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：2253214
姓名：李闯
指导教师：陆有军

日期：2024 年 10 月 29 日星期二

一、实验环境部署与硬件配置说明

实验环境部署：本实验在 Windows 平台上进行，使用 Vivado 作为开发工具，ModelSim 用于功能仿真和调试。

硬件配置说明：实验使用学校统一发放的 NEXYS 4 DDR 开发板。

二、实验的总体结构

1. 指令码设计

在本次实验中，我选择了 16 条指令，以验证 CPU 性能并实现相关功能。这些指令包括：NOP（无操作）、HALT（停止）、LOAD（加载）、STORE（存储）、ADD（加法）、ADDI（立即加法）、SUB（减法）、SUBI（立即减法）、SLL（逻辑左移）、SRL（逻辑右移）、CMP（比较）、BZ（零分支）、BNZ（非零分支）、BN（负分支）、BNN（非负分支）和 JUMP（立即跳转）。每条指令的操作码部分长度为 4 位，这样的设计确保了指令集的简洁性和有效性。

我的设计初衷是为了简化指令集，使得指令尽可能短小，以提高指令执行的效率。同时，CPU 内部操作数的长度设定为 16 位，这不仅增强了数据处理的精度，也提高了对较大数值的处理能力。此外，我为 CPU 配置了 16 个通用寄存器，以满足指令执行过程中的存储需求，确保有足够的灵活性来支持多种计算任务。

数据存储器（DMEM）的容量设定为 256 字节，这样能够有效地存储和管理程序运行过程中所需的数据，符合实际需求。

整个指令码的设计如图 1 所示，直观地展示了各指令的操作码及其对应功能，为后续的 CPU 性能验证奠定了基础。这一系列设计不仅关注了指令集的完整性和多样性，还旨在满足性能验证程序的执行需求，确保设计的合理性和实用性。

	指令	指令操作码-4 位	操作数 1-4 位	操作数 2-4 位	操作数 3-4 位	操作简介
1	NOP	0000	0000	0000	0000	空指令
2	HALT	1111	1111	1111	1111	停机指令
3	LOAD	1101	r1	r2	val3	[r1]<-dmem([r2]+val3)
4	STORE	1110	r1	r2	val3	dmem([r2]+val3)<-[r1]
5	ADD	0001	r1	r2	r3	[r1]<-[r2]+[r3]
6	ADDI	0010	r1	val2	val3	[r1]<-[r1]+{val2,val3}
7	SUB	0011	r1	r2	r3	[r1]<-[r2]-[r3]
8	SUBI	0100	r1	val2	val3	[r1]<-[r1]-{val2,val3}
9	SLL	0101	r1	r2	val3	[r1]<-[r2]<<{val3}
10	SRL	0110	r1	r2	val3	[r1]<-[r2]>>{val3}
11	CMP	0111	0000	r2	r3	r2-r3,设置标志位 NF,ZF
12	BZ	1000	r1	val2	val3	if ZF=1 跳转到 r1+{val2,val3}
13	BNZ	1001	r1	val2	val3	if ZF=0 跳转到 r1+{val2,val3}
14	BN	1010	r1	val2	val3	if NF=1 跳转到 r1+{val2,val3}
15	BNN	1011	r1	val2	val3	if NF=0 跳转到 r1+{val2,val3}
16	JUMP	1100	0000	val2	val3	跳转到{val2,val3}

图 1 指令码设计

2. 静态流水线总体结构

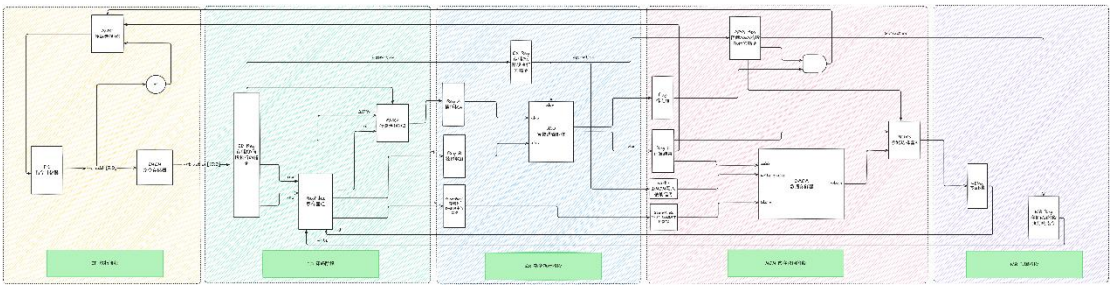


图 2 静态流水线总体结构

三、总体架构部件的解释说明

该静态流水线的上层部分主要由三个组成部分构成：简单流水线 CPU、数据存储器（DMEM）和指令存储器（IMEM）。其 CPU 部分操作分为五个阶段，分别是：IF（取指令）、ID（指令解析）、EX（执行指令）、MEM（访问数据存储器）、WB（写回阶段）。这五个阶段协同工作，确保指令能够有效地逐步处理。

在静态流水线 CPU 的内部部件中，各部分功能如下：

1. **ID_IReg[15:0]、EX_IReg[15:0]、MEM_IReg[15:0]、WB_IReg[15:0]**：这些寄存器分别用于存储各个阶段正在执行的指令码，确保各部件能够根据当前指令的需要执行相应的操作，从而实现流水线的高效运作。
2. **[15:0] RegFiles[0:15]**：这是一个 16x16 位的通用寄存器组，用于存储 CPU 在计算过程中所需的数据。寄存器组的设计为 CPU 提供了快速访问的存储空间，能够支持多种运算需求。
3. **RegA、RegB**：这两个寄存器用于存储算术逻辑单元（ALU）的两个操作数，确保 ALU 在执行运算时能够快速获得所需的输入数据。
4. **RegC1**：此寄存器用于存储 ALU 运算的结果，确保后续的操作可以使用该结果进行进一步处理。
5. **RegC2**：此寄存器专门用于在 WB 阶段存储将要写入寄存器组的值，从而实现数据的正确回写。
6. **StoreReg1、StoreReg2**：这两个寄存器用于存储在执行 STORE 指令后，需写入到数据存储器（DMEM）中的数据，确保数据能够被有效保存。
7. **PCReg[7:0]**：指令计数器，用于存储下一条指令的地址。这一部分对于控制程序的执行流程至关重要。
8. **zf、nf、cf**：这些是运算结果的状态标志位，分别表示零标志、负标志和进位标志。它们的状态用于指导后续指令的执行，特别是在条件分支时。
9. **dw**：这是 DMEM 写入使能信号，控制数据何时被写入数据存储器，确保数据的正确存储与管理。

