

计算机系统结构课程实验

总结报告

实验题目：动态流水线设计与性能定量分析

学号：2252431

姓名：孙骁远

指导教师：秦国锋

日期：2024.12.07

一、实验环境部署与硬件配置说明

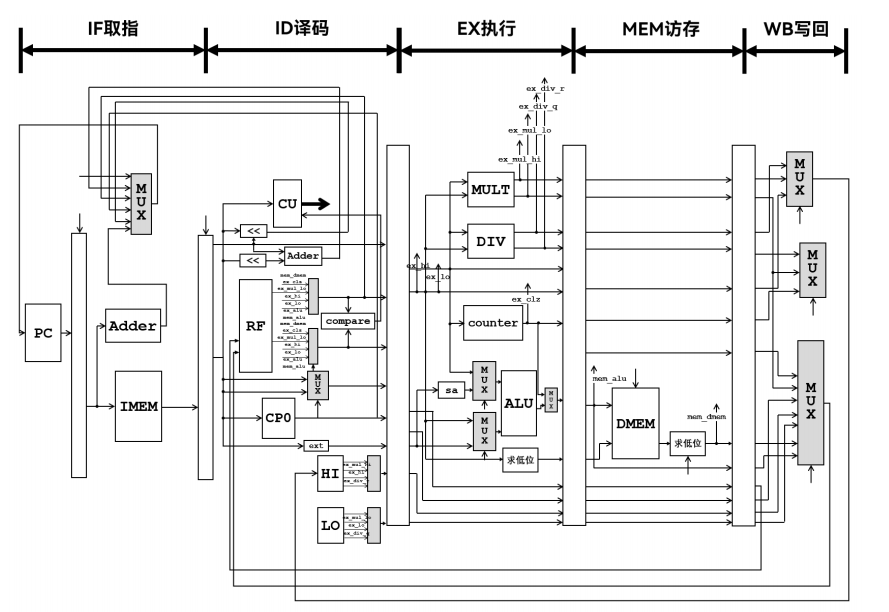
* 操作系统：windows11
* 开发环境：vivado 2016.2
* 仿真工具：vivado自带的仿真模拟器
* 硬件环境：Xilinx NEXYS A7

二、实验的总体结构

实验要求完成至少31条MIPS指令的动态流水线CPU设计，并支持中断。在CPU运行验证程序的过程中，由按键或拨动开关产生一个暂停的中断，再次按键或拨动开关结束中断，继续运行后续的运算，并在数码管上动态显示运算值。

本实验中，我完成了31条MIPS指令的动态流水线设计，可以支持中断。在下板过程中，拨动开关可以模拟中断的产生，再次拨动开关结束中断。7段数码管可以选择在计算过程中显示当前运行的指令码或者pc值。运算结束后，拨动相应开关调整至对应的寄存器，可以在数码管上动态显示运算值。

1.动态流水线的总体结构



2.指令选取说明

本次实验共选取32条指令，总体上可分为运算指令和控制指令。具体格式和指令码如下所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令名称 | 指令格式（0-31位） | | | | | |
| Bit | 31-26 | 25-21 | 20-16 | 15-11 | 10-6 | 5-0 |
| Add | 000000 | rs | rt | rd | 00000 | 100000 |
| Addu | 000000 | rs | rt | rd | 00000 | 100001 |
| Sub | 000000 | rs | rt | rd | 00000 | 100010 |
| Subu | 000000 | rs | rt | rd | 00000 | 100011 |
| And | 000000 | rs | rt | rd | 00000 | 100100 |
| Or | 000000 | rs | rt | rd | 00000 | 100101 |
| Xor | 000000 | rs | rt | rd | 00000 | 100110 |
| Nor | 000000 | rs | rt | rd | 00000 | 100111 |
| Slt | 000000 | rs | rt | rd | 00000 | 101010 |
| Sltu | 000000 | rs | rt | rd | 00000 | 101011 |
| Sll | 000000 | 00000 | rt | shamt | 00000 | 000000 |
| Srl | 000000 | 00000 | rt | shamt | 00000 | 000010 |
| Sra | 000000 | 00000 | rt | shamt | 00000 | 000011 |
| Sllv | 000000 | rs | rt | rd | 00000 | 000100 |
| Srlv | 000000 | rs | rt | rd | 00000 | 000110 |
| Srav | 000000 | rs | rt | rd | 00000 | 000111 |
| Jr | 000000 | rs | 00000 | 00000 | 00000 | 001000 |
| Addi | 001000 | rs | rt | immediate | | |
| Addiu | 001001 | rs | rt | immediate | | |
| Andi | 001100 | rs | rt | immediate | | |
| Ori | 001101 | rs | rt | immediate | | |
| Xori | 001110 | rs | rt | immediate | | |
| Lui | 001111 | rs | rt | immediate | | |
| Lw | 100011 | rs | rt | immediate | | |
| Sw | 101011 | rs | rt | immediate | | |
| Beq | 000100 | rs | rt | immediate | | |
| Bne | 000101 | rs | rt | immediate | | |
| Slti | 001010 | rs | rt | immediate | | |
| Sltiu | 001011 | rs | rt | immediate | | |
| j | 000010 | address | | | | |
| Jal | 000011 | address | | | | |
| Mul | 011100 | rs | rt | rd | 00000 | 000010 |

以上的表格详细地介绍了运算指令和控制指令，以及他们的指令格式，以便于之后的流水线设计

三、总体架构部件的解释说明

1. 动态流水线总体结构部件的解释说明

本次实验设计了一个五级动态流水线，该流水线总体结构分为五个阶

段，分别是 IF（指令获取）、ID（指令译码）、EX（执行）、MEM（访存）、WB（写回）。本次流水线实验中，我实现了对32条指令的设计。5个流水线阶段的说明如下：

1. **取指阶段（IF - Instruction Fetch）：**

 从程序计数器（PC）指向的内存地址中读取指令。

 更新程序计数器（PC）以指向下一条指令的地址。

1. **译码阶段（ID - Instruction Decode）：**

 对取出的指令进行解析，识别操作码（Opcode）和操作数。

 从寄存器文件中读取源操作数。

 处理立即数（Immediate）的符号扩展（Sign Extension）。

 检查是否需要分支或跳转。

1. **执行阶段（EX - 执行）：**

 在算术逻辑单元（ALU）中执行计算或逻辑操作。

 对分支指令进行条件判断并计算目标地址。

1. **访存阶段（MEM - 访存 Access）：**

 对数据存储器进行读取或写入操作（适用于访存指令）。

 如果是跳转指令，根据条件更新 PC。

1. **写回阶段（WB - Write Back）：**

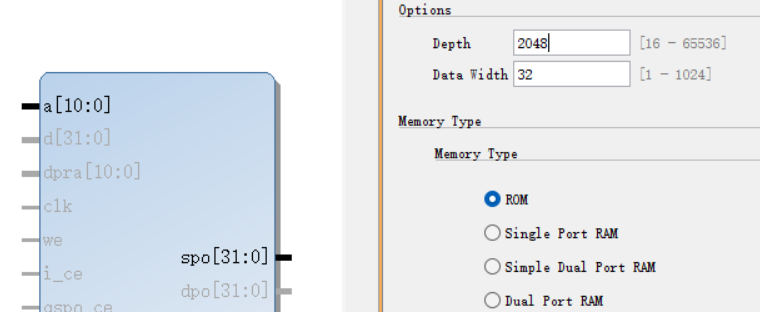
 将计算结果或访存数据写回寄存器文件。

 确保正确更新目标寄存器。

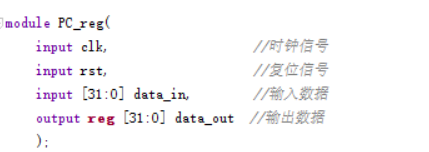
2. 动态流水线各个结构部件说明

3.2.1 指令寄存器imem 用于存储将要执行的指令，配置ip核

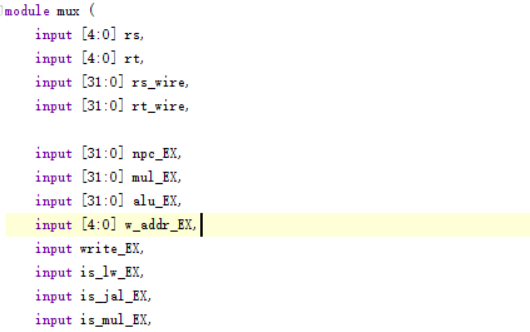




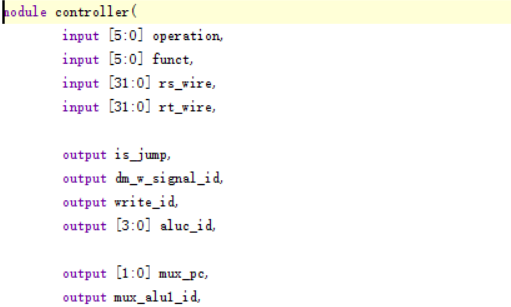
3.2.2 PC寄存器 PC\_reg，下一条PC以及当前的PC指令



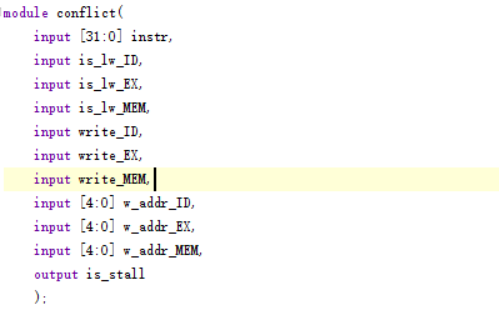
3.2.3 专用路径模块mux，将 EX阶段或者 MEM 阶段的输出结果定向前推至 ID 阶段的多路选择器中



