
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：2252431

姓名：孙骁远

指导教师：秦国锋

日期：2024.11.3

一、 实验环境部署与硬件配置说明

1. 实验环境部署：windows11 系统，使用 vivado2016.2 作为开发工具，vivado 自带的仿真工具进行仿真

2. 硬件配置说明：Xilinx FPGA 器件，Nexys DDR 开发板

二、 实验的总体结构

1. 指令选取

选取常用指令 8 条，以完成指定的功能，依次为 ADD，NOP，HALT，LOAD，STORE，CMP，BZ，BN

2. 指令码设计

用 4 位数表示操作码可以很好的表示 8 条指令，4 位表示一个寄存器或者立即数，后 8 位可以拼接起来表示一个立即数。具体设计如下，该 CPU 可执行 8 条指令，拥有 16 个通用寄存器

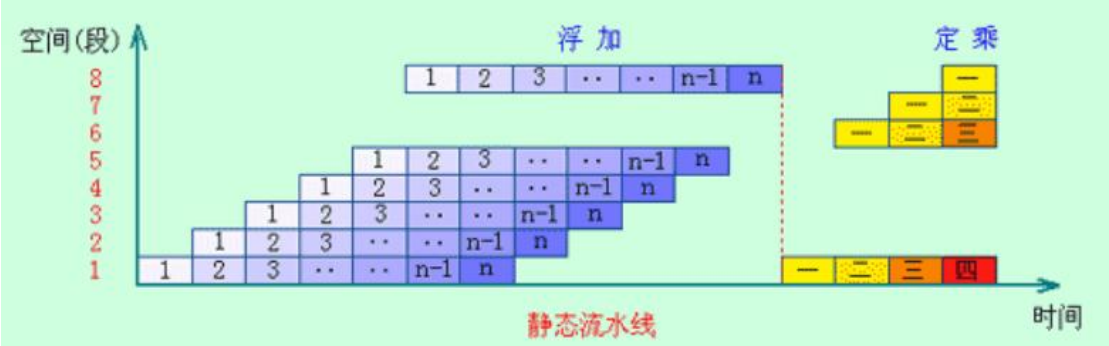
**指令码格式**

| 种类 | 操作码 | op1 | op1 | op1 | operation |
|---|---|---|---|---|---|
| ADD | 0010 | r1 | r2 | r3 | r1<-r2+r3 |
| NOP | 0000 | 0000 | 0000 | 0000 | no operation |
| HALT | 0001 | 0000 | 0000 | 0000 | halt |
| LOAD | 1101 | r1 | r2 | val | r1<-dmem[r2+val] |

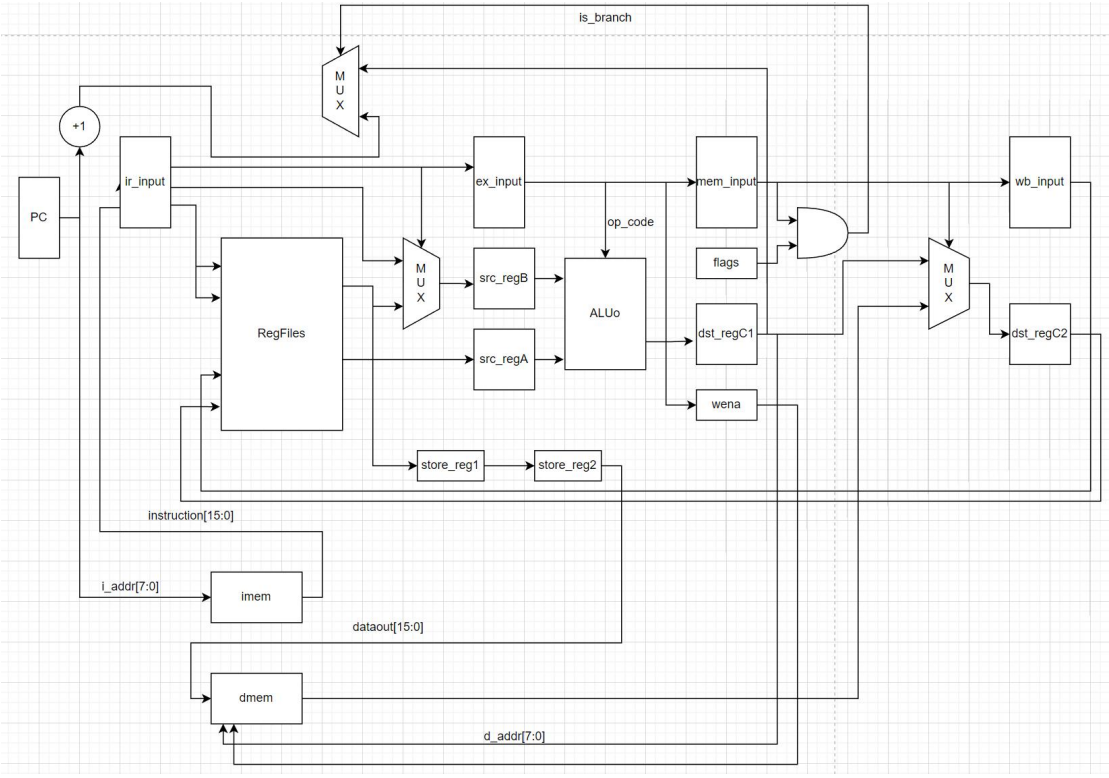
| | | | | | |
|---|---|---|---|---|---|
| STORE | 1110 | r1 | r2 | val | r1->dmem[r2+val] |
| CMP | 0111 | 0000 | r2 | r3 | r2-r3<br>set cf, nf, zf |
| BZ | 1011 | r1 | val1 | val2 | jump to r1+{val1,val2} |
| BN | 1001 | r1 | val1 | val2 | jump to r1+{val1,val2} |

