同濟大學
TONGJI UNIVERSITY

计算机系统结构课程实验

总结报告

实验题目：简单的流水线 CPU 设计与性能分析

学号：2151769

姓名：吕博文

指导教师：陆有军

日期：2023.11.25

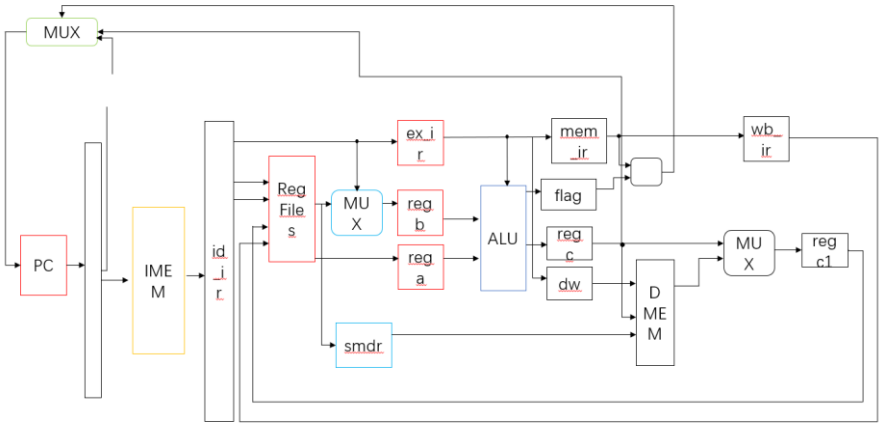
一、 实验环境部署与硬件配置说明

本次实验使用的实验环境是 Vivado 软件，Modelsim 软件，具体配置以及建立项目编写文件的格式和上学期完成 CPU 时的部署配置基本一致，这里不做过多说明。

二、 实验的总体结构

1、 静态流水线的总体结构

本次实验实现的是简单流水线 CPU 的设计，我们根据实验指导书中给出的流水线功能部件介绍，画出简单流水线 CPU 的数据通路以及总体结构如下所示：



三、 总体架构部件的解释说明

1、 静态流水线总体结构部件的解释说明

本次实验实现的 CPU 设计流水线工作主要分为 5 个阶段：
取址（IF）→译码（ID）→执行（EX）→访存（MEM）→回写（WB）
相比于上学期的 CPU31 条和 CPU54 条，其他模块如 ALU.v，MUL.v，DIV.v，PCReg.v 等没有过多的变化，主要发生变换的是 cpu 模块，不同于简单 CPU 单周期一条指令的执行，简单流水线 CPU 要求我们有以上提到的基本的 5 个步骤并解决一定的数据冲突，竞争冒险等行为。
下面分别对 5 个步骤进行具体模块解析：
（1） IF 阶段：

```
//IF阶段
// 声明程序计数器（if_pc）和指令（if_inst）的线路
wire[31:0] if_pc;
wire[31:0] if_inst;
```

```
//IF阶段赋值
assign if_pc = pc_out;
assign if_inst = IM_inst;
```

总的来说，IF 阶段中主要实现的功能是：从内存中取出指令并更新程序计数器的过程，如果涉及到的指令为条件分支指令，还需要提前计算出条件分支转移的地址。

（2） ID 阶段

```
// ID 阶段
// 用于符号扩展和零扩展数据的线路，以及控制信号
wire[31:0] if_sext_16, id_uext_16, id_uext_5, id_sext_18;
wire[4:0] id_rsc, id_rtc;
wire [31:0] id_rs, id_rt;
wire id_pc_ena;
wire[31:0] id_pc_in, id_npc;
wire[31:0] id_alu_a, id_alu_b;
wire[4:0] id_cp0_raddr;
wire[31:0] id_cp0_rdata, id_cp0_epcout, id_cp0_status;
wire[31:0] id_pass_data;
```

```
// 设置 ID 阶段的程序计数器使能信号为 1
assign id_pc_ena = 1'b1;
// 将 ID 阶段的下一个程序计数器连接到当前程序计数器
assign id_npc = if_id_npc;

// 将 ID 阶段的寄存器编号连接到相应的目的寄存器
assign id_rsc = idRs;
assign id_rtc = idRt;
// 将 ID 阶段的寄存器数据连接到相应的目的寄存器
assign id_rs = rs;
assign id_rt = rt;
```

ID 级主要对指令进行译码并生成控制信号，具体生成控制信号的代码见附录。

（3） EX 阶段

```
//ID-EX
reg[31:0] id_ex_alua,id_ex_alub;
reg[31:0] id_ex_pass_data;
reg[31:0] id_ex_inst;

//EX
// 执行阶段（EX）的算术运算单元 A、B、和输出 O 的线路
wire[31:0] ex_alua, ex_alub, ex_aluo;
// 执行阶段的乘法器输入 A、B、无符号输入 A、B、以及输出 Z 的线路
wire[31:0] ex_mula, ex_mulb, ex_multua, ex_multub;
wire[63:0] ex_mulz, ex_multuz;
// 执行阶段的除法器和除法器（无符号）控制信号
wire ex_div_start, ex_div_busy, ex_divu_start, ex_divu_busy;
// 执行阶段的除法器输入：被除数、除数、商和余数（有符号）
wire[31:0] ex_div_dividend, ex_div_divisor, ex_div_q, ex_div_r;
// 执行阶段的除法器输入：被除数、除数、商和余数（无符号）
wire[31:0] ex_divu_dividend, ex_divu_divisor, ex_divu_q, ex_divu_r
```

```
//EX
reg div_start_reg,divu_start_reg;

always@(*)begin
    divu_start_reg = 0;
    div_start_reg = 0;
    if(exOpDivu&&!stall[`STAGE_EX]&&!divuBusy)begin
        divu_start_reg = 1;
    end
    if(exOpDiv&&!stall[`STAGE_EX]&&!divBusy)begin
        div_start_reg = 1;
    end
end
```

EX 阶段是流水线 CPU 中的指令执行阶段，这里包含了 ALU 模块，乘法器，触发器模块和多路选择器等多个模块，是流水线中主要负责计算的一个阶段,其中几个调用的模块如下图所示：

```
ALU cpu_alu(
    .a(alua),
    .b(alub),
    .aluc(aluc),
    .r(aluo),
    .zero(zeroFlag),
    .carry(carryFlag),
    .negative(negFlag),
    .overflow(overflowFlag)
    );
```

```
DIV cpu_div(
    .dividend(dividend),
    .divisor(divisor),
    .div_res_q(divq),
    .div_res_r(divr),
    .div_busy(divBusy),
    .divu_res_q(divuq),
    .divu_res_r(divur),
    .divu_busy(divuBusy)
);

MUL cpu_mul(
    .a(multa),
    .b(multb),
    .mult_res(mulz),
    .multu_res(multuz)
);
```

（4） MEM 阶段

```
// MEM 阶段
// 存储器数据的字节和半字扩展
wire[31:0] mem_byte_ext, mem_half_ext;
// 数据存储器（Data Memory）读写控制信号
wire mem_dm_r, mem_dm_w;
// 数据存储器读取和写入的数据、以及地址
wire[31:0] mem_dm_rdata, mem_dm_wdata, mem_dm_addr;
// 用于指定字节使能的线路
wire[3:0] mem_byte_ena;
```

```
// EX_MEM 寄存器更新
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        // 复位时，将 EX_MEM 寄存器清零
        ex_mem_div_r <= 32'b0;
        ex_mem_mulz <= 64'b0;
        ex_mem_aluo <= 32'b0;

        ex_mem_inst <= 32'b0;
        ex_mem_overflowFlag <= 1'b0;
        ex_mem_div_q <= 32'b0;
        ex_mem_pass_data <= 32'b0;

    end
    else if (stall[`STAGE_EX] && !stall[`STAGE_ME]) begin
        // 如果有流水线暂停信号，清零 EX_MEM 寄存器
        ex_mem_div_r <= 32'b0;
        ex_mem_mulz <= 64'b0;
        ex_mem_aluo <= 32'b0;
        ex_mem_pass_data <= 32'b0;
        ex_mem_div_q <= 32'b0;
        ex_mem_inst <= 32'b0;
        ex_mem_overflowFlag <= 1'b0;
    end
    else if (!stall[`STAGE_EX]) begin
        // 如果没有流水线暂停信号，则更新 EX_MEM 寄存器
        ex_mem_div_r <= exOpDiv ? ex_div_r : exOpDivu ? ex_divu_r : 32'b0;
        ex_mem_mulz <= exOpMul ? ex_mulz : exOpMultu ? ex_multuz : 64'b0;
        ex_mem_div_q <= exOpDiv ? ex_div_q : exOpDivu ? ex_divu_q : 32'b0;

        ex_mem_pass_data <= id_ex_pass_data;
        ex_mem_inst <= id_ex_inst;
        ex_mem_aluo <= ex_aluo;
        ex_mem_overflowFlag <= overflowFlag;
    end
end
```

总的来说 MEM 阶段的主要任务是处理设计内存访问的指令，包括加载和存储操作，包含数据存储器模块、选择数据长度、符号扩展器和多路选择器。

（5） WB 阶段

```
// MEM-WB 阶段寄存器
// 存储传递给下一阶段的数据
reg[31:0] mem_wb_pass_data;
// 存储除法器结果的商和余数
reg[31:0] mem_wb_div_q, mem_wb_div_r;
// 存储 ALU 溢出标志
reg mem_wb_overflowFlag;  // ALU 溢出标志
// 存储乘法器结果
reg[63:0] mem_wb_mulz;
// 存储 ALU 的输出
reg[31:0] mem_wb_aluo;
// 存储指令
reg[31:0] mem_wb_inst;


//WB
wire wb_rf_wena;
wire[4:0]wb_rdc,wb_cp0_waddr;
wire[31:0]wb_rd,wb_cp0_wdata;
wire wb_hi_wena,wb_lo_wena;
wire[31:0]wb_hi_wdata,wb_lo_wdata;
```

```
// MEM_WB 寄存器更新
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        // 复位时，将 MEM_WB 寄存器清零
        mem_wb_div_q <= 32'b0;
        mem_wb_div_r <= 32'b0;
        mem_wb_aluo <= 32'b0;
        mem_wb_mulz <= 64'b0;
        mem_wb_pass_data <= 32'b0;
        mem_wb_inst <= 32'b0;
        mem_wb_overflowFlag <= 32'b0;
    end
    else if (stall[`STAGE_ME] && !stall[`STAGE_WB]) begin
        // 如果有流水线暂停信号，清零 MEM_WB 寄存器
        mem_wb_div_q <= 32'b0;
        mem_wb_div_r <= 32'b0;
        mem_wb_aluo <= 32'b0;
        mem_wb_mulz <= 64'b0;
        mem_wb_pass_data <= 32'b0;
        mem_wb_inst <= 32'b0;
        mem_wb_overflowFlag <= 32'b0;
    end
    else if (!stall[`STAGE_ME]) begin
        // 如果没有流水线暂停信号，则更新 MEM_WB 寄存器
        mem_wb_mulz <= ex_mem_mulz;
        mem_wb_div_q <= ex_mem_div_q;
        mem_wb_div_r <= ex_mem_div_r;
        mem_wb_aluo <= ex_mem_aluo;
        if (meOpLw) begin
            mem_wb_pass_data <= mem_dm_rdata;
        end
        else if (meOpLb || meOpLbu) begin
            mem_wb_pass_data <= mem_byte_ext;
        end
        else if (meOpLh || meOpLhu) begin
            mem_wb_pass_data <= mem_half_ext;
        end
        else begin
            mem_wb_pass_data <= ex_mem_pass_data;
        end
        mem_wb_inst <= ex_mem_inst;
        mem_wb_overflowFlag <= ex_mem_overflowFlag;
    end
end
```

该阶段是流水线五个阶段的最后一个阶段，其主要任务是将得到的结果写回寄存器文件中。

（6） 一些其他的模块（这里只给出模块定义，具体文件内容见附件）

a) 顶层模块 sccomp_dataflow.v:

```
module sccomp_dataflow(
    input clk_in,
    input rst,
    output [7:0]o_seg,
    output [7:0]o_sel
    );
```

b) IMEM.v:

```
module IMEM(
    input [10:0] addr,
    output [31:0] instr
    );

    dist_mem_gen_0 instr_mem(
        .a(addr),
        .spo(instr)
    );
endmodule
```

c) DMEM.v:

```
module DMEM(
    input clk,
    input rst,
    input ena,
    input DM_W,
    input DM_R,
    input[3:0]byteEna,
    input [31:0] DM_addr,
    input [31:0] DM_wdata,
    output [31:0] DM_rdata
    );
```

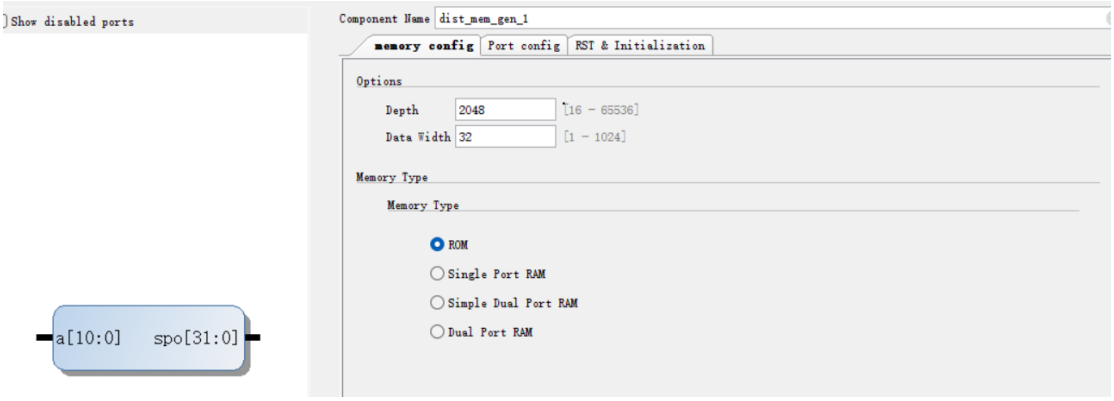
d) PCreg.v:

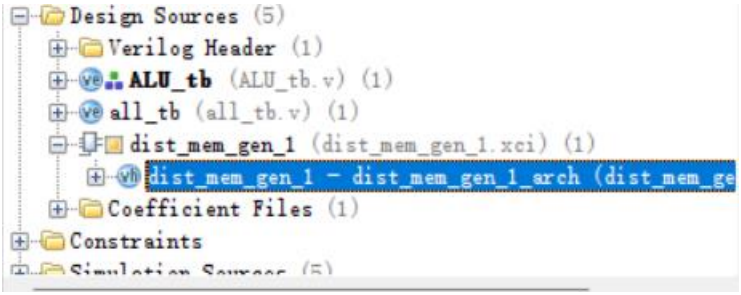
```
module PCReg(
    input clk,
    input rst,
    input ena,
    input [31:0]pc_in,
    output  reg[31:0] pc_out
    );
    always@(negedge clk or posedge rst)
    begin
        if(rst)pc_out<=32'h00400000;//这是MARS中取指令的开始地址
        else if(rst==0&&ena)pc_out<=pc_in;
    end
endmodule
```

