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：
姓名：
指导教师：陆有军

日期：2025.11.3

一、 实验环境部署与硬件配置说明

本实验运行在 windows 上，使用 vivado2016 进行开发与仿真。使用 Nexys DDR4 开发板进行下板实验。

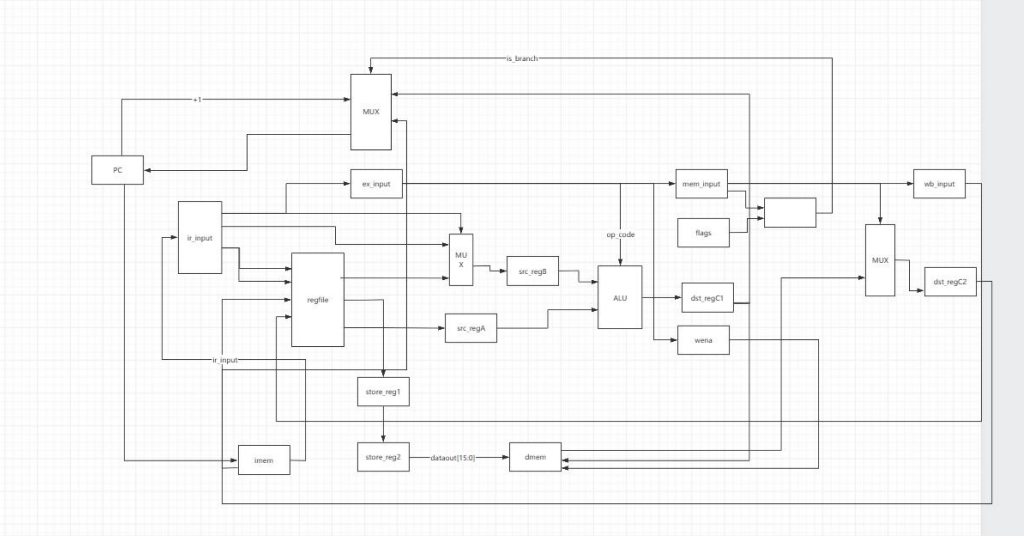
二、 实验的总体结构

1、 8 条指令流水线的总体结构

指令选取：本实验共选取常用指令 15 条，以完成指定的功能，依次为：NOP、HALT、ADD、ADDI、SUB、SUBI、CMP、LOAD、STORE、SHR、SHL、SHRI、SHLI、BZ、BN

指令码设计：指令有 15 条，采用四位操作码，后接 12 位作为立即数/寄存器。该 cpu 可执行 15 条指令，有 16 个通用寄存器。

种类	操作码	op1	op2	op3	操作描述（中文）
NOP	0	0	0	0	空操作，不执行任何指令
HALT	111	0	0	0	停止，终止程序执行
ADD	1	r1	r2	r3	将寄存器r2与r3的值相加，结果存入r1
ADDI	1000	r1	r1	val	将立即数val与寄存器r1相加，结果写入r1
SUB	1101	r1	r2	r3	将寄存器r2减去r3，结果写入r1
SUBI	1110	r1	r1	val	将立即数val从寄存器r1中减去，结果写入r1
CMP	10	0	r2	r3	比较寄存器r2与r3的值，设置标志位 cf,zf,nf
LOAD	11	r1	r2	val	从数据存储器地址(r2+val)读取数据，存入寄存器r1
STORE	100	r1	r2	val	将寄存器r1的数据写入数据存储器地址(r2+val)
SHR	1001	r1	r2	val	寄存器r2逻辑右移val位，结果写入r1
SHL	1010	r1	r2	val	寄存器r2逻辑左移val位，结果写入r1
SHRI	1011	r1	r1	val	寄存器r1逻辑右移val位，写回r1
SHLI	1100	r1	r1	val	寄存器r1逻辑左移val位，写回r1
BZ	101	r1	val1	val2	若零标志位zf=1，则跳转到地址(r1+{val1,val2})
BN	110	r1	val1	val2	若负标志位nf=1，则跳转到地址(r1+{val1,val2})



该流水线使用哈佛架构，每一条指令均经过取址（IF）、取值（ID）、计算（EX）、访存（MEM）、写回（WB）五个阶段。数据在数据通路中流动

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

流水线主要分为 CPU，imem，dmem 三部分，imem 与 dmem 通过数据总线和 cpu 之间进行交换数据的通信

2、 具体部件的作用

imem、dmem：

分别为指令存储器和数据存储器。其中 imem 为只读存储器（ROM），用于存放指令序列；dmem 为可读写的 RAM，在时钟上升沿进行写入，在组合逻辑中异步读取，实现数据访问。

PC（程序计数器）：

用于存放下一条要取的指令地址。IF 阶段根据 PC 从 imem 中读取指令，并在 BZ/BN 分支指令成立时更新 PC，实现跳转。

ALU（算术逻辑单元）：

位于 EX 阶段，用于执行所有算术与逻辑运算，包括加法、减法、逻辑移位、比较以及地址计算等。

ALU 输出同时用于设置标志位，如 z_flag（零标志）、n_flag（负标志）、c_flag（进位/借位标志）。

instr_ID、instr_EX、instr_MEM、instr_WB：

用于存放流水线四个阶段（ID、EX、MEM、WB）的指令。每一级设置独立寄存器以保证指令可以完整流水执行，并支持分支冲刷（flush）和 load-use 冒险的停顿（stall）。

RegFiles：

共 16 个通用寄存器（R0－R15）。由 WB 阶段在 RegWrite 为 1 时写回，其他阶段仅读取。

alu_op_A、alu_op_B：

源操作数寄存器，位于 EX 阶段。用于存放从 ID/EX 寄存器得到的两个操作数，并通过旁路（forwarding）单元进行更新，

从而解决数据相关导致的冒险。

mem_write_data_EX：

STORE 指令专用的数据寄存器。用于保证 STORE 写入的数据在流水化结构中不丢失，并支持 store-data 的前推（store forwarding），从而使 STORE 指令不会阻塞流水线。

wb_data_MEM、wb_data_final：

两个结果寄存器，用于 MEM→WB 阶段的数据传递。wb_data_MEM 存放 LOAD 的读取值或 ALU 的结果，wb_data_final 提供额外一拍的旁路支持，用于解决 WB 阶段数据的再次前推。

