```
///////////////multiCycleDatapath_code//////////////
module multiCycleDatapath_code(
    clk,
    rst,
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
    ALUSrcB,
    RegSrc,
    ResultSrc,
    ALU_flags,
    R1out,
    R2out
);

input wire  clk;
input wire  rst;
input wire  PCWrite;
input wire  MemWrite;
input wire  IRWrite;
input wire  ImmSrc;
input wire  RegWrite;
input wire  ALUSrcA;
input wire  [1:0] AdrSrc;
input wire  [3:0] ALUControl;
input wire  [1:0] ALUSrcB;
input wire  [2:0] RegSrc;
input wire  [1:0] ResultSrc;
output wire [3:0] ALU_flags;
output wire [7:0] R1out;
output wire [7:0] R2out;

wire    [15:0] WREGout;
wire    [3:0] X;
wire    [7:0] SYNTHESIZED_WIRE_0;
wire    [7:0] SYNTHESIZED_WIRE_31;
wire    [7:0] SYNTHESIZED_WIRE_2;
wire    [7:0] SYNTHESIZED_WIRE_3;
wire    [7:0] SYNTHESIZED_WIRE_32;
wire    [2:0] SYNTHESIZED_WIRE_5;
wire    [2:0] SYNTHESIZED_WIRE_6;
wire    [7:0] SYNTHESIZED_WIRE_33;
wire    [7:0] SYNTHESIZED_WIRE_34;
wire    [7:0] SYNTHESIZED_WIRE_9;
wire    [15:0] SYNTHESIZED_WIRE_11;
wire    [2:0] SYNTHESIZED_WIRE_12;
wire    [2:0] SYNTHESIZED_WIRE_13;
wire    [2:0] SYNTHESIZED_WIRE_14;
wire    [7:0] SYNTHESIZED_WIRE_16;
wire    [7:0] SYNTHESIZED_WIRE_17;
wire    [7:0] SYNTHESIZED_WIRE_35;
wire    [7:0] SYNTHESIZED_WIRE_22;
wire    [7:0] SYNTHESIZED_WIRE_23;
wire    [7:0] SYNTHESIZED_WIRE_24;
wire    [7:0] SYNTHESIZED_WIRE_25;
wire    [15:0] SYNTHESIZED_WIRE_36;




InstDataMemMC    b2v_inst(
    .clk(clk),
    .memWE(MemWrite),
    .memA(SYNTHESIZED_WIRE_0),
    .memWD(SYNTHESIZED_WIRE_31),
    .memRD(SYNTHESIZED_WIRE_36));


simpleREG    b2v_inst10(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_2),
    .SREGout(SYNTHESIZED_WIRE_9));
    defparam    b2v_inst10.W = 8;


simpleREG    b2v_inst11(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_3),
    .SREGout(SYNTHESIZED_WIRE_31));
    defparam    b2v_inst11.W = 8;


simpleREG    b2v_inst12(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_32),
    .SREGout(SYNTHESIZED_WIRE_24));
    defparam    b2v_inst12.W = 8;


muxWTwoToOne    b2v_inst13(
    .s0(RegSrc[2]),
    .I0(SYNTHESIZED_WIRE_5),
    .I1(WREGout[7:5]),
    .out(SYNTHESIZED_WIRE_12));
    defparam    b2v_inst13.W = 3;


muxWTwoToOne    b2v_inst14(
    .s0(RegSrc[1]),
    .I0(WREGout[4:2]),
    .I1(WREGout[10:8]),
    .out(SYNTHESIZED_WIRE_13));
    defparam    b2v_inst14.W = 3;


muxWTwoToOne    b2v_inst15(
    .s0(RegSrc[0]),
    .I0(WREGout[10:8]),
    .I1(SYNTHESIZED_WIRE_6),
```

```verilog
	.out(SYNTHESIZED_WIRE_14));
	defparam	b2v_inst15.W = 3;


muxWTwoToOne	b2v_inst16(
	.s0(RegSrc[0]),
	.I0(SYNTHESIZED_WIRE_33),
	.I1(SYNTHESIZED_WIRE_34),
	.out(SYNTHESIZED_WIRE_16));
	defparam	b2v_inst16.W = 8;


extendImm	b2v_inst17(
	.ImmSrc(ImmSrc),
	.Instr70(WREGout[7:0]),
	.extendedImm(SYNTHESIZED_WIRE_22));


muxWTwoToOne	b2v_inst18(
	.s0(ALUSrcA),
	.I0(SYNTHESIZED_WIRE_9),
	.I1(SYNTHESIZED_WIRE_34),
	.out(SYNTHESIZED_WIRE_17));
	defparam	b2v_inst18.W = 8;


shrinkImm	b2v_inst19(
	.Instr150(SYNTHESIZED_WIRE_11),
	.instr70(SYNTHESIZED_WIRE_25));


RegisterFileMC	b2v_inst2(
	.clk(clk),
	.rst(rst),
	.WE(RegWrite),
	.A1(SYNTHESIZED_WIRE_12),
	.A2(SYNTHESIZED_WIRE_13),
	.A3(SYNTHESIZED_WIRE_14),
	.R6(SYNTHESIZED_WIRE_33),
	.WD3(SYNTHESIZED_WIRE_16),
	.R1(R1out),
