**武汉大学计算机学院**

**本科生实验报告**

**MIPS 流水线 CPU设计与实现**

专 业 名 称： 计算机科学与技术

课 程 名 称：计算机组成原理实验

指 导 教 师： 徐爱萍 教授

学 生 学 号： 2017301500335

学 生 姓 名： 张永康

二〇一九年五月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 日期：

摘 要

计算机组成原理实验的实验目的是熟悉现代处理器的基本工作原理、掌握单周期和流水线处理器的设计方法。

实验设计主要遵循预期设计的顶层体系结构图，分别对每个流水级进行详细实现，在CPU设计中还要考虑数据相关性、阻塞问题, 本次实验采用预测分支总不发生、ID级判定分支、ID级和EX级加入全旁路，从而解决不同的数据相关性问题和控制冒险问题。

实验内容主要包括：

1. 单周期MIPS处理器的设计与实现
2. 流水线MIPS处理器的设计与实现
3. 针对分支语句引发的控制冒险加入相应阻塞
4. 针对RAW(先写后读)类型数据相关加入全旁路
5. 基于ModelSim的功能仿真
6. 下载至FPGA开发板进行实际测试

最终使用 Mentor公司的仿真软件 Modelsim对流水线MIPS模型各模块进行功能仿真,将仿真的结果与MARS软件运行得到的预期结果进行对比,结果完全相同；之后在Xilinx FPGA开发板上成功执行了数组排序程序，与MARS软件运行得到的预期结果相同。以上表明模型设计正确，测试结果符合设计要求。

**关键词：**MIPS；处理器；流水线

目 录

[1 实验目的和意义 1](#_Toc24584)

[1.1 实验目的 1](#_Toc25527)

[1.2 实验意义 1](#_Toc26965)

[2 实验设计 2](#_Toc3372)

[2.1 概述 2](#_Toc3970)

[2.2 实验环境 2](#_Toc23330)

[2.2.1 Verilog HDL简介 2](#_Toc27590)

[2.2.2 ModelSim简介 3](#_Toc23807)

[2.2.3 MARS 简介 3](#_Toc26326)

[2.3 硬件设计 3](#_Toc7164)

[2.3.1 CPU总体结构 3](#_Toc756)

[2.3.2 IF/ID寄存器（IF\_ID.v） 4](#_Toc32640)

[2.3.3 ID/EX寄存器（ID/EX.v） 6](#_Toc26481)

[2.3.4 EX/MEM寄存器（EX\_MEM.v） 7](#_Toc272)

[2.3.5 MEM/WB寄存器（MEM\_WB.v） 8](#_Toc16720)

[2.3.6 ID级模块（ID.v） 9](#_Toc5358)

[2.3.7 EX级模块（EX.v） 11](#_Toc27112)

[2.3.8 MEM级模块（MEM.v） 12](#_Toc5291)

[2.3.9程序计数器（PcUnit.v） 13](#_Toc3005)

[2.3.10寄存器堆（RegFile.v） 13](#_Toc9004)

[2.3.11算术逻辑运算单元（Alu.v） 14](#_Toc32479)

[2.3.12扩展单元（Extender.v） 15](#_Toc32245)

[2.3.13指令存储器（IMem.v） 15](#_Toc8598)

[2.3.14控制器（Ctrl.v） 15](#_Toc3513)

[2.3.15模型机（Pipeline\_Mips.v） 16](#_Toc22313)

[2.4 软件设计 17](#_Toc22656)

[2.4.1 指令宏定义（instruction\_def.v） 17](#_Toc26099)

[2.4.2 控制逻辑宏定义（ctrl\_encode\_def.v） 18](#_Toc18914)

[2.4.3 总线位宽宏定义（bus\_def.v） 20](#_Toc30513)

[2.4.4程序计数器（PcUnit.v） 20](#_Toc10786)

[2.4.5寄存器堆（RegFile.v） 21](#_Toc20008)

[2.4.6算术逻辑运算单元（Alu.v） 23](#_Toc9014)

[2.4.7扩展单元（Extender.v） 24](#_Toc14744)

[2.4.8指令存储器（IMem.v） 25](#_Toc27392)

[2.4.9控制器（Ctrl.v） 25](#_Toc6543)

[2.4.10 ID级（ID.v） 34](#_Toc9275)

[2.4.11 EX级（EX.v） 40](#_Toc25887)

[2.4.12 MEM级（MEM.v） 44](#_Toc21484)

[2.4.13 IF/ID寄存器（IF\_ID.v） 45](#_Toc13960)

[2.4.14 ID/EX寄存器（ID\_EX.v） 46](#_Toc30559)

[2.4.15 EX/MEM寄存器（EX\_MEM.v） 51](#_Toc22984)

[2.4.16 MEM/WB寄存器（MEM\_WB.v） 53](#_Toc26353)

[2.4.17 模型机（Pipeline\_Mips.v） 55](#_Toc30909)

[2.5 设计结果分析 64](#_Toc21553)

[2.5.1测试文件 64](#_Toc6456)

[2.5.2 测试结果分析 65](#_Toc26155)

[2.5.2.1 lui $1, 0x1000 指令 65](#_Toc1195)

[2.5.2.2 ori $1, $1, 0x00001008 指令 65](#_Toc19665)

[2.5.2.3 sw $1, 0($0)指令 66](#_Toc11872)

[2.5.2.4 addi $6,$6,-1 指令 66](#_Toc11484)

[2.5.2.5 lw $2,0($1) 指令 67](#_Toc5214)

[2.5.2.6 slt $4,$3,$2 指令 67](#_Toc4005)

[2.5.2.7 beq $4, $5, swap指令 67](#_Toc14428)

[2.5.2.8 EX级旁路 68](#_Toc7869)

[2.5.2.9 ID级旁路 70](#_Toc31099)

[2.5.2.10 EX级阻塞控制 70](#_Toc29006)

[2.5.2.11 ID级阻塞控制 70](#_Toc7323)

[2.5.2.12 ID级、EX级同时阻塞的控制 72](#_Toc8594)

[2.5.3 FPGA开发板测试 73](#_Toc132)

[总 结 78](#_Toc21211)

[参考文献 79](#_Toc9281)

[教师评语评分 80](#_Toc21761)

# 1 实验目的和意义

## 1.1 实验目的

融会贯通计算机组成与设计课程所教授的知识,通过对知识的综合应用,加深对CPU系统各模块的工作原理及相互联系的认识。

学习采用EDA( Electronic Design Automation)技术设计MIPS单周期CPU多周期/流水线CPU的技术与方法

培养科学研究的独立工作能力,取得CPU设计与仿真的实践和经验。

## 1.2 实验意义

(1)通过对指令在流水线中的执行过程的分析，掌握MIPS指令集中各类指令的特点、功能和实现细节。

(2)通过对流水线的设计以及各模块的实现,并用仿真软件对所设计的流水线各模块进行功能仿真。更加深刻的握的MIPS32指令架构的相关概念。

(3)针对流水线执行过程中的各类问题，学习旁路、分支预测、阻塞等技术的实现细节。

(4)学习Modelsim仿真软件、Vivado FPGA开发工具、MARS(MIPS汇编仿真软件)的使用。

# 2 实验设计

## 2.1 概述

本设计采用五级流水线结构,也就是将指令的执行过程分为五个阶段进行,如图2.1所示,这五个阶段分别是取值阶段(IF)、译码阶段(ID)、执行阶段(EX)、访存阶段(MEM)和回写阶段(WB)。

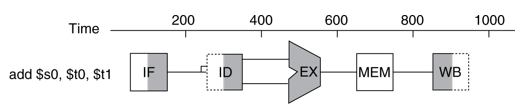


图2.1 五级流水线示意图

指令执行时各阶段的主要工作如下

取指阶段(IF):取指阶段要完成的工作就是从指令存储器读出PC寄存器指定的当前地址中的指令信息,并对下一次要访问的指令地址信息进行预判。

译码阶段(ID):对根据取指阶段取出的指令的32位二进制编码方式进行翻译指令,此阶段会得到指令的操作类型,以及要读取的寄存器数据。

执行阶段(EX):按照译码阶段给出来的操作数、运算类型进行算数运算,并给出运算结。如果是 Load/Store指令,那么还会计算出 Load/Store的目标地址。

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

回写阶段(WB):将运算结果保存到目标寄存器。

## 2.2 实验环境

### 2.2.1 Verilog HDL简介

Verilog HDL是一种硬件描述语言，用于从算法级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可以介于简单的门和完整的电子数字系统之间。数字系统能够按层次描述，并可在相同描述中显式地进行时序建模。

Verilog HDL 语言具有下述描述能力：设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监控和设计验证方面的时延和波形产生机制。所有这些都使用同一种建模语言。此外，Verilog HDL语言提供了编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

Verilog HDL语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用Verilog仿真器进行验证。语言从C编程语言中继承了多种操作符和结构。Verilog HDL提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog HDL语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然，完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

### 2.2.2 ModelSim简介

Mentor公司的ModelSim是业界最优秀的HDL语言仿真软件，它能提供友好的仿真环境，是业界唯一的单内核支持VHDL和Verilog混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk技术、和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护IP核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是FPGA/ASIC设计的首选仿真软件。

### 2.2.3 MARS 简介

MARS是一个轻量级的交互式开发环境（ide），用于mips汇编语言编程，旨在与Patterson和Hennessy编写的教材《计算机组成和设计：硬件/软件接口》一书相结合，在教育领域使用。Mars已经在Softpedia实验室使用多个行业领先的安全解决方案进行了测试，发现完全没有广告软件/间谍软件组件。Mars4.3是100%免费的，它不包含任何形式的恶意软件，包括间谍软件、病毒、特洛伊木马和后门。

## 2.3 硬件设计

### **2.3.1 CPU总体结构**

在本设计中要实现的五级流水线包含了取指、译码、执行、访存和回写五个阶段。

取指阶段要完成的工作就是从指令存储器读出PC寄存器指定的当前地址中的指令信息,并根据控制信号对下一次要访问的指令地址信息进行预判,IF/ID寄存器模块是存放指令信息的流水线寄存器。

译码阶段要对取出的指令信息进行译码分析,从寄存器堆中取出源操作数并给出转移类指令的跳转地址,因此译码阶段设计了用于指令译码和转移判断的ID模块以及用于存放数据的 Regfile 模块、 ID/EX寄存器模块,ID/EX寄存器用于存放译码阶段得到的各类信息。

执行阶段会对译码得到的各类数据进行相关运算,此阶段设计有EX模块、DⅣ模块和 EX/MEM寄存器模块,需要执行的大部分运算都会在EX模块中的ALU运算单元内完成,执行完成后的各类数据会存放在流水线寄存器 EX/MEM中。

访存阶段会对存储器访问指令和异常情况进行处理,这些指令的执行会发生在MEM模块中,RAM模块用于存放所需的数据,访存阶段完成后结果会存放在 MEM/WB寄存器中。

回写阶段会将当前指令访存取数，或经执行阶段运算得到的结果写回寄存器堆Regfile。

CPU总体结构如图2.2所示，其中包括程序计数器（PCUnit)、指令存储器（IMem）、寄存器组（RegFile）、运算器（ALU）、数据扩展单元（EXT）、控制器（Ctrl），五个流水线级(IF、ID、EX、MEM、WB)，以及五个流水线级之间的寄存器(IF/ID、ID/EX、EX/MEM、MEM/WB)。

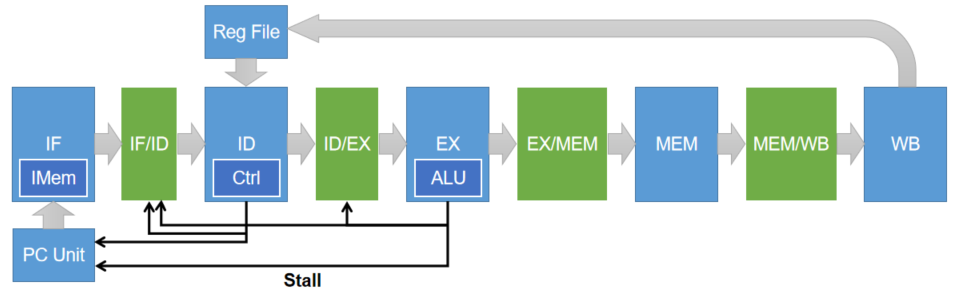


图2.2 处理器模块构成图

数据通路图如2.3所示，RTL综合图如图2.4所示。

### 2.3.2 IF/ID寄存器（IF\_ID.v）

**（1）功能描述**

IF级与ID级之间暂存数据的寄存器。

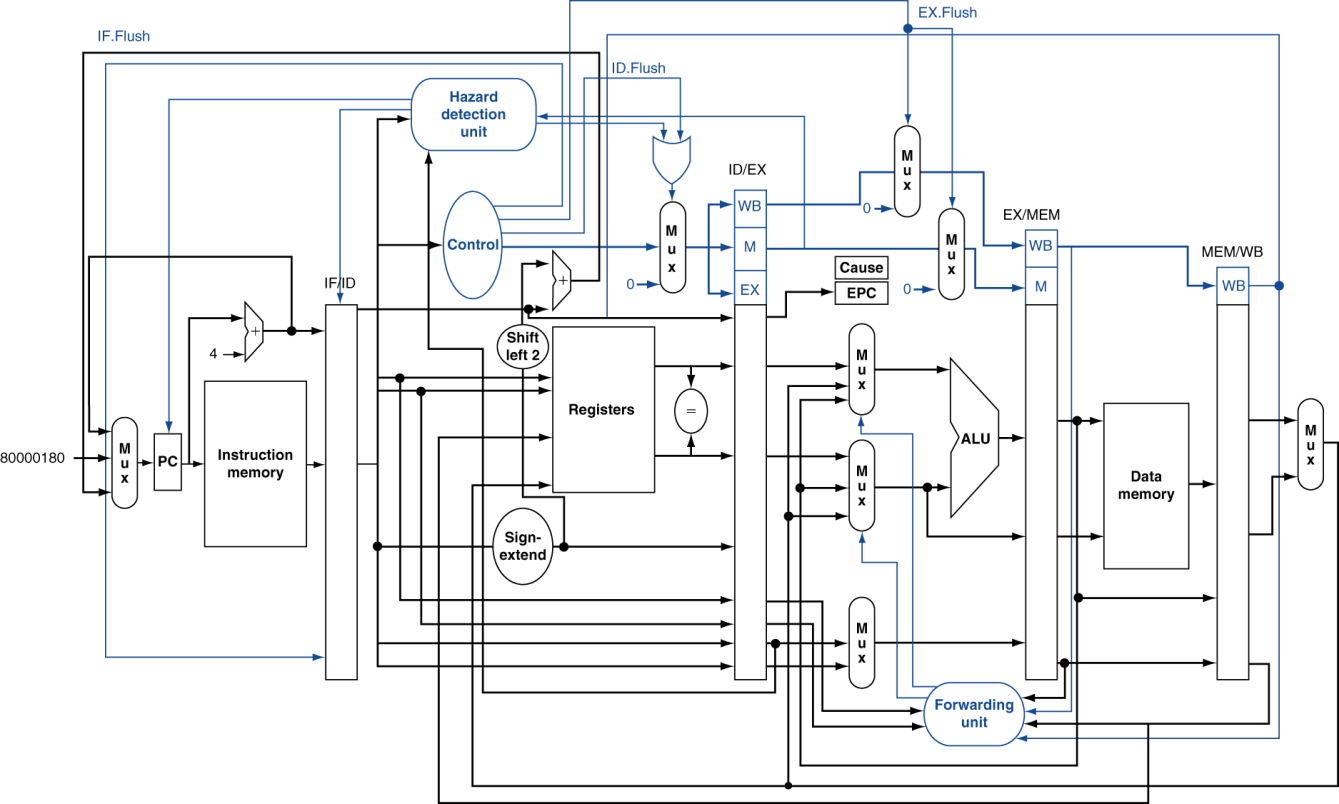


图2.3 数据通路图

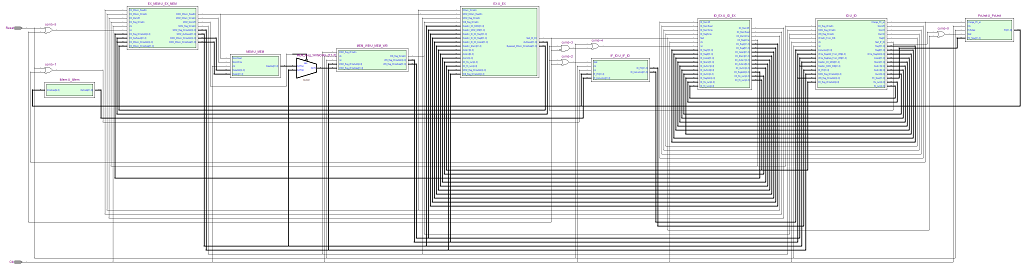


图2.4 RTL综合图

**（2）模块接口**

IF/ID寄存器接口定义如表2.1所示。

表2.1 IF/ID寄存器接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 重置信号 |
| Stall | input | ID级流水线暂停信号 |
| [`Word\_Bus] IF\_PC | input | IF级的PC值 |
| [`Word\_Bus]IF\_Instruction | input | IF级从指令存储器取出的指令 |
| [`Word\_Bus] ID\_PC | output | ID级的PC值 |
| [`Word\_Bus] ID\_Instruction | output | ID级的指令 |

