# Lab7 - mips-cpu 设计

**金泽文 PB15111604**

## 实验目的：

设计更为全面，更为优秀的 mips cpu，以加深对 Computer Organization and Design 的理解。

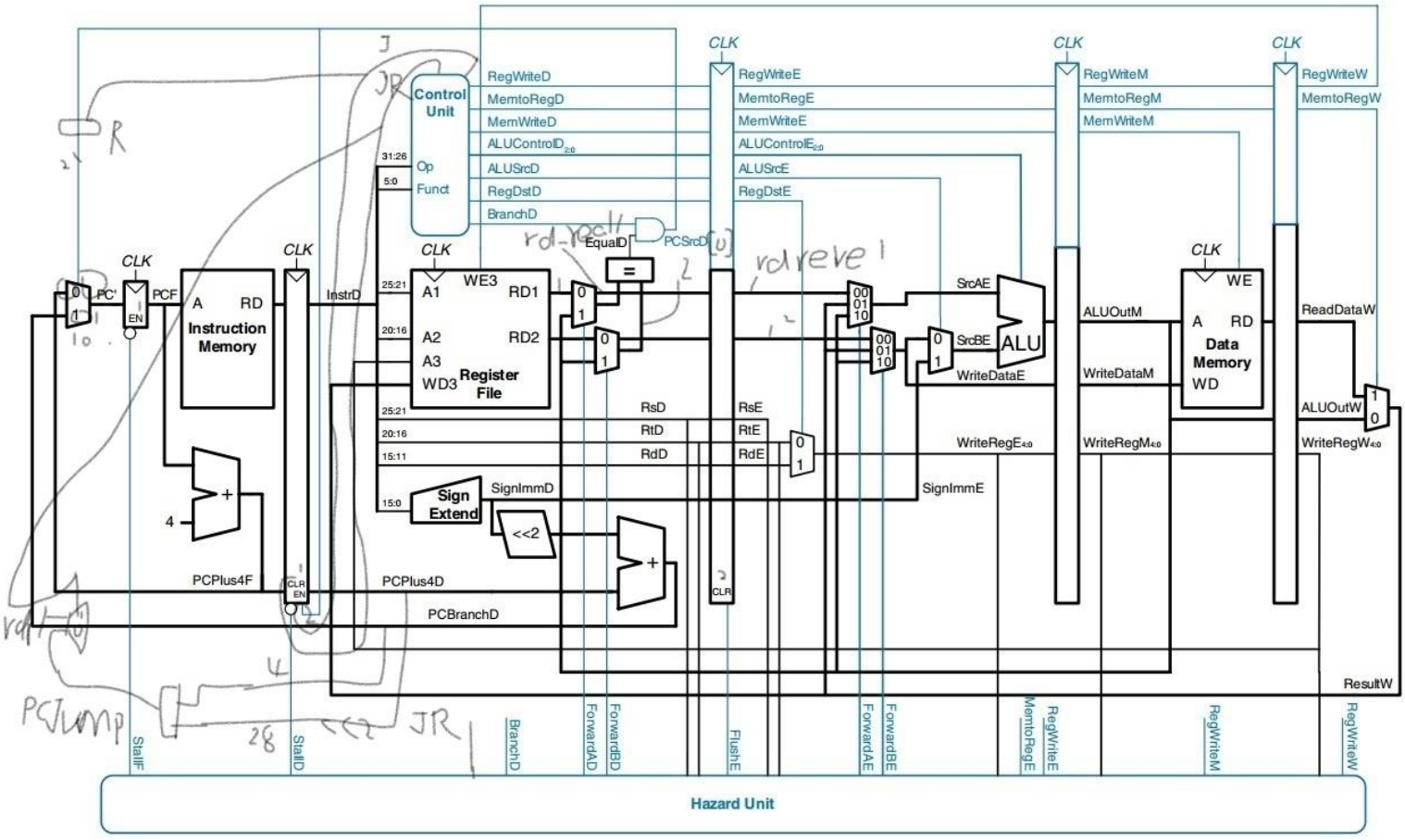
## 实验内容：

设计并实现流水线、静态分支预测功能，与对 16 个基本指令基础上的 20 条指令的处理功能。

实验分析与设计：

首先是对流水线整体的思考。根据 COD 书上的内容和 **David Money Harris** 的《Digital Design and Computer Architecture》上面的讲解，进行设计。