	.R2(R2out),
	.RD1(SYNTHESIZED_WIRE_2),
	.RD2(SYNTHESIZED_WIRE_3));


ConstantValueGenerator	b2v_inst20(
	.Data_on_Bus(SYNTHESIZED_WIRE_5));
	defparam	b2v_inst20.BUS_DATA = 3'b110;
	defparam	b2v_inst20.DATA_WIDTH = 3;


ConstantValueGenerator	b2v_inst21(
	.Data_on_Bus(SYNTHESIZED_WIRE_6));
	defparam	b2v_inst21.BUS_DATA = 3'b111;
	defparam	b2v_inst21.DATA_WIDTH = 3;


ConstantValueGenerator	b2v_inst22(
    .Data_on_Bus(SYNTHESIZED_WIRE_23));
    defparam    b2v_inst22.BUS_DATA = 3'b100;
    defparam    b2v_inst22.DATA_WIDTH = 8;


ALU_MC  b2v_inst3(
    .A(SYNTHESIZED_WIRE_17),
    .ALUcontrol(ALUControl),
    .B(SYNTHESIZED_WIRE_35),
    .N(X[3]),
    .Z(X[2]),
    .CO(X[1]),
    .OVF(X[0]),
    .Y(SYNTHESIZED_WIRE_32));
    defparam    b2v_inst3.W = 8;


muxWFourToOne   b2v_inst4(
    .s0(AdrSrc[0]),
    .s1(AdrSrc[1]),
    .I0(SYNTHESIZED_WIRE_34),
    .I1(SYNTHESIZED_WIRE_33),


    .out(SYNTHESIZED_WIRE_0));
    defparam    b2v_inst4.W = 8;


muxWFourToOne   b2v_inst5(
    .s0(ALUSrcB[0]),
    .s1(ALUSrcB[1]),
    .I0(SYNTHESIZED_WIRE_31),
    .I1(SYNTHESIZED_WIRE_22),
    .I2(SYNTHESIZED_WIRE_23),

    .out(SYNTHESIZED_WIRE_35));
    defparam    b2v_inst5.W = 8;


muxWFourToOne   b2v_inst6(
    .s0(ResultSrc[0]),
    .s1(ResultSrc[1]),
    .I0(SYNTHESIZED_WIRE_24),
    .I1(SYNTHESIZED_WIRE_25),
    .I2(SYNTHESIZED_WIRE_32),
    .I3(SYNTHESIZED_WIRE_35),
    .out(SYNTHESIZED_WIRE_33));
    defparam    b2v_inst6.W = 8;


writeEnableREG  b2v_inst7(
    .clk(clk),
    .rst(rst),
    .WE(IRWrite),
    .DATA(SYNTHESIZED_WIRE_36),
    .WREGout(WREGout));
    defparam    b2v_inst7.W = 16;
```

```
simpleREG   b2v_inst9(
    .clk(clk),
    .rst(rst),
    .DATA(SYNTHESIZED_WIRE_36),
    .SREGout(SYNTHESIZED_WIRE_11));
    defparam    b2v_inst9.W = 16;


writeEnableREG4PC   b2v_PCregister(
    .clk(clk),
    .rst(rst),
    .WE(PCWrite),
    .DATA(SYNTHESIZED_WIRE_33),
    .WREGout(SYNTHESIZED_WIRE_34));

assign  ALU_flags = X;

endmodule
/*for the pc counter initialize with 0*/
module writeEnableREG4PC (DATA,clk,rst,WREGout,WE);
input rst,clk,WE;
input [7:0] DATA;
output reg [7:0] WREGout=8'b00000000;


always @(posedge clk)//due to sync reset issue
begin
    if(rst==1)
        begin
            WREGout<=0;
        end

    if(WE==1)
        begin
            WREGout<=DATA;
        end


end

endmodule






module ALU_MC #(parameter W=8) (ALUcontrol,A,B,Y,N,Z,CO,OVF);
//negative and zero bits are affected by ALU op
//CO and OVF are affected by arithmetics
input [3:0] ALUcontrol;
input [W-1:0] A,B;
output reg [W-1:0] Y; //output
output reg CO,OVF,N,Z; //cpsr
wire [W-1:0] Bcomp=~B; //bitwise not
wire [W-1:0] Acomp=~A; //bitwise not
reg E;
always @(*)
begin
```

```
    case(ALUcontrol)
    4'b0000: //add
    begin
        //update the overflow bit according to the signs
                {CO, Y} = A + B;
                    if (A[W-1] ~^ B[W-1]) //if the signs are the same
                        OVF = Y[W-1] ^ A[W-1];
                    else
                        OVF = 0;


    end
    4'b0001: //subt a-b
    begin
//update the overflow bit according to the signs
                    {CO, Y} = A + Bcomp+1;
                    if ((A[W-1] ^ B[W-1]))
                        OVF = Y[W-1] ^ A[W-1];
                    else
