Project: pipelined 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.

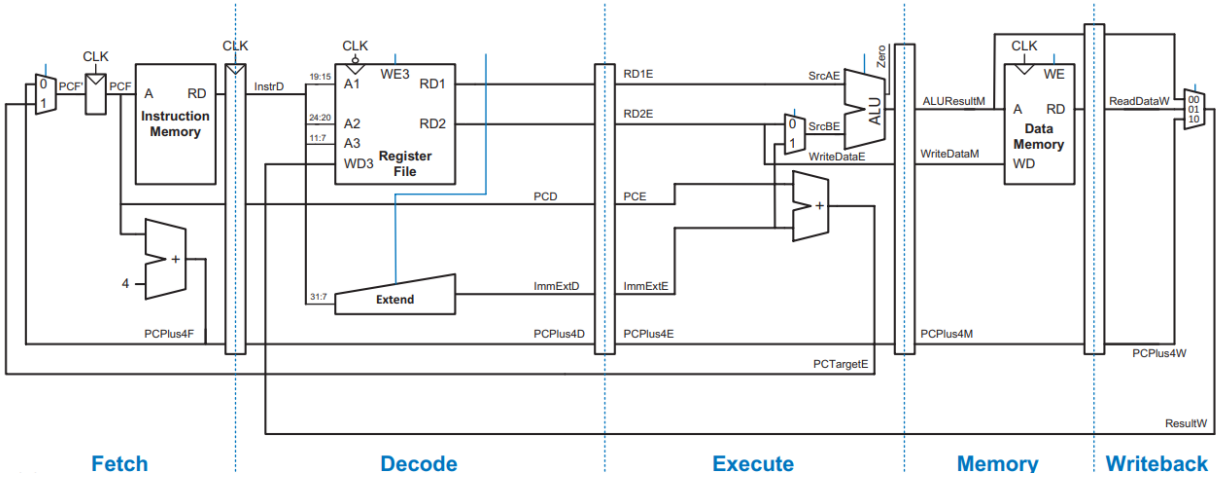
Pipeline RISC-V is an update of the single cycle RISC-V in terms of execution time and it's very fast for the programs that doesn't require forwarding the result of an instruction or using branchs and jumps.

The pipleline RISC-V contains 5 stages:

1- Fetch (F)
2- Decode (D)
3- Execute (E)
4- Memory (M)
5- Writeback (W)

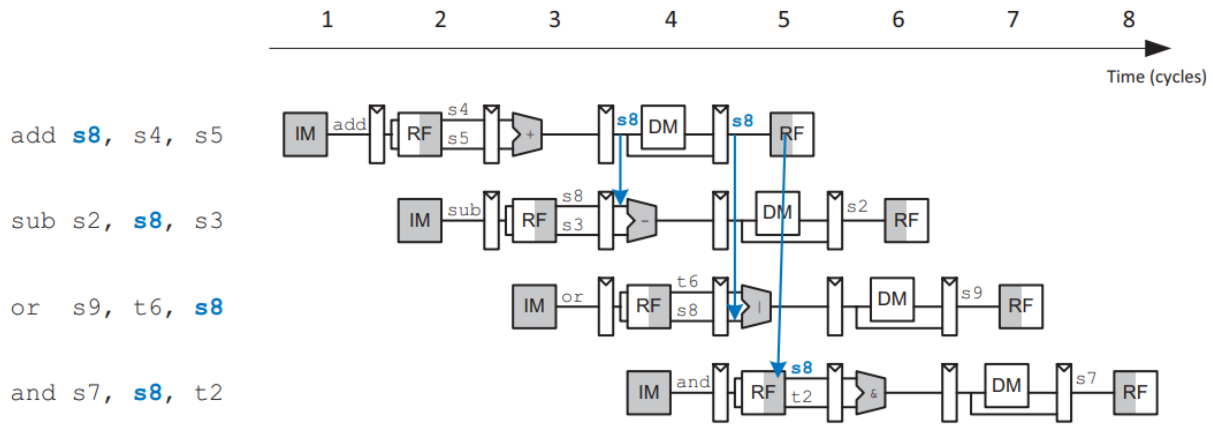
Every stage takes one smaller cycle to do its work. In the fetch stage we fetch the new instruction from the instruction memory. In the decode stage the processor read the operands from the register file and decode the instruction for the control unit. In the execute stage the ALU evaluates the output based on the operands and the ALUctrl. In the Memory stage data get saved in the data memory. In the writeback stage data get saved in the register file.

the combinational blocks between every stage and another works as one unit so the data will propagate together until the last stage. we can notice that the register file will be used twice in one instruction first it will read the operands in the decode stage and then save the data in the register file. So the best way to handle this situation is that the register file reads the operands at the positive edge of the clk and data is written back to the register file at the negative edge of the clock.