### 2.3.3 ID/EX寄存器（ID/EX.v）

**（1）功能描述**

ID级与EX级之间暂存数据的寄存器。

**（2）模块接口**

ID/EX寄存器接口定义如表2.2所示。

表2.2 ID/EX寄存器接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 重置信号 |
| Stall | input | 暂停信号 |
| [`Word\_Bus] ID\_Reg1 | input | ID级取出的寄存器1的值 |
| [`Word\_Bus] ID\_Reg2 | input | ID级取出的寄存器2的值 |
| [`Word\_Bus] ID\_Instant | input | ID级取出的经扩展得到的立即数的值 |
| [`Word\_Bus]ID\_Shamt | input | 来自ID/EX的Shamt偏移量 |
| [`AluSrc\_Bus] ID\_AluSrc1 | input | ALU源操作数1控制信号 |
| [`AluSrc\_Bus] ID\_AluSrc2 | input | ALU源操作数2控制信号 |
| [4:0]ID\_AluCtrl | input | 来自ID/EX的ALU控制信号 |
| ID\_MemWrite | input | 来自ID级的存储器写信号 |
| ID\_MemRead | input | 来自ID级的存储器读信号 |
| ID\_RegWrite | input | 来自ID级的寄存器堆写回信号 |
| ID\_Mem2R | input | 来自ID级的Mem to Reg信号 |
| [`Reg\_Addr\_Bus] ID\_RegAddr | input | 来自ID级的被写回的寄存器号 |
| [`Reg\_Addr\_Bus] ID\_Rs\_num | input | 来自ID级的第一个被读出的寄存器号 |
| [`Reg\_Addr\_Bus] ID\_Rt\_num | input | 来自ID级的第二个被读出的寄存器号 |
| [`Word\_Bus] EX\_Reg1 | output | 送入EX级的寄存器1的值 |
| [`Word\_Bus] EX\_Reg2 | output | 送入EX级的寄存器2的值 |
| [`Word\_Bus] EX\_Instant | output | 送入EX级的经扩展得到的立即数的值 |
| [`Word\_Bus] EX\_Shamt | output | 送入EX级的Shamt偏移量 |
| [`AluSrc\_Bus] EX\_AluSrc1 | output | 送入EX级的ALU源操作数1控制信号 |
| [`AluSrc\_Bus] EX\_AluSrc2 | output | 送入EX级的ALU源操作数2控制信号 |
| [4:0] EX\_AluCtrl | output | 送入EX级的ALU控制信号 |
| EX\_MemWrite | output | 送入EX级的存储器写信号 |
| EX\_MemRead | output | 送入EX级的存储器读信号 |
| EX\_RegWrite | output | 送入EX级的写回寄存器堆信号 |
| EX\_Mem2R | output | 送入EX级的Mem to Reg信号 |
| [`Reg\_Addr\_Bus] EX\_RegAddr | output | 送入EX级的要被写回的寄存器号 |
| [`Reg\_Addr\_Bus] EX\_Rs\_num | output | 送入EX级的第一个被读出寄存器号 |
| [`Reg\_Addr\_Bus] EX\_Rt\_num | output | 送入EX级的第二个被读出寄存器号 |

### 2.3.4 EX/MEM寄存器（EX\_MEM.v）

**（1）功能描述**

EX级与MEM级之间暂存数据的寄存器。

**（2）模块接口**

EX/MEM寄存器接口定义如表2.3所示。

表2.3 EX/MEM寄存器接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 重置信号 |
| [`Reg\_Addr\_Bus] EX\_Reg\_WriteAddr | input | EX级指令要写入的寄存器的地址 |
| EX\_Reg\_WriteEn | input | =1 表示EX级指令要写寄存器 |
| [`Word\_Bus] EX\_AluResult | input | EX级指令ALU的结果 |
| EX\_Mem2R | input | =1表示EX级指令选择将存储器读出的数据送入寄存器 |
| [`DMem\_Addr\_Bus] EX\_DMem\_WriteAddr | input | EX级指令要写入的存储器单元的地址 |
| EX\_DMem\_WriteEn | input | EX级指令存储器写入使能信号 |
| EX\_DMem\_ReadEn | input | EX级指令存储器读出使能信号 |
| [`Word\_Bus] EX\_DMem\_WriteData | input | EX级指令要写入存储器的数据 |
| [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr | output | MEM级指令要写入寄存器的地址 |
| reg MEM\_Reg\_WriteEn | output | =1 表示MEM级指令要写寄存器 |
| [`Word\_Bus] MEM\_AluResult | output | MEM级指令ALU结果 |
| reg MEM\_Mem2R | output | =1 表示MEM级指令选择将存储器读出的数据送入寄存器 |
| [`DMem\_Addr\_Bus] MEM\_DMem\_WriteAddr | output | MEM级指令写入存储器单元的地址 |
| reg MEM\_DMem\_WriteEn | output | MEM级指令存储器写入使能信号 |
| reg MEM\_DMem\_ReadEn | output | MEM级指令存储器读出使能信号 |
| [`Word\_Bus] MEM\_DMem\_WriteData | output | MEM级指令要写入存储器的数据 |

### 2.3.5 MEM/WB寄存器（MEM\_WB.v）

**（1）功能描述**

MEM级与WB级之间暂存数据的寄存器。

**（2）模块接口**

MEM/WB寄存器接口定义如表2.4所示。

表2.4 MEM/WB寄存器接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 重置信号 |
| [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr | input | MEM级指令写回寄存器的地址 |
| MEM\_Reg\_WriteEn | input | MEM级指令写回寄存器使能信号 |
| [`Word\_Bus] MEM\_Reg\_WriteData | input | MEM级指令写回寄存器的数据 |
| [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr | output | WB级指令写回寄存器的地址 |
| reg WB\_Reg\_WriteEn | output | WB级指令写回寄存器使能信号 |
| [`Word\_Bus] WB\_Reg\_WriteData | output | WB级指令写回寄存器的数据 |

### 2.3.6 ID级模块（ID.v）

**（1）功能描述**

流水线ID级模块。

**（2）模块接口**

ID级模块接口定义如表2.5所示。

表2.5 ID级模块接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 重置信号 |
| [`Word\_Bus] Instruction | input | 32位指令 |
| [`Reg\_Addr\_Bus] Write\_RegAddr\_From\_WB | input | 来自WB级指令的写回寄存器的地址 |
| [`Word\_Bus] Write\_RegData\_From\_WB | input | 来自WB级指令的要写回寄存器的数据 |
| WriteEn\_From\_WB | input | 来自WB级指令的写回寄存器使能信号 |
| [`Word\_Bus] Reg1 | output | ID级指令从寄存器堆读出的第一个寄存器的数据 |
| [`Word\_Bus] Reg2 | output | ID级指令从寄存器堆读出的第二个寄存器的数据 |
| [`Reg\_Addr\_Bus]  Write\_RegAddr | output | ID级指令要写回的寄存器的地址 |
| [`Word\_Bus] Instant | output | ID级指令的立即数 |
| [`Word\_Bus] Shamt | output | ID级指令的位移量(用于SLL |
| jump | output | 跳转使能信号 |
| RegDst | output | 写入寄存器选择(Rt或Rd) |
| Branch | output | 分支使能信号 |
| MemR | output | 内存读信号 |
| Mem2R | output | 选择将从内存读出的数据作为写回寄存器的数据 |
| MemW | output | 内存写信号 |
| RegW | output | 写回寄存器信号 |
| [`AluSrc\_Bus] AluSrc1 | output | ALU的第一个输入的选择信号 |
| [`AluSrc\_Bus] AluSrc2 | output | ALU的第二个输入的选择信号 |
| [4:0] Aluctrl | output | ALU的控制信号 |
| [`Word\_Bus] DataSrc\_EX\_MEM | input | 来自EX\_MEM的旁路数据 |
| [`Word\_Bus] DataSrc\_MEM\_WB | input | 来自MEM\_WB的旁路数据 |
| [`Word\_Bus] ID\_PC | input | ID级指令所处的PC值 |
| MEM\_Reg\_WriteEn | input | MEM级指令的寄存器写回使能信号 |
| [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr | input | MEM级指令的寄存器写回地址 |
| WB\_Reg\_WriteEn | input | WB级指令的寄存器写回使能信号 |
| [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr | input | WB级指令的寄存器写回地址 |
| [`Word\_Bus] PC\_New | output | 分支或跳转发生后的新的PC值 |
| Change\_PC\_en | output | 用PC\_New改变当前PC值的使能信号 |

### 2.3.7 EX级模块（EX.v）

**（1）功能描述**

流水线EX级模块。

**（2）模块接口**

EX级模块接口定义如表2.6所示。

表2.6 EX级模块接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| [`Word\_Bus] DataSrc\_EX\_MEM | input | 来自EX\_MEM的旁路数据 |
| [`Word\_Bus] DataSrc\_MEM\_WB | input | 来自MEM\_WB的旁路数据 |
| [`Word\_Bus] DataSrc\_ID\_EX\_Reg1 | input | 来自ID/EX第1个寄存器的数据 |
| [`Word\_Bus] DataSrc\_ID\_EX\_Reg2 | input | 来自ID/EX第2个寄存器的数据 |
| [`Word\_Bus] DataSrc\_ID\_EX\_Instant | input | 来自ID/EX的立即数 |
| [`Word\_Bus] DataSrc\_Shamt | input | 来自ID/EX的5位位移量Shamt |
| [`AluSrc\_Bus] AluSrc1 | input | ID级判定的ALU第1个数据的来源 |
| [`AluSrc\_Bus] AluSrc2 | input | ID级判定的ALU第2个数据的来源 |
| [4:0]AluCtrl | input | ALU控制信号，决定ALU执行什么类型的运算 |
| [`Reg\_Addr\_Bus] EX\_Rs\_num | input | EX级的指令的Rs寄存器号 |
| [`Reg\_Addr\_Bus] EX\_Rt\_num | input | EX级的指令的Rt寄存器号 |
| MEM\_Reg\_WriteEn | input | MEM级的指令的寄存器写回信号 |
| [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr | input | MEM级的指令的寄存器写回地址 |
| WB\_Reg\_WriteEn | input | WB级的指令的寄存器写回信号 |
| [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr | input | WB级的指令的寄存器写回地址 |
| [`Word\_Bus] AluResult | output | ALU运算结果 |
| Zero | output | Zero=1表示ALU的运算结果为全零 |
| DMem\_WriteEn | input | EX级的指令的存储器写入信号 |
| [`Word\_Bus] Bypassed\_DMem\_WriteData | output | 将要送给MEM级的被旁路过的送入存储器的数据 |
| MEM\_DMem\_ReadEn | input | MEM级存储器读信号 |
| Stall\_ID\_EX | output | ID/EX寄存器暂停的信号 |

### 2.3.8 MEM级模块（MEM.v）

（1）功能描述

流水线MEM级模块。

（2）模块接口

MEM级模块接口定义如表2.7所示。

表2.7 MEM级模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| MemWrite | input | 存储器写信号 |
| MemRead | input | 存储器读信号 |
| clk | input | 时钟信号 |
| [4:0] DataAddr | input | 地址 |
| [31:0] DataIn | input | 要写入存储器的数据 |
| [31:0] DataOut | output | 从存储器取出的数据 |

### 2.3.9程序计数器（PcUnit.v）

（1）功能描述

记录当前的PC值，当ID级判定为分支或跳转发生时，更新为新的PC值。

（2）模块接口

程序计数器模块接口定义如表2.8所示。

表2.8 程序计数器模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| [`Word\_Bus] PC | output | 当前PC值 |
| IF\_ID\_Flush | output | IF/ID寄存器清空信号 |
| PcReSet | input | PC值重置信号 |
| Change\_PC\_en | input | PC值置为PC\_New信号 |
| Stall | input | PC值计数暂停信号 |
| Clk | input | 时钟信号 |
| [`Word\_Bus] PC\_New | input | PC值的新值PC\_New |

### 2.3.10寄存器堆（RegFile.v）

（1）功能描述

保存32个32位寄存器的值，支持双端口读、单端口写。

（2）模块接口

寄存器堆模块接口定义如表2.9所示。

表2.9 寄存器堆模块接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| clk | input | 时钟信号 |
| rst | input | 复位信号 |
| WriteEn | input | 写使能信号 |
| [4:0] WriteAddr | input | 被写入寄存器的地址 |
| [31:0] WriteData | input | 要写入寄存器的数据 |
| [4:0] ReadAddr1 | input | 被读出的第1个寄存器地址 |
| [4:0] ReadAddr2 | input | 被读出的第2个寄存器地址 |
| [31:0] DataOut1 | output | 被读出的第1个寄存器数据 |
| [31:0] DataOut2 | output | 被读出的第2个寄存器数据 |

### 2.3.11算术逻辑运算单元（Alu.v）

（1）功能描述

ALU主要功能是完成对输入数据的进行加法、减法、与、或、左移、右移等运算。

（2）模块接口

ALU模块接口定义如表2.10所示。

表2.10 ALU模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| [`Word\_Bus] DataIn1 | input | ALU的第一个输入 |
| [`Word\_Bus] DataIn2 | input | ALU的第二个输入 |
| [4:0] AluCtrl | input | ALU的控制信号 |
| [`Word\_Bus] AluResult | output | ALU运算结果 |
| Zero | output | Zero=1表明ALU运算结果全0 |

### 2.3.12扩展单元（Extender.v）

（1）功能描述

主要功能是将原32位指令中的低16位的常数数据扩展为32位立即数数据。

（2）模块接口

扩展单元模块接口定义如表2.11所示。

表2.11 扩展单元模块接口定义

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| [15:0] Imm16 | input | 来自指令低16位的立即数 |
| [1:0] EXTOp | input | 扩展器控制信号 |
| [31:0] Imm32 | output | 扩展得到的32位立即数 |

### 2.3.13指令存储器（IMem.v）

（1）功能描述

存储32位指令。根据输入地址，将指令取出送入IF级。

（2 ）模块接口

指令存储器模块接口定义如表2.12所示。

表2.12 指令存储器模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| [`IMem\_Addr\_Bus] ImAdress | input | 当前指令的地址 |
| [`Word\_Bus] OpCode | output | 输出的32位指令 |

### 2.3.14控制器（Ctrl.v）

（1）功能描述

根据输入的32位指令，译码获得各类控制信号的值。

（2 ）模块接口

控制器模块接口定义如表2.13所示。

表2.13 控制器模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| [5:0]OpCode | input | 指令的6位操作码 |
| [5:0]funct | input | 指令的6位功能码 |
| jump | output | 跳转使能信号 |
| RegDst | output | 写入寄存器选择(Rt或Rd) |
| Branch | output | 分支使能信号 |
| MemR | output | 内存读信号 |
| Mem2R | output | 选择将从内存读出的数据作为写回寄存器的数据 |
| MemW | output | 内存写信号 |
| RegW | output | 写回寄存器信号 |
| [`AluSrc\_Bus] AluSrc1 | output | ALU的第一个输入的选择信号 |
| [`AluSrc\_Bus] AluSrc2 | output | ALU的第二个输入的选择信号 |
| [1:0] ExtOp | output | ID级的扩展器的控制信号 |
| [4:0] Aluctrl | output | ALU的控制信号 |

### 2.3.15模型机（Pipeline\_Mips.v）

（1）功能描述

顶层代码

（2 ）模块接口

表2.14是下板子时所需的接口，仿真时不需要。

表2.14 模型机模块接口定义

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Clk\_init | Input | 输入的原始时钟信号 |
| rstn | input | 输入的全局复位信号 |
| [15:0] sw\_i | Input | 15个开关，用于选择输出和控制要显示的存储器单元地址 |
| disp\_seg\_o | output | 数码管八选一信号 |
| disp\_an\_o | output | 单个数码管的控制信号 |