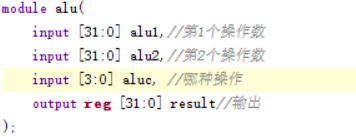
3.2.4 controller，用于指令译码，产生控制信号



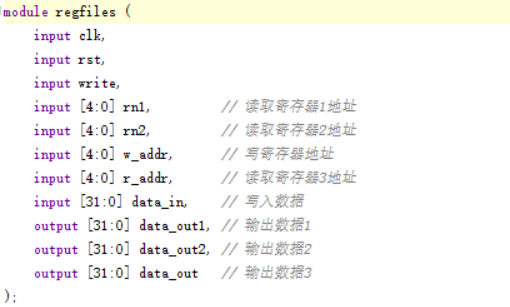
3.2.5 conflict，判断指令是否存在冲突，存在冲突就暂停流水线



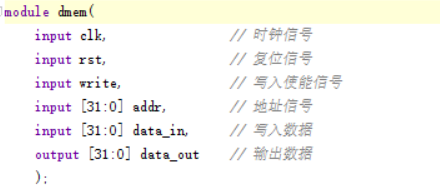
3.2.6 运算器alu



3.2.7 寄存器堆regfiles



3.2.8 dmem，实现数据存储功能



3.2.9 IF/ID , ID/EX , EX/MEM , MEM/WB级间寄存器模块，用于传递各级输出的控制信号和读出的数据，以及级间的计算结果。

四、实验仿真过程

1、动态流水线的仿真过程

各条指令的仿真测试随 CPU 模块的搭建共同进行，最终仿真时，选择第六部分中的数学模型并转为coe文件，用其初始化imem中的ip核。

2、编写testbench程序

module cpu\_top\_tb();

reg clk = 1;

reg rst = 1;

reg [4:0] raddr = 5'b00001; // 初始读取地址

wire [31:0] instr;

wire [31:0] pc;

wire [31:0] odata;

integer count = 0;

integer count\_nop = 0;

always #10 clk = ~clk;

initial begin

#1 rst = 0;

end

// 在时钟上升沿统计

always @(posedge clk) begin

count <= count + 1;

if (instr == 32'b0)

count\_nop <= count\_nop + 1;

end

cpu\_top cpu\_top\_inst (

.clk(clk),

.rst(rst),

.raddr(raddr),

.instr(instr),

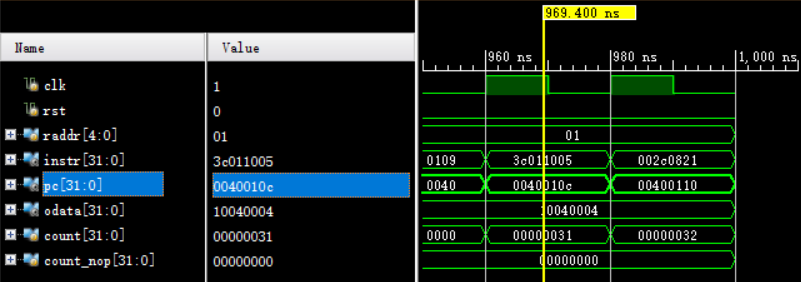
.pc(pc),

.odata(odata)

);

Endmodule

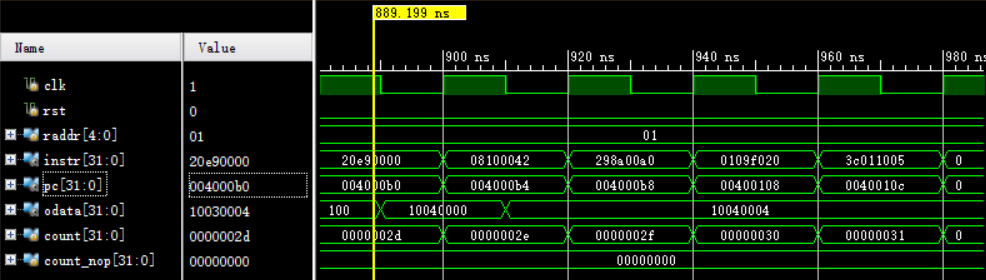
3、用vivado自带的仿真工具进行仿真



五、实验仿真的波形图及某时刻寄存器值的物理意义

1、动态流水线的波形图及某时刻寄存器值的物理意义

波形图如下

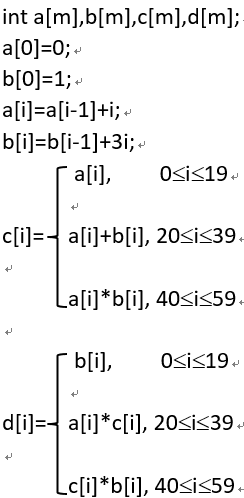


显示当前运算的指令码（instr），PC值（pc），输出值（odata），计数器（count）

每当clk信号到上升沿，下一条指令进入流水线，count + 1

具体运算值将在下板测试中说明

六、实验验算数学模型及算法程序



对应C++程序如下：

#include <iostream>

using namespace std;

int main() {

int a[60];

int b[60];

int c[60];

unsigned int d[60];

// 初始化数组

for (int i = 0; i < 60; i++) {

if (i == 0) {

a[0] = 0;

b[0] = 1;

c[0] = a[0];

d[0] = b[0];

}

else if (i > 0 && i <= 19) {

a[i] = a[i - 1] + i;

b[i] = b[i - 1] + 3 \* i;

c[i] = a[i];

d[i] = b[i];

}

else if (i >= 20 && i <= 39) {

a[i] = a[i - 1] + i;

b[i] = b[i - 1] + 3 \* i;

c[i] = a[i] + b[i];

d[i] = a[i] \* c[i];

}

else {

a[i] = a[i - 1] + i;

b[i] = b[i - 1] + 3 \* i;

c[i] = a[i] \* b[i];

d[i] = b[i] \* c[i];

}

}

cout << a[59] << " ,转为16进制是:" << hex << a[59] << endl;

cout << b[59] << " ,转为16进制是:" << hex << b[59] << endl;

cout << c[59] << " ,转为16进制是:" << hex << c[59] << endl;

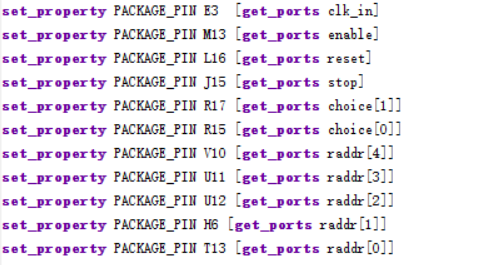
cout << d[59] << " ,转为16进制是:" << hex << d[59] << endl;

return 0;

}

七、实验验算程序下板测试过程与实现

顶层文件：



用J15模拟中断，M13代表使能信号，L16代表复位信号

R17，R15控制在运算时7段数码管的输出,。若R17，R15同时为1，显示指令码；若R17为1，R15为0，显示PC值

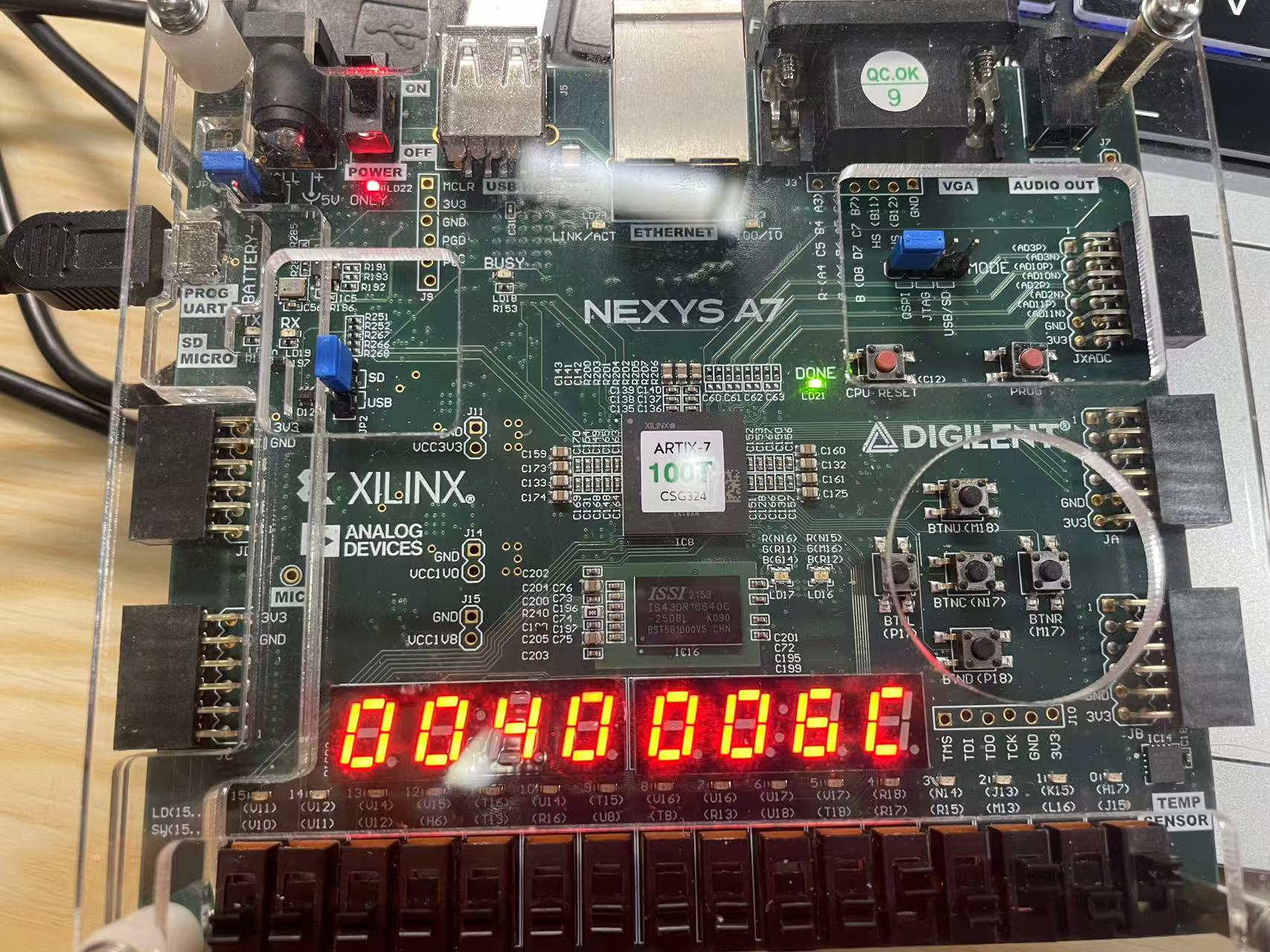
将32个寄存器映射到V10至T13引脚上，设定寄存器6,7,8,9分别存储a[m], b[m], c[m], d[m] 的值

