**武汉大学计算机学院**

**本科生实验报告**

**计算机组成原理实验**

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 ：冯晶

学 生 学 号 ：

学 生 姓 名 ：

二○二一年五月

**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掌握CPU运算的基本流程与实现方法，熟悉流水线在CPU设计中的作用，了解数据冒险、控制冒险的解决方案。

实验设计主要遵循CPU数据通路设计原理。

实验内容主要包括：设计一个单周期CPU和一个流水线CPU。

对单周期CPU进行指令扩展，至少支持以下指令：

• add/sub/and/or/slt/sltu/addu/subu

• addi/ori/lw/sw/beq

• j/jal

• sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr

用流水线方法实现流水线CPU，并解决数据冒险和控制冒险的问题。

用ModelSim软件对实现的CPU进行测试。

实验结论为通过本次实验，我充分了解了单周期CPU和流水线CPU的工作原理与实现方法。

**关键词：**单周期CPU；流水线CPU

目录

[1 实验目的和意义 1](#_Toc71217399)

[1.1 实验目的 1](#_Toc71217400)

[1.2 实验意义 1](#_Toc71217401)

[2 实验环境介绍 2](#_Toc71217402)

[2.1 Verilog HDL 2](#_Toc71217403)

[2.2 MARS 2](#_Toc71217404)

[2.3 ModelSim 2](#_Toc71217405)

[2.4 Vivado 3](#_Toc71217406)

[2.5 Nexys4 DDR 3](#_Toc71217407)

[3 单周期CPU概要设计 4](#_Toc71217408)

[3.1 总体设计 4](#_Toc71217409)

[3.2 PC（程序计数器） 4](#_Toc71217410)

[3.2.1 功能描述 4](#_Toc71217411)

[3.2.2 模块接口 4](#_Toc71217412)

[3.3 IM（指令存储器） 4](#_Toc71217413)

[3.3.1 功能描述 4](#_Toc71217414)

[3.3.2 模块接口 4](#_Toc71217415)

[3.4 RF（寄存器文件） 5](#_Toc71217416)

[3.4.1 功能描述 5](#_Toc71217417)

[3.4.2 模块接口 5](#_Toc71217418)

[3.5 ALU（运算器） 5](#_Toc71217419)

[3.5.1 功能描述 5](#_Toc71217420)

[3.5.2 模块接口 5](#_Toc71217421)

[3.6 NPC（PC计算器） 6](#_Toc71217422)

[3.6.1 功能描述 6](#_Toc71217423)

[3.6.2 模块接口 6](#_Toc71217424)

[3.7 EXT（数据扩展单元） 6](#_Toc71217425)

[3.7.1 功能描述 6](#_Toc71217426)

[3.7.2 模块接口 6](#_Toc71217427)

[3.8 DM（数据存储器） 7](#_Toc71217428)

[3.8.1 功能描述 7](#_Toc71217429)

[3.8.2 模块接口 7](#_Toc71217430)

[3.9 Ctrl（寄存器文件） 7](#_Toc71217431)

[3.9.1 功能描述 7](#_Toc71217432)

[3.9.2 模块接口 7](#_Toc71217433)

[3.10 MUX（多路选择器） 8](#_Toc71217434)

[3.10.1 功能描述 8](#_Toc71217435)

[3.10.2 模块接口 8](#_Toc71217436)

[3.11 SCCPU（单周期CPU核心模块） 8](#_Toc71217437)

[3.11.1 功能描述 8](#_Toc71217438)

[3.11.2 模块接口 8](#_Toc71217439)

[3.12 SCCOMP（顶层模块） 9](#_Toc71217440)

[3.12.1 功能描述 9](#_Toc71217441)

[3.12.2 模块接口 9](#_Toc71217442)

[3.13 ctrl\_encode\_def（宏定义模块） 9](#_Toc71217443)

[4 单周期CPU详细设计 10](#_Toc71217444)

[4.1 CPU总体结构 10](#_Toc71217445)

[4.2 PC（程序计数器） 10](#_Toc71217446)

[4.3 IM（指令存储器） 11](#_Toc71217447)

[4.4 RF（寄存器文件） 11](#_Toc71217448)

[4.5 ALU（运算器） 12](#_Toc71217449)

[4.6 NPC（PC计算器） 14](#_Toc71217450)

[4.7 EXT（数据扩展单元） 15](#_Toc71217451)

[4.8 DM（数据存储器） 15](#_Toc71217452)

[4.9 Ctrl（控制器） 16](#_Toc71217453)

[4.10 MUX（多路选择器） 20](#_Toc71217454)

[4.11 SCCPU（单周期CPU核心模块） 24](#_Toc71217455)

[4.12 SCCOMP（顶层模块） 28](#_Toc71217456)

[4.13 ctrl\_encode\_def（宏定义模块） 29](#_Toc71217457)

[5 流水线CPU概要设计 31](#_Toc71217458)

[5.1 总体设计 31](#_Toc71217459)

[5.2 PC（程序计数器） 31](#_Toc71217460)

[5.2.1 功能描述 31](#_Toc71217461)

[5.2.2 模块接口 31](#_Toc71217462)

[5.3 IM（指令存储器） 31](#_Toc71217463)

[5.3.1 功能描述 31](#_Toc71217464)

[5.3.2 模块接口 32](#_Toc71217465)

[5.4 RF（寄存器文件） 32](#_Toc71217466)

[5.4.1 功能描述 32](#_Toc71217467)

[5.4.2 模块接口 32](#_Toc71217468)

[5.5 ALU（运算器） 33](#_Toc71217469)

[5.5.1 功能描述 33](#_Toc71217470)

[5.5.2 模块接口 33](#_Toc71217471)

[5.6 EXT（数据扩展单元） 33](#_Toc71217472)

[5.6.1 功能描述 33](#_Toc71217473)

[5.6.2 模块接口 33](#_Toc71217474)

[5.7 DM（数据存储器） 33](#_Toc71217475)

[5.7.1 功能描述 33](#_Toc71217476)

[5.7.2 模块接口 34](#_Toc71217477)

[5.8 Ctrl（寄存器文件） 34](#_Toc71217478)

[5.8.1 功能描述 34](#_Toc71217479)

[5.8.2 模块接口 34](#_Toc71217480)

[5.9 ID（IF/ID段间寄存器） 35](#_Toc71217481)

[5.9.1 功能描述 35](#_Toc71217482)

[5.9.2 模块接口 35](#_Toc71217483)

[5.10 EXE（ID/EX段间寄存器） 36](#_Toc71217484)

[5.10.1 功能描述 36](#_Toc71217485)

[5.10.2 模块接口 36](#_Toc71217486)

[5.11 MEM（EX/MEM段间寄存器） 37](#_Toc71217487)

[5.11.1 功能描述 37](#_Toc71217488)

[5.11.2 模块接口 37](#_Toc71217489)

[5.12 WB（MEM/WB段间寄存器） 38](#_Toc71217490)

[5.12.1 功能描述 38](#_Toc71217491)

[5.12.2 模块接口 38](#_Toc71217492)

[5.13 MUX（多路选择器） 38](#_Toc71217493)

[5.13.1 功能描述 38](#_Toc71217494)

[5.13.2 模块接口 38](#_Toc71217495)

[5.14 SCCPU（流水线CPU核心模块） 39](#_Toc71217496)

[5.14.1 功能描述 39](#_Toc71217497)

[5.14.2 模块接口 39](#_Toc71217498)

[5.15 SCCOMP（顶层模块） 39](#_Toc71217499)

[5.15.1 功能描述 39](#_Toc71217500)

[5.15.2 模块接口 39](#_Toc71217501)

[5.16 ctrl\_encode\_def（宏定义模块） 40](#_Toc71217502)

[6 流水线CPU详细设计 41](#_Toc71217503)

[6.1 CPU总体结构 41](#_Toc71217504)

[6.2 PC（程序计数器） 41](#_Toc71217505)

[6.3 IM（指令存储器） 42](#_Toc71217506)

[6.4 RF（寄存器文件） 42](#_Toc71217507)

[6.5 ALU（运算器） 43](#_Toc71217508)

[6.6 EXT（数据扩展单元） 44](#_Toc71217509)

[6.7 DM（数据存储器） 45](#_Toc71217510)

[6.8 Ctrl（控制器） 46](#_Toc71217511)

[6.9 MUX（多路选择器） 52](#_Toc71217512)

[6.10 ID（IF/ID段间寄存器） 56](#_Toc71217513)

[6.11 EXE（ID/EX段间寄存器） 57](#_Toc71217514)

[6.12 MEM（EX/MEM段间寄存器） 58](#_Toc71217515)

[6.13 WB（MEM/WB段间寄存器） 60](#_Toc71217516)

[6.14 SCCPU（流水线CPU核心模块） 61](#_Toc71217517)

[6.15 SCCOMP（顶层模块） 66](#_Toc71217518)

[6.16 ctrl\_encode\_def（宏定义模块） 68](#_Toc71217519)

[7 测试及结果分析 70](#_Toc71217520)

[7.1 仿真代码及分析 70](#_Toc71217521)

[7.2 仿真测试结果 71](#_Toc71217522)

[7.3 下载测试代码及分析 100](#_Toc71217523)

[7.4 下载测试结果 102](#_Toc71217524)

[8 实验心得 103](#_Toc71217525)

[参考文献 104](#_Toc71217526)

[教师评语评分 105](#_Toc71217527)

# 1 实验目的和意义

## 1.1 实验目的

（1）理解CPU的工作原理；

（2）理解流水线方法的工作原理；

（3）掌握解决数据冒险和控制冒险的方法；

（4）掌握数字信号仿真方法和Nexys4 DDR开发板的使用。

## 1.2 实验意义

（1）深刻了解计算机内部CPU的组成和原理；

（2）熟练使用Verilog HDL语言；

（3）学会使用ModelSim软件进行仿真；

（4）了解程序如何控制开发板工作。

# **2 实验环境介绍**

## 2.1 Verilog HDL

Verilog HDL是一种硬件描述语言，以文本形式来描述数字系统硬件的结构和行为。用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

## 2.2 MARS

能够将汇编代码编译为机器代码。

## 2.3 ModelSim

Mentor公司的ModelSim是业界最优秀的HDL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，具有如下主要特点。

（1）RTL级和门级优化，本地编译结构，编译仿真速度快。

（2）单内核VHDL和Verilog混合仿真。

（3）源代码模板和助手，项目管理。

（4）集成了性能分析、波形比较、代码覆盖等功能。

（5）数据流ChaseX。

（6）Signal Spy。

（7）C和Tcl/Tk接口，C调试。

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了强有力的手段，是FPGA/ASIC设计的首选仿真软件。ModelSim具有多个版本。首先是大的版本，从ModelSim4.7开始，最新版为10.5。在大版本的基础上还有小版本，小版本以小写英文字母作为主要区分，例如，ModelSim10.4版就有10.a、10.b、10.c几个不同的版本。除去大版本和小版本，还有SE、DE、PE三个不同的版本。

## 2.4 Vivado

Xilinx公司于2012年发布了新一代的Vivado设计套件，改变了传统的设计环境和设计方法，打造了一个最先进的设计实现流程，可以让用户更快地实现设计收敛。Vivado设计套件不仅包含传统上寄存器传输级（RTL）到比特流的FPGA设计流程，而且提供了系统级的设计流程，全新的系统级设计的中心思想是基于知识产权（Intellectual Property，IP）核的设计。与前一代的ISE设计平台相比，Vivado设计套件在各方面的性能都有了明显的提升。

## 2.5 Nexys4 DDR

Nexys4 DDR开发板搭载Xilinx Artix-7 FPGA芯片，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。Nexys4 DDR开发板集成了USB、以太网和其他端口，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。几个内置的外设：包括一个加速度计，一个温度传感器，微机电系统数字麦克风，扩音器和大量的I/O设备使Nexys4 DDR在不需要任何其他组件的情况下就能满足广泛的设计需求。新一代的Nexys4 DDR最值得被关注的改良是将原先的16MB的CellularRAM升级为128MB的DDR2 SDRAM内存。

# 3 单周期CPU概要设计

## 3.1 总体设计

单周期CPU总体结构包括PC（程序计数器)、IM（指令存储器）、RF（寄存器文件）、ALU（运算器）、NPC（PC计算器）、EXT（数据扩展单元）、DM（数据存储器）、Ctrl（控制器）和MUX（多路选择器）。

## 3.2 PC（程序计数器）

### 3.2.1 功能描述

PC用来给出下一个指令的地址。

### 3.2.2 模块接口

表3.1 PC（程序计数器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| NPC[31:0] | I | 不考虑重置信号的下一条PC |
| PC[31:0] | O | 指向的下一条的PC |

## 3.3 IM（指令存储器）

### 3.3.1 功能描述

存储指令。

### 3.3.2 模块接口

表3.2 IM（指令存储器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[8:2] | I | 访问地址 |
| dout[31:0] | O | 读出的指令 |

## 3.4 RF（寄存器文件）

### 3.4.1 功能描述

保存寄存器文件，并支持对通用寄存器的访问。

### 3.4.2 模块接口

表3.3 RF（寄存器文件）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| RFWr | I | 寄存器写使能信号 |
| A1[4:0] | I | 需要读的寄存器1的地址 |
| A2[4:0] | I | 需要读的寄存器2的地址 |
| A3[4:0] | I | 需要写的寄存器3的地址 |
| WD[31:0] | I | 需要写的寄存器的数据 |
| reg\_sel[4:0] | I | 需要用于调试的寄存器的地址 |
| RD1[31:0] | O | 需要读的寄存器1的数据 |
| RD2[31:0] | O | 需要读的寄存器2的数据 |
| reg\_data[31:0] | O | 需要用于调试的寄存器的数据 |

## 3.5 ALU（运算器）

### 3.5.1 功能描述

完成对输入数据的进行加法、减法、与、或、或非、左移、右移运算以及小于则置1、判断两个操作数是否相等。

### 3.5.2 模块接口

表3.4 ALU（运算器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| A[31:0] | I | 操作数A |
| B[31:0] | I | 操作数B |
| ALUOp[3:0] | I | 需要进行的运算 |
| C[31:0] | O | 运算结果 |
| Zero | O | 两操作数是否相等 |

## 3.6 NPC（PC计算器）

### 3.6.1 功能描述

完成对下一个PC值的计算。

### 3.6.2 模块接口

表3.5 NPC（PC计算器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| PC[31:0] | I | 当前PC |
| NPCOp[1:0] | I | 下一步PC的操作 |
| IMM[25:0] | I | 立即数 |
| PCJR[31:0] | I | PC跳转的值 |
| NPC[31:0] | O | 下一个PC值 |

## 3.7 EXT（数据扩展单元）

### 3.7.1 功能描述

将16位的数据扩展为32位数据。

### 3.7.2 模块接口

表3.6 EXT（数据扩展单元）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Imm16[15:0] | I | 需要进行扩展的数据 |
| EXTOp | I | 扩展方式的控制信号 |
| Imm32[31:0] | O | 扩展结果 |

## 3.8 DM（数据存储器）

### 3.8.1 功能描述

存储数据。

### 3.8.2 模块接口

表3.7 DM（数据存储器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| DMWr | I | 写信号 |
| addr[8:2] | I | 访问地址 |
| din[31:0] | I | 输入到数据寄存器的数据 |
| dout[31:0] | O | 读出的结果 |

## 3.9 Ctrl（寄存器文件）

### 3.9.1 功能描述

控制信号的单元。

### 3.9.2 模块接口

表3.8 Ctrl（寄存器文件）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Op[5:0] | I | 指令格式中的op域 |
| Funct[5:0] | I | 指令格式中的func域 |
| Zero | I | ALU的相等零信号 |
| RegWrite | O | 写寄存器信号 |
| MemWrite | O | 写存储器信号 |
| EXTOp | O | 位扩展信号 |
| ALUOp[3:0] | O | ALU的运算选择信号 |
| NPCOp[1:0] | O | 下一条PC选择信号 |
| ALUSrc | O | ALU操作数B的来源 |
| ARegSel | O | ALU操作数A的来源 |
| GPRSel[1:0] | O | 目标寄存器选择 |
| WDSel[1:0] | O | 寄存器写数据选择 |

## 3.10 MUX（多路选择器）

### 3.10.1 功能描述

从输入的数据中选择一个作为输出。

### 3.10.2 模块接口

表3.9 MUX（多路选择器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| d0, d1, d2… | I | 供选择数据 |
| s | I | 选择信号 |
| y | O | 选择后的数据 |

## 3.11 SCCPU（单周期CPU核心模块）

### 3.11.1 功能描述

将单周期CPU相关重要部件结合。

### 3.11.2 模块接口

表3.10 SCCPU（单周期CPU核心模块）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| instr[31:0] | I | 指令 |
| readdata[31:0] | I | 存储器读出的数据 |
| reg\_sel[4:0] | I | 用于调试的寄存器地址 |
| PC[31:0] | O | PC |
| MemWrite | O | 写存储器信号 |
| aluout[31:0] | O | ALU输出结果 |
| writedata[31:0] | O | 写往存储器的数据 |
| reg\_data[31:0] | O | 调试寄存器的数据 |

## 3.12 SCCOMP（顶层模块）

### 3.12.1 功能描述

完整的单周期CPU。

### 3.12.2 模块接口

表3.11 SCCOMP（顶层模块）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rstn | I | 重置信号的反信号 |
| reg\_sel[4:0] | I | 用于调试的寄存器地址 |
| reg\_data[31:0] | O | 调试寄存器的数据 |

