Kuwait University
College of Engineering and Petroleum
Computer Engineering Department

CpE-Computer Organization
Semester: Spring
Section No. 01A

Project Assignment Part 3

Student name:

Hanan Ahmed ALSaleh - 2201129601

Asmaa Adel Alazmi - 2201122708

Instructor Name: Dr. Sami Hbib

TA Name: Eng. Zainab Bahbahani

Date: 13/Dec/2024

Table of Contents

Table of Figures

Tables

Phase 1:

I. Problem Statement

This project involves designing two essential components in digital systems: an Arithmetic Logic Unit (ALU) and a Register File. Using Verilog hardware description language and Quartus II software, the objectives are as follows:

1. ALU Design

Develop an 8-bit ALU with 4 operations (Addition, Subtraction, Logic AND, and Logic OR), controlled by a 4-bit control signal (InputControl). The ALU must take two 8-bit inputs, perform the specified operation, and produce an 8-bit output with flags for zero.

The following table showcases the possible arithmetic operations and their corresponding code:

Operation	Symbol	Binary code
Addition	add	0000
Subtract	sub	0010
Logic And	and	0110
Logic Or	or	0100

Table 1- Phase 1 ALU operations

2. Register File Design

Create a register file containing 8 registers of size 8 bits. The register file should be addressable using 3-bit addresses and allowing read and write operations, with two read ports and one write port. The following are the constraints for both the inputs & the outputs:

- **Read1** : input represents address of register source 1 (rs1) of size 3.
- **Read2**: input represents address of register source 2 (rs2) of size 3 bits.
- **Data1**: content of register source 1 of size 8 bits.
- **Data2**: content of register source 2 of size 8 bits.
- **Writereg** : input represent address of register destination (rd) of size 3 bits.
- **WriteData**: 8 bits data to be written on address register specified by *WriteReg*.
- **RegWrite**: control signal that is when equal to 1 the register file will allow 8 bits data at input *WriteReg* to be written to destination register address specified by *WriteReg*.
- **Register 0:** address 000 its value always zero and cannot be changed.

3. Objectives

a) Implement and test the ALU and register file designs in Verilog.

b) Simulate the designs and verify functionality using test cases provided.

c) Ensure accurate flag results for the ALU and correct data handling in the register file.

II. Results

1. ALU Simulation Results

The following table shows the expected output based on test inputs for the ALU operations:

inputs			Expected output		Actual output on waveform	
Input1	Input2	InputControl	Result	Zero flag	Result	Zero flag
0xFF	0xAA	0000	1A9	0	A9	0
0x35	0xAA	0110	20	0	20	0
0xFD	0x30	0010	CD	0	CD	0
0xEA	0xF1	0100	FB	0	FB	0
0x00	0x01	0000	01	0	01	0
0x00	0x01	0010	FFFF	0	FF	0
0xAA	0x55	0010	55	0	55	0
0xAA	0xFF	0110	AA	0	AA	0

Table 2 - Phase 1 ALU results

2. Register File Simulation Results

The test cases provided involve different register addresses and data values to test the register file. The results are recorded as follows (based on hexadecimal values). Note that for the file register both the expected value and the actual value match:

Read1	Read2	Writereg	RegWrite	WriteData	Data1	Data2
		101	1	0x35		
		111	1	0xE1		
		100	1	0xAA		
		010	1	0x55		
110					00	
100					AA	
	101					35
	010					55

Table 3 - Phase 1 Registe File results

III. Waveform

The waveform captures demonstrate the correct timing and functionality of the ALU and register file, validating the outputs against the provided input tables. The waveform results were used in the previous tables as the actual output values.