总体按照下图设计。




（原图来自这本书，为了实现 j,jr 加了一些东西。大体参考了这本书对于单周期 cpu 的设计。）

为了避免数据冒险，首先要设置前推寄存器 ForwardAE，和 ForwardBE。

```
if((RsE != 0) && (RsE == WriteRegM) && RegWriteM)
    ForwardAE=2;
else if((RsE != 0) && (RsE == WriteRegW) && RegWriteW)
        ForwardAE=1;
```

判断条件就是，注意要优先判断 M，后判断 W。同时为了考虑 lw 对 reg 的影

```
lwstall = ((RsD == RtE) | (RtD == RtE)) & MemtoRegE;
```

```
StallF = lwstall
StallD = lwstall
FlushE = lwstall
```

响，需要考虑 stall 和 flush

为了避免控制冒险，需要考虑 beq 等和 jr 等。

```
assign ForwardAD = (RsD != 0) & (RsD == WriteRegM) &
    RegWriteM;
assign ForwardBD = (RtD != 0) & (RtD == WriteRegM) &
    RegWriteM;
```

```
wire branchstall = (BranchD | JumpR ) & RegWriteE &(
    WriteRegE == RsD | WriteRegE == RtD) | (BranchD|
    JumpR ) & MemtoRegM & (WriteRegM == RsD | WriteRegM
    == RtD);
```

将以上封装在 hazard 模块中。

对于 16 条指令（以及额外实现的 4 条指令），设置 control 模块，在其中类比 lab6 针对 opcode 和 funct。针对运算，由于 funct 对应运算和我们之前实现的 alu 不一致，所以设置 alu_decoder 转码模块。

对于分支预测，我实现了静态分支预测。

只需要简单的根据寄存器读出来的两个值 RD1 和 RD2 是否相等，设置 EqualD 与 control 输出的 BranchD 进行比较即可。另外考虑到 J,JR,需要根据 control 输出的 Jump 和 JumpR 进一步判断。

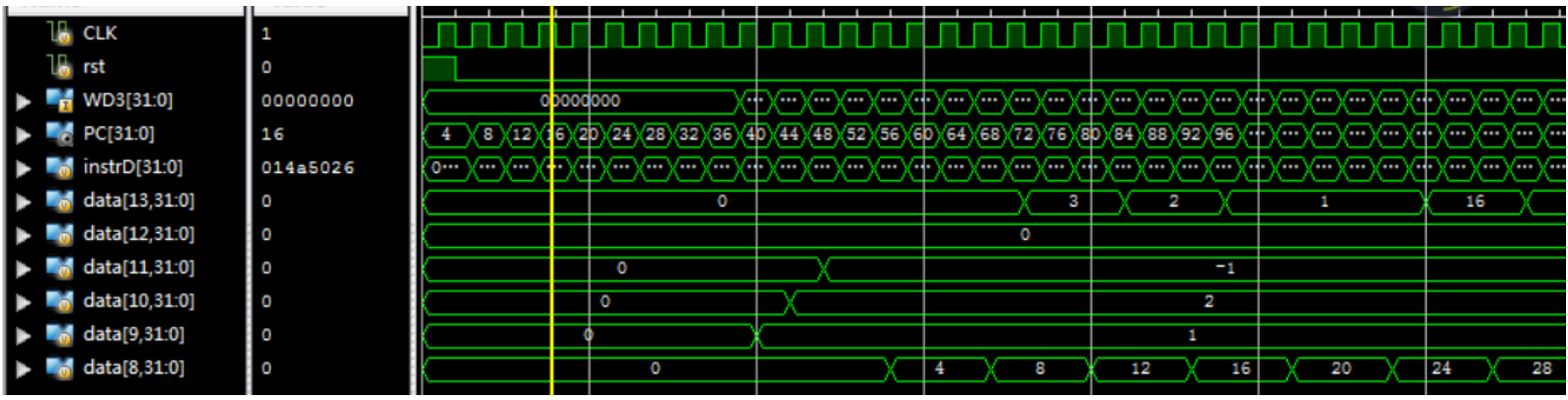
为了针对 beq，bgtz，以及加了 4 条 b 指令后继续沿用 equalD，所以直接设置 equalD 模块得到 equalD 结果，用于判断 PCSrcD。