## 3.13 ctrl\_encode\_def（宏定义模块）

定义相关控制信号的宏。

# 4 单周期CPU详细设计

## 4.1 CPU总体结构

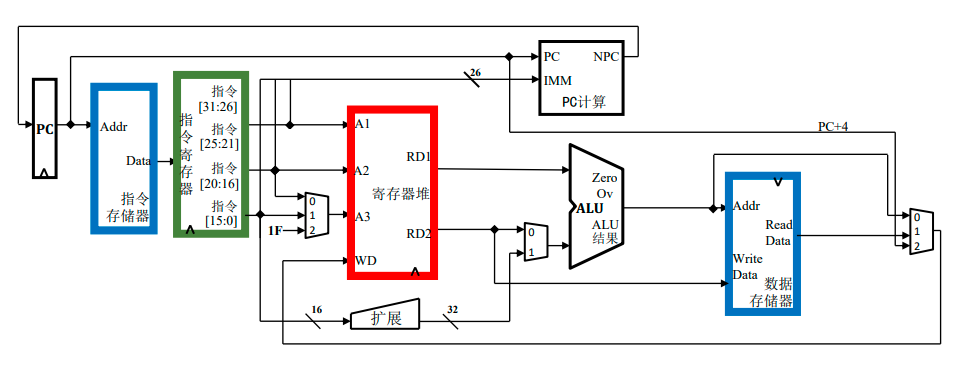
单周期CPU的总体结构如图4.1所示。

图4.1 单周期CPU总体结构

## 4.2 PC（程序计数器）

当重置信号rst在上升沿时，PC重置为全0，其他情况下当时钟信号在上升沿时，PC变为下一个PC值（NPC）。

module PC( clk, rst, NPC, PC );

input clk;

input rst;

input [31:0] NPC;

output reg [31:0] PC;

always @(posedge clk, posedge rst)

if (rst)

PC <= 32'h0000\_0000;

else

PC <= NPC;

endmodule

## 4.3 IM（指令存储器）

内含ROM用于存储指令。

module im(input [8:2] addr,

output [31:0] dout );

reg [31:0] ROM[127:0];

assign dout = ROM[addr]; // word aligned

endmodule

## 4.4 RF（寄存器文件）

当写寄存器信号（RFWr）有效时，将数据写入寄存器。

module RF( input clk,

input rst,

input RFWr,

input [4:0] A1, A2, A3,

input [31:0] WD,

output [31:0] RD1, RD2,

input [4:0] reg\_sel,

output [31:0] reg\_data);

reg [31:0] rf[31:0];

integer i;

always @(posedge clk, posedge rst)

if (rst) begin // reset

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

rf[i] <= 0; // i;

end

else

if (RFWr) begin

rf[A3] <= WD;

$display("r[00-07]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", 0, rf[1], rf[2], rf[3], rf[4], rf[5], rf[6], rf[7]);

$display("r[08-15]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[8], rf[9], rf[10], rf[11], rf[12], rf[13], rf[14], rf[15]);

$display("r[16-23]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[16], rf[17], rf[18], rf[19], rf[20], rf[21], rf[22], rf[23]);

$display("r[24-31]=0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X, 0x%8X", rf[24], rf[25], rf[26], rf[27], rf[28], rf[29], rf[30], rf[31]);

$display("r[%2d] = 0x%8X,", A3, WD);

end

assign RD1 = (A1 != 0) ? rf[A1] : 0;

assign RD2 = (A2 != 0) ? rf[A2] : 0;

assign reg\_data = (reg\_sel != 0) ? rf[reg\_sel] : 0;

endmodule

## 4.5 ALU（运算器）

根据运算器操作选择信号（ALUOp）进行对应的运算。

`include "ctrl\_encode\_def.v"

module alu(A, B, ALUOp, C, Zero);

input signed [31:0] A, B;

input [3:0] ALUOp;

output signed [31:0] C;

output Zero;

reg [31:0] C;

integer i;

always @( \* ) begin

case ( ALUOp )

`ALU\_NOP: C = A; // NOP

`ALU\_ADD: C = A + B; // ADD

`ALU\_SUB: C = A - B; // SUB

`ALU\_AND: C = A & B; // AND/ANDI

`ALU\_OR: C = A | B; // OR/ORI

`ALU\_SLT: C = (A < B) ? 32'd1 : 32'd0; // SLT/SLTI

`ALU\_SLTU: C = ({1'b0, A} < {1'b0, B}) ? 32'd1 : 32'd0;

`ALU\_NOR: C = ~(A | B);

`ALU\_SLL: C = B << A; //sll

`ALU\_SRL: C = B >> A; //srl

`ALU\_SRA: C = B >>> A; //sra

`ALU\_SLLV: C = B << A; //sllv

`ALU\_SRLV: C = B >> A; //srlv

`ALU\_SLL16: C = B << 16; //lui

default: C = A; // Undefined

endcase

end // end always

assign Zero = (C == 32'b0);

endmodule

## 4.6 NPC（PC计算器）

根据下一个PC的操作信号（NPCOp）计算出下一个PC。

`include "ctrl\_encode\_def.v"

module NPC(PC, NPCOp, PCJR, IMM, NPC); // next pc module

input [31:0] PC; // pc

input [1:0] NPCOp; // next pc operation

input [25:0] IMM; // immediate

input [31:0] PCJR;

output reg [31:0] NPC; // next pc

wire [31:0] PCPLUS4;

assign PCPLUS4 = PC + 4; // pc + 4

always @(\*) begin

case (NPCOp)

`NPC\_PLUS4: NPC = PCPLUS4;

`NPC\_BRANCH: NPC = PCPLUS4 + {{14{IMM[15]}}, IMM[15:0], 2'b00};

`NPC\_JUMP: NPC = {PCPLUS4[31:28], IMM[25:0], 2'b00};

`NPC\_JR: NPC = PCJR;

default: NPC = PCPLUS4;

endcase

end // end always

endmodule

## 4.7 EXT（数据扩展单元）

根据扩展信号（EXTOp）将16位立即数扩展为32位。

module EXT( Imm16, EXTOp, Imm32 );

input [15:0] Imm16;

input EXTOp;

output [31:0] Imm32;

assign Imm32 = (EXTOp) ? {{16{Imm16[15]}}, Imm16} : {16'd0, Imm16}; // signed-extension or zero extension

endmodule

## 4.8 DM（数据存储器）

存储数据，在写存储器信号（DMWr）有效时写数据。

module dm(clk, DMWr, addr, din, dout);

input clk;

input DMWr;

input [8:2] addr;

input [31:0] din;

output [31:0] dout;

reg [31:0] dmem[127:0];

always @(posedge clk)

if (DMWr) begin

dmem[addr[8:2]] <= din;

$display("dmem[0x%8X] = 0x%8X,", addr << 2, din);

end

assign dout = dmem[addr[8:2]];

endmodule

## 4.9 Ctrl（控制器）

根据指令控制各元器件的输入信号。

module ctrl(Op, Funct, Zero,

RegWrite, MemWrite,

EXTOp, ALUOp, NPCOp,

ALUSrc, GPRSel, WDSel, ARegSel

);

input [5:0] Op; // opcode

input [5:0] Funct; // funct

input Zero;

output RegWrite; // control signal for register write

output MemWrite; // control signal for memory write

output EXTOp; // control signal to signed extension

output [3:0] ALUOp; // ALU opertion

output [1:0] NPCOp; // next pc operation

output ALUSrc; // ALU source for B

output ARegSel; //ALU source for A

output [1:0] GPRSel; // general purpose register selection

output [1:0] WDSel; // (register) write data selection

// r format

wire rtype = ~|Op;

wire i\_add = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // add

wire i\_sub = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // sub

wire i\_and = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0]; // and

wire i\_or = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]& Funct[0]; // or

wire i\_slt = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; // slt

wire i\_sltu = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // sltu

wire i\_addu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; // addu

wire i\_subu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]& Funct[0]; // subu

wire i\_jr = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; //jr 001000

wire i\_jalr = rtype&~Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; //jalr 001001

wire i\_nor = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]& Funct[0]; //nor 100111

wire i\_sll = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; //sll 000000

wire i\_srl = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]&~Funct[0]; //srl 000010

wire i\_sra = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]& Funct[1]& Funct[0]; //sra 000011

wire i\_sllv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&~Funct[1]&~Funct[0]; //sllv 000100

wire i\_srlv = rtype&~Funct[5]&~Funct[4]&~Funct[3]& Funct[2]& Funct[1]&~Funct[0]; //srlv 000110

// i format

wire i\_addi = ~Op[5]&~Op[4]& Op[3]&~Op[2]&~Op[1]&~Op[0]; // addi

wire i\_ori = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]& Op[0]; // ori

wire i\_lw = Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // lw

wire i\_sw = Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]& Op[0]; // sw

wire i\_beq = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]&~Op[0]; // beq

wire i\_bne = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]& Op[0]; //bne 000101

wire i\_slti = ~Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]&~Op[0]; //slti 001010

wire i\_lui = ~Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0]; //lui 001111

wire i\_andi = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]&~Op[0]; //001100

// j format

wire i\_j = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]&~Op[0]; // j

wire i\_jal = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // jal

// generate control signals

assign RegWrite = rtype | i\_lw | i\_addi | i\_ori | i\_jal | i\_jalr | i\_slti | i\_lui | i\_andi; // register write

assign MemWrite = i\_sw; // memory write

assign ALUSrc = i\_lw | i\_sw | i\_addi | i\_ori | i\_slti | i\_lui | i\_andi; // ALU B is from instruction immediate

assign ARegSel = i\_sll | i\_srl | i\_sra;

assign EXTOp = i\_addi | i\_lw | i\_sw | i\_lui | i\_andi; // signed extension

// GPRSel\_RD 2'b00

// GPRSel\_RT 2'b01

// GPRSel\_31 2'b10

assign GPRSel[0] = i\_lw | i\_addi | i\_ori | i\_lui | i\_andi;

assign GPRSel[1] = i\_jal | i\_jalr;

// WDSel\_FromALU 2'b00

// WDSel\_FromMEM 2'b01

// WDSel\_FromPC 2'b10

assign WDSel[0] = i\_lw;

assign WDSel[1] = i\_jal | i\_jalr;

// NPC\_PLUS4 2'b00

// NPC\_BRANCH 2'b01

// NPC\_JUMP 2'b10

// NPC\_JR 2'b11

assign NPCOp[0] = (i\_beq & Zero) | (i\_bne & ~Zero) | i\_jr | i\_jalr;

assign NPCOp[1] = i\_j | i\_jal | i\_jr | i\_jalr;

// `define ALU\_NOP 4'b0000

// `define ALU\_ADD 4'b0001

// `define ALU\_SUB 4'b0010

// `define ALU\_AND 4'b0011

// `define ALU\_OR 4'b0100

// `define ALU\_SLT 4'b0101

// `define ALU\_SLTU 4'b0110

// `define ALU\_NOR 4'b0111

// `define ALU\_SLL 4'b1000

// `define ALU\_SRL 4'b1001

// `define ALU\_SRA 4'b1010

//`define ALU\_SLLV 4'b1011

//`define ALU\_SRLV 4'b1100

//`define ALU\_SLL16 4'b1101

assign ALUOp[0] = i\_add | i\_lw | i\_sw | i\_addi | i\_and | i\_slt | i\_addu | i\_nor | i\_srl | i\_sllv |i\_slti |i\_lui | i\_andi;

assign ALUOp[1] = i\_sub | i\_beq | i\_and | i\_sltu | i\_subu | i\_bne | i\_nor | i\_sra | i\_sllv | i\_andi;

assign ALUOp[2] = i\_or | i\_ori | i\_slt | i\_sltu | i\_nor | i\_srlv | i\_slti | i\_lui;

assign ALUOp[3] = i\_sll | i\_sra | i\_srl | i\_sllv | i\_srlv | i\_lui;

endmodule

## 4.10 MUX（多路选择器）

从多个输入中按照选择信号（s）选择一个作为输出。

// mux2

module mux2 #(parameter WIDTH = 8)

(d0, d1,

s, y);

input [WIDTH-1:0] d0, d1;

input s;

output [WIDTH-1:0] y;

assign y = ( s == 1'b1 ) ? d1:d0;

endmodule

// mux4

module mux4 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [1:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

2'b00: y\_r = d0;

2'b01: y\_r = d1;

2'b10: y\_r = d2;

2'b11: y\_r = d3;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

// mux8

module mux8 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

d4, d5, d6, d7,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [WIDTH-1:0] d4, d5, d6, d7;

input [2:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

3'd0: y\_r = d0;

3'd1: y\_r = d1;

3'd2: y\_r = d2;

3'd3: y\_r = d3;

3'd4: y\_r = d4;

3'd5: y\_r = d5;

3'd6: y\_r = d6;

3'd7: y\_r = d7;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

// mux16

module mux16 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

d4, d5, d6, d7,

d8, d9, d10, d11,

d12, d13, d14, d15,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [WIDTH-1:0] d4, d5, d6, d7;

input [WIDTH-1:0] d8, d9, d10, d11;

input [WIDTH-1:0] d12, d13, d14, d15;

input [3:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

4'd0: y\_r = d0;

4'd1: y\_r = d1;

4'd2: y\_r = d2;

4'd3: y\_r = d3;

4'd4: y\_r = d4;

4'd5: y\_r = d5;

4'd6: y\_r = d6;

4'd7: y\_r = d7;

4'd8: y\_r = d8;

4'd9: y\_r = d9;

4'd10: y\_r = d10;

4'd11: y\_r = d11;

4'd12: y\_r = d12;

4'd13: y\_r = d13;

4'd14: y\_r = d14;

4'd15: y\_r = d15;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

## 4.11 SCCPU（单周期CPU核心模块）

组合若干单周期CPU用到的元器件。

module sccpu( clk, rst, instr, readdata, PC, MemWrite, aluout, writedata, reg\_sel, reg\_data);

input clk; // clock

input rst; // reset

input [31:0] instr; // instruction

input [31:0] readdata; // data from data memory

output [31:0] PC; // PC address

output MemWrite; // memory write

output [31:0] aluout; // ALU output

output [31:0] writedata; // data to data memory

input [4:0] reg\_sel; // register selection (for debug use)

output [31:0] reg\_data; // selected register data (for debug use)

wire RegWrite; // control signal to register write

wire EXTOp; // control signal to signed extension

wire [3:0] ALUOp; // ALU opertion

wire [1:0] NPCOp; // next PC operation

wire [1:0] WDSel; // (register) write data selection

wire [1:0] GPRSel; // general purpose register selection

wire ALUSrc; // ALU source for B

wire ARegSel; // ALU sourec for A

wire Zero; // ALU ouput zero

wire [31:0] NPC; // next PC

wire [4:0] rs; // rs

wire [4:0] rt; // rt

wire [4:0] rd; // rd

wire [5:0] Op; // opcode

wire [5:0] Funct; // funct

wire [31:0] shamt;

wire [15:0] Imm16; // 16-bit immediate

wire [31:0] Imm32; // 32-bit immediate

wire [25:0] IMM; // 26-bit immediate (address)

wire [4:0] A3; // register address for write

wire [31:0] WD; // register write data

wire [31:0] RD1; // register data specified by rs

wire [31:0] B; // operator for ALU B

wire [31:0] A;

assign Op = instr[31:26]; // instruction

assign Funct = instr[5:0]; // funct

assign rs = instr[25:21]; // rs

assign rt = instr[20:16]; // rt

assign rd = instr[15:11]; // rd

assign shamt = {27'b0, instr[10:6]};

assign Imm16 = instr[15:0];// 16-bit immediate

assign IMM = instr[25:0]; // 26-bit immediate

// instantiation of control unit

ctrl U\_CTRL (

.Op(Op), .Funct(Funct), .Zero(Zero),

.RegWrite(RegWrite), .MemWrite(MemWrite),

.EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),

.ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel), .ARegSel(ARegSel)

);

// instantiation of PC

PC U\_PC (

.clk(clk), .rst(rst), .NPC(NPC), .PC(PC)

);

// instantiation of NPC

NPC U\_NPC (

.PC(PC), .NPCOp(NPCOp),.PCJR(RD1), .IMM(IMM), .NPC(NPC)

);

// instantiation of register file

RF U\_RF (

.clk(clk), .rst(rst), .RFWr(RegWrite),

.A1(rs), .A2(rt), .A3(A3),

.WD(WD),

.RD1(RD1), .RD2(writedata),

.reg\_sel(reg\_sel),

.reg\_data(reg\_data)

);

// mux for register data to write

mux4 #(5) U\_MUX4\_GPR\_A3 (

.d0(rd), .d1(rt), .d2(5'b11111), .d3(5'b0), .s(GPRSel), .y(A3)

);

// mux for register address to write

mux4 #(32) U\_MUX4\_GPR\_WD (

.d0(aluout), .d1(readdata), .d2(PC + 4), .d3(32'b0), .s(WDSel), .y(WD)

);

// mux for signed extension or zero extension

EXT U\_EXT (

.Imm16(Imm16), .EXTOp(EXTOp), .Imm32(Imm32)

);

// mux for ALU B

mux2 #(32) U\_MUX\_ALU\_B (

.d0(writedata), .d1(Imm32), .s(ALUSrc), .y(B)

);

mux2 #(32) U\_MUX\_ALU\_A(

.d0(RD1), .d1(shamt), .s(ARegSel), .y(A)

);

// instantiation of alu

alu U\_ALU (

.A(A), .B(B), .ALUOp(ALUOp), .C(aluout), .Zero(Zero)

);

endmodule

## 4.12 SCCOMP（顶层模块）

完整的单周期CPU。

module sccomp(clk, rstn, reg\_sel, reg\_data);

input clk;

input rstn;

input [4:0] reg\_sel;

output [31:0] reg\_data;

wire [31:0] instr;

wire [31:0] PC;

wire MemWrite;

wire [31:0] dm\_addr, dm\_din, dm\_dout;

wire rst = ~rstn;

// instantiation of single-cycle CPU

sccpu U\_SCPU(

.clk(clk), // input: cpu clock

.rst(rst), // input: reset

.instr(instr), // input: instruction

.readdata(dm\_dout), // input: data to cpu

.MemWrite(MemWrite), // output: memory write signal

.PC(PC), // output: PC

.aluout(dm\_addr), // output: address from cpu to memory

.writedata(dm\_din), // output: data from cpu to memory

.reg\_sel(reg\_sel), // input: register selection

.reg\_data(reg\_data) // output: register data

);