前推与冒险检测单元（ForwardA、ForwardB、Forward_Store、load_use_stall）：

ForwardA、ForwardB： 用于源操作数的旁路，解决 EX/MEM、MEM/WB、WB_final 三种情况的数据冒险。

Forward_Store： 专用于 STORE 指令写回数据的旁路。

load_use_stall： 检测 LOAD-USE 冒险，若下一条指令需要刚加载的数据，则插入一个气泡并暂停流水线一周期。

标志位寄存器（z_flag、n_flag、c_flag）：

用于保存最近一次 CMP/ADD/SUB 等运算后的标志位状态。其中：z_flag = 1 表示结果为 0，n_flag = 1 表示结果为负

c_flag 保存进位/借位状态

这些标志位用于 BZ（零跳转）和 BN（负跳转）等分支指令。

wena（数据存储器写使能）：

数据写信号寄存器。wena = 1 表示当前周期允许向 dmem 写入（即正在执行 STORE 指令）；wena = 0 表示禁止写入，确保 LOAD 与其他指令不会意外修改内存内容。

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

flag：nf，zf，cf 等等标志位寄存器

四、 实验仿真过程

1、 15 条指令流水线的仿真过程

本实验首先编写并测试完整的 C 程序，用于实现“比萨塔摔鸡蛋”模型的全部流程，包括区间二分查找、成本计算

与最终结果统计。随后将该 C 程序转换为仅由本 CPU 所实现的 15 条指令构成的汇编程序，并对所有控制流（循环、分支）、算术运算、LOAD/STORE 操作和标志位判断进行了等效改写，使其能够在自定义的流水线 CPU 架构上正确执行。

在汇编代码完成后，将每条指令按照自定义指令格式编码为 16 位机器码，并逐条进行校验，确保：立即数范围符合指令格式（例如 4 位 offset、8 位 imm），分支跳转目标地址正确，所有源寄存器、目的寄存器均符合流水线语义，没有使用 CPU 未支持的扩展指令。

修正后的机器码被写入 .coe 文件中，用作指令存储器（imem）的初始化内容。加载 .coe 文件后，Vivado 会在综合/仿真阶段自动将该文件写入 ROM，使得 CPU 在上电或复位后能够从地址 0 开始按序取指执行本实验程序。

在仿真过程中，CPU 按照 5 级流水线（IF → ID → EX → MEM → WB）依次执行程序，通过前推单元（forwarding）、LOAD-USE 冒险检测（stall）、STORE 数据旁路、分支冲刷流水等机制保证了指令之间的数据一致性。最终，经多次仿真验证，CPU 能够完整执行整个摔鸡蛋实验程序，最终在 R14、R15 中得到与 C 程序运行结果完全一致的数据，说明 15 条指令的流水化处理过程正确可靠。

五、 实验仿真的波形图及某时刻寄存器值的物理意义

15 条指令流水线的波形图及某时刻寄存器值的物理意义，下图为本实验设计的五级流水线 CPU 的仿真波形。波形中展示了 CPU 启动信号、取指地址、当前指令、寄存器堆内容以及若干关键内部信号。通过观察波形，可以验证各条指令在 IF → ID → EX → MEM → WB 五个阶段中依次推进，并能够正确进行冒险处理、旁路转发、LOAD-USE 暂停以及分支跳转冲刷。图中展示的主要信号包括：