## 3. 静态流水线的总体结构

**静态流水线的时空图**




**总体结构总线图**



## 4. 结构解释

①总体使用哈佛结构，将程序指令存储和数据存储物理上分开，即拥有两个独立的存储器，分别用于存储指令和数据。同时通过数据总线和指令总线和 CPU 进行数据交换。

②流水线的设计分为 IF，ID，EX，MEM，WB 五个阶段，具体解释如下：

1. **取指阶段（IF）：**

- 从内存或指令缓存中获取下一条指令，并将其传送到流水线中。
- 负责维护程序计数器（PC），用于指向当前要执行的指令地址。

2. **指令解码阶段（ID）：**

- 解码取到的指令以确定操作码和操作数。
- 根据指令类型，选择适当的寄存器数据，准备执行操作。
- 可能涉及寄存器堆的读取，标志位检查，以及分支预测的决策。

3. **执行阶段（EX）：**

- 在算术逻辑单元（ALU）中执行算术和逻辑操作。
- 对于分支指令，此阶段也可以评估跳转条件并决定是否进行分支。
- 计算内存访问指令的地址。

4. **存储器访问阶段（ MEM）：**

- 如果指令需要数据访问（如 LOAD 或 STORE），该阶段进行内存读取或写入操作。
- 负责从数据存储器或缓存中提取数据，或将数据写入内存。

5. **写回阶段（WB）：**

- 将计算或内存访问的结果写回到寄存器堆中。

- 确保数据可供后续指令使用，完成指令的生命周期。

### ③通过设计多个寄存器实现指令的流水执行

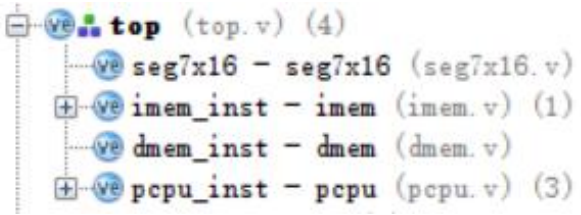
**时空图如下**

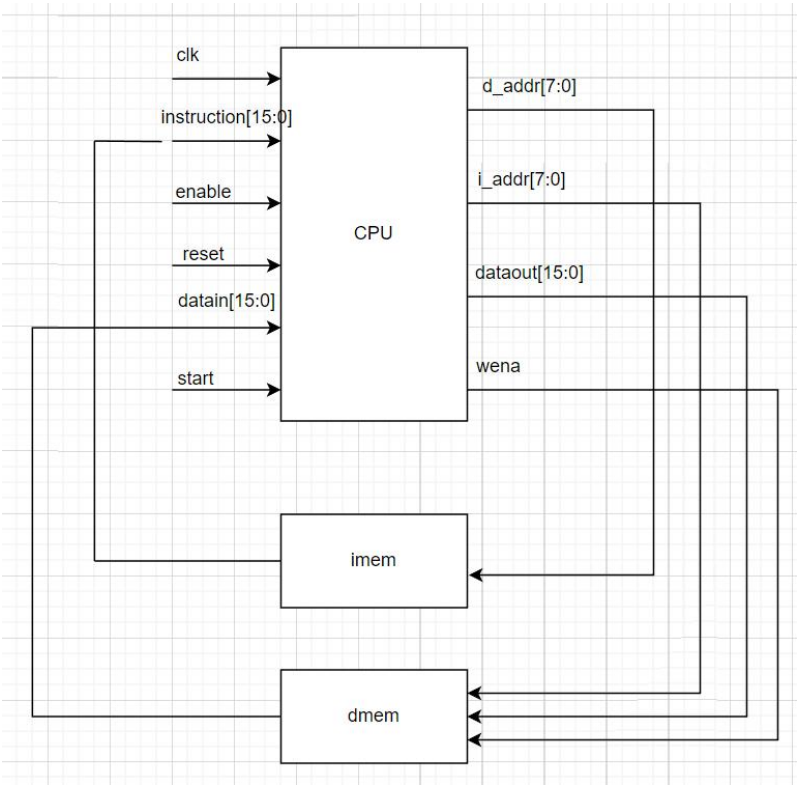
| | 时钟周期 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 指令编号 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 指令i | IF | ID | EX | MEM | WB | | | | |
| 指令i+1 | | IF | ID | EX | MEM | WB | | | |
| 指令i+2 | | | IF | ID | EX | MEM | WB | | |
| 指令i+3 | | | | IF | ID | EX | MEM | WB | |
| 指令i+4 | | | | | IF | ID | EX | MEM | WB |

## 三、 总体架构部件的解释说明

### 1、 8 条指令流水线总体结构部件的解释说明

主体分为 4 部分，pcpu，imem，dmem，seg7x16；imem 和 dmem 通过指令总线和数据总线与 CPU 进行数据的交换，pcpu 实现流水线的具体功能，seg7x16 负责在数码管上显示数据。利用 top.v 模块将以上几部分连接在一起






pcpu 又分为 IF，EX，MEM 三个子模块，并且在 pcpu.v 里实现 ID，WB 的功能


## 2、 静态流水线具体部件的解释说明

**主要变量如下**

```
reg cpu_state, next_cpu_state;//CPU的状态
reg [15:0] general_reg[15:0]; //通用寄存器堆
reg [15:0] src_regA, src_regB;//两个源操作数寄存器

wire [15:0] id_input, ex_input, mem_input, wb_input;//各阶段的指令传递

wire is_branch;//是否进行跳转的标志位
wire zf, nf, cf, cf_buf;//运算的标志位
wire [15:0] dst_regC1, dst_regC2, store_reg1, store_reg2, ALU_result;//两个目标寄存器，两个保证流水线不断
```

①PC 寄存器：用于存储下一条要执行的指令的地址，设计中地址位 8 位，即指令存储器的深度为 256

②id_input，ex_input，mem_input，wb_input：存放对应阶段流入的指令，确保了指令在 CPU 的不同阶段之间流水式传递，提高了指令执行的效率

③RegFiles：16 个通用寄存器堆，用于存储 CPU 中的临时数据和变量

④src_regA，src_regB：两个源操作数寄存器，在 ID 阶段根据指令类型的不同将这两个寄存器进行赋值

⑤store_reg1，store_reg2：用于 STORE 指令的两个寄存器，使用两个以保证两个 STORE 指令可以连续执行（流水）

⑥ALU：运算部件，在 EX 阶段进行计算，并置标志位，能进行加法，减法，移位等操作

⑦dst_regC1，dst_regC2：两个结果寄存器，分别在 MEM 和 WB 阶段将结果传出，设置两个是为了保证指令的流水执行

⑧wena：使能信号（写信号）有效，当它是高电平时，允许数据写入数据存储器

⑨flag：各种标志位寄存器，如 nf，zf，cf

⑩imem，dmem：指令存储器和数据存储器，指令存储器位 ROM，数据存储器为 RAM，在使能信号和写入信号有效的上升沿写入，指令和数据的宽度均为 16 位，深度位 256。

## 四、 实验仿真过程

### 1、 testbench 的仿真过程

编写 testbench 的 top_tb.v 文件，设置时钟周期，使用 vivado 自带的仿真器进行仿真，查看寄存器堆中的数据与 Visual Studio 的数据是否相同，确保 CPU 运算的正确性