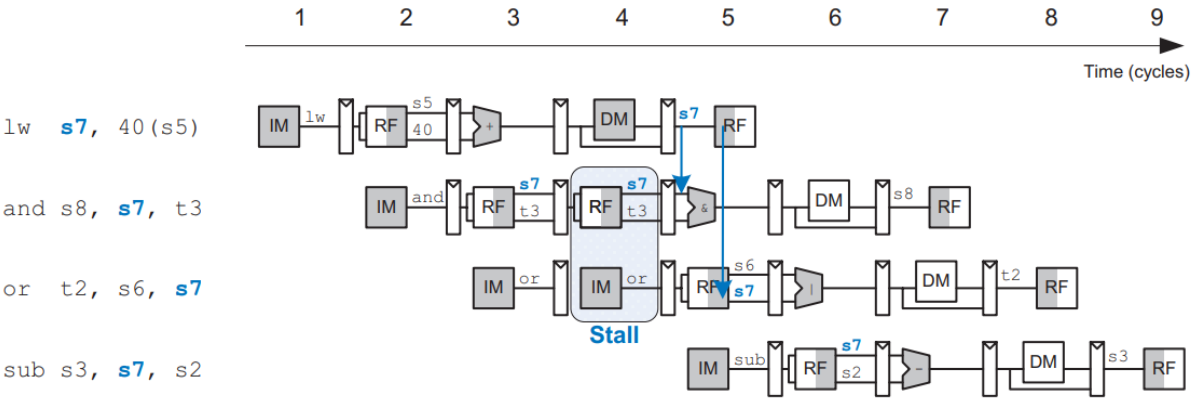


The pipeline architecture is different than the single cycle that it has **flip flops** between every stage and the next one so the data can propagate as one unit, and **two 3:1muxes** to select the operands for the ALU in case of the forwarding, and **hazard unit** to control the conditions of **forwarding, stalling, flushing**.

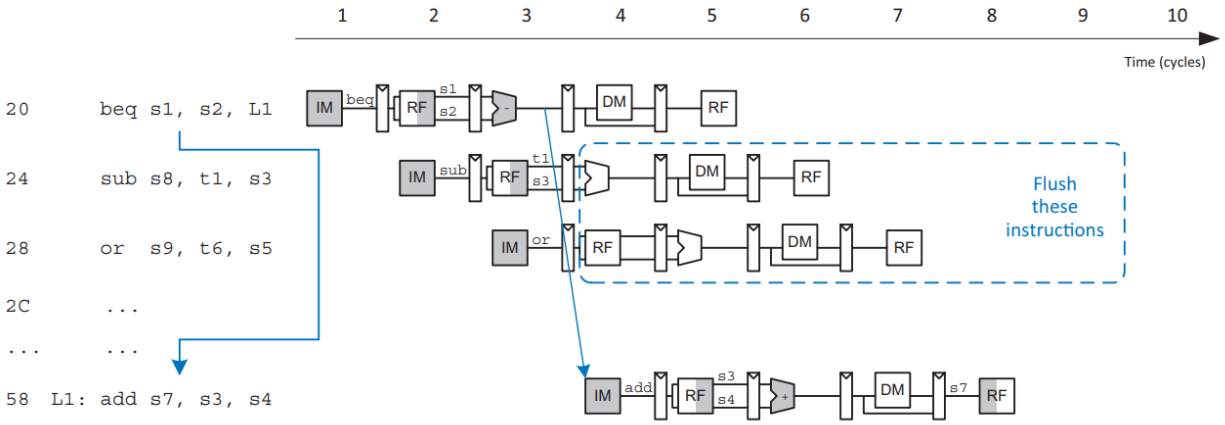
Forwarding is the case that the source register of an instruction in the execute stage(rs1E , rs2E) match the destination register of an instruction in the memory or writeback stage (rdM or rdW).



