南京農業大學

本科生课程设计

设计报告

题　　目: 基于仿真的32位虚拟计算机设计与实现

课　　程: 计算机组成原理与系统结构课程设计

姓　　名: 陈 扬

班　　级: 网络工程161班

学　　号: 19316117

指导教师: 赵 力

2018 年 10 月 7 日

南京农业大学信息科技学院

目录

基于仿真的 32 位虚拟计算机设计与实现

网络工程 161 班　陈扬

摘要：本文针对计算机组成原理与系统结构课程的学习要求，为进一步研究学习不同指令在 CPU 中的执行情况及流水线过程、尝试利用不同的方案解决流水线阻滞，基于 C/C++语言，编程设计实现了 32 位的虚拟计算机。该虚拟计算机使用 MIPS32 指令集，能够模拟运行大部分 MIPS32 指令、展示指令在计算机中的流水线过程、利用暂停执行的方法解决数据冲突。最后，为了能够更生动形象地展示流水线过程，本文利用计算机图形库 graphics.h，编程实现了流水线生成程序，配合系统的使用，完成了设计任务书的所有目标。

关键词：虚拟计算机、MIPS32 指令集、流水线、C/C++

1 项目简介

项目介绍：利用 C/C++语言开发基于 MIPS32 指令系统的虚拟计算机。要求：

（1）完成 MIPS32 指令的取指、译码、计算、访存和写回五个步骤的软件模拟
（2）能够向系统输入机器语言源程序
（3）能够对内部寄存器进行初始花
（4）能够运行程序
（5）能够查看运行结果，能够反映指令的执行过程
（6）能够模拟五段流水线的执行过程
（7）能够解决数据相关的问题
（8）能够反映流水线的执行过程

开发语言：C/C++

开发环境：Windows10 1803、Visual Studio 2017

2 功能介绍

2.1 概述

为了满足“项目简介”中所提出的程序设计要求，故需要对程序的功能有明确的划分，让使用者能够很快上手并熟练的使用该程序。在本节中，笔者将对程序的各个功能模块做简单的介绍，以帮助使用者能够较快的知晓该程序的模块划分、各个模块的功能以及各个模块的使用方法。

本程序共分 8 个功能模块，分别为：1-指令控制、2-指令执行、3-内存控制、4-寄存器控制、5-流水线、6-使用说明、7-关于、8-退出。各个功能模块有其不同的功能及作用，读者从功能模块的名字大概可以猜测。每一个功能模块内部还有不同的小模块，具体的介绍将在下文详细展开描述。

为了能够更形象的展示流水线的过程，笔者还单独使用 C 语言编制了一个基于 graphics 图形库的程序，将控制台输出的信息以图形化方式呈现。此功能模块也将在下文做具体的介绍。

2.2 功能描述

2.2.1 指令控制

本功能模块负责指令的输入输出。程序在运行之前，必须有已经输入的指令序列，

否则将不能够正常执行而直接退出。如果用户想要正常的使用该程序，则必须将已经准备好的指令序列输入。因为该虚拟计算机基于 MIPS32 指令集，所以每一条指令的长度固定为 32 位长度的 0/1 序列。

在输入时，用户可以选择单行或多行输入，但是在输入完毕后，必须另起一行，并以”##”作为结束标识，否则程序将不能够正确读取并译码。

在该模块下，有 4 个子模块，提供更加准确的指令控制，分别为：输入指令、查看指令、修改指令、清空指令，详细介绍如下：

（1）输入指令：提供单行或多行输入指令的功能，输入的指令将顺序存储在内存的 00H-FFH 区域。每一条 32 位的指令占用 4 个内存空间。

（2）查看指令：从内存的 00H 开始读取，直至 FFH，若读取的内存区域不为 00H 内容，则判断有指令。程序调用“译码”功能，将 32 位的指令译码为能够阅读的汇编代码。

（3）修改指令：用户指定一个需要修改的指令地址，并给出要修改的指令。程序将根据指令地址与指令内容修改。注意，如果提供的指令地址与指令内容非法，程序将拒绝执行，要求用户重新提供正确的地址与内容。

（4）清空指令：清空程序内存区域中指令区域的所有指令内容，即将内存 00H-FFH 区域全部强制填为 00H。

2.2.2 指令执行

本功能模块负责指令的执行控制。当用户输入指令后，便可以使用该功能模块模拟执行指令。注意，此功能模块所提供的指令执行是串行执行方法，与“项目简介”中所描述的“流水线”功能无关。

该功能模块下可细分为 4 个子模块，分别为：全部执行、单步执行、执行记录、清除记录，详细介绍如下：

（1）全部执行：一步执行完（内存指令区域的）全部指令，当执行完后，程序会列出所执行过的全部指令内容[1]。

（2）单步执行：每次执行一条（内存指令区域）指令，当执行完后，程序会列出所执行过的指令的指令内容。

在“全部执行”与“单步执行”过程中，程序每次执行一条指令，都会向”record.txt”文件中写入一条执行记录，写入方式为追加写入，每次执行都会新起一行，不会修改之前写入过的记录。执行记录详细记录了指令执行的各项细节，包括：执行时间、指令内容、指令解释、涉及到的寄存器。用户可以在程序目录下的”record.txt”文件中查看具体的细节。

（3）执行记录：查看存储在程序同目录下”record.txt”文件中的内容，即所有的指令执行记录。

（4）清除记录：将存储在程序同目录下”record.txt”文件中的内容全部清空。

2.2.3 内存控制

本功能模块负责内存内容的添加修改。模拟计算机提供了内存模块，来存储与程序运行相关的数据。内存模块具体分为指令区域与数据区域，指令区域的地址范围为：00H-FFH，数据区域的地址范围为：100H-2FFH。对指令操作时，只影响指令区域，不影响数据区域；对数据操作时，只影响数据区域，不影响指令区域。

本功能模块下可细分为 4 个子模块，分别为：初始化内存（全 0）、初始化内存（指定值）、修改内存、查看内存，详细介绍如下：

（1）初始化内存（全 0）：将内存数据区域 100H-2FFH 全部修改为 00H。

（2）初始化内存（指定值）：用户给出需要修改的指定值，程序将数据区域 100H-

[1] 具体有：执行时间、指令内容、指令解释、涉及到的寄存器信息

2FFH 全部修改为用户给出的指定值。

（3）修改内存：该功能可以指定修改某一特定地址的内存值为某指定值。程序将会要求用户给出需要修改的内存地址与要修改的内存的值并做出修改。注意，若用户给出的内存地址与数据非法，程序将会拒绝执行，并要求用户重新给出可用的数据。

（4）查看内存：程序按照内存地址顺序给出每一个地址的内容。在显示数据时，将先按照指令区域、数据区域进行分类，然后，以每行 10 个展示。

2.2.4 寄存器控制

本功能模块负责寄存器内容的添加修改。模拟计算机提供了寄存器模块，来存储程序运行所需要的各个寄存器数据与寄存器本身。根据 MIPS32 指令集的设计，程序中共有 32 个不同的寄存器[2]，对应不同的功能。

本功能模块下可细分为 4 个子模块，分别为：初始化寄存器（全 0）、初始化寄存器（指定值）、修改寄存器、查看寄存器，详细介绍如下：

（1）初始化寄存器（全 0）：将 32 个寄存器全部修改为 00H。

（2）初始化寄存器（指定值）：用户给出需要修改的指定值，程序将 32 个寄存器统一修改为用户给出的指定值。

（3）修改寄存器：该功能可以指定修改某一特定的寄存器值。程序将会要求用户给出需要修改的寄存器序号与要修改的寄存器值并做出修改。注意，若用户给出的寄存器序号与数据非法，程序将会拒绝执行，并要求用户重新给出可用的数据。

（4）查看寄存器：程序读取 32 个寄存器的值，并将他们的数据以 16 进制的方式显示出来，供使用者查阅。

2.2.5 流水线

本功能模块负责将程序中已有的指令序列使用流水线的方式展示并运行。该功能模块是整个程序的核心，也是整个程序设计的精华所在，笔者设计的程序架构、模块划分及整合以及独特的算法思想，将在这一模块中集中体现。

为了能够区别模拟计算机中“流水线”功能的独特与其优越性，故单独设计这一功能模块并将它和其他模块区别是有必要的。

该功能模块为用户详细了解程序内部流水线的运行机制提供了方法，利用图形化的方法[3]展示了流水线五个步骤[4]的详细划分。

本功能模块下可细分为 3 个功能模块，分别为：生成流水线、单步执行、清空流水线，详细介绍如下：

（1）生成流水线：程序根据当前内存指令区域的指令，依照“流水线算法”（该算法在后文将会详细介绍），生成程序的流水线过程。由于该模拟计算机对于流水线阻滞的处理方法为暂停执行，故采用了图示的方式进行展示。注意，使用该功能，只会生成流水线图，但程序并不会真正执行指令，所以使用完该功能后，程序的 PC 指针位置、内存数据、寄存器数据都不会改变。同时，生成的流水线图将存储在程序目录下的”pipeline.txt”和”graphics.txt”文件中。其中，”pipeline.txt”文件为记录文件，供读者查阅使用，”graphics.txt”文件为图像生成缓存，供“流水线绘图.exe”调用绘图。

（2）单步执行：程序根据当前内存指令区域的指令，依照“流水线算法”（该算法在后文将会详细介绍），单步生成程序的流水线过程。根据流水线时间划分的原则，使用单步执行也将逐个时间点展示具体的操作过程。每执行一步，程序将会暂停，展示流水线当前的推进情况、所有寄存器中的数据、每一条指令当前所处的状态，同时，PC 指针、内存、寄存器中的值都会根据实际的情况改变。用户可以选择是否继续单步执行，

[2] 详见《MIPS R2000 Assembler Language》文档

[3] 该图形化方法基于 graphics.h 图形库

[4] 指 取值(Fetch)、译码(Decode)、执行(Execute)、访存(Memory)、写回(WriteBack)五个步骤

若选择返回菜单，则流水线的推进过程直接结束。单步执行的记录存储于程序目录下的”pipeline_step.txt”文件中，同时，也会生成一份图像生成缓存，存储在”graphics.txt”文件中，供“流水线绘图.exe”调用绘图。

（3）清空流水线：程序将位于程序目录下的”pipeline.txt”、”pipeline_step.txt”与”graphics.txt”文件全部清空。

2.2.6 使用说明

本功能模块具体介绍了程序的使用说明与注意事项。该功能模块中所提供的使用说明与注意事项与本文档的第二部分“功能描述”完全一致，若读者已经熟读并记住了本节内容，则不必再重复查看程序的这一部分。

2.2.7 关于

本功能模块介绍了程序的作者信息。具体的信息有：项目名称、作者、学号、班级。

2.2.8 退出

本功能模块负责对程序所使用的内存空间进行释放，并安全的结束程序运行，退出程序。

2.2.9 流水线绘图

“流水线绘图”程序是一个单独的、使用 C 语言编写的、基于 graphics 图形库的程序，目的是将图像生成缓存生成为具体的流水线图形图像。对于“流水线绘图”程序的设计方法将在后文作具体的介绍。该程序没有具体的使用说明，只需要执行即可。注意，要使得改程序能够正常执行，请保证在该程序目录下有”graphics.txt”文档以供读取，同时，也要保证”graphics.txt”的格式正确。

3 设计介绍

3.1 设计思路

对于系统的设计，首先需要明确该系统所需要的框架结构。为了能够准确反映 MIPS32 指令集以及展示流水线的功能特性，故需要设计一个较大的、可变的、能够随时扩展的框架。

根据流水线的五个阶段特征，故先将系统结构简单分为取值、译码、执行、访存、写回五个模块，对应于 fetch、decoding、execute、register、memory 五个头文件及对应的 cpp 文件。同时，为了扩展程序的功能，使得有更多的可变细节，故继续扩展模块。

添加记录 log 模块，以对程序串行执行指令做出记录。记录 log 模块提供了多个对记录操作的函数接口，以方便在“译码”、“执行”、“访存”、“写回”阶段在 record.txt 文件中写入相关的记录。

添加 overalldata 模块，以对整个程序所需要使用的关键数据进行声明与赋值。其他模块只需要使用 include 方法将此头文件包含，即可共享全局唯一的程序数据。在此模块中，笔者使用了 C++ 11 的新特性，使用了 constexpr int 代替了 const 语句与 define 语句。常量表达式 const 语句的值不会改变且在编译过程中就能够得到计算结果；C++11 新标准规定，允许将变量声明为 constexpr 类型以便由编译器来验证变量的值是否是一个常量表达式。define 宏是预处理命令，即在预编译阶段进行字节替换，在内存中会存在有多份内容的拷贝，运行效率远没有使用其他两个命令来得高。因此，适当的使用 constexpr 命令能够加快程序的运行。

添加 panel 模块，专门用来管理程序的控制台菜单显示。该模块提供了一个完整的菜单控制方法，即使用者通过数字键盘来控制控制台菜单的切换及功能的选择执行。

添加 pipeline 模块，用来添加流水线功能。流水线功能模块是整个程序的核心功能模块，提供了一系列有关于流水线的功能，具体的功能已经在上一节叙述。

添加控制 control 模块，以控制全部其他模块的协调运行。根据 C++的特性，程序使

用面向对象的设计方法，每一个模块为一个对象。全部的对象在控制 control 模块中创建一个唯一的实例。在控制 control 模块中编写不同的函数，以对不同的实例及其接口进行调用，从而完成整个程序的有序运行。可以说，控制模块是整个程序各个模块之间的连接，功能至关重要。

另外，为了能够生动形象的展示流水线的运行情况，使用不同颜色方块来表示五个不同的阶段，故使用 C 语言基于 graphics 图形库另外编写了一个流水线生成程序。该程序读取程序目录下的”record.txt”文档，即可自动生成图形图像。

