```
    /// UNIFIED DATAPATH and CONTROLLER DESIGN CODE ///
module datapath_controller_MC(
input clk,
input RESET,
input RUN,
output wire [7:0] R1reg,
output wire [7:0] R2reg
);

wire [1:0]  w_cond;
wire [1:0]  w_OP;
wire [2:0]  w_type;
wire           w_PCWrite;
wire [1:0]  w_AdrSrc;
wire           w_MemWrite;
wire           w_IRWrite;
wire [2:0]  w_RegSrc;
wire           w_RegWrite;
wire           w_ImmSrc;
wire           w_ALUSrcA;
wire [1:0]  w_ALUSrcB;
wire [3:0]  w_ALUControl;
wire [1:0]  w_ResultSrc;
wire [3:0]  w_flags;
wire [2:0]  w_Rd;

MultiCycle_Controller my_MC_controller(
//inputs
.cond(w_cond),
.OP(w_OP),
.type(w_type),
.flags(w_flags),
.Rd(w_Rd),
.RUN(RUN),
.clk(clk),
.PCWrite(w_PCWrite),
.AdrSrc(w_AdrSrc),
.MemWrite(w_MemWrite),
.IRWrite(w_IRWrite),
.RegSrc(w_RegSrc),
.RegWrite(w_RegWrite),
.ImmSrc(w_ImmSrc),
.ALUSrcA(w_ALUSrcA),
.ALUSrcB(w_ALUSrcB),
.ALUControl(w_ALUControl),
.ResultSrc(w_ResultSrc)
);


 multi_cycle_datapath my_MC_datapath(
    .clk(clk),
    .rst(RESET),
    .PCWrite(w_PCWrite),
    .MemWrite(w_MemWrite),
    .IRWrite(w_IRWrite),
    .ImmSrc(w_ImmSrc),
    .RegWrite(w_RegWrite),
    .ALUSrcA(w_ALUSrcA),
    .AdrSrc(w_AdrSrc),
```

```
    .ALUControl(w_ALUControl),
    .ALUSrcB(w_ALUSrcB),
    .RegSrc(w_RegSrc),
    .ResultSrc(w_ResultSrc),
    .R1out(R1reg),
    .R2out(R2reg),
    .ALU_flags(w_flags),
    .cond(w_cond),
    .OP(w_OP),
    .type(w_type),
    .Rd(w_Rd)

);

endmodule


                        ///DATAPATH CODE////
module multi_cycle_datapath(
    clk,
    rst,
    //control signal inputs
    PCWrite,
    MemWrite,
    IRWrite,
    ImmSrc,
    RegWrite,
    ALUSrcA,
    AdrSrc,
    ALUControl,
    ALUSrcB,
    RegSrc,
    ResultSrc,

    //desmostration purpose registers
    R1out,
    R2out,

    //controller inputs which is output for the datapath
    ALU_flags,
    cond,
    OP,
    type,
    Rd

);


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 [7:0] R1out;
output wire [7:0] R2out;

output wire [1:0]  cond;
output wire [1:0]  OP;
output wire [2:0]  type;
output wire [3:0]  ALU_flags;
output wire [2:0]  Rd;



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),
    .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));

    //assignment for the required controller inputs
assign  ALU_flags = X;
assign cond = WREGout[1:0];
assign OP = WREGout[15:14];
assign type = WREGout[13:11];
assign  Rd=WREGout[10:8];
```

```
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

    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
```

//CONTROLLER DESIGN CODE//

```
module MultiCycle_Controller(
//inputs
input [1:0]  cond,
input [1:0]  OP,
input [2:0]  type,
input [3:0]  flags,
input [2:0]  Rd,
input        RUN,
input        clk,
//outputs
output reg       PCWrite,
output reg [1:0] AdrSrc,
output reg       MemWrite,
output reg       IRWrite,
output reg [2:0] RegSrc,
output reg       RegWrite,
output reg       ImmSrc,
output reg       ALUSrcA,
output reg [1:0] ALUSrcB,
output reg [3:0] ALUControl,
output reg [1:0] ResultSrc
);

reg [2:0] state_counter=3'b000;
reg [3:0] FLAG_REG;

always @(posedge clk)
            begin

                if(RUN==1)
                     begin
                        //fetch
```

```
                    if(state_counter==3'b000)
                        begin
                            PCWrite=1;
                            IRWrite=1;
                            ALUSrcA=1;
                            AdrSrc=2'b00;
                            RegWrite=0;
                            ALUControl=4'b0000;
                            ALUSrcB=2'b10;
                            ResultSrc=2'b10;
                            state_counter=state_counter+3'b001;
                        end

                    //decode
                    else if(state_counter==3'b001)
                        begin
                            PCWrite=0;
                            MemWrite=0;
                            IRWrite=0;
                            RegWrite=0;
                            ALUSrcA=1;
                            ALUControl=4'b0000;
                            ALUSrcB=2'b10;
                            //regSrc assignment
                            if(OP==2'b11) //branch
                                begin
                                    if(type==3'b000)//b
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b001)//bl
                                    begin
                                        RegSrc=3'b001;
                                    end

                                    else if(type==3'b011)//beq
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b100)//bne
                                    begin
                                        RegSrc=3'b000;
                                    end

                                    else if(type==3'b101)//bc
