University of Moratuwa

Circuits and Systems
EN3030
Team Zigma

RISC-V processor Design

Group members :
Jayashinghe D.R. 190262L
Mathotaarachchi M.M. 190388D
Rukmal M.A.D. 190531L
Udara A.G.N. 190636M

Lecturer :
Dr. Ajith Pasqual

13/02/2023

Contents

1 Introduction

A processor is an integrated electronic circuit that performs the calculations that run a computer. A processor performs arithmetical, logical, input/output, and other basic instructions that are passed from an operating system. Most other processes are dependent on the operations of a processor.

As a fulfillment for EN3030 module project, we were tasked with designing and implementing a RISC-V RV32I processor with a direct mapping cache having victim cache functionality. This document is the final fulfillment of our project.

1.1 Functionality

We designed our processor to support single-cycle instructions and the instruction types that our processor supports are as follows.

- R Type
- I Type
- S Type
- SB Type
- U Type
- UJ Type

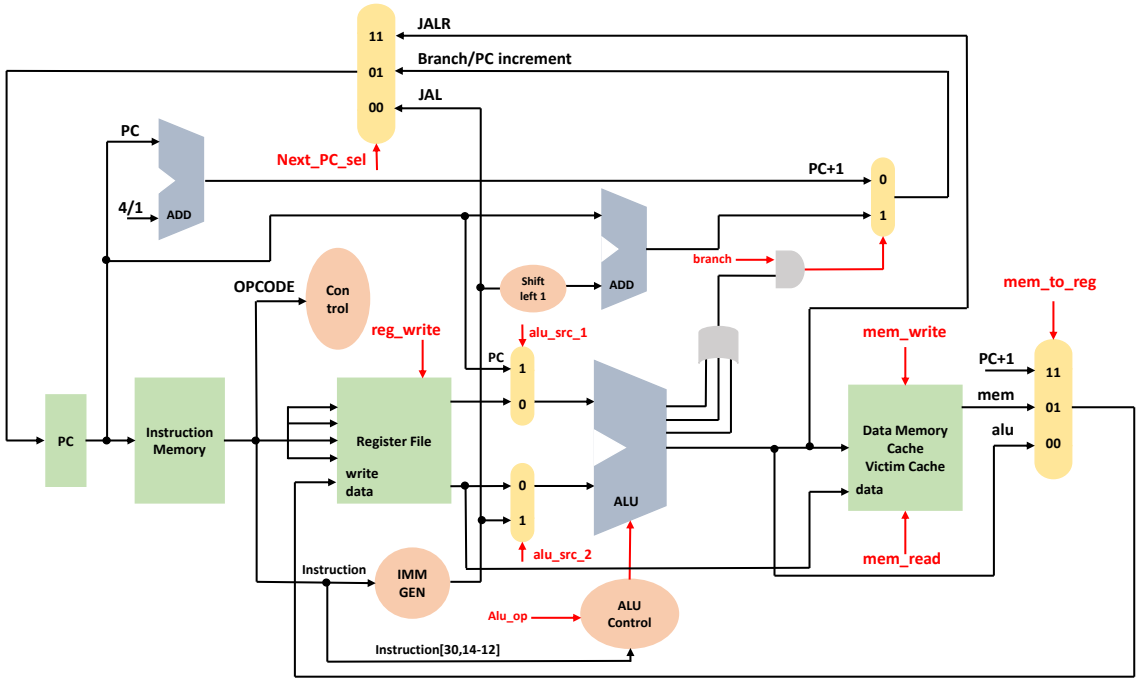

Figure 1: Datapath

2 Method

Our design comprises with following sub parts.

- Program counter
- PC selection and branching
- CLK input and management
- Main Control Unit
- Instruction Memory
- Instruction Decode
- Register File
- ALU input selection
- ALU
- ALU Control
- Data Memory

2.1 Program counter

In RISC-V, the program counter (PC) is a special register that holds the address of the instruction being executed. This is 32-bit in size. The PC is automatically incremented by 4 (for 32-bit implementations) after each instruction is executed, to keep track of the sequence of instructions in the program.

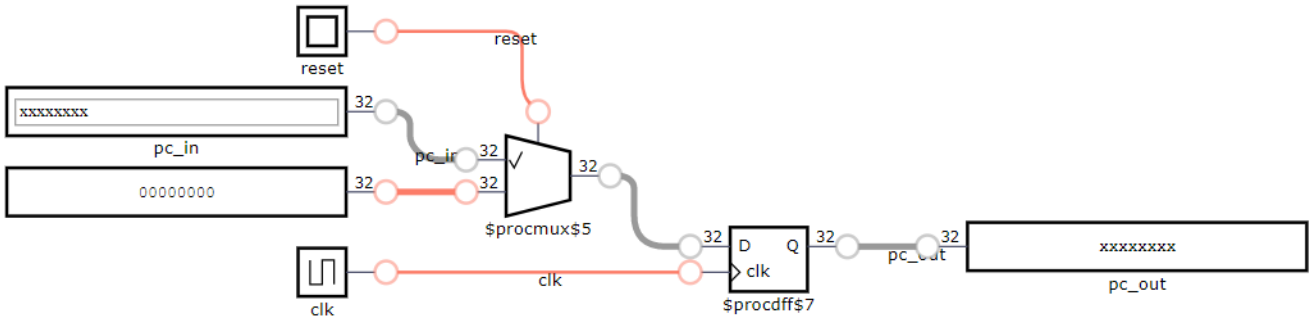

Figure 2: PC RTL view

The RISC-V architecture provides a set of instructions for manipulating the PC, including jump and branch instructions.The branch instructions can be used to change the flow of execution in a program by updating the PC with the address of a target instruction. In RISC-V, the PC is also used to implement subroutines and other control structures. When a subroutine is called, the current value of the PC is pushed onto the stack, and

the PC is updated with the address of the first instruction in the subroutine. When the subroutine returns, the PC is popped from the stack, returning control to the instruction called the subroutine.

In our implementation, PC is initiated to 0 and we have designed the instruction memory with the memory width of a word. So after the instruction is executed we need to increment the PC only by 1 (not by 4). The triggering signal to the PC is the clock. At the positive edge of the clock, the PC register gives the next PC value as the output. For increment the PC, a simple adder is used. This adder adds 1 to the current value of the PC and generates the new PC value.

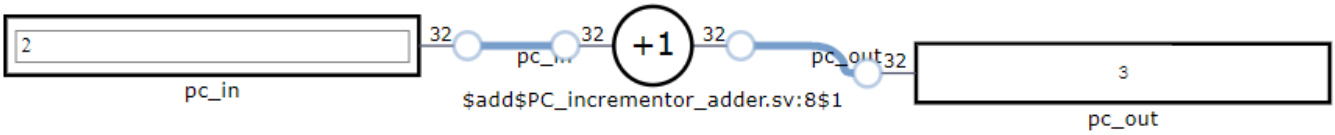

Figure 3: PC incrementor RTL view

2.2 PC selection and branching

Branching is a fundamental aspect of computer architecture, and it refers to the process of determining the next instruction to be executed based on the outcome of a previous instruction. In the RISC-V architecture, branching is performed using the program counter (PC) register.

In RISC-V, branching can occur in two ways: unconditional branching and conditional branching.

- Unconditional Branching: In unconditional branching, the program execution is transferred to a specified address regardless of the outcome of any previous instruction. This is achieved by loading the desired target address into the PC register."jal" and "jalr" are the two instructions responsible for this.
- Conditional Branching: In conditional branching, the program execution is transferred to a specified address only if a certain condition is met. This is achieved using conditional branches, such as "beq" (branch if equal) or "bne" (branch if not equal). These instructions compare the values in two registers, and based on the result of the comparison, the PC register is updated with the target address or the next instruction address is fetched from the current PC value.