start：CPU 的启动信号，在高电平时允许 CPU 从 PC=0 开始执行程序。

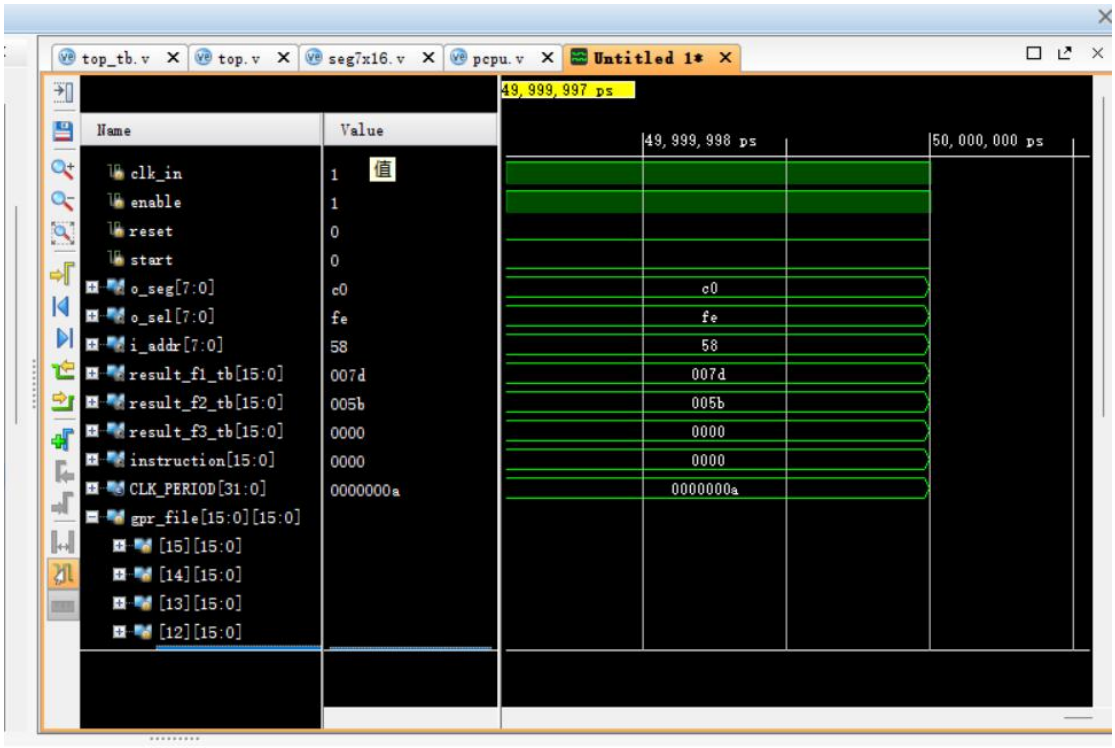
instruction：当前取出的指令（来自指令存储器 imem）

i_addr：当前取指地址（PC 值），可观察到程序执行过程中跳转指令导致的 PC 跳变。

general_reg[15:0]：通用寄存器堆的内容，包括 r0－r15。其中 r14、r15 分别用于输出实验结果（最终耐摔层数、总成本等）。

result_f1、result_f2、result_f3：分别监测 R2、R14 与 ALU 输出的调试端口，可用于验证关键步骤计算是否正确。

通过完整仿真，CPU 能够正确取指、译码、执行、访存、写回，并顺利跑通摔鸡蛋算法程序，实现 C 程序的等价执行



以下寄存器和信号为本 CPU 运行过程中关键的内部状态，其物理意义如下：

① cpu_state、next_cpu_state

用于表示 CPU 当前状态与下一拍的状态。

cpu_state 处于 CPU_EXEC 时，流水线正常运行。

当遇到 HALT 指令或外部禁止信号 enable=0 时转入 CPU_IDLE。

② instr_ID、instr_EX、instr_MEM、instr_WB

对应 ID、EX、MEM、WB 四级流水线寄存器，用于存放目前在该阶段执行的指令，保证每条指令在五级流水线中按顺序推进。

这些寄存器支持：

分支冲刷（flush）

load-use 冒险导致的暂停（stall）

指令自然向后推进（流动）

③ reg_A_read_ID、reg_B_read_ID（源操作数寄存器）

用于在 ID 阶段从通用寄存器堆中读取操作数。

reg_A_read_ID：源寄存器 Rs1

reg_B_read_ID：源寄存器 Rs2

两者的内容可能被旁路单元修改以解决数据相关冒险。

④ mem_write_data_EX（STORE 写数据寄存器）

用于 STORE 指令将要写入 dmem 的数据，它在 EX 阶段确定，并在 MEM 阶段输出到数据存储器 dataout。

此寄存器支持 store-data forwarding，保证连续 STORE 指令也能流水执行而不阻塞。

⑤ wb_data_MEM、wb_data_final（目标寄存器写回）

wb_data_MEM：LOAD 的 data_in 或 ALU 的结果，用于在 WB 阶段写回寄存器堆。

wb_data_final：额外一拍的数据缓存，用于 WB → EX 的旁路，

解决三周期前的指令结果读写冲突。

两者共同保证流水线在高并发下仍能正确执行算术和访存指令。

⑥ general_reg[15:0]（通用寄存器堆）

共 16 个 16 位寄存器，用户程序核心变量的全部存储位置，包括：

R1：当前楼层 low

R2：当前楼层 high

R3：floor_to_test

R4：last_floor

R5：total_up_floors

R6：total_down_floors

R7：broken_eggs

R8：total_throws

R14、R15：最终实验结果

仿真后可以看到 R14、R15 的值与 C 程序输出完全一致。

⑦ z_flag、n_flag、c_flag（标志位寄存器）

用于保存最近一次 ALU 运算的结果：

z_flag=1：结果为 0

n_flag=1：结果为负值

c_flag=1：产生进位或借位

在 BZ（zero）与 BN（negative）跳转指令中使用这些标志位进行分支判断。

⑧ alu_out_EX、alu_out_MEM（ALU 结果）

alu_out_EX：EX 阶段 ALU 运算结果

alu_out_MEM：用于 MEM 阶段的地址或中间值

这两个信号便于观察每条指令的计算过程，尤其是 LOAD/STORE 地址计算与二分查找中间变量的变化。

⑨ dataout、datain（访存操作）

dataout：STORE 指令写入 dmem 的内容

datain：LOAD 指令从 dmem 读出的数据

在仿真中可以看到 LOAD 读出的值随程序推进不断更新，与 C 程序中模拟摔蛋逻辑一致。

⑩ d_addr（数据存储器地址）

由 EX 阶段计算得到，用于对 dmem 的地址访问。

在程序运行过程中可以看到 d_addr 随不同指令变化，证明取/存数据过程正常流水化。

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、 编写 C 语言程序， 采用二分的方法进行耐摔值的查找，

```c
#include <stdio.h>
#include <stdlib.h> // 包含 standard library，便于在所有环境中编译

int main() {

    // --- 实验常量定义 ---
    const int N_FLOORS = 64;    // 总楼层数
    const int F_TRUE = 38;       // 鸡蛋真正的耐摔值

    // --- 成本系数定义 ---
    // 物质匮乏时期 (p1=2, p2=1, p3=4)
    const int P1_SCARCITY = 2;
    const int P2_SCARCITY = 1;
    const int P3_SCARCITY = 4;

    // 人力成本增长时期 (p1=4, p2=1, p3=2)
    const int P1_LABOR    = 4;
    const int P2_LABOR    = 1;
    const int P3_LABOR    = 2;

    // --- 实验指标变量 (对应 CPU 寄存器或内存中的 m, n, h, count) ---
    int total_up_floors = 0;   // m (上的楼层总数)
    int total_down_floors = 0; // n (下的楼层总数)
    int broken_eggs = 0;       // h (摔破的鸡蛋总数)
    int total_throws = 0;      // count (摔的总次数)

    // --- 算法变量 ---
    int high = N_FLOORS;     // 搜索区间上界
    int low = 1;             // 搜索区间下界
    int last_floor = N_FLOORS; // 初始站在最高层 (128)
    int floor_to_test = 0;   // 当前试探楼层
    int result_F = 0;        // 最终找到的耐摔值

    // ------------------------------------------------
    // Phase 1: 二分查找循环 (直到 high - low < 2)
    // ------------------------------------------------
    while ((high - low) >= 2) {
        total_throws++;

        // 核心计算: floor_to_test = (high + low) / 2
        floor_to_test = (high + low) / 2;

        // 成本计算 m 和 n (无 abs)
        int diff = floor_to_test - last_floor; // 计算楼层差值

        if (diff > 0) {
            // diff > 0 (上楼): 累加 m
            total_up_floors += diff;
        } else {
            // diff <= 0 (下楼或原地): 累加 n
            int abs_diff = 0 - diff; // 0 - diff 得到正的绝对值
            total_down_floors += abs_diff;
        }
        last_floor = floor_to_test; // 更新上次楼层
```

```
55
56          // 模拟摔蛋 (test_egg: floor_to_test > F_TRUE)
57          if (floor_to_test > F_TRUE) {
58              broken_eggs++;
59              high = floor_to_test - 1;
60          } else {
61              low = floor_to_test;
62          }
63      }
64
65      // ------------------------------------------------
66      // Phase 2: 最终确认 (处理 high - low = 0 或 1 的情况)
67      // ------------------------------------------------
68      if (high == low) {
69          result_F = high;
70      }
71      else { // high = low + 1, 需试摔 high 楼层
72          total_throws++;
73
74          // 成本计算 (测试 high 楼层)
75          int diff = high - last_floor;
76          if (diff > 0) {
77              total_up_floors += diff;
78          } else {
79              int abs_diff = 0 - diff;
80              total_down_floors += abs_diff;
81          }
82
83          // 最终试摔 high 楼层
84          if (high > F_TRUE) {
85              broken_eggs++;
```