另外值得一说的是，为了将单个周期控制在一个时钟周期，所以将寄存器文件的触发沿改为了 CLK 的下降沿。

最后，感谢 llxx 老师和四位助教一个学期以来的知道和陪伴！你们都是坠胖的！

实验结果：

仿真波形：




内存：

| | | |
|---|---|---|
| 0x0 | 0 | 0 |
| 0x2 | 3 | 2 |
| 0x4 | 1 | 1 |
| 0x6 | 16 | 3 |
| 0x8 | 0 | -2 |
| 0xA | 1 | 0 |
| 0xC | 8 | 2 |
| 0xE | 0 | 0 |
| 0x10 | 0 | 0 |
| 0x12 | 0 | 0 |

实验代码：

Top：

```
module top(
input                CLK,
input                rst
);
wire    [31:0]       PC;
wire    [31:0]       PCPlus4F;
reg     [31:0]       PCPlus4D;
wire    [31:0]       PCBranchD;
reg     [31:0]       PCF;
wire                 BranchD;
wire                 Jump;
wire                 JumpR;
wire    [31:0]       instr;
reg     [31:0]       instrD;
wire    [31:0]       SignImmD;
reg     [31:0]       SignImmE;
wire                 EN1;
wire                 EN2;
wire                 RegWriteD;
reg                  RegWriteE;
reg                  RegWriteM;
reg                  RegWriteW;
wire                 MemtoRegD;
reg                  MemtoRegE;
reg                  MemtoRegM;
reg                  MemtoRegW;
wire                 MemWriteD;
reg                  MemWriteE;
reg                  MemWriteM;
wire    [2:0]        ALUControlD;
wire    [2:0]        ALUOp;
reg     [2:0]        ALUControlE;
wire                 ALUSrcD;
reg                  ALUSrcE;
wire                 RegDstD;
reg                  RegDstE;
wire    [31:0]       RD1;
wire    [31:0]       RDreal1;
reg     [31:0]       RDrealreal1;
wire    [31:0]       RD2;
wire    [31:0]       RDreal2;
reg     [31:0]       RDrealreal2;
wire    [1:0]        PCSrcD;
wire                 EqualD;
wire    [4:0]        RsD = instrD[25:21];
wire    [4:0]        RtD = instrD[20:16];
wire    [4:0]        RdD = instrD[15:11];
reg     [4:0]        RsE;
reg     [4:0]        RtE;
reg     [4:0]        RdE;
wire    [31:0]       WriteDataE;
reg     [31:0]       WriteDataM;
wire    [31:0]       ALUOutE;
reg     [31:0]       ALUOutM;
reg     [31:0]       ALUOutW;
wire    [4:0]        WriteRegE;
reg     [4:0]        WriteRegM;
reg     [4:0]        WriteRegW;
wire    [31:0]       SrcAE;
wire    [31:0]       SrcBE;
wire    [31:0]       ReadDataM;
reg     [31:0]       ReadDataW;
wire                 StallF;
wire                 StallD;
wire                 ForwardAD;
wire                 ForwardBD;
wire    [1:0]        ForwardAE;
wire    [1:0]        ForwardBE;
wire                 FlushE;
wire                 CLR;
wire    [31:0]       PCJump;
wire    [31:0]       PCJumpR;
wire    [31:0]       ResultW = MemtoRegW ? ReadDataW : ALUOutW;
assign PC = PCSrcD == 0 ? PCPlus4F : PCSrcD == 1 ? PCBranchD : PCJumpR;
assign EN1 = ~StallF;
assign EN2 = ~StallD;
assign PCPlus4F = PCF + 32'd4;
assign PCSrcD[0] = EqualD && BranchD;
assign PCSrcD[1] = Jump;
assign RDreal1 = ForwardAD ? ALUOutM : RD1;
assign RDreal2 = ForwardBD ? ALUOutM : RD2;
assign CLR = PCSrcD || Jump || JumpR;
assign SrcAE = ForwardAE == 0 ? RDrealreal1 : (
    ForwardAE == 1 ? ResultW : ALUOutM);
assign WriteDataE = ForwardBE == 0 ? RDrealreal2 : (
    ForwardBE == 1 ? ResultW : ALUOutM);
assign SrcBE = ALUSrcE ? SignImmE : WriteDataE;
assign WriteRegE = RegDstE ? RdE : RtE;
assign PCJump[27:0] = instrD[25:0] << 2;
assign PCJump[31:28] = PCPlus4D[31:28];
assign PCJumpR = JumpR ? RDreal1 : PCJump;
assign PCBranchD = (SignImmD << 2) + PCPlus4D;
always@(posedge CLK or posedge rst) begin
    if(rst)begin
        instrD <= 0;
        PCPlus4D <= 0;
    end
    else if(EN2) begin
        if(CLR)
        begin
            instrD<=0;
            PCPlus4D<=0;
        end
        else
        begin
            instrD<=instr;
            PCPlus4D<=PCPlus4F;
        end
    end
end
always@(posedge CLK or posedge rst) begin
    if(rst)begin
        RDrealreal1 <= 0;
        RDrealreal2 <= 0;
        RsE <= 0;
        RtE <= 0;
        RdE <= 0;
        RegWriteE <= 0;
        MemtoRegE <= 0;
        MemWriteE <= 0;