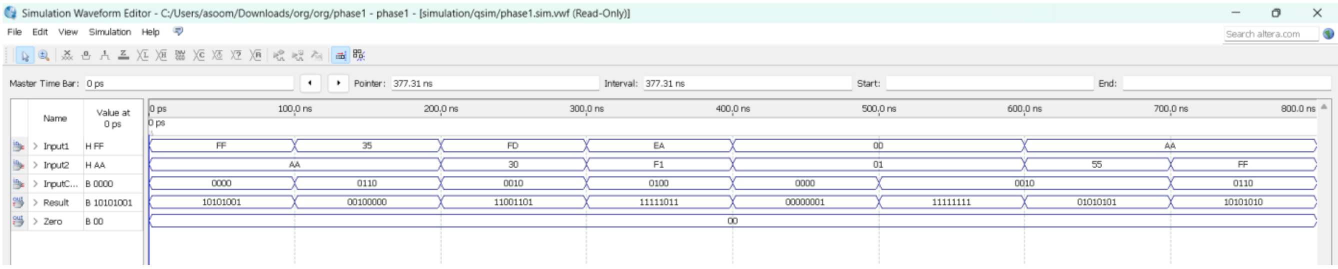


Figure 1 - ALU waveform

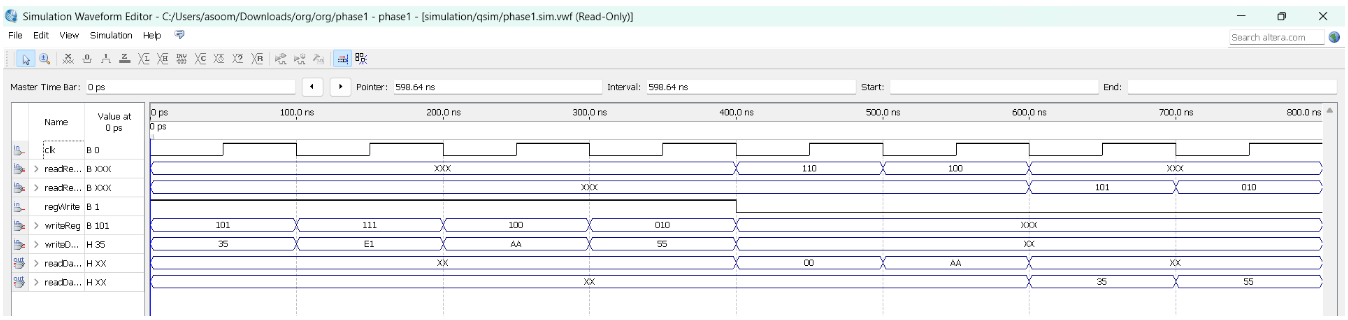


Figure 2 - Register File waveform

Phase 2:

I. Problem Statement

This project involves designing two essential components in digital systems: a read only memory (ROM) and a random-access memory (RAM). Using the block diagrams in Quartus II software, the objectives are as follows:

1. Read Only Memory (ROM)

Read Only memory is the memory used in the Data path to store the instructions named as Instruction memory IM. Design a Read Only Memory in block diagram schematic that has 8-bits address line and word size 16-bits in Quartus II 13.1 software

Design a Read Only Memory that has the following:

- The ROM has two inputs:
    1. Input1: 8-bit address line
    2. Input2: Clock
- Memory initialization file (mif) that includes a set of instructions (in this phase you will insert random data)
- Output1: 16-bit instruction(in this phase it will be random data)

- No control signal, new instruction is read every clock cycle.

2. Random Access Memory (RAM)

Random Access Memory (RAM) is a memory used in the datapath to store or load data accessed by the instruction known as Data Memory (DM).

Design a Random Access Memory that has the following inputs:

- Input1: 8-bits address line
- Input2: 8-bits data line to be written to the RAM
- Input3: Clock
- Memory Initialization File (mif) that contains the RAM initialized data (initialize it to zeros)
- Output1: 8-bits data to be read from the RAM
- Control Signals:
- ReadE: Control signal to enable reading from the RAM
- WriteE: Control Signal to enable writing to the RAM

3. Objectives

a) Implement and test the ROM and RAM using the block diagrams.

b) Simulate the designs and verify functionality using random test cases.

c) Ensure accurate flag results for the ROM and RAM by generating waveforms.

II. Results

- ROM

The following table shows the expected output based on test inputs for the ROM operations:

The values of input and output are in Hex

Address	Input	Output
0	AA	AA
1	01	01
2	FF	FF
3	00	00
4	67	67

Table 4 - Phase 2 ROM results

- RAM

The following table contains the sample data that was tested on the RAM component. Note that the clock is positive edge so the action either write or read will be done on the positive edge of the clock. The values for the fields Address, dataIn and DataOut are in Hex while REadE and WriteE are in binary.

Address	ReadE	WriteE	DataIn	DataOut
10	0	1	0A	xx
20	0	1	0B	xx
30	0	1	0C	xx
40	0	1	0D	xx
10	1	0	xx	0A
20	1	0	xx	0B
30	1	0	xx	0C
40	1	0	xx	0D