// instantiation of data memory

dm U\_DM(

.clk(clk), // input: cpu clock

.DMWr(MemWrite), // input: ram write

.addr(dm\_addr[8:2]), // input: ram address

.din(dm\_din), // input: data to ram

.dout(dm\_dout) // output: data from ram

);

// instantiation of intruction memory (used for simulation)

im U\_IM (

.addr(PC[8:2]), // input: rom address

.dout(instr) // output: instruction

);

endmodule

## 4.13 ctrl\_encode\_def（宏定义模块）

定义相关控制信号的宏。

// NPC control signal

`define NPC\_PLUS4 2'b00

`define NPC\_BRANCH 2'b01

`define NPC\_JUMP 2'b10

`define NPC\_JR 2'b11

// ALU control signal

`define ALU\_NOP 4'b0000

`define ALU\_ADD 4'b0001

`define ALU\_SUB 4'b0010

`define ALU\_AND 4'b0011

`define ALU\_OR 4'b0100

`define ALU\_SLT 4'b0101

`define ALU\_SLTU 4'b0110

`define ALU\_NOR 4'b0111

`define ALU\_SLL 4'b1000

`define ALU\_SRL 4'b1001

`define ALU\_SRA 4'b1010

`define ALU\_SLLV 4'b1011

`define ALU\_SRLV 4'b1100

`define ALU\_SLL16 4'b1101

# 5 流水线CPU概要设计

## 5.1 总体设计

流水线CPU总体结构包括PC（程序计数器)、IM（指令存储器）、RF（寄存器文件）、ALU（运算器）、EXT（数据扩展单元）、DM（数据存储器）、Ctrl（控制器）、ID（IF/ID段间寄存器）、EXE（ID/EX段间寄存器）、MEM（EX/MEM段间寄存器）、WB（MEM/WB段间寄存器）、MUX（多路选择器）。

可以看出流水线CPU只是在单周期CPU的基础上增加了各段之间的寄存器单元以及相关的冒险检测单元，即将单周期CPU进行了优化。

## 5.2 PC（程序计数器）

### 5.2.1 功能描述

PC用来给出下一个指令的地址。

### 5.2.2 模块接口

表5.1 PC（程序计数器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| NPC[31:0] | I | 不考虑阻塞的下一条PC |
| ID\_nostall | I | 根据ID级的控制器决定是否锁定PC，暂停一个周期 |
| IF\_PC[31:0] | O | 指向的下一条的PC |

## 5.3 IM（指令存储器）

### 5.3.1 功能描述

存储指令。

### 5.3.2 模块接口

表5.2 IM（指令存储器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| addr[8:2] | I | 访问地址 |
| dout[31:0] | O | 读出的指令 |

## 5.4 RF（寄存器文件）

### 5.4.1 功能描述

保存寄存器文件，并支持对通用寄存器的访问。

### 5.4.2 模块接口

表5.3 RF（寄存器文件）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| RFWr | I | 寄存器写使能信号 |
| A1[4:0] | I | 需要读的寄存器1的地址 |
| A2[4:0] | I | 需要读的寄存器2的地址 |
| A3[4:0] | I | 需要写的寄存器3的地址 |
| WD[31:0] | I | 需要写的寄存器的数据 |
| reg\_sel[4:0] | I | 需要用于调试的寄存器的地址 |
| RD1[31:0] | O | 需要读的寄存器1的数据 |
| RD2[31:0] | O | 需要读的寄存器2的数据 |
| reg\_data[31:0] | O | 需要用于调试的寄存器的数据 |

## 5.5 ALU（运算器）

### 5.5.1 功能描述

完成对输入数据的进行加法、减法、与、或、或非、左移、右移运算以及小于则置1。

### 5.5.2 模块接口

表5.4 ALU（运算器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| A[31:0] | I | 操作数A |
| B[31:0] | I | 操作数B |
| ALUOp[3:0] | I | 需要进行的运算 |
| C[31:0] | O | 运算结果 |

## 5.6 EXT（数据扩展单元）

### 5.6.1 功能描述

将16位的数据扩展为32位数据。

### 5.6.2 模块接口

表5.5 EXT（数据扩展单元）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Imm16[15:0] | I | 需要进行扩展的数据 |
| EXTOp | I | 扩展方式的控制信号 |
| Imm32[31:0] | O | 扩展结果 |

## 5.7 DM（数据存储器）

### 5.7.1 功能描述

存储数据。

### 5.7.2 模块接口

表5.6 DM（数据存储器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| DMWr | I | 写信号 |
| addr[8:2] | I | 访问地址 |
| din[31:0] | I | 输入到数据寄存器的数据 |
| dout[31:0] | O | 读出的结果 |

## 5.8 Ctrl（寄存器文件）

### 5.8.1 功能描述

控制信号的单元。

### 5.8.2 模块接口

表5.7 Ctrl（寄存器文件）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| ID\_Op[5:0] | I | 指令格式中的op域 |
| ID\_Funct[5:0] | I | 指令格式中的func域 |
| ID\_rs[4:0] | I | IF/ID级的第一个寄存器号 |
| ID\_rt[4:0] | I | IF/ID级的第二个寄存器号 |
| ID\_Zero | I | IF/ID级两寄存器相等信号 |
| EXE\_RegWrite | I | ID/EX级写寄存器信号 |
| MEM\_RegWrite | I | EX/MEM级写寄存器信号 |
| EXE\_mem\_to\_reg | I | ID/EX级存储器向寄存器信号 |
| MEM\_mem\_to\_reg | I | EX/MEM级存储器向寄存器信号 |
| EXE\_writereg\_num[4:0] | I | ID/EX级写寄存器号 |
| MEM\_writereg\_num[4:0] | I | EX/MEM级写寄存器号 |
| ID\_RegWrite | O | IF/ID级写寄存器信号 |
| ID\_mem\_to\_reg | O | IF/ID级存储器向寄存器信号 |
| ID\_writereg\_to\_rt[1:0] | O | IF/ID级写寄存器到第二个寄存器信号 |
| ID\_memwrite | O | IF/ID级写存储器信号 |
| ID\_extOp | O | IF/ID级数据扩展信号 |
| ID\_aluOp[3:0] | O | IF/ID级ALU操作信号 |
| ID\_npcOp[1:0] | O | IF/ID级下一条PC操作信号 |
| ID\_jal | O | IF/ID级jal类指令信号 |
| ID\_alua | O | IF/ID级ALU第一个数据来源信号 |
| ID\_alub | O | IF/ID级ALU第二个数据来源信号 |
| ID\_nostall | O | IF/ID级阻塞信号 |
| ID\_forwarda[1:0] | O | IF/ID级第一个数据旁路信号 |
| ID\_forwardb[1:0] | O | IF/ID级第二个数据旁路信号 |

## 5.9 ID（IF/ID段间寄存器）

### 5.9.1 功能描述

IF/ID段间寄存器。

### 5.9.2 模块接口

表5.8 ID（IF/ID段间寄存器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| ID\_nostall | I | IF/ID级阻塞信号 |
| IF\_INS[31:0] | I | IF级的指令 |
| IF\_PC[31:0] | I | IF级的PC |
| ID\_INS[31:0] | O | ID级的指令 |
| ID\_PC[31:0] | O | ID级的PC |

## 5.10 EXE（ID/EX段间寄存器）

### 5.10.1 功能描述

EXE（ID/EX段间寄存器）。

### 5.10.2 模块接口

表5.9 EXE（ID/EX段间寄存器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| ID\_RegWrite | I | IF/ID级写寄存器信号 |
| ID\_mem\_to\_reg | I | IF/ID级存储器向寄存器信号 |
| ID\_memwrite | I | IF/ID级写存储器信号 |
| ID\_aluOp[3:0] | I | IF/ID级ALU操作信号 |
| ID\_jal | I | IF/ID级jal类指令信号 |
| ID\_alua | I | IF/ID级ALU第一个数据来源信号 |
| ID\_alub | I | IF/ID级ALU第二个数据来源信号 |
| ID\_PC[31:0] | I | ID级的PC |
| ID\_A[31:0] | I | IF/ID级第一个数据 |
| ID\_B[31:0] | I | IF/ID级第二个数据 |
| ID\_IMM32[31:0] | I | IF/ID级立即数 |
| ID\_writereg\_num[4:0] | I | IF/ID级写寄存器号 |
| EXE\_RegWrite | O | ID/EX级写寄存器信号 |
| EXE\_mem\_to\_reg | O | ID/EX级存储器向寄存器信号 |
| EXE\_memwrite | O | ID/EX级写存储器信号 |
| EXE\_aluOp[3:0] | O | ID/EX级ALU操作信号 |
| EXE\_jal | O | ID/EX级jal类指令信号 |
| EXE\_alua | O | ID/EX级ALU第一个数据来源信号 |
| EXE\_alub | O | ID/EX级ALU第二个数据来源信号 |
| EXE\_PC[31:0] | O | EX级的PC |
| EXE\_A[31:0] | O | ID/EX级第一个数据 |
| EXE\_B[31:0] | O | ID/EX级第二个数据 |
| EXE\_IMM32[31:0] | O | ID/EX级立即数 |
| EXE\_writereg\_num[4:0] | O | ID/EX级写寄存器号 |

## 5.11 MEM（EX/MEM段间寄存器）

### 5.11.1 功能描述

MEM（EX/MEM段间寄存器）。

### 5.11.2 模块接口

表5.10 MEM（EX/MEM段间寄存器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| EXE\_RegWrite | I | ID/EX级写寄存器信号 |
| EXE\_mem\_to\_reg | I | ID/EX级存储器向寄存器信号 |
| EXE\_memwrite | I | ID/EX级写存储器信号 |
| EXE\_B[31:0] | I | ID/EX级第二个数据 |
| EXE\_C[31:0] | I | ID/EX级第三个数据 |
| EXE\_writereg\_num[4:0] | I | ID/EX级写寄存器号 |
| MEM\_RegWrite | O | EX/MEM级写寄存器信号 |
| MEM\_mem\_to\_reg | O | EX/MEM级存储器向寄存器信号 |
| MEM\_memwrite | O | EX/MEM级写存储器信号 |
| MEM\_B[31:0] | O | EX/MEM级第一个数据 |
| MEM\_C[31:0] | O | EX/MEM级第二个数据 |
| MEM\_writereg\_num[4:0] | O | EX/MEM级写寄存器号 |

## 5.12 WB（MEM/WB段间寄存器）

### 5.12.1 功能描述

WB（MEM/WB段间寄存器）。

### 5.12.2 模块接口

表5.11 WB（MEM/WB段间寄存器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| MEM\_RegWrite | I | EX/MEM级写寄存器信号 |
| MEM\_mem\_to\_reg | I | EX/MEM级存储器向寄存器信号 |
| MEM\_C[31:0] | I | EX/MEM级第三个数据 |
| MEM\_read[31:0] | I | EX/MEM级读数据 |
| MEM\_writereg\_num[4:0] | I | EX/MEM级写寄存器号 |
| WB\_RegWrite | O | MEM/WB级写寄存器信号 |
| WB\_mem\_to\_reg | O | MEM/WB级存储器向寄存器信号 |
| WB\_C[31:0] | O | MEM/WB级第三个数据 |
| WB\_read[31:0] | O | MEM/WB级读数据 |
| WB\_writereg\_num[4:0] | O | MEM/WB级写寄存器号 |

## 5.13 MUX（多路选择器）

### 5.13.1 功能描述

从输入的数据中选择一个作为输出。

### 5.13.2 模块接口

表5.12 MUX（多路选择器）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| d0, d1, d2… | I | 供选择数据 |
| s | I | 选择信号 |
| y | O | 选择后的数据 |

## 5.14 SCCPU（流水线CPU核心模块）

### 5.14.1 功能描述

将流水线CPU相关重要部件结合。

### 5.14.2 模块接口

表5.13 SCCPU（流水线CPU核心模块）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rst | I | 重置信号 |
| instr[31:0] | I | 指令 |
| readdata[31:0] | I | 存储器读出的数据 |
| reg\_sel[4:0] | I | 用于调试的寄存器地址 |
| PC[31:0] | O | PC |
| MemWrite | O | 写存储器信号 |
| aluout[31:0] | O | ALU输出结果 |
| writedata[31:0] | O | 写往存储器的数据 |
| reg\_data[31:0] | O | 调试寄存器的数据 |

## 5.15 SCCOMP（顶层模块）

### 5.15.1 功能描述

完整的流水线CPU。

### 5.15.2 模块接口

表5.14 SCCOMP（顶层模块）模块接口

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| rstn | I | 重置信号的反信号 |
| reg\_sel[4:0] | I | 用于调试的寄存器地址 |
| reg\_data[31:0] | O | 调试寄存器的数据 |

## 5.16 ctrl\_encode\_def（宏定义模块）

定义相关控制信号的宏。

# 6 流水线CPU详细设计

## 6.1 CPU总体结构

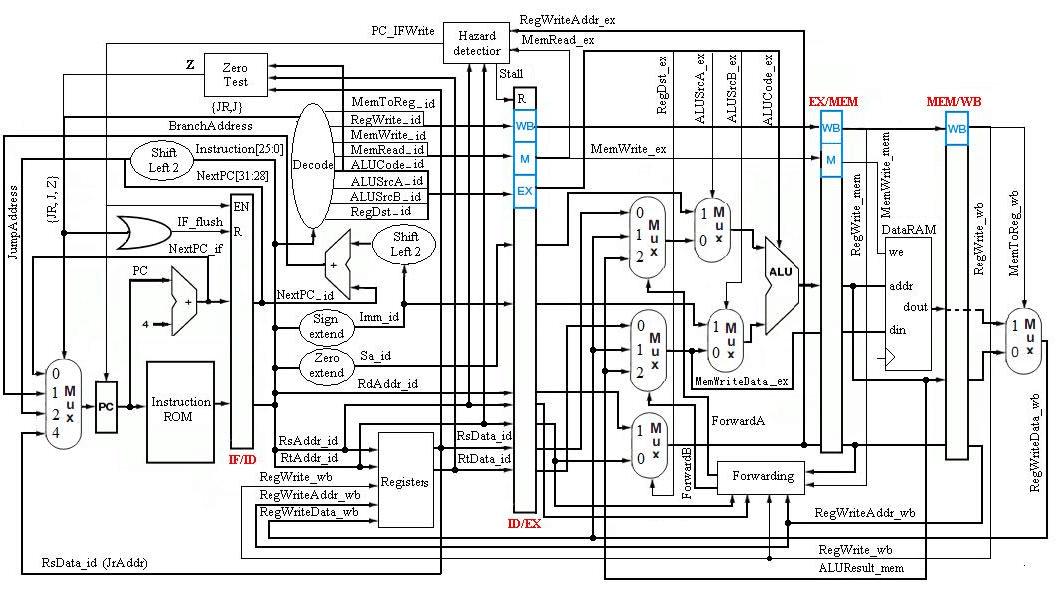
流水线CPU总体结构如图6.1所示。

图6.1 流水线CPU总体结构

## 6.2 PC（程序计数器）

当重置信号rst在上升沿时，PC重置为全0，其他情况下当时钟信号在上升沿时，PC变为下一个PC值（NPC）。

module PC( clk, rst, NPC, IF\_PC ,ID\_nostall);

input clk;

input rst,ID\_nostall;

input [31:0] NPC;

output reg [31:0] IF\_PC;

always @(posedge clk, posedge rst)

if (rst)

IF\_PC <= 32'h0000\_0000;

else if(ID\_nostall)

IF\_PC <= NPC;

endmodule

## 6.3 IM（指令存储器）

内含ROM用于存储指令。

module im(input [8:2] addr,

output [31:0] dout );

reg [31:0] ROM[127:0];

assign dout = ROM[addr];

endmodule

## 6.4 RF（寄存器文件）

当写寄存器信号（RFWr）有效时，将数据写入寄存器。

module RF( input clk,

input rst,

input RFWr,

input [4:0] A1, A2, A3,

input [31:0] WD,

output [31:0] RD1, RD2,

input [4:0] reg\_sel,

output [31:0] reg\_data);

reg [31:0] rf[31:0];

integer i;

//目前暂定为时钟的下降沿写入数据，上升沿复位

always @(negedge clk, posedge rst)

if (rst) begin // reset

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

rf[i] <= 0; // i;

end

else

if (RFWr) begin

rf[A3] <= WD;

end

assign RD1 = (A1 != 0) ? rf[A1] : 0;

assign RD2 = (A2 != 0) ? rf[A2] : 0;

assign reg\_data = (reg\_sel != 0) ? rf[reg\_sel] : 0;

endmodule

## 6.5 ALU（运算器）

根据运算器操作选择信号（ALUOp）进行对应的运算。

`include "ctrl\_encode\_def.v"

module alu(A, B, ALUOp, C);

//定义端口输入输出

input signed [31:0] A, B;

input [3:0] ALUOp;

output signed [31:0] C;

reg [31:0] C;

integer i;

always @( \* )

begin

case ( ALUOp)