## 2.4 软件设计

### 2.4.1 指令宏定义（instruction\_def.v）

// OP

`define INSTR\_RTYPE\_OP 6'b000000

`define INSTR\_LB\_OP 6'b100000

`define INSTR\_LH\_OP 6'b100001

`define INSTR\_LBU\_OP 6'b100100

`define INSTR\_LHU\_OP 6'b100101

`define INSTR\_LW\_OP 6'b100011

`define INSTR\_SB\_OP 6'b101000

`define INSTR\_SH\_OP 6'b101001

`define INSTR\_SW\_OP 6'b101011

`define INSTR\_ADDI\_OP 6'b001000

`define INSTR\_ADDIU\_OP 6'b001001

`define INSTR\_ANDI\_OP 6'b001100

`define INSTR\_ORI\_OP 6'b001101

`define INSTR\_XORI\_OP 6'b001110

`define INSTR\_LUI\_OP 6'b001111

`define INSTR\_SLTI\_OP 6'b001010

`define INSTR\_SLTIU\_OP 6'b001011

`define INSTR\_BEQ\_OP 6'b000100

`define INSTR\_BNE\_OP 6'b000101

`define INSTR\_BGEZ\_OP 6'b000001

`define INSTR\_BGTZ\_OP 6'b000111

`define INSTR\_BLEZ\_OP 6'b000110

`define INSTR\_BLTZ\_OP 6'b000001

`define INSTR\_J\_OP 6'b000010

`define INSTR\_JAL\_OP 6'b000011

// Funct

`define INSTR\_ADD\_FUNCT 6'b100000

`define INSTR\_ADDU\_FUNCT 6'b100001

`define INSTR\_SUB\_FUNCT 6'b100010

`define INSTR\_SUBU\_FUNCT 6'b100011

`define INSTR\_AND\_FUNCT 6'b100100

`define INSTR\_NOR\_FUNCT 6'b100111

`define INSTR\_OR\_FUNCT 6'b100101

`define INSTR\_XOR\_FUNCT 6'b100110

`define INSTR\_SLT\_FUNCT 6'b101010

`define INSTR\_SLTU\_FUNCT 6'b101011

`define INSTR\_SLL\_FUNCT 6'b000000

`define INSTR\_SRL\_FUNCT 6'b000010

`define INSTR\_SRA\_FUNCT 6'b000011

`define INSTR\_SLLV\_FUNCT 6'b000100

`define INSTR\_SRLV\_FUNCT 6'b000110

`define INSTR\_SRAV\_FUNCT 6'b000111

`define INSTR\_JR\_FUNCT 6'b001000

`define INSTR\_JALR\_FUNCT 6'b001001

`define INSTR\_BGEZ\_RT 5'b00001

`define INSTR\_BLTZ\_RT 5'b00000

### 2.4.2 控制逻辑宏定义（ctrl\_encode\_def.v）

// NPC control signal

`define NPC\_PLUS4 2'b00

`define NPC\_BRANCH 2'b01

`define NPC\_JUMP 2'b10

// EXT control signal

`define EXT\_ZERO 2'b00

`define EXT\_SIGNED 2'b01

`define EXT\_HIGHPOS 2'b10

// ALU control signal

`define ALUOp\_NOP 5'b00000

`define ALUOp\_ADDU 5'b00001

`define ALUOp\_ADD 5'b00010

`define ALUOp\_SUBU 5'b00011

`define ALUOp\_SUB 5'b00100

`define ALUOp\_AND 5'b00101

`define ALUOp\_OR 5'b00110

`define ALUOp\_NOR 5'b00111

`define ALUOp\_XOR 5'b01000

`define ALUOp\_SLT 5'b01001

`define ALUOp\_SLTU 5'b01010

`define ALUOp\_EQL 5'b01011

`define ALUOp\_BNE 5'b01100

`define ALUOp\_GT0 5'b01101

`define ALUOp\_GE0 5'b01110

`define ALUOp\_LT0 5'b01111

`define ALUOp\_LE0 5'b10000

`define ALUOp\_SLL 5'b10001

`define ALUOp\_SRL 5'b10010

`define ALUOp\_SRA 5'b10011

//My Update

`define ALUOp\_LUI 5'b10100

// GPR control signal

`define GPRSel\_RD 2'b00

`define GPRSel\_RT 2'b01

`define GPRSel\_31 2'b10

`define WDSel\_FromALU 2'b00

`define WDSel\_FromMEM 2'b01

`define WDSel\_FromPC 2'b10

// Memory control signal

`define BE\_SB 2'b00

`define BE\_SH 2'b01

`define BE\_SW 2'b10

`define ME\_LB 3'b000

`define ME\_LBU 3'b001

`define ME\_LH 3'b010

`define ME\_LHU 3'b011

`define ME\_LW 3'b100

//Alu Source 1 Control

`define AluSrc1\_Reg1 3'b000

`define AluSrc1\_Shamt 3'b001

`define AluSrc1\_EX\_MEM 3'b010

`define AluSrc1\_MEM\_WB 3'b011

`define AluSrc1\_ID\_EX 3'b100

//Alu Source 2 Control

`define AluSrc2\_Reg2 3'b000

`define AluSrc2\_16BitInstant 3'b001

`define AluSrc2\_EX\_MEM 3'b010

`define AluSrc2\_MEM\_WB 3'b011

`define AluSrc2\_ID\_EX 3'b100

`define RegDst\_High 1'b1 //使用20:16作为写寄存器的地址

`define RegDst\_Low 1'b0 //使用15:11作为写寄存器的地址

`define Mem2R\_From\_Alu 1'b 0

`define Mem2R\_From\_Mem 1'b 1

### 2.4.3 总线位宽宏定义（bus\_def.v）

`define Word\_Bus 31:0 //The bus size of a word(32 bits)

`define Reg\_Addr\_Bus 4:0 //The bus size of register file

`define DMem\_Addr\_Bus 4:0 //The bus size of Data Memory

`define IMem\_Addr\_Bus 9:0 //The bus size of Instruction Memory

`define NOPRegAddr 5'b00000

`define NOPDMemAddr 5'b00000

`define ZeroWord 32'h00000000

`define AluSrc\_Bus 2:0 //The bus size of the ALUScr control signal

### 2.4.4程序计数器（PcUnit.v）

`include "bus\_def.v"

module PcUnit(output reg [`Word\_Bus] PC,

            input PcReSet,

            input Change\_PC\_en,

            input Stall,

            input Clk,

            input [`Word\_Bus] PC\_New);

    integer i;

    reg [31:0] temp;

    always@(posedge Clk or posedge PcReSet)

     begin

        if(PcReSet == 1) begin //重置信号为1，则重置PC值为0

            PC <= 32'h0000\_3000;

        end

       else if(Change\_PC\_en==1) begin //改变PC值使能信号为1，表明分支或跳转发生

           PC<=PC\_New; //则用新的PC值替换原有PC值

        end

         //若阻塞信号为1，则PC值不+4，这样在下一周期取指时还会取出本周期的指令

        else if(Stall==0) begin //否则PC值+4

            PC <= PC+4;

        end

    end

endmodule

### 2.4.5寄存器堆（RegFile.v）

module RegFile(input clk, //时钟信号

input rst, //复位信号

input WriteEn, //写使能信号

input [4:0] WriteAddr, //被写入寄存器的地址

input [31:0] WriteData, //要写入寄存器的数据

//读端口1的地址和输出数据

input [4:0] ReadAddr1, //被读出的第1个寄存器地址

output [31:0] DataOut1, //被读出的第1个寄存器数据

//读端口2的地址和输出数据

input [4:0] ReadAddr2, //被读出的第2个寄存器地址

output [31:0] DataOut2); //被读出的第2个寄存器数据

        reg [31:0] Registers[31:0];

        integer i;

        always@(posedge clk or posedge rst)

         begin

if(rst==1) begin

$display("Clear the reg file!");

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

Registers[i]<=0;

end

end

//如果写使能信号为1，则进行写操作

                else if(WriteEn == 1)

                    Registers[WriteAddr] = WriteData;

$display("R[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

0, Registers[1], Registers[2], Registers[3], Registers[4],

Registers[5], Registers[6], Registers[7]);

$display("R[08-15]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

Registers[8], Registers[9], Registers[10], Registers[11], Registers[12], Registers[13], Registers[14], Registers[15]);

$display("R[16-23]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", Registers[16], Registers[17], Registers[18], Registers[19], Registers[20], Registers[21], Registers[22], Registers[23]);

$display("R[24-31]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", Registers[24], Registers[25], Registers[26], Registers[27], Registers[28], Registers[29], Registers[30], Registers[31]);

end

//若某个读地址和写入地址相同，则直接将要写入的数据送到读出数据口

//(否则，本次写入的数据要到下个周期才生效，本次读出的数据是上个周期//的老数据，是错的)

        assign DataOut1 = (ReadAddr1==0)?0:

(WriteEn==1&&WriteAddr==ReadAddr1)?

WriteData:Registers[ReadAddr1];

        assign DataOut2 = (ReadAddr2==0)?0:

(WriteEn==1&&WriteAddr==ReadAddr2)?

WriteData:Registers[ReadAddr2];

endmodule

### 2.4.6算术逻辑运算单元（Alu.v）

`include "ctrl\_encode\_def.v"

`include "bus\_def.v"

module Alu( input [`Word\_Bus] DataIn1,    //输入: ALU的第一个输入

            input [`Word\_Bus] DataIn2,    //输入: ALU的第二个输入

            input [4:0] AluCtrl,          //输入: ALU的控制信号

            output reg[`Word\_Bus] AluResult,    //输出: ALU运算结果

            output reg Zero //输出: Zero=1表明ALU运算结果全0

);

  initial               //Initialize

   begin

     Zero = 0;

     AluResult = 0;

   end

  always@(DataIn1 or DataIn2 or AluCtrl)

  begin

   case(AluCtrl)

   `ALUOp\_ADD: AluResult = DataIn1+DataIn2;

   `ALUOp\_ADDU: AluResult = DataIn1+DataIn2;

   `ALUOp\_SUB: AluResult = DataIn1-DataIn2;

   `ALUOp\_SUBU: AluResult = DataIn1-DataIn2;

   `ALUOp\_OR: AluResult = DataIn1|DataIn2;

   `ALUOp\_AND: AluResult = DataIn1&DataIn2;

   `ALUOp\_SLL: AluResult = DataIn2<<DataIn1;

   `ALUOp\_SRL: AluResult = DataIn2>>DataIn1;

   `ALUOp\_SLT:

   begin

   if(DataIn1[31]==1&&DataIn2[31]==0)

   AluResult=1;

   else if(DataIn1[31]==0&&DataIn2[31]==1)

   AluResult=0;

   else

   AluResult = (DataIn1<DataIn2)?1:0;

   end

   `ALUOp\_EQL: AluResult = DataIn1-DataIn2;

   `ALUOp\_BNE: AluResult = DataIn1-DataIn2;

   `ALUOp\_LUI: AluResult = DataIn2;

   default: ;

   endcase

   if(AluCtrl==`ALUOp\_BNE)

   begin

   case(AluResult)

   0: Zero=0;

   default: Zero=1;

   endcase

   end

   else

   begin

   case(AluResult)

   0: Zero=1;

   default: Zero=0;

   endcase

   end

  end

endmodule

### 2.4.7扩展单元（Extender.v）

`include "ctrl\_encode\_def.v"

module Extender(

input [15:0] Imm16, //输入: 来自指令低16位的立即数

input [1:0] EXTOp, //输入: 扩展器控制信号

output reg [31:0] Imm32 //输出: 扩展得到的32位立即数

);

always @(\*) begin

case (EXTOp)

`EXT\_ZERO: Imm32 = {16'd0, Imm16};

`EXT\_SIGNED: Imm32 = {{16{Imm16[15]}}, Imm16};

`EXT\_HIGHPOS: Imm32 = {Imm16, 16'd0};

default: ;

endcase

end

endmodule

### 2.4.8指令存储器（IMem.v）

`include "bus\_def.v"

module IMem(OpCode,ImAdress);

     input [`IMem\_Addr\_Bus] ImAdress; //当前指令的地址

     output [`Word\_Bus] OpCode; //输出的32位指令

     reg [`Word\_Bus] Opcode;

     reg [`Word\_Bus] IMem[1023:0];

    always@(ImAdress)

     begin

         $display("IMem[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

IMem[0],IMem[1],IMem[2],IMem[3],IMem[4],IMem[5],IMem[6],IMem[7]);

         Opcode = IMem[ImAdress];

     end

    assign OpCode = Opcode;

endmodule

### 2.4.9控制器（Ctrl.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: ID级控制器

\*/

`include "ctrl\_encode\_def.v"

`include "instruction\_def.v"

`include "bus\_def.v"

module Ctrl( input [5:0] OpCode, //输入: 指令的6位操作码

input [5:0] funct, //输入: 指令的6位功能码

output reg jump, //输出: 跳转使能信号

output reg RegDst, //输出: 写入寄存器选择(Rt或Rd)

output reg Branch, //输出: 分支使能信号

output reg MemR, //输出: 内存读信号,lw指令有效

//输出: 选择将从内存读出的数据作为写回寄存器的数据,lw指令有效output reg Mem2R,

output reg MemW, //输出: 内存写信号,sw指令有效

//输出: 写回寄存器信号,所有需要向寄存器写回结果的指令均有效

output reg RegW,

output reg [`AluSrc\_Bus] AluSrc1, //输出: ALU的第一个输入的选择信号

output reg [`AluSrc\_Bus] AluSrc2, //输出: ALU的第二个输入的选择信号

output reg[1:0] ExtOp, //输出: ID级的扩展器的控制信号

output reg[4:0] Aluctrl //输出: ALU的控制信号

);

initial begin

jump=0;

RegDst=0;

Branch=0;

MemR=0;

Mem2R=0;

MemW=0;

RegW=0;

AluSrc1=0;

AluSrc2=0;

ExtOp=0;

Aluctrl=0;

end

always @(OpCode or funct)

begin

case(OpCode)

`INSTR\_RTYPE\_OP: //R Type Instruction

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc2=`AluSrc2\_Reg2;

ExtOp=`EXT\_ZERO;

RegDst=0;

case(funct)

`INSTR\_ADD\_FUNCT: //Add

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_ADD;

end

`INSTR\_ADDU\_FUNCT: //Addu

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_ADDU;

end

`INSTR\_SUB\_FUNCT: //Sub

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_SUB;

end

`INSTR\_SUBU\_FUNCT: //Subu

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_SUBU;

end

`INSTR\_SLL\_FUNCT: //SLL Instruction

begin

RegW=1;

AluSrc1=`AluSrc1\_Shamt;

Aluctrl=`ALUOp\_SLL;

end

`INSTR\_SRL\_FUNCT: //SRL Instruction

begin

RegW=1;

AluSrc1=`AluSrc1\_Shamt;

Aluctrl=`ALUOp\_SRL;

end

`INSTR\_SLT\_FUNCT: //SLT Instruction

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_SLT;

end

`INSTR\_AND\_FUNCT: //AND Instruction

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_AND;

end

`INSTR\_OR\_FUNCT: //OR Instruction

begin

RegW=1;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=`ALUOp\_OR;

end

default:

begin

RegW=0;

AluSrc1=`AluSrc1\_Reg1;

Aluctrl=0;

end

endcase

end

`INSTR\_ORI\_OP: //Ori

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_SIGNED;

RegDst=1;

RegW=1;

Aluctrl=`ALUOp\_OR;

end

`INSTR\_LW\_OP: //LW Instruction

begin

Branch=0;

jump=0;

Mem2R=1;

MemW=0;

MemR=1;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_SIGNED;

RegDst=1;

RegW=1;

Aluctrl=`ALUOp\_ADD;

end

`INSTR\_SW\_OP: //SW Instruction

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=1;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_SIGNED;

RegDst=0;

RegW=0;

Aluctrl=`ALUOp\_ADD;

end

`INSTR\_BEQ\_OP: //BEQ Instruction

begin

Branch=1;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_Reg2;

ExtOp=`EXT\_SIGNED;

RegDst=0;

RegW=0;

Aluctrl=`ALUOp\_EQL;

end

`INSTR\_LUI\_OP: //LUI Instruction

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_HIGHPOS;

RegDst=1;

RegW=1;

Aluctrl=`ALUOp\_LUI;

end

`INSTR\_BNE\_OP: //BNE Instruction

begin

Branch=1;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_Reg2;

ExtOp=`EXT\_SIGNED;

RegDst=0;

RegW=0;

Aluctrl=`ALUOp\_BNE;

end

`INSTR\_J\_OP: //Jump Instruction

begin

Branch=0;

jump=1;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_Reg2;

ExtOp=`EXT\_ZERO;

RegDst=0;

RegW=0;

Aluctrl=`ALUOp\_BNE;

end

`INSTR\_SLTI\_OP: //SLTI Instruction

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_SIGNED;

RegDst=1;

RegW=1;

Aluctrl=`ALUOp\_SLT;

end

`INSTR\_ADDI\_OP: //ADDI Instruction

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_16BitInstant;

