**计算机组成原理课程设计**

|  |  |
| --- | --- |
| 题 目 | 五级流水线处理器设计 |

|  |  |  |
| --- | --- | --- |
| 学生姓名 | | 陈明富 |
| 学 | 号 | 161710228 |
| 学 | 院 | 计算机科学与技术学院 |
| 专 | 业 | 计算机科学与技术 |
| 班 | 级 | 1617102 |
| 指导教师 | | 施慧彬 |

二〇一九年七月

**南京航空航天大学**

**实验报告诚信承诺书**

本人郑重声明：所呈交的毕业设计（论文）（题目：带冒险流水线CP设

计 36条 ）是本人在导师的指导下独立进行研究所取得的成果。尽本人

所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名：陈明富 2019 年 7 月 7日

（学号）：161710228

流水线 MIPS CPU课程设计报告

摘 要

为了提升对单周期 MIPS CPU 的理解能力和动手能力，充分地理解和掌握计算机组成原理课程的相关知识。深入理解计算机内部指令的执行原理和实现方式，设计简单的单周期CPU，计算机指令执行过程中控制信号和数据通路部件的联系，基本部件如存储器、组合逻辑元件和控制器的实现和处理方式。通过处理数据冒险、控制冒险解决流水线的冲突问题，设计出符合硬件逻辑要求的仿真。

关键词**：**流水线，MIPS，数据冒险，控制

Pipeline MIPS CPU Course Design Report

Abstract

In order to improve the understanding and hands-on ability of the single-cycle MIPS CPU, fully understand and master the knowledge of the computer composition principle course. In-depth understanding of the execution principle and implementation of computer internal instructions, design of a simple single-cycle CPU, the connection of control signals and data path components in the execution of computer instructions, basic components such as memory, combinatorial logic components and controller implementation and processing. By dealing with data adventures, controlling risk-solving pipeline conflicts, and designing simulations that meet hardware logic requirements.

Key Words**：**pipeline ；MIP; data hazard; control

目 录

[1. 课程设计说明](#_Toc30724_WPSOffice_Level1) [7](#_Toc30724_WPSOffice_Level1)

[1.1 流水线设计说明](#_Toc23205_WPSOffice_Level2) [7](#_Toc23205_WPSOffice_Level2)

[2. 模块化和层次化设计说明](#_Toc23205_WPSOffice_Level1) [8](#_Toc23205_WPSOffice_Level1)

[2.1 设计图](#_Toc2618_WPSOffice_Level2) [8](#_Toc2618_WPSOffice_Level2)

[2.2 模块化设计图](#_Toc24882_WPSOffice_Level2) [8](#_Toc24882_WPSOffice_Level2)

[2.3 层次化设计](#_Toc6165_WPSOffice_Level2) [9](#_Toc6165_WPSOffice_Level2)

[3. 具体模块化定义](#_Toc2618_WPSOffice_Level1) [10](#_Toc2618_WPSOffice_Level1)

[3.1 PC模块](#_Toc22754_WPSOffice_Level2) [10](#_Toc22754_WPSOffice_Level2)

[3.2 NPC模块](#_Toc13149_WPSOffice_Level2) [10](#_Toc13149_WPSOffice_Level2)

[3.3 IM\_4k模块](#_Toc28516_WPSOffice_Level2) [10](#_Toc28516_WPSOffice_Level2)

[3.4 ALU模块](#_Toc12566_WPSOffice_Level2) [11](#_Toc12566_WPSOffice_Level2)

[3.5 MUX模块](#_Toc16924_WPSOffice_Level2) [11](#_Toc16924_WPSOffice_Level2)

[3.6 RegFile模块](#_Toc9834_WPSOffice_Level2) [12](#_Toc9834_WPSOffice_Level2)

[3.7 dm\_4k模块](#_Toc9808_WPSOffice_Level2) [12](#_Toc9808_WPSOffice_Level2)

[3.8 Ctrl模块](#_Toc1886_WPSOffice_Level2) [13](#_Toc1886_WPSOffice_Level2)

[3.9 Extend模块](#_Toc19557_WPSOffice_Level2) [14](#_Toc19557_WPSOffice_Level2)

[3.10 IFID寄存器](#_Toc25965_WPSOffice_Level2) [14](#_Toc25965_WPSOffice_Level2)

[3.11 IDEX寄存器](#_Toc16361_WPSOffice_Level2) [14](#_Toc16361_WPSOffice_Level2)

[3.12 EXMEM寄存器](#_Toc26089_WPSOffice_Level2) [16](#_Toc26089_WPSOffice_Level2)

[3.13 MEMWB寄存器](#_Toc8444_WPSOffice_Level2) [17](#_Toc8444_WPSOffice_Level2)

[3.14 MIPS模块](#_Toc16446_WPSOffice_Level2) [18](#_Toc16446_WPSOffice_Level2)

[3.15 Forward模块](#_Toc10747_WPSOffice_Level2) [18](#_Toc10747_WPSOffice_Level2)

[3.16 load\_use模块](#_Toc11113_WPSOffice_Level2) [18](#_Toc11113_WPSOffice_Level2)

[3.17 controlhazard模块](#_Toc27552_WPSOffice_Level2) [18](#_Toc27552_WPSOffice_Level2)

[4. 测试代码及结果](#_Toc24882_WPSOffice_Level1) [18](#_Toc24882_WPSOffice_Level1)

[4.1 具体Verilog代码实现](#_Toc7209_WPSOffice_Level2) [18](#_Toc7209_WPSOffice_Level2)

[4.2 ModelSim模拟及测试](#_Toc15085_WPSOffice_Level2) [38](#_Toc15085_WPSOffice_Level2)

[4.3 测试仿真结果](#_Toc435_WPSOffice_Level2) [39](#_Toc435_WPSOffice_Level2)

[5. 心得体会](#_Toc6165_WPSOffice_Level1) [42](#_Toc6165_WPSOffice_Level1)

[参 考 文 献](#_Toc14647_WPSOffice_Level1) [43](#_Toc14647_WPSOffice_Level1)

[附 录](#_Toc22754_WPSOffice_Level1) [43](#_Toc22754_WPSOffice_Level1)

# 课程设计说明

## 流水线设计说明

处理了一条指令的执行过程被分成五个阶段，每个阶段由相应的功能部件完成。如果将各阶段看成相应的流水段，则指令的执行过程就构成了一条指令流水线。

1. Ifetch 阶段:将PC的值作为地址到指令存储器IM中取指令，并计算PC+4，送到PC输入端。
2. Reg/Dec 阶段：根据指令中的Rs和Rt的值到寄存器堆中取出相应寄存器的值，同时对指令中的操作码op字段进行译码，生成相应的控制信号。寄存器堆可看成是寄存器读口和寄存器写口两个功能部件。
3. Exec 阶段：由具体指令确定，不同的指令经过ID段译码后得到不同的控制信号，用来控制执行部件进行不同的操作。
4. Mem 阶段：由具体指令确定，有Branch和MemWr两个控制信号。
5. Wr阶段: 由具体指令确定，将数据写到寄存器中。