In both cases, the branching operation is performed by updating the value in the PC register. So there should be a separate adder to increment the PC count by a supplied value from the instruction. For that, an adder was implemented to add the immediate offset to the current PC value and calculate the next PC value.

For branching instructions, we raised three flags from ALU.

- "eq" - branch if r1 and r2 register values are equal.
- "a_lt_b" - branch if r1 value is less than r2 value .
- "a_lt_ub" - branch if unsigned r1 value less than unsigned r2 value.

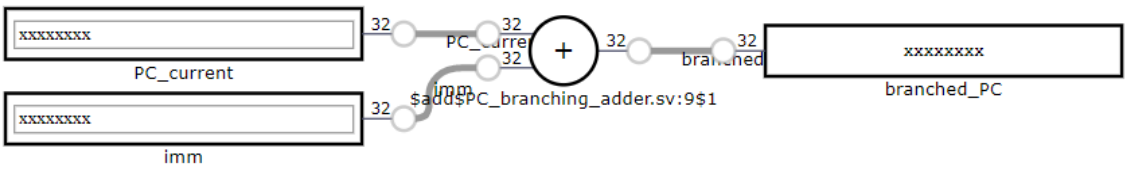

Figure 4: Immediate offset adder to PC RTL view

The logic was designed to generate a branching command if at least one flag is raised with branch control signal from the control unit.

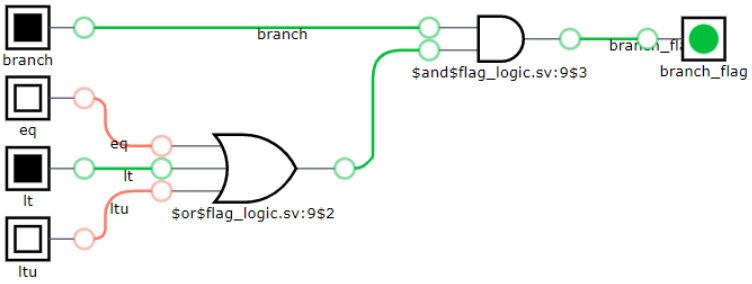

Figure 5: Branching logic RTL view

The branching command is used to select how next PC address is calculated with a help of multiplexer.

Branch command	**Operation**
0	PC increment by 1
1	PC shift by given offset

Other than branching instructions, unconditional jump instructions also change the PC value by a given amount. So we had to choose the next PC value from "branched_PC", "JAL" and "JALR". For this, we generated 'next_pc_sel' command from control logic and gives the correct next PC value to PC with the help of another multiplexer.

next_pc_sel	**next_PC**
10	JALR
00	PC+imm or PC+1
01	JAL

2.3 CLK and input management

The inputs for the RISC V are,

- Input clock.
- Reset.

- Instructions.

The Input clock provides timing sense for the RISC V and the reset pin input provides the reset functionality for the register bank and PC counter. It will initiate the modules to be 0. The instructions are provided via a .mif file containing HEX codes for the program. You can pass the instruction data in the real-time via Quartus In-System Memory Content Editor. The memory is created using an IP core of a 1 port RAM. The clock is provided via a high 50MHZ for the module to get combinations logic behaviour. The base clock for the other modules are 50KHZ. A clock divider is used for the clock slowing. We have to downgrade the clock to provide enough timing for the combinations delays.

2.4 Instruction Memory

A Single clocked, 1 port RAM IP core from the Quartus to use an inbuilt memory module. This provides the ability to Read and write data to the registers in run time. There will be 32Bit instructions and 1024 total instructions at a time. This can be extended if there is need to be.

2.5 Instruction Decode

With the different type of instructions, the instruction encoding is different. We can differentiate them mostly by the 7 bit opcode. We have identified 9 types of instructions,

- Arithmetic and Shift Operations based on R instructions.
- Immediate Operations with Arithmetic and logic operations.
- Load Operations, with similar syntax to Immediate Operations.
- Store Operations.
- Branch - Jump Operations.
- J type.
- JALR with similar syntax to Immediate Operations.
- Upper type, LUI.
- Upper AUIPC.

2.6 Register Bank

With Total of 32 registers, our design have only one hardwired register which provides integer 0, located at address 0. All other registers are to be used as anyway user like. There are no constrains.

Decoded data from instructions will provide input addresses for the module. The Write enable is provided by the Main Control Unit. With a clock event, after the all calculations are done, we will write to the provided address, if the instruction is required to do so.

Inputs	
Register Address 1	For register output selection
Register Address 2	For register output selection
Write Address	Where to be stored the data
Write Enable	If writing needs to be done
Write Data	The data which need to be stored
Clock	Timing event for the Data writing process
Outputs	
Data 1	Data stored at location of Address 1
Data 2	Data stored at location of Address 2

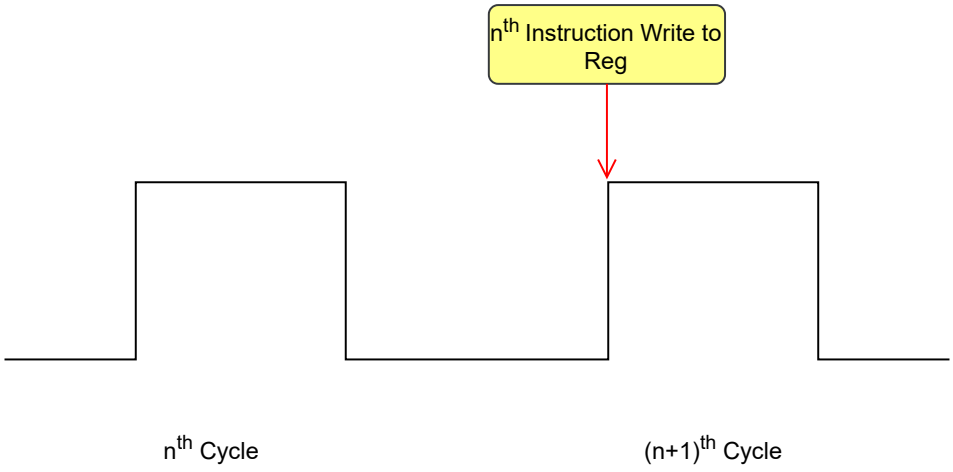

Figure 6: Clock Details for reg

In the single-cycle design, we have to restrict clock usage as much as possible. Here, the clock trigger is used only for register write.

Since the PC increment also happen at the same clock trigger, with the path delays, the access of the written data from the previous instruction writing will not be a problem.

2.7 ALU input selection

For the inputs of ALU, We have to choose between 4 inputs.

- Register input 1
- PC
- Register input 2
- Immediate value

There are two independent 2×1 multiplexers to Select for two inputs of the ALU unit. The multiplexers are controlled with 2 flags provided by the control unit and the flags are dependent on the opcodes. For further clarifications refer to the Main controller section.

2.8 ALU

To execute all the Arithmetic and Logic tasks except for the PC increment and branching increment, we rely on the ALU of our processor.
It has total of 11 tasks. Getting control signals from ALU control unit with help of 4bit *FCODE*, we can differentiate those unique tasks.

