```verilog
`timescale 1ns / 1ps
// NPC control signal
`define NPC_PLUS4   3'b000
`define NPC_BRANCH  3'b001
`define NPC_JUMP    3'b010
`define NPC_JALR 3'b100

// ALU control signal
`define ALU_NOP   3'b000
`define ALU_ADD   3'b001
`define ALU_SUB   3'b010
`define ALU_AND   3'b011
`define ALU_OR    3'b100

`define ALUOp_nop 5'b00000
`define ALUOp_lui 5'b00001
`define ALUOp_auipc 5'b00010
`define ALUOp_add 5'b00011
`define ALUOp_sub 5'b00100
`define ALUOp_bne 5'b00101
`define ALUOp_blt 5'b00110
`define ALUOp_bge 5'b00111
`define ALUOp_bltu 5'b01000
`define ALUOp_bgeu 5'b01001
`define ALUOp_slt 5'b01010
`define ALUOp_sltu 5'b01011
`define ALUOp_xor 5'b01100
`define ALUOp_or 5'b01101
`define ALUOp_and 5'b01110
`define ALUOp_sll 5'b01111
`define ALUOp_srl 5'b10000
`define ALUOp_sra 5'b10001

//EXT CTRL itype, stype, btype, utype, jtype
`define EXT_CTRL_ITYPE_SHAMT 6'b100000
`define EXT_CTRL_ITYPE       6'b010000
`define EXT_CTRL_STYPE       6'b001000
`define EXT_CTRL_BTYPE       6'b000100
`define EXT_CTRL_UTYPE       6'b000010
`define EXT_CTRL_JTYPE       6'b000001

`define GPRSel_RD 2'b00
`define GPRSel_RT 2'b01
`define GPRSel_31 2'b10

`define WDSel_FromALU 2'b00
`define WDSel_FromMEM 2'b01
`define WDSel_FromPC 2'b10




`define dm_word 3'b000
`define dm_halfword 3'b001
`define dm_halfword_unsigned 3'b010
```

```verilog
`define dm_byte 3'b011
`define dm_byte_unsigned 3'b100



module test(
        input clk,
        input rstn,
        input [15:0] sw_i,
        output [7:0] disp_an_o,
        output [7:0] disp_seg_o
    );

    SCPU_TOP scpu(
        .clk(clk),
        .rstn(rstn),
        .sw_i(sw_i),
        .disp_an_o(disp_an_o),
        .disp_seg_o(disp_seg_o)
        );
endmodule

module clk_16divider(
    input clk,
    output wire clk_16div
);
    reg [8:0] counter;

    always @(posedge clk) begin
        counter <= counter + 1;
    end

    assign clk_16div = counter[8];

endmodule

module an_selector(
    input clk,
    input rstn,
    input [15:0]sw_i,
    input [63:0]i_data,
    output reg [7:0] an,
    output reg [7:0] seg
);
    reg [2: 0] selector;

    always @(posedge clk) begin
        selector <= selector + 1;
    end

    reg [63:0] i_data_store;
    always@(posedge clk,negedge rstn)
    begin
        if(!rstn)
            i_data_store<=0;
        else
            i_data_store<=i_data;
    end
```

```
reg [7:0]seg_data_r;

always@(*)begin
if(sw_i[0]==0)begin //文本模式
    case(selector)
       0:seg_data_r=i_data_store[3:0];
       1:seg_data_r=i_data_store[7:4];
       2:seg_data_r=i_data_store[11:8];
       3:seg_data_r=i_data_store[15:12];
       4:seg_data_r=i_data_store[19:16];
       5:seg_data_r=i_data_store[23:20];
       6:seg_data_r=i_data_store[27:24];
       7:seg_data_r=i_data_store[31:28];
    endcase end
else if(sw_i[0]==1)begin //图形模式
    case(selector)
       0:seg_data_r=i_data_store[7:0];
       1:seg_data_r=i_data_store[15:8];
       2:seg_data_r=i_data_store[23:16];
       3:seg_data_r=i_data_store[31:24];
