

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：2052697

姓名：刘毅

指导教师：陆有军

日期：2022.11.19

1. 实验环境部署与硬件配置说明

1.实验软件环境：

1.1操作系统版本

**版本 Windows 11 家庭中文版**

**版本 21H2**

**操作系统版本 22000.1219**

**体验 Windows 功能体验包 1000.22000.1219.0**

1.2硬件描述语言

Verilog HDL语言

1.3硬件编程环境

（1）vivado（所使用的硬件编程软件） ：

**Vivado v2016.2 (64-bit)  
SW Build 1577090 on Thu Jun 2 16:32:40 MDT 2016  
IP Build 1577682 on Fri Jun 3 12:00:54 MDT 2016**   
**Centerpoint XML**  
The Initial Developer of the Original Code is CenterPoint - Connective Software  
Software Engineering GmbH. Portions created by CenterPoint - Connective Software  
Software Engineering GmbH. are Copyright © 1998-2000 CenterPoint -  
Connective Software Engineering GmbH. All Rights Reserved.   
Source code for Centerpoint XML is available at http://www.cpointc.com/XML/  
**NLView Schematic Engine**   
Copyright © Concept Engineering  
Static Timing Analysis by Parallax Software Inc.  
Copyright © Parallax Software Inc.  
**Java 2 Standard Edition**  
Includes portions of software from RSA Security, Inc. and some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/  
**Powered by JIDE** - http://www.jidesoft.com



（2）modelsim（所使用的仿真测试软件）：

**ModelSim PE 10.4c**

**Revision: 2015.07**

**Date: Jul 20 2015**

**Copyright 1991-2015 Mentor Graphics Corporation**

**All rights reserved.**

**Tcl and other GUI Copyrights**

**SystemC Copyrights**

**GPL/LGPL Copyrights**

**Mentor Graphics Support**

**North America Toll Free: (800) 547-4303**

**http://www.mentor.com/supportnet**

2.实验硬件配置

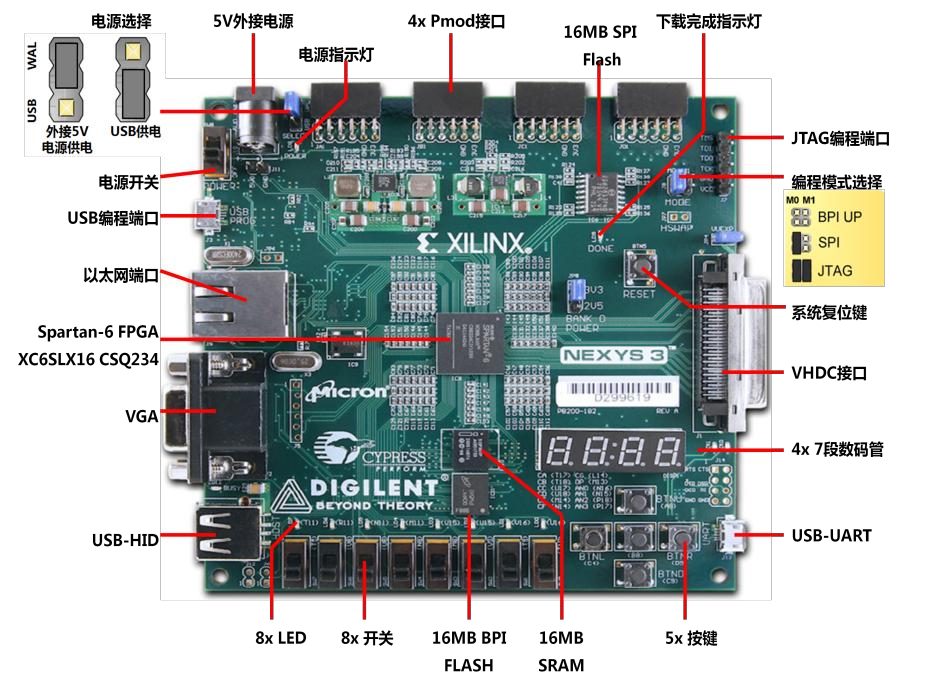
2.1上位机配置

处理器 Intel(R) Core(TM) i7-10875H CPU @ 2.30GHz 2.30 GHz

机带 RAM 16.0 GB (15.8 GB 可用)

系统类型 64 位操作系统, 基于 x64 的处理器

2.2操作板配置



 Xilinx Spartan6 XC6LX16-CS324

 16M 字节 Micron 公司的 Cellular RAM

 16M 字节 Micron 公司的 BPI PCM Flash

 16M 字节 Micron 公司的 Quad-mode SPI PCM Flash

 10/100 SMSC LAN8710 PHY

 Digilent Adept USB 接口提供电源、程序烧录和数据传输

 USB-UART,USB-HID(OTG)

