CPU学习笔记

HIT LAB

Lab1：基本组合逻辑设计

1.1. 加法器

请自行阅读，理解计算机中加法器的原理。实验要求设计有2个32位数输入和1个进位输入，产生1个32位的加法和结果和1个向高位的进位的加法器。

> adder.v

```
`timescale 1ns / 1ps     #定义时钟周期
module adder(                  #模块化adder
    input   [31:0]  ain,     #设置32bit输入ain，[]内限制宽度，高位在前
    input   [31:0]  bin,
    input           cin,     #设置上一级进位cin
    input           clk,     #设置clk是为了在测试中设置
    output   [31:0]  sout,
    output          co
);
assign {co, sout} = ain + bin + cin;   #注意，这里的写法{进位，求和}可以直接得到加法结果
endmodule
```

> adder_tb.v

```
`timescale 1ns / 1ps

module adder_tb;
    reg [31: 0] ain, bin;        #输入input在测试tb文件中用reg定义，因为输入寄存需要数值
    reg cin, clk;

    wire [31: 0] sout;           #输出output在测试tb文件中用wire定义，因为需要查看输出信号
    wire co;

    initial begin                #初始化输入参数
```

```
11          ain = 31'b0;
12          bin = 31'b0;
13          cin = 0;
14          clk = 0;
15      end
16
17      always #10 clk = ~clk;          #每隔10ns将clk翻转，生成时钟信号
18
19      always@(posedge clk) begin      #总是在clk信号上升沿触发，可以理解为if的条件
20          ain = $random;              #直接用$random表示为随机数，长度为所定义的长度
21          bin = $random;
22          cin = {$random} % 2;        #用$random/n表示[-(n-1),(n-1)]，{$random}/n表示
    [0,n-1]
23      end
24
25      adder u_adder(                  #例化信号线
26          .ain    (ain    ),
27          .bin    (bin    ),
28          .cin    (cin    ),
29          .clk    (clk    ),
30          .sout   (sout    ),
31          .co    (co    )
32      );
33
34  endmodule
```

1.2. ALU

基于前面完成的加法器设计，设计一个简单的算术逻辑单元，能够执行以下16种算术逻辑运算操作。实验要求ALU支持如下16种运算操作。

序号	运算操作	序号	运算操作
1	F=A 加 B	9	F=/A
2	F=A 加 B 加 Cin	10	F=/B
3	F=A 减 B	11	F=A+B
4	F=A 减 B 减 Cin	12	F=AB
5	F=A向左移位	13	F=A⊙B
6	F=A向右移位	14	F=A⊕B
7	F=A	15	F=/(AB)
8	F=B	16	F=0

alu.v

```
`define HALF_ADD 4'b0000            #宏定义ALU中操作数的操作码
`define FULL_ADD 4'b0001
`define SUB 4'b0010
`define SUB_CIN 4'b0011
`define SLL 4'b0100
`define RLL 4'b0101
`define A 4'b0110
`define B 4'b0111
`define NA 4'b1000
`define NB 4'b1001
`define OR 4'b1010
`define AND 4'b1011
`define XNOR 4'b1100
`define XOR 4'b1101
`define NAND 4'b1110
`define ZERO 4'b1111

`timescale 1ns / 1ps

module alu(
    input   [31: 0]     A, B,         #定义输入A,B,Cin ALU控制信号alu_sel
    input               Cin, clk,
    input   [3: 0]      alu_sel,

    output  [31: 0]     F,
    output              Cout
);
    reg [31: 0]     result;           #定义中间寄存器reg保存计算结果

    assign F = result;
    assign Cout = (A & B)|(A & Cin)|(B & Cin);    #全加算法中进位信号（有待优化）

    always@ (*) begin
        case(alu_sel)             #用case选择ALU控制信号，选择不同的计算方法
            `HALF_ADD:   result <= A + B;          #注意宏定义后要在前面加 `
            `FULL_ADD:   result <= A + B + Cin;    #不用宏定义可以直接  4'b0001
            `SUB:        result <= A - B;          #如果想在case中某一句赋值多个变量，可以用
            `SUB_CIN:    result <= A - B - Cin;    #begin {添加内容}  end 实现

            `SLL:        result <= A << 1;
            `RLL:        result <= A >> 1;
            `A:          result <= A;
            `B:          result <= B;
```

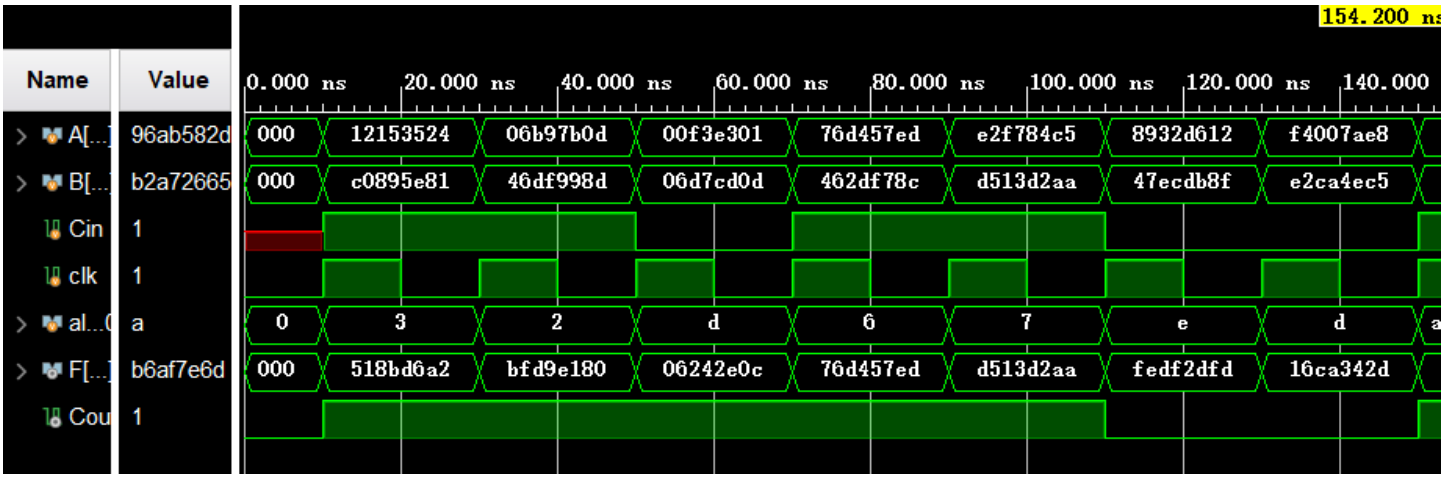
```
44
45              `NA:          result <= ~A;
46              `NB:          result <= ~B;
47              `OR:          result <= A | B;
48              `AND:         result <= A & B;
49
50              `XNOR:        result <= ~(A ^ B);
51              `XOR:         result <= A ^ B;
52              `NAND:        result <= ~(A & B);
53              `ZERO:        result <= 32'h00000000;
54
55              default:     result <= A + B;
56          endcase
57      end
58
59  endmodule
```

alu_tb.v

```
1  `timescale 1ns / 1ps
2
3  module alu_tb;
4
5      reg [31: 0] A, B;
6      reg Cin, clk;
7      reg [3: 0]  alu_sel;
8
9      wire [31: 0] F;
10     wire Cout;
11
12     alu u_alu(
13         .A      (A      ),
14         .B      (B      ),
15         .Cin    (Cin    ),
16         .clk    (clk    ),
17         .alu_sel(alu_sel),
18         .F      (F      ),
19         .Cout   (Cout   )
20     );
21
22     initial begin
23         A = 32'h00000000;
24         B = 32'h00000000;
25         alu_sel = 4'h0;
26         clk = 0;
```

```
27      end
28
29      always #10 clk = ~clk;
30
31      always@ (posedge clk) begin
32          A = $random;
33          B = $random;
34          Cin = $random;
35          alu_sel = $random;
36      end
37
38  endmodule
```

> **仿真结果**



Lab2：内存与寄存器堆

2.1. 寄存器堆

学习 MIPS 计算机中寄存器堆的设计原理。实验要求设计寄存器堆，包含1个写端口和2个读端口。

寄存器堆的信号说明如下。

名称	宽度	方向	描述
clk	1	IN	时钟信号
raddr1	5	IN	寄存器堆读地址1
rdata1	32	OUT	寄存器堆读返回数据1
raddr2	5	IN	寄存器堆读地址2
rdata2	32	OUT	寄存器堆读返回数据2
we	1	IN	寄存器堆写使能
waddr	5	IN	寄存器堆写地址
wdata	32	IN	寄存器堆写数据

regfile_tb.v

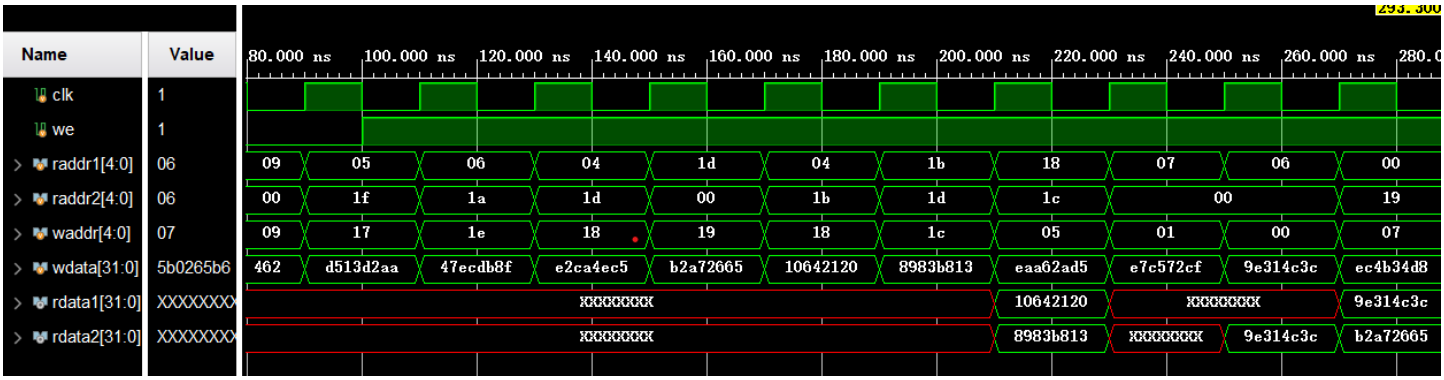
```
1  `timescale 1ns / 1ps
2
3  module regfile(
4      input           clk, we,                    #we-write enable 写寄存器需要使能
   信号，读不需要
5      input [4: 0]    raddr1, raddr2, waddr,      #5bit地址线，索引32个寄存器
6      input [31: 0]   wdata,                      #32bit数据线，数据位宽为32位
7
8      output [31: 0]   rdata1, rdata2
9  );
10
11     reg [31: 0]  Datareg[31: 0];        #Datareg表示32个寄存器，构成寄存器堆
12                                         #第一个[31:0]表示reg宽度为32bit
13                                         #第二个[31:0]表示索引的十进制范围，即reg个数
   为32个
14     always@ (posedge clk) begin
15         if(we)
16             Datareg[waddr] <= wdata;    #waddr索引regfile中的一个，写入wdata
17     end
18
19     assign rdata1 = Datareg[raddr1];    #读操作直接assign
20     assign rdata2 = Datareg[raddr2];
21
22 endmodule
```

regfile_tb.v

```
1  `timescale 1ns / 1ps
2
```

```
3  module regfile_tb;
4
5      reg           clk, we;
6      reg [4: 0]    raddr1, raddr2, waddr;
7      reg [31: 0]   wdata;
8
9      wire [31: 0]   rdata1, rdata2;
10
11     regfile u_regfile(
12         .clk(clk),
13         .we(we),
14         .raddr1(raddr1),
15         .raddr2(raddr2),
16         .waddr(waddr),
17         .wdata(wdata),
18         .rdata1(rdata1),
19         .rdata2(rdata2)
20     );
21
22     always #10 clk = ~clk;
23
24     initial begin
25         clk = 0;
26         we = 0;
27         raddr1 = 0;
28         raddr2 = 0;
29         waddr = 0;
30         wdata = 0;
31         #100 we =1;
32     end
33
34     always@ (posedge clk) begin
35         waddr = ($random)%10;
36         wdata = $random;
37         raddr1 = ($random)%10;
38         raddr2 = ($random)%10;
39     end
40
41 endmodule
```

仿真结果



分析：

1. 在Datareg中未写入数据时，读出对应地址的reg数据为******；
2. 在we信号使能后，时钟clk上升沿可以按照写地址索引到的reg写数据；
3. 写入的数据被保存在reg后，按照对应的读地址索引可以读出之前写入的对应位置的数据。