Stalling is the case that we have to disenable fetch and decode stage in order to give the load word instruction (lw) chance to read the data from the data memory as it can't be forward until the end of the memory stage. Flushing the execute stage by resetting its outputs will be necessary in this case to prevent the false data to interfere with the stalled data



Flushing is necessary in the case of branch instructions (beq) as the PC can't decide which instruction will be next whether it's the next instruction (PC+4) or the instruction at the new branch (PCtarget). So we should assume that the next instruction will be executed (PC+4) and continue as the branch will be false. In case of the branch is actually false then nothing will change, but in case the branch is true then we should flush the next two instructions and start fetching the new instruction in the branch and that will require to flush the decode and the execute stages.



To calculate the execution 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 = 1.25 (approximately for pipelined cycle), T_c is calculated by evaluate the critical path for the longest combinational logic between stages which is the excute stage.

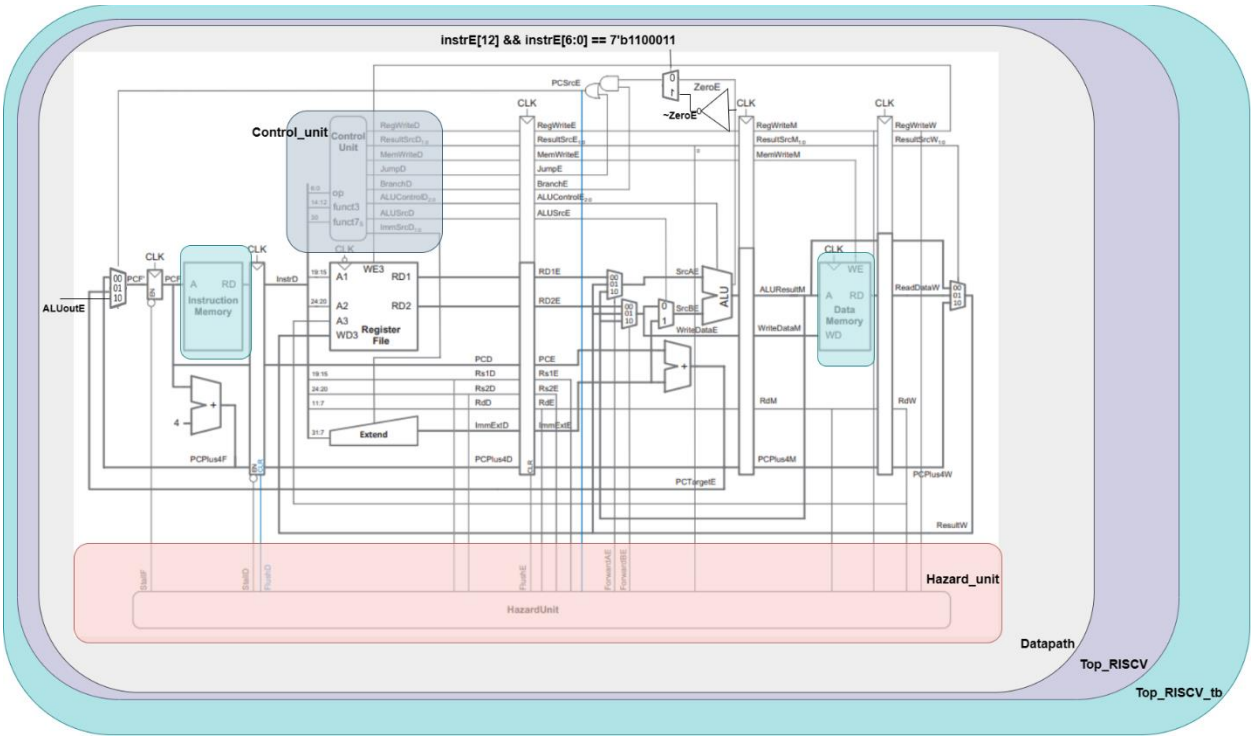
$$T_c = t_{pcq_{pc}} + t_{alu} + 4t_{mux} + t_{and-or} + t_{setup}$$

$$T_c = 10 + 50 + 4 * 10 + 10 + 10 = 120\,ps$$

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

$$T_{exec} = 16 * 1.25 * 120 = \quad 2400\ ps = 2.4\ ns$$

2. Block diagram



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 [n-1:0] d,
output reg [n-1:0] q
);

always@(posedge clk)
begin
  q <= d ;
end

endmodule
```

3.4 flip flop with reset

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

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

endmodule
```

