University of Moratuwa

Department of Electronic and Telecommunication Engineering

EN3023 - Electronic Design Realization

Project report- Imposters

Name	Index Number
C.N. Abeywansa	190005E
W.A.D.K. De Silva	190128H
M.S.K. Gunasekara	190202F
G. I. Hewasura	190234E

February 13, 2023

Contents

Instruction Set Architecture

The ISA is a RISCV architecture. 38 basic instructions are implemented including the U type and the J type. All instructions are 32 bit fixed length.

Instruction set

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

R – type

R-type instructions are for register-register operations. R-type instructions specify two source registers (rs1 and rs2) and a destination register (rd). The funct10 field is an additional opcode field.

I – type

I- type instructions are for immediate and load operations.I-type instructions specify one source register (rs1) and a destination register (rd). The second source operand is a sign-extended 12-bit immediate, encoded contiguously in bits 21–10. The funct3 field is a second opcode field.

S-type

S-type instructions are for store operations.

B-type

B-type instructions are for conditional branch operations. The 12-bit immediate is sign-extended, shifted left one bit, then added to the current pc to give the target address.

U-type

U-type instructions are for incrementing the PC register by an immediate value specified by the instruction. A 20'b0 is added concatenated to the end of the immediate and added to the current PC value.

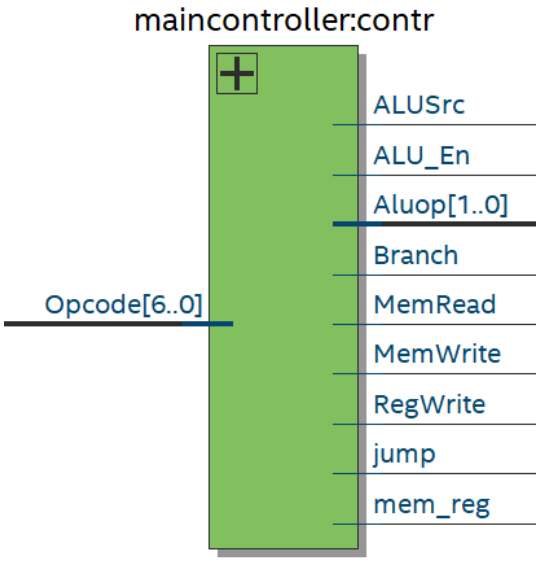
J-types.

Absolute jumps (J) and jump and link (JAL) instructions use the J-type format. The 25-bit jump target offset is sign-extended and shifted left one bit to form a byte offset, then added to the pc to form the jump target address. Jumps can therefore target a ±32 MB range. JAL stores the address of the instruction following the jump (pc+4) into register x1

Instruction	Description
ADD	rd = rs1 + rs2
SUB	rd = rs1 - rs2
XOR	rd = rs1 ^ rs2
OR	rd = rs1 | rs2
AND	rd = rs1 & rs2
SLL	rd = rs1 << rs2
SRL	rd = rs1 >> rs2
SRA	rd = rs1 >> rs2
SLT	rd = (rs1 < rs2)?1:0
SLTU	rd = (rs1 < rs2)?1:0
ADDI	rd = rs1 + imm
XORI	rd = rs1 ^ imm
ORI	rd = rs1 | imm
ANDI	rd = rs1 & imm
SLLI	rd = rs1 << imm[0:4]
SRLI	rd = rs1 >> imm[0:4]
SRAI	rd = rs1 >> imm[0:4]
STLI	rd = (rs1 < imm)?1:0
SLTIU	rd = (rs1 < imm)?1:0
LB	rd = M[rs1+imm][0:7]
LH	rd = M[rs1+imm][0:15]
LW	rd = M[rs1+imm][0:31]
LBU	rd = M[rs1+imm][0:7]
LHU	rd = M[rs1+imm][0:15]
SB	M[rs1+imm][0:7] = rs2[0:7]

SH	M[rs1+imm][0:15] = rs2[0:15]
SW	M[rs1+imm][0:31] = rs2[0:31]
BEQ	if(rs1 == rs2) PC += imm
BNE	if(rs1 != rs2) PC += imm
BLT	if(rs1 < rs2) PC += imm
BGE	if(rs1 >= rs2) PC += imm
BLTU	if(rs1 < rs2) PC += imm
BGEU	if(rs1 >= rs2) PC += imm
JAL	rd = PC+4; PC += imm
LUI	rd = imm << 12
AUIPC	rd = PC + (imm << 12)

Main controller



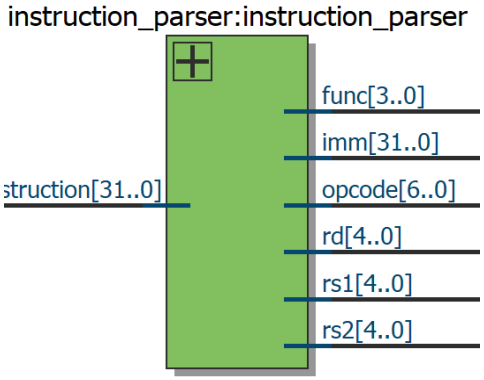
Main controller gets the opcode of the 32 bit instruction as the input from the Instruction passer and depending on the opcode the controller decides on 10 control signals and provides them to the specified modules through the data path. The 10 control signals are;