四、 实验仿真过程

1、 静态流水线的仿真过程

首先向 Vivado 中导入 coe 指令文件，生成具体的 IP 核：

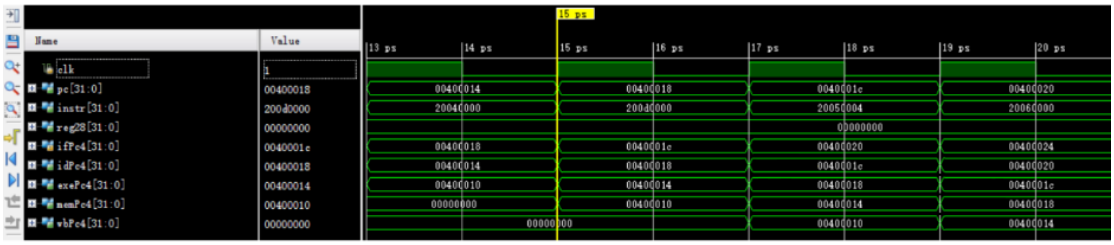




在 test_tb 文件测试下：

```
module all_tb();
    reg clk;
    reg rst;
//  wire [31:0] inst;
    wire [31:0] pc;
    wire [31:0]if_inst,id_inst,ex_inst,me_inst,wb_inst;
//  wire[31:0]rf_31,rf_2,rf_1,a,b,r,z;
//  wire [2:0]curState;
//  reg [31:0]fpc;
    wire[7:0]o_seg,o_sel;
    wire[4:0]STALL;
//  integer counter=0;
//  integer file_output;
//   sccomp_dataflow uut(clk,rst,if_inst,id_inst,ex_inst,me_inst,wb_inst,pc,STALL,o_seg,o_sel);
    initial
    begin
//          file_output=$fopen("D:\result.txt");
//          fpc<=32'h00400000;
            clk <= 1'b0;
            rst<=1'b1;
            #0.05 rst <= 1'b0;

    end

    always
     begin
    #0.0125 clk <= ~clk;
     end

//     sccomp_dataflow uut(.clk_in(clk),.reset(rst),.inst(inst),.pc(pc),.rf31(rf_31),.rf1(rf_1),.rf2(rf_2));

sccomp_dataflow uut(clk,rst,o_seg,o_sel);
```

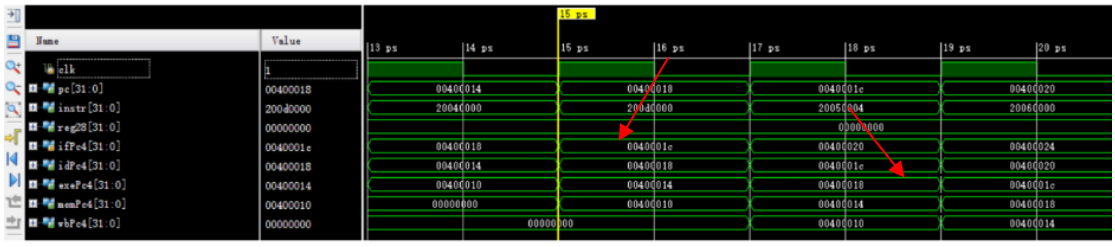
得到波形图：



五、 实验仿真的波形图及某时刻寄存器值的物理意义

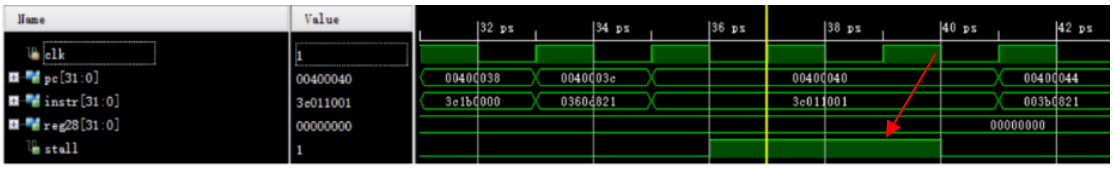
1、 静态流水线的波形图及某时刻寄存器值的物理意义

针对上方的波形图进行解释：



我们横向观察这个波形图，分别是每一级的信号输出从上到下对应于 clk 信号，pc 信号……;纵向看，是每个时钟周期内每一级的当前输出，也可以看出，五级流水线完成了各级之间的信号传递。
如果遇到数据冲突的情况时，我们可以通过观察 stall 变量的值为

1 的情况来观察流水线是否有效的解决了冲突问题：



我们看到当 stall 值为 1 时代表发生冲突，这里给出的解决方案是暂停周期，直至冲突不存在之后继续流水线。

六、 流水线 CPU 实验性能验证模型

实验性能验证模型：比萨塔摔鸡蛋游戏。两个同学在可变换层数的比萨塔上摔鸡蛋，一个同学秘密设定同一批鸡蛋耐摔值；另一个同学在指定层高的比萨塔拿着鸡蛋往下摔，用最少的摔次数和摔破的鸡蛋数求出鸡蛋的耐摔值。假定在耐摔值的楼层及其下面楼层，鸡蛋摔不破，可以重复使用，否则鸡蛋摔破。要求模型的算法输出包括：摔的总次数、摔的总鸡蛋数、最后摔的鸡蛋是否摔破。请使用 C 语言设计该验证模型的算法，并把 C 语言汇编为 RISC-V 指令汇编程序，同时利用编译器生成 RISC-V 指令集可执行目标程序。

算法分析：

分析该问题后我们发现，该问题可以使用动态规划算法解决，我们记 f(n,m)为 n 层楼，m 各鸡蛋所需要的次数,那么我已经得到了转移方程:

f(0,m) = 0 （m>=1）

f(n,1) = n (n>=1)

f(n,m) = min{max{f(i-1,m-1},f(n-i,m)}} + 1 (1<=i<=n)

根据该转移方程，我们可以编写出.c 文件，下面给出算法的核心实现部分：

```c
int eggDrop(int floors, int eggs) {
    int dp[MAX_FLOORS + 1][MAX_EGGS + 1];

    for (int i = 1; i <= floors; i++) {
        dp[i][1] = i;
        dp[i][0] = 0;
    }

    for (int j = 1; j <= eggs; j++) {
        dp[0][j] = 0;
    }

    for (int i = 2; i <= floors; i++) {
        for (int j = 2; j <= eggs; j++) {
            dp[i][j] = INT_MAX;
            for (int x = 1; x <= i; x++) {
                int res = 1 + max(dp[x - 1][j - 1], dp[i - x][j]);
                if (res < dp[i][j]) {
                    dp[i][j] = res;
                }
            }
        }
    }

    return dp[floors][eggs];
}
```

.asm 文件：

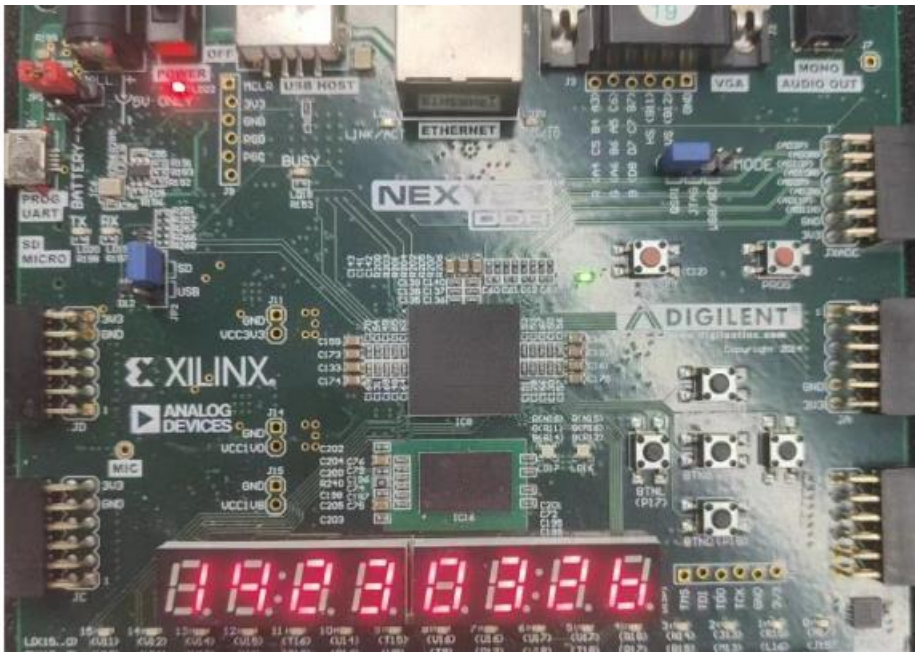
```
.global min
min:
    bge a0, a1, min_exit
    mv a0, a1
min_exit:
    ret

.global eggDrop
eggDrop:
    # Arguments: a0 = floors, a1 = eggs
    # Return: a0 = result

    # Initialize dp array
    li t0, 1
    li t1, MAX_FLOORS
init_floors_loop:
    beq t0, t1, init_eggs
    li t2, 1
init_eggs_loop:
    beq t2, MAX_EGGS, init_floors_next
    mul t3, t0, (MAX_EGGS + 1)
    add t4, t3, t2
```

```
    beqz t2, init_eggs_zero
    li t5, 0
    sw t5, 0(t4)
    j init_eggs_next
init_eggs_zero:
    sw t0, 0(t4)
init_eggs_next:
    addi t2, t2, 1
    j init_eggs_loop
init_floors_next:
    addi t0, t0, 1
    j init_floors_loop

    # Main calculation
    li t0, 2
    li t1, 2
calc_floors_loop:
    bge t0, MAX_FLOORS, calc_exit
    li t2, 2
calc_eggs_loop:
    bge t2, MAX_EGGS, calc_floors_next
    li t3, INT_MAX
    li t4, 1
calc_x_loop:
    bge t4, t0, calc_x_next
    lw t5, 0(t4)
    sub t6, t4, t2
    lw t7, 0(t6)
    add t8, t7, 1
    blt t5, t8, calc_x_next
    bge t3, t8, calc_x_next
    li t3, t8
calc_x_next:
    addi t4, t4, 1
    j calc_x_loop
calc_floors_next:
    addi t0, t0, 1
    j calc_floors_loop
calc_exit:
    mv a0, t3
    ret

.global main
main:
```

```
# Main function
# Initialize floors and eggs
li a0, 10
li a1, 2

# Call eggDrop function
call eggDrop

# Print result
mv a1, a0
li a0, 1
ecall

# Exit program
li a0, 10
ecall
```

七、 实验验算程序下板测试过程与实现

我们通过 Divider 分频器模块调整时钟周期，Seg7x16.v 七段数码管文件将最终结果显示在实验板上生成 bit 流运行如下：



经实验验证，结果正确。

八、 流水线的性能指标定性分析（包括：吞吐率、加速比、效率及相关与冲突分析、CPU 的运行时间及存储器空间的使用）

1、 静态流水线的性能指标定性分析

1. 吞吐率（Throughput）

吞吐率表示在单位时间内系统处理的任务数量。对于流水线 CPU，吞吐率可以通过每个时钟周期执行的指令数来衡量。经计算

吞吐率 = 1228/2173 = 0.565/时钟周期

2. 加速比（Speedup）

加速比是用来比较两个不同系统或两种不同实现之间性能提升的度量。对于流水线 CPU 的设计，加速比可以通过比较流水线执行和非流水线执行的性能来获得。经计算得：

加速比 = （207*4 + 106*3 + 896*4 + 25*2 + 198*3）/2173 = 2.473

3. 效率（Efficiency）

效率是流水线 CPU 的性能与理论最优性能之间的比率。理论最优性能是指在没有任何冲突和浪费的情况下，系统可以达到的最高性能。经计算得：

效率 = （207*4 + 106*3 + 896*4 + 25*2 + 198*3）/（2173*5） = 49.46%

4. 冲突分析

冲突分析包括数据冒险、控制冒险和结构冒险等。通过分析流水线执行过程中的指令冲突，可以识别瓶颈并提出优化建议。例如，数据冒险可能导致流水线停顿，需要引入数据前推等技术来解决。

在本次 CPU 流水线设计中，如果一条指令在 ID 阶段发现它使用到了现在正处于 EXE 阶段或者 MEM 阶段的指令所要写的寄存器中的数据时，则 stall 信号有效，等待前置指令全部进入 WB 阶段再继续执行。

而控制相关方面的冲突，则是利用延迟槽和提前判断分支技术，在 ID 阶段进行转移条件的判断，如果满足转移条件，则修改 PC 为转移目标地址，令处于延迟槽的指令失效，如果延迟槽内的也是分支指令，则禁止其对 PC 的修改。

九、 总结与体会

在本次实验中，我成功地完成了一个简单的 5 级流水线 CPU 的设计。这是一次非常有挑战性的任务，同时也是一个充实而有趣的学习过程。以下是我在这个实验中的一些总结和个人体会：

1. 学术理论的应用：这个实验让我深刻理解了流水线架构的工作原理，以及它是如何通过分阶段执行指令来提高 CPU 性能的。我不仅理论上了解了流水线的优势，还亲身体验了如何在实际设计中应用这些概念。

2. 技术挑战与解决方案： 在实验过程中，我面临了一些技术挑战，如数据冒险和流水线停顿。通过深入研究和学习，我成功地引入了一些解决方案，例如数据前推和指令重排序，以最大程度地减小冒险对性能的影响。

3. 实际测试与验证：我深刻体会到了测试与验证在硬件设计中的关键性。通过创建多样化的测试用例，我能够验证 CPU 的正确性和性能，并及时发现和解决一些潜在的问题。