Task	**FCODE**
Add Two Numbers	0000
Subtract Two Numbers	0001
Left Shift One Number	0010
Signed Compare	0011
Unsigned Compare	0100
XOR bitwise	0101
Right Shift	0110
Signed Right Shift	0111
OR	1000
AND	1001
Address Save and Jump	1010

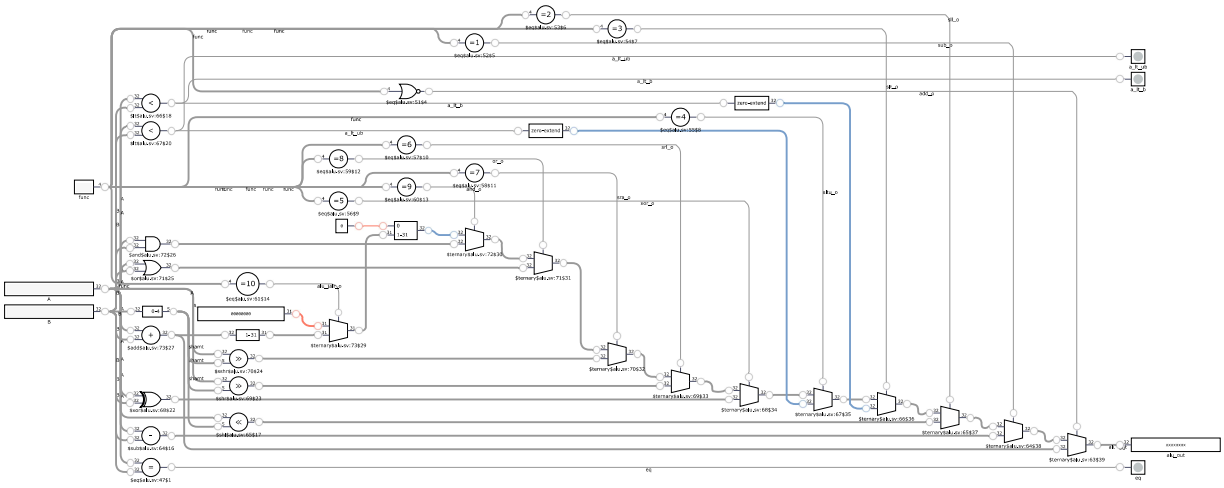

Figure 7: ALU RLT view

2.9 Data Memory

RISC-V is a load-store architecture, which means that all data processing operations are performed on data stored in registers. The data memory of a RISC-V processor consists of two types of memory: cache memory(with victim cache functionality) and main memory. Cache memory is a small, fast memory that is used to temporarily store frequently accessed data. It is typically divided into multiple levels, with each level having a larger size and a longer access time than the previous level. The purpose of the cache memory is to reduce the time it takes to access data from the main memory, which is slower. Main memory is the primary storage area for data in a RISC-V processor. It is typically

composed of DRAM (dynamic random access memory) chips and is organized into a hierarchy of memory banks, with each bank consisting of multiple memory modules. The size and speed of the main memory depend on the specific implementation of the RISC-V processor.
A victim cache is a hardware cache designed to decrease conflict misses and improve hit latency for direct-mapped caches. It is employed at the refill path of a Level 1 cache, such that any cache line which gets evicted from the cache is cached in the victim cache.
The RISC-V architecture provides instructions for loading data from memory into registers and storing data from registers into memory. These instructions can be used to read and write data from the cache memory and main memory.
Two control signals and two inputs (one from ALU and one from the register file) come into the data memory and one output comes out from the data memory.

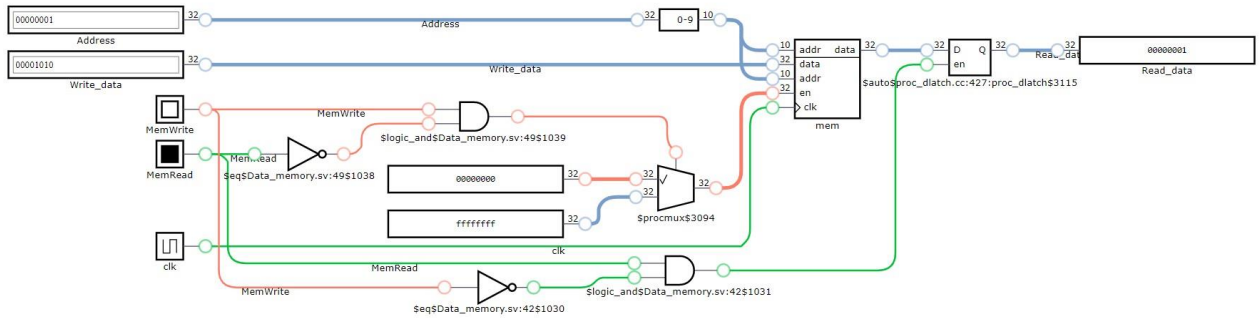

Figure 8: Data Memory RTL view

3 Control Signals

The control signals for the datapath are generated by 2 units; main controller and ALU controller.

3.1 Main Controller

The main controller gives the necessary control signals to the modules with the single cycle datapath. The main control unit takes as input the 7-bit opcode from the instruction and it outputs 9 control signals. The following table contains the control signal types and their destinations.

Control Signal	Signal bit size	Function
alu_src_1	1	Select the first input to the ALU (data from register file or PC)
alu_src_2	1	Select the second input to the ALU (data from register file or immediate value from instruction)
mem_to_reg	2	Select which data is written to the register file at the mux (data from memory, ALU output or incremented PC)
reg_write	1	Enable data write to registerfile
mem_read	1	Enable memory read for load instruction
mem_write	1	Enable memory write for store instruction
branch	1	Enable branching
alu_op	2	This signal is sent to the ALU controller indicating the ALU operation type
next_pc_sel	2	Selects the next value of the PC

Table 1: Main control signals

3.2 ALU Controller

The ALU controller takes 2 inputs. One input is the 2-bit *alu_op* which comes from the main control unit and other input is 4 bits decoded from the intructions. These 4 bits comes from the *function 3 (Instruction[14:12])* and the *6th bit of function 7 (Instruction[30])* in the instruction. Based on the input signals the ALU controller will output a 4-bit signal which will correspond to the ALU function required by the instruction such as addition, subtraction, bit shifting, comparison, etc.

The ALU OP from main controller categorises the signals into 4 types based on the opcode.

ALU Op	Relevant instructions
00	I-type (load) and S-type
01	SB-type (branching)
10	R-type and I-type (execpt load)
11	U-type and J-type

Table 2: ALU OP signal taken as input to ALU controller

4 Discussion

4.1 Verifying the Funcationality

Designing individual modules

We took a modular approach to processor design by initally designing the individual functional blocks such as the ALU, resigster file, controller, memory, etc. We used System Verilog as our HDL. We tested them individually using test benches on EDA simulation tools avaible on Xilinx Vivado and Modelsim Altera.

Creating the top module and simulations

After verifying the functionality of the individual blocks we moved on to connecting them in the top module. Here had to resolve several issues related to setting the clock. After that, we moved on to testing the top module using a testbench. Since our datapath was complete, we started testing our instructions. During this stage of testing, we were able to find some issues with modules which we have not encountered before, as well as connectivity issues. After resolving them we were able to successfully execute a sequence of instructions and verify the functionality through simulation.

FPGA implementation

We were given the Altera DE2-115 FPGA board. We used the Quartus II to program the FPGA. The main issue we encountered during our FPGA implemetation was with the original 50MHz clock begin too fast. We used a clock divider module and got the clock down to a suitable value. We found that the maximum clock at which we could execute instructions without any timing issue was about 400Khz. After resolving some other minor error, we were able to successfully run several programs in our processor. We tested a few codes that covered the range of our instructions. To read the internal values of the registers during runtime, we used the tool SingalTap Logic Analyser avaible in Quartus Environment. As, for the instrction memory, we used a RAM block avaiable in Quartus to load the instructions.