        ALUControlE <= 0;
        ALUSrcE <= 0;
        RegDstE <= 0;
        SignImmE <= 0;
    end
    else begin
        RDrealreal1 <= FlushE ? 0 : RD1 ;
        RDrealreal2 <= FlushE ? 0 : RD2 ;
        RsE <= FlushE ? 0 : RsD;
        RtE <= FlushE ? 0 : RtD;
        RdE <= FlushE ? 0 : RdD;
        RegWriteE <= FlushE ? 0 : RegWriteD;
        MemtoRegE <= FlushE ? 0 : MemtoRegD;
        MemWriteE <= FlushE ? 0 : MemWriteD;
        ALUControlE <= FlushE ? 0 : ALUControlD;
        ALUSrcE <= FlushE ? 0 : ALUSrcD;
        RegDstE <= FlushE ? 0 : RegDstD;
        SignImmE <= FlushE ? 0 : SignImmD;
    end
end
always@(posedge CLK or posedge rst) begin
    if(rst)begin
        PCF <= 0;

        RegWriteM <= 0;
        MemtoRegM <= 0;
        MemWriteM <= 0;
        ALUOutM <= 0;
        WriteDataM <= 0;
        WriteRegM <= 0;
    end
    else begin
        if(EN1)
            PCF <= PC;
        RegWriteM <= RegWriteE;
        MemtoRegM <= MemtoRegE;
        MemWriteM <= MemWriteE;
        ALUOutM <= ALUOutE;
        WriteDataM <= WriteDataE;
        WriteRegM <= WriteRegE;
    end
end
always@(posedge CLK or posedge rst) begin
    if(rst) begin
        RegWriteW <= 0;
        MemtoRegW <= 0;
        ReadDataW <= 0;
        ALUOutW <= 0;
        WriteRegW <= 0;
    end
    else begin
        RegWriteW <= RegWriteM;
        MemtoRegW <= MemtoRegM;
        ReadDataW <= ReadDataM;
        ALUOutW <= ALUOutM;
        WriteRegW <= WriteRegM;
    end
end
control      u_control(
.Op          (instrD[31:26]       ),
.Funct       (instrD[5:0]         ),
.RegWriteD   (RegWriteD           ),
.MemtoRegD   (MemtoRegD           ),
.MemWriteD   (MemWriteD           ),
.ALUOp       (ALUOp               ),
.ALUSrcD     (ALUSrcD             ),
.RegDstD     (RegDstD             ),
.BranchD     (BranchD             ),
.Jump        (Jump                ),
.JumpR       (JumpR               )
);
REG_FILE    u_REG_FILE(
.CLK         (CLK                 ),
.rst         (rst                 ),
.A1          (instrD[25:21]       ),
.A2          (instrD[20:16]       ),
.A3          (WriteRegW           ),
.WD3         (ResultW             ),
.WE3         (RegWriteW           ),
.RD1         (RD1                 ),
.RD2         (RD2                 )
);
InsMem     u_InsMem(
           PCF[7:2],
           instr
);
DataMem    u_DataMem(
           ALUOutM[6:2],
           WriteDataM,
           CLK,
           MemWriteM,
           ReadDataM
);
sign_extend  u_sign_extend(
           instrD[15:0],
           SignImmD
);
alu          u_alu(
.SrcAE       (SrcAE               ),
.SrcBE       (SrcBE               ),
.ALUControlE (ALUControlE         ),
.ALUOutE     (ALUOutE             )
);
alu_decoder  u_alu_decoder(
.ALUOp       (ALUOp               ),
.Funct       (instrD[5:0]         ),
.ALUControl  (ALUControlD         )
);
hazard       u_hazard(
.WriteRegE   (WriteRegE           ),
.WriteRegM   (WriteRegM           ),
.WriteRegW   (WriteRegW           ),
.RegWriteE   (RegWriteE           ),
.RegWriteM   (RegWriteM           ),
.RegWriteW   (RegWriteW           ),
.MemtoRegE   (MemtoRegE           ),
.MemtoRegM   (MemtoRegM           ),
.BranchD     (BranchD             ),
.JumpR       (JumpR               ),
.RtE         (RtE                 ),
.RsE         (RsE                 ),
.RtD         (RtD                 ),
.RsD         (RsD                 ),
.StallF      (StallF              ),
.StallD      (StallD              ),
.ForwardAD   (ForwardAD           ),
.ForwardBD   (ForwardBD           ),
.FlushE      (FlushE              ),
.ForwardAE   (ForwardAE           ),
.ForwardBE   (ForwardBE           )
);
equald       u_equald(
.RDreal1     (RDreal1             ),
.RDreal2     (RDreal2             ),
.Op          (instrD[31:26]       ),
.RtD         (RtD                 ),
.EqualD      (EqualD              )
);
endmodule
```

