Project: Single Cycle RISC-V

1. Introduction

in the area of Computer Architecture, it was originally designed to support research and education, for academic and industrial applications RISC-V instruction set architecture (ISA) is now set to become a standard free and open architecture. For the success and adoption of RISC-V, 32-bit, 64-bit and 128-bit address spaces support by RISC-V.

A minimal set of instructions adequate to provide a reasonable target for assemblers, linkers, compilers and operating systems, the ISA is separated into a small base integer ISA. The set of compatible tool chains which includes the above Suits, provided by RISC-V foundation.

In this architecture it provides the following set of **RV321 instructions:**

R-Type: add, sub, and, or

I-Type: addi, andi, ori, lw, jalr

B-Type: beq, bne

J-Type: jal

S-Type: sw

You can add more instructions by modifying the architecture in terms of muxes and the width of control lines.

To calculate the excuation time:

$$T_{exec} = \quad instructions * CPI\left(\frac{cycle}{instruction}\right) * T_c\left(\frac{seconds}{cycle}\right)$$

In our example we testcase we have #instructions = 16 instructions , CPI = 1 (single cycle), T_c is calculated by evaluate the critical path for the longest instruction which is lw instruction.

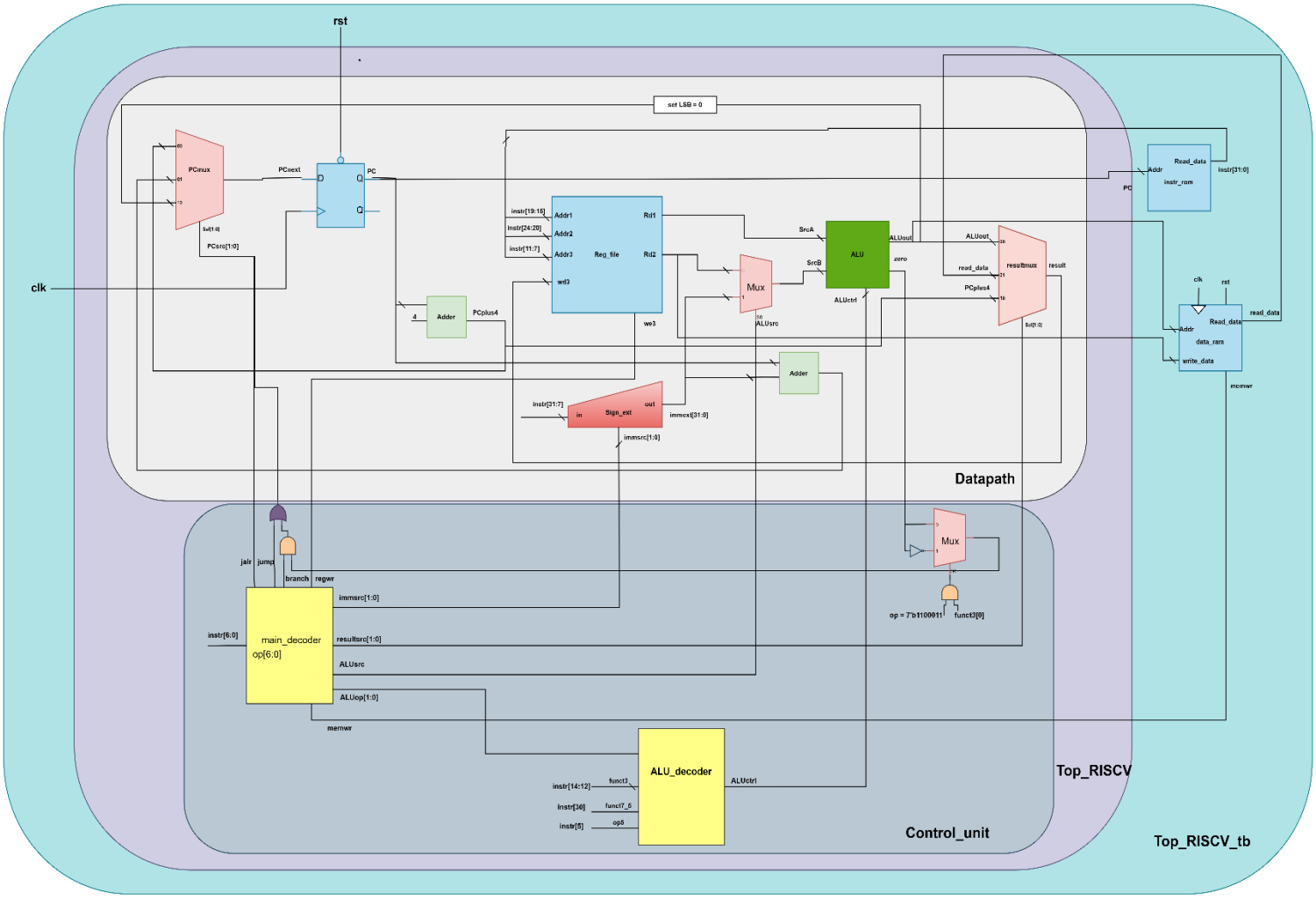
$$T_c = t_{pcq_{pc}} + 2t_{mem} + t_{RFread} + t_{alu} + t_{mux} + t_{RFsetup}$$