                                    begin
                                        RegSrc=3'b000;
                                    end
                                    else if(type==3'b110)//bnc
                                    begin
                                        RegSrc=3'b000;
                                    end
                                    else if(type==3'b010)//bi
                                    begin
                                        RegSrc=3'b101;
                                    end
                                end
```

```
                        else if(OP==2'b00) //data
                            begin
                                RegSrc=3'b100;
                            end
                        else if(OP==2'b01) //shift
                            begin
                                RegSrc=3'b100;
                            end

                        else if(OP==2'b10) //memory
                            begin
                            if(type[2:1]==2'b10)//str
                                begin
                                RegSrc=3'b110;
                                end
                            else // ldr/i
                                begin
                                RegSrc=3'b100;
                                end
                            end


                        state_counter=state_counter+3'b001;
                    end

                    //execution phase
                else if(state_counter==3'b010) //cycle 3
                    begin

                    if(OP==2'b00)//data
                        begin
                            ALUSrcA=0;

                            case(type)
                                    3'b000: ALUControl=4'b0000; //add
                                    3'b010: ALUControl=4'b0001; //sub
                                    3'b100: ALUControl=4'b0010; //and
                                    3'b101: ALUControl=4'b0011; //orr
                                    3'b110: ALUControl=4'b0100; //xor
                                    3'b111: ALUControl=4'b0101; //clr
                            endcase
                            ALUSrcB=2'b00;
                            RegSrc=3'b100;
                            ResultSrc=2'b10;
                            RegWrite=0;
                            FLAG_REG=flags; //update the flags for data
processing

                            state_counter=state_counter+3'b001;
                        end
                    else if(OP==2'b01)//shift
                        begin
                            ALUSrcA=0;

                            case(type)
                                    3'b000: ALUControl=4'b0110; //rol
                                    3'b001: ALUControl=4'b0111; //ror
                                    3'b010: ALUControl=4'b1000; //lsl
                                    3'b011: ALUControl=4'b1010; //asr
```

```
                                        3'b100: ALUControl=4'b1001; //lsr
                                    endcase
                                    ALUSrcB=2'b00;
                                    RegSrc=3'b100;
                                    ResultSrc=2'b10;
                                    RegWrite=0;
                                    FLAG_REG=flags;
                                    state_counter=state_counter+3'b001;
                                end
                            else if(OP==2'b10)//memory
                                begin

                                    case(type[2:1])
                                    2'b00://ldr cycle 3
                                        begin
                                            ImmSrc=0;
                                            ALUSrcB=2'b01;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;

                                            state_counter=state_counter+3'b001;
                                        end
                                    2'b01://ldi last cycle
                                        begin
                                            ImmSrc=1;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b11;
                                            RegWrite=1;

                                            state_counter=3'b000;//go to next
cycle

                                        end
                                    2'b10://str cycle 3
                                        begin
                                            ImmSrc=0;
                                            ALUSrcB=2'b01;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;

                                            state_counter=state_counter+3'b001;
                                        end
                                    endcase
                                end