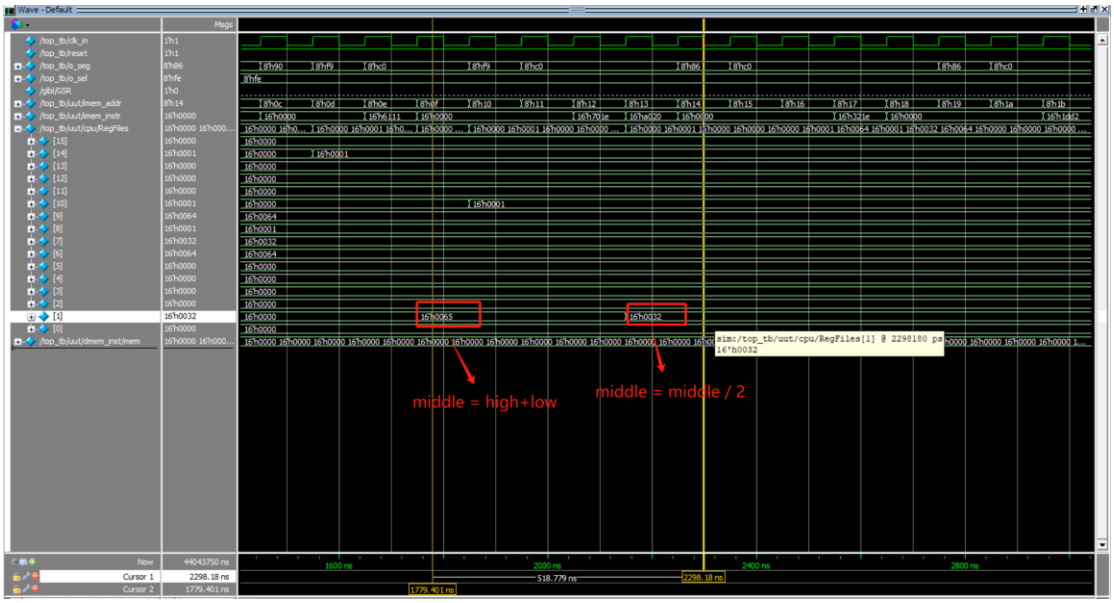
这一系列设计不仅确保了指令的顺利执行，也为 CPU 的整体性能提升奠定了基础。通过合理的结构布局和信号管理，静态流水线能够有效地提升指令处理的速度和效率。

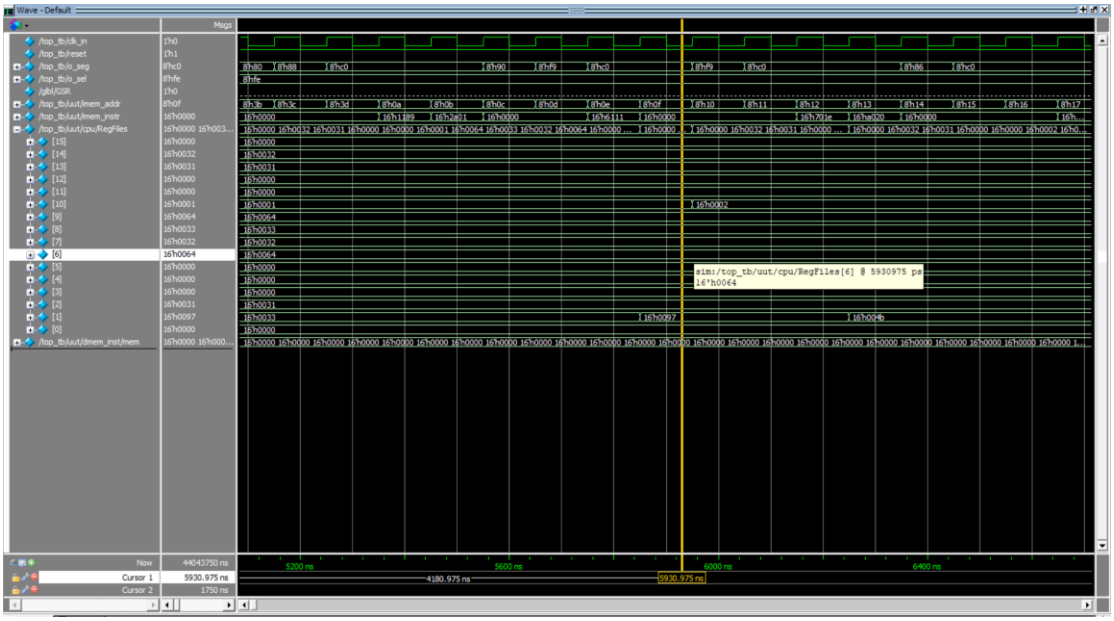
四、实验仿真过程

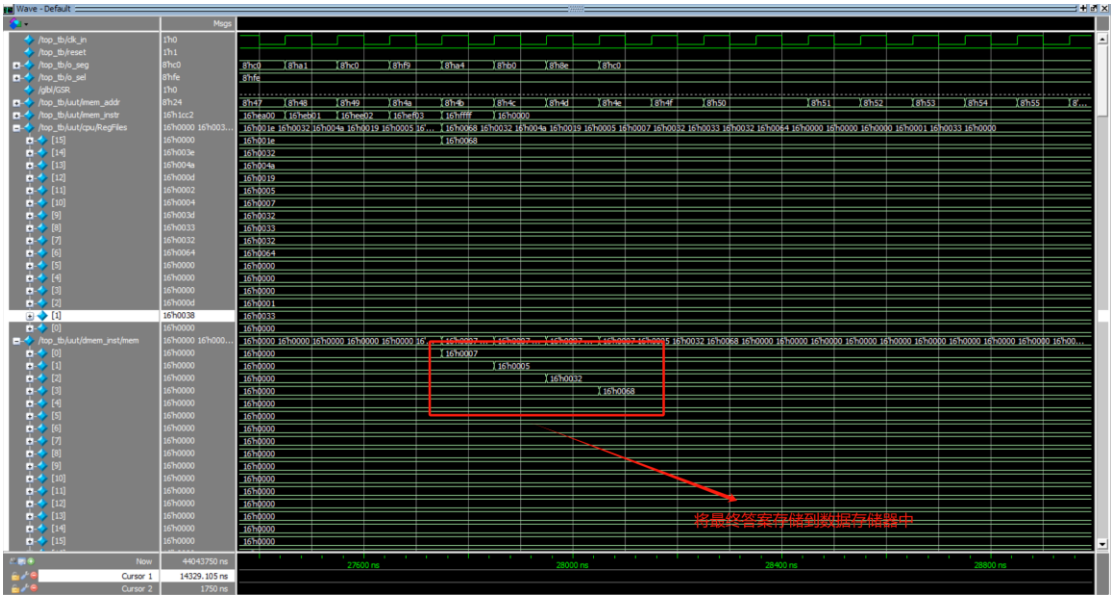
1. **编写 C 语言测试程序：**首先，需编写一个用于仿真测试的 C 语言程序。该程序应涵盖多种操作和数据处理，以确保对设计的静态流水线 CPU 的全面测试。程序应设计得尽可能简单明了，方便后续的手动汇编和测试。
2. **手动汇编为汇编语言代码：**接下来，将上述 C 语言测试程序按照预先设定好的指令集手动汇编成汇编语言代码。在此过程中，需要特别注意指令的顺序安排，以避免流水线中的数据冲突和资源竞争。同时，适当增加 NOP（无操作）指令，作为填充指令，以解决流水线运行中的资源相关问题，确保每个指令在其对应的阶段能够顺利执行。
3. **转化为二进制机器代码：**随后，按照指令集的规定，将调整好的汇编语言代码手动转化为二进制机器代码。这一步骤需确保每条指令的二进制表示准确无误，以便在后续使用。最终，将这些二进制代码形成 coe 文件，并使用该文件来初始化指令存储器（IMEM），为后续的仿真测试提供必要的指令集。
4. **编写测试基准文件进行功能仿真测试：**最后，通过编写相应的测试基准（testbench）文件进行功能仿真测试。这一文件将调用 IMEM 中的指令，并监控 CPU 的执行过程。通过观察流水线 CPU 的运行情况，确认其是否按照预先设想的功能和性能要求进行操作，从而验证设计的正确性和有效性。

