principles and developed our skills in tackling complex engineering challenges.