Table 5 - Phase 2 RAM results

III. Waveforms

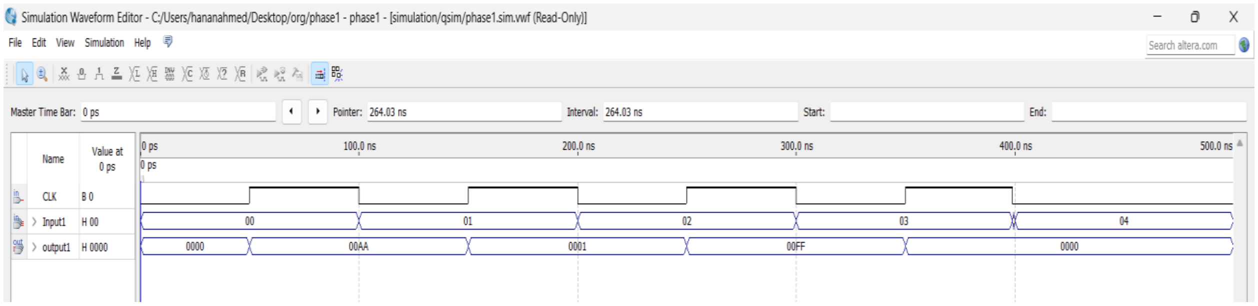


Figure 3 - ROM waveform

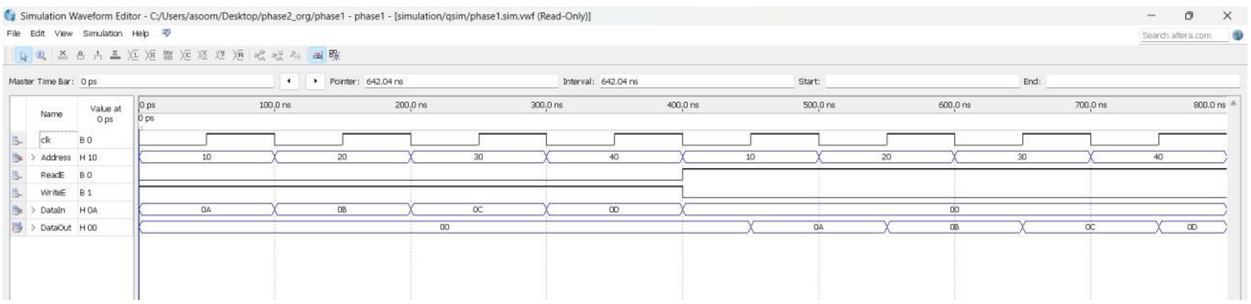


Figure 4 - RAM waveform

IV. Block Diagram (BDF)

- ROM

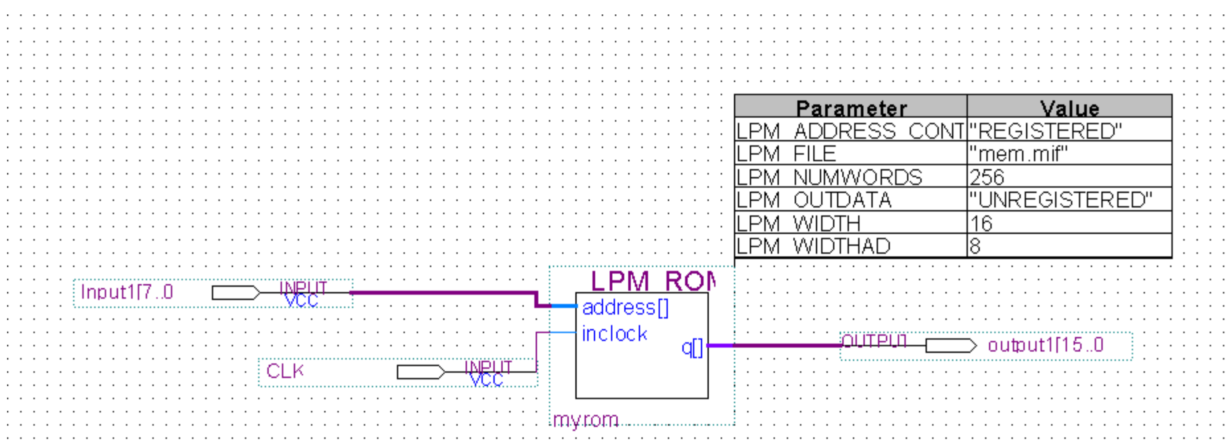


Figure 5 - ROM BDF

Figure 6 - Phase 2 ROM .mif

- **RAM**

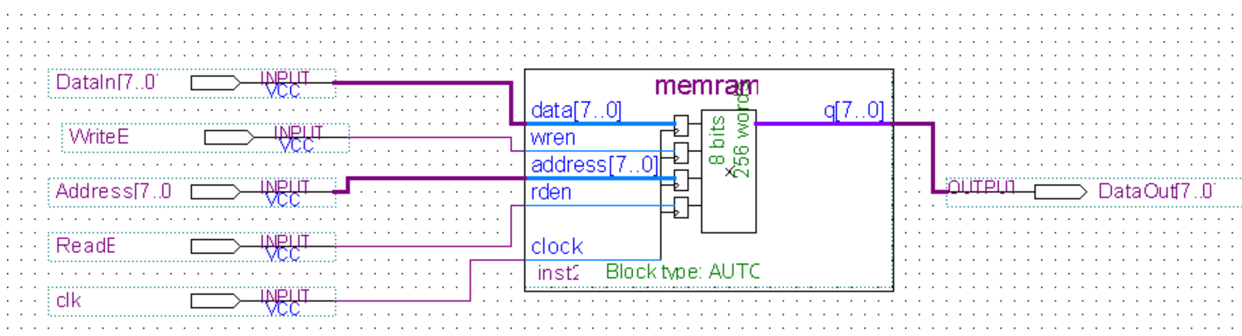


Figure 7 - RAM BDF

Phase 3:

I. Problem Statement

Design a datapath and control unit for the RISC-X architecture based on the provided specifications. RISC-X includes an instruction set (R-type, I-type, and S-type), a register file, instruction memory (ROM), and data memory (RAM). Customize the datapath and control unit to meet RISC-X requirements, such as removing the ALU control unit. Some extra components like MUXs and ADDERs will be required to be used to implement RISC-X. The datapath should handle incrementing the program counter (PC) such that the next instruction plays automatically, the value of the incrementation following the specified requirements should be PC+1 (PC + one byte). The value of the PC should be initialized as 0x00 at the start of each test. Note that the ALU from phase 1 should be updated to satisfy this phase's requirements.

The final design should output the following values:

4. PC address (8-bits) in HEX..
5. Current instruction (16-bits) in HEX.
6. The read values for both Rs1 and Rs2 in HEX.
7. The ALU result in HEX.
8. The ALU zero value in Binary.
9. The control unit signals (8 signals) in Binary.

And the only input required is the clock.