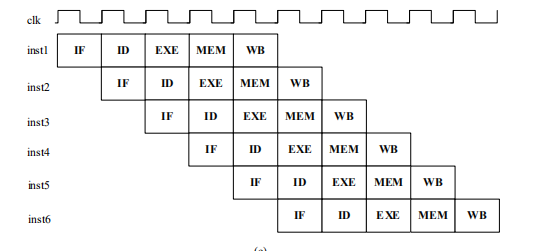


图1.1 5级流水线CPU时空图

# 模块化和层次化设计说明

## 2.1 设计图

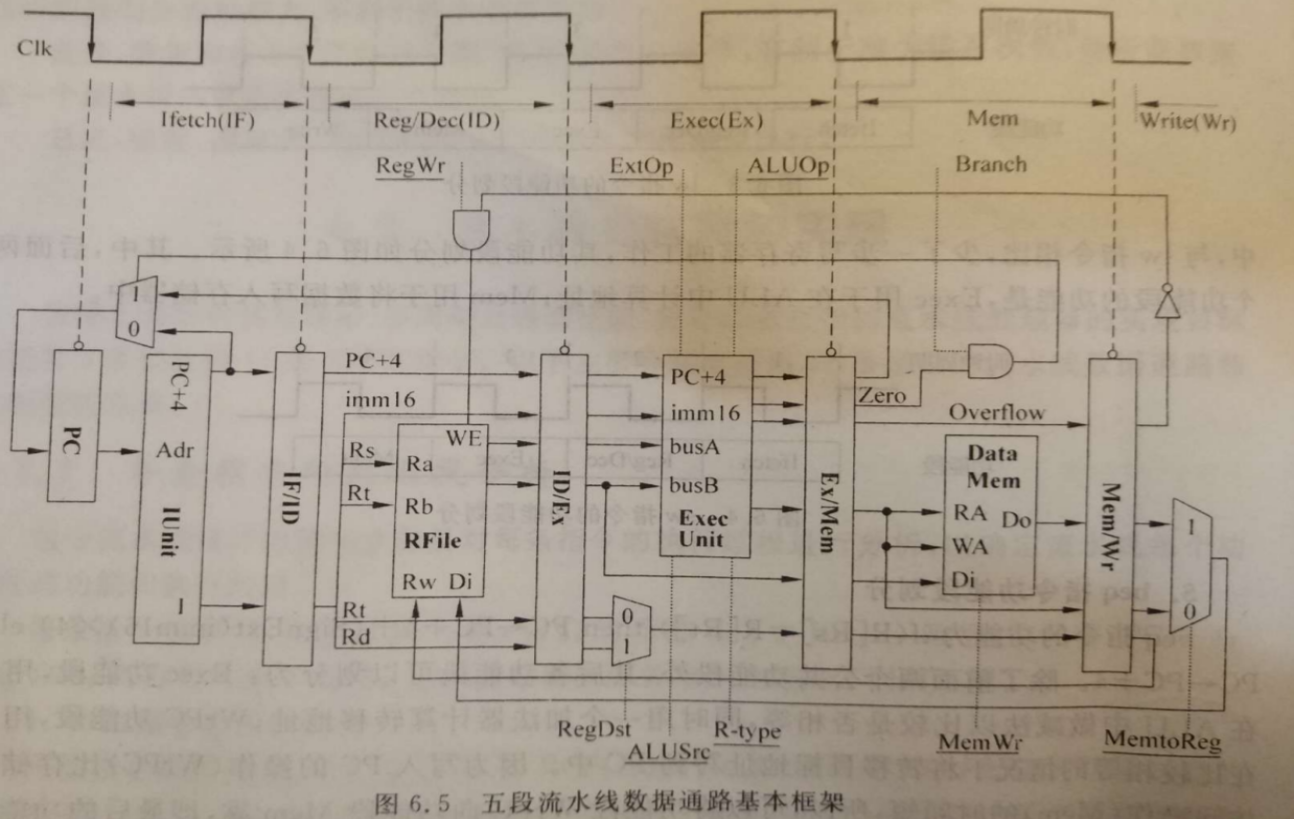


图1.1 5级流水线数据通路基本框架

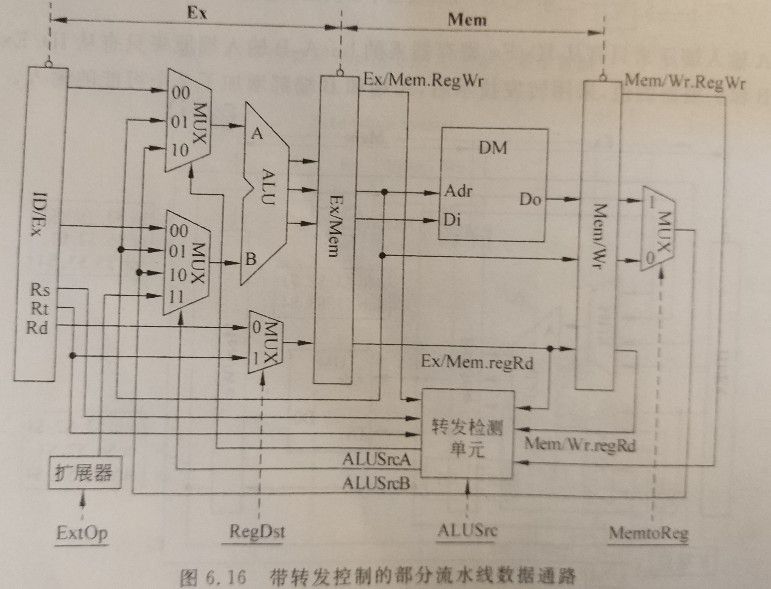


图1.2 带转发的流水线部分数据通路

2.2 模块化设计图

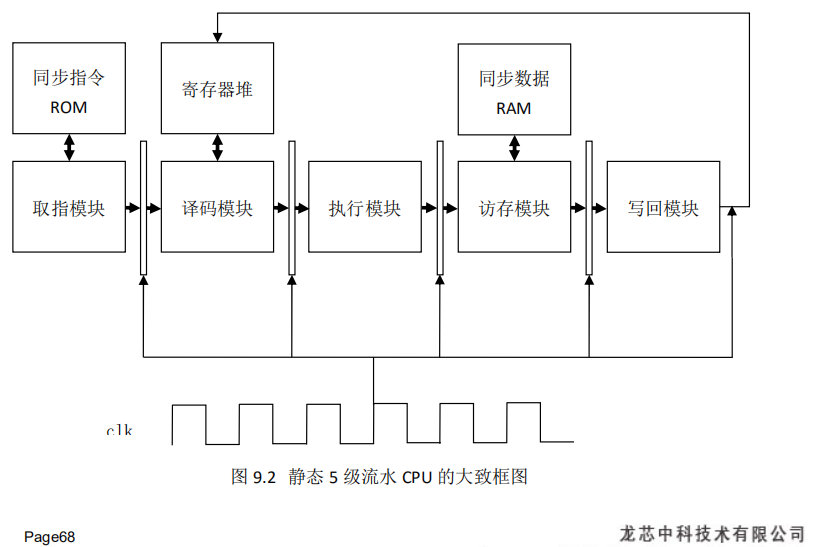


图1.3 5级流水线CPU框图

2.3 层次化设计

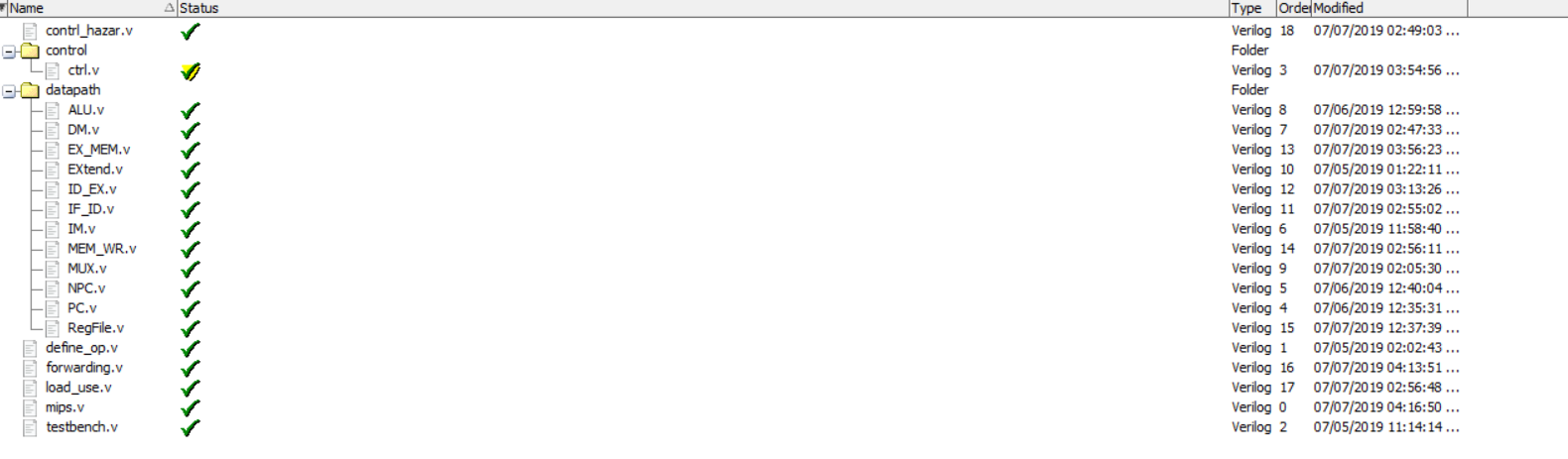


图1.4 流水线文件层次框图

# 具体模块化定义

3.1 PC模块

（1）基本描述描述：

用来输出当前指令的地址

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| reset | I | 复位信号 |
| bubble | I | 阻塞信号 |
| NPC [31:0] | I | 下一地址 |
| PC [31:0] | O | 输出地址 |

表1.1 PC模块定义

（3）功能定义：

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 复位 | rst=1 时，将 pc 置为 0X0000\_3000 |
| 2 | 输出地址 | 时钟信号到来时，将 NPC 赋给 PC |

表1.2 PC功能定义

3.2 NPC模块

（1）基本描述：

计算下一条指令并输出

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Zero | I | ALU 计算结果：1 表示当前两寄存器(rs,rt)值相等;0 表示不相等。 |
| Branch | I | 分支指令，1为是，0为否 |
| Jump | I | 是否为J指令，1为是，0为否 |
| Imm16[15:0] | I | 需要扩展的立即数 |
| Target[15:0 | I | J跳转的立即数 |
| PC [31:2] | I | 当前正常的指令地址 |
| NPC [31:2] | O | 输出地址 |

表1.3 NPC模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 输出地址 | 正常顺序的取指地址 |
| 2 | 输出地址 | 根据 zero 与 Branch 的值输出下一  条指令的地址 |
| 3 | 输出J跳转地址 | 根据 jump 和 target 的值输出下一  条指令的地址 |

表1.4 NPC功能定义

3.3 im\_4k模块

（1）基本描述：

取指令模块，指令内存大小为 4K，初始化从 data.txt 载入指令。根据输入的指令地址，输出当前位置存储的指令。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Addr[31:2] | I | 指令地址 |
| Dout[31:0] | O | 指令 |

表1.5 im\_4k模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 载入指令 | 初始化载入data.txt中的指令 |
| 2 | 输出指令 | 根据指令地址，输出指令 |

表1.6 im\_4k功能定义

3.4 ALU模块

（1）基本描述：

实现基本的R型指令以及其他需要进行ALU运算的指令，如 addu，subu，slt，and，nor， or，xor，sll，srl，sltu ，sllv，sra， srav，srlv，lui，slti，sltiu 等基本操作。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| A[31:0] | I | 输入数据端口 |
| B[31:0] | I | 输入数据端口 |
| Shf[4:0] | I | 移位操作 |
| ALUOp[4:0] | I | 选择ALU操作 |
| Result[31:0] | O | 计算结果 |
| Zero | O | 0标志 |

表1.7 ALU模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 输出计算结果 | 根据 alu 控制信号，输出A 与 B的计算结果 |
| 2 | 输出零标识位 | 根据计算结果判断是否为0,1表示为0；0表示不为0 |
| 3 | 基本运算 | 进行加法、减法、逻辑左右移、变量移、与或非、异或等操作 |

表1.8 ALU功能定义

3.5 MUX模块

（1）基本描述：

根据输入信号选择不同的输出结果，有不同位宽的选择器；有位宽为5的选择rd和rt，以及32位的选择器。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Muxop | I | 选择控制信号 |
| Cin\_a[4:0] | I | 待选择a |
| Cin\_b[4:0] | I | 待选择b |

表1.9 MUX模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 5位的选择器 | 选择输出对应信号的5位结果 |
| 2 | 32位的选择器 | 选择输出对应信号的32位结果 |

表1.10 MUX功能定义

3.6 RegFile模块

（1）基本描述：

在时钟信号控制下，根据输入的两个寄存器地址，输出相应寄存器的值，根据寄存器写信号和寄存器地址，将输入的数据选择写入寄存器。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| CLK | I | 时钟信号 |
| WEn | I | 写使能，在时钟上升沿到来时，为1则写数据 |
| RA[4:0] | I | Rs寄存器 |
| RB[4:0] | I | Rt寄存器 |
| RW[4:0] | I | 被写数据的寄存器Rd或者Rt |
| busW[31:0] | I | 待写的数据 |
| busA[31:0] | O | Rs寄存器存储的内容 |
| busB[31:0] | O | Rt寄存器存储的内容 |
| CLK | I | 时钟信号 |
| WEn | I | 写使能，在时钟上升沿到来时，为1则写数据 |
| RA[4:0] | I | Rs寄存器 |