3.5 flip flop with enable and reset

```
module flip_flop_en #(parameter n = 32) (
input wire clk,
input wire rst,
input wire en,
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 if(en)
    begin
      q <= d ;
    end
end

endmodule
```

3.6 adder

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

assign  out = in1 + in2 ;

endmodule
```

3.7 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
```

3.8 reg file

```
module Reg_file (
input wire       clk,
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@(negedge clk )
begin
  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
```

3.9 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
```

3.10 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
```

3.11 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
```

3.12 data_path

```
module datapath (
//global inputs
input wire clk,
input wire rst,
//instr memory inputs
input wire [31:0] instrF,
//data memory inputs
input wire [31:0] read_dataM,
//CU inputs
input wire [1:0]  immsrcD,
input wire        ALUsrcD,
input wire [1:0]  ALUctrlD,
input wire [1:0]  resultsrcD,
input wire        regwrD,
input wire        jumpD,
input wire        jalrD,
input wire        branchD,
input wire        memwrD,
//hazard unit inputs
input wire [1:0] forwardAE,
input wire [1:0] forwardBE,
input wire       stallF,
input wire       stallD,
input wire       flushE,
input wire       flushD,
//CU outputs
output wire [31:0] instrD,
//hazard unit outputs
output wire [4:0] rs1E,
output wire [4:0] rs2E,
output wire [4:0] rdM,
output wire [4:0] rdW,
output wire       regwrM,
output wire       regwrW,
output wire [4:0] rs1D,
output wire [4:0] rs2D,
output wire [4:0] rdE,
output wire       resultsrcE0,
output wire       PCsrcE0,
//instr memory outputs
output wire [31:0] PCF,
//data memory outputs
output wire [31:0] ALUoutM,
output wire [31:0] write_dataM,
output wire        memwrM
);
```

```
wire [31:0] PCnext , PCplus4F , PCplus4D , PCplus4E , PCplus4M  , PCplus4W, PCtargetE , PCD, PCE ;
wire [31:0] resultW ;
wire [31:0] SrcA , SrcB ;
wire [31:0] immextD , immextE ;

wire [31:0] instrE ;
wire        jalrE ,branchE , jumpE,regwrE , memwrE , ALUsrcE ;
wire [31:0] rd1D, rd2D ,rd1E, rd2E ;
wire [31:0] ALUoutE , ALUoutW;
wire [31:0] write_dataE ;
wire [1:0]  PCsrcE , ALUctrlE;
wire [1:0]  resultsrcE , resultsrcM , resultsrcW ;
wire [31:0] read_dataW ;
wire [4:0] rdD ;

wire zero ;
wire zero_new ;
assign zero_new = (instrE[12] && instrE[6:0] == 7'b1100011)? !zero : zero ;
assign PCsrcE = {jalrE, ((zero_new & branchE) | jumpE) } ;

assign resultsrcE0 = resultsrcE[0] ;
assign PCsrcE0 = PCsrcE[0] ;
assign rs1D = instrD[19:15] ;
assign rs2D = instrD[24:20] ;
assign rdD = instrD[11:7] ;

//flip flops between fetch and decode
flip_flop_en #(32) u_ff1(
.clk(clk),
.rst(flushD),
.en(~stallD),
.d(instrF),
.q(instrD)
);

flip_flop_en #(32) u_ff2(
.clk(clk),
.rst(flushD),
.en(~stallD),
.d(PCF),
.q(PCD)
);

flip_flop_en #(32) u_ff3(
.clk(clk),
.rst(flushD),
.en(~stallD),
.d(PCplus4F),
.q(PCplus4D)
);

//flip flops between decode and excute
flip_flop_rst #(1) u_ff4(
.clk(clk),
.rst(flushE),
.d(regwrD),
.q(regwrE)
);

flip_flop_rst #(2) u_ff5(
.clk(clk),
.rst(flushE),
.d(resultsrcD),
.q(resultsrcE)
);
```