系统的结构如图 3.1-1 所示。

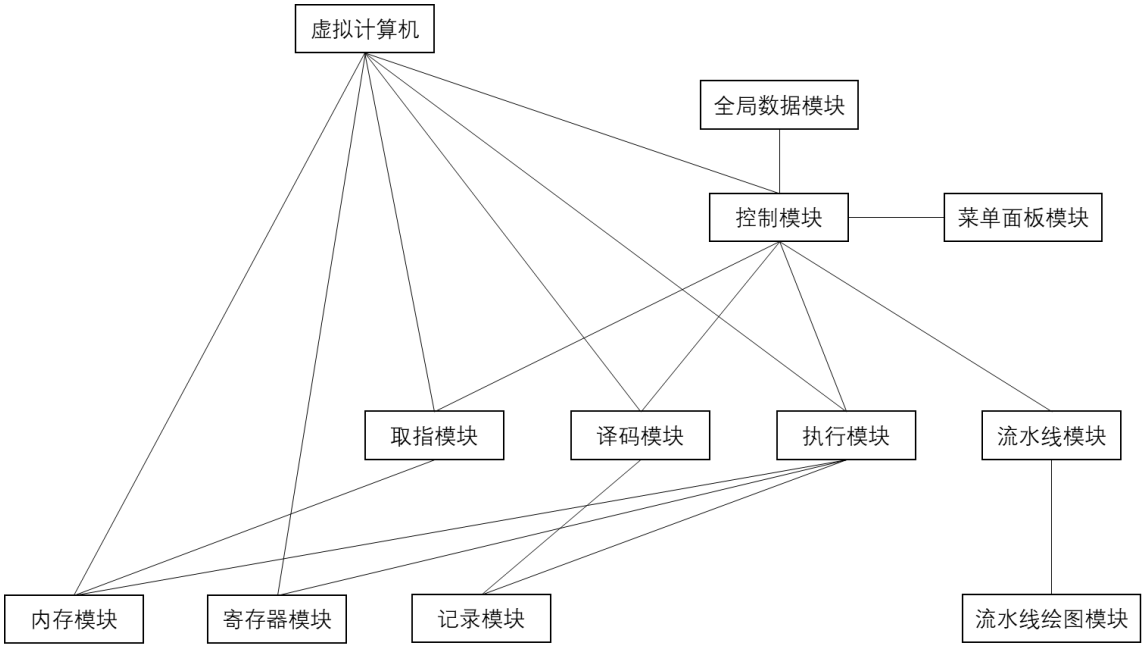


图 3.1-1

3.2 数据结构设计

程序不同模块的设计，数据结构至关重要。本节将具体描述程序的各个功能模块所使用的数据结构及其运行机制。有些功能模块中不包含数据结构，故直接省略。

记录 log 模块使用了一个二维数组 regname[REGISTER_NUM][REGISTER_MAX_NAME_SIZE]来记录 32 个寄存器的名称。程序直接访问寄存器数组，给出具体的寄存器序号，即可获取寄存器名字。

内存 memory 模块使用了一个一维数组 mem[MEMORY_SIZE]来存储内存，该数组使用字符类型 char。在 win32 系统中，一个 char 类型对应于 8 个 bit 位，能够用 00H-FFH 来表示。而 char 类型与 int、long、float、double 类型之间能够进行强制类型转换，故使用 char 类型能够很好地满足内存的数据存储需要。

寄存器 register 模块使用了一个一维数组 reg[REGISTER_NUM]来存储 32 个寄存器的值，该数组使用整形数据类型 int。模拟计算机的字长为 32 位，故一次能够处理的最长数据类型为 32 位，正好对应于 32 位的整形 int 数据字长。

全局数据 overalldata 模块使用了大量的 constexpr int 类型的数据声明了程序所需要使用的关键数据。其中，每一条指令能够译码出的部分为：rs、rt、rd、shamt、immediate、address，这些数据部分在程序运行时往往需要同时进行传递。因此，使用结构体数据类型将他们全部包含在内，即声明 struct InstructionStruct 结构体类型，里面定义 char 类型的 rs、rt、rd、shamt，short 类型的 immediate、address。若有指令不需要某些数据，则将其赋值为一个特殊的值，和其他值区分开来。

流水线 pipeline 模块为程序的核心模块，其数据结构的设计同时体现出流水线算法

的核心思想。此处只介绍流水线的数据结构设计，详细的算法设计在下文将做详细的说明。在该模块中，使用了 5 个数组来存储不同的数据。使用二维数组 pipeline_coordinates[PIPELINE_X_NUM][PIPELINE_Y_NUM]来表示流水线的二维 XY 坐标系；使用一维数组 register_time_state[PIPELINE_Y_NUM]来表示每个时刻的寄存器状态；使用一维数组 function_state[PIPELINE_Y_NUM]来表示每个时刻的五个部件的状态；使用一维数组 instruction_start[PIPELINE_X_NUM]来表示每一条指令的开始位置；使用一维数组 if_write[PIPELINE_X_NUM]来表示二维 XY 坐标系中每一行是否被用。

3.3 功能模块设计

该节将会详细介绍程序设计中的每一个功能模块的详细设计方法，用户可以参考这一节的叙述，自己编写出一个相同结构的程序。各个功能模块的包含关系与调用关系在上一节已有叙述，故本节只单独地介绍每个模块的设计方法。

3.3.1 内存模块

内存模块（类）提供了程序中关于内存的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.内存数组
2.内存指令区域的长度
3.内存数据区域的长度
4.当前内存状态
5.当前指令地址

函数部分：

1.默认构造函数
2.析构函数
3.设置内存状态为忙碌
4.设置内存状态为空闲
5.返回内存当前状态
6.初始化全部内存 0
7.初始化全部内存 指定值
8.向指定内存写入数据
9.从指定内存读取数据
10.重置当前的添加指令位置
11.获取当前的添加指令位置
12.指令位置自增

3.3.2 寄存器模块

寄存器模块（类）提供了程序中关于寄存器的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.所有的 32 个寄存器
2.寄存器当前状态

函数部分：

1.默认构造函数
2.析构函数
3.设置寄存器为忙碌

4.设置寄存器为空闲
5.返回寄存器当前状态
6.初始化全部寄存器 0
7.初始化全部寄存器 指定值
8.向指定的寄存器中写入数据
9.从指定的寄存器中读取数据

3.3.3 取指模块

取指模块（类）提供了程序中关于取指令的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.PC 指针
2.当前取指令状态

函数部分：

1.默认构造函数
2.析构函数
3.设置当前取指状态为忙碌
4.设置当前取指状态为空闲
5.返回取指类当前状态
6.获取 PC 指针的值
7.PC 指针自增
8.修改 PC 指针的值
9.重置 PC 指针的值
10.从内存的指定区域取指令

3.3.4 译码模块

译码模块（类）提供了程序中关于译码的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.当前译码状态

函数部分：

1.默认构造函数
2.析构函数
3.设置当前译码状态为忙碌
4.设置当前译码状态为空闲
5.返回取指类当前状态
6.提取从 m 位到 n 位的值
7.对指令进行译码
8.检测该指令是否为写入$zero0 寄存器的指令

3.3.5 执行模块

执行模块（类）提供了程序中关于执行的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.当前执行的状态

2.当前指令数量

函数部分：

1.默认构造函数

2.析构函数

3.设置当前执行的状态为忙碌

4.设置当前执行的状态为空闲

5.返回取指类当前状态

6.对数据进行符号位扩展

7.对数据进行 0 扩展

8.R-Type 执行（addu、subu、and、or、xor、nor、sll、srl、jr）

9.I-Type 执行（addi、andi、ori、xori、lw、sw、beq）

10.J-Type 执行（j）

11.指令执行控制

12.指令过程预知

3.3.6 记录模块

记录模块（类）提供了程序中关于记录的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.寄存器名字

2.写记录的文件

3.判断是否第一次写文件

4.当前记录标号

函数部分：

1.默认构造函数

2.析构函数

3.初始化寄存器名字

4.获取寄存器名字

5.初始化记录标号 current_log_no

6.获取指令记录内容

7.获取指令内容

8.根据指令内容，写文件

9.开始写文件

10.结束写文件

11.删除所有记录

3.3.7 全局数据模块

全局数据模块(类)提供了程序中程序运行所必须使用的数据的获取声明定义功能。在该模块中，除了译码得到参数的结构体之外，其他的数据都使用 constexpr 类型声明。

因为数据过多，具体的声明定义将在“详细设计”中介绍，此处只做大概介绍。

1.寄存器的参数

2.内存的参数

3.取指类的参数

4.译码类的参数

5.指令中译码得到的所有参数 结构体

6.指令中的 FUNC 标识情况

7.17 条命令的对应序号
8.执行类的参数
9.控制类的参数
10.面板选择类的参数
11.流水线的参数

3.3.8 菜单面板模块

菜单面板模块（类）提供了程序中关于菜单面板选择、跳转的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

函数部分：

1.菜单选择控制
2.显示主菜单
3.指令控制菜单
4.指令执行菜单
5.内存控制菜单
6.寄存器控制菜单
7.流水线菜单
8.使用说明菜单
9.关于菜单
10.退出菜单

3.3.9 流水线模块

流水线模块（类）提供了程序中关于流水线的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.流水线的二维 XY 坐标系
2.流水线的二维 XY 坐标系，用于单步执行
3.每个时刻的寄存器状态
4.每个时刻的五个部件的状态
5.每一条指令的开始位置
6.二维 XY 坐标系中每一行是否被用

函数部分：

1.默认构造函数
2.析构函数
3.初始化全部数据
4.初始化 XY 坐标系
5.初始化每时刻的寄存器状态数组
6.初始化每时刻五个部件的状态数组
7.初始化每条指令的开始位置
8.初始化每条指令是否已被解析
9.写入 Ban 指令
10.判断指令需要读的寄存器，返回一个 int 类型的 32 位数据，对应表示 32 个寄存器
11.判断指令需要写的寄存器，返回一个 int 类型的 32 位数据，对应表示 32 个寄存器

12.对传入的数据，在特定位置写入特定的 1 / 0
13.返回某一位的值
14.返回当前二维 XY 坐标系中已经用的流水线条数
15.检查前面的指令是否已经执行完毕
16.检查某一条指令是否已经执行完毕
17.对单条指令进行解析，生成流水线
18.单步模拟执行流水线
19.判断当前时间点是否还有指令需要执行
20.将二维 XY 坐标系中的数据写入到文件中
21.将二维 XY 坐标系中的数据写入到单步执行记录文件中
22.判断是否在某一时间点存在某一动作，如果存在则返回该指令的编号，如果不存在，则返回特定值

3.3.10 控制模块

控制模块（类）提供了程序中关于全局控制、模块协调调用的控制功能。该模块中定义了相关的数据与函数，具体的名称与形式参数在“详细设计”中会做介绍，本处只介绍他们的意义与所实现的功能。

数据部分：

1.译码类的实例
2.执行类的实例
3.取指类的实例
4.记录类的实例
5.内存类的实例
6.菜单面板类的实例
7.流水线类的实例
8.寄存器类的实例
9.当前共有的指令条数

函数部分：

1.默认构造函数
2.析构函数
3.程序入口函数
4.面板控制
5.按任意键返回主菜单
6.将输入的字符串转化为指令
7.检查输入的指令是否违法
8.输入指令
9.查看指令
10.修改指令
11.清空指令
12.重置当前的所有的指令条数
13.刷新当前的所有的指令条数
14.检测$zero 寄存器的状态，并重置为 0
15.执行一条指令
16.执行全部指令
17.单步执行指令
18.检查指令执行记录

19.清除指令执行记录
20.初始化内存 全 0
21.初始化内存 指定值
22.修改内存
23.查看内存
24.初始化寄存器 全 0
25.初始化寄存器 指定值
26.修改寄存器
27.查看寄存器
28.查看流水线图
29.单步模拟执行流水线
30.查看当前 PC 指针的值
31.清空流水线
32.退出

3.3.11 流水线绘图模块

流水线绘图模块（程序）提供了流水线绘图的方法。该模块单独为一个程序，基于 graphics 图形库。

由于该程序（模块）单独使用 C 语言编写，故只有面向过程的方法，没有面向对象。使用面向过程能够很好地适应该程序的需求，根据绘图缓存文件绘制出相应的图形。

这里只介绍该程序（模块）使用的函数，具体的算法在“详细设计”中做详细说明。

数据部分：

1.指令区长度
2.流水线 X 轴长度
3.流水线 Y 轴长度
4.绘图开始时的左开始位置
5.绘图开始时的上开始位置
6.每一个绘图的方块的宽度
7.每一个绘图的方块的高度
8.各个绘图方块的颜色（取指、取指等待、译码、译码等待、执行、访存、访存等待、写回、禁止、空）

函数部分：

1.文字绘制
2.图形绘制（空、取指、取指等待、译码、译码等待、执行、执行等待、访存、访存等待、写回、禁止）
3.主函数

4 关键算法

4.1 流水线生成

每一条指令的处理，都可分为 5 个阶段：取指、译码、执行、访存、写回，每一个阶段的具体解释为:取值,根据 PC 的值,从存储器中取指令并装入到 IR 中,同时 PC+4；译码/读寄存器，译码，读寄存器堆，判断是否为转移指令，计算下条地址；地址/执行周期，对上一周期准备好的操作数进行运算；访问存储器，load 根据上一周期的有效地址进行读；store 根据上一周期的有效地址进行写；写回，将结果写入寄存器堆。

本节将详细介绍程序中流水线生成的具体算法。该算法的思想基于二维 XY 坐标系，来模拟流水线的每条指令的过程。这样设计，不仅有利于使用者阅读，也有利于使用绘

图程序生成相应的图形图像。

4.1.1 各项数据的意义

该算法使用了多种不同数据类型的一维与二维数组。

定义了 char 类型的数组 pipeline_coordinates[PIPELINE_X_NUM][PIPELINE_Y_NUM]，该二维数组用来表示流水线的二维 XY 坐标系。其中 PIPELINE_X_NUM 为坐标系的纵轴最大长度，其值在 overalldata 中有定义，也可理解为程序最大支持的指令条数；PIPELINE_Y_NUM 为坐标系的横轴最大长度，其值在 overalldata 中有定义，也可理解为程序最大支持的时间点个数。该二维数组中，每一个元素在流水线生成时都将被填入一个字符，X 代表未定，F 代表译码、D 代表译码、E 代表执行、M 代表访存、W 代表写回、B 代表禁止执行。另外，还定义了一个 pipeline_coordinates_type_2[PIPELINE_X_NUM][PIPELINE_Y_NUM]数组，其变量的意义与上文所说的相同，该二维数组用来暂时存储要写入到文件中的值或者单步执行的值。

定义了 int 类型的数组 register_time_state[PIPELINE_Y_NUM]，该数组用来表示流水线中每个时刻的寄存器状态。其中，PIPELINE_Y_NUM 为坐标轴的横轴的最大长度，也可理解为程序最大支持的时间点个数，在 overalldata 中有定义。读者可以注意到，该数组的长度与二维 XY 坐标系的横轴长度相同，且每一个数据的为 32 位，正好对应 32 个寄存器。当进行流水线生成时，该数组即用来表示每一个时刻，32 个寄存器的被占用状态。