`ALU\_NOP: C = A; // NOP

`ALU\_ADD: C = A + B; // ADD

`ALU\_SUB: C = A - B; // SUB

`ALU\_AND: C = A & B; // AND/ANDI

`ALU\_OR: C = A | B; // OR/ORI

`ALU\_SLT: C = (A < B) ? 32'd1 : 32'd0; // SLT/SLTI

`ALU\_SLTU: C = ({1'b0, A} < {1'b0, B}) ? 32'd1 : 32'd0; //SLTU

`ALU\_SLL: C = B << A[10:6]; //SLL

`ALU\_SLLV: C = B << A[4:0]; //SLLV

`ALU\_SRL: C = B >> A[10:6]; //SRL

`ALU\_SRLV: C = B >> A[4:0]; //SRLV

`ALU\_NOR: C = ~(A|B) ; //NOR

`ALU\_LUI: C = B << 16;

default: C = A; // Undefined

endcase

end // end always

endmodule

## 6.6 EXT（数据扩展单元）

根据扩展信号（EXTOp）将16位立即数扩展为32位。

module EXT( Imm16, EXTOp, Imm32 );

input [15:0] Imm16;

input EXTOp;

output [31:0] Imm32;

assign Imm32 = (EXTOp) ? {{16{Imm16[15]}}, Imm16} : {16'd0, Imm16};

endmodule

## 6.7 DM（数据存储器）

存储数据，在写存储器信号（DMWr）有效时写数据。

module dm(clk, DMWr, addr, din, dout);

input clk;

input DMWr;

input [8:2] addr;

input [31:0] din;

output [31:0] dout;

reg [31:0] dmem[127:0];

always @(posedge clk)

if (DMWr) begin

dmem[addr[8:2]] <= din;

$display("dmem[0x%8X] = 0x%8X,", addr << 2, din);

end

assign dout = dmem[addr[8:2]];

endmodule

## 6.8 Ctrl（控制器）

根据指令控制各元器件的输入信号。

module ctrl(ID\_Op, ID\_Funct,ID\_rs,ID\_rt,ID\_Zero,

EXE\_RegWrite,MEM\_RegWrite,EXE\_mem\_to\_reg,MEM\_mem\_to\_reg,EXE\_writereg\_num,MEM\_writereg\_num,

ID\_RegWrite,ID\_mem\_to\_reg,ID\_writereg\_to\_rt,ID\_memwrite,

ID\_extOp, ID\_aluOp, ID\_npcOp,ID\_jal,ID\_alua,ID\_alub,ID\_nostall,ID\_forwarda,ID\_forwardb

);

input [5:0] ID\_Op, ID\_Funct;

input [4:0] ID\_rs,ID\_rt,EXE\_writereg\_num,MEM\_writereg\_num;

input ID\_Zero,EXE\_RegWrite,MEM\_RegWrite,EXE\_mem\_to\_reg,MEM\_mem\_to\_reg;

output ID\_RegWrite,ID\_mem\_to\_reg,ID\_memwrite,ID\_jal,ID\_extOp,ID\_alua,ID\_alub,ID\_nostall;

output [1:0] ID\_npcOp,ID\_forwarda,ID\_forwardb,ID\_writereg\_to\_rt;

output [3:0] ID\_aluOp;

reg [1:0] ID\_forwarda,ID\_forwardb;

// r format

wire rtype = ~|ID\_Op;

wire i\_add = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]&~ID\_Funct[1]&~ID\_Funct[0]; // add

wire i\_sub = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]& ID\_Funct[1]&~ID\_Funct[0]; // sub

wire i\_and = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]& ID\_Funct[2]&~ID\_Funct[1]&~ID\_Funct[0]; // and

wire i\_or = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]& ID\_Funct[2]&~ID\_Funct[1]& ID\_Funct[0]; // or

wire i\_slt = rtype& ID\_Funct[5]&~ID\_Funct[4]& ID\_Funct[3]&~ID\_Funct[2]& ID\_Funct[1]&~ID\_Funct[0]; // slt

wire i\_sltu = rtype& ID\_Funct[5]&~ID\_Funct[4]& ID\_Funct[3]&~ID\_Funct[2]& ID\_Funct[1]& ID\_Funct[0]; // sltu

wire i\_addu = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]&~ID\_Funct[1]& ID\_Funct[0]; // addu

wire i\_subu = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]& ID\_Funct[1]& ID\_Funct[0]; // subu

wire i\_sll = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]& ~ID\_Funct[1]& ~ID\_Funct[0];

wire i\_srl = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&~ID\_Funct[2]& ID\_Funct[1]& ~ID\_Funct[0];

wire i\_jalr = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&ID\_Funct[3]&~ID\_Funct[2]& ~ID\_Funct[1]& ID\_Funct[0];

wire i\_jr = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&ID\_Funct[3]&~ID\_Funct[2]& ~ID\_Funct[1]& ~ID\_Funct[0];

wire i\_nor = rtype& ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&ID\_Funct[2]& ID\_Funct[1]& ID\_Funct[0];

wire i\_sllv = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&ID\_Funct[2]& ~ID\_Funct[1]& ~ID\_Funct[0];

wire i\_srlv = rtype& ~ID\_Funct[5]&~ID\_Funct[4]&~ID\_Funct[3]&ID\_Funct[2]& ID\_Funct[1]& ~ID\_Funct[0];

// i format

wire i\_addi = ~ID\_Op[5]&~ID\_Op[4]& ID\_Op[3]&~ID\_Op[2]&~ID\_Op[1]&~ID\_Op[0]; // addi

wire i\_ori = ~ID\_Op[5]&~ID\_Op[4]& ID\_Op[3]& ID\_Op[2]&~ID\_Op[1]& ID\_Op[0]; // ori

wire i\_lw = ID\_Op[5]&~ID\_Op[4]&~ID\_Op[3]&~ID\_Op[2]& ID\_Op[1]& ID\_Op[0]; // lw

wire i\_sw = ID\_Op[5]&~ID\_Op[4]& ID\_Op[3]&~ID\_Op[2]& ID\_Op[1]& ID\_Op[0]; // sw

wire i\_beq = ~ID\_Op[5]&~ID\_Op[4]&~ID\_Op[3]& ID\_Op[2]&~ID\_Op[1]&~ID\_Op[0]; // beq

wire i\_lui = ~ID\_Op[5]&~ID\_Op[4]&ID\_Op[3]& ID\_Op[2]&ID\_Op[1]&ID\_Op[0];//lui

wire i\_slti = ~ID\_Op[5]&~ID\_Op[4]&ID\_Op[3]& ~ID\_Op[2]&ID\_Op[1]&~ID\_Op[0];//slti

wire i\_bne = ~ID\_Op[5]&~ID\_Op[4]&~ID\_Op[3]& ID\_Op[2]&~ID\_Op[1]&ID\_Op[0];//bne

wire i\_andi = ~ID\_Op[5]&~ID\_Op[4]&ID\_Op[3]& ID\_Op[2]&~ID\_Op[1]&~ID\_Op[0];//andi

// j format

wire i\_j = ~ID\_Op[5]&~ID\_Op[4]&~ID\_Op[3]&~ID\_Op[2]& ID\_Op[1]&~ID\_Op[0]; // j

wire i\_jal = ~ID\_Op[5]&~ID\_Op[4]&~ID\_Op[3]&~ID\_Op[2]& ID\_Op[1]& ID\_Op[0]; // jal

//用于检测lw冒险的时候lw的下一条指令是否使用rs，rt寄存器。

wire i\_rs = i\_add | i\_sub | i\_and |i\_or | i\_slt | i\_sltu | i\_addu | i\_subu | i\_jalr | i\_jr | i\_nor | i\_sllv | i\_srlv | i\_addi | i\_ori | i\_lw | i\_sw | i\_beq | i\_slti | i\_bne | i\_andi;

wire i\_rt = i\_add | i\_sub | i\_and |i\_or | i\_slt | i\_sltu | i\_addu | i\_subu | i\_sll | i\_srl | i\_nor | i\_sllv | i\_srlv | i\_beq | i\_bne | i\_sw;

//允许写寄存器

assign ID\_RegWrite = (i\_add |i\_sub| i\_and|i\_or|i\_slt|i\_sltu|i\_addu|i\_subu| i\_lw | i\_addi | i\_ori | i\_jal | i\_lui | i\_slti | i\_sll | i\_srl | i\_jalr| i\_andi | i\_nor | i\_sllv | i\_srlv) & ID\_nostall; // register write

//允许写ROM

assign ID\_memwrite = i\_sw & ID\_nostall; // memory write

//写入寄存器的来源是内存中读出来的,否则就是从alu中读出来的

assign ID\_mem\_to\_reg = i\_lw;

//lw/sw型数据冒险，停止后序指令一个周期

assign ID\_nostall = ~(EXE\_RegWrite & EXE\_mem\_to\_reg & (EXE\_writereg\_num !=0) & ( i\_rs & (EXE\_writereg\_num == ID\_rs) | i\_rt & (EXE\_writereg\_num == ID\_rt)));

//默认情况下指令的写是写到rd寄存器，但是在一些立即数包括lw的运算上是写到rt上的

//更奇怪的是jal是写到31号寄存器的,但是却没有31号寄存器的标识！！

assign ID\_writereg\_to\_rt [0] = i\_lw | i\_addi | i\_ori | i\_lui | i\_slti | i\_andi;

assign ID\_writereg\_to\_rt [1] = i\_jal;

//默认情况下往寄存器中写的是数，但在jal和jalr的情况下，往31号寄存中写的是PC地址

assign ID\_jal = i\_jal | i\_jalr;

//正常进行高位填充0进行符号拓展，但是在有符号数的情况下就是填充最高位进行拓展

assign ID\_extOp = i\_addi | i\_lw | i\_sw | i\_bne | i\_beq;

//正常情况下alua是来自forwarda信号控制的多路选择器的，也就是寄存器的，但在sll，srl却是来自shamt字段的，此时就要传递符号拓展后的立即数然后切片的

assign ID\_alua = i\_sll | i\_srl;

//正常情况下alub是来自forwardb信号控制的多路选择器的，也就是寄存器的，但是在addi等的情况下却来自拓展后的立即数

assign ID\_alub = i\_addi | i\_andi | i\_ori | i\_lw | i\_sw | i\_lui | i\_slti;

//默认情况下是进行pc+4，只有出现跳转指令时

//我们定义01 bpc就是立即数+地址跳转

//10 npc寄存器跳转 典型的有jr jalr

//11 jpc就是指令跳转，典型的有j，jal；

assign ID\_npcOp[1] = i\_jr | i\_jalr | i\_j | i\_jal;

assign ID\_npcOp[0] = i\_beq & ID\_Zero | i\_bne & ~ID\_Zero | i\_j | i\_jal;

//旁路

always @ (ID\_rs,ID\_rt,EXE\_RegWrite,EXE\_mem\_to\_reg,EXE\_writereg\_num,MEM\_RegWrite,MEM\_mem\_to\_reg,MEM\_writereg\_num)

begin

ID\_forwarda = 2'b00;//默认没有冒险

if(EXE\_RegWrite & (EXE\_writereg\_num!=0) &(EXE\_writereg\_num ==ID\_rs) & ~EXE\_mem\_to\_reg)

begin

ID\_forwarda = 2'b01;//发生EXE-ID级R型冒险

end

else if(MEM\_RegWrite & (MEM\_writereg\_num!=0) &(MEM\_writereg\_num ==ID\_rs) &~MEM\_mem\_to\_reg)

begin

ID\_forwarda = 2'b10;//发生MEM-ID级冒险

end

else if(MEM\_RegWrite & (MEM\_writereg\_num!=0) & (MEM\_writereg\_num==ID\_rs) & MEM\_mem\_to\_reg)

begin

ID\_forwarda = 2'b11;//发生lw型的冒险

end

ID\_forwardb = 2'b00;//默认没有冒险

if(EXE\_RegWrite & (EXE\_writereg\_num!=0) &(EXE\_writereg\_num ==ID\_rt) & ~EXE\_mem\_to\_reg)

begin

ID\_forwardb = 2'b01;//发生EXE-ID级R型冒险

end

else if(MEM\_RegWrite & (MEM\_writereg\_num!=0) &(MEM\_writereg\_num ==ID\_rt) &~MEM\_mem\_to\_reg)

begin

ID\_forwardb = 2'b10;//发生MEM-ID级冒险

end

else if(MEM\_RegWrite & (MEM\_writereg\_num!=0) & (MEM\_writereg\_num==ID\_rt) & MEM\_mem\_to\_reg)

begin

ID\_forwardb = 2'b11;//发生lw型的冒险

end

end

//执行ALU指令

assign ID\_aluOp[0] = i\_add | i\_lw | i\_sw | i\_addi | i\_and | i\_slt | i\_addu | i\_sll | i\_andi | i\_nor |i\_srlv;

assign ID\_aluOp[1] = i\_sub | i\_beq | i\_and | i\_sltu | i\_subu | i\_sll | i\_slti | i\_bne | i\_andi | i\_sllv|i\_srlv;

assign ID\_aluOp[2] = i\_or | i\_ori | i\_slt | i\_sltu | i\_sll | i\_slti | i\_lui ;

assign ID\_aluOp[3] = i\_srl | i\_nor |i\_sllv|i\_srlv | i\_lui;

endmodule

## 6.9 MUX（多路选择器）

从多个输入中按照选择信号（s）选择一个作为输出。

// mux2

module mux2 #(parameter WIDTH = 8)

(d0, d1,

s, y);

input [WIDTH-1:0] d0, d1;

input s;

output [WIDTH-1:0] y;

assign y = ( s == 1'b1 ) ? d1:d0;

endmodule

// mux4

module mux4 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [1:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

2'b00: y\_r = d0;

2'b01: y\_r = d1;

2'b10: y\_r = d2;

2'b11: y\_r = d3;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

// mux8

module mux8 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

d4, d5, d6, d7,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [WIDTH-1:0] d4, d5, d6, d7;

input [2:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

3'd0: y\_r = d0;

3'd1: y\_r = d1;

3'd2: y\_r = d2;

3'd3: y\_r = d3;

3'd4: y\_r = d4;

3'd5: y\_r = d5;

3'd6: y\_r = d6;

3'd7: y\_r = d7;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

// mux16

module mux16 #(parameter WIDTH = 8)

(d0, d1, d2, d3,

d4, d5, d6, d7,

d8, d9, d10, d11,

d12, d13, d14, d15,

s, y);

input [WIDTH-1:0] d0, d1, d2, d3;

input [WIDTH-1:0] d4, d5, d6, d7;

input [WIDTH-1:0] d8, d9, d10, d11;

input [WIDTH-1:0] d12, d13, d14, d15;

input [3:0] s;

output [WIDTH-1:0] y;

reg [WIDTH-1:0] y\_r;

always @( \* ) begin

case ( s )

4'd0: y\_r = d0;

4'd1: y\_r = d1;

4'd2: y\_r = d2;

4'd3: y\_r = d3;

4'd4: y\_r = d4;

4'd5: y\_r = d5;

4'd6: y\_r = d6;

4'd7: y\_r = d7;

4'd8: y\_r = d8;

4'd9: y\_r = d9;

4'd10: y\_r = d10;

4'd11: y\_r = d11;

4'd12: y\_r = d12;

4'd13: y\_r = d13;

4'd14: y\_r = d14;

4'd15: y\_r = d15;

default: ;

endcase

end // end always

assign y = y\_r;

endmodule

## 6.10 ID（IF/ID段间寄存器）

储存IF/ID级相关数据。

module ID(clk,rst,ID\_nostall,IF\_INS,ID\_INS,IF\_PC,ID\_PC);

input clk,rst,ID\_nostall;

input [31:0] IF\_INS,IF\_PC;

output reg [31:0] ID\_INS,ID\_PC;

always@(posedge clk,posedge rst)

begin

if(rst)

ID\_INS =32'b0;

else if(ID\_nostall)

begin

ID\_INS = IF\_INS;

ID\_PC=IF\_PC+4;

end

end

endmodule

## 6.11 EXE（ID/EX段间寄存器）

储存ID/EX级相关数据。

module EXE (clk,rst,ID\_RegWrite,ID\_mem\_to\_reg,ID\_memwrite,ID\_aluOp, ID\_jal,ID\_alua,ID\_alub,ID\_PC,ID\_A,ID\_B,ID\_IMM32,ID\_writereg\_num,

EXE\_RegWrite,EXE\_mem\_to\_reg,EXE\_memwrite,EXE\_aluOp, EXE\_jal,EXE\_alua,EXE\_alub,EXE\_PC,EXE\_A,EXE\_B,EXE\_IMM32,EXE\_writereg\_num

);

input clk,rst,ID\_RegWrite,ID\_mem\_to\_reg,ID\_memwrite,ID\_jal,ID\_alua,ID\_alub;

input [4:0] ID\_writereg\_num;

input [3:0] ID\_aluOp;

input [31:0] ID\_PC,ID\_A,ID\_B,ID\_IMM32;

output reg EXE\_RegWrite,EXE\_mem\_to\_reg,EXE\_memwrite,EXE\_jal,EXE\_alua,EXE\_alub;

output reg [4:0] EXE\_writereg\_num;

output reg [3:0] EXE\_aluOp;

output reg [31:0] EXE\_PC,EXE\_A,EXE\_B,EXE\_IMM32;

always @(posedge clk,posedge rst)

begin

if(rst) begin

EXE\_A=0;

EXE\_B=0;

EXE\_IMM32=0;

EXE\_PC=0;

EXE\_RegWrite=0;

EXE\_aluOp=0;

EXE\_alua=0;

EXE\_alub=0;

EXE\_jal=0;

EXE\_mem\_to\_reg=0;

EXE\_memwrite=0;

EXE\_writereg\_num=0;

end

else begin

EXE\_A=ID\_A;

EXE\_B=ID\_B;

EXE\_IMM32=ID\_IMM32;

EXE\_PC=ID\_PC+4;

EXE\_RegWrite=ID\_RegWrite;

EXE\_aluOp=ID\_aluOp;

EXE\_alua=ID\_alua;

EXE\_alub=ID\_alub;

EXE\_jal=ID\_jal;

EXE\_mem\_to\_reg=ID\_mem\_to\_reg;

EXE\_memwrite=ID\_memwrite;

EXE\_writereg\_num=ID\_writereg\_num;

end

end

endmodule

## 6.12 MEM（EX/MEM段间寄存器）

储存EX/MEM级相关数据。

module MEM(clk,rst,EXE\_C,EXE\_RegWrite,EXE\_mem\_to\_reg,EXE\_memwrite,EXE\_B,EXE\_writereg\_num,