1. Register write
2. Mem- write

3. Mem- read
4. Branch
5. ALU Src
6. ALU op (2 bits)
7. J-type
8. ALU enable
9. Mem-to-Register

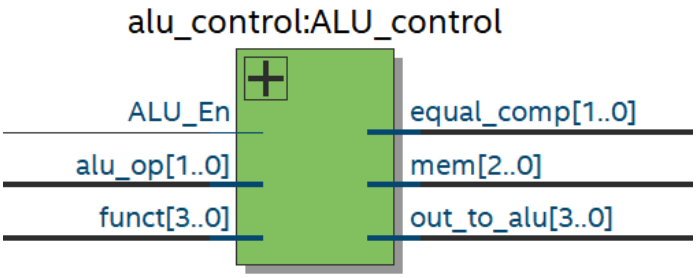
Instruction Type	Control signal
R- type	1000000000
I-type load	1010110000
I-type imm	1000101000
s-type	0100110001
sb-type	0001011000
jal-type	10001xx110
lui-type	10001xx010
auipc-type	00000xx010

Instruction passer



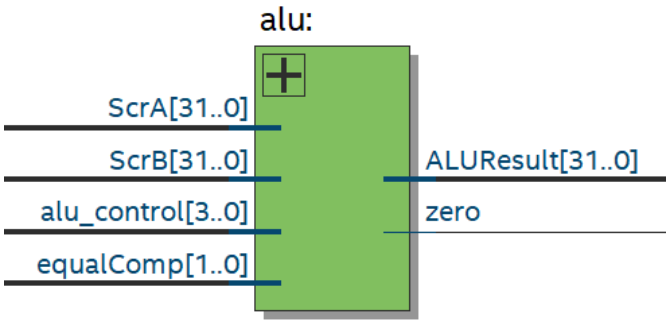
The input to the instruction passer is the 32 bit instruction straight from the instruction memory. Depending on the 7 bit Opcode of the instruction the instruction passer decides on the instruction type and outputs the values o the 5 bit read address 1, 5 bit read address 2, 5 bit write address to the register file and a 32 bit immediate value.

ALU controller



The ALU controller gets the 4 bit instruction[14:12,30] as an input from the instruction passer, and two control signals 1 bit ALUEn and 2 bit ALUop from the maincontroller. Depending on the input bits the controller determines which operation should be performed by the ALU and pass it over to the ALU as the 4 bit ALU control signal 2 bit equal_comp bit. Another three bit control signal is passed over to the data memory which will decide whether to signed or unsigned versions of read word, half word or byte.

Arithmetic and Logic Unit (ALU)



ALU is designed to perform operations between two 32 bit operands and produce 32 bit outputs. The control signals given as outputs from the ALU controller (4 bits) determines which operation will be performed on the operands. The 2 bit equal_comp input signal from the ALU controller determines which operation should be performed by the comperator part of the ALU. The table below contains the list of operations.

Operation	Control signal	Description
ADD	0010	Add the two operands
SUB	0110	Subtract the two operands

OR	0011	Perform OR operation between the two operands
AND	0000	Perform AND operation between the two operands
SLL	0100	Perform logical left shift on the operand (op1<<op2)
SRL	1000	Perform logical right shift on the operand (op1>>op2)
SLT	0101	Compare the two operands and write 1 to destination register if op1<op2
SLTU	0111	Compare the unsigned versions of the two operands and write 1 to destination register if op1<op2

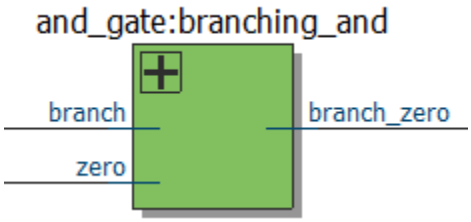
Operation	**Control signal (equal_comp)**	**Description**
BEQ	11	Compares the two operands (perform XOR operation) and raise zero flag if the ALU result is 32’b0
BNE	01	Compares the two operands (perform XOR operation) and raise zero flag if the ALU result is not 32’b0
BLT	00	Perform subtraction and raise zero flag if the sign of ALU result is 1

BGE	10	Perform subtraction and raise zero flag if the sign of ALU result is

Aside from the ALU result the ALU maintains a zero flag which will be used for branching instructions. The zero flag is set to one when the resultant from the ALU is 32'b0.

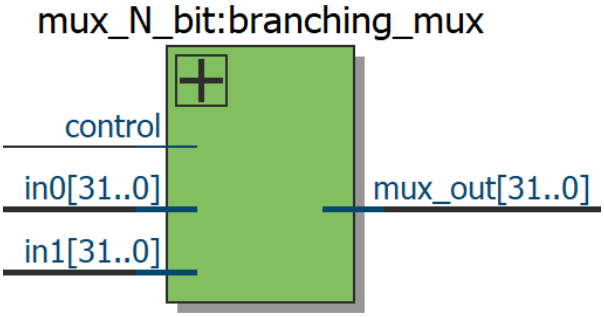
The ALU has two inputs to receive the two operands. One input is directly from the register file and the other input is connected to a MUX that allows to give the immediate from the instruction or the read value from the register depending on the control signal (ALUSrc) given to the MUX as the selection input.

AND Gate



The two inputs to the AND gate are zero flag from the ALU and the branching control signal from the main controller. Depending on the inputs the AND gate decides on whether to branch or not and pass a one bit input to the branching mux.

MUXs



J-Type selection MUX

The MUX gets two 32 bit inputs and a one bit control signal from the main controller as the control signal and outputs a 32 bit output to the write data input to the register file. The control signal choose either the PC-value+4 or the data memory output depending on the instruction type.