### 2、 下板过程

编写用于仿真测试的 C 程序，将 C 程序转化为对应的指令序列，保证指令在实现的指令范围内，同时使用 NOP 指令和调整指令顺序方式解决冲突。

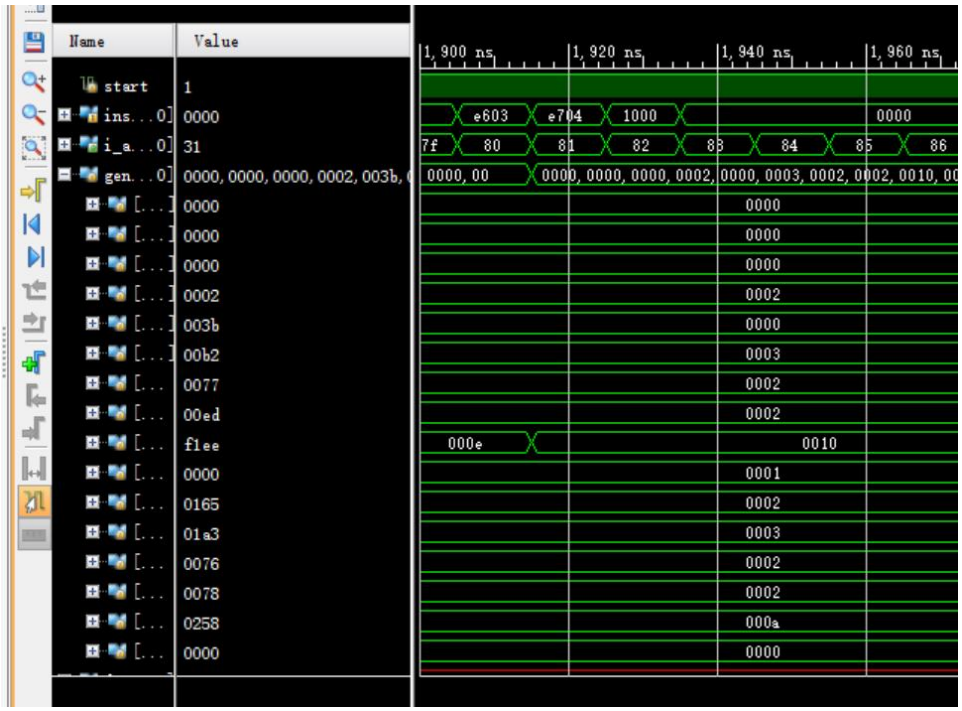
将指令序列转为 mips32 汇编语言，形成 coe 文件

使用 coe 文件初始化 ip 核，带入 imem 中，配置 xdc 文件进行下板仿真

## 五、 实验仿真的波形图及某时刻寄存器值的物理意义

### 1、 8 条指令流水线的波形图

包含 start（开始信号），instruction（指令），i_addr（指令地址），general_reg（通用寄存器堆）




2、 寄存器值的物理意义

①cpu_state、next_cpu_state：CPU 当前和下一个状态寄存器

②id_input、ex_input、mem_input、wb_input，存储正在某个阶段执行的对应的指令

③src_regA，src_regB 两个源操作数寄存器

④dst_regC1，dst_regC2 两个目标寄存器，两个保证流水线不断

⑤store_reg1，store_reg2 用于 store 指令的两个寄存器，两个保证流水线不断

⑥general_reg[15:0]通用寄存器堆

⑦zf，nf，cf 运算的标志位

⑧ALU_result 运算器的运算结果

⑨cf_buf 存储溢出位

## 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。用你的模型评价该游戏在两个不同历史时期花费的总成本 f=m*p1+n*p2+h*p3，m 为上的楼层总数，n 为下的楼层总数，h 为摔破的鸡蛋总数，p1 为每上 1 层的成本，p2 为每下 1 层的成本，p2 为每个鸡蛋的成本；在物质匮乏时期，p1=2，p2=1，p3=4；在人力成本增长时期，p1=4，p2=1，p3=2。请使用 C 语言设计该验证模型的算法，并把 C 语言汇编为 MIPS 或 RISC-V 指令汇编程序，同时利用编译器生成 MIPS 或 RISC-V 指令集可执行目标程序。

1.首先使用二分法查找耐摔值，计算出摔的总次数，摔的总鸡蛋数，最后一个鸡蛋是否摔碎（这里楼层数 20 和耐摔值 2 已经写定）。同时根据中间楼层，high，low 的值计算总上楼层数，总下楼层数

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int totalFloors = 20; // 楼层数
    int fallResistance = 2; // 鸡蛋的耐摔值
    int result = 0; // 最终结果
    int count = 0; // 总次数
    int totalEggs = 0; // 摔碎的总鸡蛋数
    int isBroken = 0; // 最后一个鸡蛋是否破碎（1破碎）

    int high = totalFloors, low = 1;
    int stumbFloor = 0;

    int totalUpFloors = 0; // 总上楼层数
    int totalDownFloors = 0; // 总下楼层数
```

```c
while (1) {
    count++;
    stumbFloor = (high + low) / 2;

    if (stumbFloor <= fallResistance) {
        isBroken = 0;
        totalUpFloors += (stumbFloor - low); // 记录上楼层数
        low = stumbFloor;
    }
    else {
        isBroken = 1;
        totalEggs++;
        totalDownFloors += (high - stumbFloor); // 记录下楼层数
        high = stumbFloor - 1;
    }

    if ((high - low) < 2) {
        break;
    }
}
```

```c
count++;
if (high <= fallResistance) {
    isBroken = 0;
    result = high;
    totalUpFloors += (high - low); // 最后一次的上楼层数
}
else {
    isBroken = 1;
    totalEggs++;
    result = low;
    totalDownFloors += (high - low); // 最后一次的下楼层数
}
```

然后计算两个时期的成本

```
// 成本计算
int p1_materialScarcity = 2, p2_materialScarcity = 1, p3_materialScarcity = 4;
int p1_laborCostIncrease = 4, p2_laborCostIncrease = 1, p3_laborCostIncrease = 2;

// 物质匮乏时期的总成本
int cost_materialScarcity = totalUpFloors * p1_materialScarcity +
    totalDownFloors * p2_materialScarcity +
    totalEggs * p3_materialScarcity;