2.2. RAM

学习 RAM 的读写时序，并定制一块 RAM IP 核，在顶层文件中实例化。实验要求实现同步 RAM，定制深度为65536，定制宽度为32。

RAM 顶层模块的接口信号说明如下。

名称	宽度	方向	描述
clk	1	IN	时钟信号
ram_wen	1	IN	同步 RAM 写使能，置位时写入
ram_addr	16	IN	同步 RAM 地址信号，表示读/写地址
ram_wdata	32	IN	同步 RAM 写数据信号，表示写入数据
ram_rdata	32	OUT	同步 RAM 读数据信号，表示读出数据

Lab3：给定指令系统的处理器设计

3.1. 实验内容

- 根据《计算机体系结构》第五章里 MIPS 指令集的基本实现，设计并实现一个符合实验指令的**非流水**处理器， 包括 Verilog 语言的实现和 FPGA 芯片的编程实现，要求该处理器可以**通过所提供的自动测试环境**。

3.1.1. 处理器功能

本实验的任务是设计一个简单的RISC处理器，该处理器是在给定的指令集（与MIPS32类似）下构建的，支持12条指令。假定存储器分为数据缓冲存储器和指令缓冲存储器，且都可以在一个时钟周期内

完成一次同步存取操作，时钟信号和CPU相同。处理器的指令字长为32位，包含32个32位通用寄存器R0~R31，1个32位的指令寄存器IR和1个32位的程序计数器PC，1个256×32位指令缓冲存储器，1个256×32位的数据缓冲存储器。

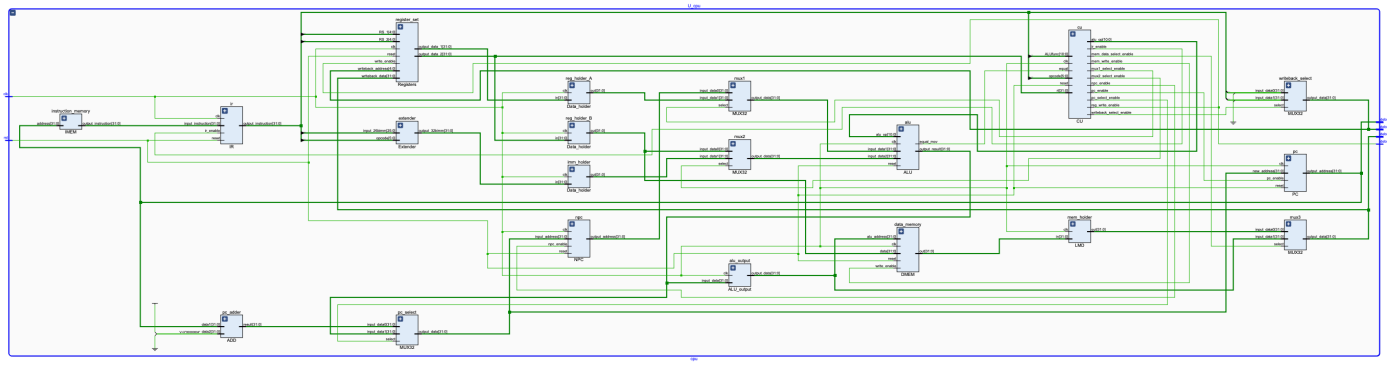
3.1.2. 指令系统定义

处理器所支持的指令包括 `LW`， `SW`， `ADD`， `SUB`， `SLL`，`AND`， `OR`， `XOR`， `SLT`， `MOVZ`， `BNE`， `J`。

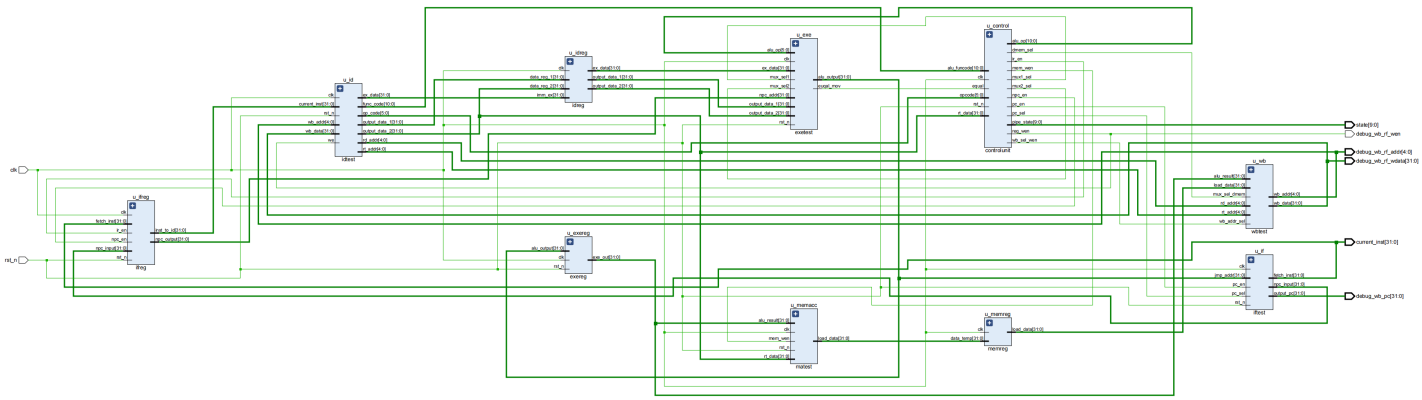
其中仅有 `LW` 和 `SW` 是字访存指令，所有的存储器访问都通过这两条指令完成； `ADD`、 `SUB`、 `SLL`、 `AND`、 `OR`、 `XOR`、 `SLT`、 `MOVZ` 是运算指令，他们都在处理器内部完成； `BNE` 是分支跳转指令，根据寄存器的内容进行相对跳转； `J` 是无条件转移指令。

> 整体结构

参考资料[1]中生成的电路RTL图如下：



修改后，全时钟上升沿同步触发逻辑如下：



该结构将寄存器if_reg, id_reg, exe_reg, mem_reg单独抽出，相当于为整个电路增加了四拍的延迟。

其中if, id, exe, ma, wb模块的具体RTL图下面列出。

下面列出全时钟上升沿触发的CPU实现代码

单周期处理器设计　CPUsyn:

cpu.vh　　verilog header 头文件　　定义一些文件路径

```
1  //宏定义指令、数据、寄存器初始化信息的文件路径
2  `define TRACE_FILE_PATH
   "E:\\HITCPU\\CPU_Design_Lab4\\CPUsyn\\CPUsyn.data\\cpu_trace"
3  `define INST_FILE_PATH
   "E:\\HITCPU\\CPU_Design_Lab4\\CPUsyn\\CPUsyn.data\\inst_data.txt"
4  `define DATA_FILE_PATH
   "E:\\HITCPU\\CPU_Design_Lab4\\CPUsyn\\CPUsyn.data\\data_data.txt"
5  `define GPRS_FILE_PATH
   "E:\\HITCPU\\CPU_Design_Lab4\\CPUsyn\\CPUsyn.data\\reg_data.txt"
6
7  `define ADD 6'b100000        //定义运算类型指令的功能码func_code(区别于操作码opcode)
8  `define SUB 6'b100010
9  `define AND 6'b100100
10 `define OR 6'b100101
11 `define XOR 6'b100110
12 `define SLT 6'b101010
13 `define MOVZ 6'b001010
14 `define SLL 6'b000000
15 `define SW 6'b101011             //定义非运算类型指令的操作码opcode
16 `define LW 6'b100011             //(SW, LW, BNE, J指令牺牲功能码地址换取操作数)
17 `define BNE 6'b000101
18 `define J 6'b000010
```

imem.v　　存储指令数据（组合逻辑实现，注意区别于数据存储的时序逻辑实现方式）

```
1  `include "cpu.vh"            //下面用到头文件中的指令数据文件
2
3  module imem(         //enter addr fetch inst
4      input [31: 0]   addr,
5
6      output[31: 0]   output_inst
7  );
8
9      reg [31: 0] data[255: 0];      //256×32 inst mem          //能容纳256条32-bit
   的指令
10                                    //这里索引data的数组大小为256
11     initial begin
12         $readmemh(`INST_FILE_PATH, data);        //从路径下的.txt文件读出存储的数据
13     end
```

```
                        //需注意，该路径下.txt文件应该存有32条32-bit的指令数据
    assign output_inst = data[addr / 4];        //addr      //用指令PC索引取出存储的指令信息

endmodule
```

regfile.v　　寄存器堆

```
`include "./cpu.vh"     //下面用到头文件中的寄存器初始化数据文件

module regfile(         //寄存器文件，与Lab3-3.3.1中不同之处在于添加了复位信号rst_n
    input           clk, we, rst_n,
    input [4: 0]    rs1, rs2, wb_addr,    //在取出指令后进入id_stage，通过指令rs,rt寄存器地址索引
    input [31: 0]   wdata,    //wb_addr和wb_data分别为写回write_back的索引地址和待写回的数据

    output [31: 0]   output_data_1, output_data_2        //源寄存器保存的数据信息
);

    reg [31: 0] datareg[31: 0];

    initial begin
        $readmemb(`GPRS_FILE_PATH, datareg);
    end

    always@ (posedge clk) begin
        if(!rst_n) begin
//            rdata1 <= 32'h00000000;          //reg wire = <= conflict
//            rdata2 <= 32'h00000000;        //在always@中，无法对wire进行赋值，无论 =, <=
        end                                    //只能对reg进行阻塞或非阻塞赋值
        else begin
            if(we) begin
                datareg[waddr] <= wdata;
            end
        end
    end

    assign rdata1 = datareg[raddr1];        //assign实际上是把reg的值给到wire类型信号上
    assign rdata2 = datareg[raddr2];

endmodule
```

pc.v　　本质上是一个寄存器，用于提供下一条执行指令的地址

```
module pc(
    input   [31 :0]     new_addr,
    input               clk,
    input               rst_n,
    input               pc_en,

    output reg [31: 0]  output_pc           //将module中输出直接定义为reg，可以在always中赋值
);

    always@ (posedge clk) begin
        if(!rst_n)
            output_pc <= 0 - 4;              //output_pc = 0xfffffffc
        else            //减4是因为默认(pc_add为组合逻辑)第一次pc+4后取指，这样刚好第一条地址为0
            output_pc <= (pc_en == 1)? new_addr : output_pc;
    end         //若pc使能，则令output_pc = new_addr，否则pc值保持不变

endmodule
```

add.v　　加法器，用于地址的计算

```
module pcadd(
    input   [31: 0] index1, index2,

    output  [31: 0] result
);

    assign  result = index1 + index2;

endmodule
```

pcsel.v　　三选二选择器

```
module pcsel(
    input   [31: 0]     jmp_addr,       //分支指令跳转目标地址
    input   [31: 0]     pc_addr,        //顺序执行PC的地址（PC+4）
    input               pc_sel,         //控制选择信号  由control_unit提供

    output  [31: 0]     next_addr
```

```
);

    assign next_addr = (pc_sel == 1)? jmp_addr : pc_addr;
                                    //若pc_sel为1，则执行跳转后的指令，若无效，则顺序执行
endmodule
```

npc.v　　和pc.v功能一致(寄存器)，用来暂存pc+4后的地址

```
module npc(          //对于跳转指令，需要PC作为基址，因此将npc送入后端计算单元
    input   [31 :0]     input_addr,
    input               clk,
    input               rst_n,
    input               npc_en,

    output reg [31: 0]  output_addr
);

    always@ (posedge clk) begin
        if(!rst_n)
            output_addr <= 0;
        else
            output_addr <= (npc_en == 1)? input_addr : output_addr;
    end

endmodule
```

instreg.v　　本质上是个寄存器，存放正要执行的指令

```
module instreg(
    input   [31: 0]     input_inst,
    input               ir_en,
    input               clk,
    input               rst_n,

    output reg  [31: 0] output_inst
);

    always@ (posedge clk) begin
        if(!rst_n)
            output_inst <= 32'h00000000;            //复位时存储nop指令
        else
            output_inst <= (ir_en == 1)? input_inst : output_inst;
```