 8 位 VGA

 100MHz 晶振

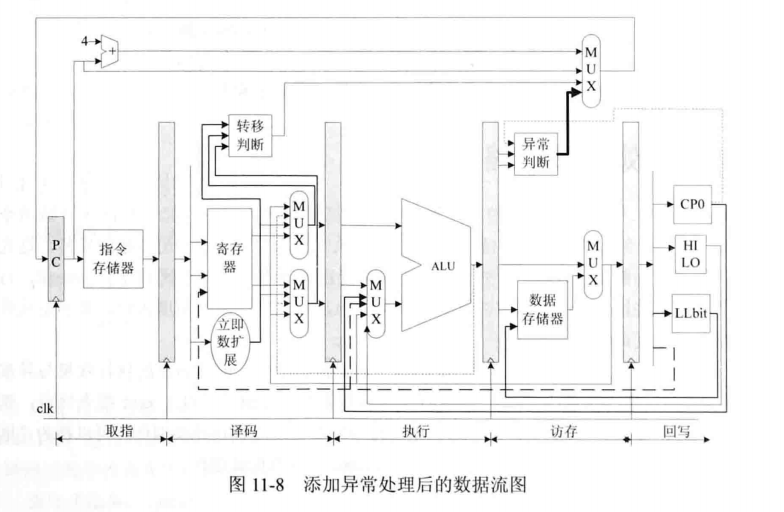
 8 个拨码开关, 4 个按钮

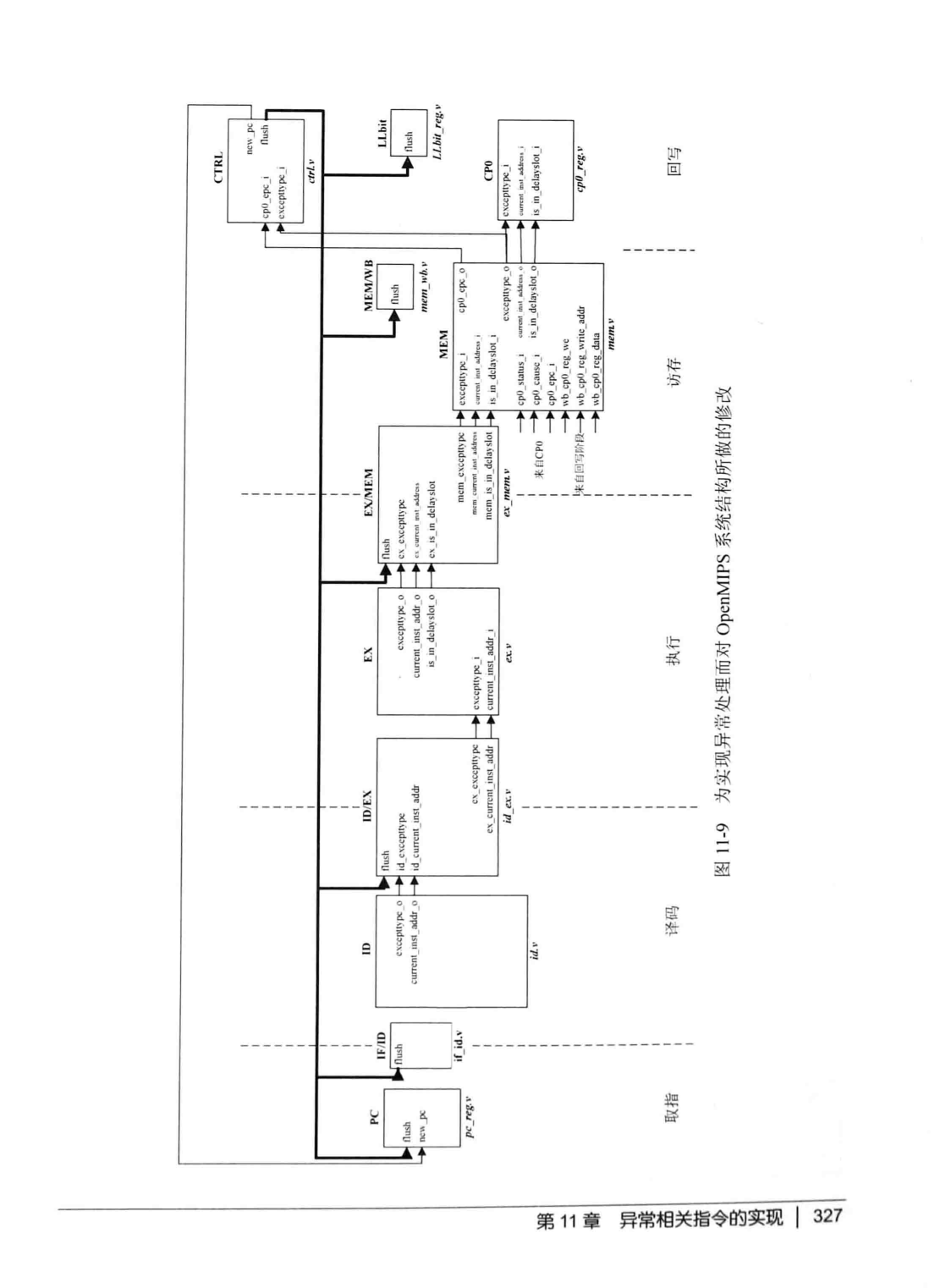
 4 个 7 段数码管, 8 个 LED

 4 个双层 Pmod 接口，一个 VHDC 接口

1. 实验的总体结构
2. 静态流水线的总体结构

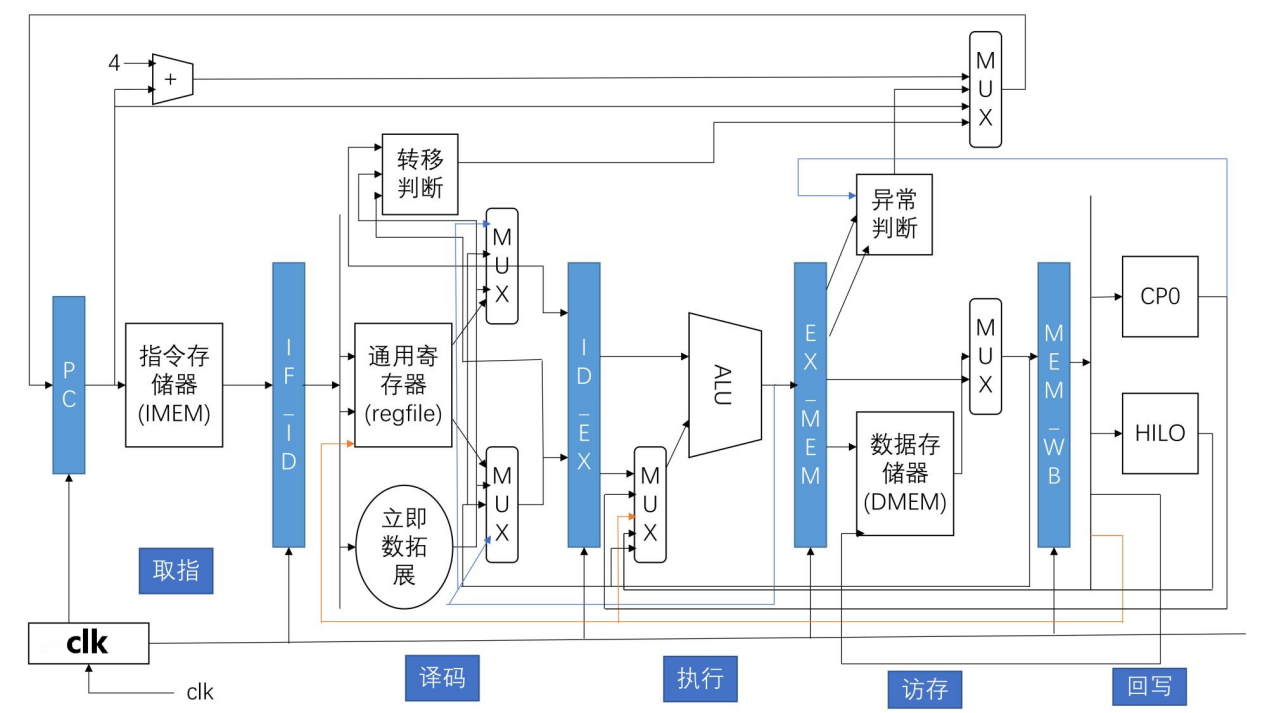
以下为实现过程中参考的数据通路，按照这样的数据通路设计了此次实验的静态流水线（部分指令实现了动态，但是大部分还是没有）总体结构。需指出的是，下面两张架构图实现的内容有：（1）逻辑、移位操作和空指令；（2）移动操作指令；（3）算术操作指令；（4）转移操作指令；（5）加载存储指令；（6）协处理器访问指令；（7）异常相关指令。下面会详细介绍实现了哪些指令。



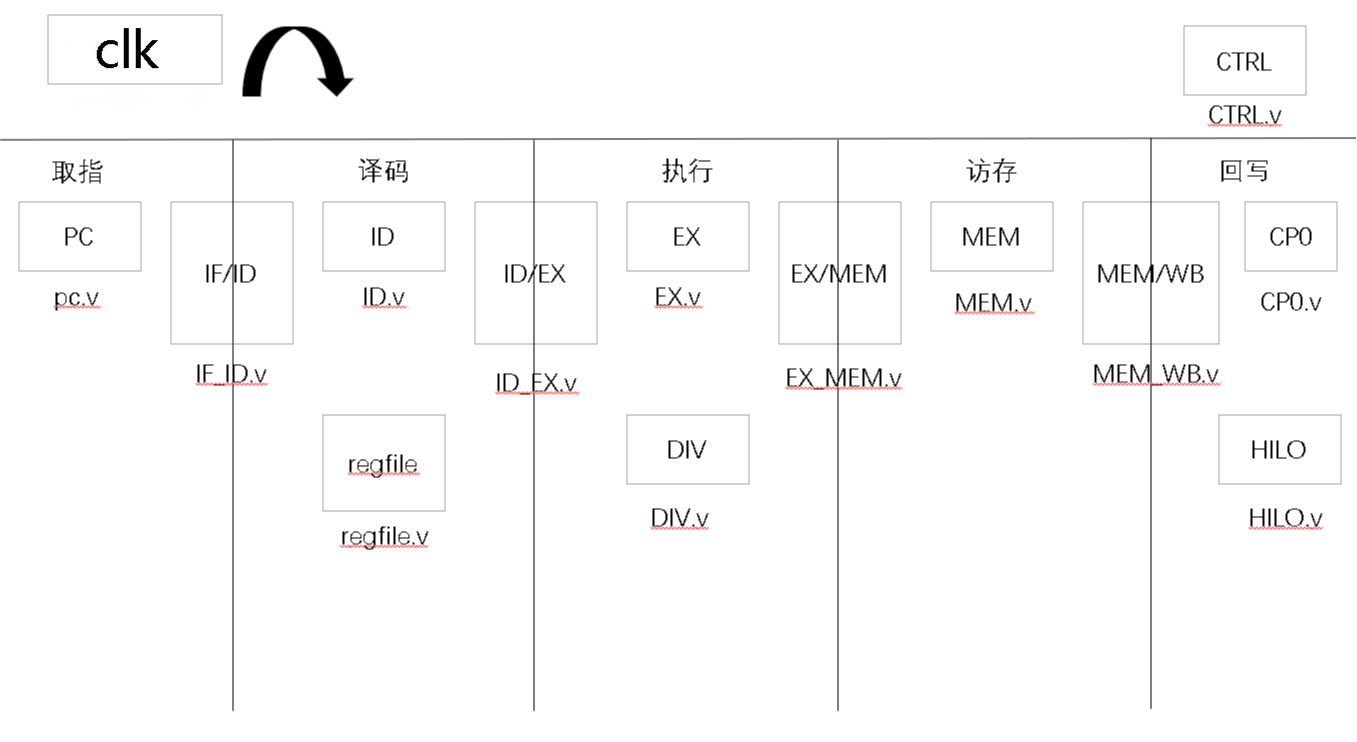


图源：《自己动手写CPU》

参考上面的两张图，得出了下面的本次设计的流水线MIPS架构CPU的总体结构图——

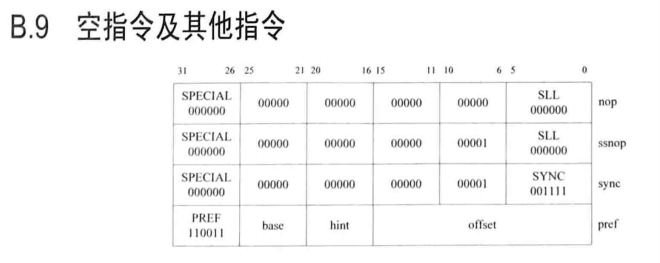
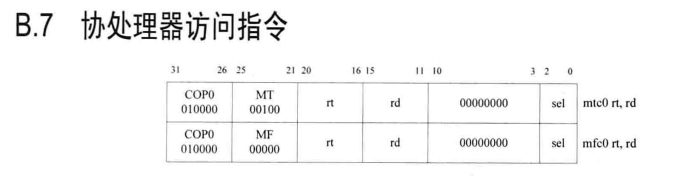
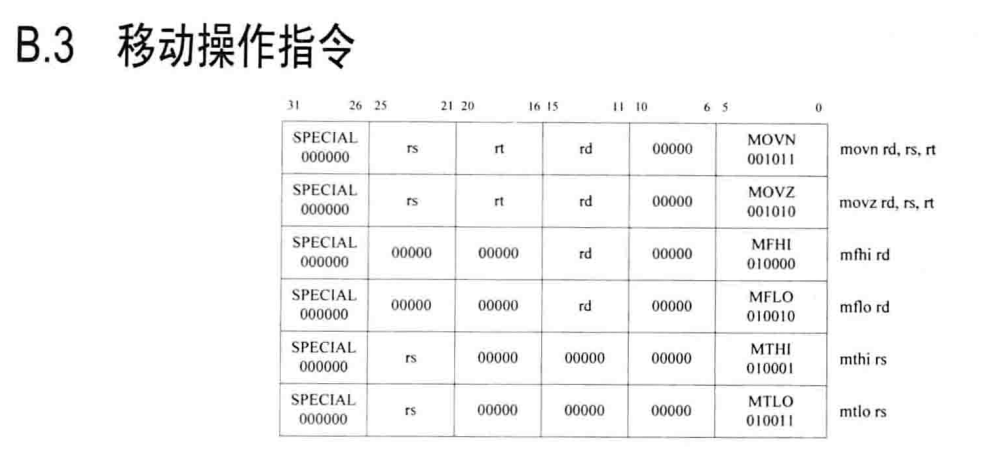
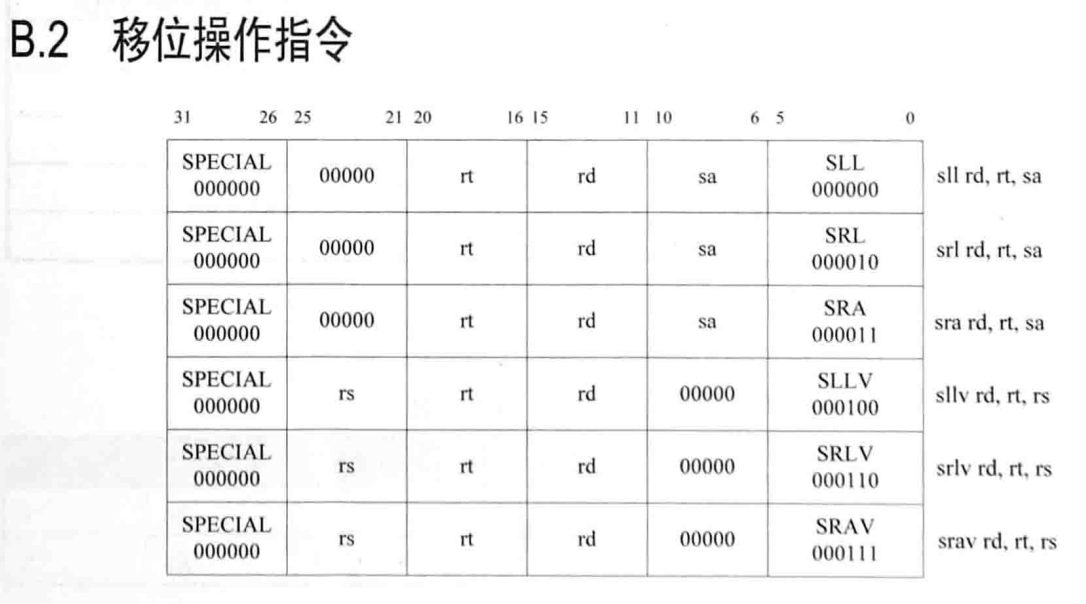


以及简单的按流水线顺序部署的部件图如下：



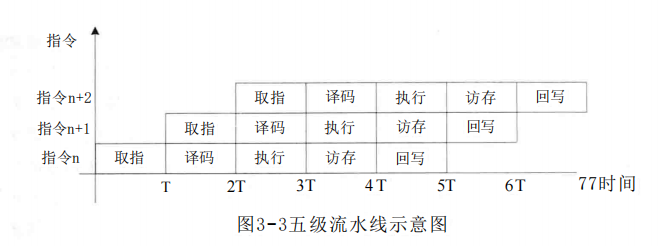
1. 静态流水线实现的指令

总共9类指令，指令的格式如下图所示——



1. 总体架构部件的解释说明
   1. 静态流水线总体结构部件的解释说明

本实验实现的流水线CPU是MIPS架构下的五级流水线CPU（指令执行分五个阶段，如下图），具体实现架构图上文已有。参照上面给出的流水线CPU总体架构图，接下来对每个部件的功能做出简要的解释：



图源：《自己动手写CPU》

（1）取指阶段：从指令存储器读出指令，同时确定下一条指令地址。具体模块有：

PC模块：给出指令地址，其中实现指令寄存器PC，该寄存器的值就是指令的地址。对应文件是pc.v。

IF/ID模块：实现取指与译码阶段之间的寄存器，将取指结果（取得指令、指令地址等信息）在下一个时钟周期传递到译码阶段。 对应文件是IF\_ID.v。

（2）译码阶段：含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。具体模块有：

ID模块：对指令进行译码，译码结果包括运算类型、运算所需源操作数、要写入的目标寄存器。对应文件是ID.v。

regfile模块：实现32个32位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作。对应文件是regfile.v。