ExtOp=`EXT\_SIGNED;

RegDst=1;

RegW=1;

Aluctrl=`ALUOp\_ADD;

end

default:

begin

Branch=0;

jump=0;

Mem2R=0;

MemW=0;

MemR=0;

AluSrc1=`AluSrc1\_Reg1;

AluSrc2=`AluSrc2\_Reg2;

ExtOp=`EXT\_ZERO;

RegDst=0;

RegW=0;

Aluctrl=0;

end

endcase

end

endmodule

### 2.4.10 ID级（ID.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: ID级模块

\*/

`include "ctrl\_encode\_def.v"

`include "instruction\_def.v"

`include "bus\_def.v"

module ID( input clk, //输入: 时钟信号

    input rst, //输入: 重置信号

input [`Word\_Bus] Instruction, //输入: 32位指令

//输入: 来自WB级指令的写回寄存器的地址

    input [`Reg\_Addr\_Bus] Write\_RegAddr\_From\_WB,

//输入: 来自WB级指令的要写回寄存器的数据

     input [`Word\_Bus] Write\_RegData\_From\_WB,

     input WriteEn\_From\_WB, //输入: 来自WB级指令的写回寄存器使能信号

//输出: ID级指令从寄存器堆读出的第一个寄存器的数据

output [`Word\_Bus] Reg1,

//输出: ID级指令从寄存器堆读出的第二个寄存器的数据

output [`Word\_Bus] Reg2,

//输出: ID级指令要写回的寄存器的地址

     output [`Reg\_Addr\_Bus] Write\_RegAddr,

     output [`Word\_Bus] Instant, //输出: ID级指令的立即数

output [`Word\_Bus] Shamt, //输出: ID级指令的位移量(用于SLL,SRL指令)

output jump,    //输出: 跳转使能信号

     output RegDst,  //输出: 写入寄存器选择(Rt或Rd)

     output Branch,  //输出: 分支使能信号

     output MemR,    //输出: 内存读信号,lw指令有效

 //输出: 选择将从内存读出的数据作为写回寄存器的数据,lw指令有效

     output Mem2R,

     output MemW,    //输出: 内存写信号,sw指令有效

//输出: 写回寄存器信号,所有需要向寄存器写回结果的指令均有效

     output RegW,

     output [`AluSrc\_Bus] AluSrc1, //输出: ALU的第一个输入的选择信号

output [`AluSrc\_Bus] AluSrc2, //输出: ALU的第二个输入的选择信号

          output [4:0] Aluctrl, //输出: ALU的控制信号

         input [`Word\_Bus] DataSrc\_EX\_MEM, //来自EX\_MEM的旁路数据

     input [`Word\_Bus] DataSrc\_MEM\_WB, //来自MEM\_WB的旁路数据

input [`Word\_Bus] ID\_PC, //ID级指令所处的PC值

    input MEM\_Reg\_WriteEn, //MEM级指令的寄存器写回使能信号

//MEM级指令的寄存器写回地址

     input [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr,

     input WB\_Reg\_WriteEn, //WB级指令的寄存器写回使能信号

     input [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr, //WB级指令的寄存器写回地址

//对beq,bne,j型指令有效,分支或跳转发生后的新的PC值

          output [`Word\_Bus] PC\_New,

//对beq,bne,j型指令有效,用PC\_New改变当前PC值的使能信号

     output Change\_PC\_en,

         output Stall\_IF\_ID, //ID级流水线暂停信号

     input [`Reg\_Addr\_Bus] EX\_Reg\_WriteAddr, //EX级指令的寄存器写回地址

     input EX\_Reg\_WriteEn, //EX级指令的寄存器写回使能信号

         output [`Reg\_Addr\_Bus] Rs\_num, //ID级指令的Rs寄存器编号

     output [`Reg\_Addr\_Bus] Rt\_num, //ID级指令的Rt寄存器编号

     input EX\_MemRead, //EX级指令的存储器读信号

     input MEM\_MemRead //MEM级指令的存储器读信号

);

wire [5:0] OpCode; //6位操作码

wire [5:0] funct; //R型指令功能码

wire [1:0] ExtOp; //扩展器控制信号

wire [25:0] JumpAddr; //J型指令的26位常数

wire [15:0] Instant\_16bit; //I型指令的16位立即数

assign OpCode = Instruction[31:26];

assign funct[5:0] = Instruction[5:0];

assign Shamt[31:0] = {27'd0,Instruction[10:6]};

assign JumpAddr[25:0] = Instruction[25:0];

assign Instant\_16bit[15:0]=Instruction[15:0];

assign Rs\_num[`Reg\_Addr\_Bus] = Instruction[25:21];

assign Rt\_num[`Reg\_Addr\_Bus] = Instruction[20:16];

assign Write\_RegAddr = (RegDst==`RegDst\_High)?

            Instruction[20:16]:Instruction[15:11];

Ctrl U\_Ctrl( .OpCode(OpCode),

     .funct(funct),

     .jump(jump),

     .RegDst(RegDst),

     .Branch(Branch),

     .MemR(MemR),

     .Mem2R(Mem2R),

     .MemW(MemW),

     .RegW(RegW),

     .AluSrc1(AluSrc1),

.AluSrc2(AluSrc2),

     .ExtOp(ExtOp),

     .Aluctrl(Aluctrl));

    RegFile U\_RegFile( .clk(clk), //时钟信号

.rst(rst), //复位信号

.WriteEn(WriteEn\_From\_WB), //写使能信号

.WriteAddr(Write\_RegAddr\_From\_WB), //被写入寄存器的编号

.WriteData(Write\_RegData\_From\_WB), //要写入寄存器的数据

//读端口1

.ReadAddr1(Rs\_num), //第一个被读寄存器编号

.DataOut1(Reg1), //第一个被读寄存器数据

         //读端口2

         .ReadAddr2(Rt\_num), //第二个被读寄存器编号

.DataOut2(Reg2) //第二个被读寄存器数据

     );

  /\*

    ID级旁路控制逻辑

    由于beq,bne指令的判定分支是在ID级实现的，因此类似于EX级，ID级也需要加入

旁路解决数据相关问题。

    当来自EX,MEM,WB级的旁路数据中有多个同时有效时，应优先使用最早的数据.

    换言之，三种旁路数据的优先级为: EX>MEM>WB。

    \*/

Extender U\_Extender( .Imm16(Instant\_16bit),

.EXTOp(ExtOp),

.Imm32(Instant));

wire [`AluSrc\_Bus] Bypassed\_AluSrc1/\*synthesis noprune\*/;

wire [`AluSrc\_Bus] Bypassed\_AluSrc2/\*synthesis noprune\*/;

wire [`Word\_Bus] Branch\_Data1/\*synthesis noprune\*/;

wire [`Word\_Bus] Branch\_Data2/\*synthesis noprune\*/;

wire [`Word\_Bus] ID\_PC\_plus4/\*synthesis noprune\*/;

wire MEM\_WB\_Rs\_Bypass\_En,MEM\_WB\_Rt\_Bypass\_En,EX\_MEM\_Rs\_Bypass\_En,

        EX\_MEM\_Rt\_Bypass\_En,ID\_EX\_Rs\_Bypass\_En,ID\_EX\_Rt\_Bypass\_En;

     //=1表明MEM/WB级Rs旁路数据是有效的

    assign MEM\_WB\_Rs\_Bypass\_En=WB\_Reg\_WriteEn==1&&

                                WB\_Reg\_WriteAddr!=0&&

                                WB\_Reg\_WriteAddr==Rs\_num;

    //=1表明EX/MEM级Rs旁路数据是有效的

    assign EX\_MEM\_Rs\_Bypass\_En=MEM\_Reg\_WriteEn==1&&

                                MEM\_Reg\_WriteAddr!=0&&

                                MEM\_Reg\_WriteAddr==Rs\_num;

    //=1表明ID/EX级Rs旁路数据是有效的

    assign ID\_EX\_Rs\_Bypass\_En=EX\_Reg\_WriteEn==1&&

                                EX\_Reg\_WriteAddr!=0&&

                                EX\_Reg\_WriteAddr==Rs\_num;

    //=1表明MEM/WB级Rt旁路数据是有效的

    assign MEM\_WB\_Rt\_Bypass\_En=WB\_Reg\_WriteEn==1&&

                                WB\_Reg\_WriteAddr!=0&&

                                WB\_Reg\_WriteAddr==Rt\_num;

    //=1表明EX/MEM级Rt旁路数据是有效的

    assign EX\_MEM\_Rt\_Bypass\_En=MEM\_Reg\_WriteEn==1&&

                                MEM\_Reg\_WriteAddr!=0&&

                                MEM\_Reg\_WriteAddr==Rt\_num;

    //=1表明ID/EX级Rt旁路数据是有效的

    assign ID\_EX\_Rt\_Bypass\_En=EX\_Reg\_WriteEn==1&&

                                EX\_Reg\_WriteAddr!=0&&

                                EX\_Reg\_WriteAddr==Rt\_num;

    //经旁路得到的ALU的第1个源操作数的来源

    assign Bypassed\_AluSrc1= (MEM\_WB\_Rs\_Bypass\_En==1&&

EX\_MEM\_Rs\_Bypass\_En==0&&ID\_EX\_Rs\_Bypass\_En==0)?

             `AluSrc1\_MEM\_WB:

            (EX\_MEM\_Rs\_Bypass\_En==1&&ID\_EX\_Rs\_Bypass\_En==0)?

            `AluSrc1\_EX\_MEM:

            (ID\_EX\_Rs\_Bypass\_En==1)?

            `AluSrc1\_ID\_EX:

            `AluSrc1\_Reg1;

     //经旁路得到的ALU的第2个源操作数的来源

    assign Bypassed\_AluSrc2=

            (MEM\_WB\_Rt\_Bypass\_En==1&&EX\_MEM\_Rt\_Bypass\_En==0&&

ID\_EX\_Rt\_Bypass\_En==0)?

            `AluSrc2\_MEM\_WB:

            (EX\_MEM\_Rt\_Bypass\_En==1&&ID\_EX\_Rt\_Bypass\_En==0)?

            `AluSrc2\_EX\_MEM:

            (ID\_EX\_Rt\_Bypass\_En==1)?

            `AluSrc2\_ID\_EX:

            `AluSrc2\_Reg2;

    //经旁路得到的ALU的第1个源操作数

    assign Branch\_Data1=(Bypassed\_AluSrc1==`AluSrc1\_Reg1)?Reg1:

                    (Bypassed\_AluSrc1==`AluSrc1\_EX\_MEM)?DataSrc\_EX\_MEM:

                    DataSrc\_MEM\_WB;

    //经旁路得到的ALU的第2个源操作数

    assign Branch\_Data2=(Bypassed\_AluSrc2==`AluSrc2\_Reg2)?Reg2:

            (Bypassed\_AluSrc2==`AluSrc2\_EX\_MEM)?DataSrc\_EX\_MEM:

            DataSrc\_MEM\_WB;

    //ID级指令所对应PC值+4

    assign ID\_PC\_plus4=ID\_PC+4;

    //=1表示发生了跳转或分支，需要改变PC单元的PC值

    assign Change\_PC\_en=(Stall\_IF\_ID==0)&&((OpCode==`INSTR\_J\_OP)||

            (OpCode==`INSTR\_BEQ\_OP&&Branch\_Data1==Branch\_Data2)||

            (OpCode==`INSTR\_BNE\_OP&&Branch\_Data1!=Branch\_Data2));

  /\*

    ID级阻塞控制逻辑

    由于beq,bne指令的判定分支是在ID级实现的，因此类似于EX级，

ID级也需要加入因数据相关引发的阻塞。当前一条数据为lw/R类型且发生

先写后读(Read after write)类型数据相关时，需要暂停ID及其之前的所有单元

一个周期。

    阻塞发生条件：

    当EX级的寄存器写回信号为1，对应指令的写入寄存器号不为0，

且等于ID级指令的Rs或Rt时。

 \*/

 //=1表示需要暂停IF/ID寄存器，同时PC Unit,IF,ID级都暂停，而EX,MEM,WB级继续执行

assign Stall\_IF\_ID=(OpCode==`INSTR\_BEQ\_OP||OpCode==`INSTR\_BNE\_OP)&&

                        (Bypassed\_AluSrc1==`AluSrc1\_ID\_EX||(Bypassed\_AluSrc1==

`AluSrc1\_EX\_MEM&&MEM\_MemRead==1)||

                        Bypassed\_AluSrc2==`AluSrc2\_ID\_EX||(Bypassed\_AluSrc2==

`AluSrc2\_EX\_MEM&&MEM\_MemRead==1));

 //跳转或分支改变PC单元的新PC值

assign PC\_New=(Stall\_IF\_ID==1)?ID\_PC\_plus4:

            (OpCode==`INSTR\_J\_OP)?{ID\_PC\_plus4[31:28],JumpAddr,2'd0}:

            (OpCode==`INSTR\_BEQ\_OP&&Branch\_Data1==Branch\_Data2)?

(ID\_PC\_plus4+(Instant<<2)):

            (OpCode==`INSTR\_BNE\_OP&&Branch\_Data1!=Branch\_Data2)?

(ID\_PC\_plus4+(Instant<<2)):

            ID\_PC;

endmodule

### 2.4.11 EX级（EX.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: EX级模块

\*/

`include "ctrl\_encode\_def.v"

`include "bus\_def.v"

module EX( input [`Word\_Bus] DataSrc\_EX\_MEM, // //来自EX\_MEM的旁路数据

     input [`Word\_Bus] DataSrc\_MEM\_WB, //来自MEM\_WB的旁路数据

             input [`Word\_Bus] DataSrc\_ID\_EX\_Reg1, //来自ID/EX第1个寄存器的数据

input [`Word\_Bus] DataSrc\_ID\_EX\_Reg2, //来自ID/EX第2个寄存器的数据

input [`Word\_Bus] DataSrc\_ID\_EX\_Instant, //来自ID/EX的立即数

             input [`Word\_Bus] DataSrc\_Shamt, //来自ID/EX的5位位移量Shamt

input [`AluSrc\_Bus] AluSrc1, //ID级判定的ALU第1个数据的来源

input [`AluSrc\_Bus] AluSrc2, //ID级判定的ALU第2个数据的来源

     input [4:0] AluCtrl, //ALU控制信号，决定ALU执行什么类型的运算

               input [`Reg\_Addr\_Bus] EX\_Rs\_num, //EX级的指令的Rs寄存器号

             input [`Reg\_Addr\_Bus] EX\_Rt\_num, //EX级的指令的Rt寄存器号

             input MEM\_Reg\_WriteEn, //MEM级的指令的寄存器写回信号

//MEM级的指令的寄存器写回地址

             input [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr,

             input WB\_Reg\_WriteEn, //WB级的指令的寄存器写回信号

             input [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr, //WB级的指令的寄存器写回地址

     output [`Word\_Bus] AluResult, //ALU运算结果

             //output reg[Reg\_Addr\_Bus] Reg\_Addr,

     output Zero, //Zero=1表示ALU的运算结果为全零

             input DMem\_WriteEn, //EX级的指令的存储器写入信号

//将要送给MEM级的被旁路过的送入存储器的数据

             output [`Word\_Bus] Bypassed\_DMem\_WriteData,

             //Signal for lw stall

             input MEM\_DMem\_ReadEn, //MEM级存储器读信号

             output Stall\_ID\_EX);            //ID/EX寄存器暂停的信号

    wire [`AluSrc\_Bus] Bypassed\_AluSrc1/\*synthesis noprune\*/;

    wire [`AluSrc\_Bus] Bypassed\_AluSrc2/\*synthesis noprune\*/;

    //ALU源操作数

    wire [`Word\_Bus] Alu\_Data1/\*synthesis noprune\*/;

    wire [`Word\_Bus] Alu\_Data2/\*synthesis noprune\*/;

    /\*

     EX级旁路控制逻辑

     当来自MEM,WB级的旁路数据同时有效时，应优先使用MEM级的数据。

    \*/

    //旁路后的ALU第1个操作数来源

    assign Bypassed\_AluSrc1=(AluSrc1==`AluSrc1\_Reg1&&WB\_Reg\_WriteEn==1&&

            WB\_Reg\_WriteAddr!=0&&WB\_Reg\_WriteAddr==EX\_Rs\_num&&

            !(MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0&&

MEM\_Reg\_WriteAddr==EX\_Rs\_num))?

            `AluSrc1\_MEM\_WB:

            (AluSrc1==`AluSrc1\_Reg1&&MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0

&&MEM\_Reg\_WriteAddr==EX\_Rs\_num)?