       4:seg_data_r=i_data_store[39:32];
       5:seg_data_r=i_data_store[47:40];
       6:seg_data_r=i_data_store[55:48];
       7:seg_data_r=i_data_store[63:56];
    endcase end
end

function [7:0] seven_seg_translator;
    input [7:0] seg_data;
    input sw;
    input r;
    if(!r)begin
        seven_seg_translator= 8'hff;
        end
    else if(sw==0)begin
    case (seg_data) // 共阳极数码管
        4'h0: seven_seg_translator = 8'hc0;
        4'h1: seven_seg_translator = 8'hf9;
        4'h2: seven_seg_translator = 8'ha4;
        4'h3: seven_seg_translator = 8'hb0;
        4'h4: seven_seg_translator = 8'h99;
        4'h5: seven_seg_translator = 8'h92;
        4'h6: seven_seg_translator = 8'h82;
        4'h7: seven_seg_translator = 8'hf8;
        4'h8: seven_seg_translator = 8'h80;
        4'h9: seven_seg_translator = 8'h90;
        4'hA: seven_seg_translator = 8'h88;
        4'hB: seven_seg_translator = 8'h83;
        4'hC: seven_seg_translator = 8'hC6;
        4'hD: seven_seg_translator = 8'hA1;
        4'hE: seven_seg_translator = 8'h86;
        4'hF: seven_seg_translator = 8'h8E;
        default: seven_seg_translator = 8'hff;
    endcase end
    else begin seven_seg_translator =seg_data; end
endfunction
```

```
always @(selector) begin
    case (selector)
        3'b000: begin
            an <= 8'b1111_1110;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b001: begin
            an <= 8'b1111_1101;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b010: begin
            an <= 8'b1111_1011;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b011: begin
            an <= 8'b1111_0111;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b100: begin
            an <= 8'b1110_1111;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b101: begin
            an <= 8'b1101_1111;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b110: begin
            an <= 8'b1011_1111;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        3'b111: begin
            an <= 8'b0111_1111;
            seg <= seven_seg_translator(seg_data_r,sw_i[0],rstn);
        end
        default: begin
            an <= 8'b1111_1111;
            seg <= 8'hff;
        end
    endcase
end

endmodule

module SCPU_TOP(
input clk,  //100MHZ CLK
input rstn,  //reset signal
input [15:0] sw_i, //sw_i[15]---sw_i[0]
output wire [7:0] disp_an_o, //8位数码管位选
output wire[7:0] disp_seg_o //数码管8段数据
);
    //clk module
    reg[31:0]clkdiv;
    wire clk_cpu,clk_dis,clk_cpu_on;

    always@(posedge clk or negedge rstn)begin
        if(!rstn)   clkdiv<=1'b0;
        else clkdiv<=clkdiv+1'b1;
    end
    assign clk_cpu_on=(sw_i[15])? clkdiv[27]:clkdiv[25];
    assign clk_cpu=(sw_i[1])? 0:clk_cpu_on;
    assign clk_dis=(sw_i[15])? clkdiv[27]:clkdiv[25];

    //control module
    wire [31:0]inst_in;
    wire [6:0] Op = inst_in[6:0];  // op
    wire [6:0] Funct7 = inst_in[31:25]; // funct7
    wire [2:0] Funct3 = inst_in[14:12]; // funct3
    wire [4:0] rs1 = inst_in[19:15];  // rs1
    wire [4:0] rs2 = inst_in[24:20];  // rs2