                        OVF = 0;
    end
    4'b0010:
    begin
            Y=A&B;    //and
            CO=0;
            OVF=0;

    end

        4'b0011:
    begin

            Y=A|B;    //or
            CO=0;
            OVF=0;


    end

        4'b0100:
    begin

            Y=A^B;    //xor
            CO=0;
            OVF=0;


    end

    4'b0101:
    begin

            Y=0;    //clear
            CO=0;
            OVF=0;


    end
```

```verilog
    4'b0110:
    begin

            Y={A[6:0],A[7]};   //rol
            CO=0;
            OVF=0;


    end

        4'b0111:
    begin

            Y={A[0],A[7:1]};   //ror
            CO=0;
            OVF=0;


    end


    4'b1000:  //shift left lsl
    begin
           Y=A<<1;
            CO=0;
            OVF=0;
    end

    4'b1001:  //shift right lsr
    begin
            Y=A>>1;
            CO=0;
            OVF=0;
    end


    4'b1010:
    begin
            Y={A[7],A[7:1]};   //arithmetic shift right
            CO=0;
            OVF=0;
    end


    endcase
end


always @(*)
begin
        N = Y[W-1];
        Z = ~|Y;
end

endmodule

module extendImm
(
//data,shift no needfor the extendedImm
//memory inst imm5 for the ldr and str
//memory inst imm8 for the immediate
//branch no need to extend
//input port
input     ImmSrc,
input     [7:0] Instr70,
//output port
output    [7:0]  extendedImm
);
//ImmSrc 1 no change
//ImmSrc 0 imm5
assign extendedImm=ImmSrc ? Instr70 :{3'b0,Instr70[4:0]};


endmodule

//DATA MEMORY

 module InstDataMemMC
 (      // input ports
      input                    clk,
      input     [7:0]          memA,//memory address according to the address write or
read occur
      input     [7:0]          memWD,//memory write data, it specifies the memory data
which can be written
      input                    memWE,//memory write enable
      // output port
      output    [15:0]          memRD
 );


      reg        [15:0]         DATAmem  [255:0];

        //also maximum PC value is 252
        //however PC values are 0-4-8...252
        //memWD values are extended to 16 bits
        //initialize the memory

        integer i;
      initial
          begin
                //instructionMemory initialization
                //we have 16 bits in the memORY it is used instruction
                //instructions
                /*
                    register file initial contents
                    register_R[3] = 8'b00001111; //15
                    register_R[4] = 8'b00011111; //31
                    register_R[5] = 8'b00111111; //63

                */

                    DATAmem[0]  =  16'b0000_0001_0111_0000;//add rd 1 rn 3 rm 4
then result is " 46" initially r3=15 r4=31 r5=63
                    DATAmem[4]  =  16'b0001_0010_1010_1100;//sub  r2=r5-r3 "48"
                    DATAmem[8]  =  16'b0010_0001_0111_0000;//and  r1=r4&r3 "15"
                    DATAmem[12] =  16'b0010_1010_0111_0100;//orr  r2=r3|r5  "63"
                    DATAmem[16] =  16'b0011_0001_0110_0000;//xor  r1=r3^r0  "15"
because r0 initially zero
```

```
                    DATAmem[20] =  16'b0011_1010_0000_0000;//clr r2 loaded with 0

                    //shift operations shift rn and store it in rd
                    DATAmem[24] =  16'b0100_0001_0110_0000;//rol r1=rol r3