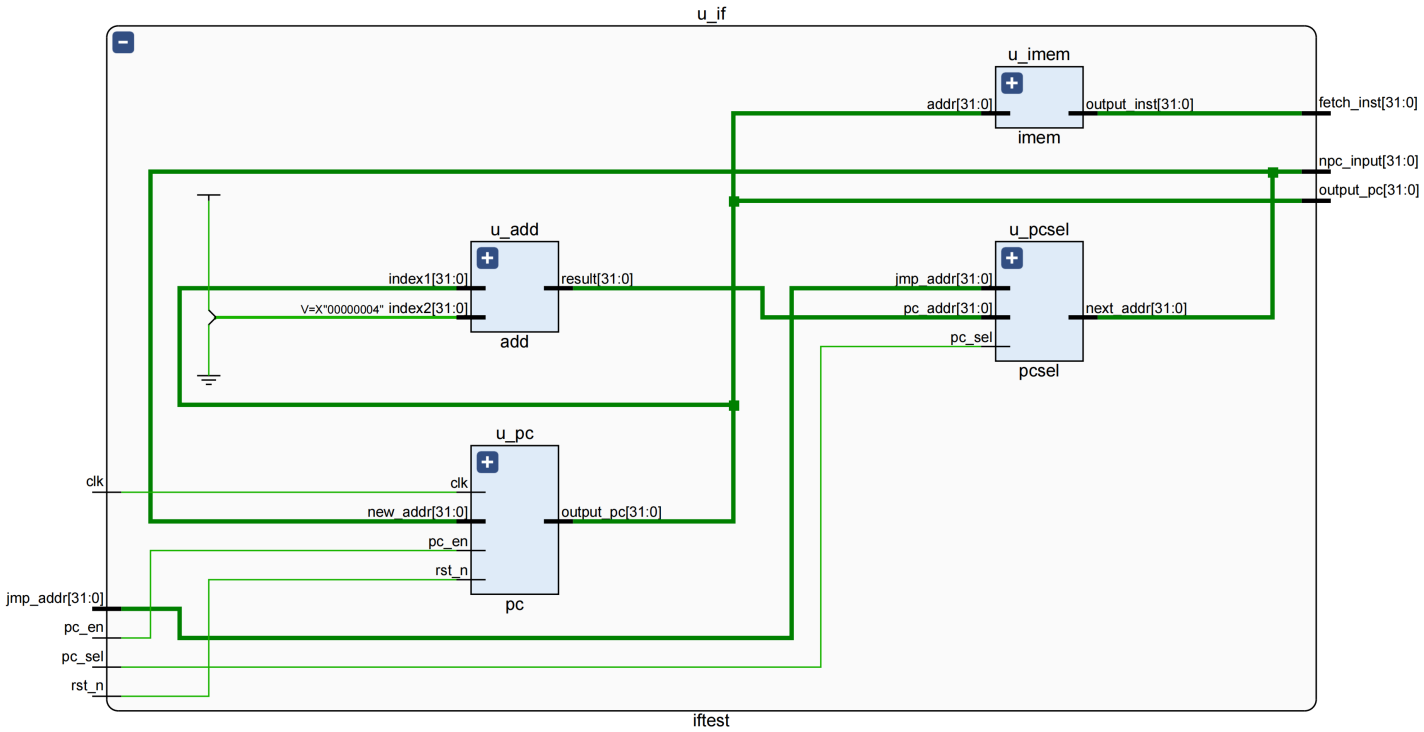
```
    end

endmodule
```

iftest.v 取指模块

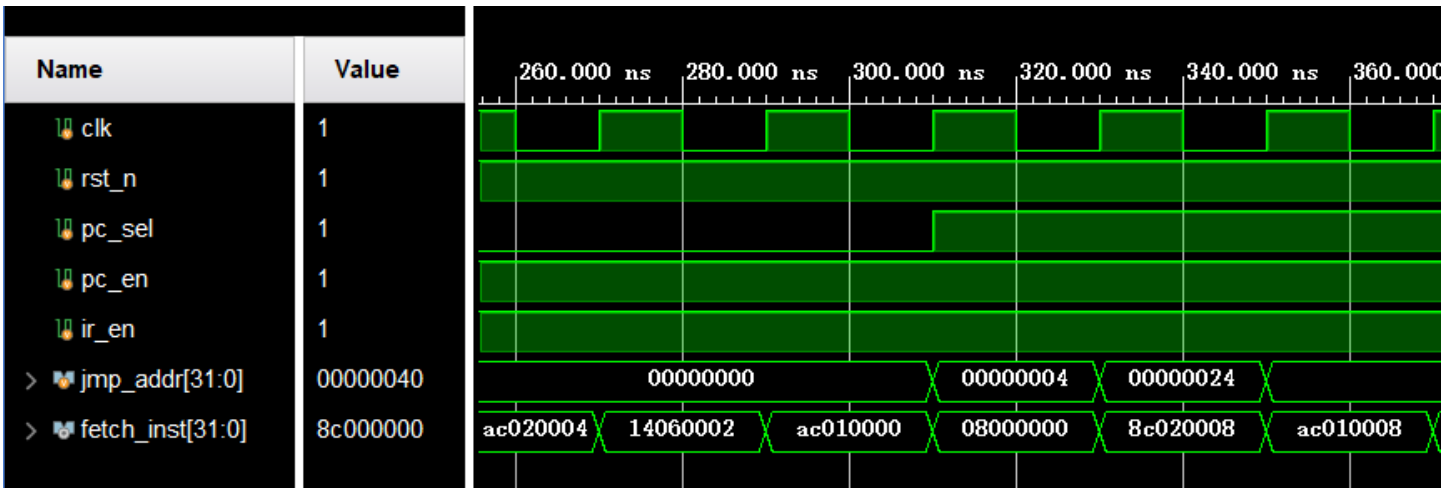
```
`timescale 1ns / 1ps

module iftest(                          //fetch instruction test
    input           clk, rst_n, pc_sel, pc_en, ir_en,     //pc使能、pc选择器使能，指令寄存器使能
    input   [31: 0] jmp_addr,                             //跳转地址

    output  [31: 0] fetch_inst, npc_input, output_pc      //查看取得指令是否正确
);

    wire    [31: 0] new_addr;                             //在顶层模块化多个module                          //用wire定义模块之间连接的信号线
    wire    [31: 0] output_inst;
    wire    [31: 0] pc_addr;
    wire    [31: 0] npc_addr;

    assign npc_input = new_addr;        //将npc定义在if_reg中

    imem u_imem(                        //instruction memory //指令存储器
        .addr(output_pc),                //用pc寄存器输出索引imem中指令数据
        .output_inst(fetch_inst)        //读出mem中指令送入寄存器instreg
    );

    pc u_pc(                            //program count     //pc 实际是个寄存器
        .new_addr(new_addr),            //pc的输入地址，默认是32'fffffffc 即 0-4
        .clk(clk),
        .rst_n(rst_n),
        .pc_en(pc_en),
        .output_pc(output_pc)           //得到的pc直接用以索引imem
    );

    add u_add(                          //实现顺序取指令，PC+4    组合逻辑
        .index1(output_pc),
        .index2(32'h00000004),
        .result(pc_addr)
    );

    pcsel u_pcsel(                        //二选一，跳转地址 or 顺序地址(PC+4)
        .jmp_addr(jmp_addr),
```

```
38          .pc_addr(pc_addr),
39          .pc_sel(pc_sel),
40
41          .next_addr(new_addr)
42      );
43
44  endmodule
```

iftest.RTL



iftest.sim



可以看到在pc_sel未使能前，取指动作一直是顺序进行的，取得的指令内容也是顺序得到的结果。

在pc_sel使能后，pc将由跳转指令得到的跳转目标地址提供，可以看到在输入跳转地址0x00000004后的一拍，取得了该地址对应的指令内容0x8c020008。

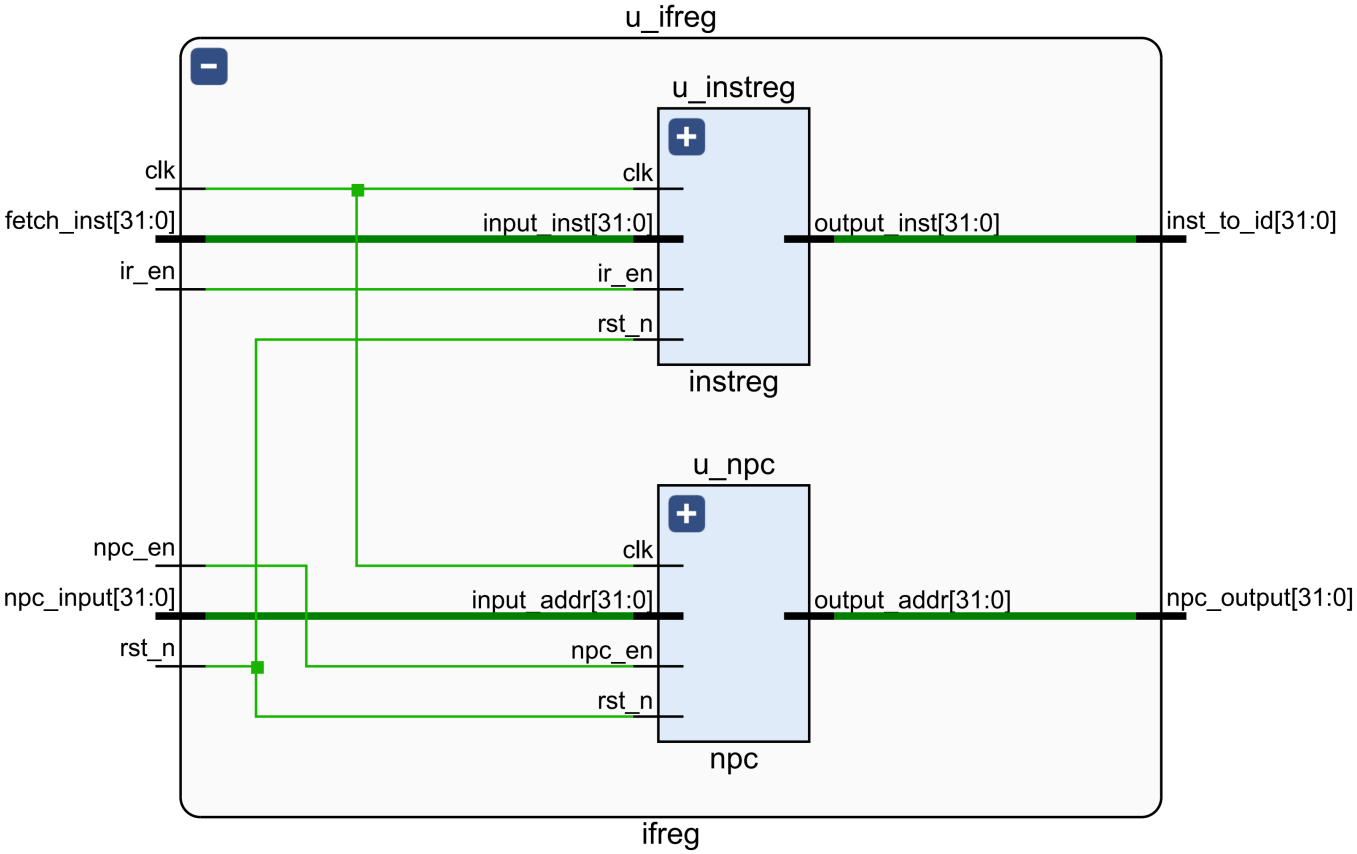
这里引出一个问题： 在pc_sel和jmp_addr使能后，在当前时钟上升沿取得的指令并不是跳转指令取得的，而是顺序的下一条指令，应该是这样的吗？