// 人力成本增长时期的总成本
int cost_laborCostIncrease = totalUpFloors * p1_laborCostIncrease +
    totalDownFloors * p2_laborCostIncrease +
    totalEggs * p3_laborCostIncrease;

printf("Total drops: %d, Total broken eggs: %d, Last egg broken status: %d\n", count, totalEggs, isBroken);
printf("Cost in material scarcity period: %d\n", cost_materialScarcity);
printf("Cost in labor cost increase period: %d\n", cost_laborCostIncrease);
```

2. 编写对应的汇编指令，根据加减关系在 Mars 中写出汇编指令

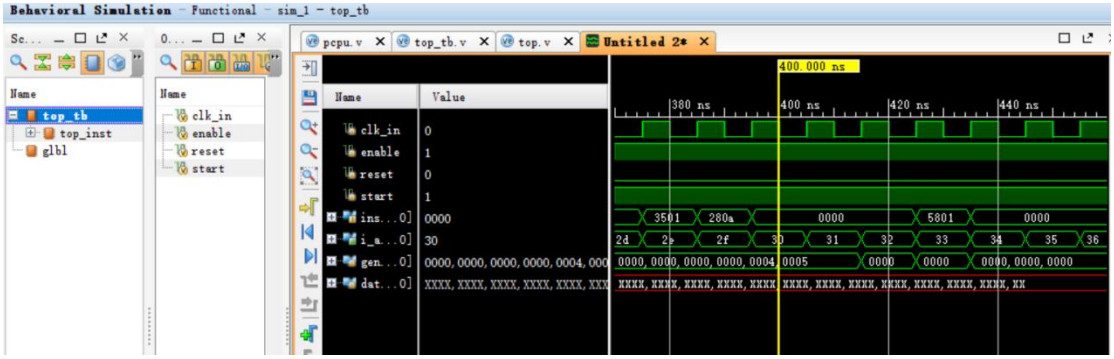
```
# stumbFloor = (high + low) / 2
add $t4, $t1, $t2        # t4 = high + low
srl $t3, $t4, 1          # stumbFloor = (high + low) / 2

# if (stumbFloor <= fallResistance)
lw $t5, fallResistance
ble $t3, $t5, not_broken

# broken:
# 记录鸡蛋破碎
lw $t0, totalEggs
addi $t0, $t0, 1
sw $t0, totalEggs
li $t0, 1
sw $t0, isBroken

# totalDownFloors += (high - stumbFloor)
lw $t0, totalDownFloors
sub $t6, $t1, $t3        # t6 = high - stumbFloor
add $t0, $t0, $t6
sw $t0, totalDownFloors
```

3. 转为 16 进制 coe 文件

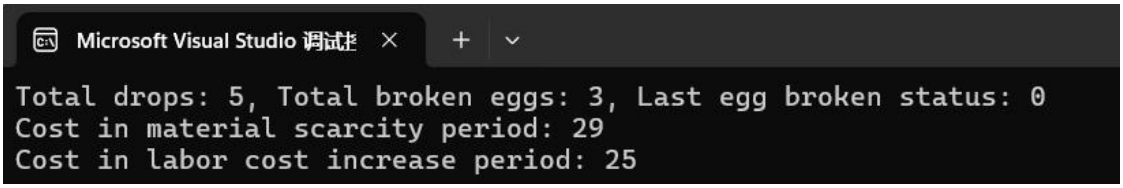
```
memory_initialization_radix = 16;
memory_initialization_vector =
24080014
3c011001
ac280000
24080002
3c011001
ac280004
```

4. 将 coe 文件导入并进行仿真

dmem 初始值未知，所以为 xxxxxxxx




5. 结果比较




```
Total drops: 5, Total broken eggs: 3, Last egg broken status: 0

Cost in material scarcity period: 29

Cost in labor cost increase period: 25

— program is finished running —
```

## 七、 实验验算程序下板测试过程与实现

### 1、配置顶层文件，加入分频和 7 段数码管以方便观察现象

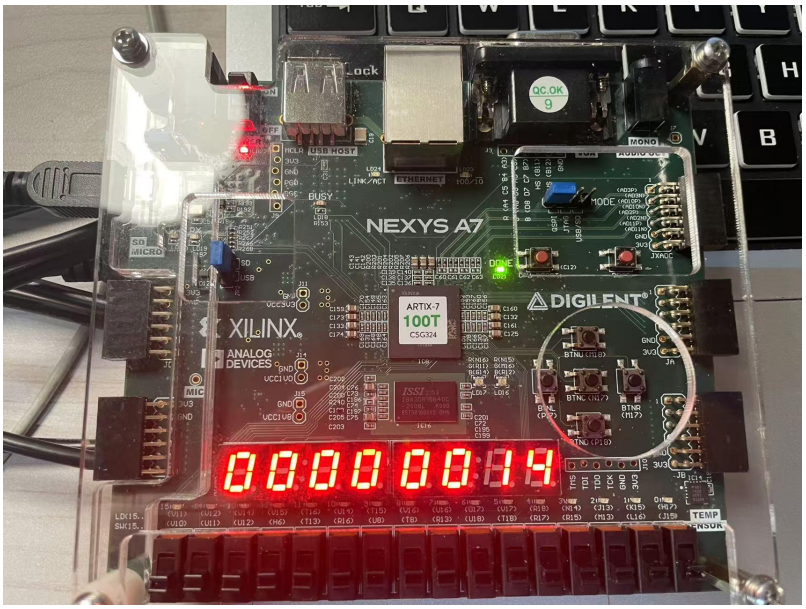
```
module top(
    input clk_in,
    input enable,
    input reset,
    input start,
    output [7:0] o_seg,
    output [7:0] o_sel
    );
    wire [15:0] instruction;
    wire [15:0] datain;
    wire [7:0] i_addr;
    wire [7:0] d_addr;
    wire wena;
    wire [15:0] dataout;

