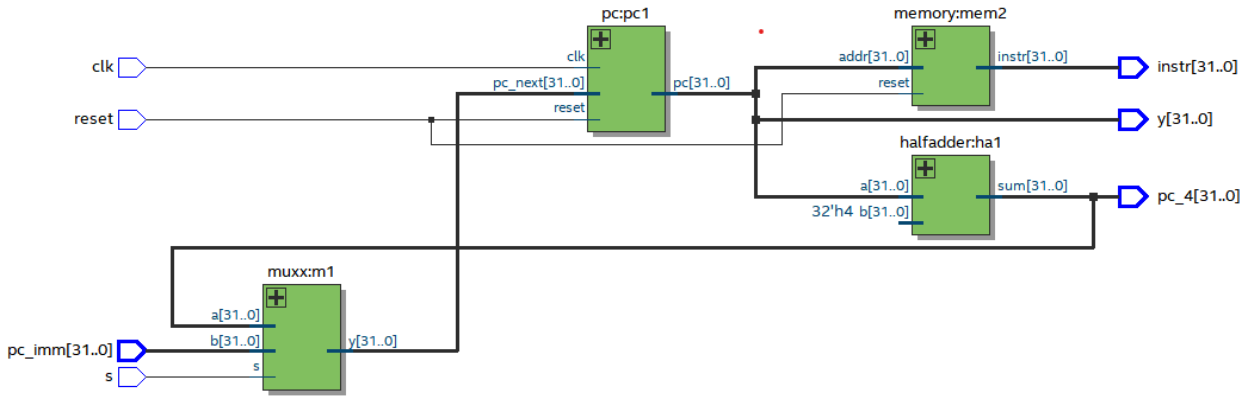
Cod RSIC-V processor code

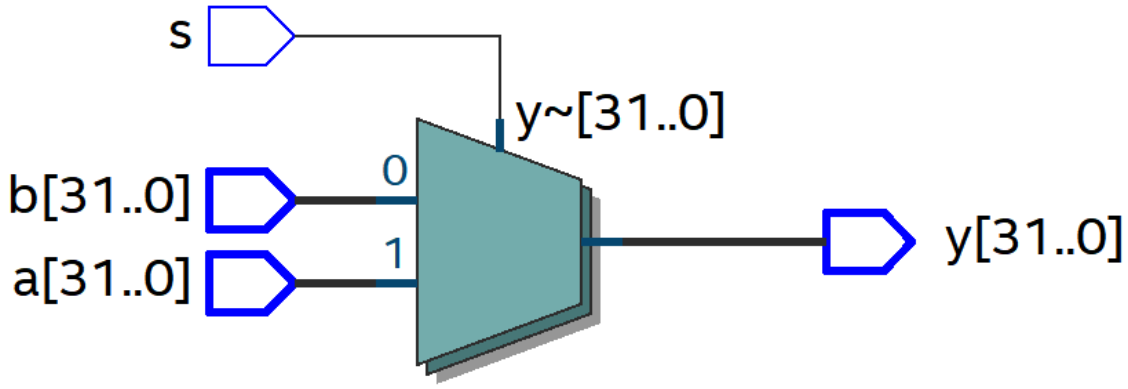
IF stage:-

```
module IFstage#(parameter n=32)(
  input logic [n-1:0] pc_imm,
  input logic s,clk,reset,
  output logic [n-1:0] instr,
  output logic [n-1:0] pc_4,
  output logic [n-1:0] y);
  logic [n-1:0] x;
 // const bit z=4;
  muxx m1(.a(pc_4),.b(pc_imm),.s(s),.y(x));
  pc pc1(.pc_next(x),.clk(clk),.reset(reset),.pc(y));
  memory mem2(.addr(y),.reset(reset),.instr(instr));
  halfadder ha1(.a(y),.b(4),.sum(pc_4));
endmodule
```