定义了 int 类型的数组 function_state[PIPELINE_Y_NUM]，该数组用来表示流水线中每个时刻五个部件的状态。其中，PIPELINE_Y_NUM 为坐标轴的横轴的最大长度，也可理解为程序最大支持的时间点个数，在 overalldata 中有定义。读者可以注意到，该数组的长度与二维 XY 坐标系的横轴长度相同。当流水线生成时，每一个时间点的 int 类型数据，使用其低 5 位，来标记 5 个部件（取指 Fetch、译码 Decode、执行 Execute、访存 Memory、写回 WriteBack）的被占用情况。

定义了 int 类型的数组 instruction_start[PIPELINE_X_NUM]，该数组用来表示流水线中每一条指令的取指 Fetch 的开始位置。其中 PIPELINE_X_NUM 为坐标轴的纵轴的最大长度，也可理解为程序最多支持的指令的条数，在 overalldata 中有定义。一般的，如果流水线全部流通未有阻滞，则该数组应该是一个递增的序列（0、1、2、3……），但在现实情况中，往往会有不同的情况发生，所以该数组中的值会随时进行修改。

定义了 bool 类型的数组 if_write[PIPELINE_X_NUM]，该数组用来表示每一条指令是否已经生成流水线或二维 XY 坐标系中，纵轴的每一行是否已经被用。其中，PIPELINE_X_NUM 为坐标轴的最大长度，也可理解为程序最多支持的指令的条数，在 overalldata 中有定义。该数组用来辅助判断每一条指令是否已经被处理，或某一条前面的指令是否已经处理完毕。

4.1.2 二维 XY 坐标系

算法所使用的二维 XY 坐标系，是一种直接的表示方法。在生成流水线时，程序按照 instruction_start 数组中对应的值，在某一条指令对应的行的对应列开始读取/写入，因为程序是顺序处理的，所以当处理到某一条指令时，其前面的指令都已经被处理完毕。且其下面的行都已经被填上字符标记，则当处理这一条指令时，只需要读取其前面的各个字符，以及 register_time_state、function_state 数组中的对应值即可。

二维 XY 坐标系的示意图如图 4.1-1 所示。

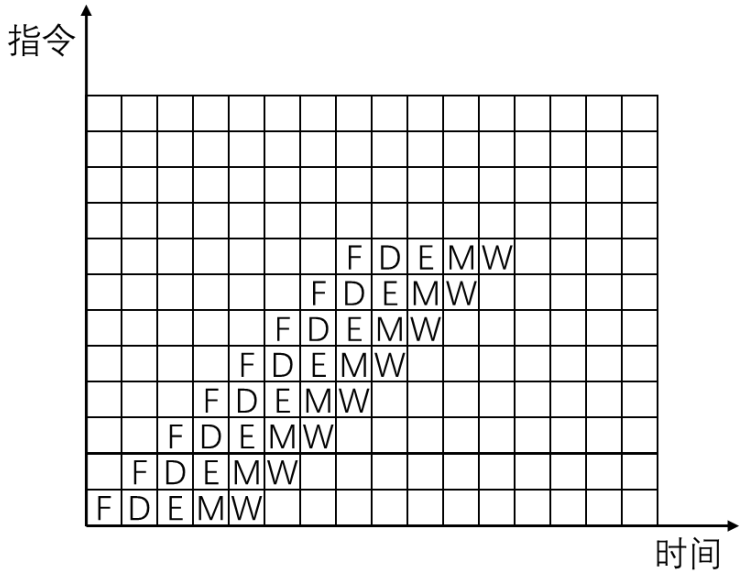


图 4.1-1

4.1.3 寄存器占用检测与标记

根据题目的要求与假设，每条指令都必须占用 5 个阶段，且每一个阶段所占用的时间相同。在这样的假设条件下，经过验证，数据相关的两种情况读后写、写后写可不用考虑，只需要考虑写后读的数据相关情况。若发生写后读的情况，则需要将指令在取指阶段暂停执行，直至上一条指令完成写入步骤才可继续向前推进。

根据这样的想法，可使用 register_time_state 数组标记每一个时刻 32 个寄存器的被占用情况。且该数组的每一个元素皆为 int 类型，都为 32 个 bit 位，可使用位运算进行修改，以节省空间、加快执行速度。若被占用，则为 1；若读取或空闲，则为 0。

当某一条指令推进到某一时刻时，通过特定的函数检测自身所需要读取的指令在下一时刻是否已经被占用，若检测到没有被占用，则可以继续推进；若检测到被占用，则开始停顿，在下一时刻写入“等待”对应的字符。同时，利用特定的函数检测该指令自身所需要写入的寄存器，同样的，也在下一时刻往 register_time_state 数组中写入，方便之后的指令的检测。

寄存器占用检测示意如图 4.1-2 所示。

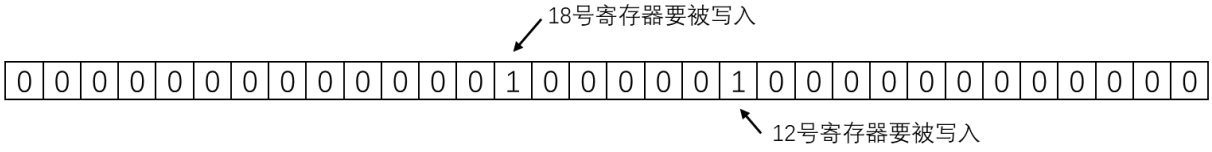


图 4.1-2

4.1.4 部件占用检测与标记

该功能的主要目的是为了解决资源相关。虽题目并未要求解决，但在实际中不得不考虑这种阻滞带来的影响。若只考虑数据相关而不考虑资源相关，则生成的流水线将会不切合实际。

与寄存器的占用检测类似，该功能使用 function_state 数组来记录每一时刻的各个部件的被占用情况。因为指令的执行过程中只涉及到 5 个部件，故使用位运算标记时，也只使用低 5 位，即 0-4 位。若被占用，则为 1；若读取或空闲，则为 0。

当某一条指令推进到某一时刻时，通过特定的函数检测自身所需要使用的部件在下个时刻是否已经被占用，若没有被占用，即可往前推进；若被占用，则在此时间点写入

等待，直到需要使用的部件为空闲。同时，也在此时刻写入指令本身当前所占用的部件，方便之后的指令的检测。

部件占用检测示意如图 4.1-3 所示。

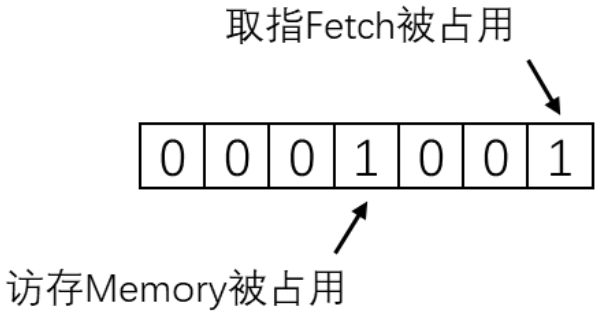


图 4.1-3

4.1.5 时间点检测

时间点的检测是以每一个时间点为单位，而不是以每一条指令为单位。在二维 XY 坐标系中的体现便是以列作为单位进行遍历查询。

对时间点的检测，可覆盖多条指令，并准确得到在该时间点的寄存器被占用状态和五个部件的被占用状态。

时间点检测技术是流水线生成步骤中必不可少的一项重要功能。检测的内容包括：是否发生数据相关、是否发生资源相关。

时间点检测的基本原理如图 4.1-5 所示。

经过上面的介绍，读者可以发现，流水线二维 XY 坐标系 pipeline_coordinates 的长度与 register_time_state 数组和 function_state 数组的长度相同。即表明他们被访问、被修改的时间是相同的。二维 XY 坐标系 pipeline_coordinates 负责标记每个时刻的各条指令推进情况；寄存器状态数组 register_time_state 负责表示每个时刻 32 个寄存器的被占用情况；部件状态数组 function_state 负责表示每个时刻 5 个部件的被占用情况。他们的对应关系如图 4.1-6 所示。

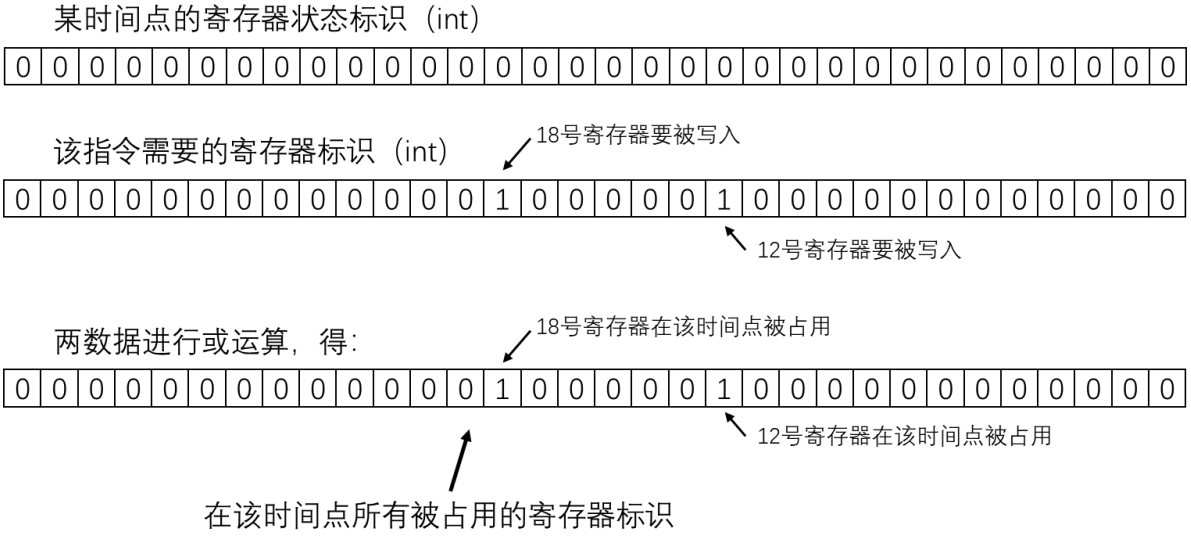


图 4.1-5

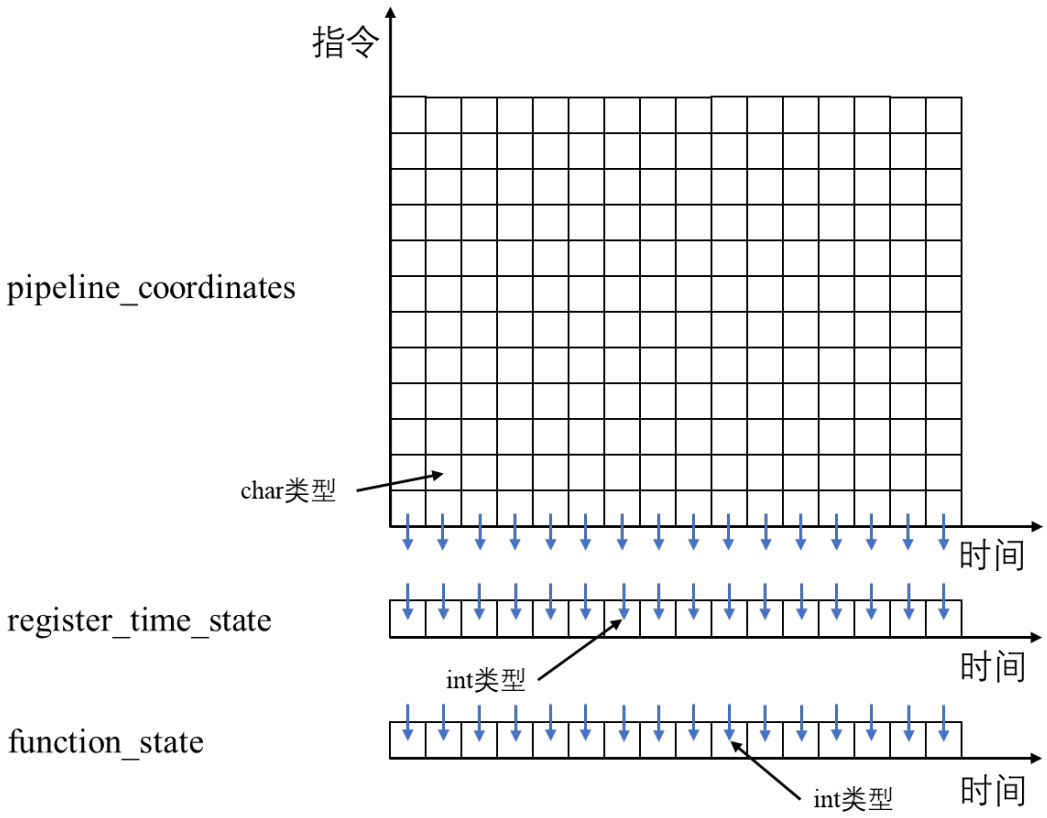


图 4.1-6

4.1.6 生成步骤

有了前面的叙述做准备，现在描述流水线的具体生成步骤将会比较简单，具体的步骤如下：

①从第一条指令开始遍历，直至最后一条指令结束
②得到指令的起始位置 address
③判断取指 Fetch 部件是否被占用
　①若 Fetch 部件被占用，则等待
　②若 Fetch 部件未被占用，则继续推进
④Decode 解码，得到该指令的类型，需要读 / 写的寄存器
⑤判断译码 Decode 部件是否被占用
　①不被占用，判断要读的寄存器是否被占用要写
　　①如果要读的寄存器被占用，则等待
　　②如果要读的寄存器不被占用，则继续推进
　②被占用，则继续等待
⑥判断 Execute 执行部件是否被占用
　①若 Execute 部件被占用，则等待
　②若 Execute 部件未被占用，则继续推进
⑦判断 Memory 访存部件是否被占用
　①不被占用，判断该条指令是否有比前面指令先完成的可能
　　①没有超前结束的可能，则继续推进
　　②有超前结束的可能，则等待
　②被占用，则等待
⑧执行写回操作

⑨结束

4.2 流水线单步执行

流水线的单步执行，是基于流水线基础上的软件模拟。当程序自动生成流水线的二维 XY 图后，单步执行即可按照每一个时间点来读取各条指令所执行的动作，并作出相应的模拟。