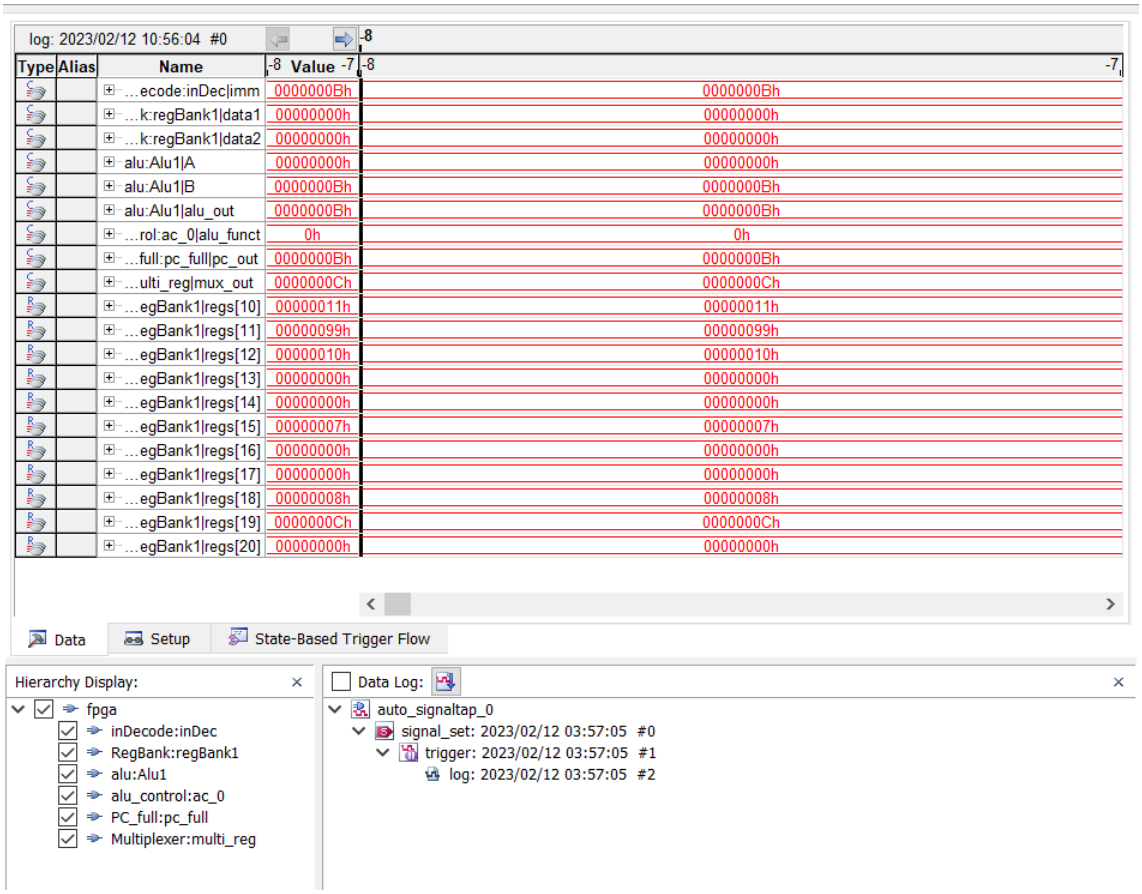

Figure 9: Signal Tap Logic Analyser Tool displaying current register values

4.2 Processor Specifications

Instruction architecture	RISCV
Number of supported RISC-V instructions	28
Supported instructions types	R, I, S, B, U, J types
Maximum clock	200kHz
Total logic elements	66 748
Total registers	40 365

Table 3: Processor Specifications

Flow Summary	
Flow Status	Successful - Sun Feb 12 23:55:58 2023
Quartus II 64-Bit Version	13.1.0 Build 162 10/23/2013 SJ Web Edition
Revision Name	riscv_processor
Top-level Entity Name	Zigma_RISCV
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	43,179 / 114,480 (38 %)
Total combinational functions	26,804 / 114,480 (23 %)
Dedicated logic registers	33,963 / 114,480 (30 %)
Total registers	33963
Total pins	2 / 529 (< 1 %)
Total virtual pins	0
Total memory bits	32,768 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 10: Flow Summary after compilation

5 Conclusion

The RISC-V processor we designed can be used for a range of applications mainly based on integer arithmetic. With our testing we were able to verify the possibility of executing programs containing control loops, arrays, etc.

By extending this instruction set, this processor can be made to support system instructions and floating point numbers. Through optimization, the processor can be made to run at a higher clock. By implementing the datapath using pipe-lining instead of single cycle design, the CPU time can be greatly increased.

6 Appendix

6.1 Verilog codes

Program counter

```
`timescale 1ns / 1ps

module PC(
        input clk,
        input reset, //reset
    input [31:0] pc_in,
    output reg [31:0] pc_out
    );

initial begin
        pc_out<=0;
        end

always @(posedge clk)      //Assigns pc_out equal to pc_in at every clock cycle
        begin
                if(reset==1)
                begin
                        pc_out<=0;
                end
                else
                begin
                        pc_out<=pc_in;
                end
        end

endmodule
```

PC Incrementor Adder

```
`timescale 1ns / 1ps

module PC_incrementor(
    input [31:0] pc_in,
    output [31:0] pc_out
    );

assign pc_out=pc_in+32'b00000000000000000000000000000001;  //increment PC val by 1

endmodule
```

PC Incrementor selecting multiplexer

```
module mux_4(
    input [31:0] jal,
    input [31:0] br_pc,
    input [31:0] jalr,
    input [1:0] sel,
    output reg [31:0] y
    );

   always @(*) begin
      case (sel)
         2'b00: y = br_pc;
         2'b01: y = jal;
         2'b10: y = jalr;
      endcase
   end
endmodule
```

PC Branching Adder

```
`timescale 1ns / 1ps

module adder(
    input [31:0] PC_current,
    input [31:0] imm,
    output [31:0] branched_PC
    );

assign branched_PC=PC_current+imm; //add PC and immediate
//(assumed imm incremented by 4)

endmodule
```

Flag Logic

```
module flag_logic(
    input eq,
    input lt,
    input ltu,
    input branch,
    output branch_flag
);

assign branch_flag = branch & (eq | lt | ltu);


endmodule
```

Register Bank

```
`timescale 1ns / 1ps



module RegBank(
    input             clk,
    input      [ 4:0] r_addr1,
    input      [ 4:0] r_addr2,
    input      [ 4:0] w_addr,
    input             write_en,
         input                                reset,
    input      [31:0] write_data,
    output reg [31:0] data1, //to alu directly
    output reg [31:0] data2 // send to the input selector between imm or reg
    );

    reg [31:0] regs[31:0]; // registers, there are 32 total.

   //Initially make all registers 0
    //writing data in the address.

    always @(posedge clk) // will execute on next pc update. same clock
        begin
                  if (reset) begin
                        regs[0]=32'b0;                regs[16]=32'b0;
            regs[1]=32'b0;                regs[17]=32'b0;
            regs[2]=32'b0;                regs[18]=32'b0;
            regs[3]=32'b0;                regs[19]=32'b0;
            regs[4]=32'b0;                regs[20]=32'b0;
            regs[5]=32'b0;                regs[21]=32'b0;
            regs[6]=32'b0;                regs[22]=32'b0;
            regs[7]=32'b0;                regs[23]=32'b0;
            regs[8]=32'b0;                regs[24]=32'b0;
            regs[9]=32'b0;                regs[25]=32'b0;
            regs[10]=32'b0;                regs[26]=32'b0;
            regs[11]=32'b0;                regs[27]=32'b0;
            regs[12]=32'b0;                regs[28]=32'b0;
            regs[13]=32'b0;                regs[29]=32'b0;
            regs[14]=32'b0;                regs[30]=32'b0;
            regs[15]=32'b0;                regs[31]=32'b0;


                  end

        else begin
                        if ((write_en == 1'b1) && (w_addr != 5'b00000))
```

```
                                        begin
                                                regs[w_addr] = write_data;
                            end
                           end
        end

    always @(*)
        begin
            data1 = regs[r_addr1];
            data2 = regs[r_addr2];
        end


endmodule
```