通过以上步骤，可以系统地测试和验证静态流水线 CPU 的设计，确保其在实际运行中的表现与预期一致。

五、实验仿真的波形图及某时刻寄存器值的物理意义


middle = high+low
middle = middle / 2




将最终答案存储到数据存储器中

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. C 语言程序

采用二分查找的方式确定最终鸡蛋的耐摔值，同时在查找过程中不断更新摔碎的鸡蛋数、代价值等其他重要的参数量，该算法情况最差的时间复杂度 O(logn)

```c
/*
    2024年秋季学期计算机系统结构课程流水线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。

    使用该模型计算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
*/

#include <stdio.h>

// 物质匮乏时期：p1=2，p2=1，p3=4
#define COST_UP_LOW 2
#define COST_DOWN_LOW 1
#define COST_BROKEN_LOW 4

// 人力成本增长时期：p1=4，p2=1，p3=2
#define COST_UP_HIGH 4
#define COST_DOWN_HIGH 1
#define COST_BROKEN_HIGH 2

void solve(int floors, int critical_floor, int cost_up, int cost_down, int cost_broken)
{
    // 采用二分的方式进行问题解决
    int low = 1, high = floors;
    int egg_drops = 0, eggs_broken = 0; // 记录扔鸡蛋的数量 以及 摔碎鸡蛋的数量
    int last_floor = 1;                 // 记录当前所处的位置
    int total_cost = 0;                 // 记录总代价

    while (low <= high)
    {
        int mid = (low + high) / 2;
        egg_drops++;

        // 计算从当前楼层到达mid楼层的代价
        if (mid > last_floor)
        {
            total_cost += (mid - last_floor) * cost_up; // 上楼代价
        }
        else if (mid < last_floor)
        {
            total_cost += (last_floor - mid) * cost_down; // 下楼代价
        }

        if (mid > critical_floor)
        {
            eggs_broken++;
            total_cost += cost_broken;
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }

        last_floor = mid; // 更新当前楼层
    }

    printf("检查的总楼层数: %d\n", egg_drops);
    printf("摔碎的鸡蛋数量: %d\n", eggs_broken);
    printf("最终投掷的楼层: %d\n", last_floor);
    printf("总代价: %d\n", total_cost);
}

int main()
{
    int floors = 100;        // 假设有100层楼
    int critical_floor = 50; // 假设临界楼层为50

    printf("物质匮乏时期的成本评估:\n");
    solve(floors, critical_floor, COST_UP_LOW, COST_DOWN_LOW, COST_BROKEN_LOW);

    printf("\n人力成本增长时期的成本评估:\n");
    solve(floors, critical_floor, COST_UP_HIGH, COST_DOWN_HIGH, COST_BROKEN_HIGH);

    return 0;
}
```

图 3 流水线 CPU 实验性能验证模型 C 语言程序

2. 汇编语言程序

```
00 ADDI r6 6 4  // 楼层数 -- 初始化为100=6*16+4  -> 0010 0110 0110 0100
01 ADDI r7 3 2   // 临界楼层 -- 初始化为50  -> 0010 0111 0011 0010
02 ADDI r10 0 0  // egg_drops 扔下的鸡蛋数量 -- 初始化为0 -> 0010 1010 0000 0000
03 ADDI r11 0 0  // eggs_broken 已经摔碎的数量 -- 初始化为0 -> 0010 1011 0000 0000
04 ADDI r8 0 1    // low -- 将low初始化为0 -> 0010 1000 0000 0001
05 ADD r9 r0 r6   // high -- 将high初始化为总楼层数 -> 0001 1001 0000 0110
06 ADDI r12 0 0  // floor_up 记录上楼的数量 -- 初始化为0 -> 0010 1100 0000 0000
07 ADDI r13 0 0  // floor_down 记录下楼的数量 -- 初始化为0 -> 0010 1101 0000 0000
08 ADDI r14 0 1  // last_position 当前所处的位置 -- 初始化为1 -> 0010 1110 0000 0001
09 ADDI r15 0 0  // total_cost 总代价值 -- 初始化为0 -> 0010 1111 0000 0000
0a ADD r1 r8 r9 // 计算 low+high -> 0001 0001 1000 1001
0b ADDI r10 0 1  // egg_drops++ -> 0010 1010 0000 0001
0c NOP
0d NOP
0e SRL r1 r1 1  // 计算 middle=(low+high)>>1 -> 0110 0001 0001 0001
0f NOP
10 NOP
11 NOP
12 CMP r1 r14 // 将middle层与last_position进行比较 进行计算r1-r14设置
NF ZF ->0111 0000 0001 1110
13 BN r0 2 0 // 当middle<last_floor时跳转到 r0+{x,x} -> 1010 0000 0010 0000
14 NOP
15 NOP
16 NOP
17 SUB r2 r1 r14 // r2=middle-last_floor -> 0011 0010 0001 1110
18 NOP
19 NOP
1a NOP
1b ADD r13 r13 r2 // 记录上楼层数 -> 0001 1101 1101 0010
1c JUMP 2 5 // 跳转到{x,x} -> 1100 0000 0010 0101
1d NOP
1e NOP
1f NOP
20 SUB r2 r14 r1 -> 0011 0010 1110 0001
21 NOP
22 NOP
23 NOP
24 ADD r12 r12 r2 // 记录下楼层数 -> 0001 1100 1100 0010
25 CMP r7 r1 // 将临界楼层和middle层进行比较 -> 0111 0000 0111 0001
26 BN r0 3 3 // 如果临界楼层<middle层 -- 鸡蛋摔碎 -- 跳转到r0+{x,x} -> 1010 0000 0011 0011
27 ADD r14 r0 r1 // last_position=middle -> 0001 1110 0000 0001
28 NOP
29 NOP
```

```
2a ADDI r1 0 1 // middle=middle+1 -> 0010 0001 0000 0001
2b NOP
2c NOP
2d NOP
2e ADD r8 r0 r1  // low = middle -> 0001 1000 0000 0001
2f JUMP 3 9  // 跳转到{x,x} -> 1100 0000 0011 1001
30 NOP
31 NOP
32 NOP
33 ADDI r11 0 1 // eggs_broken++ -> 0010 1011 0000 0001
34 SUBI r1 0 1 // high=middle-1 -> 0100 0001 0000 0001
35 NOP
36 NOP
37 NOP
38 ADD r9 r0 r1 // -> 0001 1001 0000 0001
39 CMP r9 r8  // 将 high 和 low 进行比较  high-low -> 0111 0000 1001 1000
3a BNN r0 0 a // 跳转至 r0+{x,x}继续进行循环计算 -> 1011 0000 0000 1010
3b NOP
3c NOP
3d NOP
3e ADD r15 r15 r11  // 摔碎的鸡蛋数量 -> 0001 1111 1111 1011
3f NOP
40 NOP
41 NOP
42 ADD r15 r15 r12  // 上楼代价 -> 0001 1111 1111 1100
43 NOP
44 NOP
45 NOP
46 ADD r15 r15 r13  // 下楼代价 -> 0001 1111 1111 1101
47 STORE r10 r0 0  // 存储扔下的鸡蛋数量 -> 1110 1010 0000 0000
48 STORE r11 r0 1  // 存储摔碎的鸡蛋数量 -> 1110 1011 0000 0001
49 STORE r14 r0 2  // 存储最终确认的临界楼层 -> 1110 1110 0000 0010
4a STORE r15 r0 3  // 存储最终所耗代价 -> 1110 1111 0000 0011
4b HALT // 停机 -> 1111 1111 1111 1111
```

