Department of Electrical and Computer Engineering
Air University Islamabad
6th Semester

Digital System Design
Complex Engineering Activity
Instructors: Engr. Usman Hameed

Osama Anees Mirza 210286
Rayyan Munir 210301
Zafir Asad 210299

26 May 2024

Contents

List of Tables

List of Figures

1 Objective:

The aim of this intricate engineering activity is to introduce students to the concepts and processes involved in designing and implementing a datapath using Verilog, a hardware description language. A datapath is a crucial component of a computer's central processing unit (CPU) responsible for performing arithmetic and logical operations on data. The objective is to provide students with practical experience and a deep understanding of the inner workings of a CPU's datapath. This comprehensive engineering activity encompasses the following aspects:

- Familiarizing students with the principles and theory behind datapath design.
- Hands-on experience in implementing a datapath using Verilog.
- Gaining insight into the various stages and components of a CPU's datapath.
- Understanding the interaction and flow of data within the datapath.
- Practicing the design and implementation of arithmetic and logical operations.
- Exploring different control mechanisms and their integration with the datapath.

2 Resources Required

- Xilinx ISE Suite
- iverilog (Icarus Verilog)
- VS-Code
- A Computer (Not a broken one)

3 Problem Statement:

We were tasked with designing a complete data path of any RISC V Implementation. The datapath should consist of the following components:

1. Register File
2. Arithmetic Logic Unit
3. Control Unit
4. Instruction Memory
5. Program Counter
6. Data Memory
7. Instruction decoder

4 Introduction:

In this task we are going to design a 16 bit single cycle RICS core which supports MIPS like instructions because its instruction set is very easy to understand + its extendable meaning if we want to add more instructions like multiply or divide or jump instruction we can easily do that. Right now we just need to execute R-type, I-type and J-type instruction because we are trying to make our design perform everything that a RISC SPM can.

5 The Format Of Our Instructions:

Before proceeding further we need to understand the format of our instruction. We are using 16 bits and only require to execute Register-type, Immediate-type and Jump-type instructions. The following table explains our instruction set:

5.1 R-Type Instruction:

Register Type Instruction					
Instruction	**op**	**rs**	**rt**	**rd**	**function**
add	000	3 bit	3bit	3bit	0000
sub	000	3 bit	3bit	3bit	0001
and	000	3 bit	3bit	3bit	0010
or	000	3 bit	3bit	3bit	0011

5.2 I-Type Instruction:

Immediate Type Instruction				
Instruction	**op**	**rs**	**rt**	**Shift Amount/Function**
lw	001	3 bit	3bit	0000000
sw	010	3 bit	3bit	0000000
addi	100	3 bit	3bit	0000000

5.3 J-Type Instruction:

Jump Type Instruction		
Instruction	**op**	**Instruction memory Address**
j	011	13 bit

These tables are modified versions of [2].

6 Procedure

6.1 Arithmetic Logic Unit (ALU)

6.1.1 Introduction:

We will design an ALU that can perform a subset of the ALU operations of a full Processor ALU.

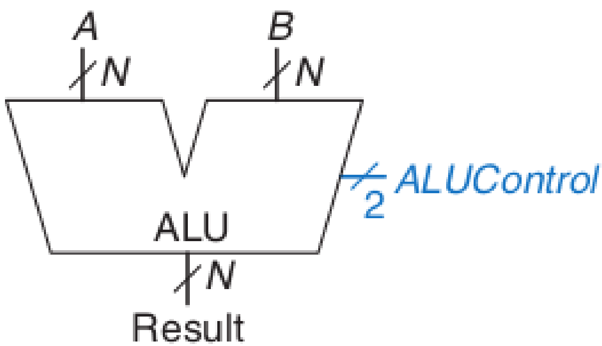


Figure 1: ALU [1]

In this exercise, we will develop an ALU that will take two 2-inputs, A and B and is able to execute the following instructions:

ALU Control	Instruction
000 (add)	lw,sw
000 (add)	add
001 (add)	sub
010 (and)	and
011 (or)	or

The ALU will generate a 16-bit output that we will call 'Result' and an additional 1-bit flag 'Zero' that will be set to 'logic-1' if all the bits of 'Result' are 0. The different operations will be selected by a 3-bit control signal called

'ALUControl' according to the following table.

For example, when the 'ALUControl' input is '011', the function Result = A or B should be calculated. It is easy to see that there are many values of 'ALUControl' for which no operation has been defined. It is not very important what the circuit does when 'ALUControl' has these values, since the 'Result' will simply be ignored in these cases. You can use this to your advantage to simplify the circuit. Right now, the described operations may look random, but once we learn more about the Instruction set architecture, these choices will make more sense.

6.1.2 Block Diagram:

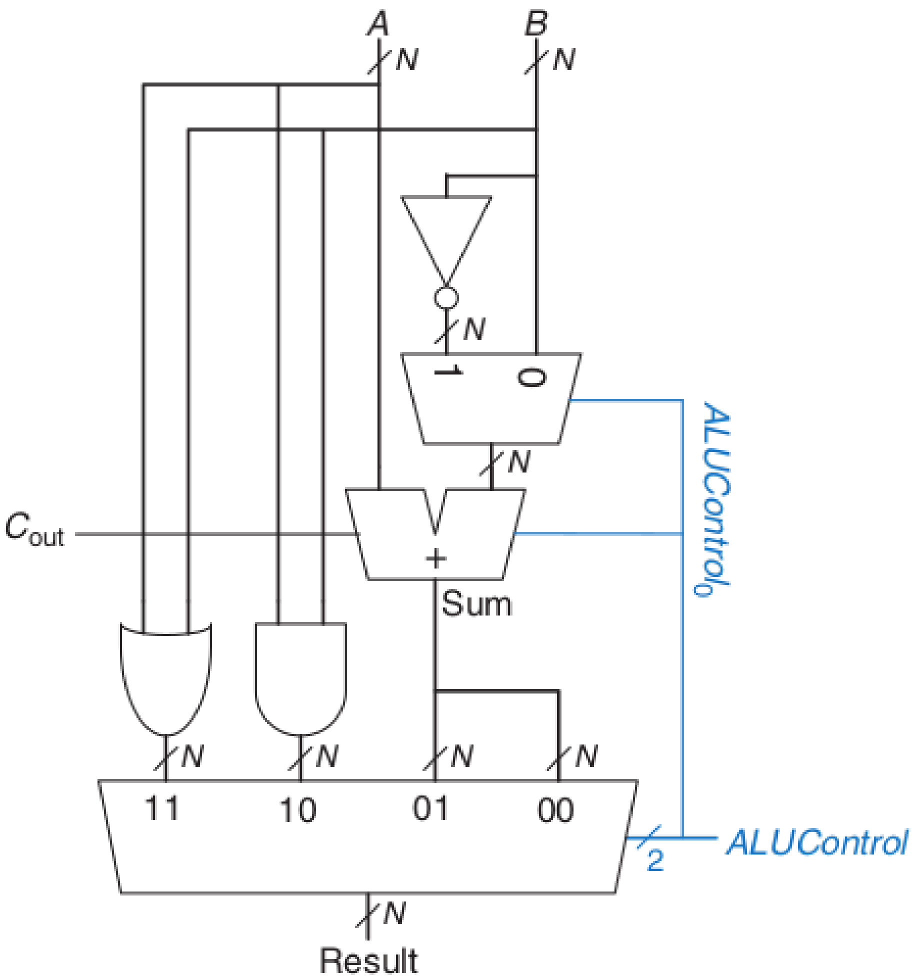


Figure 2: Where N = 16 [1].

6.2 Control Unit

6.2.1 Introduction:

RISC-V consists of defining the following instruction formats: R-type, I-type, S-Type, B-Type, U-type, and J-type. R-type instructions operate on three registers. I-type, S-type and B-type instructions operate on two registers and a 12-bit immediate. U-type and J-type (jump) instructions operate on one 20-bit immediate. Here our only concern is of R-type and I-type.

Table 1: Control Unit Flags

Instruction Type	op flag(input)	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	ALUOp(2 bits)	Jump_EN
R-Type	000	1	0	0	1	0	0	00	0
lw	001	0	1	1	1	1	0	11	0
sw	010	0	1	0	0	1	1	11	0
j	011	0	0	0	0	0	0	00	1

6.2.2 Control Unit Flags:

6.2.3 Block Diagram:

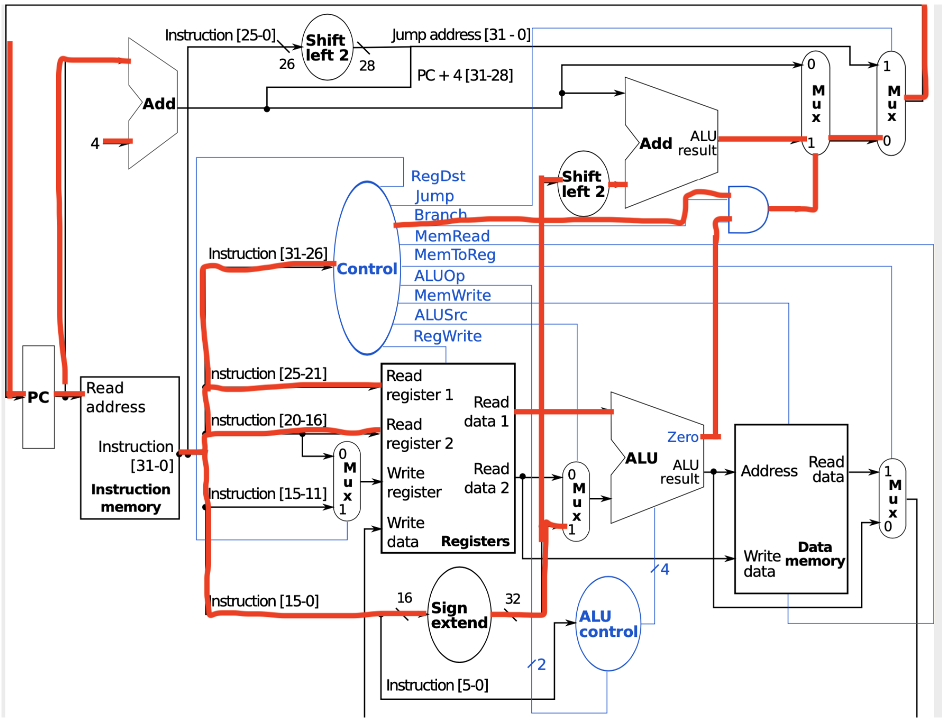


Figure 3: Control Unit Path.
Ignore The Non R-Type Flags.

6.3 ALU Control

The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal. The ALU decoder uses this ALUOp signal in conjunction with the funct field and opcode bit to compute ALUControl. The meaning of the ALUOp signal is given in Table below:

6.3.1 ALUOp Table:

ALUOp	Meaning
00	add
01	subtract
10	look at funct fields and opcode bit
11	N/A

Table below is a truth table for the ALU decoder. The logic of ALUControl was covered in the above chapter. When ALUOp is 00 or 01, the ALU should add or subtract, respectively. When ALUOp is 10, the decoder examines the function fields and operand bit to determine the ALUControl. The control signals for each instruction were described as we built the datapath.

6.4 Memory Elements

These include:

1. Program Counter

2. Instruction Memory
3. Register File
4. Data Memory

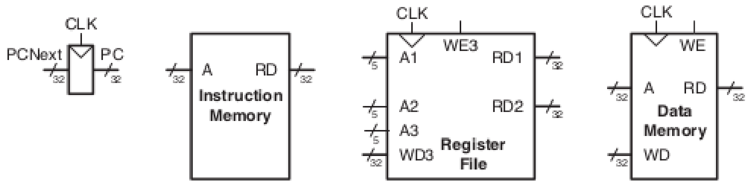


Figure 4: Memory Elements [1]

6.4.1 Program Counter:

The Program Counter (PC) is a fundamental component in computer architecture responsible for storing the memory address of the next instruction to be executed. In the case of a 16-bit Program Counter, it can store a 16-bit memory address, allowing it to access up to 65,536 memory locations.

The Program Counter works in conjunction with the Instruction Fetch stage of the processor's pipeline. During each clock cycle, the PC increments by one to point to the next memory location. It fetches the instruction stored at that memory address and sends it to the Instruction Decoder for further processing.

When an instruction requires a branch or jump, the Program Counter is modified to point to the new memory address indicated by the branch instruction. This allows the processor to change the sequence of instruction execution and alter the program flow.

The Program Counter is a vital component for sequential instruction execution, ensuring the processor fetches and executes instructions in the correct order. It plays a crucial role in maintaining program flow and enabling the processor to follow the control flow of the program being executed.

6.4.2 Instruction Memory:

The Instruction Memory is a crucial component in computer architecture responsible for storing the program instructions that the processor executes. In the case of a 16-bit Instruction Memory, it can store and provide access to 16-bit instructions.

The Instruction Memory works in conjunction with the Program Counter (PC) and the Instruction Decoder. The PC holds the memory address of the next instruction to be fetched, and the Instruction Memory fetches the instruction from the corresponding memory location.

The Instruction Memory operates by receiving the memory address from the PC and retrieving the instruction stored at that address. It then sends the instruction to the Instruction Decoder, which interprets the instruction and initiates the necessary operations.

The Instruction Memory is typically implemented as a read-only memory (ROM) or as a portion of the main memory in a computer system. It is initialized with the program instructions prior to program execution and remains constant during runtime.

The Instruction Memory's primary function is to provide the processor with the instructions needed to execute a program. It ensures that the processor fetches the correct instructions in the correct order, enabling the execution of complex programs and algorithms.

6.4.3 Register File

The Register File is a component that stores and provides access to a set of 16-bit general-purpose registers in a computer architecture. It allows the processor to quickly read and write data during instruction execution, improving performance and efficiency.

6.4.4 Data Memory

The data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

The instruction memory, register file, and data memory are all read combinationally. In other words, if the address changes, the new data appears at RD after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must be set up sometime before the clock edge and must remain stable until a hold time after the clock edge. Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits.

7 Verilog Implementation

We used Xilinx along with Icarus Verilog and VS-Code to code, test and make its RTL. We have separated all the individual components into separate files as it is much easier to detect errors and maintain the code base.