MEM\_C,MEM\_RegWrite,MEM\_mem\_to\_reg,MEM\_memwrite,MEM\_B,MEM\_writereg\_num,

);

input clk,rst,EXE\_RegWrite,EXE\_mem\_to\_reg,EXE\_memwrite;

input [31:0] EXE\_B,EXE\_C;

input [4:0] EXE\_writereg\_num;

output reg MEM\_RegWrite,MEM\_mem\_to\_reg,MEM\_memwrite;

output reg [31:0] MEM\_B,MEM\_C;

output reg [4:0] MEM\_writereg\_num;

always @(posedge clk,posedge rst)

begin

if(rst)

begin

MEM\_B=0;

MEM\_C=0;

MEM\_RegWrite=0;

MEM\_mem\_to\_reg=0;

MEM\_memwrite=0;

MEM\_writereg\_num=0;

end

else begin

MEM\_B=EXE\_B;

MEM\_C=EXE\_C;

MEM\_RegWrite=EXE\_RegWrite;

MEM\_mem\_to\_reg=EXE\_mem\_to\_reg;

MEM\_memwrite=EXE\_memwrite;

MEM\_writereg\_num=EXE\_writereg\_num;

end

end

endmodule

## 6.13 WB（MEM/WB段间寄存器）

储存MEM/WB级相关数据。

module WB(clk,rst,MEM\_RegWrite,MEM\_mem\_to\_reg,MEM\_read,MEM\_C,MEM\_writereg\_num,

WB\_RegWrite,WB\_mem\_to\_reg,WB\_read,WB\_C,WB\_writereg\_num

);

input clk,rst,MEM\_RegWrite,MEM\_mem\_to\_reg;

input [31:0] MEM\_C,MEM\_read;

input [4:0] MEM\_writereg\_num;

output reg WB\_RegWrite,WB\_mem\_to\_reg;

output reg [31:0] WB\_C,WB\_read;

output reg [4:0] WB\_writereg\_num;

always @(posedge clk,posedge rst)

begin

if(rst)

begin

WB\_C=0;

WB\_RegWrite=0;

WB\_mem\_to\_reg=0;

WB\_read=0;

WB\_writereg\_num=0;

end

else begin

WB\_C=MEM\_C;

WB\_RegWrite=MEM\_RegWrite;

WB\_mem\_to\_reg=MEM\_mem\_to\_reg;

WB\_read=MEM\_read;

WB\_writereg\_num=MEM\_writereg\_num;

end

end

endmodule

## 6.14 SCCPU（流水线CPU核心模块）

组合若干流水线CPU用到的元器件。

module sccpu( clk, rst, instr, readdata,PC, MemWrite, aluout, writedata, reg\_sel, reg\_data);

input clk;

input rst; // reset

input [31:0] instr; // instruction

input [31:0] readdata; // data from data memory

output [31:0] PC; // PC address

output MemWrite; // memory write

output [31:0] aluout; // ALU output

output [31:0] writedata; // data to data memory

input [4:0] reg\_sel; // register selection (for debug use)

output [31:0] reg\_data; // selected register data (for debug use)

wire [15:0] ID\_IMM16;

wire [31:0] ID\_IMM32,EXE\_IMM32;

wire [31:0] NPC;

wire [31:0] jpc,npc,bpc;

wire [31:0] ID\_INS;

wire [31:0] ID\_PC;

wire [31:0] ID\_A,ID\_B;

wire [31:0] ID\_reada,ID\_readb;

wire [4:0] ID\_writereg\_num;

wire [31:0] WB\_writereg;//写向寄存器的数据

wire [5:0] ID\_Op, ID\_Funct;

wire [4:0] ID\_rs,ID\_rt,ID\_rd,EXE\_writereg\_num,MEM\_writereg\_num;

wire ID\_Zero,EXE\_RegWrite,MEM\_RegWrite,EXE\_mem\_to\_reg,MEM\_mem\_to\_reg;

wire ID\_RegWrite,ID\_mem\_to\_reg,ID\_memwrite,ID\_jal,ID\_extOp,ID\_alua,ID\_alub,ID\_nostall;

wire [1:0] ID\_npcOp,ID\_forwarda,ID\_forwardb,ID\_writereg\_to\_rt;

wire [3:0] ID\_aluOp;

wire EXE\_memwrite,EXE\_jal,EXE\_alua,EXE\_alub;

wire [3:0] EXE\_aluOp;

wire [31:0] EXE\_PC,EXE\_A,EXE\_B,EXE\_C;

wire WB\_RegWrite,WB\_mem\_to\_reg;

wire [31:0] WB\_C,WB\_read;

wire [4:0] WB\_writereg\_num;

wire [31:0] ALU\_A,ALU\_B,ALU\_C;

//用于更新PC，IF级作用

PC U\_PC(.rst(rst), .IF\_PC(PC),.clk(clk),.NPC(NPC),.ID\_nostall(ID\_nostall));

//此时将jal，和j指令集合到了一起。但是j是跳转指令低26，而jal仅仅16位，但是此次试验PC地址不会太大，故此处可暂时将

assign jpc={ID\_PC[31:26],ID\_INS[25:0],2'b00};

//用于寄存器跳转，主要有jalr，jr

assign npc=ID\_A;

//用于bne，beq跳转

assign bpc=ID\_PC+{14'b0,ID\_IMM16,2'b00};

//切分指令段

assign ID\_Op=ID\_INS[31:26];

assign ID\_Funct=ID\_INS[5:0];

assign ID\_rs=ID\_INS[25:21];

assign ID\_rt=ID\_INS[20:16];

assign ID\_rd=ID\_INS[15:11];

assign ID\_IMM16=ID\_INS[15:0];

//用于bne,beq的ID级判断

assign ID\_Zero=(ID\_A ==ID\_B)?1'b1:1'b0;

//ID级更新作用

ID U\_ID(.clk(clk),.rst(rst),.IF\_INS(instr),.IF\_PC(PC),.ID\_INS(ID\_INS),.ID\_PC(ID\_PC),.ID\_nostall(ID\_nostall));

//控制器，核心部件

ctrl U\_CTRL(.ID\_Op(ID\_Op), .ID\_Funct(ID\_Funct),.ID\_rs(ID\_rs),.ID\_rt(ID\_rt),.ID\_Zero(ID\_Zero),

.EXE\_RegWrite(EXE\_RegWrite),.MEM\_RegWrite(MEM\_RegWrite),.EXE\_mem\_to\_reg(EXE\_mem\_to\_reg),.MEM\_mem\_to\_reg(MEM\_mem\_to\_reg),.EXE\_writereg\_num(EXE\_writereg\_num),.MEM\_writereg\_num(MEM\_writereg\_num),

.ID\_RegWrite(ID\_RegWrite),.ID\_mem\_to\_reg(ID\_mem\_to\_reg),.ID\_writereg\_to\_rt(ID\_writereg\_to\_rt),.ID\_memwrite(ID\_memwrite),

.ID\_extOp(ID\_extOp), .ID\_aluOp(ID\_aluOp), .ID\_npcOp(ID\_npcOp),.ID\_jal(ID\_jal),.ID\_alua(ID\_alua),.ID\_alub(ID\_alub),.ID\_nostall(ID\_nostall),.ID\_forwarda(ID\_forwarda),.ID\_forwardb(ID\_forwardb)

);

//决定下一个pc信号，取决于控制器的输出

mux4 #(32) U\_MUX4\_NPC (.d0(PC+4),.d1(bpc),.d2(npc),.d3(jpc),.s(ID\_npcOp),.y(NPC));

//决定是否进行立即数的有符号数的拓展

EXT U\_EXT(.Imm16(ID\_IMM16),.EXTOp(ID\_extOp),.Imm32(ID\_IMM32));

//决定是否进行旁路，进行旁路进行什么旁路

mux4 #(32) U\_MUX4\_FOEWARDA (.d0(ID\_reada),.d1(EXE\_C),.d2(aluout),.d3(readdata),.s(ID\_forwarda),.y(ID\_A));

//决定是否进行旁路，进行旁路进行什么旁路

mux4 #(32) U\_MUX4\_FOEWARDB (.d0(ID\_readb),.d1(EXE\_C),.d2(aluout),.d3(readdata),.s(ID\_forwardb),.y(ID\_B));

//决定写入寄存器的内容的来源，是EXE级计算出的，还是MEM级读出来的

mux2 #(32) U\_MUX2\_WRITEREG (.d0(WB\_C),.d1(WB\_read), .s(WB\_mem\_to\_reg), .y(WB\_writereg));

//寄存器模块

RF U\_RF(.clk(clk),.rst(rst),.RFWr(WB\_RegWrite),.A1(ID\_rs),.A2(ID\_rt),.A3(WB\_writereg\_num),.WD(WB\_writereg),.RD1(ID\_reada),.RD2(ID\_readb),.reg\_sel(reg\_sel),.reg\_data(reg\_data));

//决定写入的寄存器号的来源，31号还是rd段或者rt段

mux4 #(5) U\_MUX4\_RT\_RD (.d0(ID\_rd),.d1(ID\_rt),.d2(5'b11111),.d3(5'b0),.s(ID\_writereg\_to\_rt),.y(ID\_writereg\_num));

//EXE级更新作用

EXE U\_EXE(.clk(clk),.rst(rst),.ID\_RegWrite(ID\_RegWrite),.ID\_mem\_to\_reg(ID\_mem\_to\_reg),.ID\_memwrite(ID\_memwrite),.ID\_aluOp(ID\_aluOp),. ID\_jal( ID\_jal),.ID\_alua(ID\_alua),.ID\_alub(ID\_alub),.ID\_PC(ID\_PC),.ID\_A(ID\_A),.ID\_B(ID\_B),.ID\_IMM32(ID\_IMM32),.ID\_writereg\_num(ID\_writereg\_num),

.EXE\_RegWrite(EXE\_RegWrite),.EXE\_mem\_to\_reg(EXE\_mem\_to\_reg),.EXE\_memwrite(EXE\_memwrite),.EXE\_aluOp(EXE\_aluOp),.EXE\_jal(EXE\_jal),.EXE\_alua(EXE\_alua),.EXE\_alub(EXE\_alub),.EXE\_PC(EXE\_PC),.EXE\_A(EXE\_A),.EXE\_B(EXE\_B),.EXE\_IMM32(EXE\_IMM32),.EXE\_writereg\_num(EXE\_writereg\_num)

);

//MEM级更新作用

MEM U\_MEM(.clk(clk),.rst(rst),.EXE\_C(EXE\_C),.EXE\_RegWrite(EXE\_RegWrite),.EXE\_mem\_to\_reg(EXE\_mem\_to\_reg),.EXE\_memwrite(EXE\_memwrite),.EXE\_B(EXE\_B),.EXE\_writereg\_num(EXE\_writereg\_num),

.MEM\_C(aluout),.MEM\_RegWrite(MEM\_RegWrite),.MEM\_mem\_to\_reg(MEM\_mem\_to\_reg),.MEM\_memwrite(MemWrite),.MEM\_B(writedata),.MEM\_writereg\_num(MEM\_writereg\_num)

);

//WB级更新作用

WB U\_WB(.clk(clk),.rst(rst),.MEM\_RegWrite(MEM\_RegWrite),.MEM\_mem\_to\_reg(MEM\_mem\_to\_reg),.MEM\_read(readdata),.MEM\_C(aluout),.MEM\_writereg\_num(MEM\_writereg\_num),

.WB\_RegWrite(WB\_RegWrite),.WB\_mem\_to\_reg(WB\_mem\_to\_reg),.WB\_read(WB\_read),.WB\_C(WB\_C),.WB\_writereg\_num(WB\_writereg\_num)

);

//决定ALUC是来自PC还是来自ALU计算的结果

mux2 #(32) U\_MUX2\_ALUC (.d0(ALU\_C),.d1(EXE\_PC),.s(EXE\_jal),.y(EXE\_C));

//决定ALUa是来自ID级的旁路选择器的结果还是来自立即数

mux2 #(32) U\_MUX2\_ALUA (.d0(EXE\_A),.d1(EXE\_IMM32),.s(EXE\_alua),.y(ALU\_A));

//决定ALUb是来自ID级的旁路选择器的结果还是来自立即数

mux2 #(32) U\_MUX2\_ALUB (.d0(EXE\_B),.d1(EXE\_IMM32),.s(EXE\_alub),.y(ALU\_B));

//进行ALU运算

alu U\_ALU(.A(ALU\_A),.B(ALU\_B),.ALUOp(EXE\_aluOp),.C(ALU\_C));

endmodule

## 6.15 SCCOMP（顶层模块）

完整的流水线CPU。

`timescale 1ns/1ns

module sccomp(clk, rstn, reg\_sel);

input clk;

input rstn;

input [4:0] reg\_sel;

output [7:0] seg;

output [7:0] dig;

wire [31:0] instr;

wire [31:0] PC;

wire MemWrite;

wire [31:0] dm\_addr, dm\_din, dm\_dout;

wire rst = ~rstn;

wire [31:0] reg\_data;

// instantiation of single-cycle CPU

sccpu U\_SCCPU(

.clk(clk), // input: cpu clock

.rst(rst), // input: reset

.instr(instr), // input: instruction

.readdata(dm\_dout), // input: data to cpu

.MemWrite(MemWrite), // output: memory write signal

.PC(PC), // output: PC

.aluout(dm\_addr), // output: address from cpu to memory

.writedata(dm\_din), // output: data from cpu to memory

.reg\_sel(reg\_sel), // input: register selection

.reg\_data(reg\_data) // output: register data

);

// instantiation of data memory

dm U\_DM(

.clk(clk), // input: cpu clock

.DMWr(MemWrite), // input: ram write

.addr(dm\_addr[8:2]), // input: ram address

.din(dm\_din), // input: data to ram

.dout(dm\_dout) // output: data from ram

);

// instantiation of intruction memory (used for simulation)

im U\_IM (

.addr(PC[8:2]), // input: rom address

.dout(instr) // output: instruction

);

endmodule

## 6.16 ctrl\_encode\_def（宏定义模块）

定义相关控制信号的宏。

// NPC control signal

`define NPC\_PLUS4 2'b00

`define NPC\_BPC 2'b01

`define NPC\_JPC 2'b10

`define NPC\_RPC 2'b11

// ALU control signal

`define ALU\_NOP 4'b0000

`define ALU\_ADD 4'b0001

`define ALU\_SUB 4'b0010

`define ALU\_AND 4'b0011

`define ALU\_OR 4'b0100

`define ALU\_SLT 4'b0101

`define ALU\_SLTU 4'b0110

`define ALU\_SLL 4'b0111

`define ALU\_SRL 4'b1000

`define ALU\_NOR 4'b1001

`define ALU\_SLLV 4'b1010

`define ALU\_SRLV 4'b1011

`define ALU\_LUI 4'b1100

# 7 测试及结果分析

## 7.1 仿真代码及分析

机器代码 汇编代码 PC

20020005 main: addi $2, $0, 5 0

3403000c ori $3, $0, 12 4

00620823 subu $1, $3, $2 8

00013842 srl $7, $1, 1 12

08000019 call\_a: j a 16

00000000 nop 20

00e22025 or $4, $7, $2 24

00642824 and $5, $3, $4 28

00a42820 add $5, $5, $4 32

10a7001f beq $5, $7, end 36

00000000 nop 40

0064202b sltu $4, $3, $4 44

10800004 beq $4, $0, around 48

00000000 nop 52

20050000 addi $5, $0, 0 56

20050000 addi $5, $0, 0 60

20050000 addi $5, $0, 0 64

00e2202a around: slt $4, $7, $2 68

00853821 addu $7, $4, $5 72

00e23822 sub $7, $7, $2 76

ac670044 sw $7, 68($3) 80

8c020050 lw $2, 80($0) 84

08000029 j end 88

00000000 nop 92

20020001 addi $2, $0, 1 96

00073880 a: sll $7, $7, 2 100

0c00001f call\_b: jal b 104

00000000 nop 108

201f0018 addi $31,$0,24 112

03e00008 jr $31 116

00000000 nop 120

3c01ffaa b: lui $1, 0xFFAA 124

00e1082a slt $1, $7, $1 128

14200007 bne $1, $0, end 132

00000000 nop 136

00e23822 sub $7, $7, $2 140

00073842 srl $7, $7, 1 144

00e10827 nor $1, $7, $1 148

0027082b sltu $1, $1, $7 152

03e00008 jr $31 156

00000000 nop 160

ac030054 end: sw $3, 84($0) 164

8c670048 lw $7, 72($3) 168

ac670044 sw $7, 68($3) 172

8c660044 lw $6, 68($3) 176

00c53020 add $6, $6, $5 180

0800002e loop: j loop 184

00000000 nop 188

## 7.2 仿真测试结果

**addi $2, $0, 5**

该指令仿真结果如图7.1所示，该指令执行结束后将2号寄存器的值变为5，执行后的PC、指令、寄存器状态如图7.2所示。由于该指令后的第二条指令为subu $1, $3, $2，有数据冒险，需要在MEM级旁路，故ID\_forwardb会变为10。

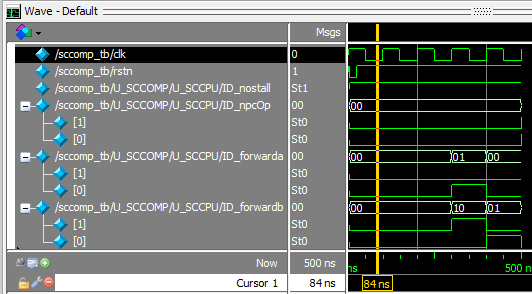


图7.1

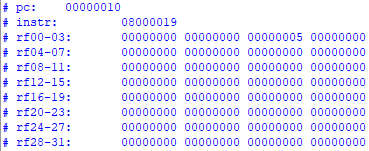


图7.2

**ori $3, $0, 12**

该指令仿真结果如图7.3所示，该指令执行结束后将3号寄存器的值变为12，执行后的PC、指令、寄存器状态如图7.4所示。由于该指令后的第一条指令为subu $1, $3, $2，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