$$T_c = 10 + 2 * 100 + 50 + 50 + 10 + 10 = 330\, ps$$

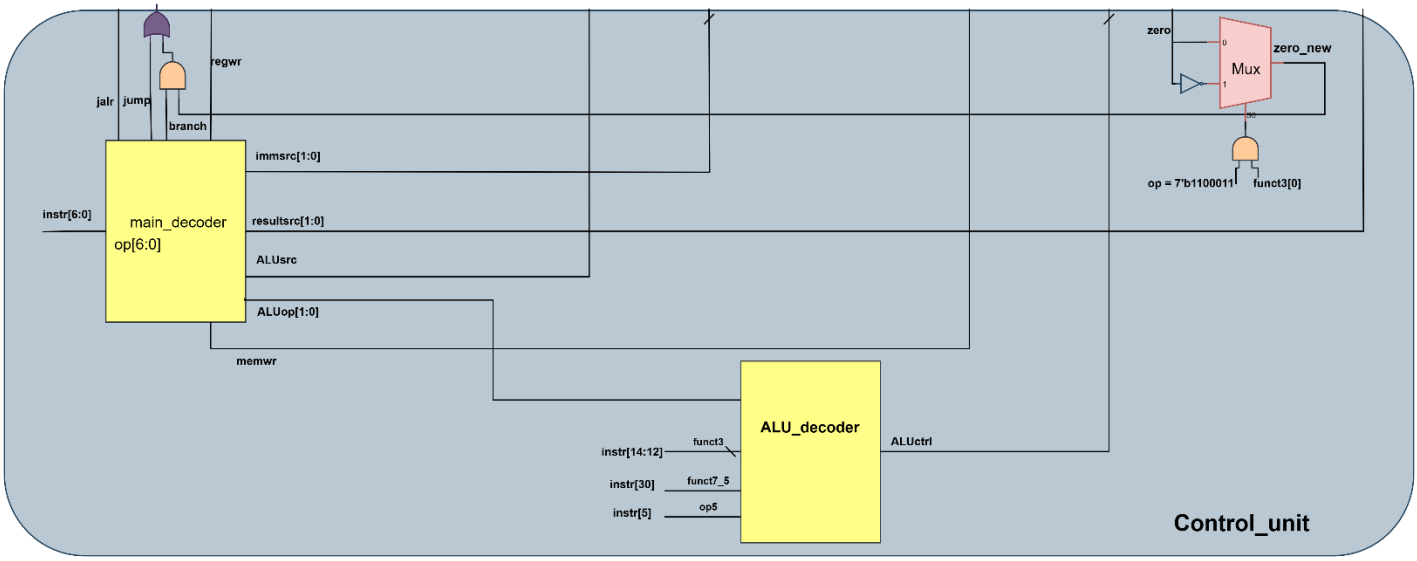
We can now evaluate the minimum execution time for our single-cycle RISC-V:

$$T_{exec} = 16 * 1 * 330 = \quad 5280\, ps = 5.28\, ns$$

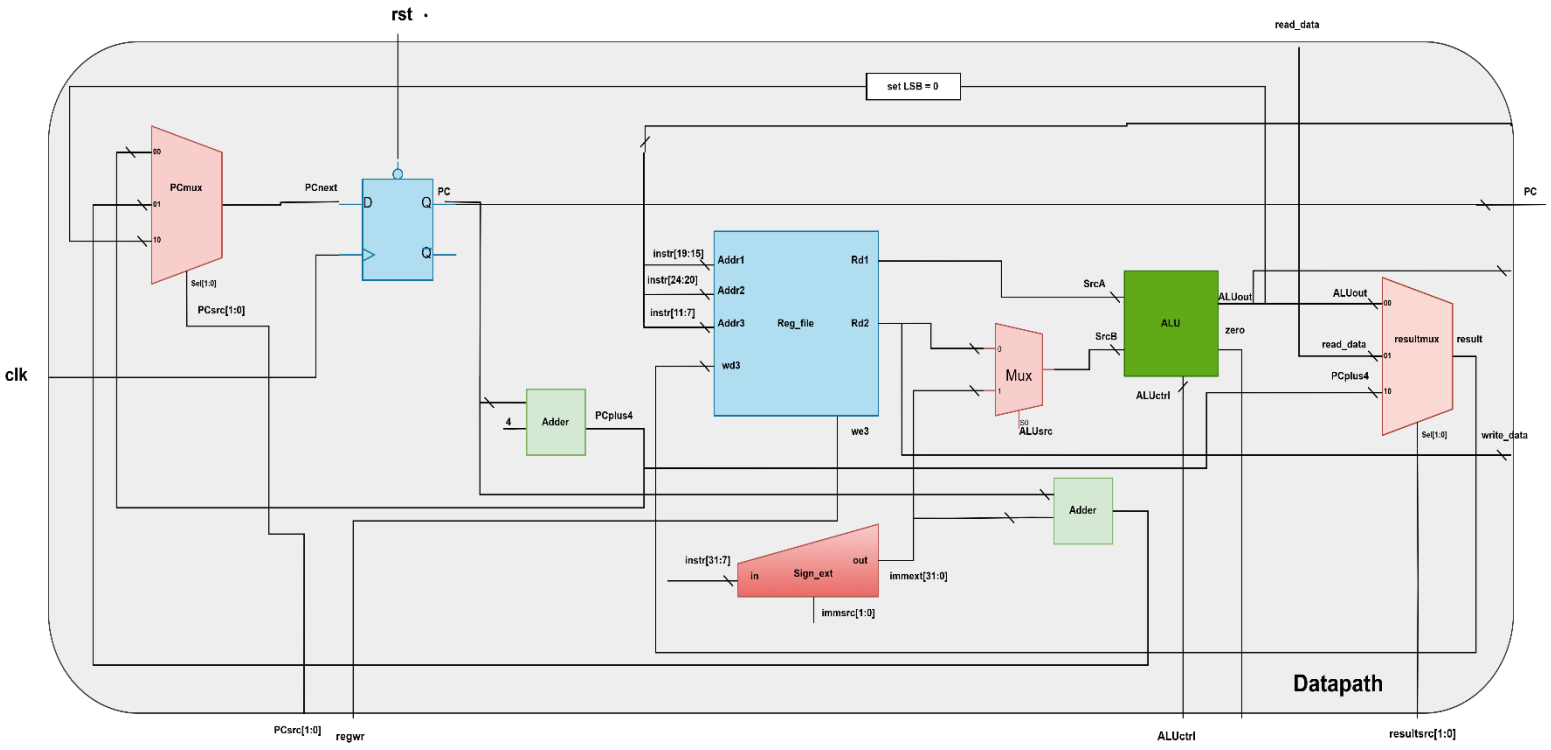
2.block diagram



2.1 control unit



2.2 data path



3. Design codes:

3.1 mux2x1

```
module mux2x1 #(parameter n = 32)(
input wire sel    ,
input wire [n-1:0] in0 , in1 ,
output reg [n-1:0] out
);

always@(*)
 begin
  if(sel)
    begin
        out = in1 ;
      end
  else
      begin
        out = in0 ;
      end

end

endmodule
```