II. RISC-X Architecture

1. Instruction set

lb rd,x(rs1)	rd = M[x + rs1], and x is 3 bits positive number
sb rs2,x(rs1)	M[x + rs1] = rs2, and x is 8 bit positive number
add rd,rs1,rs2	rd = rs1 + rs2
addi rd, rs1, x	rd = rs1 + x , and x is 3 bits positive number
nand rd,rs1,rs2	rd = rs1 ~& rs2
nandi rd, rs1, x	rd = rs1 ~& x, and x is 3 bits positive number
bz rs1, rs2, label	If rs1 = rs2 then branch to PC = PC + label

2. Instruction format

R-type

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">func1</td><td colspan="3">rs2</td><td colspan="3">rs1</td><td colspan="3">rd</td><td colspan="4">opcode</td></tr>
</table>

The R-type includes add, nand and nor instructions

Operation	Opcode	Func1
add	0001	001
nand	0001	011

I-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
func1			imm[2..0]			rs1			rd			opcode			

The I-type includes addi, lb, sb and beq instruction

Operation	Opcode	Func1
addi	0011	001
Nandi	0011	011
lb	0010	001
bz	0101	001

S-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
func1			rs2			rs1			imm[2..0]			opcode			

The I-type includes sb instruction

Operation	Opcode	Func1
sb	0100	001

3. The system has a register file, which consists of 8 registers with size of 8bits each

   X0 = 0

   X1 to x14 multi-purpose registers

   The registers are addressed with three bits such that:

Register	Address
X0	000 Note always its value = 0
X1	001
X2	010
X3	011
X4	100
X5	101
X6	110
X7	111

4. Instruction memory (ROM) that has 8-bits address line and word size 16-bits, the ROM is word addressable, and the detailed specifications explained in detailed in phase II
5. Data Memory (RAM) 8 bits address line and 8 bits data line, the RAM is byte addressable, and the detailed specifications explained in detail in phase II.

III. Results

The following code snippet was tested:

```
addi x1, x0, 3
sb x1, 0(x0)
addi x2, x0, 2
sb x2, 3(x0)
add   x3,   x2,   x1
nand x4, x2, x1
```

The instructions were assigned to an address according to their order in the code. Starting form address 0x01 to 0x06. The instructions were first converted into an instruction format as follows:

Instruction	Instruction Format	
addi x1, x0, 3	001 011 000 001 0011	0x2C13
sb x1, 0(x0)	001 010 000 000 0100	0x2804
addi x2, x0, 2	001 010 000 010 0011	0x2823
sb x2, 3(x0)	001 010 000 011 0100	0x2834
add x3, x2, x1	001 001 010 011 0001	0x2531
nand x4, x2, x1	011 001 010 100 0001	0x6541

Table 6 - Phase 3 results instructions format

The instructions in their HEX format were then stored in the ROM memory which acts as the instruction memory for RISC-X.

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0000	2C13	2804	2823	2834	2531	6541	0000	....
8	0000	0000	0000	0000	0000	0000	0000	0000	....
16	0000	0000	0000	0000	0000	0000	0000	0000	....

Figure 8 - Phase 3 ROM .mif data

The results after running the previous code are:

Instruction code	PC	Inst. fetched	Read Rs1	Read Rs2	ALU result
addi x1, x0, 3	0x02	0x2C13	0x00	0x00	0x03
sb x1, 0(x0)	0x03	0x2804	0x00	0x00	0x00

addi x2, x0, 2	0x04	0x2823	0x00	0x00	0x02
sb x2, 3(x0)	0x05	0x2834	0x00	0x02	0x03
add x3, x2, x1	0x06	0x2531	0x02	0x03	0x05
nand x4, x2, x1	0x07	0x6541	0x02	0x03	0xFD

Table 7 - Phase 3 reuslts

Note that the PC points to the next instruction not the current one. The waveform section below includes more detailed results (control signals).

IV. Waveform

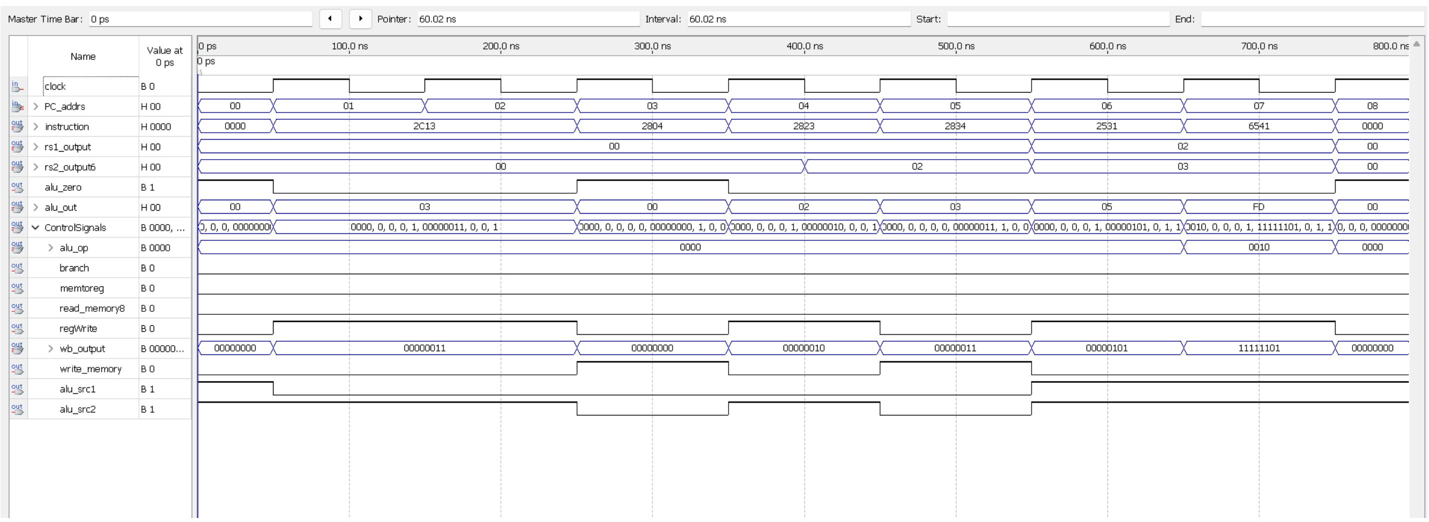
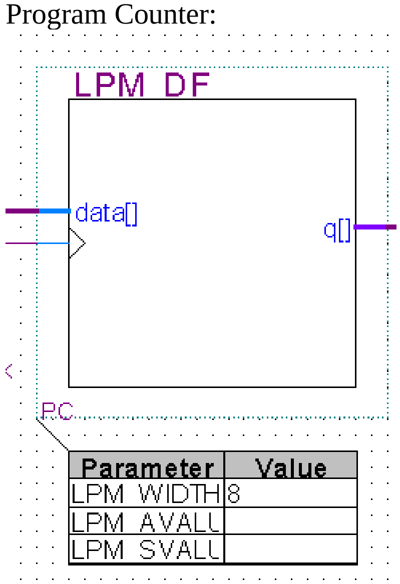