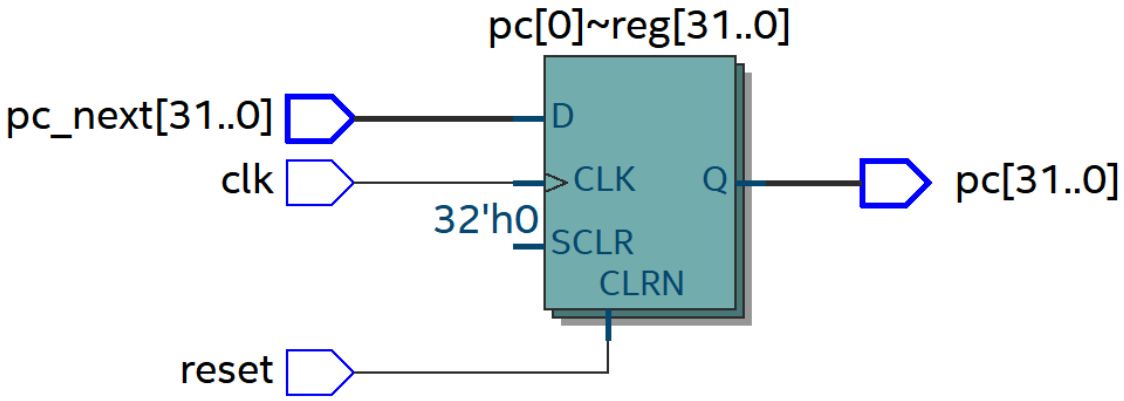
Muxx :-

```
module muxx#(parameter n=32)(
  input s,
  input logic [n-1:0]a,b,
  output logic[n-1:0] y);
  assign y = s? a: b;
endmodule
```



Pc :-

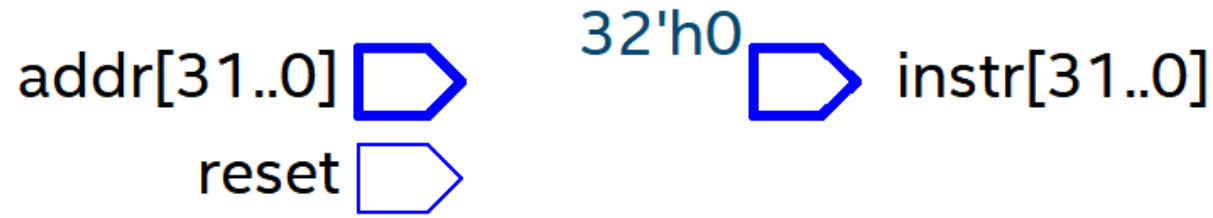
```
//syncronous--flipflop
// Code your design here
module pc#(parameter N=32)(
  input logic[N-1:0] pc_next,
  input logic clk,reset,
  output logic[N-1:0] pc);
  always @(posedge clk or posedge reset) begin
    if (reset)
       pc <= 0;  // Reset triggered by clock edge
    else
       pc <= pc_next;     // Data updated on clock edge
   end
endmodule
```



Memory:-

```
module memory#(parameter n=32, DW=32,DL=1024)(
 input logic [n-1:0] addr,
 input reset,
 output logic [n-1:0] instr);
```

```
 logic [DW-1:0]mem[0:DL-1];
  assign instr = reset?32'b0:mem[addr];
endmodule
```