```
flip_flop_rst #(1) u_ff6(
.clk(clk),
.rst(flushE),
.d(memwrD),
.q(memwrE)
);

flip_flop_rst #(1) u_ff7(
.clk(clk),
.rst(flushE),
.d(jumpD),
.q(jumpE)
);

flip_flop_rst #(1) u_ff8(
.clk(clk),
.rst(flushE),
.d(jalrD),
.q(jalrE)
);

flip_flop_rst #(1) u_ff9(
.clk(clk),
.rst(flushE),
.d(branchD),
.q(branchE)
);

flip_flop_rst #(2) u_ff10(
.clk(clk),
.rst(flushE),
.d(ALUctrlD),
.q(ALUctrlE)
);


flip_flop_rst #(1) u_ff11(
.clk(clk),
.rst(flushE),
.d(ALUsrcD),
.q(ALUsrcE)
);

flip_flop_rst #(32) u_ff12(
.clk(clk),
.rst(flushE),
.d(rd1D),
.q(rd1E)
);

flip_flop_rst #(32) u_ff13(
.clk(clk),
.rst(flushE),
.d(rd2D),
.q(rd2E)
);

flip_flop_rst #(32) u_ff14(
.clk(clk),
.rst(flushE),
.d(PCD),
.q(PCE)
);

flip_flop_rst #(5) u_ff15(
.clk(clk),
.rst(flushE),
.d(rs1D),
.q(rs1E)
);
```

```
flip_flop_rst #(5) u_ff16(
.clk(clk),
.rst(flushE),
.d(rs2D),
.q(rs2E)
);

flip_flop_rst #(5) u_ff17(
.clk(clk),
.rst(flushE),
.d(rdD),
.q(rdE)
);

flip_flop_rst #(32) u_ff18(
.clk(clk),
.rst(flushE),
.d(immextD),
.q(immextE)
);

flip_flop_rst #(32) u_ff19(
.clk(clk),
.rst(flushE),
.d(instrD),
.q(instrE)
);

flip_flop_rst #(32) u_ff20(
.clk(clk),
.rst(flushE),
.d(PCplus4D),
.q(PCplus4E)
);
//flip flops between excute and memory
flip_flop #(1) u_ff21(
.clk(clk),
.d(regwrE),
.q(regwrM)
);

flip_flop #(2) u_ff22(
.clk(clk),
.d(resultsrcE),
.q(resultsrcM)
);

flip_flop #(1) u_ff23(
.clk(clk),
.d(memwrE),
.q(memwrM)
);

flip_flop #(32) u_ff24(
.clk(clk),
.d(ALUoutE),
.q(ALUoutM)
);

flip_flop #(32) u_ff25(
.clk(clk),
.d(write_dataE),
.q(write_dataM)
);

flip_flop #(5) u_ff26(
.clk(clk),
.d(rdE),
.q(rdM)
);
```

```
flip_flop #(32) u_ff27(
.clk(clk),
.d(PCplus4E),
.q(PCplus4M)
);

//flip flops between memory and writeback
flip_flop #(1) u_ff28(
.clk(clk),
.d(regwrM),
.q(regwrW)
);

flip_flop #(2) u_ff29(
.clk(clk),
.d(resultsrcM),
.q(resultsrcW)
);

flip_flop #(32) u_ff30(
.clk(clk),
.d(ALUoutM),
.q(ALUoutW)
);

flip_flop #(32) u_ff31(
.clk(clk),
.d(read_dataM),
.q(read_dataW)
);

flip_flop #(5) u_ff32(
.clk(clk),
.d(rdM),
.q(rdW)
);

flip_flop #(32) u_ff33(
.clk(clk),
.d(PCplus4M),
.q(PCplus4W)
);

flip_flop_en #(32) u_ff(
.clk(clk),
.rst(rst),
.en(~stallF),
.d(PCnext),
.q(PCF)
);

mux3x1 #(32) u_pcmux (
.sel(PCsrcE)   ,
.in0(PCplus4F) ,
.in1(PCtargetE) ,
.in2( {ALUoutE[31:1],1'b0} ),
.out(PCnext)
);

Reg_file u_regf (
.clk(clk),
.Addr1(rs1D),
.Addr2(rs2D),
.Addr3(rdW),
.wd3(resultW),
.we3(regwrW),
.rd1(rd1D),
.rd2(rd2D)
);
```

```
Sign_ext u_signext(
.in(instrD[31:7]),
.opcode(immsrcD),
.out(immextD)
);

mux3x1 #(32) u_forwardAEmux (
.sel(forwardAE)   ,
.in0(rd1E) ,
.in1(resultW) ,
.in2(ALUoutM),
.out(SrcA)
);

mux3x1 #(32) u_forwardBEmux (
.sel(forwardBE)   ,
.in0(rd2E) ,
.in1(resultW) ,
.in2(ALUoutM),
.out(write_dataE)
);

mux2x1 #(32) u_alumux (
.sel(ALUsrcE)   ,
.in0(write_dataE) ,
.in1(immextE) ,
.out(SrcB)
);

adder u_adderplus4 (
.in1(PCF),
.in2(32'd4),
.out(PCplus4F)
);

adder u_addertarget (
.in1(PCE),
.in2(immextE),
.out(PCtargetE)
);