test.cpp

```
91
92      // ------------------------------------------------
93      // 结果输出与成本计算
94      // ------------------------------------------------
95      printf("--- 比萨塔摔鸡蛋模型验证 ---\n");
96      printf("总楼层 N: %d, 真实耐摔值 F_true: %d\n", N_FLOORS, F_TRUE);
97      printf("找到的耐摔楼层 F: %d\n", result_F);
98
99      printf("\n--- 实验指标 ---\n");
100     printf("摔的总次数: %d\n", total_throws);
101     printf("摔破的鸡蛋总数 h: %d\n", broken_eggs);
102     printf("上的楼层总数 m: %d\n", total_up_floors);
103     printf("下的楼层总数 n: %d\n", total_down_floors);
104
105     // 成本计算 f = m*p1 + n*p2 + h*p3
106     long m_val = total_up_floors;
107     long n_val = total_down_floors;
108     long h_val = broken_eggs;
109
110     // 成本分析 - 物质匮乏时期
111     long cost_scarcity = m_val * P1_SCARCITY + n_val * P2_SCARCITY + h_val * P3_S
112     printf("\n--- 成本分析 (f = m*p1 + n*p2 + h*p3) ---\n");
113     printf("物质匮乏时期 (p1=2, p2=1, p3=4) 总成本: %ld\n", cost_scarcity);
114
115     // 成本分析 - 人力成本增长时期
116     long cost_labor = m_val * P1_LABOR + n_val * P2_LABOR + h_val * P3_LABOR;
117     printf("人力成本增长时期 (p1=4, p2=1, p3=2) 总成本: %ld\n", cost_labor);
118
119     return 0;
120 }
```

2、 编写对应的指令序列（汇编语言）

手动编写汇编指令，用于之后转换成机器码（部分截图如下）：

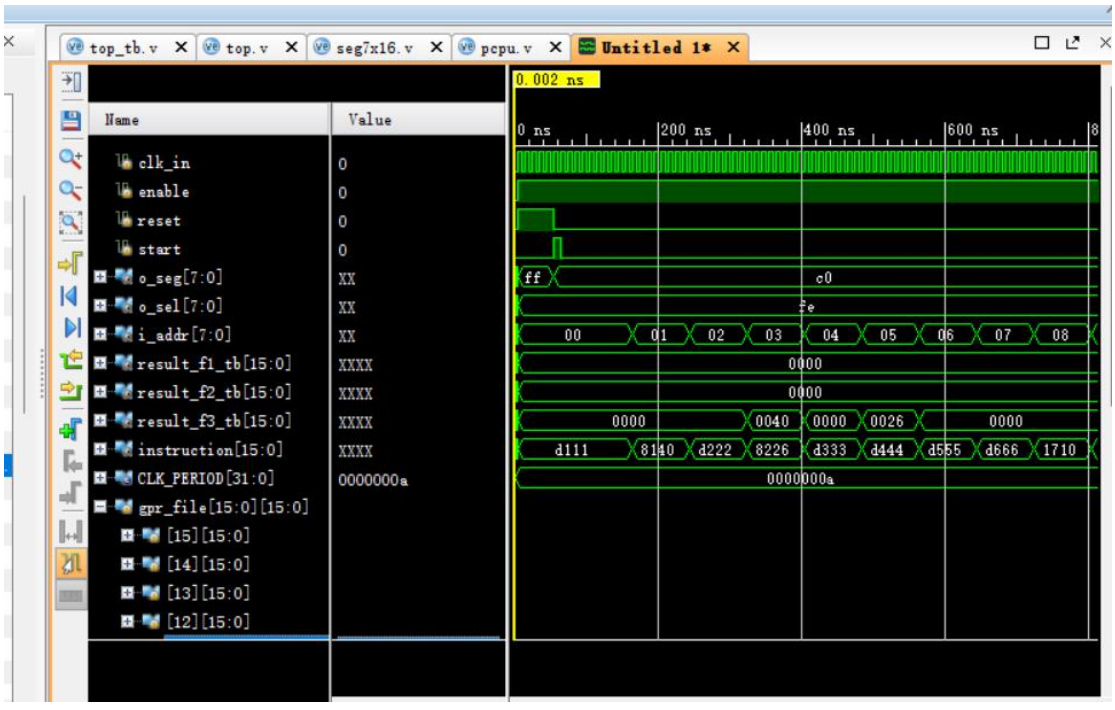
```
SUB  R1, R1, R1
ADDI R1, 0x40       ; N_FLOORS = 64

SUB  R2, R2, R2
ADDI R2, 0x26       ; F_TRUE = 38

SUB  R3, R3, R3     ; total_up = 0
SUB  R4, R4, R4     ; total_down = 0
SUB  R5, R5, R5     ; broken_eggs = 0
SUB  R6, R6, R6     ; total_throws = 0

ADD  R7, R1, R0     ; high = 64
SUB  R8, R8, R8
ADDI R8, 0x1        ; low = 1

ADD  R9, R1, R0     ; last_floor = N_FLOORS

SUB  R12, R12, R12 ; result_F = 0
```