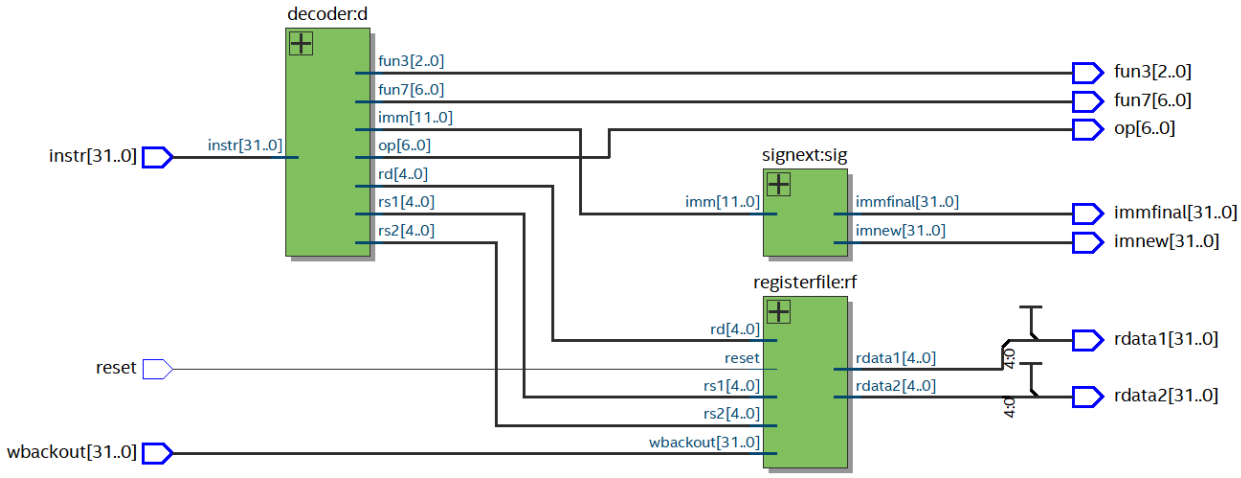


Halfadder:-

```
module halfadder#(parameter n=32)(a,b,sum,Cout);
  input logic [n-1:0] a, b;
  output logic[n-1:0] sum,Cout;
  always_comb
 begin
   sum = a ^ b;
   Cout = a & b;
  end
endmodule
```

ID stage:-

```
module IDstagae#(parameter n=32)(
              input logic reset,
              input logic [n-1:0] instr,
              input logic [n-1:0] wbackout,
              output logic [2:0]fun3,
              output logic[n-1:0] rdata1,rdata2,
              output logic [6:0]fun7,
              output logic [6:0] op,
              output logic[n-1:0] immfinal,imnew);
              logic [11:0] imm;
              logic [4:0] rd,rs1,rs2;
              decoder d(.op(op),.instr(instr),.rd(rd),.rs1(rs1),.rs2(rs2),.imm(imm),.fun3(fun3),.fun7(fun7));
              registerfile
rf(.rd(rd),.rs1(rs1),.rs2(rs2),.wbackout(wbackout),.reset(reset),.rdata1(rdata1),.rdata2(rdata2));
              signext sig(.imm(imm),.immfinal(immfinal),.imnew(imnew));
            endmodule
```



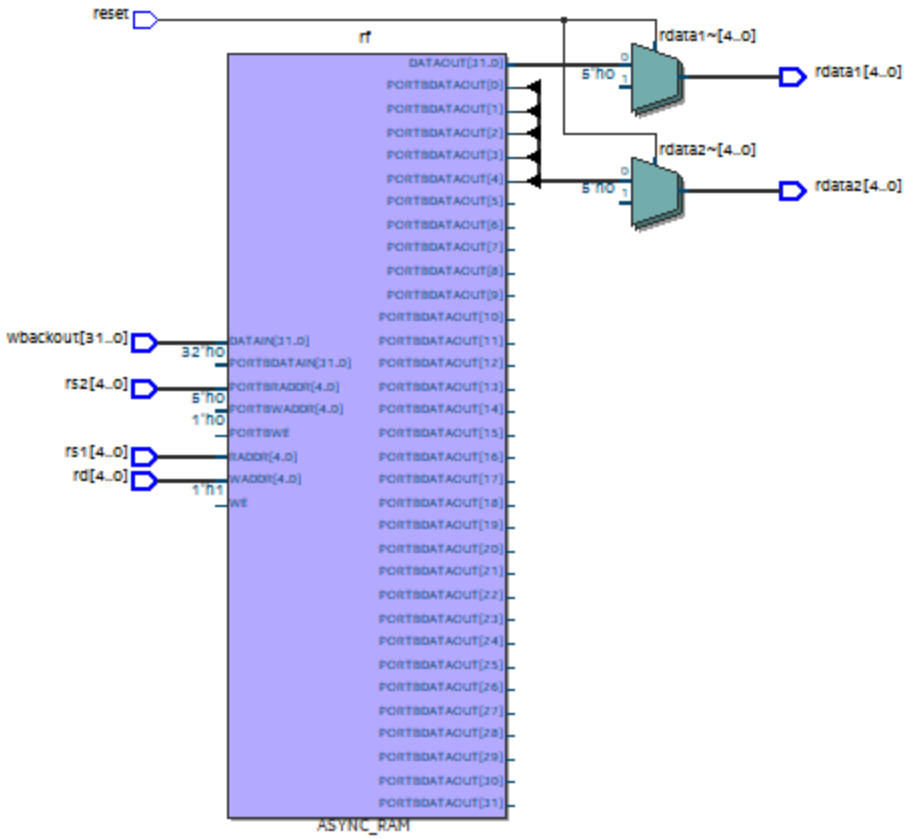
Decoder:-

```
module decoder#(parameter n=32)(instr,rd,rs1,rs2,imm,fun3,fun7,op);
input logic [n-1:0] instr;
output logic [4:0] rd,rs1,rs2;
output logic [11:0] imm;
output logic [2:0]fun3;
output logic [6:0]fun7;
output logic[6:0] op;
assign op=instr[6:0];
always@(*)
 begin
 case(op)
 7'b0110011://R-type
   begin
            rd=instr[11:7];
                    rs1=instr[19:15];
                    rs2=instr[24:20];
                    fun3=instr[14:12];
                    fun7=instr[31:25];
           end
          7'b0010011://I-type
           begin
            rd=instr[11:7];
                    rs1=instr[19:15];
                    imm=instr[31:25];
                    fun3=instr[14:12];
           end
           7'b0100011://S-type
   begin
            rd=instr[11:7];
                    rs1=instr[19:15];
                    rs2=instr[24:20];
```

```
                fun3=instr[14:12];
                imm=instr[11:5];
            end
            7'b1100011://B-type
    begin
             imm={instr[31],instr[7],instr[30:25],instr[11:6]};
                rs1=instr[19:15];
                rs2=instr[24:20];
                fun3=instr[14:12];
            end
                7'b0000011://L-type
            begin
             rd=instr[11:7];
                rs1=instr[19:15];
                imm=instr[31:25];
                fun3=instr[14:12];
            end
           endcase
          end
endmodule
```