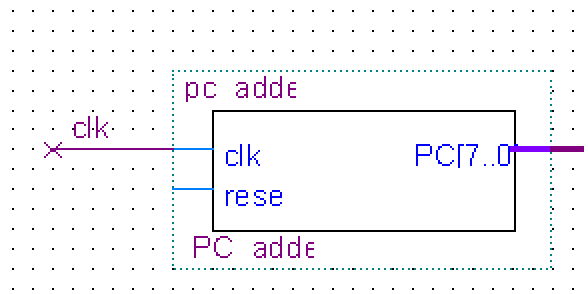
Figure 9 - Phase 3 waveform

V. Required Components

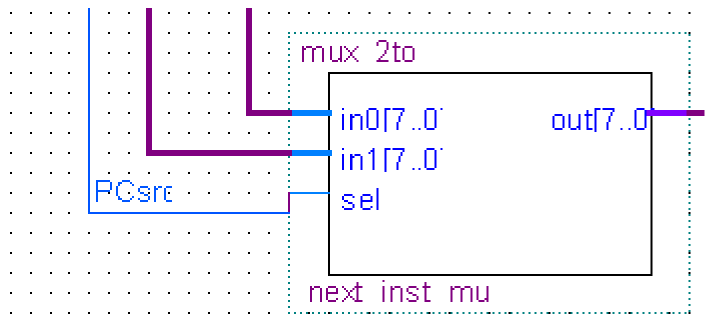
1. Program Counter:



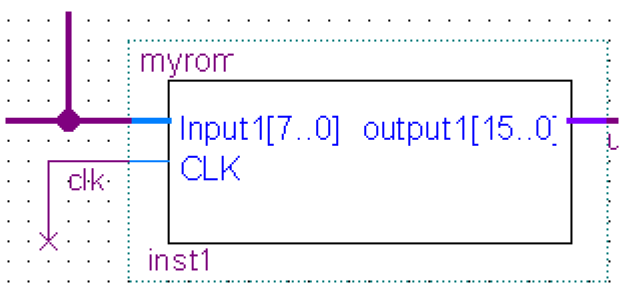
2. PC Adder: This adder initializes the PC value to 0x00 when the program first runs then increments the PC value by one byte. It also has two inputs, a clock and a reset input to reset the PC to 0x00.



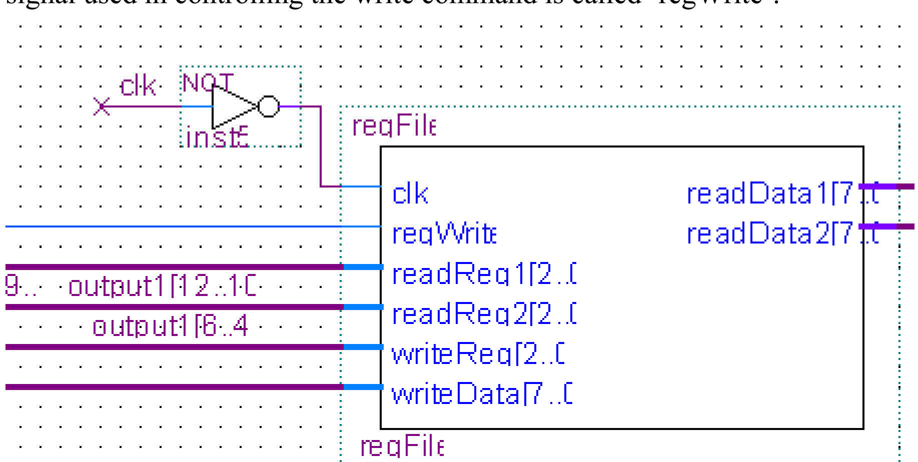
3. PC source MUX: This mux takes two inputs, PC+1 and PC+JUMP and the selection line is connected to the control unit. . The control signal is called ‘PCsrc.



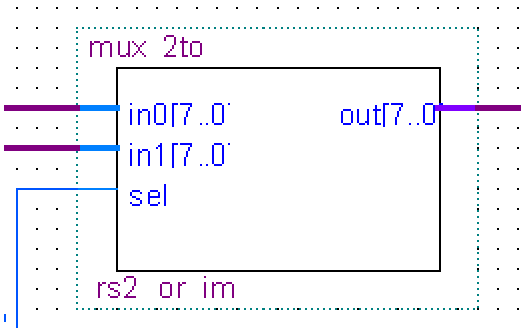
4. ROM: this unit acts as the instruction memory; it fetches the instruction after it takes its address from the PC unit.