    wire [4:0] rd = inst_in[11:7];  // rd
    wire [11:0] iimm = inst_in[31:20];//addi 指令立即数，lw指令立即数
    wire [11:0] simm = {inst_in[31:25],inst_in[11:7]}; //sw指令立即数
    wire [11:0] bimm=  {inst_in[31],inst_in[7], inst_in[30:25], inst_in[11:8]};
    wire [19:0] u_imm = {inst_in[31:12]};
    wire [19:0] j_imm = {inst_in[31], inst_in[19:12], inst_in[20],
inst_in[30:21]};
    wire Zero,RegWrite,MemWrite,ALUSrc,GE;
    wire [1:0] WDSel;
    wire [5:0] EXTOp;
    wire [2:0] DMType,NPCOp;
    wire [4:0] ALUOp;

    ctrl u_ctrl(
    .Op(Op),  //opcode
    .Funct7(Funct7),  //funct7
    .Funct3(Funct3),    // funct3
    .Zero(Zero),
    .GE(GE),
    .RegWrite(RegWrite), // control signal for register write
    .MemWrite(MemWrite), // control signal for memory write
    .EXTOp(EXTOp),    // control signal to signed extension
    .ALUOp(ALUOp),    // ALU opertion
    .NPCOp(NPCOp),    // next pc operation
    .ALUSrc(ALUSrc),    // ALU source for b
    .DMType(DMType), //dm r/w type
    .WDSel(WDSel)    // (register) write data selection  (MemtoReg)
);

    //PC module
    wire [31:0]     immout;
    wire  signed [31:0] aluout;
    wire [31:0] PC,NPC;

    PC u_pc(
    .clk(clk_cpu),
    .rstn(rstn),
    .NPC(NPC),
    .PCwr(~sw_i[1]),
    .sw_i(sw_i[5:2]),
    .PC(PC));

    NPC u_npc(
    .PC(PC),
    .NPCOp(NPCOp),
    .IMM(immout),
    .aluout(aluout),
    .NPC(NPC));

    //ROM module
    wire[31:0] ROM_data;
    wire[11:0]rom_addr;

    assign rom_addr=PC/4;
    assign inst_in=(sw_i[1])? 0:ROM_data;

    dist_mem_im U_IM(
        .a(rom_addr),
        .spo(ROM_data)
        );

    //RF module
    wire [31:0]RD1,RD2;
    reg [31:0] WD;
    //display
    reg[5:0]rf_addr;
    parameter REG_NUM = 32;

    wire [4:0] rf_rs1;
    assign rf_rs1=(sw_i[1])? rf_addr:rs1;

    RF u_rf(
    .clk(clk_cpu),  //分频后的主时钟 CLK
    .rstn(rstn),    //reset signal
    .RFWr(RegWrite),//Rfwrite = mem2reg
    .sw_i(sw_i),    //sw_i[15]---sw_i[0]
    .A1(rf_rs1),
    .A2(rs2),
    .A3(rd),        // Register Num
    .WD(WD),        //Write data
    .RD1(RD1),
    .RD2(RD2)       //Data output port
);

    //ALU module
    wire  signed [31:0] A,B;
    assign A=RD1;
    //DISPLAY
    reg [2:0]alu_addr;
    parameter ALU_NUM = 4;

    ALU u_alu(
        .A(A),
        .B(B),
        .ALUOp(ALUOp),
        .C(aluout),
        .Zero(Zero),
        .GE(GE)
    );

    //DM module
    wire[5:0]   ram_addr;
    wire[31:0]  din;
    wire[31:0]  dout;
    assign din=RD2;
    //display
    reg [4:0]dm_addr;
    parameter DM_NUM=16;
    assign ram_addr=(sw_i[1])?dm_addr:aluout;

    wire [2:0]DMType_;
    assign DMType_ =(sw_i[1])? 3'b011:DMType;

    DM u_dm(
    .clk(clk_cpu),  //100MHZ CLK
    .rstn(rstn),
    .DMWr(MemWrite),  //write signal
    .addr(ram_addr),
    .din(din),
    .DMType(DMType_),
    .dout(dout)
    );


    //IMM module
    wire[4:0] iimm_shamt;
    wire[19:0]  uimm,jimm;

    assign iimm_shamt = inst_in[24:20];
    assign iimm = inst_in[31:20];
    assign simm = {inst_in[31:25], inst_in[11:7]};