                            else if(OP==2'b11)//branch cycle 3 last cycle
                                    begin
                                        case(type)
                                        3'b000: //und. branch
                                        begin
                                            PCWrite=1;
                                            ALUSrcA=0;
                                            ALUControl=4'b0000;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b10;
                                            state_counter=3'b000;
                                        end

                                            3'b001: // branch link
                                            begin
```

```
                                    PCWrite=1;
                                    ALUSrcA=0;
                                    ALUControl=4'b0000;
                                    ALUSrcB=2'b01;
                                    ResultSrc=2'b10;
                                    RegWrite=1;
                                    state_counter=3'b000;

                                end

                                3'b011: //beq
                                begin
                                //branch cycle 3 equal means Z=1 nzcv
                                //look at the FLAG_REG register value to
decide execue or not
                                if(FLAG_REG[2]==1)
                                begin
                                    PCWrite=1;
                                end
                                else
                                begin
                                PCWrite=0;
                                end

                                ALUSrcA=0;
                                ALUControl=4'b0000;
                                ALUSrcB=2'b01;
                                ResultSrc=2'b10;
                                state_counter=3'b000;

                                end

                                3'b100: //bne
                                //branch cycle 3 not equal means Z=0 nzcv
                                begin
                                if(FLAG_REG[2]==0)
                                begin
                                    PCWrite=1;
                                end
                                else
                                begin
                                PCWrite=0;
                                end

                                ALUSrcA=0;
                                ALUControl=4'b0000;
                                ALUSrcB=2'b01;
                                ResultSrc=2'b10;
                                state_counter=3'b000;

                                end

                                3'b101: //bc
                                begin
                                //branch cycle 3 carry set means c=1 nzcv
                                if(FLAG_REG[1]==1)
                                        begin
                                            PCWrite=1;
                                        end
```

```
                                        else
                                                begin
                                                    PCWrite=0;
                                                end

                                        ALUSrcA=0;
                                        ALUControl=4'b0000;
                                        ALUSrcB=2'b01;
                                        ResultSrc=2'b10;
                                        state_counter=3'b000;
                                        end

                                        3'b110: //bnc
                                        begin
                                        //branch cycle 3 not carry set means c=0
nzcv
                                        if(FLAG_REG[1]==0)
                                                begin
                                                    PCWrite=1;
                                                end
                                        else
                                                begin
                                                    PCWrite=0;
                                                end

                                            ALUSrcA=0;
                                            ALUControl=4'b0000;
                                            ALUSrcB=2'b01;
                                            ResultSrc=2'b10;
                                            state_counter=3'b000;

                                        end

                                        3'b010: //bi
                                        begin
                                            ALUSrcB=2'b00;
                                            ResultSrc=2'b11;