Control signal	MUX out
1	PC_value+4
0	Data memory output

ALU input selection MUX

The MUX gets two inputs, one from the Instruction passer (32 bit immediate value) and another 32 bit input from the register file. Depending on the control signal ALUSrc provided from the maincontroller the MUX chooses either of the inputs and pass it on to the ALU as the second operand.

Control signal	MUX out
1	Immediate value
0	Register file data

Branching MUX

The branching MUX gets the 32 bit current PC + 32 bit immediate value passed from the add only ALU and PC+4 value as the two inputs and choose either of them depending on the branching control signal provided as the output from the AND gate.

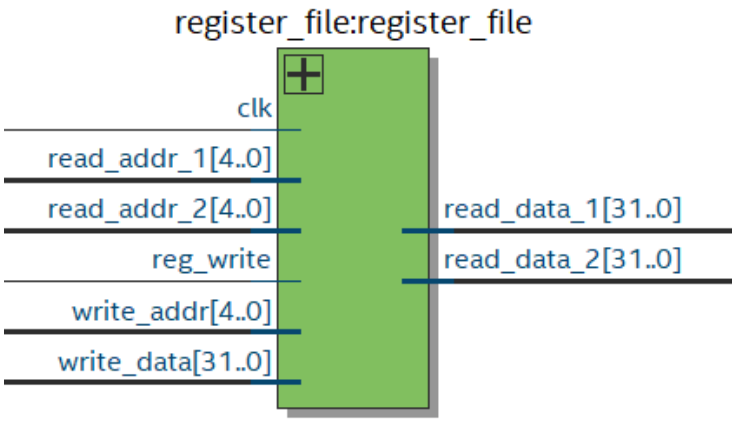
Control signal	MUX out
1	Immediate value + Current PC value
0	PC value + 4

Data memory MUX

The control signal memory to register given from the controller chooses the ALU result or the data read from the data memory and pass it on to the J-Type selection MUX.

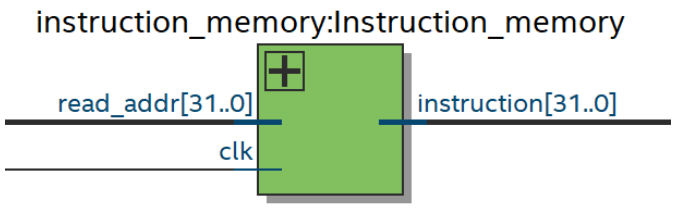
Control signal	MUX out
1	Data memory output
0	ALU result

Register File



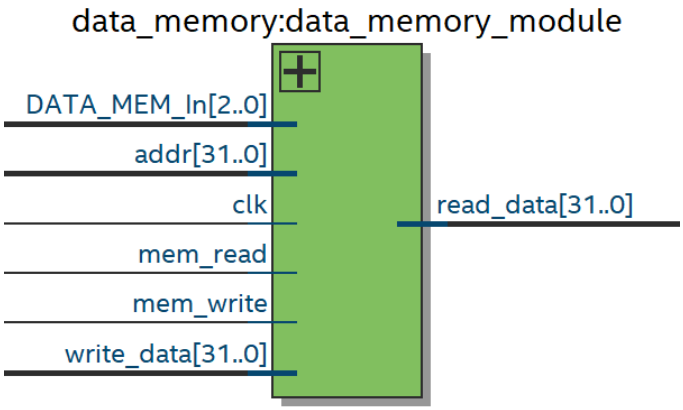
The register file was created with the ability to read two registers and write to one register in one clock cycle. Each register has a width of 32 bits. The register file contains 32 registers (x0 to x31), where x0 is a special read-only register called the Zero register, which always holds the value 0x00000000. Other 31 registers can be written and read as normal. The reg_write is used as a write enable switch to write the data in the write_data bus to the register specified by the address in the write_addr bus. read_data_1 and read_data_2 always output the data in the registers specified by read_addr_1 and read_addr_2, respectively. The reset pin sets all the registers to 0x00000000.

Instruction Memory



All the prefetched instructions are stored inside the instruction memory and the relevant address to be fetched will be provided by the program counter output. When the output of the program counter (pc_out) reaches the instruction memory and gets triggered by the positive edge of the clock the instruction in the relevant address will be sent out from the instruction memory to the instruction passer where all the decoding happens. In the instruction memory we designed we can hold a set of 32 instructions at a time. Since our instruction memory was word addressable, we had to shift the address always by two bits to cancel out the effect of incrementing the pc count by 4.