4.2.1 二维 XY 坐标系的读取

对二维 XY 坐标系的读取，只需要读取之前使用流水线生成算法生成的 pipeline_coordinates 数组即可，以时间点为单位，逐个读取。

4.2.2 时间点检测

对二维 XY 坐标系的时间点检测，即以列为单位，逐个取出 pipeline_coordinates 数组中的每一列，并进行处理。

4.2.3 各部件调用

程序根据取得的每一列的信息，逐个分析出每条指令当前所处于的状态。接着，交由控制 control 部件对其他的各个模块进行调用，以完成流水线的单步模拟。

4.3 流水线绘图

流水线的绘图是一个单独的程序，利用 C 语言编写，基于 graphics 图形库。该程序的目的是为了将程序生成的流水线的相关信息用图形的方式进行展示。因编程语言的缘故，故只能利用面向过程的设计思想。

4.3.1 图形库调用

利用网上提供的用于 VS2017 软件的图形库，来进行图形的绘制。该图形库包含有一个头文件 graphics.h，与两个库文件 EasyXa.lib、EasyXw.lib。

4.3.2 过程绘制

程序读取流水线生成的缓存文件，即可绘制出相应的图形。缓存文件以二维 XY 坐标系的方式给出了各条指令，各个时间点的状态。每一个状态都有一个相对应的字符。根据这些字符的含义，即可绘制。

5 测试及结果

该节具体介绍了程序的运行情况，并配以图片做具体的展示。由于程序的功能较多，且很多功能的使用已经在前文说明，因此没有必要全部展示。此处只展示程序的核心部分，即指令执行模块与流水线模块的运行情况。其他的功能，读者可以根据前文的介绍，自行运行程序测试。

5.1 菜单面板

打开程序，即可看见主菜单。如图 5.1-1 所示。

主菜单下又可细分为 8 个项目，输入对应的序号，即可继续进入。每一个项目的功能及作用已经在前文做过详细介绍，因此本初不再赘述。各个子菜单的界面如图 5.1-2 所示。

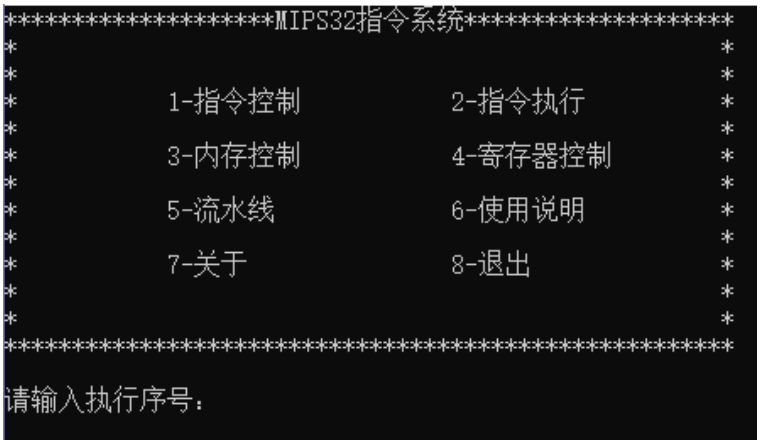
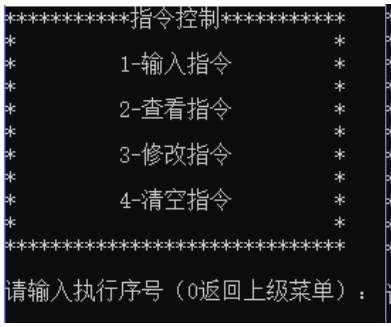
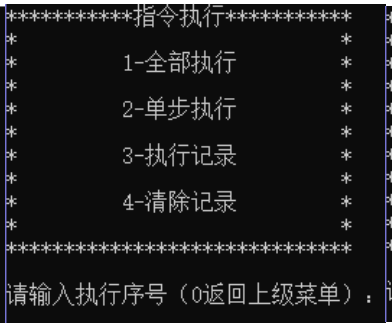
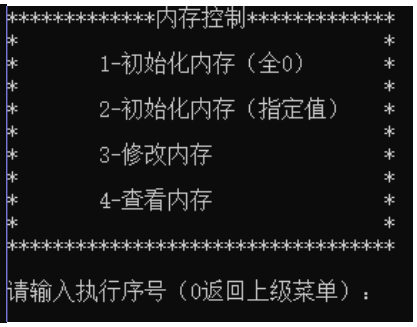
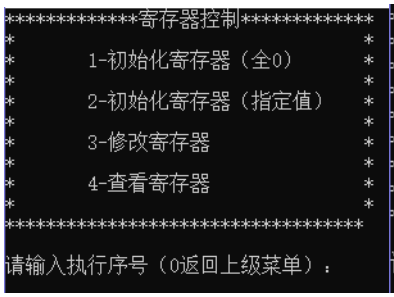

MIPS32指令系统
1-指令控制
2-指令执行
3-内存控制
4-寄存器控制
5-流水线
6-使用说明
7-关于
8-退出
请输入执行序号：

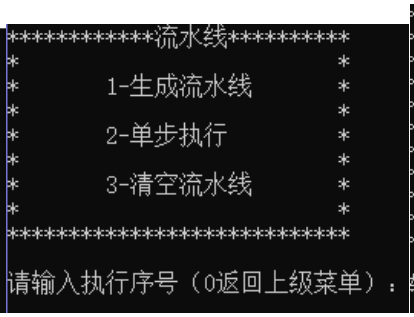
图 5.1-1

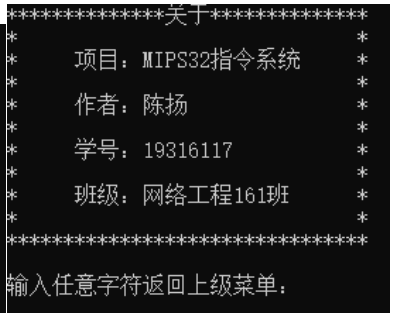

指令控制
1-输入指令
2-查看指令
3-修改指令
4-清空指令
请输入执行序号（0返回上级菜单）：


指令执行
1-全部执行
2-单步执行
3-执行记录
4-清除记录
请输入执行序号（0返回上级菜单）：


内存控制
1-初始化内存（全0）
2-初始化内存（指定值）
3-修改内存
4-查看内存
请输入执行序号（0返回上级菜单）：


寄存器控制
1-初始化寄存器（全0）
2-初始化寄存器（指定值）
3-修改寄存器
4-查看寄存器
请输入执行序号（0返回上级菜单）：


流水线
1-生成流水线
2-单步执行
3-清空流水线
请输入执行序号（0返回上级菜单）：


关于
项目：MIPS32指令系统
作者：陈扬
学号：19316117
班级：网络工程161班
输入任意字符返回上级菜单：

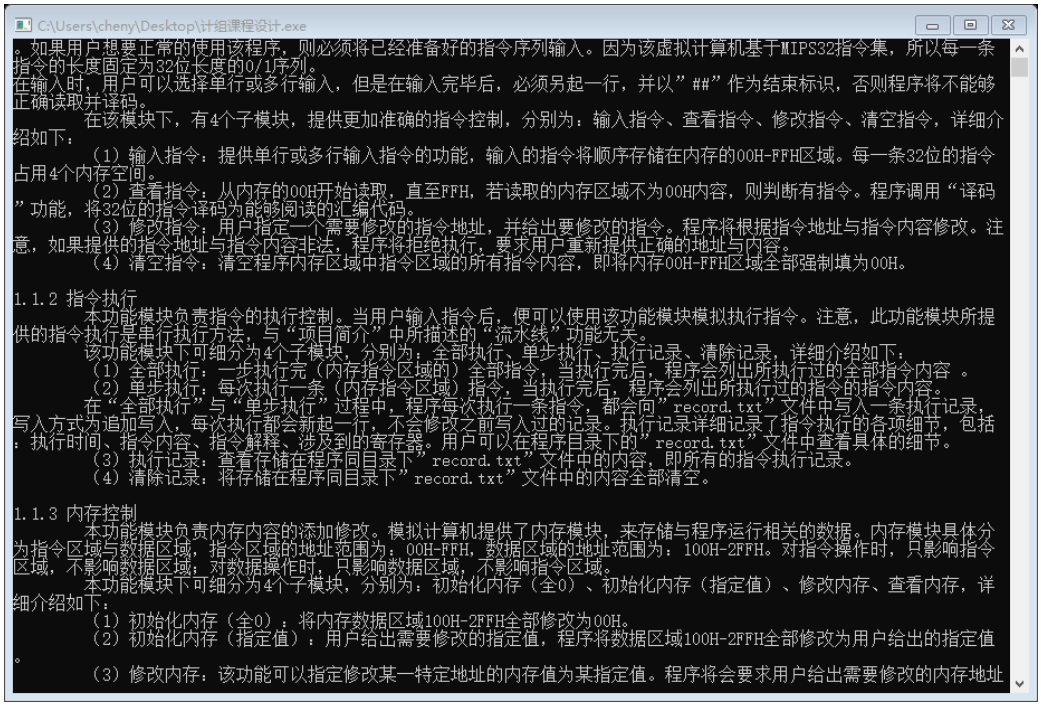


图 5.1-2

5.2 指令执行

5.2.1 测试前准备

为了能够准确的反映程序内部的运行状况，且使得结果较为明显。故在测试前，先对内存与寄存器做初始化。内存全部初始化为 100，寄存器全部初始化为 200。

初始化后查看内存与寄存器的值，如图 5.2-1、5.2-2

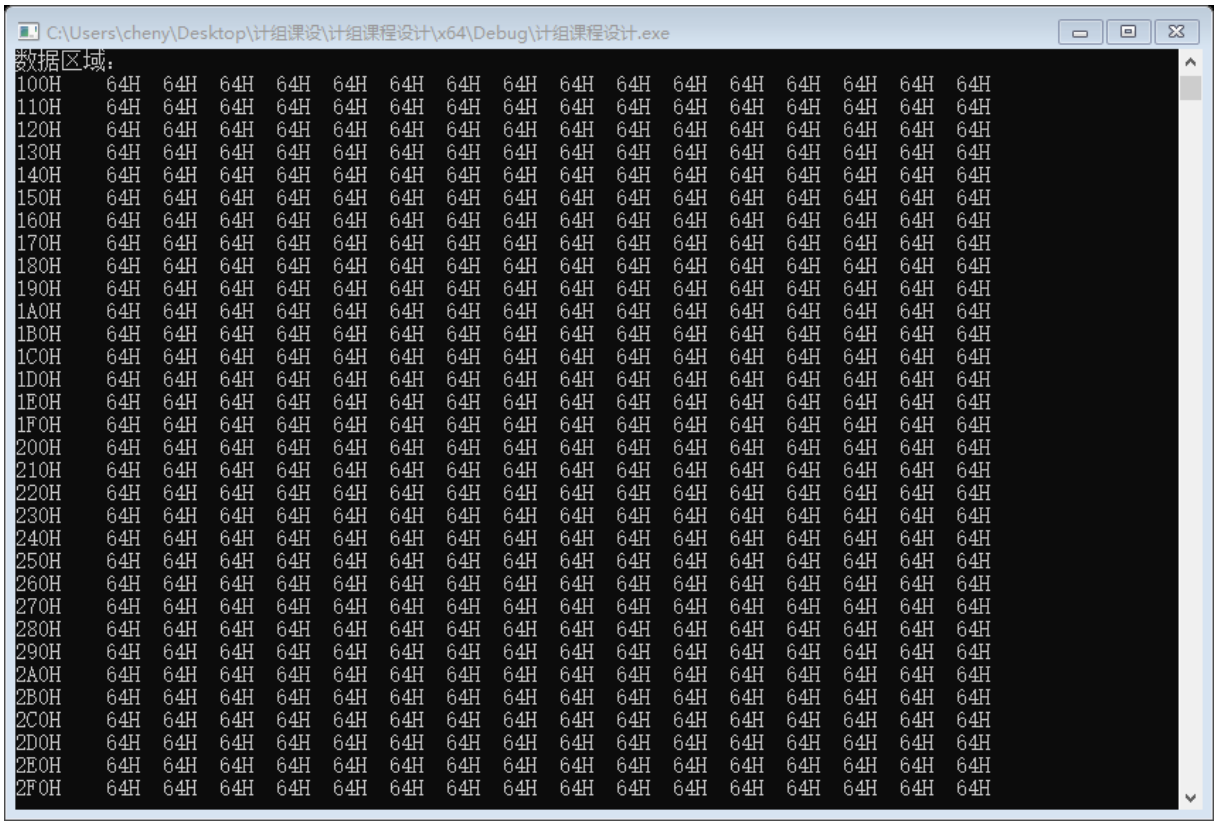


图 5.2-1

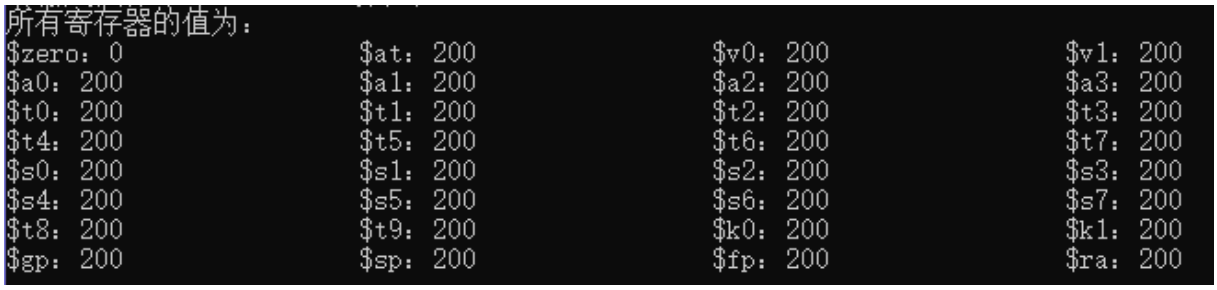


图 5.2-2

测试使用的指令为：

00000000001000100001100000100001	addu $v1, $at, $v0
00000000100001010011000000100011	subu $a2, $a0, $a1
00000000111010000100100000100100	and $t1, $a3, $t0
00000001010010110110000000100101	or $t4, $t2, $t3
00000001101011100111100000100110	xor $t7, $t5, $t6
00000010000100011001000000100111	nor $s2, $s0, $s1
00000000000100111010000010000000	sll $s4, $s3, 2
00000000000101011011000100000010	srl $s6, $s5, 4
00100010111110000110101001010010	addi $t8, $s7, 27218
00110011001110101010101101010100	andi $k0, $t9, 43860
00110111011111000101010001100110	ori $gp, $k1, 21606
00111011101111100101000110001100	xori $fp, $sp, 20876