**流水线运算过程中观察现象**

1. **某时刻的指令值**

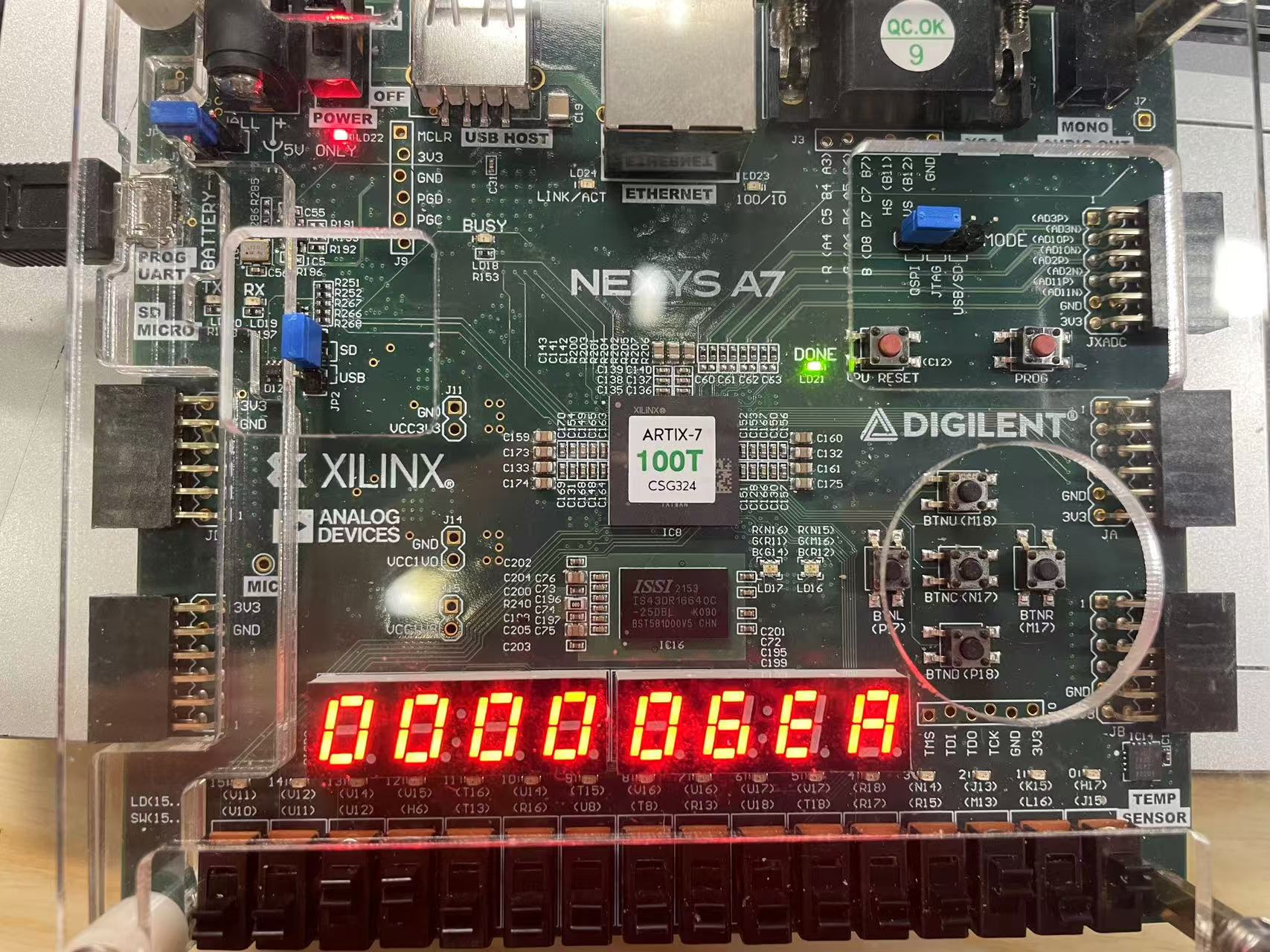


1. **某时刻PC的值**

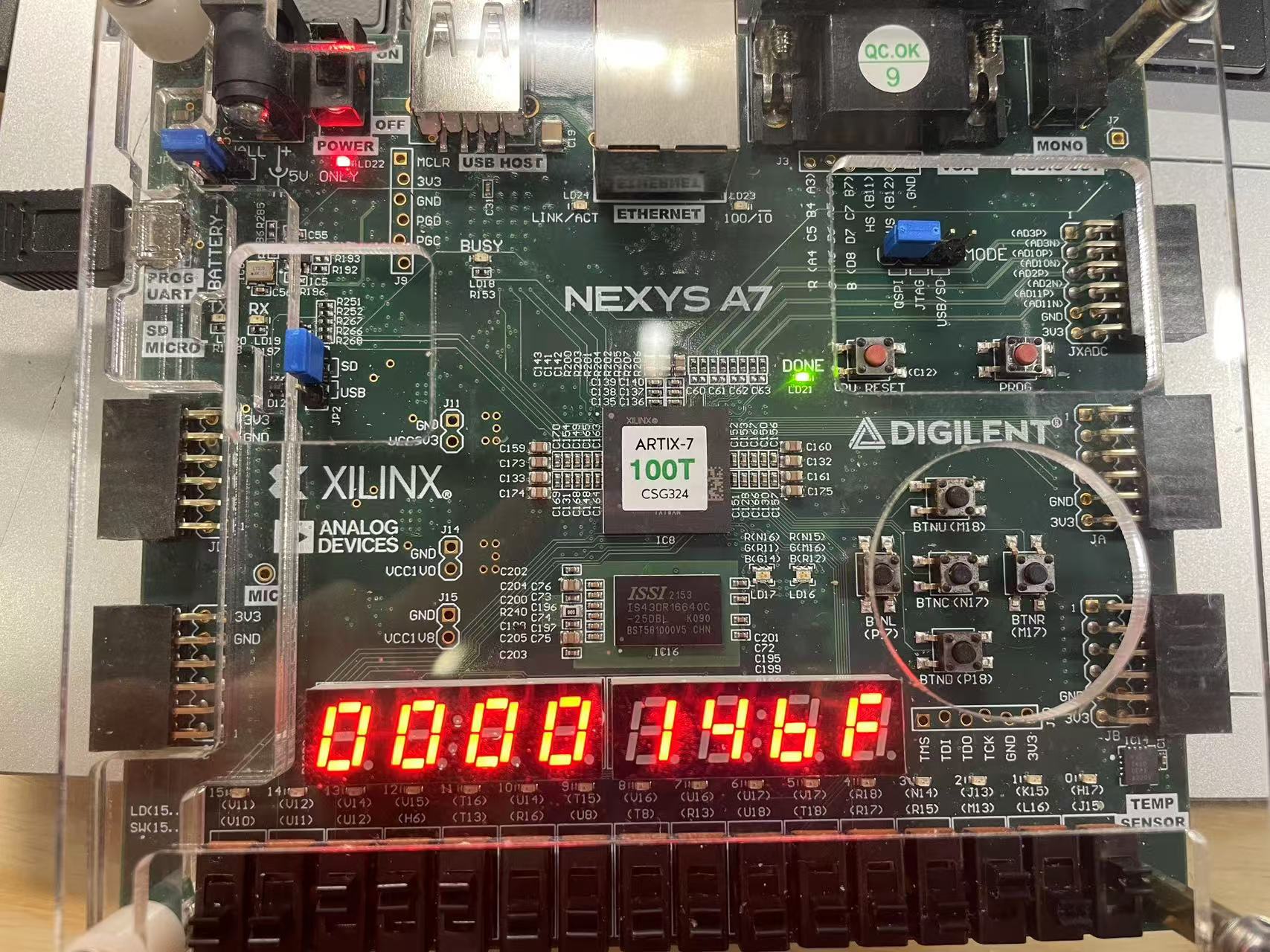


运算结束后，查看4个寄存器的值

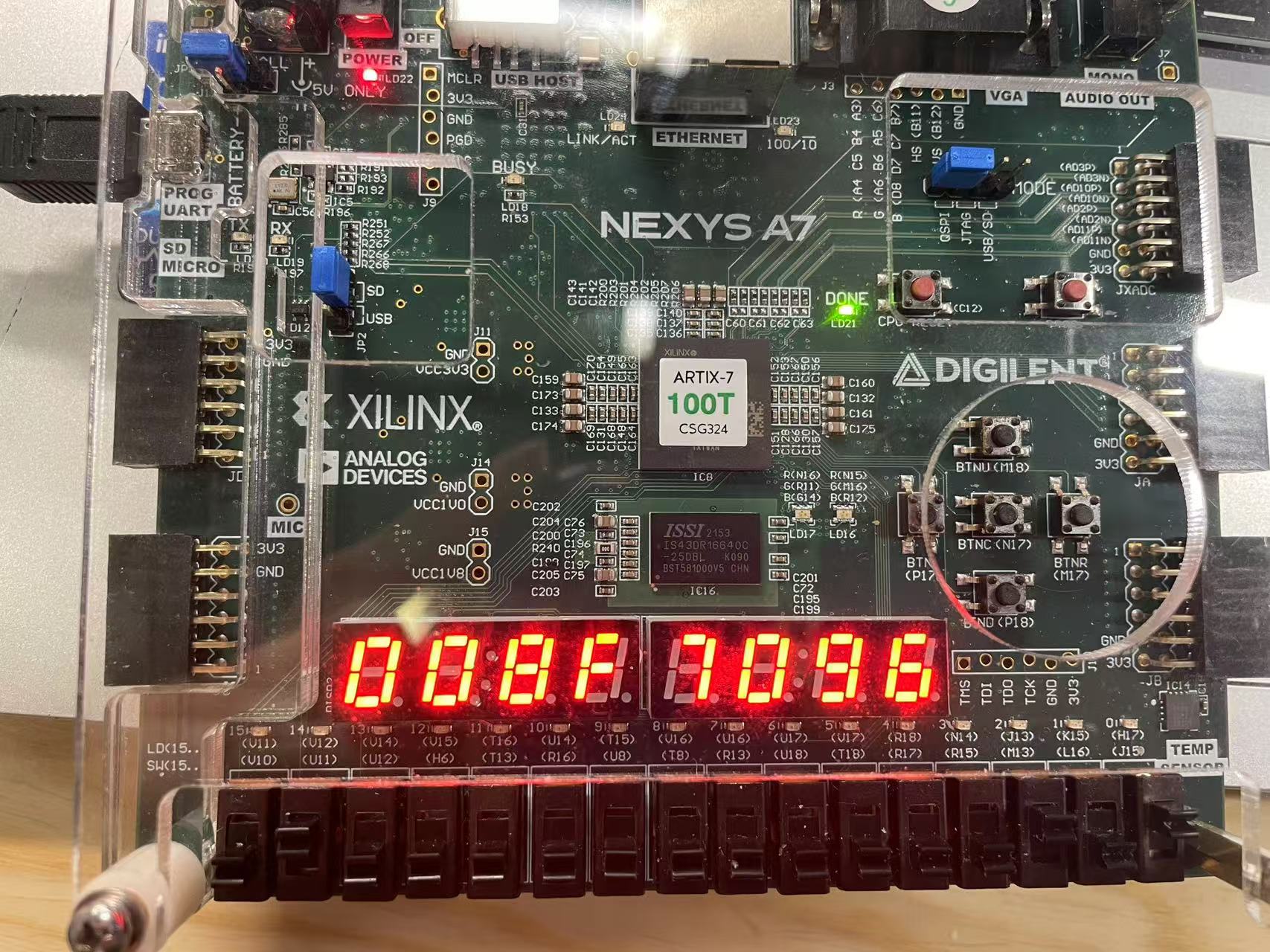
a[59]:



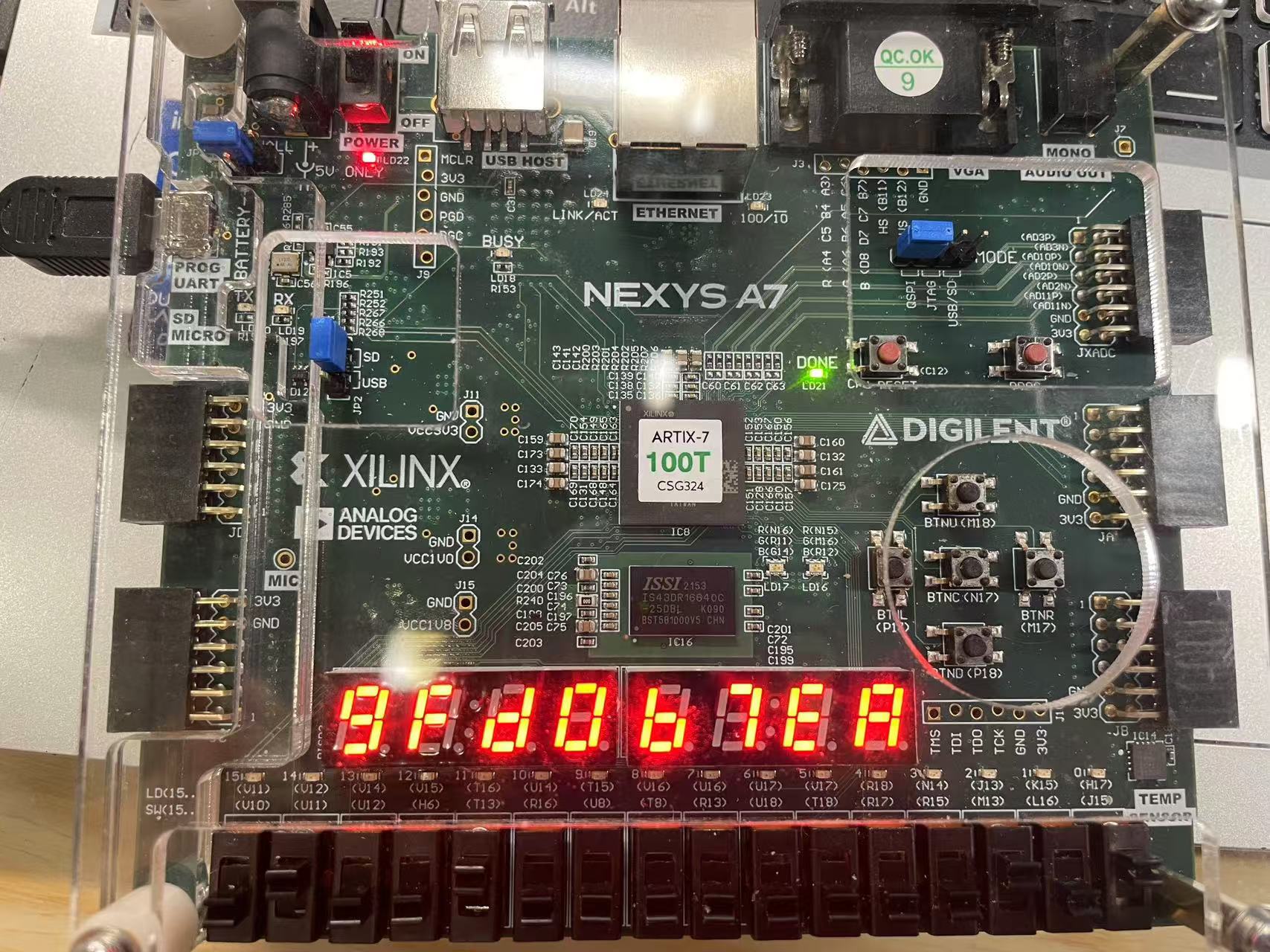
b[59]：



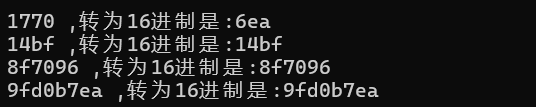
c[59]:



d[59]:

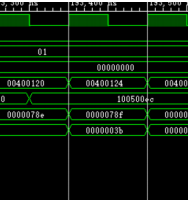


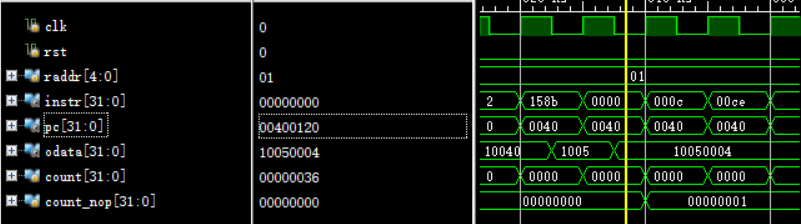
以上结果与Visual Studio运算结果一致



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）

1、动态流水线的性能指标定性分析





**吞吐率：**

根据图像，程序结束时一共运行了78f条指令，其中空指令的数量是3b

因此吞吐率 = （1935-59）/1935 = 0.968

**加速比：**

非流水线所使用的时钟周期数量：(1935-59)\*5

流水线所使用的时钟周期数量：(1935+5-1)

加速比为 = 4.838

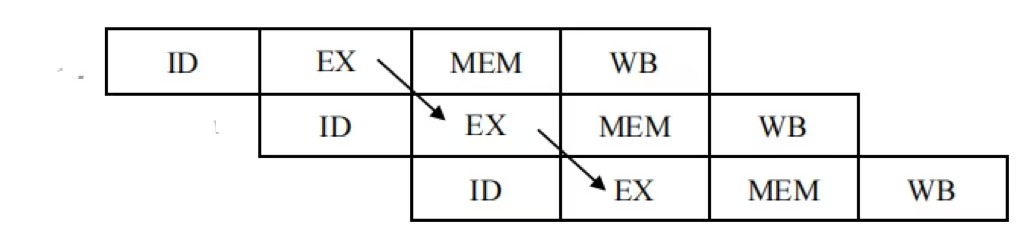
**效率：**

效率 = = = 0.968

**数据相关分析：**

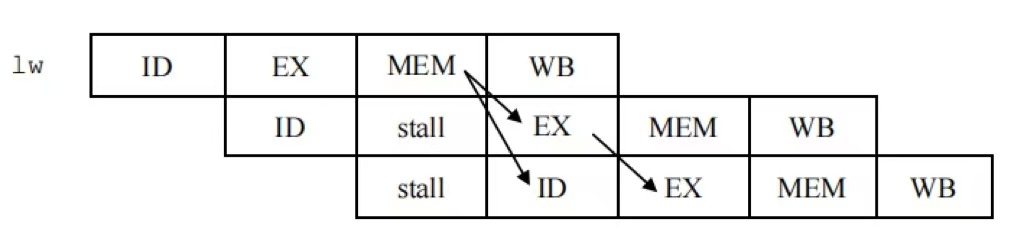
情况1，前一条指令的目的操作数是后一条指令的源操作数

采用内部前推解决，即在ID输出EX所需源操作数时增加多路选择器，将之前周期EX结果或MEM结果存入该多路选择器的输入中，通过控制信号进行数据传递的判断。示意图如下



情况2，多条指令数据相关

如果存在lw指令，不能采用1中的定向技术，只能将其前推至EX级以消除数据冲突，暂停流水线。示意图如下



**控制相关分析：**为了优化分支指令和跳转指令的执行，可以采用延迟槽和提前判断分支技术，将转移条件的判断提前到 ID（指令译码）阶段。这种方法可以使分支指令的延迟仅为一个周期**。**通过这种优化，可以减少流水线阻塞，提高处理器的执行效率。

具体实现方式如下：

* 如果分支成功发生，延迟槽中的指令将被置为无效，并阻止 ID 阶段 的指令产生任何写入操作。同时，将程序计数器（PC）修改为目标地址。
* 如果延迟槽中的指令本身是另一条跳转或分支指令，则必须禁止它对 PC 的进一步修改，以确保分支行为的正确性。

**冲突分析：**

1. 在本次动态流水线CPU的设计中，相对于静态流水线的CPU，添加了内部前推，延迟槽和提前分支判断等机制，解决了部分的先写后读冲突和控制冲突，大大减少了流水线CPU流水停止的概率，提高了流水线CPU的执行效率。
2. 在运算过程中，存在前半周期写、后半周期读不奏效的现象，这会导致读出的寄存器值仍为旧值，因此可以在寄存器堆模块中加入多路选择器mux，将写入的值作为读到的值输出，可以解决该冲突。

九、总结与体会

在本次实验中，我设计并实现了一款基于 MIPS 指令集的五级动态流水线 CPU，并且支持32条指令。相比之前的简易静态流水线设计，此次实验的动态流水线在架构上进行了显著优化，特别是增加了访存（MEM）和写回（WB）阶段的数据前推机制，以有效解决数据相关性问题。数据前推技术使得动态流水线能够在处理过程中减少数据冒险的影响，从而实现更高效的指令执行，显著提高了整体资源的利用率。接下来，我将对本次实验的关键点和收获进行详细总结。

在设计与实现 54 条指令的动态流水线CPU的过程中，我学习整理了教材中的相关内容，这为后续的模块设计和代码编写奠定了坚实的理论基础。在设计阶段，我对CPU进行了详细的模块划分，并逐一实现了每个功能模块。为了验证CPU的正确性并测试其性能，我编写了一套汇编验算程序。该程序涵盖了算术逻辑指令、跳转指令、分支指令以及访存指令等多种类型的指令，能够全面考察CPU的运行性能和处理能力。仿真阶段，我通过 Vivado 和 Modelsim 对这套程序进行测试，确保CPU运行的正确性。最后，我将程序下载到开发板上，通过7段数码管的显示验证了CPU的功能正确。

在验算程序正确运行后，我进行了吞吐率、加速比和效率等方面的性能分析。通过这些指标的分析，我能够对流水线设计的优劣做出客观评价，并验证其在提升指令执行效率方面的有效性。此外，我还学习了数据前推、延迟槽等技术；这些措施解决了相关性冲突，确保流水线的正确性，还能显著提升动态流水线 CPU 的整体性能。

总之，通过验算程序的测试、实验板上的硬件验证以及性能分析，我不仅全面验证了 CPU 的正确性和功能完整性，还对流水线的性能进行了深入评估。这一过程让我加深了对计算机体系结构的理解，同时也显著提升了我在硬件设计和性能优化方面的实际应用能力。

十、附件（所有程序）

1、动态流水线的设计程序

**下板模块 dcpu\_onboard.v**

module dcpu\_onboard(

input clk\_in,//输入时钟

input enable,//使能信号

input reset,//复位信号

input stop,//中断信号

input [1:0] choice,//选择输出指令还是pc

input [4:0] raddr,//寄存器（5位 32个）

//7位数码管

output [7:0] o\_seg,

output [7:0] o\_sel

);

//内部信号定义

reg [24 : 0] cnt;

wire [31 : 0] inst, pc, odata, display;

always @ (posedge clk\_in, posedge reset, posedge stop)

if (reset) cnt <= 0;

else if (stop) cnt <= cnt;

else cnt <= cnt + 1'b1;

wire clk\_cpu = cnt[14];//表示分频

assign display = choice[0] ? inst : (choice[1] ? pc : odata);

cpu\_top cpu\_top\_inst (

.clk(clk\_cpu),

.rst(reset),

.raddr(raddr),

.instr(inst),

.pc(pc),

.odata(odata)