ID\_EX模块：在ID和EX模块之间传递信息，等一个时钟周期后，将源操作数、运算类型、写控制信号、写地址等信息传递到EX模块。

（3）执行阶段：按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是Load/Store指令，那么还会计算Load/Store的目标地址。具体模块有：

EX模块：根据译码阶段的结果，进行指定的运算，给出运算结果。这些运算包括逻辑运算、算术运算、移动运算等等，需要分别在该模块中实现，该模块对应一般冯诺依曼架构中的ALU单元。该模块对应文件是EX.v。

DIV模块：进行除法运算的模块，因为进行除法运算需要做流水线的暂停（并且一般需要连续暂停32个时钟周期以等待结果计算得到），所以需要特意实现该模块完成除法相关的指令。对应文件是DIV.v。

EX/MEM模块：实现执行与访存阶段之间的寄存器，将执行阶段的结果在下一个时钟周期传递到访存阶段。对应文件是EX\_MEM.v。

（4）访存阶段：如果是Load/Store指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行。具体模块有：

MEM模块：如果是加载、存储指令，那么会对数据存储器进行访问。此外，还会在该模块进行异常判断。对应文件是MEM.v。

MEM/WB模块：实现访存与回写阶段之间的寄存器，将访存阶段的结果在下一个时钟周期传递到回写阶段。对应文件是MEM\_WB.v。

（5）回写阶段：将运算结果保存到目标寄存器。具体模块有：

CP0模块：对应MIPS架构中的协处理器CP0。协处理器一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展，具有

与处理器核独立的寄存器。MIPS32架构提供了最多4个协处理器，分别是CPO～CP3。CP0是用来实现系统控制的协处理器，需要实现。CP0主要功能有：配置CPU工作状态、高速缓存控制、异常控制、存储单元管理控制等。本模块实现其中的部分功能，对应的是CP0.v文件。

HILO模块：实现寄存器HI、LO，在乘法、除法指令的处理过程中会使用这两个寄存器。对应的是HILO.v文件。

（6）其他：主要是全局的控制模块

CTRL模块，用来控制整个流水线的暂停、清除等等操作。在多周期流水线CPU中，其负责在div和divu指令的时候暂停控制信号等待除法完成。对应文件CTRL.v。

* 1. 静态流水线对指令冲突问题的解决方案

流水线CPU实现的过程中，必然会遇到一些指令操作数的冲突问题，这些问题及其在本实验中的解决方案主要如下。

（1）数据相关。数据相关问题主要分为三种，即

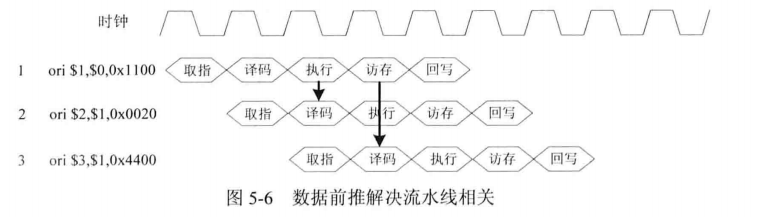
RAW，Read After Write，所谓先写后读。假设指令j是在指令i后面执行的指令，RAW表示指令i将数据写入寄存器后，指令j才能从这个寄存器读取数据。如果指令j在指令i写入寄存器前尝试读出该寄存器的内容，将得到不正确的数据。

WAR，即Write After Read，所谓先读后写，假设指令j是在指令i后面执行的指令，WAR表示指令i读出数据后，指令j才能写这个寄存器。如果指令j在指令i读出数据前就写该寄存器，将使得指令i读出的数据不正确。

WAW，即Write after Write，所谓先写后写，假设指令j是在指令i后面执行的指令，WAW表示i将数据写入寄存器后，j才能将数据写入寄存器，如果j抢在i前面写入数据，最后就会导致该寄存器中的值不是最新值。

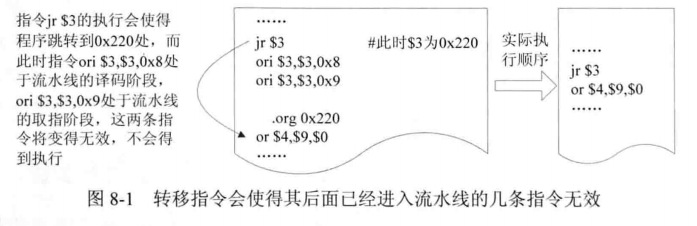
对于此次实现的流水线CPU而言，只在流水线回写阶段才会写寄存器，就没有WAW相关；又因只在流水线译码阶段读寄存器，就没有WAR相关。于是要解决的就是RAW相关。这一问题已经完成了动态流水线（数据前推）方式解决，而非暂停一个周期。

数据前推：将计算结果从其产生处直接送到其他指令需要处或所有需要的功能单元处，避免流水线暂停。本实验中所有数据相关的问题都通过这一方法解决。一个具体实例如下：



图源：《自己动手写CPU》

（2）控制相关，即流水线中转移指令或者其他需要修改PC的指令之间存在的相关关系。本实验通过静态流水线的方式解决了这类相关问题。这些指令修改了PC是值，可能会造成后面的几条指令无效（发生跳转的情况下），一个具体实例如下。

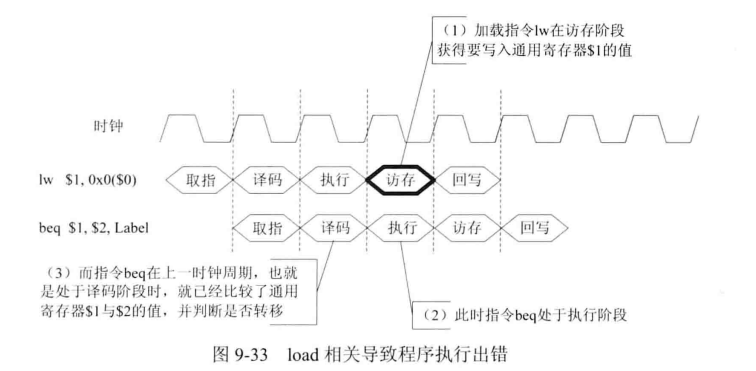


为解决这一问题，先引入了延迟槽的概念。如果流水线中存在不跳转指令，且发生跳转，则可能导致下面的两条指令无效（判断是否跳转需要两个周期时间），这就浪费了两个时钟周期。就可以把跳转指令后面的两条指令作为延迟槽，延迟槽总是被执行。如果未发送跳转，那么两个时钟周期就没有浪费；如果发生了跳转，那么两个时钟周期还是浪费了。这样降低了损失，这是动态流水线方法解决控制相关的方法。

但是本实验是通过静态方法解决了，引入了延迟槽，但是在延迟槽中卡了nop指令（即空指令），这样实质上是让流水线暂停了，这样损失了性能，但是简单的解决了问题。

（3）load相关

以下为一个load相关的实例。这种相关一般是由于load指令向相关寄存器中保存数据，其后如有指令涉及这一寄存器，就造成了load相关。这一问题无法再通过数据前推解决了。



OpenMIPS解决load相关的方法是：在译码阶段检查当前指令与上一条指令是否存在load 相关，如果存在load相关，那么就让流水线的译码、取指阶段暂停，而执行、访存、回写阶段继续，相当于插入一个空指令，这样处于执行阶段的加载指令会继续运行，不受影响，当其运行到访存阶段时，将加载得到的数据前推到译码阶段，然后，流水线可以继续运行。

* 1. 静态流水线暂停的处理方案

因为本实验中如除法指令等指令在流水线执行阶段占用多个时钟周期，因此需要暂停流水线，以等待这些多周期指令执行完毕，一种直观的实现方法是：要暂停流水线，只需保持取指令地址PC的值不变，同时保持流水线各个阶段的寄存器(也即IF\_ID、ID\_EX、EX\_MEM、MEM\_WB模块的输出)不变。

设计一个CTRL模块来完成这个任务。CTRL模块的输入来自ID、EX模块的请求暂停信号stallreq。CTRL模块对暂停请求信号进行判断，然后输出流水线暂停信号stall。stall输出到PC、IF\_ID、ID\_EX、EX\_MEM、MEM\_WB等模块，从而控制PC的值，以及流水线各个阶段的寄存器。