    assign bimm = {inst_in[31], inst_in[7], inst_in[30:25], inst_in[11:8]};
    assign uimm = {inst_in[31:12]};
    assign jimm = {inst_in[31], inst_in[19:12], inst_in[20], inst_in[30:21]};

    EXT u_ext(
    .iimm_shamt(iimm_shamt), //
    .iimm(iimm),  //instr[31:20], 12 bits
    .simm(simm), //instr[31:25, 11:7], 12 bits
    .bimm(bimm),//instrD[31],instrD[7], instrD[30:25], instrD[11:8], 12 bits
    .uimm(uimm),
    .jimm(jimm),
    .EXTOp(EXTOp),
    .immout(immout)
);
    //mul module
    always @* begin
        case(WDSel)
            `WDSel_FromALU:   WD=aluout;
            `WDSel_FromMEM:   WD=dout;
            `WDSel_FromPC:    WD=PC+4;
            default:          WD=WD;
        endcase
    end
    assign B=(ALUSrc)?immout:RD2;

    //display module
    reg [31:0]i_data;
    always@(posedge clk_dis or negedge rstn)begin
        if(!rstn)begin
            rf_addr     <=0;
            alu_addr    <=0;
            dm_addr     <=0;
        end
        else if(sw_i[14]==1 && sw_i[13]==0 && sw_i[12]==0 && sw_i[11]==0)begin
                i_data<=ROM_data;
        end
        else if(sw_i[14]==0 && sw_i[13]==1 && sw_i[12]==0 && sw_i[11]==0)begin
            if(rf_addr==REG_NUM)begin
                rf_addr<=0;
                i_data<=32'hffff_ffff;
            end
            else begin
                i_data={2'b00,rf_addr[5:0],RD1[23:0]};
                rf_addr=rf_addr+1'b1;
            end
        end
        else if(sw_i[14]==0 && sw_i[13]==0 && sw_i[12]==1 && sw_i[11]==0)begin
            if(alu_addr==ALU_NUM)begin
                alu_addr<=0;
                i_data<=32'hffff_ffff;
            end
            else begin
                case(alu_addr)
                    2'b00:      i_data<=A;
                    2'b01:      i_data<=B;
                    2'b10:      i_data<=aluout;
                    2'b11:      i_data<=Zero;
                    default:    i_data<=i_data;
                endcase
                alu_addr=alu_addr+1'b1;
            end
        end
        else if(sw_i[14]==0 && sw_i[13]==0 && sw_i[12]==0 && sw_i[11]==1)begin
            if(dm_addr==DM_NUM)begin
                dm_addr<=0;
                i_data<=32'hffff_ffff;
            end
            else begin
                i_data={dm_addr[3:0],dout[27:0]};
                dm_addr=dm_addr+1'b1;
            end
        end
        else
            i_data<=32'h0000_0000;
    end


    wire clk_16div;
    clk_16divider clk16div(
        .clk(clk),
        .clk_16div(clk_16div)
    );

    an_selector an_st(
        .clk(clk_16div),
        .rstn(rstn),
        .sw_i(sw_i),
        .i_data(i_data),
        .an(disp_an_o),
        .seg(disp_seg_o)
    );
endmodule

module ALU(
input signed [31:0]     A, B,  //alu input num
input [4:0]             ALUOp, //alu how to do
output reg signed [31:0]    C, // alu result
output wire  Zero,
output wire GE
);
    wire [31:0] AU=A;
    wire [31:0] BU=B;
    always@(*)begin
        case(ALUOp)
            `ALUOp_add:C=A+B;