3. 机器指令（COE 文件）

MachineCode.coe

```
memory_initialization_radix=2;
memory_initialization_vector=
0010011001100100
0010011100110010
0010101000000000
0010101100000000
0010100000000001
0001100100000110
0010110000000000
0010110100000000
0010111000000001
0010111100000000
0001000110001001
0010101000000001
0000000000000000
0000000000000000
0110000100010001
0000000000000000
0000000000000000
0000000000000000
0111000000011110
1010000000100000
0000000000000000
0000000000000000
0000000000000000
0011001000011110
0000000000000000
```

图 4 流水线 CPU 实验性能验证模型机器指令 coe 文件(部分)

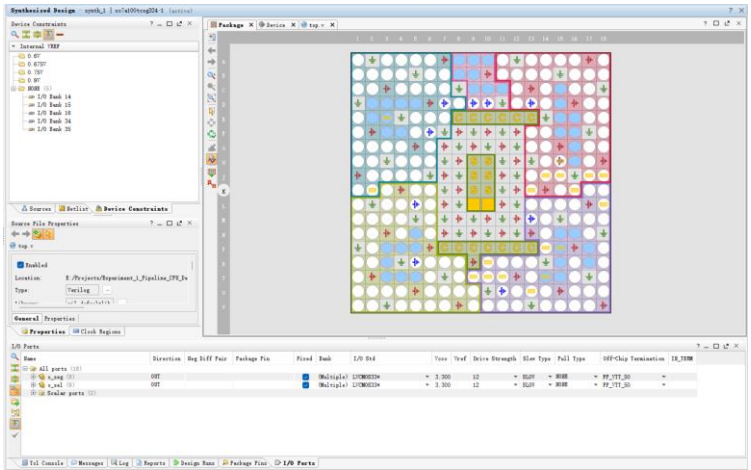
七、实验验算程序下板测试过程与实现

为完成下板测试过程，首先需要在顶层文件引入时钟分频器以及七段数码管显示模块：

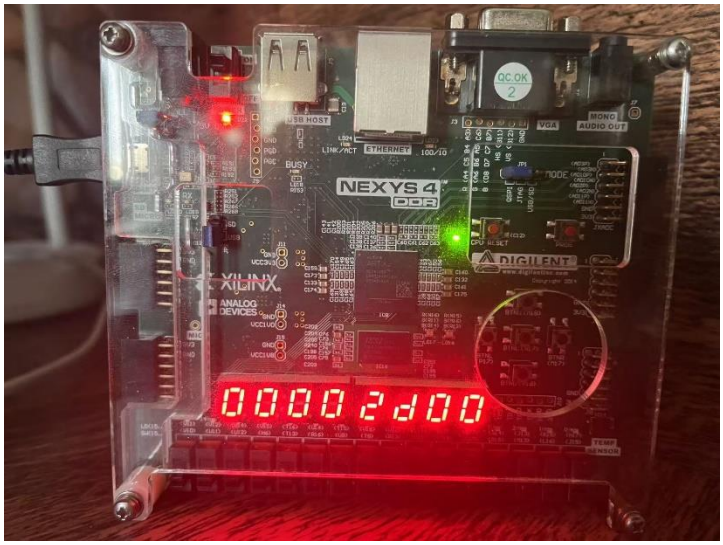
```
    // 实例化分频器
//    assign clk_in = clk;
    clk_divider divider(
        clk,
        reset,
        clk_in
    );
```

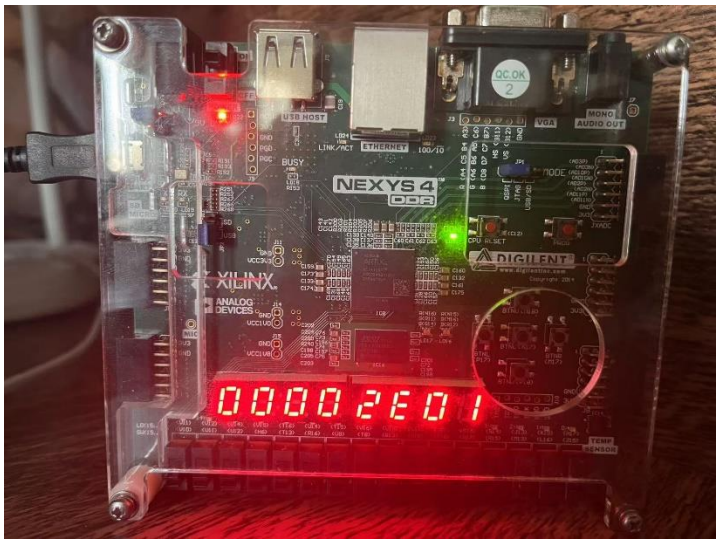
```
// 实例化数码管
seg7x16 show(
    .clk(clk),
    .reset(reset),
    .cs(1'b1),
    .i_data({16'b0, imem_instr}),
    .o_seg(o_seg),
    .o_sel(o_sel)
);
```