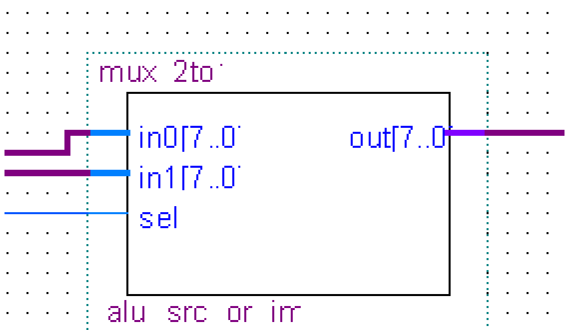
5. Register File unit: This unit takes 3 addresses (Rs1, Rs2, Rd) from the instruction format forwarded by the ROM and reads the registers Rs1 and Rs2 and outputs their value. It also updates the value of the register Rd using data forwarded by either the ALU or the RAM. Note that the register file needs a ‘not’ gate on the clock such that it operates on the negative edge of the clock. . The control signal used in controlling the write command is called ‘regWrite’.



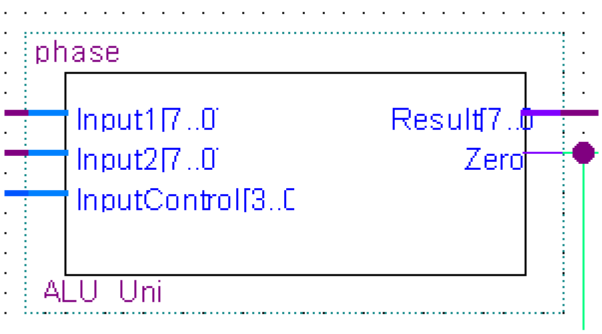
6. Rs2 or I-Type immediate MUX: This mux takes two inputs Rs2 and an immediate value. Depending on the instruction used, the control unit will send data to the selection line of this mux and the output will be used in the next mux. . The control signal is called ‘ALUsrc’.



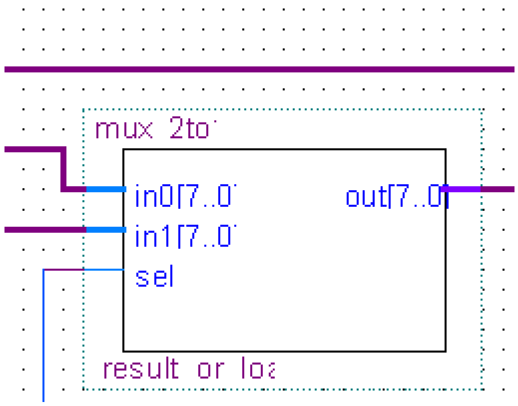
7. Rs2 or S-Type immediate MUX: This mux takes two inputs an S-Type immediate value and the result from the previous mux. Depending on the selection line that is controlled by the control unit it will output one of them straight to the ALU unit. The control signal is called ‘ALUsrc2’.



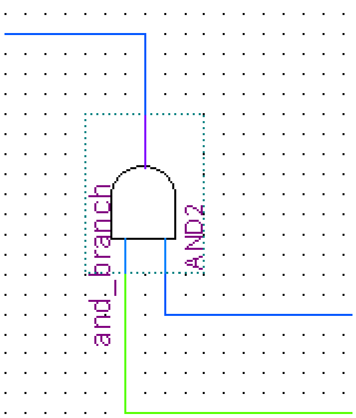
8. ALU unit: This unit does all the arithmetic operations, it takes two inputs one is always Rs1 and the other is the result of a mux. This unit outputs a 8-bit result and forwards in to the next mux and a one bit ALUzero result.



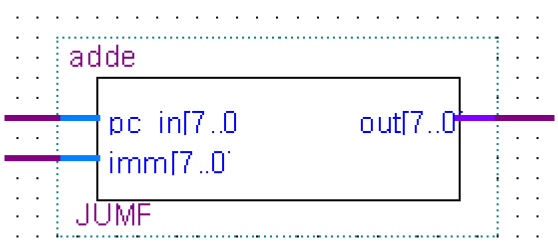
9. Result or memory MUX: This mux controls whether the Rd value stores the result from the ALU or loads a value from the memory. It takes two inputs the ALU result, and the data loaded from the memory. The selection line is controlled by the signal ‘memtoreg’.



10. Branch AND gate: This gate controls whether the branch id permissible or not. It takes two inputs one is the control signal ‘Branch’ and the other is the ALUzero result.



11. Jump ADDER: This adder is responsible for incrementing the PC by an immediate jump value for the branch instruction.



VI. Custom Control Unit

The following control unit is responsible for deciding which route the datapath takes depending on the instruction taken from the ROM. This unit analyzes the opcode and fun1 provided in the 16-bit instruction, then it outputs the eight control signals accordingly. The eight signals are as following:

1. **branch** (1-bit): if 1 then branch.
2. **regWrite** (1-bit): if 1 then write to rd.
3. **memtoreg** (1-bit): if 0 then write to rd using the ALU result else use the loaded data.
4. **alu_src** (1-bit): if 1 then use Rs2 value else use the immediate value (I-Type MUX).
5. **write_memory** (1-bit): if 1 then write to memory.