3、 将指令序列转化为机器码

编写了一个 python 程序将汇编代码转换为机器码，部分截图如下：

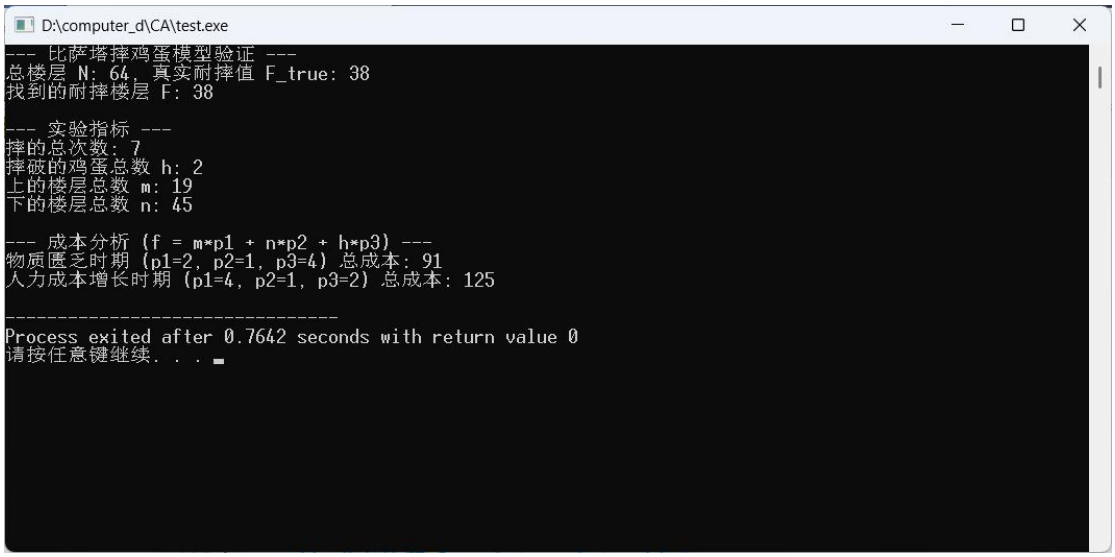
```python
# --- 4. Branch Type (BZ, BN) ---
elif instruction in ['BN', 'BZ']:
    if len(operands) != 1:
        raise ValueError(f"分支指令 {instruction} 格式应为 BZ LABEL")

    target_label = operands[0]
    target_addr = labels.get(target_label)

    if target_addr is None:
        raise ValueError(f"未找到标签地址: {target_label}")

    # ⭐ 核心修正: 适配 Verilog 中的 Reg + Imm 寻址
    # 策略: 使用 R0 ('0000') 作为基址寄存器 Rs (instr[11:8]),
    # 并将目标地址 Target_Addr 编码为 8 位立即数 Imm (instr[7:0])。
    # 假设 R0=0, Verilog计算 Target = Reg[R0] + Imm。
```