7.1 Program Counter

7.1.1 RTL Diagram

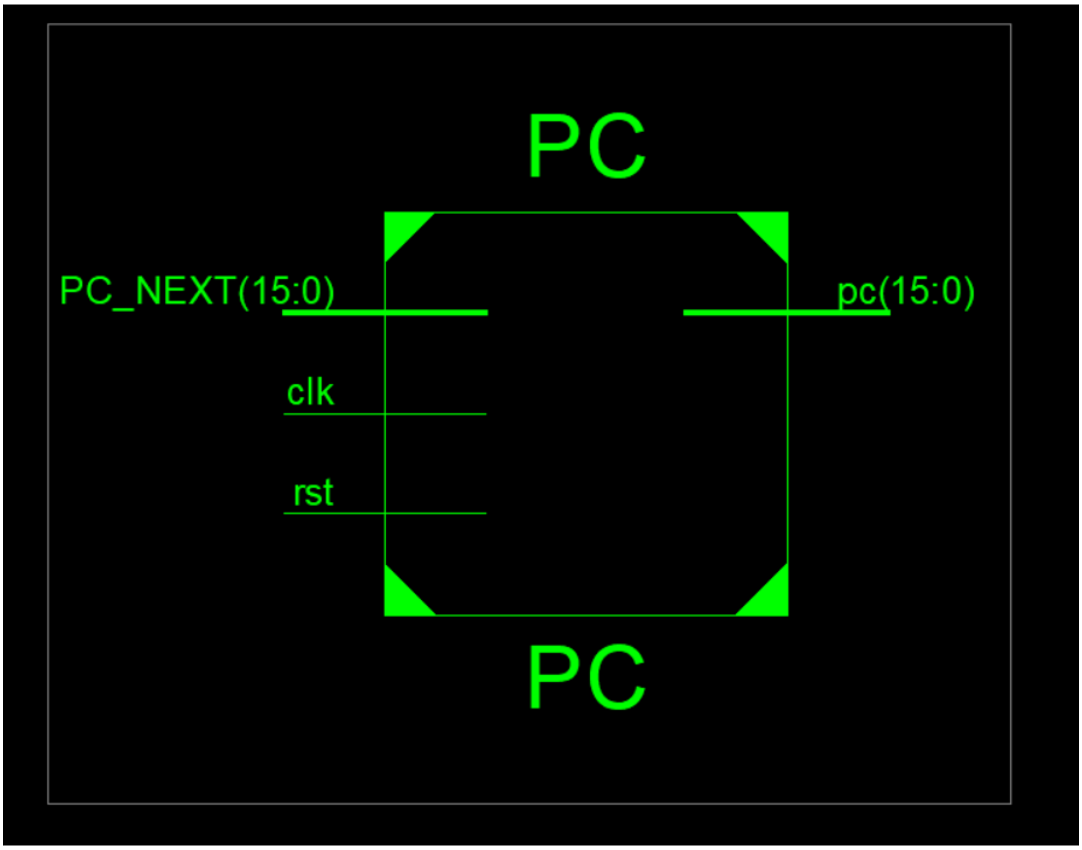


Figure 5: Program Counter Block Diagram

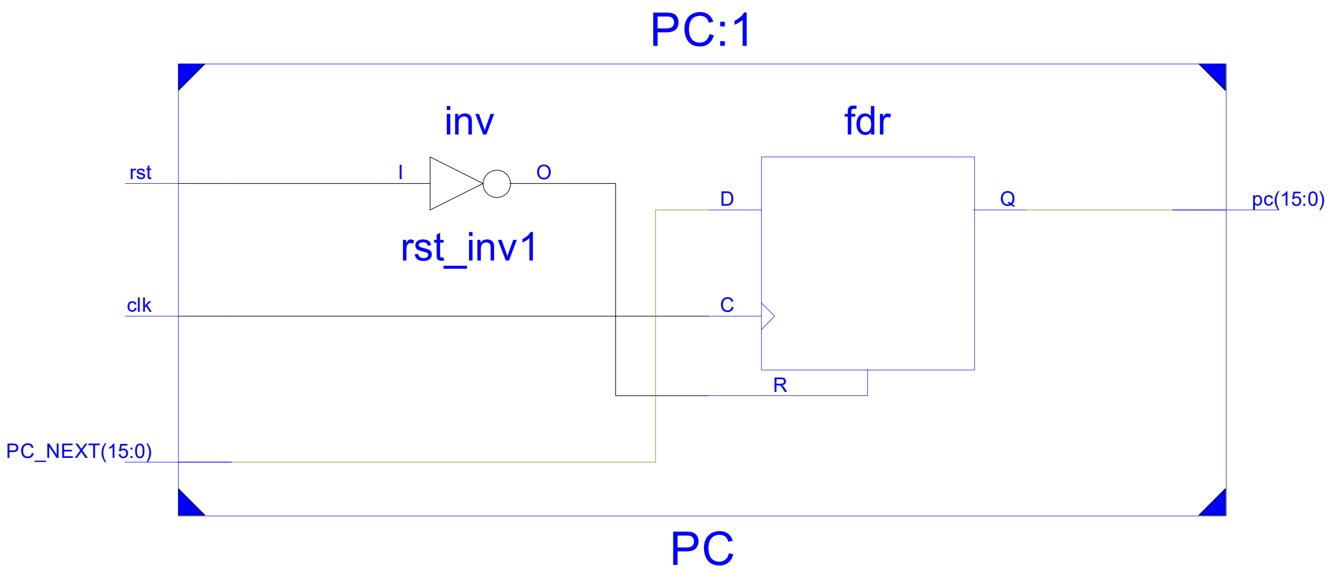


Figure 6: Program Counter RTL

Code:

```
module PC(PC_NEXT,pc,rst,clk);
    //Declaring Inputs:
    input [15:0] PC_NEXT;
    input clk,rst;

    //Declaring Outputs:
    output reg [15:0] pc;
    always @(posedge clk) begin //On every Positive Edge of the clock do:
        if (rst == 1'b0)// Active Low Values.
            begin
                pc <= 16'b0000000000000000;
            end
        else
            begin
                pc <= PC_NEXT;
            end
    end
endmodule
```

7.2 Program Counter Adder

7.2.1 RTL Diagram

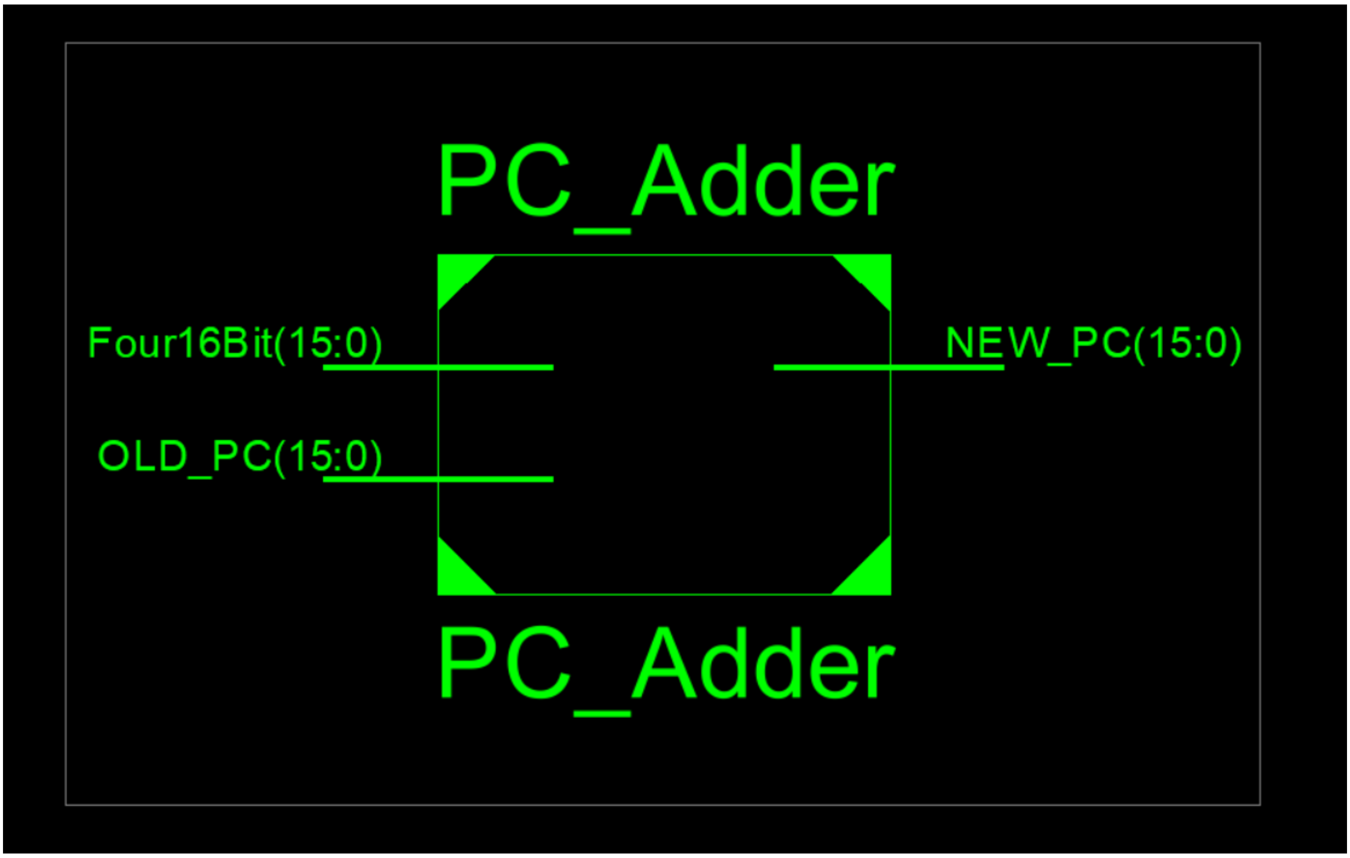


Figure 7: Program Counter Adder Block Diagram

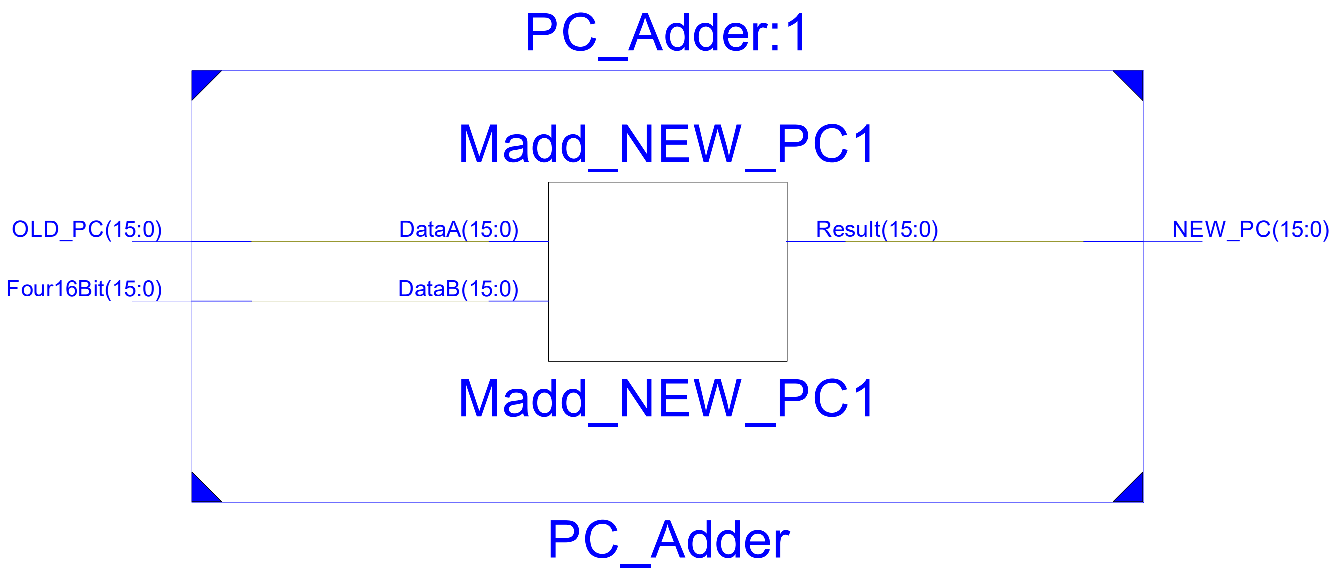


Figure 8: Program Counter Adder RTL

Code:

```
module PC_Adder (OLD_PC,NEW_PC,Four16Bit);

    //Declaring Inputs:
    input [15:0]OLD_PC,Four16Bit;

    //Declaring Outputs:
    output [15:0] NEW_PC;

    //Assigning Outputs:
    assign NEW_PC = OLD_PC + Four16Bit;

endmodule
```