表1.11 RegFile模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 写数据到对应寄存器中 | 根据写使能和时钟，写数据到制定的寄存器 |
| 2 | 读取对应的寄存器内的数据 | 在时钟控制下，读取对应寄存器中的内容数据 |

表1.12 RegFile功能定义

3.7 dm\_4k模块

（1）基本描述：

用于存储数据的的数据存储器，内存。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Address[11：0] | I | 地址线 |
| Bitop | I | bit操作 |
| Extop | I | Bit操作时的外扩展信号 |
| CLK | I | 时钟信号 |
| WE | I | 写使能 |
| din[31:0] | I | 待写数据 |
| dout[31:0] | O | 输出数据 |

表1.13 dm\_4k模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读数据内存 | 根据地址读出内存中的数据，word |
| 2 | 读数据内存 | 根据地址读出内存中的数据，bit字节 |

表1.14 dm\_4k功能定义

3.8 Ctrl模块

（1）基本描述：

根据指令中的op和func得到，R型指令的基本是一致的，特别地处理逻辑右移和逻辑左移以及算术右移时需要产生shfop标志；因为bgez和bltz的op域相同，无法产生可以区分的控制信号，所以需要再根据rt产生ALUop。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| op[5:0] | I | 指令高6位 |
| func[5:0] | I | 指令的func段 |
| rt[4:0] | I | rt寄存器 |
| zero | I | ALU中得到的0标志 |
| PCWre | O | PC修改标志 |
| ALUSrc | O | ALU端选择立即数或者busB |
| RegWr | O | 寄存器写使能信号 |
| RegDst | O | 选择rd或rt |
| MentoReg | O | 选择ALU输出端或者内存数据信号 |
| MenWr | O | DM写使能 |
| ExtOp | O | 符号扩展信号 |
| cin | O | 进位 |
| Branch | O | 分支指令信号 |
| Jump | O | j跳转指令信号 |
| Shfop | O | 逻辑左右移和算术右移信号 |
| Bitop | O | dm的bit操作信号 |
| ALUOp[4:0] | O | ALUop控制信号 |

表1.15 RegFile模块定义

（3）功能定义

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 输出控制信号 | 根据指令的op域和func段等，输出所有的控制信号 |
| 2 | 输出几个逻辑和算术位移信号 | Shfop用于几个逻辑位移 |
| 3 | 生成bgez和bltz的ALUop | 根据rt生成bgez和bltz |

表1.16 Ctrl功能定义

3.9 Extend模块

（1）基本描述：

将16位的立即数进行符号或者0扩展为32位，将8位的立即数符号或者0扩展为32位.

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Extop | I | 符号或0扩展控制信号 |
| Immediate | I | 立即数输入 |
| Extendimmdiate | O | 扩展输出 |

表1.17 Extend模块定义

（3）功能定义：

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 符号扩展为32位 | 根据控制信号进行扩展 |
| 2 | 0扩展为32位 | 根据控制信号进行扩展 |

表1.18 Extend功能定义

3.10 IFID寄存器

（1）基本描述：

IFID流水段寄存器用于实现取指与译码阶段，，时钟信号上升沿时把信号传给下一流水段或寄存器。

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| Clk | I | 时钟信号 |
| flush | I | 清空信号，置为0 |
| stall | I | 阻塞信号 |
| Pc\_in | I | 输入地址 |
| Inst\_in | I | 输入指令 |
| Pc\_out | O | 译码阶段的指令对应的地址 |
| Inst\_out | O | 译码阶段指令 |

表1.19 IFID模块定义

3.11 IDEX寄存器

（1）基本描述：

将16位的立即数进行符号或者0扩展为32位，将8位的立即数符号或者0扩展为32位.

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| clk | I | 时钟信号 |
| flush | I | 清空信号 |
| stall | I | 阻塞信号 |
| RegWr\_in | I | 寄存器写使能信号 |
| RegDst\_in | I | 选择写寄存器 |
| ALUsrc\_in | I | ALUB输入端选择信号 |
| MemWr\_in | I | Dm写使能信号 |
| MemRead\_in | I | DM读数据使能信号 |
| Extop | I | 符号扩展信号 |
| MemtoReg\_in | I | 选择写回寄存器堆的数据端口 |
| Bitop\_in | I | bit操作信号 |
| Branch\_in | I | 分支跳转指令信号 |
| Jump\_in | I | J跳转 |
| ALUop[4:0] | I | ALUop域 |
| Rs\_in[4:0] | I | Rs |
| Rt\_in[4:0] | I | Rt |
| Rd\_in[4:0] | I | Rd |
| Target\_in[25:0] | I | J跳转的立即数 |
| Imm16\_in[15:0] | I | Branch的立即数 |
| Inst\_in[31:0] | I | 指令 |
| busA\_in[31:0] | I | 寄存器堆中读取busA |
| busB\_in[31:0] | I | 寄存器堆中读取busA |
| RegWr\_out | O | 对应的输出信号，等待用于下一流水段的使用 |
| RegDst\_out | O |
| ALUsrc\_out | O |
| MemWr\_out | O |
| MemRead\_out | O |
| Extop\_out | O |
| MemtoReg\_out | O |
| Bitop\_out | O |
| Branch\_out | O |
| Jump\_out | O |
| ALUop\_out[4:0] | O |
| Rs\_out[4:0] | O |
| Rs\_out[4:0] | O |
| Rs\_iout4:0] | O |
| Target\_out[25:0] | O |
| Imm16\_out[15:0] | O |
| Inst\_out[31:0] | O |
| busA\_out[31:0] | O |
| busB\_out[31:0] | O |

表1.20 IDEX模块定义

3.12 EXMEM寄存器

（1）基本描述：

将16位的立即数进行符号或者0扩展为32位，将8位的立即数符号或者0扩展为32位.

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 信号名 |
| clk | I | 时钟信号 |
| flush | I | 清空信号 |
| stall | I | 阻塞信号 |
| RegWr\_in | I | 寄存器写使能信号 |
| RegDst\_in | I | 选择写寄存器 |
| ALUsrc\_in | I | ALUB输入端选择信号 |
| MemWr\_in | I | Dm写使能信号 |
| MemRead\_in | I | DM读数据使能信号 |
| Extop | I | 符号扩展信号 |
| MemtoReg\_in | I | 选择写回寄存器堆的数据端口 |
| Bitop\_in | I | bit操作信号 |
| Branch\_in | I | 分支跳转指令信号 |
| Jump\_in | I | J跳转 |
| Zero | I | 0标志 |
| Rs\_in[4:0] | I | Rs |
| Rt\_in[4:0] | I | Rt |
| Rd\_in[4:0] | I | Rd |
| Target\_in[25:0] | I | J跳转的立即数 |
| Imm16\_in[15:0] | I | Branch的立即数 |
| Inst\_in[31:0] | I | 指令 |
| Result\_in[31:0] | I | ALU运算结果 |
| busB\_in[31:0] | I | 寄存器堆中读取busA |
| RegWr\_out | O | 对应的输出信号，等待用于下一流水段的使用 |
| RegDst\_out | O |
| ALUsrc\_out | O |
| MemWr\_out | O |
| MemRead\_out | O |
| Extop\_out | O |
| MemtoReg\_out | O |
| Bitop\_out | O |
| Branch\_out | O |
| Jump\_out | O |
| Zero | O |
| Rs\_out[4:0] | O |
| Rs\_out[4:0] | O |
| Rs\_iout4:0] | O |
| Target\_out[25:0] | O |
| Imm16\_out[15:0] | O |
| Inst\_out[31:0] | O |
| Result\_out[31:0] | O |
| busB\_out[31:0] | O |

表1.21 EXMEM模块定义

3.13 MEMWB寄存器

（1）基本描述：

将16位的立即数进行符号或者0扩展为32位，将8位的立即数符号或者0扩展为32位.

（2）接口定义：

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 信号名 |
| clk | I | 时钟信号 |
| stall | I | 阻塞信号 |
| RegWr\_in | I | 寄存器写使能信号 |
| RegDst\_in | I | 选择写寄存器 |
| MemtoReg\_in | I | 选择写回寄存器堆的数据端口 |
| Rt\_in[4:0] | I | Rt |
| Rd\_in[4:0] | I | Rd |
| Result\_in[31:0] | I | ALU运算结果 |
| Dout\_in[31:0] | I | DM中取出的数据 |
| RegWr\_out | O | 对应的输出信号，等待用于下一流水段的使用 |
| RegDst\_out | O |
| MemtoReg\_out | O |
| Rt\_out[4:0] | O |
| Rd\_iout4:0] | O |
| Result\_out[31:0] | O |
| \_out[31:0] | O |

表1.22 MEMWB模块定义

3.14 MIPS模块

（1）基本描述：

顶层模块，用于连接各模块。

（2）接口定义：

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

表1.23 MIPS模块定义

3.15 Forward模块

（1）基本描述：

处理数据冒险，用于转发数据。

（2）接口定义：

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

表1.24 Forward模块定义

3.16 load\_use模块

（1）基本描述：

处理数据冒险，用于转发数据。

（2）接口定义：

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

表1.25 load\_use模块定义

3.17 controlhazard模块

（1）基本描述：

处理数据冒险，用于转发数据。

（2）接口定义：

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

表1.25 controlhazard模块定义

# 测试代码及结果

## 具体Verilog代码实现

## PC模块

module PC(CLK,reset,bubble,NPC,PC);

input CLK; // 时钟

input reset; // 重置信号

input bubble; // 阻塞信号

input [31:2] NPC; // PC指令地址

output reg[31:2] PC ;

initial begin

PC <= 32'h0000\_3000;

end

always@(posedge CLK)

begin

if (reset == 1) PC <= 32'h0000\_3000;

else

begin

if (bubble) PC <= PC; //阻塞PC地址保持不变

else PC <= NPC;

end

end

Endmodule

## NPC模块

module NPC(PC,Zero,Branch,Jump,target,imm16,Npc);

input Zero;

input Branch;

input Jump;

input [31:2] PC;

input [15:0] imm16;

input [25:0] target;

output[31:2] Npc;

wire [31:0] imm32;

wire [31:2] JNPC;

wire [31:2] BNPC;

wire [31:2] NNPC;