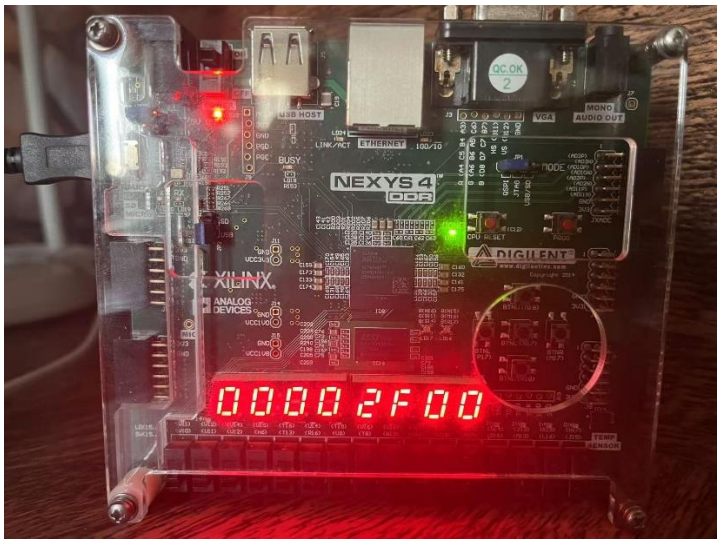
随后进行综合、布线、产生 bitstream：

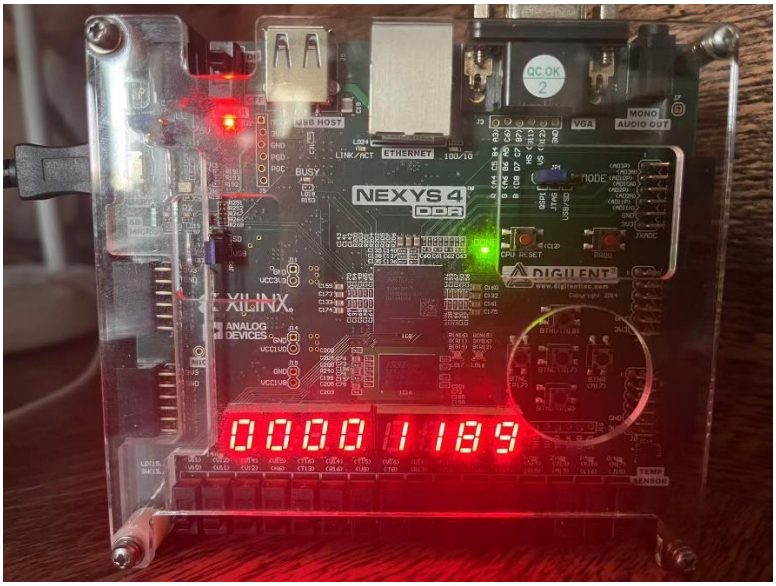


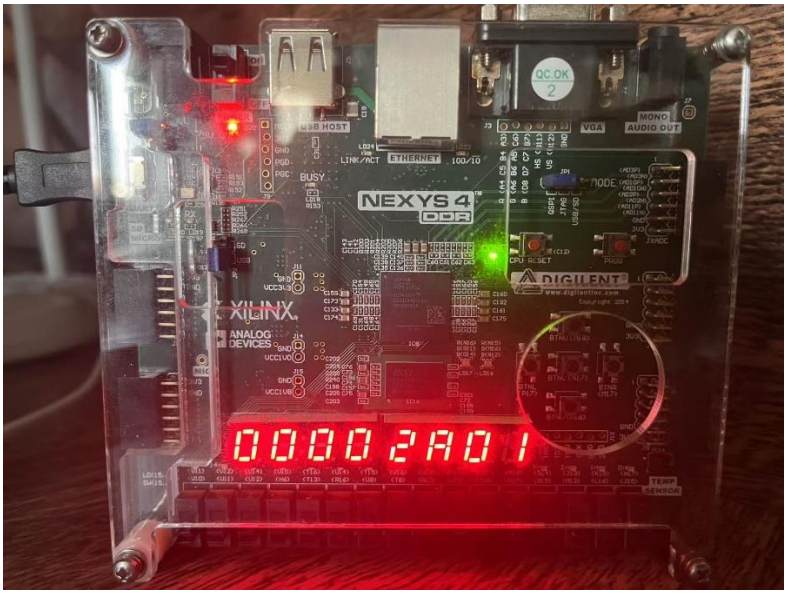
最后下板进行验证，下板运行时数码管上显示的是当前处于 IF 阶段指令的指令码：

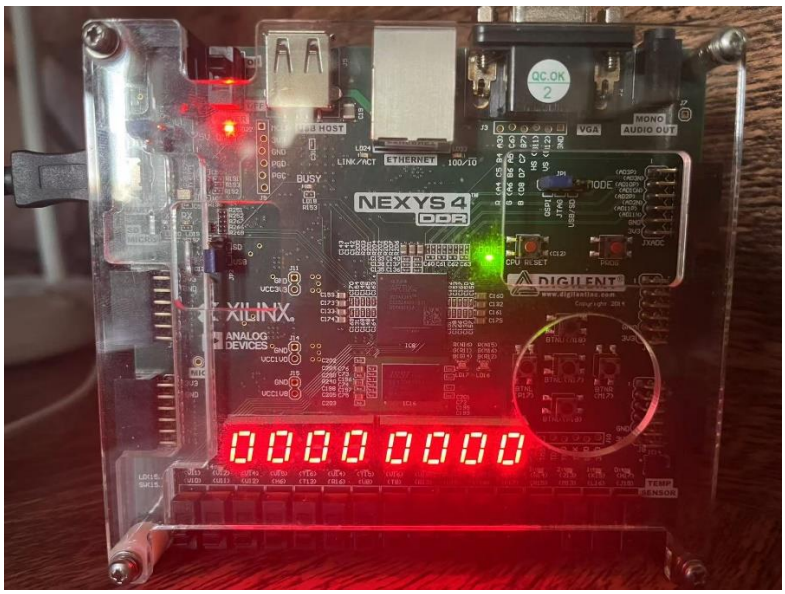







NEXYS 4
DDR
0000 1189


NEXYS 4
DDR
0000 2A01


NEXYS 4
DDR
0000 0000

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）

在本实验中完成一次测试程序，大概需要完成 300 条指令，完成一条指令大概需要 100ns 左右。

1. 吞吐率

$$TP = \frac{n}{(m+n-1)\Delta t}$$

当 n 趋近于无穷大时为10^7，在本测试程序中大概为$9.868 * 10^6$。

2. 加速比

$$S = \frac{m*n*\Delta t}{(m+n-1)*\Delta t} = \frac{m*n}{m+n-1}$$

当 n 趋近于无穷大时为 5，在本测试程序中大概为 4.9342。

3. 效率及冲突分析

$$E = \frac{m*n*\Delta t}{m*(m+n-1)*\Delta t} = \frac{n}{m+n-1}$$

当 n 趋近于无穷大时为 1，在本测试程序中大概为 0.9868。

1. 写读冲突/写写冲突

写读冲突发生在一个指令正在写入寄存器或内存的同时，另一个指令试图从同一个寄存器或内存中读取数据。由于流水线的并行执行特性，这种情况容易导致读取到不正确的值。

写写冲突发生在两个或多个指令同时试图写入同一个寄存器或内存地址时。这种情况可能导致最终结果的不确定性，因为哪个写操作先执行是不可预测的。

解决方案：

1）指令调整：可以通过调整指令的顺序，将写操作与紧随其后的读操作分开。这样可以确保在读操作执行之前，写操作已经完成。

2）添加 NOP 指令：在写读冲突的情况下，适当插入 NOP 指令可以有效地延迟读操作，使得写操作有足够的时间完成。这种方式虽然降低了流水线的效率，但能有效防止数据不一致问题。

2. 跳转指令的冲突

跳转指令在流水线中执行时，会改变指令流。这可能导致流水线中已经加载的指令无效，从而造成冲突。

解决方案：

1）添加 NOP 指令：在跳转指令后面添加 NOP 指令是解决跳转冲突的一种常见做法。这些 NOP 指令为后续指令的加载提供时间，避免在跳转发生后立即执行不再需要的指令。