将这些指令输入到程序中，准备执行。

5.2.2 全部执行

使用“全部执行”功能，一次执行全部的指令。结果如图 5.2-3 所示。

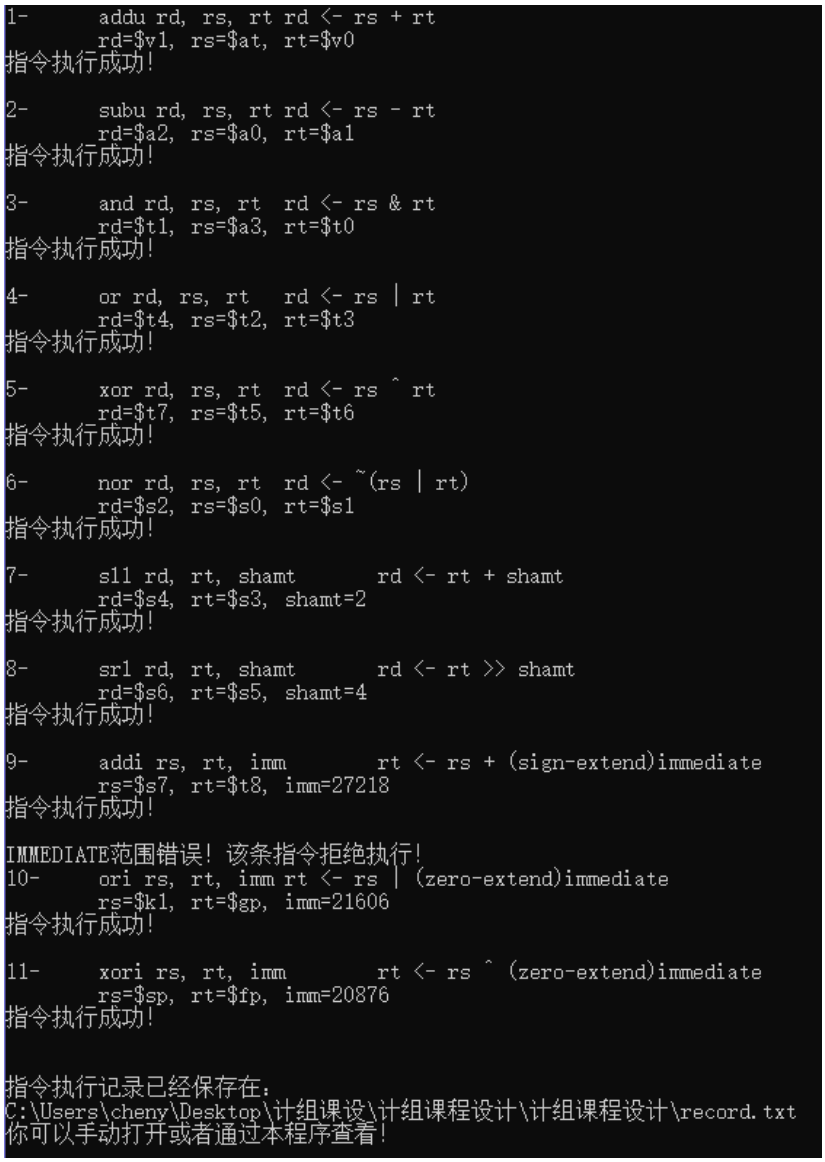


图 5.2-3

5.2.3 单步执行

使用“单步执行”功能，开始单步执行输入的指令。没次执行一部，程序便会停顿，询问是否继续执行，使用者可以选择继续执行或者终止执行。如图 5.2-4、5.2-5 所示。

当执行完毕后，用户可以通过控制台的导航功能，执行“查看寄存器”与“查看内存”功能，查看寄存器与内存修改后的值。

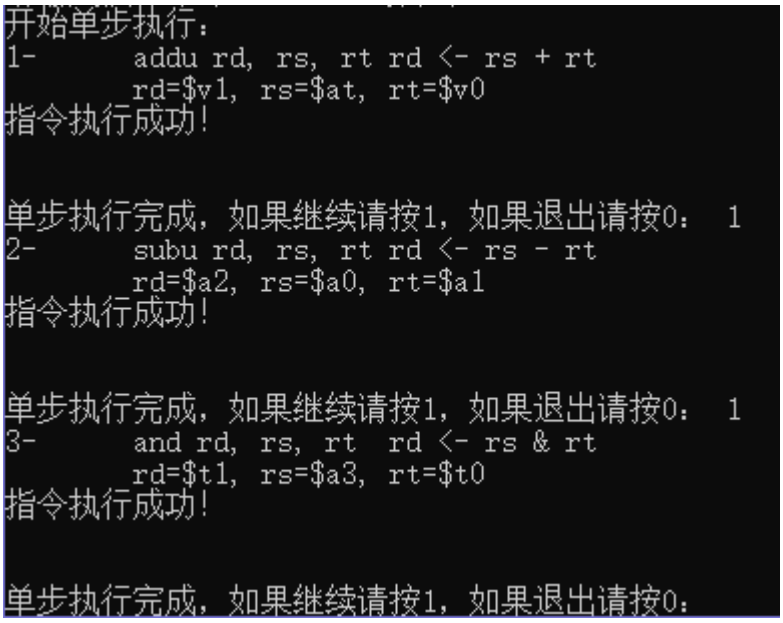


图 5.2-4

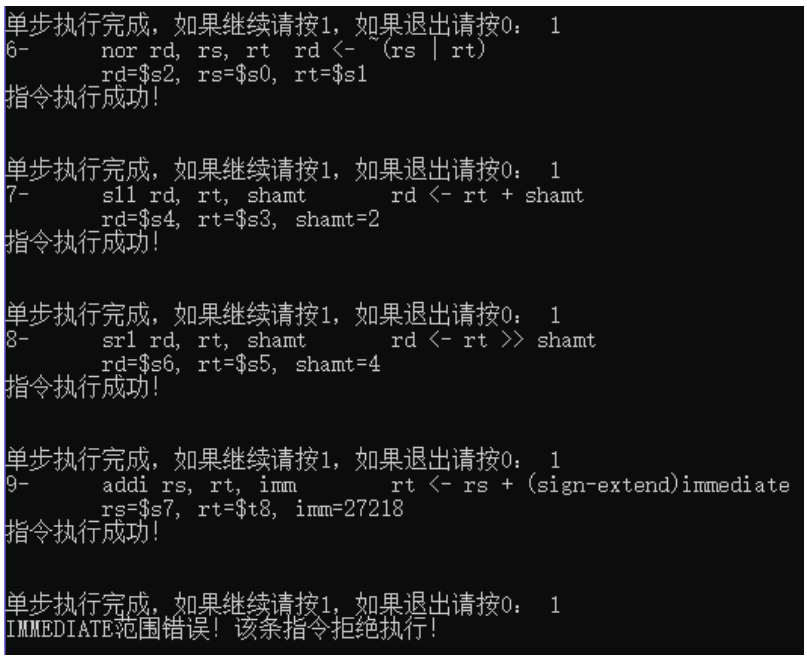


图 5.2-5

5.3 流水线

5.3.1 全部流通情况

如“指令执行”一样，为了能够准确的反映程序内部的运行状况，且使得结果较为明显。故在测试前，先对内存与寄存器做初始化。内存全部初始化为 100，寄存器全部初始化为 200。

准备全部流通的指令，指令为：

00000000001000100001100000100001	addu $v1, $at, $v0
00000000100001010011000000100011	subu $a2, $a0, $a1
00000000111010000100100000100100	and $t1, $a3, $t0
00000001010010110110000000100101	or $t4, $t2, $t3
00000001101011100111100000100110	xor $t7, $t5, $t6
00000010000100011001000000100111	nor $s2, $s0, $s1

```
00000000000100111010000010000000    sll $s4, $s3, 2
00000000000101011011000100000010    srl $s6, $s5, 4
00100010111110000110101001010010    addi $t8, $s7, 27218
00110011001110101010101101010100    andi $k0, $t9, 43860
00110111011111000101010001100110    ori $gp, $k1, 21606
00111011101111100101000110001100    xori $fp, $sp, 20876
```

使用“生成流水线”功能测试。在控制台界面只给出流水线的形状，如图 5.3-1 所示。

同时，若用户执行“流水线绘图”程序，则程序将会根据当前目录下生成的缓存文件，自动绘制出与控制台界面相同的图形，如图 5.3-2 所示。

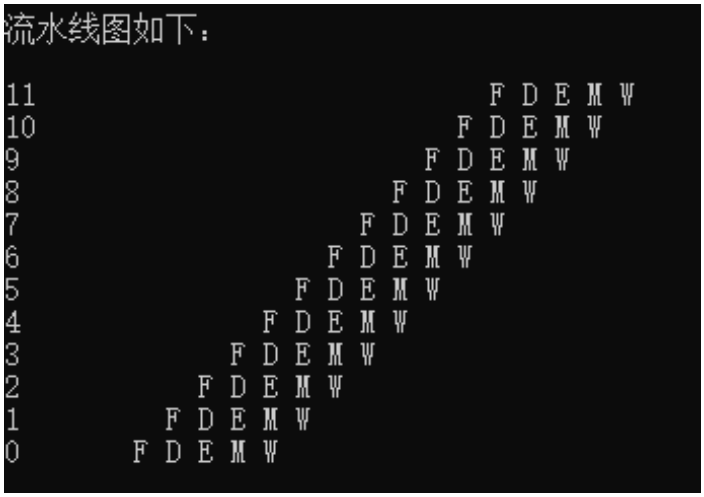


图 5.3-1

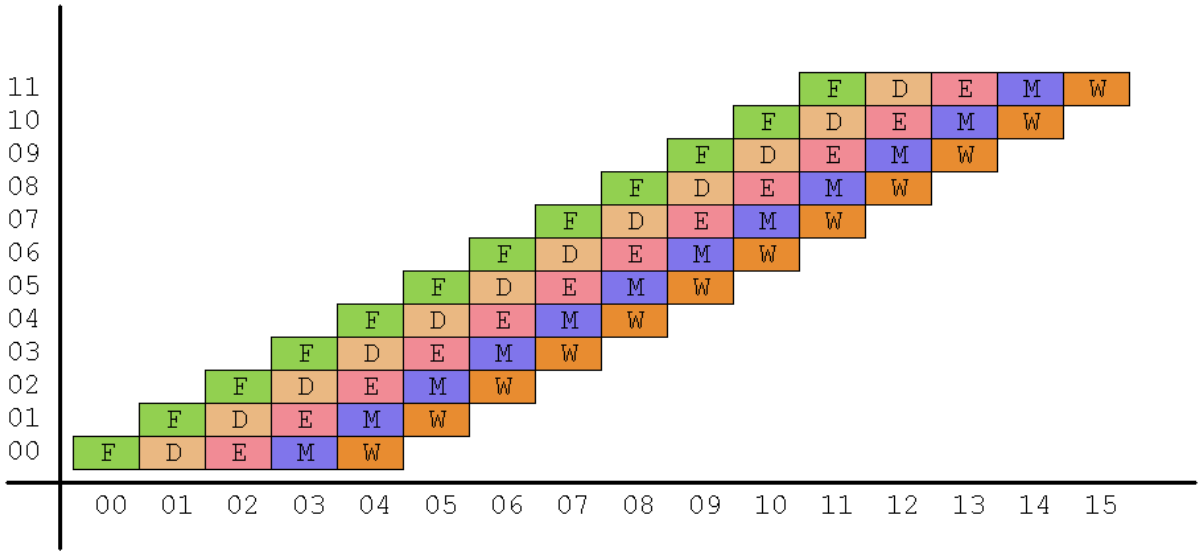


图 5.3-2

使用“单步执行”功能测试，与“生成流水线功能”不同，该功能能够单步模拟每个时间点 CPU 内部的运行过程与结果。并给出相应的信息、所有寄存器的值、PC 指针当前的值。用户可以通过这些信息，来更好的理解体会流水线的功能特性。相关示意图如图 5.3-3、5.3-4 所示。

在控制台界面中，程序将会详细的描述每一条指令当前的状态（取值、译码、执行、访存、写回），并对每一个当前的状态做出解释说明。

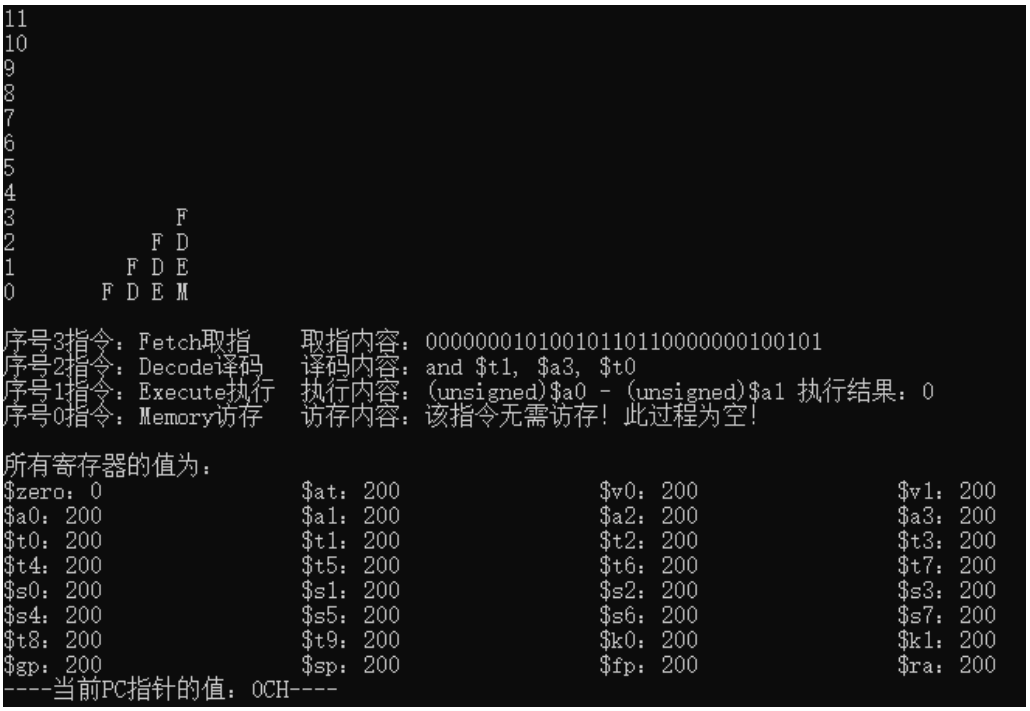


图 5.3-3

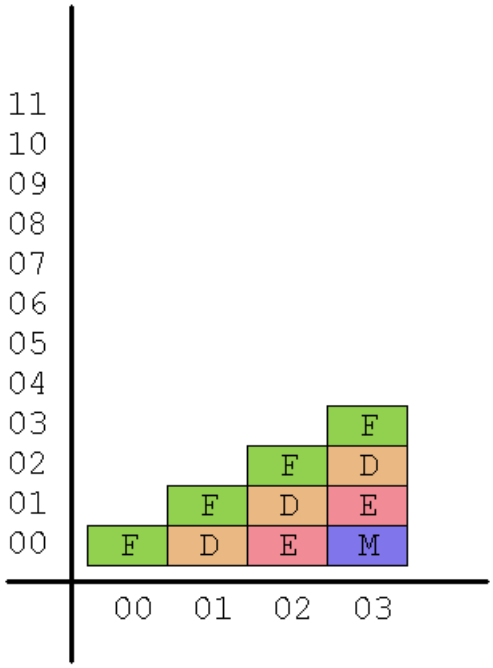


图 5.3-4

5.3.2 写$zero 寄存器情况

准备存在写$zero 寄存器的指令，指令为：

00000000001000100001100000100001	addu $v1, $at, $v0
00000000100001010011000000100011	subu $a2, $a0, $a1
00000000111010000000000000100100	and $zero, $a3, $t0
00000001010010110000000000100101	or $zero, $t2, $t3