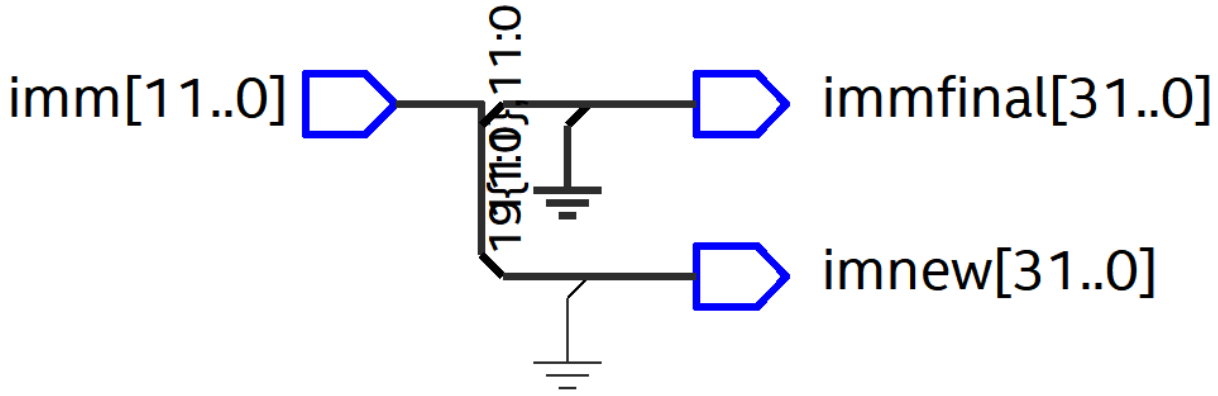
Register file:-

```
module registerfile#(parameter n=32)(
input logic[4:0]rd,rs1,rs2,
input logic [n-1:0] wbackout,
input logic reset,
output logic[4:0] rdata1,rdata2);
logic rw;
logic [n-1:0]rf[n-1:0];
assign rdata1=reset?0:rf[rs1];
assign rdata2=reset?0:rf[rs2];

always@(rw) begin
   rf[rd]=wbackout;
end

endmodule
```



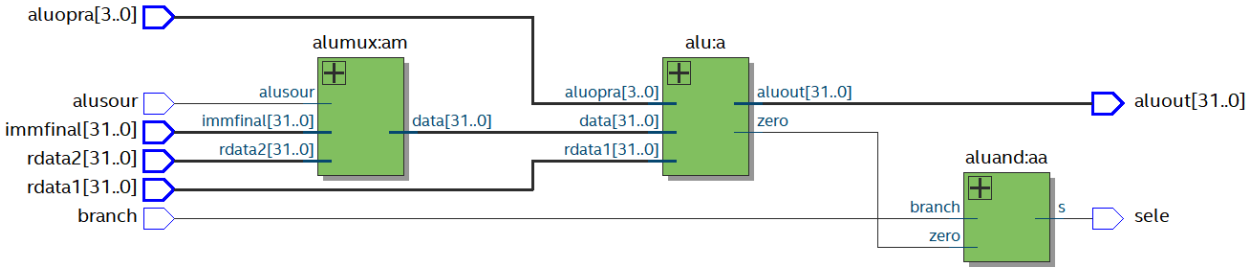
Sign extender (along with shifter):-

```
module signext#(parameter n=32)(
input logic[11:0] imm,
output logic[n-1:0] immfinal,
output logic [n-1:0]imnew);
logic[11:0] im;
assign imnew={{19{imm[11]}},imm[11:0],1'b0};
assign im=imm[11];
always@(*)
  begin
            if(im[11]==1)
            begin
                     immfinal={{20{imm[11]}}, imm};
                    end
            else
            begin
             immfinal={20'b0, imm};
            end
           end
endmodule
```