Memory multiplexer

```
`timescale 1ns / 1ps

module Multiplexer(
    input [31:0] Read_data,
    input [31:0] Address,
    input [31:0] PC4,
    input [1:0] MemtoReg,
    output reg [31:0] mux_out
    );

    always @(*) begin
        case (MemtoReg)
          2'b00: mux_out = Address;
          2'b01: mux_out = Read_data;
          2'b11: mux_out = PC4;
          default: mux_out = 32'b0;
        endcase
    end
endmodule
```

Branching multiplexer

```
`timescale 1ns / 1ps

module mux(
    input [31:0] in0,
    input [31:0] in1,
    input sel,
    output [31:0] out
    );
```

```
assign out=sel?in1:in0;   //if sel is 1, choose in1 else choose in0

endmodule
```

Instruction Memory

```
`timescale 1 ps / 1 ps
// synopsys translate_on
module myTop (
        address,
        clock,
        data,
        wren,
        q);

        input         [9:0]  address;
        input          clock;
        input         [31:0]  data;
        input          wren;
        output        [31:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
        tri1           clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

        wire [31:0] sub_wire0;
        wire [31:0] q = sub_wire0[31:0];

        altsyncram        altsyncram_component (
                              .address_a (address),
                              .clock0 (clock),
                              .data_a (data),
                              .wren_a (wren),
                              .q_a (sub_wire0),
                              .aclr0 (1'b0),
                              .aclr1 (1'b0),
                              .address_b (1'b1),
                              .addressstall_a (1'b0),
                              .addressstall_b (1'b0),
                              .byteena_a (1'b1),
                              .byteena_b (1'b1),
                              .clock1 (1'b1),
                              .clocken0 (1'b1),
                              .clocken1 (1'b1),
                              .clocken2 (1'b1),
```

```
                                .clocken3 (1'b1),
                                .data_b (1'b1),
                                .eccstatus (),
                                .q_b (),
                                .rden_a (1'b1),
                                .rden_b (1'b1),
                                .wren_b (1'b0));
        defparam
                altsyncram_component.clock_enable_input_a = "BYPASS",
                altsyncram_component.clock_enable_output_a = "BYPASS",
                altsyncram_component.init_file = "instr.mif",
                altsyncram_component.intended_device_family = "Cyclone IV E",
                altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=
                altsyncram_component.lpm_type = "altsyncram",
                altsyncram_component.numwords_a = 1024,
                altsyncram_component.operation_mode = "SINGLE_PORT",
                altsyncram_component.outdata_aclr_a = "NONE",
                altsyncram_component.outdata_reg_a = "UNREGISTERED",
                altsyncram_component.power_up_uninitialized = "FALSE",
                altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE
                altsyncram_component.widthad_a = 10,
                altsyncram_component.width_a = 32,
                altsyncram_component.width_byteena_a = 1;


endmodule
```

Instruction Decoder

```
`timescale 1ns / 1ps

module inDecode (
    input      [31:0] inst,
    output reg [ 6:0] opcode,
    output reg [ 4:0] rsAddr,    //rs1
    output reg [ 4:0] rdAddr,    //rd
    output reg [ 4:0] shamt,     //SHIFT VAL AND RS2
    output reg [ 3:0] alu_func,  // FOR ALU_contoller
    //func_3 + 1 bit for add sub chnage like things
    output reg [31:0] imm        // i TYPE  imm and branching
    //output reg [31:0] label    // for braching
);

  always @(*) begin
    opcode = inst[6:0];  //Opcode is always the first 7 bits

    if(opcode==7'b0110011)        //Arithmetic and Shift Operations
                begin
```

```
  rsAddr   = inst[19:15];
  rdAddr   = inst[11:7];
  shamt    = inst[24:20];
  alu_func = {inst[30], inst[14:12]};
  imm      = 32'b0;
  //label    = 32'b0;
end

    else if(opcode==7'b0010011)   //Immediate Operations
            begin
  rsAddr   = inst[19:15];
  rdAddr   = inst[11:7];
  shamt    = 5'b00000;  // no need second address
  alu_func = {1'b0, inst[14:12]};
  imm      = $signed(inst[31:20]);
  //label    = 32'b0;  //only for U and UJ
end
    else if(opcode==7'b0000011)
            begin
  rsAddr   = inst[19:15];  //rs1
  rdAddr   = inst[11:7];  //rd
  shamt    = 5'b00000;
  alu_func = {1'b0, inst[14:12]};
  imm      = $signed(inst[31:20]);
  // label    = 32'b0;
end

else if(opcode==7'b0100011)
            begin
  rsAddr   = inst[19:15];  //rs1
  rdAddr   = 5'b00000;    //inst[23:19];
  shamt    = inst[24:20];  // We need this
  alu_func = {1'b0, inst[14:12]};
  imm      = $signed({inst[31:25], inst[11:7]});
  // label    = 32'b0;  //No need this

end

    else if(opcode==7'b1100011)    //Branch/Jump Operations
            begin
  rsAddr   = inst[19:15];  // rs1 adddress
  rdAddr   = 5'b0;
  shamt    = inst[24:20];
  alu_func = {1'b0, inst[14:12]};
  imm      = $signed({ inst[31], inst[7], inst[30:25], inst[11:8], 1'b0});
  // label    = 32'b0;
end
```

```
    else if(opcode==7'b1101111)    //J - JAL
            begin
  rsAddr   = 5'b0;
  rdAddr   = inst[11:7];
  shamt    = 5'b0;
  alu_func = 4'b0;
  imm      = {inst[31], inst[19:12], inst[20], inst[30:21], 1'b0};
  // imm      = 32'b0;
  // label    = {inst[31], inst[19:12], imm[20], imm[30:21], 1'b0};
end

else if(opcode==7'b1100111)    //I-type  JALR
            begin
  rsAddr   = inst[19:15];
  rdAddr   = inst[11:7];
  shamt    = 5'b00000;  // no need second address
  alu_func = {1'b0, inst[14:12]};
  imm      = $signed(inst[31:20]);
  // label    = 32'b0;  //only for U and UJ
end

else if(opcode==7'b0110111)    //Upper LUI
            begin
  rsAddr   = 5'b0;
  rdAddr   = inst[11:7];
  shamt    = 5'b0;
  alu_func = 4'b0;
  imm      = $signed({inst[31:12], 12'b0});
  // imm      = 32'b0;
  // label    = $signed({inst[31:12], 12'b0});
end

else if(opcode==7'b0010111)    //Upper AUIPC
            begin
  rsAddr   = 5'b0;
  rdAddr   = inst[11:7];
  shamt    = 5'b0;
  alu_func = 4'b0;
  imm      = $signed({inst[31:12], 12'b0});
  // imm      = 32'b0;
  // label    = $signed({inst[31:12], 12'b0});
end

else  //invalid instruction0
begin
  rsAddr   = 5'b0;
```

```
        rdAddr   = 5'b0;
        shamt    = 5'b0;
        alu_func = 4'b0;
        imm      = 32'b0;
        //label    = 32'b0;
      end
    end

endmodule
```