3.2 mux3x1

```
module mux3x1 #(parameter n = 32)(
input wire [1:0] sel    ,
input wire [n-1:0] in0 , in1 , in2,
output reg [n-1:0] out
);

always@(*)
 begin
  if(sel == 2'b10 )
    begin
        out = in2 ;
      end
  else if (sel == 2'b01)
      begin
        out = in1 ;
      end
    else if (sel == 2'b00)
      begin
        out = in0 ;
      end
     else
       begin
         out = in0 ;
       end
end

endmodule
```

3.3 flip flop

```
module flip_flop #(parameter n = 32) (
input wire clk,
input wire rst,
input wire [n-1:0] d,
output reg [n-1:0] q
);

always@(posedge clk or posedge rst)
begin
  if(rst)
    begin
      q <= 0 ;
    end
  else
    begin
      q <= d ;
    end
end

endmodule
```

3.4 adder

```
module adder (
input wire [31:0] in1,
input wire [31:0] in2,
output wire [31:0] out
);

assign  out = in1 + in2 ;

endmodule
```

3.5 sign extend

```
module Sign_ext (
input wire [31:7] in,
input wire [1:0] opcode,
output reg [31:0] out
);

always@(*)
begin
  case(opcode)
    2'b00 : //I-type instruction
    out = { {20{in[31]}} , in[31:20] } ;
    2'b01 : //S-type instruction
    out = { {20{in[31]}} , in[31:25] , in[11:7] } ;
    2'b10 : //B-type instruction
    out = { {20{in[31]}} , in[7] , in[31:25] , in[11:8] , 1'b0} ;
    2'b11 : //J-type instruction
    out = { {12{in[31]}} , in[19:12] , in[20] , in[30:21] , 1'b0} ;
    default : out = 32'hxxxxxxxx ;
  endcase
end

endmodule
```

3.6 reg file

```
module Reg_file (
input wire        clk,
input wire        rst,
input wire [4:0] Addr1,
input wire [4:0] Addr2,
input wire [4:0] Addr3,
input wire [31:0] wd3,
input wire         we3,
output reg [31:0] rd1,
output reg [31:0] rd2
);

reg [31:0] temp [0:31] ;
integer i ;
//clocked writing
always@(posedge clk or posedge rst)
begin
  if(rst)
    begin
      for(i=0 ; i < 16 ; i = i+1)
      begin
        temp[i] <= 'h0 ;
      end
    end
  if(we3)
    begin
      temp[Addr3] <= wd3 ;
    end
end

//combinational reading
always@(*)
begin
  if(Addr1 == 0)
    begin
       rd1 = 0 ;
    end
  else
    begin
       rd1 = temp[Addr1] ;
    end

  if(Addr2 == 0)
    begin
       rd2 = 0 ;
    end
  else
    begin
       rd2 = temp[Addr2] ;
    end

end

endmodule
```

3.7 ALU

```
module Alu (
input wire [1:0] ALUctrl ,
input wire [31:0] A , B   ,
output reg [31:0] ALUout,
output wire       zero
);

assign zero = (ALUout == 0)? 1 : 0 ;

always@(*)
 begin
   case(ALUctrl)
     2'b00:  ALUout = A + B ;
     2'b01:  ALUout = A - B ;
     2'b10:  ALUout = A & B ;
     2'b11:  ALUout = A | B ;
     default:  ALUout = 0 ;
      endcase
end

endmodule
```

3.8 ALU decoder

```
module Alu_decoder (
input wire [1:0]  ALUop,
input wire [2:0]  funct3,
input wire        funct7_5,
input wire        op_5,
output reg [1:0] ALUctrl
);

always@(*)
begin
  case(ALUop)
    2'b00 : ALUctrl = 2'b00 ; //adding for lw,sw,jalr
    2'b01 : ALUctrl = 2'b01 ; //subtructing for beq,bne
    2'b10 : //R,I-type instructions
    begin
      case(funct3)
        3'b000 :
        begin
        if({op_5,funct7_5} == 3'b11)
            begin
              ALUctrl = 2'b01 ; //subtraction for sub
            end
          else
            begin
              ALUctrl = 2'b00 ; //adding for add,addi
            end
        end
        3'b111 : ALUctrl = 2'b10 ;//anding for and,andi
        3'b110 : ALUctrl = 2'b11 ;//oring for or,ori
        default : ALUctrl = 2'bxx ;
      endcase
    end
    default : ALUctrl = 2'bxx ;
  endcase
end
endmodule
```