Extend16 extnpc(1,imm16,imm32);

assign BNPC = PC+1+imm32[29:0];

assign JNPC = {PC[31:28],target};

assign NNPC = PC + 1;

assign Npc = Jump?JNPC:((Branch&Zero)?BNPC:NNPC);

endmodule

## EXtend模块

//16位->32位

module Extend16(

input Extop,

input [15:0] immediate,

output [31:0] extendImmediate

);

assign extendImmediate[15:0] = immediate;

assign extendImmediate[31:16] = Extop ? (immediate[15] ? 16'hffff : 16'h0000) : 16'h0000;

endmodule

## MUX模块

//16位->32位

//选择写寄存器 rd or rt

// 5位2路

module Mux2\_5(

input MuxOp,

input [4:0] cin\_a,

input [4:0] cin\_b,

output [4:0] out

);

assign out = MuxOp ? cin\_b : cin\_a;

endmodule

//32位2路

module Mux2\_32(

input MuxOp,

input [31:0] cin\_a,

input [31:0] cin\_b,

output [31:0] out

);

assign out = MuxOp ? cin\_b : cin\_a;

endmodule

//32位3路

module Mux3\_32(

input [1:0] MuxOp,

input [31:0] cin\_a,

input [31:0] cin\_b,

input [31:0] cin\_c,

output[31:0] out

);

reg [31:0] reg\_out;

always @( \* ) begin

case (MuxOp)

2'b00: reg\_out = cin\_a;

2'b01: reg\_out = cin\_b;

2'b10: reg\_out = cin\_c;

default: ;

endcase

end

assign out = reg\_out;

endmodule

//32位4路

module Mux4\_32(

input [1:0] MuxOp,

input [31:0] cin\_a,

input [31:0] cin\_b,

input [31:0] cin\_c,

input [31:0] cin\_d,

output[31:0] out

);

reg [31:0] reg\_out;

always @( \* ) begin

case ( MuxOp )

2'b00: reg\_out = cin\_a;

2'b01: reg\_out = cin\_b;

2'b10: reg\_out = cin\_c;

2'b11: reg\_out = cin\_d;

default: ;

endcase

end

assign out = reg\_out;

endmodule

## Im\_4k模块

module im\_4k(

input [11:2] addr, // 读取指令存储地址

output [31:0] dout ); // 指令内容

reg[31:0] im[0:1023]; // 指令存储器

initial

begin

$readmemh("data.txt", im); //读取文件中的指令 b-2进制 h-16进制

end

assign dout = im[addr];

endmodule

## dm\_4k模块

module dm\_4k( addr,Bitop,Extop,din, we,MemRead, clk, dout ) ;

input [11:0] addr ;

input Bitop; // 判断是否进行字节的读写

input Extop; // bit读时扩展

input MemRead; // 读使能

input [31:0] din ; // 待写数据

input we; //写使能

input clk ;

output reg [31:0] dout ; //输出数据 bit时候需要进行位扩展

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

reg [1:0] pos; //地址的后两位 据此选择bit

reg [7:0] bit;

reg [31:0] Ext\_bit;

always@(clk)

begin

pos <= addr[1:0];

if(we)

begin

if(!Bitop)

begin

dm[addr] <= din;

end

else

case(pos)

2'b00: dm[addr[11:2]] = {dm[addr[11:2]][31:8],din[7:0] };

2'b01: dm[addr[11:2]] = {dm[addr[11:2]][31:16],din[7:0],dm[addr[11:2]][7:0] };

2'b10: dm[addr[11:2]] = {dm[addr[11:2]][31:25],din[7:0],dm[addr[11:2]][15:0] };

2'b11: dm[addr[11:2]] = {din[7:0],dm[addr[11:2]][23:0] };

endcase

end

// 读内存

else

begin

if(MemRead)

begin

if(!Bitop) dout <= dm[addr[11:2]];

else

begin

case(pos)

2'b00: bit <= dm[addr[11:2]][7:0];

2'b01: bit <= dm[addr[11:2]][15:8];

2'b10: bit <= dm[addr[11:2]][23:16];

2'b11: bit <= dm[addr[11:2]][31:24];

endcase

if(Extop)

begin

if(bit[7]) Ext\_bit <= {24'hffffff,bit};

else

Ext\_bit <= {24'h000000,bit};

end

else Ext\_bit = {24'b0,bit};

dout <= Ext\_bit;

end

end

end

end

endmodule

## ALU模块

module Alu(A,B,ALUop,shf,Result,Zero);

input[31:0] A,B;

input[4:0] ALUop;

input [4:0] shf; //移位

output[31:0] Result;

reg[31:0] Result;

output reg Zero; //0标识

initial

begin

Zero = 0;

end

always@(ALUop or A or B)

begin

case (ALUop)

5'b00000: Result = A+B; // addu

5'b00001: Result = $signed(A) < 32'b0 ? 1:0; // bgez 大于或等于0跳转(小于，zero标志为1)

5'b00010: Result= A-B; // sub,subu,beq

5'b00011: Result = $signed(A)>0 ? 0:1; // bgtz 大于0跳转

5'b00100: Result = ($unsigned(A)<$unsigned(B))?1:0; // sltu

5'b00101: Result = ($signed(A)<$signed(B))?1:0; // slt

5'b00110: Result = A&B; // and

5'b00111: Result = ~(A|B); // nor

5'b01000: Result = A|B; // or

5'b01001: Result = A^B; // xor

5'b01010: Result = B<<shf; // sll

5'b01011: Result = B<<A; // sllv

5'b01100: Result = $signed(B)>>shf; // sra,srl

5'b01101: Result = $signed(B)>>A; // srav

5'b01110: Result = $unsigned(B)>>A; // srlv

5'b01111: Result = $signed(A) > 0 ? 1:0; // blez 小于或等于0跳转

5'b10000: Result = $signed(A)<0 ? 0:1; // bltz 小于0跳转

5'b10001: Result = B<<5'b10000; // lui 左移16位

5'b10010: Result = A!= B ? 0:1; // bne

5'b10010: Result = A; // jalr,jr

endcase

Zero =(Result)?0:1 ;

end

endmodule

## RegFile模块

module RF(clk,WEn,RW,RA,RB,busW,busA,busB);

input clk,WEn;

input[4:0] RW,RA,RB;

input[31:0] busW;

output [31:0] busA,busB;

reg[31:0] regfile[0:31];

integer i;

initial

begin

for(i = 0; i < 31; i = i + 1) regfile[i] <= 0;

end

///时钟下降沿到来写

////先写后读

always@(clk )

begin

if(WEn)

regfile[RW] <= busW;

end

assign busA = regfile[RA];

assign busB = regfile[RB] ;

Endmodule

## IFID\_REG模块

module IFID (clk,flush,stall,pc\_in ,inst\_in, pc\_out,inst\_out);

input clk;

input flush;//清空

input stall;

input [31:0] pc\_in; // 当前取指令地址

input [31:0] inst\_in;//

output reg [31:0] pc\_out; //下一条指令地址

output reg[31:0] inst\_out;//

reg [31:0]pc;

reg [31:0] inst;

always @(posedge clk )

begin

if(flush==1)

begin

pc <=0;

inst <= 0;

end

else if(stall==1)

begin

inst\_out <= inst;

pc\_out<= pc;

end

else

begin

pc = pc\_in;

inst = inst\_in;

inst\_out <= inst\_in;

pc\_out <= pc\_in;

end

end

endmodule

## IDEX\_REG模块

module IDEX (clk,flush,stall,RegWr\_in,RegDst\_in,ALUsrc\_in,MemWr\_in,MemRead\_in,Extop\_in,MemToReg\_in,Bitop\_in,Branch\_in,Jump\_in,

ALUop\_in,rs\_in,rt\_in,rd\_in,target\_in,imm16\_in,inst\_in,busA\_in,busB\_in,

RegWr\_out,RegDst\_out,ALUsrc\_out,MemWr\_out,MemRead\_out,Extop\_out,MemToReg\_out,Bitop\_out,Branch\_out,Jump\_out,

ALUop\_out,rs\_out,rt\_out,rd\_out,target\_out,imm16\_out,inst\_out,busA\_out,busB\_out

);

input clk;//时钟信号

input flush;//清空

input stall; //阻塞

////Input：

input RegWr\_in;

input RegDst\_in;

input ALUsrc\_in;

input MemWr\_in;

input MemRead\_in;

input Extop\_in;

input MemToReg\_in;

input Bitop\_in;

input Branch\_in;

input Jump\_in;

input [4:0] ALUop\_in;

input [4:0] rs\_in ;

input [4:0] rt\_in;

input [4:0] rd\_in;

input [25:0] target\_in;

input [15:0] imm16\_in;

input [31:0] inst\_in;

input [31:0] busA\_in;

input [31:0] busB\_in;

//// Output:

output reg RegWr\_out;

output reg RegDst\_out;

output reg ALUsrc\_out;

output reg MemWr\_out;

output reg MemRead\_out;

output reg Extop\_out;

output reg MemToReg\_out;

output reg Bitop\_out;

output reg Branch\_out;

output reg Jump\_out;

output reg [4:0] ALUop\_out;

output reg [4:0] rs\_out ;

output reg [4:0] rt\_out;

output reg [4:0] rd\_out;

output reg [25:0] target\_out;

output reg [15:0] imm16\_out;

output reg [31:0] inst\_out;

output reg [31:0] busA\_out;

output reg [31:0] busB\_out;

always@(posedge clk)

begin

if ( flush ) //清空信号

begin

RegWr\_out <= 0;

RegDst\_out <= 0;

ALUsrc\_out <= 0;

MemWr\_out <= 0;

MemRead\_out <= 0 ;

Extop\_out <= 0;

MemToReg\_out <= 0;

Bitop\_out <= 0;

Branch\_out <= 0;

Jump\_out <= 0;

ALUop\_out <= 0;

rs\_out <= 0;

rd\_out <= 0;

rt\_out <= 0;

target\_out <= 0;

imm16\_out <= 0;

inst\_out <= 0;

busA\_out <= 0;

busB\_out <= 0;

end

else if(!stall) //阻塞信号

begin

RegWr\_out <= RegWr\_in;

RegDst\_out <= RegDst\_in;

ALUsrc\_out <= ALUsrc\_in;

MemWr\_out <= MemWr\_in;

MemRead\_out <= MemRead\_in;

Extop\_out <= Extop\_in;

MemToReg\_out <= MemToReg\_in;

Bitop\_out <= Bitop\_in;

Branch\_out <= Branch\_in;

Jump\_out <= Jump\_in;