Reg_file

```
module REG_FILE(
input                CLK,
input                rst,
input [4:0]          A1,
input [4:0]          A2,
input [4:0]          A3,
input [31:0]         WD3,
input                WE3,
output reg [31:0]    RD1,
output reg [31:0]    RD2
);

reg [31:0] data [31:0];
always@(negedge CLK or posedge rst)
begin
    if (rst) begin
        data[0] <= 32'b0;
        data[1] <= 32'b0;
        data[2] <= 32'b0;
        data[3] <= 32'b0;
        data[4] <= 32'b0;
        data[5] <= 32'b0;
        data[6] <= 32'b0;
        data[7] <= 32'b0;
        data[8] <= 32'b0;
        data[9] <= 32'b0;
        data[10] <= 32'b0;
        data[11] <= 32'b0;
        data[12] <= 32'b0;
        data[13] <= 32'b0;
        data[14] <= 32'b0;
        data[15] <= 32'b0;
        data[16] <= 32'b0;
        data[17] <= 32'b0;
        data[18] <= 32'b0;
        data[19] <= 32'b0;
        data[20] <= 32'b0;
        data[21] <= 32'b0;
        data[16] <= 32'b0;
        data[17] <= 32'b0;
        data[18] <= 32'b0;
        data[19] <= 32'b0;
        data[20] <= 32'b0;
        data[21] <= 32'b0;
        data[22] <= 32'b0;
        data[23] <= 32'b0;
        data[24] <= 32'b0;
        data[25] <= 32'b0;
        data[26] <= 32'b0;
        data[27] <= 32'b0;
        data[28] <= 32'b0;
        data[29] <= 32'b0;
        data[30] <= 32'b0;
        data[31] <= 32'b0;
    end
    else if(WE3)
        data[A3]<= WD3;
end

always@(*)
begin
    if(A1)
        RD1 = data[A1];
    else
        RD1 = 32'h0;
end

always@(*)begin
    if(A2)
        RD2 = data[A2];
    else
        RD2 = 32'h0;
end
endmodule

```

Alu :