            `ALUOp_sub:C=A-B;
            `ALUOp_blt:C=(A>=B);
            `ALUOp_bge:C=(A>=B);
            `ALUOp_bltu:C=( $unsigned(A) >= $unsigned(B));
            `ALUOp_bgeu:C=( $unsigned(A) >= $unsigned(B));
            `ALUOp_sll:C=A<<($unsigned(B));
            `ALUOp_srl:C=A>>($unsigned(B));
            `ALUOp_sra:C=A>>>($unsigned(B));
            default:C=0;
        endcase
    end

    assign Zero=(C==0);
    assign GE=C;
endmodule

module RF(
input   clk,                            //分频后的主时钟 CLK
input   rstn,                               //reset signal
input   RFWr,                           //Rfwrite = mem2reg
input   [15:0] sw_i,                    //sw_i[15]---sw_i[0]
input   [4:0] A1, A2, A3,       // Register Num
input   [31:0] WD,                      //Write data
output wire [31:0] RD1, RD2 //Data output port
);
    reg [31:0]rfs[31:0];
    integer p;

   assign RD1=rfs[A1];
   assign RD2=rfs[A2];

   always@(posedge clk or negedge rstn)begin
         if(!rstn)begin
         for(p=0;p<32;p=p+1)begin
                rfs[p]<=0;
         end
         end
         else if(A3!=0 && sw_i[1]==0 && RFWr==1) rfs[A3]<=WD;
         else rfs[A3]<=rfs[A3];
    end
endmodule
```

```
module DM(
input   clk,  //100MHZ CLK
input  rstn,
input   DMWr,  //write signal
input [5:0]     addr,
input [31:0]    din,
input [2:0]     DMType,
output reg [31:0]   dout
);
    reg [7:0] dmem[127:0];
    integer i;

    always@(posedge clk or negedge rstn)begin
    if(!rstn)begin
        for(i=0;i<128;i=i+1)begin
                dmem[i]<=0;
        end
    end
    else if(DMWr==1)begin
        case(DMType)
        `dm_word:begin
            dmem[addr]<=din[7:0];
            dmem[addr+1]<=din[15:8];
            dmem[addr+2]<=din[23:16];
            dmem[addr+3]<=din[31:24];
            end
        `dm_halfword:begin
            dmem[addr]<=din[7:0];
            dmem[addr+1]<=din[15:8];
            end
        `dm_halfword_unsigned:begin
            dmem[addr]<=din[7:0];
            dmem[addr+1]<=din[15:8];
            end
        `dm_byte:dmem[addr]<=din[7:0];
        `dm_byte_unsigned:dmem[addr]<=din[7:0];
        default:dmem[addr]<=dmem[addr];
        endcase
    end
    end

    always@(*)begin
        case(DMType)
        `dm_word:dout={dmem[addr+3][7:0],dmem[addr+2][7:0],dmem[addr+1]
[7:0],dmem[addr][7:0]};
        `dm_halfword:dout={{16{dmem[addr+1][7]}},dmem[addr+1][7:0],dmem[addr]
[7:0]};
        `dm_halfword_unsigned:dout={16'b0,dmem[addr+1][7:0],dmem[addr][7:0]};
        `dm_byte:dout={{24{dmem[addr][7]}},dmem[addr][7:0]};
        `dm_byte_unsigned:dout={24'b0,dmem[addr][7:0]};
        default:dout=0;
        endcase
    end
```

```
endmodule

module ctrl(
input [6:0] Op,  //opcode
input [6:0] Funct7,  //funct7
input [2:0] Funct3,    // funct3
input Zero,
input GE,
output RegWrite, // control signal for register write
output MemWrite, // control signal for memory write
output  [5:0]EXTOp,    // control signal to signed extension
output [4:0] ALUOp,    // ALU opertion