4、 将 coe 文件导入，并进行仿真

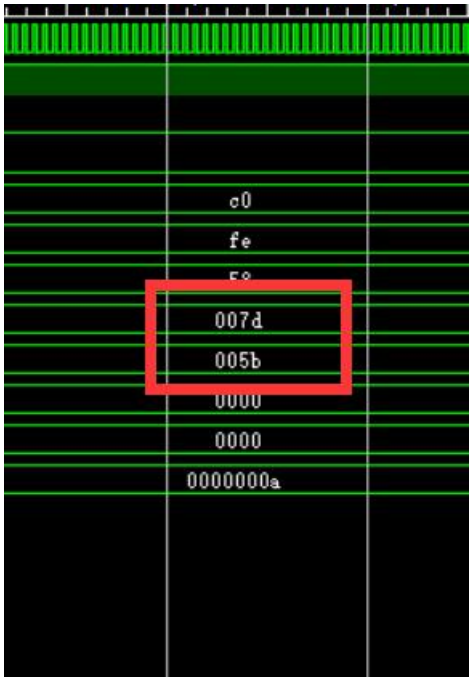


5、结果比较

C 语言程序结果：



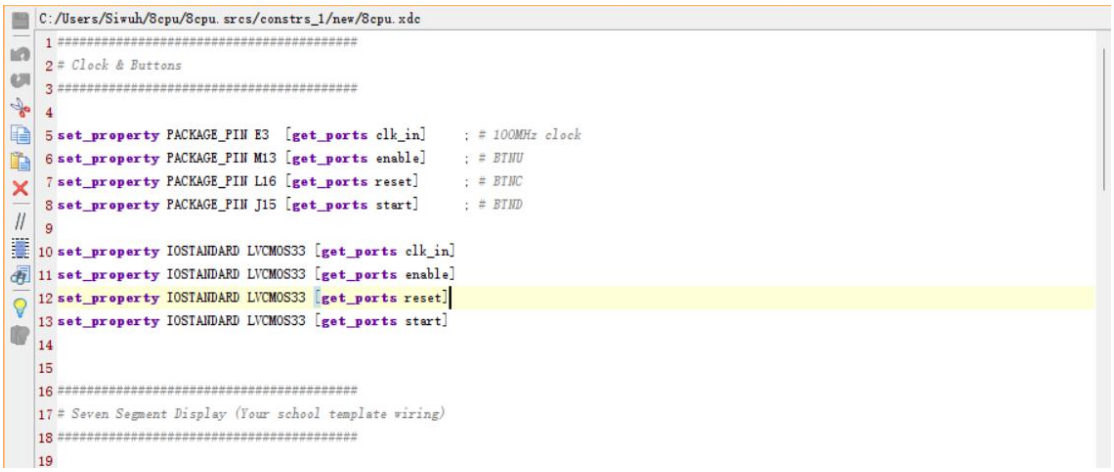
流水线 CPU 计算结果：



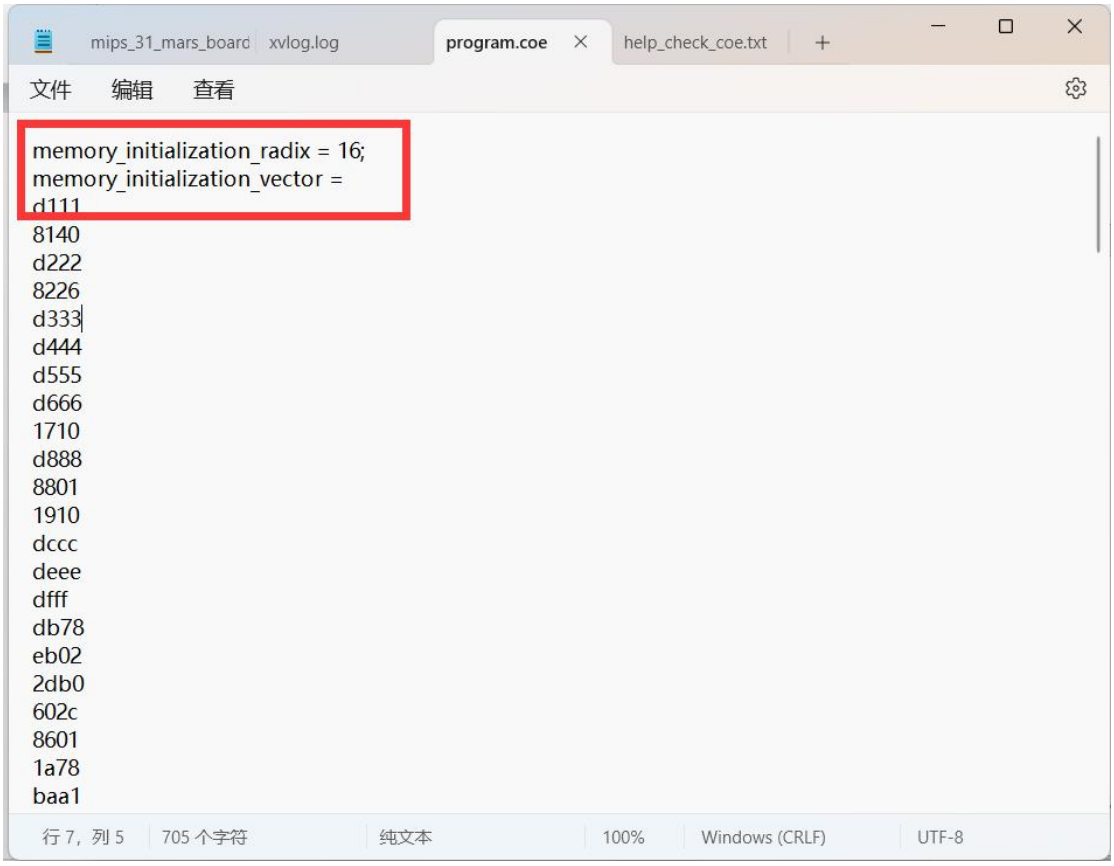
结果相同，说明 c 语言程序和机器码都正确

七、 实验验算程序下板测试过程与实现

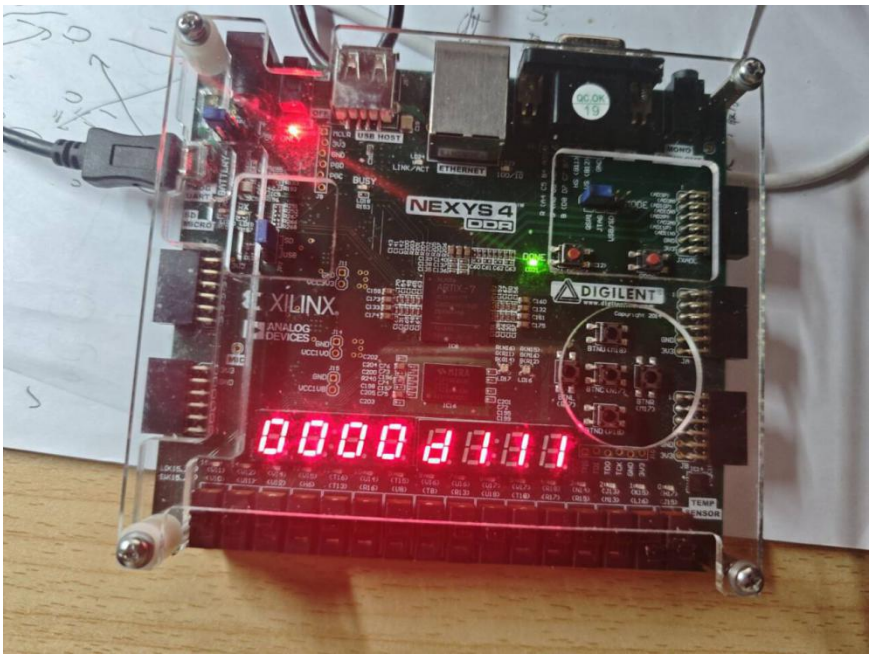
1.配置 XDC 文件，部分截图如下



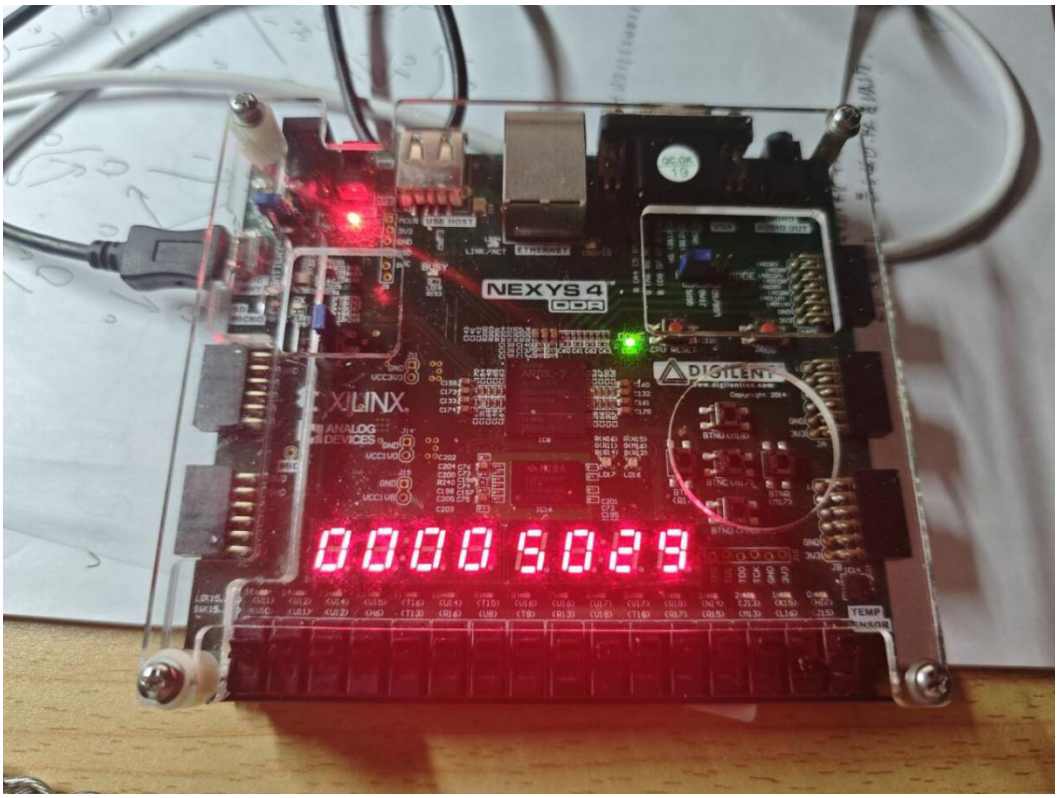
2. 在板子上显示出指令



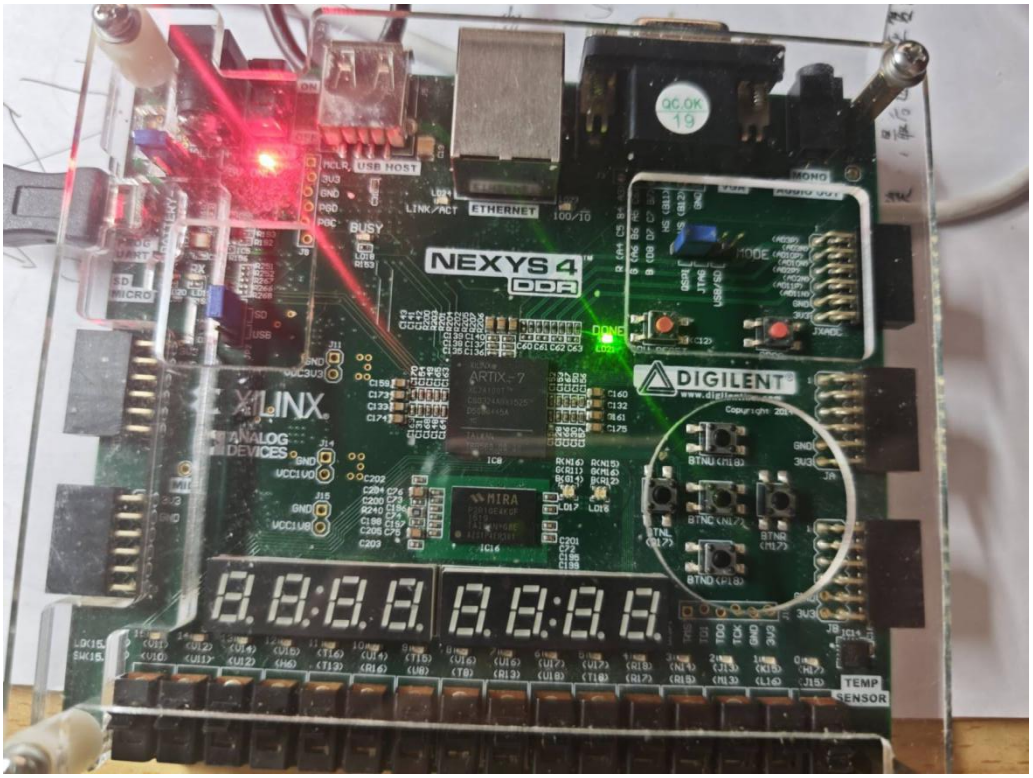