`AluSrc1\_EX\_MEM:AluSrc1;

        //旁路后的ALU第2个操作数来源

    assign Bypassed\_AluSrc2=(AluSrc2==`AluSrc2\_Reg2&&WB\_Reg\_WriteEn==1&&

            WB\_Reg\_WriteAddr!=0&&WB\_Reg\_WriteAddr==EX\_Rt\_num&&

            !(MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0&&

MEM\_Reg\_WriteAddr==EX\_Rt\_num))?

            `AluSrc2\_MEM\_WB:

            (AluSrc2==`AluSrc2\_Reg2&&MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0

&&MEM\_Reg\_WriteAddr==EX\_Rt\_num)?

            `AluSrc2\_EX\_MEM:

            AluSrc2;

    always@(Bypassed\_AluSrc1 or Bypassed\_AluSrc2) begin

        $display("Dbg Alu: SRC1=%8X / SRC2=%8X",Bypassed\_AluSrc1,Bypassed\_AluSrc2);

    end

    //旁路后的ALU第1个操作数

    assign Alu\_Data1=(Bypassed\_AluSrc1==`AluSrc1\_Reg1)?DataSrc\_ID\_EX\_Reg1:

                    (Bypassed\_AluSrc1==`AluSrc1\_Shamt)?DataSrc\_Shamt:

                    (Bypassed\_AluSrc1==`AluSrc1\_EX\_MEM)?DataSrc\_EX\_MEM:

                    DataSrc\_MEM\_WB;

    //旁路后的ALU第2个操作数

    assign Alu\_Data2=(Bypassed\_AluSrc2==`AluSrc2\_Reg2)?DataSrc\_ID\_EX\_Reg2:

            (Bypassed\_AluSrc2==`AluSrc2\_16BitInstant)?DataSrc\_ID\_EX\_Instant:

            (Bypassed\_AluSrc2==`AluSrc2\_EX\_MEM)?DataSrc\_EX\_MEM:

            DataSrc\_MEM\_WB;

    assign Bypassed\_DMem\_WriteData=(DMem\_WriteEn==0)?0:

            (WB\_Reg\_WriteEn==1&&

            WB\_Reg\_WriteAddr!=0&&WB\_Reg\_WriteAddr==EX\_Rt\_num&&

            !(MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0&&

MEM\_Reg\_WriteAddr==EX\_Rt\_num))?

            DataSrc\_MEM\_WB:

            (MEM\_Reg\_WriteEn==1&&MEM\_Reg\_WriteAddr!=0&&

MEM\_Reg\_WriteAddr==EX\_Rt\_num)?

            DataSrc\_EX\_MEM: DataSrc\_ID\_EX\_Reg2;

  /\*

    EX级阻塞控制逻辑

    当前一条数据为lw类型且发生先写后读(Read after write)类型数据相关时，

需要暂停EX及其之前的所有单元一个周期。

    阻塞发生条件：当MEM级的存储器读信号为1，对应指令的写入寄存器号不为0，

且等于EX级指令的Rs或Rt时，则暂停。

  \*/

    assign Stall\_ID\_EX=(Bypassed\_AluSrc1==`AluSrc1\_EX\_MEM||Bypassed\_AluSrc2==

`AluSrc2\_EX\_MEM)&&MEM\_DMem\_ReadEn==1;

    Alu U\_Alu(  .DataIn1(Alu\_Data1),        //1st input data of ALU

                .DataIn2(Alu\_Data2),        //2nd input data of ALU

                .AluCtrl(AluCtrl),      //ALU control signal

                .AluResult(AluResult),      //The output of ALU

                .Zero(Zero));

endmodule

### 2.4.12 MEM级（MEM.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: MEM级(数据存储器)

\*/

`include "bus\_def.v"

module MEM( input MemWrite, //存储器写信号

input MemRead, //存储器读信号

input clk, //时钟信号

input [4:0] DataAddr, //地址

input [31:0] DataIn, //要写入存储器的数据

             output [31:0] DataOut //从存储器取出的数据

);

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

always@(posedge clk) //Write the mem when pos edge comes

    begin

        if(MemWrite)

             DMem[DataAddr] = DataIn;

        $display("addr=%8X",DataAddr);//addr to DM

     $display("din=%8X",DataIn);//data to DM

     $display("Mem[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

DMem[0],DMem[4],DMem[8],DMem[12],DMem[16],DMem[20],DMem[24],DMem[28]);

    end

 assign DataOut = DMem[DataAddr];

endmodule

### 2.4.13 IF/ID寄存器（IF\_ID.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: IF/ID寄存器

\*/

`include "bus\_def.v"

module IF\_ID(input clk, //输入: 时钟信号

input rst, //输入: 重置信号

input Stall, //输入: ID级流水线暂停信号

input [`Word\_Bus] IF\_PC, //输入: IF级的PC值

input [`Word\_Bus] IF\_Instruction, //输入: IF级从指令存储器取出的指令

     output reg [`Word\_Bus] ID\_PC, //输出: ID级的PC值

output reg [`Word\_Bus] ID\_Instruction //输出: ID级的指令

);

initial begin

ID\_PC<=0;

ID\_Instruction<=0;

end

always@(posedge clk) begin

if (rst==1) begin

ID\_PC<=0;

ID\_Instruction<=0;

$display("IF/ID Reset!");

end

else if(Stall==1) begin

ID\_PC<=ID\_PC;

ID\_Instruction<=ID\_Instruction;

       end

    else begin

ID\_PC<=IF\_PC;

ID\_Instruction<=IF\_Instruction;

$display("IF/ID: %8X %8X",ID\_PC,ID\_Instruction);

end

end

endmodule

### 2.4.14 ID/EX寄存器（ID\_EX.v）

`include "bus\_def.v"

module ID\_EX(input clk,

input rst,

     //Signal for stall

     input Stall,

     //Information came from ID stage

//1. Reg1&2 , intant number, shamt value in ID

input [`Word\_Bus] ID\_Reg1, //ID级取出的寄存器1的值

input [`Word\_Bus] ID\_Reg2, //ID级取出的寄存器2的值

input [`Word\_Bus] ID\_Instant, //ID级取出的经扩展得到的立即数的值

input [`Word\_Bus]   ID\_Shamt, //来自ID/EX的Shamt偏移量

//2. Control signal

//--2.1 EX级控制信号

input [`AluSrc\_Bus] ID\_AluSrc1, //ALU源操作数1控制信号

input [`AluSrc\_Bus] ID\_AluSrc2, //ALU源操作数2控制信号

input [4:0] ID\_AluCtrl, //来自ID/EX的ALU控制信号

//--2.2 MEM级控制信号

input ID\_MemWrite,

input ID\_MemRead,

//input [`DMem\_Addr\_Bus] ID\_MemAddr,

//--2.3 WB级控制信号

input ID\_RegWrite,

input ID\_Mem2R,

input [`Reg\_Addr\_Bus] ID\_RegAddr,

//3. rs,rt,rd寄存器号

input [`Reg\_Addr\_Bus] ID\_Rs\_num,

input [`Reg\_Addr\_Bus] ID\_Rt\_num,

     //input [`Reg\_Addr\_Bus] ID\_Rd\_num,

     //Information that sends into EX stage

//1. Reg1&2 , intant number, shamt value in ID

output reg [`Word\_Bus] EX\_Reg1, //ID级取出的寄存器1的值

output reg [`Word\_Bus] EX\_Reg2, //ID级取出的寄存器2的值

output reg [`Word\_Bus] EX\_Instant, //ID级取出的经扩展得到的立即数的值

output reg [`Word\_Bus] EX\_Shamt, //来自ID/EX的Shamt偏移量

//2. Control signal

//--2.1 EX级控制信号

output reg [`AluSrc\_Bus] EX\_AluSrc1, //ALU源操作数1控制信号

output reg [`AluSrc\_Bus] EX\_AluSrc2, //ALU源操作数2控制信号

output reg [4:0] EX\_AluCtrl, //来自ID/EX的ALU控制信号

//--2.2 MEM级控制信号

output reg EX\_MemWrite,

output reg EX\_MemRead,

//output reg [`DMem\_Addr\_Bus] EX\_MemAddr,

//--2.3 WB级控制信号

output reg EX\_RegWrite,

output reg EX\_Mem2R,

output reg [`Reg\_Addr\_Bus] EX\_RegAddr,

//3. rs,rt,rd寄存器号

output reg [`Reg\_Addr\_Bus] EX\_Rs\_num,

    output reg [`Reg\_Addr\_Bus] EX\_Rt\_num

     //output reg [`Reg\_Addr\_Bus] EX\_Rd\_num,

);

initial begin

EX\_Reg1<=0;

EX\_Reg2<=0;

EX\_Instant<=0;

EX\_Shamt<=0;

EX\_AluSrc1<=0;

EX\_AluSrc2<=0;

EX\_AluCtrl<=0;

EX\_MemWrite<=0;

EX\_MemRead<=0;

//EX\_MemAddr<=0;

//--2.3 WB级控制信号

EX\_RegWrite<=0;

EX\_Mem2R<=0;

EX\_RegAddr<=0;

EX\_Rs\_num<=0;

EX\_Rt\_num<=0;

//EX\_Rd\_num<=0;

end

always@(posedge clk) begin

if(rst==1) begin

EX\_Reg1<=0;

EX\_Reg2<=0;

EX\_Instant<=0;

EX\_Shamt<=0;

EX\_AluSrc1<=0;

EX\_AluSrc2<=0;

EX\_AluCtrl<=0;

EX\_MemWrite<=0;

EX\_MemRead<=0;

//EX\_MemAddr<=0;

//--2.3 WB级控制信号

EX\_RegWrite<=0;

EX\_Mem2R<=0;

EX\_RegAddr<=0;

EX\_Rs\_num<=0;

EX\_Rt\_num<=0;

//EX\_Rd\_num<=0;

end

 else if(Stall==1) begin

EX\_Reg1<=EX\_Reg1;

EX\_Reg2<=EX\_Reg2;

EX\_Instant<=EX\_Instant;

EX\_Shamt<=EX\_Shamt;

EX\_AluSrc1<=EX\_AluSrc1;

EX\_AluSrc2<=EX\_AluSrc2;

EX\_AluCtrl<=EX\_AluCtrl;

EX\_MemWrite<=EX\_MemWrite;

EX\_MemRead<=EX\_MemRead;

//EX\_MemAddr<=ID\_MemAddr;

//--2.3 WB级控制信号

EX\_RegWrite<=EX\_RegWrite;

EX\_Mem2R<=EX\_Mem2R;

EX\_RegAddr<=EX\_RegAddr;

EX\_Rs\_num<=EX\_Rs\_num;

EX\_Rt\_num<=EX\_Rt\_num;

//EX\_Rd\_num<=ID\_Rd\_num;

  end

else begin

EX\_Reg1<=ID\_Reg1;

EX\_Reg2<=ID\_Reg2;

EX\_Instant<=ID\_Instant;

EX\_Shamt<=ID\_Shamt;

EX\_AluSrc1<=ID\_AluSrc1;

EX\_AluSrc2<=ID\_AluSrc2;

EX\_AluCtrl<=ID\_AluCtrl;

EX\_MemWrite<=ID\_MemWrite;

EX\_MemRead<=ID\_MemRead;

//EX\_MemAddr<=ID\_MemAddr;

//--2.3 WB级控制信号

EX\_RegWrite<=ID\_RegWrite;

EX\_Mem2R<=ID\_Mem2R;

EX\_RegAddr<=ID\_RegAddr;

EX\_Rs\_num<=ID\_Rs\_num;

EX\_Rt\_num<=ID\_Rt\_num;

//EX\_Rd\_num<=ID\_Rd\_num;

end

end

endmodule

### 2.4.15 EX/MEM寄存器（EX\_MEM.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: EX/MEM寄存器

\*/

`include "bus\_def.v"

module EX\_MEM( input clk,

input rst,

//Information came from EX stage

//1. Reg in WB

//EX级指令要写入的寄存器的地址

input [`Reg\_Addr\_Bus] EX\_Reg\_WriteAddr,

input EX\_Reg\_WriteEn, //=1 表示EX级指令要写寄存器

input [`Word\_Bus] EX\_AluResult, //EX级指令ALU的结果

//=1表示EX级指令选择将存储器读出的数据送入寄存器

input EX\_Mem2R,

//2. DMem in MEM

//EX级指令要写入的存储器单元的地址

input [`DMem\_Addr\_Bus] EX\_DMem\_WriteAddr,

input EX\_DMem\_WriteEn, //EX级指令存储器写入使能信号

input EX\_DMem\_ReadEn, //EX级指令存储器读出使能信号

input [`Word\_Bus] EX\_DMem\_WriteData, //EX级指令要写入存储器的数据

//3. 为支持旁路加入的部分信息

     //input [`Reg\_Addr\_Bus] EX\_Rs\_num,

     //input [`Reg\_Addr\_Bus] EX\_Rt\_num,

//Information that sends into MEM stage

//1. Reg in WB

//MEM级指令要写入寄存器的地址

output reg [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr,

output reg MEM\_Reg\_WriteEn, //=1 表示MEM级指令要写寄存器

output reg [`Word\_Bus] MEM\_AluResult, //MEM级指令ALU结果

//=1 表示MEM级指令选择将存储器读出的数据送入寄存器

output reg MEM\_Mem2R,

//2. DMem in MEM

//MEM级指令写入存储器单元的地址

output reg [`DMem\_Addr\_Bus] MEM\_DMem\_WriteAddr,

output reg MEM\_DMem\_WriteEn, //MEM级指令存储器写入使能信号

output reg MEM\_DMem\_ReadEn, //MEM级指令存储器读出使能信号

//MEM级指令要写入存储器的数据

output reg [`Word\_Bus] MEM\_DMem\_WriteData

);

initial begin

MEM\_Reg\_WriteAddr<=`NOPRegAddr;

MEM\_Reg\_WriteEn<=0;

MEM\_AluResult<=0; //The data that needs to be written into a register

MEM\_Mem2R<=0;

MEM\_DMem\_WriteAddr<=`NOPDMemAddr;

MEM\_DMem\_WriteEn<=0;

MEM\_DMem\_ReadEn<=0;

MEM\_DMem\_WriteData<=`ZeroWord;

//MEM\_Rs\_num<=0;

//MEM\_Rt\_num<=0;

end

always@(posedge clk) begin

if(rst==1) begin

MEM\_Reg\_WriteAddr<=`NOPRegAddr;

MEM\_Reg\_WriteEn<=0;

MEM\_AluResult<=0; //The data that needs to be written into a register

MEM\_Mem2R<=0;

MEM\_DMem\_WriteAddr<=`NOPDMemAddr;

MEM\_DMem\_WriteEn<=0;

MEM\_DMem\_ReadEn<=0;

MEM\_DMem\_WriteData<=`ZeroWord;

//MEM\_Rs\_num<=0;

//MEM\_Rt\_num<=0;

end

else begin

MEM\_Reg\_WriteAddr<=EX\_Reg\_WriteAddr;

MEM\_Reg\_WriteEn<=EX\_Reg\_WriteEn;

MEM\_AluResult<=EX\_AluResult; //The data that needs to be written into a register

MEM\_Mem2R<=EX\_Mem2R;

MEM\_DMem\_WriteAddr<=EX\_DMem\_WriteAddr;

MEM\_DMem\_WriteEn<=EX\_DMem\_WriteEn;

MEM\_DMem\_ReadEn<=EX\_DMem\_ReadEn;

MEM\_DMem\_WriteData<=EX\_DMem\_WriteData;

//MEM\_Rs\_num<=EX\_Rs\_num;

//MEM\_Rt\_num<=EX\_Rt\_num;

end

end

endmodule

### 2.4.16 MEM/WB寄存器（MEM\_WB.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: MEM/WB寄存器

\*/

`include "bus\_def.v"

module MEM\_WB( input clk,

input rst,

//Information came from MEM stage

//1. Reg in WB

input [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr, //MEM级指令写回寄存器的地址

input MEM\_Reg\_WriteEn, //MEM级指令写回寄存器使能信号

//input [`Word\_Bus] MEM\_AluResult,

//The data that needs to be written into a register

input [`Word\_Bus] MEM\_Reg\_WriteData, //MEM级指令写回寄存器的数据

//input MEM\_Mem2R,

//Information that sends into WB stage

//1. Reg in WB

//WB级指令写回寄存器的地址

output reg [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr,

output reg WB\_Reg\_WriteEn, //WB级指令写回寄存器使能信号

output reg [`Word\_Bus] WB\_Reg\_WriteData //WB级指令写回寄存器的数据

);

initial begin

WB\_Reg\_WriteAddr<=0;

WB\_Reg\_WriteEn<=0;

WB\_Reg\_WriteData<=0;

end

always@(posedge clk) begin

if(rst==1) begin

WB\_Reg\_WriteAddr<=0;

WB\_Reg\_WriteEn<=0;

WB\_Reg\_WriteData<=0;

end

else begin

WB\_Reg\_WriteAddr<=MEM\_Reg\_WriteAddr;