4. 持续学习与未来展望：这个实验是我硬件设计领域的一次重要经验。在完成项目的过程中，我积累了大量的知识，并对计算机体系结构有

了更深层次的理解。未来，我期待进一步学习和改进我的设计，也希望能够应用这些知识到更为复杂和先进的 CPU 设计中。

十、 附件（所有程序）

1、 静态流水线的设计程序

（1）ALU.v:

```
1. `timescale 1ns / 1ps
2.
3.
4.
5. module ALU(
6.     input [31:0]a,
7.     input [31:0]b,
8.     input [3:0]alu_func,
9.     output  [31:0]res,
10.    output zero,
11.    output carry,
12.    output negative,
13.    output overflow
14.    );
15.            //参数中的 alu_func 代表执行运算的类型，由指令的 opt 和 func 决定，在传入 ALU 模块之前就计算好了
16.    //下面定义几个运算类型的表示,按照文档顺序定义
17.    parameter ADD = 0 ;
18.    parameter ADDU =1;
19.    parameter SUB = 2;
20.    parameter SUBU = 3;
21.    parameter AND = 4;
22.    parameter OR = 5;
23.    parameter XOR = 6;
24.    parameter NOR = 7;
25.    parameter SLT = 8;
26.    parameter SLTU = 9;
27.    parameter SLL = 10;
28.    parameter SRL = 11;
29.    parameter SRA = 12;
30.    parameter SLLV = 13;
31.    parameter SRLV = 14;
32.    parameter SRAV = 15 ;
33.    parameter LUI = 16;
34.    wire signed [31:0]sign_a,sign_b;//将给定数据转化为有符号数方便接下来进行计算
```

```
    assign sign_a = a;
    assign sign_b = b;
    reg[32:0]tmp_res;//暂时存储结果
    always@(*)
    begin
        case(alu_func)
        ADD : tmp_res<=sign_a+sign_b;
        ADDU : tmp_res<=a+b;
        SUB : tmp_res<=sign_a-sign_b;
        SUBU : tmp_res<=a-b;
        AND : tmp_res<=a & b;
        OR : tmp_res<=a | b;
        XOR : tmp_res<= a ^ b;
        NOR : tmp_res<= ~(a|b);
        SLT : tmp_res<=(sign_a<sign_b);
        SLTU : tmp_res<=(a<b);
        SLL : tmp_res<=(b<<a);
        SRL : tmp_res<=(b>>a);
        SRA : tmp_res<=(sign_b>>>sign_a);
        SLLV : tmp_res<= (b<<a[4:0]);
        SRLV : tmp_res<=(b>>a[4:0]);
        SRAV : tmp_res<=(sign_b>>>sign_a[4:0]);
        LUI : tmp_res<={b[15:0],16'b0};
        endcase
    end
    assign res = tmp_res[31:0];
    assign zero=(tmp_res==32'b0)?1:0;
    assign carry = tmp_res[32];
    assign overflow = tmp_res[32];
    assign negative = tmp_res[31];
endmodule
```

（2）CP0.v:

```
`timescale 1ns / 1ps

module CP0(
    input clk,
    input rst,
  input mfc0,
    input mtc0,//cp0 write
  input eret,//是否为错误返回信号
  input exception,
    input [31:0]pc,//指令
```

```
    input [4:0] addr,//address
    input [31:0] wdata,//向 cp0 写的数据
    input teq_exc,
    input[3:0]cause,//异常原因
output [31:0] status,
    output [31:0]rdata,//cp0 读出的数据
    output[31:0]exc_addr //返回异常保存的地址
    );

    parameter  STATUS_POS = 12;
    parameter  CAUSE_POS = 13;
    parameter  EPC_POS = 14;
    parameter  SYS_ERR = 4'b1000;
    parameter  BREAK_ERR = 4'b1001;
    parameter  TEQ_ERR = 4'b1101;

    reg[31:0] cp0_reg[0:31];
    // wire [31:0] status;
    assign status = cp0_reg[STATUS_POS];
    // wire exception;
    assign exception = (status[0] == 1)&&
    ((status[1] == 1&& cause == SYS_ERR)||
    (status[2] == 1&&cause == BREAK_ERR)||
    (status == 1&&cause == TEQ_ERR));

    assign rdata = cp0_reg[addr];
    assign exc_addr = (eret?cp0_reg[EPC_POS]:32'h00400004);

    integer i;
    always @(negedge clk or posedge rst)
    begin
        if(rst == 1)begin
            for(i = 0;i<32;i=i+1)
                cp0_reg[i]<=32'b0;
        end
        else
        begin
            if(mtc0 == 1)
                cp0_reg[addr]<=wdata;
            else if(exception)begin
                cp0_reg[STATUS_POS]<=(status<<5);
                cp0_reg[CAUSE_POS]<={24'b0,cause,2'b0};
                cp0_reg[EPC_POS]<=pc;
            end
```

```
            else if(eret == 1)begin
                cp0_reg[STATUS_POS]<=(status>>5);
            end
        end
    end
endmodule
```

