班级中序号 84

**南京航空航天大学**

**计算机组成原理课设报告**

|  |  |
| --- | --- |
| 题 目 | 流水线MIPS CPU设计与仿真验证 |

|  |  |
| --- | --- |
| 学生姓名 | 谢骏鑫 |
| 学 号 | 162110132 |
| 学 院 | 16院 |
| 专 业 | 计算机科学与技术 |
| 班 级 | 1621101 |
| 课程教师 | 施慧彬 |

二〇二三年 六月

**南京航空航天大学**

**实验报告诚信承诺书**

本人郑重声明：所呈交的报告是本人独立进行研究所取得的成果。尽本人所知，除了特别加以标注引用的内容外，本报告不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名： 谢骏鑫 2023年 7 月 1 日

（学号）：1621110132

流水线MIPS CPU设计与仿真验证

摘 要

在学习完计算机组成原理课程之后，在完成了实验大作业对单周期CPU的设计之后，为了进一步理解流水线CPU的工作逻辑和实现方式，提升设计的CPU的工作效率，加强自己对计算机组成原理课程和CPU工作方式的认识和理解，在参考教材设计模式的基础上，独立设计完成了支持36条MIPS基础指令集的五级流水线CPU的设计，并初步解决了这些指令中会涉及到的数据冒险和控制冒险的问题，同时在VSCode信号仿真上测试了实验所给的用于36条MIPS指令的测试程序，结果正确无误。

关键词**：**计算机组成原理，MIPS，流水线CPU

Pipeline MIPS CPU design and simulation verification

Abstract

After learning the course of Principles of computer composition, and completing the design of single-cycle CPU in the experiment, in order to further understand the working logic and implementation mode of pipeline CPU, improve the work efficiency of the designed CPU, and strengthen my knowledge and understanding of the course of principles of computer composition and the working mode of CPU, on the basis of referring to the textbook design mode, Independently designed and completed the design of five-level pipeline CPU supporting 36 MIPS basic instruction sets, and initially solved the problem of data risk and control risk involved in these instructions. At the same time, the test program for 36 MIPS instructions given by the experiment was tested on VSCode signal simulation, and the results were correct.

KEYWORDS: Principles of computer composition，MIPS，Pipeline CPU

目录

[第一章 引言 1](#_Toc139137637)

[1.1 五级流水线CPU设计说明图 1](#_Toc139137638)

[1.2 五级流水线CPU总电路设计图 2](#_Toc139137639)

[1.3 五级流水线CPU各模块定义 3](#_Toc139137640)

[1.3.1 PC模块 3](#_Toc139137641)

[1.3.2 NPC模块 4](#_Toc139137642)

[1.3.3 pc2addr模块 6](#_Toc139137643)

[1.3.4 IM模块 7](#_Toc139137644)

[1.3.5 IF/ID模块 7](#_Toc139137645)

[1.3.6 registers模块 8](#_Toc139137646)

[1.3.7 dm\_in\_mux模块 11](#_Toc139137647)

[1.3.8 ID/Ex模块 12](#_Toc139137648)

[1.3.9 Control模块 16](#_Toc139137649)

[1.3.10 extend模块 20](#_Toc139137650)

[1.3.11 alu\_in1\_mux模块 20](#_Toc139137651)

[1.3.12 alu\_in2\_mux模块 21](#_Toc139137652)

[1.3.13 alu模块 22](#_Toc139137653)

[1.3.14 LoadUse\_detect模块 24](#_Toc139137654)

[1.3.15 Ex/Mem模块 25](#_Toc139137655)

[1.3.16 addr2dm模块 30](#_Toc139137656)

[1.3.17 dm\_4k模块 30](#_Toc139137657)

[1.3.18 sb模块 31](#_Toc139137658)

[1.3.19 Branch\_fc模块 32](#_Toc139137659)

[1.3.20 Jump\_fc模块 33](#_Toc139137660)

[1.3.21 forward\_detect模块 34](#_Toc139137661)

[1.3.22 Mem/Wr模块 38](#_Toc139137662)

[1.3.23 mux\_memtoreg模块 41](#_Toc139137663)

[1.3.24 mips模块 42](#_Toc139137664)

[1.4 五级流水线CPU冒险处理方式 42](#_Toc139137665)

[1.4.1 转发到ALU单元的写后读数据冒险处理 42](#_Toc139137666)

[1.4.2 转发到DM单元的写后读数据冒险处理 46](#_Toc139137667)

[1.4.3 load-use数据冒险处理 47](#_Toc139137668)

[1.4.4 分支跳转简单预测和直接跳转的控制冒险处理 52](#_Toc139137669)

[1.5 五级流水线CPU各个指令控制信号取值标图 59](#_Toc139137670)

[1.6 五级流水线CPU各个指令的设计逻辑 59](#_Toc139137671)

[1.6.1 R型指令 60](#_Toc139137672)

[1.6.2 I型指令 62](#_Toc139137673)

[1.6.3 J型指令 65](#_Toc139137674)

[第二章 代码实现和运行结果验证 65](#_Toc139137675)

[2.1 具体Verilog代码实现 65](#_Toc139137676)

[2.1.1 PC模块 65](#_Toc139137677)

[2.1.2 NPC模块 66](#_Toc139137678)

[2.1.3 pc2addr模块 69](#_Toc139137679)

[2.1.4 IM模块 69](#_Toc139137680)

[2.1.5 IF/ID模块 70](#_Toc139137681)

[2.1.6 registers模块 71](#_Toc139137682)

[2.1.7 dm\_in\_mux模块 73](#_Toc139137683)

[2.1.8 ID/Ex模块 74](#_Toc139137684)

[2.1.9 Control模块 78](#_Toc139137685)

[2.1.10 extend模块 87](#_Toc139137686)

[2.1.11 alu\_in1\_mux模块 87](#_Toc139137687)

[2.1.12 alu\_in2\_mux模块 88](#_Toc139137688)

[2.1.13 alu模块 89](#_Toc139137689)

[2.1.14 LoadUse\_detect模块 93](#_Toc139137690)

[2.1.15 Ex/Mem模块 94](#_Toc139137691)

[2.1.16 addr2dm模块 97](#_Toc139137692)

[2.1.17 dm\_4k模块 97](#_Toc139137693)

[2.1.18 sb模块 98](#_Toc139137694)

[2.1.19 Branch\_fc模块 100](#_Toc139137695)

[2.1.20 Jump\_fc模块 102](#_Toc139137696)

[2.1.21 forward\_detect模块 103](#_Toc139137697)

[2.1.22 Mem/Wr模块 106](#_Toc139137698)

[2.1.23 mux\_memtoreg模块 107](#_Toc139137699)

[2.1.24 mips模块 108](#_Toc139137700)

[2.1.25 tmips模块（testbench） 112](#_Toc139137701)

[2.2 ModelSim上模拟仿真测试指令的信号结果 113](#_Toc139137702)

[第三章 总结与展望 120](#_Toc139137703)

[参考文献 122](#_Toc139137704)

[附录 123](#_Toc139137705)

[附录1 五级流水线CPU测试所用的指令程序 123](#_Toc139137706)

[附录2 实现的36条MIPS指令解析表图 128](#_Toc139137707)

# 第一章 引言

## 1.1 五级流水线CPU设计说明图

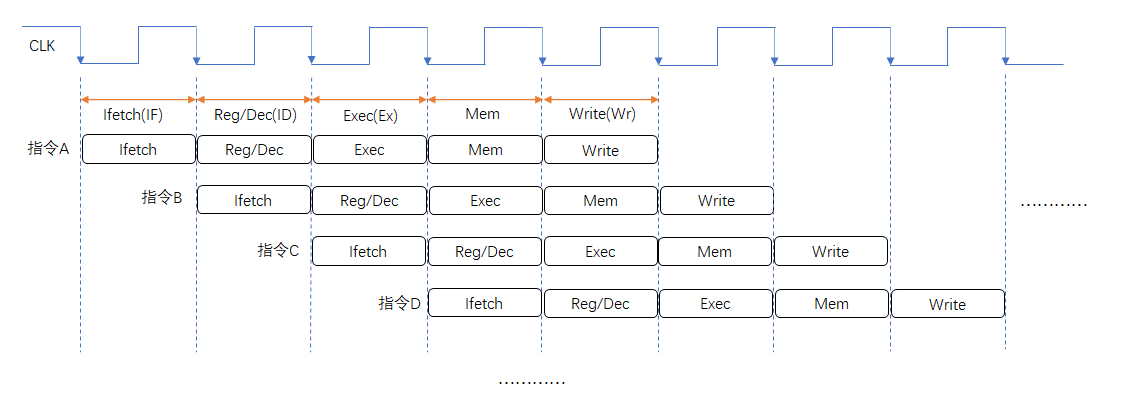


图1.1 五级流水结构

图1.1展示了我在计算机组成原理课程设计的流水线CPU的设计中采用的五级流水线的结构，参考教材上的设计，对要求的36条MIPS基础指令进行逻辑分析之后，使用五级流水的模式进行设计，即将一条指令的执行分为五个阶段来完成，如图所示。

第一个阶段是Ifetch(IF)阶段，即取指阶段，这一阶段主要是PC模块间的交互，主要是将PC的值作为地址（本课设的流水线设计中会将PC的值经过一个模块转换为指令存储器中可以取指的地址值）送到指令存储器IM中取指令，在流水线CPU中会在这一阶段直接计算PC+4送到PC模块作为下一条指令的地址，从而可以在下一个时钟周期到来时直接继续进行下一条指令的Ifetch流水段工作。

第二个阶段是Reg/Dec(ID)阶段，也即译码阶段，这个阶段主要是寄存器堆模块间的交互，这一阶段要对取出来的指令进行译码操作并且送控制单元生成相应的控制信号，并且根据指令中的Rs和Rt的值到寄存器堆中取出相应寄存器堆的值，并通过段寄存器送给下一阶段进行本指令下一阶段的继续执行。

第三个阶段是Exec(Ex)阶段，即执行阶段，不同的指令经过前面的取指和译码阶段已经得到不同的控制信号了，并会通过段寄存器送到这一阶段，这一阶段参与交互的主要模块是ALU算术逻辑单元模块，在控制信号的控制下，指令需要进行算术逻辑运算的部分在这一阶段进行，得到的结果会通过段寄存器送到下一个阶段进行使用。

第四个阶段是Mem阶段，也即访存阶段，参与交互的主要模块单元是数据存储器模块，对于数据存储器的读和写主要在这个阶段进行，如果是读数据存储器，成功读取的值将会通过段寄存器送到下一个阶段进行寄存器写回或者为其他用处使用。在本课设流水线CPU的设计中，对于分支转移和跳转指令，也统一放到了Mem段进行，在这一阶段，所有和跳转有关的信号和数值信息都已经通过段寄存器正确地进行了传递，通过npc模块即可得到跳转之后的地址，不过这对应地会产生控制冒险，解决方式在后续内容会有相关分析。

最后一个阶段，即第五个阶段是Write(Wr)阶段，即写回阶段，这一阶段相对来讲比较简单，主要交互的部件是寄存器写口，这一阶段就是将经过前四个阶段得到的相关的数值信息写回寄存器堆中的目的寄存器，相关逻辑都会在控制信号的控制下进行，而相关的控制信号也会随着段寄存器之间的传递依次向后传递，经历这个阶段成功写回之后，一条指令才算完整地顺利执行完成了。

## 1.2 五级流水线CPU总电路设计图

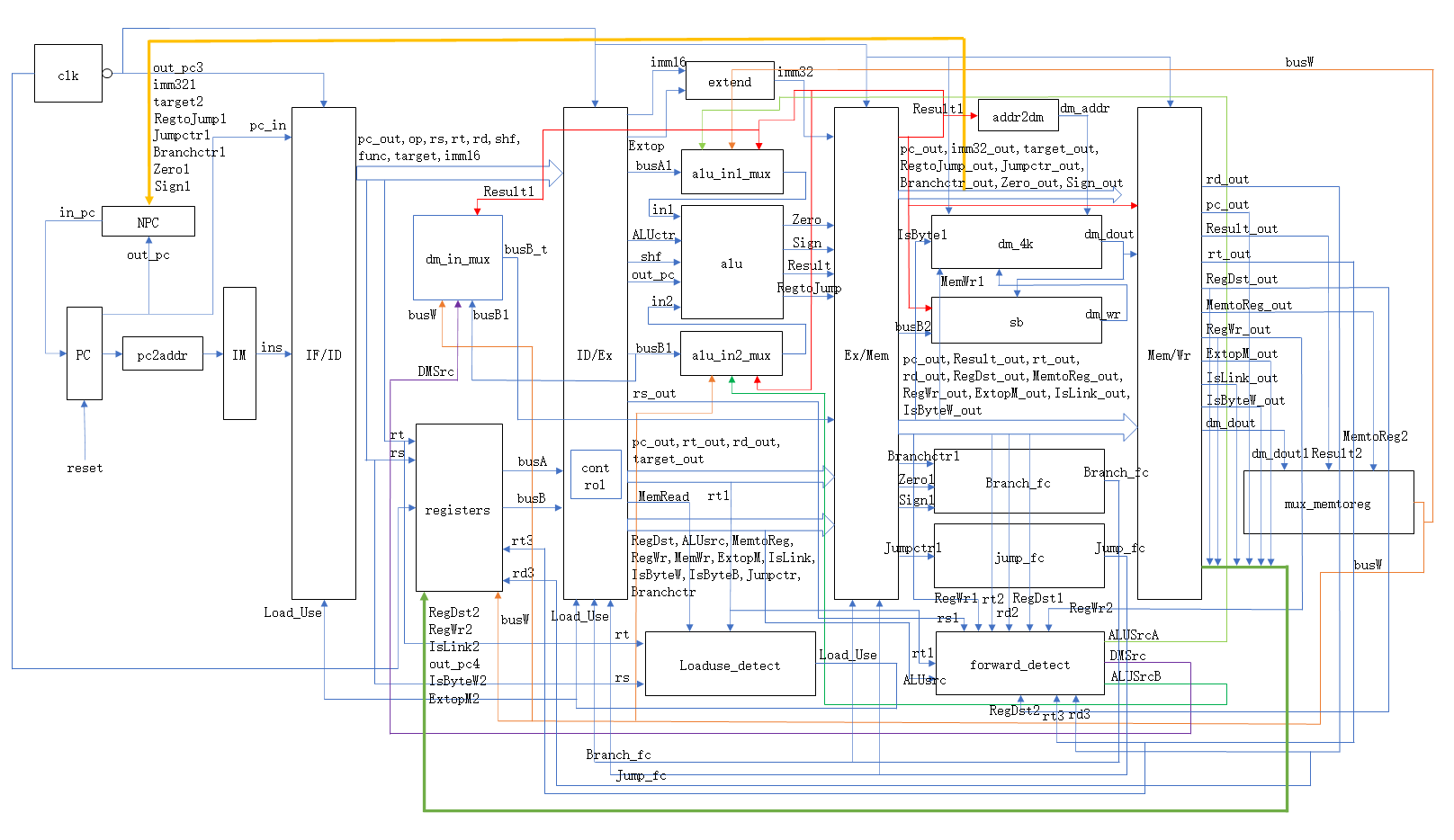


图1.2 五级流水线总电路设计图

图1.2展示了我在这次课程设计中涉及地五级流水线CPU的总电路设计图，图中包含了本流水线CPU中所有的模块以及它们之间的逻辑互联关系，同时图中标注了所有的控制信号和相关数值的信息，它们在流水线中的传递和转发逻辑流程都在图中非常清楚地体现出来了（具体浏览时可以将文档放大从而更加清楚地观察其中的逻辑）。

## 1.3 五级流水线CPU各模块定义

需要说明的是，各个模块虽然是独立编写的，最终在顶层模块mips模块进行例化和整合连接，但是各个模块之间都是有紧密联系的，是相互服务的，因此不同模块间的某个接口可能刚好对应其他模块的某个接口，因此在下面对每个模块进行描述和接口定义分析时，一些接口和信号在某个模块中详细说明之后，为了避免重复解释，使得报告内容略显臃肿，当这个接口和信号在其他模块再次出现时将会一笔带过，不再详细说明了，因此若有某个信号或者接口描述简单的话，在其他模块基本会有它的详细说明。

### 1.3.1 PC模块

（1）基本描述

在我的五级流水线CPU设计逻辑中，PC模块的作用是来选择下一条需要取指令的PC值，从功能接口上来看，它的作用就是传入in\_pc,在时钟clk上升沿更新pc作为out\_pc输出进行下一条指令的取指并且运行。而in\_pc的计算逻辑主要在npc模块中进行，这基本和实验中的单周期CPU的PC模块设计逻辑保持了一致，其中增加了Load\_Use信号，主要是为了处理load-use数据冒险使用的，具体逻辑可以参考1.4节冒险处理逻辑的分析，在PC模块的作用就是为了阻塞PC更新。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| in\_pc[31:0] | I | 传入的PC值用来更新计算下一条的PC值 |
| clk | I | 时钟信号 |
| reset | I | 复位信号，用来将PC值复位更新为0（使用扩展要求，其实是更新为0x00003000） |
| out\_pc[31:0] | O | PC的输出，下一条指令的取指地址 |
| Load\_Use | I | 解决load-use数据冒险需要使用的信号，用于阻塞PC值的更新 |

### 1.3.2 NPC模块

（1）基本描述

npc模块，就是所说的“下地址”逻辑模块，在指令的执行过程中，下一条指令的不可能只是通过PC+4计算得到的，还存在着分支跳转和J型的跳转指令，这些都需要重新计算下一条PC的地址值，具体就是在npc模块里面实现，这里传入了控制分支和跳转的控制信号以及相关判断的标志位，并根据实验测试要求的方式进行PC的重新拼接和赋值，这里基本都是实验设计的单周期CPU保持一致，但是在我设计的五级流水线CPU中，支跳转的指令和直接跳转的指令都需要进行到Mem阶段才可以计算得到正确的跳转指令，而流水线的执行不可能等到对应的跳转指令执行到Mem阶段才继续取指，这会产生控制冒险，具体解决逻辑可以参看1.4节，接口上体现为新增了pc\_add接口，这个是相当于PC自增的逻辑，保证流水线永远是按照PC+4去取下一条指令的PC，而out\_pc是经过流水线段寄存器保持在后面阶段产生的，是用于对分支和跳转这类指令执行到Mem阶段后能够在npc模块中正确计算得到跳转地址的。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| pc\_add[31:0] | I | 当流水线CPU执行的阶段还没有办法判断是否可以跳转时，传递的pc\_add就是上一条取指的PC地址值，并且经过PC+4得到下一条指令的PC从而取指 |
| out\_pc[31:0] | I | 经过Ex阶段从前面的流水段寄存器保存和传递得到的PC值会被送到这个接口，为了让分支和跳转类指令在执行到Mem阶段时能够在npc模块正确计算得到下一条指令的地址。 |
| imm32[31:0] | I | I型指令解析出来的立即数字段并且扩展为32位的结果，用来进行分支跳转PC的计算 |
| target[25:0] | I | J型指令解析出来的target字段，用来进行直接跳转PC的计算 |
| RegtoJump[31:0] | I | 跳转寄存器指令的需要，传递的某个需要跳转的寄存器的值，用来更新PC值 |
| Jumpctr[1:0] | I | 跳转指令的控制信号，00控制不跳转，01控制类j指令的跳转，10控制类jr指令的跳转 |
| Branchctr[2:0] | I | 分支指令的控制信号，根据要实现的指令实现，主要控制等于、不等、大于等于0、大于0、小于等于0和小于0的分支跳转 |
| Zero | I | ALU计算得到的标志位信息，用来表明要跳转判断的两个数是否相等 |
| Sign | I | ALU计算得到的标志位信息，用来表明要跳转判断的两个数的大小关系 |
| in\_pc | O | 根据上述控制逻辑最终得到的正确的下一条指令的PC地址，用来传递给PC模块得到取指PC |

### 1.3.3 pc2addr模块

（1）基本描述

正如模块名字一样，将32位PC值转换为可以从指令存储器读取对应指令的10位地址值，这是由于指令存储器im\_4k的接口要求的，指令存储器是以一个字为一个单位存放，而PC值是按照字节进行读取，因此在取指令时需要先对PC值做一个转换才可以。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| out\_pc[31:0] | I | 传入的32位PC值 |
| addr[11:2] | O | 输出的10位地址值，能够从指令存储器的接口中正确读取传入PC对应的指令 |

### 1.3.4 IM模块

（1）基本描述

指令存储器模块，仍旧保留了按照大实验要求的指令存储器接口模块的设计，具体的存储器是1024个字大小的，故只有10位的取指地址，这是经过pc2addr模块得到的结果，输出就是32位的取得的具体指令编码。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[11:2] | I | 10位取指地址 |
| dout[31:0] | O | 取得的具体的指令的编码 |

### 1.3.5 IF/ID模块

（1）基本描述

这个模块本质上就是充当从IF段到ID段的段寄存器作用，其作用就是将在IF段生成的相关信号在后面阶段需要使用的通过这个段寄存器向后传递，保证指令在下一个阶段能够使用到正确的值，正如下面接口定义所描述的那样，经过IF段译码的工作后，这个段寄存器保存的就是对指令译码之后的结果，传入的ins就是经PC模块得到的PC到指令存储器取得的指令，在实际模块实现中，我将译码的过程合并到了这个模块中，因此这个模块同时兼并译码工作和段寄存器的作用。其中，Load\_Use输入信号是解决load-use数据冒险使用的，有关冒险处理的逻辑在下一小节会专门分析。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| pc\_in[31:0] | I | 传入的PC值，由于后续的阶段需要使用PC的数值信息，故需要在段寄存器中保存 |
| ins[31:0] | I | 根据PC值取出来的指令 |
| pc\_out[31:0] | O | 经段寄存器保存传递给下一阶段的PC值 |
| op[5:0] | O | 译码得到的传递给下一阶段使用的op字段 |
| rs[4:0] | O | 译码得到的传递给下一阶段使用的rs字段 |
| rt[4:0] | O | 译码得到的传递给下一阶段使用的rt字段 |
| rd[4:0] | O | 译码得到的传递给下一阶段使用的rd字段 |
| shf[4:0] | O | 译码得到的传递给下一阶段使用的shf字段 |
| func[5:0] | O | 译码得到的传递给下一阶段使用的func字段 |
| imm16[15:0] | O | 译码得到的传递给下一阶段使用的imm16字段 |
| target[25:0] | O | 译码得到的传递给下一阶段使用的target字段 |
| Load\_Use | I | 解决Load\_Use数据冒险需要使用的信号 |

### 1.3.6 registers模块

（1）基本描述

非常核心和重要的一个模块——寄存器堆模块，因为所有指令归根结底无外乎就是读写寄存器，所以控制着整个CPU的32个寄存器的寄存器堆的设计显得很重要，同时又有较大的自主设计的灵活性。我在设计单周期CPU的时候，将所有相关的不同的读写寄存器类型都集中到了这个模块进行处理，并且通过传入相关的控制信号，来控制相关的读写逻辑，但是转移到当前这个五级流水线CPU的设计中，为了更好地契合流水线之间的工作方式和流水段之间保存和传递信号的逻辑，我对单周期CPU设计的寄存器堆做了一定的修改，主要是将读寄存器的两个寄存器编号端口和写寄存器的两个寄存器编号端口分开进行设计，为了避免原先为实现sb指令设计的逻辑产生冒险，将原先为了实现sb指令而处理写入寄存器的数据重新整合到了sb模块中，并在寄存器堆中去除了像IsByteB、dm\_addr这样的接口，使得整个寄存器堆模块的设计更加统一和简洁。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号，在时钟上升沿的会进行寄存器写使能判断并决定是否进行寄存器写的操作 |
| rs[4:0] | I | rs寄存器编号，读这个寄存器送出对应的值到busA端口 |
| rt[4:0] | I | rt寄存器编号，读这个寄存器送出对应的值到busB端口 |
| w\_rt[4:0] | I | 最终在写回寄存器时的rt寄存器的编号，主要是需要经过段寄存器的保存和传递之后 |
| w\_rd[4:0] | I | 最终在写回寄存器时的rd寄存器的编号，主要是需要经过段寄存器的保存和传递之后 |
| RegDst | I | 控制信号，决定是写rt寄存器还是rd寄存器，为1时写rd寄存器，反之写rt寄存器 |
| RegWr | I | 寄存器写使能信号，控制是否可以写寄存器 |
| IsLink | I | 是否需要链接控制信号，为真时需要将当前的PC+4送入31号ra寄存器 |
| out\_pc[4:0] | I | 传入的当前的PC值用来实现链接更新ra寄存器的需要 |
| IsByteW | I | 控制信号，控制是否是按字节将内容写入寄存器，为真时表示按字节写入 |
| ExtopM | I | 控制在按字节写寄存器的时候是否需要对写入的字节进行符号扩展，为真时表示需要进行符号扩展 |
| busA[31:0] | O | 寄存器堆从rs寄存器读出来的busA端口的值 |
| busB[31:0] | O | 寄存器堆从rt寄存器读出来的busB端口的值 |
| busW[31:0] | I | 最终在流水线的写回Wr阶段需要写回寄存器堆的数据 |

### 1.3.7 dm\_in\_mux模块

（1）基本描述

为了解决写后读这类数据冒险且和sw类装载字（或者字节）发生数据相关的情况而设计的模块，类似1.3.11和1.3.12增加的在ALU运算单元的两个源操作数端口前增加的多路选择器，这个模块是增加在数据存储器写端口前的多路选择器，冒险处理和判断的逻辑详见1.4节，相关信号的具体控制功能可见1.3.21 forward\_detect模块的定义介绍，这里不再赘述了。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| busB[31:0] | I | 输入的busB信息，即从寄存器堆读出来的busB内容 |
| Result[31:0] | I | 输入的Result信息，即经Ex阶段运算得到的结果 |
| mem\_result[31:0] | I | 输入的mem\_result信息，即经Mem阶段得到的最终写回寄存器堆的busW信息 |
| DMSrc[1:0] | I | 输入的DMSrc信号，用来控制选择哪个值作为输出从而最终写入数据存储器 |
| dm\_write[31:0] | O | 输出的最终写到数据存储器的内容 |