    reg [24 : 0] cnt;
    always @ (posedge clk_in, negedge reset)
    if (reset)
        cnt <= 0;
    else
        cnt <= cnt + 1'b1;
    wire clk_cpu = cnt[24];
    seg7x16(clk_in, reset, 1, {8'b0000_0000, instruction}, o_seg, o_sel);
    pcpu pcpu_inst(clk_cpu, enable, reset, start, instruction, datain, i_addr, d_addr, wena, dataout);
    imem imem_inst(i_addr, instruction);
    dmem dmem_inst(clk_cpu, enable, wena, d_addr, dataout, datain);
endmodule

```

# 2、配置 xdc 文件

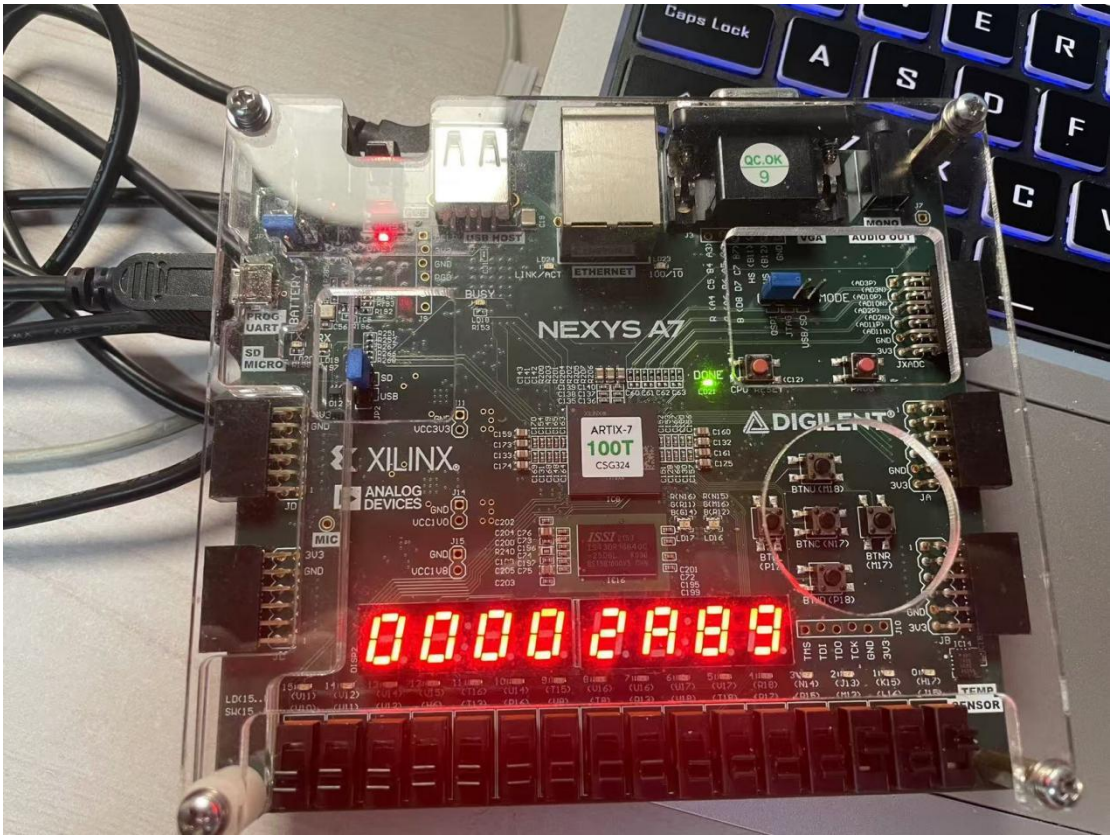
```
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {o_seg[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports enable]
set_property IOSTANDARD LVCMOS33 [get_ports start]
set_property IOSTANDARD LVCMOS33 [get_ports clk_in]