);

seg7x16 seg7x16\_inst (

.clk(clk\_in),

.reset(reset),

.cs(enable),

.i\_data(display),

.o\_seg(o\_seg),

.o\_sel(o\_sel)

);

Endmodule

**顶层模块 cpu\_top.v**

module cpu\_top(

input clk, // 时钟信号

input rst, // 复位信号

input [4:0] raddr, // 寄存器地址

output [31:0] instr,// 指令输出

output [31:0] pc, // 程序计数器

output [31:0] odata // 数据输出

);

wire [31:0] address;

assign address = pc - 32'h00400000;//减去偏移量

cpu cpu\_inst(

.clk(clk), // 时钟信号

.rst(rst), // 复位信号

.instr(instr), // 输入指令

.raddr(raddr), // 寄存器地址

.pc(pc), // 程序计数器

.odata(odata) // 输出数据

);

imem imem\_inst(

.pc(address[12:2]),// 提取高10位地址

.instr(instr) // 输出指令

);

Endmodule

**7段数码管seg7x16.v**

module seg7x16(

input clk, //时钟信号

input reset, //复位信号

input cs, //片选信号

input [31:0] i\_data, //需要数码管输出的内容

output [7:0] o\_seg, //输出内容

output [7:0] o\_sel //片选信号

);

reg [14:0] cnt;//分频信号

always @ (posedge reset, posedge clk) begin

if (reset) cnt <= 0;

else cnt <= cnt + 1'b1;

end

wire seg7\_clk = cnt[14]; // 频率分频

// 3位地址计数器，用于循环选择8位显示

reg [2:0] seg7\_addr;

always @ (posedge seg7\_clk, posedge reset) begin

if(reset) seg7\_addr <= 0;

else seg7\_addr <= seg7\_addr + 1'b1;

end

// 下面使用移位来简化位选信号的生成

reg [7:0] o\_sel\_r;

always @(\*) begin

o\_sel\_r = ~(8'b1 << seg7\_addr);

end

//存储输入数据

reg [31:0] i\_data\_store;

always @ (posedge clk, posedge reset) begin

if(reset == 1'b1)

i\_data\_store <= 0;

else if(cs)

i\_data\_store <= i\_data;

end

// 选择当前的4位段数据

reg [7:0] seg\_data\_r;

always @(\*) begin

seg\_data\_r = i\_data\_store[(seg7\_addr \* 4) +: 4];

end

function [7:0] seg\_decoder(input [3:0] digit);

case (digit)

4'h0: seg\_decoder = 8'hC0;

4'h1: seg\_decoder = 8'hF9;

4'h2: seg\_decoder = 8'hA4;

4'h3: seg\_decoder = 8'hB0;

4'h4: seg\_decoder = 8'h99;

4'h5: seg\_decoder = 8'h92;

4'h6: seg\_decoder = 8'h82;

4'h7: seg\_decoder = 8'hF8;

4'h8: seg\_decoder = 8'h80;

4'h9: seg\_decoder = 8'h90;

4'hA: seg\_decoder = 8'h88;

4'hB: seg\_decoder = 8'h83;

4'hC: seg\_decoder = 8'hC6;

4'hD: seg\_decoder = 8'hA1;

4'hE: seg\_decoder = 8'h86;

4'hF: seg\_decoder = 8'h8E;

default: seg\_decoder = 8'hFF;

endcase

endfunction

// 赋值给 o\_seg\_r

reg [7:0] o\_seg\_r;

always @(posedge clk or posedge reset) begin

if (reset)

o\_seg\_r <= 8'hFF;

else

o\_seg\_r <= seg\_decoder(seg\_data\_r);

end

assign o\_sel = o\_sel\_r;

assign o\_seg = o\_seg\_r;

endmodule

**CPU总模块 cpu.v:**

module cpu(

input clk, //时钟信号

input rst, //复位信号

input [31:0] instr,//指令输出

input [4:0] raddr, //寄存器地址

output [31:0] pc, //程序计数器

output [31:0] odata//数据输出

);

//#########################################################################

//# IF 部分

//#########################################################################

wire is\_jump, is\_stall;

wire [31:0] npc\_if, pc\_id, pc\_if, instr\_if;

assign npc\_if = pc + 32'd4;

assign instr\_if = (is\_stall) ? 32'b0 : instr;

assign pc\_if = (is\_stall) ? pc : (is\_jump) ? pc\_id : npc\_if;

PC\_reg PC\_inst(

.clk(clk),

.rst(rst),

.data\_in(pc\_if),

.data\_out(pc)

);

//#########################################################################

//# IF\_ID 流水线

//#########################################################################

wire [31:0] instr\_id, npc\_id;

regs\_IF\_ID regs\_IF\_ID\_inst(

.clk(clk),

.rst(rst),

.npc\_if(npc\_if),

.instr\_if(instr\_if),

.npc\_id(npc\_id),

.instr\_id(instr\_id)

);

//#########################################################################

//# ID 部分

//#########################################################################

//inst--ID指令解码

wire [5:0] op,func;

wire [4:0] rs,rt,rd,shamt;

wire [15:0] immediate;

wire [25:0] index;

wire [31:0] shamt\_id, uimmediate\_id, simmediate\_id, offset;

assign func = instr\_id[5:0];

assign shamt = instr\_id[10:6];

assign immediate = instr\_id[15:0];

assign index = instr\_id[25:0];

assign op = instr\_id[31:26];

assign rs = instr\_id[25:21];

assign rt = instr\_id[20:16];

assign rd = instr\_id[15:11];

assign shamt\_id = {27'b0, shamt};

assign uimmediate\_id = {16'b0, immediate};

assign simmediate\_id = {{16{immediate[15]}}, immediate};

assign offset = {{14{immediate[15]}}, immediate, 2'b0};

//ID选择

wire [1:0] mux\_pc;

wire [31:0] rs\_wire, rt\_wire;

wire [31:0] pc\_offset, pc\_index, pc\_rs;

assign pc\_rs = rs\_wire;

assign pc\_offset = pc + offset;

assign pc\_index = {npc\_if[31:28], index, 2'b0};

assign pc\_id = (mux\_pc[1]) ? pc\_offset : (mux\_pc[0]) ? pc\_rs : pc\_index;

//#########################################################################

//# 数据选择

//#########################################################################

wire write\_ex, write\_mem, conflict\_lw;

wire is\_lw\_ex, is\_jal\_ex, is\_mul\_ex;

wire is\_lw\_mem, is\_jal\_mem, is\_mul\_mem;

wire [4:0] w\_addr\_ex, w\_addr\_mem;

wire [31:0] alu\_ex, mul\_ex, npc\_ex;

wire [31:0] alu\_mem, mul\_mem, npc\_mem;

wire [31:0] rs\_mux, rt\_mux;

mux mux\_INST(rs, rt, rs\_wire, rt\_wire, npc\_ex, mul\_ex, alu\_ex, w\_addr\_ex, write\_ex, is\_lw\_ex, is\_jal\_ex, is\_mul\_ex,

npc\_mem, mul\_mem, alu\_mem, w\_addr\_mem, write\_mem, is\_lw\_mem, is\_jal\_mem, is\_mul\_mem, rs\_mux, rt\_mux, conflict\_lw);

//#########################################################################

//# 控制器与检测冲突

//#########################################################################

wire dm\_w\_signal\_id, write\_id;

wire is\_lw\_id, is\_jal\_id, is\_mul\_id;

wire [3:0] aluc\_id;

reg [4:0] w\_addr\_id;

wire mux\_alu1\_id;

wire [1:0] mux\_alu2\_id, mux\_w\_addr\_id, mux\_wdata\_id;

always @(\*) begin

if (mux\_w\_addr\_id[1])

w\_addr\_id = 5'd31;

else if (mux\_w\_addr\_id[0])

w\_addr\_id = rd;

else

w\_addr\_id = rt;

end

controller controller\_INST(op, func, rs\_mux, rt\_mux, is\_jump, dm\_w\_signal\_id,

write\_id, aluc\_id, mux\_pc, mux\_alu1\_id, mux\_alu2\_id,

mux\_w\_addr\_id, is\_lw\_id, is\_jal\_id, is\_mul\_id);

//判断有无冲突

conflict\_judge conflict\_judge\_INST(instr, is\_lw\_id, is\_lw\_ex, is\_lw\_mem, write\_id,

write\_ex, write\_mem, w\_addr\_id, w\_addr\_ex,

w\_addr\_mem, is\_stall);

//#########################################################################

//# ID\_EXE流水线寄存器

//#########################################################################

wire dm\_w\_signal\_ex;

wire mux\_alu1\_ex;

wire dm\_w\_id\_real;

wire [1:0] mux\_alu2\_ex, mux\_wdata\_ex;

wire [3:0] aluc\_ex;

wire [31:0] shamt\_ex, simmediate\_ex, uimmediate\_ex;

wire [31:0] rs\_wire\_ex, rt\_wire\_ex, dm\_wdata\_ex;

wire judge;

assign dm\_w\_id\_real = judge;

assign judge = dm\_w\_signal\_id && (~conflict\_lw);

regs\_ID\_EX regs\_ID\_EX\_INST(clk,rst, dm\_w\_id\_real, write\_id, is\_lw\_id, is\_jal\_id, is\_mul\_id, mux\_alu1\_id,

mux\_alu2\_id, aluc\_id, npc\_id, w\_addr\_id, shamt\_id, simmediate\_id, uimmediate\_id,

rs\_mux, rt\_mux, rt\_mux, dm\_w\_signal\_ex, write\_ex, is\_lw\_ex, is\_jal\_ex,

is\_mul\_ex, mux\_alu1\_ex, mux\_alu2\_ex, aluc\_ex, npc\_ex, w\_addr\_ex,

shamt\_ex, simmediate\_ex, uimmediate\_ex, rs\_wire\_ex, rt\_wire\_ex, dm\_wdata\_ex);

//#########################################################################

//# EXE阶段

//#########################################################################

reg [31:0] alu1, alu2;

wire [63:0] mul\_out;

always @(\*) begin

if (mux\_alu1\_ex) alu1 = shamt\_ex;

else alu1 = rs\_wire\_ex;

if (mux\_alu2\_ex[1])

alu2 = rt\_wire\_ex;

else if (mux\_alu2\_ex[0])

alu2 = uimmediate\_ex;

else

alu2 = simmediate\_ex;

end

//初始化alu

alu alu\_inst(

.alu1(alu1),

.alu2(alu2),

.aluc(aluc\_ex),

.result(alu\_ex)

);

assign mul\_out = alu1 \* alu2;

assign mul\_ex = mul\_out[31:0];