output [2:0] NPCOp,    // next pc operation
output ALUSrc,   // ALU source for b
output [2:0] DMType, //dm r/w type
output [1:0] WDSel   // (register) write data selection  (MemtoReg)
);   //R_type
  wire rtype  = ~Op[6]&Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //0110011
  wire
i_add=rtype&~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]&~Funct3[1]&~Funct3[0]; // add 0000000 000
  wire
i_sub=rtype&~Funct7[6]&Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~Fu
nct7[0]&~Funct3[2]&~Funct3[1]&~Funct3[0]; // sub 0100000 000
  wire
i_sll=rtype&~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&~Funct3[2]&~Funct3[1]&Funct3[0];
  wire
i_sra=rtype&~Funct7[6]&Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~Fu
nct7[0]&Funct3[2]&~Funct3[1]&Funct3[0];
  wire
i_srl=rtype&~Funct7[6]&~Funct7[5]&~Funct7[4]&~Funct7[3]&~Funct7[2]&~Funct7[1]&~F
unct7[0]&Funct3[2]&~Funct3[1]&Funct3[0];

  //i_l type
  wire itype_l  = ~Op[6]&~Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //0000011
  wire i_lb=itype_l&~Funct3[2]& ~Funct3[1]& ~Funct3[0]; //lb 000
  wire i_lh=itype_l&~Funct3[2]& ~Funct3[1]& Funct3[0];  //lh 001
  wire i_lw=itype_l&~Funct3[2]& Funct3[1]& ~Funct3[0];  //lw 010

  // i_i type
  wire itype_r  = ~Op[6]&~Op[5]&Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; //0010011
  wire i_addi  =  itype_r& ~Funct3[2]& ~Funct3[1]& ~Funct3[0]; // addi 000 func3

  // s format
  wire stype  = ~Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0];//0100011
  wire i_sw   = ~Funct3[2]&Funct3[1]&~Funct3[0];// sw 010
  wire i_sb=stype& ~Funct3[2]& ~Funct3[1]&~Funct3[0];
  wire i_sh=stype&& ~Funct3[2]&~Funct3[1]&Funct3[0];

  // b format
  wire btype  = Op[6]&Op[5]&~Op[4]&~Op[3]&~Op[2]&Op[1]&Op[0]; // 1100011
  wire i_beq = btype & (Funct3 == 3'b000);
  wire i_bne = btype & (Funct3 == 3'b001);
  wire i_blt = btype & (Funct3 == 3'b100);
  wire i_bge = btype & (Funct3 == 3'b101);
  wire i_bltu = btype & (Funct3 == 3'b110);
  wire i_bgeu = btype & (Funct3 == 3'b111);
  //j type ALUOp_jal   5'b10010
  wire i_jal=Op[6]&Op[5]&~Op[4]&Op[3]&Op[2]&Op[1]&Op[0];//1101111
  wire i_jalr =  Op[6] & Op[5] & ~Op[4] & ~Op[3] & Op[2] & Op[1] & Op[0];// 110
0111

  assign RegWrite   = rtype | itype_r| itype_l  |i_jal | i_jalr; // register
write
  assign MemWrite   = stype;              // memory write
  assign ALUSrc     = itype_r | stype | itype_l | i_jal |i_jalr; // ALU B is
from instruction immediate

  //操作指令生成运算类型aluop
  //ALUOp_nop 5'b00000
  //ALUOp_add 5'b00011
  //ALUOp_sub 5'b00100
//`define ALUOp_blt 5'b00110
//`define ALUOp_bge 5'b00111
//`define ALUOp_bltu 5'b01000
//`define ALUOp_bgeu 5'b01001
//`define ALUOp_sll 5'b01111
//`define ALUOp_srl 5'b10000
//`define ALUOp_sra 5'b10001
  assign ALUOp[0]= i_add|i_addi|stype|itype_l|i_jal| i_jalr|i_bgeu |i_bge|
i_sra|i_sll;
  assign ALUOp[1]= i_add|i_addi|stype|itype_l|i_jal | i_jalr |i_bge|i_blt|i_sll;
  assign ALUOp[2]= i_sub | i_beq|i_bne|i_bge|i_blt|i_sll;
  assign ALUOp[3]= i_bltu| i_bgeu|i_sll;
  assign ALUOp[4]= i_srl| i_sra;