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" }]
```

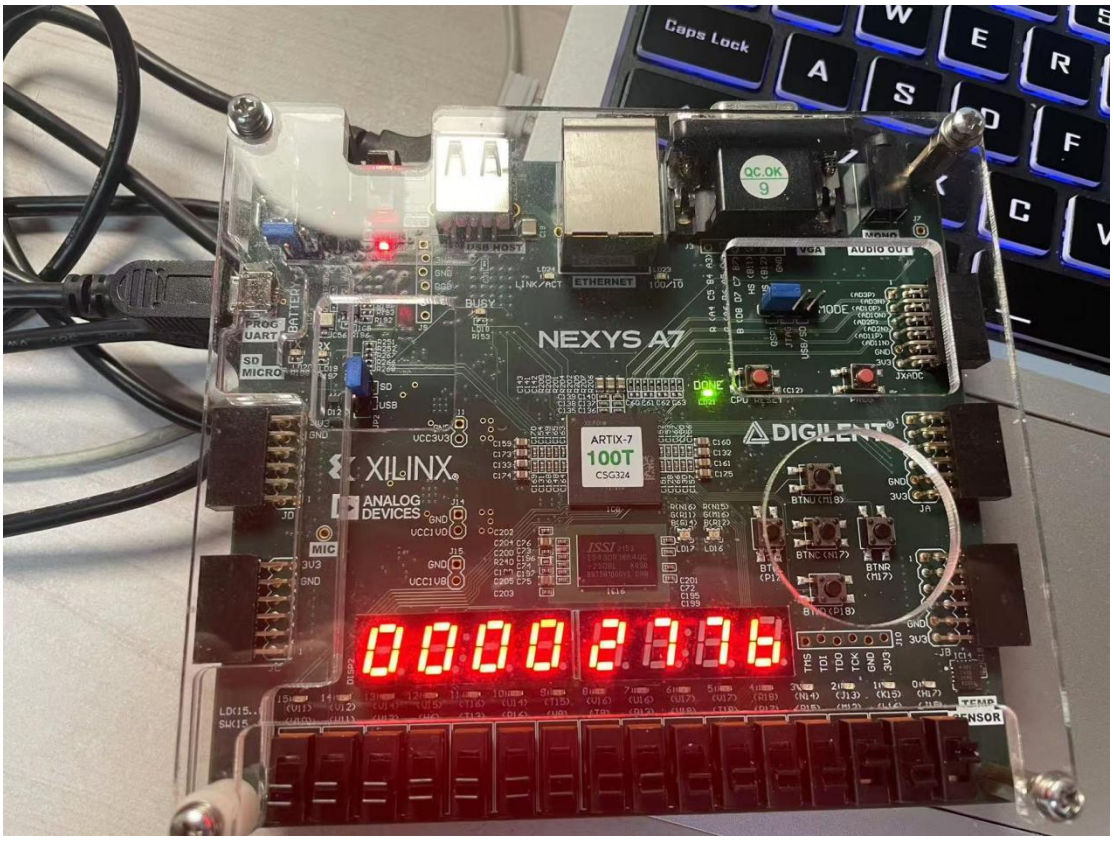
# 3、综合，布线，下板

实验结果如下图所示，成功显示第一条指令

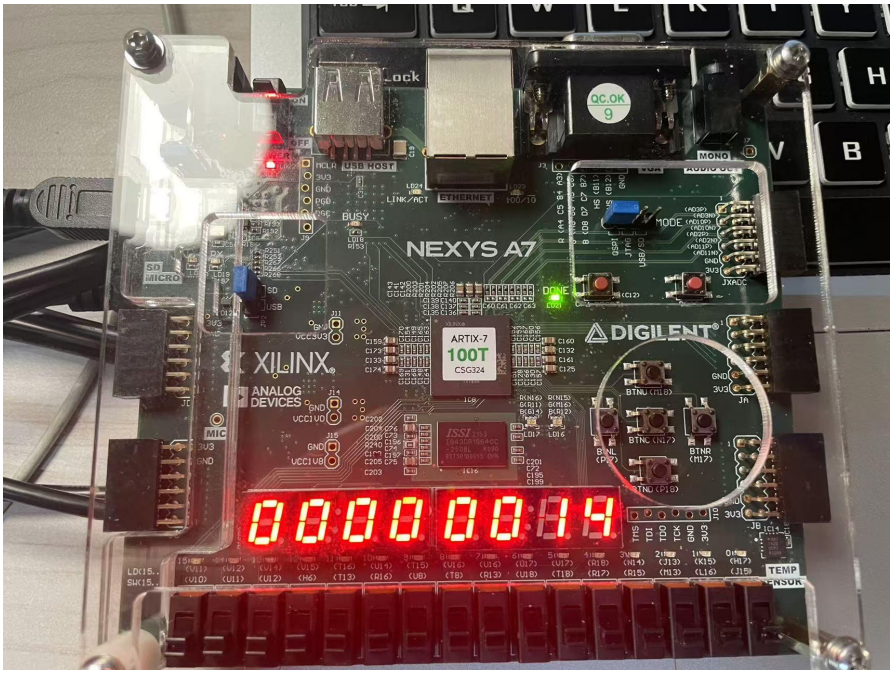


开启 enable 和 start 能够正常流动



关闭 enable 能停止流动，reset 能复位




## 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）

通过阅读时序报告可知延迟大概在 90ns 左右，因此时钟周期最短可设置为 90ns 附近

吞吐率：最大吞吐率=1/t=11111111，实际为 10893246

加速比：(n*m)/(m+n-1)，当 n 趋近于无穷大时，加速比趋近于 5，在测试程序中实际为 4.901

效率：n*m/(m*(m+n-1))，当 n 趋近于无穷大时，效率趋近于 1，实际测试程序中约为 0.9804

冲突分析：在处理先写后读的数据冲突时，我们可以通过插入 NOP（无操作）指令或者重新安排指令的执行顺序来缓解这种冲突。NOP 指令作为一种占位符，可以暂时占用 CPU 周期而不执行任何实际操作，从而为数据写入提供必

要的时间缓冲，确保数据在被读取前已经稳定。然而，在处理跳转指令时，情况则有所不同。跳转指令依赖于条件标志位或地址的计算结果，而这些结果可能因为计算延迟而无法即时可用。这种延迟导致的数据冲突，我们称之为控制冲突，它不能通过简单的指令重排来解决，因为跳转决策本身就依赖于这些延迟的结果。在这种情况下，NOP 指令成为必要的缓冲手段，以等待跳转条件的最终确定。尽管 NOP 指令在解决这些冲突中发挥了重要作用，但它们也导致了 CPU 资源的浪费，因为 CPU 在执行 NOP 时实际上是在空转，没有进行任何有价值的计算。这种空转不仅降低了 CPU 的工作效率，还可能影响整体的程序性能。因此，在设计 CPU 和编译器时，需要权衡 NOP 指令的使用，以优化程序的执行效率和资源利用率。

CPU 运行时间：测试程序大约执行了几百条指令，运行时间在 20000ns 左右

存储器空间的使用：使用指令存储器和数据存储器。测试程序只在最后 STORE 了 5 个值进数据存储器，其余计算均在寄存器，因此数据存储器空间消耗为 5，指令存储器消耗空间为指令的数量为 130。

# 九、 总结与体会

在本次实验中，我设计了一套包含 8 条指令的指令集，并且用二进制码表示这些指令，并且成功构建了一个五阶段流水线 CPU，用以执行这些指令。随后，我编写了一个模拟“摔鸡蛋”问题的验证程序，将其编译成汇编语言，并进一步转换为机器码，加载到 CPU 中进行执行，以验证程序的正确性和性能。

通过这一过程，我不仅深入理解了指令设计的基本方法和核心原理，还掌握了流水线 CPU 的设计和实现技巧，深刻理解了流水线中每个阶段的关键作用及其重要性。在将汇编语言程序转换为机器码的过程中，我学会了如何巧妙地运用 NOP 指令和调整指令执行顺序来解决流水线中可能出现的数据冲突和控制冲突，这些技巧对于保证流水线的顺畅运行至关重要。

这次实验不仅加深了我对指令设计、流水线 CPU 设计以及性能评估的理解，还让我对解决指令冲突的策略有了更加深刻的认识。通过实践，我体会到了理论应用于实际问题解决过程中的挑战与乐趣，这对我的学习和未来的研究工作都有着极大的帮助。

# 十、附件（所有程序）

**top.v：顶层模块，用于实例化 pcpu，imem，dmem 三个模块**

```
module top(
    input clk_in,//输入时钟信号
    input reset,//复位信号
    input enable,//使能信号
    input start,//开始
    output [7:0] o_seg,//7 段显示器的显示数据输出
    output [7:0] o_sel//7 段显示器的位选信号
    );

    wire [7:0] i_addr,d_addr;//取指令，读数据位置
    wire [15:0] instruction, datain, dataout;//指令代码，写入数据，读出数据
    wire wena;//高电平使能信号

    reg [24 : 0] cnt;
    always @ (posedge clk_in, negedge reset) begin
    if (reset)
        cnt <= 0;
    else
        cnt <= cnt + 1'b1;
    end
    wire clk_cpu = cnt[24];
    //######  实例化 seg7x16 模块  ########//
    seg7x16 display_inst(
        .clk(clk_in),
        .reset(reset),
        .cs(1'b1),
        .i_data({8'b0, instruction}),
        .o_seg(o_seg),
        .o_sel(o_sel)
    );
    //######  实例化 imem 模块  ######//
    imem imem_inst(
```

```
        .pc(i_addr),
        .instr(instruction)
    );
    //######  实例化 dmme 模块  ######//
    dmem dmem_inst(
        .clk(clk_cpu),
        .ram_ena(enable),
        .wena(wena),
        .addr(d_addr),
        .data_in(datain),
        .data_out(dataout)
    );
    //######  实例化 pcpu 模块  ######//
    pcpu pcpu_inst(
        .clk(clk_cpu),
        .enable(enable),
        .reset(reset),
        .start(start),
        .instruction(instruction),
        .datain(datain),
        .i_addr(i_addr),
        .d_addr(d_addr),
        .wena(wena),
        .dataout(dataout)
    );
Endmodule
```

**seg7x16 模块：用于 7 段数码管显示**

```
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;
    always @ (posedge clk, posedge reset) 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
```

### imem 模块：实现指令存储器并初始化 ip 核

```
module imem(
    input [7:0] pc,         // 程序计数器
    output [15:0] instr // 输出 16 位指令
);
dist_mem_gen_0 dist_mem_gen_0_inst(pc, instr);
endmodule
```

### dmem 模块：实现数据存储器