### 1.3.8 ID/Ex模块

（1）基本描述

此模块主要是充当将从ID段生成和传送过来的信号保存下来并送给Ex段使用的段寄存器作用，同时在具体实现时，从ID段译码工作完成后得到的各个字段信息可以用来生成整个五级流水线CPU需要使用的控制信号，因此我将生成控制信号的控制器单元合并到了这个模块中，因此这个模块不仅充当了从ID段到Ex段段寄存器的角色，也兼并了控制器模块的角色，实现上就是在模块中例化了控制器模块，因此这个模块会输出所有的控制信号，具体的控制信号的作用和描述在后面的控制器模块会有描述。从段寄存器的角度来看，ID段会从寄存器堆取数，同时传递到ID段的一些译码得到的字段信息在后续的阶段和模块中也需要使用，故也需要在段寄存器中保存并传递给下一阶段。此外，Load\_Use、Branch\_fc、Jump\_fc这三个输入信号是冒险处理使用的，相关逻辑和描述在下一小节会进行分析。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| pc\_in[31:0] | I | 从前一阶段传递过来的PC值的信息，后续阶段需要使用，保存在段寄存器中并送给下一阶段使用 |
| op\_in[5:0] | I | 从前一阶段传递过来的指令的op字段信息，用于生成控制信号 |
| func\_in[5:0] | I | 从前一阶段传递过来的指令的func字段信息，用于生成控制信号 |
| rt\_in[4:0] | I | 从前一阶段传递过来的指令的rt字段信息，用于生成控制信号，同时后续阶段也需要使用这个rt寄存器的信息 |
| rd\_in[4:0] | I | 从前一阶段传递过来的指令的rd字段信息，后续阶段需要使用这个rd寄存器的信息 |
| rs\_in[4:0] | I | 从前一阶段传递过来的指令的rs字段信息，后续阶段需要使用这个rs寄存器的信息 |
| shf\_in[4:0] | I | 从前一阶段传递过来的指令的shf字段信息，后续阶段需要使用这个shf移位量的信息 |
| target\_in[25:0] | I | 从前一阶段传递过来的指令的target字段信息，后续阶段比如跳转时需要使用这个target的信息 |
| imm16\_in[15:0] | I | 从前一阶段传递过来的指令的imm16字段信息，即I型指令的立即数字段，后续阶段比如ALU计算单元需要使用这个信息 |
| busA\_in[31:0] | I | 从前一阶段传递过来的经过寄存器堆取数后的busA信息，用于在后续阶段进行使用，主要是参与ALU运算 |
| busB\_in[31:0] | I | 从前一阶段传递过来的经过寄存器堆取数后的busB信息，用于在后续阶段进行使用，主要是参与ALU运算和写数据存储器等操作 |
| rt\_out[4:0] | O | 经段寄存器保存传递给下一阶段的rt信息 |
| rd\_out[4:0] | O | 经段寄存器保存传递给下一阶段的rd信息 |
| rs\_out[4:0] | O | 经段寄存器保存传递给下一阶段的rs信息 |
| pc\_out[31:0] | O | 经段寄存器保存传递给下一阶段的pc值的信息 |
| shf\_out[4:0] | O | 经段寄存器保存传递给下一阶段的shf信息 |
| target\_out[25:0] | O | 经段寄存器保存传递给下一阶段的target信息 |
| imm16\_out[15:0] | O | 经段寄存器保存传递给下一阶段的imm16信息 |
| busA\_out[31:0] | O | 经段寄存器保存传递给下一阶段的busA信息 |
| busB\_out[31:0] | O | 经段寄存器保存传递给下一阶段的busB信息 |
| RegDst | O | 控制器生成RegDst控制信号，送到后续阶段进行电路的逻辑控制 |
| ALUsrc | O | 控制器生成ALUsrc控制信号，送到后续阶段进行电路的逻辑控制 |
| MemtoReg | O | 控制器生成MemtoReg控制信号，送到后续阶段进行电路的逻辑控制 |
| RegWr | O | 控制器生成RegWr控制信号，送到后续阶段进行电路的逻辑控制 |
| MemWr | O | 控制器生成MemWr控制信号，送到后续阶段进行电路的逻辑控制 |
| Extop | O | 控制器生成Extop控制信号，送到后续阶段进行电路的逻辑控制 |
| ExtopM | O | 控制器生成ExtopM控制信号，送到后续阶段进行电路的逻辑控制 |
| IsLink | O | 控制器生成IsLink控制信号，送到后续阶段进行电路的逻辑控制 |
| IsByteW | O | 控制器生成IsByteW控制信号，送到后续阶段进行电路的逻辑控制 |
| IsByteB | O | 控制器生成IsByteB控制信号，送到后续阶段进行电路的逻辑控制 |
| ALUctr[3:0] | O | 控制器生成ALUctr控制信号，送到后续阶段进行电路的逻辑控制 |
| Jumpctr[1:0] | O | 控制器生成Jumpctr控制信号，送到后续阶段进行电路的逻辑控制 |
| Branchctr[2:0] | O | 控制器生成Branchctr控制信号，送到后续阶段进行电路的逻辑控制 |
| MemRead | O | 控制器生成MemRead控制信号，送到后续阶段进行电路的逻辑控制 |
| Load\_Use | I | 解决load-use数据冒险需要的信号 |
| Branch\_fc | I | 解决分支跳转控制冒险需要的信号 |
| Jump\_fc | I | 解决直接跳转控制冒险需要的信号 |

### 1.3.9 Control模块

（1）基本描述

本模块是生成五级流水线CPU所有控制信号的控制器模块，五级流水结构上它是在ID译码阶段生成的，实现上合并到了ID\_Ex模块中进行了，这一部分我会在下表中描述每个控制信号的作用，大部分信号和单周期CPU设计中是保持一致的，在其他地方当需要引用这些信号时将不再具体阐释其具体功能逻辑了。同样，在五级流水线结构中，部分控制信号根据其使用的场景和范围需要经过段寄存器在不同阶段进行传送，送到需要用到的地方进行使用。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| op[5:0] | I | 传入的指令op字段信息 |
| func[5:0] | I | 传入的指令func段信息，一般为R型指令所使用 |
| rt[4:0] | I | 传入的指令rt段信息，bgez和bltz指令区分的需要 |
| RegDst | O | 控制信号，控制写哪个寄存器，为1时写rd寄存器，否则写rt寄存器 |
| ALUsrc | O | 控制信号，决定ALU单元使用哪个值进行运算，为1时选择经扩展器的立即数参与运算，否则选择从寄存器堆读出来的busB参与运算，在实现冒险处理时ALU选择的源操作数会有一定的区别，详见下一小节讲解冒险处理的逻辑 |
| MemtoReg | O | 控制信号，控制从数据存储器读出的值是否需要写寄存器，为1表示选择从数据存储器中读出来的值写寄存器，否则选择ALU运算得到的结果写寄存器 |
| RegWr | O | 控制信号，寄存器写使能，控制是否可以写寄存器堆，为1表示需要将内容写到寄存器中，否则不允许写 |
| MemWr | O | 控制信号，存储器写使能，控制是否可以写数据存储器，为1表示需要将内容写到数据存储器中，否则不允许写 |
| Extop | O | 控制信号，控制是否需要对16位立即数作符号扩展，为1表示需要作符号扩展，否则作零扩展 |
| ExtopM | O | 控制信号，实现lb指令的需要，控制对取出来的字节是否作符号扩展，为1表示需要对字节作符号扩展，否则作零扩展 |
| IsLink | O | 控制信号，链接需要，控制是否需要将pc+4的值写入31号寄存器（即ra寄存器），为1表示在寄存器堆单元写使能打开的情况下将对应的PC数值写入31号ra寄存器，反之不作 |
| IsByteW | O | 控制信号，控制写寄存器时是写入字（4个字节）还是写入字节，为1时表示要写入寄存器的内容是对应传入的写数据信息的低8位信息，否则正常写入字 |
| IsByteB | O | 控制信号，sb指令需要，控制sb模块是选择将字的内容写入数据存储器还是将字节的内容写入数据存储器，为1时表示将字节内容写入数据存储器，否则正常写入字 |
| ALUctr[3:0] | O | 控制信号，4位ALUctr，控制ALU单元执行何种运算，具体哪种信号控制何种运算可以参考1.5节每条指令的控制信号的取指信息和第二章所附的具体代码ALU模块的实现 |
| Jumpctr[1:0] | O | 控制信号，控制需要跳转的方式，是直接跳转还是寄存器跳转，00表示不跳转，01表示直接跳转，10表示寄存器跳转 |
| Branchctr[2:0] | O | 控制信号，控制分支跳转的类型，根据不同类型判断相对于的分支跳转条件，000表示不需要分支跳转，001表示相等跳转，010表示不等跳转，011表示大于等于0跳转，100表示小于0跳转，101表示大于0跳转，110表示小于等于0跳转 |
| MemRead | O | 相较于单周期CPU新增加的一个控制信号，主要是用于判断这一条指令是否是装载字或者装载字节类型的指令，用于处理解决load-use数据冒险 |

### 1.3.10 extend模块

（1）基本描述

扩展器模块，就是对立即数进行扩展的，I型指令中解析得到的立即数是16位的，参与运算的元素都是32位的，并且需要根据相关指令的需求决定是零扩展还是符号扩展，这是由控制信号Extop决定的。这个模块在课设的五级流水线CPU的设计中依旧和是实验中的单周期CPU保持了一致。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| imm16[15:0] | I | 传入的16位立即数 |
| Extop | I | 控制是进行零扩展还是符号扩展 |
| imm32[31:0] | O | 输出的扩展完成的32位立即数 |

### 1.3.11 alu\_in1\_mux模块

（1）基本描述

解决数据冒险和转发的需要，在ALU算术逻辑运算单元的第一个源操作数端口增加一个多路选择器，从而进行相关数值信息的转发解决像写后读这样的数据冒险，具体的冒险处理逻辑可以参见1.4节冒险处理的分析，具体信号的功能作用在1.3.21的forward\_detect模块的接口定义有清楚的说明，这里不再赘述了。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| busA[31:0] | I | 输入的用于选择的busA信息，即读寄存器堆的结果 |
| Result[31:0] | I | 输入的用于选择的Result信息，提前转发的经Ex阶段的运算得到的结果 |
| mem\_result[31:0] | I | 输入的用于选择的mem\_result信息，提前转发的经Mem阶段确定的要写回寄存器堆的结果 |
| ALUSrcA[1:0] | I | 输入的控制选择哪一条信息作为输出的ALUSrcA信号 |
| in1[31:0] | O | 输出的最终送到ALU运算单元第一个源操作数端口的操作数 |

### 1.3.12 alu\_in2\_mux模块

（1）基本描述

和1.3.11 alu\_in1\_mux模块一样的，区别是用来选择ALU运算单元第二个源操作数端口的操作数。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| busB[31:0] | I | 输入的用于选择的busB信息，即读寄存器堆的结果 |
| Result[31:0] | I | 输入的用于选择的Result信息，提前转发的经Ex阶段的运算得到的结果 |
| mem\_result[31:0] | I | 输入的用于选择的mem\_result信息，提前转发的经Mem阶段确定的要写回寄存器堆的结果 |
| imm32[31:0] | I | 输入的用于选择的imm32信息，即经过ID阶段并且经扩展器按照对应控制信号进行扩展后的32位立即数 |
| ALUSrcB[1:0] | I | 输入的控制选择哪一条信息作为输出的ALUSrcB信号 |
| in2[31:0] | O | 输出的最终送到ALU运算单元第二个源操作数端口的操作数 |

### 1.3.13 alu模块

（1）基本描述

五级流水线CPU进行算术逻辑运算的核心单元，整个ALU算术逻辑运算单元模块和实验设计的单周期CPU是一致的，并没有任何改动。所有指令需要的各种运算都在这个单元中进行，支持对传入的数值进行无符号加法、无符号减法、高位拼接、小于比较置位、按位与、按位或非、按位或、按位异或、逻辑左移、逻辑右移、算术右移等运算。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ALUctr[3:0] | I | 控制信号，4位ALU运算控制信号，控制当前的ALU单元对传入的值进行何种运算 |
| in1[31:0] | I | ALU单元的输入值的1号端口，不过传入它的值需要经过一个用于处理了写后读冒险的三路多选一选择器 |
| in2[31:0] | I | ALU单元的输入值的2号端口，不过传入它的值需要经过一个用于处理了写后读冒险的四路多选一选择器 |
| shf[4:0] | I | R型指令中非变量的移位运算需要的shamt字段信息 |
| out\_pc[31:0] | I | 链接指令的需要将PC的值写入31号寄存器，因此传入这个值用来特殊的运算 |
| Zero | O | ALU单元运算生成的标志位信息，Zero为真表示传入的两个计算值相等，反之则不等 |
| Sign | O | ALU单元运算生成的标志位信息，Sign为真表示in1大于in2，反之则小于 |
| Result[31:0] | O | ALU单元对传入的in1和in2运算生成的结果 |
| RegtoJump[31:0] | O | 寄存器跳转的需要，需要读取寄存器的值，作为ALU的输出统一处理 |

### 1.3.14 LoadUse\_detect模块

（1）基本描述

在数据冒险中，转发是能够解决大部分RAW（写后读）的问题，但是对于像lw（lb、lbu）这样的加载字类型的指令后面紧跟的R型或I型运算类指令的相关性问题，由于lw指令只有在Mem段结束时才能得到数据存储器中的结果，因此需要重新设计相关的逻辑来解决load-use产生的冒险问题。这个模块就是用来判断是否产生了load-use数据冒险，相关判断逻辑和处理逻辑可见1.4节或者第二章所附具体的实现代码。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ID\_Ex\_MemRead | I | 传入的从ID段到Ex段的MemRead控制信号，用来判断上一条指令是否时load类型的指令 |
| ID\_Ex\_rt[4:0] | I | 传入的从ID段到Ex段的rt寄存器的编号信息，用来判断指令间是否发生了load-use数据冒险 |
| IF\_ID\_rs[4:0] | I | 传入的从IF段到ID段的rs寄存器的编号信息，用来判断指令间是否发生了load-use数据冒险 |
| IF\_ID\_rt[4:0] | I | 传入的从IF段到ID段的rt寄存器的编号信息，用来判断指令间是否发生了load-use数据冒险 |
| Load\_Use | O | 经模块内部判断逻辑之后输出的Load\_Use信号，为真表示发生了load-use数据冒险，需要进行相关处理，否则没有发生数据冒险 |

### 1.3.15 Ex/Mem模块

（1）基本描述

本模块主要是充当从Ex阶段将相关信号保存下来并传送给下一阶段使用的段寄存器的作用，实现上本模块并没有什么需要特别注明的地方，就是将从前面的阶段传送过来的值保存并传送给下一阶段进行使用，由于这个段寄存器承接的是Ex执行阶段，下一阶段是Mem访存阶段，很多数值信息都需要被进行使用和运算，因此这个模块涉及的接口信号数量较为庞大，但是每个信号的相关作用从其命名中就可以看出来，可以参考如下表格。其中还有两个输入的信号Branch\_fc和Jump\_fc，主要是为了解决跳转指令产生的控制冒险，具体可以参见1.4节有关冒险处理的具体逻辑。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| pc\_in[31:0] | I | 前面阶段传送过来的PC值 |
| rt\_in[4:0] | I | 前面阶段传送过来的rt寄存器信息 |
| rd\_in[4:0] | I | 前面阶段传送过来的rd寄存器信息 |
| target\_in[25:0] | I | 前面阶段传送过来的target字段信息 |
| imm32\_in[31:0] | I | 前面阶段传送过来的经过扩展器扩展的32为立即数imm32信息 |
| busB\_in[31:0] | I | 前面阶段传送过来的busB信息，已经经过冒险处理，并且送到下一阶段用于写数据存储器 |
| Zero\_in | I | 前面阶段传送过来的经过ALU运算单元得到用于控制分支跳转的Zero标志位信息 |
| Sign\_in | I | 前面阶段传送过来的经过ALU运算单元得到用于控制分支跳转的Sign标志位信息 |
| Result\_in[31:0] | I | 前面阶段传送过来的经过ALU运算单元得到的运算结果的信息 |
| RegtoJump\_in[31:0] | I | 前面阶段传送过来的经过ALU运算单元得到的用于寄存器跳转的寄存器对应的跳转地址的信息 |
| RegDst\_in | I | 前面阶段传送过来的RegDst控制信号 |
| MemtoReg\_in | I | 前面阶段传送过来的MemtoReg控制信号 |
| RegWr\_in | I | 前面阶段传送过来的RegWr控制信号 |
| MemWr\_in | I | 前面阶段传送过来的MemWr控制信号 |
| ExtopM\_in | I | 前面阶段传送过来的ExtopM控制信号 |
| IsLink\_in | I | 前面阶段传送过来的IsLink控制信号 |
| IsByteW\_in | I | 前面阶段传送过来的IsByteW控制信号 |
| IsByteB\_in | I | 前面阶段传送过来的IsByteB控制信号 |
| Jumpctr\_in[1:0] | I | 前面阶段传送过来的Jumpctr控制信号 |
| Branchctr\_in[2:0] | I | 前面阶段传送过来的Branchctr控制信号 |
| pc\_out[31:0] | O | 经过段寄存器保存然后送下一阶段使用的PC值 |
| rt\_out[4:0] | O | 经过段寄存器保存然后送下一阶段使用的rt寄存器信息 |
| rd\_out[4:0] | O | 经过段寄存器保存然后送下一阶段使用的rd寄存器信息 |
| target\_out[25:0] | O | 经过段寄存器保存然后送下一阶段使用的target字段信息，用于跳转计算下一条PC值的需要 |
| imm32\_out[31:0] | O | 经过段寄存器保存然后送下一阶段使用的且经过扩展的32为立即数信息，主要也是用于分支跳转计算下一条PC的偏移量 |
| busB\_out[31:0] | O | 经过段寄存器保存然后送下一阶段使用的busB信号，主要用于写数据存储器 |
| Zero\_out | O | 经过段寄存器保存然后送下一阶段使用的Zero标志位信息，用于分支跳转的判断 |
| Sign\_out | O | 经过段寄存器保存然后送下一阶段使用的Sign标志位信息，用于分支跳转的判断 |
| Result\_out[31:0] | O | 经过段寄存器保存然后送下一阶段使用的经ALU运算单元计算得到的结果Result信息 |
| RegtoJump\_out[31:0] | O | 经过段寄存器保存然后送下一阶段使用的用于寄存器跳转的从相应寄存器读出来的跳转地址信息 |
| RegDst\_out | O | 经过段寄存器保存然后送下一阶段使用的RegDst控制信号信息 |
| MemtoReg\_out | O | 经过段寄存器保存然后送下一阶段使用的MemtoReg控制信号信息 |
| RegWr\_out | O | 经过段寄存器保存然后送下一阶段使用的RegWr控制信号信息 |
| MemWr\_out | O | 经过段寄存器保存然后送下一阶段使用的MemWr控制信号信息 |
| ExtopM\_out | O | 经过段寄存器保存然后送下一阶段使用的ExtopM控制信号信息 |
| IsLink\_out | O | 经过段寄存器保存然后送下一阶段使用的IsLink控制信号信息 |
| IsByteW\_out | O | 经过段寄存器保存然后送下一阶段使用的IsByteW控制信号信息 |
| IsByteB\_out | O | 经过段寄存器保存然后送下一阶段使用的IsByteB控制信号信息 |
| Jumpctr\_out[1:0] | O | 经过段寄存器保存然后送下一阶段使用的Jumpctr控制信号信息 |
| Branchctr\_out[2:0] | O | 经过段寄存器保存然后送下一阶段使用的Branchctr控制信号信息 |
| Branch\_fc | I | 解决分支跳转控制冒险需要的信号 |
| Jump\_fc | I | 解决直接跳转控制冒险需要的信号 |

### 1.3.16 addr2dm模块

（1）基本描述

正如模块名字命名一样，这是将32位的地址转换为10位的可以用来读写数据存储器的地址，在指令执行逻辑中计算得到的地址都是32位的，并且是按照字节进行编址的，因此在按字存储的数据存储器中，需要取32位地址中的[11:2]位进行转换。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| in\_addr[31:0] | I | 传入的32位地址 |
| addr[11:2] | O | 输出可以读写数据存储器的10位地址 |

### 1.3.17 dm\_4k模块

（1）基本描述

数据存储器模块，接口的设计和大实验要求的依旧保持了一致，作为五级流水线CPU的数据存储单元，存储单元按照一个字（4个字节）为单位进行设计，大小一共1024个字，所以地址只需要10位，支持读存储单元的内容和在写使能的控制下将内容写进存储单元中去。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[11:2] | I | 传入的控制读写内容的地址 |
| din[31:0] | I | 要写入的一个字的内容 |
| we | I | 存储器写使能信号，控制是否可以将内容写进存储器 |
| clk | I | 时钟控制信号，在时钟上升沿检查是否可写 |
| dout[31:0] | O | 从数据存储器指定地址读出来的内容 |

### 1.3.18 sb模块

（1）基本描述

正如这个模块命名一样，这个模块是为了实现sb指令单独设计的，即将从寄存器读出来的值的最后8位的字节信息写入数据存储器，由于只能修改指定地址处的字节信息，并且是按照小端方式进行访问和存储，所以自己采取的逻辑是对所有要写数据存储器的数据都经过这个模块进行逻辑处理，通过IsByteB信号控制是否需要按字节写入，并且将处理之后得到的真正要写入数据存储器的值输出并写入数据存储器。相比较在实验中单周期CPU将sb指令放到寄存器堆中去一并处理，这个方式更加清楚明晰，并且原来的方式交互的两个模块不在同一个流水线阶段，会产生难以处理的冒险问题，而采取单独封装称为sb模块并且都安排到Mem段进行统一处理，也避免了原先设计的冒险问题。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| dm\_addr[31:0] | I | 通过ALU单元计算得到的需要访问数据存储器的32位地址信息，当然是经过流水段寄存器保存和传送得到的数据 |
| busB[31:0] | I | 通过读寄存器堆得到的busB数据信息，当然是经过流水段寄存器保存和传送得到的数据 |
| dm\_read[31:0] | I | 读数据存储器得到的相关字的内容，由于sb指令实现设计的逻辑需要 |
| dm\_write[31:0] | O | 最终经过模块内的逻辑处理得到的写入数据存储器的32位，即一个字大小的内容 |
| IsByteB | I | 传入的控制信号，用来控制是否需要写入字节，实现sb指令的需要 |

### 1.3.19 Branch\_fc模块

（1）基本描述

在五级流水线的设计中，除了需要解决像写后读这样的数据冒险之外，还需要解决由于分支和跳转这类指令带来的控制冒险，即在正常情况下，指令在流水线中总是按照PC+4这样的顺序执行，但是当遇到像Branch或者Jump这样的跳转指令时就会产生控制冒险，改变指令的执行顺序，需要让流水线中正常执行的指令阻塞，正如这个模块的命名，是用来判断指令是否会发生分支跳转，产生相关的信号进行控制，如果Branch\_fc为真，则需要阻塞流水线，至于控制冒险的具体处理逻辑详见1.4节的分析，下面模块用到的信息都可以在npc模块的介绍中看到详细的介绍，本质上就是一个跳转判断的逻辑。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Branchctr[2:0] | I | 传入的分支跳转指令的控制信号，控制当前要判断的指令是否是分支跳转指令，以及是哪一种类型的分支跳转指令 |
| Zero | I | 判断跳转指令是否相等的条件 |
| Sign | I | 判断跳转指令对于操作数的大小关系 |
| Branch\_fc | O | 输出的信号，为真表示发生了跳转，流水线的顺序需要被改变，否则不发生跳转 |

### 1.3.20 Jump\_fc模块

（1）基本描述

和1.3.19 Branch\_fc模块描述地基本上是一致的，不过这个模块是用来产生判断是否会发生直接跳转现象的信号。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Jumpctr[1:0] | I | 传入的描述跳转指令的控制信号 |
| Jump\_fc | O | 由Jumpctr得到的是否发生跳转的输出信号，为真时表示发生了直接跳转，指令流水线顺序需要发生改变，否则不需要跳转 |