                    DATAmem[28] =  16'b0100_1010_1000_0000;//ror r2= ror r4
                    DATAmem[32] =  16'b0101_0001_1010_0000;//lsl r1= r5*2 126
                    DATAmem[36] =  16'b0101_1010_1000_0000;//asr r2=asr r4
                    DATAmem[40] =  16'b0110_0001_1010_0000;//lsr r1= r5/2 31

                    //memory instructions rd=r2
                    DATAmem[44] =  16'b1000_0010_0110_0101; //ldr r2,[r3,5]
                    DATAmem[48] =  16'b1001_0010_1111_1111;//ldi r2 255
                    DATAmem[52] =  16'b1010_0010_0110_0101; // str r2,[r3,5]
                    //rd=1
                    DATAmem[56] =  16'b1000_0001_0110_0101; // ldr r1,[r3,5]
33+5=38

                    //branch instructions
                    DATAmem[60] =  16'b1100_0000_0000_1000; //b pc+8+imm8(8)=76

                    //B TO #76 branch here "B 76"
                    DATAmem[76] =  16'b1001_0010_0100_1100;//ldi r2 76
                    DATAmem[80] =  16'b1100_1000_0001_0000; //bl branch with link
to the 104

                    //BL TO THE 104 "BL 104"
                    DATAmem[104]    =  16'b1001_0010_0110_1000;//ldi r2 104
                    DATAmem[108]    =  16'b1100_0000_0101_1000; //b
pc+8+imm8(64)=204

                    //with branch at the 108 jump here
                    DATAmem[204]    =  16'b1001_0010_1101_1000;//ldi r2 216

                    //verfy branch indirect
                    //bi r2
                    DATAmem[208]    =  16'b1101_0000_0000_1000; //bi r2

                    DATAmem[216]    =  16'b1001_0010_1111_1111;//jump to here
after bi instruction
                //end of the instruction

                    DATAmem[220]    =  16'b0000_0000_0000_0000;
                    DATAmem[224]    =  16'b0000_0000_0000_0000;
                    DATAmem[228]    =  16'b0000_0000_0000_0000;
                    DATAmem[232]    =  16'b0000_0000_0000_0000;
                    DATAmem[236]    =  16'b0000_0000_0000_0000;
                    DATAmem[240]    =  16'b0000_0000_0000_0000;
                    DATAmem[244]    =  16'b0000_0000_0000_0000;
                    DATAmem[248]    =  16'b0000_0000_0000_0000;
                    DATAmem[252]    =  16'b0000_0000_0000_0000;



//data memory initialization
//data initialization
//instructions at the 0 4 8 12 ... 252 therefore others can be assigned randomly
           for(i=0;i<256;i=i+1)
              begin
                    if(i%3'd4!=0)
```

```
                        begin
                            DATAmem[i] = i;
                        end
              end

          end
            /*
        initial begin
        $readmemh("memory.txt",DATAmem,0,255);
        end
                    */


        //memory write operation
      always @(posedge clk) begin
           if (memWE)
              begin
               DATAmem[memA] = {8'b0,memWD};//extended value is stored the memory
              end
              else
              begin
              //nothing
              end
      end

      assign memRD = DATAmem[memA]; //it provides the data which is specified by
the address

 endmodule



/*
Implement a W-bit 2 to 1 and a W-bit 4 to 1 multiplexers, where W is a parameter
specifying the
data width of the input.