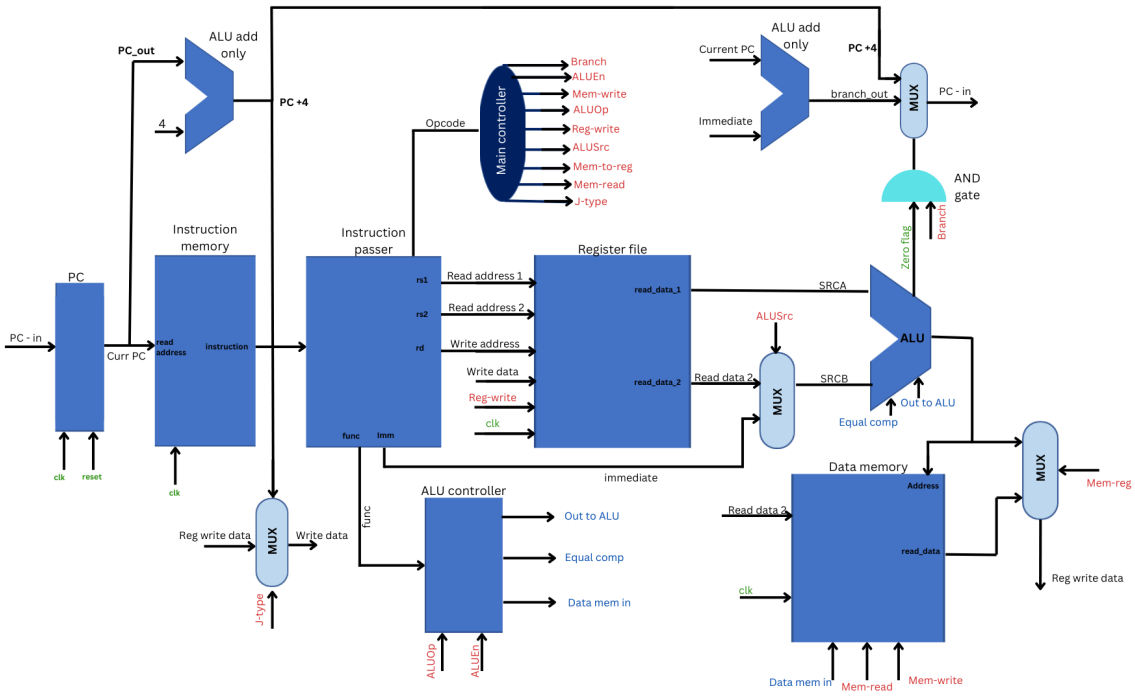
Data Memory



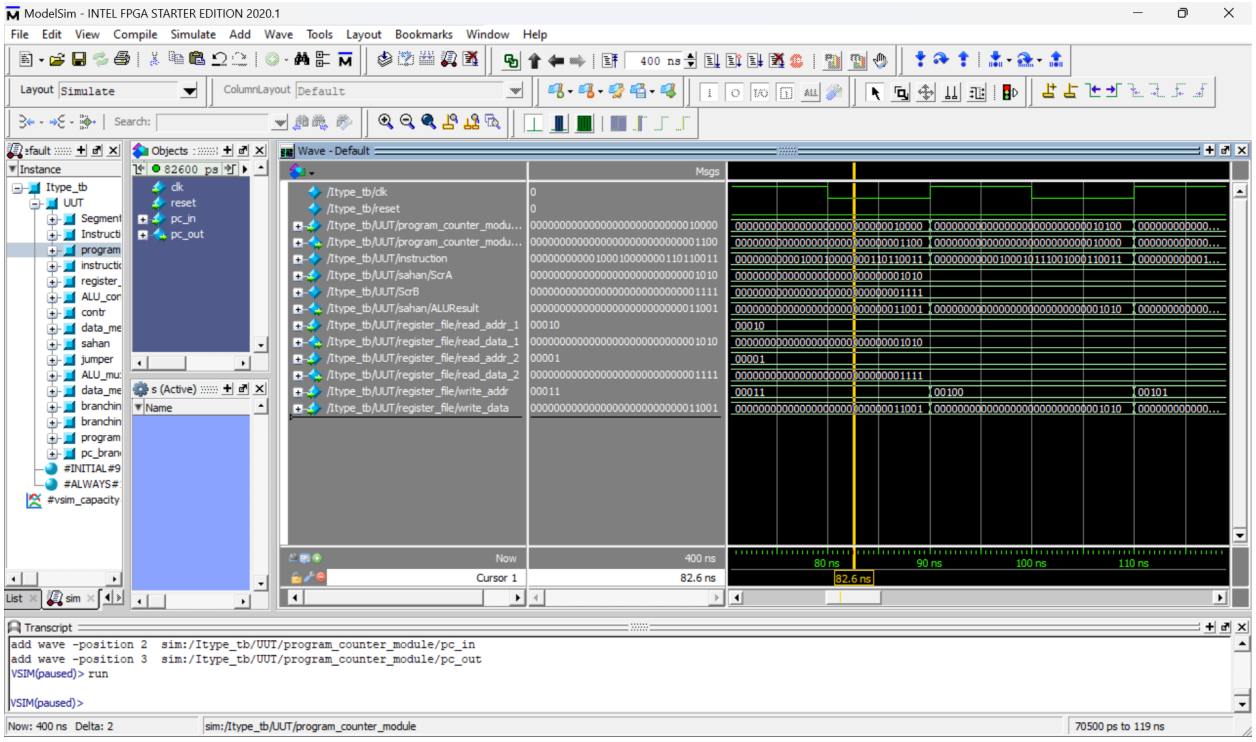
Data memory is the place where all the program executable operands are stored. In the data memory we designed we have 64 locations to store the data that comes to the data memory. There are 2 control signals which come to the data memory as mem_write and mem_read. And a three-bit signal is coming from the ALU (Arithmetic Logic Unit) control which is named as DATA_MEM_IN. The three bits in DATA_MEM_IN contains the signed unsigned identifying bit and two bits to distinguish whether the load or store data to be a byte, half word, or a word. If the mem_read signal is enabled the data will be sent from the data memory as preferred by the DATA_MEM_IN signal. If mem_write is enabled the data comes to the data memory from the register files will get stored in the relevant address (which is calculated by the ALU) by checking whether a byte, half word or a word needs to be stored.