### 1.3.21 forward\_detect模块

（1）基本描述

在五级流水线CPU中，流水段的设计不可避免地会产生数据冒险，而在非“乱序”执行地基本流水线中，所有的数据冒险都属于RAW（写后读）数据冒险，即前面指令的目的操作数后紧接着后面指令的源操作数，在流水线执行过程中，前面的指令还没有执行完最后一个Wr写回阶段时，目的寄存器的值还没有更新，而后面的指令就需要用到该寄存器的值，这就是写后读数据冒险。具体的解决逻辑会在1.4节具体介绍，这个模块就是为了通过转发的方式处理这样的数据冒险而设计的，这个模块可以通过传入的几个段寄存器中保存的信息来区分是否发生了数据冒险从而决定是否需要转发，以及确定转发的数据是应该转发哪里的数据到哪些端口，这些具体的逻辑这里不展开了，具体功能逻辑介绍详见1.4节。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Ex\_Mem\_RegWr | I | 解决数据冒险和检测是否需要转发的需要，传入的从Ex段传送到Mem段的RegWr寄存器写使能信号 |
| Ex\_Mem\_rt[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从Ex段传送到Mem段的rt寄存器编号信息 |
| Ex\_Mem\_rd[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从Ex段传送到Mem段的rd寄存器编号信息 |
| Ex\_Mem\_RegDst | I | 解决数据冒险和检测是否需要转发的需要，传入的从Ex段传送到Mem段的RegDst确定写目的寄存器的控制信号信息 |
| Mem\_Wr\_RegWr | I | 解决数据冒险和检测是否需要转发的需要，传入的从Mem段传送到Wr段的RegWr寄存器写使能信号 |
| Mem\_Wr\_rt[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从Mem段传送到Wr段的rt寄存器编号信息 |
| Mem\_Wr\_rd[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从Mem段传送到Wr段的rd寄存器编号信息 |
| Mem\_Wr\_RegDst | I | 解决数据冒险和检测是否需要转发的需要，传入的从Mem段传送到Wr段的RegDst确定写目的寄存器的控制信号信息 |
| ID\_Ex\_rs[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从ID段传送到Ex段的rs寄存器编号的信息 |
| ID\_Ex\_rt[4:0] | I | 解决数据冒险和检测是否需要转发的需要，传入的从ID段传送到Ex段的rt寄存器编号的信息 |
| ALUSrc | I | 解决数据冒险和检测是否需要转发的需要，为确保正确进行转发，需要将生成的用于确定ALU单元第二个源操作数端口是来自寄存器堆还是来自立即数送到这个模块 |
| ALUSrcA[1:0] | O | 在这个模块里会进行判断是否发生了写后读的数据冒险并根据相关送进的信息确定是否需要转发，输出的这个信号取值决定了ALU运算单元第一个源操作数来源，若为00，则来源寄存器堆读出来的busA信息，若为01，则需要转发上一条指令在Ex阶段经ALU运算得到的结果到ALU单元的第一个源操作数端口，若为10，则需要转发上一条指令经Mem段得到的最终写回寄存器的值 |
| ALUSrcB[1:0] | O | 同上一个ALUSrcA字段的逻辑基本是一样的，不过是来决定转发到ALU运算单元第二个源操作数端口的数据信息。同时需要利用传进来的ALUSrc控制信号去判断是否需要选择立即数。若为00，则选择从寄存器堆中读出来的busB信息，若为01，则转发经Ex阶段运算得到的结果Result信息，若为10，则转发经Mem段确定的要写回寄存器堆的busW信息，若为11，则选择经过扩展的32位立即数imm32 |
| DMSrc[1:0] | O | 上述对ALU运算单元的转发逻辑已经可以解决绝大多数运算相关指令间的数据冒险了，但是对于前面指令的目的寄存器和随后sw类存储字的指令的源寄存器发生的数据相关还是没有办法处理，为了处理这类冒险，需要在数据存储器的写数据端增加一个多路选择器，DMSrc就是控制选择转发哪些信息的，若为00，则正常选择busB来写数据存储器，若为01，则选择经Ex阶段计算得到的结果Result写入，若为10，则选择经Mem阶段确定的写回寄存器堆的busW来写入 |

### 1.3.22 Mem/Wr模块

（1）基本描述

这个模块也是充当一个从Mem阶段传送数据到Wr阶段的段寄存器作用，会保存下来要传送的数据，作为五级流水线最后一个段寄存器，它的作用仅仅是将从前一个阶段传送过来的数据保存然后传送给Wr写回阶段进行使用，因而它的逻辑和接口相对来讲是比较简单的，需要传送的数据也是在将数据写回寄存器堆需要的，在我的设计中，相关的冒险处理也是不需要对这个段寄存器模块进行处理的。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| pc\_in[31:0] | I | 前面阶段传送过来的PC值，用于最后一个写回阶段的使用，主要是链接的需要 |
| Result\_in[31:0] | I | 前面阶段传送过来的ALU单元最终计算的结果，用于最后一个写回阶段的使用 |
| rt\_in[4:0] | I | 前面阶段传送过来的rt寄存器信息，用于最后一个写回阶段的使用，作为可选的要写入的目标寄存器 |
| rd\_in[4:0] | I | 前面阶段传送过来的rd寄存器信息，用于最后一个写回阶段的使用，作为可选的要写入的目标寄存器 |
| RegDst\_in | I | 前面阶段传送过来的RegDst控制信号信息，用于最后一个写回阶段确定目标寄存器使用 |
| MemtoReg\_in | I | 前面阶段传送过来的MemtoReg控制信号信息，用于最后一个写回阶段确定是将ALU计算得到的结果写回寄存器还是将从数据存储器读出的结果写回寄存器 |
| RegWr\_in | I | 前面阶段传送过来的RegWr控制信号信息，用于最后一个写回阶段确定是否可以写寄存器使用 |
| ExtopM\_in | I | 前面阶段传送过来的ExtopM控制信号信息，用于最后一个写回阶段确定是否需要对写入的字节（即lb指令）作符号扩展 |
| IsLink\_in | I | 前面阶段传送过来的IsLink控制信号信息，用于最后一个写回阶段确定是否需要将对应的PC值更新到31号ra寄存器中 |
| IsByteW\_in | I | 前面阶段传送过来的IsByteW控制信号信息，用于最后一个写回阶段确定是按字写入寄存器还是按字节写入寄存器 |
| dm\_read\_in[31:0] | I | 前面阶段传送过来的dm\_read\_in信息，即在前一个Mem阶段访存得到的读数据存储器的内容，保存下来送到下一个阶段进行使用 |
| pc\_out[31:0] | O | 传递给下一阶段使用的PC信息 |
| Result\_out[31:0] | O | 传递给下一阶段使用的Result信息 |
| rt\_out[4:0] | O | 传递给下一阶段使用的rt寄存器信息 |
| rd\_out[4:0] | O | 传递给下一阶段使用的rd寄存器信息 |
| RegDst\_out | O | 传递给下一阶段使用的RegDst信息 |
| MemtoReg\_out | O | 传递给下一阶段使用的MemtoReg信息 |
| RegWr\_out | O | 传递给下一阶段使用的RegWr信息 |
| ExtopM\_out | O | 传递给下一阶段使用的ExtopM信息 |
| IsLink\_out | O | 传递给下一阶段使用的IsLink信息 |
| IsByteW\_out | O | 传递给下一阶段使用的IsByteW信息 |
| dm\_read\_out[31:0] | O | 传递给下一阶段使用的dm\_read\_out信息 |

### 1.3.23 mux\_memtoreg模块

（1）基本描述

正如模块名字所述，本模块是一个二选一多路选择器，来选择将哪一个值写入寄存器堆，传入信号MemtoReg控制是选择ALU单元运算的结果写入寄存器堆还是选择从数据存储器读出来的值写入寄存器堆。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ALU\_out[31:0] | I | 传入的ALU单元运算得到的结果 |
| datamem\_out[31:0] | I | 传入的从数据存储器dm读出来的值 |
| MemtoReg | I | 控制信号，为1时选择datamem\_out,否则选择ALU\_out |
| memtoreg\_out[31:0] | O | 输出，最终将被写入寄存器堆的值 |

### 1.3.24 mips模块

（1）基本描述

最后一个模块是设计的CPU的顶层mips模块，这个模块只有时钟信号和PC复位信号两个输入的接口，在这个模块中，就是将整个五级流水线CPU设计中的所有模块进行例化，统一组织并且将它们联系起来，CPU功能的测试就是经过这个模块进行的。

（2）接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| reset | I | PC复位信号 |

## 1.4 五级流水线CPU冒险处理方式

这一部分来具体介绍我所设计的五级流水线CPU所遇到的冒险情况和具体所采取的处理冒险的手段和方式。

### 1.4.1 转发到ALU单元的写后读数据冒险处理

在报告其他模块也或多或少提及了这样一个冒险，引起这样的写后读数据冒险的原因在于后面指令需要用到前面指令结果时而前面指令的结果还没有产生，这个在五级流水线的设计中是非常容易出现的，比如第一条指令是addu指令，第二条指令是subu指令，第三条指令是or指令，addu指令的目的寄存器是15号寄存器，而subu指令和or指令的源操作数寄存器（第一个或者第二个）也是15号寄存器，而addu正确将运算的结果写入到15号寄存器需要在五级流水线第五个阶段Wr写回阶段才可以，而第二条和第三条指令计算需要用到对应的数据则在Ex阶段进行，并且在ID阶段的后半阶段需要将正确的值传送给下一阶段Ex阶段进行使用。如果出现第一条指令和第四条指令之间发生了数据相关问题，根据所设计的五级流水线CPU的结构和工作逻辑（可参考1.1五级流水线CPU总设计说明图）我们可以通过更改寄存器堆的写口和读口的逻辑，将写和读寄存器分别控制在前、后半个时钟周期内操作来解决，这个体现在我这次设计的具体代码上表现为，寄存器堆在时钟上升沿的时候才会进行写，而所有段寄存器都是在时钟下降沿才开始写。

处理这样的冒险的手段有很多种，比如软件上插入空操作指令，硬件上插入气泡，或者在数据通路上采用转发的技术，也就是将数据通路生成的中间数据（会通过流水段寄存器逐级传送）提前转发到ALU的输入端，也是我在这次课设中解决这一类冒险采用的手段。

经过如上的分析，我们可以知道，在Ex段结束时已经得到了计算结果，而这个正确的新值被存放在Ex/Mem流水段寄存器中（比如上面所举例子的addu指令和subu指令间的数据相关），因此需要将这个流水段寄存器中存放的经Ex段计算结束的新值转发到ALU运算单元的输入端，而对于间隔了一条指令的数据相关指令（比如上面例子的addu指令和or指令间的数据相关），可以通过将Mem/Wr流水段寄存器中的对应值转发到ALU运算单元的输入端口。因此，原来的ALU运算单元第一个输入端A需要增加一个三路选择器，第二个输入端B需要增加一个四路选择器，在1.3.11 alu\_in1\_mux模块和1.3.12 alu\_in2\_mux模块的描述和分析中也有详细的阐述。

除了需要对应的转发处理的逻辑，还需要判断检测的逻辑，也就是1.3.21 forward\_detect模块中的逻辑，对于这类冒险的检测，我们参考对应模块的如下代码截图（报告中2.1.21部分也附上了对应的代码）进行描述和分析，同时也可以在报告的1.3.21部分参考对本模块的接口定义的描述和分析进一步参考。

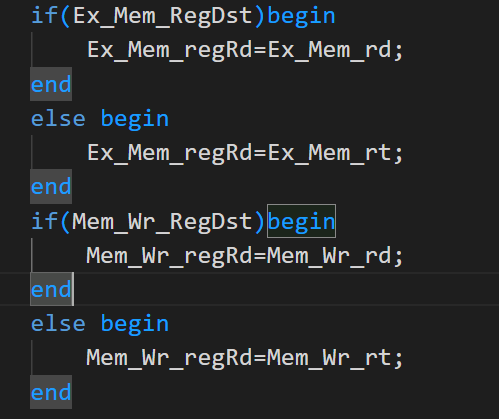


图1.4.1.1 代码片段1

图1.4.1.1是通过传入的对应段寄存器的寄存器编号信息和写目的寄存器的控制信号来生成临时的在各个段寄存器的写目的寄存器的编号。

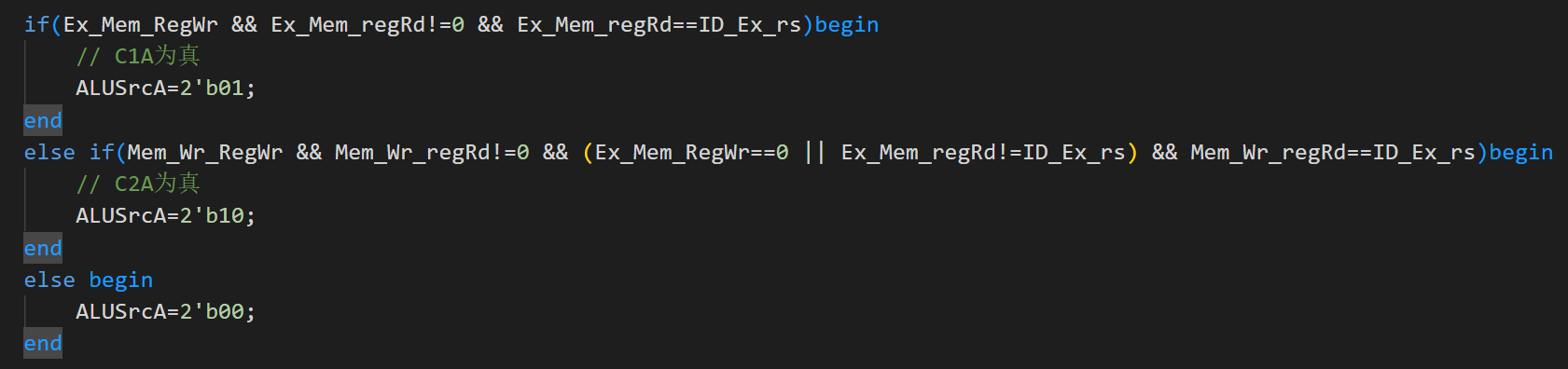


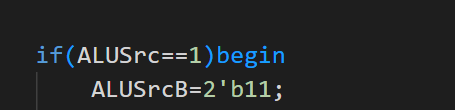
图1.4.1.2 代码片段2

图1.4.1.2展示了生成对应的ALU第一个输入端口A的多路选择器信号，可以看到对应的逻辑，当Ex/Mem段寄存器的寄存器写使能信号为真并且此段下保存的写目的寄存器的编号不为0（这是考虑到0号寄存器在MIPS中是恒为0的zero寄存器，不应该被修改，所以不需要转发），并且此段下保存的写目的寄存器的编号与ID/Ex段下保存的rs寄存器编号相同，说明当前指令的目的操作数与随后第一条指令所用的源操作数rs寄存器发生了数据相关，因此需要将经过Ex段计算得到的结果转发到ALU的A输入端。

否则，当Mem/Wr阶段的寄存器写使能信号为真并且此段下的写目的寄存器编号不为0（理由同上）时，并且Ex/Mem段的写目的寄存器与ID/Ex段的源操作数寄存器编号不同（这个是因为可能存在多条连续的指令关于同一个寄存器数据相关，在这种情况下，如果不加这个判断，会使得两个判断均为真可能会产生意外的错误，不过上述代码的逻辑下应该是没有问题的），同时还需要Mem/Wr段寄存器的写目的寄存器和ID/Ex段的rs寄存器相同，即说明了本条指令的目的操作数是随后第二条指令所用的源操作数，这中间发生了数据相关。同时可以看到上述代码，在判断Ex/Mem段寄存器的写目的寄存器编号与ID/Ex的rs寄存器编号不等时，还增加了一条或逻辑的判断条件，即如果Ex/Mem段寄存器下的寄存器写使能信号为假时，可以忽略到后面那个判断，这个可以参看下面一小节1.4.2节的数据冒险处理分析。

如果上述两条都不满足，那么就默认选择从寄存器堆读出来的busA的内容送到ALU的A输入端即可。

对于ALU运算单元的另外一个输入端B而言，相关的逻辑是几乎一致的，区别主要在于B输入端需要增加一个取立即数的送入端口，所以需要一个四路选择器，并且当ALUsrc控制信号为真时，即原来的B输入端需要选择立即数时，这个时候就不需要考虑转发了，因为立即数是在指令译码阶段直接给出的，不需要从寄存器中取并进行转发，同时在上面A输入端判断是否发生冒险时用的是ID/Ex段寄存器的rs寄存器编号，根据数据通路的设计（可以参考1.2五级流水线CPU的总电路设计图），在ALU的B输入端应该根据ID/Ex段寄存器的rt寄存器编号进行判断，相关代码片段如下图1.4.1.3所示。



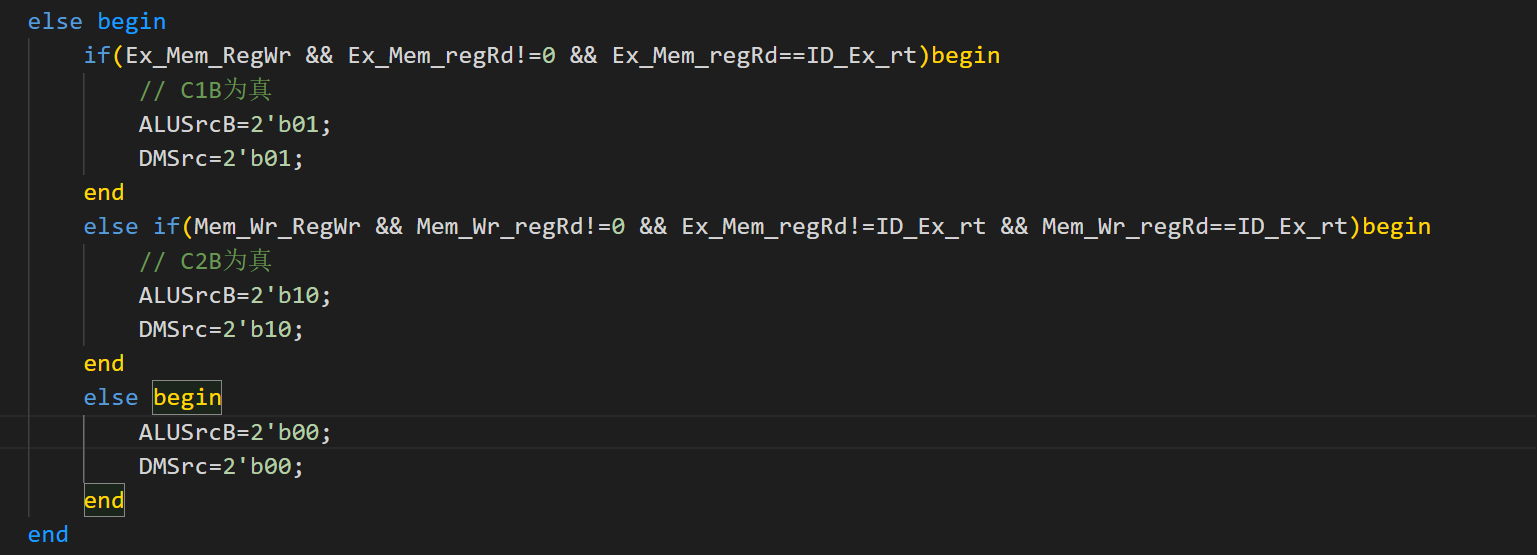


图1.4.1.3 代码片段3

这样处理之后，大部分的RAW（写后读）的数据冒险问题基本就被解决了。

### 1.4.2 转发到DM单元的写后读数据冒险处理

对于1.4.1通过转发来处理经典写后读数据冒险的逻辑只考虑了像一开始所举的那个例子一样的情况，即三条指令都是需要打开寄存器写使能信号的，但是对于前面指令的目的寄存器和随后sw类指令的源寄存器发生数据相关的情况是无法解决的，即像连续的这两条指令“addu $s3,$2,$1”和“sw $3,0($s1)”的指令，同时，如果这两条指令后面跟一条像“subu $4,$3,$1”这样的addu指令的目的寄存器是subu的源操作数寄存器并且也是sw指令的源操作数寄存器且这条sw类指令位于两者中间，由于涉及到写数据存储器的问题并且sw指令是写使能关闭的，即本条指令随后一条的指令的RegWr非真但是也与随后第二条指令发生了数据相关，且由于与sw类指令发生数据相关时的源操作数寄存器对应的是rt寄存器的编号，导致1.4.1对于ALU运算单元的输入端A的本应该生效的第二个逻辑判断失效了。因而如代码中所示的，将原先的“Ex\_Mem\_regRd!=ID\_Ex\_rs”改变为了“Ex\_Mem\_Reg==0 || Ex\_Mem\_regRd!=ID\_Ex\_rs”，代码片段如下图1.4.2.1所示。

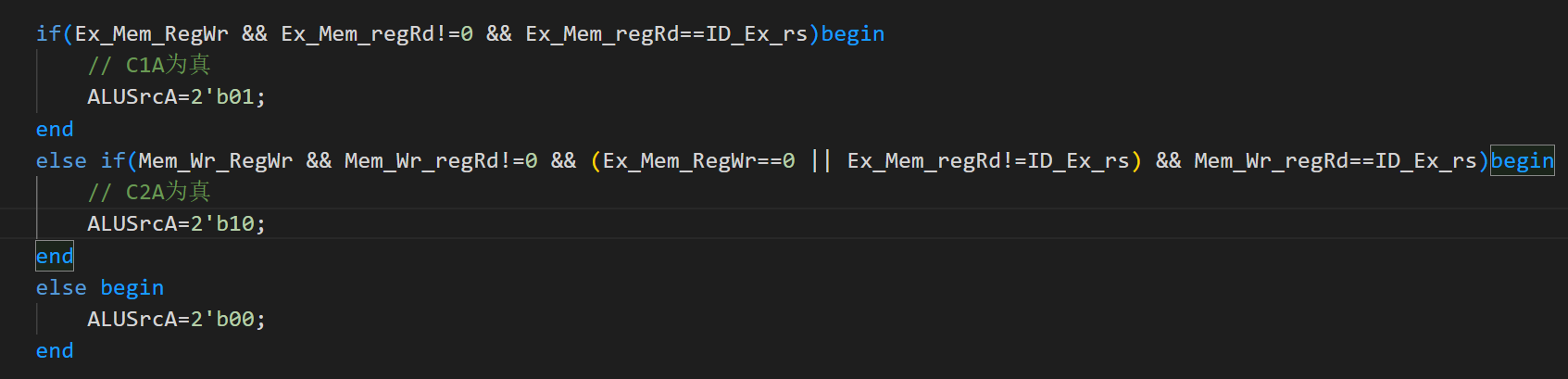


图1.4.2.1 代码片段1

此外，对于上述例子中第一条指令addu和第二条指令sw之间的数据相关，由于是在写数据存储器时写入的数据来自前一条指令的目的寄存器，而前一条指令还没有将正确的值写入数据存储器，类似采取1.4.1的判断逻辑，通过在数据存储器DM的写数据端增加一个多路选择器（这一块也可以参考1.3.7 dm\_in\_mux模块的接口介绍），根据冒险检测产生的控制逻辑信号选择是写入正常情况下从寄存器堆读取的并经段寄存器保存和传送的busB的内容，还是经Ex阶段保存在Ex/Mem段寄存器的上一条指令产生的ALU结果，还是上上条指令经Mem阶段在Wr写回阶段要写回寄存器堆的busW的内容。而具体的多选控制信号DMSrc根据对sw类指令的解析，其源操作数寄存器对应的是rt寄存器，因此检测判断的逻辑和1.4.1情况中对于ALU运算单元B输入端的情况几乎一致的，也不再赘述重复单一的内容了，相关代码逻辑如下图1.4.2.2所示。

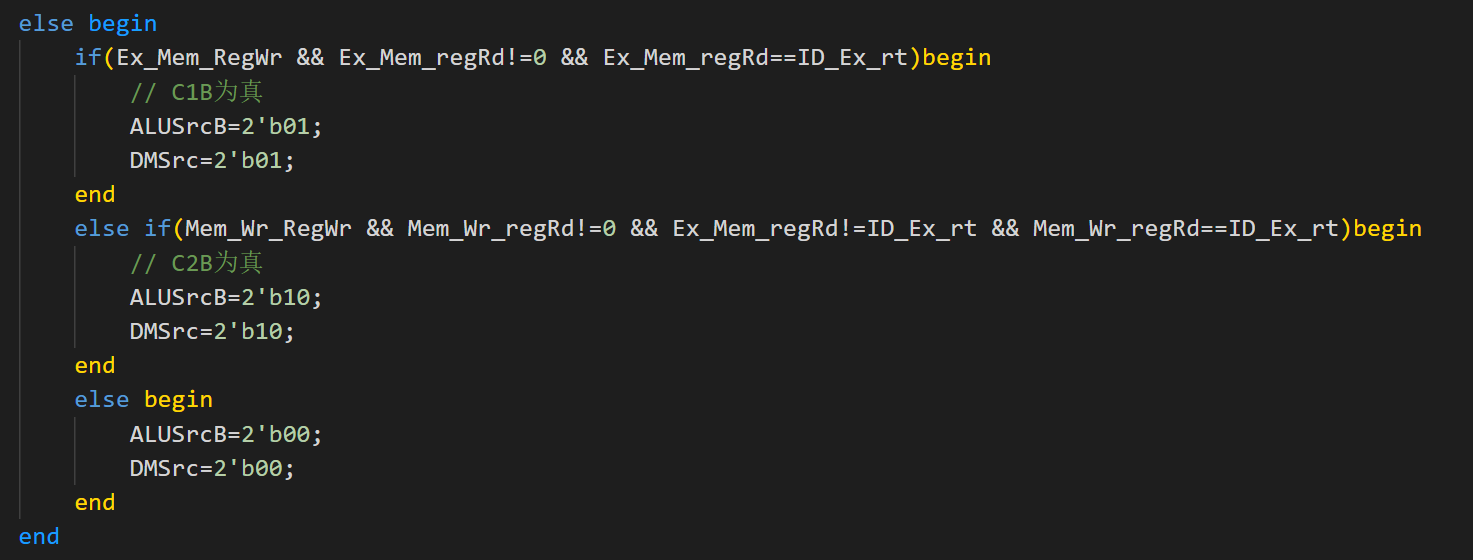
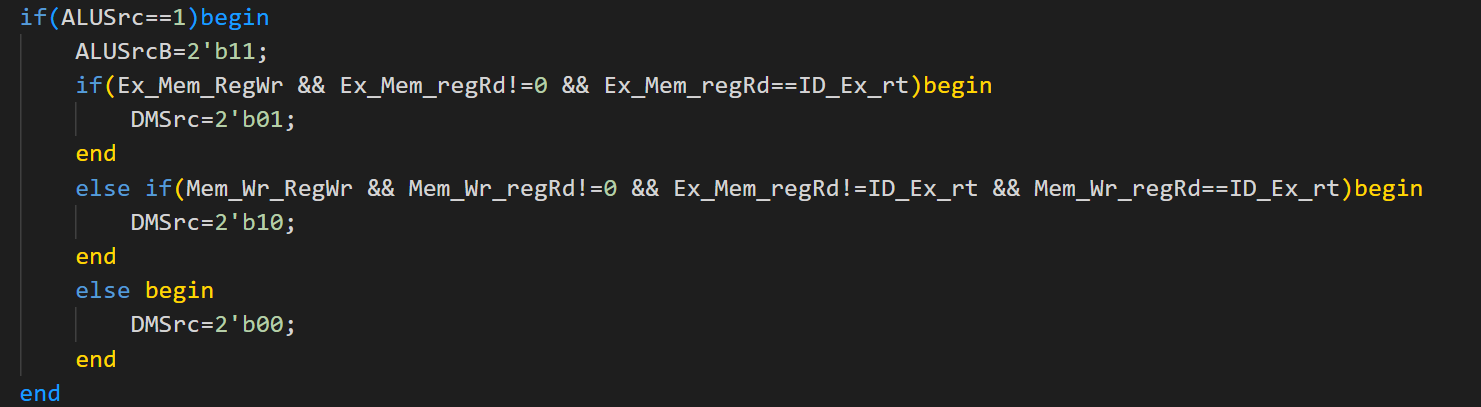