7.3 Instruction Memory

7.3.1 RTL Diagram

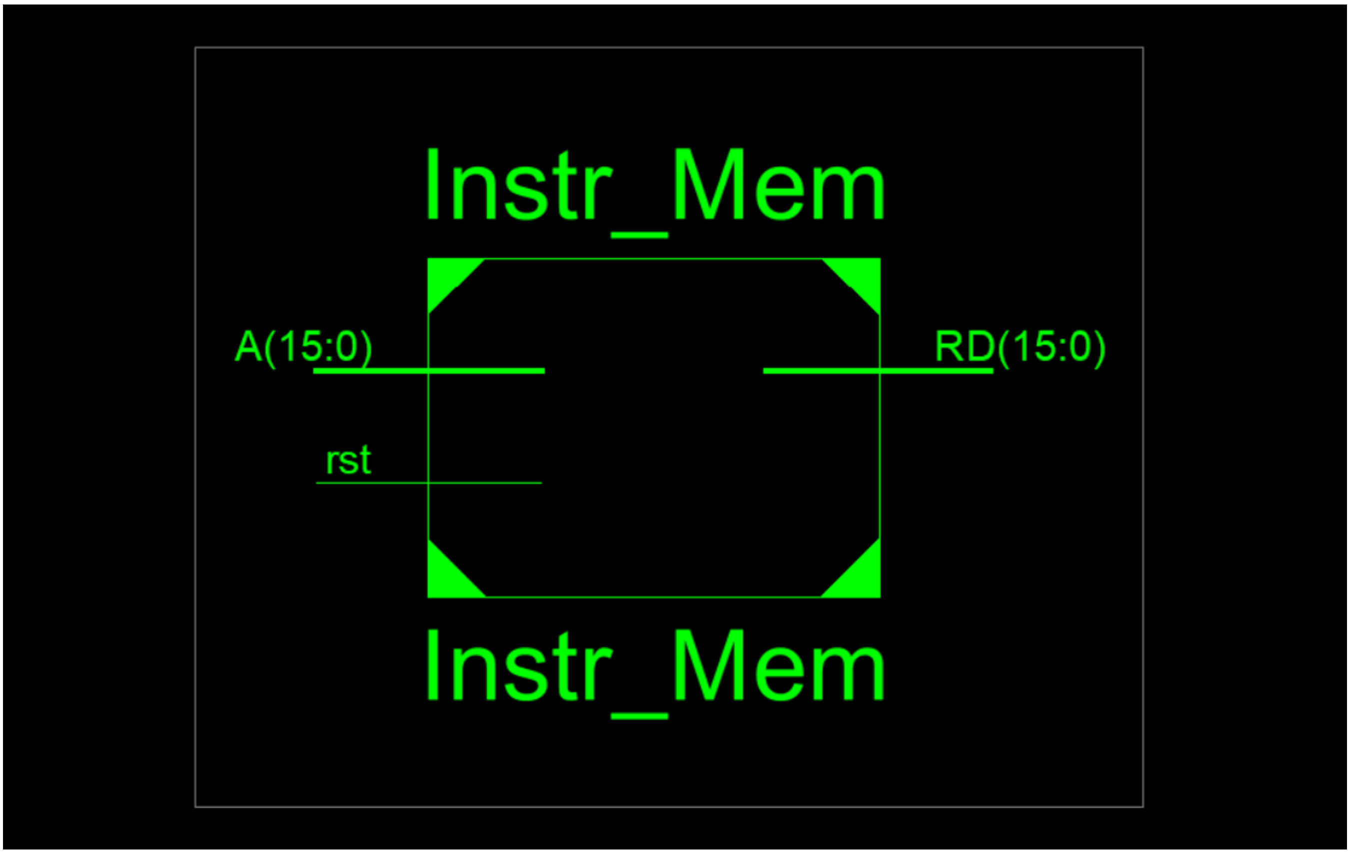


Figure 9: Instruction Memory Block Diagram

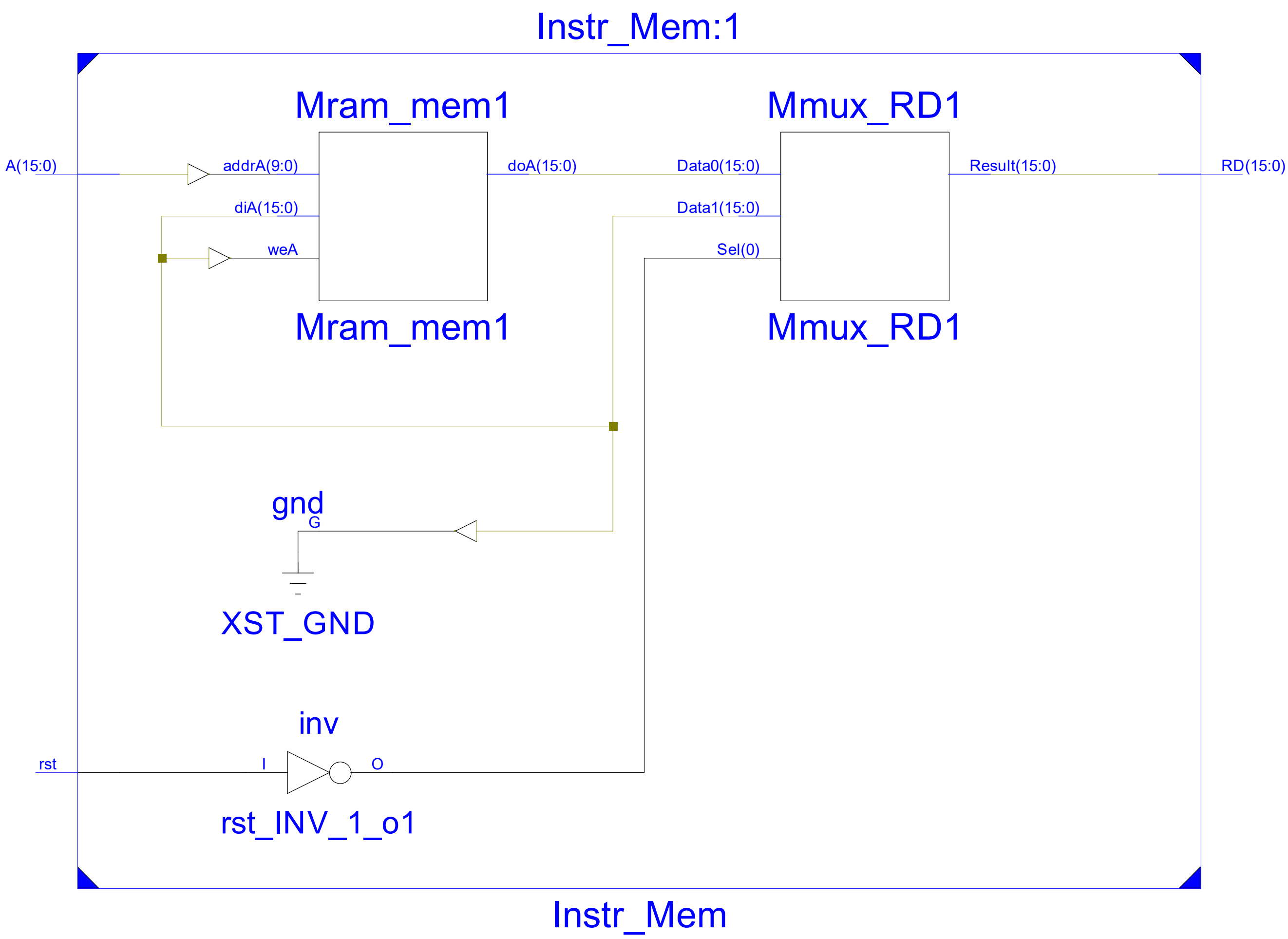


Figure 10: Instruction Memory RTL

Code:

```
module Instr_Mem (A,RD,rst);
input rst;
input [15:0]A;
output [15:0]RD;

reg [15:0] mem [1023:0];

assign RD = (~rst) ? {16{1'b0}} : mem[A[15:2]];


initial begin

  mem[0] = 16'b0000100100000100; // addi $4,$Zero,04
  //          Fun  RD RT RS OP  // RD = RS (Operator) RT
  mem[1] = 16'b0000101100110000; // add $5,$6,$4
  // mem[0] = 16'b0000001011010000; // add $1,$2,$3
  mem[2] = 16'b0001001011010000; // sub $1,$2,$3
  mem[3] = 16'b0010001011010000; // and $1,$2,$3
  mem[4] = 16'b0011001011010000; // or $1,$2,$3
  mem[5] = 16'b0000101100110000; // add $5,$6,$4
  mem[6] = 16'b0000000101000010; // sw $5,M[0]
  mem[7] = 16'b0000000000000011; // Jump 0
end

endmodule
```

7.4 Register File

7.4.1 RTL Diagram

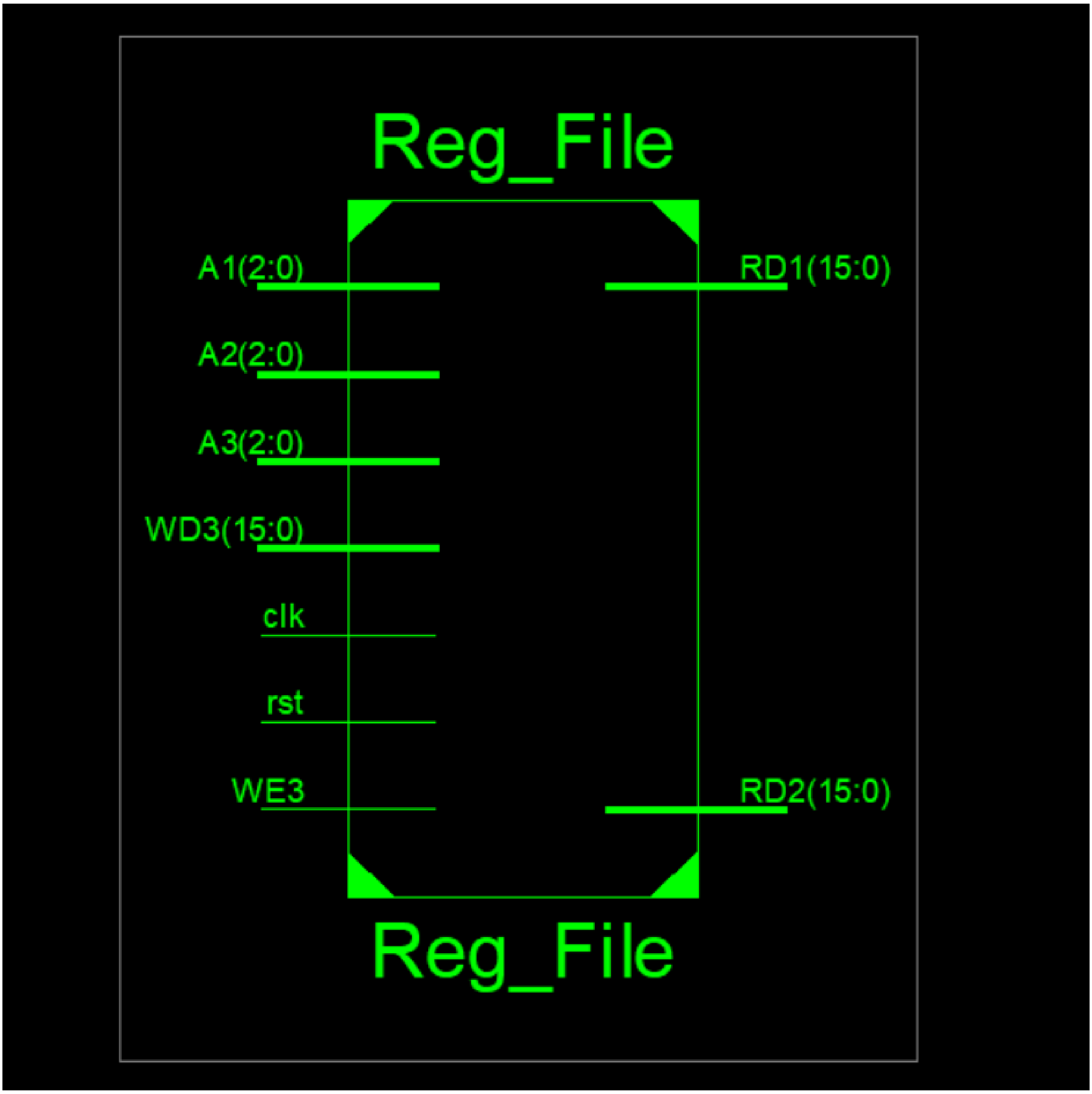


Figure 11: Register File Block Diagram

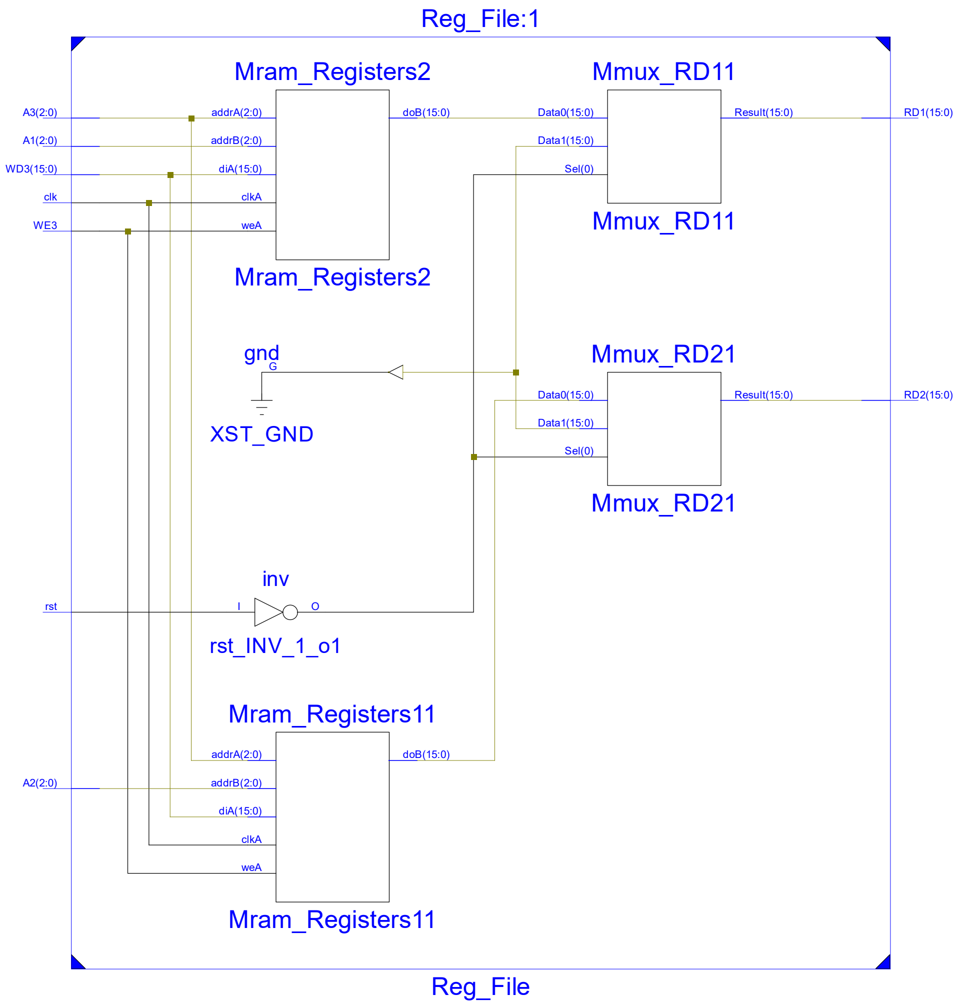


Figure 12: Register File RTL

Code:

```
module Reg_File (A1,A2,A3,WD3,WE3,clk,rst,RD1,RD2);
    //NOTES:
    // WD3 = Write Data
    // WE3 = Write Enable (Comming From Control Unit's RegWrite Output)

    //Declaring Inputs:
    input [2:0] A1,A2,A3;
    input [15:0] WD3;
    input clk,rst,WE3;

    //Declaring Outputs:
    output [15:0] RD1,RD2;

    //Creation Of The Memory:
    reg[15:0] Registers [15:0];

    //Read Functionality:
    assign RD1 = (!rst) ? 16'b0000000000000000 : Registers[A1];
    assign RD2 = (!rst) ? 16'b0000000000000000 : Registers[A2];

    //Write Functionality:
    always @(posedge clk) begin// Positive Edge Of The Clock
        if (WE3)
            begin
                Registers[A3] <= WD3;
            end
    end


    //We can Initialize Data Of Registers From Here!
```

```
    initial begin
        Registers[0] = 16'b0000000000000000;// $zero
        Registers[1] = 16'b0000000000000001;
        Registers[2] = 16'b0000000000000010;
        Registers[3] = 16'b0000000000000011;
        // Registers[4] = 16'b0000000000000100;
        Registers[5] = 16'b0000000000000101;
        Registers[6] = 16'b0000000000000110;
    end

endmodule
```