Operation	Control Signal (DATA_MEM_IN)
Load Byte	001
Load Halfword	010
Load word	011
Load Byte Unsigned	101
Load Halfword Unsigned	110
Store Byte	001
Store Halfword	010
Store Word	011

Data path


PC_out
ALU add only
4
PC +4
Opcode
Main controller
Branch
ALUEn
Mem-write
ALUOp
Reg-write
ALUSrc
Mem-to-reg
Mem-read
J-type
Current PC
ALU add only
Immediate
PC +4
branch_out
MUX
PC - in
AND gate
Zero flag
Branch
Instruction memory
Instruction passer
Register file
PC
PC - in
Curr PC
read address
instruction
clk
reset
rs1
rs2
rd
func
Imm
Read address 1
Read address 2
Write address
Write data
Reg-write
clk
read_data_1
read_data_2
SRCA
ALUSrc
Read data 2
SRCB
ALU
Out to ALU
Equal comp
immediate
Reg write data
MUX
Write data
J-type
func
ALU controller
Out to ALU
Equal comp
Data mem in
ALUOp
ALUEn
Data memory
Address
read_data
Read data 2
clk
Data mem in
Mem-read
Mem-write
MUX
Mem-reg
Reg write data

Test bench



The test bench was coded with a time scale of 1 ns/ 100 ps. Initially, it resets the clock, and set the reset wire high for 15 ns. This allows to initialize the Program counter to run the first instruction. Then, it uses a timed loop to run the clock at 50 MHz. The following is the code used for the test bench.

```
`timescale 1ns/100ps

module main_tb;

reg clk,reset;

main UUT (clk, reset, ALU_result);

initial
begin
    clk = 0;
    reset = 1;
    #15;reset = 0;
    #400
    $stop;
    //$finish;
end
always begin #10 clk=~clk; end
endmodule
```