图1.4.2.2 sw指令产生的数据相关问题的判断逻辑

### 1.4.3 load-use数据冒险处理

经过1.4.1和1.4.2的转发处理逻辑已经可以解决极大一部分的RAW（写后读）数据冒险了，但是还有一类正如本小节标题所描述的数据冒险问题，就是像类似lw指令随后根R型或者I型运算类指令的相关性问题，即需要从数据存储器取数到寄存器中，但是这个数值还没有取出来，后面的指令的源操作数就是要使用对应的寄存器中的数据。

对于上面描述的这样的数据相关问题称为load-use类型的数据冒险，具体一点来讲，可以使用这样的例子，有这样的连续的几条指令执行序列：“lw $1,0($2)”“sub $4,$1,$3”“or $8,$1,$9”，lw指令只有在Mem段结束时才能得到DM数据存储器中的结果，然后送Mem/Wr寄存器，在Wr段前半周期$1中才能存入新值，但随后的sub指令在Ex阶段就要取$1的值，因而得到的时旧值，根据1.4.1和1.4.2的综合转发逻辑，ALU的输入端要么来自上条指令在Ex段生成且存放在Ex/Mem段寄存器中的值，要么来自上上条指令的执行结果，因此是无法解决的。

对于这类的load-use数据冒险，最简单的做法就是由编译器在load指令之后插入nop指令来解决，也可以在程序编译时进行优化，通过调整指令顺序来避免出现load-use现象。那么对于设计的CPU只能从硬件角度来处理load-use冒险，则必须在流水线数据通路中增加load-use冒险检测部件，并且在检测到发生load-use冒险时必须进行流水线阻塞处理。

经过上述分析之后，我们的load-use检测模块可以参考1.3.14 LoadUse\_detect模块中的对接口的描述和分析，具体的判断逻辑可以参考如下代码，即如图1.4.3.1所示。

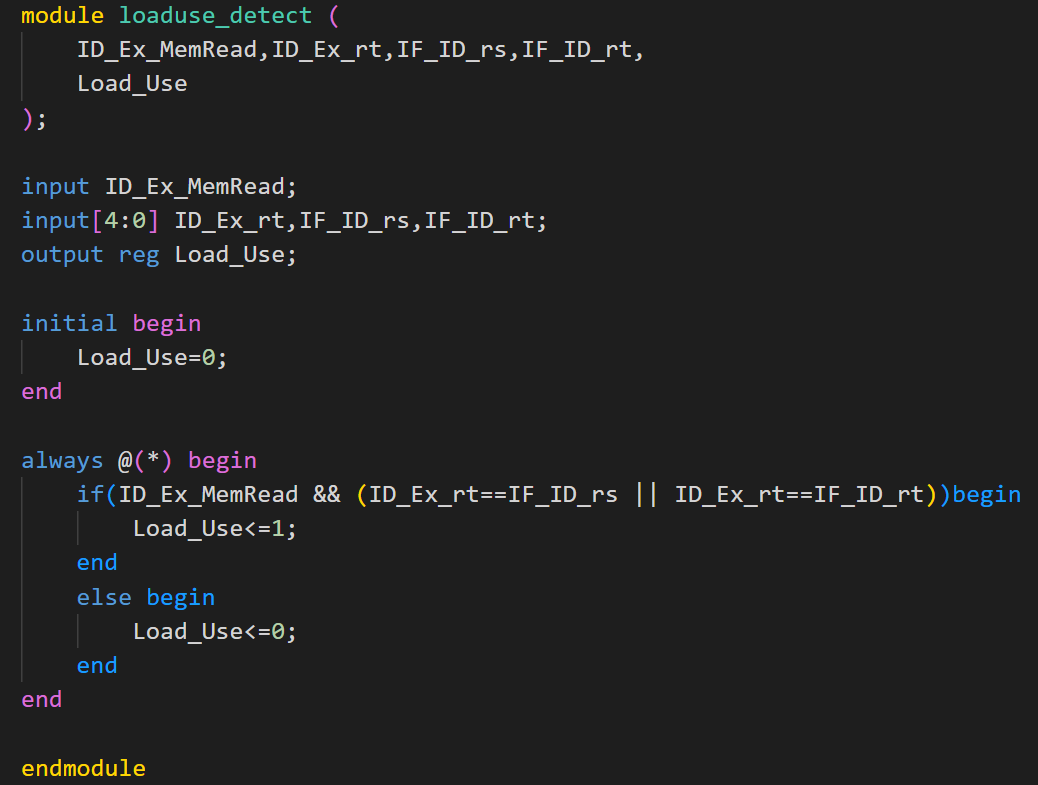


图1.4.3.1 检测load-use冒险发生的逻辑

从这个代码中可以看出，我们检测load-use发生的条件就是上一条指令时load指令，并且从数据存储器装入寄存器的数据是当前指令的源操作数，原则上load-use冒险的检测应该是越早越好，但是再早也要在load类指令随后的指令被取出并被译码之后，因此可以将该模块置于译码ID阶段，那么根据五级流水线的工作结构模式，ID/Ex段寄存器中存放的MemRead控制信号就是可以用来判断上一条指令是否是load类型指令的（MemRead控制信号只有当指令是load类型指令，具体在设计的36条基础MIPS指令中含有lw，lb和lbu三条指令），并且此时ID/Ex段寄存器中存放的rt寄存器就是对应于上条是load指令情况下的目的寄存器，IF/ID段寄存器中存放的rs和rt寄存器编号就是对应本条指令的两个源操作数寄存器，当与上条指令load指令的从数据存储器装入寄存器的寄存器编号相同时则说明发生了load-use数据冒险。

经过了判断和检测，就需要进行解决，对于上例所示的指令间发生的load-use数据冒险，在硬件层面的解决，则必须让紧随load后的两条指令停顿一个时钟周期后继续执行，这可以通过将这两条指令的执行结果清除并让它们延迟一个时钟周期来实现。具体就是在代码中实现了如下三个操作：（a）将ID/Ex流水段寄存器中的所有控制信号清零；（b）保持IF/ID流水段寄存器的值不变；（c）保持PC的值不变。这三个操作对应到具体代码实现中分别如下图1.4.3.2、图1.4.3.3和图1.4.3.4所示（完整代码可以参看2.1节具体Verilog代码实现中的2.1.8 ID/Ex模块、2.1.5 IF/ID模块和2.1.1 PC模块）。

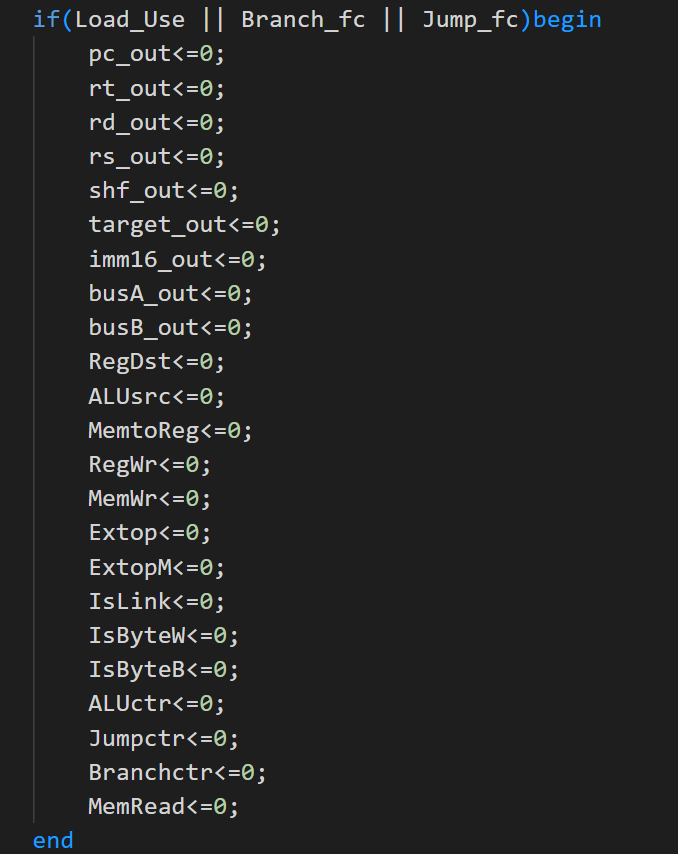


图1.4.3.2 ID/Ex段解决load-use冒险的逻辑

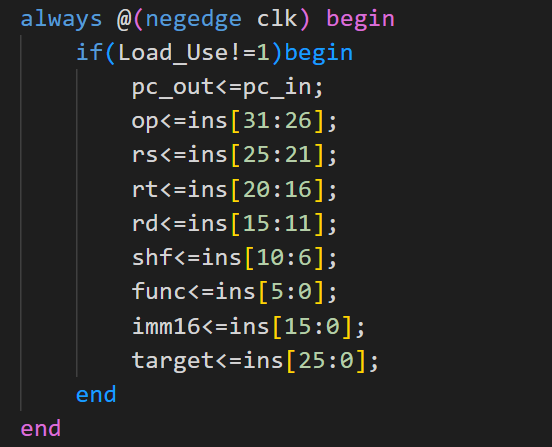


图1.4.3.3 IF/ID模块解决load-use冒险的逻辑

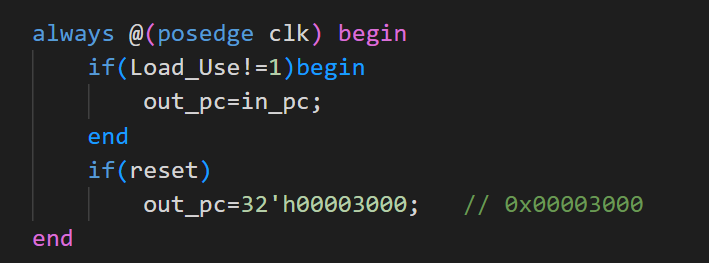


图1.4.3.4 PC模块解决load-use冒险的逻辑

当检测到发生了load-use数据冒险时，会产生一个Load\_Use的控制信号，使它为真，当它为真时，ID/Ex段寄存器模块会将要输出的所有控制信号清零，IF/ID段寄存器模块只有当Load\_Use控制信号不真时才会将传进来的新值写入并输出，进而实现了当Load\_Use控制信号为真即产生了load-use数据冒险时保持了IF/ID流水段寄存器中的值不变的要求，同样地，在PC更新模块，只有当Load\_Use控制信号不真时才可以写入新的PC，进而实现了保持PC的值不变的要求，即均实现了上述需要的（a）（b）（b）三个处理手段，解决了load-use数据冒险。

### 1.4.4 分支跳转简单预测和直接跳转的控制冒险处理

经过1.4.1、1.4.2和1.4.3的处理，基本上所遇到的数据冒险都能解决，但是这都是基于非“乱序”执行的流水线来讲的，在正常的指令序列中，经常会出现分支跳转和直接跳转这类的改变指令执行顺序而产生控制冒险，对于这类跳转产生的控制冒险，一般解决的手段就是在产生了跳转的信息之后将流水线阻塞,即将相关段寄存器产生的信号清零，使多执行的指令产生的结果不会影响到正常执行的指令。处理手段在具体代码中的体现如下图1.4.4.1和图1.4.4.2所示。

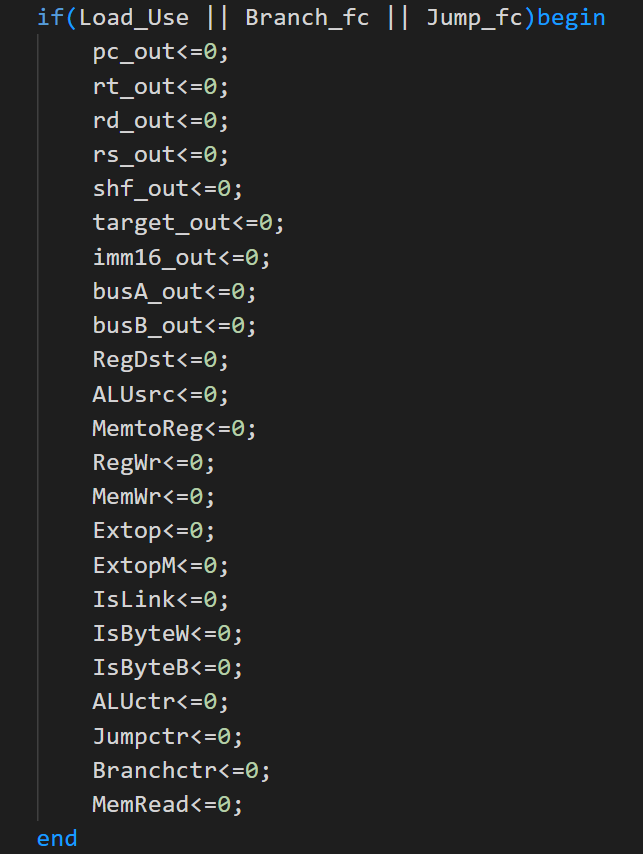


图1.4.4.1 ID/Ex段寄存器的阻塞逻辑

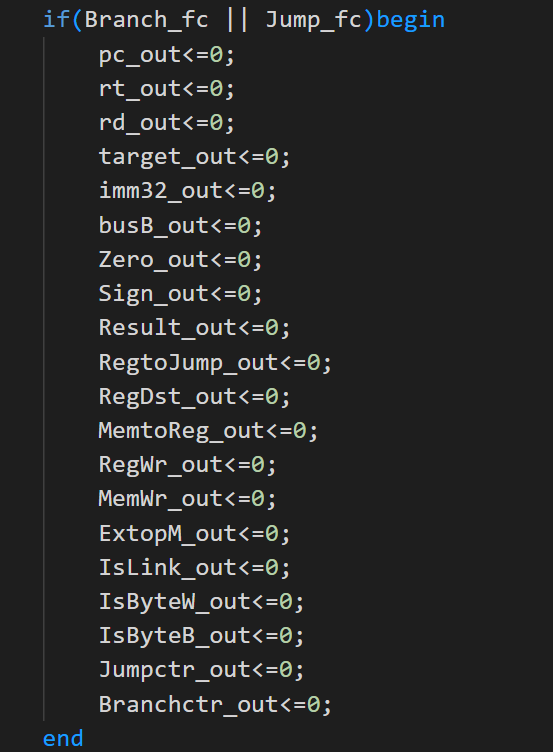


图1.4.4.2 Ex/Mem段寄存器的阻塞逻辑

图1.4.4.1是ID/Ex流水段寄存器阻塞的实现，Branch\_fc信号或者Jump\_fc信号为真时说明产生了分支跳转或者直接跳转改变了流水线的执行顺序，因此需要将段寄存器阻塞，即将生成的所有信号清空就可以了（相当于插入一个气泡），而图1.4.4.2是Ex/Mem流水段寄存器阻塞的实现，也是同样的逻辑，当Branch\_fc和Jump\_fc信号为真时，清空段寄存器的保存的所有信号输出即可。

对于直接跳转指令，当译码得到这样的指令时是一定会发生跳转的，就必须需要阻塞流水线，但是对于分支类的跳转指令，分支跳转的条件不一定一定会触发跳转，因此可以在流水线引入分支预测的方式，如果预测跳转的情况正确的话，那么就可以避免因为阻塞流水线而带来的延迟损失时间片，在本课设的五级流水线CPU设计中引入了分支预测，并且使用的是简单预测方式，即总是预测跳转或者总是预测不跳转，在设计也是采用总是预测不跳转的方式，让PC总是按照PC+4的方式来取指，当预测错误，即发生了跳转，Branch\_fc信号为真，对应的流水段会在相关信号的控制信号阻塞。

对于生成Branch\_fc和Jump\_fc信号的模块分别对应于1.3.19 Branch\_fc模块和1.3.20 Jump\_fc模块，它们的逻辑是非常简单的，就是根据是否跳转的条件和相关控制信号进行简单逻辑判断并且输出即可，相关代码逻辑如下图1.4.4.3和图1.4.4.4所示。

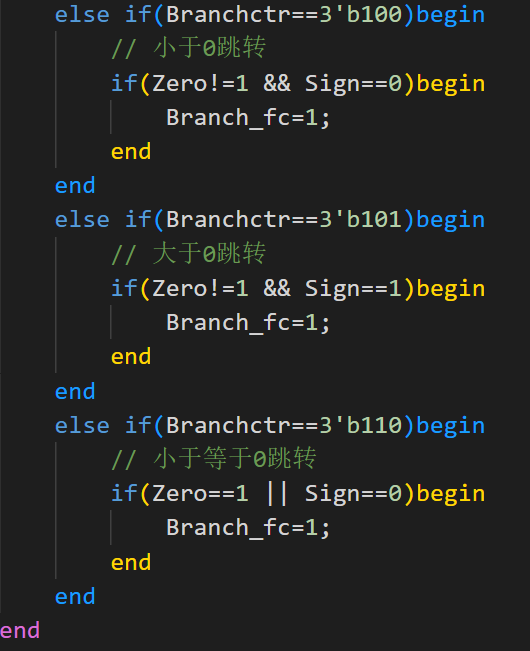
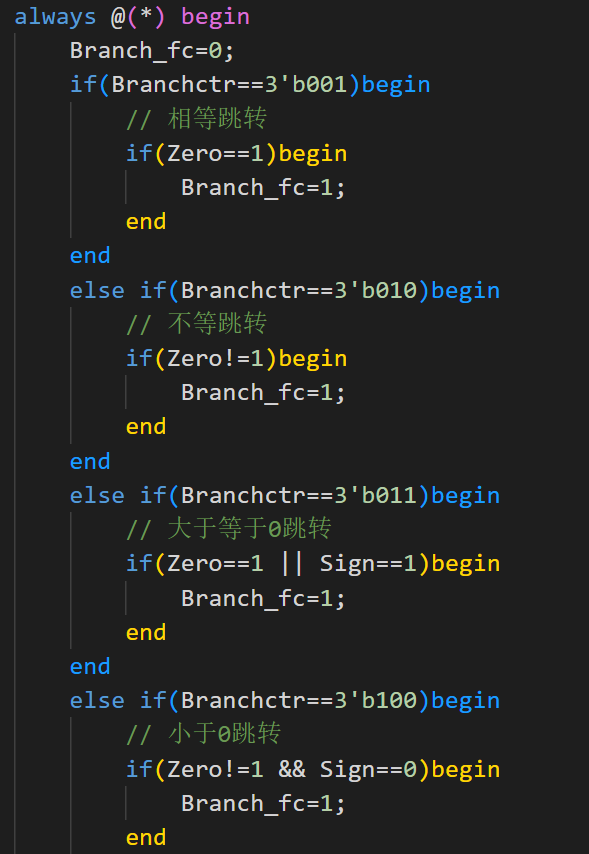


图1.4.4.3 Branch\_fc模块判断逻辑

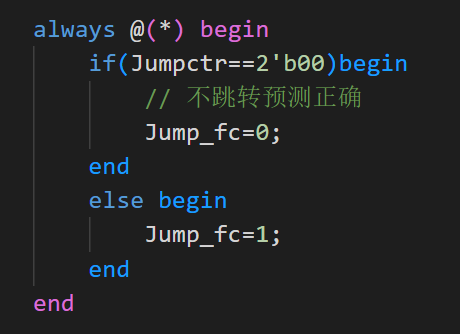


图1.4.4.4 Jump\_fc模块判断逻辑

最后说明一下，这两个模块都是置于Mem段进行的，只是因为经过Ex段之后对于分支指令和直接跳转指令都可以得到需要用于判断是否跳转以及计算跳转之后的目标PC地址值所需要的相关信号和数据信息并被送到npc模块计算下一条指令，按照五级流水线的执行结构（可以参考1.1的结构设计说明图），此时应该已经有后面三条顺序执行的指令被取出来了，那么应该让流水线阻塞三个时间周期，即延迟执行的周期数为3，即应该阻塞Ex/Mem段寄存器、ID/Ex段寄存器和IF/ID寄存器，不过我在设计过程中受到课设要求中“将beq 这类指令的判定跳转提前到第二阶段完成，来提高效率”的启发，由于实现这样的效果会破坏设计的统一性，并且带来不必要的麻烦，所以还是将分支跳转类指令和直接跳转类指令计算并且去更新npc模块都统一置于Mem阶段，同时为了尽可能减少延迟执行周期数，我在模块设计上将ID阶段译码的工作兼并到了IF/ID模块中去（可参考1.3.5 IF/ID模块的实现逻辑和接口介绍），从而使得当跳转指令执行到这一阶段时只顺序执行了两条指令，这样只需要阻塞如上所述的两个段寄存器，将原本为3的延迟执行周期数降低为2，提升了一定的效率的同时，还没有打破设计的统一性（本人在设计上是非常喜欢保持统一性的，所以最终设计上采取了这样的技巧式的实现方式）。

到此为止，整个1.4节分析描述完了本五级流水线CPU设计中所遇到的所有冒险情况并且和对应的具体的解决方式都详细介绍如上了。需要说明的是，这里所提及的冒险情况并不是绝对的，也就是当需要设计的指令集中增加或者扩展了更多的指令之后，可能在某些指令之间发生的冒险当前已有的逻辑是无法顺利解决的，而且目前现有的可以解决冒险的逻辑也仅仅是建立在自己拟写出来并测试通过的基础上，是不能保证所有情况都是正确无误的。此外，除了出现冒险的情况不绝对之外，对于解决冒险的手段也不是唯一的，事实上除了结构冒险这样的硬件资源冲突的冒险问题，其他的冒险都是有多种解决手段的，起始在上面分析冒险情况和处理方式也都有提及。对于本流水线设计中对于控制冒险处理的方式，采取是一种最简单的方式，即简单预测并且总是预测不跳转的方式，事实上，动态预测的方式应该被作为在解决由分支跳转指令引起的控制冒险问题的首选手段，它利用分支指令发生转移的历史情况进行预测，并且根据实际执行情况动态调整预测位，它解决这类控制冒险的效率损失相比简单预测来讲是极低的。当然，控制冒险还有另外一类由于异常或中断引起的，不过相关指令也没有设计就不赘述了。

综上，我在本课设中的五级流水线CPU中的冒险检测和处理方式的所有内容就如上所述了。

## 1.5 五级流水线CPU各个指令控制信号取值标图



图1.5 五级流水线CPU各个指令控制信号取值表

如上图1.5所示，展示了本次课程设计中的五级流水线CPU各个指令对应的控制信号的具体取值，相比较实验中的单周期CPU，五级流水线的控制信号几乎保持了一致，唯一新增的，就是为了处理load-use的数据冒险新增了一个用于判断上条指令是否是Load指令的Memread信号。从这个表中，可以非常清楚地看到每条指令对应的控制信号的取值，在本课设报告其他部分如果遇到对指令控制逻辑不清楚或者解释不够清楚的地方，可以暂且回到这里参考这张表（具体浏览时可以放大更清楚地观察到具体的指令和相对应的控制取值）。

## 1.6 五级流水线CPU各个指令的设计逻辑

需要说明的是，在实验中实现的单周期CPU支持了36条MIPS基础指令，在课程设计五级流水线设计中，按照课程设计的要求，依旧是设计支持了一样的36条MIPS基础指令，具体的36条指令的解析可以参看附录2所附内容。并且流水线CPU和单周期CPU主要的区别在于增加了流水段的划分和段寄存器保存传送数据，同时需要做好冒险处理部分，具体的指令实现逻辑与单周期CPU的设计几乎没有区别，个别指令比如sb指令由于冒险处理的原因，逻辑上做出了一点修改，详细内容可见如下对36条指令的解析。

### 1.6.1 R型指令

（1）无符号加法：addu rd,rs,rt

将rs寄存器和rt寄存器的值相加并且写到rd寄存器中，无符号加法因此不作溢出判断，实现逻辑很简单，从寄存器堆读出相应的值送到ALU单元进行运算，并将相应的结果再写回寄存器堆即可。

（2）无符号减法：subu rd,rs,rt

将rs寄存器的值减去rt寄存器的值相减并写到rd寄存器中，同样不作溢出判断，实现逻辑同无符号加法，ALU单元中作无符号减法运算即可。

（3）有符号比较，小于置位：slt rd,rs,rt

比较rs寄存器和rt寄存器中的值，若小于则将rd寄存器置为1，否则置0，实现逻辑与前面一样。

（4）按位与：and rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位与运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位与运算即可。

（5）按位或非：nor rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位或非运算即可。

（6）按位或：or rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位或非运算即可。

（7）按位异或：xor rd,rs,rt

将rs寄存器和rt寄存器中的值进行按位或非运算并写回rd寄存器，逻辑同前面的指令，ALU单元作按位异或运算即可。

（8）逻辑左移：sll rd,rt,shf

将rt寄存器中的值逻辑左移shf位并将结果写回rd寄存器，shf字段信息已经提前被解析出来并且送到ALU单元，根据控制信号进行相关运算即可。

（9）逻辑右移：srl rd,rt,shf

将rt寄存器中的值逻辑右移shf位并将结果写回rd寄存器，逻辑是平凡的，ALU单元中的运算采取逻辑右移即可。

（10）无符号小于置位：sltu rd,rs,rt

将rs寄存器和rt寄存器中的值进行比较，若小于则将1写回rd寄存器，这里的有符号和无符号比较在代码中不需要特意加以区分，可以共用一个ALU计算方式控制信号。

（11）跳转寄存器并链接：jalr rs

这一条指令特殊安排一个ALUctr的控制信号用来计算，当是jalr运算时，ALU单元将rs寄存器中的值写到RegtoJump端口中并送到npc模块单元，同时这条指令生成的控制信号会更新相关的跳转信号Jumpctr，同时本条指令解析格式中的rd寄存器为31号寄存器，所以将ALU运算的结果表示为当前的PC值+4之后的值，通路逻辑会将这个结果自动写回31号ra寄存器。