6. **read_memory** (1-bit): if 1 then load from memory.
7. **alu_src2** (1-bit): if 1 then use the I-Type MUX result, else use the immediate value of the S-Type instruction.
8. **alu_op** (4-bits):
    a. If 0000 then ADD operation.
    b. If 0010 then NAND operation.
    c. If 0100 then BZ operation (check for zero by subtracting zero from Rs1).
    d. If 1000 then AND operation.

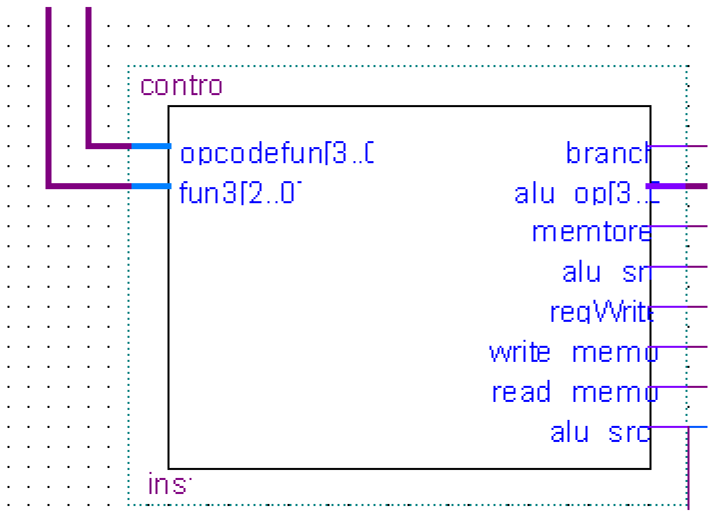


Figure 10 - Control Unit BDF

VII. Block Diagram (BDF)

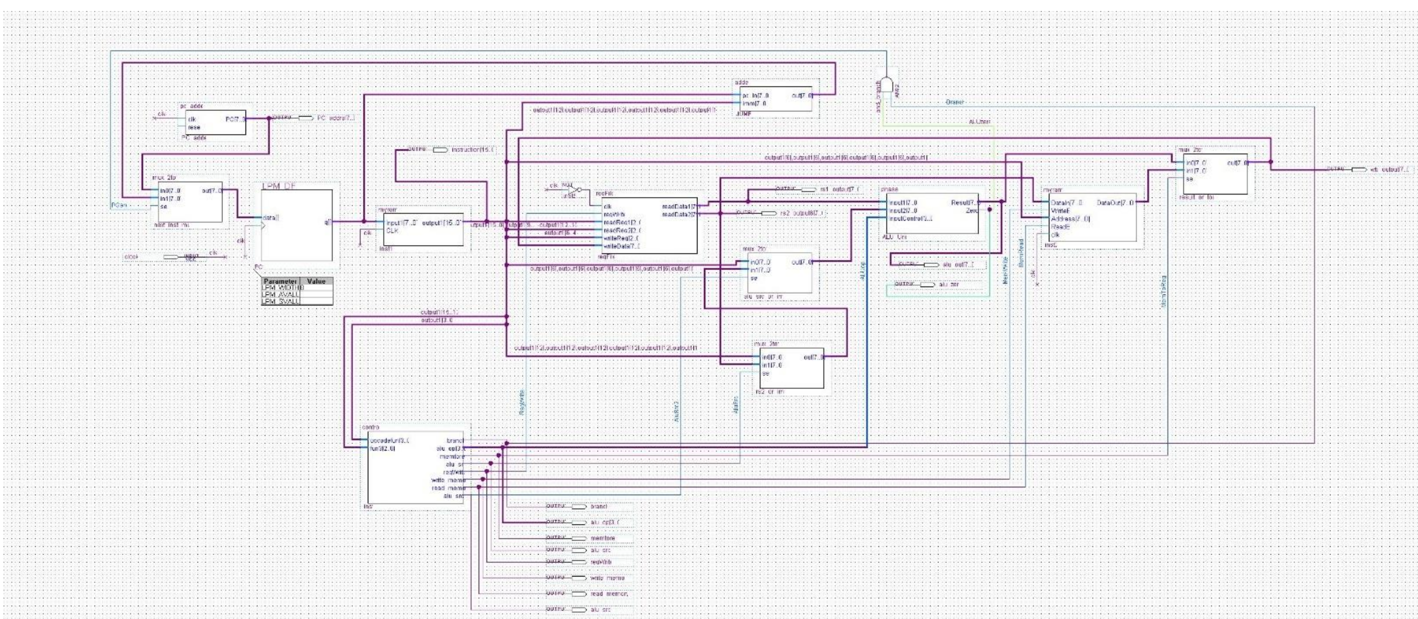


Figure 11- Phase 3 BDF

VIII. Conclusion

In “designing RISC-X” we noticed that the project was helpful in understanding how to create a data path. It taught us to write Verilog code to exist ALU, Register File, and Control Unit. Using Quartus, we successfully built a system capable of executing key instructions such as arithmetic operations, logical operations, and memory access operations, each component, including the ALU, register file, instruction memory, and data memory, was carefully designed and integrated to meet the RISC-X specifications. The system is capable of performing basic operations like addition, NAND, and memory access. This project helped us understand how components like the ALU, register file, and control unit work together to execute instructions.