ALUop\_out <= ALUop\_in;

rd\_out <= rd\_in;

rs\_out <= rs\_in;

rt\_out <= rt\_in;

target\_out <= target\_in;

imm16\_out <= imm16\_in;

inst\_out <= inst\_in;

busA\_out <= busA\_in;

busB\_out <= busB\_in;

end

end

endmodule

## EXMEM\_REG模块

module EXMEM (clk,flush,stall,RegWr\_in,RegDst\_in,MemWr\_in,MemRead\_in,MemtoReg\_in,Branch\_in,Jump\_in,Zero\_in,Extop\_in,Bitop\_in,rs\_in,rt\_in,rd\_in,busB\_in,Result\_in,

RegWr\_out,RegDst\_out,MemWr\_out,MemRead\_out,MemtoReg\_out,Branch\_out,Jump\_out,Zero\_out,Extop\_out,Bitop\_out,rs\_out,rt\_out,rd\_out,busB\_out,Result\_out

);

input clk;

input flush;

input stall;

/////INput

input RegWr\_in;

input RegDst\_in;

input MemWr\_in;

input MemRead\_in;

input MemtoReg\_in;

input Branch\_in;

input Jump\_in;

input Zero\_in;

input Extop\_in;

input Bitop\_in;

input [4:0] rs\_in;

input [4:0] rt\_in;

input [4:0] rd\_in;

input [31:0] busB\_in;

input [31:0] Result\_in;

/////Output

output reg RegWr\_out;

output reg RegDst\_out;

output reg MemWr\_out;

output reg MemRead\_out;

output reg MemtoReg\_out;

output reg Branch\_out;

output reg Jump\_out;

output reg Zero\_out;

output reg Extop\_out;

output reg Bitop\_out;

output reg [4:0]rs\_out;

output reg [4:0]rt\_out;

output reg [4:0]rd\_out;

output reg [31:0] busB\_out;

output reg [31:0] Result\_out;

always @(posedge clk )

begin

if (flush) //清空

begin

RegWr\_out <= 0;

RegDst\_out <= 0;

MemWr\_out <= 0;

MemRead\_out <= 0 ;

MemtoReg\_out <= 0;

Branch\_out <= 0;

Jump\_out <= 0;

Zero\_out <= 0;

Extop\_out <= 0;

Bitop\_out <= 0;

rs\_out <= 0;

rt\_out <= 0;

rd\_out <= 0;

busB\_out <= 0;

Result\_out <= 0;

end

if(!stall) //非阻塞

begin

RegWr\_out <= RegWr\_in;

RegDst\_out <= RegDst\_in;

MemWr\_out <= MemWr\_in;

MemRead\_out <= MemRead\_in;

MemtoReg\_out <= MemtoReg\_in;

Branch\_out <= Branch\_in;

Jump\_out <= Jump\_in;

Zero\_out <= Zero\_in;

Extop\_out <= Extop\_in;

Bitop\_out <= Bitop\_in;

rs\_out <= rs\_in;

rt\_out <= rt\_in;

rd\_out <= rd\_in;

busB\_out <= busB\_in;

Result\_out <= Result\_in;

end

end

endmodule

## MEMWB\_REG模块

module MEMWB( clk,stall,

RegWr\_in,RegDst\_in,MemToReg\_in,rt\_in,rd\_in,Dout\_in,Result\_in,

RegWr\_out,RegDst\_out,MemToReg\_out,rt\_out,rd\_out,Dout\_out,Result\_out

);

input clk;

input stall;

////Input

input RegWr\_in; //Regfile写

input RegDst\_in;//选择rd或者rt

input MemToReg\_in;//选择写Reg的内容

input [4:0] rt\_in;

input [4:0] rd\_in;

input [31:0] Dout\_in; //DM读出的内容

input [31:0] Result\_in; //ALU计算结果，DM地址或者Reg待写内容

////Output

output reg RegWr\_out;

output reg RegDst\_out;

output reg MemToReg\_out;

output reg [4:0] rt\_out;

output reg [4:0] rd\_out;

output reg [31:0] Dout\_out;

output reg [31:0] Result\_out;

always @(posedge clk )

begin

if(stall)

begin

RegWr\_out <= 0;

RegDst\_out <= 0;

MemToReg\_out <= 0;

rt\_out <= 0;

rd\_out <= 0;

Dout\_out <= 0;

Result\_out <= 0;

end

else

begin

RegWr\_out <= RegWr\_in;

RegDst\_out <= RegDst\_in;

MemToReg\_out <= MemToReg\_in;

rt\_out <= rt\_in;

rd\_out <= rd\_in;

Dout\_out <= Dout\_in;

Result\_out <=Result\_in;

end

end

Endmodule

## Forward模块

//转发主要解决

//采用助教文档中的案例

//从Reg寄存器中读取数据，数据还未被写入，即读取的是旧值

module DataHazard(Mem\_RegWr, Wr\_RegWr,MemtoReg,Mem\_Rw, Ex\_Rs, Ex\_Rt, Wr\_Rw,

Ex\_ALUSrc, Ex\_imm32, Ex\_busA, Ex\_busB, Mem\_Result,Wr\_Result, Wr\_din,

A, B);

input Mem\_RegWr, Wr\_RegWr,MemtoReg, Ex\_ALUSrc;

input [4:0] Mem\_Rw, Ex\_Rs, Ex\_Rt, Wr\_Rw;

input [31:0] Ex\_imm32; // 立即数

input [31:0] Ex\_busA;

input [31:0] Ex\_busB;

input [31:0] Mem\_Result,Wr\_Result;

input [31:0] Wr\_din;

output [31:0] A, B; //ALU的A和B输入端口

wire C1A, C1B, C2A, C2B,B2;

wire [31:0] busB;

reg [1:0] ALUSrcA, ALUSrcB;

initial

begin

ALUSrcA = 0;

ALUSrcB = 0;

end

assign C1A = Mem\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rs)||Wr\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rs); //选择Result

assign C1B = Wr\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rt)|| Mem\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rt); // 选择Result

/\*注意到此处助教给出代码的不足

C1B判定条件

原： Wr\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rt)

修改后：Wr\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rt)|| Mem\_RegWr&&(Mem\_Rw != 0)&&(Mem\_Rw == Ex\_Rt)

\*/

assign C2A = MemtoReg&&(Wr\_Rw != 0)&&(Mem\_Rw != Ex\_Rs)&&(Wr\_Rw == Ex\_Rs); //选择DM\_out

assign C2B = MemtoReg&&(Wr\_Rw != 0)&&(Mem\_Rw != Ex\_Rt)&&(Wr\_Rw == Ex\_Rt); //选择DM\_out

assign B2 = Wr\_RegWr&&(Wr\_Rw==Ex\_Rt);

always@(C1A or C2A)

begin

if (C1A !=1 && C2A != 1) ALUSrcA <= 2'b00;

if (C1A == 1) ALUSrcA <= 2'b01;

if (C2A == 1) ALUSrcA <= 2'b10;

end

always@(C1B or C2B)

begin

if (C1B != 1 && C2B != 1) ALUSrcB <= 2'b00;

if (C1B == 1) ALUSrcB <= 2'b01;

if (C2B == 1) ALUSrcB <= 2'b10;

end

assign A = (ALUSrcA == 2'b00) ? Ex\_busA :(ALUSrcA == 2'b01) ? Mem\_Result :(ALUSrcA == 2'b10) ? Wr\_din : 0 ;

/\*ALU输入A端：

00 正常情况，选择Ex\_busA，不发生数据冒险

01 选择 Mem\_Result

10 选择Wr\_din 即从Dm中读取出的数据，这时候应该 MemtoReg = 1

\*/

assign busB = B2 ?Wr\_Result: Ex\_busB;

assign B = Ex\_ALUSrc ? Ex\_imm32 :((ALUSrcB == 2'b00) ? busB:((ALUSrcB == 2'b01) ? Mem\_Result :((ALUSrcB == 2'b10) ? Wr\_din : 0)));

endmodule

## Loaduse模块

/\* 课本P211

需要实现的三个操作：

1.将ID/EX寄存器的控制信号清零

2.保持IF/ID寄存器的的值不变，在下一个时钟到来重新进行译码

3.保持PC的值不变，使得Load后面的第二条指令在下一时钟周期重新取指令

\*/

//冒险产生的结果

//保持PC不变 pc\_stall = 1

//清空ID/EX IDEX\_flush = 1

//保持IFID不变 IFID\_stall = 1

module Load\_use(IDEX\_MemRead,IFID\_rt,IFID\_rs,IDEX\_rt,pc\_stall,IFID\_stall,IDEX\_flush);

input IDEX\_MemRead;

input [4:0]IFID\_rt,IFID\_rs,IDEX\_rt;

output reg pc\_stall;

output reg IFID\_stall;

output reg IDEX\_flush;

always @(1)

begin

if((IDEX\_MemRead==1&&(IDEX\_rt==IFID\_rs||IDEX\_rt==IFID\_rt)))

begin

pc\_stall= 1;

IFID\_stall = 1;

IDEX\_flush = 1;

end

else

begin

pc\_stall= 0;

IFID\_stall = 0;

IDEX\_flush = 0;

end

end

endmodule

## Ctrlhazard模块

//处理分支跳转指令

/\*Branch & Jump

bne beq

bgez bgtz blez bltz

J

\*/

module Ctrl\_Hazar(clk,Sign,Jump,pc\_stall,IFID\_flush);

input clk;

input Sign; ///Branch 地址跳转条件

input Jump;

output reg pc\_stall;//清空信号

output reg IFID\_flush;

initial begin

pc\_stall = 0;

IFID\_flush = 0;

end

always @(1)

begin

//jump flush

if(Jump==1||Sign==1) //或者Branch满足跳转

begin

pc\_stall = 1;

IFID\_flush = 1;/// IFID——flush

end

else

begin

pc\_stall = 0;