Main control

```
`timescale 1ns / 1ps

module main_control(
                        input [6:0]                      opcode,
                        output reg                       alu_src_1,
                        output reg                       alu_src_2,
                        output reg[1:0]        mem_to_reg,
                        output reg                       reg_write,
                        output reg                       mem_read,
                        output reg                       mem_write,
                        output reg                       branch,
                        output reg[1:0]        alu_op,
                        output reg[1:0]        next_pc_sel
                                                );

        always @ (*) begin
                if (opcode == 7'b0110011) begin
                        /*Control signals for R-type & I-type(immediate)*/
                        alu_src_1                 = 1'b0;
                        alu_src_2                = 1'b0;
                        mem_to_reg                = 2'b00;
                        reg_write                 = 1'b1;
                        mem_read                        = 1'b0;
                        mem_write                = 1'b0;
                        branch                        = 1'b0;
                        alu_op                        = 2'b10;
                        next_pc_sel                = 2'b00;
                end else if (opcode == 7'b0010011) begin
                        /*I-type(immediate)*/
                        alu_src_1                 = 1'b0;
                        alu_src_2                = 1'b1;
                        mem_to_reg                = 2'b00;
                        reg_write                 = 1'b1;
                        mem_read                        = 1'b0;
                        mem_write                = 1'b0;
                        branch                        = 1'b0;
```

```
            alu_op                          = 2'b10;
            next_pc_sel                   = 2'b00;
        end else if (opcode == 7'b0000011) begin
            /*Control signals for I-type(load)*/
            alu_src_1                     = 1'b0;
            alu_src_2                    = 1'b1;
            mem_to_reg                     = 2'b01;
            reg_write                     = 1'b1;
            mem_read                            = 1'b1;
            mem_write                    = 1'b0;
            branch                           = 1'b0;
            alu_op                           = 2'b00;
            next_pc_sel                   = 2'b00;
        end else if (opcode == 7'b0100011) begin
            /*Control signals for S-type*/
            alu_src_1                     = 1'b0;
            alu_src_2                    = 1'b1;
            mem_to_reg                     = 2'b00;        /*don't care*/
            reg_write                     = 1'b0;
            mem_read                    = 1'b0;
            mem_write                     = 1'b1;
            branch                           = 1'b0;
            alu_op                           = 2'b00;
            next_pc_sel                   = 2'b00;
        end else if (opcode == 7'b1100011) begin
            /*Control signals for SB-type*/
            alu_src_1                     = 1'b0;
            alu_src_2                    = 1'b0;
            mem_to_reg                     = 2'b00;
            reg_write                     = 1'b0;
            mem_read                    = 1'b0;
            mem_write                     = 1'b0;
            branch                           = 1'b1;
            alu_op                           = 2'b01;
            next_pc_sel                   = 2'b00;
        end else if (opcode == 7'b1101111) begin
            /*JAL instrution (UJ-type)*/
            alu_src_1                     = 1'b0;
            alu_src_2                    = 1'b0;
            mem_to_reg                     = 2'b11;
            reg_write                     = 1'b1;
            mem_read                    = 1'b0;
            mem_write                     = 1'b0;
            branch                           = 1'b0;
            alu_op                           = 2'b11;
            next_pc_sel                   = 2'b01;
        end else if (opcode == 7'b1100111) begin
```

```
                /*JALR instruction (I-type)*/
                alu_src_1                = 1'b0;
                alu_src_2                = 1'b1;
                mem_to_reg               = 2'b11;
                reg_write                = 1'b1;
                mem_read                 = 1'b0;
                mem_write                = 1'b0;
                branch                   = 1'b0;
                alu_op                   = 2'b11;
                next_pc_sel              = 2'b10;
            end else if (opcode == 7'b0110111) begin
                /*LUI instruction (U-type)*/
                alu_src_1                = 1'b0;
                alu_src_2                = 1'b1;
                mem_to_reg               = 2'b00;
                reg_write                = 1'b1;
                mem_read                 = 1'b0;
                mem_write                = 1'b0;
                branch                   = 1'b0;
                alu_op                   = 2'b11;
                next_pc_sel              = 2'b00;
            end else if (opcode == 7'b0010111) begin
                /*AUIPC instuction (U-type)*/
                alu_src_1                = 1'b1;
                alu_src_2                = 1'b1;
                mem_to_reg               = 2'b00;
                reg_write                = 1'b1;
                mem_read                 = 1'b0;
                mem_write                = 1'b0;
                branch                   = 1'b0;
                alu_op                   = 2'b11;
                next_pc_sel              = 2'b00;
            end else begin
                /*Keep high impedence for any other input*/
                alu_src_1                = 1'bz;
                alu_src_2                = 1'bz;
                mem_to_reg               = 2'bzz;
                reg_write                = 1'bz;
                mem_read                 = 1'bz;
                mem_write                = 1'bz;
                branch                   = 1'bz;
                alu_op                   = 2'bzz;
                next_pc_sel              = 2'bzz;
            end
        end
endmodule
```

ALU Input Selection Multiplexer - 1

```
`timescale 1ns / 1ps


module INPUT_SEL(
    input         ALUSrc1,       // ALUSrc is Control unit flag to select the Input,
    input  [31:0] IMMval,       // IF 1, select imm value
    input  [31:0] reg_val,      // If 0, select register out put
    output [31:0] AluInput2
    );

    assign AluInput2 = ALUSrc1 ? IMMval:reg_val;

endmodule
```

ALU Input Selection Multiplexer - 2

```
`timescale 1ns / 1ps


module INPUT_SEL_2(
    input         ALUSrc2,       // ALUSrc is Control unit flag to select the Input,
        input  [31:0] Pc,       // IF 1, select imm value
        input  [31:0] reg_val,      // If 0, select register out put
        output [31:0] AluInput1
        );

        assign AluInput1 = ALUSrc2 ? Pc:reg_val;
endmodule
```

ALU Control

```
`timescale 1ns / 1ps

module alu_control (        input        [1:0]                          alu_op,
                                                 input        [3:0]
                                                 output reg        [3:0]

        wire        [5:0]        temp;

        assign temp = {alu_op, alu_funct};

        always @(*) begin
                if (temp == 6'b10_0000 | temp == 6'b00_0010 | temp == 6'b11_0000) begi
                        /* ADD for add, addi, lw, sw instructions */
                        alu_ctrl = 4'b0000;
```

```
end else if (temp == 6'b10_1000) begin
        /* SUB */
        alu_ctrl = 4'b0001;

end else if (temp == 6'b01_0001) begin
        /* SLL */
        alu_ctrl = 4'b0010;

end else if (temp == 6'b10_0010) begin
        /* SLT -> slt & slti*/
        alu_ctrl = 4'b0011;

end else if (temp == 6'b10_0011) begin
        /* ALU SLTU */
        alu_ctrl = 4'b0100;

end else if (temp == 6'b10_0100) begin
        /* XOR */
        alu_ctrl = 4'b0101;

end else if (temp == 6'b10_0101) begin
        /* SRL */
        alu_ctrl = 4'b0110;

end else if (temp == 6'b10_1101) begin
        /* SRA */
        alu_ctrl = 4'b0111;

end else if (temp == 6'b10_0110) begin
        /* OR */
        alu_ctrl = 4'b1000;

end else if (temp == 6'b10_0111) begin
        /* AND */
        alu_ctrl = 4'b1001;

end else if (temp == 6'b01_0000) begin
        /* BEQ */
        alu_ctrl = 4'b1011;

end else if (temp == 6'b01_0100) begin
        /* BLT */
        alu_ctrl = 4'b1100;

end else if (temp == 6'b01_0110) begin
        /* BLTU */
        alu_ctrl = 4'b1101;
```

```
                end else begin
                        alu_ctrl = 4'bzzzz;
                end
        end
endmodule
```