Appendix

```
// -------------------------MAIN_CODE-----------------------------------

`timescale 1ns/100ps

module Itype(input clk, input reset, output [6:0]HEX0,output [6:0]HEX1,output [6:0]HEX2,output
[6:0]HEX3,output [6:0]HEX4,output [6:0]HEX5,output [6:0]HEX6,output [6:0]HEX7);

        //PC pc (.clk(clk), .reset(rst))

  wire [31:0] instruction,ScrB,register_write_data;
        wire [1:0] Alu_op;
        wire [3:0] out_to_alu;
   wire [1:0] equal_comp;
        wire [2:0] mem;
        wire RegWrite, MemWrite, MemRead, branch, ALUSrc,jump, ALU_En,
mem_reg,zero,branch_zero;

        wire [4:0] read_addr_1;
        wire [4:0] read_addr_2;
        wire [4:0] write_addr;
        wire [31:0] write_data;
        wire [31:0] read_data_1;
        wire [31:0] read_data_2;
        wire [6:0] Opcode;
        wire [31:0] immidiate;
        wire [3:0] funct;
        wire [31:0] read_data;
        wire [31:0] branch_out;
        wire [31:0] ALU_result;

        // pc
                wire [31:0] pc_in;
                wire [31:0] pc_out;
                wire [31:0] pc_plus4;
```

```
segment Segment7(
.clk(clk),
.ALU_result(ALU_result),
.HEX0(HEX0),
.HEX1(HEX1),
.HEX2(HEX2),
.HEX3(HEX3),
.HEX4(HEX4),
.HEX5(HEX5),
.HEX6(HEX6),
.HEX7(HEX7)
);
        //wire reg_w;
instruction_memory Instruction_memory (
        .read_addr(pc_out),
        .clk(clk),
        .instruction(instruction)
        );
program_counter program_counter_module (
        .clk(clk),
        .reset(reset),
        .pc_in(pc_in),
        .pc_out(pc_out)
        );


instruction_parser instruction_parser(
                .instruction(instruction),
                .rs1(read_addr_1),
                .rs2(read_addr_2),
                .rd(write_addr),
                .opcode(Opcode),
                .imm(immidiate),
                .func(funct)
                );

        // for R type instruction
```

```
register_file register_file(
    .clk(clk),
    .read_addr_1(read_addr_1),
    .read_addr_2(read_addr_2),
    .write_addr(write_addr),
    .write_data(write_data),//.....
    .reg_write(RegW),
            .read_data_1(read_data_1),
    .read_data_2(read_data_2)
             );

alu_control ALU_control (
            .funct(funct),
            .alu_op(Alu_op),
            .out_to_alu(out_to_alu),
            .ALU_En(ALU_En),
            .equal_comp(equal_comp),
            .mem(mem)

            );
maincontroller contr(
            .Opcode(Opcode),
            .RegWrite(RegW),
            .MemWrite(MemW),
            .MemRead(MemR),
            .Branch(branch),
            .ALUSrc( ALUSrc),
            .Aluop(Alu_op),
            .jump(jump),
            .ALU_En(ALU_En),
            .mem_reg(mem_reg)
            );

data_memory data_memory_module(
    .clk(clk),
    .mem_read(MemR),
    .mem_write(MemW),
    .DATA_MEM_In(mem),
    .read_data(read_data),
    .write_data(read_data_2),
    .addr(ALU_result)
    // Add other inputs and outputs here
  );
```

```
alu ALU(
        .ScrA(read_data_1),
        .ScrB(ScrB),
        .alu_control(out_to_alu),
        .ALUResult(ALU_result),
        .zero(zero),
        .equalComp(equal_comp)
        );

mux_N_bit #(32) jumper (
                .in0(register_write_data),
                .in1(pc_plus4),
                .mux_out(write_data),
                .control( jump)
                );
mux_N_bit #(32) ALU_mux (
                .in0(read_data_2),
                .in1(immidiate),
                .mux_out(ScrB),
                .control( ALUSrc)
                );
mux_N_bit #(32) data_memory_mux (
                .in0(ALU_result),
                .in1(read_data),
                .mux_out(register_write_data),
                .control(mem_reg)
                );
mux_N_bit #(32) branching_mux (
                .in0(pc_plus4),
                .in1(branch_out),
                .mux_out(pc_in),
                .control(branch_zero)
                );
and_gate branching_and(
                .branch(branch),
                .zero(zero),
                .branch_zero(branch_zero)
                );
alu_add_only program_counter_4 (
                .in_a(pc_out),
                .in_b($signed(32'b0100)),
                //.in_b(32'h0004),
                .add_out(pc_plus4)
                ); // pc + 4
```

```
alu_add_only pc_branching (
                    .in_a(pc_out),
                    .in_b(immidiate),
                    .add_out(branch_out)
                    );
endmodule

// -------------------------CONTROLLER-----------------------------------

module maincontroller( control_signal, Opcode, Aluop, RegWrite, MemWrite, MemRead, Branch,
ALUSrc, Alu0, Alu1,jump, ALU_En, mem_reg);
        input [6:0] Opcode;
        output [1:0] Aluop;//ALUSrc,MemtoReg,RegWrite, MemRead, MemWrite, Branch,jump,
        output reg [9:0] control_signal;
        output wire RegWrite, MemWrite, MemRead, Branch, ALUSrc, Alu0, Alu1,jump, ALU_En,
mem_reg;
        assign {RegWrite, MemWrite, MemRead, Branch, ALUSrc, Alu0, Alu1,jump, ALU_En,
mem_reg} = control_signal;
        assign Aluop={Alu0, Alu1};
        always @(*)
        begin
                case(Opcode)
                        7'b0110011 : control_signal <= 10'b1000000000; // R-type
                        7'b0000011 : control_signal <= 10'b1010110000; // I-type load
                        7'b0010011 : control_signal <= 10'b1000101000; // I-type imm
                        7'b0100011 : control_signal <= 10'b0100110001; // s-type
                        7'b1100011 : control_signal <= 10'b0001011000; // sb-type
                        7'b1101111 : control_signal <= 10'b10001xx110; // jal-type
                        7'b0110111 : control_signal <= 10'b10001xx010; // lui-type
                        7'b0010111 : control_signal <= 10'b00000xx010; // auipc-type
                        default : control_signal    <= 10'b0000000010;
                endcase


        end

endmodule
```

```
// -------------------------INSTRUCTION_PARSER-----------------------------------

module instruction_parser(

  input [31:0] instruction,
  output reg [4:0] rs1,
  output reg [4:0] rs2,
  output reg [4:0] rd,
  output reg [6:0] opcode,
  output reg [31:0] imm,
  output reg [3:0] func

);

  always @(*)
  begin
    opcode = instruction[6:0];// extract opcode

          case(opcode)
                        7'b0110011 : // R-type
                                begin
                                                rd = instruction[11:7];
                                                rs1 = instruction[19:15];
                                                rs2 = instruction[24:20];
                                                imm = 32'hx;
                                                func = {instruction[14:12],instruction[30]};
                                end

                        7'b0000011 :  // I-type load
                                begin
                                                rd = instruction[11:7];
                                                rs1 = instruction[19:15];
                                                rs2 = instruction[24:20];