（12）跳转寄存器：jr rs

这一条指令和上一条指令少了链接的要求，因此实现更为简单，不需要考虑链接功能，将从rs读出来的值送到RegtoJump中去即可，npc单元中会正确计算跳转之后的下一条指令的PC。

（13）变量逻辑左移：sllv rd,rt,rs

和之前逻辑左移shf位指令的区别是，变量逻辑左移是将rt寄存器中的值逻辑左移rs寄存器中的值对应的位数，因此可以通过增加新的ALUctr计算控制信号控制运算方式，选择用rs寄存器的值进行移位运算，并将结果写回rd寄存器即可。

（14）算术右移：sra rd,rt,shf  
 算术右移和逻辑左移有一定的区别，在Verilog代码模拟中，可以通过系统函数$signed并将原来的逻辑右移符号>>改为>>>即可，其他的逻辑是一致的且平凡的。

（15）变量算术右移：srav rd,rt,rs

结合上述指令中对变量移位和逻辑移位的逻辑，这一条指令可以使用类似的逻辑进行设计，不再赘述。

（16）变量逻辑右移：srlv rd,rt,rs

逻辑基本和变量逻辑左移一致，ALUctr控制信号来控制相对于的运算方式即可，不再赘述。

### 1.6.2 I型指令

（17）立即数、无符号加法：addiu rt,rs,imm

将rs寄存器中的值加上符号扩展后的imm立即数并将结果写入rt寄存器，与前面介绍过的R型指令不一样的是目的寄存器不是rd寄存器而是rs寄存器，ALU另一个端口选择的操作数不是rt寄存器而是经扩展器扩展的立即数，这些控制单元生成的控制信号相应地都会去进行控制。ALU单元中的计算方式仍然是无符号加法。

（18）相等跳转：beq rs,rt,offset

分支跳转指令，控制单元会生成相应的对应类型的分支跳转的控制信号Branchctr，相等逻辑的判断在ALU单元中其实是执行的无符号减法的运算，根据结果是否为0生成对应的标志位Zero，npc模块会根据这个标志位和对应的分支跳转控制信号判断是否进行跳转，同时，经过扩展的offset立即数也会送到npc单元中进行地址的计算。当Branchctr控制信号取001并且Zero标志位为1时发生跳转。

（19）不等跳转：bne rs,rt,offset

与上一条相等跳转指令逻辑几乎一致，判断逻辑根据不等稍微改变一下即可，即相等判断Zero是否等于1，不等即判断Zero是否等于0，其余的执行逻辑如出一辙。当Branchctr控制信号取010并且Zero标志位为0时发送跳转。

（20）装载字：lw rt,offset(base)

这条指令base给出的是基址寄存器，加上符号扩展的立即数offset后得到在数据存储器中需要取数的地址，并经过从存储器取数后将取得的一个字的数据写回rt寄存器，计算地址在ALU中使用的无符号加法运算方式，从数据存储器取数的相关逻辑在前一部分模块定义中已经说明清楚，相关控制信号会控制正确取数并将写回寄存器的值改为从数据存储器中读出来的值。

（21）存储字：sw rt,offset(base)

与前面一条lw指令不同的是，这个是将从rt寄存器中读出来的值写到数据存储器对应的地址中去，更新相关需要的控制信号即可完成，计算目的地址在ALU单元中使用的是无符号加法的运算。

（22）立即数装载高位：lui rt,imm

这是将16位的立即数放到rt寄存器对应值的高16位，并在低16位添加0即可，这里需要注意的是控制信号中的Extop需要置为0，即不需要进行符号扩展，ALU运算单独设计一个lui的运算方式特殊处理。

（23）立即数有符号比较，小于置位：slti rt,rs,imm

将rs寄存器的值和经符号扩展的imm立即数进行比较，如果小于则将rt寄存器置1，否则置0，没有什么特殊的逻辑，控制信号保证可以选择正确的值进行比较，ALU单元中的运算方式采用小于比较置位。

（24）立即数无符号比较，小于置位：sltiu rt,rs,imm

与立即数有符号比较并没有区别，这里不再赘述。

（25）大于或等于零跳转：bgez rs,offset

分支跳转指令，比较rs寄存器中的值和0的大小关系，接下来的包括这条指令在内的和0比较的分支跳转指令，解析出来的rt寄存器的值都是0，因此在ALU运算单元中使用无符号减法，得到的结果转化为无符号数与0比较，如果小于0则说明rs的值小于0，Sign置1，否则Sign标志位置0。控制信号会生成对应的分支信号，offset经符号扩展后会送到npc单元，根据Zero（表示两个值是否相等）和Sign标志位的取值情况判断是否需要进行跳转并计算得到正确的下一条指令的PC具体值。当Branchctr控制信号取011并且Zero标志位取0或者Sign标志位取1时发生跳转。

（26）大于零跳转：bgtz rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取101并且Zero标志位取0并且Sign标志位取1时发生跳转。

（27）小于或等于零跳转：blez rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取110并且Zero标志位取1或者Sign标志位取0时发生跳转。

（28）小于零跳转：bltz rs,offset

逻辑与bgez指令一致，不再赘述。当Branchctr控制信号取100并且Zero标志位取0并且Sign标志位取0时发生跳转。

（29）装载字节并作符号扩展：lb rt,offset(base)

与前面分析的lw装载字指令的区别是本条指令写入寄存器的值是字节，这个字节就是取原来读出来的字的低8位，并且这个字节的高24位需要作符号扩展，这些是通过在原有lw指令执行的逻辑上新增了IsByteW和ExtopM两个控制信号来实现的，前者为真时表示写寄存器时写入字节，后者为真时表示写入的字节需要进行符号扩展，其余的逻辑与lw指令无异。

（30）装载字节并作无符号扩展：lbu rt,offset(base)  
 与lb指令不同的是，ExtopM控制信号控制为0对要写入的字节作零扩展即可。

（31）存储字节：sb rt,offset(base)

与sw指令不同的是，sb指令需要将字节写入具体地址的数据存储器中，但是数据存储器的接口要求是按字为单位进行存储的，因此对于计算得到的地址在数据存储器中读出来的那一个32位的字信息，一共4个字节，需要根据ALU计算得到的地址找到是哪一个字节需要被写入，并且保证不覆盖其他3个字节的信息，要注意的是这里的存储方式是按照小端进行的，因此需要注意找对应字节的顺序。为了实现sb指令这样的逻辑，正如在1.3.18节对sb模块描述的那样，实现逻辑是通过计算得到的地址先经过addr2dm模块的转换，然后先从数据存储器根据转换后的地址得到存放在存储器对应位置的一个字信息，即32位大小的数据，然后将读出来的内容送到sb模块，在sb模块中，同时还会传入在ALU运算单元计算得到访问数据存储器的地址，以及控制是否需要按字节写入的IsByteB控制信号，当这个信号为真时，根据具体地址和小端存储的方式修改从数据存储器读出来的内容指定位置的字节并将修改字节后的对应字送到数据存储器的写端口重新写入。

（32）立即数按位与：andi rt,rs,imm

ALU运算单元中采取按位与运算，且第二个源操作数选择经过零扩展的立即数进行运算即可，并将结果写回rt寄存器。

（33）立即数按位或：ori rt,rs,imm

ALU运算单元中采取按位或运算，且第二个源操作数选择经过零扩展的立即数进行运算即可，结果写回rt寄存器。

（34）立即数按位异或：xori rt,rs,imm

ALU运算单元中采取按位异或运算，且第二个源操作数选择经过零扩展的立即数，运算的结果写回rt寄存器即可。

### 1.6.3 J型指令

（35）直接跳转：j target

控制单元会生成直接跳转指令相关的控制信号，并将26位的target字段信息送到npc单元中，npc模块会根据对应的跳转信号和target来计算下一条PC的值，由于PC的起始地址为0x00003000，故在计算j指令的下一条指令的地址时偏移量需要从0x00003000进行计算。

（36）跳转和链接：jal target

与上一条直接跳转指令的区别是本条指令需要链接，这可以通过生成IsLink控制信号来让寄存器堆将当前PC+4的值写入31号ra寄存器中去。其余的跳转逻辑是一致的。

# 第二章 代码实现和运行结果验证

## 2.1 具体Verilog代码实现

### 2.1.1 PC模块

module pc (

in\_pc,clk,reset,out\_pc,

Load\_Use

);

input[31:0] in\_pc;

input clk,reset;

input Load\_Use;

output reg[31:0] out\_pc;

initial begin

out\_pc=32'h00003000; // 0x00003000

end

always @(posedge clk) begin

if(Load\_Use!=1)begin

out\_pc=in\_pc;

end

if(reset)

out\_pc=32'h00003000; // 0x00003000

end

endmodule

### 2.1.2 NPC模块

module npc (

pc\_add,out\_pc,imm32,target,RegtoJump,Jumpctr,Branchctr,Zero,Sign,in\_pc

);

input[31:0] pc\_add,out\_pc;

input[31:0] imm32;

input[25:0] target;

input[31:0] RegtoJump;

input[1:0] Jumpctr;

input[2:0] Branchctr;

input Zero,Sign;

output reg[31:0] in\_pc;

initial begin

in\_pc<=32'h00003000;

end

always @(\*) begin

in\_pc<=pc\_add+4;

if(Jumpctr!=0)begin

// 是跳转

if(Jumpctr==2'b01)begin

// in\_pc<={out\_pc[31:28],target<<2};

in\_pc<=32'h00003000+(target<<2);

end

else if(Jumpctr==2'b10)begin

// in\_pc<=RegtoJump;

in\_pc<=32'h00003000+RegtoJump;

end

end

else if(Branchctr!=0)begin

if(Branchctr==3'b001)begin

// 相等跳转

if(Zero==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b010)begin

// 不等跳转

if(Zero!=1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b011)begin

// 大于等于0跳转

if(Zero==1 || Sign==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b100)begin

// 小于0跳转

if(Zero!=1 && Sign==0)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b101)begin

// 大于0跳转

if(Zero!=1 && Sign==1)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

else if(Branchctr==3'b110)begin

// 小于等于0跳转

if(Zero==1 || Sign==0)begin

in\_pc<=out\_pc+(imm32<<2);

end

end

end

end

endmodule

### 2.1.3 pc2addr模块

module pc2addr (

out\_pc,addr

);

input[31:0] out\_pc;

output[11:2] addr;

wire[31:0] temp\_pc;

assign temp\_pc=out\_pc-32'h00003000;

assign addr=temp\_pc[11:2];

endmodule

### 2.1.4 IM模块

module im\_4k (

addr,dout

);

input[11:2] addr;

output[31:0] dout;

reg[31:0] im[0:1023];

initial begin

$readmemh("code.txt",im);

// $readmemh("code1.txt",im);

// $readmemh("code2.txt",im);

// $readmemh("code3.txt",im);

// $readmemh("code4.txt",im);

// $readmemh("code5.txt",im);

// $readmemh("code6.txt",im);

end

assign dout=im[addr];

endmodule

### 2.1.5 IF/ID模块

module IF\_IDseg (

clk,pc\_in,ins,

pc\_out,op,rs,rt,rd,shf,func,imm16,target,

Load\_Use

);

input clk;

input[31:0] pc\_in,ins;

input Load\_Use;

output reg[31:0] pc\_out;

output reg[5:0] op;

output reg[4:0] rs,rt,rd;

output reg[4:0] shf;

output reg[5:0] func;

output reg[15:0] imm16;

output reg[25:0] target;

always @(negedge clk) begin

if(Load\_Use!=1)begin

pc\_out<=pc\_in;

op<=ins[31:26];

rs<=ins[25:21];

rt<=ins[20:16];

rd<=ins[15:11];

shf<=ins[10:6];

func<=ins[5:0];

imm16<=ins[15:0];

target<=ins[25:0];

end

end

endmodule

### 2.1.6 registers模块

module registers (

clk,rs,rt,w\_rt,w\_rd,

RegDst,RegWr,IsLink,out\_pc,IsByteW,ExtopM,

busA,busB,busW

);

input clk,RegDst,RegWr,IsLink,IsByteW,ExtopM;

input[4:0] rs,rt,w\_rt,w\_rd;

input[31:0] out\_pc;

input[31:0] busW;

output reg[31:0] busA,busB;

reg[31:0] regs[0:31];

integer i,j;

initial begin

i=0;

j=0;

for(i=0;i<32;i=i+1)begin

regs[i]<=0;

end

end

always @(posedge clk) begin

if(RegWr)begin

if(IsLink)begin

regs[31]=out\_pc+4;

end

if(RegDst)begin

if(IsByteW)begin

if(ExtopM)begin

regs[w\_rd]={{24{busW[7]}},busW[7:0]};

end

else begin

regs[w\_rd]={24'b0,busW[7:0]};

end

end

else begin

regs[w\_rd]=busW;

end

end

else begin

if(IsByteW)begin

regs[w\_rt]={24'b0,busW[7:0]};

end

else begin

regs[w\_rt]=busW;

end

end

end

end

always @(\*) begin

busA<=regs[rs];

busB<=regs[rt];

end

endmodule

### 2.1.7 dm\_in\_mux模块

// 写数据存储器的多路选择器

module dm\_in\_mux (

busB,Result,mem\_result,DMSrc,

dm\_write

);

input[31:0] busB,Result,mem\_result;

input[1:0] DMSrc;

output reg[31:0] dm\_write;

initial begin

dm\_write=0;

end

always @(\*) begin

if(DMSrc==2'b00)begin

dm\_write=busB;

end

else if(DMSrc==2'b01)begin

dm\_write=Result;

end

else if(DMSrc==2'b10)begin

dm\_write=mem\_result;

end

end

endmodule

### 2.1.8 ID/Ex模块

`include "ctrl.v"

module ID\_EXseg (

clk,pc\_in,op\_in,func\_in,rt\_in,rd\_in,rs\_in,shf\_in,target\_in,imm16\_in,busA\_in,busB\_in,

rt\_out,rd\_out,rs\_out,pc\_out,shf\_out,target\_out,imm16\_out,busA\_out,busB\_out,

RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,

ALUctr,Jumpctr,Branchctr,MemRead,

Load\_Use,Branch\_fc,Jump\_fc

);

input clk;

input[31:0] pc\_in;

input[5:0] op\_in,func\_in;

input[4:0] rt\_in,rd\_in,rs\_in,shf\_in;

input[25:0] target\_in;

input[15:0] imm16\_in;

input[31:0] busA\_in,busB\_in;

input Load\_Use,Branch\_fc,Jump\_fc;

output reg[31:0] pc\_out;

output reg[4:0] rt\_out,rd\_out,rs\_out,shf\_out;

output reg[25:0] target\_out;

output reg[15:0] imm16\_out;

output reg[31:0] busA\_out,busB\_out;

output reg RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB;

output reg MemRead;

output reg[3:0] ALUctr;

output reg[1:0] Jumpctr;

output reg[2:0] Branchctr;

wire RegDst\_t,ALUsrc\_t,MemtoReg\_t,RegWr\_t,MemWr\_t,Extop\_t,ExtopM\_t,IsLink\_t,IsByteW\_t,IsByteB\_t;

wire MemRead\_t;

wire[3:0] ALUctr\_t;

wire[1:0] Jumpctr\_t;

wire[2:0] Branchctr\_t;

control my\_control(op\_in,func\_in,rt\_in,RegDst\_t,ALUsrc\_t,MemtoReg\_t,RegWr\_t,MemWr\_t,

Extop\_t,ExtopM\_t,IsLink\_t,IsByteW\_t,IsByteB\_t,ALUctr\_t,Jumpctr\_t,Branchctr\_t,MemRead\_t);

always @(negedge clk) begin

pc\_out<=pc\_in;

rt\_out<=rt\_in;

rd\_out<=rd\_in;

rs\_out<=rs\_in;

shf\_out<=shf\_in;

target\_out<=target\_in;

imm16\_out<=imm16\_in;

busA\_out<=busA\_in;

busB\_out<=busB\_in;

RegDst<=RegDst\_t;

ALUsrc<=ALUsrc\_t;

MemtoReg<=MemtoReg\_t;

RegWr<=RegWr\_t;

MemWr<=MemWr\_t;

Extop<=Extop\_t;

ExtopM<=ExtopM\_t;

IsLink<=IsLink\_t;

IsByteW<=IsByteW\_t;

IsByteB<=IsByteB\_t;

ALUctr<=ALUctr\_t;

Jumpctr<=Jumpctr\_t;

Branchctr<=Branchctr\_t;

MemRead<=MemRead\_t;

if(Load\_Use || Branch\_fc || Jump\_fc)begin

pc\_out<=0;

rt\_out<=0;

rd\_out<=0;

rs\_out<=0;

shf\_out<=0;

target\_out<=0;

imm16\_out<=0;

busA\_out<=0;

busB\_out<=0;

RegDst<=0;

ALUsrc<=0;

MemtoReg<=0;

RegWr<=0;

MemWr<=0;

Extop<=0;

ExtopM<=0;

IsLink<=0;

IsByteW<=0;

IsByteB<=0;

ALUctr<=0;

Jumpctr<=0;

Branchctr<=0;

MemRead<=0;

end

end

endmodule

### 2.1.9 Control模块

module control (

op,func,rt,

RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,

ALUctr,Jumpctr,Branchctr,

MemRead

);

input[5:0] op,func;

input[4:0] rt;

output reg RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB;

output reg MemRead;

output reg[3:0] ALUctr;

output reg[1:0] Jumpctr;

output reg[2:0] Branchctr;

// 使用朴素的判别方式

always @(\*) begin

if(op==6'b000000)begin

// R型

// init

Branchctr=3'b000;

Jumpctr=2'b00;

ALUctr=4'b0000;

RegDst=1;

ALUsrc=0;

MemtoReg=0;

RegWr=1;

MemWr=0;

Extop=0;

ExtopM=0;

IsLink=0;

IsByteW=0;

IsByteB=0;

MemRead=0;

if(func==6'b100001)begin

// addu

ALUctr=4'b0000; // 无符号加

end

else if(func==6'b100011)begin

// subu

ALUctr=4'b0001; // 无符号减

end

else if(func==6'b101010)begin

// slt

ALUctr=4'b0010; // 小于置位

end

else if(func==6'b100100)begin

// and

ALUctr=4'b0011; // 按位与

end

else if(func==6'b100111)begin

// nor

ALUctr=4'b0100; // 按位或非

end

else if(func==6'b100101)begin

// or

ALUctr=4'b0101; // 按位或

end

else if(func==6'b100110)begin

// xor

ALUctr=4'b0110; // 按位异或

end

else if(func==6'b000000)begin

// sll

ALUctr=4'b0111; // 逻辑左移

end

else if(func==6'b000010)begin

// srl

ALUctr=4'b1000; // 逻辑右移

end

else if(func==6'b101011)begin

// sltu

ALUctr=4'b0010; // 小于置位

end

else if(func==6'b001001)begin

// jalr

ALUctr=4'b1001; // jalr特殊1

Jumpctr=2'b10;

end

else if(func==6'b001000)begin

// jr

ALUctr=4'b1010; // jr特殊2

Jumpctr=2'b10;

end

else if(func==6'b000100)begin

// sllv

ALUctr=4'b1011; // 变量的左移

end

else if(func==6'b000011)begin

// sra

ALUctr=4'b1100; // 算术右移

end

else if(func==6'b000111)begin

// srav

ALUctr=4'b1101; // 变量的算术右移

end

else if(func==6'b000110)begin

// srlv

ALUctr=4'b1110; // 变量的逻辑右移

end

end

else begin

// 非R型

Branchctr=3'b000;

Jumpctr=2'b00;

ALUctr=4'b0000;

RegDst=0;

ALUsrc=1;

MemtoReg=0;

RegWr=1;

MemWr=0;

Extop=0;

ExtopM=0;

IsLink=0;

IsByteW=0;

IsByteB=0;

MemRead=0;

if(op==6'b001001)begin

// addiu

ALUctr=4'b0000;

Extop=1;

end

else if(op==6'b000100)begin

// beq

ALUctr=4'b0001;

Branchctr=3'b001;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b000101)begin

// bne

ALUctr=4'b0001;

Branchctr=3'b010;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b100011)begin

// lw

ALUctr=4'b0000;

Extop=1;

MemtoReg=1;

MemRead=1;

end

else if(op==6'b101011)begin

// sw

ALUctr=4'b0000;

Extop=1;

RegWr=0;

MemWr=1;

end

else if(op==6'b001111)begin

// lui

ALUctr=4'b1111;

end

else if(op==6'b001010)begin

// slti

ALUctr=4'b0010;

Extop=1;

end

else if(op==6'b001011)begin

// sltiu

ALUctr=4'b0010;

Extop=1;

end

else if(op==6'b000001)begin

if(rt==5'b00001)begin

// bgez

ALUctr=4'b0001;

Branchctr=3'b011;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(rt==5'b00000)begin

// bltz

ALUctr=4'b0001;

Branchctr=3'b100;

Extop=1;

ALUsrc=0;

RegWr=0;

end

end

else if(op==6'b000111)begin

// bgtz

ALUctr=4'b0001;

Branchctr=3'b101;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b000110)begin

// blez

ALUctr=4'b0001;

Branchctr=3'b110;

Extop=1;

ALUsrc=0;

RegWr=0;

end

else if(op==6'b100000)begin

// lb

ALUctr=4'b0000;

Extop=1;

ExtopM=1;

MemtoReg=1;

IsByteW=1;

MemRead=1;

end

else if(op==6'b100100)begin

// lbu

ALUctr=4'b0000;

Extop=1;

MemtoReg=1;

IsByteW=1;

MemRead=1;

end

else if(op==6'b101000)begin

// sb

ALUctr=4'b0000;

Extop=1;

RegWr=0;

MemWr=1;

IsByteB=1;

MemtoReg=1;

end

else if(op==6'b001100)begin

// andi

ALUctr=4'b0011;

end

else if(op==6'b001101)begin

// ori

ALUctr=4'b0101;

end

else if(op==6'b001110)begin

// xori

ALUctr=4'b0110;

end

else if(op==6'b000010)begin

// j

Jumpctr=2'b01;

RegWr=0;

end

else if(op==6'b000011)begin

// jal

Jumpctr=2'b01;

RegWr=0;

IsLink=1;

end

end

end

endmodule

### 2.1.10 extend模块

module extend (

imm16,imm32,Extop

);

input[15:0] imm16;

input Extop;

output[31:0] imm32;

reg[31:0] temp;

always @(\*) begin

if(Extop==0)

temp={16'b0,imm16[15:0]}; // 零扩展

else

temp={{16{imm16[15]}},imm16[15:0]}; // 符号扩展

end

assign imm32=temp;

endmodule

### 2.1.11 alu\_in1\_mux模块

module alu\_in1\_mux (

busA,Result,mem\_result,ALUSrcA,

in1

);

input[31:0] busA,Result,mem\_result;

input[1:0] ALUSrcA;

output reg[31:0] in1;

initial begin

in1=0;

end

always @(\*) begin

if(ALUSrcA==2'b00)begin

in1=busA;

end

else if(ALUSrcA==2'b01)begin

in1=Result;

end

else if(ALUSrcA==2'b10)begin

in1=mem\_result;

end

end

endmodule

### 2.1.12 alu\_in2\_mux模块

module alu\_in2\_mux (

busB,Result,mem\_result,imm32,ALUSrcB,

in2

);

input[31:0] busB,Result,mem\_result,imm32;

input[1:0] ALUSrcB;

output reg[31:0] in2;

initial begin

in2=0;

end

always @(\*) begin

if(ALUSrcB==2'b00)begin

in2=busB;

end

else if(ALUSrcB==2'b01)begin

in2=Result;

end

else if(ALUSrcB==2'b10)begin

in2=mem\_result;

end

else if(ALUSrcB==2'b11)begin

in2=imm32;

end

end

endmodule

### 2.1.13 alu模块

module alu (

ALUctr,in1,in2,shf,out\_pc,Zero,Sign,Result,RegtoJump

);

input[3:0] ALUctr;

input[31:0] in1,in2;

input[4:0] shf;

input[31:0] out\_pc;

output reg Zero,Sign;

output reg[31:0] Result,RegtoJump;

initial begin

Zero=0;

Sign=0;

Result=0;

RegtoJump=0;

end

always @(\*) begin

case(ALUctr)

4'b0000:begin

// 无符号加

Result=in1+in2;

Zero=0;

Sign=0;

end

4'b0001:begin

// 无符号减

Result=in1-in2;

if(Result==0)begin

Zero=1;

end

else begin

Zero=0;

if($signed(Result)<0)begin

Sign=0;

end

else begin

Sign=1;

end

end

end

4'b0010:begin

// 小于置位

if(in1<in2)begin

Result=1;

end

else begin

Result=0;

end

end

4'b0011:begin

// 按位与

Result=in1 & in2;

end

4'b0100:begin

// 按位或非

Result=~(in1 | in2);

end

4'b0101:begin

// 按位或

Result=in1 | in2;

end

4'b0110:begin

// 按位异或

Result=in1 ^ in2;

end

4'b0111:begin

// 逻辑左移

Result=in2 << shf;

end

4'b1000:begin

// 逻辑右移

Result=in2 >> shf;

end

4'b1001:begin

// jalr

RegtoJump=in1;

Result=out\_pc+4;

end

4'b1010:begin

// jr

RegtoJump=in1;