IFID\_flush = 0;

end

end

Endmodule

## define模块

// define OP

`define RTYPE 6'b000000

`define LB 6'b100000

`define LBU 6'b100100

`define LW 6'b100011

`define SB 6'b101000

`define SW 6'b101011

`define ADDI 6'b001000

`define ADDIU 6'b001001

`define ANDI 6'b001100

`define ORI 6'b001101

`define XORI 6'b001110

`define LUI 6'b001111

`define SLTI 6'b001010

`define SLTIU 6'b001011

`define BEQ 6'b000100

`define BNE 6'b000101

`define BGEZ 6'b000001

`define BGTZ 6'b000111

`define BLEZ 6'b000110

`define BLTZ 6'b000001

`define J 6'b000010

//`define JAL 6'b000011

// func域

`define ADD\_func 6'b100000

`define ADDU\_func 6'b100001

`define SUB\_func 6'b100010

`define SUBU\_func 6'b100011

`define AND\_func 6'b100100

`define NOR\_func 6'b100111

`define OR\_func 6'b100101

`define XOR\_func 6'b100110

`define SLT\_func 6'b101010

`define SLTU\_func 6'b101011

`define SLL\_func 6'b000000

`define SRL\_func 6'b000010

`define SRA\_func 6'b000011

`define SLLV\_func 6'b000100

`define SRLV\_func 6'b000110

`define SRAV\_func 6'b000111

`define JR\_func 6'b001000

`define JALR\_func 6'b001001

`define BGEZ\_Rt 5'b00001

`define BLTZ\_Rt 5'b00000

## Crtl模块

`include "define\_op.v"

module

Ctrl(op,func,rt,ALUSrc,RegWr,RegDst,MemtoReg,MemWr,MemRead,Extop,Branch,Jump,Bitop,ALUop);

input [5:0] op;

input [5:0] func;

input [4:0] rt; //用于区别 bltz与bgez

output reg ALUSrc; //ALU选择

output reg RegWr; //Regfile写使能

output reg RegDst; //Regfile写地址选择，rd or rt

output reg MemtoReg; //写回结果内容选择

output reg MemWr; //存储器写使能

output reg MemRead;

output reg Extop; //符号扩展

output reg Branch;

output reg Jump;

output reg Bitop; //DM的字节

output reg [4:0] ALUop; //ALU操作

initial

begin Branch = 0;

Jump = 0 ;

Bitop =0;

end

always@(op or func )

begin

Branch = 0;

Jump = 0 ;

Bitop =0;

MemRead = 0;

case(op)

`RTYPE: //R型指令 000000

begin

ALUSrc = 0;

RegDst = 1 ;

MemWr = 0;

MemtoReg = 0;

Branch = 0;

Jump = 0;

Bitop = 0;

case(func)

`ADD\_func: //6'b100000

begin

RegWr = 1;

ALUop = 5'b00000;

end

`ADDU\_func : //6'b100001

begin

RegWr = 1;

ALUop = 5'b00000;

end

`SUB\_func : //6'b100010

begin

RegWr = 1;

ALUop = 5'b00010;

end

`SUBU\_func : //6'b100011

begin

RegWr = 1;

ALUop = 5'b00010;

end

`AND\_func : //6'b100100

begin

RegWr = 1;

ALUop = 5'b00110;

end

`NOR\_func : //6'b100111

begin

RegWr = 1;

ALUop = 5'b00111;

end

`OR\_func : //6'b100101

begin

RegWr = 1;

ALUop = 5'b01000;

end

`XOR\_func : //6'b100110

begin

RegWr = 1;

ALUop = 5'b01001;

end

`SLT\_func : //6'b101010

begin

RegWr = 1;

ALUop = 5'b00101;

end

`SLTU\_func : //6'b101011

begin

RegWr = 1;

ALUop = 5'b00100;

end

`SLL\_func : //6'b000000

begin

RegWr = 1;

ALUop = 5'b01010;

end

`SRL\_func : //6'b000010

begin

RegWr = 1;

ALUop = 5'b01100;

end

`SRA\_func : //6'b000011

begin

RegWr = 1;

ALUop = 5'b01100;

end

`SLLV\_func : //6'b000100

begin

RegWr = 1;

ALUop = 5'b01011;

end

`SRLV\_func : //6'b000110

begin

RegWr = 1;

ALUop = 5'b01110;

end

`SRAV\_func : //6'b000111

begin

RegWr = 1;

ALUop = 5'b01101;

end

// `JR\_func : //6'b001000

// `JALR\_func : //6'b001001

default:;

endcase

end

`ADDI ://addi 6'b001000

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

end

`ADDIU://addiu 6'b001001

begin

RegWr = 1;

RegDst = 0; // rt

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

end

`ANDI://andi 6'b001100

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b00110;//and

end

`ORI://ori 6'b001101

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b01000;//or

end

`XORI://xori 6'b001110

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b01001;//xor

end

`LW ://lw 6'b100011

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemRead = 1;

MemtoReg = 1;

Extop = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

Bitop = 0;

end

`SW://sw 6'b101011

begin

RegWr = 0;

ALUSrc = 1;

MemWr = 1;

MemtoReg = 0;

Extop = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

Bitop = 1;

end

`LUI://lui 6'b001111

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 0;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b10001;

end

`LBU://lbu 6'b100100

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 1;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

Bitop = 1;

end

`LB://lb 6'b100000

begin

RegWr = 1;

RegDst = 0;

ALUSrc = 1;

MemWr = 0;

MemtoReg = 1;

Extop = 0;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

Bitop = 1;

end

`SB://sb 6'b101000

begin

RegWr = 0;

ALUSrc = 1;

MemWr = 1;

Extop = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00000;

Bitop = 1;

end

`SLTI://slti 6'b001010

begin

RegWr = 1;

RegDst = 0;

Extop = 1;

MemWr = 0;

MemtoReg = 0;

ALUSrc = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00101;

end

`SLTIU://sltiu 6'b001011

begin

RegWr = 1;

RegDst = 0;

Extop = 1;

MemWr = 0;

MemtoReg = 0;

ALUSrc = 1;

Jump = 0;

Branch = 0;

ALUop = 5'b00100;

end

6'b000001://bgez区别

if(rt==`BGEZ\_Rt)

begin

RegWr = 0;

ALUSrc = 0;

MemWr = 0;

ALUop = 5'b00001;

Branch = 1;

Jump = 0;

end

else //bltz区别

begin

RegWr = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 1;

ALUop = 5'b10000;

Jump = 0;

end

`BGTZ://bgtz 6'b000111

begin

RegWr = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 1;

ALUop = 5'b00011;

Jump = 0;

end

`BLEZ://blez 6'b000110

begin

RegWr = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 1;

ALUop = 5'b01111;

Jump = 0;

end

`BEQ://beq 6'b000100

begin

RegWr = 0;

RegDst = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 1;

Jump = 0;

ALUop = 5'b00010;

end

`BNE: // bne 6'b000101

begin

RegWr = 0;

RegDst = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 1;

Jump = 0;

ALUop = 5'b10010;

end

`J://j 6'b000010

begin

RegWr = 0;

ALUSrc = 0;

MemWr = 0;

Branch = 0;

Jump = 1;

end

default:;

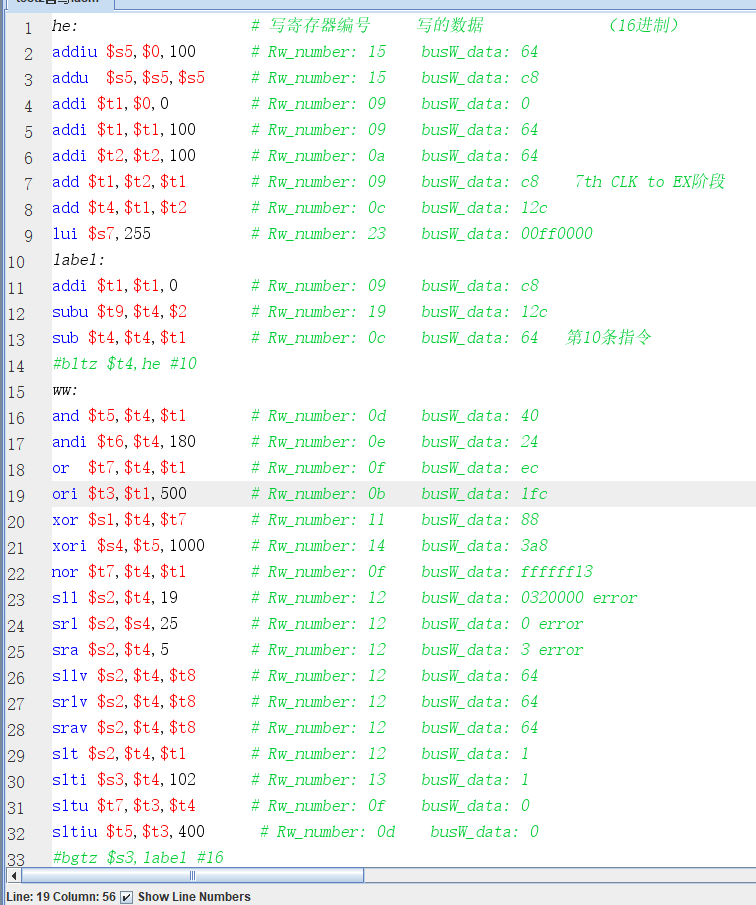
endcase

end

endmodule

## ModelSim模拟及测试

1. 使用自己写的测试代码，如下所示



具体测试代码：test.asm

he: # 写寄存器编号 写的数据 （16进制）

addiu $s5,$0,100 # Rw\_number: 15 busW\_data: 64

addu $s5,$s5,$s5 # Rw\_number: 15 busW\_data: c8

addi $t1,$0,0 # Rw\_number: 09 busW\_data: 0

addi $t1,$t1,100 # Rw\_number: 09 busW\_data: 64

addi $t2,$t2,100 # Rw\_number: 0a busW\_data: 64

add $t1,$t2,$t1 # Rw\_number: 09 busW\_data: c8 7th CLK to EX阶段

add $t4,$t1,$t2 # Rw\_number: 0c busW\_data: 12c

lui $s7,255 # Rw\_number: 23 busW\_data: 00ff0000

label:

addi $t1,$t1,0 # Rw\_number: 09 busW\_data: c8

subu $t9,$t4,$2 # Rw\_number: 19 busW\_data: 12c

sub $t4,$t4,$t1 # Rw\_number: 0c busW\_data: 64 第10条指令

#bltz $t4,he #10

ww:

and $t5,$t4,$t1 # Rw\_number: 0d busW\_data: 40

andi $t6,$t4,180 # Rw\_number: 0e busW\_data: 24

or $t7,$t4,$t1 # Rw\_number: 0f busW\_data: ec

ori $t3,$t1,500 # Rw\_number: 0b busW\_data: 1fc

xor $s1,$t4,$t7 # Rw\_number: 11 busW\_data: 88

xori $s4,$t5,1000 # Rw\_number: 14 busW\_data: 3a8

nor $t7,$t4,$t1 # Rw\_number: 0f busW\_data: ffffff13

sll $s2,$t4,19 # Rw\_number: 12 busW\_data: 0320000 error

srl $s2,$s4,25 # Rw\_number: 12 busW\_data: 0 error

sra $s2,$t4,5 # Rw\_number: 12 busW\_data: 3 error

sllv $s2,$t4,$t8 # Rw\_number: 12 busW\_data: 64

srlv $s2,$t4,$t8 # Rw\_number: 12 busW\_data: 64

srav $s2,$t4,$t8 # Rw\_number: 12 busW\_data: 64

slt $s2,$t4,$t1 # Rw\_number: 12 busW\_data: 1

slti $s3,$t4,102 # Rw\_number: 13 busW\_data: 1

sltu $t7,$t3,$t4 # Rw\_number: 0f busW\_data: 0

sltiu $t5,$t3,400 # Rw\_number: 0d busW\_data: 0

#bgtz $s3,label #16

sw $t1,100($t2) # Datamem: c8 data: c8

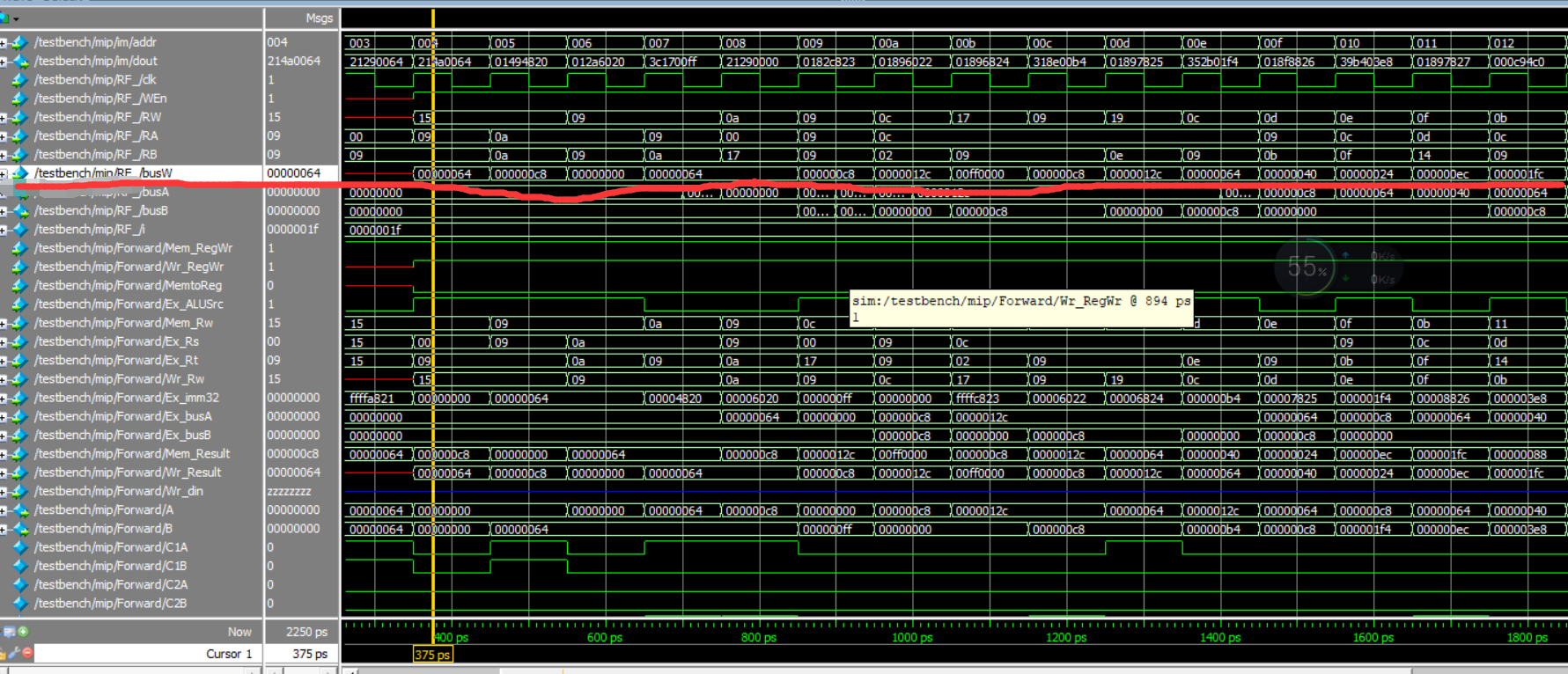
lw $s3,100($t2) # Rw\_number: 13 busW\_data: c8