exe stage:-

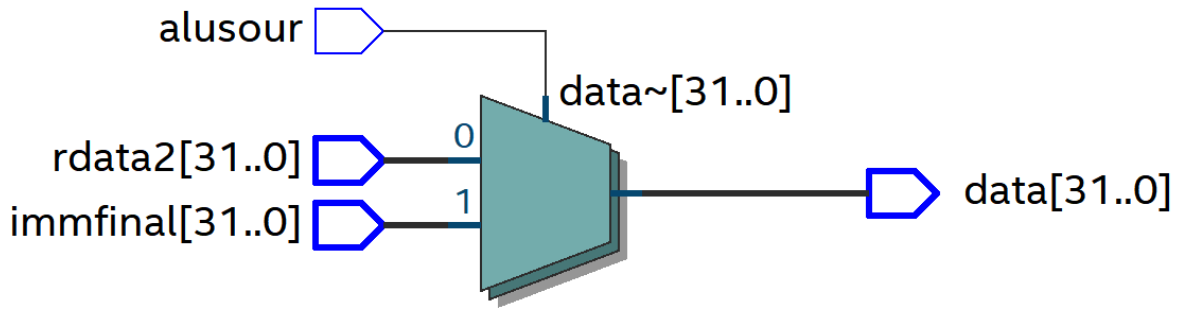
```
module EXXstage#(parameter n=32)(
input logic [n-1:0]rdata1,rdata2,immfinal,
input logic [3:0]aluopra,
input logic alusour,branch,
output logic [n-1:0]aluout,
output logic sele);
logic[n-1:0] data;
logic zero;
alumux am(.rdata2(rdata2),.immfinal(immfinal),.alusour(alusour),.data(data));
aluand aa(.branch(branch),.zero(zero),.s(sele));
alu a(.aluopra(aluopra),.data(data),.zero(zero),.rdata1(rdata1),.aluout(aluout));
endmodule
```



---

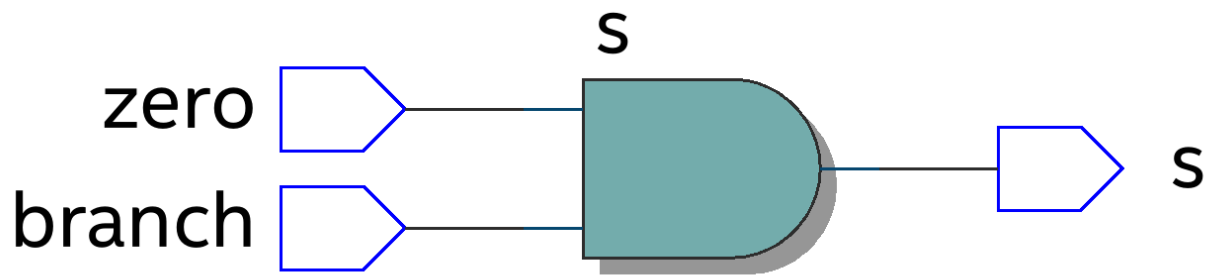
alumux:-

```
module alumux#(parameter n=32)(
input logic [n-1:0] immfinal, rdata2,
input logic alusour,
output logic[n-1:0] data );
 assign data = alusour? immfinal : rdata2;
endmodule
```



aluand:-

```
module aluand(input logic zero , branch,
output logic s);
assign s=zero & branch;
endmodule
```