Result=0;

end

4'b1011:begin

// 变量逻辑左移

Result=in2 << in1;

end

4'b1100:begin

// 算术右移

Result=$signed(in2) >>> shf;

end

4'b1101:begin

// 变量算术右移

Result=$signed(in2) >>> in1;

end

4'b1110:begin

// 变量的逻辑右移

Result=in2 >> in1;

end

4'b1111:begin

// lui特殊运算

Result={in2,16'd0};

end

endcase

end

endmodule

### 2.1.14 LoadUse\_detect模块

module loaduse\_detect (

ID\_Ex\_MemRead,ID\_Ex\_rt,IF\_ID\_rs,IF\_ID\_rt,

Load\_Use

);

input ID\_Ex\_MemRead;

input[4:0] ID\_Ex\_rt,IF\_ID\_rs,IF\_ID\_rt;

output reg Load\_Use;

initial begin

Load\_Use=0;

end

always @(\*) begin

if(ID\_Ex\_MemRead && (ID\_Ex\_rt==IF\_ID\_rs || ID\_Ex\_rt==IF\_ID\_rt))begin

Load\_Use<=1;

end

else begin

Load\_Use<=0;

end

end

endmodule

### 2.1.15 Ex/Mem模块

module EX\_MEMseg (

clk,pc\_in,rt\_in,rd\_in,target\_in,imm32\_in,busB\_in,

Zero\_in,Sign\_in,Result\_in,RegtoJump\_in,

RegDst\_in,MemtoReg\_in,RegWr\_in,MemWr\_in,ExtopM\_in,IsLink\_in,IsByteW\_in,IsByteB\_in,

Jumpctr\_in,Branchctr\_in,

pc\_out,rt\_out,rd\_out,target\_out,imm32\_out,busB\_out,

Zero\_out,Sign\_out,Result\_out,RegtoJump\_out,

RegDst\_out,MemtoReg\_out,RegWr\_out,MemWr\_out,ExtopM\_out,IsLink\_out,IsByteW\_out,IsByteB\_out,

Jumpctr\_out,Branchctr\_out,

Branch\_fc,Jump\_fc

);

input clk,Zero\_in,Sign\_in;

input RegDst\_in,MemtoReg\_in,RegWr\_in,MemWr\_in,ExtopM\_in,IsLink\_in,IsByteW\_in,IsByteB\_in;

input[4:0] rt\_in,rd\_in;

input[31:0] pc\_in,imm32\_in,busB\_in,Result\_in,RegtoJump\_in;

input[25:0] target\_in;

input[1:0] Jumpctr\_in;

input[2:0] Branchctr\_in;

input Branch\_fc,Jump\_fc;

output reg Zero\_out,Sign\_out;

output reg RegDst\_out,MemtoReg\_out,RegWr\_out,MemWr\_out,ExtopM\_out,IsLink\_out,IsByteW\_out,IsByteB\_out;

output reg[4:0] rt\_out,rd\_out;

output reg[25:0] target\_out;

output reg[31:0] pc\_out,imm32\_out,busB\_out,Result\_out,RegtoJump\_out;

output reg[1:0] Jumpctr\_out;

output reg[2:0] Branchctr\_out;

always @(negedge clk) begin

pc\_out<=pc\_in;

rt\_out<=rt\_in;

rd\_out<=rd\_in;

target\_out<=target\_in;

imm32\_out<=imm32\_in;

busB\_out<=busB\_in;

Zero\_out<=Zero\_in;

Sign\_out<=Sign\_in;

Result\_out<=Result\_in;

RegtoJump\_out<=RegtoJump\_in;

RegDst\_out<=RegDst\_in;

MemtoReg\_out<=MemtoReg\_in;

RegWr\_out<=RegWr\_in;

MemWr\_out<=MemWr\_in;

ExtopM\_out<=ExtopM\_in;

IsLink\_out<=IsLink\_in;

IsByteW\_out<=IsByteW\_in;

IsByteB\_out<=IsByteB\_in;

Jumpctr\_out<=Jumpctr\_in;

Branchctr\_out<=Branchctr\_in;

if(Branch\_fc || Jump\_fc)begin

pc\_out<=0;

rt\_out<=0;

rd\_out<=0;

target\_out<=0;

imm32\_out<=0;

busB\_out<=0;

Zero\_out<=0;

Sign\_out<=0;

Result\_out<=0;

RegtoJump\_out<=0;

RegDst\_out<=0;

MemtoReg\_out<=0;

RegWr\_out<=0;

MemWr\_out<=0;

ExtopM\_out<=0;

IsLink\_out<=0;

IsByteW\_out<=0;

IsByteB\_out<=0;

Jumpctr\_out<=0;

Branchctr\_out<=0;

end

end

endmodule

### 2.1.16 addr2dm模块

module addr2dm (

in\_addr,addr

);

input[31:0] in\_addr;

output[11:2] addr;

assign addr=in\_addr[11:2];

endmodule

### 2.1.17 dm\_4k模块

module dm\_4k (

addr,din,we,clk,dout

);

input[11:2] addr;

input[31:0] din;

input we,clk;

output reg[31:0] dout;

reg[31:0] dm[0:1023];

integer i;

initial begin

for(i=0;i<1024;i=i+1)begin

dm[i]<=0;

end

end

always @(negedge clk) begin

if(we==1)begin

dm[addr]<=din;

end

end

always @(\*) begin

dout<=dm[addr];

end

endmodule

### 2.1.18 sb模块

module sb (

dm\_addr,busB,dm\_read,dm\_write,

IsByteB

);

input IsByteB;

input[31:0] dm\_addr;

input[31:0] busB,dm\_read;

output reg[31:0] dm\_write;

integer j;

initial begin

dm\_write=0;

j=0;

end

always @(\*) begin

if(IsByteB)begin

j=dm\_addr%4;

if(j==0)begin

dm\_write<={dm\_read[31:8],busB[7:0]};

end

else if(j==1)begin

dm\_write<={dm\_read[31:16],busB[7:0],dm\_read[7:0]};

end

else if(j==2)begin

dm\_write<={dm\_read[31:24],busB[7:0],dm\_read[15:0]};

end

else begin

dm\_write<={busB[7:0],dm\_read[23:0]};

end

end

else begin

dm\_write<=busB;

end

end

endmodule

### 2.1.19 Branch\_fc模块

module branch\_fc (

Branchctr,Zero,Sign,

Branch\_fc

);

input[2:0] Branchctr;

input Zero,Sign;

output reg Branch\_fc;

always @(\*) begin

Branch\_fc=0;

if(Branchctr==3'b001)begin

// 相等跳转

if(Zero==1)begin

Branch\_fc=1;

end

end

else if(Branchctr==3'b010)begin

// 不等跳转

if(Zero!=1)begin

Branch\_fc=1;

end

end

else if(Branchctr==3'b011)begin

// 大于等于0跳转

if(Zero==1 || Sign==1)begin

Branch\_fc=1;

end

end

else if(Branchctr==3'b100)begin

// 小于0跳转

if(Zero!=1 && Sign==0)begin

Branch\_fc=1;

end

end

else if(Branchctr==3'b101)begin

// 大于0跳转

if(Zero!=1 && Sign==1)begin

Branch\_fc=1;

end

end

else if(Branchctr==3'b110)begin

// 小于等于0跳转

if(Zero==1 || Sign==0)begin

Branch\_fc=1;

end

end

end

endmodule

### 2.1.20 Jump\_fc模块

// jump和branch统一放到mem段处理了

module jump\_fc (

Jumpctr,

Jump\_fc

);

input[1:0] Jumpctr;

output reg Jump\_fc;

initial begin

Jump\_fc=0;

end

always @(\*) begin

if(Jumpctr==2'b00)begin

// 不跳转预测正确

Jump\_fc=0;

end

else begin

Jump\_fc=1;

end

end

endmodule

### 2.1.21 forward\_detect模块

module forward\_detect (

Ex\_Mem\_RegWr,

Ex\_Mem\_rt,Ex\_Mem\_rd,Ex\_Mem\_RegDst,

Mem\_Wr\_RegWr,

Mem\_Wr\_rt,Mem\_Wr\_rd,Mem\_Wr\_RegDst,

ID\_Ex\_rs,ID\_Ex\_rt,

ALUSrc,

ALUSrcA,ALUSrcB,DMSrc

);

input Ex\_Mem\_RegWr,Mem\_Wr\_RegWr;

input Ex\_Mem\_RegDst,Mem\_Wr\_RegDst;

input ALUSrc;

input[4:0] Ex\_Mem\_rt,Ex\_Mem\_rd,Mem\_Wr\_rt,Mem\_Wr\_rd;

input[4:0] ID\_Ex\_rs,ID\_Ex\_rt;

output reg[1:0] ALUSrcA,ALUSrcB,DMSrc;

reg[4:0] Ex\_Mem\_regRd,Mem\_Wr\_regRd;

initial begin

ALUSrcA=0;

ALUSrcB=0;

DMSrc=0;

end

always @(\*) begin

if(Ex\_Mem\_RegDst)begin

Ex\_Mem\_regRd=Ex\_Mem\_rd;

end

else begin

Ex\_Mem\_regRd=Ex\_Mem\_rt;

end

if(Mem\_Wr\_RegDst)begin

Mem\_Wr\_regRd=Mem\_Wr\_rd;

end

else begin

Mem\_Wr\_regRd=Mem\_Wr\_rt;

end

if(Ex\_Mem\_RegWr && Ex\_Mem\_regRd!=0 && Ex\_Mem\_regRd==ID\_Ex\_rs)begin

// C1A为真

ALUSrcA=2'b01;

end

else if(Mem\_Wr\_RegWr && Mem\_Wr\_regRd!=0 && (Ex\_Mem\_RegWr==0 || Ex\_Mem\_regRd!=ID\_Ex\_rs) && Mem\_Wr\_regRd==ID\_Ex\_rs)begin

// C2A为真

ALUSrcA=2'b10;

end

else begin

ALUSrcA=2'b00;

end

if(ALUSrc==1)begin

ALUSrcB=2'b11;

if(Ex\_Mem\_RegWr && Ex\_Mem\_regRd!=0 && Ex\_Mem\_regRd==ID\_Ex\_rt)begin

DMSrc=2'b01;

end

else if(Mem\_Wr\_RegWr && Mem\_Wr\_regRd!=0 && Ex\_Mem\_regRd!=ID\_Ex\_rt && Mem\_Wr\_regRd==ID\_Ex\_rt)begin

DMSrc=2'b10;

end

else begin

DMSrc=2'b00;

end

end

else begin

if(Ex\_Mem\_RegWr && Ex\_Mem\_regRd!=0 && Ex\_Mem\_regRd==ID\_Ex\_rt)begin

// C1B为真

ALUSrcB=2'b01;

DMSrc=2'b01;

end

else if(Mem\_Wr\_RegWr && Mem\_Wr\_regRd!=0 && Ex\_Mem\_regRd!=ID\_Ex\_rt && Mem\_Wr\_regRd==ID\_Ex\_rt)begin

// C2B为真

ALUSrcB=2'b10;

DMSrc=2'b10;

end

else begin

ALUSrcB=2'b00;

DMSrc=2'b00;

end

end

end

endmodule

### 2.1.22 Mem/Wr模块

module MEM\_WRseg (

clk,pc\_in,Result\_in,rt\_in,rd\_in,

RegDst\_in,MemtoReg\_in,RegWr\_in,ExtopM\_in,IsLink\_in,IsByteW\_in,

dm\_read\_in,

pc\_out,Result\_out,rt\_out,rd\_out,

RegDst\_out,MemtoReg\_out,RegWr\_out,ExtopM\_out,IsLink\_out,IsByteW\_out,

dm\_read\_out

);

input clk;

input RegDst\_in,MemtoReg\_in,RegWr\_in,ExtopM\_in,IsLink\_in,IsByteW\_in;

input[4:0] rt\_in,rd\_in;

input[31:0] pc\_in,Result\_in;

input[31:0] dm\_read\_in;

output reg RegDst\_out,MemtoReg\_out,RegWr\_out,ExtopM\_out,IsLink\_out,IsByteW\_out;

output reg[4:0] rt\_out,rd\_out;

output reg[31:0] pc\_out,Result\_out;

output reg[31:0] dm\_read\_out;

always @(negedge clk) begin

pc\_out<=pc\_in;

rt\_out<=rt\_in;

rd\_out<=rd\_in;

Result\_out<=Result\_in;

RegDst\_out<=RegDst\_in;

MemtoReg\_out<=MemtoReg\_in;

RegWr\_out<=RegWr\_in;

ExtopM\_out<=ExtopM\_in;

IsLink\_out<=IsLink\_in;

IsByteW\_out<=IsByteW\_in;

dm\_read\_out<= dm\_read\_in;

end

endmodule

### 2.1.23 mux\_memtoreg模块

module mux\_memtoreg (

ALU\_out,datamem\_out,MemtoReg,memtoreg\_out

);

input[31:0] ALU\_out;

input[31:0] datamem\_out;

input MemtoReg;

output reg[31:0] memtoreg\_out;

always @(\*) begin

if(MemtoReg==1)

memtoreg\_out<=datamem\_out;

else

memtoreg\_out<=ALU\_out;

end

endmodule

### 2.1.24 mips模块

`timescale 1ns/1ns

`include "addr2dm.v"

`include "alu.v"

`include "dm.v"

`include "EX\_MEMseg.v"

`include "extend.v"

`include "ID\_EXseg.v"

`include "IF\_IDseg.v"

`include "im.v"

`include "MEM\_WRseg.v"

`include "mux.v"

`include "npc.v"

`include "pc.v"

`include "pc2addr.v"

`include "registers.v"

`include "sb.v"

`include "forward\_detect.v"

`include "aluin\_mux.v"

`include "dmin\_mux.v"

`include "loaduse\_detect.v"

`include "branch\_fc.v"

`include "jump\_fc.v"

module mips (

clk,reset

);

input clk,reset;

wire[31:0] in\_pc,out\_pc,out\_pc1,out\_pc2,out\_pc3,out\_pc4;

wire[11:2] im\_addr;

wire[31:0] ins;

wire[5:0] op,op1;

wire[4:0] rs,rt,rd,rt1,rd1,rs1,rt2,rd2,rt3,rd3;

wire[4:0] shf,shf1;

wire[5:0] func;

wire[15:0] imm16,imm161;

wire[25:0] target,target1,target2;

wire[31:0] imm32,imm321;

wire RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,MemRead;

wire RegDst1,MemtoReg1,RegWr1,MemWr1,Extop1,ExtopM1,IsLink1,IsByteW1,IsByteB1;

wire RegDst2,MemtoReg2,RegWr2,ExtopM2,IsLink2,IsByteW2;

wire[3:0] ALUctr;

wire[1:0] Jumpctr,Jumpctr1;

wire[2:0] Branchctr,Branchctr1;

wire[31:0] busA,busB,busW,busA1,busB1,busB2,busB\_t;

wire[31:0] Result,RegtoJump,Result1,RegtoJump1,Result2;

wire[31:0] in1,in2;

wire Zero,Sign,Zero1,Sign1;

wire[11:2] dm\_addr;

wire[31:0] dm\_dout,dm\_wr,dm\_dout1,dm\_wrpre;

wire[1:0] ALUSrcA,ALUSrcB,DMSrc;

wire Load\_Use,Branch\_fc,Jump\_fc;

pc my\_pc(in\_pc,clk,reset,out\_pc,Load\_Use);

pc2addr my\_pc2addr(out\_pc,im\_addr);

im\_4k my\_im\_4k(im\_addr,ins);

IF\_IDseg my\_IF\_IDseg(clk,out\_pc,ins,out\_pc1,op,rs,rt,rd,shf,func,imm16,target,

Load\_Use);

registers my\_registers(clk,rs,rt,rt3,rd3,RegDst2,RegWr2,IsLink2,out\_pc4,IsByteW2,ExtopM2,busA,busB,busW);

ID\_EXseg my\_ID\_EXseg(clk,out\_pc1,op,func,rt,rd,rs,shf,target,imm16,busA,busB,

rt1,rd1,rs1,out\_pc2,shf1,target1,imm161,busA1,busB1,

RegDst,ALUsrc,MemtoReg,RegWr,MemWr,Extop,ExtopM,IsLink,IsByteW,IsByteB,

ALUctr,Jumpctr,Branchctr,MemRead,

Load\_Use,Branch\_fc,Jump\_fc

);

extend my\_extend(imm161,imm32,Extop);

alu\_in1\_mux my\_alu\_in1\_mux(busA1,Result1,busW,ALUSrcA,

in1);

alu\_in2\_mux my\_alu\_in2\_mux(busB1,Result1,busW,imm32,ALUSrcB,

in2);

alu my\_alu(ALUctr,in1,in2,shf1,out\_pc2,Zero,Sign,Result,RegtoJump);

dm\_in\_mux my\_dm\_in\_mux(busB1,Result1,busW,DMSrc,busB\_t);

loaduse\_detect my\_loaduse\_detect(MemRead,rt1,rs,rt,Load\_Use);

EX\_MEMseg my\_EX\_MEMseg(clk,out\_pc2,rt1,rd1,target1,imm32,busB\_t,

Zero,Sign,Result,RegtoJump,

RegDst,MemtoReg,RegWr,MemWr,ExtopM,IsLink,IsByteW,IsByteB,

Jumpctr,Branchctr,

out\_pc3,rt2,rd2,target2,imm321,busB2,

Zero1,Sign1,Result1,RegtoJump1,

RegDst1,MemtoReg1,RegWr1,MemWr1,ExtopM1,IsLink1,IsByteW1,IsByteB1,

Jumpctr1,Branchctr1,

Branch\_fc,Jump\_fc);

npc my\_npc(out\_pc,out\_pc3,imm321,target2,RegtoJump1,Jumpctr1,Branchctr1,Zero1,Sign1,in\_pc);

addr2dm my\_addr2dm(Result1,dm\_addr);

dm\_4k my\_dm\_4k(dm\_addr,dm\_wr,MemWr1,clk,dm\_dout);

sb my\_sb(Result1,busB2,dm\_dout,dm\_wr,IsByteB1);

// 转发及控制冒险检测单元

forward\_detect my\_forward\_detect(RegWr1,

rt2,rd2,RegDst1,

RegWr2,

rt3,rd3,RegDst2,

rs1,rt1,

ALUsrc,

ALUSrcA,ALUSrcB,DMSrc);

branch\_fc my\_branch\_fc(Branchctr1,Zero1,Sign1,Branch\_fc);

jump\_fc my\_jump\_fc(Jumpctr1,Jump\_fc);

MEM\_WRseg my\_MEM\_WRseg(clk,out\_pc3,Result1,rt2,rd2,

RegDst1,MemtoReg1,RegWr1,ExtopM1,IsLink1,IsByteW1,

dm\_dout,

out\_pc4,Result2,rt3,rd3,

RegDst2,MemtoReg2,RegWr2,ExtopM2,IsLink2,IsByteW2,

dm\_dout1);

mux\_memtoreg my\_mux\_memtoreg(Result2,dm\_dout1,MemtoReg2,busW);

endmodule

### 2.1.25 tmips模块（testbench）

`timescale 1ns/1ns

`include "mips.v"

module testmips;

reg clk,reset;

mips example(clk,reset);

initial begin

$dumpfile("wave.vcd");

$dumpvars;

reset=0;

end

always begin

clk=0;#5;

clk=1;#5;

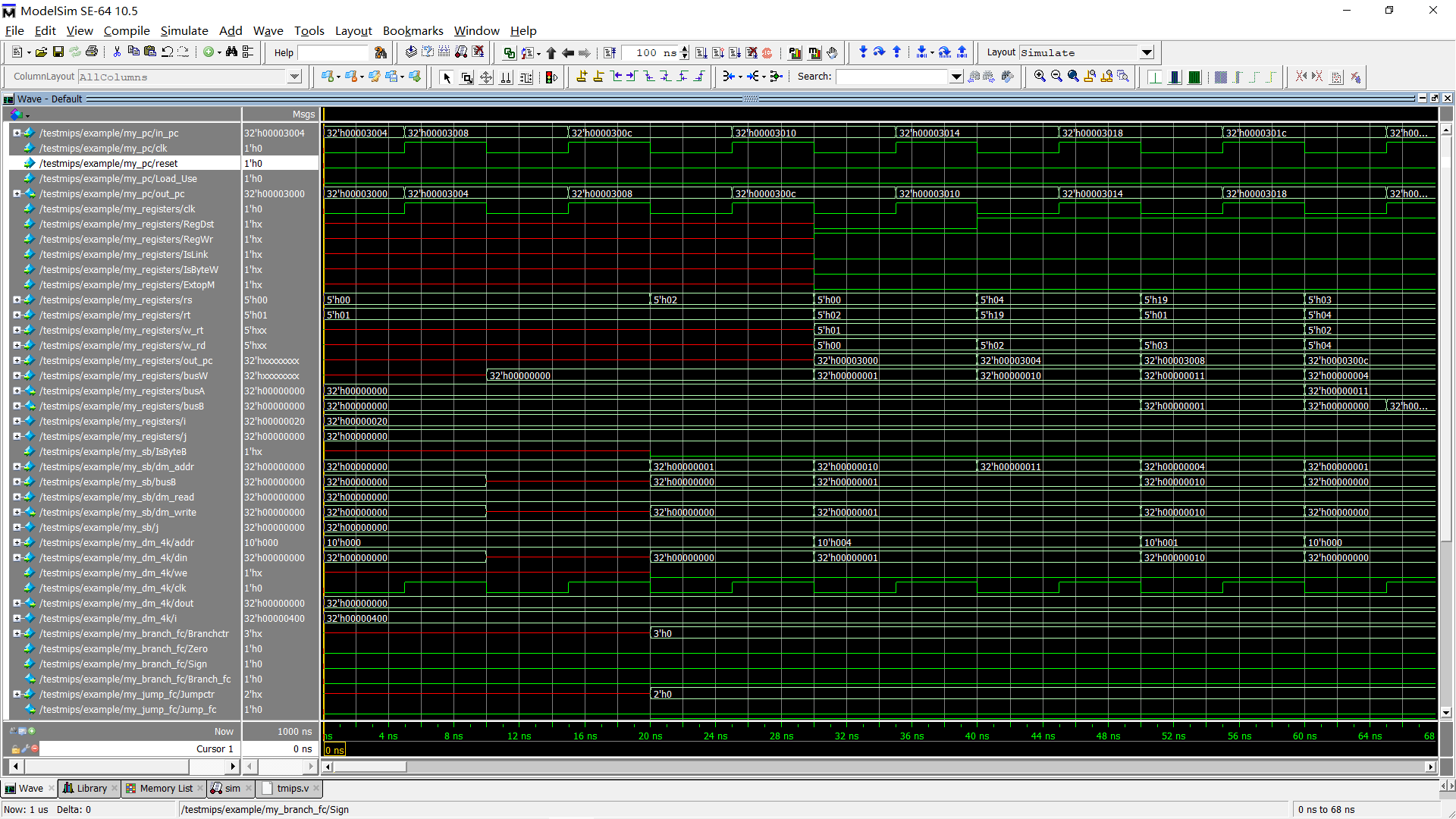
if($time>=1000)$finish;