ALU

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Zigma
// Engineer: Nuwan Udara
//
// Create Date: 02.02.2023 14:21:28
// Design Name:
// Module Name: alu
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module alu(alu_out, eq, a_lt_b, a_lt_ub, func, A, B);

        input [3:0] func;
        input [31:0] A;
        input [31:0] B;
        output[31:0] alu_out;
        output eq, a_lt_b, a_lt_ub;// sign?

        wire add_o, sub_o, sll_o, slt_o, sltu_o, xor_o, srl_o, sra_o, or_o, and_o, alu

    parameter
       func_ADD             = 4'b0000,
                func_SUB            = 4'b0001,
                func_SLL           = 4'b0010,
                func_SLT             = 4'b0011,
                func_SLTU            = 4'b0100,
                func_XOR           = 4'b0101,
                func_SRL           = 4'b0110,
```

```
                func_SRA           = 4'b0111,
                func_OR            = 4'b1000,
                func_AND           = 4'b1001,
                func_ADD_JALR = 4'b1010,
                func_BEQ                        = 4'b1011,
                func_BLT                = 4'b1100,
                func_BLTU                = 4'b1101;

    wire [4:0] shamt = B[4:0];

    //assign eq = A==B;
    assign a_lt_b = 1'b0;//$signed(A) < $signed(B);
    assign a_lt_ub = 1'b0;

assign add_o      = func==func_ADD;
assign sub_o      = func==func_SUB;
assign sll_o      = func==func_SLL;
assign slt_o      = func==func_SLT;
assign sltu_o     = func==func_SLTU;
assign xor_o      = func==func_XOR;
assign srl_o      = func==func_SRL;
assign sra_o      = func==func_SRA;
assign or_o       = func==func_OR;
assign and_o      = func==func_AND;
assign alu_jalr_o = func==func_ADD_JALR;
     assign beq_0               = func==func_BEQ;
     assign blt_0               = func==func_BLT;
     assign bltu_0               = func==func_BLTU;

     assign eq = beq_0 ? A==B:
                                 blt_0 ? $signed(A) < $signed(B):
                                 bltu_0 ? A<B:
                                 1'b0;


    assign alu_out = add_o ? A + B :
                     sub_o ? $signed(A - B) :
                     sll_o ? A << shamt :
                     slt_o ? ($signed(A) < $signed(B) ? {{32-1{1'b0}},1'b1} : 0)
                     sltu_o ? (A < B ? {{32-1{1'b0}},1'b1} : 0) :
                     xor_o ? A ^ B :
                     srl_o ? A >> shamt :
                     sra_o ? $signed(A) >>> shamt:
                     or_o ? A | B :
                     and_o ? A & B :
                     alu_jalr_o ? (A + B) & 32'hFFFFFFFE :
                     32'b0;
```

```
endmodule
```

Data Memory

```
`timescale 1ns / 1ps

module Data_memory(
          input clk,
    input [31:0] Address,
    input [31:0] Write_data,
    input MemRead,
    input MemWrite,
    output reg [31:0] Read_data
    );

          reg [31:0] mem [1023:0]; // 1024 32-bit words of memory


          initial begin
      for (int i = 0; i < 1024; i++)
      begin
              mem[i] = 0;
              end
              //mem[6] =32'b11101;

    end

  always @(*)
    begin
            if (MemRead == 1) begin
                Read_data = mem[Address[9:0]];
              end
    end

  always @(posedge clk)
    begin
      if (MemWrite == 1) begin
        mem[Address[9:0]] <= Write_data;
      end

    end
endmodule
```

Processor (Top Module)

```
`timescale 1ns / 1ps
/****************** This is the TOP module*****************/
```

```
module Zigma_RISCV(
                input clk_max,
                input reset
    );

        wire clk;        //clock after dividing
        Clock_divider clk_div0(
                .clock_in        (clk_max),
                .clock_out        (clk)
        );

        wire [31:0] pc_number;
        wire [31:0] data = 32'h0;
        wire wren = 0;                /*write diables for instruction memory*/
        wire [31:0]        inst_out_inst_Mem;

        /**********Instruction Memory**********/
        /*implemented in RAM module*/
         myTop instr_mem0(
                .address (pc_number[9:0]),
                .clock   (clk_max),
                .data    (data),
                .wren    (wren),
                .q       (inst_out_inst_Mem)
        );

        /**********Instruction Decoder********/
        wire [6:0] inDec_opcode;
        wire [4:0] inDec_rs1Add;
        wire [4:0] inDec_rs2Add;
        wire [ 4:0] inDec_rdAddr;
        wire [ 3:0] inDec_alu_func;
        wire [31:0]  inDec_imm;

    inDecode inDec(
      .inst(inst_out_inst_Mem),
      .opcode(inDec_opcode),
      .rsAddr( inDec_rs1Add),
      .rdAddr(inDec_rdAddr),
      .shamt(inDec_rs2Add),
      .alu_func(inDec_alu_func),
      .imm(inDec_imm)
    );

    /***************register bank**********************/
         wire        regB_write_en;
    wire [31:0] regB_write_data;
```

```
wire [31:0] regB_data1;
wire [31:0] regB_data2;

RegBank regBank1(
  .clk(clk),
  .r_addr1(inDec_rs1Add),
  .r_addr2(inDec_rs2Add),
  .w_addr(inDec_rdAddr),
  .write_en(regB_write_en),
            .reset(~reset),
  .write_data(regB_write_data),
  .data1(regB_data1),
  .data2(regB_data2)
);

    //select between register data or immediate data for alu
    wire         inp_ALUSrc_2;
    wire [31:0] inp_AluInput2;

    INPUT_SEL input_sel1(
            .ALUSrc1(inp_ALUSrc_2),
            .IMMval(inDec_imm),
            .reg_val(regB_data2),
            .AluInput2(inp_AluInput2)
     );

    wire         inp_ALUSrc1    ;  // ALUSrc is Control unit flag to select the Inp
    wire  [31:0] inp_1_reg_val;       // If 0, select register out put
    wire [31:0]  inp_AluInput1;

    // select between register and pc for alu
    INPUT_SEL_2 input_sel2(
            .ALUSrc2(inp_ALUSrc1),
            .Pc(pc_number),
            .reg_val(regB_data1),
            .AluInput1(inp_AluInput1)
    );

    /************ALU*************************/
    wire[31:0] alu_alu_out;
    wire alu_eq, alu_a_lt_b, alu_a_lt_ub;
    wire [3:0] alu_ctrl;

    alu Alu1(
            .func(alu_ctrl),
            .A(inp_AluInput1),
            .B(inp_AluInput2),
```

```
        .alu_out(alu_alu_out),
        .eq(alu_eq),
        .a_lt_b(alu_a_lt_b),
        .a_lt_ub(alu_a_lt_ub)
);

/****************** Main contoler *****************/
wire  mem_read, mem_write, branch;
wire [1:0] mem_to_reg, next_pc_sel;
wire [1:0] alu_op;

main_control mc_0 (
        .opcode                         (inDec_opcode),
        .alu_src_1              (inp_ALUSrc1),
        .alu_src_2              (inp_ALUSrc_2),
        .mem_to_reg              (mem_to_reg),
        .reg_write              (regB_write_en),
        .mem_read                (mem_read),
        .mem_write          (mem_write),
        .branch                         (branch),
        .alu_op                         (alu_op),
        .next_pc_sel        (next_pc_sel)
);

/******************ALU controler******************/
alu_control ac_0 (
                .alu_op                         (alu_op),
                .alu_funct               (inDec_alu_func),
                .alu_ctrl               (alu_ctrl)
);

/******************Data Memory*****************/
wire [31:0] dat_mem_output;