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

mux3x1 #(32) u_resultmux (
.sel(resultsrcW)   ,
.in0(ALUoutW) ,
.in1(read_dataW) ,
.in2(PCplus4W),
.out(resultW)
);

endmodule
```

3.13 control_unit

```
module control_unit(
//instr memory inputs
input wire [6:0]  opD,
input wire [2:0]  funct3D,
input wire        funct7_5D,
//datapath outputs
output wire [1:0]  immsrcD,
output wire        ALUsrcD,
output wire [1:0]  ALUctrlD,
output wire [1:0]  resultsrcD,
output wire        regwrD,
output wire        jumpD,
output wire        jalrD,
output wire        branchD,
//data memory output
output wire       memwrD
);

wire [1:0] ALUopD ;

main_decoder u_md (
.op(opD),
.jump(jumpD),
.jalr(jalrD),
.branch(branchD),
.immsrc(immsrcD),
.ALUsrc(ALUsrcD),
.ALUop(ALUopD), //
.resultsrc(resultsrcD),
.regwr(regwrD),
.memwr(memwrD)
);


Alu_decoder u_ad (
.ALUop(ALUopD),
.funct3(funct3D),
.funct7_5(funct7_5D),
.op_5(opD[5]),
.ALUctrl(ALUctrlD)
);

endmodule
```

3.14 hazard unit

```
module hazard_unit (
//fowarding inputs
input wire       rst,
input wire [4:0] rs1E,
input wire [4:0] rs2E,
input wire [4:0] rdM,
input wire [4:0] rdW,
input wire       regwrM,
input wire       regwrW,
//stalling inputs
input wire [4:0] rs1D,
input wire [4:0] rs2D,
input wire [4:0] rdE,
input wire       resultsrcE0,
//flushing inputs
input wire       PCsrcE0,
```

```
//forwarding outputs
output reg [1:0] forwardAE,
output reg [1:0] forwardBE,
//stalling outputs
output reg       stallF ,
output reg       stallD,
output reg       flushE,
//flushing outputs
output reg       flushD
);

always@(*)
begin
     if( (rs1E == rdM) && regwrM && rs1E != 0 )
      begin
        forwardAE = 2'b10 ;
      end
    else if( (rs1E == rdW) && regwrW && rs1E != 0 )
      begin
        forwardAE = 2'b01 ;
      end
    else
      begin
        forwardAE <= 2'b00 ;
      end

    if( (rs2E == rdM) && regwrM && rs2E != 0 )
      begin
        forwardBE = 2'b10 ;
      end
    else if( (rs2E == rdW) && regwrW && rs2E != 0 )
      begin
        forwardBE = 2'b01 ;
      end
    else
      begin
         forwardBE = 2'b00 ;
      end

end

always@(*)
begin
  if(rst)
    begin
      stallF = 1'b0 ;
      stallD = 1'b0 ;
    end
  else if(( (rdE == rs1D) || (rdE == rs2D) ) && resultsrcE0 )
    begin
      stallF = 1'b1 ;
      stallD = 1'b1 ;
    end
  else
    begin
      stallF = 1'b0 ;
      stallD = 1'b0 ;
    end
end

always@(*)
begin
  if(rst)
    begin
      flushD = 1'b0 ;
    end
  else if(PCsrcE0)
    begin
      flushD = 1'b1  ;
    end
  else
    begin
      flushD = 1'b0 ;
    end
```

```
  if(rst)
    begin
      flushE = 1'b0 ;
    end
  else if((( (rdE == rs1D) || (rdE == rs2D) ) && resultsrcE0 ) || PCsrcE0)
    begin
      flushE = 1'b1 ;
    end
  else
    begin
      flushE = 1'b0 ;
    end

end

endmodule
```

3.15 top_RISCV

```
module top_RISCV #(parameter n = 10 , m = 32)(
input wire clk,
input wire rst,
input wire  [31:0] instrF,
output wire [n-1:0] addr,
output wire [m-1:0] write_dataM,
output wire         memwrM,
output wire [31:0] read_dataM,
output wire [31:0] PCF,
output wire [31:0] instrD
);