4. CPU 运行时间及存储空间的使用

CPU 运行时间大概在 30000ns 左右；在本次测试程序中只在最后添加了有关数据存储器操作的指令，其余指令均在寄存器中完成，并且仅使用了数据存储器的四个存储单元。

九、总结与体会

在本次实验中，我深入探讨了从设计指令码到实现静态流水线 CPU 的全过程，积累了丰富的经验和体会。

1. 指令码设计的重要性

首先，指令码的设计是构建 CPU 的基础。在设计阶段，我选择了适合的指令集，以确保 CPU 能够执行各种基本运算和数据操作。通过合理设置操作码长度和指令种类，我确保了指令的简洁性与灵活性。这不仅提高了指令的执行效率，还为后续的流水线设计奠定了坚实的基础。

2. 流水线结构的理解与实现

在构建静态流水线 CPU 的过程中，我认识到流水线结构的核心优势在于提高指令执行的并行度。将指令处理分为五个阶段，使得每个阶段可以同时处理不同指令，从而显著提升了 CPU 的吞吐量。尽管静态流水线相较于动态流水线在灵活性上有所不足，但其设计相对简单，适合于初步的 CPU 架构学习。

3. 资源冲突的处理

在实验中，我深刻体会到资源冲突对流水线性能的影响。为了应对数据依赖和结构冲突，我学习了如何合理安排指令的顺序，并通过插入 NOP 指令来解决潜在的问题。这一过程强调了流水线调度的重要性，促使我更加注重指令间的依赖关系，以确保流水线的高效运行。

4. 测试与验证的必要性

实验过程中，我通过编写 C 语言测试程序，将其手动汇编为汇编语言，再转化为二进制机器代码，最后使用测试基准文件进行仿真测试。这一系列步骤让我意识到，系统的测试与验证对于确保设计的正确性至关重要。通过仿真测试，我能够直观地观察到 CPU 的运行状态，及时发现并解决问题。

5. 面对挑战的收获

在实验过程中，我也遇到了许多挑战。例如，在调试阶段，如何高效地定位问题和优化设计成为我必须面对的任务。通过不断调整和测试，我逐渐掌握了如何在复杂系统中进行调试和优化，提升了自己的问题解决能力。

通过本次实验，我不仅提升了对指令集设计和流水线结构的理解，还在实践中获得了宝贵的经验。这一过程让我更加明确了 CPU 设计中的关键因素，为我未来的学习和研究奠定了基础。未来，我希望能够继续深化对计算机体系结构的理解，探索更为复杂的动态流水线和其他高级特性。

十、附件（所有程序）

1. Top 模块

```
`timescale 1ns / 1ps


module top(
```

```
    input clk,       // 时钟信号
    input reset,        // 全局复位信号
    output [7:0] o_seg, // 数码管输出
    output [7:0] o_sel  // 数码管选择信号
    );

    parameter ADDR_WIDTH = 8;   // 地址位宽
    parameter DATA_WIDTH = 16;  // 数据位宽
    parameter INSTR_WIDTH = 16; // 指令位宽
    parameter REG_COUNT  = 16;  // 寄存器组数量
    parameter MEM_SIZE = 256;   // 存储器大小(单元数量)

    // 数据和指令存储器地址信号
    wire [ADDR_WIDTH-1:0] dmem_addr;
    wire [ADDR_WIDTH-1:0] imem_addr;

    // 数据和指令存储器 I/O 信号
    wire [DATA_WIDTH-1:0] dmem_data_out;
    wire [DATA_WIDTH-1:0] dmem_data_in;
    wire [INSTR_WIDTH-1:0] imem_instr;

    // 控制信号
    wire dmem_enable;
    wire dmem_write;
    wire clk_in;

    // 实例化分频器
//    assign clk_in = clk;
    clk_divider divider(
        clk,
        reset,
        clk_in
     );

    // 实例化数据存储器 DMEM
    DMEM #(
        .ADDR_WIDTH(8),    // 设置地址宽度为 8 位
        .DATA_WIDTH(16),   // 设置数据宽度为 16 位
        .MEM_SIZE(256)     // 设置存储器大小为 256
    ) dmem_inst (
        .clk_in(clk_in),
        .reset(reset),
```

```
        .enable(1'b1),
        .write(dmem_write),
        .addr(dmem_addr), // 使用低 8 位作为地址
        .idata(dmem_data_in),
        .odata(dmem_data_out)
    );

    // 实例化 IMEM 模块
    disrom_256x16b imem_inst (
        .a(imem_addr),      // input wire [7 : 0] a
        .spo(imem_instr)    // output wire [15 : 0] spo
    );

    // 实例化流水线 CPU 模块
    PipelineCPU cpu(
        .clk_in(clk_in),
        .enable(1'b1),
        .reset(reset),
        .start(1'b1),
        .imdata_i(imem_instr),
        .dmdata_i(dmem_data_out),
        .imaddr(imem_addr),
        .dmaddr(dmem_addr),
        .dmwrite(dmem_write),
        .dmdata_o(dmem_data_in)
    );

    // 实例化数码管
    seg7x16 show(
        .clk(clk),
        .reset(reset),
        .cs(1'b1),
        .i_data({16'b0, imem_instr}),
        .o_seg(o_seg),
        .o_sel(o_sel)
    );

endmodule
```

2. DMEM 模块

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

```
module DMEM #(
    parameter ADDR_WIDTH = 8,  // 地址位宽
    parameter DATA_WIDTH = 16, // 数据位宽
    parameter MEM_SIZE = 256   // 存储器大小(单元数量)
)(
    input clk_in,         // 时钟信号
    input reset,          // 复位信号
    input enable,         // 存储器使能信号，高电平有效
    input write,          // 存储器写入信号，高电平有效
    input [ADDR_WIDTH-1:0] addr,     // 存储器地址信号
    input [DATA_WIDTH-1:0] idata,    // 输入数据
    output [DATA_WIDTH-1:0] odata    // 输出数据
    );

    integer i;  // 用于遍历存储单元进行复位

    reg [DATA_WIDTH - 1:0] mem [0:MEM_SIZE - 1];

    // 读取操作：在 `enable` 为高电平时输出对应地址的存储内容，否则为高阻
态
    assign odata = (enable == 1'b1) ? mem[addr] :
{DATA_WIDTH{1'bz}};

    // 写入操作和复位操作：在时钟上升沿，根据 `enable`、`write` 和
`reset` 控制信号执行
    always @(posedge clk_in or negedge reset) begin
        if (!reset) begin
            for (i = MEM_SIZE; i != 0; i = i - 1) begin
                mem[i - 1] <= {DATA_WIDTH{1'b0}}; // 清零每个存储
单元
            end
        end
        else if (enable == 1'b1 && write == 1'b1) begin
            mem[addr] <= idata; // 写入数据到指定地址
        end
    end

endmodule
```