当暂停发生时，PC维持原有值不变保持取指、译码、执行阶段的寄存器不变，但是可以允许访存、回写阶段的指令继续运行。

1. 实验仿真过程
   1. 静态流水线的仿真过程

（1）前仿真

前仿真按照这样的方法进行：

1.提供一个测试某某指令的mips汇编文件，将其汇编为机器码。

2.将机器码写到指令存储器之中，CPU从中取指。

3.运行前仿真的测试程序，输出每个周期CPU中通用寄存器、PC和指令的信息到一个文件中。

4.用MARS汇编器汇编原测试某某指令的汇编文件，输出result.txt文件（保存得也是每个周期CPU中通用寄存器、PC和指令的信息）。

5.对比result.txt文件和前仿真程序输出的信息，观察最后二者结果是否一致，如果一致，则说明对该指令前仿真的测试是成功的。

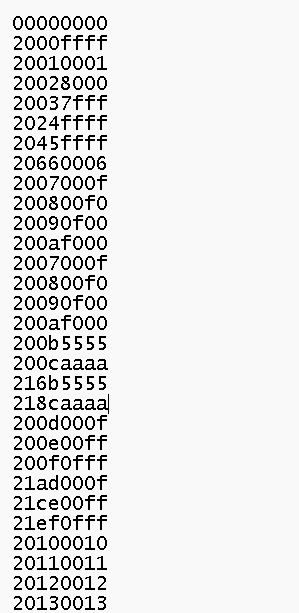
所有指令的测试文件保存在“cpu\CPUtest指令示例”这个路径下面。

以下以addi指令的测试为例，说明过程：

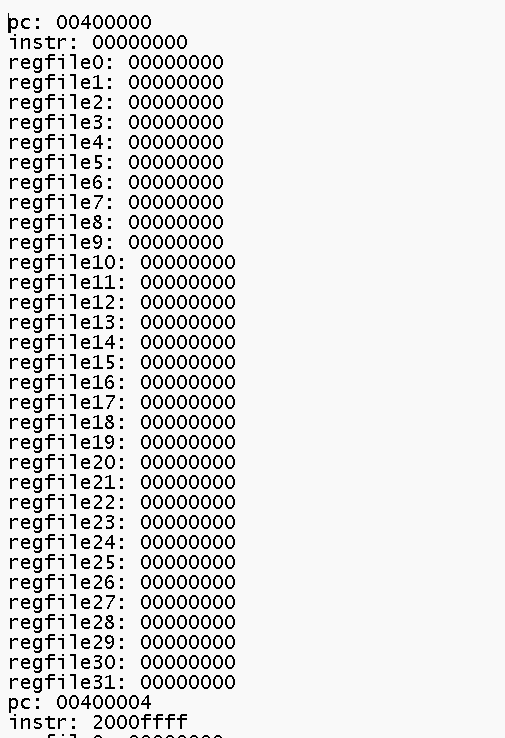
addi指令测试文件如下：



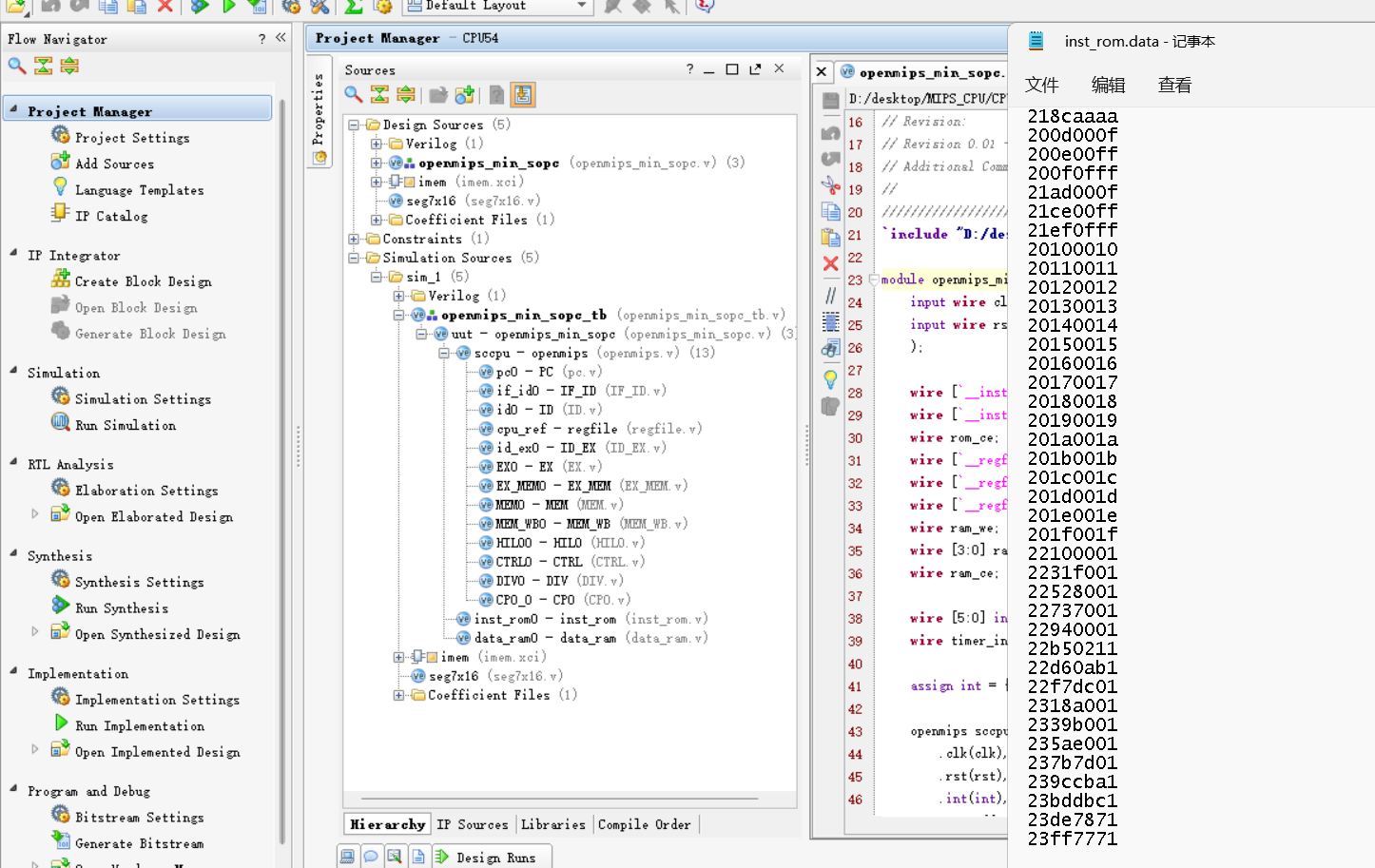
机器码为：



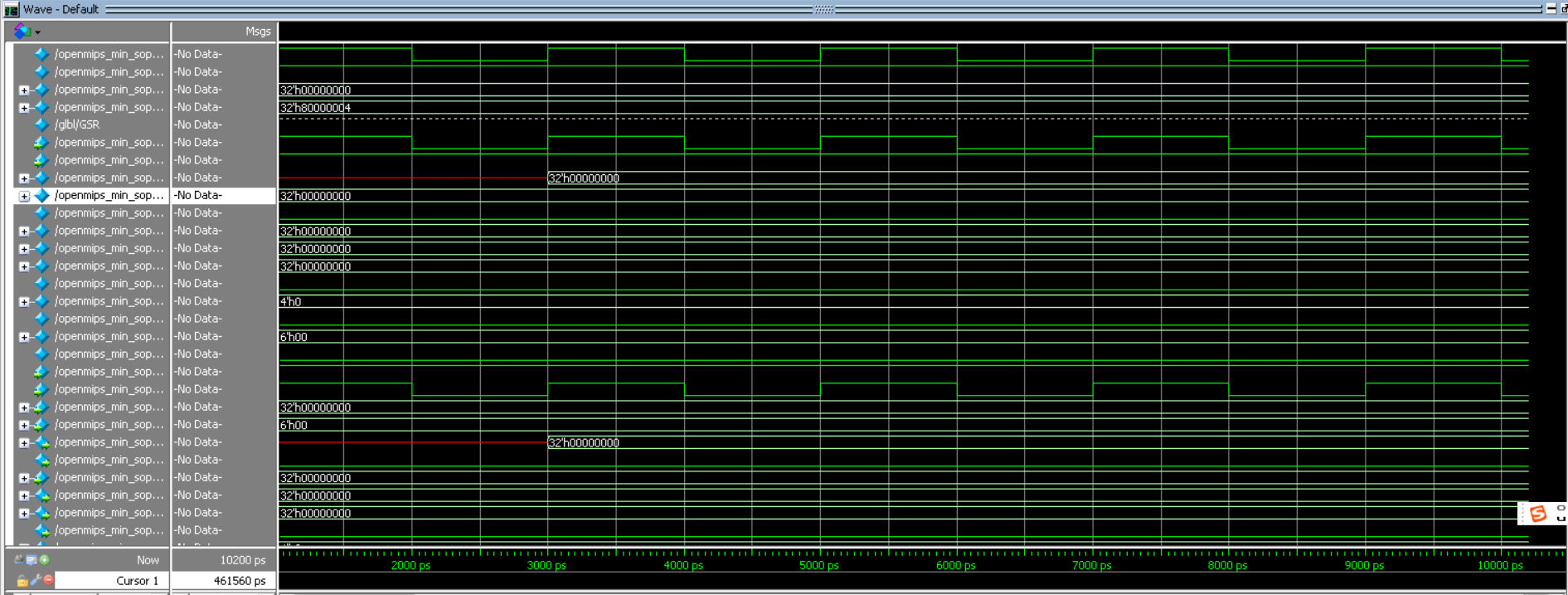
送到MARS汇编器中汇编得到的result.txt如下：



在vivado中联合modelsim运行前仿真的测试程序：



运行modelsim仿真得到结果如下（下一章会详细解释这个波形图）：



生成文件进行对比，结果如下（运行得到的最终结果完全一致，说明前仿真成功，即原CPU流水线代码没有问题，下图左侧为MARS生成的，右侧是CPU前仿真生成的）：

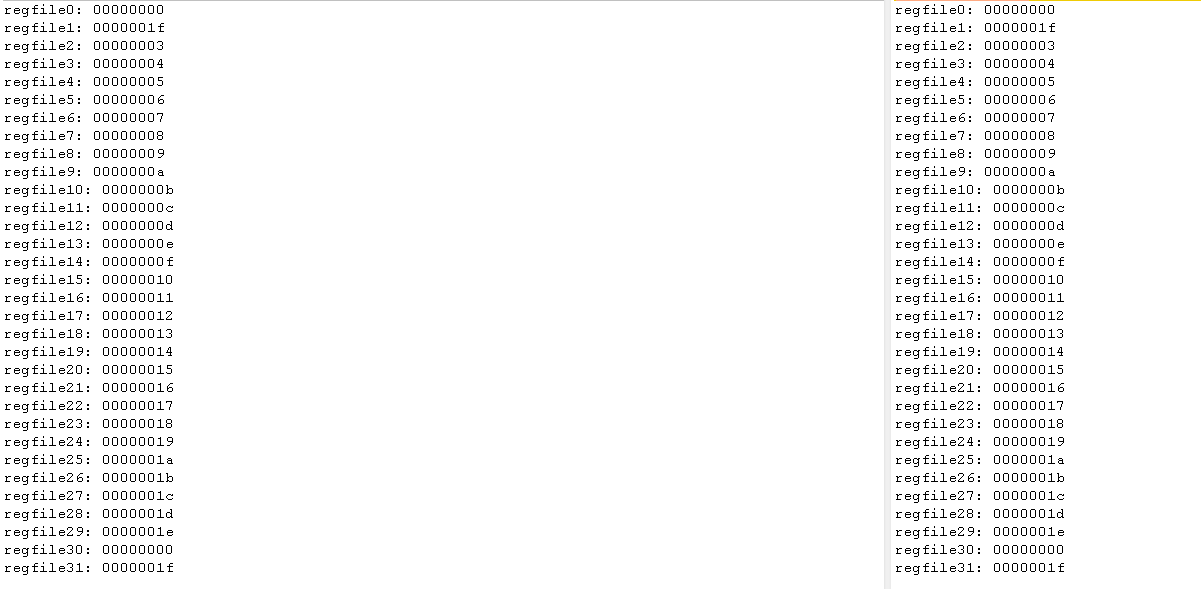


下面为其他几条指令的前仿真测试结果（篇幅限制，这里只举几条指令例子）——

beq（左侧为MARS生成的，右侧是CPU前仿真生成的） :



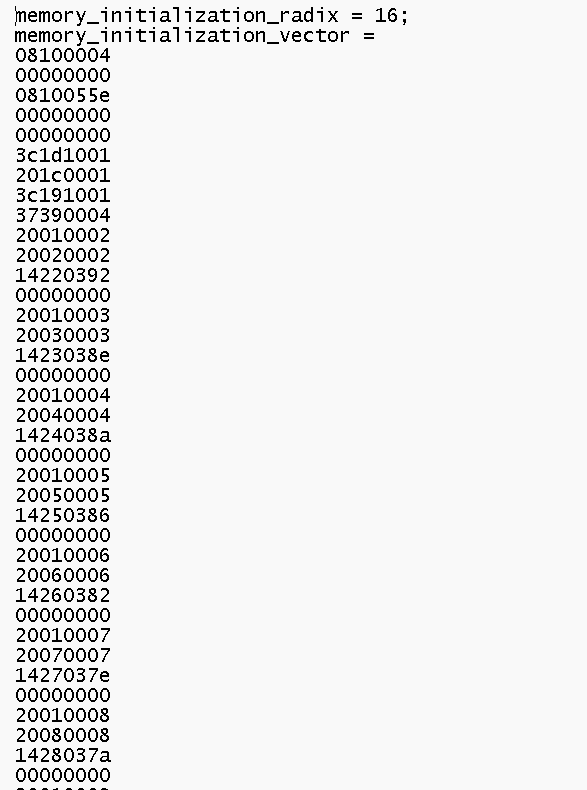
lwsw（左侧为MARS生成的，右侧是CPU前仿真生成的） :