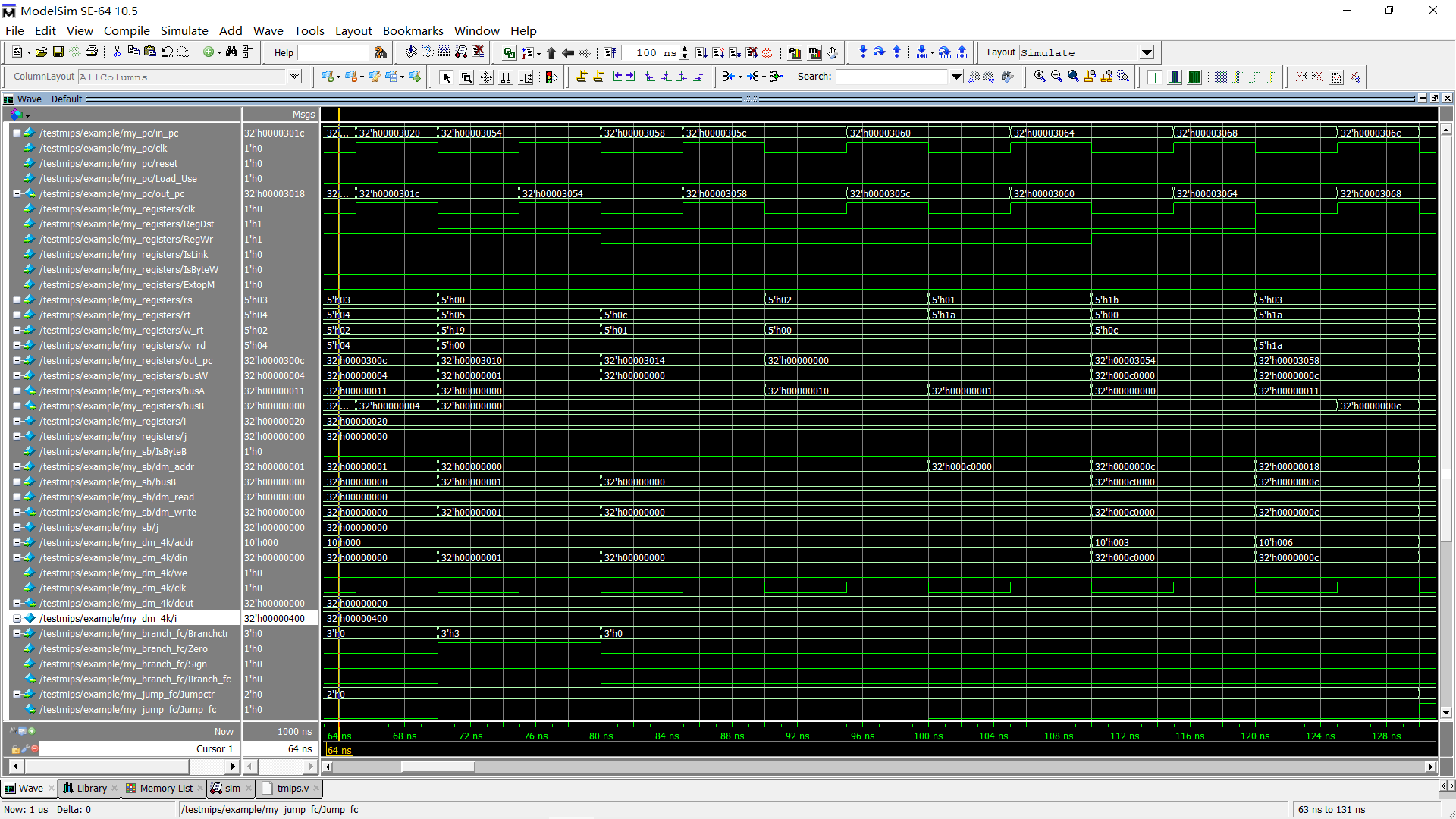
end

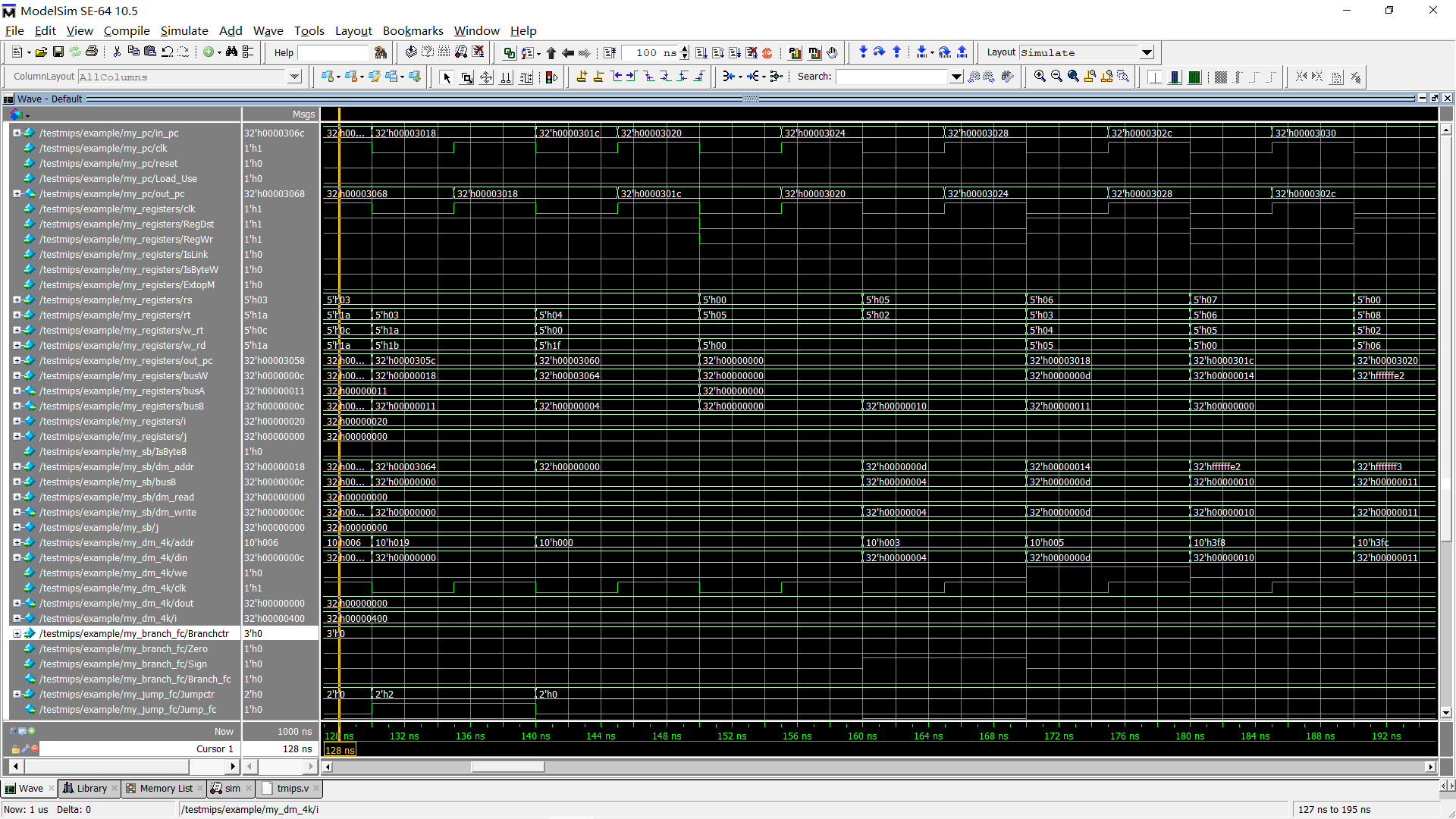
endmodule

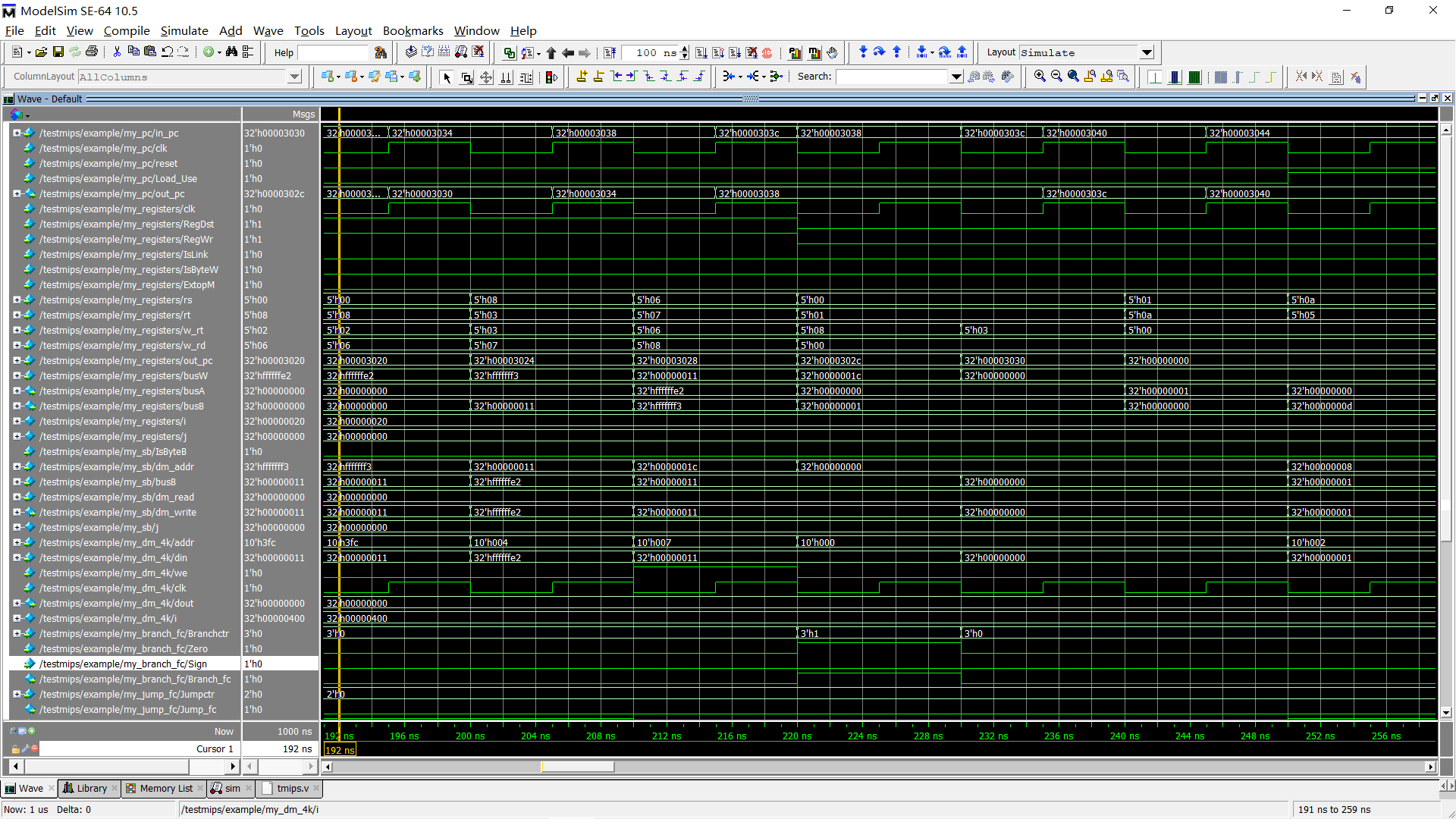
## 2.2 ModelSim上模拟仿真测试指令的信号结果

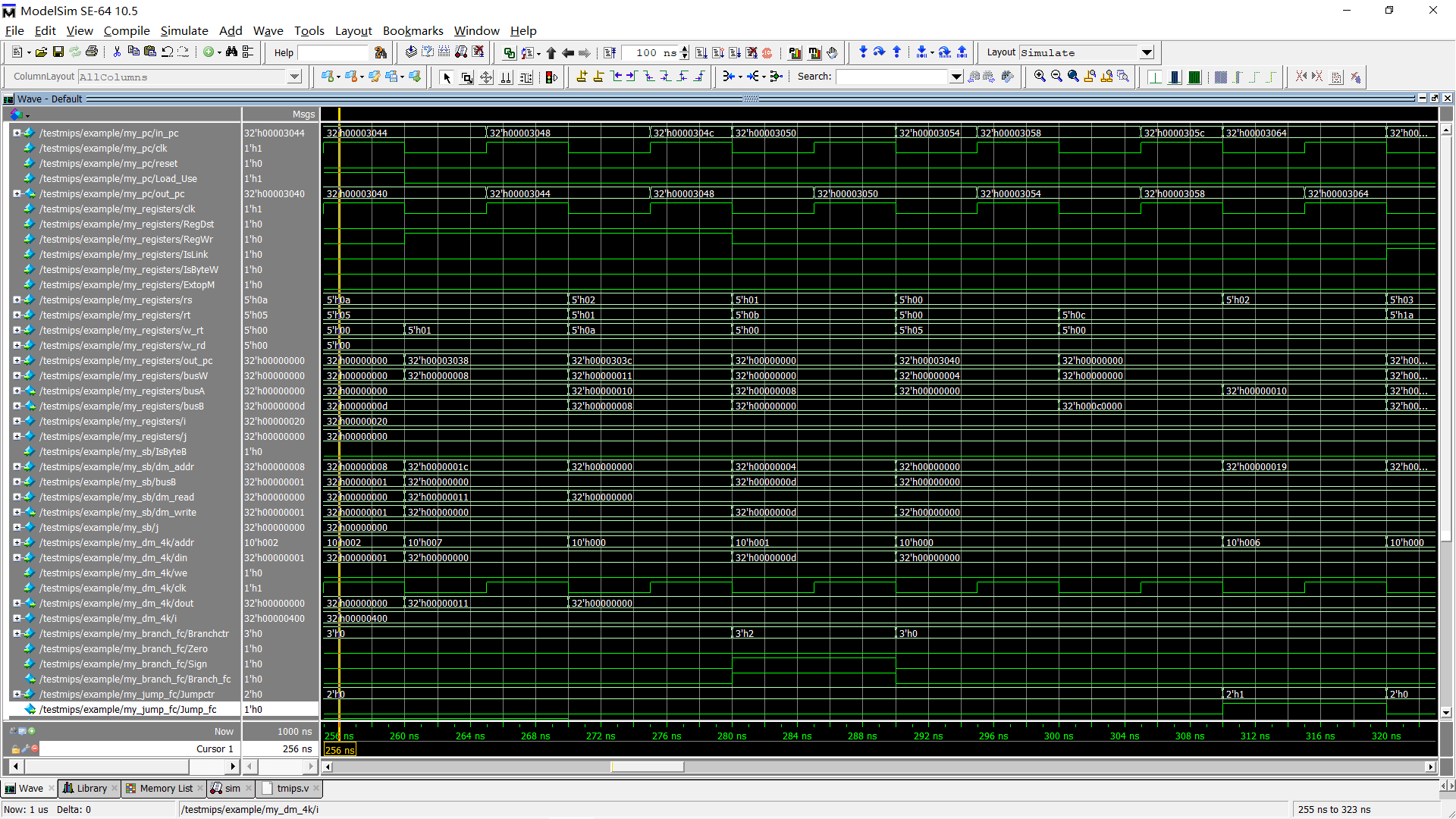
如下的截图是自己设计的流水线CPU在ModelSim仿真上截取的部分重要模块的信号的结果，由于流水线设计的定义的变量和信号过于繁多，导致如果全部加入ModelSim截图时也无法截得全貌，故选取了在观察判断流水线CPU在所给的36条MIPS指令测试程序下的正确性的一些模块的信号输入输出值，这些模块主要为PC模块、寄存器堆模块、数据存储器模块、用于sb指令特别实现的sb模块、转发数据冒险检测模块等等。