3.9 main decoder

```
module main_decoder (
input wire [6:0] op,
output reg         jump,
output reg         jalr,
output reg         branch,
output reg [1:0]   immsrc,
output reg         ALUsrc,
output reg [1:0]   ALUop,
output reg [1:0]   resultsrc,
output reg         regwr,
output reg         memwr
);
always@(*)
begin
  case(op)
    7'b0000011 : //lw instruction
    begin
      regwr     = 1'b1  ;
      immsrc    = 2'b00 ;
      ALUsrc    = 1'b1  ;
      memwr     = 1'b0  ;
      resultsrc = 2'b01 ;
      branch    = 1'b0  ;
      ALUop     = 2'b00 ;
      jump      = 1'b0  ;
      jalr      = 1'b0  ;
    end
    7'b0100011 : //sw instruction
    begin
      regwr     = 1'b0  ;
      immsrc    = 2'b01 ;
      ALUsrc    = 1'b1  ;
      memwr     = 1'b1  ;
      resultsrc = 2'bxx ;
      branch    = 1'b0  ;
      ALUop     = 2'b00 ;
      jump      = 1'b0  ;
      jalr      = 1'b0  ;
    end
    7'b0110011 : //R-type instruction
    begin
      regwr     = 1'b1  ;
      immsrc    = 2'bxx ;
      ALUsrc    = 1'b0  ;
      memwr     = 1'b0  ;
      resultsrc = 2'b00 ;
      branch    = 1'b0  ;
      ALUop     = 2'b10 ;
      jump      = 1'b0  ;
      jalr      = 1'b0  ;
    end
    7'b1100011 : //beq instruction
    begin
      regwr     = 1'b0  ;
      immsrc    = 2'b10 ;
      ALUsrc    = 1'b0  ;
      memwr     = 1'b0  ;
      resultsrc = 2'bxx ;
      branch    = 1'b1  ;
      ALUop     = 2'b01 ;
      jump      = 1'b0  ;
      jalr      = 1'b0  ;
    end
```

```
    7'b0010011 : //I-type instruction (except jalr)
    begin
      regwr     = 1'b1  ;
      immsrc    = 2'b00 ;
      ALUsrc    = 1'b1  ;
      memwr     = 1'b0  ;
      resultsrc = 2'b00 ;
      branch    = 1'b0  ;
      ALUop     = 2'b10 ;
      jump      = 1'b0  ;
      jalr      = 1'b0  ;
    end
    7'b1101111 : //jal instruction
    begin
      regwr     = 1'b1  ;
      immsrc    = 2'b11 ;
      ALUsrc    = 1'bx  ;
      memwr     = 1'b0  ;
      resultsrc = 2'b10 ;
      branch    = 1'b0  ;
      ALUop     = 2'bxx ;
      jump      = 1'b1  ;
      jalr      = 1'b0  ;
    end
    7'b1100111 : //jalr instruction
    begin
      regwr     = 1'b1  ;
      immsrc    = 2'b00 ;
      ALUsrc    = 1'b1  ;
      memwr     = 1'b0  ;
      resultsrc = 2'b10 ;
      branch    = 1'b0  ;
      ALUop     = 2'b00 ;
      jump      = 1'b0  ;
      jalr      = 1'b1  ;
    end
    default :
    begin
      regwr     = 1'bx  ;
      immsrc    = 2'bxx ;
      ALUsrc    = 1'bx  ;
      memwr     = 1'bx  ;
      resultsrc = 2'bxx ;
      branch    = 1'bx  ;
      ALUop     = 2'bxx ;
      jump      = 1'bx  ;
      jalr      = 1'bx  ;
    end
  endcase
end

endmodule
```