3. PipelineCPU 模块

```
`timescale 1ns / 1ps
`include "opcodes.vh"

// 模块名: PipelineCPU
// 功能: 模拟一个简单的五级流水线 CPU，通过流水线结构分解指令执行过程
// 模块描述:
// 该流水线 CPU 模块包含五个阶段，每个阶段完成指令执行的不同部分。
// 模块通过时钟驱动各流水线阶段的信号流动，逐步实现指令的取指、译码、执行、内存操作和写回功能。
//  1. IF（取指阶段）: 获取下一条指令的地址，并从程序存储器取出指令
//  2. ID（指令译码阶段）: 解析指令字段，确定操作码和操作数
//  3. EX（执行阶段）: 进行算术或逻辑操作，根据操作码执行不同的运算
//  4. MEM（内存访问阶段）: 如果指令涉及存储器，则在此阶段读取或写入数据
//  5. WB（写回阶段）: 将计算结果或加载的数据写回通用寄存器

module PipelineCPU(
    input clk_in,               // 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] imdata_i,      // 输入指令
    input [15:0] dmdata_i,      // 存储器读取数据
    output [7:0] imaddr,        // 输出指令地址 -- 用于获取指令
    output [7:0] dmaddr,        // 输出数据地址 -- 用于读取/写入存储器
    output dmwrite,             // 数据写入信号
    output [15:0] dmdata_o      // 存储器写入数据
    );

    parameter ADDR_WIDTH = 8;   // 地址位宽
    parameter DATA_WIDTH = 16;  // 数据位宽
    parameter INSTR_WIDTH = 16; // 指令位宽
    parameter REG_COUNT  = 16;  // 寄存器数量

    reg [7:0] PCReg;            // 指令计数器
    reg [INSTR_WIDTH-1:0] ID_IReg, EX_IReg, MEM_IReg, WB_IReg;
// 各阶段指令寄存器

    reg [DATA_WIDTH-1:0] RegFiles[REG_COUNT-1:0];  // 寄存器组
    reg [DATA_WIDTH-1:0] ALUo;                     // ALU 计算结果
```

```
    reg [DATA_WIDTH-1:0] RegA, RegB, RegC1, RegC2;    // ALU 相
关寄存器
    reg [DATA_WIDTH-1:0] StoreReg1, StoreReg2;        // 存储需要
存储到 DMEM 中的数据

    reg CPUState, CPUNextState; // CPU 状态机
    reg zf, nf, cf, dw;         // 标志位

    // CPU 状态机控制
    always@(posedge clk_in) begin
        if(!reset) begin
            CPUState <= `IDLE;
        end else begin
            CPUState <= CPUNextState;
        end
    end
    always@(*) begin
        case(CPUState)
            `IDLE: begin
                if(enable == 1'b1 && start == 1'b1) begin
                    CPUNextState <= `EXEC;
                end else begin
                    CPUNextState <= `IDLE;
                end
            end
            `EXEC: begin
                if(enable == 1'b0 || start == 1'b0 ||
WB_IReg[15:12] == `HALT) begin
                    CPUNextState <= `IDLE;
                end else begin
                    CPUNextState <= `EXEC;
                end
            end
        endcase
    end

    // IF 阶段
    // 该阶段的任务是根据 PC 的值从指令内存中读取一条指令，并且设置下一个周
期 PC 的值
    assign imaddr = PCReg;
    always@(posedge clk_in or negedge reset) begin
        if(reset == 1'b0) begin
```

```
            ID_IReg <= {`NOP, 12'b0};
            PCReg <= 8'b0;
        end else if(CPUState == `EXEC) begin
            ID_IReg <= imdata_i;
            if((MEM_IReg[15:12] == `JUMP)
            ||(MEM_IReg[15:12] == `BZ && zf == 1'b1)
            ||(MEM_IReg[15:12] == `BN && nf == 1'b1)
            ||(MEM_IReg[15:12] == `BNZ && !zf)
            ||(MEM_IReg[15:12] == `BNN && !nf)) begin
                // 跳转指令满足条件时
                PCReg <= RegC1[7:0];
            end else begin
                PCReg <= PCReg + 1;
            end
        end
    end

    // ID 阶段
    always@(posedge clk_in or negedge reset) begin
        if(reset == 1'b0) begin
            EX_IReg <= {`NOP, 12'b0};
            RegA <= 16'b0;
            RegB <= 16'b0;
            StoreReg1 <= 16'b0;
        end else if(CPUState == `EXEC) begin
            EX_IReg <= ID_IReg;
            // 为 StoreReg1 赋值
            if(ID_IReg[15:12] == `STORE) begin
                StoreReg1 <= RegFiles[ID_IReg[11:8]];
            end else begin
                StoreReg1 <= StoreReg1;
            end
            // 为操作数 A 赋值
            if(ID_IReg[15:12] == `JUMP) begin
                RegA <= 16'b0;
            end else if(ID_IReg[15:12] == `ADDI
            || ID_IReg[15:12] == `SUBI
            || ID_IReg[15:12] == `BN
            || ID_IReg[15:12] == `BNN
            || ID_IReg[15:12] == `BZ
            || ID_IReg[15:12] == `BNZ) begin
                // RegA 使用 r1 位置的操作数
```

```
            RegA <= RegFiles[ID_IReg[11:8]];
        end else if(ID_IReg[15:12] == `ADD
        || ID_IReg[15:12] == `SUB
        || ID_IReg[15:12] == `LOAD
        || ID_IReg[15:12] == `STORE
        || ID_IReg[15:12] == `SLL
        || ID_IReg[15:12] == `SRL
        || ID_IReg[15:12] == `CMP) begin
            // RegA 使用 r2 位置的操作数
            RegA <= RegFiles[ID_IReg[7:4]];
        end else begin
            RegA <= RegA;
        end
        // 为操作数 B 赋值
        if(ID_IReg[15:12] == `LOAD
        || ID_IReg[15:12] == `STORE
        || ID_IReg[15:12] == `SLL
        || ID_IReg[15:12] == `SRL) begin
            RegB <= {12'b0, ID_IReg[3:0]};
        end else if(ID_IReg[15:12] == `ADDI
        || ID_IReg[15:12] == `SUBI
        || ID_IReg[15:12] == `BN
        || ID_IReg[15:12] == `BNN
        || ID_IReg[15:12] == `BZ
        || ID_IReg[15:12] == `BNZ
        || ID_IReg[15:12] == `JUMP) begin
            RegB <= {8'b0, ID_IReg[7:0]};
        end else if(ID_IReg[15:12] == `ADD
        || ID_IReg[15:12] == `SUB
        || ID_IReg[15:12] == `CMP) begin
            RegB <= RegFiles[ID_IReg[3:0]];
        end else begin
            RegB <= RegB;
        end
    end
end

// EX 阶段
// 该阶段将执行 ALU 运算并设置标志寄存器
always@(posedge clk_in or negedge reset) begin
    if(reset == 1'b0) begin
        MEM_IReg <= {`NOP, 12'b0};
```

```
            RegC1 <= 16'b0;
            StoreReg2 <= 16'b0;
            dw <= 1'b0;
            zf <= 1'b0;
            nf <= 1'b0;
            cf <= 1'b0;
        end else if(CPUState == `EXEC) begin
            MEM_IReg <= EX_IReg;
            RegC1 <= ALUo;
            // 标志位赋值
            if((EX_IReg[15:12] == `ADD)
            ||(EX_IReg[15:12] == `CMP)) begin
                if(ALUo == 16'b0) begin
                    zf <= 1'b1;
                end else begin
                    zf <= 1'b0;
                end
                if(ALUo[15] == 1'b1) begin
                    nf <= 1'b1;
                end else begin
                    nf <= 1'b0;
                end
            end else begin
                zf <= zf;
                nf <= nf;
                cf <= cf;
            end
            if(EX_IReg[15:12] == `STORE) begin
                dw <= 1'b1;
                StoreReg2 <= StoreReg1;
            end else begin
                dw <= 1'b0;
                StoreReg2 <= StoreReg2;
            end
        end
    end
    // ALU 计算过程
    always@(*) begin
        if(EX_IReg[15:12] == `ADDI
        || EX_IReg[15:12] == `ADD
        || EX_IReg[15:12] == `BN
        || EX_IReg[15:12] == `BNN
```

```
        || EX_IReg[15:12] == `BZ
        || EX_IReg[15:12] == `BNZ
        || EX_IReg[15:12] == `JUMP
        || EX_IReg[15:12] == `STORE
        || EX_IReg[15:12] == `LOAD) begin
            ALUo <= RegA + RegB;
        end else if(EX_IReg[15:12] == `SUB || EX_IReg[15:12] ==
`SUBI || EX_IReg[15:12] == `CMP) begin
            ALUo <= RegA - RegB;
        end else if(EX_IReg[15:12] == `SLL) begin
            ALUo <= (RegA << RegB[3:0]);
        end else if(EX_IReg[15:12] == `SRL) begin
            ALUo <= (RegA >> RegB[3:0]);
        end  else begin
            ALUo <= ALUo;
        end
    end

    // MEM 阶段
    // 该阶段要根据指令功能和上一阶段的运行结果决定是否要访问内存以及如何
访问
    assign dmaddr = RegC1[7:0];
    assign dmwrite = dw;
    assign dmdata_o = StoreReg2;
    assign BranchFlag = 1'b1;
    always@(posedge clk_in or negedge reset) begin
        if(reset == 1'b0) begin
            WB_IReg <= {`NOP, 12'b0};
            RegC2 <= 16'b0;
        end else if(CPUState == `EXEC) begin
            WB_IReg <= MEM_IReg;
            if(MEM_IReg[15:12] == `LOAD) begin
                RegC2 <= dmdata_i;
            end else begin
                RegC2 <= RegC1;
            end
        end
    end

    // WB 阶段
    // 该阶段同样根据指令的功能以及上一阶段的结果决定是否要修改寄存器的值
以及如何修改
```

```
    always@(posedge clk_in or negedge reset) begin
        if(reset == 1'b0) begin
            RegFiles[15] <= {DATA_WIDTH{1'b0}};
            RegFiles[14] <= {DATA_WIDTH{1'b0}};
            RegFiles[13] <= {DATA_WIDTH{1'b0}};
            RegFiles[12] <= {DATA_WIDTH{1'b0}};
            RegFiles[11] <= {DATA_WIDTH{1'b0}};
            RegFiles[10] <= {DATA_WIDTH{1'b0}};
            RegFiles[9] <= {DATA_WIDTH{1'b0}};
            RegFiles[8] <= {DATA_WIDTH{1'b0}};
            RegFiles[7] <= {DATA_WIDTH{1'b0}};
            RegFiles[6] <= {DATA_WIDTH{1'b0}};
            RegFiles[5] <= {DATA_WIDTH{1'b0}};
            RegFiles[4] <= {DATA_WIDTH{1'b0}};
            RegFiles[3] <= {DATA_WIDTH{1'b0}};
            RegFiles[2] <= {DATA_WIDTH{1'b0}};
            RegFiles[1] <= {DATA_WIDTH{1'b0}};
            RegFiles[0] <= {DATA_WIDTH{1'b0}};
        end else if(CPUState == `EXEC) begin
            if((WB_IReg[15:12] == `LOAD)
            || (WB_IReg[15:12] == `ADD)
            || (WB_IReg[15:12] == `ADDI)
            || (WB_IReg[15:12] == `SUB)
            || (WB_IReg[15:12] == `SUBI)
            || (WB_IReg[15:12] == `SRL)
            || (WB_IReg[15:12] == `SLL)) begin
                RegFiles[WB_IReg[11:8]] <= RegC2;
            end
        end
    end
endmodule
```