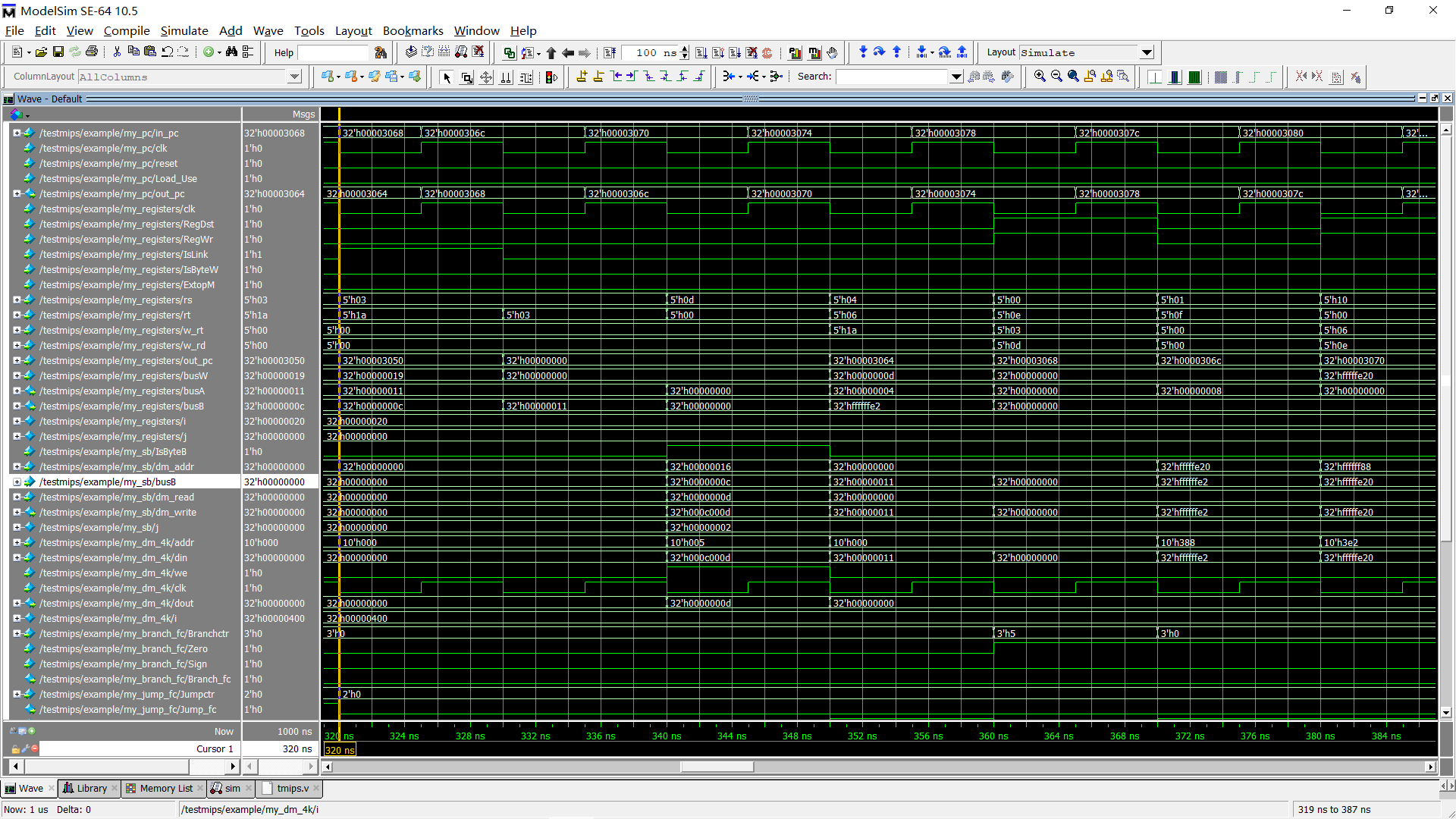


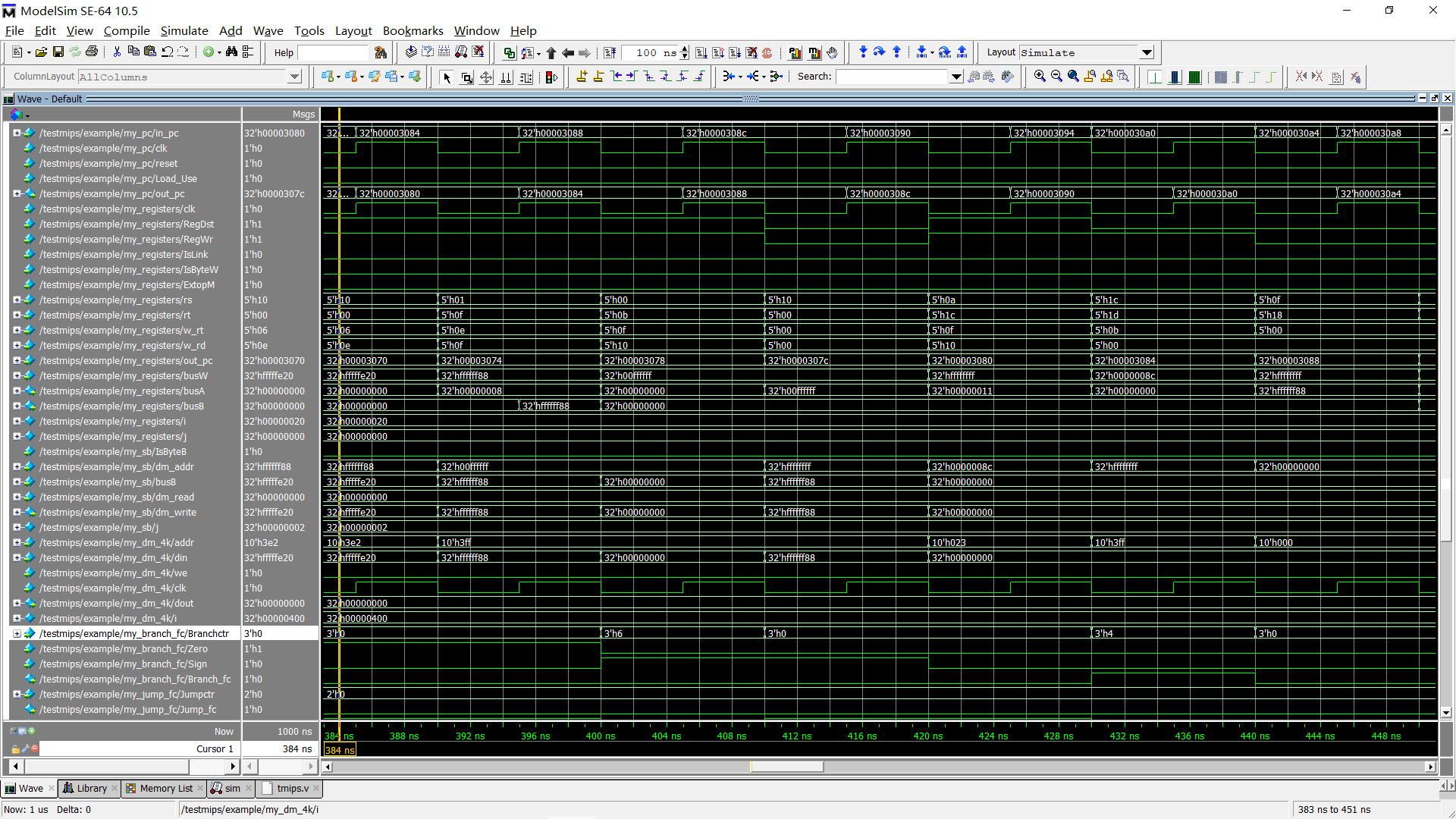


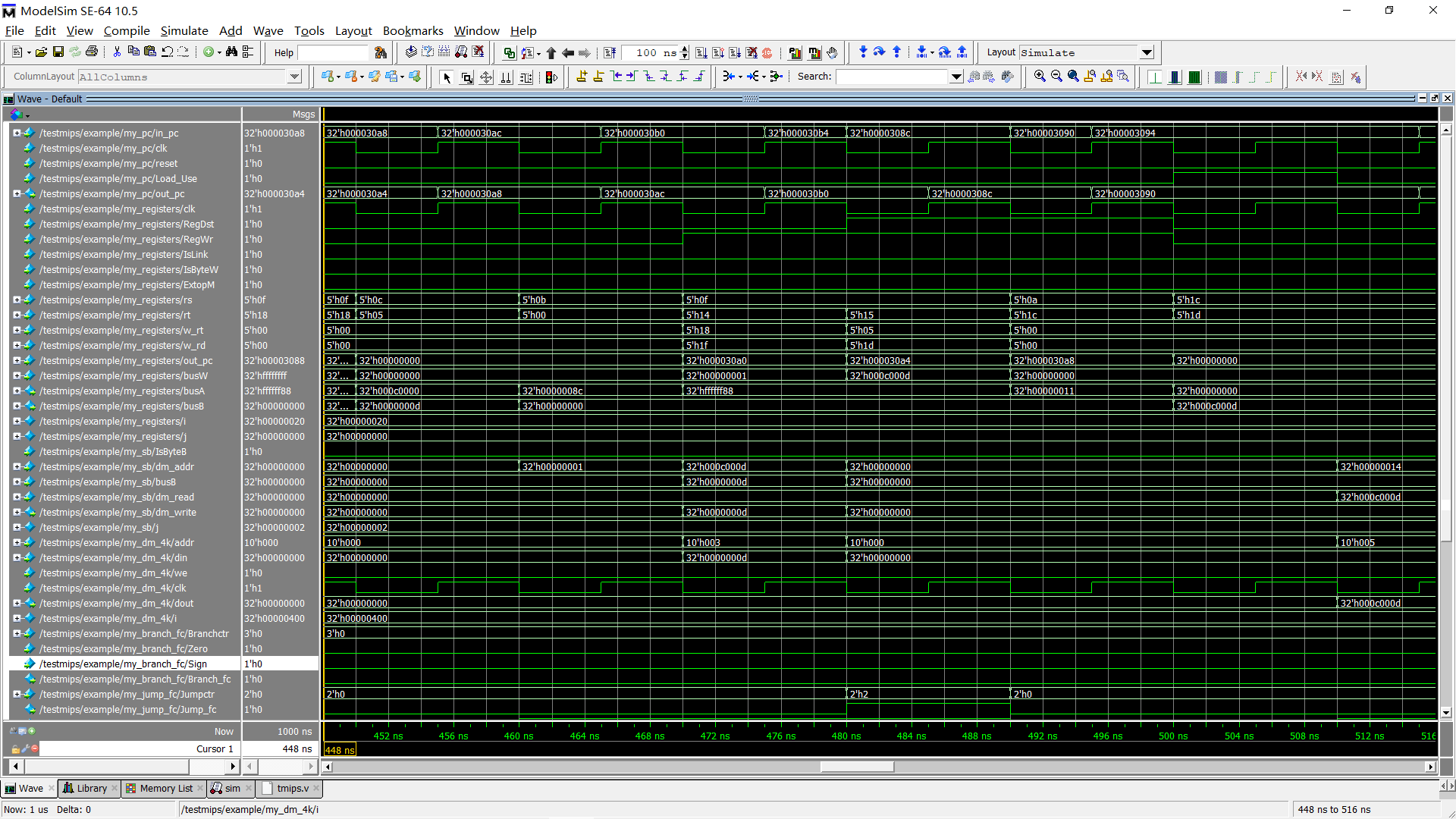


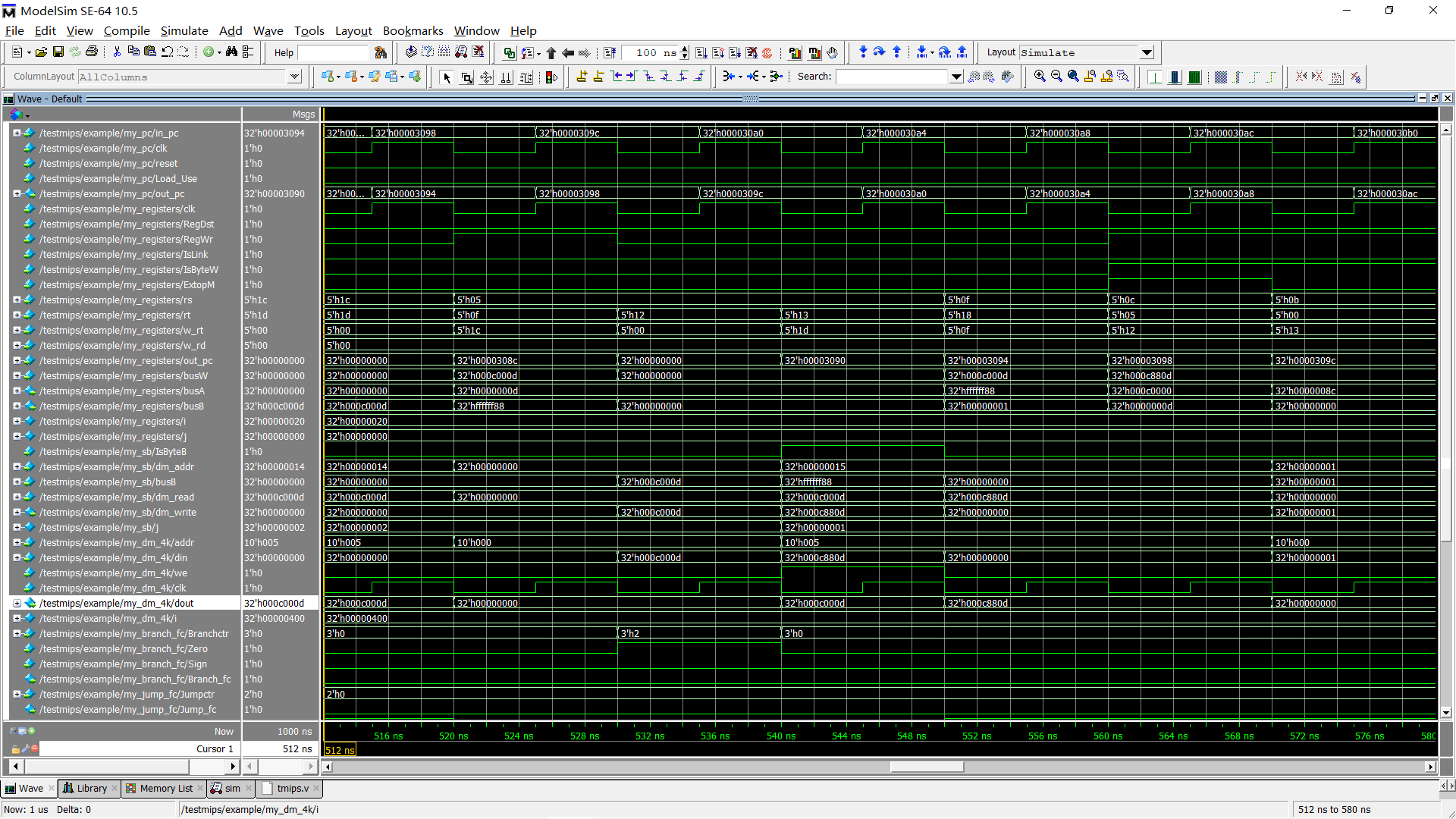


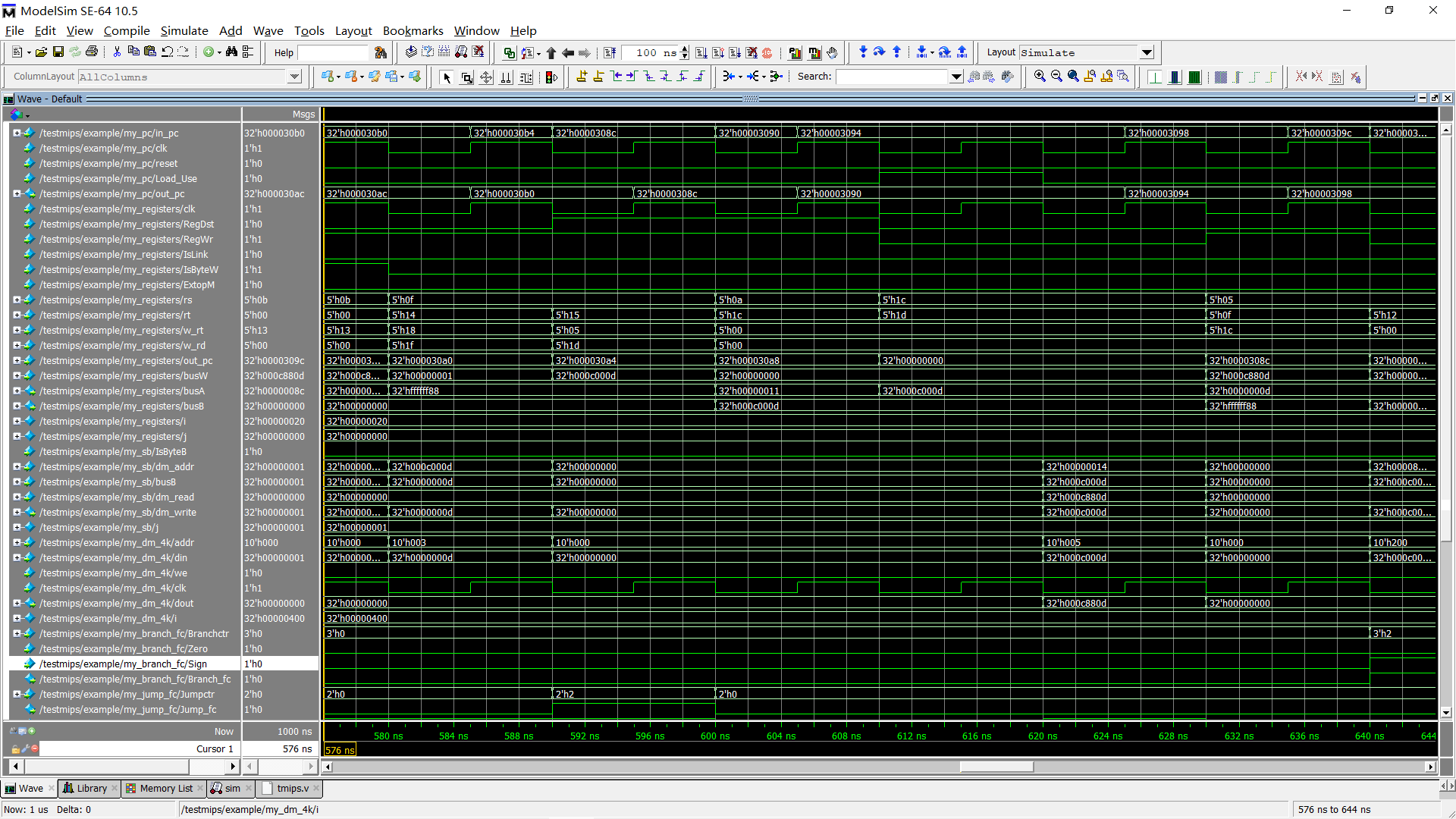


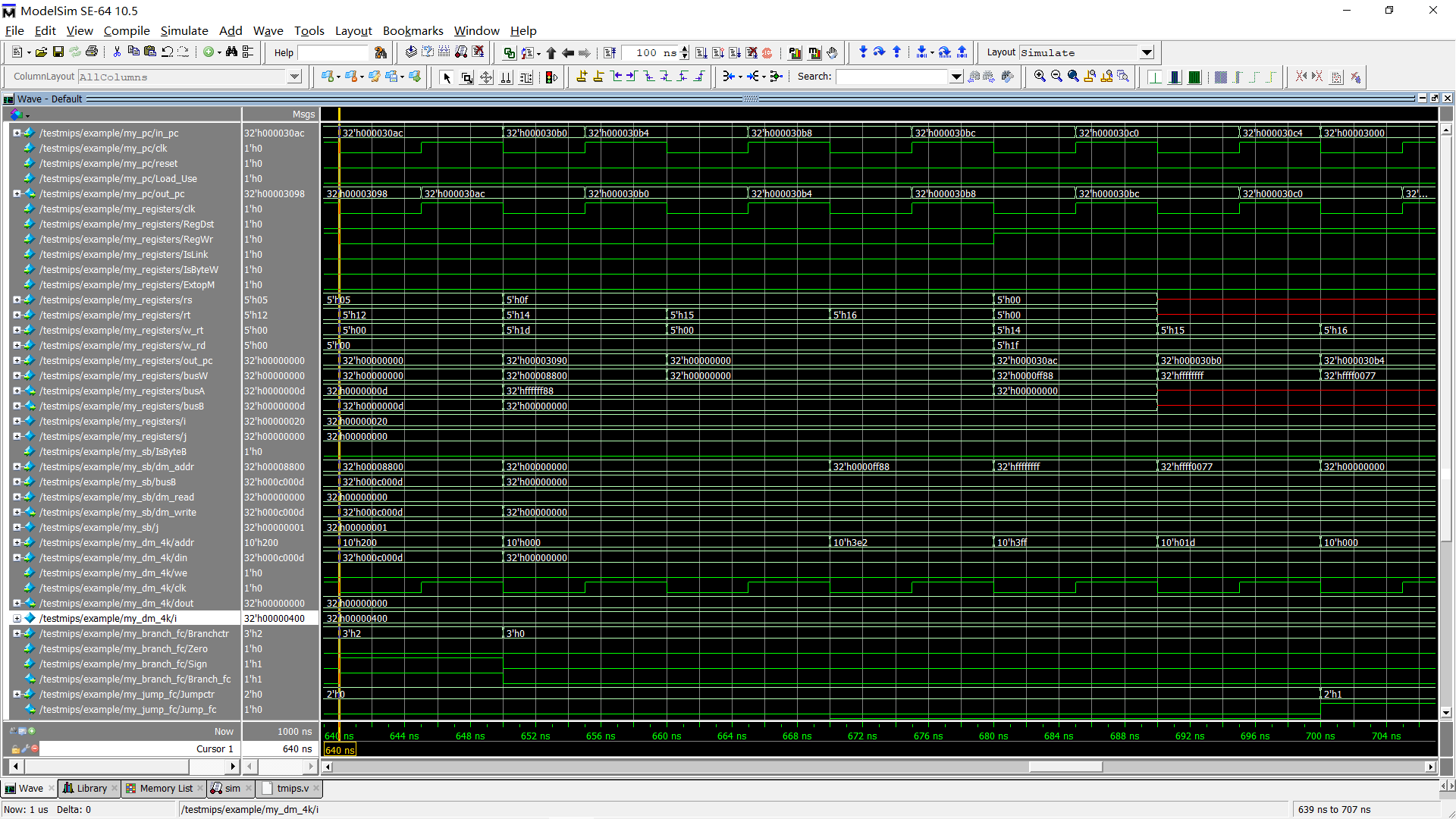












根据上面截图所附的这些模块的相关信息就可以充分验证自己设计的流水线CPU在验证指令测试集上的正确性。从如上的信号截图可以清楚地看到，五级流水线工作模式下PC的变化，在本课程设计中的五级流水线CPU，当遇到分支跳转和直接跳转这类改变流水线指令执行顺序的控制冒险时，会将流水线阻塞两个时钟周期，所以看到在跳转相关指令的PC继续顺序计算了两个时钟周期之后才跳转到预计的正确的PC地址，并且阻塞逻辑保证阻塞期间的段寄存器中保存的相关信号会被清空，不会出现将错误数据写入的情况。综合流水线执行的流程和逻辑，以及在应对相关冒险时的处理措施，再对比上面对应的在寄存器堆中写入的目的寄存器和写入的具体值，和所给的测试36条指令的测试程序预计的执行结果是完全一致的，并且在最后执行完了一轮测试程序之后PC也再一次跳转回了起始地址0x00003000，所设计的五级流水线CPU顺利且正确地通过测试。

红线的说明：在本流水线中所出现的所有红线都是因为没有初始化造成的，不过由于五级流水线结构的设计，一些信号值需要经过段寄存器的逐层传递才能被更新，以上述截图中寄存器堆的模块为例，相关的输入信号sim:/testmips/example/my\_registers/RegDst、sim:/testmips/example/my\_registers/RegWr、sim:/testmips/example/my\_registers/IsLink、sim:/testmips/example/my\_registers/IsByteW、sim:/testmips/example/my\_registers/ExtopM、sim:/testmips/example/my\_registers/w\_rt、sim:/testmips/example/my\_registers/w\_rd、sim:/testmips/example/my\_registers/out\_pc等信号，在顶层模块代码中也可以清楚看到，这些信号在经过Mem/Wr段寄存器才会被送到寄存器堆，故在第四个周期下降沿才会有对应的值送过去，所以未定态的红线占据了如图所示的周期数（在设计时时钟是从下降沿开始，PC更新是在时钟上升沿，段寄存器传递更新是在是在下降沿），其他的红线长短不一也是这个原因，都是因为没有初始化引起的，并且由于流水线逐级传递的结构特征所致的。并且可以看到，从第五个时钟周期开始，后面直到测试指令的第一轮循环结束（时钟刻度上越690ｎｓ的位置）都没有存在任何红线，后面进入新的一轮了，由于在简单预测的原因，执行到最后一条指令的时候，会继续向后读取下一条指令，可是指令存储器中已经没有指令了，所以取出了空指令，由于在代码中并没有对指令存储器进行初始化清零，所以取出的指令是未定态的，导致解码得到的值也是未定态的，所以出现了部分信号在第二轮开始前的几个时钟周期内红线的情况，之后即恢复正常了。对于某些特殊的信号是一开始一个时钟周期内为零而后又变成了红线是因为一开始在相关模块中加入了初始化为零的操作，但是由于在顶层模块的逻辑和段寄存器的传递，要传入这个端口的信号在当前时钟周期内还是未定态，故而导致出现了某些信号值一开始的一个时钟周期为绿线正常而后又变成红线。由于没有初始化和流水线五级流水结构逐级传递的特性，导致部分信号出现红线的情况基本已经全部清楚地讲解了，也可以看出，在进入第五个时钟周期之后，所有信号全部为绿线，不再存在红线的情况。

# 第三章 总结与展望

写到这里的时候，不禁看了一眼这篇报告已经达到的字数，不觉间已经达到了27000字，本人也是感慨万千，意识到自己在计算机组成原理这门课程中所做的课程设计也算是告一段落了，课程设计是在写完课程实验大作业的单周期CPU之后进行的，本以为在有了单周期CPU的实践基础之上，流水线CPU的设计应该会较为顺利，可是事实上自己还是遇到了不少的阻力，从一开始的没有任何经验、完全摸不着头脑、找不到方向如何开始，到开始写代码之后遇到一个又一个的毫无预料的bug，以及在处理流水线冒险时的为了考虑逻辑时长动辄几个小时连续坐在电脑前进行调试，再到最终完成36条基础MIPS指令的设计并且成功测试通过老师所给的测试指令程序，再到现在来到这里写课程设计报告的总结与展望，回顾整个过程，不能说是不艰辛的，但是相比过程中的阻力和困难给自己带来的压力和痛苦，通过自己夜以继日的努力成功克服了它们给自己带来的喜悦和满足会更让人幸福。

对于整个课程设计的五级流水线CPU，是在实验大作业中对单周期CPU的基础上进行扩展得到的，当然正如前面的内容所讲的，其中对于像sb这样的指令的设计逻辑进行修改与重做，使这条指令执行的逻辑更加清楚和分明。整个CPU设计中对于模块的组织是做到了模块化的基本需求，每个模块都有自己负责的部分和承担的责任，设计上是相互独立的，在布局上又是每个模块之间都是有着相互联系、密不可分的关系的，这使得整个CPU设计层次分明，逻辑清楚。在参考教材设计的同时，也保留了自我设计的独立性和灵活性，对于一些模块和处理融入了自己的猜想和设计，也尽可能地在不打破整体设计的统一性和完整性下去对一些逻辑进行优化和重组，这些相关的设计实现在整篇报告中的前面内容的正文部分都有着体现和描述。在最后的总结部分就不再赘述了。

整体上，我是设计出了一个具有五级流水线阶段，即取指、译码、执行、访存和写回五个阶段，支持实验指导书上要求的36条基础MIPS指令，能够处理绝大多数的RAW（写后读）的数据冒险和跳转类的指令引起的流水线执行顺序改变的控制冒险问题，当然结构冒险这样的硬件资源冲突问题在给定的指令存储器和数据存储器设计要求中已经得以解决了，对于分支跳转指令实现支持总是预测不跳转的简单分支预测方式，能够顺利执行36条基础MIPS指令和正确运行老师所给的指令测试程序的五级流水线CPU。

通过这次五级流水线CPU的设计，虽然限于某些客观条件的原因，对于一些扩展的指令要求并没有去进行扩展和实现，但是经历了整个设计的过程，一直到现在完善课程设计的报告到这里，自己的收获是非常非常多的，私认为，这是难以用文字来形容的。首先，自己对流水线CPU的工作方式和实现逻辑有了非常深刻的认识和理解，对计算机组成原理这门课程有了更加清楚的认识，对整个计算机体系结构有了具体的了解并且激发了自己去探索的浓厚兴趣与热情（自己本身就是对计算机系统这块有着非常浓厚的兴趣，这学期早在写计算机组成原理实验和课设的CPU之前，起始已经通过全国大学生计算机系统能力CSCC竞赛独立完成了竞赛平台要求的操作系统内核的设计，这个当时也是耗费了我们大量的精力和心血，并且可以说是远远超过CPU设计的），也极大地加强了自己的代码编写能力和项目组织管理能力，整个编写和调试的过程也很好地磨练了自我的意志，在与同学和舍友一起进行设计过程中的问题的交流和方案的探讨过程中也很好地增强了自己的团队协作能力和沟通与交流的能力，还有等等等等，总之，受益匪浅！

当然，我想也是非常有必要，在这里感谢自己的计算机组成原理课程的任课老师施慧彬老师！施老师在整个教学和实验过程中，一丝不苟，兢兢业业，努力将自己的学识传授给我们，勤奋认真地批改我们的作业，积极关心并且能够及时解决学生的问题，施老师在整个过程的辛勤付出，是我这次实验和课设能够顺利设计出来的关键因素！

最后，也希望自己能够好好借助这次设计流水线CPU的过程，将得到的收获和锻炼的能力内化为自己实实在在的实力，练就计算机能力相关的过硬本领。在日后的学习和探索的过程中，也能够像这次CPU设计一样，一步一步去克服自己遇到的困难，迎难而上，不惧险阻。同时，我也将以更高的标准要求自己，以更严格的规矩约束自己，努力向着自己的目标去追求，不断攀登高峰，成为计算机领域的精英，成为能够投身祖国建设大业的领尖人才！

# 参考文献

[1] 袁春风, 唐杰, 杨若瑜, 李俊等. 计算机组成与系统结构[M]. 第3版, 北京: 清华大学出版社, 2022.

# 附录

## 附录1 五级流水线CPU测试所用的指令程序

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令地址 | 汇编指令 | 结果描述 |  | 机器指令的机器码 |
| **16** 进制 | 二进制 |
| 00H | addiu $1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_0001 |
| 04H | sll $2, $1,#4 | [$2] = 0000\_0010H | 00011100 | 0000\_0000\_0000\_0001\_0001\_0001\_0000\_0000 |
| 08H | addu $3, $2,$1 | [$3] = 0000\_0011H | 00411821 | 0000\_0000\_0100\_0001\_0001\_1000\_0010\_0001 |
| 0CH | srl $4, $2,#2 | [$4] = 0000\_0004H | 00022082 | 0000\_0000\_0000\_0010\_0010\_0000\_1000\_0010 |
| 10H | slti $25,$4,#5 | [$25]=0000\_0001H | 28990005 | 0010\_1000\_1001\_1001\_0000\_0000\_0000\_0101 |
| 14H | bgez $25,#16 | 跳转到 54H | 07210010 | 0000\_0111\_0010\_0001\_0000\_0000\_0001\_0000 |
| 18H | subu $5, $3,$4 | [$5] = 0000\_000DH | 00642823 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0011 |
| 1CH | sw $5, #20($0) | Mem[0000\_0014H] =  0000\_000DH | AC050014 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0100 |
| 20H | nor $6, $5,$2 | [$6] = FFFF\_FFE2H | 00A23027 | 0000\_0000\_1010\_0010\_0011\_0000\_0010\_0111 |
| 24H | or $7, $6,$3 | [$7] = FFFF\_FFF3H | 00C33825 | 0000\_0000\_1100\_0011\_0011\_1000\_0010\_0101 |
| 28H | xor $8, $7,$6 | [$8] = 0000\_0011H | 00E64026 | 0000\_0000\_1110\_0110\_0100\_0000\_0010\_0110 |
| 2CH | sw $8, #28($0) | Mem[0000\_001CH] =  0000\_0011H | AC08001C | 1010\_1100\_0000\_1000\_0000\_0000\_0001\_1100 |
| 30H | beq $8, $3,#2 | 跳转到 38H | 11030002 | 0001\_0001\_0000\_0011\_0000\_0000\_0000\_0010 |
| 34H | slt $9, $6,$7 | 不执行 | 00C7482A | 0000\_0000\_1100\_0111\_0100\_1000\_0010\_1010 |
| 38H | addiu $1, $0,#8 | [$1] = 0000\_0008H | 24010008 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_1000 |
| 3CH | lw $10,#20($1) | [$10]=0000\_0011H | 8C2A0014 | 1000\_1100\_0010\_1010\_0000\_0000\_0001\_0100 |
| 40H | bne $10,$5,#4 | 跳转到 50H | 15450004 | 0001\_0101\_0100\_0101\_0000\_0000\_0000\_0100 |
| 44H | and $11,$2,$1 | 不执行 | 00415824 | 0000\_0000\_0100\_0001\_0101\_1000\_0010\_0100 |
| 48H | sw $11,#28($1) | 不执行 | AC2B001C | 1010\_1100\_0010\_1011\_0000\_0000\_0001\_1100 |
| 4CH | sw $4, #16($1) | 不执行 | AC240010 | 1010\_1100\_0010\_0100\_0000\_0000\_0001\_0000 |
| 50H | jal #25 | 跳转到 64H,  [$31] =0000\_0054H | 0C000019 | 0000\_1100\_0000\_0000\_0000\_0000\_0001\_1001 |
| 54H | lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011\_1100\_0000\_1100\_0000\_0000\_0000\_1100 |
| 58H | srav $26,$12,$2 | [$26] = 0000\_000CH | 004CD007 | 0000\_0000\_0100\_1100\_1101\_0000\_0000\_0111 |
| 5CH | sllv $27,$26,$1 | [$27]=0000\_0018H | 003AD804 | 0000\_0000\_0011\_1010\_1101\_1000\_0000\_0100 |
| 60H | jalr $27 | 跳转到 18H , | 0360F809 | 0000\_0011\_0110\_0000\_1111\_1000\_0000\_1001 |
|  |  | [$31]=0000\_0064H |  |  |
| 64H | sb $26,#5($3) | 取出$26的最低字节处的0x0C, 将0x0C存入MEM[0000\_0016H] 字节单元处。表示一个字时，  Mem[0000\_0014H]  000C\_000DH | A07A0005 | 1010\_0000\_0111\_1010\_0000\_0000\_0000\_0101 |
| 68H | sltu $13,$3,$3 | [$13]= 0000\_0000H | 0063682B | 0000\_0000\_0110\_0011\_0110\_1000\_0010\_1011 |
| 6CH | bgtz $13,#3 | 不跳转 | 1DA00003 | 0001\_1101\_1010\_0000\_0000\_0000\_0000\_0011 |
| 70H | sllv $14,$6,$4 | [$14] =FFFF\_FE20H | 00867004 | 0000\_0000\_1000\_0110\_0111\_0000\_0000\_0100 |
| 74H | sra $15,$14,#2 | [$15] =FFFF\_FF88H | 000E7883 | 0000\_0000\_0000\_1110\_0111\_1000\_1000\_0011 |
| 78H | srlv $16,$15,$1 | [$16] =00FF\_FFFFH | 002F8006 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0110 |
| 7CH | blez $16,#8 | 不跳转 | 1A000008 | 0001\_1010\_0000\_0000\_0000\_0000\_0000\_1000 |
| 80H | srav $16,$15,$1 | [$16] =FFFF\_FFFFH | 002F8007 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0111 |
| 84H | addiu $11,$0,#140 | [$11] = 0000\_008CH | 240B008C | 0010\_0100\_0000\_1011\_0000\_0000\_1000\_1100 |
| 88H | bltz $16, #6 | 跳转到 A0H | 06000006 | 0000\_0110\_0000\_0000\_0000\_0000\_0000\_0110 |
| 8CH | lw $28,#3($10) | [$28] = 第一次执行时赋值000C\_000DH，  /第二次执行时赋值  000C\_880DH  测试45条指令的汇编代码执行情况说明中的‘/’类似地表示多次依次执行的结果。 | 8D5C0003 | 1000\_1101\_0101\_1100\_0000\_0000\_0000\_0011 |
| 90H | bne $28,$29,#7 | 第一次执行时,不跳转  /第二次执行时跳转到指令地址ACH处执行 | 179D0007 | 0001\_0111\_1001\_1101\_0000\_0000\_0000\_0111 |
| 94H | sb $15,#8($5) | Mem[0000\_0014H] =  000C\_880DH | A0AF0008 | 1010\_0000\_1010\_1111\_0000\_0000\_0000\_1000 |
| 98H | lb $18,#8($5) | [$18] =FFFF\_FF88H | 80B20008 | 1000\_0000\_1011\_0010\_0000\_0000\_0000\_1000 |
| 9CH | lbu $19,#8($5) | [$19] = 0000\_0088H | 90B30008 | 1001\_0000\_1011\_0011\_0000\_0000\_0000\_1000 |
| A0H | sltiu $24,$15,#0xFFFF | [$24] = 0000\_0001H | 2DF8FFFF | 0010\_1101\_1111\_1000\_1111\_1111\_1111\_1111 |
| A4H | or $29,$12,$5 | [$29] = 000C000DH | 0185E825 | 0000\_0001\_1000\_0101\_1110\_1000\_0010\_0101 |
| A8H | jr $11 | 跳转到8CH | 01600008 | 0000\_0001\_0110\_0000\_0000\_0000\_0000\_1000 |
| ACH | andi $20,$15,#0xFFFF | [$20] = 0000\_FF88H | 31F4FFFF | 0011\_0001\_1111\_0100\_1111\_1111\_1111\_1111 |
| B0H | ori  $21,$15,#0xFFFF | [$21] =FFFF\_FFFFH | 35F5FFFF | 0011\_0101\_1111\_0101\_1111\_1111\_1111\_1111 |
| B4H | xori $22,$15,#0xFFFF | [$22] = FFFF\_0077H | 39F6FFFF | 0011\_1001\_1111\_0110\_1111\_1111\_1111\_1111 |
| B8H | j #00H | 跳转指令，到 00H | 08000000 | 0000\_1000\_0000\_0000\_0000\_0000\_0000\_0000 |

## 附录2 实现的36条MIPS指令解析表图



图附2 36条MIPS指令解析表

图附2是自己对照实验指导书在实验开始对36条MIPS指令实现的解析表图，其中解析了每条指令的类型和对应的字段信息，以及各自的功能和预期的计算方式，方便在代码设计中对于控制单元的编写和ALU控制信号的设计和生成。（具体浏览时可以将word文档放大更清楚查看具体内容和细节）