wire [1:0]  PCsrc ;
wire [1:0]  immsrcD ;
wire        ALUsrcD ;
wire [1:0]  ALUctrlD ;
wire [1:0]  resultsrcD ;
wire        regwrD , regwrM , regwrW ;
wire [31:0] ALUoutM ;
wire [1:0]  forwardAE, forwardBE;
wire [4:0]  rs1E , rs2E ;
wire [4:0]  rdE ,rdM , rdW ;
wire [4:0]  rs1D , rs2D ;
wire        jumpD ,jalrD , branchD ;
wire        stallF , stallD, flushD , flushE ;
wire        resultsrcE0 , PCsrcE0 ;

assign addr = ALUoutM[9:0] ;

datapath u_dp (
.clk(clk),
.rst(rst),
//instr memory inputs
.instrF(instrF),
//data memory inputs
.read_dataM(read_dataM),
//CU inputs
.immsrcD(immsrcD),
.ALUsrcD(ALUsrcD),
.ALUctrlD(ALUctrlD),
.resultsrcD(resultsrcD),
.regwrD(regwrD),
.jumpD(jumpD),
.jalrD(jalrD),
.branchD(branchD),
.memwrD(memwrD),
//hazard unit inputs
.forwardAE(forwardAE),
.forwardBE(forwardBE),
.stallF(stallF),
.stallD(stallD),
.flushE(flushE),
.flushD(flushD),
//CU outputs
.instrD(instrD),
```

```
//hazard unit outputs
.rs1E(rs1E),
.rs2E(rs2E),
.rdM(rdM),
.rdW(rdW),
.regwrM(regwrM),
.regwrW(regwrW),
.rs1D(rs1D),
.rs2D(rs2D),
.rdE(rdE),
.resultsrcE0(resultsrcE0),
.PCsrcE0(PCsrcE0),
//instr memory outputs
.PCF(PCF),
//data memory outputs
.ALUoutM(ALUoutM),
.write_dataM(write_dataM),
.memwrM(memwrM)
);

control_unit u_cu (
.opD(instrD[6:0]),
.funct3D(instrD[14:12]),
.funct7_5D(instrD[30]),
//datapath outputs
.immsrcD(immsrcD),
.ALUsrcD(ALUsrcD),
.ALUctrlD(ALUctrlD),
.resultsrcD(resultsrcD),
.regwrD(regwrD),
.jumpD(jumpD),
.jalrD(jalrD),
.branchD(branchD),
//data memory output
.memwrD(memwrD)
);


hazard_unit u_hu (
.rst(rst),
.rs1E(rs1E),
.rs2E(rs2E),
.rdM(rdM),
.rdW(rdW),
.regwrM(regwrM),
.regwrW(regwrW),
//stalling inputs
.rs1D(rs1D),
.rs2D(rs2D),
.rdE(rdE),
.resultsrcE0(resultsrcE0),
//flushing inputs
.PCsrcE0(PCsrcE0),
//forwarding outputs
.forwardAE(forwardAE),
.forwardBE(forwardBE),
//stalling outputs
.stallF(stallF),
.stallD(stallD),
.flushE(flushE),
//flushing outputs
.flushD(flushD)
);

endmodule
```

3.16 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
```

3.17 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 [m-1:0] mem [0:2**(n-1)-1] ;
integer i ;

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

assign read_data = mem[addr[n-1:2]] ;

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 [31:0]  instrF ;
wire [n-1:0] addr;
wire [m-1:0] write_dataM;
wire         memwrM ;
wire [31:0] read_dataM;
wire [31:0] PCF;
wire [31:0] instrD;

//instantiation
top_RISCV #(10,32) u_top (
.clk(clk),
.rst(rst),
.instrF(instrF),
.addr(addr),
.write_dataM(write_dataM),
.memwrM(memwrM),
.read_dataM(read_dataM),
.PCF(PCF),
.instrD(instrD)
);

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

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

initial
begin
  clk = 0 ;
  forever #250 clk = ~clk ; //clk with period 500ps
end

initial
begin
  rst = 1'b1  ;
  #500
  rst = 1'b0 ;
end
```

```
always@(negedge clk)
begin
  if(memwrM)
    begin
      if(write_dataM == 2 && addr == 96)
        begin
          $display("time = %0t , write_dataM = %4d ,  addr = %8d ,testcase1 passed (first sw)",
$time , write_dataM , addr) ;
        end
      else if(write_dataM == 4 && addr == 92)
        begin
          $display("time = %0t , write_dataM = %4d ,  addr = %8d ,testcase2 passed (second
sw)", $time , write_dataM , addr) ;
              #500
          $stop ;
        end
      else
        begin
          $display("time = %0t , write_dataM = %4d ,  addr = %8d ,testcase1,2 faild", $time ,
write_dataM , 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