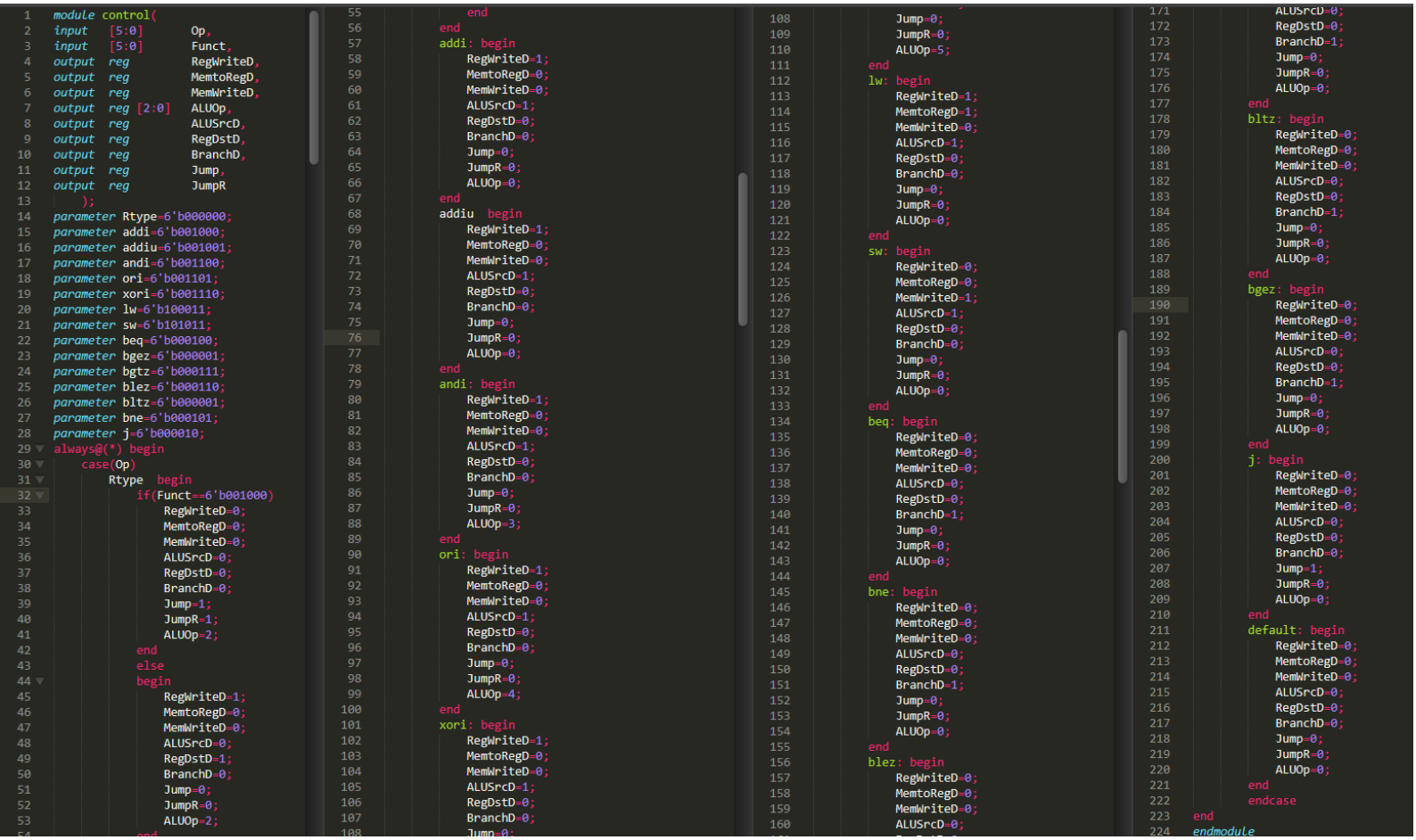
```
module alu(
input   signed  [31:0]  SrcAE,
input   signed  [31:0]  SrcBE,
input           [2:0]   ALUControlE,
output  reg     [31:0]  ALUOutE
);

always@(*)
begin
    case(ALUControlE)
        3'h0: ALUOutE = 32'h0;
        3'h1: ALUOutE = SrcAE + SrcBE;
        3'h2: ALUOutE = SrcAE - SrcBE;
        3'h3: ALUOutE = SrcAE & SrcBE;
        3'h4: ALUOutE = SrcAE | SrcBE;
        3'h5: ALUOutE = SrcAE ^ SrcBE;
        3'h6: ALUOutE = ~(SrcAE | SrcBE);
        default: ALUOutE = 32'h0;
    endcase
end

endmodule