//#########################################################################

//# EXE\_MEM 流水线寄存器

//#########################################################################

wire DM\_W\_SIGNAL\_MEM;

wire [1:0] MUX\_WDATA\_MEM;

wire [31:0] DM\_WDATA\_MEM;

regs\_EX\_MEM regs\_EX\_MEM\_INST(clk, rst, dm\_w\_signal\_ex, write\_ex,

is\_lw\_ex, is\_jal\_ex, is\_mul\_ex, w\_addr\_ex,npc\_ex, alu\_ex, mul\_ex,

dm\_wdata\_ex, DM\_W\_SIGNAL\_MEM, write\_mem, is\_lw\_mem,

is\_jal\_mem, is\_mul\_mem, w\_addr\_mem, npc\_mem, alu\_mem,

mul\_mem, DM\_WDATA\_MEM

);

//#########################################################################

//# MEM部分

//#########################################################################

wire [31:0] dm\_addr, dm\_rdata, dm\_wdata;

wire dm\_w;

assign dm\_addr = alu\_mem;

assign dm\_wdata = DM\_WDATA\_MEM;

assign dm\_w = DM\_W\_SIGNAL\_MEM;

dmem dmem\_INST(

.clk(~clk),

.rst(rst),

.write(dm\_w),

.addr(dm\_addr),

.data\_in(dm\_wdata),

.data\_out(dm\_rdata)

);

//#########################################################################

//# MEM\_WB 流水线寄存器

//#########################################################################

wire [1:0] mux\_wdata\_wb;

wire [4:0] w\_addr\_wb;

wire write\_wb, is\_lw\_wb, is\_jal\_wb, is\_mul\_wb;

wire [31:0] alu\_wb, mul\_wb, npc\_wb, dm\_rdata\_wb;

wire [31:0] wb\_signals[3:0]; // 数组化存储信号

// 映射信号

assign alu\_wb = wb\_signals[0];

assign mul\_wb = wb\_signals[1];

assign npc\_wb = wb\_signals[2];

assign dm\_rdata\_wb = wb\_signals[3];

// regs\_MEM\_WB 实例化

regs\_MEM\_WB regs\_MEM\_WB\_INST(clk, rst, write\_mem, is\_lw\_mem,

is\_jal\_mem, is\_mul\_mem, w\_addr\_mem, alu\_mem,mul\_mem,

npc\_mem, dm\_rdata, write\_wb, is\_lw\_wb, is\_jal\_wb, is\_mul\_wb,

w\_addr\_wb, wb\_signals[0], wb\_signals[1], wb\_signals[2], wb\_signals[3]

);

//#########################################################################

//# WB 阶段

//#########################################################################

reg [31:0] wdata\_wb;

always @(\*) begin

if (is\_lw\_wb)

wdata\_wb = dm\_rdata\_wb;

else if (is\_jal\_wb)

wdata\_wb = npc\_wb;

else if (is\_mul\_wb)

wdata\_wb = mul\_wb;

else

wdata\_wb = alu\_wb;

end

regfiles regfiles\_INST(

.clk(clk),

.rst(rst),

.write(write\_wb),

.rn1(rs),

.rn2(rt),

.w\_addr(w\_addr\_wb),

.r\_addr(raddr),

.data\_in(wdata\_wb),

.data\_out1(rs\_wire),

.data\_out2(rt\_wire),

.data\_out(odata)

);

Endmodule

**PC寄存器 PC\_reg：**

module PC\_reg(

input clk, //时钟信号

input rst, //复位信号

input [31:0] data\_in, //输入数据

output reg [31:0] data\_out //输出数据

);

localparam RESET\_VALUE = 32'h00400000;

always@(posedge rst or posedge clk) begin

if(rst==1)

data\_out <= RESET\_VALUE;

else

data\_out <= data\_in;

end

endmodule

IF\_ID阶段流水线

module regs\_IF\_ID(

input clk, //时钟信号

input rst, //复位信号

input [31:0] npc\_if, //程序计数器值

input [31:0] instr\_if, //指令获取阶段

output reg [31:0] npc\_id = 32'b0,//解码阶段的程序计数器值

output reg [31:0] instr\_id = 32'b0 //解码阶段的指令

);

localparam RESET\_VALUE = 32'b0;

always @(posedge clk or posedge rst) begin

if(rst==1) begin

//复位指令

{npc\_id, instr\_id} <= {RESET\_VALUE, RESET\_VALUE};

end

else begin

//更新指令

{npc\_id, instr\_id} <= {npc\_if, instr\_if};

end

end

endmodule

**多路选择器 mux.v:**

module mux (

input [4:0] rs,

input [4:0] rt,

input [31:0] rs\_wire,

input [31:0] rt\_wire,

input [31:0] npc\_EX,

input [31:0] mul\_EX,

input [31:0] alu\_EX,

input [4:0] w\_addr\_EX,

input write\_EX,

input is\_lw\_EX,

input is\_jal\_EX,

input is\_mul\_EX,

input [31:0] npc\_MEM,

input [31:0] mul\_MEM,

input [31:0] alu\_MEM,

input [4:0] w\_addr\_MEM,

input write\_MEM,

input is\_lw\_MEM,

input is\_jal\_MEM,

input is\_mul\_MEM,

output reg [31:0] rs\_mux = 32'b0,

output reg [31:0] rt\_mux = 32'b0,

output reg conflict\_lw = 1'b0

);

reg conflict\_lw\_rs = 1'b0;

reg conflict\_lw\_rt = 1'b0;

// 公共任务：处理寄存器的 MUX 逻辑

task handle\_mux(

input [4:0] reg\_addr,

input [31:0] wire\_data,

input [31:0] npc\_stage,

input [31:0] mul\_stage,

input [31:0] alu\_stage,

input [4:0] w\_addr\_stage,

input write\_stage,

input is\_lw\_stage,

input is\_jal\_stage,

input is\_mul\_stage,

output reg [31:0] mux\_out,

output reg conflict\_lw\_out

);

begin

if (write\_stage && w\_addr\_stage == reg\_addr && reg\_addr != 5'b0) begin

if (is\_jal\_stage) begin

conflict\_lw\_out = 1'b0;

mux\_out = npc\_stage;

end else if (is\_mul\_stage) begin

conflict\_lw\_out = 1'b0;

mux\_out = mul\_stage;

end else if (is\_lw\_stage) begin

conflict\_lw\_out = 1'b1;

mux\_out = 32'b0;

end else begin

conflict\_lw\_out = 1'b0;

mux\_out = alu\_stage;

end

end else begin

conflict\_lw\_out = 1'b0;

mux\_out = wire\_data;

end

end

endtask

always @(\*) begin

conflict\_lw = conflict\_lw\_rs || conflict\_lw\_rt;

end

// 处理 rs 寄存器的 MUX 逻辑

always @(\*) begin

handle\_mux(

rs, rs\_wire,

npc\_EX, mul\_EX, alu\_EX, w\_addr\_EX, write\_EX, is\_lw\_EX, is\_jal\_EX, is\_mul\_EX,

rs\_mux, conflict\_lw\_rs

);

if (!conflict\_lw\_rs) begin

handle\_mux(

rs, rs\_mux,

npc\_MEM, mul\_MEM, alu\_MEM, w\_addr\_MEM, write\_MEM, is\_lw\_MEM, is\_jal\_MEM, is\_mul\_MEM,

rs\_mux, conflict\_lw\_rs

);

end

end

// 处理 rt 寄存器的 MUX 逻辑

always @(\*) begin

handle\_mux(

rt, rt\_wire,

npc\_EX, mul\_EX, alu\_EX, w\_addr\_EX, write\_EX, is\_lw\_EX, is\_jal\_EX, is\_mul\_EX,

rt\_mux, conflict\_lw\_rt

);

if (!conflict\_lw\_rt) begin

handle\_mux(

rt, rt\_mux,

npc\_MEM, mul\_MEM, alu\_MEM, w\_addr\_MEM, write\_MEM, is\_lw\_MEM, is\_jal\_MEM, is\_mul\_MEM,

rt\_mux, conflict\_lw\_rt

);

end

end

endmodule

**控制器controller：**

module controller(

input [5:0] operation,

input [5:0] funct,

input [31:0] rs\_wire,

input [31:0] rt\_wire,

output is\_jump,

output dm\_w\_signal\_id,

output write\_id,

output [3:0] aluc\_id,

output [1:0] mux\_pc,

output mux\_alu1\_id,

output [1:0] mux\_alu2\_id,

output [1:0] mux\_waddr\_id,

output is\_lw\_id,

output is\_jal\_id,

output is\_mul\_id

);

wire r\_type = (operation == 6'b000000);

wire ADD = r\_type & (funct == 6'b100000);

wire ADDU = r\_type & (funct == 6'b100001);

wire SUB = r\_type & (funct == 6'b100010);

wire SUBU = r\_type & (funct == 6'b100011);

wire AND = r\_type & (funct == 6'b100100);

wire OR = r\_type & (funct == 6'b100101);

wire XOR = r\_type & (funct == 6'b100110);

wire NOR = r\_type & (funct == 6'b100111);

wire SLT = r\_type & (funct == 6'b101010);

wire SLTU = r\_type & (funct == 6'b101011);

wire SLL = r\_type & (funct == 6'b000000);

wire SRL = r\_type & (funct == 6'b000010);

wire SRA = r\_type & (funct == 6'b000011);

wire SLLV = r\_type & (funct == 6'b000100);

wire SRLV = r\_type & (funct == 6'b000110);

wire SRAV = r\_type & (funct == 6'b000111);

wire JR = r\_type & (funct == 6'b001000);

wire MUL = (operation == 6'b011100) & (funct == 6'b000010);

wire ADDI = (operation == 6'b001000);

wire ADDIU = (operation == 6'b001001);

wire ANDI = (operation == 6'b001100);

wire ORI = (operation == 6'b001101);

wire XORI = (operation == 6'b001110);

wire LUI = (operation == 6'b001111);

wire LW = (operation == 6'b100011);

wire SW = (operation == 6'b101011);

wire BEQ = (operation == 6'b000100);

wire BNE = (operation == 6'b000101);

wire SLTI = (operation == 6'b001010);

wire SLTIU = (operation == 6'b001011);

wire J = (operation == 6'b000010);

wire JAL = (operation == 6'b000011);

// 对输出赋值

assign is\_jal\_id = JAL;

assign is\_lw\_id = LW;

assign is\_mul\_id = MUL;

assign dm\_w\_signal\_id = SW;

assign write\_id = ~(JR | SW | BEQ | BNE | J);

assign is\_jump = JR | J | JAL | (BEQ & (rs\_wire == rt\_wire)) | (BNE & (rs\_wire != rt\_wire));

assign aluc\_id = {

LUI | SLL | SLLV | SLT | SLTI | SLTIU | SLTU | SRA | SRAV | SRL | SRLV,

AND | ANDI | NOR | OR | ORI | SLL | SLLV | SRA | SRAV | SRL | SRLV | XOR | XORI,

ADD | ADDI | BEQ | BNE | LW | NOR | SLL | SLLV | SLT | SLTI | SLTIU | SLTU | SUB | SW | XOR | XORI,

BEQ | BNE | NOR | OR | ORI | SLT | SLTI | SRL | SRLV | SUB | SUBU

};

assign mux\_pc = (J | JAL) ? 2'b00 :