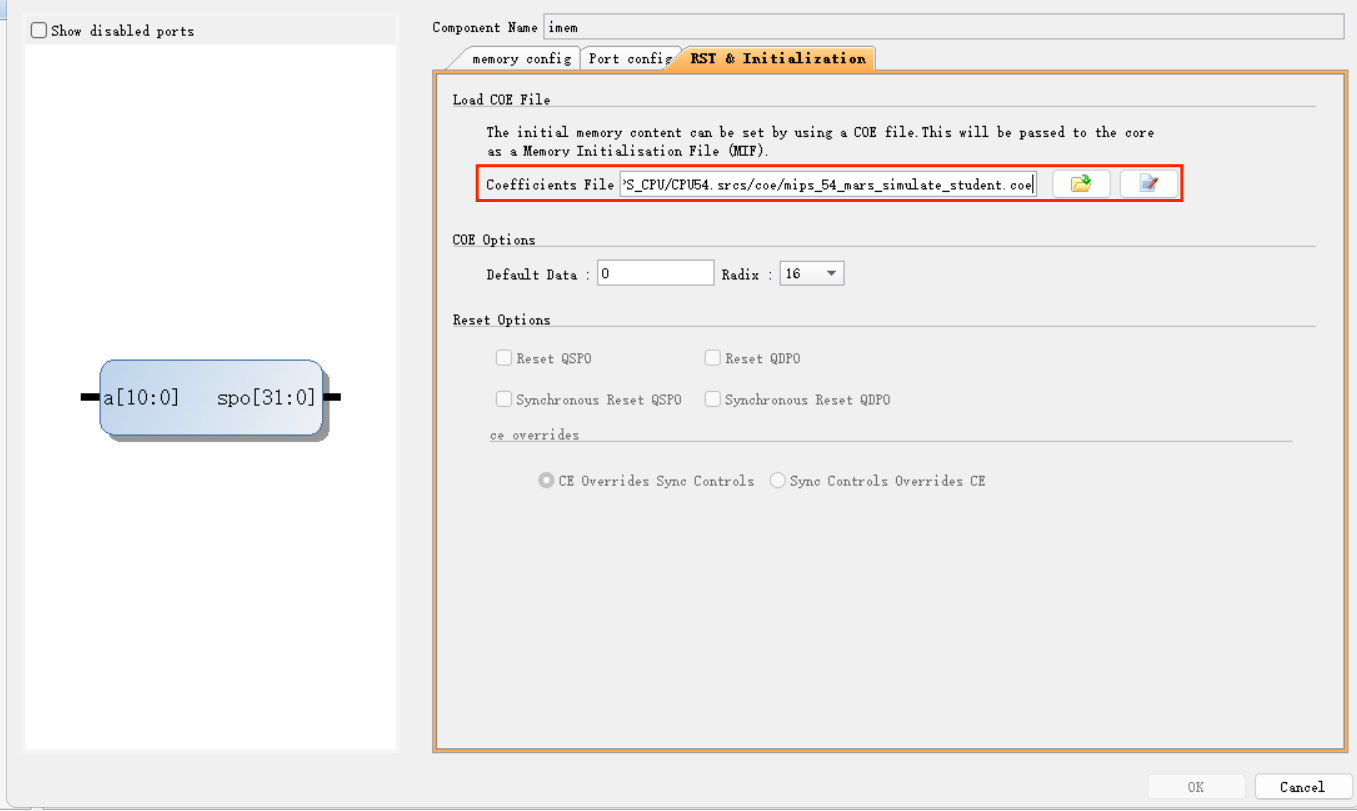
（2）后仿真

后仿真与前仿真的不同之处在于，前仿真是通过在文件中写入需要运行的指令，然后CPU启动之后把文件中的指令加载到指令存储器中，从而运行指令。后仿真需要使用vivado自带的只读存储器ip核作为指令存储器，在CPU开始运行前就把指令加载到ip核中，仿真运行得到结果。

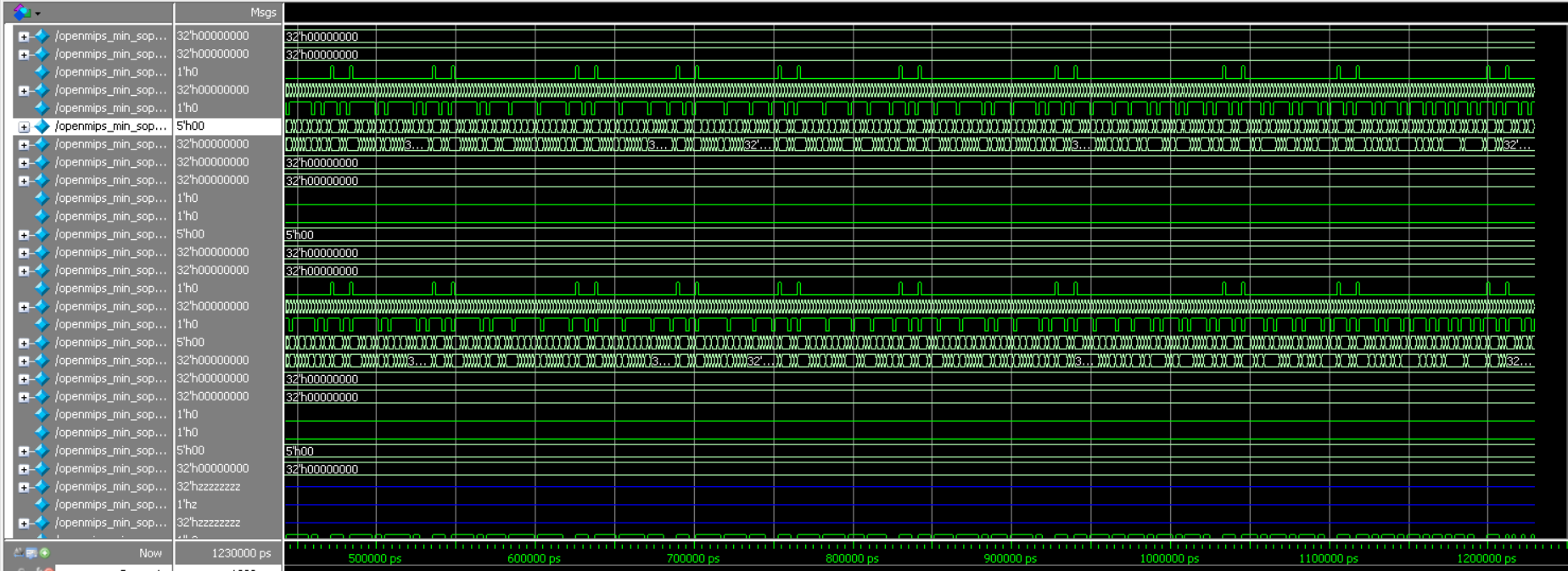
使用需要测试的指令的机器码构造出后仿真测试的coe文件（该文件保存在“cpu\mips\_54\_mars\_simulate\_student.coe”这个路径下面），如下：



ip核加载入该文件中的数据：



使用modelsim进行仿真，得到波形图如下：





由仿真结果仔细观察，发现没有时序上的毛刺，说明后仿真成功，CPU不存在时序问题。

1. 实验仿真的波形图及某时刻寄存器值的物理意义
   1. 静态流水线的波形图及某时刻寄存器值的物理意义

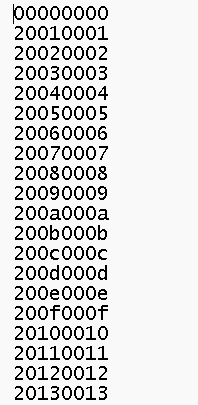
（1）前仿真波形图物理意义解释

此处简单就前仿真波形图中一些波形的物理意义做出解释，由于信号量过多，没办法一一做出详细解释，所以这里只能挑选有代表性且比较重要的信号做解释。



inst\_addr:该信号即为CPU中的PC的值，由上面的波形图可以看到，PC的值每个时钟周期增加4，因为一条指令的宽度就是32位，4字节。说明本实验实现的PC寄存器是成功的，其按照正常时序变化。

inst:该信号使CPU读取的指令，对比读取的文件中的指令序列（如下图），可以看到指令的读取和地址也是一一对应的，且每一个时钟周期都有一条指令被读取进来，说明CPU中取指实现也是正确的。



接下来解释几种特殊情况下的信号波形：



如上图为跳转指令发生时的流水线中信号的变化，可以看到的是inst\_addr信号发生的突变，从00000008变化到00000088，可见确实跳转指令（实际上00000008地址的指令是j指令无条件跳转）发挥了作用。此后读取到的信号也是跳转指令后面的信号。 

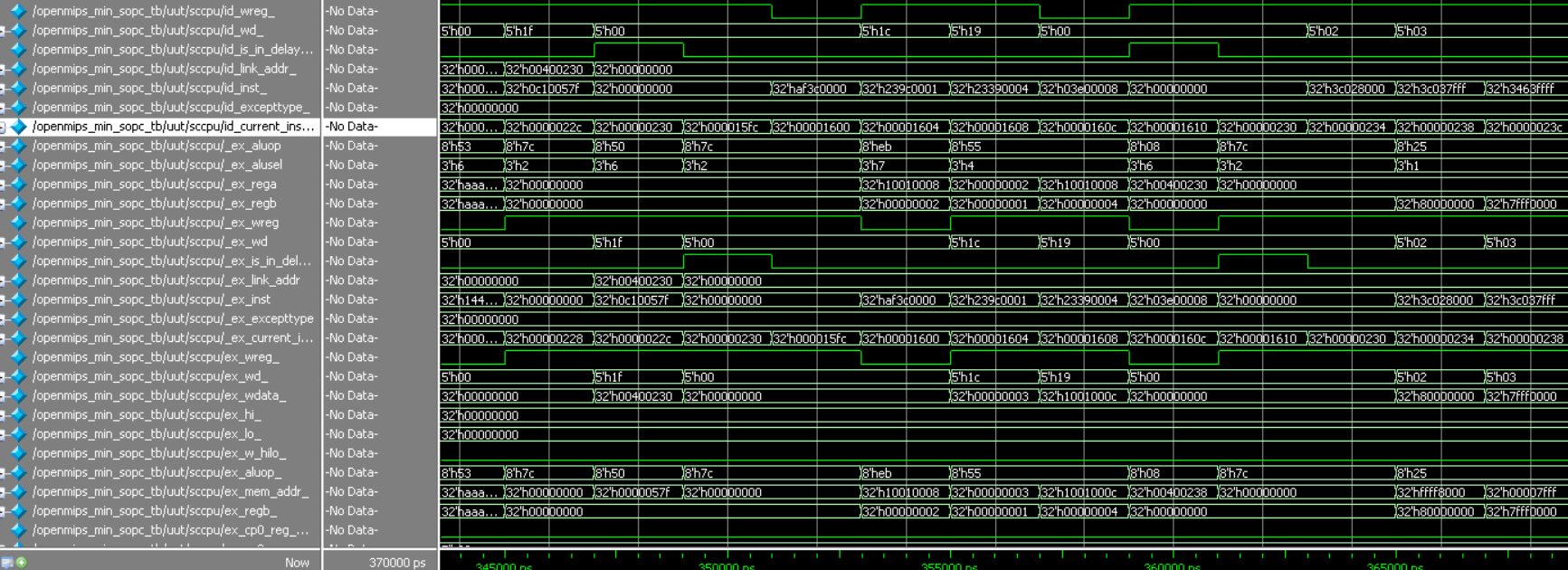
如上图为有流水线暂停发生时的信号的变化，可以看到inst\_addr信号在很长一段时间保持不变（即PC不再改变），读取到的指令inst也不变化，说明取指阶段被暂停，说明流水线暂停也成功实现。