                                            PCWrite=1;
                                            state_counter=3'b000;
                                        end

                                        3'b111: //end
                                        begin
                                            //RUN=0; //end of the instructions
                                            state_counter=3'b000;
                                        end
                                        endcase
                                    end

                                end//cycle 3 ends

                                //execution phase
                                else if(state_counter==3'b011) //cycle 4
                                begin

                                    if(OP[1]==1'b0)
                                    begin
                                        ResultSrc=2'b00;
                                        RegWrite=1;
```

```
                                    state_counter=3'b000;//data and shift end
                                end
                                else if(OP==2'b10)
                                begin
                                    if(type[2:1]==2'b10)//str
                                            begin
                                            ResultSrc=2'b00;
                                            AdrSrc=2'b01;
                                            MemWrite=1;
                                            state_counter=3'b000; //end store
                                            end
                                    else if(type[2:1]==2'b00) //ldr
                                            begin
                                                ResultSrc=2'b00;
                                                AdrSrc=2'b01;
                                                MemWrite=0;

state_counter=state_counter+3'b001;//cycle 4 for the ldr
                                            end

                                end
                            end

                            else if(state_counter==3'b100) //cycle 5
                            begin
                            ResultSrc=2'b01;
                            RegWrite=1;
                            state_counter=3'b000; //end ldr
                            end



                    end//run


    end//always


endmodule
```

```
                ///TESTBENCH OF THE CONTROL UNIT///

module testbench_MC_Controller();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg [1:0]  cond;
reg [1:0]  OP;
reg [2:0]  type;
reg [3:0]  flags;
reg          RUN;
reg          clk;


//outputs
wire         PCWrite;
wire [1:0] AdrSrc;
wire         MemWrite;
wire         IRWrite;
wire [2:0] RegSrc;
wire         RegWrite;
wire         ImmSrc;
wire         ALUSrcA;
wire [1:0] ALUSrcB;
wire [3:0] ALUControl;
wire [1:0] ResultSrc;
// instantiate device under test
MultiCycle_Controller DUT(
//inputs
cond,
OP,
type,
flags,
RUN,
clk,

//outputs
PCWrite,
AdrSrc,
MemWrite,
IRWrite,
RegSrc,
RegWrite,
ImmSrc,
ALUSrcA,
ALUSrcB,
ALUControl,
ResultSrc
);



initial
begin

RUN=1;
//inputs
cond=2'b00;
```

```
OP=2'b00;
type=3'b000;
flags=4'b0000;

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
always // no sensitivity list, so it always executes
    begin
//fetch
OP=2'b00;
type=3'b000;

#400; //sub
flags=4'b1011;//////flag update
OP=2'b00;
type=3'b010;

#400; //and

OP=2'b00;
type=3'b100;

#400; //orr

OP=2'b00;
type=3'b101;

#400; //xor

OP=2'b00;
type=3'b110;

#400; //clr

OP=2'b00;
type=3'b111;

#400; //rol

OP=2'b01;
type=3'b000;

#400; //ror

OP=2'b01;
type=3'b001;

#400; //lsl

OP=2'b01;
type=3'b010;
```

```
#400; //asr

OP=2'b01;
type=3'b011;

#400; //lsr

OP=2'b01;
type=3'b100;

#400; //ldi

OP=2'b10;
type[2:1]=2'b01;

#300; //ldr

OP=2'b10;
type[2:1]=2'b00;

#500; //str

OP=2'b10;
type[2:1]=2'b10;

#400; //branch und.

OP=2'b11;
type=3'b000;

#300; //branch bl

OP=2'b11;
type=3'b001;

#300; //branch ind.

OP=2'b11;
type=3'b010;

#300; //branch eq

//nzcv
flags=4'b0100; //z=1 means equal case
OP=2'b11;
type=3'b011;

#300; //branch not eq

//nzcv
flags=4'b1011; //z=0 means not equal case
OP=2'b11;
type=3'b100;
#300;

//bc
flags=4'b1010; //c=1 means carry set case
OP=2'b11;
type=3'b101;
```

```
#300;

//bnc
flags=4'b1000; //c=0 means not cary set case
OP=2'b11;
type=3'b110;
#300;


//END inst
flags=4'b1000;
OP=2'b11;
type=3'b111;
#300;
end

endmodule
```

//THE TESTBENCH OF THE UNIFIED CODE//

```
module testbench_MC_Data_Controller();
//inputs are reg
//outputs are wire
//assume that bus width is 3 to test the result
//inputs
reg clk;
reg RESET;
reg RUN;
wire [7:0] R1reg;
wire [7:0] R2reg;
// instantiate device under test
datapath_controller_MC DUT(
clk,
RESET,
RUN,
R1reg,
R2reg
);

initial
begin
RESET=1;
#100;
RESET=0;
RUN=1;
end

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