3.10 data_path

```
module datapath (
//global inputs
input wire clk,
input wire rst,
//instr memory inputs
input wire [31:0] instr,
//data memory inputs
input wire [31:0] read_data,
//CU inputs
input wire [1:0]  PCsrc,
input wire [1:0]  immsrc,
input wire        ALUsrc,
input wire [1:0]  ALUctrl,
input wire [1:0]  resultsrc,
input wire        regwr,
//instr memory outputs
output wire [31:0] PC,
//data memory outputs
output wire        zero,
output wire [31:0] ALUout,
output wire [31:0] write_data
);

wire [31:0] PCnext , PCplus4 , PCtarget ;
wire [31:0] result ;
wire [31:0] SrcA , SrcB ;
wire [31:0] immext ;


flip_flop #(32) u_ff(
.clk(clk),
.rst(rst),
.d(PCnext),
.q(PC)
);
mux3x1 #(32) u_pcmux (
.sel(PCsrc)   ,
.in0(PCplus4) ,
.in1(PCtarget) ,
.in2( {ALUout[31:1],1'b0} ),
.out(PCnext)
);

Reg_file u_regf (
.clk(clk),
.rst(rst),
.Addr1(instr[19:15]),
.Addr2(instr[24:20]),
.Addr3(instr[11:7]),
.wd3(result),
.we3(regwr),
.rd1(SrcA),
.rd2(write_data)
);

Sign_ext u_signext(
.in(instr[31:7]),
.opcode(immsrc),
.out(immext)
);
```

```
mux2x1 #(32) u_alumux (
.sel(ALUsrc)   ,
.in0(write_data) ,
.in1(immext) ,
.out(SrcB)
);

adder u_adderplus4 (
.in1(PC),
.in2(32'd4),
.out(PCplus4)
);

adder u_addertarget (
.in1(PC),
.in2(immext),
.out(PCtarget)
);

Alu u_ALU (
.ALUctrl(ALUctrl) ,
.A(SrcA) ,
.B(SrcB) ,
.ALUout(ALUout) ,
.zero(zero)
);

mux3x1 #(32) u_resultmux (
.sel(resultsrc)   ,
.in0(ALUout) ,
.in1(read_data) ,
.in2(PCplus4),
.out(result)
);

endmodule
```

3.11 control_unit

```
module control_unit(
//instr memory inputs
input wire [6:0]  op,
input wire [2:0]  funct3,
input wire        funct7_5,
//Alu inputs
input wire        zero,
//datapath outputs
output wire [1:0]  PCsrc,
output wire [1:0]  immsrc,
output wire        ALUsrc,
output wire [1:0]  ALUctrl,
output wire [1:0]  resultsrc,
output wire        regwr,
//data memory output
output wire       memwr
);

wire [1:0] ALUop ;
wire       jump;
wire       jalr;
wire       branch;
wire       zero_new ;

assign zero_new = (funct3[0] && op == 7'b1100011)? !zero : zero ;

assign PCsrc = {jalr, ((zero_new & branch) | jump) } ;
```

```
main_decoder u_md (
.op(op),
.jump(jump),
.jalr(jalr),
.branch(branch),
.immsrc(immsrc),
.ALUsrc(ALUsrc),
.ALUop(ALUop), //
.resultsrc(resultsrc),
.regwr(regwr),
.memwr(memwr)
);

Alu_decoder u_ad (
.ALUop(ALUop),
.funct3(funct3),
.funct7_5(funct7_5),
.op_5(op[5]),
.ALUctrl(ALUctrl)
);

endmodule
```

3.12 top_RISCV

```
module top_RISCV #(parameter n = 10 , m = 32)(
input wire clk,
input wire rst,
output wire [n-1:0] addr,
output wire [m-1:0] write_data,
output wire         memwr,
output wire [31:0] read_data,
output wire [31:0] PC,
output wire [31:0] instr
);

wire [1:0]  PCsrc ;
wire [1:0]  immsrc ;
wire        ALUsrc ;
wire [1:0]  ALUctrl ;
wire [1:0]  resultsrc ;
wire        regwr ;
wire        zero ;
wire [31:0] ALUout ;

assign addr = ALUout[9:0] ;

datapath u_dp (
.clk(clk),
.rst(rst),
//instr memory inputs
.instr(instr),
//data memory inputs
.read_data(read_data),
//CU inputs
.PCsrc(PCsrc),
.immsrc(immsrc),
.ALUsrc(ALUsrc),
.ALUctrl(ALUctrl),
.resultsrc(resultsrc),
.regwr(regwr),
//instr memory outputs
.PC(PC),
//data memory outputs
.zero(zero),
.ALUout(ALUout),
.write_data(write_data)
);
```

```
control_unit u_cu (
.op(instr[6:0]),
.funct3(instr[14:12]),
.funct7_5(instr[30]),
//Alu inputs
.zero(zero),
//datapath outputs
.PCsrc(PCsrc),
.immsrc(immsrc),
.ALUsrc(ALUsrc),
.ALUctrl(ALUctrl),
.resultsrc(resultsrc),
.regwr(regwr),
//data memory output
.memwr(memwr)
);

endmodule
```