```
module dmem(
    input clk,                      // 存储器时钟信号
    input ram_ena,                  // 存储器有效信号
    input wena,                     // 写有效信号
    input [7 : 0] addr,          // 输入地址
    input [15 : 0] data_in,   // 写入数据
    output reg [15 : 0] data_out // 输出数据
    );
    reg [15:0] memory [255:0];

    always @(*) begin
        if (ram_ena == 1'b1)
            data_out = memory[addr];
        else
            data_out = 16'bz;
    end
```

```
    always @ (posedge clk) begin
        if (ram_ena && wena)
            memory[addr] = data_in;
    end

endmodule
```

**pcpu 模块：给出 IF，EX，MEM 的模块接口并且实现 ID，WB 的功能**

```
`define idle 1'b0//CPU 待机状态
`define exec 1'b1//CPU 运行状态

//定义指令的操作码
`define NOP     4'b0000
`define HALT    4'b0001
`define ADD     4'b0010
`define CMP     4'b0111
`define BN      4'b1001
`define BZ      4'b1011
`define LOAD    4'b1101
`define STORE 4'b1110

module pcpu(
    input clk,                          //时钟信号
    input enable,                       //使能信号
    input reset,                        //复位信号
    input start,                        //CPU 启动信号
    input [15:0] instruction,//输入的指令
    input [15:0] datain,//输入的数据
    output wire [7:0] i_addr,//指令地址
    output [7:0] d_addr,//数据地址
    output wire wena,//数据写入信号，高电平有效
    output [15:0] dataout//输出的数据
    );

    reg cpu_state, next_cpu_state;//CPU 的状态
    reg [15:0] general_reg[15:0]; //通用寄存器堆
    reg [15:0] src_regA, src_regB;//两个源操作数寄存器

    wire [15:0] id_input, ex_input, mem_input, wb_input;//各阶段的指令传递

    wire is_branch;//是否进行跳转的标志位
    wire zf, nf, cf, cf_buf;//运算的标志位
```

```
    wire [15:0] dst_regC1, dst_regC2, store_reg1, store_reg2, ALU_result;//两个目标寄存器，两个保证流水线不断

    //###################CPU 状态切换#######################//
    //每次时钟上升沿切换到下一个状态，若遇到复位信号则待机
    always @ (posedge clk) begin
        if (reset)
            cpu_state <= `idle;
        else
            cpu_state <= next_cpu_state;
    end
    //根据当前状态和输入确定下一个状态

    always @(*) begin
        case (cpu_state)
            `idle: next_cpu_state <= (enable && start) ? `exec : `idle;
            `exec: next_cpu_state <= (enable && wb_input[15:12] != `HALT) ? `exec : `idle;
            default: next_cpu_state <= `idle;
        endcase
    end

    //###################IF 阶段#######################//
    IF if_stage(clk, reset, instruction, cpu_state, is_branch, dst_regC1, id_input, i_addr);

    //###################ID 阶段#######################//
    assign ex_input = 16'b0000_0000_0000_0000;
    assign store_reg1 = (id_input[15:12] == `STORE) ? general_reg[id_input[11:8]] : store_reg1;
    always @ (posedge clk or posedge reset)
    begin
        if (reset)
        begin
            src_regA <= 16'b0000_0000_0000_0000;
            src_regB <= 16'b0000_0000_0000_0000;
            //store_reg1 <= 16'b0000_0000_0000_0000;
        end//复位
        else if (cpu_state == `exec)
        begin
             if ( id_input[15:12] == `BN || id_input[15:12] == `BZ)
                src_regA <= general_reg[id_input[11:8]];
            else if (id_input[15:12] == `ADD || id_input[15:12] == `CMP ||    id_input[15:12] == `LOAD ||    id_input[15:12] == `STORE)
                src_regA <= general_reg[id_input[7:4]];
            else
                src_regA <= src_regA;//根据指令类型置源操作数寄存器 A
```

```
            if (id_input[15:12] == `LOAD || id_input[15:12] == `STORE)
                src_regB <= {12'b0000_0000_0000, id_input[3:0]};
            else if ( id_input[15:12] == `BN ||    id_input[15:12] == `BZ )
                src_regB <= {8'b0000_0000, id_input[7:0]};
            else if (id_input[15:12] == `ADD || id_input[15:12] == `CMP)
                src_regB <= general_reg[id_input[3:0]];
            else
                src_regB <= src_regB;//根据指令类型置源操作数寄存器 B
        end
    end

    //###################EX 阶段######################//
    EX ex_stage(clk, reset, cpu_state, ex_input, src_regA, src_regB, store_reg1,
                ALU_result, cf_buf, zf, nf, cf, mem_input, dst_regC1, store_reg2, wena);

    //###################MEM######################//
    MEM mem_stage(clk, reset, cpu_state, mem_input, datain, dst_regC1, nf,
        zf, wb_input, dst_regC2, d_addr, dataout, is_branch, store_reg2);

    //###################WB 阶段######################//
    integer i;
    always @ (posedge clk or posedge reset)
        begin
        if (reset)begin
            for (i = 0; i < 16; i = i + 1) begin
                general_reg[i] <= 16'b0;
            end
        end
        else if (cpu_state == `exec)
        begin
            if (wb_input[15:12] == `ADD || wb_input[15:12] == `LOAD)
                general_reg[wb_input[11:8]] <= dst_regC2;
        end
    end
endmodule
```

### IF 模块：简单实现取指功能

```
`define NOP 4'b0000
`define exec 1'b1

module IF(
    input clk,                          // 时钟信号
```

```
    input reset,                        // 复位信号
    input [15:0] instruction,           // 输入的指令
    input cpu_state,                    // 当前 CPU 状态
    input is_branch,                    // 跳转标志位
    input [15:0] dst_regC1,             // 目标寄存器 C1 的值
    output reg [15:0] id_input,         // 传递到 ID 阶段的指令
    output reg [7:0] i_addr             // 指令地址
);

    // IF 阶段逻辑
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            id_input <= 16'b0000_0000_0000_0000;
            i_addr <= 8'b0000_0000;
        end else if (cpu_state == `exec) begin
            id_input <= instruction;
            i_addr <= is_branch ? dst_regC1[7:0] : i_addr + 1;
        end
    end

endmodule
```

## EX.v：简单实现流水线 CPU 中的执行（EX）阶段

```
`define idle 1'b0 // CPU 待机状态
`define exec 1'b1 // CPU 运行状态

// 定义指令的操作码
`define NOP     4'b0000
`define HALT    4'b0001
`define ADD     4'b0010
`define LOAD    4'b1101
`define STORE 4'b1110
`define CMP     4'b0111
`define BZ      4'b1011
`define BN      4'b1001

module EX (
    input clk,
    input reset,
    input cpu_state,
    input [15:0] ex_input,              // 输入指令
    input [15:0] src_regA,              // 源寄存器 A
    input [15:0] src_regB,              // 源寄存器 B
```