（2）后仿真波形图物理意义解释

以下为后仿真的一个仿真波形图



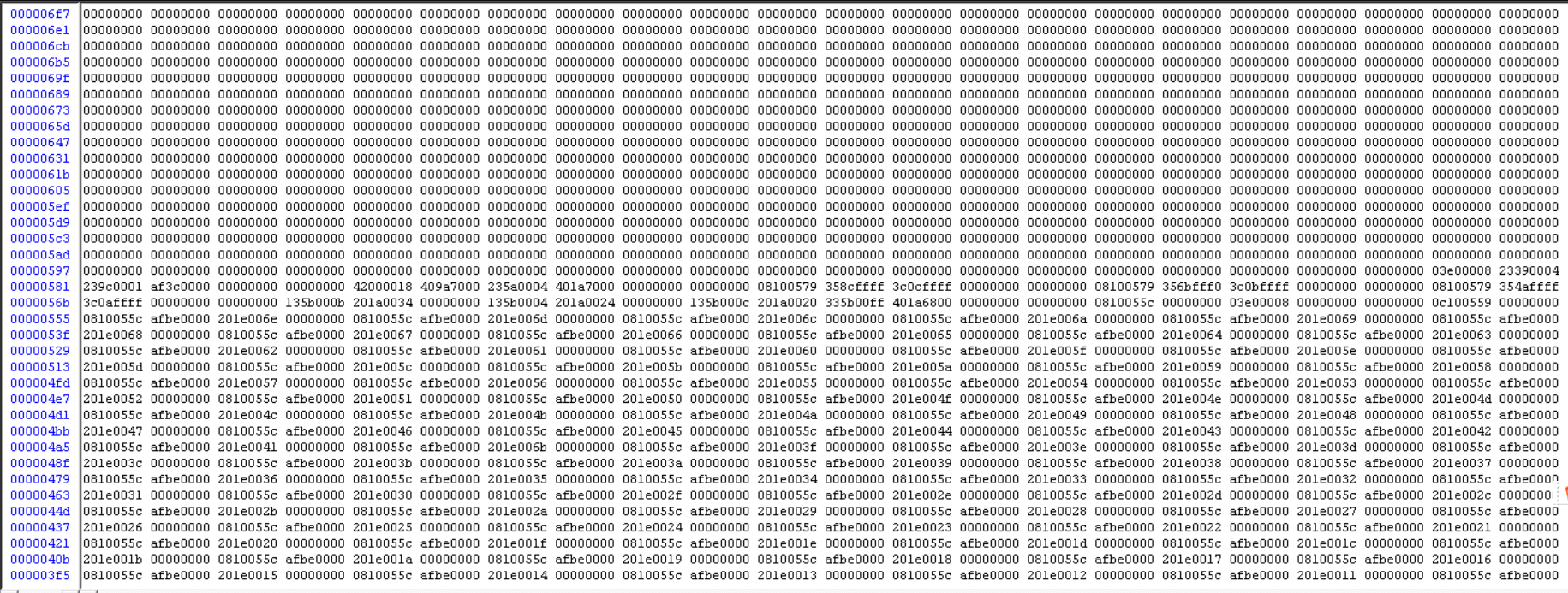
注意观察到inst\_addr的变化和inst的取指都在同一时刻，说明没有毛刺现象。



观察CPU运算中间过程的所有信号，都是随着时钟上升沿和下降沿的改变而立即改变，没有出现毛刺现象，说明后仿真是成功的，CPU没有时序问题。

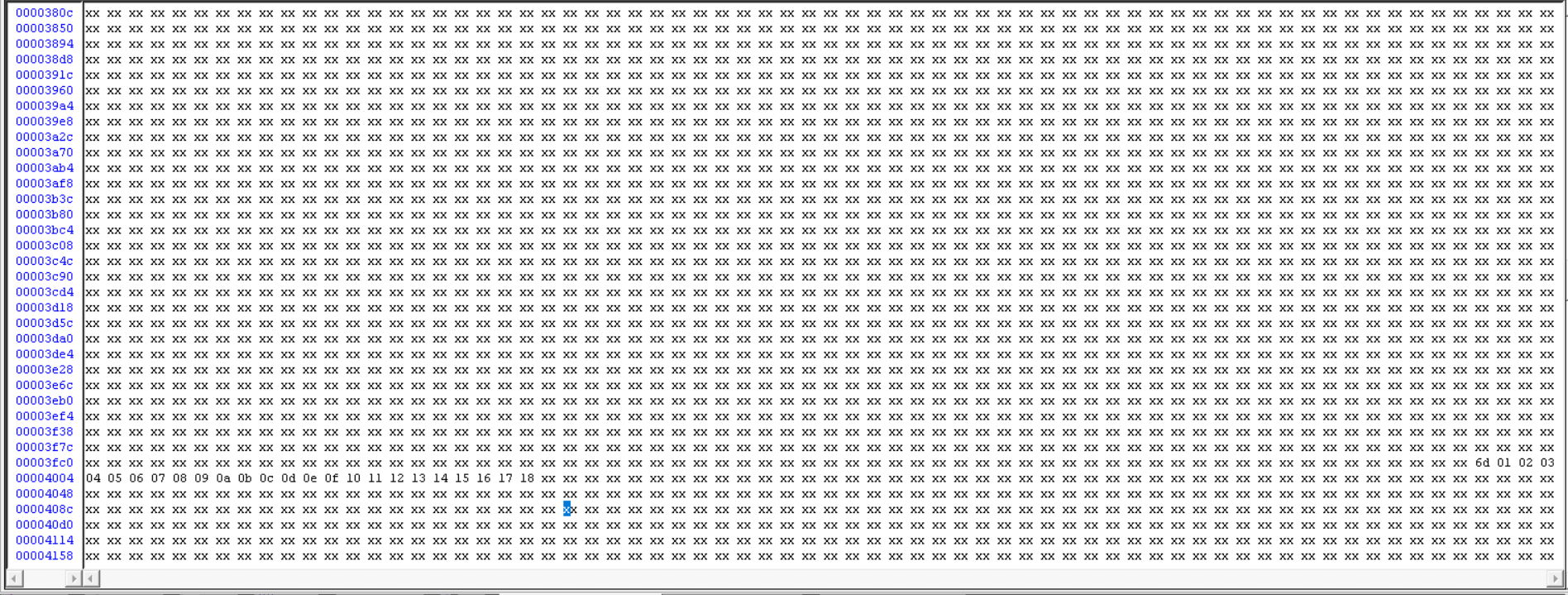
（3）某时刻寄存器的物理意义

可以通过modelsim观察仿真过程中的内存变化：



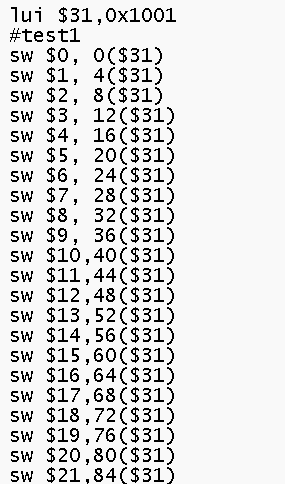
如上图为仿真过程中的指令存储器的变化，前仿真在一开始就会初始化指令存储器，后仿真在程序开始前就会加载入数据，所以指令存储器的值不会在过程中改变。指令存储器中以32位为一个单元存储，存放的是CPU要运行的指令。

以下以lwsw指令的测试为例，观察数据存储寄存器的变化。





如上图可以看到在一部分地址对应的位置已经有了存储数据，因为测试的指令集里面有存储指令（如下图）。本实验通过实现4片数据存储器进行位拓展，实现4片存储器同时存储一个字的数据（即32位分成4份存放，每份8位）。对应上图也可以看到，低位数据存放在data\_mem0，高位数据在data\_mem1等存储器中，然后下面保存的数据都很小，所以后者中数据是0（即高位为0）。同时地址也是可以对应上的，说明数据存储器的保存功能成功实现。

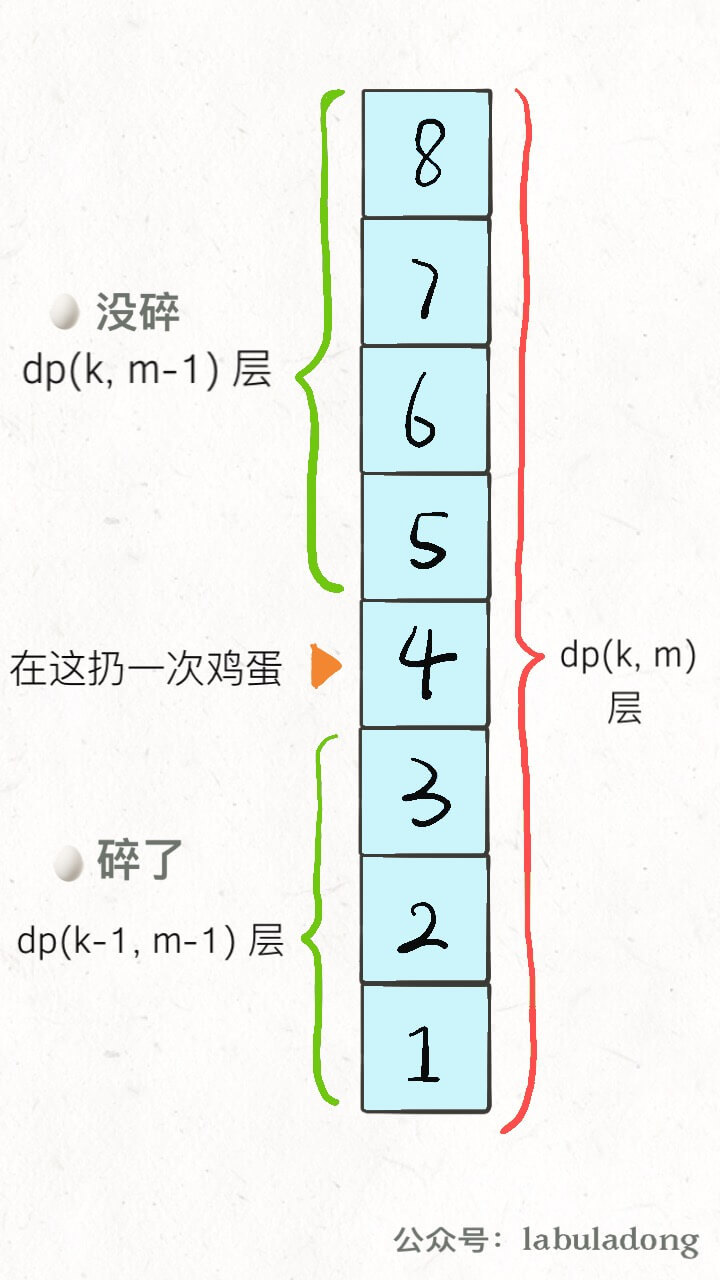


下面两张图可以看到通用寄存器详细变化过程，00400008地址的指令是20010001，即addi $1,$0,0x0001，即向1号寄存器加入1。可以看到五个时钟周期后（每个时钟周期读入一条指令，所以五个时钟周期后PC为0040001C），1号通用寄存器中就出现了计算得到的值，说明流水线CPU运行过程是正确的。经多次测试无误，下面两张图只展示其中一次测试的结果。

|  |  |
| --- | --- |
|  |  |

1. 流水线CPU实验性能验证模型

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



定义：

dp[k][m] = n

# 当前有 k 个鸡蛋，可以尝试扔 m 次鸡蛋

# 这个状态下，最坏情况下最多能确切测试一栋 n 层的楼

1、无论你在哪层楼扔鸡蛋，鸡蛋只可能摔碎或者没摔碎，碎了的话就测楼下，没碎的话就测楼上。

2、无论你上楼还是下楼，总的楼层数 = 楼上的楼层数 + 楼下的楼层数 + 1（当前这层楼）。

根据这个特点，可以写出下面的状态转移方程：

dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1

dp[k][m - 1] 就是楼上的楼层数，因为鸡蛋个数 k 不变，也就是鸡蛋没碎，扔鸡蛋次数 m 减一；

dp[k - 1][m - 1] 就是楼下的楼层数，因为鸡蛋个数 k 减一，也就是鸡蛋碎了，同时扔鸡蛋次数 m 减一。

参考：<https://zhuanlan.zhihu.com/p/92288604>

最终得到的C++代码如下：

int main()