(JR) ? 2'b01 :

(BNE | BEQ) ? 2'b11 : 2'bxx;

assign mux\_alu1\_id = SLL | SRA | SRL;

assign mux\_alu2\_id = {~(ADDI | ADDIU | LUI | LW | SLTI | SW | ANDI | ORI | SLTIU | XORI),

ANDI | ORI | SLTIU | XORI};

assign mux\_waddr\_id = {JAL, ~(ADDI | ADDIU | ANDI | LUI | LW | ORI | SLTI | SLTIU | XORI | JAL)};

endmodule

**判断冲突 conflict\_judge.v：**

module conflict\_judge(

input [31:0] instr,

input is\_lw\_ID,

input is\_lw\_EX,

input is\_lw\_MEM,

input write\_ID,

input write\_EX,

input write\_MEM,

input [4:0] w\_addr\_ID,

input [4:0] w\_addr\_EX,

input [4:0] w\_addr\_MEM,

output is\_stall

);

wire [4:0] rs, rt;

assign rt = instr[20:16];

assign rs = instr[25:21];

// 3种情况

wire condition1 = (is\_lw\_ID && write\_ID && w\_addr\_ID != 5'b0) && ((rs == w\_addr\_ID) || (rt == w\_addr\_ID));

wire condition2 = (is\_lw\_MEM && write\_MEM && w\_addr\_MEM != 5'b0) && ((rs == w\_addr\_MEM) || (rt == w\_addr\_MEM));

wire condition3 = (is\_lw\_EX && write\_EX && w\_addr\_EX != 5'b0) && ((rs == w\_addr\_EX) || (rt == w\_addr\_EX));

assign is\_stall = condition1 || condition2 || condition3;

endmodule

**ID\_EX之间流水线阶段：**module regs\_ID\_EX(

input clk,

input rst,

input dm\_w\_signal\_ID,

input write\_ID, is\_lw\_ID, is\_jal\_ID, is\_mul\_ID, mux\_alu1\_ID,

input [1:0] mux\_alu2\_ID,

input [3:0] aluc\_ID,

input [31:0] npc\_ID,

input [4:0] w\_addr\_ID,

input [31:0] shamt\_ID, simmediate\_ID, uimmediate\_ID, rs\_wire\_ID, rt\_wire\_ID, dm\_wdata\_ID,

output reg dm\_w\_signal\_EX = 1'b0,

output reg write\_EX = 1'b0,

output reg is\_lw\_EX = 1'b0,

output reg is\_jal\_EX = 1'b0,

output reg is\_mul\_EX = 1'b0,

output reg mux\_alu1\_EX = 1'b0,

output reg [1:0] mux\_alu2\_EX = 2'b0,

output reg [3:0] aluc\_EX = 4'b0,

output reg [31:0] npc\_EX = 32'b0,

output reg [4:0] w\_addr\_EX = 5'b0,

output reg [31:0] shamt\_EX = 32'b0,

output reg [31:0] simmediate\_EX = 32'b0,

output reg [31:0] uimmediate\_EX = 32'b0,

output reg [31:0] rs\_wire\_EX = 32'b0,

output reg [31:0] rt\_wire\_EX = 32'b0,

output reg [31:0] dm\_wdata\_EX = 32'b0

);

// 通用任务用于赋值或复位

task assign\_or\_reset;

input reset;

begin

dm\_w\_signal\_EX <= reset ? 1'b0 : dm\_w\_signal\_ID;

write\_EX <= reset ? 1'b0 : write\_ID;

is\_lw\_EX <= reset ? 1'b0 : is\_lw\_ID;

is\_jal\_EX <= reset ? 1'b0 : is\_jal\_ID;

is\_mul\_EX <= reset ? 1'b0 : is\_mul\_ID;

mux\_alu1\_EX <= reset ? 1'b0 : mux\_alu1\_ID;

mux\_alu2\_EX <= reset ? 2'b0 : mux\_alu2\_ID;

aluc\_EX <= reset ? 4'b0 : aluc\_ID;

npc\_EX <= reset ? 32'b0 : npc\_ID;

w\_addr\_EX <= reset ? 5'b0 : w\_addr\_ID;

shamt\_EX <= reset ? 32'b0 : shamt\_ID;

simmediate\_EX <= reset ? 32'b0 : simmediate\_ID;

uimmediate\_EX <= reset ? 32'b0 : uimmediate\_ID;

rs\_wire\_EX <= reset ? 32'b0 : rs\_wire\_ID;

rt\_wire\_EX <= reset ? 32'b0 : rt\_wire\_ID;

dm\_wdata\_EX <= reset ? 32'b0 : dm\_wdata\_ID;

end

endtask

// 主逻辑

always @(posedge rst or posedge clk) begin

if (rst)

assign\_or\_reset(1'b1); // 复位所有信号

else

assign\_or\_reset(1'b0); // 更新信号

end

endmodule

**EX\_MEM阶段流水线：**

module regs\_EX\_MEM(

input clk,

input rst,

input dm\_w\_signal\_ex,

input write\_ex, is\_lw\_ex, is\_jal\_ex, is\_mul\_ex,

input [4:0] w\_addr\_ex,

input [31:0] alu\_ex, mul\_ex, npc\_ex, dm\_wdata\_ex,

output reg dm\_w\_signal\_mem = 1'b0,

output reg write\_mem = 1'b0,

output reg is\_lw\_mem = 1'b0,

output reg is\_jal\_mem = 1'b0,

output reg is\_mul\_mem = 1'b0,

output reg [4:0] w\_addr\_mem = 5'b0,

output reg [31:0] alu\_mem = 32'b0,

output reg [31:0] mul\_mem = 32'b0,

output reg [31:0] npc\_mem = 32'b0,

output reg [31:0] dm\_wdata\_mem = 32'b0

);

// 通用任务用于赋值或复位

task assign\_or\_reset;

input reset;

begin

dm\_w\_signal\_mem <= reset ? 1'b0 : dm\_w\_signal\_ex;

write\_mem <= reset ? 1'b0 : write\_ex;

is\_lw\_mem <= reset ? 1'b0 : is\_lw\_ex;

is\_jal\_mem <= reset ? 1'b0 : is\_jal\_ex;

is\_mul\_mem <= reset ? 1'b0 : is\_mul\_ex;

w\_addr\_mem <= reset ? 5'b0 : w\_addr\_ex;

alu\_mem <= reset ? 32'b0 : alu\_ex;

mul\_mem <= reset ? 32'b0 : mul\_ex;

npc\_mem <= reset ? 32'b0 : npc\_ex;

dm\_wdata\_mem <= reset ? 32'b0 : dm\_wdata\_ex;

end

endtask

// 主逻辑

always @(posedge rst or posedge clk) begin

if (rst)

assign\_or\_reset(1'b1); // 复位所有信号

else

assign\_or\_reset(1'b0); // 更新信号

end

endmodule

**MEM\_WB之间流水线部分：**

module regs\_MEM\_WB(

input clk,

input rst,

input write\_me, is\_lw\_me, is\_jal\_me, is\_mul\_me,

input [4:0] w\_addr\_me,

input [31:0] alu\_me, mul\_me, npc\_me, dm\_rdata\_me,

output reg write\_w = 1'b0,

output reg is\_lw\_w = 1'b0,

output reg is\_jal\_w = 1'b0,

output reg is\_mul\_w = 1'b0,

output reg [4:0] w\_addr\_w = 5'b0,

output reg [31:0] alu\_w = 32'b0,

output reg [31:0] mul\_w = 32'b0,

output reg [31:0] npc\_w = 32'b0,

output reg [31:0] dm\_rdata\_w = 32'b0

);

// 定义一个任务，用于赋值或复位

task assign\_or\_reset;

input reset;

begin

write\_w <= reset ? 1'b0 : write\_me;

is\_lw\_w <= reset ? 1'b0 : is\_lw\_me;

is\_jal\_w <= reset ? 1'b0 : is\_jal\_me;

is\_mul\_w <= reset ? 1'b0 : is\_mul\_me;

w\_addr\_w <= reset ? 5'b0 : w\_addr\_me;

alu\_w <= reset ? 32'b0 : alu\_me;

mul\_w <= reset ? 32'b0 : mul\_me;

npc\_w <= reset ? 32'b0 : npc\_me;

dm\_rdata\_w <= reset ? 32'b0 : dm\_rdata\_me;

end

endtask

// 使用任务进行逻辑处理

always @(posedge rst or posedge clk) begin

if (rst)

assign\_or\_reset(1'b1); // 复位信号

else

assign\_or\_reset(1'b0); // 更新信号

end

endmodule

**数据存储器 dmem.v：**

module dmem(

input clk, // 时钟信号

input rst, // 复位信号

input write, // 写入使能信号

input [31:0] addr, // 地址信号

input [31:0] data\_in, // 写入数据

output [31:0] data\_out // 输出数据

);

integer i;

reg [31:0] memory[0:2047];

wire [31:0] addr\_TRUE;

assign data\_out = memory[addr\_TRUE[10:0]];

assign addr\_true = addr - 32'h10010000;

always @(posedge rst or negedge clk) begin

if(rst)

begin

for(i=0;i<2048;i=i+1)

memory[i] <= 0;

end

else if(write==1)

memory[addr\_TRUE[10:0]] <= data\_in;

end

endmodule

**寄存器堆 regfiles.v：**

module regfiles (

input clk,

input rst,

input write,

input [4:0] rn1, // 读取寄存器1地址

input [4:0] rn2, // 读取寄存器2地址

input [4:0] w\_addr, // 写寄存器地址

input [4:0] r\_addr, // 读取寄存器3地址

input [31:0] data\_in, // 写入数据

output [31:0] data\_out1, // 输出数据1

output [31:0] data\_out2, // 输出数据2

output [31:0] data\_out // 输出数据3

);

reg [31:0] regs[31:0];

integer i;

//从寄存器读取数据

assign data\_out = (r\_addr)?regs[r\_addr]:0;

assign data\_out1 = (rn1)?regs[rn1]:0;

assign data\_out2 = (rn2)?regs[rn2]:0;

//将数据写入寄存器

always@(negedge clk or posedge rst) begin

if(rst==1) begin

for(i=0;i<32;i=i+1)

begin

regs[i]<=0;

end

end

else if(w\_addr != 0 && write)

regs[w\_addr]<=data\_in;

end

endmodule

**运算器 alu.v：**

module alu(

input [31:0] alu1,//第1个操作数

input [31:0] alu2,//第2个操作数

input [3:0] aluc, //哪种操作

output reg [31:0] result//输出

);

wire [31:0] result0;

wire [31:0] result1;

//定义add

add add\_inst(

.add1(alu1),

.add2(alu2),

.aluc(aluc[0]),

.res(result0)