              imm = $signed(instruction[31:20]);
                                                func = {instruction[14:12],instruction[30]};
                                end

                        7'b0010011 :  // I-type imm
                                begin
                                                rd = instruction[11:7];
                                                rs1 = instruction[19:15];
                                                rs2 = 5'hx;
                                                imm = $signed(instruction[31:20]);
                                                func = {instruction[14:12],instruction[30]};
```

```
                end


            7'b0100011 :  // s-type
                begin
                                rd = instruction[11:7];
                                rs1 = instruction[19:15];
                                rs2 = instruction[24:20];
        imm = $signed({instruction[31:25],instruction[11:7]});
                                func = {instruction[14:12],instruction[30]};

                end


            7'b1100011 :  // sb-type
                begin
                                rd = 5'hx;
                                rs1 = instruction[19:15];
                                rs2 = instruction[24:20];
        imm = $signed({instruction[31],instruction[7],instruction[30:25],instruction[11:8],1'b0});
                                func = {instruction[14:12],instruction[30]};

                end


            7'b1101111 :  // jal-type
                begin
                                rd = instruction[11:7];
                                rs1 = 5'hx;
                                rs2 = 5'hx;
                                imm =
$signed({instruction[20],instruction[10:1],instruction[11],instruction[19:12]});
                                func = 4'hx;
                end


            7'b0110111 :  // lui-type
                begin
                                rd = instruction[11:7];
                                rs1 = 5'hx;
                                rs2 = 5'hx;
                                imm = $signed(instruction[31:12]);
                                func = 4'hx;
```

```
                        end


                7'b0010111 :  // auipc-type
                        begin
                                rd = instruction[11:7];
                                rs1 = 5'hx;
                                rs2 = 5'hx;
                                imm = $signed(instruction[31:12]);
                                func = 4'hx;
                        end


            endcase

  end
endmodule

// --------------------------ALU-------------------------------------

`timescale 1ns/100ps

module alu (ScrA,ScrB,alu_control,ALUResult,zero,equalComp);
   input  [31:0] ScrA,ScrB;
   input  [3:0] alu_control;
   output reg [31:0] ALUResult;
         output  reg zero;
   input [1:0]equalComp;

   wire equal_inequal;
   wire Comparatorenable;

   assign {equal_inequal,Comparatorenable} = equalComp;


always @ (*)
begin

   ALUResult='d0;
   zero='b0;

   case(alu_control)
   4'b0000: begin//AND
```

```
                zero=0;
   ALUResult = ScrA & ScrB;
end
4'b0001: begin//OR
                zero=0;
   ALUResult = ScrA | ScrB;
end
4'b0010: begin//ADD
                zero=0;
   ALUResult = ScrA + ScrB;
end
4'b0011: begin//XOR
   ALUResult = ScrA ^ ScrB;
                zero=0;
   if (Comparatorenable=='b1) begin
      if (equal_inequal=='b1) begin
         zero=(ALUResult==32'b0);
      end
                                else begin
         zero=(ALUResult!=32'b0);
      end
   end
end
4'b0100:begin//SLL
   ALUResult = ScrA << ScrB;
end
4'b0101:begin//SLT set less than
   ALUResult = ($signed(ScrA) < $signed(ScrB));// less than Correction
   if (Comparatorenable=='b1) begin
      if (equal_inequal=='b1) begin
         zero=(ALUResult!=32'b0);
      end else begin
         zero=(ALUResult==32'b0);
      end
   end
end
4'b0110://SUB
   begin
   ALUResult = $signed (ScrA)-$signed(ScrB);
   zero = ($signed(ALUResult)==$signed(1'd0));
   end
4'b0111://SLTU
   begin
      ALUResult=(ScrA<ScrB);
```

```
        if (Comparatorenable=='b1) begin
        if (equal_inequal=='b1) begin
           zero=(ALUResult!=32'b0);
        end else begin
           zero=(ALUResult==32'b0);
        end
      end
      end
   4'b1000:begin//SRL
      ALUResult = ScrA >> ScrB;
   end
   4'b1001:begin//SRLA
      ALUResult = $signed(ScrA) >>> ScrB;
   end


   default:begin
      ALUResult= 'b0;
   end
   endcase


end

endmodule

// --------------------------ALU_CONTROLLER-------------------------------------

module alu_control (alu_op, out_to_alu, funct, equal_comp, mem, ALU_En);
          input [1:0] alu_op;
          input ALU_En;
          input [3:0] funct;
          output [3:0] out_to_alu;
          output reg [1:0] equal_comp;
          output reg [2:0] mem;

          reg [3:0] out_to_alu;

          assign funct3 = {funct[3:1]};

          always @ (*)
          begin
                    if (ALU_En == 'b0)begin
                    equal_comp= 2'b00;
```

```
mem= 3'b000;
case (alu_op)
2'b00: //Rtype
        begin
                case(funct)
                4'b0000:
                        begin
                                out_to_alu <= 4'b0010;
                        end
                4'b0001:
                        begin
                                out_to_alu <= 4'b0110;
                        end
                4'b1000:
                        begin
                                out_to_alu <= 4'b0011;
                        end
                4'b1100:
                        begin
                                out_to_alu <= 4'b0001;
                        end
                4'b1110:
                        begin
                                out_to_alu <= 4'b0000;
                        end
                4'b0010:
                        begin
                                out_to_alu <= 4'b0100;
                        end
                4'b1010:
                        begin
                                out_to_alu <= 4'b1000;
                        end
                4'b1011:
                        begin
                                out_to_alu <= 4'b1001;
                        end
                4'b0100:
                        begin
                                out_to_alu <= 4'b0101;
                        end
                4'b0110:
                        begin
                                out_to_alu <= 4'b0111;
```

```
                end

                endcase
        end
2'b01:
        begin
        case(funct3)
        3'b000:
                begin
                        out_to_alu <= 4'b0010;
                end
        3'b100:
                begin
                        out_to_alu <= 4'b0011;
                end
        3'b110:
                begin
                        out_to_alu <= 4'b0001;
                end
        3'b111:
                begin
                        out_to_alu <= 4'b0000;
                end