问题分析：时钟上升沿处指令PC寄存器更新，然而此时索引指令mem取出的指令送入指令寄存器inst_reg后还需要下一拍才能读出。在[1]中通过时钟下降沿实现一个时钟周期的PC索引读出指令信息，CPUsyn中多加一拍。

ifreg.v 取指寄存器

```
module ifreg(
    input   [31: 0] fetch_inst, npc_input,
    input           clk, rst_n, ir_en, npc_en,

    output  [31: 0] inst_to_id,
    output  [31: 0] npc_output
);

    instreg u_instreg(                  //寄存指令
        .input_inst(fetch_inst),
        .ir_en(ir_en),
        .clk(clk),
        .rst_n(rst_n),

        .output_inst(inst_to_id)
    );

    npc u_npc(                          //寄存PC
        .input_addr(npc_input),
        .clk(clk),
        .rst_n(rst_n),
        .npc_en(npc_en),

        .output_addr(npc_output)
    );

endmodule
```

ifreg.RTL



registers.v　　与regfile.v文件没有区别

```
1  `include "./cpu.vh"
2
3  module registers(
4      input   [4: 0]  rs1, rs2, rd,              #用指令中源寄存器rs，目标寄存器rd表示
   地址
5      input   [31: 0] wb_data,
6      input           clk, rst_n, we,
7
8      output  [31: 0] output_data_1, output_data_2
9  );
10
11     reg [31: 0] datareg[31: 0];
12
13     assign output_data_1 = datareg[rs1];
14     assign output_data_2 = datareg[rs2];
15
16     initial begin
17         $readmemb(`GPRS_FILE_PATH, datareg);     #唯一区别在于这里用文件为datareg赋
   初值
18     end
19
```

```
    always@ (posedge clk) begin
        if(!rst_n) begin
            // output_data_1 = 32'h00000000;
        end
        else begin
            if(we == 1)
                datareg[rd] <= wb_data;
        end
    end

endmodule
```

extender.v　　对分支指令进行位拓展

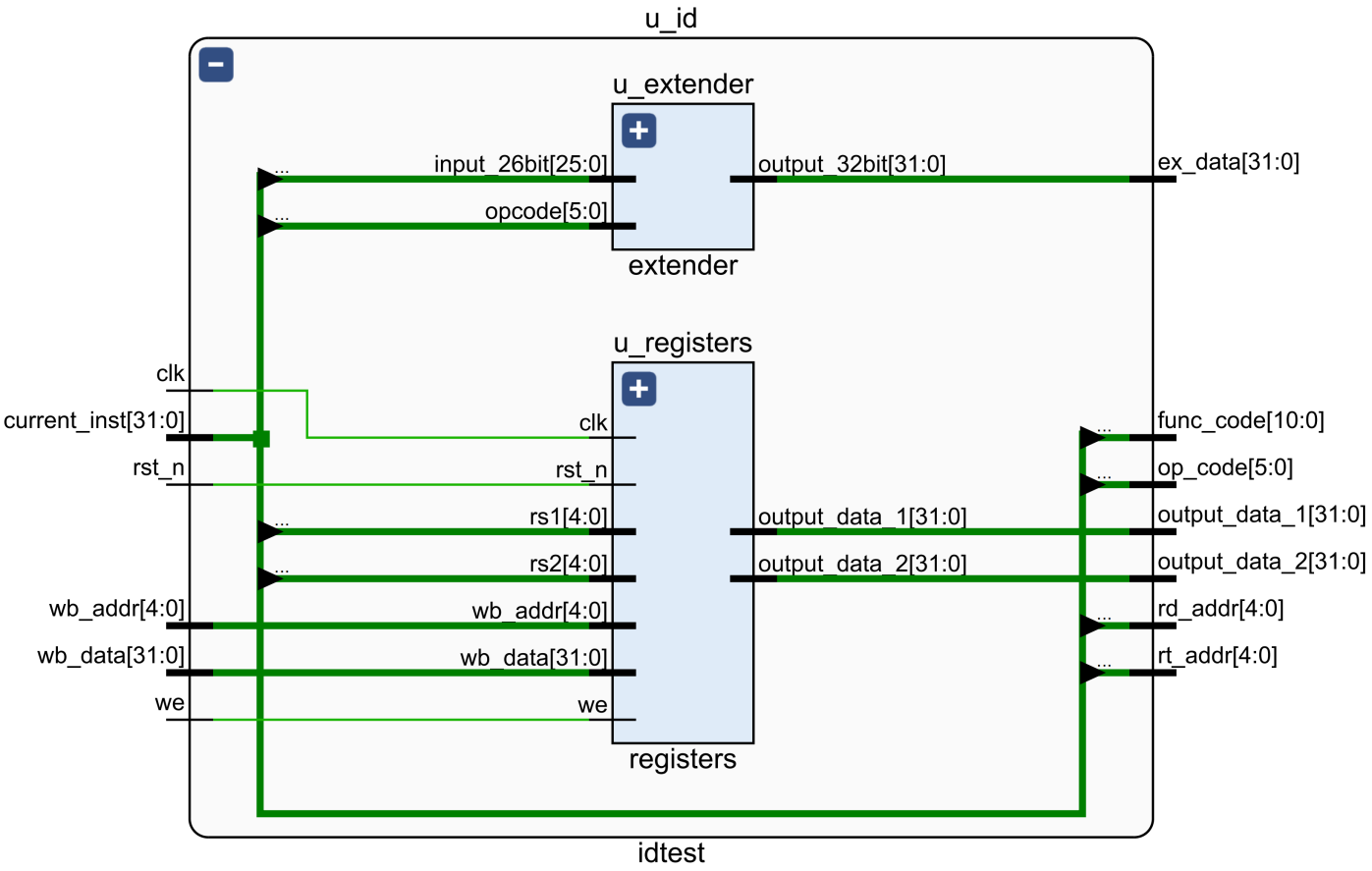
```
`include "./cpu.vh"

module extender(
    input   [5: 0]  opcode,
    input   [25: 0] input_26bit,        //jmp inst_index    //jmp指令的低26位

    output  [31: 0] output_32bit                            //拓展为32位
);

    assign output_32bit[31: 0] = (opcode == `J)? { {6{input_26bit[25]}}, input_26bit[25: 0] }
                                                : { {16{input_26bit[15]}}, input_26bit[15: 0]};
                                                    //lw,sw的偏移地址为低16位
endmodule
```

idtest.v　译码模块

```
module idtest(
    input           clk, rst_n, we,
    input   [31: 0] current_inst, wb_data,        //寄存器堆需要的写回信号(使能、地址、数据)
    input   [4: 0]  wb_addr,                      //待被分解的指令

    output  [31: 0] output_data_1, output_data_2, ex_data,  //ex_data:被拓展的立即数
    output  [4: 0]  rt_addr, rd_addr,
```

```
    output  [5: 0]  op_code,
    output  [10: 0] func_code                   //操作码送入控制单元识别当前指令
);

    wire    [31: 0]   imm_ex;

    wire    [5: 0]    opcode;
    wire    [4: 0]    rs;
    wire    [4: 0]    rt;
    wire    [4: 0]    rd;
    wire    [10: 0]   funcode;
    wire    [25: 0]   imm;

    assign opcode   = current_inst[31: 26];         //按照指令段分解指令
    assign rs       = current_inst[25: 21];
    assign rt       = current_inst[20: 16];
    assign rd       = current_inst[15: 11];
    assign funcode  = current_inst[10: 0 ];
    assign imm      = current_inst[25: 0 ];

    assign rt_addr = rt;
    assign rd_addr = rd;
    assign op_code = opcode;
    assign func_code = funcode;
    assign ex_data = imm_ex;

    registers u_registers(
        .rs1(rs),
        .rs2(rt),
        .wb_addr(wb_addr),
        .wb_data(wb_data),
        .clk(clk),
        .rst_n(rst_n),
        .we(we),
        .output_data_1(output_data_1),    //rs寄存器读出的数据
        .output_data_2(output_data_2)    //rt寄存器读出的数据
    );

    extender u_extender(        //特殊指令拓展模块
        .opcode(opcode),
        .input_26bit(imm),
        .output_32bit(imm_ex)
    );

endmodule
```

idtest.RTL



dataholder.v　　寄存器

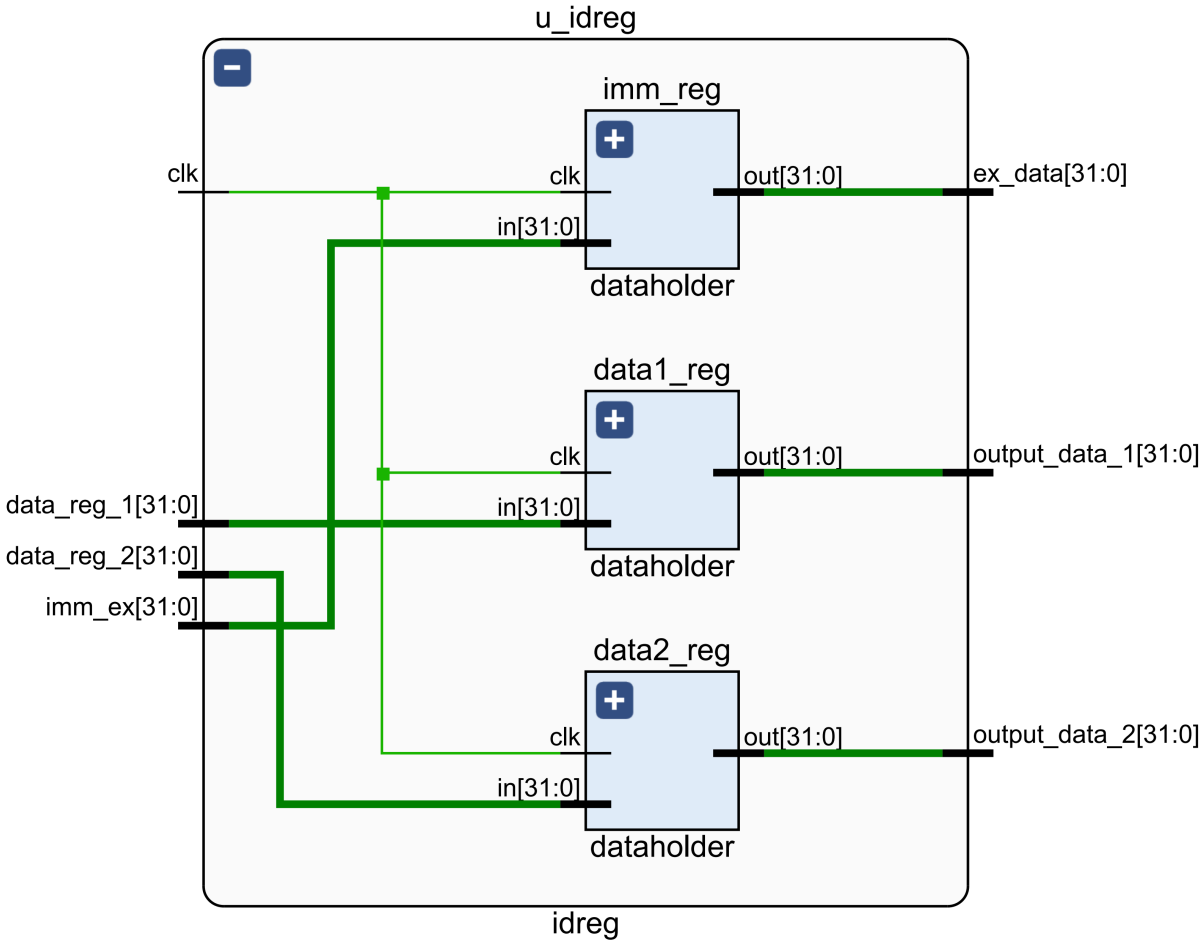
```
1  module dataholder(
2      input   [31: 0]     in,
3      input               clk,
4
5      output reg  [31: 0] out
6  );
7
8      always@(posedge clk) begin
9          out <= in;
10     end
11
12 endmodule
```

idreg.v 译码寄存器

```
1  module idreg(
```

```
    input           clk,
    input   [31: 0] data_reg_1, data_reg_2, imm_ex,

    output  [31: 0] output_data_1, output_data_2, ex_data
);

    dataholder data1_reg(
        .in(data_reg_1),
        .clk(clk),
        .out(output_data_1)   //rs寄存器读出的数据
    );

    dataholder data2_reg(
        .in(data_reg_2),
        .clk(clk),
        .out(output_data_2)    //rt寄存器读出的数据
    );

    dataholder imm_reg(        //位拓展后的数据
        .in(imm_ex),
        .clk(clk),
        .out(ex_data)
    );

endmodule
```