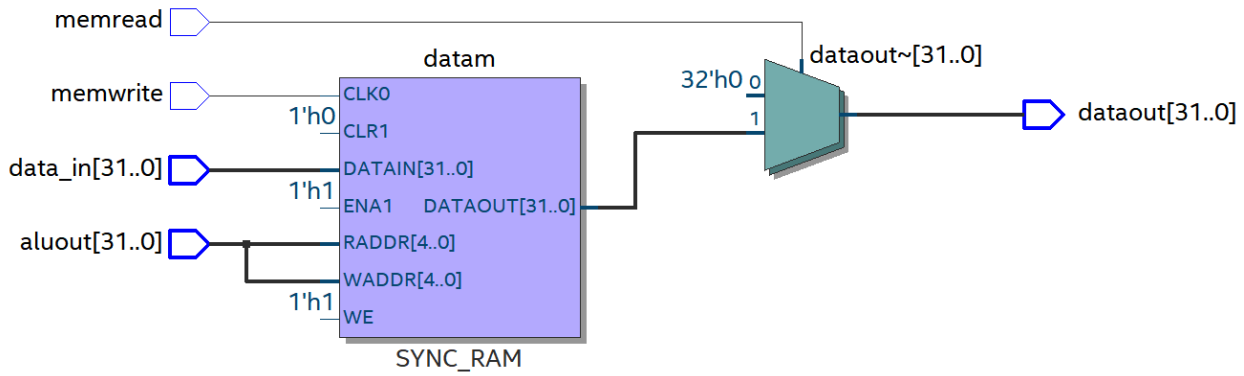
alu:-

```
module alu#(parameter n=32)(
input logic [3:0] aluopra,
input logic [n-1:0] rdata1,rdata2,data,
output logic[n-1:0] aluout,
output logic zero);
logic [3:0] x;
assign x=aluopra;
always@(*)
        begin
        case(x)
                4'b0000://and
                        begin
                                aluout=rdata1&data;
                        end
                4'b0001://or
                        begin
                                aluout=rdata1|data;
                        end
                4'b0010://add
```

```
                        begin
                                aluout=rdata1+data;
                        end
                4'b0110://subtration
                        begin
                                aluout=rdata1-data;
                        end
        endcase
if(aluout==0)
        begin
                zero=1;

        end
else
        begin
                zero=0;
        end
end
endmodule
```

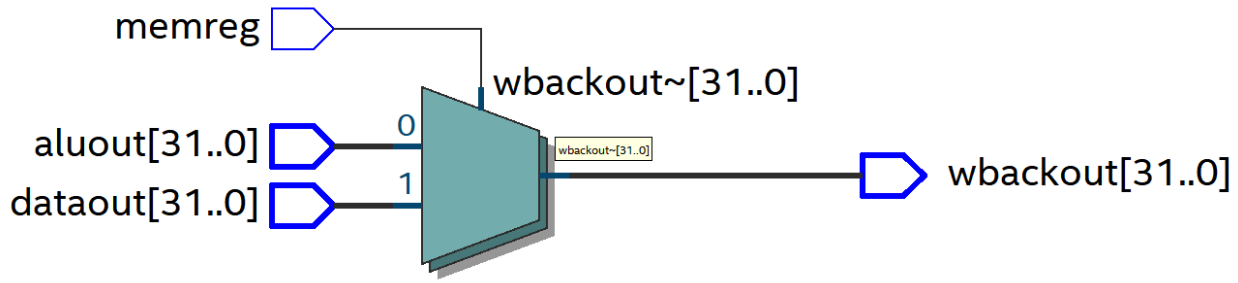
datamem stage:-

```
module datamem #(parameter n=32)(
input logic [n-1:0] aluout,data_in,
input logic memread,memwrite,
output logic [n-1:0] dataout);
logic [n-1:0] datam [n-1:0];
assign dataout=memread?datam[aluout]:32'b0;//if memread is 1 then data out is value in datam else
32'b0
always@(posedge memwrite)
        begin
          datam[aluout]=data_in;
          end
endmodule
```