4. 操作码定义头文件

```
// 文件名: opcodes.vh
// 作用: 定义流水线 CPU 中各指令的操作码，方便在不同模块中统一引用
// 使用方法: 在需要使用操作码的 Verilog 文件中加入 `include
"opcodes.vh"`
// 注意事项: 添加新操作码时，确保编码唯一，避免冲突

`define IDLE 1'b0
`define EXEC 1'b1
```

```
// 基本指令
`define NOP   4'b0000  // 无操作（No Operation）
`define HALT  4'b1111  // 停机指令，停止执行流水线

// 数据存取
`define LOAD  4'b1101  // 将数据存储器数据加载寄存器
`define STORE 4'b1110  // 将寄存器数据存入数据存储器

// 算术操作
`define ADD   4'b0001  // 加法指令
`define ADDI  4'b0010  //
`define SUB   4'b0011
`define SUBI  4'b0100
`define SLL   4'b0101
`define SRL   4'b0110

// 跳转指令
`define CMP   4'b0111
`define BZ    4'b1000
`define BNZ   4'b1001
`define BN    4'b1010
`define BNN   4'b1011
`define JUMP  4'b1100
```

5. 七段数码管展示模块

```
`timescale 1ns / 1ps

// 七段数码管,用于下板展示
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, negedge reset)
        if (reset == 1'b0)
            cnt <= 0;
        else
```

```
        cnt <= cnt + 1'b1;

    wire seg7_clk = cnt[14];

    reg [2:0] seg7_addr;

always @ (posedge seg7_clk, negedge reset)
    if(reset == 1'b0)
        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, negedge reset)
    if(reset == 1'b0)
        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, negedge reset)
        if(reset == 1'b0)
            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
```

6. 时钟信号分频器模块

```
// 时钟信号分频器,用于下板测试
module clk_divider(
    input I_CLK,
    input rst,
    output reg O_CLK
    );
```

```
    //分频器的默认分频倍数是 20
    parameter Magnification = 99_999_999;
    //parameter Magnification = 20;
    reg [31:0]counter = 0;

    initial begin
        O_CLK = 0;
    end

    always@(posedge I_CLK) begin
        //复位信号
        if(rst == 1'b0) begin
            counter = 0;
            O_CLK = 0;
        end
        //计数 CLk
        else begin
            counter = counter + 1'b1;
            if(counter == Magnification / 2 || counter ==
Magnification) begin
                if(O_CLK == 0)
                    O_CLK = 1;
                else if(O_CLK == 1)
                    O_CLK = 0;
                if(counter == Magnification) begin
                    counter = 0;
                end
            end
        end
    end

endmodule
```