> **idreg.RTL　三个寄存器，一目了然**



mux32.v　　二选一数据选择器

```
1  module mux32(
2      input   [31: 0] input_data1,
3      input   [31: 0] input_data2,
4      input           sel,
5
6      output   [31: 0] output_data
7  );
8
9      assign output_data = (sel == 1)? input_data1 : input_data2;
10
11 endmodule
```

alu.v　　运算单元

```
1  `include "./cpu.vh"
2
3  module alu(
```

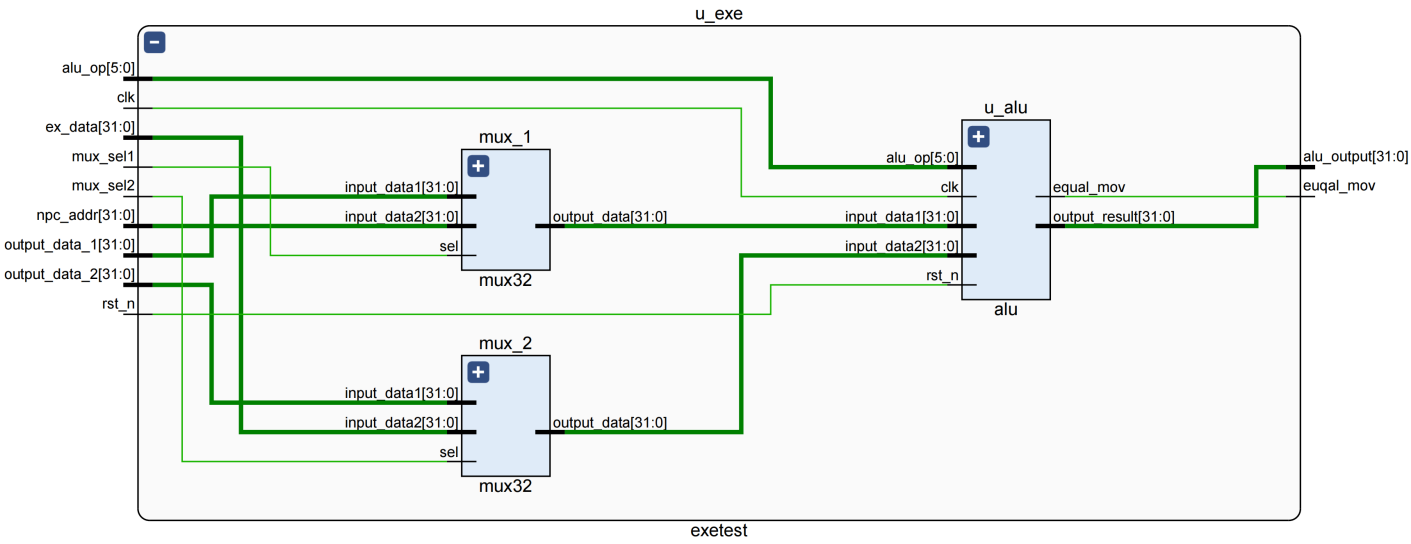
```
    input   [5: 0]  alu_op,            //实际运算码(funcode for alu, opcode for others)
    input   [31: 0] input_data1,    //rs_data or pc
    input   [31: 0] input_data2,    //rt_data or imm_ex
    input           clk,
    input           rst_n,

    output  [31: 0] output_result,
    output          equal_mov        //BNE比较寄存器数据是否相等
);

    reg [31: 0] result;
    reg         equal;
    reg [31: 0] temp;

    assign output_result = result;
    assign equal_mov = equal;

    always@ (posedge clk) begin
        if(!rst_n) begin
            result = 32'b0;
            temp = 32'b0;
        end
        else begin
            case(alu_op)
                `ADD :  result = input_data1 + input_data2;
                `SUB :  result = input_data1 - input_data2;
                `AND :  result = input_data1 & input_data2;
                `OR  :  result = input_data1 | input_data2;
                `XOR :  result = input_data1 ^ input_data2;
                `SLT :  result = (input_data1 < input_data2) ? 32'h00000001 : 32'h00000000;
                `MOVZ:  result = (input_data2 == 0) ? input_data1 : 32'h00000000;
                //以上指令对应的input1, input2分别为rs, rt寄存器读出的数据
                //以下跳转指令, input_data1为npc, input_data2为imm(拓展后)
                `BNE :  begin
                            result = (input_data2 << 2) + input_data1;
                            equal = (input_data1 - input_data2)? 1 : 0;
                        end
                `J   :  begin
                            temp = input_data2 << 2;
                            result = {input_data1[31: 26], temp[25: 0]};
                        end
               default : begin
                                result = input_data1 + input_data2;
                         end
```

```
48              endcase
49          end
50      end
51
52  endmodule
```

exetest.v 计算模块

```
1  module exetest(
2      input            clk, rst_n, mux_sel1, mux_sel2,
3      input   [31: 0]  output_data_1, output_data_2, ex_data, npc_addr,
4      input   [5: 0]   alu_op,
5
6      output           euqal_mov,
7      output  [31: 0]  alu_output
8  );
9
10     wire    [31: 0] mux1_output;
11     wire    [31: 0] mux2_output;
12
13     alu u_alu(
14         .alu_op(alu_op),
15         .input_data1(mux1_output),
16         .input_data2(mux2_output),
17         .clk(clk),
18         .rst_n(rst_n),
19         .output_result(alu_output),
20         .equal_mov(euqal_mov)
21     );
22
23     mux32 mux_1(
24         .input_data1(output_data_1),    //rs    选择进入alu的是rs寄存器数据还是npc
25         .input_data2(npc_addr),
26         .sel(mux_sel1),
27         .output_data(mux1_output)
28     );
29
30     mux32 mux_2(
31         .input_data1(output_data_2),    //rt    选择进入alu的是rt寄存器数据还是imm
32         .input_data2(ex_data),
33         .sel(mux_sel2),
34         .output_data(mux2_output)
35     );
36
37 endmodule
```

exe.RTL



alu_output.v　　运算单元输出结果寄存器

```
module alu_output(
    input   [31: 0]     input_data,
    input               clk,
    input               rst_n,

    output  reg [31: 0] output_data    //可以不定义为reg
);
    //在这里定义中间reg变量result    reg [31: 0] result
    //令assign output_data = result即可
    always@ (clk)   begin
        if(!rst_n)
            output_data = 32'h00000000;
        else
            output_data = input_data;
    end

endmodule
```

exereg.v 计算寄存器

```
module exereg(                                  //simply a reg
    input   [31: 0] alu_output,
```

```
    input           clk, rst_n,

    output  [31: 0] exe_out
);

    alu_output u_alu_output(
        .input_data(alu_output),
        .clk(clk),
        .rst_n(rst_n),
        .output_data(exe_out)
    );

endmodule
```

dmem.v　　数据存储器 时序逻辑（不同于指令存储器为组合逻辑）

```
`include "cpu.vh"

module dmem(        //enter addr fetch inst
    input [31: 0]   alu_addr,
    input [31: 0]   data,
    input           we, clk, rst_n,

    output[31: 0]   output_data
);

    reg [31: 0] data_mem[255: 0];       //256×32 inst mem

    initial begin
        $readmemh(`DATA_FILE_PATH, data_mem);        //读32条32bit的数据
    end

    assign output_data = data_mem[alu_addr / 4];

    always@ (posedge clk) begin
        if(!rst_n)
            data_mem[alu_addr / 4] = 32'h00000000;
        else begin
            if(we == 1) begin
                data_mem[alu_addr / 4] <= data;        //store指令将rt对应数据存入
data_mem
            end
        end
    end
```

```
28
29 endmodule
```

matest.v 访存模块

```
1  module matest(
2      input           clk, rst_n, mem_wen,    //sw指令使能写数据存储器
3      input   [31: 0] alu_result, rt_data,
4
5      output  [31: 0] load_data            //lw指令读出的数据
6  );
7
8      dmem u_dmem(
9          .alu_addr(alu_result),      //用以索引需要读出存储器数据的地址  如lw指令的
   imm_ex
10         .data(rt_data),            //store指令写回存储器的数据
11         .we(mem_wen),
12         .clk(clk),
13         .rst_n(rst_n),
14         .output_data(load_data)
15     );
16
17 endmodule
```

mareg.v 访存寄存器

```
1  module memreg(
2      input   [31: 0] data_temp,
3      input           clk,
4
5      output  [31: 0] load_data
6  );
7
8      datareg u_datareg(
9          .input_data(data_temp),
10         .clk(clk),
11         .output_data(load_data)
12     );
13
```

```
14 endmodule
```

wbsel.v　　写回选择

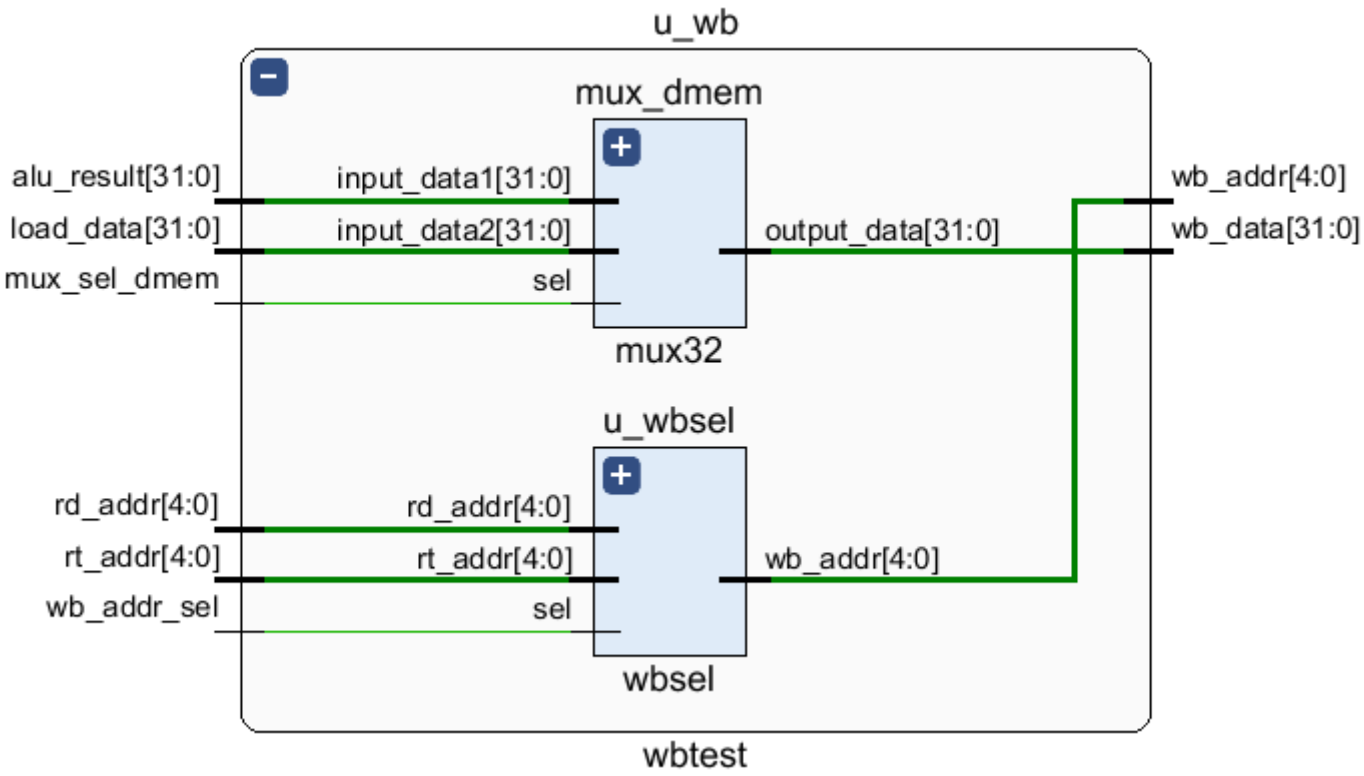
```
1  module wbsel(           //write back select
2      input           sel,
3      input   [4: 0]  rt_addr,            #rt[16: 20]
4      input   [4: 0]  rd_addr,            #rd[11: 15]
5
6      output  [4: 0]  wb_addr
7  );
8
9      assign wb_addr = (sel == 1) ? rt_addr : rd_addr;    #决定写回的寄存器是rt还是
   rd
10
11 endmodule
```

wbtest.v 写回模块

```
1  module wbtest(                   //只有load， 计算指令需要将数据写回寄存器堆
2      input           mux_sel_dmem, wb_addr_sel,    //选择写回的地址、数据
3      input   [31: 0] alu_result, load_data,    //load写回 rt， //alu写回 rd
4      input   [4: 0]  rt_addr, rd_addr,
5
6      output  [31: 0] wb_data,
7      output  [4: 0]  wb_addr
8  );
9
10     mux32 mux_dmem(
11         .input_data1(alu_result),
12         .input_data2(load_data),
13         .sel(mux_sel_dmem),
14
15         .output_data(wb_data)
16     );
17
18     wbsel u_wbsel(
19         .sel(wb_addr_sel),
20         .rt_addr(rt_addr),
21         .rd_addr(rd_addr),
22
```

```
        .wb_addr(wb_addr)
    );

endmodule
```