7.5 Control Unit

7.5.1 RTL Diagram

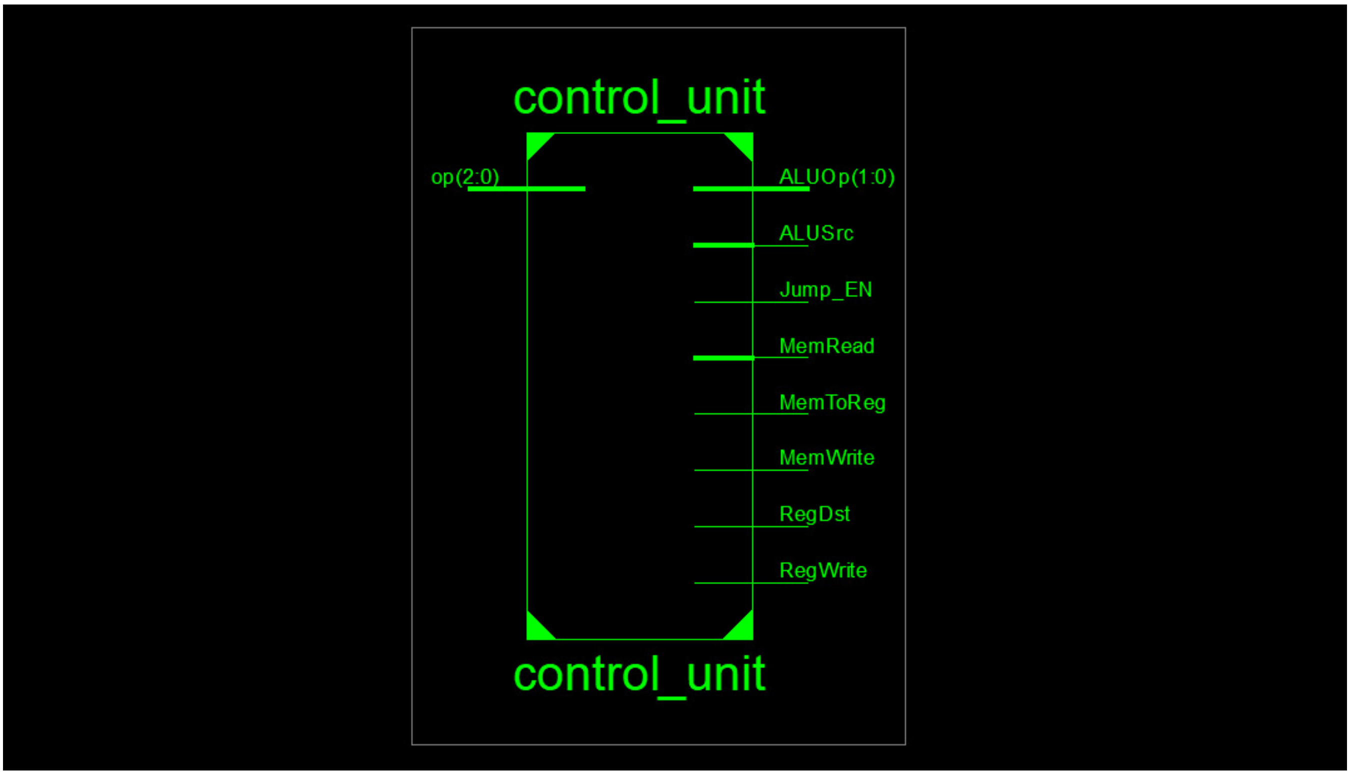


Figure 13: Control Unit Block Diagram

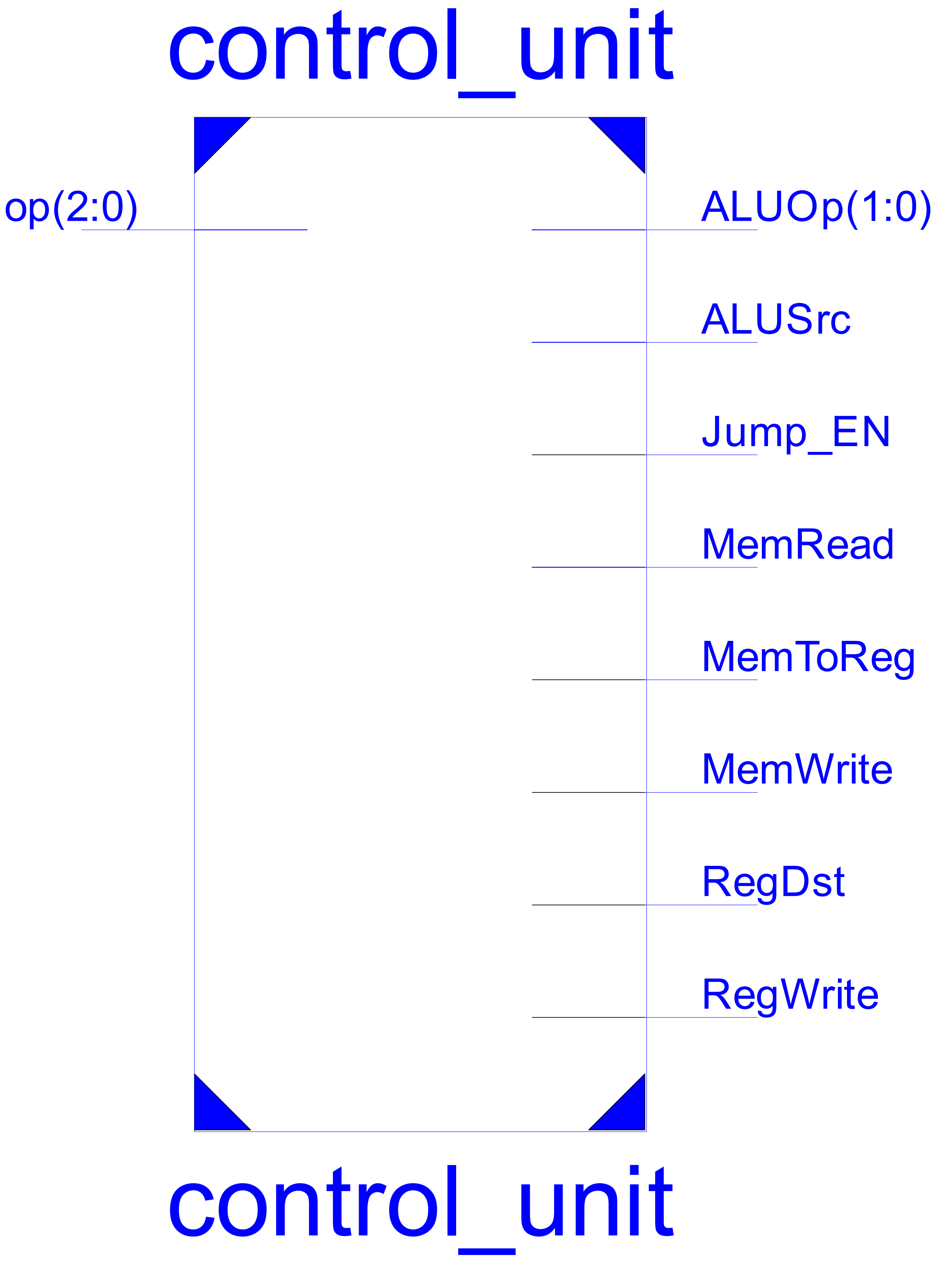


Figure 14: Control Unit RTL

Code:

```
module control_unit (op,RegWrite,MemWrite,ALUSrc,ALUOp,RegDst,MemToReg,MemRead,Jump_EN);

```

```
    //Inputs / Outputs declaration
    input [2:0] op;
    output RegWrite,MemWrite,ALUSrc,RegDst,MemToReg,MemRead,Jump_EN;
    output [1:0] ALUOp;


    //Assigning Outputs:
    // NOTES:
    // ______________________________________________________________________________________________
    // |Instruction | OP  | RegDst | ALUSrc | MemToReg | RegWrite | MemRead | MemWrite | ALUOp | Jump |
    // |  R-Type    | 000 |    1   |    0   |     0    |     1    |    0    |     0    |   00  |  0   |
    // | Load Word  | 001 |    0   |    1   |     1    |     1    |    1    |     0    |   11  |  0   |
    // |Store Word  | 010 |    0   |    1   |     0    |     0    |    0    |     1    |   11  |  0   |
    // |Jump-Type   | 011 |    0   |    0   |     0    |     0    |    0    |     0    |   00  |  1   |
    // |   addi     | 100 |    0   |    1   |     0    |     1    |    0    |     0    |   11  |  0   |
    // ----------------------------------------------------------------------------------------------
    assign RegWrite = ((op == 3'b000) | (op == 3'b001) | (op == 3'b100)) ? 1'b1 : 1'b0;// if (op == R-type | op == lw |
      op == addi)

    assign RegDst = (op == 3'b000) ? 1'b1 : 1'b0;// if (op == R-type)

    assign ALUSrc = ((op == 3'b001) | (op == 3'b010) | (op == 3'b100)) ? 1'b1 : 1'b0;// if (op == lw | op == lw | op ==
      addi)

    assign MemToReg = (op == 3'b001) ? 1'b1 : 1'b0;// if (op == lw)

    assign MemRead = ((op == 3'b001) | (op == 3'b010)) ? 1'b1 : 1'b0;// if (op == lw | op == sw)

    assign MemWrite = (op == 3'b010) ? 1'b1 : 1'b0;// if (op == sw)

    assign ALUOp = ((op == 3'b001) | (op == 3'b010) | (op == 3'b100)) ? 2'b11 : 2'b00;// if (op == lw | op == sw | op
     == addi)

    assign Jump_EN = (op == 3'b011) ? 1'b1 : 1'b0; // if (op == jump)

endmodule
```

7.6 ALU

7.6.1 RTL Diagram

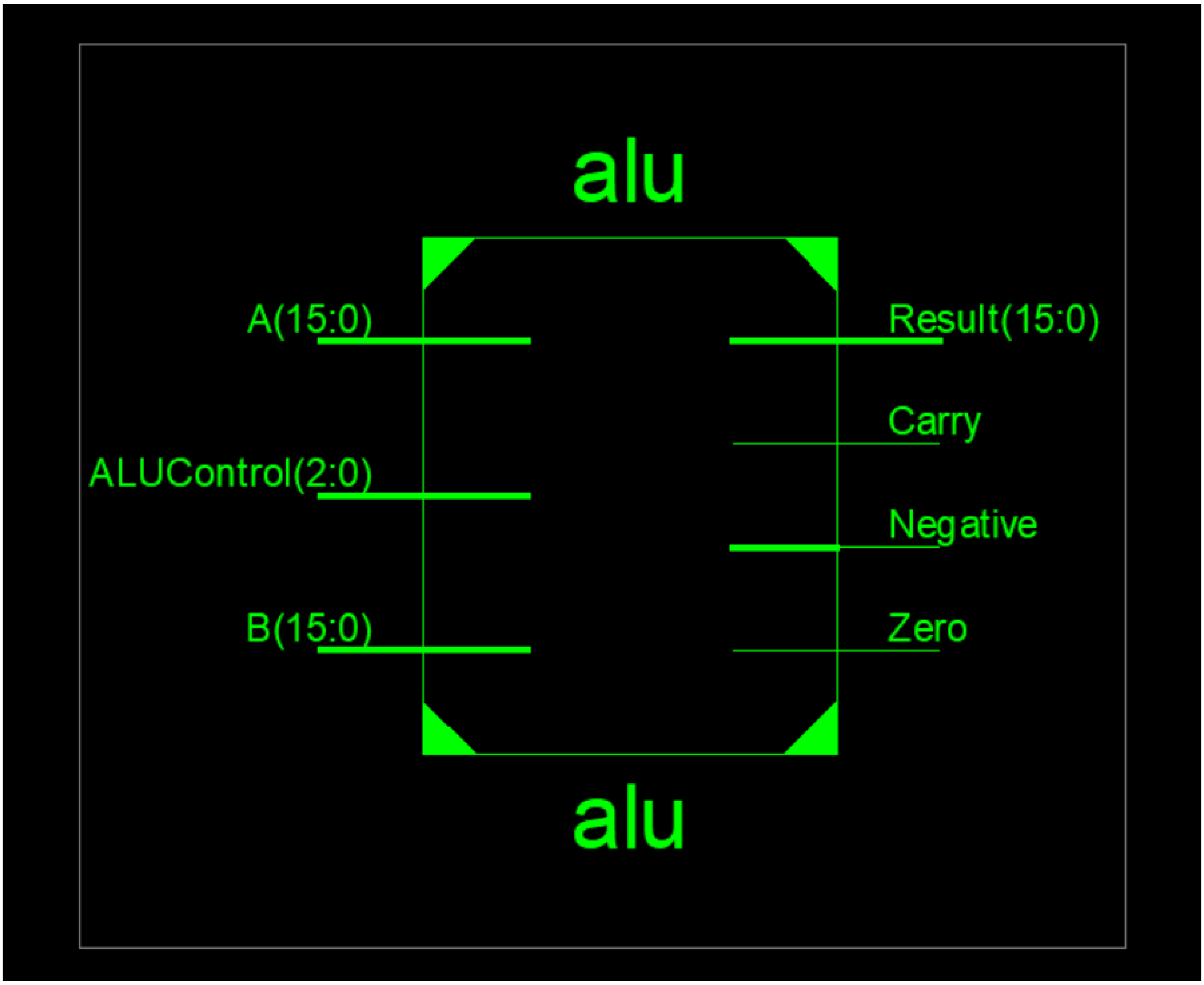


Figure 15: ALU Block Diagram

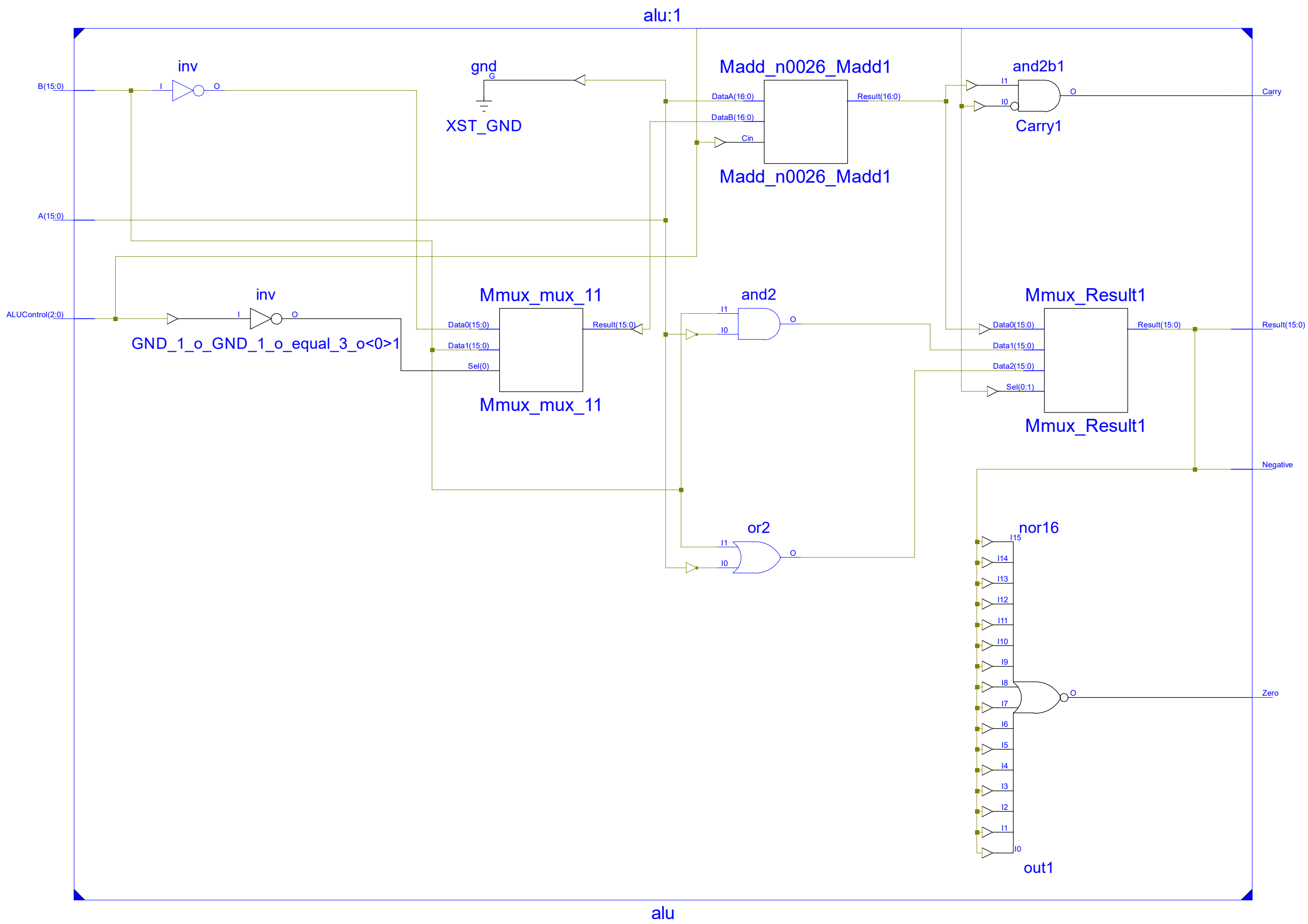


Figure 16: ALU RTL

Code:

```
module alu (A,B,ALUControl,Result,Negative,Carry,Zero);
    //declaring inputs
    input [15:0] A,B;
    input [2:0] ALUControl;

    //declaring Outputs
    output [15:0] Result;
    //Flags
    output Negative,Carry,Zero;

    //declaring intermediate wires
    //AND or OR:
    wire [15:0] a_and_b;
    wire [15:0] a_or_b;

    //Addition or Subtraction:
    wire [15:0] not_b;
    wire [15:0] sum;
    wire cout;
    //Mux
    wire [15:0] mux_1; //2's Complement Or Not
    wire [15:0] mux_2;

    //logic Design Outputs
    //AND Operation
    assign a_and_b = A & B;

    //OR Operation (Additional)
    assign a_or_b = A | B;

```

```
    //Tenary Operator
    //assign name = (Condition) ? True Value : False Value;
    assign mux_1 = (ALUControl[0] == 2'b0) ? B : not_b; //2's complement

    //Addition or Subtraction:
    assign not_b = ~B; // 2's complement of B
    assign {cout,sum} = A + mux_1 + ALUControl[0];

    //4X1 Mux:
    assign mux_2 = (ALUControl[1:0] == 2'b00) ? sum :
                   (ALUControl[1:0] == 2'b01) ? sum :
                   (ALUControl[1:0] == 2'b10) ? a_and_b : a_or_b;

    // assign mux_2 = (ALUControl[1] == 1'b0) ? sum :
    //                (ALUControl[0] == 1'b0) ? a_and_b : a_or_b;

    //Result:
    assign Result = mux_2;

    //Flag Assignments:
    assign Zero = &(~Result); //Zero Flag
    assign Negative = Result[15]; //Negative Flag
    assign Carry = cout & (~ALUControl[1]); //Carry Flag

endmodule
```