{

    int K = 5, N = 10, broken = 5; //假设给N层楼，K个鸡蛋，在第5层楼破

    int best = 0x7fffffff; //int的最大值

    register int eggs, num;

    register bool finalFlag;

    for (int z = 0; z <= K; z++) {

        bool flag = false;//初始化为没摔破

        int dp[z + 1][N + 1] = {};

        int m = 0;

        while (dp[z][m] < N) {

            m++;

            for (int k = 1; k <= z; k++) {

                dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1;

                flag = dp[k][m] >= broken;//当前层dp[k][m]超过broken楼，最后一个鸡蛋就破了

            }

        }

        if (best > m + z) {

            best = m + z;

            eggs = z;

            num = m;

            finalFlag = flag;

        }

    }

    // printf("eggs = %d\n", eggs);

    // printf("num = %d\n", num);

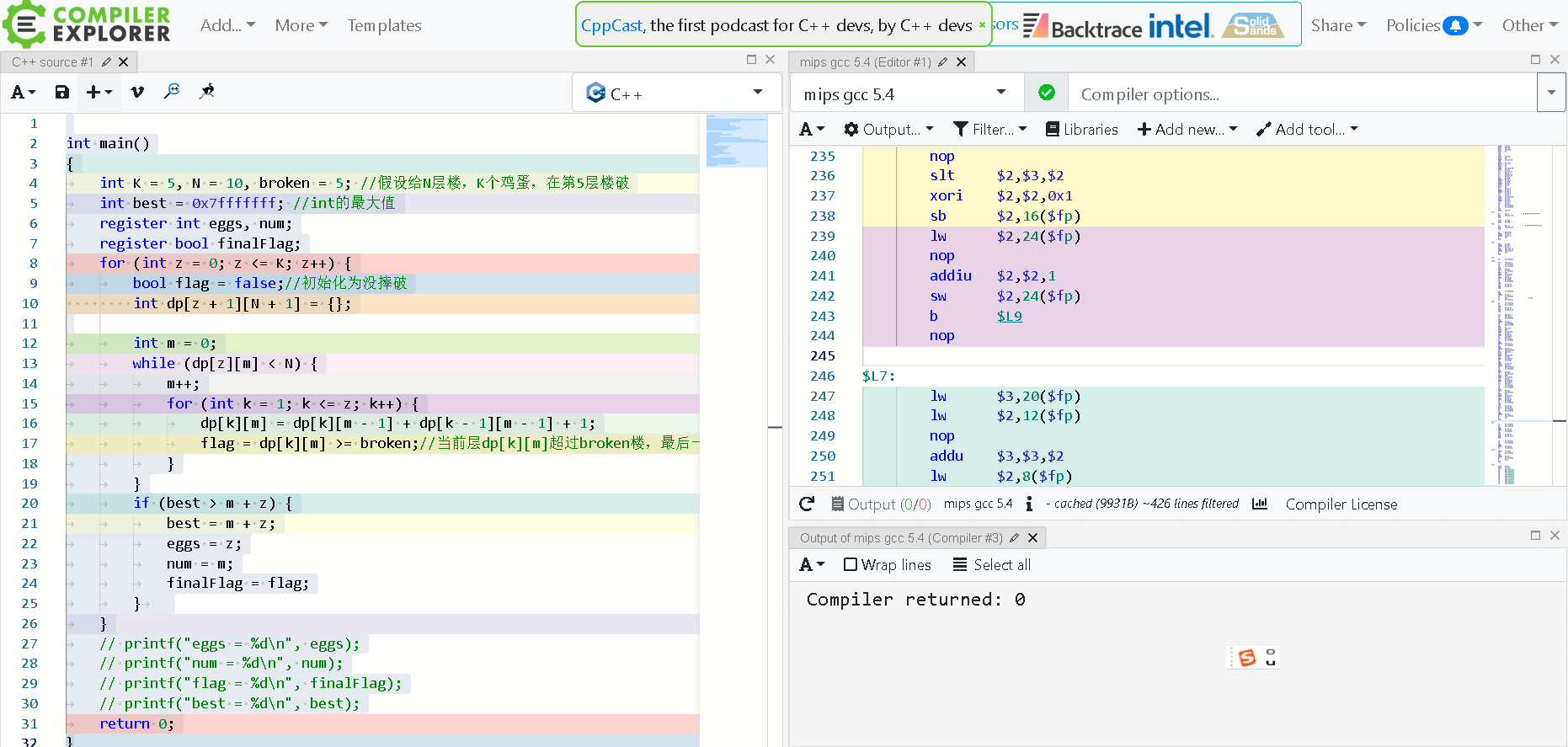
    // printf("flag = %d\n", finalFlag);

    // printf("best = %d\n", best);

    return 0;

}

使用mips gcc 5.4编译器编译为汇编语言，并稍作修改，得到的汇编语言代码如下：



main:

addiu $sp,$sp,-96

sw $fp,92($sp)

sw $23,88($sp)

sw $22,84($sp)

sw $21,80($sp)

sw $20,76($sp)

sw $19,72($sp)

sw $18,68($sp)

sw $17,64($sp)

sw $16,60($sp)

move $fp,$sp

addi $2,$zero,5 # 0x5

sw $2,28($fp)

addi $2,$zero,10 # 0xa

sw $2,32($fp)

addi $2,$zero,5 # 0x5

sw $2,36($fp)

addi $2,$zero,2147418112 # 0x7fff0000

ori $2,$2,0xffff

sw $2,8($fp)

sw $0,12($fp)

$L12:

lw $3,12($fp)

lw $2,28($fp)

nop

slt $2,$2,$3

bne $2,$0,$L2

nop

move $2,$sp

move $23,$2

sb $0,16($fp)

lw $20,32($fp)

nop

sw $20,40($fp)

move $2,$20

addiu $2,$2,1

move $13,$2

move $12,$0

srl $2,$13,27

sll $4,$12,5

or $4,$2,$4

sll $5,$13,5

move $2,$20

addiu $2,$2,1

sll $11,$2,2

lw $10,12($fp)

nop

sw $10,44($fp)

move $2,$20

addiu $2,$2,1

move $15,$2

move $14,$0

move $2,$10

addiu $2,$2,1

move $25,$2

move $24,$0

mult $14,$25

mflo $2

nop

nop

mult $24,$15

mflo $3

addu $21,$2,$3

nop

multu $15,$25

mflo $3

mfhi $2

addu $21,$21,$2

move $2,$21

srl $21,$3,27

sll $6,$2,5

or $6,$21,$6

sll $7,$3,5

move $2,$20

addiu $2,$2,1

move $17,$2

move $16,$0

move $2,$10

addiu $2,$2,1

move $19,$2

move $18,$0

mult $16,$19

mflo $2

nop

nop

mult $18,$17

mflo $3

addu $21,$2,$3

nop

multu $17,$19

mflo $3

mfhi $2

addu $21,$21,$2

move $2,$21

srl $21,$3,27

sll $8,$2,5

or $8,$21,$8

sll $9,$3,5

move $2,$20

addiu $3,$2,1

move $2,$10

addiu $2,$2,1

mult $3,$2

mflo $2

sll $2,$2,2

addiu $2,$2,3

addiu $2,$2,7

srl $2,$2,3

sll $2,$2,3

subu $sp,$sp,$2

move $2,$sp

addiu $2,$2,3

srl $2,$2,2

sll $2,$2,2

sw $2,48($fp)

lw $2,48($fp)

nop

move $21,$2

$L6:

addi $2,$zero,-1 # 0xffffffffffffffff

beq $10,$2,$L3

nop

move $3,$21

move $2,$20

$L5:

addi $22,$zero,-1 # 0xffffffffffffffff

beq $2,$22,$L4

nop

sw $0,0($3)

addiu $3,$3,4

addiu $2,$2,-1

b $L5

nop

$L4:

move $2,$20

addiu $2,$2,1

sll $2,$2,2

addu $21,$21,$2

addiu $10,$10,-1

b $L6

nop

$L3:

sw $0,20($fp)

$L10:

srl $10,$11,2

lw $3,48($fp)

lw $2,12($fp)

nop

mult $10,$2

lw $2,20($fp)

mflo $10

addu $2,$10,$2

sll $2,$2,2

addu $2,$3,$2

lw $3,0($2)

lw $2,32($fp)

nop

slt $2,$3,$2

beq $2,$0,$L7

nop

lw $2,20($fp)

nop

addiu $2,$2,1

sw $2,20($fp)

addi $2,$zero,1 # 0x1

sw $2,24($fp)

$L9:

lw $3,24($fp)

lw $2,12($fp)

nop

slt $2,$2,$3

bne $2,$0,$L10

nop

srl $20,$11,2

srl $21,$11,2

lw $2,20($fp)

nop

addiu $2,$2,-1

lw $3,48($fp)

lw $10,24($fp)

nop

mult $21,$10

mflo $10

addu $2,$10,$2

sll $2,$2,2

addu $2,$3,$2

lw $3,0($2)

srl $22,$11,2

lw $2,24($fp)

nop

addiu $21,$2,-1

lw $2,20($fp)

nop

addiu $2,$2,-1

lw $10,48($fp)

mult $22,$21

mflo $21

addu $2,$21,$2

sll $2,$2,2

addu $2,$10,$2

lw $2,0($2)

nop

addu $2,$3,$2

addiu $3,$2,1

lw $10,48($fp)

lw $2,24($fp)

nop

mult $20,$2

lw $2,20($fp)

mflo $20

addu $2,$20,$2

sll $2,$2,2

addu $2,$10,$2

sw $3,0($2)

srl $10,$11,2

lw $3,48($fp)

lw $2,24($fp)

nop

mult $10,$2

lw $2,20($fp)

mflo $10

addu $2,$10,$2

sll $2,$2,2

addu $2,$3,$2

lw $3,0($2)

lw $2,36($fp)

nop

slt $2,$3,$2

xori $2,$2,0x1

sb $2,16($fp)

lw $2,24($fp)

nop

addiu $2,$2,1

sw $2,24($fp)

b $L9

nop

$L7:

lw $3,20($fp)

lw $2,12($fp)

nop

addu $3,$3,$2

lw $2,8($fp)

nop

slt $2,$3,$2

beq $2,$0,$L11

nop

lw $3,20($fp)

lw $2,12($fp)

nop

addu $2,$3,$2

sw $2,8($fp)

$L11:

move $sp,$23

lw $2,12($fp)

nop

addiu $2,$2,1

sw $2,12($fp)

b $L12

nop

$L2:

move $2,$0

move $sp,$fp

lw $fp,92($sp)

lw $23,88($sp)

lw $22,84($sp)

lw $21,80($sp)

lw $20,76($sp)

lw $19,72($sp)

lw $18,68($sp)