```

control



```
module control(
input   [5:0]       Op,
input   [5:0]       Funct,
output  reg         RegWriteD,
output  reg         MemtoRegD,
output  reg         MemWriteD,
output  reg [2:0]   ALUOp,
output  reg         ALUSrcD,
output  reg         RegDstD,
output  reg         BranchD,
output  reg         Jump,
output  reg         JumpR
    );
parameter Rtype=6'b000000;
parameter addi=6'b001000;
parameter addiu=6'b001001;
parameter andi=6'b001100;
parameter ori=6'b001101;
parameter xori=6'b001110;
parameter lw=6'b100011;
parameter sw=6'b101011;
parameter beq=6'b000100;
parameter bgez=6'b000001;
parameter bgtz=6'b000111;
parameter blez=6'b000110;
parameter bltz=6'b000001;
parameter bne=6'b000101;
parameter j=6'b000010;
always@(*) begin
    case(Op)
        Rtype  begin
            if(Funct==6'b001000)
                RegWriteD=0;
                MemtoRegD=0;
                MemWriteD=0;
                ALUSrcD=0;
                RegDstD=0;
                BranchD=0;
                Jump=1;
                JumpR=1;
                ALUOp=2;
            end
            else
            begin
                RegWriteD=1;
                MemtoRegD=0;
                MemWriteD=0;
                ALUSrcD=0;
                RegDstD=1;
                BranchD=0;
                Jump=0;
                JumpR=0;
                ALUOp=2;
            end
        end
        addi: begin
            RegWriteD=1;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        addiu  begin
            RegWriteD=1;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        andi: begin
            RegWriteD=1;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=3;
        end
        ori: begin
            RegWriteD=1;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=4;
        end
        xori: begin
            RegWriteD=1;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=5;
        end
        lw: begin
            RegWriteD=1;
            MemtoRegD=1;
            MemWriteD=0;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        sw: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=1;
            ALUSrcD=1;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        beq: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=1;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        bne: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=1;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        blez: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=1;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        bltz: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=1;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        bgez: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=1;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        j: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=0;
            Jump=1;
            JumpR=0;
            ALUOp=0;
        end
        default: begin
            RegWriteD=0;
            MemtoRegD=0;
            MemWriteD=0;
            ALUSrcD=0;
            RegDstD=0;
            BranchD=0;
            Jump=0;
            JumpR=0;
            ALUOp=0;
        end
        endcase
end
endmodule
```

alu_decoder:

```
module alu_decoder(
    input [2:0]ALUOp,
    input [5:0]Funct,
    output reg [2:0]ALUControl
    );

    always@(*)
    begin
    case(ALUOp)
        0:  ALUControl<=1;
        1:  ALUControl<=2;
        2:
        begin
            case(Funct)
                6'b100000: ALUControl<=1;
                6'b100001: ALUControl<=1;
                6'b100010: ALUControl<=2;
                6'b100011: ALUControl<=2;
                6'b100100: ALUControl<=3;
                6'b100101: ALUControl<=4;
                6'b100110: ALUControl<=5;
                6'b100111: ALUControl<=6;
                6'b001000: ALUControl<=0;
                default:       ALUControl<=3'b000;
            endcase
        end
        3:  ALUControl<=3;
        4:  ALUControl<=4;
        5:  ALUControl<=5;
        6:  ALUControl<=6;
        7:  ALUControl<=7;
        default: ALUControl<=1;
    endcase
    end

endmodule
```