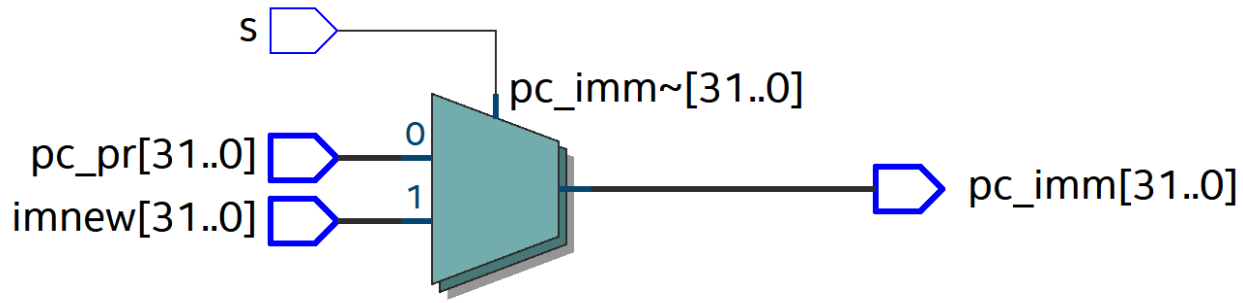
writeback stage:-

```
module writeback#(parameter n=32)(
input logic [n-1:0] aluout,dataout,
input logic memreg,
output logic[n-1:0]wbackout);
 assign wbackout = memreg? dataout : aluout;
endmodule
```



Branch mux:-

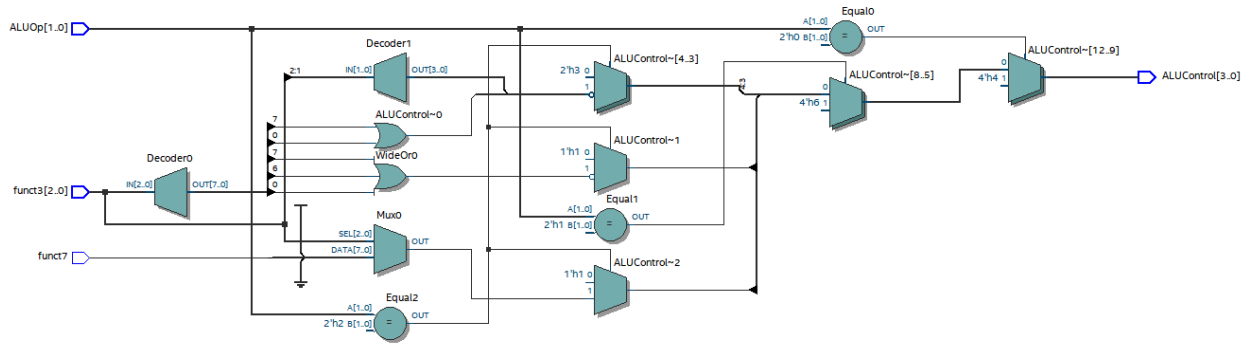
```
module branchmx#(parameter n=32)(
input logic[n-1:0] imnew,pc_pr,
input logic s,
output logic[n-1:0] pc_imm);
assign pc_imm=s?imnew:pc_pr;
endmodule
```



alucontrol code:-

```
module control( input  logic [1:0] ALUOp,
   input  logic [2:0] funct3,
   input  logic       funct7,
   output logic [3:0] ALUControl
);

   always_comb begin
```

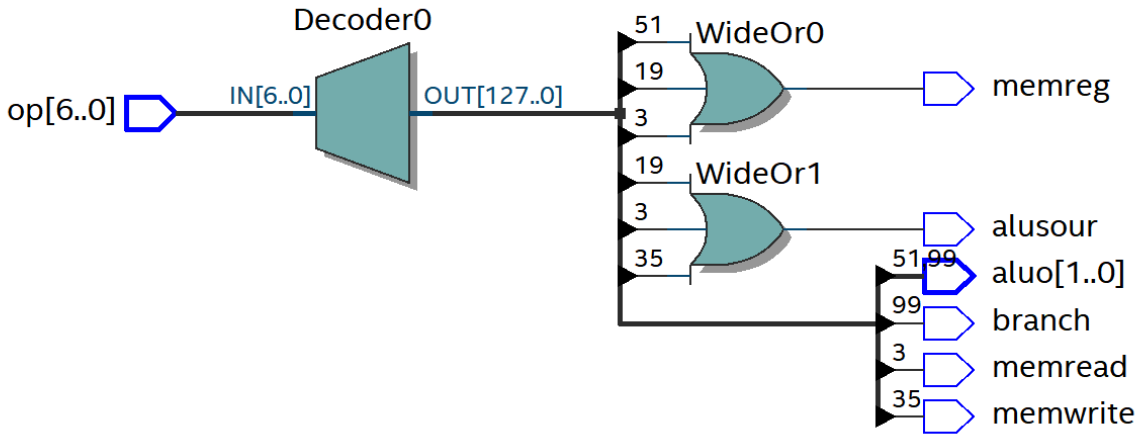
```
      ALUControl = 4'b1111;

      if (ALUOp == 2'b00) begin
         ALUControl = 4'b0010; // ADD
      end
      else if (ALUOp == 2'b01) begin
         ALUControl = 4'b0110; // SUB
      end
      else if (ALUOp == 2'b10) begin
         case (funct3)
            3'b000: ALUControl = funct7? 4'b0110 : 4'b0010; // SUB if funct7_5=1, ADD if funct7_5=0
            3'b111: ALUControl = 4'b0000; // AND
            3'b110: ALUControl = 4'b0001; // OR
            default: ALUControl = 4'b1111; // Undefined
         endcase
      end
   end

endmodule
```