Data_memory dat_mem(
  .clk(clk),
  .Address(alu_alu_out),
  .Write_data(regB_data2),
  .MemRead(mem_read),
  .MemWrite(mem_write),
  .Read_data(dat_mem_output)
);

// Mux for register input selector after the data memmory
wire [31:0] incr_pc; // use this one in Pc parts

Multiplexer multi_reg(
```

```
            .Read_data(dat_mem_output),
            .Address(alu_alu_out),                    /*direct output from ALU*/
            .PC4(incr_pc),
            .MemtoReg(mem_to_reg),
            .mux_out(regB_write_data) //sending to register bank
        );

        /*******************Program Counter*****************/
        PC_full pc_full(
            .clk(clk),
            .next_pc_sel(next_pc_sel),
            .imme(inDec_imm),
            .jalr(alu_alu_out),
            .eq(alu_eq),
            .a_lt_b(alu_a_lt_b),
            .a_lt_ub(alu_a_lt_ub),
            .branch(branch),
            .pc_out(pc_number),
            .reset(~reset),
            .pcinc_to_m2(incr_pc)
        );

endmodule
```

Direct Mapped Cache

```
`timescale 1ns / 1ps
module cache();
parameter size = 8;         // cache size
parameter index_size = 3;    // index size
reg [31:0] cache [0:size - 1]; //registers for the data in cache
reg [11 - index_size:0] tag_array [0:size - 1]; // for all tags in cache
reg valid_array [0:size - 1]; //0 - there is no data 1 - there is data

initial
    begin: initialization
        integer i;
        for (i = 0; i < size; i++)
        begin
            valid_array[i] = 6'b000000;
            tag_array[i] = 6'b000000;
        end
    end
endmodule
```

Cashe and RAM

```
module cache_and_ram(
    input [31:0] address,
    input [31:0] data,
    input clk,
    input read_mode, //mode equal to 1 when we write and equal to 0 when we read
    input write_mode,
    output [31:0] out
);
//previous values
reg [31:0] prev_address, prev_data;
reg prev_mode;
reg [31:0] temp_out;
reg [cache.index_size - 1:0] index; // for keeping index of current address
reg [11 - cache.index_size:0] tag;  // for keeping tag of current address

ram ram();
cache cache();

wire mode;

assign mode = (write_mode) & (~read_mode);
initial
    begin
        index = 0;
        tag = 0;
        prev_address = 0;
        prev_data = 0;
        prev_mode = 0;
    end
always @(posedge clk)
begin

    //check if the new input is updated

    if (prev_address != address || prev_data != data || prev_mode != mode)
    begin
    prev_address = address % ram.size;
    prev_data = data;
    prev_mode = mode;
    tag = prev_address >> cache.index_size; // tag
    index = address % cache.size;        // index value
    if (mode == 1)
    begin
         // write address to data in the ram
        ram.ram[prev_address] = data;
        //write new data to the relevant cache block if there is such one
```

```
        if (cache.valid_array[index] == 1 && cache.tag_array[index] == tag)
            cache.cache[index] = data;
        end
        else
            begin
                //write new data to the relevant cache's block,
                // because the one we addressing to will be possibly addressed one mo
                if (cache.valid_array[index] != 1 || cache.tag_array[index] != tag)
                    begin
                    cache.valid_array[index] = 1;
                    cache.tag_array[index] = tag;
                    cache.cache[index] = ram.ram[prev_address];
                    end
                    temp_out = cache.cache[index];
            end
        end
end
assign out = temp_out;
endmodule
```

6.2 Instruction Set

Mnemonic	Instruction	Description
R type		
ADD rd, rs1, rs2	Addition	rd <= rs2 + rs1
SUB rd, rs1, rs2	Subtraction	rd <= rs2 - rs1
SLL rd, rs1, rs2	Shift Left Logic	rd <= rs1 <<< rs2 (only by lower 5 bits in rs2)
SLT rd, rs1, rs2	Set Less Than	if rs1 < rs2: rd <= 1, if rs1 >= rs2: rd <= 0
SLTU rd, rs1, rs2	Set Less Than Unsigned	if rs1 < unsigned(rs2): rd <= 1, if rs1 >= unsigned(rs2): rd <= 0
XOR rd, rs1, rs2	XOR Logic	rd <= rs1 ^ rs2
SRL rd, rs1, rs2	Shift Right Logic	rd <= rs1 >>> rs2 (only by lower 5 bits in rs2)
SRA rd, rs1, rs2	Shift Right Arithmetic	rd <= rs1 >> rs2 (only by lower 5 bits in rs2)
OR rd, rs1, rs2	OR Logic	rd <= rs1 | rs2
AND rd, rs1, rs2	AND Logic	rd <= rs1 & rs2
I type		
ADDI rd, rs1, imm	Add Immediate	rd <= rs1 + imm
SLTI rd, rs1, imm	Set Less Than Immediate	if rs1 < imm: rd <= 1, If rs1 >= imm: rd <= 0
SLTIU rd, rs1, imm	Set Less Than Immediate Unsigned	if rs1 < unsigned(imm): rd <= 1, If rs1 >= unsigned(imm): rd <= 0

XORI rd, rs1, imm	XOR Immediate	rd <= rs1 ^ imm
ORI rd, rs1, imm	OR Immediate	rd <= rs1 | imm
ANDI rd, rs1, imm	AND Immediate	rd <= rs1 & imm
SLLI rd, rs1, shamt	Shift Left Logic Immediate	rd <= rs1 <<< shamt
SRLI rd, rs1, shamt	Shift Right Logic Immediate	rd <= rs1 >>> shamt
SRAI rd, rs1, shamt	Shift Right Arithmetic Immediate	rd <= rs1 >> shamt
JALR rd, imm(rs1)	Jump And Link Register	Rd <= PC + 4 PC <= rs1 + imm
Load Instructions (I type)		
LW rd, imm(rs1)	Load word	rd <= mem[rs1 + imm]
LBU rd, imm(rs1)	Load Byte Unsigned	rd <= mem[rs1 + imm] The fetched value from memory is zero
LHU rd, imm(rs1)	Load Halfword Unsigned	rd <= mem[rs1 + imm] The fetched value from memory is zero extended to make it as 32 bit
S type		
SW rs2, imm(rs1)	Store Word	rs2[31:0] <= mem[rs1 + imm]
SB type		
BEQ rs1, rs2, imm	Branch Equal	if rs1 = rs2: PC <= PC + imm*4 else: PC <= PC + 4
BNE rs1, rs2, imm	Branch Not Equal	if rs1 != rs2: PC <= PC + imm*4 else: PC <= PC + 4

BLT rs1, rs2, imm	Branch Less Than	if rs1 < rs2: PC <= PC + imm*4 else: PC <= PC + 4
BGE rs1, rs2, imm	Branch Greater Than or Equal	if rs1 >= rs2: PC <= PC + imm*4 else: PC <= PC + 4
BLTU rs1, rs2, imm	Branch Less Than Unsigned	if unsigned(rs1) < unsigned(rs2): PC <= PC + imm*4 else: PC <= PC + 4
BGEU rs1, rs2, imm	Branch Greater Than or Equal Unsigned	if unsigned(rs1) >= unsigned(rs2): PC <= PC + imm*4 else: PC <= PC + 4
U type		
LUI rd, imm	Load Upper Immediate	rd[31:12] <= imm rd[11:0] <= 12'b0
AUIPC rd, imm	Add Upper Immediate to PC	rd <= PC + offset (offset[32] = imm[20] + zero[12])
UJ type		
JAL rd, imm	Jump And Link	rd <= PC + 4 PC = PC + imm