7.7 ALU Control

7.7.1 RTL Diagram

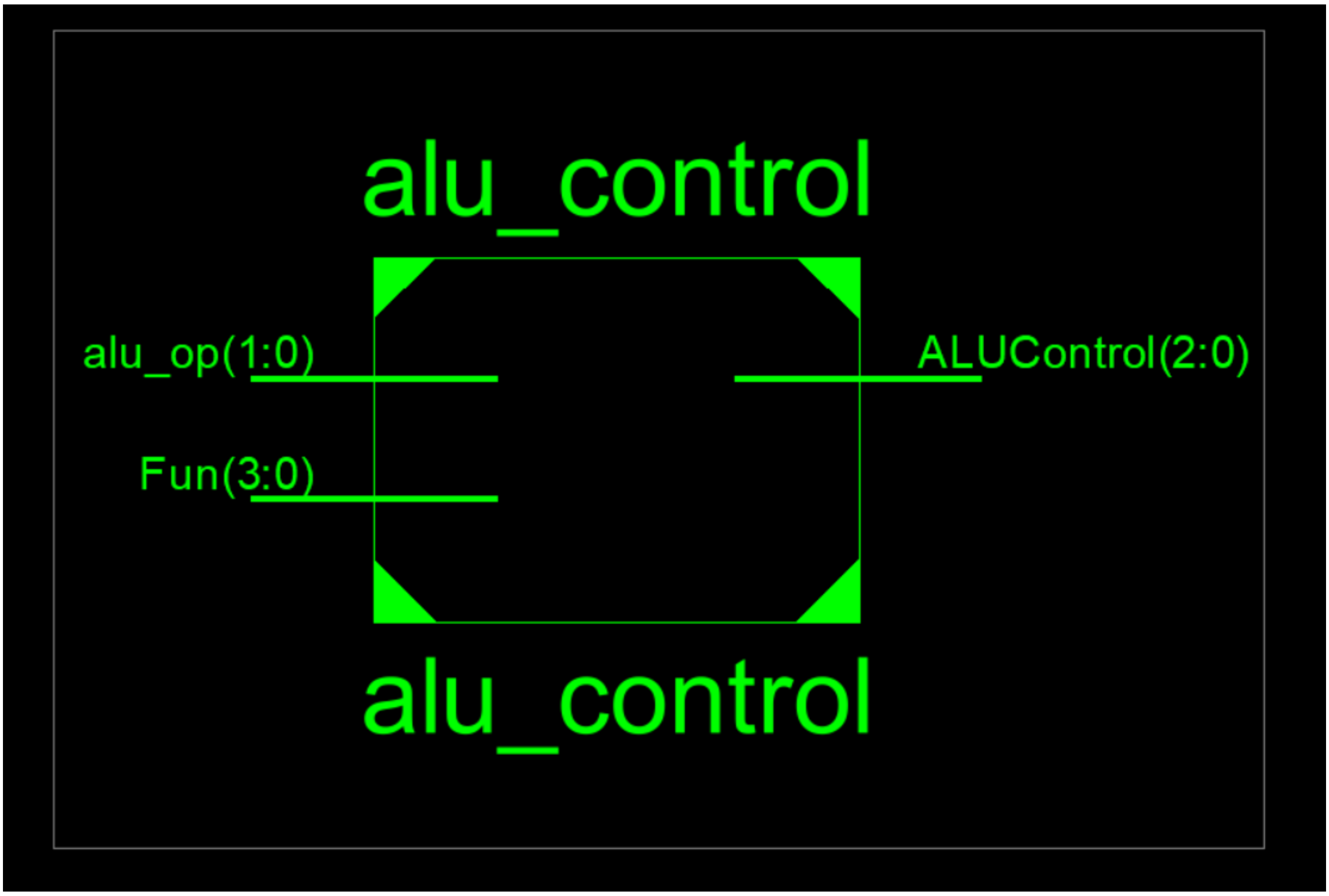


Figure 17: ALU Control Block Diagram

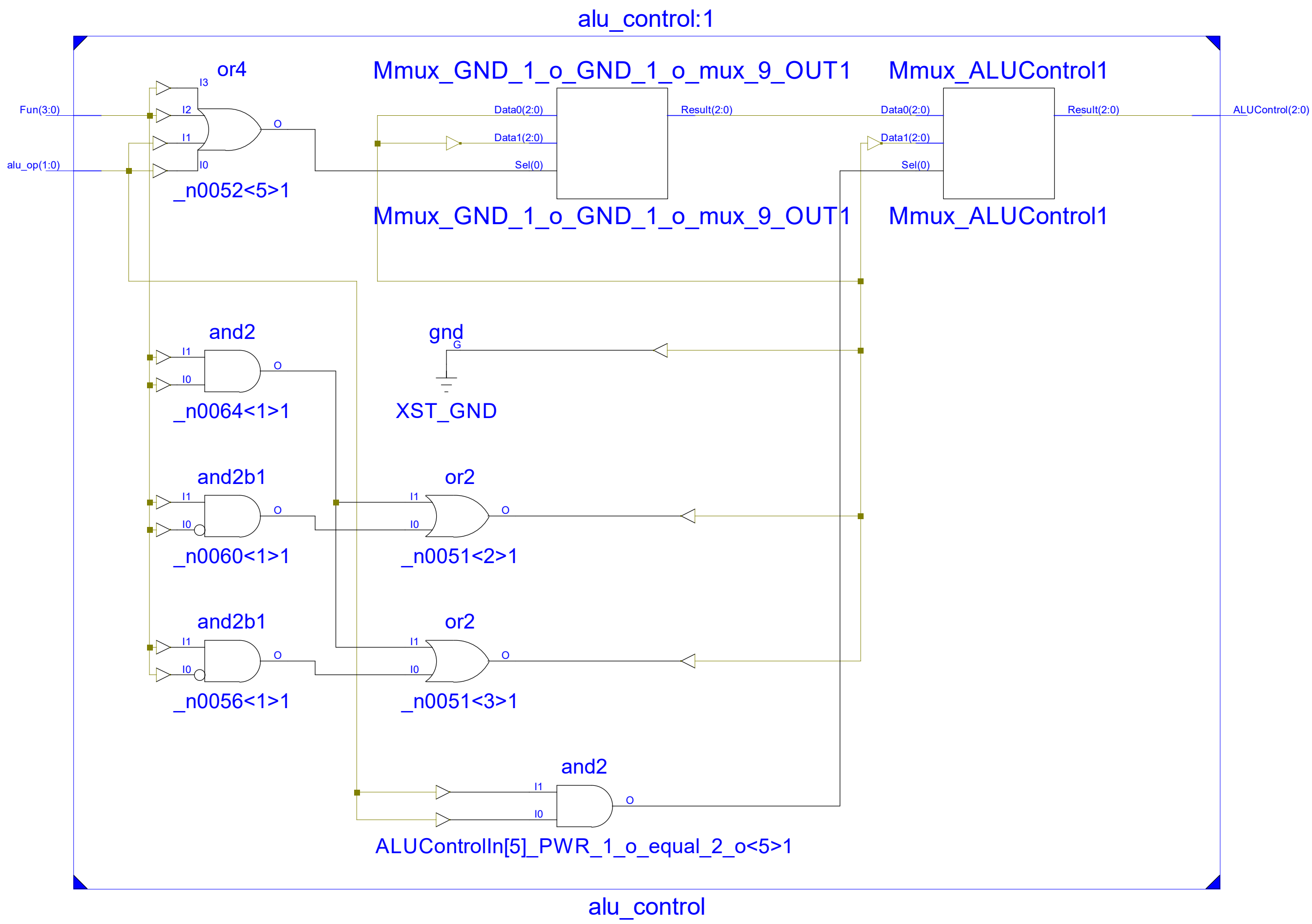


Figure 18: ALU Control RTL

Code:

```
module alu_control (alu_op,ALUControl,Fun);
    //Declaring Inputs:
    input [1:0] alu_op;
    input [3:0] Fun;

    //Declaring Outputs:
    output [2:0] ALUControl;

    //Intermediate Wire
    wire [5:0] ALUControlIn;
    assign ALUControlIn = {alu_op,Fun}; //Contatinate. Same Pattern {Fun,alu_op}!={alu_op,Fun};

    //Tenary Operator
    assign ALUControl = (ALUControlIn == 6'b11xxxx) ? 3'b000: // lw,sw I-type
                        (ALUControlIn == 6'b000000) ? 3'b000: // Add R-type
                        (ALUControlIn == 6'b000001) ? 3'b001: // Sub R-type
                        (ALUControlIn == 6'b000010) ? 3'b010: // AND R-type
                        (ALUControlIn == 6'b000011) ? 3'b011: // Or R-type
                        3'b000;
endmodule
```

7.8 Sign Extender

7.8.1 RTL Diagram

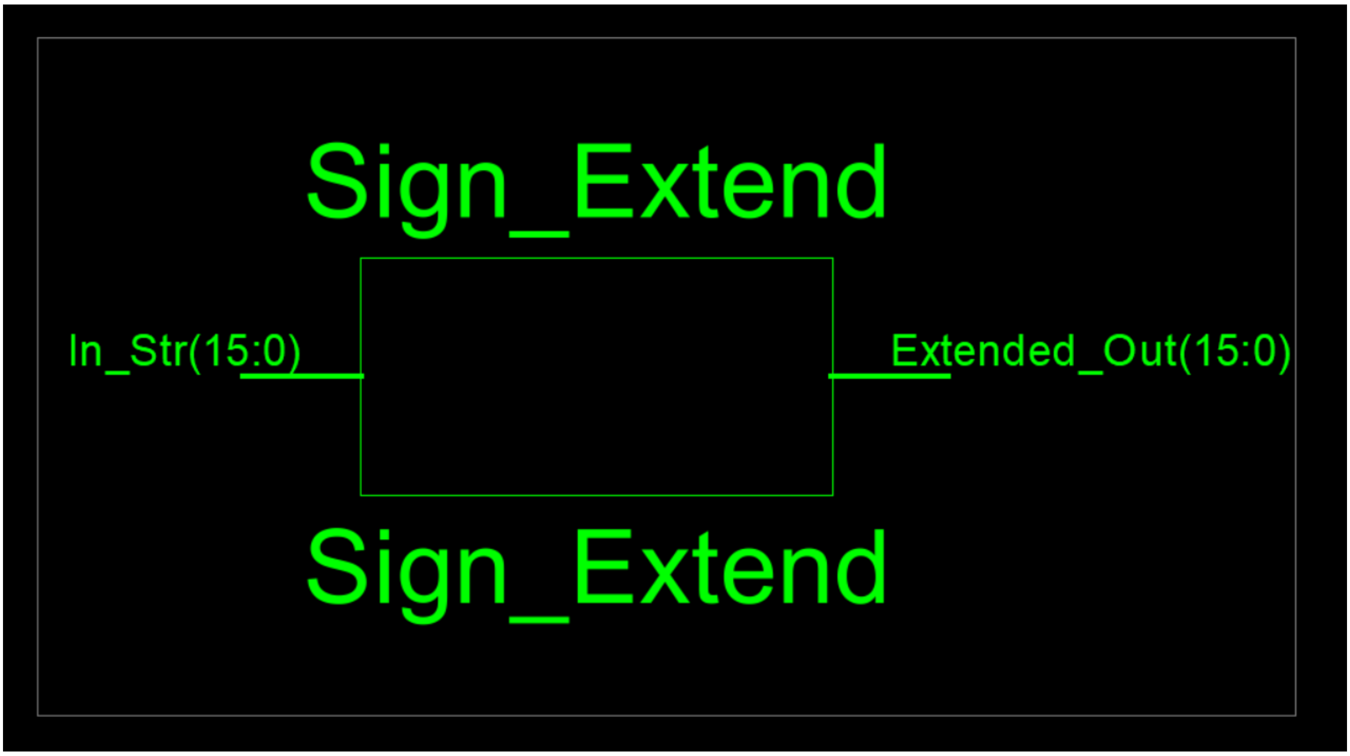


Figure 19: Sign Extender Block Diagram

Code:

```
module Sign_Extend (In_Str,Extended_Out);
    //Declaring Inputs:
    input [15:0] In_Str;

    //Declaring Outputs:
    output [15:0] Extended_Out;

    //Assigning Outputs:
    assign Extended_Out = (In_Str[9]) ? {{9{1'b1}},In_Str[15:9]}:
                                        {{9{1'b0}},In_Str[15:9]};
endmodule
```

7.9 Mux 2X1

7.9.1 RTL Diagram

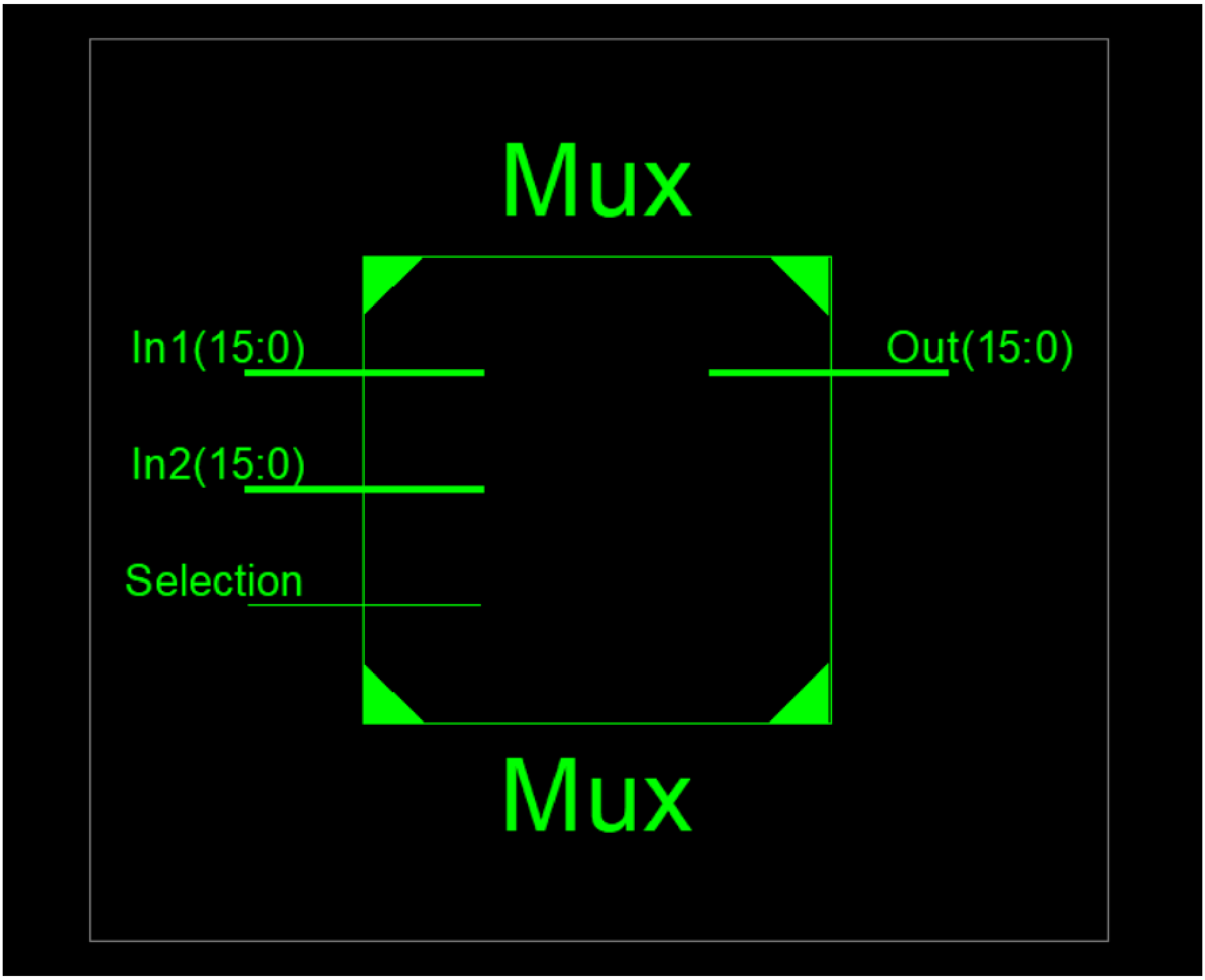


Figure 20: 2X1 Multiplexer 16 bit

Code:

```
module Mux (In1,In2,Selection,Out);
    //Declaring Inputs:
    input [15:0] In1,In2;
    input Selection;

    //Declaring Output:
    output [15:0] Out;

    //Assigning Output:
    assign Out = (~Selection) ? In1 : In2;
endmodule
```