3. 成功显示第一条指令 d111



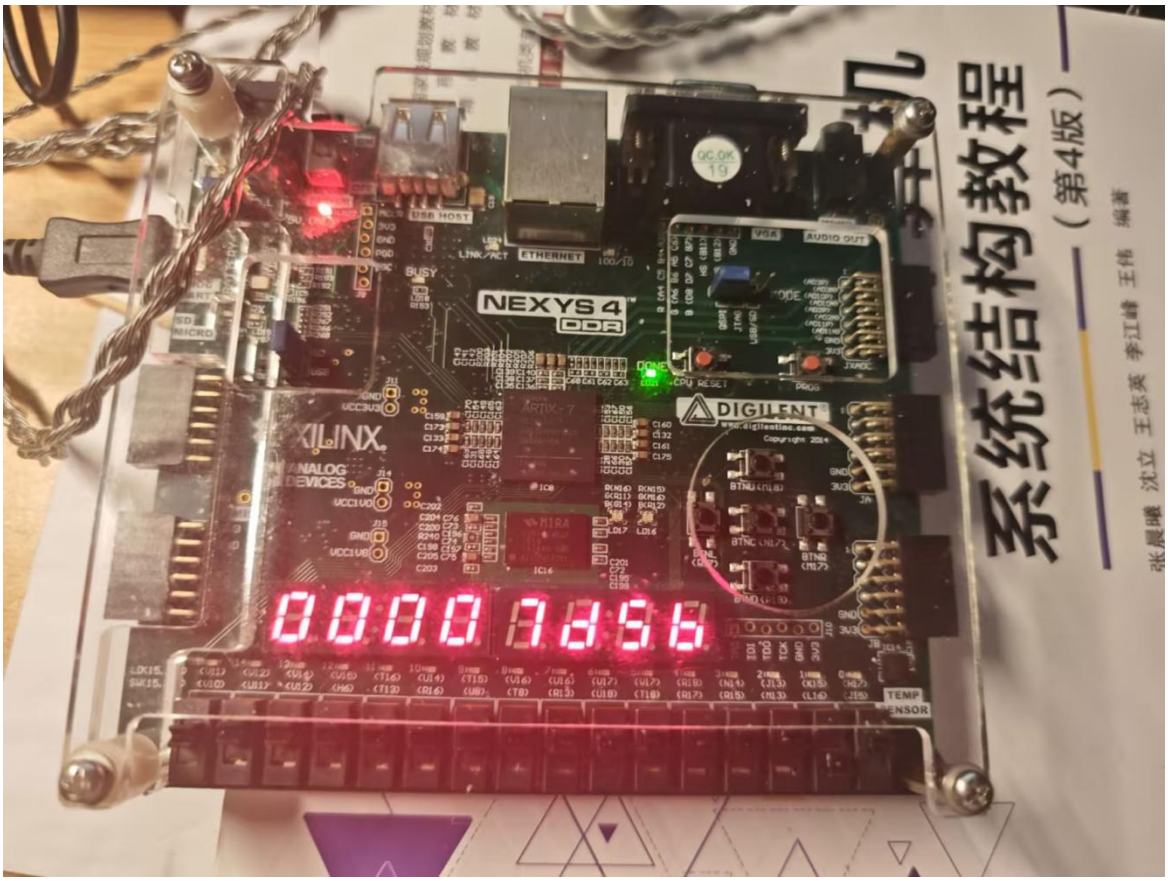
4. 后面程序正常流水：



5. 关闭 enable，能够正常停下：



更改 top.v 之后，可以将结果也显示在开发板上（与前面一致）：



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、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

时序报告显示，运行延时约为 90ns，本次程序在 N=64、F_true=38 的条件下运行，共经历初始化阶段、6 次 Phase 1 二分查找循环、Phase 2 楼层确认和 Phase 3 成本计算。