hazard:

```
module hazard(
input                 BranchD,
input                 MemtoRegE,
input                 RegWriteE,
input                 MemtoRegM,
input                 RegWriteM,
input                 RegWriteW,
input                 JumpR,
input       [4:0]     RsD,
input       [4:0]     RtD,
input       [4:0]     RsE,
input       [4:0]     RtE,
input       [4:0]     WriteRegE,
input       [4:0]     WriteRegM,
input       [4:0]     WriteRegW,
output                StallF,
output                StallD,
output                ForwardAD,
output                ForwardBD,
output                FlushE,
output reg  [1:0]     ForwardAE,
output reg  [1:0]     ForwardBE
    );
```

```
wire lwstall = ((RsD == RtE) | (RtD == RtE)) & MemtoRegE;
wire branchstall = (BranchD | JumpR ) & RegWriteE &(
    WriteRegE == RsD | WriteRegE == RtD) | (BranchD|
    JumpR ) & MemtoRegM & (WriteRegM == RsD | WriteRegM
    == RtD);

always@(*) begin
    ForwardAE=0;
    ForwardBE=0;
    if((RsE != 0) && (RsE == WriteRegM) && RegWriteM)
        ForwardAE=2;
    else if((RsE != 0) && (RsE == WriteRegW) && RegWriteW
        )
            ForwardAE=1;
    if((RtE != 0) && (RtE == WriteRegM) && RegWriteM)
        ForwardBE=2;
    else if((RtE != 0) && (RtE == WriteRegW) && RegWriteW
        )
            ForwardBE=1;
end

assign ForwardAD = (RsD != 0) & (RsD == WriteRegM) &
    RegWriteM;
assign ForwardBD = (RtD != 0) & (RtD == WriteRegM) &
    RegWriteM;

assign StallF = lwstall | branchstall;
assign StallD = lwstall | branchstall;
assign FlushE = lwstall | branchstall;

endmodule
```

equald:

```
module equald(
    input signed [31:0] RDreal1,
    input signed [31:0] RDreal2,
    input        [5:0]  Op,
    input        [4:0]  RtD,
    output reg          EqualD
    );

    parameter beq = 6'b000100;
    parameter bgez = 6'b000001;
    parameter bgtz = 6'b000111;
    parameter blez = 6'b000110;
    parameter bltz = 6'b000001;
    parameter bne = 6'b000101;

    always@(*)
    begin
        case(Op)
        beq:    if(RDreal1 == RDreal2)
                EqualD = 1;
                else
                EqualD = 0;
        bgez:   if(RtD == 5'b00001)//bgez
                        begin
                            if(RDreal1> = 0)
                            EqualD = 1;
                            else
                            EqualD = 0;
                        end
                        else
                        begin
                            if(RDreal1<0)//bltz
                            EqualD = 1;
                            else
                            EqualD = 0;
                        end
        bgtz:   if(RDreal1 > 0)
                EqualD = 1;
                else
                EqualD = 0;
        blez:   if(RDreal1 <= 0)
                EqualD = 1;
                else
                EqualD = 0;
        bne:    if(RDreal1 != RDreal2)
                EqualD = 1;
                else
                EqualD = 0;
        default:EqualD = 0;
        endcase
    end

endmodule
```

sign_extend:

```
module sign_extend(
    input [15:0]din,
    output [31:0]dout
    );

assign dout[15:0]=din[15:0];
assign dout[31]=din[15];
assign dout[30]=din[15];
assign dout[29]=din[15];
assign dout[28]=din[15];
assign dout[27]=din[15];
assign dout[26]=din[15];
assign dout[25]=din[15];
assign dout[24]=din[15];
assign dout[23]=din[15];
assign dout[22]=din[15];
assign dout[21]=din[15];
assign dout[20]=din[15];
assign dout[19]=din[15];
assign dout[18]=din[15];
assign dout[17]=din[15];
assign dout[16]=din[15];

endmodule

```

sim:

```
4  module sim;
5
6      // Inputs
7      reg CLK;
8      reg rst;
9
10     // Instantiate the Unit Under Test (UUT)
11     top uut (
12         .CLK(CLK),
13         .rst(rst)
14     );
15
16     always #5 CLK=~CLK;
17
18     initial begin
19         // Initialize Inputs
20         CLK = 0;
21         rst = 1;
22
23         // Wait 100 ns for global reset to finish
24         #10;
25
26         rst = 0;
27         // Add stimulus here
28
29     end
30
31 endmodule
```