wbtest.RTL



controlunit.v 控制单元

```
`include "./cpu.vh"

module controlunit(                 //为各个模块输送使能信号，为ALU输送操作码，实现控制的作用
    input   [5: 0]      opcode,
    input   [10: 0]     alu_funcode,
    input               clk, rst_n, equal,
    input   [31: 0]     rt_data,

    output  reg     pc_en,
    output  reg     pc_sel,
    output  reg     npc_en,
    output  reg     ir_en,
    output  reg     reg_wen,
```

```
14      output  reg     mem_wen,
15      output  reg     wb_sel_wen,
16      output  reg     mux1_sel,
17      output  reg     mux2_sel,
18      output  reg     dmem_sel,
19      output  reg     [10: 0] alu_op,
20      output  [9: 0]  pipe_state
21  );
22
23      parameter [5: 0] ALU = 6'b000000;       //alu opcode
24
25      reg [9: 0]  state;
26      reg [9: 0]  next_state;
27
28      assign pipe_state = state;
29
30      parameter [9: 0] state_if    = 10'b0000000001;
31      parameter [9: 0] state_id    = 10'b0000000010;
32      parameter [9: 0] state_ex    = 10'b0000000100;
33      parameter [9: 0] state_ma    = 10'b0000001000;
34      parameter [9: 0] state_wb    = 10'b0000010000;
35      parameter [9: 0] state_ifreg = 10'b0000100000;
36      parameter [9: 0] state_idreg = 10'b0001000000;
37      parameter [9: 0] state_exreg = 10'b0010000000;
38      parameter [9: 0] state_mareg = 10'b0100000000;
39      parameter [9: 0] state_wbreg = 10'b1000000000;
40
41      always@ (posedge clk) begin
42          if(!rst_n) begin            //low reset
43              pc_en       <= 0;
44              pc_sel      <= 0;
45              npc_en      <= 1;
46              ir_en       <= 0;
47              reg_wen     <= 0;
48              mem_wen     <= 0;
49              wb_sel_wen  <= 0;
50              mux1_sel    <= 0;
51              mux2_sel    <= 0;
52              dmem_sel    <= 0;
53              state       <= state_wb;      //default state is writeback
54              next_state  <= state_if;      //clk rise turn state into fetch
55          end
56          else begin
57              state   <=  next_state;
58              case(opcode)        //according to the opcode decide function
59                  ALU  : alu_op <= alu_funcode[10: 0];
60                  `SW  : alu_op <= `SW;
```

```
61                  `LW  : alu_op <= `LW;
62                  `BNE : alu_op <= `BNE;
63                  `J   : alu_op <= `J;
64                  default: alu_op <= `J;
65              endcase
66
67              case(next_state)
68                  state_if    : next_state <= state_ifreg;
69                  state_ifreg : next_state <= state_id;
70                  state_id    : next_state <= state_idreg;
71                  state_idreg : next_state <= state_ex;
72                  state_ex    : next_state <= state_exreg;
73                  state_exreg : next_state <= state_ma;
74                  state_ma    : next_state <= state_mareg;
75                  state_mareg : next_state <= state_wb;
76                  state_wb    : next_state <= state_if;
77              endcase
78              //according to the opcode and funcode decide control signal
79              pc_en       <= (state == state_wb)? 1'b1 : 1'b0;    //pc_reg
80              ir_en       <= (state == state_if)? 1'b1 : 1'b0;
81              mem_wen     <= (state == state_exreg && opcode == `SW)? 1'b1 :
    1'b0;
82              pc_sel      <= (state == state_wb && (opcode == `J || (opcode ==
    `BNE && equal == 1)))? 1'b1 : 1'b0;
83              wb_sel_wen  <= (opcode == `LW)? 1'b1 : 1'b0;
84              mux1_sel    <= (opcode == `J || opcode == `BNE)? 1'b0 : 1'b1;
     //jmp imm or rs_data
85              mux2_sel    <= (opcode == `J || opcode == `BNE || opcode == `LW ||
    opcode == `SW)? 1'b0 : 1'b1;
86              dmem_sel    <= (opcode == `LW)? 1'b0 : 1'b1;
87              if(state == state_ma && (opcode == `LW || (opcode == ALU && !
    (alu_op[5: 0] == `MOVZ && rt_data != 0)))) begin
88                  reg_wen <= 1'b1;
89              end
90              else begin
91                  reg_wen <= 1'b0;
92              end
93          end
94      end
95
96  endmodule
```

CPU.v 顶层文件

```
`timescale 1ns / 1ps

module cpu(
    input           clk,
    input           rst_n,

    output  [31:0]  debug_wb_pc,
    output          debug_wb_rf_wen,
    output  [4:0]   debug_wb_rf_addr,
    output  [31:0]  debug_wb_rf_wdata,
    output  [31:0]  current_inst,
    output  [9: 0]  state
);
    //if
    wire            pc_sel;
    wire            pc_en;
    wire            ir_en;
    wire    [31: 0] jmp_addr;
    wire    [31: 0] fetch_inst;
    wire    [31: 0] npc_input;
    wire    [31: 0] output_pc;
    wire    [31: 0] inst_to_id;
    wire    [31: 0] npc_output;
    //id
    wire            reg_wen;
    wire            npc_en;
    wire    [31: 0] wb_data;
    wire    [31: 0] pc_sel_out;
    wire    [4: 0]  wb_addr;
    wire    [31: 0] reg_output_data_1;
    wire    [31: 0] reg_output_data_2;
    wire    [31: 0] extend_data;
    wire    [4: 0]  rt_addr;
    wire    [4: 0]  rd_addr;
    wire    [5: 0]  op_code;
    wire    [10: 0] func_code;
    wire    [31: 0] id_output_data_1;
    wire    [31: 0] id_output_data_2;
    wire    [31: 0] ex_data;
    //exe
    wire            mux_sel1;
    wire            mux_sel2;
    wire    [31: 0] alu_output_1;
    wire    [5: 0]  alu_op;
    wire            equal;
    wire    [31: 0] alu_out;
    wire    [31: 0] exe_out;
```

```
48      //ma
49      wire            mem_wen;
50      wire            mux_sel_dmem;
51      wire    [31: 0] ma_data;
52      wire    [31: 0] load_data;
53      //wb
54      wire            wb_sel_wen;
55
56      assign debug_wb_pc = output_pc;
57      assign debug_wb_rf_wen = reg_wen;
58      assign debug_wb_rf_addr = wb_addr;
59      assign debug_wb_rf_wdata = wb_data;
60      assign current_inst = fetch_inst;
61
62      //if-ifreg-id-idreg-exe-exereg-mem-memreg-wb
63
64      iftest u_if(
65          .clk(clk),
66          .rst_n(rst_n),
67          .pc_sel(pc_sel),
68          .pc_en(pc_en),
69          .jmp_addr(alu_out),
70
71          .fetch_inst(fetch_inst),
72          .npc_input(npc_input),
73          .output_pc(output_pc)
74      );
75
76      ifreg u_ifreg(
77          .fetch_inst(fetch_inst),
78          .npc_input(npc_input),
79          .clk(clk),
80          .rst_n(rst_n),
81          .ir_en(ir_en),
82          .npc_en(npc_en),
83
84          .inst_to_id(inst_to_id),
85          .npc_output(npc_output)
86      );
87
88      idtest u_id(
89          .clk(clk),
90          .rst_n(rst_n),
91          .we(reg_wen),         //reg write enable
92          .current_inst(inst_to_id),
93          .wb_data(wb_data),
94          .wb_addr(wb_addr),
```

```
 95
 96         .output_data_1(reg_output_data_1),
 97         .output_data_2(reg_output_data_2),
 98         .ex_data(extend_data),
 99         .rt_addr(rt_addr),
100         .rd_addr(rd_addr),
101         .op_code(op_code),
102         .func_code(func_code)
103     );
104 
105     idreg u_idreg(
106         .clk(clk),
107         .data_reg_1(reg_output_data_1),
108         .data_reg_2(reg_output_data_2),
109         .imm_ex(extend_data),
110 
111         .output_data_1(id_output_data_1),
112         .output_data_2(id_output_data_2),
113         .ex_data(ex_data)
114     );
115 
116     exetest u_exe(
117         .clk(clk),
118         .rst_n(rst_n),
119         .mux_sel1(mux_sel1),
120         .mux_sel2(mux_sel2),
121         .output_data_1(id_output_data_1),
122         .output_data_2(id_output_data_2),
123         .ex_data(ex_data),
124         .npc_addr(npc_output),
125         .alu_op(alu_op),
126 
127         .euqal_mov(equal),
128         .alu_output(alu_out)
129     );
130 
131     exereg u_exereg(
132         .alu_output(alu_out),
133         .clk(clk),
134         .rst_n(rst_n),
135 
136         .exe_out(exe_out)
137     );
138 
139     matest u_memacc(
140         .clk(clk),
141         .rst_n(rst_n),
```

```
142         .mem_wen(mem_wen),
143         .alu_result(exe_out),
144         .rt_data(reg_output_data_2),
145 
146         .load_data(ma_data)
147     );
148 
149     memreg u_memreg(
150         .data_temp(ma_data),
151         .clk(clk),
152 
153         .load_data(load_data)
154     );
155 
156     wbtest u_wb(
157         .mux_sel_dmem(mux_sel_dmem),
158         .wb_addr_sel(wb_sel_wen),
159         .alu_result(exe_out),
160         .load_data(load_data),
161         .rt_addr(rt_addr),
162         .rd_addr(rd_addr),
163 
164         .wb_data(wb_data),
165         .wb_addr(wb_addr)
166     );
167 
168     controlunit u_control(
169         .opcode(op_code),
170         .alu_funcode(func_code),
171         .clk(clk),
172         .rst_n(rst_n),
173         .equal(equal),
174         .rt_data(reg_output_data_2),
175 
176         .pc_en(pc_en),
177         .pc_sel(pc_sel),
178         .npc_en(npc_en),
179         .ir_en(ir_en),
180         .reg_wen(reg_wen),
181         .mem_wen(mem_wen),
182         .wb_sel_wen(wb_sel_wen),
183         .mux1_sel(mux_sel1),
184         .mux2_sel(mux_sel2),
185         .dmem_sel(mux_sel_dmem),
186         .alu_op(alu_op),
187         .pipe_state(state)
188     );
```

```
189
190 endmodule
```

五级流水线处理器设计　CPU_Pipe:

CPU.v　顶层文件

```
 1 module cpu(
 2     input           clk,
 3     input           rst_n,
 4
 5     output  [31:0]  debug_wb_pc,
 6     output          debug_wb_rf_wen,
 7     output  [4:0]   debug_wb_rf_addr,
 8     output  [31:0]  debug_wb_rf_wdata,
 9     output  [31:0]  current_if_inst, current_id_inst, current_exe_inst,
   current_mem_inst,
10     output  [1: 0]  forward_signal_a, forward_signal_b,
11     output          pc_reg_en
12 );
13     //if
14     wire            pc_sel;
15     wire            if_id_wen;
16     wire    [31: 0] jmp_addr;
17     wire    [31: 0] fetch_inst;
18     wire    [31: 0] npc_input;
19     wire    [31: 0] output_pc;
20     wire    [31: 0] inst_to_id;
21     wire    [31: 0] npc_output;
22     wire    [31: 0] npc_output_if;
23     wire    [31: 0] if_pc;
24     //id
25     wire            reg_wen;
26     wire            id_ex_wen;
27     wire    [31: 0] wb_data;
28     wire    [31: 0] pc_sel_out;
29     wire    [4: 0]  wb_addr;
30     wire    [31: 0] reg_output_data_1;
31     wire    [31: 0] reg_output_data_2;
32     wire    [31: 0] extend_data;
33     wire    [5: 0]  op_code;
```

```
34      wire    [10: 0] func_code;
35      wire    [31: 0] id_output_data_1;
36      wire    [31: 0] id_output_data_2;
37      wire    [31: 0] ex_data;
38      wire    [31: 0] id_inst;
39      wire    [31: 0] id_pc;
40      wire    [31: 0] inst_to_exe;
41      wire    [31: 0] npc_output_id;
42      //exe
43      wire            mux_sel1;
44      wire            mux_sel2;
45      wire    [31: 0] alu_output_1;
46      wire    [5: 0]  alu_op;
47      wire            equal;
48      wire            exe_wen;
49      wire    [31: 0] alu_out;
50      wire    [31: 0] exe_out;
51      wire    [31: 0] exe_inst;
52      wire    [31: 0] inst_to_mem;
53      wire    [31: 0] exe_pc;
54      wire    [31: 0] wb_rt;
55      //ma
56      wire            mem_en;             //different from mem_wen
57      wire            mux_sel_dmem;
58      wire    [31: 0] ma_data;
59      wire    [31: 0] load_data;
60      wire    [31: 0] mem_inst;
61      wire    [31: 0] mem_pc;
62      wire    [31: 0] mem_alu_out;
63
64      //wb
65      wire            wb_sel_wen;
66      // forwarding unit
67      wire    [31: 0] mux41_output_1;
68      wire    [31: 0] mux41_output_2;
69      wire            id_ex_rstn;
70      wire            data_hazard_id_ex_rstn;
71      // branch predictor
72      wire            bp_wen;
73      wire            if_id_rstn;
74      wire            logic_hazard_id_ex_rstn;
75      wire            exe_pc_sel;
76      wire            exe_reg_rst_n;
77
78      assign debug_wb_pc = output_pc;
79      assign debug_wb_rf_wen = reg_wen;
80      assign debug_wb_rf_addr = wb_addr;
```

```
81      assign debug_wb_rf_wdata = wb_data;
82      assign current_if_inst = inst_to_id;
83      assign current_id_inst = id_inst;
84      assign current_exe_inst = exe_inst;
85      assign current_mem_inst = mem_inst;
86      assign pc_reg_en = pc_sel;
87
88      iftest u_if(            //取指模块（PC寄存器）
89          .clk(clk),
90          .rst_n(rst_n),
91          .pc_sel(pc_sel),  //need to be modified at bp
92          .pc_en(pc_en),              //输入跳转地址、PC+4 or jmp_addr的控制信号
93          .jmp_addr(jmp_addr),            //from bp_unit
94
95          .fetch_inst(fetch_inst),        //输出由PC索引得到imem中的指令
96          .npc_input(npc_input),          //npc_input提供跳转指令所需NPC
97          .output_pc(output_pc)
98      );
99
100     ifreg u_ifreg(        //取指寄存器
101         .fetch_inst(fetch_inst),
102         .npc_input(npc_input),
103         .pc_input(output_pc),
104         .clk(clk),
105         .rst_n(if_id_rstn),             //复位信号由分支预测单元bp_unit控制
106         .write_en(if_id_wen),
107
108         .inst_to_id(inst_to_id),        //寄存指令、PC值、NPC一拍
109         .npc_output(npc_output_if),
110         .pc_output(if_pc)
111     );
112
113     idtest u_id(            //译码模块，主要是按照rs1,rs2读取寄存器堆中的值
114         .clk(clk),
115         .rst_n(rst_n),
116         .we(reg_wen),       //寄存器堆写使能信号，由访存寄存器按照指令操作码控制
117         .current_inst(inst_to_id), //拆解指令为操作码、源寄存器、目标寄存器地址、功能
    码 or 立即数
118         .wb_data(wb_data),    //写回模块(wb_unit)经mem_reg控制后返回的写数据和写地址
119         .wb_addr(wb_addr),
120
121         .output_data_1(reg_output_data_1),    //寄存器堆读出的数据可以直接送入下一级
122         .output_data_2(reg_output_data_2),    //不需要再在id_reg中寄存一拍
123         .ex_data(ex_data)         //SW、LW、JMP类型指令所需的立即数imm位拓展
124     );                          //注意此处拓展操作为组合逻辑，为保证时序一致，需要再寄
    存一拍
125
```