经过逐条汇编级计数，本程序的总指令执行次数为 168 条。

其中，Phase 1 的 6 次循环合计执行了 120 条指令，是整个程序耗时最多的部分。跳转造成的流水线冲刷数量也集中在 Phase 1 的循环判断与楼层判断中。我们后续的分析使用该数据来计算

吞吐率：1/t≈11000000

在当前设计中，CPU 采用 5 级静态流水线（IF－ID－EX－MEM－WB），并实现了：寄存器前推（EX/MEM、MEM/WB 两级结果旁路，针对 LOAD 指令的 load-use 冒险暂停机制，在 MEM 阶段完成分支判断，被采纳分支会冲刷水线中的后续指令

本程序的最终版本中没有再使用 LOAD / STORE 指令，所有数据都保存在寄存器中，因此不会产生 load-use 冒险，流水线停顿的主要来源为：

流水线填充与排空开销（大约 4 个时钟周期）。分支指令被采纳时产生的冲刷水泡。在当前这段汇编中：Phase 1 的 while 循环、Phase 2 的 if/else、以及若干用 CMP R0, R0, R0 + BZ label 实现的“伪 JUMP”都会触发分支跳转

结合程序实际执行路径，可以估计：被采纳的分支指令数量约为 8 条左右，设计中分支在 MEM 阶段判断，每次被采纳的分支会冲刷 IF，ID，EX 三级，等效为每条被采纳分支额外损失约 3 个时钟周期

据此可以得到一个较为合理的估算公式：

流水线总周期 ≈ 指令条数 + 填充气泡 + 分支冲刷水泡 ≈ 168（指令） + 4（填充） + 8 × 3（分支冲刷）≈ 168 + 4 + 24 = 196 个时钟周期

也就是说，在当前程序和输入数据下，五级流水线 CPU 实际大约在 196 个时钟周期内完成全部 168 条指令的执行。

由此计算得出，我设计的流水线的吞吐率：

168/196=0.86%

在不考虑流水线的理想顺序实现下，可以近似认为每条指令需要经过 “取指 / 译码 / 执行 / 访存 / 写回” 五个阶段，

一个阶段一个时钟周期，则非流水线 CPU 的总周期数约为：流水线总周期 ≈ 168 条指令 × 5 周期 / 条 ≈ 840 个时钟周期

加速比 S ≈ 非流水线总周期 / 流水线总周期≈ 840 / 196 ≈ 4.28 倍

流水线效率可以近似定义为：Efficiency ≈ 实际加速比 / 理论最大加速比，理论最大加速比 ≈ 流水线级数 = 5

Efficiency ≈ 4.47 / 5 ≈ 0.89

也就是说，在这次实验程序上，五级流水线的利用率接近 90%，

数据相关方面

通过 EX/MEM、MEM/WB 的结果旁路，解决了绝大多数寄存器读后写相关（RAW）问题，例如连续的 ADD、SUB、ADDI 之间可以在下一拍直接使用上一条指令的结果，而无需停顿。

控制相关（分支相关）

所有条件分支在 MEM 阶段才完成判断，因此被采纳的分支需要冲刷后续三级流水线，等效为每次分支损失 3 个周期。

在本程序中，while 循环和后续 if/else 的结构比较规则，总共大约只有 8 次左右的“被采纳分支”，对整体 CPI 的影响有限，使得平均 CPI 仍维持在约 1.12 的较低水平。

· **结构相关**

指令存储器（imem）和数据存储器（dmem）物理上分离，指令取值与数据读写在硬件上没有结构冲突，避免了典型的“单端口存储器”资源竞争问题。16 个通用寄存器和多级流水线寄存器共同构成了良好的数据通路，在本次测试程序中没有出现结构性瓶颈。

九、 总结与体会

本次流水线 CPU 的设计与实验，使我对处理器结构、流水线机制以及硬件数据通路的构建流程有了系统而深入的理解。在实验最初，我原本计划仅实现 8 条基本指令，以完成最简单的逻辑验证。但在将真实的 C 程序（摔鸡蛋模型）转换为汇编后，我很快发现 8 条指令难以满足实际需求，无法表达必要的条件判断、移位计算和内存访存等操作。最终，我将指令数量扩展到 15 条。随着指令集的丰富，CPU 才具备了执行完整算法的能力，这也让我意识到：指令集的设计不是随意增加，而是由程序需求与硬件复杂度共同决定的工程权衡。

在流水线设计阶段，我深刻体会到实际硬件设计的严谨性。流水线在逻辑上似乎简单，但在实际实现时，每一级的寄存器推进、控制信号传递、操作数选择都必须严格保持一致，否则仅仅“取错一拍”就会导致结果彻底错误。例如，本次实验中多次出现由于前推信号或 STORE 数据一拍延迟而导致的计算偏差。每一次错误都迫使我重新检查流水线数据依赖、前推路径和冒险处理逻辑，也让我真正理解到：流水线 CPU 的本质，是对时间和数据流的精确调度，任何一个细节偏差都会导致全局崩溃。

通过不断调试 Forwarding、Load-Use Stall 和分支冲刷，我逐渐构建了稳定可靠的五级静态流水线，实现了大部分指令能做到“一周期一条”的吞吐率。特别是在分支跳转的处理上，如何正确 flush 掉 ID、EX、MEM 级错误的指令，是确保程序正确性的关键，这让我对真实 CPU 技术有了更深层的体会。

通过将 C 程序手动转为汇编，再编码为机器码写入 COE 文件，我首次完整经历了从高级语言逻辑到 FPGA 硬件执行的全流程。下板测试中，七段数码管成功显示指令地址和取指内容，CPU 能够正确执行整个摔鸡蛋算法，并输出与 C 程序一致的结果。这让我体会到把“纸上架构”变成“真实运行硬件”的成就感。

总体而言，这次实验提升了我对流水线 CPU 设计方法的理解。许多错误并非源自功能，而是源自节拍、冒险、依赖等隐藏

问题。通过不断调试和优化，我增强了对数字系统设计的把握，也为进一步学习现代计算机体系结构奠定了坚实基础。

十、 附件（所有程序）

文件夹中，

top.v,pcpu.v,dmem.v,imem.v,seg7x16.v 为 vivado 工程用到的源代码，

top_tb.v 是用于测试代码的 tb 文件,

Test.c 是用于计算摔鸡蛋模型的 C 语言程序

汇编.txt 是对 C 语言程序手动汇编得到的汇编代码

Translate.py 是将汇编代码翻译为机器码的编译程序

program.coe 是由上述编译器得到的机器码（用于放进 ip 核）