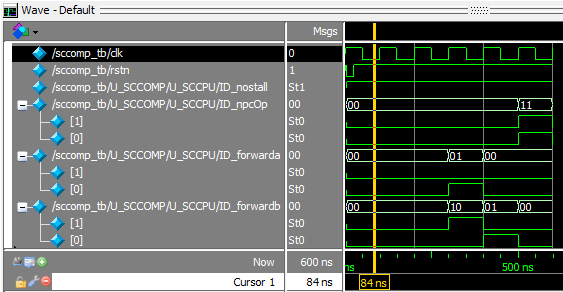


图7.3

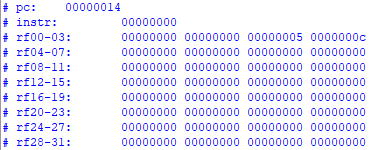


图7.4

**subu $1, $3, $2**

该指令仿真结果如图7.5所示，该指令执行结束后将1号寄存器的值变为7，执行后的PC、指令、寄存器状态如图7.6所示。由于该指令后的第一条指令为srl $7, $1, 1，有数据冒险，需要在EX级旁路，故ID\_forwardb会变为01。

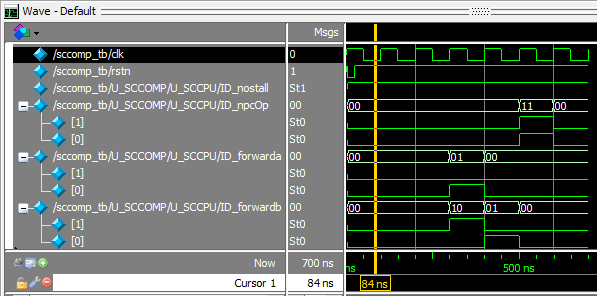


图7.5

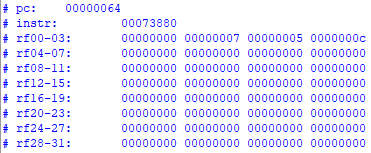


图7.6

**srl $7, $1, 1**

该指令仿真结果如图7.7所示，该指令执行结束后将7号寄存器的值变为3，执行后的PC、指令、寄存器状态如图7.8所示。该指令不存在冒险。

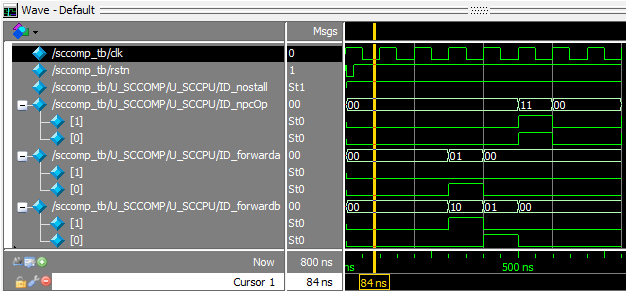


图7.7

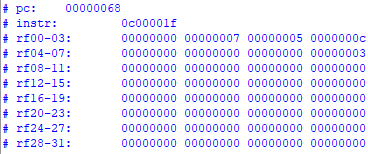


图7.8

**j a**

该指令仿真结果如图7.9所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.10所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为11。

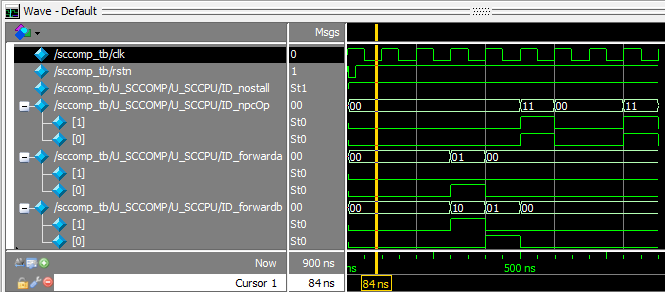


图7.9

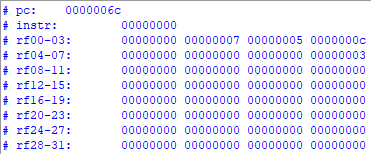


图7.10

**nop**

该指令仿真结果如图7.11所示，执行后的PC、指令、寄存器状态如图7.12所示。该指令是为了解决控制冒险而添加的。

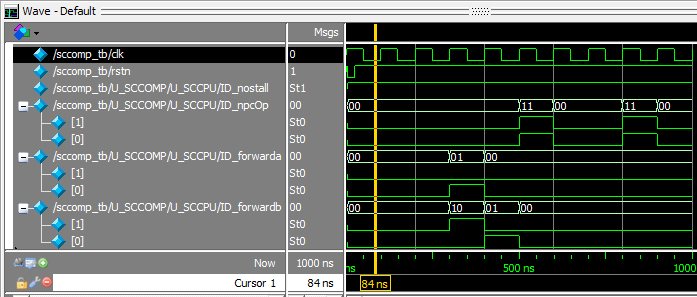


图7.11

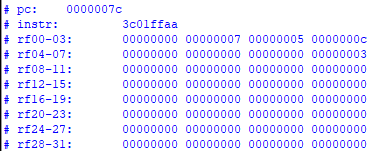


图7.12

**sll $7, $7, 2**

该指令仿真结果如图7.13所示，该指令执行结束后将7号寄存器的值变为12，执行后的PC、指令、寄存器状态如图7.14所示。该指令不存在冒险。

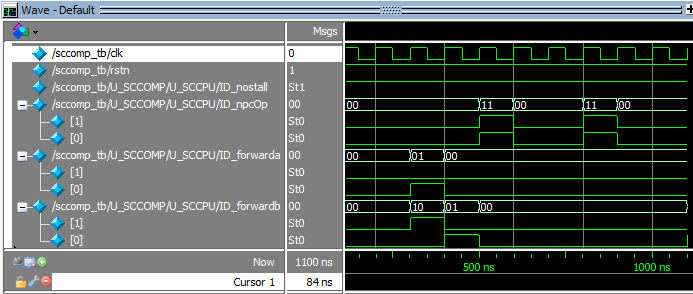


图7.13

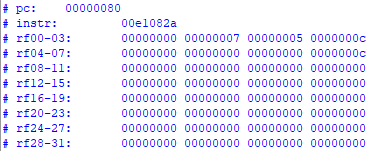


图7.14

**jal b**

该指令仿真结果如图7.15所示，该指令需要跳转，并将当前PC+4赋值给31号寄存器，执行后的PC、指令、寄存器状态如图7.16所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为11。

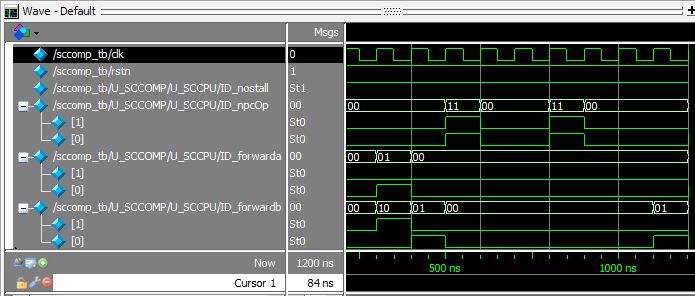


图7.15

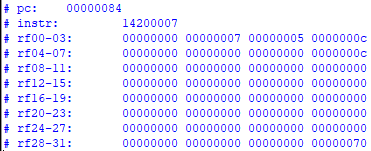


图7.16

**nop**

该指令仿真结果如图7.17所示，执行后的PC、指令、寄存器状态如图7.18所示。该指令是为了解决控制冒险而添加的。

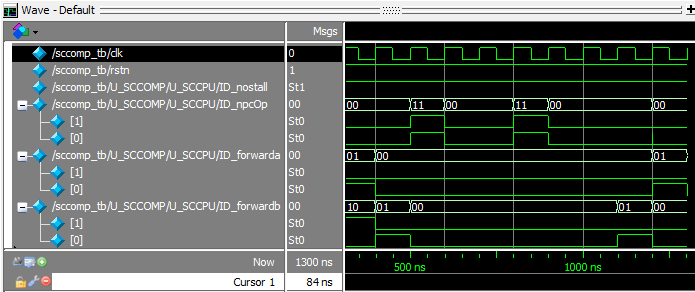


图7.17

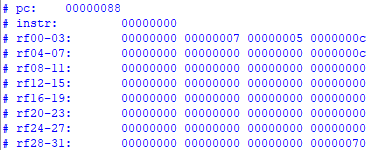


图7.18

**lui $1, 0xFFAA**

该指令仿真结果如图7.19所示，该指令执行结束后将1号寄存器的值变为0xFFAA0000，执行后的PC、指令、寄存器状态如图7.20所示。由于该指令后的第一条指令为slt $1, $7, $1，有数据冒险，需要在EX级旁路，故ID\_forwardb会变为01。

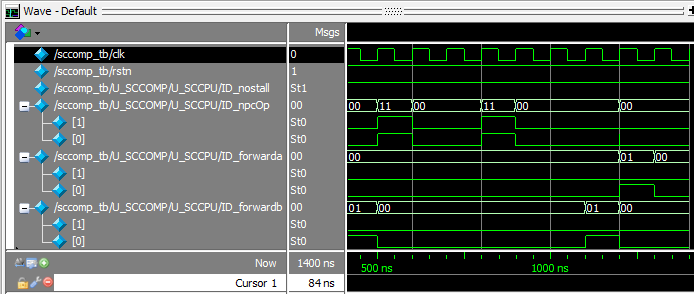


图7.19

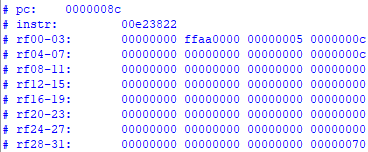


图7.20

**slt $1, $7, $1**

该指令仿真结果如图7.21所示，该指令执行结束后将1号寄存器的值变为0，执行后的PC、指令、寄存器状态如图7.22所示。由于该指令后的第一条指令为bne $1, $0, end，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

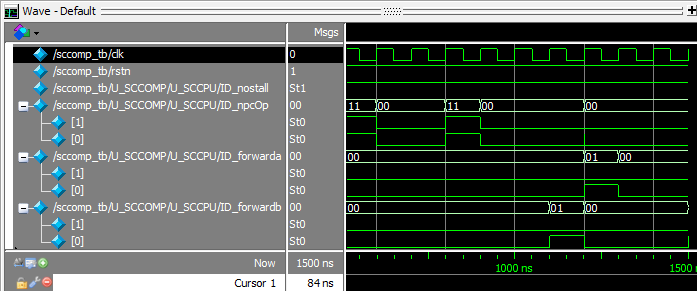


图7.21

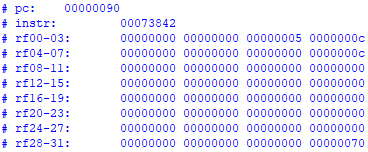


图7.22

**bne $1, $0, end**

该指令仿真结果如图7.23所示，该指令不需要跳转，执行后的PC、指令、寄存器状态如图7.24所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp为00。

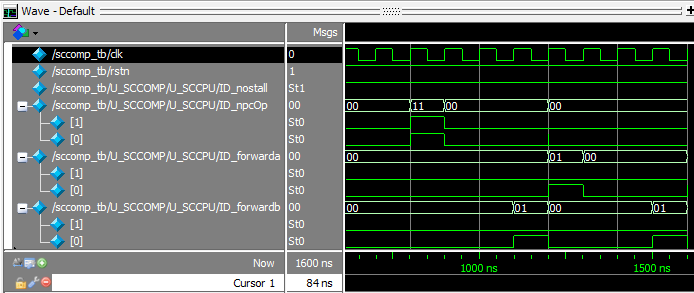


图7.23

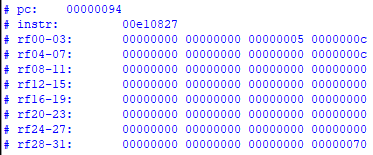


图7.24

**nop**

该指令仿真结果如图7.25所示，执行后的PC、指令、寄存器状态如图7.26所示。该指令是为了解决控制冒险而添加的。

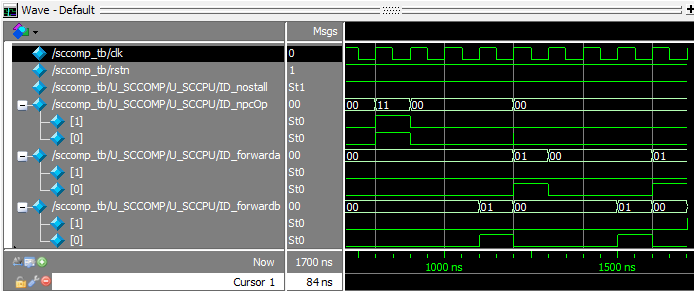


图7.25

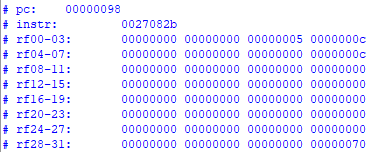


图7.26

**sub $7, $7, $2**

该指令仿真结果如图7.27所示，该指令执行结束后将7号寄存器的值变为7，执行后的PC、指令、寄存器状态如图7.28所示。由于该指令后的第一条指令为srl $7, $7, 1，有数据冒险，需要在EX级旁路，故ID\_forwardb会变为01。

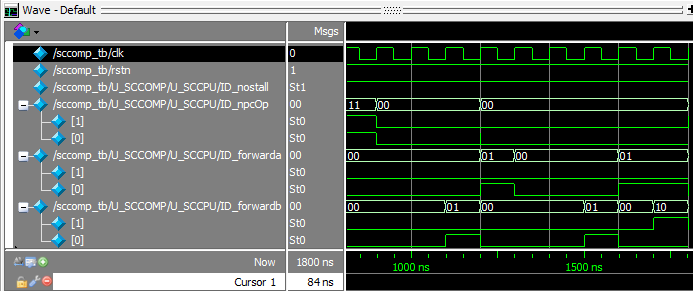


图7.27

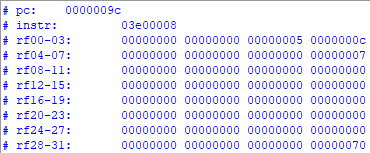


图7.28

**srl $7, $7, 1**

该指令仿真结果如图7.29所示，该指令执行结束后将7号寄存器的值变为3，执行后的PC、指令、寄存器状态如图7.30所示。由于该指令后的第一条指令为nor $1, $7, $1，第二条指令为sltu $1, $1, $7，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01，还需在MEM级旁路，故ID\_forwardb变为10。

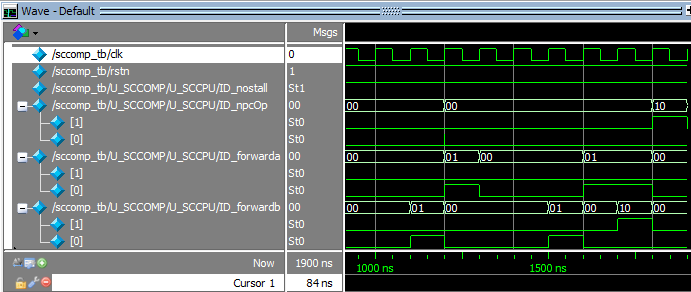


图7.29

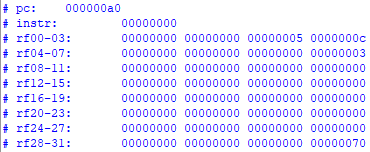


图7.30

**nor $1, $7, $1**

该指令仿真结果如图7.31所示，该指令执行结束后将1号寄存器的值变为0xFFFFFFFC，执行后的PC、指令、寄存器状态如图7.32所示。由于该指令后的第一条指令为sltu $1, $1, $7，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

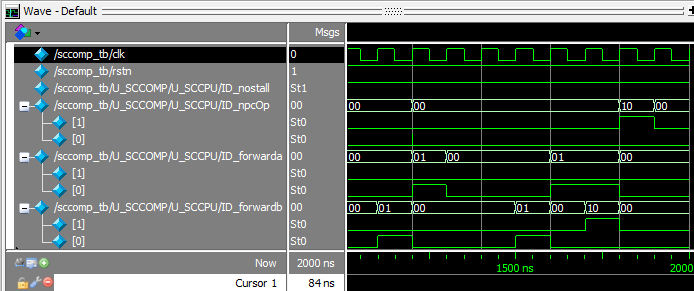


图7.31

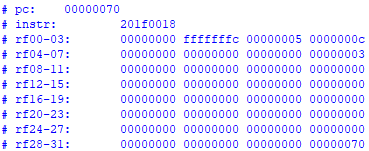


图7.32

**sltu $1, $1, $7**

该指令仿真结果如图7.33所示，该指令执行结束后将1号寄存器的值变为0，执行后的PC、指令、寄存器状态如图7.34所示。该指令不存在冒险。

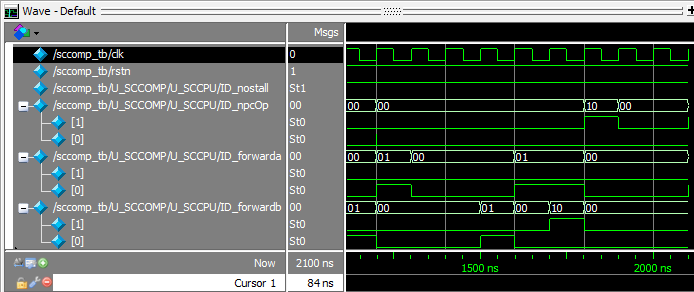


图7.33

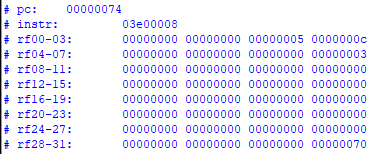
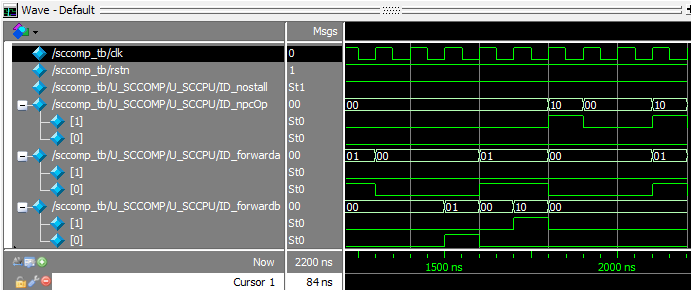