3.13 instr_rom

```
module instr_rom #(parameter n = 10 , m = 32 ) (
input wire [n-1:0] addr,
output wire [m-1:0] read_data
);

reg [m-1:0] mem [0:2**(n-2)-1] ; //n-2 for word addressable memory

initial
begin
  $readmemh("testcases.txt", mem);
end

assign read_data = mem[addr[n-1:2]] ; //[n-1:2] for word addressable memory

endmodule
```

3.14 data_ram

```
module data_ram #(parameter n = 10 , m = 32 )(
input wire    clk,
input wire    rst,
input wire    we,
input wire [n-1:0] addr,
input wire [m-1:0] write_data,
output wire [m-1:0] read_data
);

reg [7:0] mem [0:1023] ; //n-2 for word addressable memory
integer i ;

always@(posedge clk or posedge rst)
begin
  if(rst)
    begin
      for(i=0 ; i < 2**(n-2) ; i = i+1)
      begin
        mem[i] <= 'h0 ;
      end
    end
```

```
  else if(we)
    begin
      mem[addr[n-1:2]] <= write_data ; //[n-1:2] for word addressable memory
    end
end

assign read_data = mem[addr[n-1:2]] ; //[n-1:2] for word addressable memory

endmodule
```

4.simulation

4.1 top_RISCV_tb

```
`timescale 1ps/1fs
module top_RISCV_tb #(parameter n = 10 , m = 32)();

reg clk ;
reg rst ;
wire [n-1:0] addr;
wire [m-1:0] write_data;
wire         memwr ;
wire [31:0] read_data;
wire [31:0] PC;
wire [31:0] instr;

//instantiation
top_RISCV #(10,32) u_top (
.clk(clk),
.rst(rst),
.addr(addr),
.write_data(write_data),
.memwr(memwr),
.read_data(read_data),
.PC(PC),
.instr(instr)
);

instr_rom #(10,32) u_ins_rom (
.addr(PC[9:0]),
.read_data(instr)
);

data_ram #(10,32) u_data_ram (
.clk(clk),
.rst(rst),
.we(memwr),
.addr(addr),
.write_data(write_data),
.read_data(read_data)
);

initial
begin
  clk = 0 ;
  forever #500 clk = ~clk ;
end


initial
begin
  rst = 1'b1  ;
  #1000
  rst = 1'b0 ;
end
```

```
always@(negedge clk)
begin
  if(memwr)
    begin
      if(write_data == 2 && addr == 96)
        begin
          $display("time = %0t , write_data = %4d ,  addr = %8d ,testcase1 passed (first sw)",
$time , write_data , addr) ;
        end
      else if(write_data == 4 && addr == 92)
        begin
          $display("time = %0t , write_data = %4d ,  addr = %8d ,testcase2 passed (second sw)",
$time , write_data , addr) ;
          $stop ;
        end
      else
        begin
          $display("time = %0t , write_data = %4d ,  addr = %8d ,testcase1,2 faild", $time ,
write_data , addr) ;
          $stop ;
        end
    end
end
endmodule
```

4.2 assembly code and its output

main: addi x2, x0, 5 # x2 = 5 0 00500113

addi x3, x0, 12 # x3 = 12 4 00C00193

addi x7, x3, -9 # x7 = (12 - 9) = 3 8 FF718393

or x4, x7, x2 # x4 = (3 OR 5) = 7 C 0023E233

and x5, x3, x4 # x5 = (12 AND 7) = 4 10 0041F2B3

add x5, x5, x4 # x5 = 4 + 7 = 11 14 004282B3

beq x5, x7, end # shouldn't be taken 18 02728463

beq x4, x0, around # shouldn’t be taken 1C 00020463

addi x5, x0, 0 # x5 = 0 + 0 = 0 20 00000293

around: add x7, x4, x5 # x7 = (7 + 0) = 7 24 005203B3

sub x7, x7, x2 # x7 = (7 - 5) = 2 28 402383B3

sw x7, 84(x3) # **[96] = 2** 2C 0471AA23 (testcase1)