*/
module muxWFourToOne #(parameter W=1)(s0,s1,I0,I1,I2,I3,out);

input s0;
input s1;
input [W-1:0] I0;
input [W-1:0] I1;
input [W-1:0] I2;
input [W-1:0] I3;

output reg [W-1:0] out;
wire [1:0] sel ={s1,s0};



always @(s0 or s1 or I0 or I1 or I2 or I3)
    begin
         case (sel)
         2'b00 : out = I0;
         2'b01 : out = I1;
         2'b10 : out = I2;
         2'b11 : out = I3;
         endcase
    end

endmodule



/*
Implement a W-bit 2 to 1 and a W-bit 4 to 1 multiplexers, where W is a parameter
specifying the
data width of the input.
*/
module muxWTwoToOne #(parameter W=1)(s0,I1,I0,out);

input [W-1:0] I1;
input [W-1:0] I0;
input s0;
output [W-1:0] out;

assign out=s0 ? I1 : I0;

endmodule



module RegisterFileMC
 (   //input ports
      input                        clk,
      input                        rst,
      input                        WE,  //write enable signal
      input          [2:0]         A1 ,
        input          [2:0]         A2,
        input          [2:0]         A3,
      input          [7:0]        WD3,
      //output ports
      output         [7:0]        RD1,
      output         [7:0]        RD2,

        //demostration purposes
        output         [7:0]        R1,
      output         [7:0]        R2,

        input            [7:0]      R6
 );


        reg                 [7:0]    register_R [7:0]; //8 bits width and 8 bits
length

      always @ (posedge clk or posedge rst) begin

           if(rst) begin
                  //general purpose registers
               register_R[0] = 8'b0;
                    register_R[1] = 8'b0;
                    register_R[2] = 8'b0;
                    register_R[3] = 8'b00001111; //15
                    register_R[4] = 8'b00011111; //31
                    register_R[5] = 8'b00111111; //63
                    //link register
                    register_R[7] = 8'b0;

                    //pc represent the r6

           end

           else
              begin
                if(WE)
                     begin
                     register_R[A3] <= WD3;  //if WE is equal to 1 then
corresponding register is written
                end
           end

      end
      assign RD1 = (A1==3'b110) ? R6:register_R[A1];  //read data 1
      assign RD2 = register_R[A2];  //read data 2
        assign R1 = register_R[1];
      assign R2 = register_R[2];
 endmodule

 module shrinkImm
(

input    [15:0] Instr150,
//output port
output    [7:0]  instr70
);

assign instr70=Instr150[7:0];


endmodule


//it creates the constant value
module ConstantValueGenerator #(parameter DATA_WIDTH = 1,parameter BUS_DATA = 0) (
//port declerations
output wire [DATA_WIDTH-1:0] Data_on_Bus
);
//assign DATA_BUS to the bus
assign Data_on_Bus [DATA_WIDTH-1:0]=BUS_DATA;

endmodule

module simpleREG #(parameter W=1) (DATA,clk,rst,SREGout);
input rst,clk;
input [W-1:0] DATA;
output reg [W-1:0] SREGout;


always @(posedge clk)
begin
    if(rst==1)
        begin
            SREGout<=0;
        end
    else
        begin

            SREGout<=DATA;
        end

end

endmodule


module writeEnableREG #(parameter W=1) (DATA,clk,rst,WREGout,WE);
input rst,clk,WE;
input [W-1:0] DATA;
output reg [W-1:0] WREGout;


always @(posedge clk)//due to sync reset issue
begin
    if(rst==1)
        begin
            WREGout<=0;
        end
    else
        begin

            if(WE==1)
                begin
                    WREGout<=DATA;
                end

        end

end

endmodule
```

```
//////////////Textbench Multi Cycle Code///////////////
/*

input wire  clk;
input wire  rst;
input wire  PCWrite;
input wire  MemWrite;
input wire  IRWrite;
input wire  ImmSrc;
input wire  RegWrite;
input wire  ALUSrcA;
input wire  [1:0] AdrSrc;
input wire  [3:0] ALUControl;
input wire  [1:0] ALUSrcB;
input wire  [2:0] RegSrc;
input wire  [1:0] ResultSrc;

*/
module testbenchMC();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg clk;
reg rst;
reg PCWrite;
reg MemWrite;
reg IRWrite;
reg ImmSrc;
reg RegWrite;
reg ALUSrcA;
reg [1:0] AdrSrc;
reg   [3:0] ALUControl;
reg [1:0] ALUSrcB;
reg [2:0] RegSrc;
reg [1:0] ResultSrc;


//outputs
 wire   [3:0] ALU_flags;
 wire   [7:0] R1out;
 wire   [7:0] R2out;





// instantiate device under test
multiCycleDatapath_code DUT(
    clk,
    rst,
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
    ALUSrcB,
    RegSrc,
    ResultSrc,
    ALU_flags,
    R1out,
    R2out
);


initial
begin
rst=1;
#100;
rst=0;
PCWrite=0;
MemWrite=0;
IRWrite=0;
ImmSrc=1;