        3'b001:
                begin
                        out_to_alu = 4'b0100;
                end
        3'b101:
        begin
                if (funct[0]==0)
                        out_to_alu <= 4'b1000;
                else
                        out_to_alu <= 4'b1001;
        end

        endcase

        end
2'b10:
        begin
                case(funct3)
                3'b000:
                begin
                        out_to_alu <= 4'b0010;
```

```
                mem<= 3'b001;
            end
            3'b001:
            begin
                out_to_alu <= 4'b0010;
                mem<= 3'b010;
            end
            3'b010:
            begin
                out_to_alu <= 4'b0010;
                mem<= 3'b011;
            end
            3'b100:
            begin
                out_to_alu <= 4'b0010;
                mem<= 3'b101;
            end
            3'b101:
            begin
                out_to_alu <= 4'b0010;
                mem<= 3'b011;
            end

            endcase
        end
2'b11:
        begin
            case(funct3)
            3'b000:
            begin
                out_to_alu <= 4'b0011;
                equal_comp <= 2'b11;
            end
            3'b001:
            begin
                out_to_alu <= 4'b0011;
                equal_comp <= 2'b10;
            end
            3'b100:
            begin
                out_to_alu <= 4'b0101;
                equal_comp <= 2'b11;
            end
            3'b101:
```

```
                                        begin
                                                out_to_alu <= 4'b0101;
                                                equal_comp<= 2'b10;
                                        end
                                        3'b110:
                                        begin
                                                out_to_alu <= 4'b0111;
                                                equal_comp <= 2'b11;
                                        end
                                        3'b111:
                                        begin
                                                out_to_alu <= 4'b0111;
                                                equal_comp <= 2'b10;
                                        end

                                        endcase
                                end

        endcase
        end
        end
endmodule

// --------------------------ALU_ADD_ONLY-------------------------------------

module alu_add_only (in_a, in_b, add_out);
        input [31:0] in_a, in_b;
        output [31:0] add_out;
        assign add_out=in_a+in_b;
endmodule

// --------------------------AND_GATE-----------------------------------

module and_gate (input branch, zero, output branch_zero);
   assign branch_zero = branch & zero;
endmodule

// --------------------------INSTRUCTION_MEMORY-------------------------------------

module data_memory (addr, write_data, read_data, clk, mem_read, mem_write, DATA_MEM_In);

  input [31:0] addr;
  input [31:0] write_data;
  input clk, mem_read, mem_write;
```

```
 input [2:0] DATA_MEM_In;
 output reg [31:0] read_data;

reg [31:0] dmemory [63:0];
 reg [31:0] write_data_in;

wire [5:0] shifted_addr;
 wire [2:0] DATA_MEM_In;
 wire [31:0] DATA;

assign shifted_addr = addr[5:0];

 //assign DATA = dmemory[addr];

 always @(posedge clk)
 begin
    if (mem_read == 1'b1)

        begin

           case(DATA_MEM_In)
           3'b001: begin
             read_data = $signed(dmemory[shifted_addr][7:0]);

               end

           3'b010:begin
             read_data = $signed(dmemory[shifted_addr][15:0]);
               end

           3'b011: read_data = dmemory[shifted_addr];
             3'b101: read_data = $unsigned(dmemory[shifted_addr][7:0]);
             3'b110: read_data= $unsigned(dmemory[shifted_addr][15:0]);
           endcase

        end

    else if (mem_write == 1'b1)

        begin

           case(DATA_MEM_In)
           3'b001: begin
             write_data_in = $signed(write_data[7:0]);
```

```
            dmemory[shifted_addr] = write_data_in;
          end
          3'b010:begin
            write_data_in = $signed(write_data[15:0]);
            dmemory[shifted_addr] = write_data_in;
          end

          3'b011:begin
            dmemory[shifted_addr] = write_data;
          end
          endcase
        end

        end

endmodule

// --------------------------INSTRUCTION_MEMORY-------------------------------------

module instruction_memory (read_addr, instruction, clk);
        input clk;
        input [31:0] read_addr;
        output [31:0] instruction;
        reg [31:0] Imemory [63:0];
        integer k;



        wire [7:0] shifted_read_addr;
  assign shifted_read_addr = read_addr[7:0] >>> 2;
        assign instruction = Imemory[shifted_read_addr];

        always @(posedge clk)
        begin

                for (k=16; k<32; k=k+1) begin// here Ou changes k=0 to k=16
                        Imemory[k] = 32'b0;
                end
Imemory[0] = 32'b00000000111100000000000010010011;
Imemory[1] = 32'b00000000111100000000000010010011; //addi x1, x0, 15 - I
Imemory[2] = 32'b00000000101000000000000100010011; //addi x2, x0, 10 - I
Imemory[3] = 32'b00000000000010001000000110110011; //add x3, x2, x1 - R
Imemory[4] = 32'b00000000000010001011100100110011; //and x4, x2, x1 - R
Imemory[5] = 32'b00000000000010001011000101010110011; //or x5, x2, x1 - R
```

```
Imemory[6] = 32'b00000000010000011010000000100011; //sw x4, 0(x3)
Imemory[7] = 32'b00000000000000000000000000000000; //nop
Imemory[8] = 32'b00000000001100000010001000000011; //load x0, x4, 3
Imemory[9] = 32'b00000000000000000000000000000000; //nop

Imemory[10] = 32'b00000000101000000000001100010011; //addi x6, x0, 10 - I
Imemory[11] = 32'b00000000000100011000010001100011; //bne x6, x2, 8
Imemory[12] = 32'b00000000111100000000000010010011; //addi x1, x0, 15 <Skipped
Imemory[13] = 32'b00000000000000000000000000000000; //nop


        end

endmodule

// -------------------------N_BIT_MUX-----------------------------------

module mux_N_bit (in0, in1, mux_out, control);
        parameter N = 32;
        input [N-1:0] in0, in1;
        output [N-1:0] mux_out;
        input control;
        assign mux_out=control?in1:in0;
endmodule

// -------------------------PROGRAM_COUNTER-----------------------------------

module program_counter (clk, reset, pc_in, pc_out);
        input clk, reset;
        input [31:0] pc_in;
        output [31:0] pc_out;
        reg [31:0] pc_out;
        always @ (posedge clk or posedge reset)
        begin
                if(reset==1'b1)
                        pc_out<=0;
                else
                        pc_out<=pc_in;
        end
endmodule

// -------------------------TEST_BENCH-----------------------------------

`timescale 1ns/100ps
```

```
module Itype_tb;

reg clk,reset;

Itype UUT (clk, reset, ALU_result);

initial
begin
        clk = 0;
        reset = 1;
        #15;reset = 0;
        #400
        $stop;
        //$finish;
end
always begin #10 clk=~clk; end
endmodule
```