Main control code:-

```
module maincontrol(input  logic [6:0] op,
   output logic      memreg,
            output logic      alusour,
   output logic      memread,
   output logic      memwrite,
   output logic      branch,
      output logic [1:0] aluo
);

   always_comb begin
      memreg = 0;
      alusour   = 0;
      memread  = 0;
      memwrite = 0;
      branch   = 0;
```

```
    aluo    = 2'b00;

    case (op)
      7'b0110011: begin      // R-type instructions
        memreg = 1;
        aluo    = 2'b10;
      end
      7'b0010011: begin      // I-type instructions
        memreg = 1;
        alusour  = 1;
        aluo  = 2'b00;
      end
      7'b0000011: begin      // Load instructions
        memreg = 1;
        memread  = 1;
        alusour  = 1;
      end
      7'b0100011: begin      // Store instructions
        memwrite = 1;
        alusour = 1;
      end
      7'b1100011: begin      // Branch instructions
        branch  = 1;
        aluo  = 2'b01;
      end
      default: begin
      end
    endcase
  end

endmodule
```



riscV code:-

```
module riscv_singlecycle #(parameter n=32)(
   input logic clk,
   input logic reset
);
          logic [n-1:0] wbackout;     // Internal signals
   logic [n-1:0] pc, pc_next, pc_imm, pc_4;
   logic [n-1:0] instr, rdata1, rdata2, immfinal,imnew,y;
   logic [n-1:0] aluout, dataout;
   logic [3:0] aluopra;       // ALU operation code
   logic alusour, memread, memwrite, memreg, branch,sele; // Control signals
   logic [2:0] fun3;          // Function code 3
          logic [3:0] alucontrol;
          logic [1:0] aluo;
   logic [6:0] fun7;// Function code 7
          logic[6:0] op;
IFstage #(.n(n)) if_stage (
   .pc_imm(pc_imm),
   .s(sele),          // Select PC source
   .clk(clk),
   .reset(reset),
   .instr(instr),
   .pc_4(pc_4) ,
   .y(y)    // Connect the output port to the internal signal
);
   // ID Stage
   IDstagae  id_stage (
             .op(op),
      .reset(reset),
      .instr(instr),
                    .wbackout(wbackout),
      .fun3(fun3),
      .rdata1(rdata1),
      .rdata2(rdata2),
      .fun7(fun7),
      .immfinal(immfinal),
                    .imnew(imnew)
   );
   // EX Stage
   EXXstage #(.n(n)) ex_stage (
      .rdata1(rdata1),
      .rdata2(rdata2),
      .immfinal(immfinal),
      .aluopra(alucontrol),
      .alusour(alusour),
      .branch(branch),
      .aluout(aluout),
      .sele(sele) // Use memreg for deciding if we write back memory or ALU result
   );
```

```
// Data Memory
datamem #(.n(n)) data_mem (
    .aluout(aluout),
    .data_in(rdata2), // Assuming rdata2 is the data to write
    .memread(memread),
    .memwrite(memwrite),
    .dataout(dataout)
);
// Writeback Stage
writeback #(.n(n)) wb_stage (
    .aluout(aluout),
    .dataout(dataout),
    .memreg(memreg),
    .wbackout(wbackout)
);
        halfadder #(.n(n)) finaladder (
                .a(y),.b(imnew),.sum(pc_imm));

control  con( .ALUOp(aluo),.funct3(fun3),
                        .funct7(fun7),.ALUControl(alucontrol));


maincontrol mcon(.op(op),.memreg(memreg),.alusour(alusour),
.memread(memread),.memwrite(memwrite),.branch(branch),
.aluo(aluo));


endmodule
```