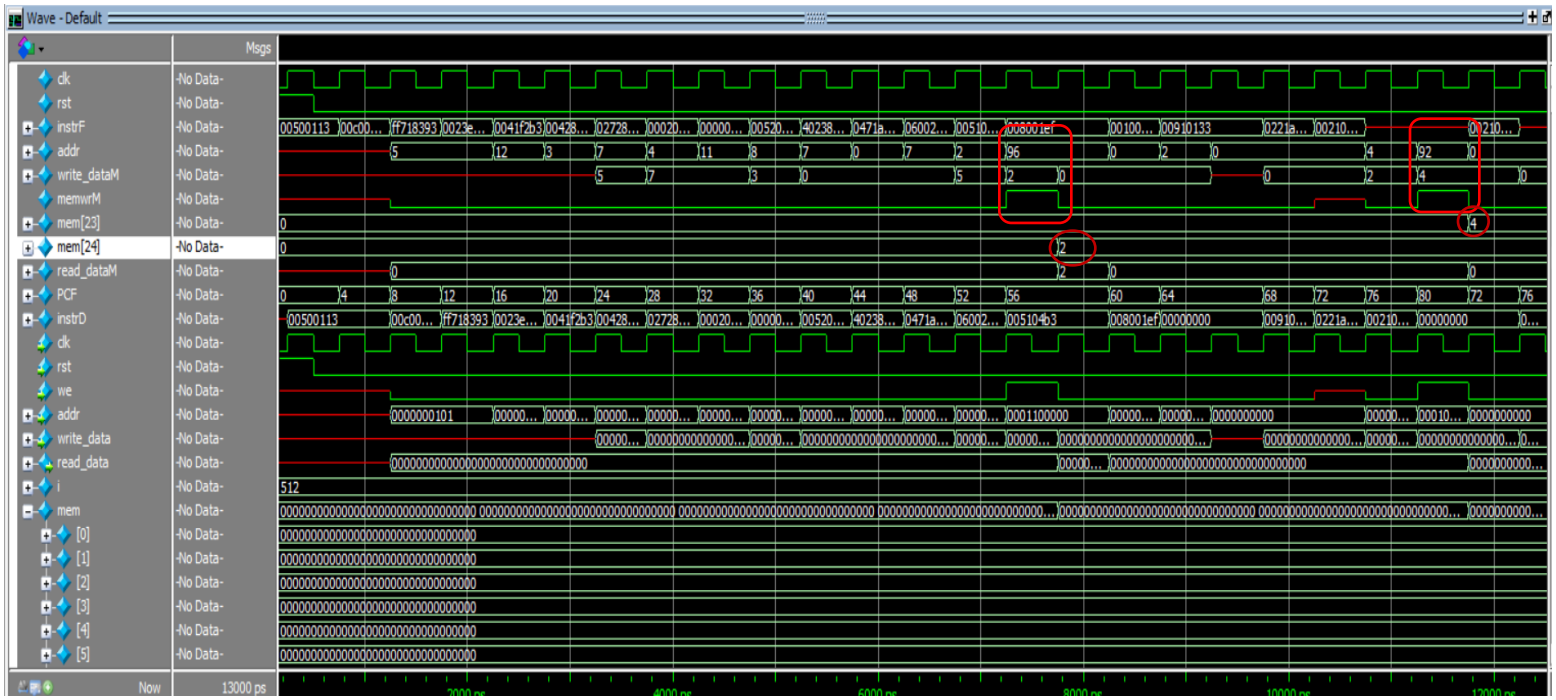
sw x2, 0x20(x3) # **[92] = 4** 44 0221A023 (testcase2)

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] = 4

mem[24]_word addressable = mem[24*4] = mem[96] = 2

```
VSIM 12> run -all
# time = 7500000 , write_dataM =    2 ,  addr =       96 ,testcase1 passed (first sw)
# time = 11500000 , write_dataM =    4 ,  addr =       92 ,testcase2 passed (second sw)
```