WB\_Reg\_WriteEn<=MEM\_Reg\_WriteEn;

WB\_Reg\_WriteData<=MEM\_Reg\_WriteData;

end

end

endmodule

### 2.4.17 模型机（Pipeline\_Mips.v）

/\*

Author: 张永康

Institution Affiliation: 武汉大学计算机学院

Description: 顶层代码

\*/

`include "bus\_def.v"

`include "ctrl\_encode\_def.v"

module Pipeline\_Mips( );

reg Clk, Reset;

wire [`Word\_Bus] IF\_Instruction;

initial begin

$readmemh( "sort\_test.txt", U\_IMem.IMem ) ;

//$readmemh( "Test\_Signal\_Pipeline.txt", U\_IMem.IMem ) ;

$monitor("PC = 0x%8X, IR = 0x%8X", U\_PcUnit.PC, IF\_Instruction );

Clk = 1 ;

Reset = 0 ;

#5 Reset = 1 ;

#20 Reset = 0 ;

end

always

   #(50) Clk = ~Clk;

wire [`Word\_Bus] PC\_out;

wire [`IMem\_Addr\_Bus] IMem\_Addr;

wire [`Word\_Bus] EX\_Instant,ID\_Instant;

wire Change\_PC\_en;

wire [`Word\_Bus] PC\_New;

wire Stall\_IF\_ID,Stall\_ID\_EX;

PcUnit U\_PcUnit(.PC(PC\_out),

            .PcReSet(Reset),

            .Change\_PC\_en(Change\_PC\_en),

            .Stall(Stall\_IF\_ID|Stall\_ID\_EX), //ID级或EX级要求阻塞时，需要暂停PC计数器

            .Clk(Clk),

            .PC\_New(PC\_New));

assign IMem\_Addr[`IMem\_Addr\_Bus]=PC\_out[11:2];

IMem U\_IMem(.OpCode(IF\_Instruction),.ImAdress(IMem\_Addr));

wire [`Word\_Bus] ID\_Instruction,ID\_PC;

 /\*

    ID和EX级阻塞同时发生的解决办法

    当ID和EX级阻塞同时发生时，优先照顾EX级阻塞的请求，

    即对EX级及其之前的所有单元阻塞一个周期。

    该周期结束后，如果ID级所需的数据仍无法通过旁路得到，

    则会再次触发ID级的阻塞。

    \*/

    IF\_ID U\_IF\_ID(.clk(Clk), //时钟信号

            .rst(Reset|Change\_PC\_en), //重置信号

//ID级或EX级要求阻塞时，需要阻塞IF/ID寄存器

           .Stall(Stall\_IF\_ID|Stall\_ID\_EX),

            .IF\_PC(PC\_out), //IF级的PC值

            .IF\_Instruction(IF\_Instruction),

            .ID\_PC(ID\_PC),

            .ID\_Instruction(ID\_Instruction)); //IF级从指令存储器取出的指令

wire [`Word\_Bus] ID\_Reg1,ID\_Reg2,ID\_Shamt;

wire jump,RegDst,Branch,MemR,Mem2R,MemW,RegW;

wire [`AluSrc\_Bus] AluSrc1,AluSrc2;

wire [4:0] Aluctrl;

wire [`Reg\_Addr\_Bus] Write\_RegAddr;

wire [`Reg\_Addr\_Bus] WB\_Reg\_WriteAddr;

wire WB\_Reg\_WriteEn,WB\_Mem2R;

wire [`Word\_Bus] WB\_Reg\_WriteData;

wire [`Reg\_Addr\_Bus] Rs\_num,Rt\_num,Rd\_num;

wire [`Word\_Bus] MEM\_AluResult;

wire [`Reg\_Addr\_Bus] MEM\_Reg\_WriteAddr;

wire MEM\_Reg\_WriteEn;

wire EX\_RegWrite;

wire [`Reg\_Addr\_Bus] EX\_RegAddr;

wire EX\_MemRead,MEM\_DMem\_ReadEn;

ID U\_ID(

        .clk(Clk),

        .rst(Reset),

        .Instruction(ID\_Instruction),

        .Write\_RegAddr\_From\_WB(WB\_Reg\_WriteAddr), //来自WB级的写寄存器的地址

        .Write\_RegData\_From\_WB(WB\_Reg\_WriteData), //来自WB级的写寄存器的地址

        .WriteEn\_From\_WB(WB\_Reg\_WriteEn), //来自WB级的写寄存器的地址

        .Reg1(ID\_Reg1),

.Reg2(ID\_Reg2),

        .Write\_RegAddr(Write\_RegAddr),

 .Instant(ID\_Instant),

        .Shamt(ID\_Shamt),

        .jump(jump),

        .RegDst(RegDst),

        .Branch(Branch),

        .MemR(MemR),

        .Mem2R(Mem2R),

     .MemW(MemW),

        .RegW(RegW),

        .AluSrc1(AluSrc1),

        .AluSrc2(AluSrc2),

        .Aluctrl(Aluctrl),

        //为支持分支与跳转加入的信号

        .DataSrc\_EX\_MEM(MEM\_AluResult), //EX\_MEM閺冧浇鐭剧紒鎾寸亯

        .DataSrc\_MEM\_WB(WB\_Reg\_WriteData), //MEM\_WB閺冧浇鐭剧紒鎾寸亯

        .ID\_PC(ID\_PC),

        .MEM\_Reg\_WriteEn(MEM\_Reg\_WriteEn),

        .MEM\_Reg\_WriteAddr(MEM\_Reg\_WriteAddr),

        .WB\_Reg\_WriteEn(WB\_Reg\_WriteEn),

        .WB\_Reg\_WriteAddr(WB\_Reg\_WriteAddr),

         .PC\_New(PC\_New),

        .Change\_PC\_en(Change\_PC\_en),

        //End

        //Signal for stall when R instruction & beq together

        .Stall\_IF\_ID(Stall\_IF\_ID),

        .EX\_Reg\_WriteAddr(EX\_RegAddr),

        .EX\_Reg\_WriteEn(EX\_RegWrite),

        .Rs\_num(Rs\_num),

        .Rt\_num(Rt\_num),

        //.Rd\_num(Rd\_num)

        .EX\_MemRead(EX\_MemRead), //EX级指令的存储器读信号

        .MEM\_MemRead(MEM\_DMem\_ReadEn) //MEM级指令的存储器读信号

    );

 wire [`Word\_Bus] EX\_Reg1,EX\_Reg2,EX\_Shamt; //来自ID/EX的Shamt偏移量

//2. Control signal

//--2.1 EX级控制信号

wire [`AluSrc\_Bus] EX\_AluSrc1,EX\_AluSrc2; //ALU源操作数2控制信号

wire [4:0] EX\_AluCtrl; //来自ID/EX的ALU控制信号

//--2.2 MEM级控制信号

wire EX\_MemWrite;

//--2.3 WB级控制信号

//3. Rs,rt,rd寄存器号

wire [`Reg\_Addr\_Bus] EX\_Rs\_num,EX\_Rt\_num,EX\_Rd\_num;

ID\_EX U\_ID\_EX(

        .clk(Clk),

//!!!!注意这里，当ID、EX同时发现要暂停流水线，

//优先照顾EX的请求，ID/EX不要清空

        .rst(Reset|(Stall\_IF\_ID&(~Stall\_ID\_EX))),

        .Stall(Stall\_ID\_EX),

        //Information came from ID stage

        //1. Reg1&2 , intant number, shamt value in ID

        .ID\_Reg1(ID\_Reg1), //ID级取出的寄存器1的值

        .ID\_Reg2(ID\_Reg2), //ID级取出的寄存器2的值

        .ID\_Instant(ID\_Instant), //ID级取出的经扩展得到的立即数的值

        .ID\_Shamt(ID\_Shamt), //来自ID/EX的Shamt偏移量

        //2. Control signal

        //--2.1 EX级控制信号

        .ID\_AluSrc1(AluSrc1), //ALU源操作数1控制信号

        .ID\_AluSrc2(AluSrc2), //ALU源操作数2控制信号

        .ID\_AluCtrl(Aluctrl), //来自ID/EX的ALU控制信号

        //--2.2 MEM级控制信号

        .ID\_MemWrite(MemW),

        .ID\_MemRead(MemR),

        //.ID\_MemAddr(),

        //--2.3 WB级控制信号

        .ID\_RegWrite(RegW),

        .ID\_Mem2R(Mem2R),

        .ID\_RegAddr(Write\_RegAddr),

        //3. RS,RT,RD寄存器号

        .ID\_Rs\_num(Rs\_num),

        .ID\_Rt\_num(Rt\_num),

        //.ID\_Rd\_num(Rd\_num),

        //Information that sends into EX stage

        //1. Reg1&2 , intant number, shamt value in ID

        .EX\_Reg1(EX\_Reg1), //ID级取出的寄存器1的值

        .EX\_Reg2(EX\_Reg2), //ID级取出的寄存器2的值

        .EX\_Instant(EX\_Instant), //ID级取出的经扩展得到的立即数的值

        .EX\_Shamt(EX\_Shamt), //来自ID/EX的Shamt偏移量

        //2. Control signal

        //--2.1 EX级控制信号

        .EX\_AluSrc1(EX\_AluSrc1), //ALU源操作数1控制信号

        .EX\_AluSrc2(EX\_AluSrc2), //ALU源操作数2控制信号

        .EX\_AluCtrl(EX\_AluCtrl), //来自ID/EX的ALU控制信号

        //--2.2 MEM级控制信号

        .EX\_MemWrite(EX\_MemWrite),

        .EX\_MemRead(EX\_MemRead),

        //--2.3 WB级控制信号

        .EX\_RegWrite(EX\_RegWrite),

        .EX\_Mem2R(EX\_Mem2R),

        .EX\_RegAddr(EX\_RegAddr),

        //3.Rs,Rt,Rd寄存器号

        .EX\_Rs\_num(EX\_Rs\_num),

        .EX\_Rt\_num(EX\_Rt\_num)

        //.EX\_Rd\_num(EX\_Rd\_num)

    );

wire [`Word\_Bus] AluResult;

wire Zero;

wire [`Word\_Bus] Bypassed\_DMem\_WriteData;

//wire MEM\_DMem\_ReadEn;

EX U\_EX(.DataSrc\_EX\_MEM(MEM\_AluResult),

     .DataSrc\_MEM\_WB(WB\_Reg\_WriteData),

             .DataSrc\_ID\_EX\_Reg1(EX\_Reg1),

.DataSrc\_ID\_EX\_Reg2(EX\_Reg2),

.DataSrc\_ID\_EX\_Instant(EX\_Instant),

             .DataSrc\_Shamt(EX\_Shamt),

.AluSrc1(EX\_AluSrc1),

.AluSrc2(EX\_AluSrc2),

     .AluCtrl(EX\_AluCtrl),

             //旁路功能所需输入

            .EX\_Rs\_num(EX\_Rs\_num),

             .EX\_Rt\_num(EX\_Rt\_num),

             .MEM\_Reg\_WriteEn(MEM\_Reg\_WriteEn),

             .MEM\_Reg\_WriteAddr(MEM\_Reg\_WriteAddr),

             .WB\_Reg\_WriteEn(WB\_Reg\_WriteEn),

             .WB\_Reg\_WriteAddr(WB\_Reg\_WriteAddr),

     .AluResult(AluResult),

             //.Reg\_Addr,

     .Zero(Zero),

             .DMem\_WriteEn(EX\_MemWrite),

             .Bypassed\_DMem\_WriteData(Bypassed\_DMem\_WriteData),

             //Signal for lw stall

             .MEM\_DMem\_ReadEn(MEM\_DMem\_ReadEn),

             .Stall\_ID\_EX(Stall\_ID\_EX));

    wire [`Word\_Bus] EX\_Reg\_WriteData,MEM\_Reg\_WriteData;

    wire [`DMem\_Addr\_Bus] MEM\_DMem\_WriteAddr;

    wire MEM\_DMem\_WriteEn;

    wire [`Word\_Bus] MEM\_DMem\_WriteData; //,MEM\_AluResult;

    wire [`Reg\_Addr\_Bus] MEM\_Rs\_num,MEM\_Rt\_num;

    EX\_MEM U\_EX\_MEM(  .clk(Clk),

     .rst(Reset|Stall\_ID\_EX),

     .EX\_Reg\_WriteAddr(EX\_RegAddr),

         .EX\_Reg\_WriteEn(EX\_RegWrite),

         .EX\_AluResult(AluResult),

         .EX\_Mem2R(EX\_Mem2R),

     .EX\_DMem\_WriteAddr(AluResult[`DMem\_Addr\_Bus]),

         .EX\_DMem\_WriteEn(EX\_MemWrite),

         .EX\_DMem\_ReadEn(EX\_MemRead),

     .EX\_DMem\_WriteData(Bypassed\_DMem\_WriteData),

         .MEM\_Reg\_WriteAddr(MEM\_Reg\_WriteAddr),

         .MEM\_Reg\_WriteEn(MEM\_Reg\_WriteEn),

         .MEM\_AluResult(MEM\_AluResult),

         .MEM\_Mem2R(MEM\_Mem2R),

         .MEM\_DMem\_WriteAddr(MEM\_DMem\_WriteAddr),

         .MEM\_DMem\_WriteEn(MEM\_DMem\_WriteEn),

         .MEM\_DMem\_ReadEn(MEM\_DMem\_ReadEn),

         .MEM\_DMem\_WriteData(MEM\_DMem\_WriteData)

     );

wire [`Word\_Bus] MEM\_DataOut;

assign MEM\_Reg\_WriteData=(MEM\_Mem2R==`Mem2R\_From\_Alu)?

MEM\_AluResult:MEM\_DataOut;

MEM U\_MEM(.MemWrite(MEM\_DMem\_WriteEn),

.MemRead(MEM\_DMem\_ReadEn),

.clk(Clk),

.DataAddr(MEM\_DMem\_WriteAddr),

.DataIn(MEM\_DMem\_WriteData),

             //Output

.DataOut(MEM\_DataOut));

 MEM\_WB U\_MEM\_WB(  .clk(Clk),

         .rst(Reset),

         .MEM\_Reg\_WriteAddr(MEM\_Reg\_WriteAddr),

         .MEM\_Reg\_WriteEn(MEM\_Reg\_WriteEn),

         .MEM\_Reg\_WriteData(MEM\_Reg\_WriteData),

         .WB\_Reg\_WriteAddr(WB\_Reg\_WriteAddr),

         .WB\_Reg\_WriteEn(WB\_Reg\_WriteEn),

         .WB\_Reg\_WriteData(WB\_Reg\_WriteData)

     );

endmodule

## 2.5 设计结果分析

### 2.5.1测试文件

lui $1, 0x1000

ori $1, 0x1008 # $1 = 0x10001008

sw $1, 0($0)

lui $1, 0x1000

ori $1, 0x1002 # $1 = 0x10001002

sw $1, 4($0)

lui $1, 0x8000

ori $1, 0x1001 # $1 = 0x80001001

sw $1, 8($0)

lui $1, 0x1000

ori $1, 0x1005 # $1 = 0x10001005

sw $1, 12($0)

lui $1, 0x8000

ori $1, 0x1000 # $1 = 0x80001000

sw $1, 16($0)

ori $1, $0, 0 # $1 = 0

ori $7, $0, 16

ori $5, $0, 1

ori $6, $0, 5

sort\_f:

ori $1, $0, 0

addi $6, $6, -1

sort:

lw $2, 0($1)

addi $1, $1, 4

lw $3, 0($1)

slt $4, $3, $2 # if $2 > $3, $4 = 1

beq $4, $5, swap # so swap($2, $3)

return:

beq $6, $0, exit

beq $1, $7, sort\_f

beq $0,$0, sort

swap:

addi $1, $1, -4

sw $3, 0($1)

addi $1, $1, 4

sw $2, 0($1)

beq $6, $0, exit

beq $0,$0, return

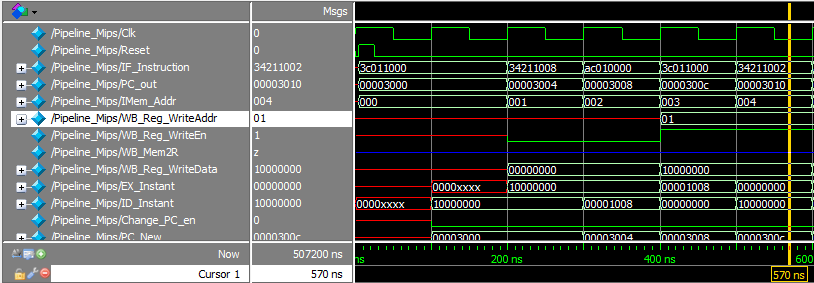
exit:

### 2.5.2 测试结果分析

#### 2.5.2.1 lui $1, 0x1000 指令

该指令的仿真结果如图2.5所示，该指令表示将常数0x1000加载到$1号寄存器高16位。仿真结果表明，该指令所在的PC寄存器地址为00003000（十六进制），该指令的机器码为3c011000（十六进制），rt寄存器编号为1，寄存器写入的数据是

0x10000000，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.5 lui $1, 0x1000 指令仿真结果

#### 2.5.2.2 ori $1, $1, 0x00001008 指令

该指令的仿真结果如图2.6所示，该指令表示将$1寄存器与常数0x0001008进行“或”运算，结果放入$1号寄存器。

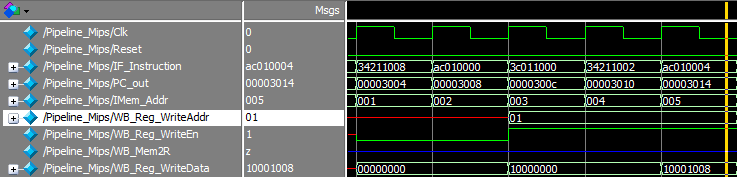
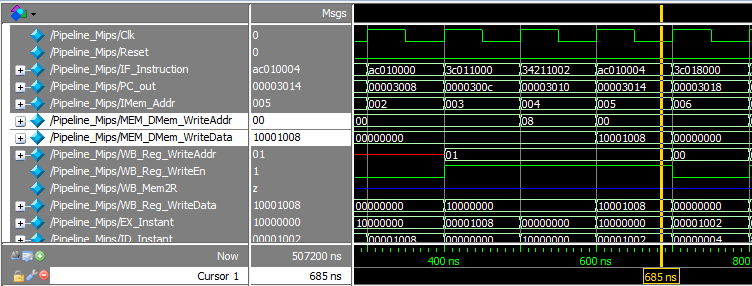


图2.6 ori $1, $1, 0x00001008 指令仿真结果

仿真结果表明，该指令所在的PC寄存器地址为00003004（十六进制），该指令的机器码为34211008（十六进制），rs寄存器编号为1，rt寄存器编号为1，寄存器写入的数据是0x0001008，与指令的预期结果是一致的，说明该指令仿真结果正确。

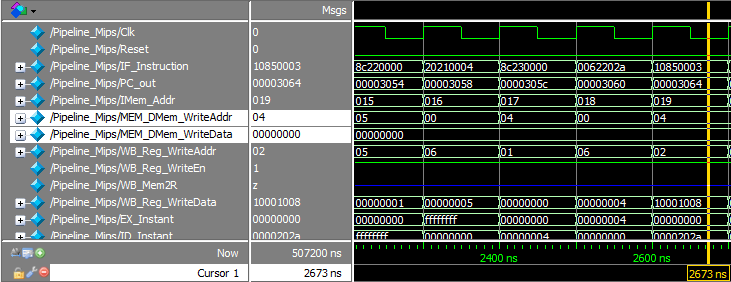
#### 2.5.2.3 sw $1, 0($0)指令

该指令的仿真结果如图2.7所示，该指令表示将$1寄存器存入$0+0的内存单元中，$0寄存器的值总是为0，因此计算的地址是0，即：将$1寄存器存入0号内存单元。仿真结果表明，该指令所在的PC寄存器地址为00003008（十六进制），该指令的机器码为ac01000（十六进制），rs寄存器编号为0，rt寄存器编号为1，应写入存储器的数据为10001008（十六进制），也就是前面指令执行后$1寄存器的内容，存储器的写入地址为0，写入的数据为10001008（见3.1CPU总体结构图中的连线），最终DMem[0]的值为10001008（十六进制），与指令的预期结果是一致的，说明该指令仿真结果正确。

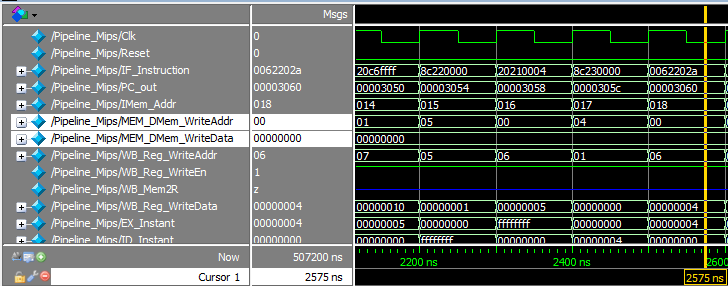
图2.7 sw $1, 0($0)指令仿真结果

#### 2.5.2.4 addi $6,$6,-1 指令

该指令的仿真结果如图2.8所示。

图2.9 lw $2,0($1) 指令仿真结果

该指令表示将寄存器$6和常数-1相加，结果写入$6号寄存器。仿真结果表明，该指令所在的PC寄存器地址为00003050（十六进制），该指令的机器码为20c6ffff（十六进制），rt寄存器编号为6，寄存器写入的数据是0x00000004，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.8 addi $6,$6,-1 指令仿真结果

#### 2.5.2.5 lw $2,0($1) 指令

该指令的仿真结果如图2.9所示，该指令表示以寄存器$1的值和0相加的结果作为基地址，取出1个字的数据，写入$2号寄存器。仿真结果表明，该指令所在的PC寄存器地址为00003054（十六进制），该指令的机器码为8c220000（十六进制），rt寄存器编号为2，存储器地址$1+0处存的数据是0x10001008，寄存器写入的数据是0x10001008，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.9 lw $2,0($1) 指令仿真结果

#### 2.5.2.6 slt $4,$3,$2 指令

该指令的仿真结果如图2.10所示，该指令表示将寄存器$3和$2的值做比较，若$3号寄存器的值更小，则置$4号寄存器的值为1，否则置其为0。仿真结果表明，该指令所在的PC寄存器地址为00003060（十六进制），该指令的机器码为0062202a（十六进制），rs寄存器编号为3，rt寄存器编号为2，rd寄存器编号为4，$3寄存器的值为0x10001002，$2寄存器的值为0x10001008，写入$4的数据是0x00000001，与指令的预期结果是一致的，说明该指令仿真结果正确。

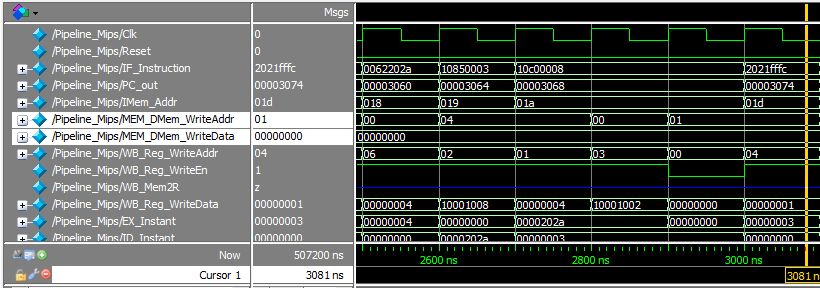
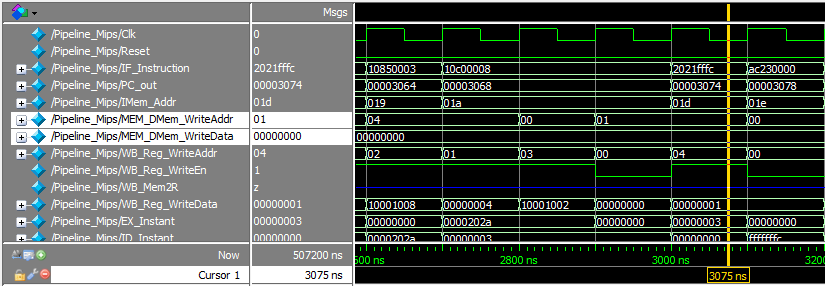


图2.10 slt $4,$3,$2 指令仿真结果

#### 2.5.2.7 beq $4, $5, swap指令

该指令的仿真结果如图2.11所示，该指令表示将$4寄存器与$5寄存器进行“减”运算，结果不保存，通过置的zero标记来决定是否转移。如果zero=1则表示$4寄存器与$5寄存器数据相等，转移到swap标号，否则就不转移。仿真结果表明，该指令所在的PC寄存器地址为00003064（十六进制），该指令的机器码为10850003（十六进制），rs寄存器编号为4，rt寄存器编号为5，寄存器$4的输出值为0x00000001，寄存器$5的输出值也为0x00000001，两者相等，根据测试文件程序应该转移到下面的addi $1,$1,-4指令，其指令地址为0x00003074，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.11 beq $4, $5, swap指令仿真结果

#### 2.5.2.8 EX级旁路

情况1：只存在EX/MEM→EX旁路，如图2.12所示。

图2.12 只存在EX/MEM→EX旁路

以上代码的仿真结果如图2.13所示。

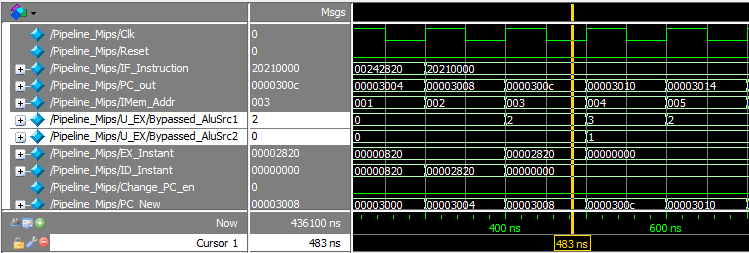
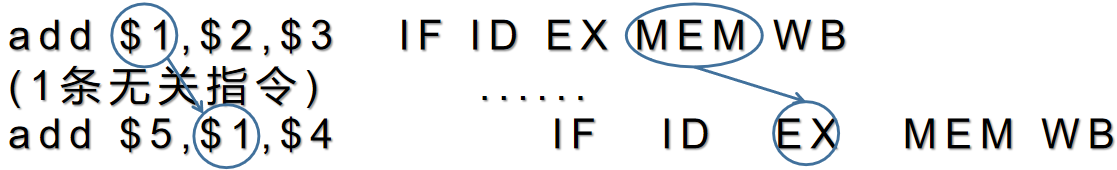


图2.13 EX级旁路仿真结果1

其中第二条指令表示将$1号和$4号寄存器相加，结果写入$5寄存器。仿真结果表明，该指令的机器码为00242820（十六进制），AluSrc1经旁路后的数据来源为2(EX/MEM)，AluSrc2经旁路后的数据来源为0(ID/EX)，与指令的预期结果是一致的，说明该指令仿真结果正确。

情况2：只存在MEM/WB→EX旁路如图2.14所示。

图2.14 只存在MEM/WB→EX旁路

执行以下指令：

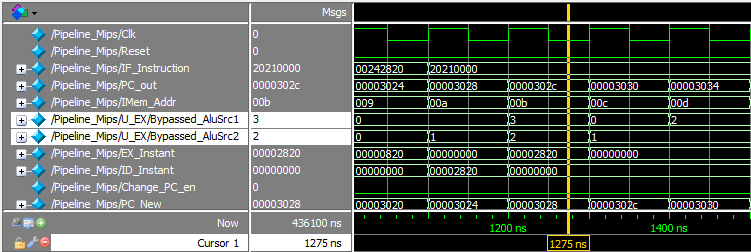
add $1,$2,$3

addi $4,$4,0

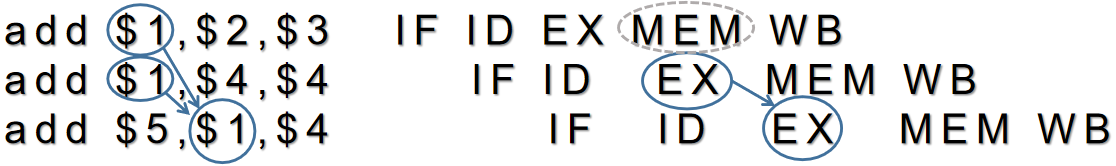
add $5,$1,$4

在第三条指令时，AluSrc1来自MEM/WB，AluSrc2来自EX/MEM。

以上代码的仿真结果如图2.15所示，其中第三条指令表示将$1号和$4号寄存器相加，结果写入$5寄存器。仿真结果表明，该指令的机器码为00242820（十六进制），AluSrc1经旁路后的数据来源为3(MEM/WB)，AluSrc2经旁路后的数据来源为2(EX/MEM)，与指令的预期结果是一致的，说明该指令仿真结果正确。

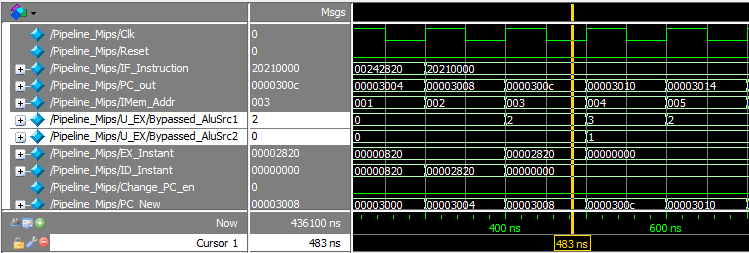
图2.15 EX级旁路仿真结果2

情况3：同时存在EX/MEM→EX旁路、MEM/WB→EX旁路（如图2.16所示），优先使用EX/MEM的旁路结果

图2.16 同时存在EX/MEM→EX旁路、MEM/WB→EX旁路

当来自EX/MEM,MEM/WB的旁路数据同时有效时，应优先使用EX/MEM级的数据。

以上代码的仿真结果如图2.17所示，其中第三条指令表示将$1号和$4号寄存器相加，结果写入$5寄存器。仿真结果表明，该指令的机器码为00242820（十六进制），AluSrc1经旁路后的数据来源为2(EX/MEM)，AluSrc2经旁路后的数据来源为0(ID/EX)，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.17 EX级旁路仿真结果3

#### 2.5.2.9 ID级旁路

由于beq,bne指令的判定分支是在ID级实现的，因此类似于EX级，ID级也需要加入旁路解决数据相关问题。

当来自EX,MEM,WB级的旁路数据中有多个同时有效时，应优先使用最早的数据，换言之，三种旁路数据的优先级为: EX>MEM>WB。

由于ID级旁路与EX级旁路高度相似，因此这里不再对其测试结果赘述。

#### 2.5.2.10 EX级阻塞控制

当前一条数据为lw类型且发生先写后读(Read after write)类型数据相关时，需要暂停EX及其之前的所有单元一个周期。

阻塞发生条件：当MEM级的存储器读信号为1，对应指令的写入寄存器号不为0，且等于EX级指令的Rs或Rt时，则暂停。

例如以下代码：

lw $1,0($4)

add $2,$1,$2

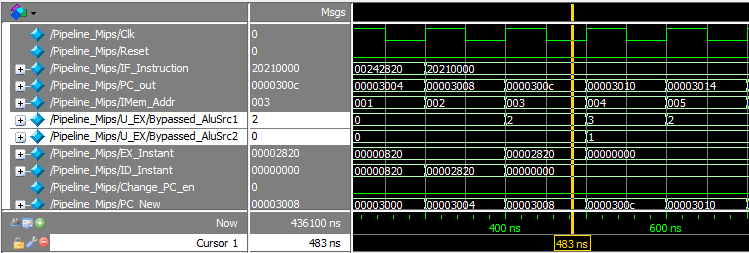
以上代码的仿真结果如图2.18所示，其中第二条指令表示将$1号和$2号寄存器相加，将结果写入$2寄存器，由于其源操作数rs($1)和第一条lw指令的rt寄存器($1)相同，因此需要在执行到EX级时暂停一周期。仿真结果表明，add指令执行到EX阶段时Stall ID/EX信号为1，与预期结果是一致的，说明该指令仿真结果正确。

图2.18 仿真结果

#### 2.5.2.11 ID级阻塞控制

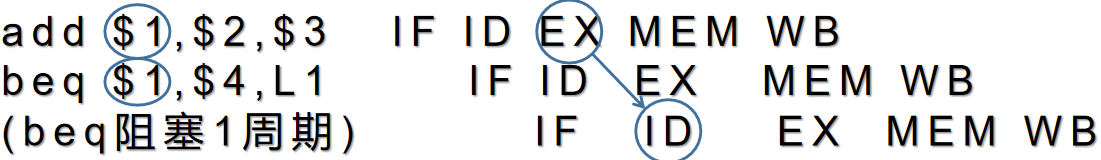
由于beq,bne指令的判定分支是在ID级实现的，因此类似于EX级，ID级也需要加入因数据相关引发的阻塞。当前一条数据为lw/R/其他要写回寄存器的指令类型且发生先写后读(Read after write)类型数据相关时，需要暂停ID及其之前的所有单元一个周期。