7.10 Mux 2x1 (RF_I_R)

7.10.1 RTL Diagram

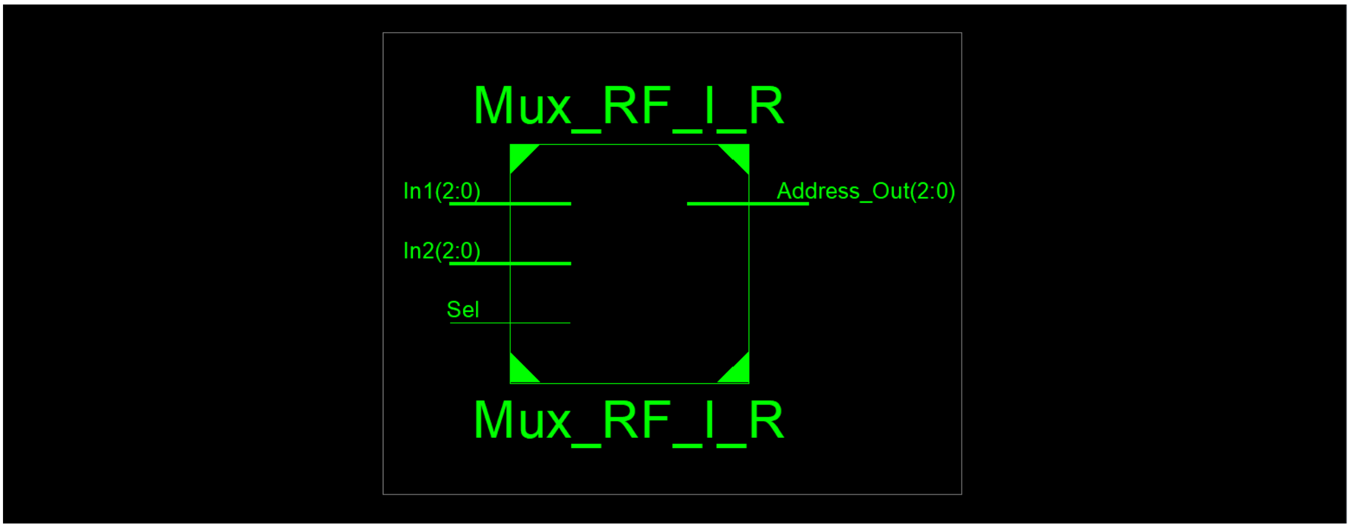


Figure 21: 2X1 Multiplexer 3 bit

Code:

```
module Mux_RF_I_R (In1,In2,Sel,Address_Out);
    input [2:0] In1,In2;
    input Sel;
    output [2:0] Address_Out;
    assign Address_Out = (~Sel) ? In1 : In2;
endmodule
```

7.11 Data Memory

7.11.1 RTL Diagram

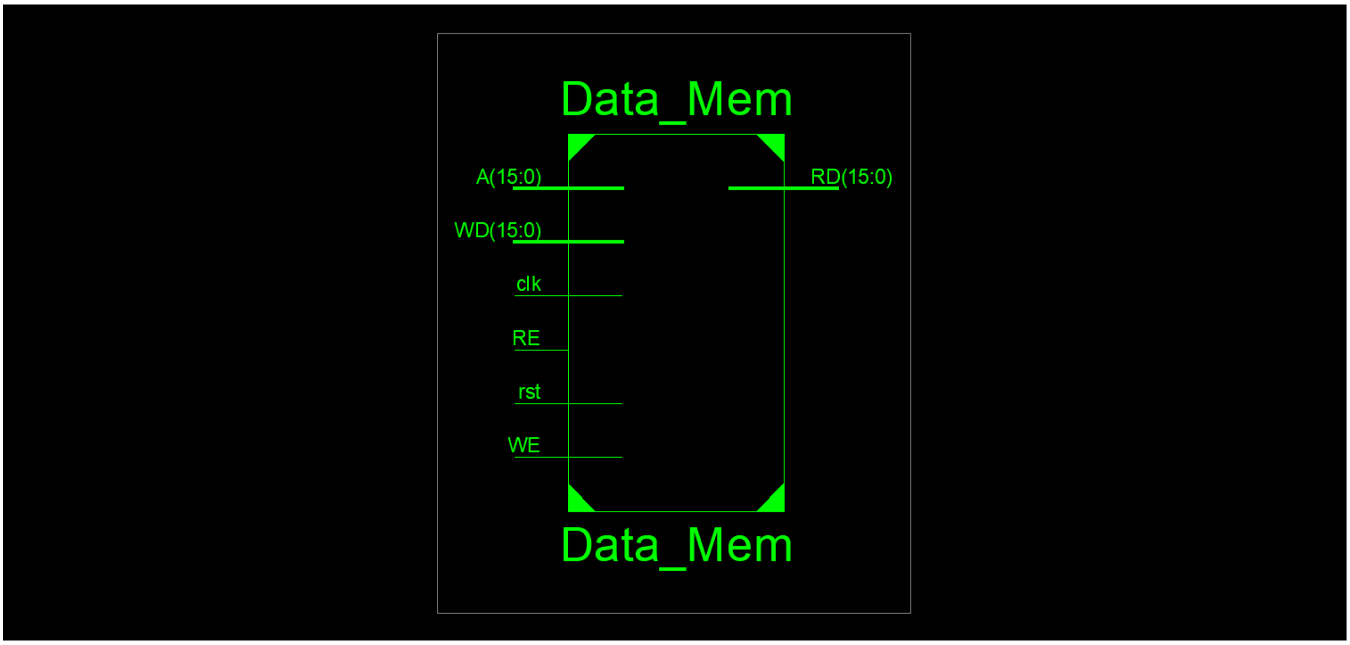


Figure 22: Data Memory Block Diagram

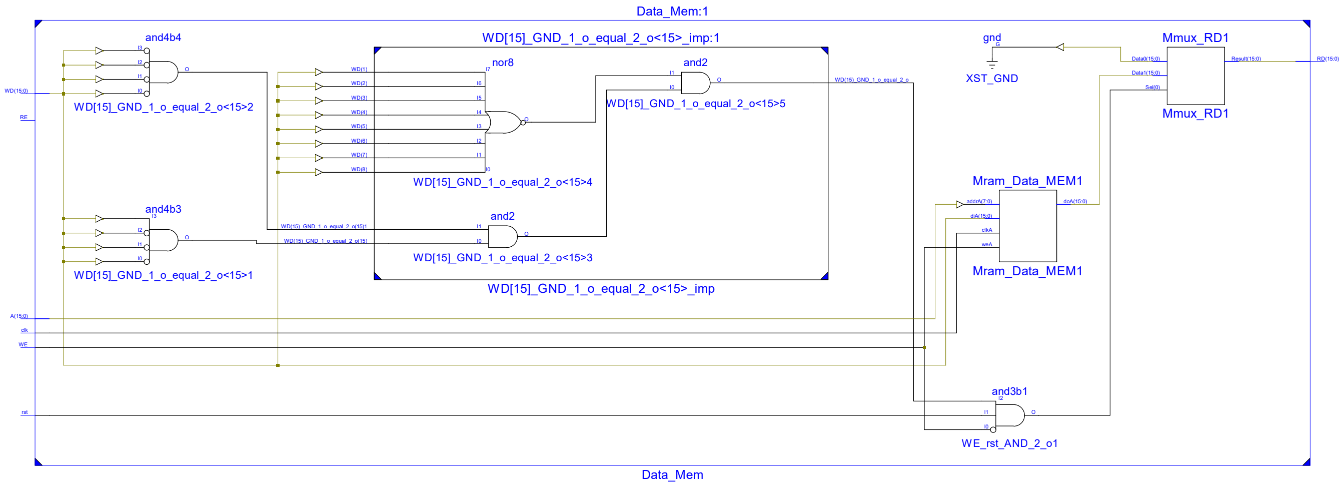


Figure 23: Data Memory RTL

Code:

```
module Data_Mem (A,WD,WE,RE,clk,RD,rst);
    //Declaring Inputs:
    input [15:0] A,WD;
    input clk,WE,RE,rst;

    //Declaruing Outputs:
    output [15:0] RD;


    //Creation OF Memory:
    reg[15:0] Data_MEM [255:0];



    //Read Functionality:
    assign RD = ((WE == 1'b0) & (WD == 1'b1) & (RE) & (rst)) ? Data_MEM[A] : 16'b0;

    //Write Functionality:
    always @(posedge clk) begin
        if(WE)
            begin
                Data_MEM[A] <= WD;
            end
    end

    //We can Initialize Data Of The Memory From Here!
    initial begin
        Data_MEM[0] = 16'b0000000000000000;
        Data_MEM[1] = 16'b0000000000000001;
        Data_MEM[2] = 16'b0000000000000010;
```

```
    end

endmodule
```