```
    output reg [15:0] ALU_result,    // ALU 运算结果
    output reg cf_buf,                // 进位标志
    output reg zf,                    // 零标志
    output reg nf,                    // 负标志
    output reg cf,                    //进位标志
    output reg [15:0] dst_regC1,      // 目标寄存器 1
    input [15:0] store_reg1,          // 存储寄存器值
    output reg [15:0] store_reg2,     // 存储数据
    output reg wena,                  // 写使能信号
    output reg [15:0] mem_input       // MEM 阶段输入
);

    // Perform operations in the EX stage
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            mem_input <= {`NOP, 12'b0};
            dst_regC1 <= 16'b0;
            store_reg2 <= 16'b0;
            wena <= 1'b0;
            zf <= 1'b0;
            nf <= 1'b0;
            cf_buf <= 1'b0; // 初始化进位标志
        end else if (cpu_state == `exec) begin
            mem_input <= ex_input; // 将输入指令传递到 MEM 阶段

            // Update dst_regC1 and flags for arithmetic operations
            case (ex_input[15:12])
                `ADD: begin
                    {cf_buf, ALU_result} <= src_regA + src_regB; // 执行加法
                    dst_regC1 <= ALU_result;
                end
                `CMP: begin
                    {cf_buf, ALU_result} <= src_regA - src_regB; // 执行比较
                    dst_regC1 <= ALU_result;
                end
                `STORE: begin
                    wena <= 1'b1; // 允许写操作
                    store_reg2 <= store_reg1; // 保存要存储的数据
                end
                default: begin
                    ALU_result <= 16'b0; // 其他指令默认为 0
                    dst_regC1 <= 16'b0; // 保持目标寄存器 1 为 0
                end
            endcase
```

```
                // 更新标志位
                zf <= (ALU_result == 16'b0);
                nf <= ALU_result[15];
            end
        end

endmodule
```

## MEM.v：简单实现流水线 CPU 中的存储器访问（MEM）阶段

```
`define idle 1'b0 // CPU 待机状态
`define exec 1'b1 // CPU 运行状态

// 定义指令的操作码
`define NOP     4'b0000
`define HALT    4'b0001
`define ADD     4'b0010
`define LOAD    4'b1101
`define STORE 4'b1110
`define CMP     4'b0111
`define BZ      4'b1011
`define BN      4'b1001

module MEM (
    input clk,                          // 时钟
    input reset,                        // 复位信号
    input cpu_state,                    // CPU 状态
    input [15:0] mem_input,             // MEM 阶段的输入指令
    input [15:0] datain,                // 输入的数据
    input [15:0] dst_regC1,             // EX 阶段传递的目标寄存器值
    input nf,                           // 负标志位
    input zf,                           // 零标志位
    output reg [15:0] wb_input,         // WB 阶段的输入
    output reg [15:0] dst_regC2,        // 最终目标寄存器的值
    output [7:0] d_addr,                // 数据地址
    output [15:0] dataout,              // 输出的数据
    output is_branch,                   // 是否进行跳转的标志位
    input [15:0] store_reg2             // 存储寄存器值
);

    // 数据地址和输出数据
    assign d_addr = dst_regC1[7:0];
    assign dataout = store_reg2;
```

```
    // 跳转条件判断（BN,BZ 指令）
    assign is_branch = (mem_input[15:12] == `BN && nf) ||
                               (mem_input[15:12] == `BZ && zf);

    // MEM 阶段操作逻辑
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            wb_input <= 16'b0000_0000_0000_0000;
            dst_regC2 <= 16'b0000_0000_0000_0000;
        end
        else if (cpu_state == `exec) begin
            wb_input <= mem_input; // 将输入指令传递到 WB 阶段
            dst_regC2 <= (mem_input[15:12] == `LOAD) ? datain : dst_regC1;
        end
    end

endmodule
```

**Pcpu.xdc：约束文件，用于仿真模拟**

```
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 start]

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 reset]
set_property IOSTANDARD LVCMOS33 [get_ports enable]
set_property IOSTANDARD LVCMOS33 [get_ports start]
set_property IOSTANDARD LVCMOS33 [get_ports clk_in]

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" }]
```

### stumbEggs.c：按照要求实现摔鸡蛋程序并计算两个时期的花费成本

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int totalFloors = 20; // 楼层数
    int fallResistance = 2; // 鸡蛋的耐摔值
    int result = 0; // 最终结果
    int count = 0; // 总次数
    int totalEggs = 0; // 摔碎的总鸡蛋数
    int isBroken = 0; // 最后一个鸡蛋是否破碎（1 破碎）

    int high = totalFloors, low = 1;
    int stumbFloor = 0;
```

```
    int totalUpFloors = 0; // 总上楼层数
    int totalDownFloors = 0; // 总下楼层数

    while (1) {
        count++;
        stumbFloor = (high + low) / 2;

        if (stumbFloor <= fallResistance) {
            isBroken = 0;
            totalUpFloors += (stumbFloor - low); // 记录上楼层数
            low = stumbFloor;
        }
        else {
            isBroken = 1;
            totalEggs++;
            totalDownFloors += (high - stumbFloor); // 记录下楼层数
            high = stumbFloor - 1;
        }

        if ((high - low) < 2) {
            break;
        }
    }

    count++;
    if (high <= fallResistance) {
        isBroken = 0;
        result = high;
        totalUpFloors += (high - low); // 最后一次的上楼层数
    }
    else {
        isBroken = 1;
        totalEggs++;
        result = low;
        totalDownFloors += (high - low); // 最后一次的下楼层数
    }

    // 成本计算
    int p1_materialScarcity=2,p2_materialScarcity=1, p3_materialScarcity = 4;
    int p1_laborCostIncrease=4, p2_laborCostIncrease=1,p3_laborCostIncrease
= 2;
    // 物质匮乏时期的总成本
    int cost_materialScarcity = totalUpFloors * p1_materialScarcity +
```

```
    totalDownFloors * p2_materialScarcity +totalEggs * p3_materialScarcity;

    // 人力成本增长时期的总成本
    int cost_laborCostIncrease = totalUpFloors * p1_laborCostIncrease +
    totalDownFloors * p2_laborCostIncrease +totalEggs * p3_laborCostIncrease;

    printf("Total drops: %d, Total broken eggs: %d, Last egg broken
status: %d\n", count, totalEggs, isBroken);
    printf("Cost in material scarcity period: %d\n", cost_materialScarcity);
    printf("Cost in labor cost increase period: %d\n",
cost_laborCostIncrease);

    return 0;
}
```