图7.34

**jr $31**

该指令仿真结果如图7.35所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.36所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为10。



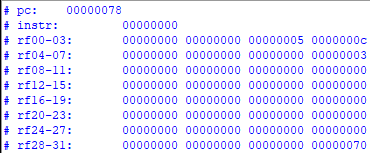
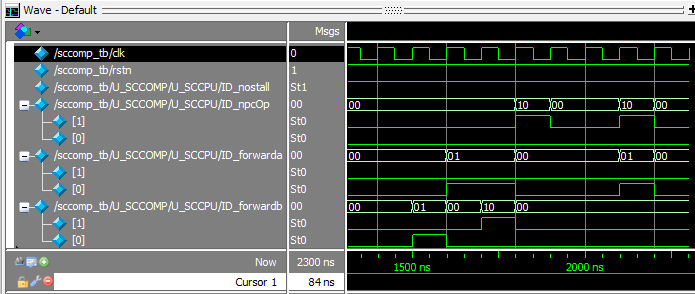
图7.35

图7.36

**nop**

该指令仿真结果如图7.37所示，执行后的PC、指令、寄存器状态如图7.38所示。该指令是为了解决控制冒险而添加的。



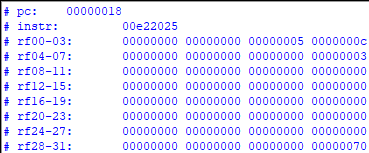
图7.37

图7.38

**addi $31,$0,24**

该指令仿真结果如图7.39所示，该指令执行结束后将31号寄存器的值变为24，执行后的PC、指令、寄存器状态如图7.40所示。由于该指令后的第一条指令为jr $31，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

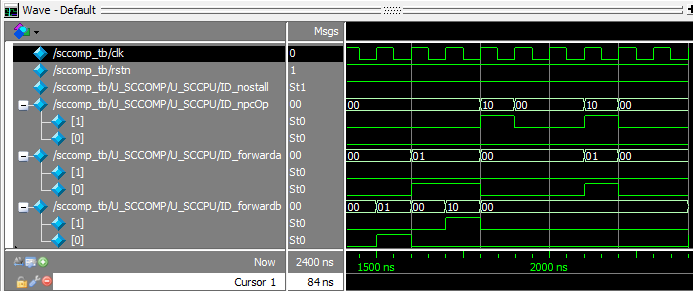


图7.39

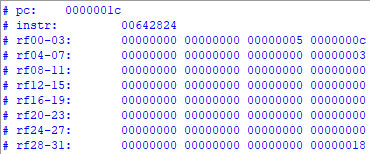


图7.40

**jr $31**

该指令仿真结果如图7.41所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.42所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为10。

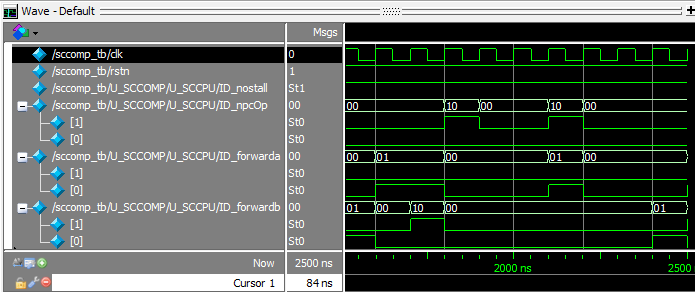


图7.41

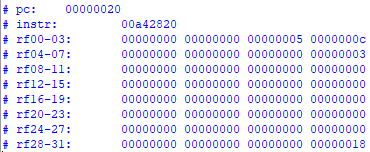


图7.42

**nop**

该指令仿真结果如图7.43所示，执行后的PC、指令、寄存器状态如图7.44所示。该指令是为了解决控制冒险而添加的。

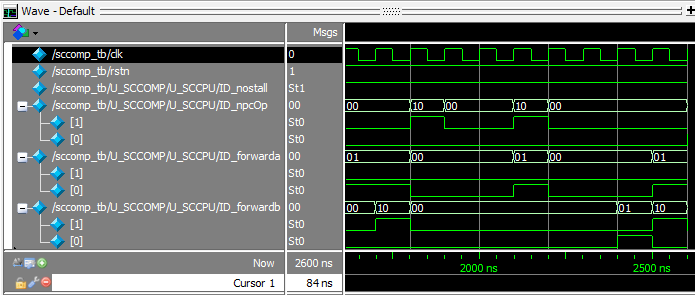


图7.43

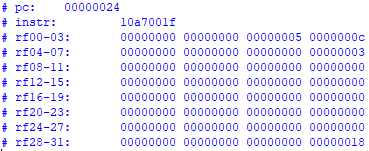


图7.44

**or $4, $7, $2**

该指令仿真结果如图7.45所示，该指令执行结束后将4号寄存器的值变为7，执行后的PC、指令、寄存器状态如图7.46所示。由于该指令后的第一条指令为and $5, $3, $4，第二条指令为add $5, $5, $4，有数据冒险，需要在EX级旁路，故ID\_forwardb会变为01，需要在MEM级旁路，故ID\_forwardb会变为10。

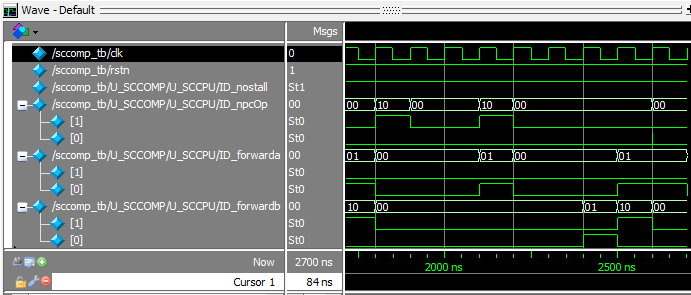


图7.45

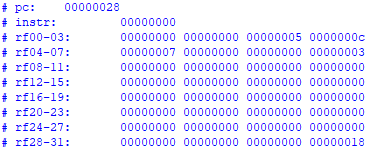


图7.46

**and $5, $3, $4**

该指令仿真结果如图7.47所示，该指令执行结束后将5号寄存器的值变为4，执行后的PC、指令、寄存器状态如图7.48所示。由于该指令后的第一条指令为add $5, $5, $4，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

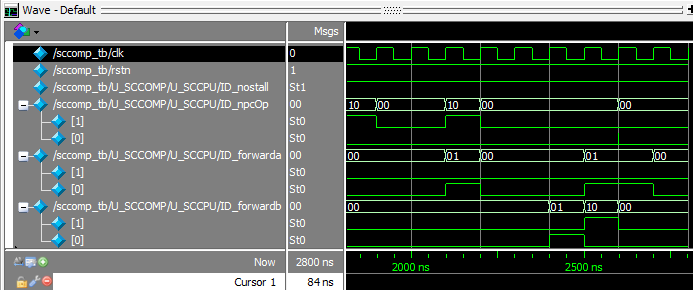


图7.47

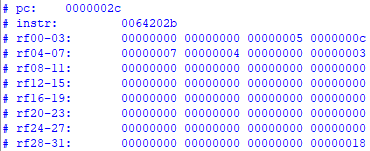


图7.48

**add $5, $5, $4**

该指令仿真结果如图7.49所示，该指令执行结束后将5号寄存器的值变为11，执行后的PC、指令、寄存器状态如图7.50所示。由于该指令后的第一条指令为beq $5, $7, end，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

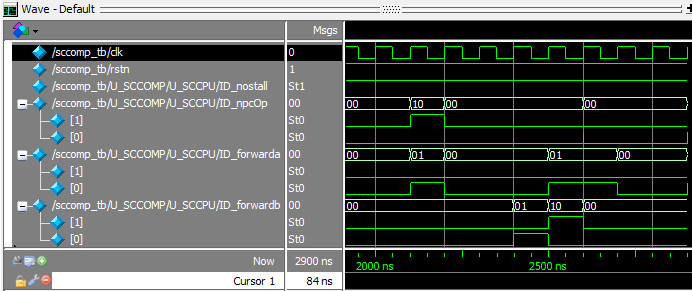


图7.49

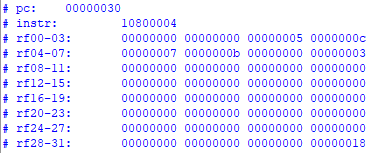


图7.50

**beq $5, $7, end**

该指令仿真结果如图7.51所示，该指令不需要跳转，执行后的PC、指令、寄存器状态如图7.52所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp为00。

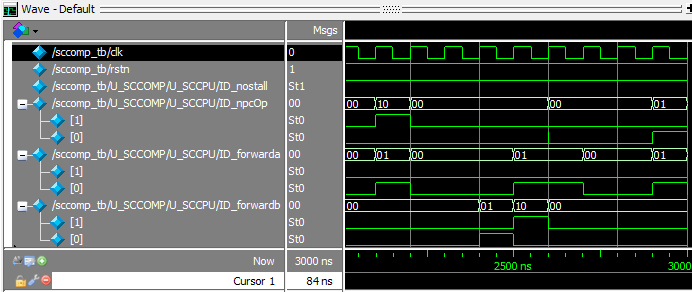


图7.51

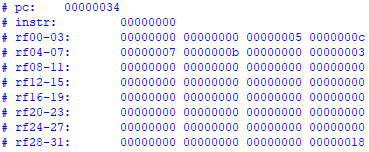


图7.52

**nop**

该指令仿真结果如图7.53所示，执行后的PC、指令、寄存器状态如图7.54所示。该指令是为了解决控制冒险而添加的。

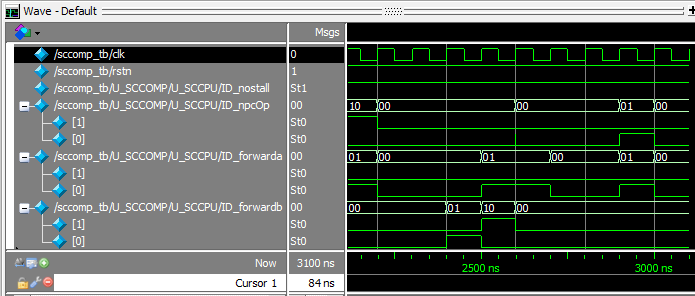


图7.53

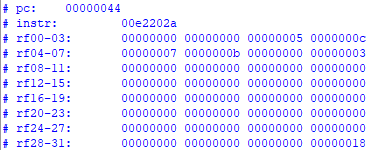


图7.54

**sltu $4, $3, $4**

该指令仿真结果如图7.55所示，该指令执行结束后将4号寄存器的值变为0，执行后的PC、指令、寄存器状态如图7.56所示。由于该指令后的第一条指令为beq $4, $0, around，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

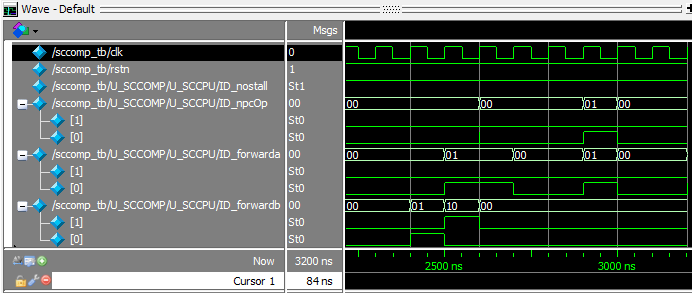


图7.55

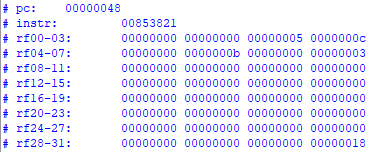


图7.56

**beq $4, $0, around**

该指令仿真结果如图7.57所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.58所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp为01。

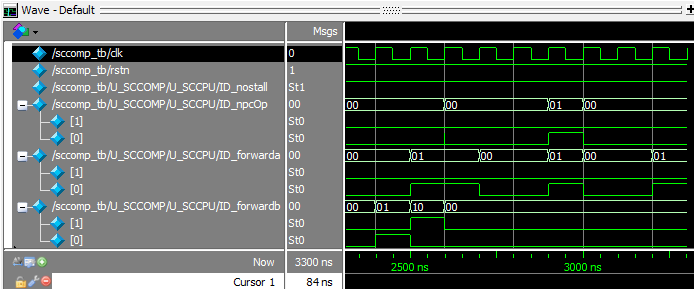


图7.57

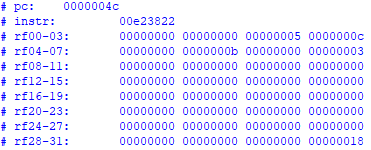


图7.58

**nop**

该指令仿真结果如图7.59所示，执行后的PC、指令、寄存器状态如图7.60所示。该指令是为了解决控制冒险而添加的。

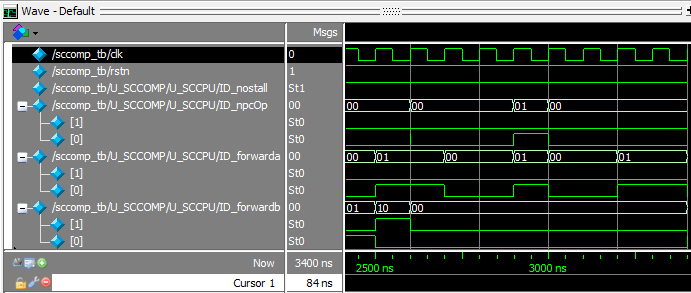


图7.59

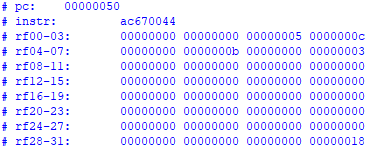


图7.60

**slt $4, $7, $2**

该指令仿真结果如图7.61所示，该指令执行结束后将4号寄存器的值变为1，执行后的PC、指令、寄存器状态如图7.62所示。由于该指令后的第一条指令为addu $7, $4, $5，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

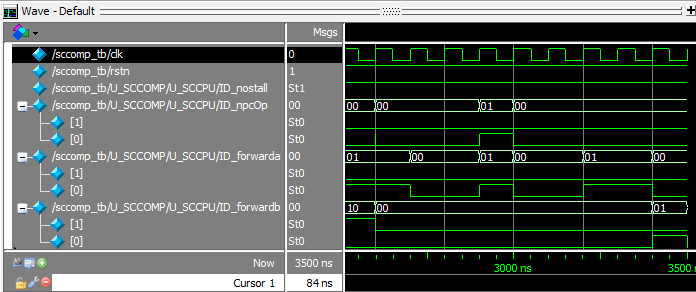


图7.61

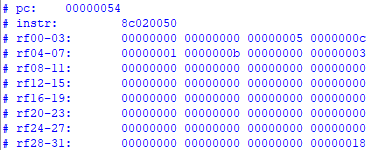


图7.62

**addu $7, $4, $5**

该指令仿真结果如图7.63所示，该指令执行结束后将7号寄存器的值变为12，执行后的PC、指令、寄存器状态如图7.64所示。由于该指令后的第一条指令为sub $7, $7, $2，有数据冒险，需要在EX级旁路，故ID\_forwarda会变为01。

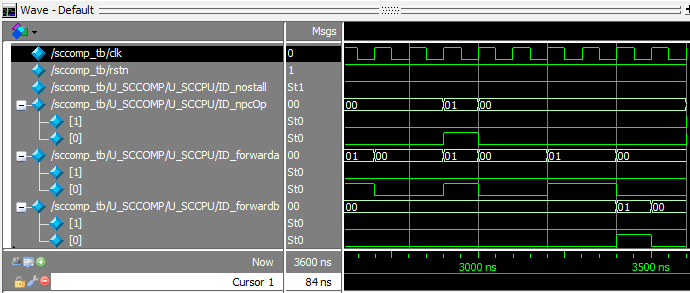


图7.63

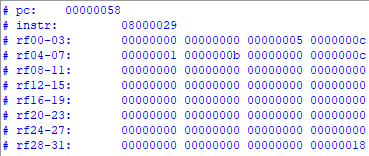


图7.64

**sub $7, $7, $2**

该指令仿真结果如图7.65所示，该指令执行结束后将7号寄存器的值变为7，执行后的PC、指令、寄存器状态如图7.66所示。由于该指令后的第一条指令为sw $7, 68($3)，有数据冒险，需要在EX级旁路，故ID\_forwardb会变为01。

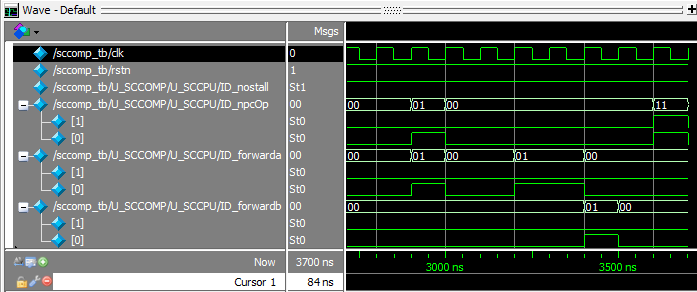


图7.65

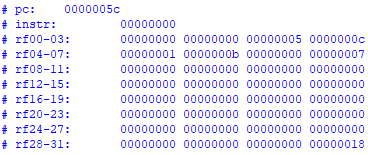
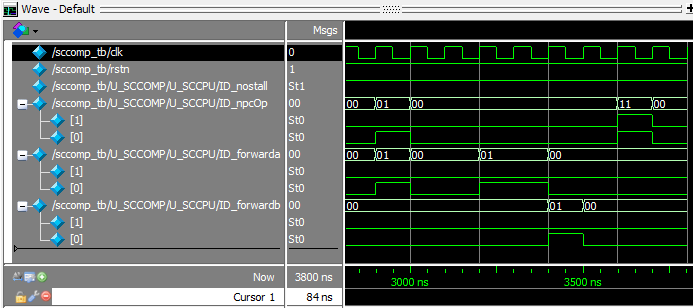