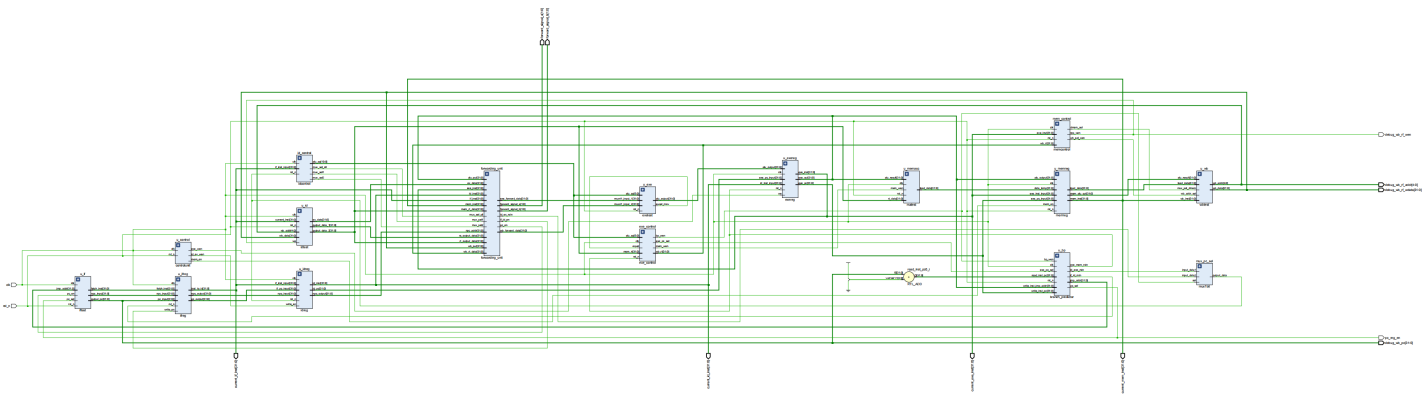
```
126     idreg u_idreg(              //译码寄存器
127         .clk(clk),
128         .rst_n(id_ex_rstn),
129         .write_en(id_ex_wen),     //id_reg复位信号由前递单元、分支预测单元选择控制
130         .if_inst_input(inst_to_id),     //输入if_reg中保存的指令、PC和NPC
131         .if_pc_input(if_pc),
132         .npc_input(npc_output_if),
133 
134         .id_inst(id_inst),
135         .id_pc(id_pc),
136         .npc_output(npc_output_id)
137     );    //注意整个DECODE在一拍下输出指令、PC、NPC、源寄存器保存的数据和拓展后的立即数
138 
139     idcontrol id_control(         //译码控制单元    为ALU提供操作码
140         .if_inst_input(inst_to_id),
141         .clk(clk),
142         .rst_n(id_ex_rstn),
143 
144         .mux_sel_sll(mux_sel_sll),     //mux_sel_sll decide SLL(rt_data) or JMP
145         .mux_sel1(mux_sel1),           //mux_sel1 decide ALU(rs_data) or others
146         .mux_sel2(mux_sel2),           //mux_sel2 decide ALU(rt_data) or
    JMP.LW.SW(extend imm)
147         .alu_op(alu_op)
148     );
149 
150     exetest u_exe(                  //执行单元
151         .rst_n(rst_n),                //操作码由id模块提供，不同与单周期可以用单个控制
    单元直接提供
152         .alu_op(alu_op),              //这样操作保证了在流水线中ALU执行的指令是实时的
153         .mux41_input_1(mux41_output_1),     //输入由前递模块(forwarding unit)提供
154         .mux41_input_2(mux41_output_2),
155 
156         .euqal_mov(equal),            //输出BNE指令所需比较相等信号以及ALU输出
157         .alu_output(alu_out)          //这里的ALU为组合逻辑，保证数据同步发出
158     );
159 
160     exereg u_exereg(            //执行寄存器
161         .alu_output(alu_out),
162         .id_inst_input(id_inst),
163         .exe_pc_input(id_pc),
164         .clk(clk),
165         .rst_n(exe_reg_rst_n),     //复位信号由bp_unit提供
166         .we(exe_wen),
167 
168         .exe_out(exe_out),     //将ALU输出、指令和PC寄存一拍
169         .exe_inst(exe_inst),
170         .exe_pc(exe_pc)
```

```
171     );
172 
173     exe_control exe_control(         //执行控制单元
174         .alu_op(alu_op),    //op_code
175         .clk(clk),
176         .rst_n(exe_reg_rst_n),
177         .mem_rt(reg_output_data_2),    //对MOVZ指令的特殊处理，判断rt_data的值确定是否写回
178         .equal(equal),
179 
180         .mem_wen(mem_wen),         //按照当前处理指令操作码控制数据存储器dmem写使能信号
181         .bp_wen(bp_wen),           //判断如果为分支指令，更新bp_unit中PHT、BTB
182         .exe_pc_sel(exe_pc_sel),  //真实计算出分支指令是否需要发生跳转
183         .wb_rt(wb_rt)             //寄存mem_rt
184     );
185 
186     matest u_memacc(         //访存模块    LW、SW
187         .clk(clk),
188         .rst_n(rst_n),
189         .mem_wen(mem_wen),
190         .alu_result(exe_out),
191         .rt_data(reg_output_data_2),    //SW指令向对应地址存入rt_data
192 
193         .load_data(ma_data)             //LW指令读出DMEM中存储的数据
194     );
195 
196     memreg u_memreg(         //访存寄存器
197         .data_temp(ma_data),
198         .clk(clk),
199         .rst_n(rst_n),
200         .mem_en(mem_en),
201         .exe_inst_input(exe_inst),
202         .exe_pc_input(exe_pc),
203         .alu_output(exe_out),
204 
205         .load_data(load_data),         //寄存DMEM读出数据、指令、PC和ALU计算输出值
206         .mem_inst(mem_inst),
207         .mem_pc(mem_pc),
208         .mem_alu_out(mem_alu_out)
209     );
210 
211     memcontrol mem_control(    //访存控制模块
212         .exe_inst(exe_inst),
213         .clk(clk),
214         .rst_n(rst_n),
215         .wb_rt(wb_rt),
```

```
216
217         .wb_sel_wen(wb_sel_wen),    //控制写回的地址和数据
218         .dmem_sel(mux_sel_dmem),
219         .reg_wen(reg_wen)
220     );
221
222     wbtest u_wb(
223         .mux_sel_dmem(mux_sel_dmem),
224         .wb_addr_sel(wb_sel_wen),
225         .alu_result(mem_alu_out),     //ALU类指令返回计算结果
226         .load_data(load_data),        //LW返回DMEM数据
227         .wb_inst(mem_inst),           //返回rt or rd
228
229         .wb_data(wb_data),
230         .wb_addr(wb_addr)
231     );
232
233     controlunit u_control(            //控制单元
234         .clk(clk),
235         .rst_n(rst_n),
236          //always enable id_exe exe_mem mem_wb reg write control
237          //always disable mem_wb rst
238         .id_ex_wen(id_ex_wen),
239         .exe_wen(exe_wen),
240         .mem_en(mem_en),
241         .pipe_state(state)
242     );
243
244     forwarding_unit forwarding_unit(        //前递单元
245         .if_inst(inst_to_id),
246         .id_inst(id_inst),
247         .exe_inst(exe_inst),
248         .mem_inst(mem_inst),
249         .mem_rt_data(reg_output_data_2),
250         .wb_rt_data(wb_rt),
251         .alu_out(exe_out),
252         .wb_out(wb_data),
253         .npc_addr(npc_output_id),
254         .rs_output_data(reg_output_data_1),
255         .rt_output_data(reg_output_data_2),
256         .ex_data(ex_data),
257         .mux_sel_sll(mux_sel_sll),
258         .mux_sel1(mux_sel1),
259         .mux_sel2(mux_sel2),
260
261         .pc_en(pc_en),
262         .if_id_en(if_id_wen),
```

```
263         .id_ex_rstn(data_hazard_id_ex_rstn),
264         .exe_forward_data(mux41_output_1),
265         .wb_forward_data(mux41_output_2),
266         .forward_signal_a(forward_signal_a),
267         .forward_signal_b(forward_signal_b)
268     );
269 
270     branch_predictor u_bp(              //分支预测单元
271         .clk(clk),
272         .rst_n(rst_n),
273         .bp_wen(bp_wen),
274         .exe_pc_sel(exe_pc_sel),
275         .write_inst_pc(exe_pc),
276         .write_inst_jmp_addr(exe_out),
277         .read_inst_pc(output_pc),
278 
279         .jmp_addr(jmp_addr),
280         .pc_sel(pc_sel),
281         .if_id_rstn(if_id_rstn),
282         .id_exe_rstn(logic_hazard_id_ex_rstn),
283         .exe_mem_rstn(exe_reg_rst_n)
284     );
285 
286     mux1bit mux_pc_sel(
287         .input_data1(logic_hazard_id_ex_rstn),
288         .input_data2(data_hazard_id_ex_rstn),
289         .sel(bp_wen),
290 
291         .output_data(id_ex_rstn)
292     );
293 
294 endmodule
```