（3）CPU.v：

```
`timescale 1ns / 1ps
`include "define_cpu.vh"
//cpu 模块
module CPU(
    input rst,
    input clk,
    output DM_W,//DM 写信号
    output DM_R,//DM 读信号
    input [31:0] IM_inst,
    output [31:0] DM_addr,//DM 地址
    input [31:0] DM_rdata,//DM 读到的数据
    output [31:0] PC_out,//PC 输出结果
    output [31:0] DM_wdata,
    output [3:0] Byte_ena
    );

//IF 阶段
// 声明程序计数器（if_pc）和指令（if_inst）的线路
wire[31:0] if_pc;
wire[31:0] if_inst;

// IF-ID 阶段寄存器
// 保存下一个程序计数器和 ID 阶段的指令
reg[31:0] if_id_npc;
reg[31:0] if_id_inst;

// ID 阶段
// 用于符号扩展和零扩展数据的线路，以及控制信号
wire[31:0] if_sext_16, id_uext_16, id_uext_5, id_sext_18;
wire[4:0] id_rsc, id_rtc;
wire [31:0] id_rs, id_rt;
wire id_pc_ena;
wire[31:0] id_pc_in, id_npc;
wire[31:0] id_alu_a, id_alu_b;
```

```
35.wire[4:0] id_cp0_raddr;
36.wire[31:0] id_cp0_rdata, id_cp0_epcout, id_cp0_status;
37.wire[31:0] id_pass_data;
38.
39.//ID-EX
40.reg[31:0] id_ex_alua,id_ex_alub;
41.reg[31:0] id_ex_pass_data;
42.reg[31:0] id_ex_inst;
43.
44.//EX
45.// 执行阶段（EX）的算术运算单元 A、B、和输出 O 的线路
46.wire[31:0] ex_alua, ex_alub, ex_aluo;
47.// 执行阶段的乘法器输入 A、B、无符号输入 A、B、以及输出 Z 的线路
48.wire[31:0] ex_mula, ex_mulb, ex_multua, ex_multub;
49.wire[63:0] ex_mulz, ex_multuz;
50.// 执行阶段的除法器和除法器（无符号）控制信号
51.wire ex_div_start, ex_div_busy, ex_divu_start, ex_divu_busy
   ;
52.// 执行阶段的除法器输入：被除数、除数、商和余数（有符号）
53.wire[31:0] ex_div_dividend, ex_div_divisor, ex_div_q, ex_di
   v_r;
54.// 执行阶段的除法器输入：被除数、除数、商和余数（无符号）
55.wire[31:0] ex_divu_dividend, ex_divu_divisor, ex_divu_q, ex
   _divu_r;
56.
57.// EX-MEM 阶段寄存器
58.// 存储除法器结果的商和余数
59.reg[31:0] ex_mem_div_q, ex_mem_div_r;
60.// 存储乘法器结果
61.reg[63:0] ex_mem_mulz;
62.// 存储 ALU 的输出
63.reg[31:0] ex_mem_aluo;
64.// 存储传递给下一阶段的数据
65.reg[31:0] ex_mem_pass_data;
66.// 存储指令
67.reg[31:0] ex_mem_inst;
68.// 存储 ALU 溢出标志
69.reg ex_mem_overflowFlag;  // ALU 溢出标志
70.
71.// MEM 阶段
72.// 存储器数据的字节和半字扩展
73.wire[31:0] mem_byte_ext, mem_half_ext;
74.// 数据存储器（Data Memory）读写控制信号
75.wire mem_dm_r, mem_dm_w;
```

```
// 数据存储器读取和写入的数据、以及地址
wire[31:0] mem_dm_rdata, mem_dm_wdata, mem_dm_addr;
// 用于指定字节使能的线路
wire[3:0] mem_byte_ena;


// MEM-WB 阶段寄存器
// 存储传递给下一阶段的数据
reg[31:0] mem_wb_pass_data;
// 存储除法器结果的商和余数
reg[31:0] mem_wb_div_q, mem_wb_div_r;
// 存储 ALU 溢出标志
reg mem_wb_overflowFlag;  // ALU 溢出标志
// 存储乘法器结果
reg[63:0] mem_wb_mulz;
// 存储 ALU 的输出
reg[31:0] mem_wb_aluo;
// 存储指令
reg[31:0] mem_wb_inst;


//WB
wire wb_rf_wena;
 wire[4:0]wb_rdc,wb_cp0_waddr;
 wire[31:0]wb_rd,wb_cp0_wdata;
 wire wb_hi_wena,wb_lo_wena;
 wire[31:0]wb_hi_wdata,wb_lo_wdata;

 //others
 reg[31:0]HI,LO;
 reg[4:0]stall;


 //IF 阶段的译码
 wire [31:0]lbw_tmp1;
 wire [4:0] ifRt = if_inst[20:16];
 wire [5:0] ifFunc = if_inst[5:0];
 wire [5:0] ifOp =if_inst[31:26];
 wire [4:0] ifRs = if_inst[25:21];


 wire ifOpLui = (ifOp == `OP_LUI);
 wire ifOpXori = (ifOp == `OP_XORI);
```

```
120.  wire ifOpSlti = (ifOp == `OP_SLTI);
121.  wire ifOpAddu = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   ADDU);
122.  wire ifOpAnd = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_A
   ND);
123.  wire ifOpBeq = (ifOp == `OP_BEQ);
124.  wire ifOpAddi = (ifOp == `OP_ADDI);
125.  wire ifOpAddiu = (ifOp == `OP_ADDIU);
126.  wire ifOpJ = (ifOp == `OP_J);
127.  wire ifOpJal = (ifOp == `OP_JAL);
128.  wire ifOpJr = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_JR
   );
129.  wire ifOpAndi = (ifOp == `OP_ANDI);
130.  wire ifOpOri = (ifOp == `OP_ORI);
131.  wire ifOpSltiu = (ifOp == `OP_SLTIU);
132.  wire ifOpBne = (ifOp == `OP_BNE);
133.  wire ifOpLw = (ifOp == `OP_LW);
134.
135.
136.
137.  wire ifOpSll = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_S
   LL && if_inst != 32'b0);
138.  wire ifOpSllv = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   SLLV);
139.  wire ifOpSltu = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   SLTU);
140.  wire ifOpSra = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_S
   RA);
141.  wire ifOpSrl = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_S
   RL);
142.  wire ifOpSubu = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   SUBU);
143.
144.  wire ifOpAdd = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_A
   DD);
145.  wire ifOpSub = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_S
   UB);
146.
147.  wire ifOpSlt = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_S
   LT);
148.  wire ifOpSrlv = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   SRLV);
149.  wire ifOpSrav = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   SRAV);
```

```
150.  wire ifOpClz = (ifOp == `OP_SPECIAL2 && ifFunc == `FUNCT_
   CLZ);
151.
152.  wire ifOpDivu = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   DIVU);
153.  wire ifOpSw = (ifOp == `OP_SW);
154.  wire ifOpXor = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_X
   OR);
155.  wire ifOpNor = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_N
   OR);
156.  wire ifOpOr = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_OR
   );
157.  wire ifOpEret = (ifOp == `OP_COP0 && ifFunc == `FUNCT_ERE
   T);
158.  wire ifOpJalr = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   JALR);
159.  wire ifOpLbu = (ifOp == `OP_LBU);
160.  wire ifOpLhu = (ifOp == `OP_LHU);
161.  wire ifOpSb = (ifOp == `OP_SB);
162.  wire ifOpSh = (ifOp == `OP_SH);
163.  wire ifOpLb = (ifOp == `OP_LB);
164.
165.  wire ifOpLh = (ifOp == `OP_LH);
166.  wire ifOpMfc0 = (ifOp == `OP_COP0 && ifRs == `RS_MF);
167.  wire ifOpMthi = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   MTHI);
168.  wire ifOpMtlo = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   MTLO);
169.  wire ifOpMul = (ifOp == `OP_SPECIAL2 && ifFunc == `FUNCT_
   MUL);
170.  wire ifOpMfhi = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   MFHI);
171.  wire ifOpMflo = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_
   MFLO);
172.
173.  wire ifOpTeq = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_T
   EQ);
174.  wire ifOpBreak = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT
   _BREAK);
175.  wire ifOpDiv = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT_D
   IV);
176.  wire ifOpBgez = (ifOp == `OP_REGIMM && ifRt == `RT_BGEZ);
177.  wire ifOpMtc0 = (ifOp == `OP_COP0 && ifRs == `RS_MT);
```

```
178.  wire ifOpMultu = (ifOp == `OP_SPECIAL && ifFunc == `FUNCT
   _MULTU);
179.  wire ifOpSyscall = (ifOp == `OP_SPECIAL && ifFunc == `FUN
   CT_SYSCALL);
180.
181.
182.  //ID 阶段的译码
183.  wire [5:0] idOp =if_id_inst[31:26];
184.  wire [4:0] idRt = if_id_inst[20:16];
185.  wire [5:0] idFunc = if_id_inst[5:0];
186.  wire [4:0] idRs = if_id_inst[25:21];
187.  wire [31:0]lbw_tmp2;
188.
189.
190.  wire idOpAndi = (idOp == `OP_ANDI);
191.  wire idOpOri = (idOp == `OP_ORI);
192.  wire idOpSltiu = (idOp == `OP_SLTIU);
193.  wire idOpAddi = (idOp == `OP_ADDI);
194.  wire idOpAddiu = (idOp == `OP_ADDIU);
195.  wire idOpLui = (idOp == `OP_LUI);
196.
197.  wire idOpAddu = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   ADDU);
198.  wire idOpAnd = (idOp == `OP_SPECIAL && idFunc == `FUNCT_A
   ND);
199.  wire idOpXori = (idOp == `OP_XORI);
200.  wire idOpSlti = (idOp == `OP_SLTI);
201.  wire idOpBne = (idOp == `OP_BNE);
202.  wire idOpJ = (idOp == `OP_J);
203.
204.  wire idOpBeq = (idOp == `OP_BEQ);
205.  wire idOpJal = (idOp == `OP_JAL);
206.  wire idOpJr = (idOp == `OP_SPECIAL && idFunc == `FUNCT_JR
   );
207.  wire idOpXor = (idOp == `OP_SPECIAL && idFunc == `FUNCT_X
   OR);
208.  wire idOpNor = (idOp == `OP_SPECIAL && idFunc == `FUNCT_N
   OR);
209.  wire idOpOr = (idOp == `OP_SPECIAL && idFunc == `FUNCT_OR
   );
210.  wire idOpLw = (idOp == `OP_LW);
211.
212.  wire idOpSll = (idOp == `OP_SPECIAL && idFunc == `FUNCT_S
   LL && if_id_inst != 32'b0);
```

```
213.  wire idOpSllv = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   SLLV);
214.
215.  wire idOpSubu = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   SUBU);
216.  wire idOpSrav = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   SRAV);
217.  wire idOpClz = (idOp == `OP_SPECIAL2 && idFunc == `FUNCT_
   CLZ);
218.  wire idOpSw = (idOp == `OP_SW);
219.  wire idOpAdd = (idOp == `OP_SPECIAL && idFunc == `FUNCT_A
   DD);
220.  wire idOpSltu = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   SLTU);
221.
222.  wire idOpSub = (idOp == `OP_SPECIAL && idFunc == `FUNCT_S
   UB);
223.  wire idOpSlt = (idOp == `OP_SPECIAL && idFunc == `FUNCT_S
   LT);
224.  wire idOpSrlv = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   SRLV);
225.  wire idOpSra = (idOp == `OP_SPECIAL && idFunc == `FUNCT_S
   RA);
226.  wire idOpSrl = (idOp == `OP_SPECIAL && idFunc == `FUNCT_S
   RL);
227.
228.
229.  wire idOpLhu = (idOp == `OP_LHU);
230.  wire idOpSb = (idOp == `OP_SB);
231.  wire idOpSh = (idOp == `OP_SH);
232.  wire idOpLb = (idOp == `OP_LB);
233.  wire idOpLbu = (idOp == `OP_LBU);
234.  wire idOpLh = (idOp == `OP_LH);
235.  wire idOpDivu = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   DIVU);
236.  wire idOpEret = (idOp == `OP_COP0 && idFunc == `FUNCT_ERE
   T);
237.  wire idOpMfhi = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   MFHI);
238.  wire idOpMflo = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   MFLO);
239.  wire idOpMtc0 = (idOp == `OP_COP0 && idRs == `RS_MT);
240.  wire idOpSyscall = (idOp == `OP_SPECIAL && idFunc == `FUN
   CT_SYSCALL);
```

```
241.  wire idOpTeq = (idOp == `OP_SPECIAL && idFunc == `FUNCT_T
   EQ);
242.  wire idOpMthi = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   MTHI);
243.  wire idOpMtlo = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   MTLO);
244.
245.  wire idOpMultu = (idOp == `OP_SPECIAL && idFunc == `FUNCT
   _MULTU);
246.
247.  wire idOpJalr = (idOp == `OP_SPECIAL && idFunc == `FUNCT_
   JALR);
248.  wire idOpMul = (idOp == `OP_SPECIAL2 && idFunc == `FUNCT_
   MUL);
249.  wire idOpBgez = (idOp == `OP_REGIMM && idRt == `RT_BGEZ);
250.  wire idOpBreak = (idOp == `OP_SPECIAL && idFunc == `FUNCT
   _BREAK);
251.  wire idOpDiv = (idOp == `OP_SPECIAL && idFunc == `FUNCT_D
   IV);
252.  wire idOpMfc0 = (idOp == `OP_COP0 && idRs == `RS_MF);
253.
254.
255.      // EX 阶段译码
256.      wire [31:0]lbw_tmp3;
257.      wire [5:0] exOp = id_ex_inst[31:26];
258.      wire [5:0] exRs = id_ex_inst[25:21];
259.      wire [4:0] exRt = id_ex_inst[20:16];
260.      wire [4:0] exRd = id_ex_inst[15:11];
261.      wire [5:0] exFunc = id_ex_inst[5:0];
262.
263.      //给信号赋值
264.      wire exOpAddiu = (exOp == `OP_ADDIU);
265.      wire exOpAndi = (exOp == `OP_ANDI);
266.      wire exOpAddi = (exOp == `OP_ADDI);
267.
268.
269.      wire[15:0] lbw_tmp4;
270.      wire exOpXori = (exOp == `OP_XORI);
271.      wire exOpSlti = (exOp == `OP_SLTI);
272.      wire exOpOri = (exOp == `OP_ORI);
273.      wire exOpSltiu = (exOp == `OP_SLTIU);
274.      wire exOpLui = (exOp == `OP_LUI);
275.      wire exOpAddu = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_ADDU);
```

```
276.
277.      wire exOpJal = (exOp == `OP_JAL);
278.      wire exOpAnd = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_AND);
279.
280.      wire exOpLw = (exOp == `OP_LW);
281.      wire exOpXor = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_XOR);
282.      wire exOpBeq = (exOp == `OP_BEQ);
283.      wire exOpBne = (exOp == `OP_BNE);
284.      wire exOpJ = (exOp == `OP_J);
285.      wire exOpJr = (exOp == `OP_SPECIAL && exFunc == `FUNC
   T_JR);
286.      wire exOpNor = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_NOR);
287.      wire exOpOr = (exOp == `OP_SPECIAL && exFunc == `FUNC
   T_OR);
288.
289.      wire exOpSra = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_SRA);
290.      wire exOpSrl = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_SRL);
291.      wire exOpSltu = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_SLTU);
292.      wire exOpSubu = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_SUBU);
293.      wire exOpSw = (exOp == `OP_SW);
294.      wire exOpSll = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_SLL && id_ex_inst != 32'b0);
295.      wire exOpSllv = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_SLLV);
296.
297.
298.      wire exOpSrav = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_SRAV);
299.      wire exOpClz = (exOp == `OP_SPECIAL2 && exFunc == `FU
   NCT_CLZ);
300.      wire exOpAdd = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_ADD);
301.      wire exOpSlt = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_SLT);
302.      wire exOpSrlv = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_SRLV);
```

```
303.      wire exOpDivu = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_DIVU);
304.      wire exOpEret = (exOp == `OP_COP0 && exFunc == `FUNCT
   _ERET);
305.      wire exOpSub = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_SUB);
306.
307.      wire[31:0]yff_tmp1;
308.      wire exOpJalr = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_JALR);
309.      wire exOpLb = (exOp == `OP_LB);
310.      wire exOpLhu = (exOp == `OP_LHU);
311.      wire exOpSb = (exOp == `OP_SB);
312.      wire exOpSh = (exOp == `OP_SH);
313.      wire exOpLh = (exOp == `OP_LH);
314.      wire exOpLbu = (exOp == `OP_LBU);
315.
316.      wire[31:0]icpc_tmp1;
317.      wire exOpMfc0 = (exOp == `OP_COP0 && exRs == `RS_MF);
318.      wire exOpMfhi = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_MFHI);
319.      wire exOpMthi = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_MTHI);
320.      wire exOpMtlo = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_MTLO);
321.      wire exOpMul = (exOp == `OP_SPECIAL2 && exFunc == `FU
   NCT_MUL);
322.
323.
324.      wire exOpSyscall = (exOp == `OP_SPECIAL && exFunc ==
   `FUNCT_SYSCALL);
325.      wire exOpTeq = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_TEQ);
326.      wire exOpMultu = (exOp == `OP_SPECIAL && exFunc == `F
   UNCT_MULTU);
327.      wire exOpMflo = (exOp == `OP_SPECIAL && exFunc == `FU
   NCT_MFLO);
328.      wire exOpMtc0 = (exOp == `OP_COP0 && exRs == `RS_MT);
329.      wire exOpBgez = (exOp == `OP_REGIMM && exRt == `RT_BG
   EZ);
330.      wire exOpBreak = (exOp == `OP_SPECIAL && exFunc == `F
   UNCT_BREAK);
331.      wire exOpDiv = (exOp == `OP_SPECIAL && exFunc == `FUN
   CT_DIV);
```

```
332.
333.      // ME 阶段译码
334.      wire [5:0] meOp = ex_mem_inst[31:26];
335.      wire [5:0] meFunc = ex_mem_inst[5:0];
336.      wire [5:0] meRs = ex_mem_inst[25:21];
337.      wire [4:0] meRt = ex_mem_inst[20:16];
338.      wire [4:0] meRd = ex_mem_inst[15:11];
339.      wire[7:0]cpu_tmp0;
340.
341.
342.      wire meOpSltiu = (meOp == `OP_SLTIU);
343.      wire meOpLui = (meOp == `OP_LUI);
344.      wire meOpAddi = (meOp == `OP_ADDI);
345.      wire meOpAddiu = (meOp == `OP_ADDIU);
346.      wire meOpAndi = (meOp == `OP_ANDI);
347.      wire meOpOri = (meOp == `OP_ORI);
348.
349.      wire meOpAddu = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_ADDU);
350.      wire meOpAnd = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_AND);
351.
352.      wire meOpJr = (meOp == `OP_SPECIAL && meFunc == `FUNC
   T_JR);
353.      wire meOpLw = (meOp == `OP_LW);
354.      wire meOpXor = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_XOR);
355.      wire meOpNor = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_NOR);
356.      wire meOpSlti = (meOp == `OP_SLTI);
357.      wire meOpBne = (meOp == `OP_BNE);
358.      wire meOpJ = (meOp == `OP_J);
359.      wire meOpJal = (meOp == `OP_JAL);
360.
361.      wire meOpOr = (meOp == `OP_SPECIAL && meFunc == `FUNC
   T_OR);
362.
363.      wire meOpSubu = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_SUBU);
364.      wire meOpSll = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_SLL && ex_mem_inst != 32'b0);
365.      wire meOpSllv = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_SLLV);
366.      wire meOpSw = (meOp == `OP_SW);
```

```
367.      wire meOpAdd = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_ADD);
368.      wire meOpSub = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_SUB);
369.      wire meOpSltu = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_SLTU);
370.       wire meOpBeq = (meOp == `OP_BEQ);
371.      wire meOpXori = (meOp == `OP_XORI);
372.      wire meOpSra = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_SRA);
373.      wire meOpSrav = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_SRAV);
374.      wire meOpClz = (meOp == `OP_SPECIAL2 && meFunc == `FU
   NCT_CLZ);
375.      wire meOpDivu = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_DIVU);
376.      wire meOpSrl = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_SRL);
377.
378.      wire meOpSlt = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_SLT);
379.      wire meOpSrlv = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_SRLV);
380.
381.      wire meOpEret = (meOp == `OP_COP0 && meFunc == `FUNCT
   _ERET);
382.      wire meOpJalr = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_JALR);
383.      wire meOpLb = (meOp == `OP_LB);
384.      wire meOpLbu = (meOp == `OP_LBU);
385.
386.      wire meOpMfc0 = (meOp == `OP_COP0 && meRs == `RS_MF);
387.      wire meOpBgez = (meOp == `OP_REGIMM && meRt == `RT_BG
   EZ);
388.      wire meOpBreak = (meOp == `OP_SPECIAL && meFunc == `F
   UNCT_BREAK);
389.      wire meOpDiv = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_DIV);
390.      wire meOpMfhi = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_MFHI);
391.      wire meOpLhu = (meOp == `OP_LHU);
392.
393.      wire meOpMflo = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_MFLO);
```

```
394.      wire meOpMtlo = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_MTLO);
395.      wire meOpMul = (meOp == `OP_SPECIAL2 && meFunc == `FU
   NCT_MUL);
396.      wire meOpMultu = (meOp == `OP_SPECIAL && meFunc == `F
   UNCT_MULTU);
397.      wire meOpMtc0 = (meOp == `OP_COP0 && meRs == `RS_MT);
398.      wire meOpMthi = (meOp == `OP_SPECIAL && meFunc == `FU
   NCT_MTHI);
399.      wire meOpSb = (meOp == `OP_SB);
400.      wire meOpSh = (meOp == `OP_SH);
401.      wire meOpLh = (meOp == `OP_LH);
402.
403.      wire meOpSyscall = (meOp == `OP_SPECIAL && meFunc ==
   `FUNCT_SYSCALL);
404.      wire meOpTeq = (meOp == `OP_SPECIAL && meFunc == `FUN
   CT_TEQ);
405.
406.
407.       // WB 阶段写入
408.      wire [5:0] wbOp = mem_wb_inst[31:26];
409.      wire [5:0] wbRs = mem_wb_inst[25:21];
410.      wire [4:0] wbRt = mem_wb_inst[20:16];
411.      wire [5:0] wbFunc = mem_wb_inst[5:0];
412.      wire [4:0] wbRd = mem_wb_inst[15:11];
413.      wire[15:0] cpu_tmp2;
414.
415.      wire wbOpAddi = (wbOp == `OP_ADDI);
416.      wire wbOpAddu = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_ADDU);
417.      wire wbOpAnd = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_AND);
418.      wire wbOpBeq = (wbOp == `OP_BEQ);
419.      wire wbOpAddiu = (wbOp == `OP_ADDIU);
420.
421.      wire wbOpLui = (wbOp == `OP_LUI);
422.      wire wbOpXori = (wbOp == `OP_XORI);
423.      wire wbOpSlti = (wbOp == `OP_SLTI);
424.
425.      wire wbOpBne = (wbOp == `OP_BNE);
426.      wire wbOpJ = (wbOp == `OP_J);
427.      wire wbOpJal = (wbOp == `OP_JAL);
428.      wire wbOpJr = (wbOp == `OP_SPECIAL && wbFunc == `FUNC
   T_JR);
```

```
429.      wire wbOpLw = (wbOp == `OP_LW);
430.
431.      wire wbOpOri = (wbOp == `OP_ORI);
432.      wire wbOpSltiu = (wbOp == `OP_SLTIU);
433.      wire wbOpSll = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_SLL && mem_wb_inst != 32'b0);
434.      wire wbOpSllv = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_SLLV);
435.      wire wbOpSltu = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_SLTU);
436.      wire wbOpSra = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_SRA);
437.      wire wbOpAndi = (wbOp == `OP_ANDI);
438.      wire wbOpAdd = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_ADD);
439.      wire wbOpSub = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_SUB);
440.      wire wbOpSlt = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_SLT);
441.      wire wbOpXor = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_XOR);
442.      wire wbOpNor = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_NOR);
443.      wire wbOpOr = (wbOp == `OP_SPECIAL && wbFunc == `FUNC
   T_OR);
444.
445.      wire wbOpSrl = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_SRL);
446.      wire wbOpSubu = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_SUBU);
447.      wire wbOpSw = (wbOp == `OP_SW);
448.
449.      wire[31:0] cpu_tmp3;
450.      wire wbOpSrlv = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_SRLV);
451.      wire wbOpSrav = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_SRAV);
452.      wire wbOpLbu = (wbOp == `OP_LBU);
453.      wire wbOpClz = (wbOp == `OP_SPECIAL2 && wbFunc == `FU
   NCT_CLZ);
454.
455.      wire wbOpJalr = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_JALR);
456.      wire wbOpLb = (wbOp == `OP_LB);
```

```
457.      wire wbOpBgez = (wbOp == `OP_REGIMM && wbRt == `RT_BG
   EZ);
458.      wire wbOpBreak = (wbOp == `OP_SPECIAL && wbFunc == `F
   UNCT_BREAK);
459.      wire wbOpDiv = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_DIV);
460.
461.      wire wbOpLhu = (wbOp == `OP_LHU);
462.      wire wbOpSb = (wbOp == `OP_SB);
463.      wire wbOpDivu = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_DIVU);
464.      wire wbOpEret = (wbOp == `OP_COP0 && wbFunc == `FUNCT
   _ERET);
465.
466.      wire[7:0] cpu_tmp5;
467.      wire wbOpMfc0 = (wbOp == `OP_COP0 && wbRs == `RS_MF);
468.      wire wbOpMfhi = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_MFHI);
469.      wire wbOpSh = (wbOp == `OP_SH);
470.      wire wbOpMthi = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_MTHI);
471.      wire wbOpMtlo = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_MTLO);
472.      wire wbOpLh = (wbOp == `OP_LH);
473.      wire wbOpMflo = (wbOp == `OP_SPECIAL && wbFunc == `FU
   NCT_MFLO);
474.
475.      wire wbOpMultu = (wbOp == `OP_SPECIAL && wbFunc == `F
   UNCT_MULTU);
476.      wire wbOpMtc0 = (wbOp == `OP_COP0 && wbRs == `RS_MT);
477.      wire wbOpMul = (wbOp == `OP_SPECIAL2 && wbFunc == `FU
   NCT_MUL);
478.
479.
480.      wire wbOpSyscall = (wbOp == `OP_SPECIAL && wbFunc ==
   `FUNCT_SYSCALL);
481.      wire wbOpTeq = (wbOp == `OP_SPECIAL && wbFunc == `FUN
   CT_TEQ);
482.
483.
484.  //ALU
485.  parameter LUI    =    4'b1000;    //r={b[15:0],16'b0}
486.  parameter SLT    =    4'b1011;    //r=(a-b<0)?1:0 signed
```

```
487.  parameter SLTU    =    4'b1010;    //r=(a-
   b<0)?1:0 unsigned
488.  parameter OR    =    4'b0101;    //r=a|b
489.  parameter ADDU    =    4'b0000;    //r=a+b unsigned
490.  parameter ADD    =    4'b0010;    //r=a+b signed
491.  parameter XOR    =    4'b0110;    //r=a^b
492.  parameter NOR    =    4'b0111;    //r=~(a|b)
493.
494.  parameter SLL    =    4'b1110;    //r=b<<a
495.  parameter SRA    =    4'b1100;    //r=b>>>a
496.  parameter SUBU    =    4'b0001;    //r=a-b unsigned
497.  parameter SUB    =    4'b0011;    //r=a-b signed
498.  parameter AND    =    4'b0100;    //r=a&b
499.  parameter SRL    =    4'b1101;    //r=b>>a
500.  parameter CLZ    =    4'b1111;
501.
502.  //写入 DM
503.  assign DM_wdata = mem_dm_wdata;
504.  assign DM_addr = mem_dm_addr;
505.  assign DM_W = mem_dm_w;
506.  assign DM_R = mem_dm_r;
507.
508.  assign Byte_ena = mem_byte_ena;
509.
510.  //cp0 的相关变量
511.  wire exception;
512.  wire[4:0] cause;
513.  assign cause=idOpSyscall?5'b01000:idOpBreak?5'b01001:idOp
   Teq?5'b01101:5'b11111;