图7.66

**sw $7, 68($3)**

该指令仿真结果如图7.67所示，该指令执行结束后将7号寄存器的值存至地址为80的存储器，执行后的PC、指令、寄存器状态如图7.68所示。该指令不存在冒险。



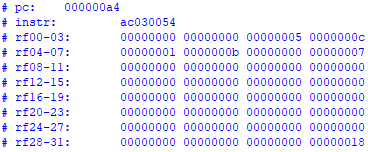
图7.67

图7.68

**lw $2, 80($0)**

该指令仿真结果如图7.69所示，该指令执行结束后将2号寄存器的值变为7，执行后的PC、指令、寄存器状态如图7.70所示。该指令不存在冒险。

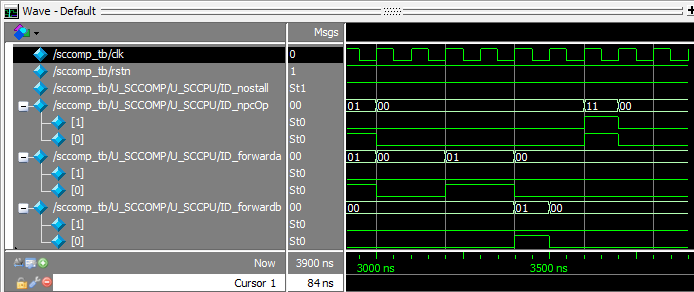


图7.69

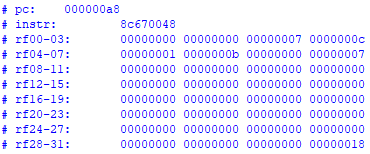


图7.70

**j end**

该指令仿真结果如图7.71所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.72所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为11。

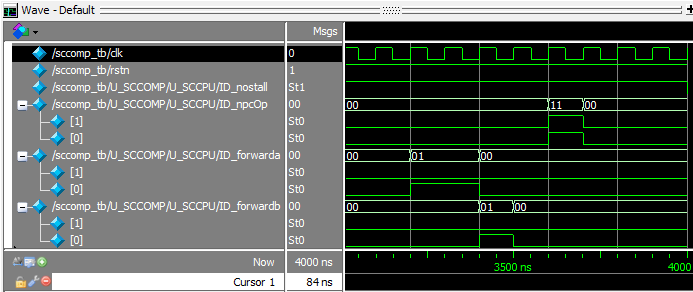


图7.71

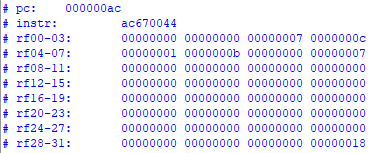


图7.72

**nop**

该指令仿真结果如图7.73所示，执行后的PC、指令、寄存器状态如图7.74所示。该指令是为了解决控制冒险而添加的。

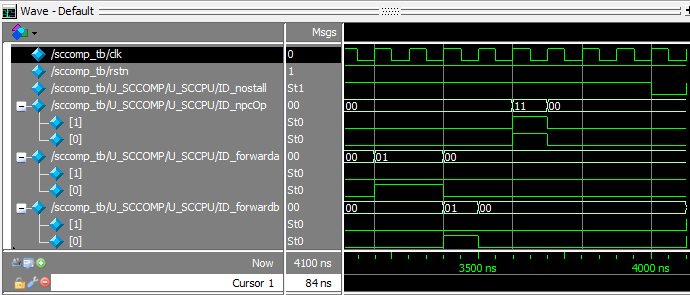


图7.73

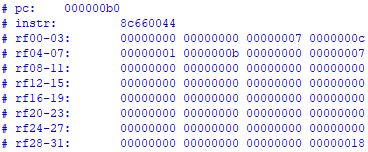


图7.74

**sw $3, 84($0)**

该指令仿真结果如图7.75所示，该指令执行结束后将3号寄存器的值存至地址为84的存储器，执行后的PC、指令、寄存器状态如图7.76所示。该指令不存在冒险。

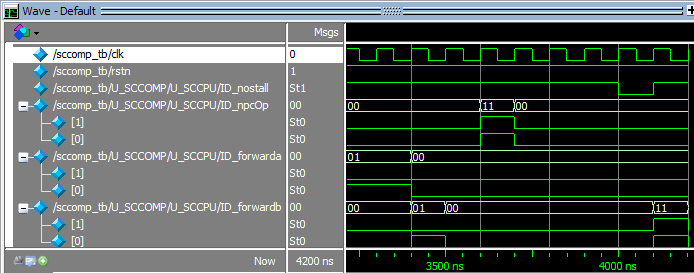


图7.75

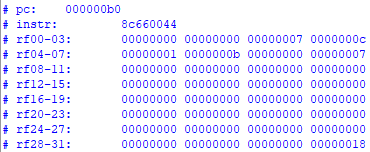


图7.76

**lw $7, 72($3)**

该指令仿真结果如图7.77所示，该指令执行结束后将7号寄存器的值变为12，执行后的PC、指令、寄存器状态如图7.78所示。该指令不存在冒险。由于该指令后的第一条指令为sw $7, 68($3)，有数据冒险，且是lw型冒险，故ID\_forwardb会变为11，且会阻塞一个时钟周期。

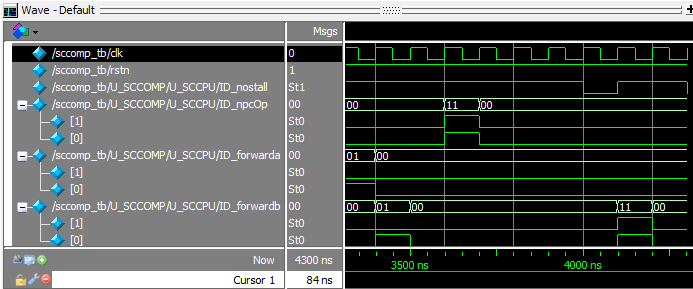


图7.77

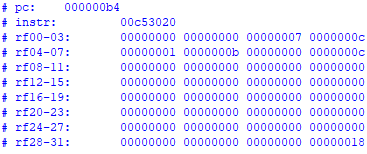


图7.78

**sw $7, 68($3)**

该指令仿真结果如图7.79所示，该指令执行结束后将7号寄存器的值存至地址为80的存储器，执行后的PC、指令、寄存器状态如图7.80所示。该指令不存在冒险。

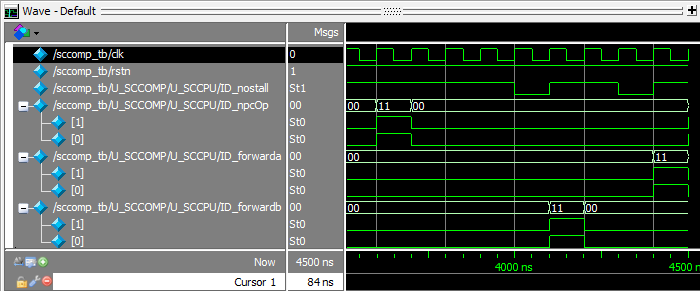


图7.79

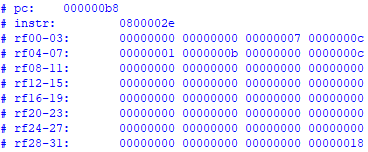


图7.80

**lw $6, 68($3)**

该指令仿真结果如图7.81所示，该指令执行结束后将7号寄存器的值变为12，执行后的PC、指令、寄存器状态如图7.82所示。由于该指令后的第一条指令为add $6, $6, $5，有数据冒险，且是lw型冒险，故ID\_forwarda会变为11，且会阻塞一个时钟周期。

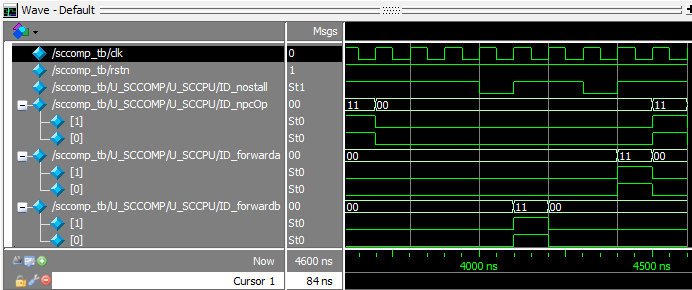


图7.81

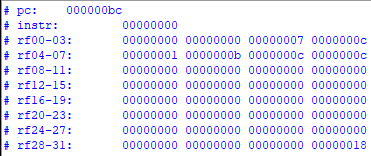


图7.82

**add $6, $6, $5**

该指令仿真结果如图7.83所示，该指令执行结束后将6号寄存器的值变为23，执行后的PC、指令、寄存器状态如图7.84所示。该指令不存在冒险。

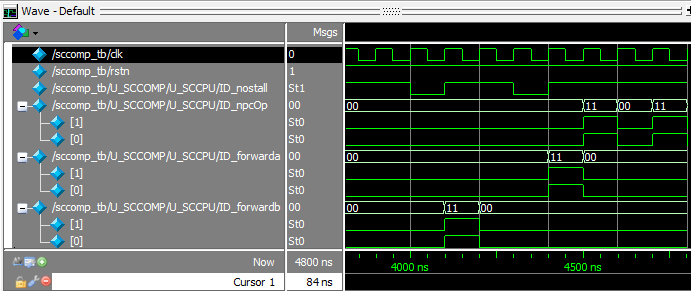


图7.83

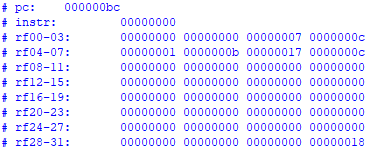


图7.84

**j loop**

该指令仿真结果如图7.85所示，该指令需要跳转，执行后的PC、指令、寄存器状态如图7.86所示。该指令有控制冒险，故在该指令后有一空指令，ID\_npcOp变为11。从此开始陷入死循环，即程序结束。

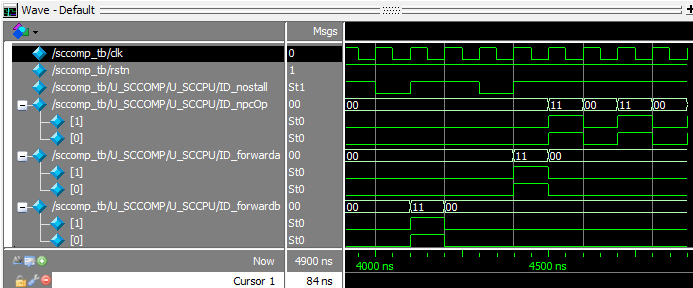


图7.85

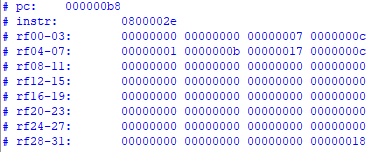


图7.86

## 7.3 下载测试代码及分析

机器代码 汇编代码 PC

3c020150 lui $2, 0x0150 0

34420196 ori $2, $2, 0x0196 4

ac020000 sw $2, 0($0) 8

200b0008 addi $11, $0, 8 12

8c010000 lw $1, 0x0($0) 16

00001020 add $2, $0, $0 20

2004000f addi $4, $0, 0x0f 24

00243824 loop1: and $7, $1, $4 28

00024880 sll $9, $2, 2 32

01273806 srlv $7, $7, $9 36

00042900 sll $5, $4, 4 40

00406020 add $12, $2, $0 44

00e06820 add $13, $7, $0 48

20430001 addi $3, $2, 1 52

106b000d loop2: beq $3, $11, checkswap 56

00000000 nop 60

00254024 and $8, $1, $5 64

00035080 sll $10, $3, 2 68

01484006 srlv $8, $8, $10 72

01a8702a slt $14, $13, $8 76

11c00003 beq $14, $0, incrLoop2 80

00000000 nop 84

01006820 add $13, $8, $0 88

00606020 add $12, $3, $0 92

00052900 incrLoop2: sll $5, $5, 4 96

20630001 addi $3, $3, 1 100

0800000e j loop2 104

00000000 nop 108

004c702a checkswap: slt $14, $2, $12 112

11c00003 beq $14, $0, incrLoop1 116

00000000 nop 120

0c000034 jal swap 124

00000000 nop 128

00042100 incrLoop1: sll $4, $4, 4 132

20420001 addi $2, $2, 1 136

144bffe3 bne $2, $11, loop1 140

00000000 nop 144

ac010004 result: sw $1, 0x04($0) 148

08000c25 j result 152

00000000 nop 156

2005000f swap: addi $5, $0, 0x0f 160

000c5080 sll $10, $12, 2 164

01452804 sllv $5, $5, $10 168

00853025 or $6, $4, $5 172

00c03027 nor $6, $6, $0 176

00260824 and $1, $1, $6 180

012d4004 sllv $8, $13,$9 184

00280825 or $1, $1, $8 188

01473804 sllv $7, $7, $10 192

00270825 or $1, $1, $7 196

03e00008 jr $31 200

00000000 nop 204

这段指令将一个八位十六进制数的各位升序排列，最初的八位十六进制数为0x01500196，排序后的八位十六进制数应为0x00011569。

## 7.4 下载测试结果

将上述指令下板测试，得到的结果如图7.87所示。

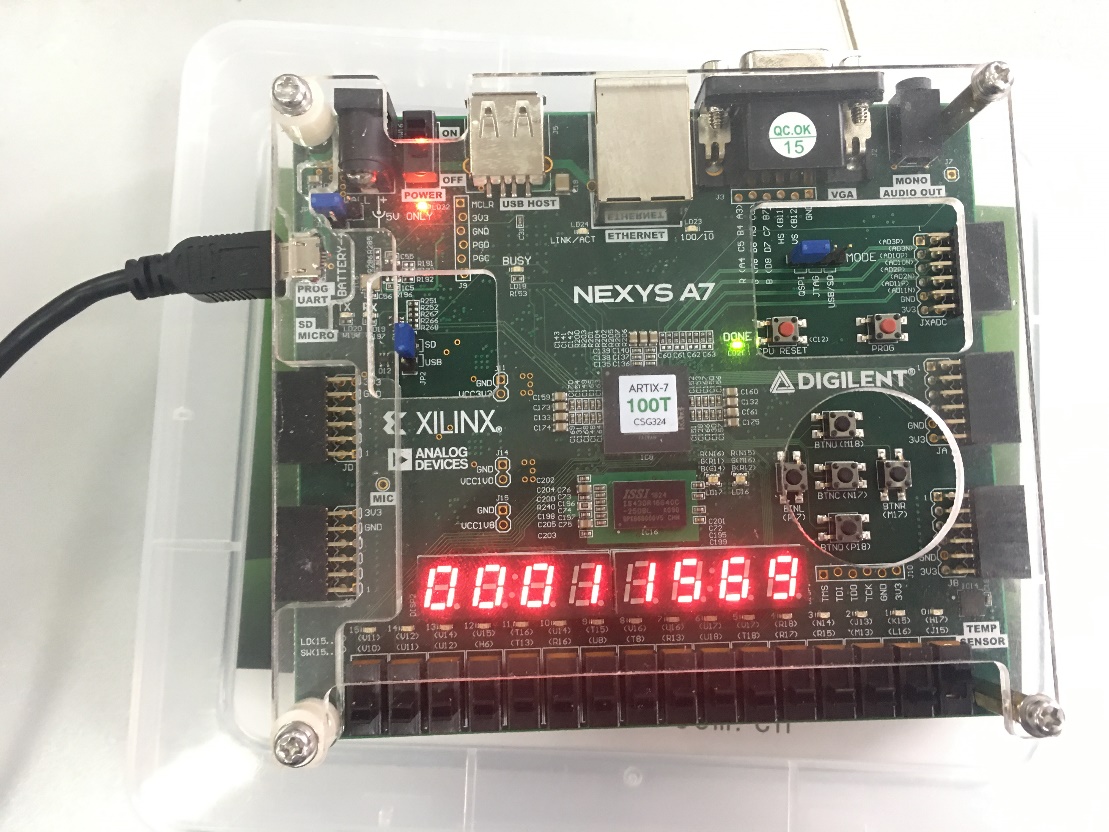


图7.87 下载测试结果

可以看到排序后的学号输出，测试结果正确。

# 8 实验心得

在该实验中我收获了许多有价值的知识，也克服了许多问题，实验心得如下：

（1）深入理解了单周期CPU中各个部件起到的作用；

（2）认识了流水线对于增强CPU工作效率的价值；

（3）在编写CPU的Ctrl单元时，要处理各种各样的逻辑关系，尤其是涉及到流水线CPU的相关控制，编写时需要十分谨慎，反复检查和试验；

（4）对于数据冒险和控制冒险，要考虑到发生冒险的各种情况，合理地选择旁路；

（5）掌握了仿真代码的流程，能够从代码中预测仿真的结果，也能从仿真的结果解释信号变化的原因；

（6）在将程序烧入开发板时，遇到了开发板数码管无响应的情况，经过检查发现仿真用的延迟代码在下板时是无效的，需要用循环强行添加延迟，从而让数码管显示数据。

# 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**].清华大学出版社，2018年8月

[2] 王党辉，康继昌，安建峰，等.计算机组成与设计：硬件/软件接口（原书第5版）[**M**].机械工业出版社，2020年4月

# 教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）