阻塞发生条件：

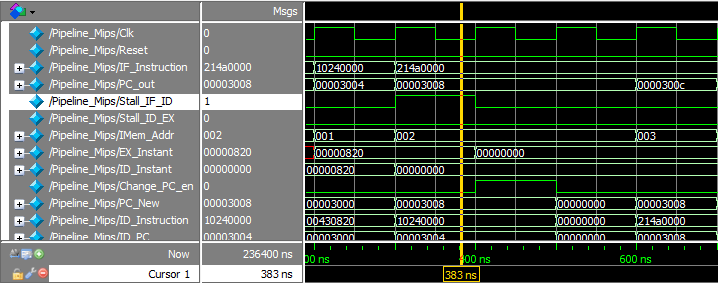
①当MEM级的存储器读信号为1，且旁路数据来源于EX/MEM。

②当EX级的寄存器写回信号为1，且旁路数据来源于ID/EX。

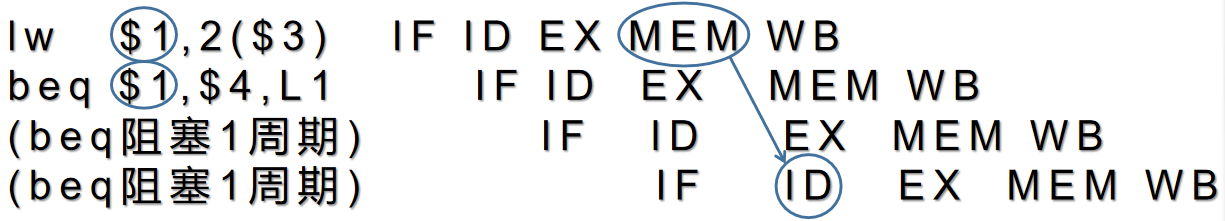
情况1：beq/bne前是一个R型/其他要写回寄存器的指令类型，阻塞1周期如图2.19所示。

图2.19 阻塞1周期

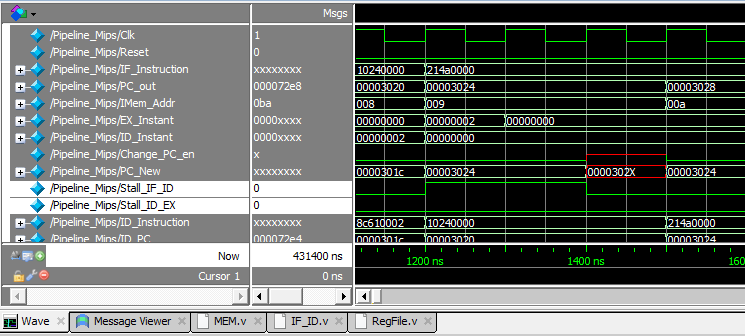
以上代码的仿真结果如图2.20所示，其中第二条指令表示将$1号和$4号寄存器做比较，若相等则跳转到L1。仿真结果表明，该指令的机器码为10240000（十六进制），执行到ID阶段时Stall IF/ID信号为1，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.20 阻塞控制仿真结果1

情况2：beq/bne前是一个lw指令，阻塞2周期，如图2.21所示。

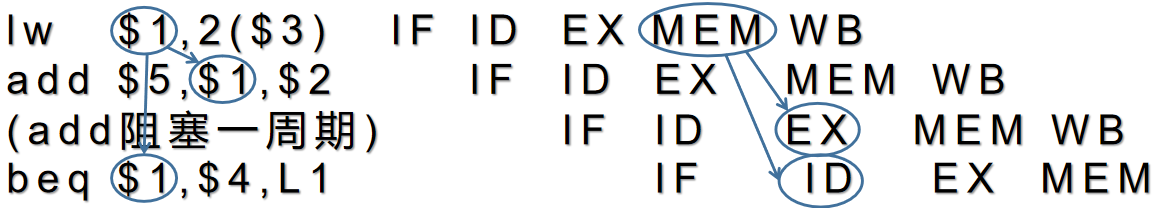
图2.21 阻塞2周期

以上代码的仿真结果如图2.22所示，其中第二条指令表示将$1号和$4号寄存器做比较，若相等则跳转到L1。仿真结果表明，该指令的机器码为10240000（十六进制），执行到ID阶段时Stall IF/ID信号为1，并持续两周期，表明beq指令在ID阶段阻塞了两个周期，与指令的预期结果是一致的，说明该指令仿真结果正确。

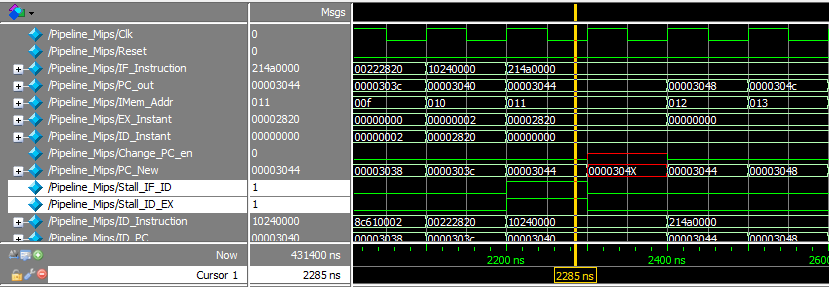
图2.22 阻塞控制仿真结果2

#### 2.5.2.12 ID级、EX级同时阻塞的控制

当ID和EX级阻塞同时发生时，优先照顾EX级阻塞的请求，即对EX级及其之前的所有单元阻塞一个周期。该周期结束后，如果ID级所需的数据仍无法通过旁路得到，则会再次触发ID级的阻塞。如图2.23所示，时刻4时，ID和EX级阻塞同发生。

图2.23 ID级、EX级同时阻塞

以上代码的仿真结果如图2.24所示，其中第二条指令表示将$1号和$5号寄存器相加，结果存入$5寄存器；第三条指令表示将$1号和$4号寄存器做比较，若相等则跳转到L1。仿真结果表明，add指令执行到EX阶段时Stall IF/ID,Stall ID/EX信号均为1，只持续一周期，表明add指令在EX阶段阻塞了一个周期，与指令的预期结果是一致的，说明该指令仿真结果正确。

图2.24 ID级、EX级同时阻塞控制仿真结果图

### 2.5.3 FPGA开发板测试

Nexys 4 DDR是一款Digilent多孔RAM-based Nexys开发板的简易替代品（如图2.24所示）。搭载Xilinx®Artix™-7 FPGA芯片，Nexys 4 DDR是一个打开即用型的数字电路开发平台，帮助使用者能够在课堂环境下实现诸多工业领域的应用。相比早期版本，经优化后的Artix-7 FPGA芯片能够实现更高性能的逻辑，并且能提供更多的容量，更好的性能以及更丰富的资源。配有高容量的大型FPGA芯片（Xilinx产品编号XC7A100T- 1CSG324C）并集成了USB，以太网和其它端口，Nexys 4 DDR开发板能实现从理论型组合电路到强大的嵌入式处理器的多种设计。

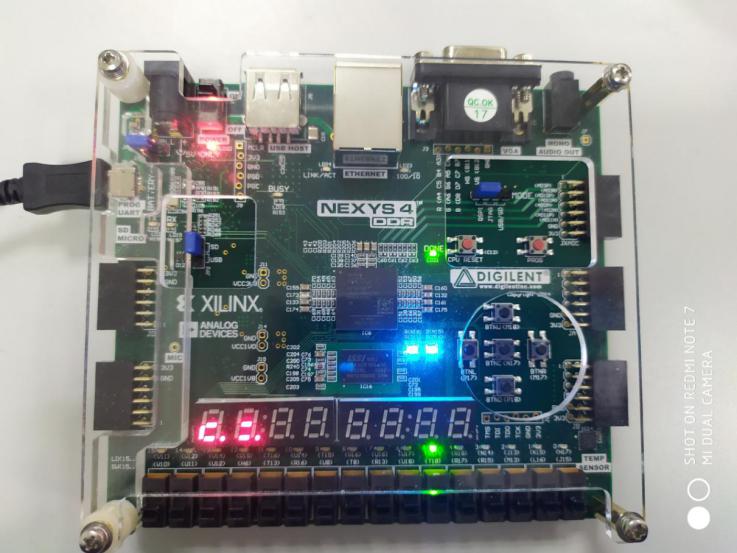
****

图2.25 Nexys 4 DDR开发板

在MEM级加入10个开关，用于输入要显示数据的地址，以及一个数码管译码器，用于输出对应地址单元的存储器数据。

在EX级加入一个数码管译码器，用于输出ALU运算结果。

使用开关SW 14，选择是显示ALU运算结果还是存储器数据。在运行程序时，输出ALU运算结果；在程序运行完成后，拨动该开关，从而可以观察存储器中的数据。

在顶层文件中加入以下代码，以显示运行过程中ALU的运算结果：

wire [7:0] disp\_seg\_o\_ALU,disp\_an\_o\_ALU;

seg7x16 U\_seg7x16(.clk(Clk\_init),

.reset(Reset),

.i\_data(AluResult),

.o\_seg(disp\_seg\_o\_ALU ),

.o\_sel(disp\_an\_o\_ALU ));

在MEM级代码中加入以下代码：

assign disp\_data=DMem[sw\_i[9:0]];

//The unit used for presenting result(write back to reg or save into memory)

// in digital tubes

seg7x16 U\_seg7x16(.clk(Clk\_init),

.reset(Reset),

.i\_data(disp\_data),

.o\_seg(disp\_seg\_o ),

.o\_sel(disp\_an\_o ));

在顶层文件中加入以下代码，以使用开关选择是显示ALU运算结果还是存储器数据：

assign disp\_seg\_o=(sw\_i[14]==1)?disp\_seg\_o\_ALU:disp\_seg\_o\_MEM;

assign disp\_an\_o=(sw\_i[14]==1)?disp\_an\_o\_ALU:disp\_an\_o\_MEM;

使用以下代码作为数码管译码器：

//`timescale 1ns / 1ps

module seg7x16(

input clk, //输入的高频时钟信号

input reset, //数码管复位信号

input [31:0] i\_data, //要输出的32位数据

output [7:0] o\_seg, //数码管输出控制信号

output [7:0] o\_sel //数码管位码，其中只有一位为0，代表当前被点亮的数码管

);

reg [14:0] cnt;

always @ (posedge clk, posedge reset)

if (reset)

cnt <= 0;

else

cnt <= cnt + 1'b1;

wire seg7\_clk = cnt[14];

reg [2:0] seg7\_addr;

always @(posedge seg7\_clk, posedge reset)

if(reset)

     seg7\_addr <= 0;

else

     seg7\_addr <= seg7\_addr + 1'b1;

reg [7:0] o\_sel\_r;

always @ (\*)

   case(seg7\_addr)

     7 : o\_sel\_r = 8'b01111111;

     6 : o\_sel\_r = 8'b10111111;

     5 : o\_sel\_r = 8'b11011111;

     4 : o\_sel\_r = 8'b11101111;

     3 : o\_sel\_r = 8'b11110111;

     2 : o\_sel\_r = 8'b11111011;

     1 : o\_sel\_r = 8'b11111101;

     0 : o\_sel\_r = 8'b11111110;

    endcase

   reg [31:0] i\_data\_store;

   always @ (posedge clk, posedge reset)

   if(reset)

     i\_data\_store <= 0;

     else

     i\_data\_store <= i\_data;

    reg [7:0] seg\_data\_r;

   always @ (\*)

   case(seg7\_addr)

     0 : seg\_data\_r = i\_data\_store[3:0];

     1 : seg\_data\_r = i\_data\_store[7:4];

     2 : seg\_data\_r = i\_data\_store[11:8];

     3 : seg\_data\_r = i\_data\_store[15:12];

     4 : seg\_data\_r = i\_data\_store[19:16];

     5 : seg\_data\_r = i\_data\_store[23:20];

     6 : seg\_data\_r = i\_data\_store[27:24];

     7 : seg\_data\_r = i\_data\_store[31:28];

    endcase

   reg [7:0] o\_seg\_r;

   always @ (posedge clk, posedge reset)

   if(reset)

     o\_seg\_r <= 8'hff;

     else

     case(seg\_data\_r)

     4'h0 : o\_seg\_r <= 8'hC0;

4'h1 : o\_seg\_r <= 8'hF9;

4'h2 : o\_seg\_r <= 8'hA4;

4'h3 : o\_seg\_r <= 8'hB0;

4'h4 : o\_seg\_r <= 8'h99;

4'h5 : o\_seg\_r <= 8'h92;

4'h6 : o\_seg\_r <= 8'h82;

4'h7 : o\_seg\_r <= 8'hF8;

4'h8 : o\_seg\_r <= 8'h80;

4'h9 : o\_seg\_r <= 8'h90;

4'hA : o\_seg\_r <= 8'h88;

4'hB : o\_seg\_r <= 8'h83;

4'hC : o\_seg\_r <= 8'hC6;

4'hD : o\_seg\_r <= 8'hA1;

4'hE : o\_seg\_r <= 8'h86;

4'hF : o\_seg\_r <= 8'h8E;

     endcase

   assign o\_sel = o\_sel\_r;

   assign o\_seg = o\_seg\_r;

endmodule

如图2.26所示，程序运行完成后，拨动开关SW14至显示存储器内容模式，拨动SW9~SW0选择存储器单元地址，依次显示地址0、4、8、12、16的数据为80001008，80001001，10001002，10001005，10001008，与预期结果完全一致，表明流水线模型工作正常。

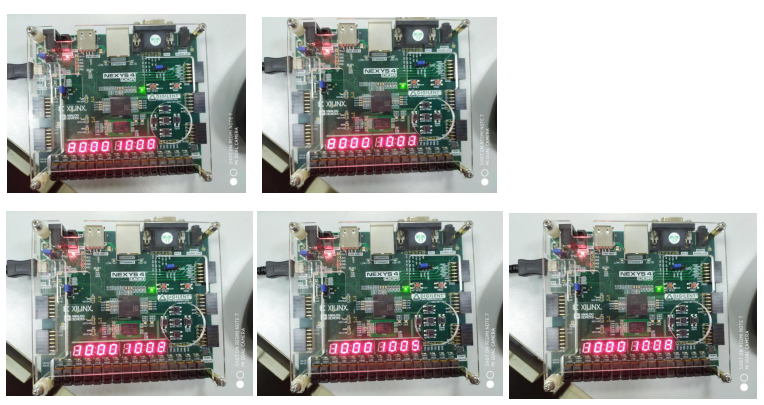


图2.26 7段代码显示结果

# 总 结

本实验主要基于MIPS架构的流水线微处理器设计，并通过对流水线中各阶段的主要工作的分析进而更加详细的划分出每个阶段所要涉及到的模块。本实验主要完成了以下几个方面的工作：

(1)详细分析了MIPS架构的指令集，并对各类指令的编码方式和功能进行了分析，为处理器的设计打下基础。

(2)认真分析了流水线的结构划分，以及这些模块所要实现的具体具体功能。流水线的设计能提高指令的

(3)针对流水线处理器执行过程中的数据相关、控制冒险问题给出了合理的解决办法。

(4)使用Modelsim仿真软件对所设计的流水线进行功能仿真。更加深刻地掌握了MIPS架构流水线处理器的技术细节。

(5)在Xilinx FPGA开发板上下载测试，验证流水线模型的正确性。

众所周知，计算机组成原理是计算机科学与技术专业最重要、最核心的课程，是区分计算机科班与非科班学生的分水岭。受教学模式所限，国内的同学对该课程的理解基本上仅停留于应试水平，对计算机系统的实现缺乏深入理解。

与其他课程的实验课相比，这门实验课显然难度更高、挑战更多，然而，实验过程中的重重困难却激发起了我的浓厚兴趣。

在此衷心感谢计算机组成原理教学组的各位老师，打破常规，将《计算机组成与设计：硬件/软件接口》这本国外的”圣经”级教材引入了我们的课堂，并花费巨资购买大量Xilinx FPGA开发板供我们做实验，通过实战检验自己对计组课程的掌握程度。

受时间所限，本实验中，我只实现了基本版的流水线MIPS处理器，没有支持乘法、除法指令，以及分支预测、中断和异常机制。

日后有空，我还会深入阅读计组教材的升级版《计算机体系结构：量化研究方法》，研究现代处理器技术，并尝试实现支持多发射、乱序执行、SIMD(数据级并行)、和动态分支预测的处理器。

# 参考文献

1. 张冬冬，王力生，郭玉臣.数字逻辑与组成原理实践教程[M]. 2018年第1版.北京:清华大学出版社，2018年：起止页码.
2. （美）帕特森等著.计算机组成与设计[M]，北京：机械工业出版社，2015年：起止页码.
3. 雷思磊. 自己动手写CPU[M]， 北京:电子工业出版社，2014年：

# 教师评语评分

评语：

评分：

评阅人：

年 月 日