514.  assign exception = idOpSyscall||idOpBreak||idOpTeq;
515.  reg[31:0] cpu_exec;
516.
517.  wire[4:0] cp0_addr;
518.  assign cp0_addr = wbOpMtc0?wb_cp0_waddr:idOpMfc0?id_cp0_r
   addr:5'bz;
519.
520.  // 程序计数器（PC）相关信号
521.  // 输出当前 PC 的值
522.  wire[31:0] pc_out, pc_in;  // pc_reg
523.  // 控制 PC 使能信号
524.  wire pc_ena;
525.  // 将 ID 阶段的 PC 使能信号传递给 PC 使能
526.  assign pc_ena = id_pc_ena;
527.  // 将 PC 输出连接到 pc_out
```

```
528.  assign PC_out = pc_out;
529.  // 根据 ID 阶段的 PC 使能信号选择输入，如果不使能则输入为未知（32'bz）
530.  assign pc_in = id_pc_ena ? id_pc_in : 32'bz;
531.
532.
533.
534.  //alu
535.  // 算术逻辑单元（ALU）信号
536.  wire[31:0] alua, alub, aluo;
537.
538.  // 将执行阶段的 ALU 输入连接到 ALU 输入 A 和 B
539.  assign alua = ex_alua;
540.  assign alub = ex_alub;
541.  // ALU 标志信号
542.  wire zeroFlag, negFlag, overflowFlag, carryFlag;
543.
544.  // ALU 控制信号
545.  wire[3:0] aluc;
546.
547.  // 根据执行阶段的操作码选择 ALU 的操作
548.  assign aluc = (exOpAddi || exOpLw ||exOpJal  || exOpSw || exOpJalr  || exOpAdd|| exOpLb || exOpLbu
549.                 || exOpLhu || exOpSb || exOpSh || exOpLh) ? ADD :
550.                 (exOpAddiu || exOpAddu) ? ADDU : (exOpSubu) ? SUBU : (exOpSub) ? SUB :
551.                 (exOpAndi || exOpAnd) ? AND :
552.                 (exOpOr || exOpOri) ? OR : (exOpXor || exOpXori) ? XOR : (exOpNor) ? NOR : (exOpLui) ? LUI :
553.                 (exOpSlt || exOpSlti) ? SLT : (exOpSltu || exOpSltiu) ? SLTU : (exOpSra || exOpSrav) ? SRA :
554.                 (exOpSll || exOpSllv) ? SLL : (exOpSrl || exOpSrlv) ? SRL : (exOpClz) ? CLZ : 4'bz;
555.
556.
557.
558.  // 寄存器文件（Register File）的定义变量
559.  // 用于指定读取的目的寄存器编号
560.  wire[4:0] rdc, rtc, rsc;
561.  wire[4:0] rkc;
562.  // 寄存器写使能信号
563.  wire RF_W;
564.  // 连接 WB 阶段的寄存器写使能信号
```

```
565.  assign RF_W = wb_rf_wena;
566.  // 用于连接 ID 阶段的目的寄存器编号
567.  assign rtc = id_rtc;
568.  assign rsc = id_rsc;
569.  // 用于连接 WB 阶段的目的寄存器编号
570.  assign rdc = wb_rdc;
571.  // 用于连接 WB 阶段的目的寄存器数据
572.  assign rd = wb_rd;
573.
574.
575.  // 乘法器相关信号
576.  wire[63:0] mulz, multuz;
577.  wire[31:0] mula, mulb, multua, multub;
578.
579.  // 将执行阶段的乘法器输入连接到乘法器输入 A、B、无符号输入 A、
     B
580.  assign multua = ex_multua;
581.  assign multub = ex_multub;
582.  assign mula = ex_mula;
583.  assign mulb = ex_mulb;
584.
585.  // 除法器相关信号
586.  wire divBusy, divuBusy;
587.  wire divStart, divuStart;
588.  wire[63:0] div_res_lbw,divu_res_lbw;
589.  wire[31:0] divisor, udivisor;
590.  wire[31:0] dividend, udividend, divq, divr, divuq, divur;
591.
592.  // 连接执行阶段的除法器控制信号和输入
593.  assign dividend = ex_div_dividend;
594.  assign divStart = ex_div_start;
595.  assign divuStart = ex_divu_start;
596.  assign udividend = ex_divu_dividend;
597.
598.  assign divisor = ex_div_divisor;
599.  assign udivisor = ex_divu_divisor;
600.
601.
602.
603.  //IF 阶段赋值
604.  assign if_pc = pc_out;
605.  assign if_inst = IM_inst;
606.
607.  // 从指令中提取并零扩展 5 位立即数
```

```
608.  wire[4:0] imm5 = if_id_inst[10:6];
609.  assign id_uext_5 = {27'b0, imm5};
610.  wire[17:0] imm18 = {if_id_inst[15:0], 2'b0};
611.  wire[15:0] imm16 = if_id_inst[15:0];
612.  // 从指令中提取并零扩展 16 位立即数
613.  assign if_sext_16 = imm16[15] == 0 ? {16'b0, imm16} : {16
   'hffff, imm16};
614.  assign id_uext_16 = {16'b0, imm16};
615.  assign id_sext_18 = imm18[17] == 0 ? {14'b0, imm18} : {14
   'b11111111111111, imm18};
616.
617.  // 设置 ID 阶段的程序计数器使能信号为 1
618.  assign id_pc_ena = 1'b1;
619.  // 将 ID 阶段的下一个程序计数器连接到当前程序计数器
620.  assign id_npc = if_id_npc;
621.
622.  // 将 ID 阶段的寄存器编号连接到相应的目的寄存器
623.  assign id_rsc = idRs;
624.  assign id_rtc = idRt;
625.  // 将 ID 阶段的寄存器数据连接到相应的目的寄存器
626.  assign id_rs = rs;
627.  assign id_rt = rt;
628.
629.
630.  assign id_pc_in =((!idOpBne)&&(!idOpBeq)&&(!idOpJ)&&(!idO
   pJal)&&(!idOpJr)
631.                      &&(!idOpEret)&&(!idOpBgez)&&(!idOpJal
   r))?id_npc:
632.                      idOpBeq?id_rs==id_rt?id_npc+id_sext_1
   8:id_npc:
633.                      idOpBne?id_rs==id_rt?id_npc:id_npc+id
   _sext_18:
634.                      idOpJ?{id_npc[31:28],if_id_inst[25:0]
   ,2'b0}:
635.                      idOpJal?{id_npc[31:28],if_id_inst[25:
   0],2'b0}:
636.                      idOpJr?id_rs:idOpEret?id_cp0_epcout:i
   dOpJalr?id_rs:
637.                      idOpBgez?$signed(id_rs)>=0?id_npc+id_
   sext_18:id_npc:32'bz;
638.  assign id_alu_b = idOpClz?id_rs:(idOpAddi||idOpAddiu||idO
   pSltiu||idOpSlti||idOpLw||idOpSw||idOpLb
639.                     ||idOpLbu||idOpLhu||idOpSb||idOpSh||idO
   pLh)?if_sext_16:
```

```
640.                    (idOpAndi||idOpOri||idOpXori||idOpLui)?
    id_uext_16:(idOpJal||idOpJalr)?32'd0:id_rt;
641.
642.  reg[31:0] alu_c;
643.  assign id_alu_a = (idOpLui||idOpClz)?32'bz:(idOpJal||idOp
    Jalr)?id_npc:(idOpSll||idOpSra||idOpSrl)?
644.                  id_uext_5:id_rs;
645.
646.  wire[7:0] alu_d;
647.  assign id_cp0_raddr = if_id_inst[15:11];
648.  assign id_pass_data = idOpMfc0?id_cp0_rdata:idOpMfhi?HI:i
    dOpMflo?LO:(idOpMtc0||idOpSw||idOpSb||idOpSh)?id_rt:
649.                        (idOpMthi||idOpMtlo)?id_rs:32'bz;
650.
651.
652.  //EX
653.  reg div_start_reg,divu_start_reg;
654.
655.  always@(*)begin
656.      divu_start_reg = 0;
657.      div_start_reg = 0;
658.      if(exOpDivu&&!stall[`STAGE_EX]&&!divuBusy)begin
659.          divu_start_reg = 1;
660.      end
661.      if(exOpDiv&&!stall[`STAGE_EX]&&!divBusy)begin
662.          div_start_reg = 1;
663.      end
664.  end
665.
666.
667.
668.  // 将 ID 阶段的 ALU 输入连接到执行阶段的 ALU 输入 A 和 B
669.  assign ex_alua = id_ex_alua;
670.  assign ex_alub = id_ex_alub;
671.
672.  // 将 ALU 的输出连接到执行阶段的 ALU 输出
673.  assign ex_aluo = aluo;
674.
675.  // 将 ID 阶段的乘法器输入连接到执行阶段的乘法器输入 A 和 B
676.  assign ex_mula = id_ex_alua;
677.  assign ex_mulb = id_ex_alub;
678.  // 将执行阶段的乘法器输出和无符号输出连接到相应的输出信号
679.  assign ex_mulz = mulz;
680.  assign ex_multuz = multuz;
```

```
681.  assign ex_multua = id_ex_alua;
682.  assign ex_multub = id_ex_alub;
683.
684.  // 将除法器的繁忙状态和启动信号连接到执行阶段的相应信号
685.  assign ex_divu_busy = divuBusy;
686.  assign ex_div_start = div_start_reg;
687.  assign ex_divu_start = divu_start_reg;
688.  assign ex_div_busy = divBusy;
689.  // 将 ID 阶段的 ALU 输入连接到执行阶段的除法器输入和无符号除
     法器输入
690.  assign ex_div_dividend = id_ex_alua;
691.  assign ex_div_divisor = id_ex_alub;
692.  // 将除法器的输出结果和无符号输出结果连接到相应的输出信号
693.  assign ex_divu_dividend = id_ex_alua;
694.  assign ex_divu_divisor = id_ex_alub;
695.  assign ex_div_q = divq;
696.  assign ex_div_r = divr;
697.  assign ex_divu_q = divuq;
698.  assign ex_divu_r = divur;
699.
700.
701.  // MEM 阶段
702.
703.  // 判断是否为读操作
704.  assign mem_dm_r = meOpLw || meOpLb || meOpLbu || meOpLh |
     | meOpLhu;
705.  // 判断是否为写操作
706.  assign mem_dm_w = meOpSw || meOpSb || meOpSh;
707.  assign mem_dm_rdata = DM_rdata;
708.  // 将数据存储器的读取数据和执行阶段传递的数据连接到 MEM 阶段的
     数据输入
709.  assign mem_dm_wdata = ex_mem_pass_data;
710.  // 根据指令类型设置字节使能信号
711.  assign mem_byte_ena = meOpSw ? 4'b1111 : meOpSb ? 4'b0001
      : meOpSh ? 4'b0011 : 4'b0000;
712.
713.  // 对于 lb 操作，进行符号或零扩展
714.  assign mem_byte_ext = meOpLb ? (mem_dm_rdata[7] == 0 ? {2
     4'b0, mem_dm_rdata[7:0]} : {24'hffffff, mem_dm_rdata[7:0]})
715.                               : meOpLbu ? {24'b0, mem_dm_rdat
     a[7:0]} : 32'bz;
716.  // 对于 lh 操作，进行符号或零扩展
```

```
717.  assign mem_half_ext = meOpLh ? (mem_dm_rdata[15] == 0 ? {
   16'b0, mem_dm_rdata[15:0]} : {16'hffff, mem_dm_rdata[15:0]}
   )
718.                               : meOpLhu ? {16'b0, mem_dm_rda
   ta[15:0]} : 32'bz;
719.  // 将执行阶段的 ALU 输出作为 MEM 阶段的数据存储器地址
720.  assign mem_dm_addr = ex_mem_aluo;
721.
722.
723.  // 写回阶段的寄存器文件写使能信号
724.  assign wb_rf_wena = (!mem_wb_overflowFlag && wbOpAddi) ||
    wbOpAndi || wbOpAddiu || wbOpOri || wbOpSltiu || wbOpLui |
   | wbOpXori || wbOpSlti
725.                       || wbOpAddu || wbOpJal || wbOpAnd ||
   wbOpLw || wbOpXor || wbOpNor || wbOpOr || wbOpSll || wbOpSl
   lv || wbOpSltu
726.                       || wbOpSra || wbOpSrl || wbOpSubu ||
   (wbOpAdd && !mem_wb_overflowFlag) || wbOpSub || 0 || wbOpSl
   t || wbOpSrlv || wbOpSrav || wbOpClz
727.                       || wbOpJalr || wbOpLb || wbOpLbu || w
   bOpLhu || wbOpLh || wbOpMfc0 || wbOpMfhi || wbOpMflo || wbO
   pMul;
728.
729.  wire[31:0]wb_tmp0;
730.  // 写回阶段的目的寄存器编号
731.  assign wb_rdc = (wbOpAddi || wbOpAddiu || wbOpAndi || wbO
   pOri || wbOpSltiu || wbOpLui || wbOpXori || wbOpSlti || wbO
   pLw
732.                   || wbOpLb || wbOpLbu || wbOpLhu || wbOpLh
    || wbOpMfc0) ? wbRt : (wbOpAddu || wbOpAnd || wbOpXor || w
   bOpNor
733.                   || wbOpOr || wbOpSll || wbOpSllv || wbOpS
   ltu || wbOpSra || wbOpSrl || wbOpSubu || wbOpAdd || wbOpSub
    || wbOpSlt
734.                   || wbOpSrlv || wbOpSrav || wbOpClz || (wb
   OpJalr && wbRd != 5'b0) || wbOpMfhi || wbOpMflo || wbOpMul)
    ? wbRd : (wbOpJal || (wbOpJalr && wbRd == 5'b0)) ? 5'd31 :
    32'bz;
735.  wire[31:0] wb_tmp1;
736.  // 写回阶段的 CP0 写地址
737.  assign wb_cp0_waddr = wbOpMtc0 ? wbRd : 5'bz;
738.
739.  // 写回阶段的目的寄存器数据
```

```
740.  assign wb_rd = ((wbOpAddi && !mem_wb_overflowFlag) || wbO
   pAddiu || wbOpAndi || wbOpOri || wbOpSltiu || wbOpLui || wb
   OpXori || wbOpSlti
741.                       || wbOpAddu || wbOpJal || wbOpAnd ||
   wbOpXor || wbOpNor || wbOpOr || wbOpSll || wbOpSllv || wbOp
   Sltu
742.                       || wbOpSra || wbOpSrl || wbOpSubu ||
   (wbOpAdd && !mem_wb_overflowFlag) || wbOpSub || wbOpSlt ||
   wbOpSrlv || wbOpSrav || wbOpClz
743.                       || wbOpJalr) ? mem_wb_aluo : (wbOpLw
   || wbOpLb || wbOpLbu || wbOpLhu || wbOpLh || wbOpMfc0 || wb
   OpMfhi || wbOpMflo) ? mem_wb_pass_data :
744.                       (wbOpMul) ? mem_wb_mulz[31:0] : 32'bz
   ;
745.  wire[31:0]wb_tmp2;
746.  // 写回阶段的 CP0 写数据
747.  assign wb_cp0_wdata = wbOpMtc0 ? mem_wb_pass_data : 32'bz
   ;
748.
749.
750.  // 写回阶段的 HI 寄存器数据
751.  assign wb_hi_wdata = (wbOpDivu || wbOpDiv) ? mem_wb_div_r
    : wbOpMultu ? mem_wb_mulz[63:32] : wbOpMthi ? mem_wb_pass_
   data : 32'bz;
752.
753.  wire wb_tmp5;
754.  assign wb_tmp5 = 1;
755.  // 写回阶段的 HI 寄存器写使能
756.  assign wb_hi_wena = wbOpDivu || wbOpDiv || wbOpMultu || w
   bOpMthi;
757.
758.  // 写回阶段的 LO 寄存器写使能
759.  assign wb_lo_wena = wbOpDivu || wbOpDiv || wbOpMultu || w
   bOpMtlo;
760.  // 写回阶段的 LO 寄存器数据
761.  assign wb_lo_wdata = (wbOpDivu || wbOpDiv) ? mem_wb_div_q
    : wbOpMultu ? mem_wb_mulz[31:0] : wbOpMtlo ? mem_wb_pass_d
   ata : 32'bz;
762.
763.  wire wRegId,wRegEx,wRegMe,rRsId,rRtId;
764.
765.  wire[4:0] wAddrId,wAddrEx,wAddrMe;
766.  wire[4:0] wb_cnt0;
767.  wire wb_kase;
```

```
768.  assign wb_kase = 0;
769.
770.  //写会寄存器选择信号
771.
772.  assign wRegEx = exOpAddi||exOpAddiu||exOpAndi||exOpOri||e
   xOpSltiu||exOpLui||exOpXori||exOpSlti
773.                        ||exOpAddu||exOpJal||exOpAnd||exOpLw|
   |exOpXor||exOpNor||exOpOr||wb_kase||exOpSll||exOpSllv||exOp
   Sltu
774.                        ||exOpSra||exOpSrl||exOpSubu||exOpAdd
   ||exOpSub||exOpSlt||exOpSrlv||exOpSrav||exOpClz
775.                        ||exOpJalr||exOpLb||exOpLbu||exOpLhu|
   |exOpLh||exOpMfc0||exOpMfhi||exOpMflo||exOpMul;
776.
777.  wire[31:0] wb_a;
778.  assign wRegId = idOpMul||idOpAddi||idOpAddiu||idOpAndi||i
   dOpOri||idOpSltiu||idOpLui||idOpSrav||idOpXori||idOpSlti
779.                        ||idOpAddu||idOpJal||idOpAnd||idOpLw|
   | wb_kase||idOpXor||idOpNor||idOpOr||idOpSll||idOpSllv||idO
   pSltu
780.                        ||idOpSra||idOpSrl||idOpSubu||idOpAdd
   ||idOpSub||idOpSlt||idOpSrlv||idOpClz
781.                        ||idOpJalr||idOpLb|| wb_kase||idOpLbu
   ||idOpLhu||idOpLh||idOpMfc0||idOpMfhi||idOpMflo;
782.
783.  assign rRsId = idOpAdd||idOpAddi||idOpAddiu||idOpAddu||id
   OpAnd||idOpAndi||idOpBeq||idOpBgez||idOpBne||idOpClz||idOpD
   iv||idOpDivu||idOpJalr||idOpJr||idOpLb
784.                     ||idOpLbu||idOpLh||idOpLhu||idOpLw||idOpM
   thi||idOpMtlo||idOpMul||idOpMultu||idOpNor||idOpOr||idOpOri
   ||idOpSb||idOpSh||idOpSllv
785.                     ||idOpSlt||idOpSlti||idOpSltiu||idOpSltu|
   |idOpSrav||idOpSrlv||idOpSub||idOpSubu||idOpSw||idOpXor||id
   OpXori;
786.  wire[31:0] wb_b;
787.  assign wRegMe = (meOpAddi&&!ex_mem_overflowFlag)||meOpAdd
   iu||wb_kase||meOpAndi||meOpOri||meOpSltiu||meOpLui||meOpXor
   i||meOpSlti
788.                                                ||meOpAddu||meOpJ
   al||meOpAnd||meOpLw||meOpXor||meOpNor||meOpOr||meOpSll||meO
   pSllv||meOpSltu
789.                                                ||meOpSra||meOpSr
   l||meOpSubu||(meOpAdd&&!ex_mem_overflowFlag)||meOpSub||meOp
   Slt||meOpSrlv||meOpSrav||meOpClz
```

```
790.                                                ||meOpJalr||meOpL
    b||wb_kase||meOpLbu||meOpLhu||meOpLh||meOpMfc0||meOpMfhi||m
    eOpMflo||meOpMul;
791.  wire wRs_id;
792.  assign rRtId = idOpSlt||idOpAdd||idOpAddu||idOpAnd||idOpB
    eq||idOpBne||idOpDiv||idOpDivu||idOpMul||idOpMultu||idOpNor
    ||idOpOr||idOpSb||idOpSh||idOpSllv
793.                  ||idOpSltu||idOpSrav||idOpSrlv||idOpSub||i
    dOpSubu||idOpSw||idOpXor
794.                  ||idOpMtc0||idOpSll||idOpSra||idOpSrl||wb_
    kase;
795.
796.  //
797.  assign wAddrEx = (exOpAddi||exOpAddiu||exOpAndi||exOpOri|
    |exOpSltiu||exOpLui||exOpXori||exOpSlti||exOpLw
798.                    ||exOpLb||exOpLbu||exOpLhu||exOpLh||exOpM
    fc0)?exRt:(exOpAddu||exOpAnd||exOpXor||exOpNor
799.                    ||exOpOr||exOpSll||exOpSllv||exOpSltu||ex
    OpSra||exOpSrl||exOpSubu||exOpAdd||exOpSub||exOpSlt
800.                    ||exOpSrlv||exOpSrav||exOpClz||(exOpJalr&
    &exRd!=5'b0)||exOpMfhi||exOpMflo||exOpMul)?exRd:(exOpJal||(
    exOpJalr&&exRd==5'b0))?5'd31:32'bz;
801.  //
802.  assign wAddrId = (idOpAddi||wb_kase||idOpAddiu||idOpAndi|
    |idOpOri||idOpSltiu||idOpLui||idOpXori||idOpSlti||idOpLw
803.                    ||idOpLb||idOpLbu||idOpLhu||idOpLh||idOpM
    fc0)?idRt:(idOpAddu||idOpAnd||idOpXor||idOpNor
804.                    ||idOpOr||idOpSll||idOpSllv||idOpSltu||id
    OpSra||idOpSrl||idOpSubu||idOpAdd||idOpSub||idOpSlt
805.                    ||idOpSrlv||idOpSrav||wb_kase||idOpClz||(
    idOpJalr&&if_id_inst[15:11]!=5'b0)||idOpMfhi||idOpMflo||idO
    pMul)?if_id_inst[15:11]:(idOpJal||(idOpJalr&&if_id_inst[15:
    11]==5'b0))?5'd31:32'bz;
806.  //
807.  assign wAddrMe = (meOpAddi||meOpAddiu||meOpAndi||wb_kase|
    |meOpOri||meOpSltiu||meOpLui||meOpXori||meOpSlti||meOpLw
808.                                    ||meOpLb||meOpLbu||meOpLh
    u||meOpLh||meOpMfc0)?meRt:(meOpAddu||meOpAnd||meOpXor||meOp
    Nor
809.                                    ||meOpOr||meOpSll||meOpSl
    lv||meOpSltu||meOpSra||meOpSrl||meOpSubu||meOpAdd||meOpSub|
    |meOpSlt
810.                                    ||meOpSrlv||meOpSrav||wb_
    kase||meOpClz||(meOpJalr&&meRd!=5'b0)||meOpMfhi||meOpMflo||
```

```
   meOpMul)?meRd:(meOpJal||(meOpJalr&&meRd==5'b0))?5'd31:32'bz
   ;
811.
812.
813.  //处理竞争冒险
814.  always @ (*) begin
815.      stall = `CTRL_STALLW'b0;
816.      // 检查 IF 阶段的冒险
817.      if (ifOpJr || ifOpBne || ifOpBeq || ifOpBgez||ifOpJal
   r) begin
818.          if (wRegEx && (wAddrEx == ifRs || ((ifOpBne || if
   OpBeq) && wAddrEx == ifRt))) begin
819.              stall = `CTRL_STALL_IF;
820.          end
821.          if (wRegId && (wAddrId == ifRs || ((ifOpBne || if
   OpBeq) && wAddrId == ifRt))) begin
822.              stall = `CTRL_STALL_IF;
823.          end
824.          if (wRegMe && (wAddrMe == ifRs || ((ifOpBne || if
   OpBeq) && wAddrMe == ifRt))) begin
825.              stall = `CTRL_STALL_IF;
826.          end
827.      end
828.      // 检查 ID 阶段的冒险
829.      if (wRegEx) begin
830.          if (rRsId && wAddrEx == idRs ||  wAddrEx == idRt
   && rRtId ) begin
831.              stall = `CTRL_STALL_ID;
832.          end
833.      end
834.
835.      // 检查涉及协处理器指令的冒险
836.      if ((exOpMthi || exOpDiv || exOpMultu|| exOpDivu) &&
   idOpMfhi) begin
837.          stall = `CTRL_STALL_ID;
838.      end
839.      if ((exOpMtlo || exOpDiv ||  exOpMultu|| exOpDivu) &&
    idOpMflo) begin
840.          stall = `CTRL_STALL_ID;
841.      end
842.      if (idOpMfc0 && exOpMtc0) begin
843.          stall = `CTRL_STALL_ID;
844.      end
845.
```

```
846.      // 检查ME阶段涉及协处理器指令的冒险
847.      if ((meOpMthi || meOpMultu || meOpDiv || meOpDivu) &&
    idOpMfhi) begin
848.          stall = `CTRL_STALL_ID;
849.      end
850.      if ((meOpMtlo || meOpMultu || meOpDiv || meOpDivu) &&
    idOpMflo) begin
851.          stall = `CTRL_STALL_ID;
852.      end
853.      if (meOpMtc0 && idOpMfc0) begin
854.          stall = `CTRL_STALL_ID;
855.      end
856.
857.      // 检查ME阶段的冒险
858.      if (wRegMe) begin
859.          if (rRsId && wAddrMe == idRs || rRtId && wAddrMe
   == idRt) begin
860.              stall = `CTRL_STALL_ID;
861.          end
862.      end
863.      // 检查EX阶段涉及除法指令的冒险
864.      if ((exOpDiv && divBusy) || (exOpDivu && divuBusy)) b
   egin
865.          stall = `CTRL_STALL_EX;
866.      end
867.  end
868.
869.
870.   // IF/ID 寄存器更新
871.  always @(posedge clk or posedge rst) begin
872.      if (rst) begin
873.          // 复位时，将 IF/ID 寄存器清零
874.          if_id_npc <= 32'b0;
875.          if_id_inst <= 32'b0;
876.      end
877.      else if (stall[`STAGE_IF] && (!stall[`STAGE_ID])) beg
   in
878.          // 如果有流水线暂停信号，清零 IF/ID 寄存器
879.          if_id_inst <= 32'b0;
880.      end
881.      else if (!stall[`STAGE_IF]) begin
882.          // 如果没有流水线暂停信号，则更新 IF/ID 寄存器
883.          if_id_inst <= if_inst;
884.
```

```
885.          // 根据指令类型更新下一条指令地址
886.          if (ifOpSyscall || ifOpTeq || ifOpBreak) begin
887.              if_id_npc <= 32'h00400004;
888.          end
889.          else begin
890.              if_id_npc <= if_pc + 32'h4;
891.          end
892.      end
893.  end
894.
895.
896.  // ID_EX 寄存器更新
897.  always @ (posedge clk or posedge rst) begin
898.      if (rst) begin
899.          // 复位时，将 ID_EX 寄存器清零
900.          id_ex_pass_data <= 32'b0;
901.          id_ex_inst <= 32'b0;
902.          id_ex_alua <= 32'b0;
903.          id_ex_alub <= 32'b0;
904.      end
905.      else if (!stall[`STAGE_EX] && stall[`STAGE_ID]) begin
906.          // 如果有流水线暂停信号，清零 ID_EX 寄存器
907.          id_ex_alua <= 32'b0;
908.          id_ex_alub <= 32'b0;
909.          id_ex_pass_data <= 32'b0;
910.          id_ex_inst <= 32'b0;
911.      end
912.      else if (!stall[`STAGE_ID]) begin
913.          // 如果没有流水线暂停信号，则更新 ID_EX 寄存器
914.          id_ex_alua <= id_alu_a;
915.          id_ex_alub <= id_alu_b;
916.          id_ex_pass_data <= id_pass_data;
917.          id_ex_inst <= if_id_inst;
918.      end
919.  end
920.
921.
922.  // EX_MEM 寄存器更新
923.  always @ (posedge clk or posedge rst) begin
924.      if (rst) begin
925.          // 复位时，将 EX_MEM 寄存器清零
926.          ex_mem_div_r <= 32'b0;
927.          ex_mem_mulz <= 64'b0;
928.          ex_mem_aluo <= 32'b0;
```

```
929.
930.          ex_mem_inst <= 32'b0;
931.          ex_mem_overflowFlag <= 1'b0;
932.          ex_mem_div_q <= 32'b0;
933.          ex_mem_pass_data <= 32'b0;
934.
935.      end
936.      else if (stall[`STAGE_EX] && !stall[`STAGE_ME]) begin
937.          // 如果有流水线暂停信号，清零 EX_MEM 寄存器
938.          ex_mem_div_r <= 32'b0;
939.          ex_mem_mulz <= 64'b0;
940.          ex_mem_aluo <= 32'b0;
941.          ex_mem_pass_data <= 32'b0;
942.          ex_mem_div_q <= 32'b0;
943.          ex_mem_inst <= 32'b0;
944.          ex_mem_overflowFlag <= 1'b0;
945.      end
946.      else if (!stall[`STAGE_EX]) begin
947.          // 如果没有流水线暂停信号，则更新 EX_MEM 寄存器
948.          ex_mem_div_r <= exOpDiv ? ex_div_r : exOpDivu ? e
    x_divu_r : 32'b0;
949.          ex_mem_mulz <= exOpMul ? ex_mulz : exOpMultu ? ex
    _multuz : 64'b0;
950.          ex_mem_div_q <= exOpDiv ? ex_div_q : exOpDivu ? e
    x_divu_q : 32'b0;
951.
952.          ex_mem_pass_data <= id_ex_pass_data;
953.          ex_mem_inst <= id_ex_inst;
954.          ex_mem_aluo <= ex_aluo;
955.          ex_mem_overflowFlag <= overflowFlag;
956.      end
957.  end
958.
959.  // MEM_WB 寄存器更新
960.  always @ (posedge clk or posedge rst) begin
961.      if (rst) begin
962.          // 复位时，将 MEM_WB 寄存器清零
963.          mem_wb_div_q <= 32'b0;
964.          mem_wb_div_r <= 32'b0;
965.          mem_wb_aluo <= 32'b0;
966.          mem_wb_mulz <= 64'b0;
967.          mem_wb_pass_data <= 32'b0;
968.          mem_wb_inst <= 32'b0;
969.          mem_wb_overflowFlag <= 32'b0;
```

```
970.      end
971.      else if (stall[`STAGE_ME] && !stall[`STAGE_WB]) begin
972.          // 如果有流水线暂停信号，清零 MEM_WB 寄存器
973.          mem_wb_div_q <= 32'b0;
974.          mem_wb_div_r <= 32'b0;
975.          mem_wb_aluo <= 32'b0;
976.          mem_wb_mulz <= 64'b0;
977.          mem_wb_pass_data <= 32'b0;
978.          mem_wb_inst <= 32'b0;
979.          mem_wb_overflowFlag <= 32'b0;
980.      end
981.      else if (!stall[`STAGE_ME]) begin
982.          // 如果没有流水线暂停信号，则更新 MEM_WB 寄存器
983.          mem_wb_mulz <= ex_mem_mulz;
984.          mem_wb_div_q <= ex_mem_div_q;
985.          mem_wb_div_r <= ex_mem_div_r;
986.          mem_wb_aluo <= ex_mem_aluo;
987.          if (meOpLw) begin
988.              mem_wb_pass_data <= mem_dm_rdata;
989.          end
990.          else if (meOpLb || meOpLbu) begin
991.              mem_wb_pass_data <= mem_byte_ext;
992.          end
993.          else if (meOpLh || meOpLhu) begin
994.              mem_wb_pass_data <= mem_half_ext;
995.          end
996.          else begin
997.              mem_wb_pass_data <= ex_mem_pass_data;
998.          end
999.          mem_wb_inst <= ex_mem_inst;
1000.         mem_wb_overflowFlag <= ex_mem_overflowFlag;
1001.     end
1002. end
1003.
1004.
1005. //HI,LO
1006. always @ (negedge clk or posedge rst) begin
1007.     if(rst)begin
1008.         LO<=32'b0;
1009.     end
1010.     else if(wb_lo_wena)begin
1011.         LO<=wb_lo_wdata;
1012.     end
1013. end
```

```
1014.
1015. always @ (negedge clk or posedge rst) begin
1016.     if(rst)begin
1017.         HI<=32'b0;
1018.     end
1019.     else if(wb_hi_wena)begin
1020.         HI<=wb_hi_wdata;
1021.     end
1022. end
1023.
1024.
1025.
1026.
1027.
1028.  CP0 cp0(
1029.     .clk(clk),
1030.     .rst(rst),
1031.     .mfc0(idOpMfc0),
1032.     .mtc0(wbOpMtc0),
1033.     .eret(idOpEret),
1034.     .exception(exception),
1035.     .cause(cause),
1036.     .addr(cp0_addr),
1037.     .wdata(wb_cp0_wdata),
1038.     .pc(id_npc),
1039.     .rdata(id_cp0_rdata),
1040.     .status(id_cp0_status),
1041.     .exc_addr(id_cp0_epcout)
1042. );
1043.
1044. PCReg pcreg(
1045.    .clk(clk),
1046.    .rst(rst),
1047.    .ena(pc_ena),
1048.    .pc_in(pc_in),
1049.    .pc_out(pc_out)
1050. );
1051.
1052.  ALU cpu_alu(
1053.     .a(alua),
1054.     .b(alub),
1055.     .aluc(aluc),
1056.     .r(aluo),
1057.     .zero(zeroFlag),
```

```
1058.     .carry(carryFlag),
1059.     .negative(negFlag),
1060.     .overflow(overflowFlag)
1061.     );
1062.
1063. RegFile cpu_ref(
1064.     .RF_ena(1'b1),
1065.     .RF_rst(rst),
1066.     .RF_clk(clk),
1067.     .Rdc(rdc),
1068.     .Rsc(rsc),
1069.     .Rtc(rtc),
1070.     .Rd(rd),
1071.     .Rs(rs),
1072.     .Rt(rt),
1073.     .RF_W(RF_W)
1074.     );
1075.
1076. DIV cpu_div(
1077.     .dividend(dividend),
1078.     .divisor(divisor),
1079.     .div_res_q(divq),
1080.     .div_res_r(divr),
1081.     .div_busy(divBusy),
1082.     .divu_res_q(divuq),
1083.     .divu_res_r(divur),
1084.     .divu_busy(divuBusy)
1085. );
1086.
1087. MUL cpu_mul(
1088.     .a(multa),
1089.     .b(multb),
1090.     .mult_res(mulz),
1091.     .multu_res(multuz)
1092. );
1093.
1094.
1095.
1096. endmodule
```