00000001101011100111100000100110	xor $t7, $t5, $t6
00000010000100010000000000100111	nor $zero, $s0, $s1
00000000000100111010000010000000	sll $s4, $s3, 2
00000000000101011011000100000010	srl $s6, $s5, 4
00100010111110000110101001010010	addi $t8, $s7, 27218
00110011001110101010101101010100	andi $k0, $t9, 43860
00110111011111000101010001100110	ori $gp, $k1, 21606
00111011101111100101000110001100	xori $fp, $sp, 20876

其中，标为橙色的即为试图写入$zero 寄存器的指令。

使用“生成流水线”功能测试。在控制台界面只给出流水线的形状，如图 5.3-5 所示。

同时，若用户执行“流水线绘图”程序，则程序将会根据当前目录下生成的缓存文件，自动绘制出与控制台界面相同的图形，如图 5.3-6 所示。

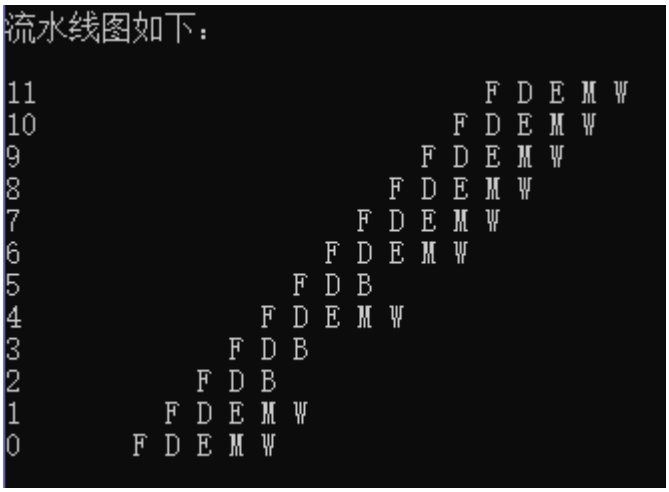


图 5.3-5

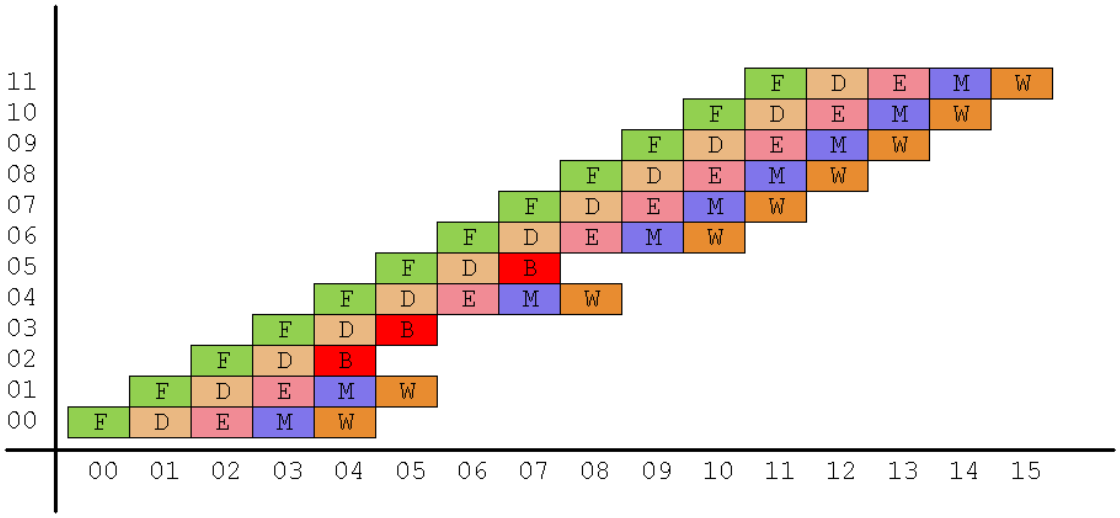


图 5.3-6

使用“单步执行”功能测试，与“生成流水线功能”不同，该功能能够单步模拟每个时间点 CPU 内部的运行过程与结果。并给出相应的信息、所有寄存器的值、PC 指针

当前的值。用户可以通过这些信息，来更好的理解体会流水线的功能特性。相关示意图如图 5.3-7、5.3-8 所示。

在控制台界面中，程序将会详细的描述每一条指令当前的状态（取值、译码、执行、访存、写回），并对每一个当前的状态做出解释说明。

```
11
10
9                       F
8                     F D
7                   F D E
6                 F D E M
5               F D B
4             F D E M W
3           F D B
2         F D B
1       F D E M W
0     F D E M W

序号9指令：Fetch取指     取指内容：00110011001110101010101101010100
序号8指令：Decode译码    译码内容：addi $s7, $t8, 27218
序号7指令：Execute执行   执行内容：$s5 >> 4       执行结果：6
序号6指令：Memory访存    访存内容：该指令无需访存！此过程为空！

所有寄存器的值为：
$zero：0                $at：100                $v0：100                $v1：200
$a0：100                $a1：100                $a2：0          $a3：100
$t0：100                $t1：100                $t2：100                $t3：100
$t4：100                $t5：100                $t6：100                $t7：0
$s0：100                $s1：100                $s2：100                $s3：100
$s4：400                $s5：100                $s6：6          $s7：100
$t8：27318              $t9：100                $k0：100                $k1：100
$gp：100                $sp：100                $fp：100                $ra：100
----当前PC指针的值：24H----
```

图 5.3-7

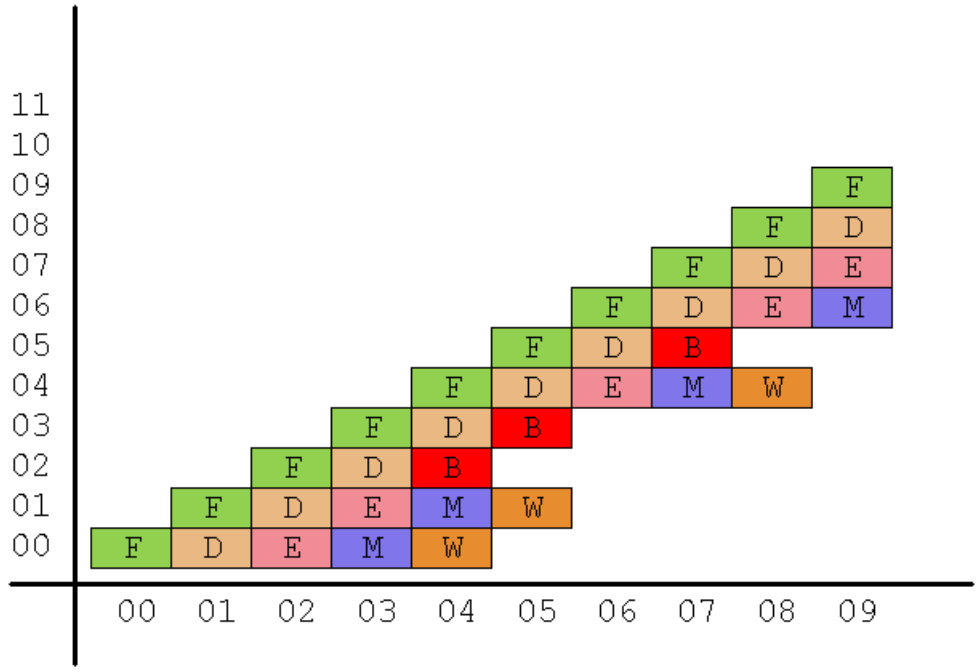


图 5.3-8

5.3.3 数据相关情况

准备存在数据相关的指令，指令为：

00000000001000100001100000100001	addu $v1, $at, $v0

00000000011001010011000000100011	subu $a2, $v1, $a1
00000000111010000100100000100100	and $t1, $a3, $t0
00000001010010110110000000100101	or $t4, $t2, $t3
00000001101011100111100000100110	xor $t7, $t5, $t6
00000001111100011001000000100111	nor $s2, $t7, $s1
00000000000100111010000010000000	sll $s4, $s3, 2
00000000000101011011000100000010	srl $s6, $s5, 4
00100010111110000110101001010010	addi $t8, $s7, 27218
00110011001110101010101101010100	andi $k0, $t9, 43860
00110111011111000101010001100110	ori $gp, $k1, 21606
00111011101111100101000110001100	xori $fp, $sp, 20876

其中，标为绿色的即为有数据相关的指令。

使用“生成流水线”功能测试。在控制台界面只给出流水线的形状，如图 5.3-9 所示。

同时，若用户执行“流水线绘图”程序，则程序将会根据当前目录下生成的缓存文件，自动绘制出与控制台界面相同的图形，如图 5.3-10 所示。

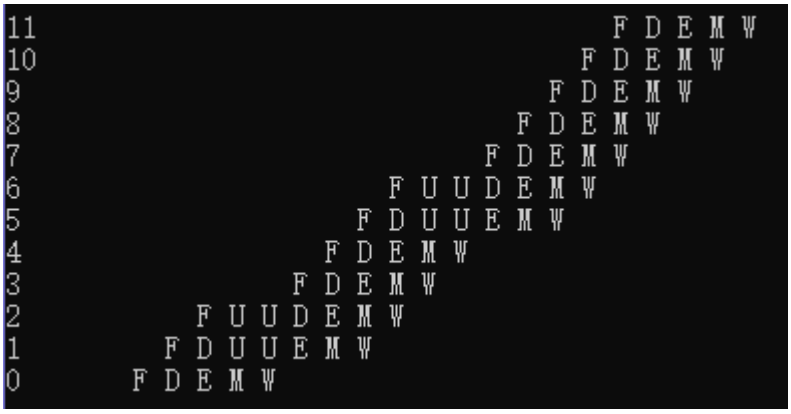


图 5.3-9

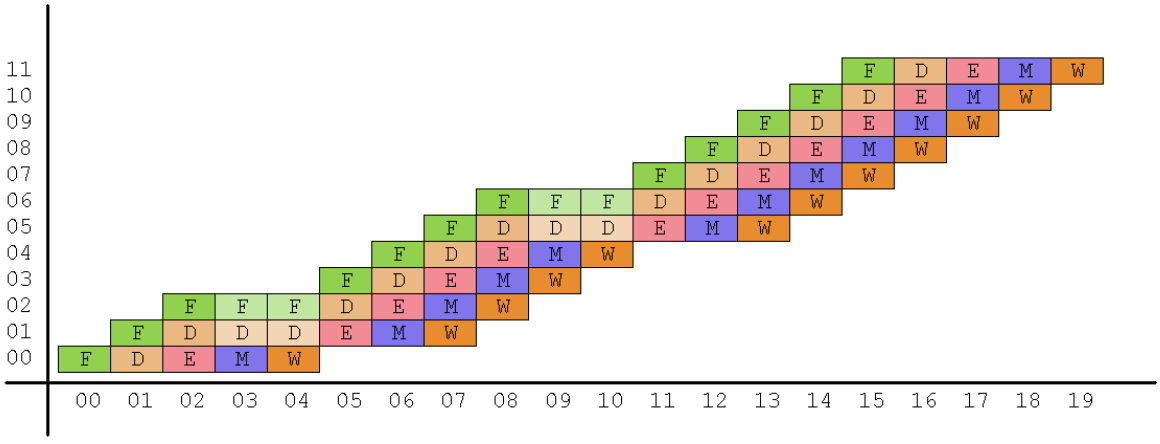


图 5.3-10

使用“单步执行”功能测试，与“生成流水线功能”不同，该功能能够单步模拟每个时间点，CPU 内部的运行过程与结果。并给出相应的信息、所有寄存器的值、PC 指针当前的值。用户可以通过这些信息，来更好的理解体会流水线的功能特性。相关示意

图如图 5.3-11、5.3-12 所示。

在控制台界面中，程序将会详细的描述每一条指令当前的状态（取值、译码、执行、访存、写回），并对每一个当前的状态做出解释说明。

图 5.3-11

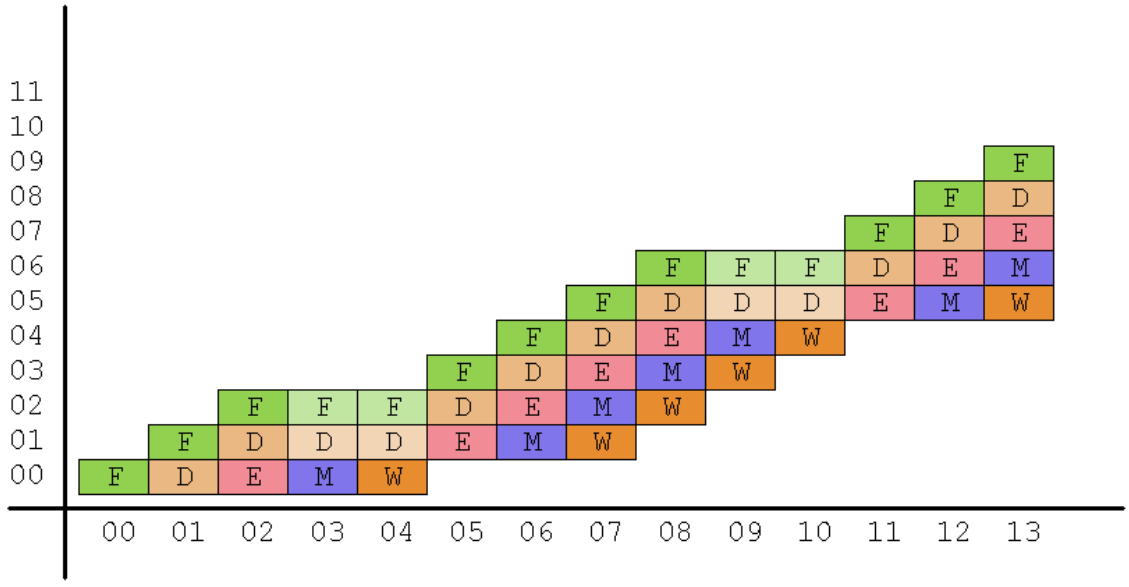


图 5.3-12

5.3.4 混合情况

准备既存在写$zero 寄存器、又有数据相关的指令，指令为：

00000000001000100001100000100001	addu $v1, $at, $v0
00000000011001010011000000100011	subu $a2, $v1, $a1
00000000111010000100100000100100	and $t1, $a3, $t0