8 Total Combined Data Path

8.1 Single Cycle MIPS

8.1.1 RTL Diagram

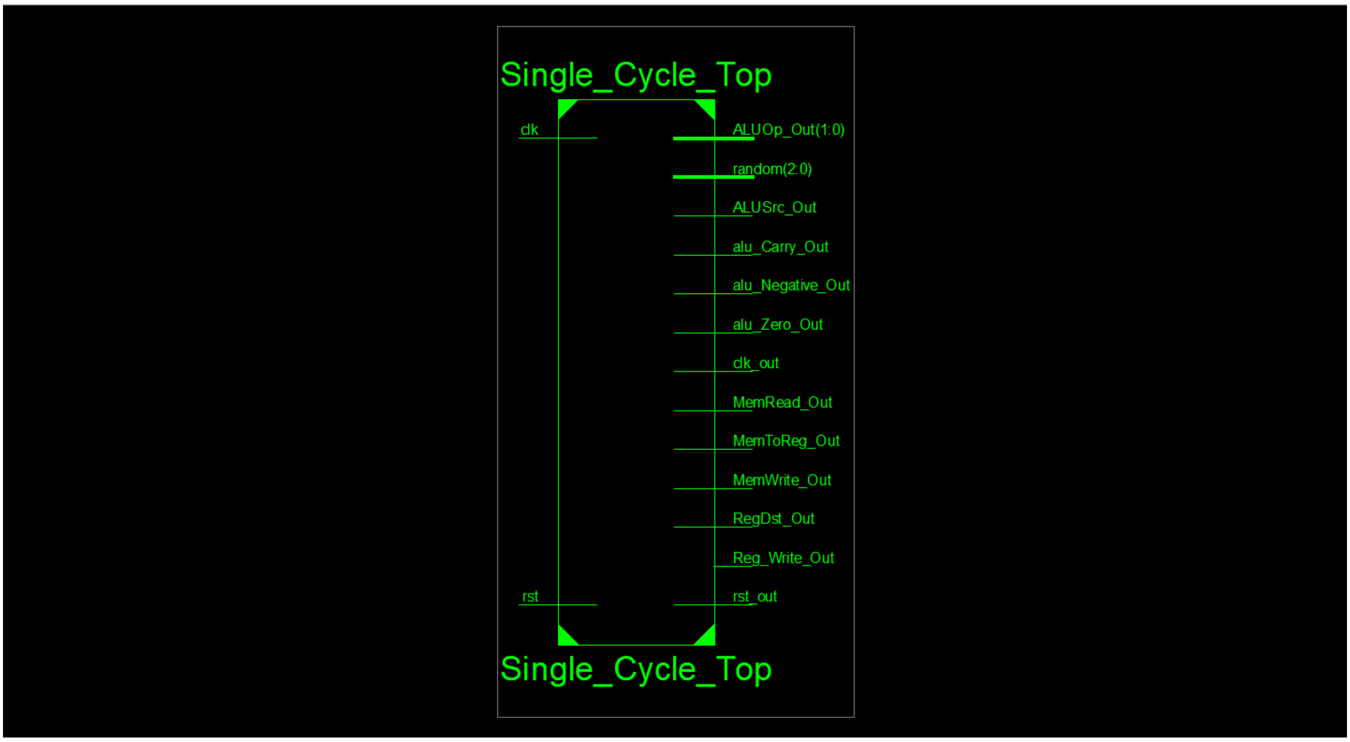


Figure 24: Data Path Block Diagram

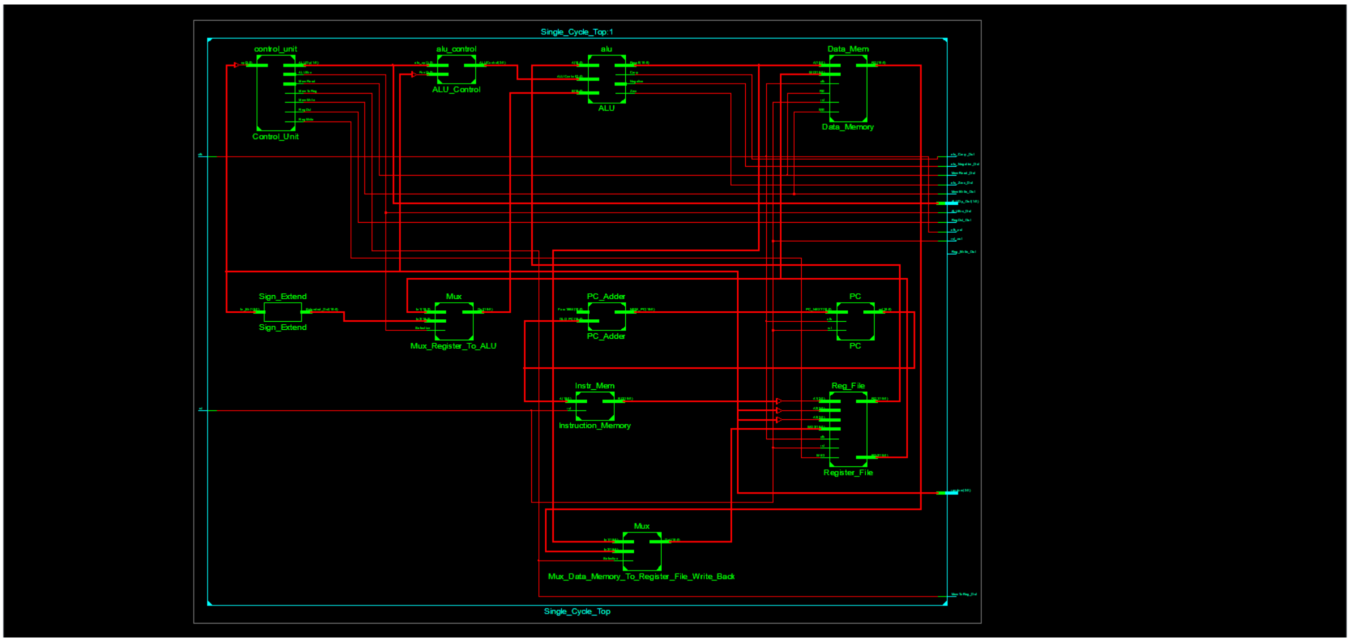


Figure 25: Data Path.
A Closer Look

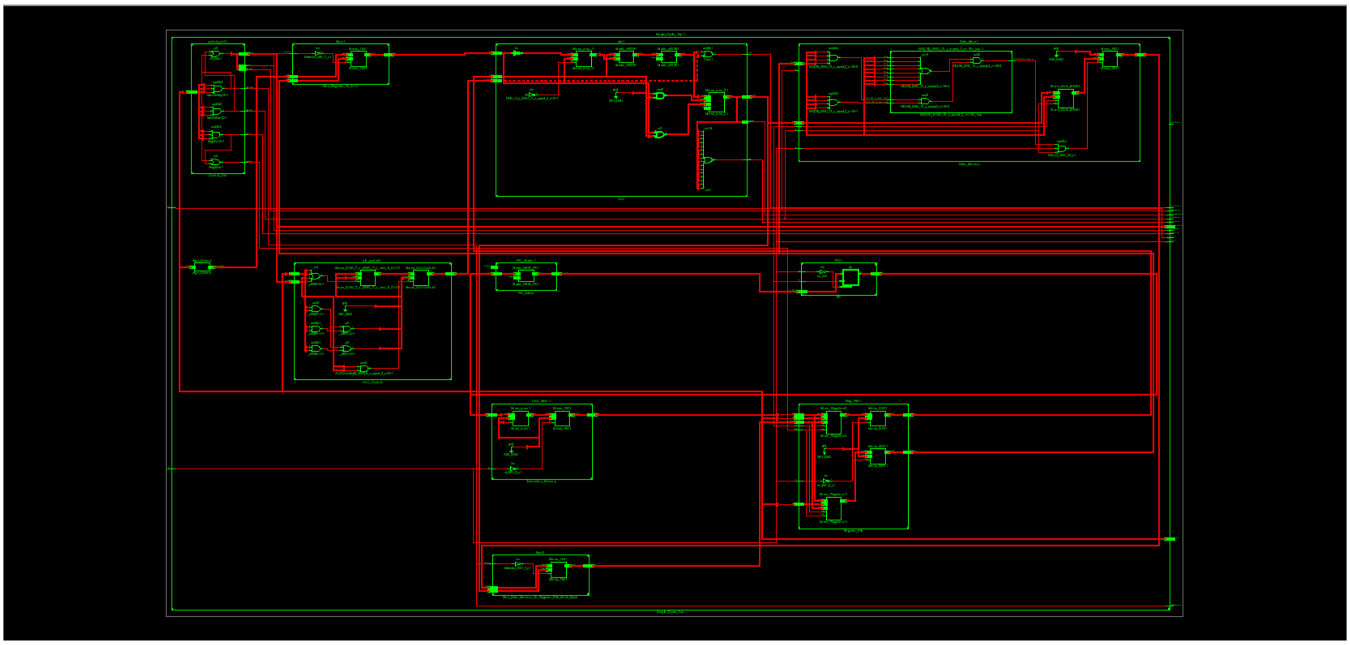


Figure 26: Data Path.
A More Better Look!

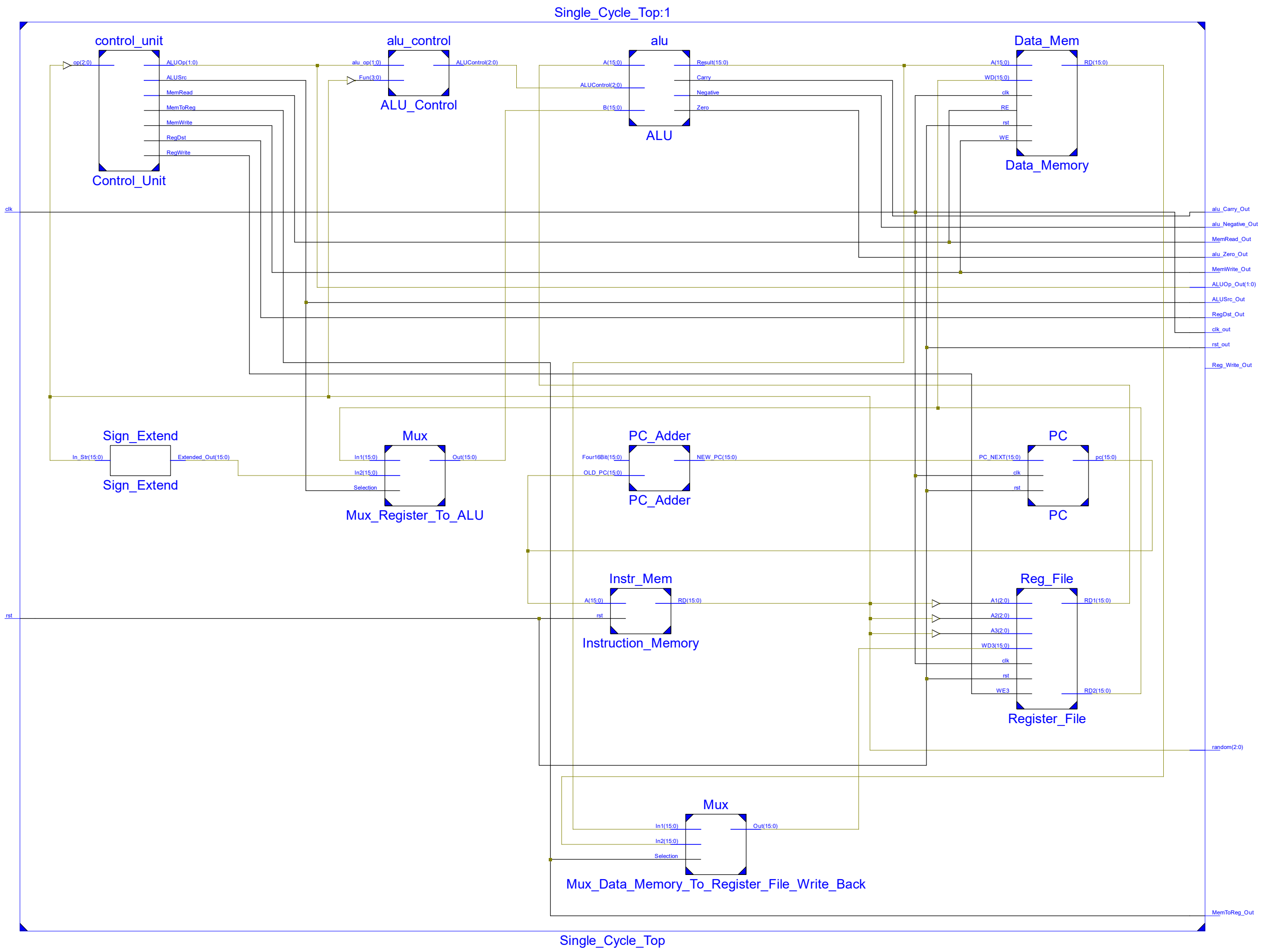


Figure 27: Data Path.
A More More Better Look!

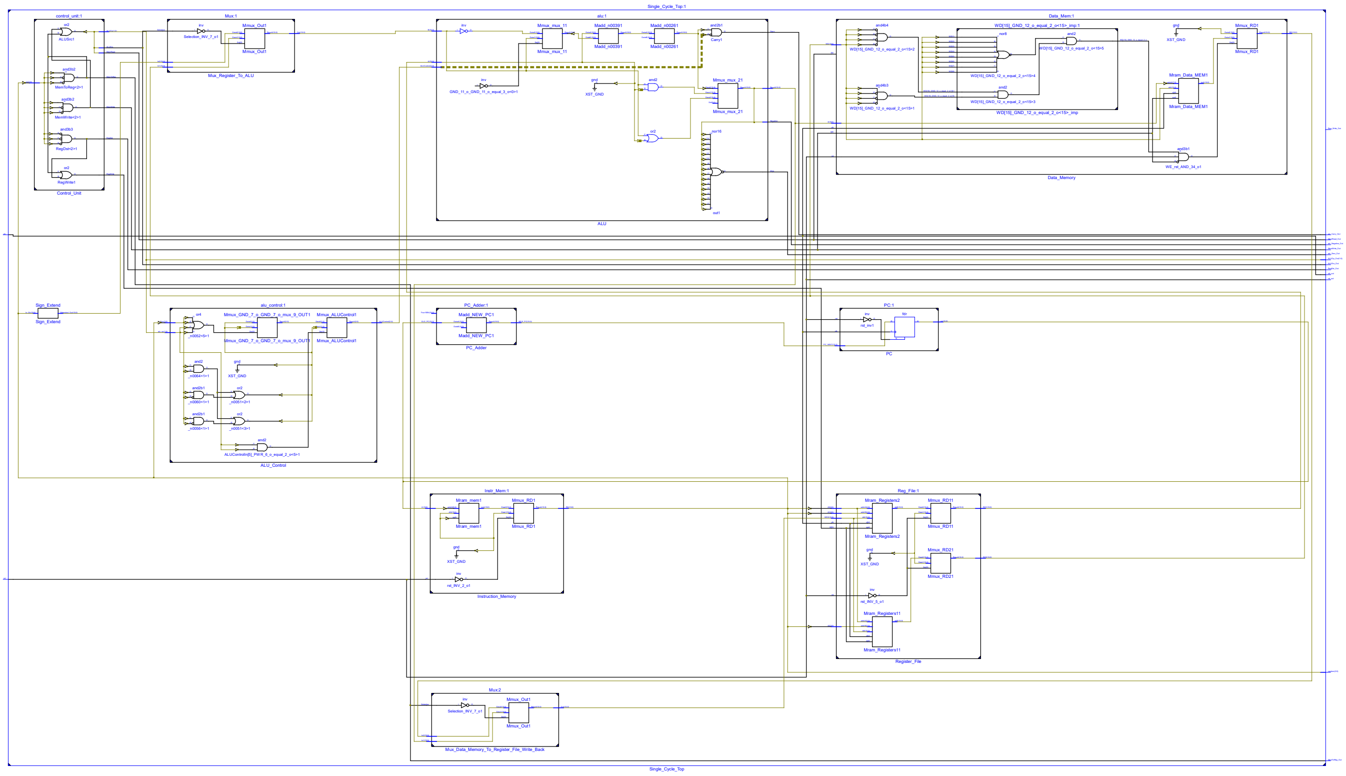


Figure 28: Data Path.
A More More More Better Look!

Code:

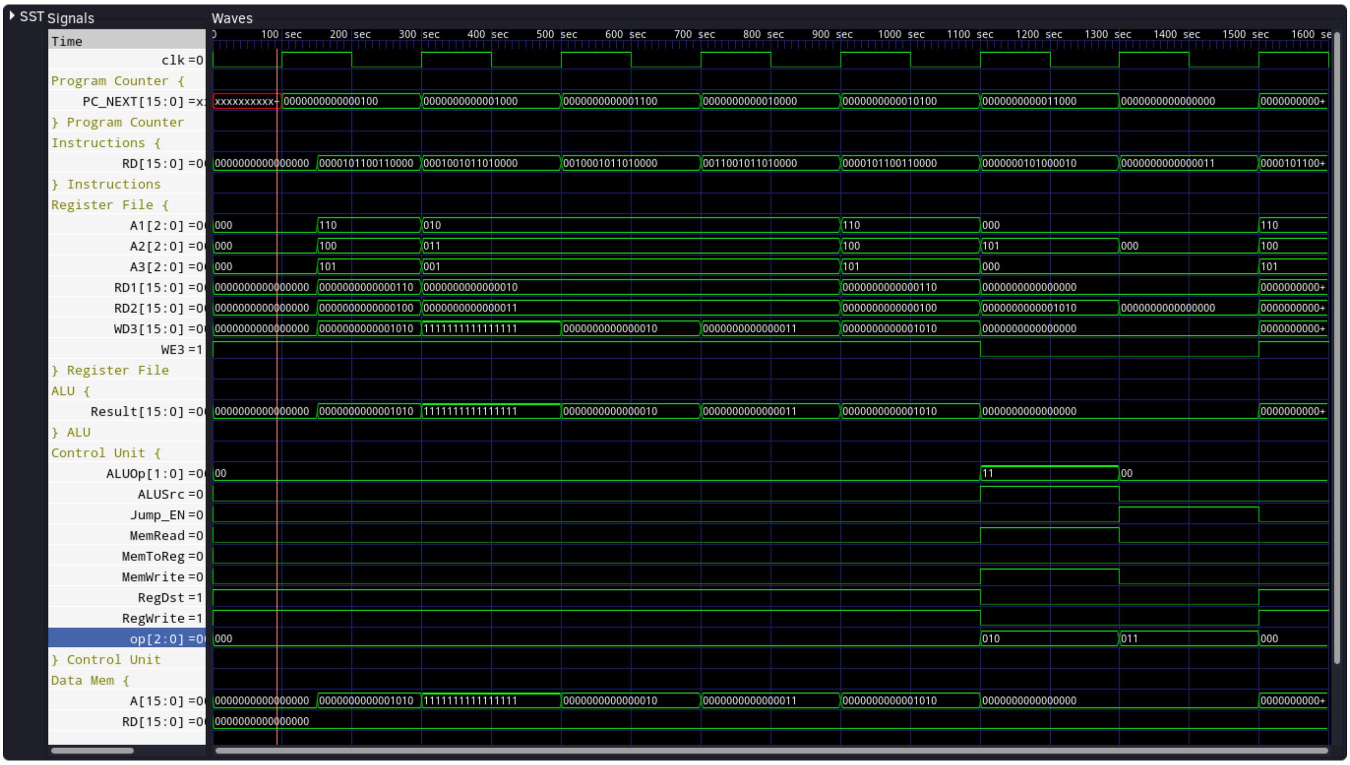
```
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Program_Counter.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Instruction_Memory.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Register_File.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Sign_Extender.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/ALU.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/ALU_Control.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Control_Unit.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Data_Memory.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/PC_Adder.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Mux.v"
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Mux_RF_I_R.v"
module Single_Cycle_Top (alu_Carry_Out,alu_Negative_Out,alu_Zero_Out,random,clk,rst,clk_out,rst_out,Reg_Write_Out,MemWrite_Out,ALUSrc_Out,RegDst_Out,MemToReg_Out,MemRead_Out,ALUOp_Out);
    //Declaring Inputs:
    input rst,clk;

    wire[15:0] PC_Top,RD_Instr,RD1_Top,RD2_Top,SignExt_Top,ALU_RESULT;// For Connecting RD (From Instruction Memory) To Register File
    //Control Unit's Wires:
    wire RegWrite,MemWrite,ALUSrc,RegDst,MemToReg,MemRead,Jump_EN;
    wire [1:0] ALUOp;
    //ALU_Control's Wires:
    wire [2:0] ALUControl_Top;
    //DataMemory:
    wire [15:0] ReadData;
    //PC's Wires:
    wire [15:0] NEW_PC;
    //Jump Address Or New PC:
```