RegWrite=0;
ALUSrcA=1;
AdrSrc=2'b11;
ALUControl=4'b0000;
ALUSrcB=2'b11;
RegSrc=3'b000;
ResultSrc=2'b00;
end

// generate clock
always // no sensitivity list, so it always executes
    begin
    clk = 0; #50; clk = 1; #50;
    end

//change the input signals according to the instructions
initial // no sensitivity list, so it always executes
    begin
//ADD instruction
//fetch
#100;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```
//execute
#100;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//SUB instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0001;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//AND instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
```

```
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0010;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ORR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
```

```
#100;
ALUSrcA=0;
ALUControl=4'b0011;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//XOR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0100;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//CLR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
```

```
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0101;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ROL instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
```

```
ALUSrcA=0;
ALUControl=4'b0110;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ROR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b0111;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//LSL instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
```

```
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b1000;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//ASR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
```

```
ALUControl=4'b1010;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;


//LSR instruction
//fetch
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//execute
#100;
ALUSrcA=0;
ALUControl=4'b1001;
ALUSrcB=2'b00;
RegSrc=3'b100;
ResultSrc=2'b10;
RegWrite=0;

//alu write back
#100;
ResultSrc=2'b00;
RegWrite=1;
//shift and data processing operations are completed


//LDR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
```

```
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memREAD Cycle 4
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=0;

//memWriteBack Cycle 5
#100;
ResultSrc=2'b01;
RegWrite=1;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```
//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;


//STR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b110;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memWrite
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=1;


//LDR instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;
```

```
//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//memADR Cycle 3
#100;
ImmSrc=0;
ALUSrcB=2'b01;
ALUSrcA=0;
ALUControl=4'b0000;

//memREAD Cycle 4
#100;
ResultSrc=2'b00;
AdrSrc=2'b01;
MemWrite=0;

//memWriteBack Cycle 5
#100;
ResultSrc=2'b01;
RegWrite=1;


//LDR and STR instructions are done
//after that branch instructions


//B instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b000;
```

```
ResultSrc=2'b10;


//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;



//BL instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
```

```
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b001;
ResultSrc=2'b10;

//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;
RegWrite=1;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;


//B instruction
//fetch-Cycle 1
#100;
RegWrite=0;
MemWrite=0;
PCWrite=1;
```

```
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b000;
ResultSrc=2'b10;


//branch cycle 3
#100;
PCWrite=1;
ALUSrcA=0;
ALUControl=4'b0000;
ALUSrcB=2'b01;
ResultSrc=2'b10;


//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;

//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;
```

```
//BI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b101;
ResultSrc=2'b10;

//bi pc write
#100;
ALUSrcB=2'b00;
ResultSrc=2'b11;
PCWrite=1;



//LDI instruction
//fetch-Cycle 1
#100;
RegWrite=0;
PCWrite=1;
IRWrite=1;
ALUSrcA=1;
AdrSrc=2'b00;
ALUControl=4'b0000;
ALUSrcB=2'b10;
ResultSrc=2'b10;

//decode Cycle 2
#100;
PCWrite=0;
MemWrite=0;
IRWrite=0;
RegWrite=0;
ALUSrcA=1;
ALUControl=4'b0000;
ALUSrcB=2'b10;
RegSrc=3'b100;
ResultSrc=2'b10;
```

```verilog
//ALU write back for LDI cycle 3
#100;
ImmSrc=1;
ALUSrcB=2'b01;
ResultSrc=2'b11;
RegWrite=1;

end

endmodule
```