00000001010010110000000000100101	or $zero, $t2, $t3
00000001010010110110000000100101	or $t4, $t2, $t3
00000001101011100111100000100110	xor $t7, $t5, $t6
00000001111100011001000000100111	nor $s2, $t7, $s1
00000000111010000000000000100100	and $zero, $a3, $t0
00000000000010011101000010000000	sll $s4, $s3, 2
00000000000010101101100010000010	srl $s6, $s5, 4
00100010111110000110101001010010	addi $t8, $s7, 27218
00110011001110101010101101010100	andi $k0, $t9, 43860
00110111011111000101010001100110	ori $gp, $k1, 21606
00111011101111100101000110001100	xori $fp, $sp, 20876

其中，标为橙色的为试图写入$zero 寄存器的指令，标为绿色的为有数据相关的指令。

使用“生成流水线”功能测试。在控制台界面只给出流水线的形状，如图 5.3-13 所示。

同时，若用户执行“流水线绘图”程序，则程序将会根据当前目录下生成的缓存文件，自动绘制出与控制台界面相同的图形，如图 5.3-14 所示。

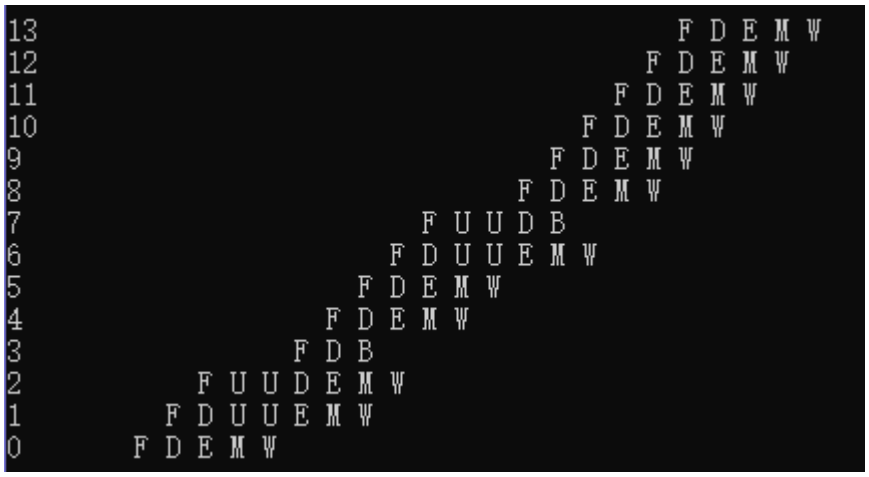


图 5.3-13

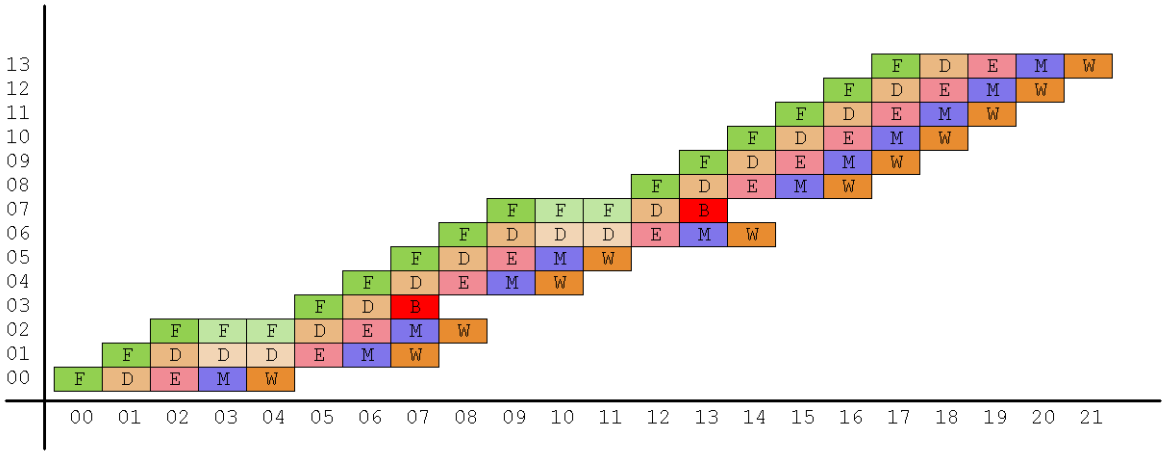


图 5.3-14

使用“单步执行”功能测试，与“生成流水线功能”不同，该功能能够单步模拟每

个时间点，CPU 内部的运行过程与结果。并给出相应的信息、所有寄存器的值、PC 指针当前的值。用户可以通过这些信息，来更好的理解体会流水线的功能特性。相关示意图如图 5.3-15、5.3-16 所示。

在控制台界面中，程序将会详细的描述每一条指令当前的状态（取值、译码、执行、访存、写回），并对每一个当前的状态做出解释说明。

图 5.3-15

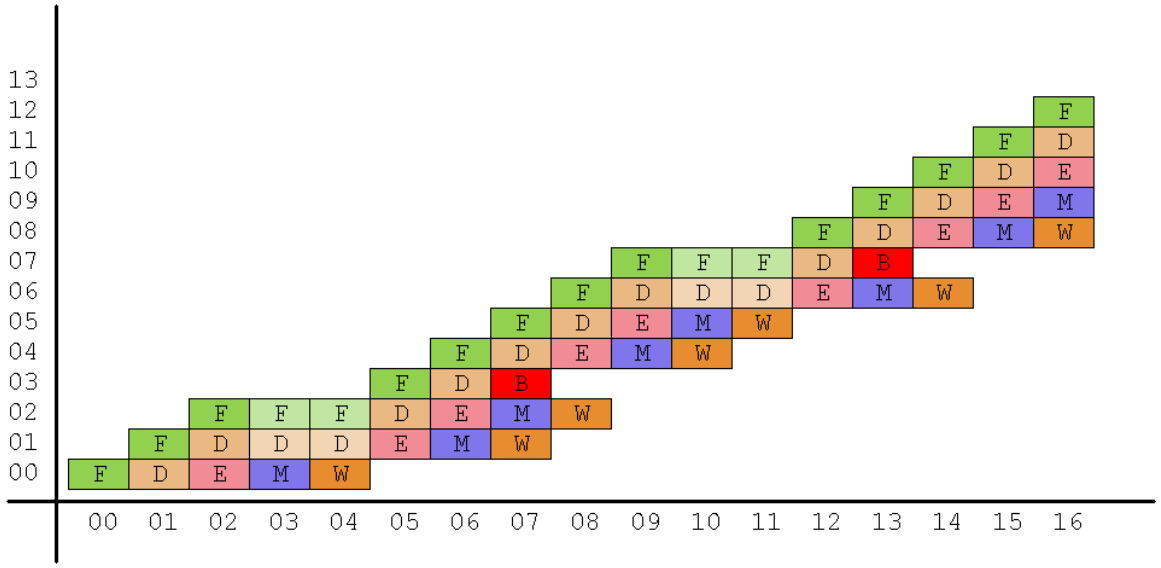


图 5.3-16

6 详细设计

6.1 内存类

```
#pragma once
#include "overalldata.h"
class CMemory
{
public:char mem[MEMORY_NUM];
private:
    int instruction_size;
    int data_size;
    int memory_state;
    int current_instruction_address;
public:
    CMemory();
    ~CMemory();
    int SetMemoryBusy();
    int SetMemoryFree();
    int ReturnCurrentState(int &);
    int InitAllMemoryWith0();
    int InitAllMemoryWithValue(int);
    int WriteToMemory(int, char);
    int ReadFromMemory(int, char &);
    int ResetCurrentInstructionAddress();
    int GetCurrentInstructionAddress();
    int AddInstructionAddress();
};
```

6.2 寄存器类

```
#pragma once
#include "overalldata.h"
#include <string>
using namespace std;
class CRegister
{
public:int reg[REGISTER_NUM];
private:
    int register_state;
public:
    CRegister();
    ~CRegister();
    int SetRegisterBusy();
    int SetRegisterFree();
    int ReturnCurrentState(int &);
    int InitAllRegisterWith0();
    int InitAllRegisterWithValue(int);
```

```
        int WriteToOneRegister(int, int);
        int ReadFromOneRegister(int, int &);
    };
```

6.3 取指类

```
    #pragma once
    #include "overalldata.h"
    class CFetch
    {
    public:int PC;
    private:
        int fetch_state;
    public:
        CFetch();
        ~CFetch();
        int SetFetchBusy();
        int SetFetchFree();
        int ReturnCurrentState(int &);
        int GetPC();
        int PCSelfAdd();
        int EditPC(int);
        int ResetPC();
        int FetchInstructionFromMemory(int, char[], int &);
    };
```

6.4 译码类

```
    #pragma once
    #include "overalldata.h"
    class CDecoding
    {
    private:
        int decoding_state;
    public:
        CDecoding();
        ~CDecoding();
        int SetDecodingBusy();
        int SetDecodingFree();
        int ReturnCurrentState(int &);
        int BitSelectTransform(int, int, int, int &);
        int InstructionsDecoding(int, InstructionStruct &, int &);
        bool InstructionCheckZero(int instruction_type, InstructionStruct data);
    };
```

6.5 执行类

```
    #pragma once
    #include <string>
    #include "overalldata.h"
    using namespace std;
```

```
class CExecute
{
private:
    int execute_state;
    int instruction_num;
public:
    CExecute();
    ~CExecute();
    int SetExecuteBusy();
    int SetExecuteFree();
    int ReturnCurrentState(int &);
    int sign_extended(int &);
    int zero_extended(int &);
    //R-Type
    int execute_addu(InstructionStruct, int[]);
    int execute_subu(InstructionStruct, int[]);
    int execute_and(InstructionStruct, int[]);
    int execute_or(InstructionStruct, int[]);
    int execute_xor(InstructionStruct, int[]);
    int execute_nor(InstructionStruct, int[]);
    int execute_sll(InstructionStruct, int[]);
    int execute_srl(InstructionStruct, int[]);
    int execute_jr(InstructionStruct, int[], int &);
    //I-Type
    int execute_addi(InstructionStruct, int[]);
    int execute_andi(InstructionStruct, int[]);
    int execute_ori(InstructionStruct, int[]);
    int execute_xori(InstructionStruct, int[]);
    int execute_lw(InstructionStruct, int[], char[]);
    int execute_sw(InstructionStruct, int[], char[]);
    int execute_beq(InstructionStruct, int[], int &);
    //J-Type
    int execute_j(InstructionStruct, int &);

    int ExecuteControl(int instruction_type, InstructionStruct instruction_data, char memory[], int reg[], int &PC);
    int ExecuteControlCheckIfOk(int instruction_type, InstructionStruct instruction_data, char memory[], int reg[], int &PC);
    int ExecuteControlAction(int instruction_type, InstructionStruct instruction_data, char memory[], int reg[], int &PC);
    string KnowExecuteDetail(int instruction_type, InstructionStruct data, char mem[], int reg[], int &PC);
    int KnowExecuteResult(int instruction_type, InstructionStruct data, char mem[], int reg[], int &PC);
    int KnowMemoryReadContent(int instruction_type, InstructionStruct data, char
```

```
mem[], int reg[]);
    string KnowWhichToWriteTo(int instruction_type, InstructionStruct data, char mem[],
int reg[]);
};
```

6.6 记录类

```
#pragma once
#include "overalldata.h"
#include <fstream>
#include <string>
#include <time.h>
using namespace std;
class CLog
{
private:
    char regname[REGISTER_NUM][REGISTER_MAX_NAME_SIZE];
    ofstream logfile;
    bool ifnew;
    int current_log_no;
public:
    CLog();
    ~CLog();
    int InitRegName();
    string GetRegisterName(int);
    int InitLogNo();
    string GetInstructionLogDetail(int, InstructionStruct);
    string GetInstructionString(int, InstructionStruct);
    int WriteLog(int, InstructionStruct);
    void LogBegin();
    void LogEnd();
    void ClearLog();
};
```

6.7 菜单面板类

```
#pragma once
#include "overalldata.h"
class CPanel
{
private:
public:
    int PanelChoose();
    int ShowMainPanel();
    int ShowChoose1Panel();
    int ShowChoose2Panel();
    int ShowChoose3Panel();
    int ShowChoose4Panel();
    int ShowChoose5Panel();
```

```
        int ShowChoose6Panel();
        int ShowChoose7Panel();
        int ShowChoose8Panel();
    };
```