CPU.RTL



forwarding_unit.v 前递模块

```
module forwarding_unit(
    input   [31: 0]     if_inst, id_inst, exe_inst, mem_inst,
    input   [31: 0]     mem_rt_data, wb_rt_data,
    input   [31: 0]     alu_out, wb_out,
    input   [31: 0]     npc_addr, rs_output_data, rt_output_data, ex_data,
    input               mux_sel_sll, mux_sel1, mux_sel2,

    output              pc_en, if_id_en, id_ex_rstn,
    output  [31: 0]     exe_forward_data, wb_forward_data,
    output  [1: 0]      forward_signal_a, forward_signal_b
);
    wire    [1: 0]      forward_a, forward_b;
    wire    [31: 0]     mux1_output, mux2_output, mux_sll_out;

    assign forward_signal_a = forward_a;
    assign forward_signal_b = forward_b;

    hazard_detection_unit hazard_detection_unit(
        .if_inst(if_inst),
        .id_inst(id_inst),
        .exe_inst(exe_inst),
        .mem_inst(mem_inst),
        .mem_rt_data(mem_rt_data),
        .wb_rt_data(wb_rt_data),

        .pc_en(pc_en),
        .if_id_en(if_id_en),    //diffrent from id_ex
        .id_ex_rstn(id_ex_rstn),  //clean id_ex_reg to make a nop
        .forward_a(forward_a),
        .forward_b(forward_b)
    );

    mux41 exe_hazard_mux(
        .input_data1(mux1_output),
        .input_data2(alu_out),
        .input_data3(wb_out),
        .input_data4(32'h00000000),
        .mux_sel(forward_a),

        .result(exe_forward_data)
    );

    mux41 mem_hazard_mux(
```

```
        .input_data1(mux2_output),
        .input_data2(alu_out),
        .input_data3(wb_out),
        .input_data4(32'h00000000),
        .mux_sel(forward_b),

        .result(wb_forward_data)
    );

    mux32 mux_sll(
        .input_data1({{27'b0}, id_inst[10: 6]}),    //rs sll need shamt
        .input_data2(npc_addr),
        .sel(mux_sel_sll),      //mux_sel_sll==1 :  out = input1
        .output_data(mux_sll_out)
    );

    mux32 mux_1(
        .input_data1(rs_output_data),    //rs sll need shamt
        .input_data2(mux_sll_out),
        .sel(mux_sel1),
        .output_data(mux1_output)
    );

    mux32 mux_2(
        .input_data1(rt_output_data),    //rt_data or extend_addr
        .input_data2(ex_data),
        .sel(mux_sel2),
        .output_data(mux2_output)
    );

endmodule
```

hazard_detection.v 数据冒险检测模块

```
`include "./cpu.vh"

module hazard_detection_unit(
    input   [31: 0]     if_inst, id_inst, exe_inst, mem_inst,
    input   [31: 0]     mem_rt_data, wb_rt_data,

    output  reg         pc_en, if_id_en, id_ex_rstn,
    output  reg [1: 0]  forward_a, forward_b
);
    wire    [5: 0]  if_op_code;
    wire    [5: 0]  id_op_code;
```

```
12      wire    [5: 0]  exe_op_code;
13      wire    [5: 0]  mem_op_code;
14
15      reg     [4: 0]  if_id_inst_rs;      //if->id
16      reg     [4: 0]  if_id_inst_rt;
17      reg     [4: 0]  if_id_inst_rd;
18      reg     [4: 0]  id_exe_inst_rs;      //id->exe
19      reg     [4: 0]  id_exe_inst_rt;
20      reg     [4: 0]  id_exe_inst_rd;
21      reg     [4: 0]  exe_mem_inst_rs;    //exe->mem
22      reg     [4: 0]  exe_mem_inst_rt;
23      reg     [4: 0]  exe_mem_inst_rd;
24      reg     [4: 0]  mem_wb_inst_rs;    //mem->wb
25      reg     [4: 0]  mem_wb_inst_rt;
26      reg     [4: 0]  mem_wb_inst_rd;
27
28      reg             mem_reg_write;
29      reg             wb_reg_write;
30      reg             hazard_exe_a;
31      reg             hazard_exe_b;
32
33      assign  if_op_code = if_inst[31: 26];
34      assign  id_op_code = id_inst[31: 26];
35      assign  exe_op_code = exe_inst[31: 26];
36      assign  mem_op_code = mem_inst[31: 26];
37
38      always@ (*) begin
39          pc_en       <=  1'b1;
40          forward_a   <=  2'b00;
41          forward_b   <=  2'b00;
42          if_id_en    <=  1'b1;
43          id_ex_rstn  <=  1'b1;
44
45          if_id_inst_rs  <= if_inst[25: 21];
46          if_id_inst_rt  <= if_inst[20: 16];
47          if_id_inst_rd  <= if_inst[15: 11];
48          id_exe_inst_rs  <= id_inst[25: 21];
49          id_exe_inst_rt  <= id_inst[20: 16];
50          id_exe_inst_rd  <= id_inst[15: 11];
51          exe_mem_inst_rs <= exe_inst[25: 21];
52          exe_mem_inst_rt <= exe_inst[20: 16];
53          exe_mem_inst_rd <= exe_inst[15: 11];
54          mem_wb_inst_rs  <= mem_inst[25: 21];
55          mem_wb_inst_rt  <= mem_inst[20: 16];
56          mem_wb_inst_rd  <= mem_inst[15: 11];
57
```

```
58          if( (id_op_code == `LW) && (if_op_code != `LW) && ((id_exe_inst_rt ==
    if_id_inst_rs) || (id_exe_inst_rt == if_id_inst_rt)) ) begin
59              pc_en       <=  1'b0;    //if inst is LW and (id/ex.rd == if/id.rs
    or id/ex.rd == if/id.rt) stop pipeline
60              if_id_en    <=  1'b0;    //for LW rd is rt actually
61              id_ex_rstn  <=  1'b0;
62          end
63          // LW, MOVZ enable mem_reg_write, wb_reg_write
64          mem_reg_write   <= (exe_op_code == `LW) || ((exe_op_code == `ALU) &&
    ((exe_inst[10:0] == 11'b00000_001010 && mem_rt_data != 32'h0) ||
    (exe_inst[10:0] != 11'b00000_001010)));
65          wb_reg_write    <= (mem_op_code == `LW) || ((mem_op_code == `ALU) &&
    ((mem_inst[10:0] == 11'b00000_001010 && wb_rt_data  != 32'h0) ||
    (mem_inst[10:0] != 11'b00000_001010)));
66          //if opcode not ALU or LW or MOVZ, no data hazard
67          if ((id_op_code != `ALU) || (mem_reg_write == 1'b0) || (wb_reg_write
    == 1'b0)) begin
68              forward_a <= 2'b00;
69              forward_b <= 2'b00;
70          end
71
72          //ex hazard     for a : exe->mem rd == id->exe rs  a = 01  control
    alu_rs_input == last_alu_output
73          if(mem_reg_write && (exe_mem_inst_rd != 0) && (exe_mem_inst_rd ==
    id_exe_inst_rs)) begin
74              forward_a   <=  2'b01;
75          end
76          //              for b : exe->mem rd == id->exe rt
77          if(mem_reg_write && (exe_mem_inst_rd != 0) && (exe_mem_inst_rd ==
    id_exe_inst_rt)) begin
78              forward_b   <=  2'b01;
79          end
80
81          hazard_exe_a <= mem_reg_write && (exe_mem_inst_rd != 0) &&
    (exe_mem_inst_rd == id_exe_inst_rs);
82          hazard_exe_b <= mem_reg_write && (exe_mem_inst_rd != 0) &&
    (exe_mem_inst_rd == id_exe_inst_rt);
83
84          //mem hazard   for a : mem->wb rd == id->exe rs  a = 01  control
    alu_rs_input == last_wb_output
85          if(wb_reg_write && (mem_wb_inst_rd != 0) && (mem_wb_inst_rd ==
    id_exe_inst_rs) && !hazard_exe_a) begin
86              forward_a   <=  2'b10;      //handle exe hazard before
87          end
88          if(wb_reg_write && (mem_wb_inst_rd != 0) && (mem_wb_inst_rd ==
    id_exe_inst_rt) && !hazard_exe_b) begin
89              forward_b   <=  2'b10;
```

```
90          end
91
92          //concern load-use    mem hazard
93          if(wb_reg_write && (mem_wb_inst_rt != 0) && (mem_wb_inst_rt ==
    id_exe_inst_rs) && (mem_op_code == `LW))
94              forward_a   <=  2'b10;
95          if(wb_reg_write && (mem_wb_inst_rt != 0) && (mem_wb_inst_rt ==
    id_exe_inst_rt) && (mem_op_code == `LW))
96              forward_b   <=  2'b10;
97
98      end
99
100 endmodule
```

branch_predictor.v 分支预测模块

```
1  module branch_predictor(
2      input               clk, rst_n, bp_wen, exe_pc_sel,
3      input   [31: 0]     write_inst_pc, write_inst_jmp_addr,
4      input   [31: 0]     read_inst_pc,
5
6      output  [31: 0]     jmp_addr,
7      output  reg         if_id_rstn, id_exe_rstn, exe_mem_rstn,
8      output              pc_sel
9  );
10     //PHT
11     reg [1: 0]  pht_state[63: 0];       //11, 10 taken    01, 00 not taken
12     //BTB
13     reg [31: 0] btb_pc[63: 0];
14     reg [31: 0] btb_jmp_addr[63: 0];
15     reg         btb_valid[63: 0];
16     reg         bp_jump;
17     reg         bp_pc_sel;
18     reg         pc_sel_choose;
19     reg [31: 0] btb_jmp;
20     wire [31: 0] read_pc;
21
22     genvar i;
23     generate
24         for(i=0; i<64; i=i+1) begin
25             initial begin
26                 btb_valid[i]    <=  1'b0;
27                 btb_pc[i]       <=  32'h0000_0000;
```

```
28                  btb_jmp_addr[i] <=  32'h0000_0000;
29                  pht_state[i]    <=  2'b11;
30              end
31          end
32      endgenerate
33
34      always@ (posedge clk or posedge bp_wen) begin
35          if(!rst_n) begin
36              btb_jmp         <=  32'h0000_0000;
37              bp_pc_sel       <=  1'b0;
38              pc_sel_choose   <=  1'b1;
39              bp_jump         <=  1'b0;
40          end
41          else begin
42              if(btb_valid[read_pc[7: 2]] == 1'b1) begin
43                  pc_sel_choose = 1'b0;
44                  btb_jmp = btb_jmp_addr[read_pc[7: 2]];
45                  if((pht_state[write_inst_pc[7: 2]] == 2'b00) ||
    (pht_state[write_inst_pc[7: 2]] == 2'b01))
46                      bp_pc_sel <= 1'b0;
47                  else
48                      bp_pc_sel <= 1'b1;
49              end
50              else begin
51                  pc_sel_choose = 1'b1;
52                  bp_pc_sel <= 1'b0;
53              end
54
55              if(bp_wen) begin
56                  if(write_inst_jmp_addr == (read_pc - 12))
57                      pc_sel_choose = 1'b0;
58
59                  if(write_inst_jmp_addr != (write_inst_pc + 4))
60                      bp_jump     <=  1'b1;
61                  else
62                      bp_jump     <=  1'b0;
63
64                  if((pht_state[write_inst_pc[7: 2]] == 2'b11) && bp_jump) begin
65                      pht_state[write_inst_pc[7: 2]] = 2'b11;
66                  end
67                  if((pht_state[write_inst_pc[7: 2]] == 2'b11) && !bp_jump) begin
68                      pht_state[write_inst_pc[7: 2]] = 2'b10;
69                  end
70                  if((pht_state[write_inst_pc[7: 2]] == 2'b10) && bp_jump) begin
71                      pht_state[write_inst_pc[7: 2]] = 2'b11;
72                  end
73                  if((pht_state[write_inst_pc[7: 2]] == 2'b10) && !bp_jump) begin
```

```
74                      pht_state[write_inst_pc[7: 2]] = 2'b01;
75                  end
76                  if((pht_state[write_inst_pc[7: 2]] == 2'b01) && bp_jump) begin
77                      pht_state[write_inst_pc[7: 2]] = 2'b10;
78                  end
79                  if((pht_state[write_inst_pc[7: 2]] == 2'b01) && !bp_jump) begin
80                      pht_state[write_inst_pc[7: 2]] = 2'b00;
81                  end
82                  if((pht_state[write_inst_pc[7: 2]] == 2'b00) && bp_jump) begin
83                      pht_state[write_inst_pc[7: 2]] = 2'b01;
84                  end
85                  if((pht_state[write_inst_pc[7: 2]] == 2'b00) && !bp_jump) begin
86                      pht_state[write_inst_pc[7: 2]] = 2'b00;
87                  end
88
89                  if(bp_jump) begin
90                      btb_valid[write_inst_pc[7: 2]]    <=  1'b1;
91                      btb_pc[write_inst_pc[7: 2]]       <=  write_inst_pc;
92                      btb_jmp_addr[write_inst_pc[7: 2]] <=  write_inst_jmp_addr;

93                  end
94              end
95          end
96      end
97
98      always@ (*) begin
99          if(!rst_n) begin
100             if_id_rstn      <=  1'b0;
101             id_exe_rstn     <=  1'b0;
102             exe_mem_rstn    <=  1'b0;
103         end
104         else begin
105             if(pc_sel_choose && exe_pc_sel && (write_inst_jmp_addr != (read_pc
    - 12))) begin
106                 if_id_rstn      =  1'b0;
107                 id_exe_rstn     =  1'b0;
108                 exe_mem_rstn    =  1'b0;
109             end
110             else begin
111                 if_id_rstn      =  1'b1;
112                 id_exe_rstn     =  1'b1;
113                 exe_mem_rstn    =  1'b1;
114             end
115         end
116     end
117
118     mux1bit mux_pc_sel(
```

```
119         .input_data1(exe_pc_sel),
120         .input_data2(bp_pc_sel),
121         .sel(pc_sel_choose),
122 
123         .output_data(pc_sel)
124     );
125 
126     mux32 mux_jmp_addr(
127         .input_data1(write_inst_jmp_addr),
128         .input_data2(btb_jmp),
129         .sel(pc_sel_choose),
130 
131         .output_data(jmp_addr)
132     );
133 
134     add u_add(
135         .index1(read_inst_pc),
136         .index2(32'h00000004),
137 
138         .result(read_pc)
139     );
140 
141 endmodule
```