IX. Verilog Code

1. **ALU unit:**

```
module phase1(Input1,Input2,InputControl,Result,Zero);
   input [7:0] Input1;
   input [7:0] Input2;
   input [3:0] InputControl;
   output reg [7:0] Result;
   output reg Zero;

   always @(Input1,Input2,InputControl) begin
      case (InputControl)

         0: Result = Input1 + Input2;        // Addition
         2: Result =  ~(Input1 & Input2);        // NAND
                                                 4: Result = Input1 - 8'b00000000;        // branch zero
                                                 6: Result = Input1 & Input2;        // AND operation

      endcase

                      if (Result == 8'b0)
   Zero <= 1'b1;  // This line doesn't end with a semicolon because the else follows
else
   Zero <= 1'b0;
   end
endmodule
```

2. **Register File Unit:**

```
module regFile (
   input wire clk,                  // Clock signal
   input wire regWrite,             // Write enable signal (Control signal)
   input wire [2:0] readReg1,        // Address of the first register to read
   input wire [2:0] readReg2,        // Address of the second register to read
   input wire [2:0] writeReg,        // Address of the register to write to
   input wire [7:0] writeData,       // Data to write into the register

          output wire [7:0] readData1,      // Data read from the first register
```

```
    output wire [7:0] readData2      // Data read from the second register
);

    // Define the register array: 8 registers, each 8 bits wide
    reg [7:0] registers [7:0];


    // Initialize register 0 to always hold 0 assuming reg 0 is also 8 bits
    initial begin
        registers[0] = 8'b0;
    end

    // Read the data from the registers
    assign readData1 = registers[readReg1];
    assign readData2 = registers[readReg2];

    // Write data to the register on the falling edge of the clock
    always @(posedge clk) begin
        if (regWrite && (writeReg != 0)) begin
            registers[writeReg] <= writeData;  // Write only if regWrite is enabled and writeReg is not 0
        end

                        //reset reg 0 to 0
        registers[0] <= 8'b0;

    end
endmodule
```

3. Control Unit:

```
module control (
    input [3:0] opcodefun,
    input [2:0] fun3,
    output reg branch,
    output reg [3:0] alu_op,
    output reg memtoreg,
    output reg alu_src,
    output reg regWrite,
    output reg write_memory,
    output reg read_memory,
            output reg alu_src2
);
    // Control logic here
    always @(*) begin
        // Initialize control signals
        alu_op = 3'b000;
        branch = 1'b0;
        memtoreg = 1'b0;
        alu_src = 1'b1;
        regWrite = 1'b0;
        write_memory = 1'b0;
        read_memory = 1'b0;
```

```
                    alu_src2 = 1'b1;

    // R-TYPE INSTRUCTIONS ------------
    if (opcodefun == 4'b0001 && fun3 == 3'b001) begin // ADD
      alu_op = 3'b000;
      regWrite = 1'b1;
    end

    if (opcodefun == 4'b0001 && fun3 == 3'b011) begin // NAND
      alu_op = 3'b010;
      regWrite = 1'b1;
    end

    // I-TYPE INSTRUCTIONS ------------
    if (opcodefun == 4'b0011 && fun3 == 3'b001) begin // ADDI
      alu_op = 4'b0000;
      regWrite = 1'b1;
      alu_src = 1'b0;
    end

    if (opcodefun == 4'b0011 && fun3 == 3'b011) begin // NANDI
      alu_op = 4'b0010;
      regWrite = 1'b1;
      alu_src = 1'b0;
    end

    if (opcodefun == 4'b0010 && fun3 == 3'b001) begin // LOAD BYTE
      alu_op = 4'b0000;
      regWrite = 1'b1;
      alu_src = 1'b0;
      read_memory = 1'b1;
    end

    if (opcodefun == 4'b0101 && fun3 == 3'b001) begin // BRANCH ZERO
      alu_op = 4'b0100;
      branch = 1'b1;
    end

    // S-TYPE INSTRUCTIONS ------------
    if (opcodefun == 4'b0100 && fun3 == 3'b001) begin // STORE BYTE
      alu_op = 4'b0000;
      write_memory = 1'b1;
      alu_src = 1'b0;
      memtoreg = 1'b0;
                              alu_src2 = 1'b0;
    end
  end
endmodule
```

4. 2 to 1 MUX:

```
module mux_2to1 (
```

```
    input [7:0] in0,     // 8-bit input 0
    input [7:0] in1,     // 8-bit input 1
    input sel,           // 1-bit select signal
    output [7:0] out     // 8-bit output
);

    assign out = (sel) ? in1 : in0; // If sel=1, select in1; otherwise, select in0

endmodule
```

5. **PC ADDER:**

```
module pc_adder(
    input wire clk,         // Clock signal
    input wire reset,       // Reset signal
    output reg [7:0] PC     // 8-bit Program Counter output
);

    // Initialize PC
    initial begin
        PC = 8'h00;         // Start PC at 0x00
    end

    // PC Update Logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            PC <= 8'h00;     // Reset PC to 0x00
        end else begin
            PC <= PC + 8'h01; // Increment PC by 1 byte (8 bits)
        end
    end

endmodule
```

6. **JUMP ADDER:**

```
module adder (
    input wire [7:0] pc_in,   // 8-bit Program Counter input
    input wire [7:0] imm,     // 8-bit immediate value
    output wire [7:0] out
);
    assign out = pc_in + imm;
endmodule
```

X. References

[1] Patterson, D. A., & Hennessy, J. L. (1998). Computer organization & design: The hardware/software interface (2nd ed.). Morgan Kaufmann.