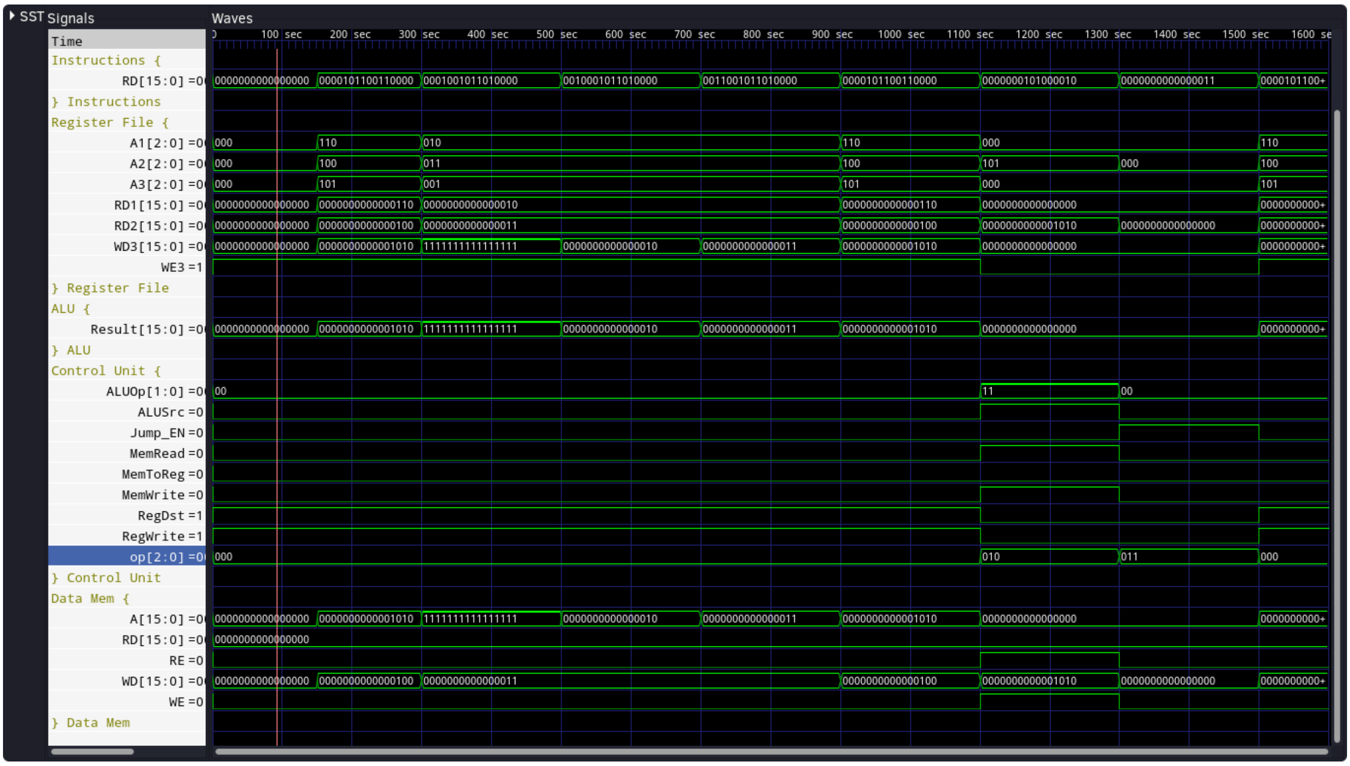
```
    wire [15:0] Jump_Adrr_Or_New_PC;
    //Mux_Register_To_ALU Result:
    wire [15:0] Mux_Register_To_ALU_Result;
    wire [15:0] Mux_Data_Memory_To_Register_File_Write_Back_Result;
    wire [2:0] Register_Address_Write;

    //Declaring Outputs:
    output clk_out,rst_out,Reg_Write_Out,MemWrite_Out,ALUSrc_Out,RegDst_Out,MemToReg_Out,MemRead_Out,alu_Negative_Out,
     alu_Zero_Out,alu_Carry_Out;
    output [1:0] ALUOp_Out;
    output [2:0] random;
    //Assigning The Outputs:
    assign clk_out = clk;
    assign rst_out = rst;
    assign Reg_Write_Out = Reg_Write_Out;
    assign MemWrite_Out = MemWrite;
    assign ALUSrc_Out = ALUSrc;
    assign RegDst_Out = RegDst;
    assign MemToReg_Out = MemToReg;
    assign MemRead_Out = MemRead;
    assign ALUOp_Out = ALUOp;
    assign random = RD_Instr[8:6];
    //assign alu_Zero_Out = ;
    //assign alu_Negative_Out = ;
    //assign alu_Carry_Out = ;

    PC PC(
        .clk(clk),
        .rst(rst),
        .pc(PC_Top),
        .PC_NEXT(Jump_Adrr_Or_New_PC)
    );


    PC_Adder PC_Adder(
        .OLD_PC(PC_Top),
        .NEW_PC(NEW_PC),
        .Four16Bit(16'b0000000000000100)
    );

    Mux Mux_PC_Jump(
        .In1(NEW_PC),
        .In2({3'b000,RD_Instr[15:3]}),
        .Selection(Jump_EN),
        .Out(Jump_Adrr_Or_New_PC)
    );

    Instr_Mem Instruction_Memory(
        .rst(rst),
        .A(PC_Top),
        .RD(RD_Instr)
    );


    control_unit Control_Unit(
        .op(RD_Instr[2:0]),
        .RegWrite(RegWrite),
        .MemWrite(MemWrite),
        .ALUSrc(ALUSrc),
        .ALUOp(ALUOp),
        .RegDst(RegDst),
        .MemToReg(MemToReg),
        .MemRead(MemRead),
        .Jump_EN(Jump_EN)
    );

    alu_control ALU_Control (
        .alu_op(ALUOp),
        .ALUControl(ALUControl_Top),
        .Fun(RD_Instr[15:12])
    );


    Mux_RF_I_R Mux_RF_I_OR_R(
        .In1(RD_Instr[11:9]),
        .In2(RD_Instr[8:6]),
```

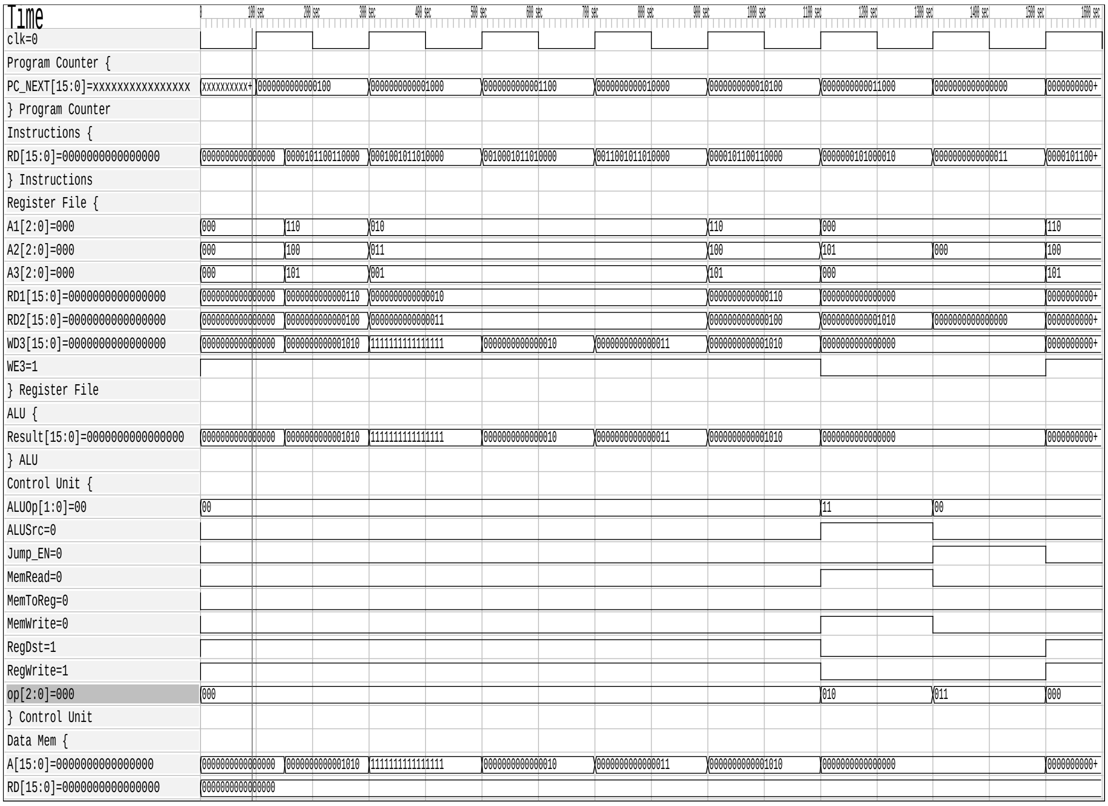
```
        .Sel(ALUSrc & RegWrite & ~(MemToReg)),
        .Address_Out(Register_Address_Write)
    );

    Reg_File Register_File(
        .A1(RD_Instr[5:3]),
        .A2(RD_Instr[8:6]),
        .A3(Register_Address_Write),
        .WD3(Mux_Data_Memory_To_Register_File_Write_Back_Result),
        .WE3(RegWrite),
        .clk(clk),
        .rst(rst),
        .RD1(RD1_Top),
        .RD2(RD2_Top)
    );


    Sign_Extend Sign_Extend(
        .In_Str(RD_Instr),
        .Extended_Out(SignExt_Top)
    );

    Mux Mux_Register_To_ALU(
        .In1(RD2_Top),
        .In2(SignExt_Top),
        .Selection(ALUSrc),
        .Out(Mux_Register_To_ALU_Result)
    );

    alu ALU(
        .A(RD1_Top),
        .B(Mux_Register_To_ALU_Result),
        .ALUControl(ALUControl_Top),
        .Result(ALU_RESULT),
        .Negative(alu_Negative_Out),
        .Carry(alu_Carry_Out),
        .Zero(alu_Zero_Out)
    );

    Data_Mem Data_Memory (
        .A(ALU_RESULT),
        .WD(RD2_Top),
        .WE(MemWrite),
        .RE(MemRead),
        .clk(clk),
        .RD(ReadData),
        .rst(rst)
    );

    Mux Mux_Data_Memory_To_Register_File_Write_Back(
        .In1(ALU_RESULT),
        .In2(ReadData),
        .Selection(MemToReg),
        .Out(Mux_Data_Memory_To_Register_File_Write_Back_Result)
    );
endmodule
```

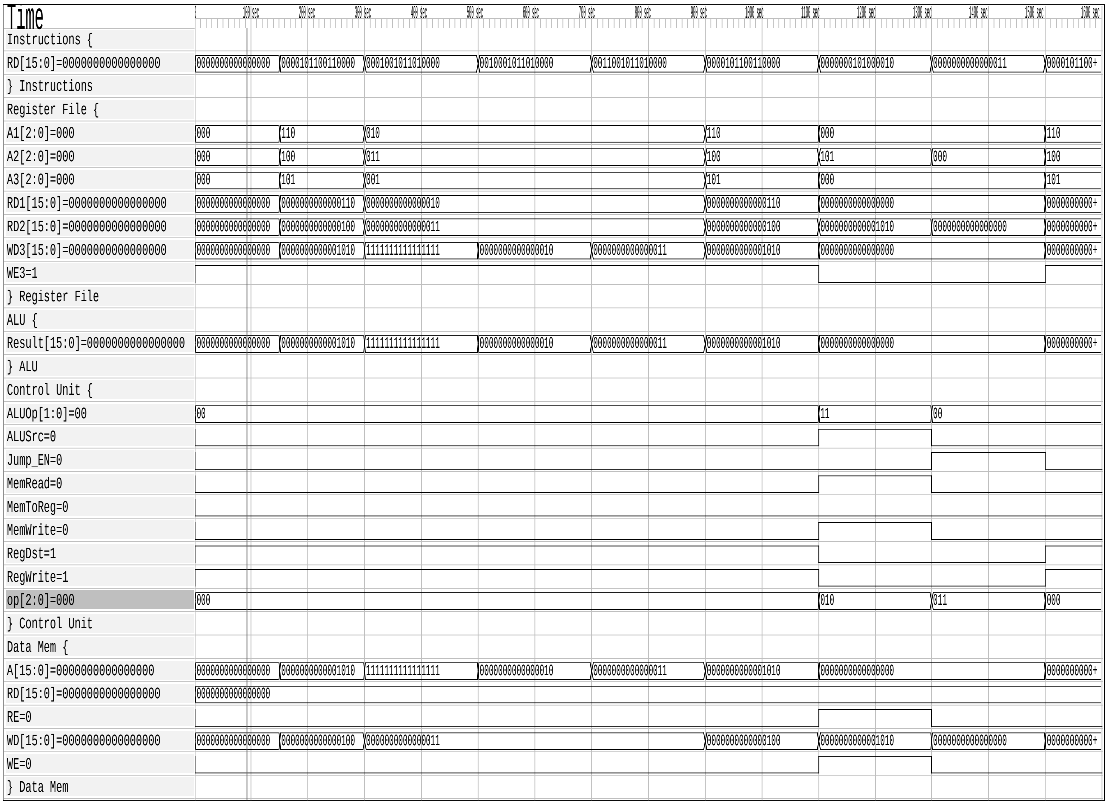
9 Testing

9.1 Output Wave Form









9.2 Test Bench

Code:

```verilog
`include "/home/baymax/Documents/Study/Air-Uni-Things/6th-Semester/Digital-System-Design/HEHEHE/Src/Single_Cycle_Top.v"
module Single_Cycle_Top_tb ();
    reg clk=1'b1,rst;

    Single_Cycle_Top Single_Cycle_Top (
        .clk(clk),
        .rst(rst)
    );
    always
        begin
            clk = ~clk;
            #100;
        end
    initial begin
        rst <= 1'b0;
        #150;

        rst <=1'b1;
        #1450;
        $finish;
    end
    initial begin
        $dumpfile("Single_Cycle.vcd");
        $dumpvars(0);
    end
endmodule
```

The following Instructions were given:

```verilog
initial begin
  //          Fun  RD RT RS OP  // RD = RS (Operator) RT
  mem[0] = 16'b0000101100110000; // add $5,$6,$4
  // mem[0] = 16'b0000001011010000; // add $1,$2,$3
  mem[1] = 16'b0001001011010000; // sub $1,$2,$3
  mem[2] = 16'b0010001011010000; // and $1,$2,$3
  mem[3] = 16'b0011001011010000; // or $1,$2,$3
  mem[4] = 16'b0000101100110000; // add $5,$6,$4
  mem[5] = 16'b0000000101000010; // sw $5,M[0]
  mem[6] = 16'b0000000000000011; // Jump 0
end
```

10 Conclusion

In conclusion, we have completed this complex engineering activity with the objective of gaining practical experience in designing and implementing a datapath using Verilog, a hardware description language. Our focus throughout this activity was to develop a deep understanding of the inner workings of a CPU's datapath and its various components.

Throughout this activity, we have worked with key elements of the datapath, including the Register File, Arithmetic Logic Unit (ALU), Control Unit, Instruction Memory, Program Counter (PC), Data Memory, and Instruction Decode. By actively engaging with these components, we have gained hands-on experience in designing and implementing the fundamental building blocks of a CPU.

The Register File serves as a storage space for data manipulation, while the ALU performs arithmetic and logical operations on that data. The Control Unit plays a crucial role in coordinating the overall operation of the datapath, ensuring proper sequencing and control flow. The Instruction Memory and Program Counter work together to fetch and execute instructions, while the Data Memory provides temporary storage for data during program execution. Lastly, the Instruction Decode phase translates the fetched instructions into control signals for the datapath components.

Through this comprehensive engineering activity, we have developed a thorough understanding of how the datapath functions and how its various components interact. This knowledge will empower us to tackle future projects and challenges in the field of computer engineering, as we can apply our expertise in designing and implementing CPU datapaths using Verilog.

References

[1] Muhammad Shameel Ansari and Hamza Shabbir. RISCV_Single_Cycle_Core, 1999. GitHub Repository: RISCV Single Cycle Core, YouTube: RISCV Single Cycle Core Playlist (https://www.youtube.com/watch?v=BVvDHhG0RoA&list=PL5AmAh9QoSK7Fwk9vOJu-3VqBng_HjGFc).

[2] FPGA4Students. Verilog code for 16-bit single cycle MIPS processor, 2017. (https://www.fpga4student.com/2017/01/verilog-code-for-single-cycle-MIPS-processor.html.