（3）define_cpu.vh:

```
1.
   //----------Instruction----------
2.
3. `define INSTR_INDEX          25:0
4. `define INSTR_RS             25:21
5. `define INSTR_RT             20:16
6. `define INSTR_RD             15:11
7. `define INSTR_SA             10:6
8. `define INSTR_OFFSET         15:0
9. `define INSTR_IMM            15:0
10.
11.
12.//----------Operation----------
13.
14.`define OP_ADDI              6'b001000
15.`define OP_ADDIU             6'b001001
16.`define OP_ANDI              6'b001100
17.`define OP_JAL               6'b000011
18.`define OP_LUI               6'b001111
19.`define OP_LB                6'b100000
20.`define OP_LBU               6'b100100
21.`define OP_LH                6'b100001
22.`define OP_LHU               6'b100101
23.`define OP_BEQ               6'b000100
24.`define OP_BNE               6'b000101
25.`define OP_COP0              6'b010000
26.`define OP_J                 6'b000010
27.
28.
29.`define OP_SLTIU             6'b001011
30.`define OP_SPECIAL           6'b000000
31.`define OP_SPECIAL2          6'b011100
32.`define OP_SB                6'b101000
33.`define OP_LW                6'b100011
34.`define OP_ORI               6'b001101
35.`define OP_REGIMM            6'b000001
36.`define OP_SLTI              6'b001010
37.`define OP_SH                6'b101001
38.`define OP_SW                6'b101011
39.`define OP_XORI              6'b001110
40.
41.`define FUNCT_ADD            6'b100000
42.`define FUNCT_CLZ            6'b100000
43.`define FUNCT_DIV            6'b011010
```

```
`define FUNCT_DIVU          6'b011011
`define FUNCT_ADDU          6'b100001
`define FUNCT_AND           6'b100100
`define FUNCT_BREAK         6'b001101
`define FUNCT_ERET          6'b011000
`define FUNCT_JALR          6'b001001
`define FUNCT_JR            6'b001000
`define FUNCT_MFHI          6'b010000
`define FUNCT_MFLO          6'b010010
`define FUNCT_MTHI          6'b010001
`define FUNCT_MTLO          6'b010011
`define FUNCT_MUL           6'b000010
`define FUNCT_MULTU         6'b011001
`define FUNCT_NOR           6'b100111
`define FUNCT_OR            6'b100101
`define FUNCT_SLL           6'b000000
`define FUNCT_SRAV          6'b000111
`define FUNCT_SRL           6'b000010
`define FUNCT_SRLV          6'b000110
`define FUNCT_SUB           6'b100010
`define FUNCT_SUBU          6'b100011
`define FUNCT_SYSCALL       6'b001100
`define FUNCT_TEQ           6'b110100
`define FUNCT_SLLV          6'b000100
`define FUNCT_SLT           6'b101010
`define FUNCT_SLTU          6'b101011
`define FUNCT_SRA           6'b000011
`define FUNCT_XOR           6'b100110

`define RS_MF               5'b00000
`define RS_MT               5'b00100
`define RT_BGEZ             5'b00001

//----------ALU----------

`define ALUCW               5   //control signal width

`define ALU_OP_NOP          5'b00000
`define ALU_OP_ADDU         5'b00001
`define ALU_OP_ADD          5'b00010
`define ALU_OP_SLTU         5'b01011
`define ALU_OP_SRA          5'b01100
`define ALU_OP_SUBU         5'b00011
`define ALU_OP_SUB          5'b00100
```

```
88.`define ALU_OP_AND           5'b00101
89.`define ALU_OP_OR            5'b00110
90.`define ALU_OP_LUI           5'b01001
91.`define ALU_OP_SLT           5'b01010
92.`define ALU_OP_SLL_SLA       5'b01101
93.`define ALU_OP_XOR           5'b00111
94.`define ALU_OP_NOR           5'b01000
95.`define ALU_OP_SRL           5'b01110
96.`define ALU_OP_CLZ           5'b01111
97.`define ALU_OP_PASSA         5'b10000
98.`define ALU_OP_PASSB         5'b10001
99.
100.
101. //----------CP0----------
102.
103. `define CP0_CAUSE_SYSCALL   32'b1000
104. `define CP0_CAUSE_BREAK     32'b1001
105. `define CP0_CAUSE_TEQ       32'b1101
106.
107. `define CP0_STATUS_ADDR     12
108. `define CP0_CAUSE_ADDR      13
109. `define CP0_EPC_ADDR        14
110.
111. `define CP0_SYSCALL_POS     1
112. `define CP0_BREAK_POS       2
113. `define CP0_TEQ_POS         3
114.
115. `define CP0_STATUS_INIT     32'b1111
116.
117.
118.
119. //----------CtrlUnit----------
120.
121. `define CTRL_STALLW         5
122. `define CTRL_STALL_ALL      5'b11111
123. `define CTRL_STALL_IF       5'b00001
124. `define CTRL_STALL_ID       5'b00011
125. `define CTRL_STALL_EX       5'b00111
126.
127. //----------Others----------
128.
129. `define STAGE_IF            0
130. `define STAGE_ID            1
131. `define STAGE_EX            2
```

```
132.  `define STAGE_ME            3
133.  `define STAGE_WB            4
```