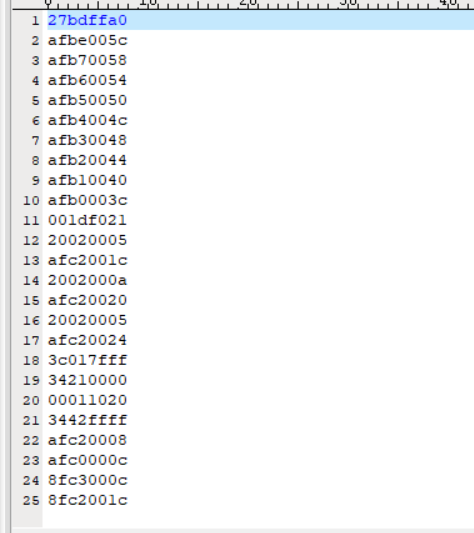
lw $17,64($sp)

lw $16,60($sp)

addiu $sp,$sp,96

nop

再生成为机器语言代码：



三份代码保存在“cpu”这一目录下面。

运行CPU程序仿真和MARS汇编得到结果如下：

30号寄存器不一样，这是因为编译生成存数的地址超过了下板的时候操作板支持的最大地址大小，所以不得不改小一点，造成取指得到xxxxxxxx。其他寄存器完全一样，说明CPU实现成功。

|  |  |
| --- | --- |
|  |  |

1. 实验验算程序下板测试过程与实现

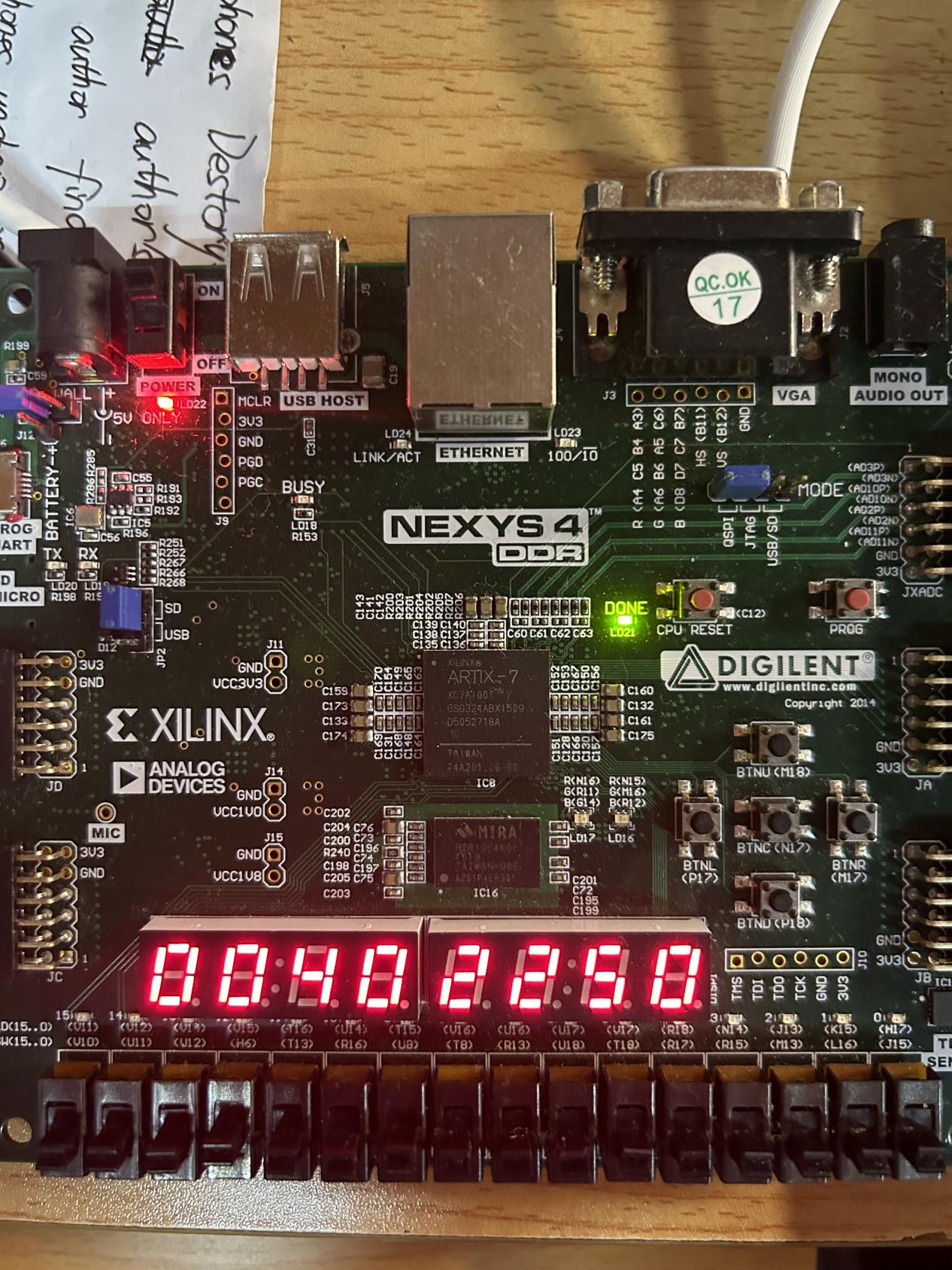
使用的下板程序（二进制机器码程序）是前面的摔鸡蛋测试程序的coe文件，该coe文件保存在“cpu”文件夹下面。

如下图为下板结果，七段数码管显示的是PC的值。

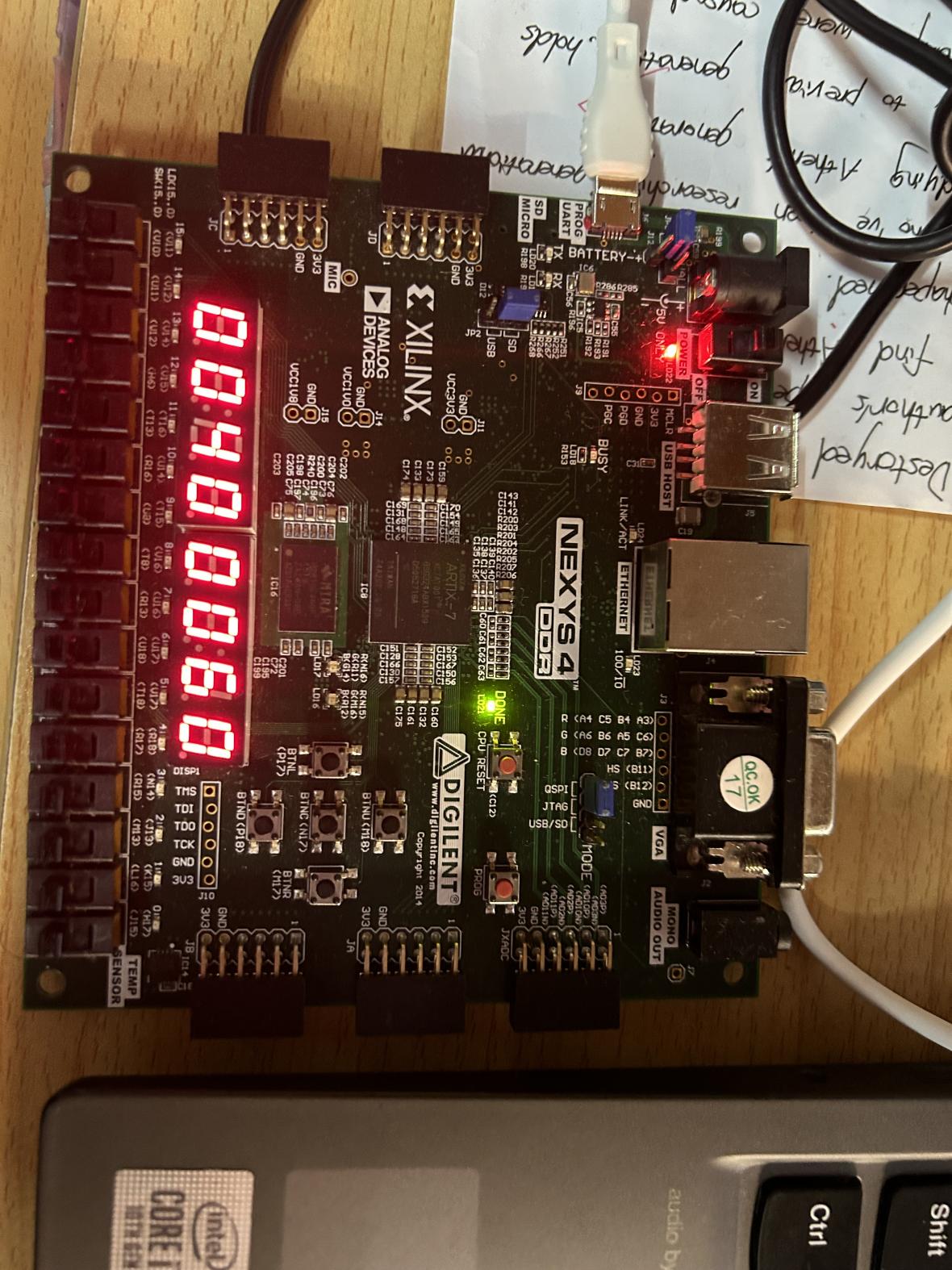
第二张图为按下重置键(reset)不放，得到的结果。

第三张图是重置之后重新开始运行的结果。

从下板结果看，实验很成功，PC的运行符合应该有的逻辑。







1. 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

分析测试使用的CPP文件，可以看出这是一个时间复杂度，空间复杂度的程序。这一程序还有改进的余地，但是本实验目的不是去优化摔鸡蛋的算法，而是用这个程序测试CPU的运行效率。

以下对CPU的性能指标进行定性的分析。

首先是吞吐率。由于使用的是流水线CPU，所以其运行效率一定高于一般的多周期CPU，从这个角度看，吞吐率要稍好一点。但是也要看到，本实验实现的CPU主体上是静态流水线CPU（除了数据相关实现了动态的数据前推），吞吐率也比不上动态流水线并经过优化的CPU。所以吞吐率也可以改进。同理，本CPU效率也不如动态流水线CPU，但比多周期CPU好。

加速比上，由于没有实现缓存机制，本CPU的加速比一定是比较差的，这可以通过以后实现缓存机制来优化。

冲突分析上，数据相关被很好的解决了，使用的是数据前推方法。但是控制相关、load相关等问题本质上都是用流水线暂停的方法来解决，所以这些冲突的解决方案还有待优化。

CPU运行时间上，这是和效率反相关的。本CPU效率比不过动态流水线CPU，但是好过多周期CPU。所以对同一份程序，运行时间比多周期CPU短，比流水线CPU长。

存储空间使用上，理论上用的是32位mips的架构，即能用空间应该是2的32次方bit这么大。但是实际没用这么大，因为本实验使用的板卡内置的空间没有这么大，限制于硬件配置，本CPU存储器空间利用上也有不足。而且，就算是对CPU能用到的存储器空间来说，因为一个程序能用到空间有限，所以也存在大量空间浪费，这些都可以改进。

1. 总结与体会

在实现流水线MIPS架构CPU的过程中，遇到了很多困难，也一一解决了，得出了下面一些主要的体会。

（1）流水线CPU的实现主要逻辑是把一条指令的执行过程划分为5个阶段，在这五个阶段中间设置连接模块。5个阶段实际处理模块是组合逻辑的，这样不会出现时序问题，并且可以迅速正确的完成对应阶段的任务。但是阶段间的连接模块却是时序逻辑的，这样就可以实现流水线并行的过程。即实际处理模块组合逻辑不受时钟影响可以继续运行，中间连接模块遵从时序控制保证指令的运行正确。

（2）静态的流水线CPU在性能上做了许多妥协。比如转移指令为避免控制相关必须在延迟槽里面放入nop指令，比如load相关只能流水线暂停。所以本次实验实现的CPU是初版的，难以满足高性能的需求。但是确实满足了正确完成任务的需求。所以，还需要对当前CPU架构多做改进。

github地址：https://github.com/Ly403/MIPS\_CPU