6.8 全局数据类

```
#pragma once
//寄存器的参数
constexpr int REGISTER_BUSY = 1;
constexpr int REGISTER_FREE = 0;
constexpr int REGISTER_NUM = 32;
constexpr int REGISTER_INIT_DATA = 0;
constexpr int REGISTER_MAX_NAME_SIZE = 6;

//内存的参数
constexpr int MEMORY_BUSY = 1;
constexpr int MEMORY_FREE = 0;
constexpr int MEMORY_NUM = 768;
constexpr int MEMORY_INIT_DATA = 0;
constexpr int MEMORY_INSTRUCTION_SIZE = 256;
constexpr int MEMORY_DATA_SIZE = MEMORY_NUM -
MEMORY_INSTRUCTION_SIZE;

//取指类的参数
constexpr int FETCH_BUSY = 1;
constexpr int FETCH_FREE = 0;
constexpr int FETCH_PC_INIT_DATA = 0;
constexpr int PC_ADD_VALUE = 4;

//译码类的参数
constexpr int DECODING_BUSY = 1;
constexpr int DECODING_FREE = 0;
struct InstructionStruct
{
    char rs;
    char rt;
    char rd;
    char shamt;
    short immediate;
    short address;
};
constexpr char RS_RT_RD_ERROR_CODE = 127;
constexpr short IMMEDIATE_ADDRESS_ERROR_CODE = -1;
constexpr int INSTRUCTION_TYPE_ERROR_CODE = -1;
constexpr int SHAMT_ERROR_CODE = 127;
//指令中的 FUNC 标识状况
```

```
constexpr int R_TYPE_OP = 0x00;
constexpr int R_TYPE_ADDU_FUNC = 0x21;
constexpr int R_TYPE_SUBU_FUNC = 0x23;
constexpr int R_TYPE_AND_FUNC = 0x24;
constexpr int R_TYPE_OR_FUNC = 0x25;
constexpr int R_TYPE_XOR_FUNC = 0x26;
constexpr int R_TYPE_NOR_FUNC = 0x27;
constexpr int R_TYPE_SLL_FUNC = 0x00;
constexpr int R_TYPE_SRL_FUNC = 0x02;
constexpr int R_TYPE_JR_FUNC = 0x08;
constexpr int I_TYPE_ADDI = 0x08;
constexpr int I_TYPE_ANDI = 0x0c;
constexpr int I_TYPE_ORI = 0x0d;
constexpr int I_TYPE_XORI = 0x0e;
constexpr int I_TYPE_LW = 0x23;
constexpr int I_TYPE_SW = 0x2b;
constexpr int I_TYPE_BEQ = 0x04;
constexpr int J_TYPE_J = 0x02;
//17 条命令的对应序号
constexpr int R_TYPE_ADDU_NO = 1;
constexpr int R_TYPE_SUBU_NO = 2;
constexpr int R_TYPE_AND_NO = 3;
constexpr int R_TYPE_OR_NO = 4;
constexpr int R_TYPE_XOR_NO = 5;
constexpr int R_TYPE_NOR_NO = 6;
constexpr int R_TYPE_SLL_NO = 7;
constexpr int R_TYPE_SRL_NO = 8;
constexpr int R_TYPE_JR_NO = 9;
constexpr int I_TYPE_ADDI_NO = 10;
constexpr int I_TYPE_ANDI_NO = 11;
constexpr int I_TYPE_ORI_NO = 12;
constexpr int I_TYPE_XORI_NO = 13;
constexpr int I_TYPE_LW_NO = 14;
constexpr int I_TYPE_SW_NO = 15;
constexpr int I_TYPE_BEQ_NO = 16;
constexpr int J_TYPE_J_NO = 17;

//执行类的参数
constexpr int EXECUTE_BUSY = 1;
constexpr int EXECUTE_FREE = 0;
constexpr int EXECUTE_CONTROL_TYPE_ERROR = 1000;
constexpr int EXECUTE_CONTROL_RD_ERROR = 2000;
constexpr int EXECUTE_CONTROL_RT_ERROR = 3000;
constexpr int EXECUTE_CONTROL_RS_ERROR = 4000;
constexpr int EXECUTE_CONTROL_SHAMT_ERROR = 5000;
```

```
constexpr int EXECUTE_CONTROL_IMMEDIATE_ERROR = 6000;
constexpr int EXECUTE_CONTROL_ADDRESS_ERROR = 7000;
constexpr int EXECUTE_CONTROL_CHECK_SUCCESS = 7777;
constexpr int EXECUTE_CONTROL_ACTION_SUCCESS = 666;
constexpr int REG_ZERO_CHANGED = 1;
constexpr int REG_ZERO_NOCHANGED = 0;
constexpr char ALLREGISTERNAME[32][6] =
{"$zero","$at","$v0","$v1","$a0","$a1","$a2","$a3","$t0","$t1","$t2","$t3","$t4","$t5",
    "$t6","$t7","$s0","$s1","$s2","$s3","$s4","$s5","$s6","$s7","$t8","$t9","$k0","$k1","$g
p","$sp","$fp","$ra" };

constexpr char HEXNAME[17] = "0123456789ABCDEF";
constexpr int INSTRUCTION_TRANSFORM_ERROR_CODE = -1;
constexpr int INSTRUCTION_CODE_OK = 1;
constexpr int INSTRUCTION_CODE_ERROR = -1;
//constexpr int INSTRUCTION_CODE_USELESS = 0;
constexpr int INSTRUCTION_ACTION_OK = 1;
constexpr int INSTRUCTION_ACTION_ERROR = 0;

//面板选择类的参数
constexpr int PANEL_MAIN_ADD = 100;
constexpr int PANEL_1_ADD = 1000;
constexpr int PANEL_2_ADD = 2000;
constexpr int PANEL_3_ADD = 3000;
constexpr int PANEL_4_ADD = 4000;
constexpr int PANEL_5_ADD = 5000;
constexpr int PANEL_6_ADD = 6000;
constexpr int PANEL_7_ADD = 7000;
constexpr int PANEL_8_ADD = 8000
constexpr int PANEL_CHOOSE_ERROR_CODE = 9999;
constexpr int PANEL_RETURN_TO_MAIN_PANEL = 999;
//流水线类的参数
constexpr int PIPELINE_X_NUM = MEMORY_INSTRUCTION_SIZE / 4;
constexpr int PIPELINE_Y_NUM = 1000;
constexpr char PIPELINE_PERIOD_NAME[5][11] =
{ "Fetch","Decode","Execute","ReadMemory","WriteBack" };
constexpr char PIPELINE_PERIOD_NAME_SHORTEN[9] = "UFDEMWXB";
constexpr int REGISTER_TIME_READ = 1;
constexpr int REGISTER_TIME_WRITE = 1;
constexpr int NOTEXISTTHISINSTRUCTIONONONONETIME = -1;
```

6.9 流水线类

```
#pragma once
#include "overalldata.h"
#include <fstream>
using namespace std;
```

```
class CPipeline
{
private:
    char pipeline_coordinates[PIPELINE_X_NUM][PIPELINE_Y_NUM];
    char pipeline_coordinates_type_2[PIPELINE_X_NUM][PIPELINE_Y_NUM];
    int register_time_state[PIPELINE_Y_NUM];
    int function_state[PIPELINE_Y_NUM];
    int instruction_start[PIPELINE_X_NUM];
    bool if_write[PIPELINE_X_NUM];

public:
    CPipeline();
    ~CPipeline();
    void ResetAll();
    void ResetPipelineCoordinates();
    void ResetRegsiterTimeState();
    void ResetFunctionState();
    void ResetInstructionStartAddress();
    void ResetIfWrite();
    void WriteBan(int,int,InstructionStruct);
    int WhichNeedRead(int, InstructionStruct);
    int WhichNeedWrite(int, InstructionStruct);
    void WriteBitToInt(int &, int, int);
    int CheckBit(int content, int address);
    int CheckInstructionNum();
    bool CheckPreviousInstructionIfDown(int, int);
    bool CheckPreviousOneInstructionIfDown(int, int);
    int CreatePipeline(int, int, InstructionStruct);
    int CreatePipelineStep(int step_num);
    bool CheckAnyInstruction(int step_num);
    void WriteCoordinatesToFile();
    void WriteCoordinatesToFileType2(int);
    int IfExistInstructionOnOneTime(int, char);
};
```

6.10 全局控制类

```
#pragma once
#include "overalldata.h"
#include "decoding.h"
#include "execute.h"
#include "fetch.h"
#include "log.h"
#include "memory.h"
#include "panel.h"
#include "pipeline.h"
#include "register.h"
```

```
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <direct.h>
using namespace std;
class CControl
{
private:
	CDecoding decoding;
	CExecute execute;
	CFetch fetch;
	CLog log;
	CMemory memory;
	CPanel panel;
	CPipeline pipeline;
	CRegister reg;
	int current_instruction_num;
public:
	CControl();
	~CControl();
	void AllBegin();
	int PanelControl();
	void PressAnyKeyBackToMainMenu();
	int TransformFromCharToInstruction(char[], int&);
	int IfInstructionOK(char []);
	void Menu11WriteInstruction();
	void Menu12CheckInstructon();
	void Menu13EditInstruction();
	void Menu14ClearInstruction();
	void ResetCurrentInstructionNum();
	void RefershCurrentInstructionNum();
	int RegZeroCheck();
	int ExecuteOneInstruction(int);
	void Menu21ExecuteAllInstruction();
	void Menu22ExecuteInstructionByStep();
	void Menu23CheckExecuteRecord();
	void Menu24ClearExecuteRecord();
	void Menu31InitMemoryWith0();
	void Menu32InitMemoryWithValue();
	void Menu33EditMemory();
	void Menu34CheckMemory();
	void Menu41InitRegisterWith0();
	void Menu42InitRegisterWithValue();
	void Menu43EditRegister();
```

```
    void Menu44CheckRegister();
    void Menu51CheckPipeline();
    void Menu52StepRunPipeline();
    void ShowPC();
    void Menu53ClearPipeline();
    void PanelMenu8Exit();
};
```

详细的代码请见文档“计组设计报告 程序源代码”。

7 结束语

7.1 总结

本系统很好的完成了设计任务书的全部要求，并且在此基础上，自行设计了流水线生成算法，不仅解决了数据相关的问题，也同时解决了结构相关的问题，有一定的创新性。

同时，考虑到控制台界面输出的信息过于复杂，为了能够让使用者更好的理解流水线的推进执行过程，还另外使用 C 语言，基于 graphics 图形库编写了流水线绘制程序。

在为期一个星期的时间内，自己独立完成了系统功能设计、程序框架设计、程序功能代码编写、测试、bug 修复、撰写报告等全部工作，付出了艰苦卓绝的努力，最终得以完成这个令自己满意的程序。

项目设计将要结束，回首这一星期的过程，收获很多。不仅对于“MIPS 指令集”与“流水线”有了更深刻的理解，还进一步巩固了自己的程序分析与设计能力。

MIPS 架构是一种简介、优化、具有高度扩展性的 RISC 架构，使用固定长度的编码。与 Intel 采用的复杂指令系统计算结构（CISC）相比，RISC 具有设计更简单、设计周期更短等优点。这一点在程序编写时给了我很大的便利。对于每一条指令，因为他们的长度相同，且每一段位置固定对应一个变量，使得译码过程变得十分简单。同时，良好的设计保证了指令对于寄存器的操作永远不会出现违规现象，很好的规避了系统错误的风险。这样的设计结构，对于流水线有着很大的益处。

流水线有利于硬件资源的充分利用。设计多级流水线，能够大大的提高硬件的利用率，从而加快程序的运行。但是，在流水线过程中，因为各种原因，难免会出现“数据相关”、“资源相关”、“控制相关”这三类问题。怎样能够有效地解决这些问题，是流水线设计中的一个难题。对于“数据相关”，目前主流的解决方案为“暂停执行”与“超前传送”。总的来说，“超前传送”的效率要比“暂停执行”更高。

最后，让我收获良多的便是，自己对于 C/C++程序设计语言的理解、对于“面向对象”设计思想的理解更加深刻，对于程序功能需求的分析能力、根据需求搭建程序框架的能力进一步加强。在程序编写过程中，大量尝试了新的方法，使用 C++的新特性。虽然全程都是一个人在参与工作，但是这也大大锻炼了自己的动手能力。

当然，对于计算机组成原理与系统结构课程的学习、对于编程能力的锻炼远远不止如此，未来仍需要不断努力，继续学习。

7.2 展望

系统完成了设计任务书的全部要求，且能够很好地运行。对于流水线的设计，在解决数据相关的同时，也解决了资源相关的问题，可以说该算法的设计较为先进。在展示流水线情况时，也使用 C 语言单独编写了流水线绘图程序，能够生动形象的展示流水线的过程。

但是，程序在很多方面仍有待完善，具体表现在以下几个方面：

1.程序使用 C++编写，基于面向对象的思想。但是，由于部件较多，某些功能模块

的类中有较多的函数，编写到后期，容易产生混乱。希望能够有更好的办法，将程序的各个函数理清，方便他人阅读代码。

2.流水线展示程序只有一张简单的图像，希望能够加入更多的细节，让使用者对流水线的过程更加清晰。

3.程序仍使用控制台的窗口，对于内存、寄存器的操作，仍需要使用键盘逐级导航到相应的菜单界面进行操作，操作麻烦。希望能够将该程序移植到 GUI 平台上，提供更直观的菜单面板来进行操作。

4.本程序对流水线阻滞的处理方式为“暂停执行”，另外一种“超前传送”的功能并未涉及。

5.程序并未提供对“控制相关”的处理方法，并未考虑若指令中的五个阶段所占用时间不同的情况。

6.未实现超标量流水线的设计与实现。

以上六个问题是我认为最急切、最想要解决的问题。若日后有机会，希望自己或者看到这一篇结题报告的同学能够将此程序继续完善。

8 参考文献

[1] 包健，冯建文，章复嘉. 计算机组成原理与系统结构[M]. 第二版. 高等教育出版社，2015

[2] 白中英，戴志涛. 计算机组成原理与系统结构[M]. 第五版. 科学出版社，2013

[3] 《MIPS R2000 Assembler Language》PDF 文档

[4] 《第三讲 MIPS32 指令系统》PDF 文档

[5] 《计组课设题目及要求 2018》PDF 文档

[6] C++ constexpr，2017
https://blog.csdn.net/Ink_cherry/article/details/74502671

[7] constexpr-C++11，2016
https://www.cnblogs.com/RainyBear/p/5742171.html

[8] C++ constexpr 类型说明符，2016
https://blog.csdn.net/yhl_leo/article/details/50864210

[9] C++文件操作详解，2017
https://www.cnblogs.com/fengliu-/p/7218970.html

[10] C 和 C++ 文件操作详解，2017
https://blog.csdn.net/freeking101/article/details/60959624

[11] MIPS architecture
https://en.wikipedia.org/wiki/MIPS_architecture

[12] MIPS32 指令集架构简单介绍，2016
https://www.cnblogs.com/blfshiye/p/5137663.html

[13] MIPS 五级流水线，2015
https://blog.csdn.net/eckotan/article/details/46533843

[14] 经典 MIPS 五级流水介绍，2018
https://wenku.baidu.com/view/7f211fc36137ee06eff91836.html

[15] 流水线相关问题及解决方法，2012
https://blog.csdn.net/zhou1519/article/details/8163500

[16] 流水线的相关以及处理方法，2016
https://blog.csdn.net/qq_33094993/article/details/53455466

[17] 流水线数据相关问题，2016
https://www.cnblogs.com/zfyouxi/p/5219160.html