);

//定义luslt

luislt luislt\_inst(

.alu1(alu1),

.alu2(alu2),

.aluc(aluc[1:0]),

.res(result1)

);

function [31:0] shift\_operation(

input [31:0] value,

input [31:0] shift,

input is\_arithmetic, // 1表示算术右移, 0表示逻辑右移或左移

input is\_left // 1表示左移, 0表示右移

);

reg [31:0] temp;

integer i;

begin

temp = value;

for (i = 0; i < 5; i = i + 1) begin

if (shift[i]) begin

if (is\_left)

temp = temp << (1 << i); // 左移

else if (is\_arithmetic)

temp = $signed(temp) >>> (1 << i); // 算术右移

else

temp = temp >> (1 << i); // 逻辑右移

end

end

shift\_operation = temp;

end

endfunction

// result 最终输出

always @(\*) begin

case (aluc[3:2])

2'b00: result = result0;

2'b10: result = result1;

2'b01: begin

case (aluc[1:0])

2'b00: result = alu1 & alu2;

2'b01: result = alu1 | alu2;

2'b10: result = alu1 ^ alu2;

default: result = ~(alu1 | alu2);

endcase

end

2'b11: begin

case (aluc[1:0])

2'b00: result = shift\_operation(alu2, alu1, 1'b1, 1'b0); // 算术右移

2'b01: result = shift\_operation(alu2, alu1, 1'b0, 1'b0); // 逻辑右移

default: result = shift\_operation(alu2, alu1, 1'b0, 1'b1); // 左移

endcase

end

default: result = 32'b0;

endcase

end

endmodule

运算器包括add.v和luislt.v两部分，

**luislt.v:**

module luislt(

input [31:0] alu1,//第一个操作数

input [31:0] alu2,//第二个操作数

input [1:0] aluc,//运算种类

output reg [31:0] res//输出结果

);

reg [2:0] cmp;

reg res\_low;

// 产生立即数结果

wire [31:0] res\_lui = {alu2[15:0], 16'b0};

wire [31:0] res\_slt = {31'b0, res\_low};

wire [32:0] alur1 = {1'b0, alu1};

wire [32:0] alur2 = {1'b0, alu2};

always @(\*) begin

cmp = (alur1 > alur2) ? 3'b100 :

(alur1 == alur2) ? 3'b010 :

3'b001;

end

always @(\*) begin

case ({aluc[0], alu1[31], alu2[31]})

3'b0\_??: res\_low = cmp[0];

3'b1\_00, 3'b1\_11: res\_low = cmp[0];

3'b1\_01: res\_low = 1'b0;

3'b1\_10: res\_low = 1'b1;

default: res\_low = 1'b0;

endcase

end

always @(\*) begin

res = aluc[1] ? res\_slt : res\_lui;

end

endmodule

**加法器add.v:**

module cla4(

input [3:0] gg, pp,

input data\_in,

output ogg,

output oper,

output [4:1] data\_out

);

wire [4:0] carry; // 包括初始输入 data\_in 和最终 carry\_out

assign carry[0] = data\_in; // 初始输入

// 使用 generate 语句生成进位逻辑

genvar i;

generate

for (i = 1; i <= 4; i = i + 1) begin : generate\_carry

assign carry[i] = gg[i-1] | (pp[i-1] & carry[i-1]);

end

endgenerate

// 连接输出

assign data\_out = carry[4:1]; // 取生成的进位信号

assign ogg = carry[4]; // 最后的进位信号作为 ogg

assign oper = &pp; // 所有 pp 为 1 时，oper 为 1

endmodule

module add1(

input add1, add2,//两个操作数

input c,

//级联生成信号

output gg,

output pp,

output data\_out//输出结果

);

assign pp = add1 | add2;

assign data\_out = add1 ^ add2 ^ c;

assign gg = add1 & add2;

endmodule

module add4(

input [3:0] add1, add2,

input data\_in,

output ogg,

output oper,

output [3:0] s

);

wire [3:0] gg;

wire [3:0] pp;

wire [4:1] c;

genvar i;

generate

for (i = 0; i < 4; i = i + 1) begin : add\_block

if (i == 0)

add1 u\_add(add1[i], add2[i], data\_in, gg[i], pp[i], s[i]);

else

add1 u\_add(add1[i], add2[i], c[i], gg[i], pp[i], s[i]);

end

endgenerate

// cla4模块负责生成进位和最终输出

cla4 u\_cla4(gg, pp, data\_in, ogg, oper, c);

endmodule

module add16(

input [15:0] add1, // 输入信号1，16位宽

input [15:0] add2, // 输入信号2，16位宽

input data\_in, // 输入信号，用于最低位加法的进位输入

output ogg, // 输出进位，表示最高位的进位输出

output oper, // 输出溢出，表示是否有溢出发生

output [15:0] s // 输出结果，16位宽

);

wire [4:1] c; // 模块之间的进位信号

wire [3:0] gg, pp; // 每个4位模块的进位生成和传递

genvar i;

generate

for (i = 0; i < 4; i = i + 1) begin : gen\_add4

if (i == 0) begin

add4 add\_1(add1[3:0], add2[3:0], data\_in, gg[0], pp[0], s[3:0]);

end else begin

add4 add\_i(add1[(i+1)\*4-1:i\*4], add2[(i+1)\*4-1:i\*4], c[i], gg[i], pp[i], s[(i+1)\*4-1:i\*4]);

end

end

endgenerate

cla4 cla4\_inst(gg, pp, data\_in, ogg, oper, c);

endmodule

module add32(

input [31:0] add1, // 输入信号1

input [31:0] add2, // 输入信号2

input data\_in, // 输入信号，用于低位加法的进位输入

output [31:0] res, // 输出结果，32位宽

output data\_out // 输出进位，表示最高位的进位输出

);

wire [1:0] gg;

wire [1:0] pp;

wire TEMP;

assign TEMP = gg[0]|(pp[0]&data\_in);

assign data\_out = gg[1]|(pp[1]&gg[0])|(pp[1]&pp[0]&data\_in);

add16 add\_1\_inst(

.add1(add1[15:0]),

.add2(add2[15:0]),

.data\_in(data\_in),

.ogg(gg[0]),

.oper(pp[0]),

.s(res[15:0])

);

add16 add\_2\_inst(

.add1(add1[31:16]),

.add2(add2[31:16]),

.data\_in(TEMP),

.ogg(gg[1]),

.oper(pp[1]),

.s(res[31:16])

);

Endmodule

module neg32(

input [31:0] data\_in,

input s,

output [31:0] data\_out

);

genvar i;

generate

for (i = 0; i < 32; i = i + 1) begin : neg\_logic

assign data\_out[i] = s ^ data\_in[i];

end

endgenerate

endmodule

**testbench文件**

module cpu\_top\_tb();

reg clk = 1;

reg rst = 1;

reg [4:0] raddr = 5'b00001; // 初始读取地址

wire [31:0] instr;

wire [31:0] pc;

wire [31:0] odata;

integer count = 0;

integer count\_nop = 0;

always #5 clk = ~clk;

initial begin

#1 rst = 0;

end

// 在时钟上升沿统计

always @(posedge clk) begin

count <= count + 1;

if (instr == 32'b0)

count\_nop <= count\_nop + 1;

end

cpu\_top cpu\_top\_inst (

.clk(clk),

.rst(rst),

.raddr(raddr),

.instr(instr),

.pc(pc),

.odata(odata)

);

Endmodule

**Xdc文件：**set\_property PACKAGE\_PIN E3 [get\_ports clk\_in]

set\_property PACKAGE\_PIN M13 [get\_ports enable]

set\_property PACKAGE\_PIN L16 [get\_ports reset]

set\_property PACKAGE\_PIN J15 [get\_ports stop]

set\_property PACKAGE\_PIN R17 [get\_ports choice[1]]

set\_property PACKAGE\_PIN R15 [get\_ports choice[0]]

set\_property PACKAGE\_PIN V10 [get\_ports raddr[4]]

set\_property PACKAGE\_PIN U11 [get\_ports raddr[3]]

set\_property PACKAGE\_PIN U12 [get\_ports raddr[2]]

set\_property PACKAGE\_PIN H6 [get\_ports raddr[1]]

set\_property PACKAGE\_PIN T13 [get\_ports raddr[0]]

set\_property PACKAGE\_PIN T10 [get\_ports {o\_seg[0]}]

set\_property PACKAGE\_PIN R10 [get\_ports {o\_seg[1]}]

set\_property PACKAGE\_PIN K16 [get\_ports {o\_seg[2]}]

set\_property PACKAGE\_PIN K13 [get\_ports {o\_seg[3]}]

set\_property PACKAGE\_PIN P15 [get\_ports {o\_seg[4]}]

set\_property PACKAGE\_PIN T11 [get\_ports {o\_seg[5]}]

set\_property PACKAGE\_PIN L18 [get\_ports {o\_seg[6]}]

set\_property PACKAGE\_PIN H15 [get\_ports {o\_seg[7]}]

set\_property PACKAGE\_PIN J17 [get\_ports {o\_sel[0]}]

set\_property PACKAGE\_PIN J18 [get\_ports {o\_sel[1]}]

set\_property PACKAGE\_PIN T9 [get\_ports {o\_sel[2]}]

set\_property PACKAGE\_PIN J14 [get\_ports {o\_sel[3]}]

set\_property PACKAGE\_PIN P14 [get\_ports {o\_sel[4]}]

set\_property PACKAGE\_PIN T14 [get\_ports {o\_sel[5]}]

set\_property PACKAGE\_PIN K2 [get\_ports {o\_sel[6]}]

set\_property PACKAGE\_PIN U13 [get\_ports {o\_sel[7]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[7]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_sel[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[7]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {o\_seg[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports clk\_in]

set\_property IOSTANDARD LVCMOS33 [get\_ports enable]

set\_property IOSTANDARD LVCMOS33 [get\_ports reset]

set\_property IOSTANDARD LVCMOS33 [get\_ports stop]

set\_property IOSTANDARD LVCMOS33 [get\_ports choice[1]]

set\_property IOSTANDARD LVCMOS33 [get\_ports choice[0]]

set\_property IOSTANDARD LVCMOS33 [get\_ports raddr[4]]

set\_property IOSTANDARD LVCMOS33 [get\_ports raddr[3]]

set\_property IOSTANDARD LVCMOS33 [get\_ports raddr[2]]

set\_property IOSTANDARD LVCMOS33 [get\_ports raddr[1]]

set\_property IOSTANDARD LVCMOS33 [get\_ports raddr[0]]

create\_clock -period 100.000 -name clk\_pin -waveform {0.000 50.000} [get\_ports clk\_in]

set\_input\_delay -clock [get\_clocks \*] 1.000 [get\_ports reset]

set\_output\_delay -clock [get\_clocks \*] 0.000 [get\_ports -filter { NAME =~ "\*" && DIRECTION == "OUT" }]