  //操作指令生成常数扩展操作
  assign EXTOp[0]  =  i_jal;
  assign EXTOp[2]  = btype;
  assign EXTOp[3]  =  stype;
  assign EXTOp[4]  =  itype_l | itype_r | i_jalr;

  //根据具体S和i_L指令生成DataMem数据操作类型编码
  // DM_word 3'b000
  //DM_halfword 3'b001
  //DM_halfword_unsigned 3'b010
  //DM_byte 3'b011
  //DM_byte_unsigned 3'b100
  assign DMType[2]  = 0;
  assign DMType[1]=i_lb | i_sb; //| i_lhu;
  assign DMType[0]=i_lh | i_sh | i_lb | i_sb;

    // NPC control signal
    //NPC_PLUS4   3'b000
    //NPC_BRANCH  3'b001
    //NPC_JUMP    3'b010
    //NPC_JALR    3'b100
  assign NPCOp[0]=
i_beq&Zero|i_bne&~Zero|i_bge&GE|i_blt&~GE|i_bgeu&GE|i_blt&~GE;
  assign NPCOp[1]=i_jal;
  assign NPCOp[2]=i_jalr;

//  `define WDSel_FromALU 2'b00
//  `define WDSel_FromMEM 2'b01
//  `define WDSel_FromPC 2'b10
```

```
  assign WDSel[0] = itype_l;
  assign WDSel[1] = i_jal | i_jalr;

endmodule

module EXT(
input [4:0] iimm_shamt, //
input [11:0]    iimm,  //instr[31:20], 12 bits
input [11:0]    simm, //instr[31:25, 11:7], 12 bits
input [11:0]    bimm,//instrD[31],instrD[7], instrD[30:25], instrD[11:8], 12
bits
input [19:0]    uimm,
input [19:0]    jimm,
input [5:0]  EXTOp,
output reg [31:0]   immout
);
    always@(*)begin
    case (EXTOp)
        `EXT_CTRL_ITYPE_SHAMT:   immout<={27'b0,iimm_shamt[4:0]};
        `EXT_CTRL_ITYPE:    immout<={ {20{ iimm[11]}},iimm[11:0]};
        `EXT_CTRL_STYPE:    immout<={ {20{ simm[11]}},simm[11:0]};
        `EXT_CTRL_BTYPE:    immout<={ {19{ bimm[11]}},bimm[11:0], 1'b0} ;
        `EXT_CTRL_UTYPE:    immout <= {uimm[19:0], 12'b0};
        `EXT_CTRL_JTYPE:    immout<={{11{ jimm[19]}},jimm[19:0],1'b0};
        default:  immout <= 32'b0;
     endcase
    end

endmodule

module NPC(
    input [31:0]PC,
    input [2:0]NPCOp,
    input [31:0]IMM,
    input [31:0]aluout,
    output reg [31:0] NPC);

    wire [31:0] PCPLUS4;
    assign PCPLUS4=PC+4;
    always@(*)begin
        case (NPCOp)
        `NPC_PLUS4:  NPC<=PCPLUS4;
        `NPC_BRANCH: NPC<=PC+IMM;
        `NPC_JUMP:   NPC<=PC+IMM;
        `NPC_JALR:   NPC<=aluout;
        default: NPC<=PCPLUS4;
        endcase
    end
endmodule

module PC(
    input clk,
    input rstn,
    input [31:0]NPC,
    input PCwr,
    input [3:0]  sw_i,
    output reg [31:0]PC);
    always@(posedge clk or negedge rstn)begin
    if(!rstn)begin
        case (sw_i)
            4'b0000:PC <= 32'h0000_0000; //beq
            4'b0001:PC <= 32'h0000_0080; //bne
            4'b0010:PC <= 32'h0000_0100; //blt
            4'b0011:PC <= 32'h0000_0180; //bge
            4'b0100:PC <= 32'h0000_0200; //bltu
            4'b0101:PC <= 32'h0000_0280; //bgeu
            4'b0110:PC <= 32'h0000_0300; //jal
            4'b0111:PC <= 32'h0000_037c; //jalr
            4'b1000:PC <= 32'h0000_03f0; //sll
            4'b1001:PC <= 32'h0000_0410; //srl
            4'b1010:PC <= 32'h0000_0430; //sra
        endcase
    end
    else if(PCwr==1)
        PC<=NPC;
    else
        PC<=PC;
    end
endmodule
```

在ip calalog 页面 输入查找关键字memory，并点击结果distributed memory 选项