lw x2, 96(x0) # x2 = [96] = 2 30 06002103

add x9, x2, x5 # x9 = (2 + 0) = 2 34 005104B3

jal x3, end # jump to end, x3 = 0x3C 38 008001EF

addi x2, x0, 1 # shouldn't execute 3C 00100113

end: add x2, x2, x9 # x2 = (2 + 2) = 4 40

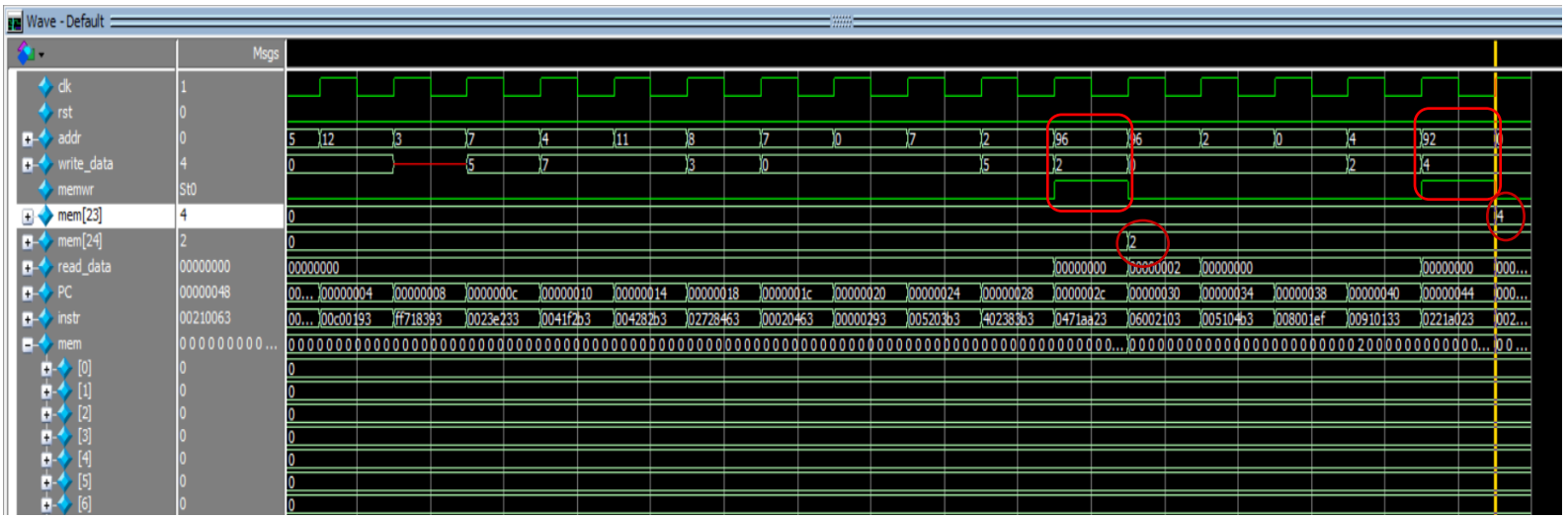
sw x2, 0x20(x3) # **[92] = 4** 44 0221A023 (testcase2)

done: beq x2, x2, done # infinite loop 48 00210063

testcases.txt content

```
00500113
00C00193
FF718393
0023E233
0041F2B3
004282B3
02728463
00020463
00000293
005203B3
402383B3
0471AA23
06002103
005104B3
008001EF
00100113
00910133
0221A023
00210063
```

4.3 simulation results



mem[23]_word addressable = mem[23*4] = mem[92]

mem[24]_word addressable = mem[24*4] = mem[96]_word addressable

```
VSIM 11> run -all
# time = 12000000 , write_data =    2 ,  addr =       96 ,testcase1 passed (first sw)
# time = 17000000 , write_data =    4 ,  addr =       92 ,testcase2 passed (second sw)
```