（4）DIV.v:

```
1. `timescale 1ns / 1ps
2.
3. module DIV(
4.     input [31:0]a,
5.     input [31:0]b,
6.     output[31:0]div_res_q,
7.     output[31:0]div_res_r,
8.     output[31:0]divu_res_q,
9.     output[31:0]divu_res_r,
10.    output div_busy,
11.    output divu_busy
12.    );
13.    wire signed[31:0]signed_a,signed_b;
14.    assign signed_a = a;
15.    assign signed_b = b;
16.    wire[31:0] signed_q,signed_r,unsigned_q,unsigned_r;
17.
18.    assign signed_r = (b == 32'b0)?32'b0:(signed_a%signed_b
   );
19.    assign signed_q = (b == 32'b0)?32'b0:(signed_a/signed_b
   );
20.
21.    assign unsigned_r = (b == 32'b0)?32'b0:(a%b);
22.    assign unsigned_q = (b == 32'b0)?32'b0:(a/b);
23.
24.    assign div_res_q = signed_q;
25.    assign div_res_r = signed_r;
26.    assign divu_res_q = unsigned_q;
27.    assign divu_res_r = unsigned_r;
28.    assign div_busy = 0;
29.    assign divu_busy = 0;
30.endmodule
```

（5）MUL.v:

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

```
3.
4. module MUL(
5.     input [31:0]a,//乘数 a
6.     input [31:0]b,//乘数 b
7.     output [63:0]mult_res,//有符号乘
8.     output [63:0]multu_res//无符号乘
9.     );
10.  wire [63:0] unsigned_a,unsigned_b;
11.  wire signed [63:0] signed_a,signed_b;
12.  assign unsigned_a = {32'b0,a};
13.  assign unsigned_b = {32'b0,b};
14.
15.  assign signed_a = {{32{a[31]}},a};
16.  assign signed_b = {{32{b[31]}},b};
17.
18.  assign mult_res = signed_a*signed_b;
19.  assign multu_res = unsigned_a*unsigned_b;
20.endmodule
```

（7） IMEM.v:

```
1. `timescale 1ns / 1ps
2.
3. module IMEM(
4.     input [10:0] addr,
5.     output [31:0] instr
6.     );
7.
8.     dist_mem_gen_0 instr_mem(
9.         .a(addr),
10.        .spo(instr)
11.    );
12.endmodule
```

（8） DMEM.v:

```
1. `timescale 1ns / 1ps
2. module DMEM(
3.     input clk,
4.     input rst,
5.     input ena,
6.     input DM_W,
7.     input DM_R,
8.     input[3:0]byteEna,
```

```
    input [31:0] DM_addr,
    input [31:0] DM_wdata,
    output [31:0] DM_rdata
    );
    wire[31:0]addr;
    reg [7:0] D_mem[0:1023];
    wire [10:0]addr0,addr1,addr2,addr3;
    wire[7:0]wByte0,wByte1,wByte2,wByte3;

    assign addr=DM_addr;
    assign addr0={addr[10:0]};
    assign addr1=addr0+11'd1;
    assign addr2=addr0+11'd2;
    assign addr3=addr0+11'd3;


    assign wByte0=byteEna[0]?DM_wdata[7:0]:D_mem[addr0];
    assign wByte1=byteEna[1]?DM_wdata[15:8]:D_mem[addr1];
    assign wByte2=byteEna[2]?DM_wdata[23:16]:D_mem[addr2];
    assign wByte3=byteEna[3]?DM_wdata[31:24]:D_mem[addr3];

    always @(negedge clk) begin
        if(rst)
        begin
        //
        end
        else if (DM_W && ena) begin
            D_mem[addr0] <= wByte0;
            D_mem[addr1] <= wByte1;
            D_mem[addr2] <= wByte2;
            D_mem[addr3] <= wByte3;
        end
    end

    assign DM_rdata[31:0] = (DM_R && ena) ? {D_mem[addr3],D
  _mem[addr2],D_mem[addr1],D_mem[addr0]} : 32'bz;
endmodule
```

（9） PCreg.v:

```
`timescale 1ns / 1ps

module PCReg(
```

```
    input clk,
    input rst,
    input ena,
    input [31:0]pc_in,
    output  reg[31:0] pc_out
    );
    always@(negedge clk or posedge rst)
    begin
        if(rst)pc_out<=32'h00400000;//这是 MARS 中取指令的开始地址
        else if(rst==0&&ena)pc_out<=pc_in;
    end
endmodule
```

（10）RegFile.v:

```
`timescale 1ns / 1ps

module RegFile(
    input RF_ena,
    input clk,
    input rst,
    input [4:0] Rdc,
    input [4:0] Rsc,
    input [4:0] Rtc,
    input [31:0] Rd,
    output [31:0] Rs,
    output [31:0] Rt,
    input RF_W
    );
    reg [31:0] array_reg[0:31];//定义 RegFile 中的寄存器
    integer i;
    always @(negedge clk or posedge rst) begin
        if (rst) begin
            for(i=0;i<32;i=i+1)array_reg[i]<=32'b0;
            //rst 信号发挥作用，全部寄存器内容清空
        end
        else begin
             //写信号且不能改变寄存器 0 中的内容
            if (RF_W == 1'b1 && RF_ena && Rdc != 5'b0)
                array_reg[Rdc] <= Rd;
        end
    end

```

```
    assign Rs = RF_ena ? array_reg[Rsc] : 32'bz;
    assign Rt = RF_ena ? array_reg[Rtc] : 32'bz;
endmodule

```

（11）sccomp_dataflow.v:

```
`timescale 1ns / 1ps

module sccomp_dataflow(
    input clk_in,
    input rst,
    output [7:0]o_seg,
    output [7:0]o_sel
    );

// 声明指令内存数据的线
wire [31:0] inst;
// 声明程序计数器的线
wire [31:0] pc;
// 声明数据存储器写和读的信号线
wire dw, dr;
// 声明写和读数据的线
wire [31:0] w_data, r_data;
// 声明指令的线
wire [31:0] instr;
// 声明数据存储器和指令存储器地址的线
wire [31:0] dm_addr;
wire [31:0] im_addr;
// 声明ALU结果的线
wire [31:0] alu_res;
// 声明时钟信号的线
wire clk;
// 声明字节使能信号的线
wire [3:0] byteEna;
// 将指令线赋值给inst
assign inst = instr;

assign dm_addr =(alu_res - 32'h1001_0000);
//这里不同于cpu31，我们选择DMEM存储数据采用标准的8位是为了方便LB等取字节指令的执行

//ROM
```

```
IMEM rom(
    .addr(im_addr[10:0]),
    .instr(instr)
);
assign im_addr = (pc - 32'h0040_0000)/4;

//CPU
CPU sc_cpu(
    .clk(clk),.rst(rst), .IM_inst(instr), .DM_rdata(r_data),
    .DM_W(dw), .DM_R(dr), .DM_wdata(w_data), .PC_out(pc),.DM_addr(alu_res),.Byte_ena(byteEna)
);
//RAM
DMEM ram(
    .clk(clk),.rst(rst),.ena(1'b1), .DM_W(dw), .DM_R(dr),.byteEna(byteEna) , .DM_addr(dm_addr), .DM_wdata(w_data),
    .DM_rdata(r_data)
);

//Seg
seg7x16 seg(
    .clk(clk_in),.reset(rst),.cs(1'b1),.i_data(pc),.o_seg(o_seg),.o_sel(o_sel)
);

//DIV
CLK_DIV div(
.clk_in(clk_in),.rst(rst),.clk_out(clk)
);

endmodule
```

（12）seg7x16.v：

```
`timescale 1ns / 1ps
module seg7x16(
    input clk,
  input reset,
  input cs,
  input [31:0] i_data,
  output [7:0] o_seg,
  output [7:0] o_sel
```

```
    );

reg [14:0] cnt;
always @ (posedge clk, posedge reset)
if (reset)
cnt <= 0;
else
cnt <= cnt + 1'b1;

wire seg7_clk = cnt[14];

reg [2:0] seg7_addr;

always @ (posedge seg7_clk, posedge reset)
if(reset)
 seg7_addr <= 0;
else
 seg7_addr <= seg7_addr + 1'b1;

reg [7:0] o_sel_r;

always @ (*)
case(seg7_addr)
 7 : o_sel_r = 8'b01111111;
 6 : o_sel_r = 8'b10111111;
 5 : o_sel_r = 8'b11011111;
 4 : o_sel_r = 8'b11101111;
 3 : o_sel_r = 8'b11110111;
 2 : o_sel_r = 8'b11111011;
 1 : o_sel_r = 8'b11111101;
 0 : o_sel_r = 8'b11111110;
endcase

reg [31:0] i_data_store;
always @ (posedge clk, posedge reset)
if(reset)
 i_data_store <= 0;
else if(cs)
 i_data_store <= i_data;

reg [7:0] seg_data_r;
always @ (*)
case(seg7_addr)
 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

reg [7:0] o_seg_r;
always @ (posedge clk, posedge reset)
if(reset)
  o_seg_r <= 8'hff;
else
  case(seg_data_r)
  4'h0 : o_seg_r <= 8'hC0;
  4'h1 : o_seg_r <= 8'hF9;
  4'h2 : o_seg_r <= 8'hA4;
  4'h3 : o_seg_r <= 8'hB0;
  4'h4 : o_seg_r <= 8'h99;
  4'h5 : o_seg_r <= 8'h92;
  4'h6 : o_seg_r <= 8'h82;
  4'h7 : o_seg_r <= 8'hF8;
  4'h8 : o_seg_r <= 8'h80;
  4'h9 : o_seg_r <= 8'h90;
  4'hA : o_seg_r <= 8'h88;
  4'hB : o_seg_r <= 8'h83;
  4'hC : o_seg_r <= 8'hC6;
  4'hD : o_seg_r <= 8'hA1;
  4'hE : o_seg_r <= 8'h86;
  4'hF : o_seg_r <= 8'h8E;
  endcase

assign o_sel = o_sel_r;
assign o_seg = o_seg_r;

endmodule
```

（13）all_tb.v：

```
module all_tb();
  reg clk;
  reg rst;
```

```
// wire [31:0] inst;
 wire [31:0] pc;
 wire [31:0]if_inst,id_inst,ex_inst,me_inst,wb_inst;
// wire[31:0]rf_31,rf_2,rf_1,a,b,r,z;
// wire [2:0]curState;
// reg [31:0]fpc;
wire[7:0]o_seg,o_sel;
wire[4:0]STALL;
// integer counter=0;
// integer file_output;
//  sccomp_dataflow uut(clk,rst,if_inst,id_inst,ex_inst,me_inst,wb_inst,pc,STALL,o_seg,o_sel);
    initial
begin
//        file_output=$fopen("D:\result.txt");
//        fpc<=32'h00400000;
        clk <= 1'b0;
        rst<=1'b1;
        #0.05 rst <= 1'b0;

end

always
 begin
#0.0125 clk <= ~clk;
 end

//    sccomp_dataflow uut(.clk_in(clk),.reset(rst),.inst(inst),.pc(pc),.rf31(rf_31),.rf1(rf_1),.rf2(rf_2));

sccomp_dataflow uut(clk,rst,o_seg,o_sel);
       endmodule
//  always@(posedge clk)
//  begin
//  counter=counter+1;
        // if (counter == 10000000000)
//         begin
//          $fclose(file_output);
//         end
//         if(pc!=fpc)
//         begin
//             counter = counter + 1;
//             $fdisplay(file_output,"pc:%h",pc-32'h00400000);
```

```
45.//              //$fdisplay(file_output,"pc:%h",pc);
46.//              $fdisplay(file_output,"instr:%h",uut.inst);
47.//              $fdisplay(file_output,"regfile0: %h",uut.scc
   pu.cpu_ref.array_reg[0]);
48.//              $fdisplay(file_output,"regfile1: %h",uut.scc
   pu.cpu_ref.array_reg[1]);
49.//              $fdisplay(file_output,"regfile2: %h",uut.scc
   pu.cpu_ref.array_reg[2]);
50.//              $fdisplay(file_output,"regfile3: %h",uut.scc
   pu.cpu_ref.array_reg[3]);
51.//              $fdisplay(file_output,"regfile4: %h",uut.scc
   pu.cpu_ref.array_reg[4]);
52.//              $fdisplay(file_output,"regfile5: %h",uut.scc
   pu.cpu_ref.array_reg[5]);
53.//              $fdisplay(file_output,"regfile6: %h",uut.scc
   pu.cpu_ref.array_reg[6]);
54.//              $fdisplay(file_output,"regfile7: %h",uut.scc
   pu.cpu_ref.array_reg[7]);
55.//              $fdisplay(file_output,"regfile8: %h",uut.scc
   pu.cpu_ref.array_reg[8]);
56.//              $fdisplay(file_output,"regfile9: %h",uut.scc
   pu.cpu_ref.array_reg[9]);
57.//              $fdisplay(file_output,"regfile10: %h",uut.sc
   cpu.cpu_ref.array_reg[10]);
58.//              $fdisplay(file_output,"regfile11: %h",uut.sc
   cpu.cpu_ref.array_reg[11]);
59.//              $fdisplay(file_output,"regfile12: %h",uut.sc
   cpu.cpu_ref.array_reg[12]);
60.//              $fdisplay(file_output,"regfile13: %h",uut.sc
   cpu.cpu_ref.array_reg[13]);
61.//              $fdisplay(file_output,"regfile14: %h",uut.sc
   cpu.cpu_ref.array_reg[14]);
62.//              $fdisplay(file_output,"regfile15: %h",uut.sc
   cpu.cpu_ref.array_reg[15]);
63.//              $fdisplay(file_output,"regfile16: %h",uut.sc
   cpu.cpu_ref.array_reg[16]);
64.//              $fdisplay(file_output,"regfile17: %h",uut.sc
   cpu.cpu_ref.array_reg[17]);
65.//              $fdisplay(file_output,"regfile18: %h",uut.sc
   cpu.cpu_ref.array_reg[18]);
66.//              $fdisplay(file_output,"regfile19: %h",uut.sc
   cpu.cpu_ref.array_reg[19]);
67.//              $fdisplay(file_output,"regfile20: %h",uut.sc
   cpu.cpu_ref.array_reg[20]);
```

```
68.//             $fdisplay(file_output,"regfile21: %h",uut.sc
   cpu.cpu_ref.array_reg[21]);
69.//             $fdisplay(file_output,"regfile22: %h",uut.sc
   cpu.cpu_ref.array_reg[22]);
70.//             $fdisplay(file_output,"regfile23: %h",uut.sc
   cpu.cpu_ref.array_reg[23]);
71.//             $fdisplay(file_output,"regfile24: %h",uut.sc
   cpu.cpu_ref.array_reg[24]);
72.//             $fdisplay(file_output,"regfile25: %h",uut.sc
   cpu.cpu_ref.array_reg[25]);
73.//             $fdisplay(file_output,"regfile26: %h",uut.sc
   cpu.cpu_ref.array_reg[26]);
74.//             $fdisplay(file_output,"regfile27: %h",uut.sc
   cpu.cpu_ref.array_reg[27]);
75.//             $fdisplay(file_output,"regfile28: %h",uut.sc
   cpu.cpu_ref.array_reg[28]);
76.//             $fdisplay(file_output,"regfile29: %h",uut.sc
   cpu.cpu_ref.array_reg[29]);
77.//             $fdisplay(file_output,"regfile30: %h",uut.sc
   cpu.cpu_ref.array_reg[30]);
78.//             $fdisplay(file_output,"regfile31: %h",uut.sc
   cpu.cpu_ref.array_reg[31]);
79.//             fpc<=pc;
80.//          end
81.//     end
82.
83.//     sccomp_dataflow uut(.clk_in(clk),.reset(rst),.inst(i
   nst),.pc(pc),.o_seg(o_seg),.o_sel(o_sel));
```