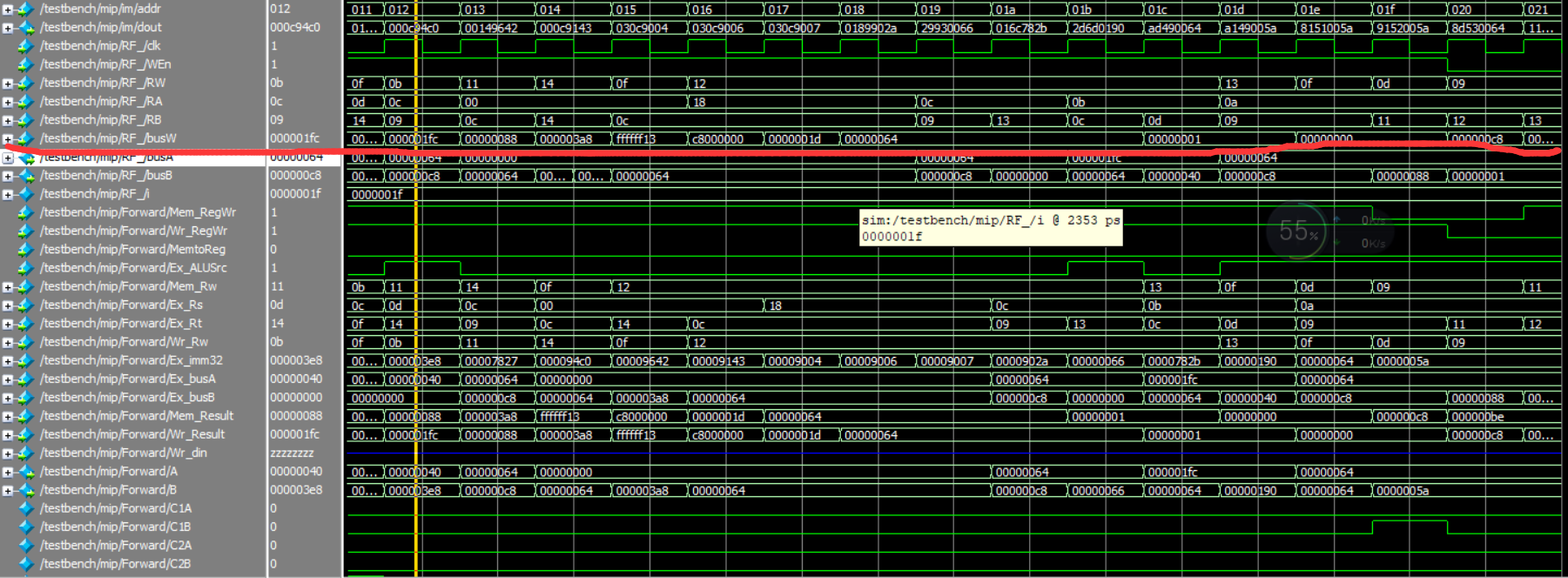
beq $t4,$t2,label #第一次会跳转至label 第20条指令

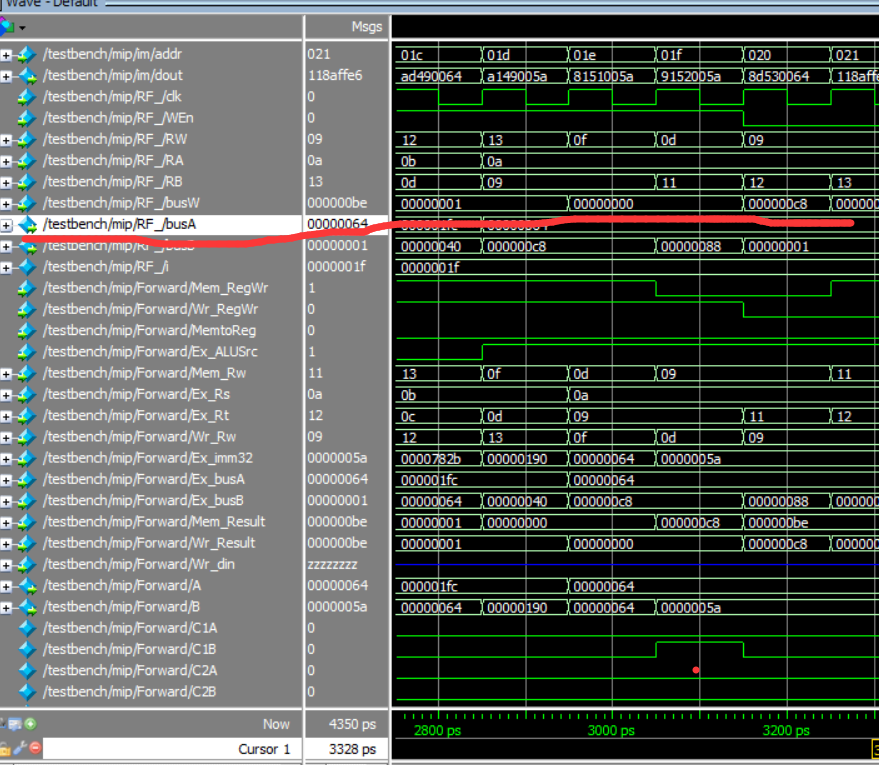
j he #地址转为0

## 测试仿真结果

像我写的汇编代码里面的注释那样的，我们只需要通过观测寄存器内数据的变化或者从Regfile的BusW是否与我们预期的一样，则可以说明代码的正确性。







通过观测busW的值与mars里面得到正确的值进行比较，可以得出以下：

指令add,addu,addi,addiu,subu,sub,lui 7

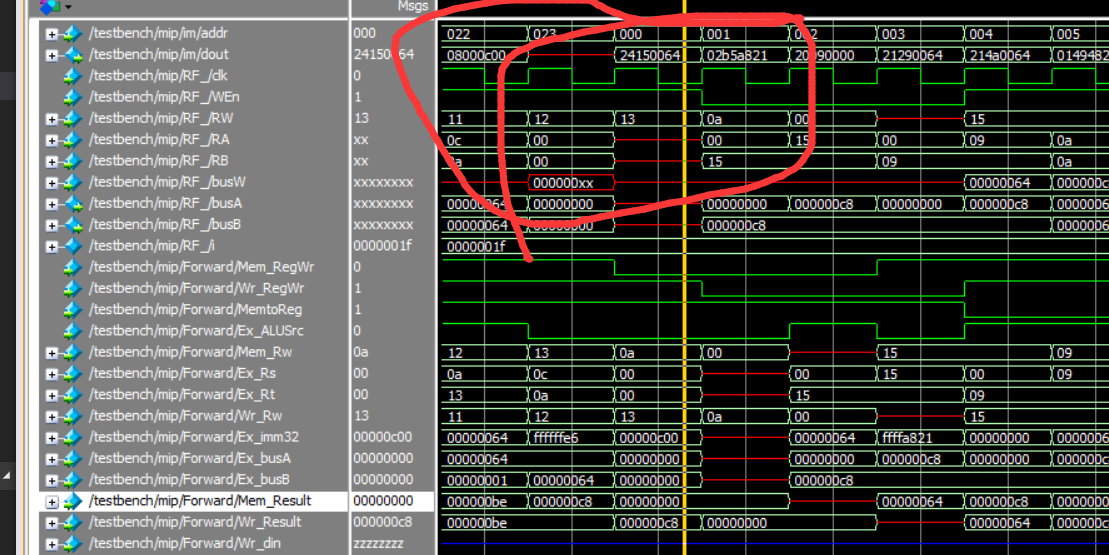
and,andi,or,ori,xor,xori,nor, 7

slt,slti,sltu,sltiu, 4

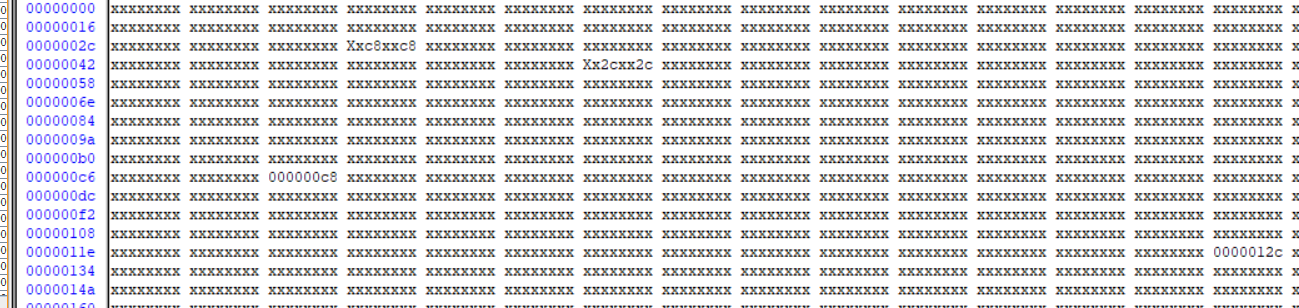
lw,lb,lbu, 3

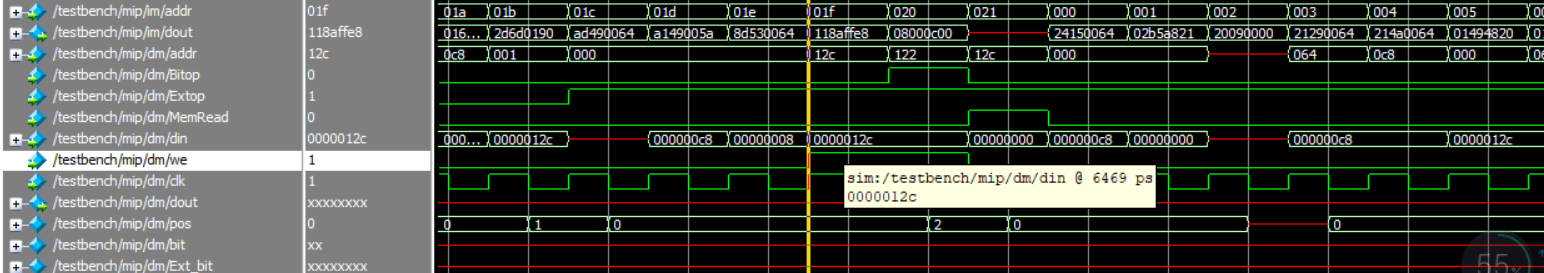
sll,sra,sllv,srav,srlv,srl,6

此处共计27条；



可知j指令跳转成功；1条





通过观测DM内存的数据可以发现，sw装载字和sb装载字节的正确性；（2条）

由于jr和jal单周期的时候也没写出来，所以没能预测，不知道对不对。

实现指令如下：

add，addu, addi, addiu,sub,subu, slt, slti,sltu,sltiu,and, nor, or, xor, andi, ori, xori,sll, srl,sllv, sra, srav, srlv, beq, bne, bgez, bgtz, blez, bltz, lw, sw, lui, j, lb, lbu, sb,

其中分支指令部分未能预测完全；共计在32条。

待验证：jalr, jr,

# 心得体会

奇怪的是，写到这里意味着课设终于要结束了，不用在过着每天写到夜里两三点的困乏，有一种如释重负的感觉同时又有一丝丝的失落。与之前写C、C++和数据结构的课设不同的是，对于那些语言你是可以轻而易举的知道并且解决掉bug的，然而计算机组成原理课设在愉快的假期到来之前，可算是给我一个难忘的煎熬。是的，代码很快就写出来了，但是太多太多的bug了，而且处理冒险问题很让人头疼，别以为自己起的变量名字自己就能不搞乱。试试一个模块几十个变量，然后又十几个模块，各个模块还不是独立的。不单是要求理论知识达到水准，动手实践的时候才能真正体现出是否掌握，大家都知道这发生数据冒险了需要转发到那去。可是，如何实现？如何一同的实现一类冒险，这可不是理论上那么简单的事情，而且还要尽量让自己的硬件模块是合理真实存在的。不得不说，这是一道坎。

因为ModelSim代码编辑不是特别的好用，所以使用了VScode进行编写，风格很清爽，而且相同的变量名选择时候回全部发亮，容易查找一些大小写错误，还有可以全局替代特定的变量，特别好；大大提高了工作效率。还有遇到了好多问题不会处理的时候，查阅资料和网上索引，以及向陈班长请教，总会得到正确的解决方案，交谈过程中收获不少。

通过这次课程设计，深有体会的一点是，一步一个脚印，慢慢地耐心地解决问题，不要浮躁。基本上我是这样完成的，先把各个模块写出来，在原来的单周期上进行修改，并且参照给的课设样例里面的内容。把那些提示的bug统统解决，还有通过VScode编辑器检查变量是否一致，不能跌在这种小错误上。暂时地，先不用去考虑带冒险的流水线，先把流水线写出来，认为的书写无冒险的流水线，看看正确吗？其次才是去考虑带冒险问题。先实现那些简单的R型或I型指令想一些跳转等指令先不考虑，定个小目标25条以上，然后再去解决，虽然最后可能没有36条，但是理论上是知道数据通路如何的，但是，哎有点复杂。

不逼自己，永远都不知道自身的潜力到底有多么的大；不熬夜到两三点，永远都不知道一天能有这么长。其实，回去审视自己这么认真努力地完成课设，不仅仅是一种自豪，更重要的是为又一次成功的坚守底线儿感到欣慰。是的，可能这份课设还存在一些我没能及时发现并且解决的bug，因为最后时间实在是不够了，有点后悔为什么不早点开写课设，可能也是最后考试太多了。但是，遗憾终究是会有的，能让我感到安慰的是真实的付出和认真对待。

参 考 文 献

[1]袁春风，等. 计算机组成与系统结构[M]. 第 2 版，北京：清华大学出版社，2010.

[2]MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set[M]. Revision 2.50, Mountain View, CA: MIPS Technologies Inc., July 1, 2005.

[3]龙芯MIPS指令集及汇编程序设计

附 录

|  |  |  |  |
| --- | --- | --- | --- |
| **汇编指令** | **结果描述** | **机器指令的机器码** | |
| **16进制** | **二进制** |
| addiu $1, $0,#1 | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_0001 |
| sll $2, $1,#4 | [$2] = 0000\_0010H | 00011100 | 0000\_0000\_0000\_0001\_0001\_0001\_0000\_0000 |
| addu $3, $2,$1 | [$3] = 0000\_0011H | 00411821 | 0000\_0000\_0100\_0001\_0001\_1000\_0010\_0001 |
| srl $4, $2,#2 | [$4] = 0000\_0004H | 00022082 | 0000\_0000\_0000\_0010\_0010\_0000\_1000\_0010 |
| slti $25,$4,#5 | [$25] = 0000\_0001H | 28990005 | 0010\_1000\_1001\_1001\_0000\_0000\_0000\_0101 |
| bgez $25,#16 | 跳转到54H | 07210010 | 0000\_0111\_0010\_0001\_0000\_0000\_0001\_0000 |
| subu $5, $3,$4 | [$5] = 0000\_000DH | 00642823 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0011 |
| sw $5, #20($0) | Mem[0000\_0014H] = 0000\_000DH | AC050014 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0100 |
| nor $6, $5,$2 | [$6] = FFFF\_FFE2H | 00A23027 | 0000\_0000\_1010\_0010\_0011\_0000\_0010\_0111 |
| or $7, $6,$3 | [$7] = FFFF\_FFF3H | 00C33825 | 0000\_0000\_1100\_0011\_0011\_1000\_0010\_0101 |
| xor $8, $7,$6 | [$8] = 0000\_0011H | 00E64026 | 0000\_0000\_1110\_0110\_0100\_0000\_0010\_0110 |
| sw $8, #28($0) | Mem[0000\_001CH] = 0000\_0011H | AC08001C | 1010\_1100\_0000\_1000\_0000\_0000\_0001\_1100 |
| beq $8, $3,#2 | 跳转到38H | 11030002 | 0001\_0001\_0000\_0011\_0000\_0000\_0000\_0010 |
| slt $9, $6,$7 | 不执行 | 00C7482A | 0000\_0000\_1100\_0111\_0100\_1000\_0010\_1010 |
| addiu $1, $0,#8 | [$1] = 0000\_0008H | 24010008 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_1000 |
| lw $10,#20($1) | [$10] = 0000\_0011H | 8C2A0014 | 1000\_1100\_0010\_1010\_0000\_0000\_0001\_0100 |
| bne $10,$5,#4 | 跳转到50H | 15450004 | 0001\_0101\_0100\_0101\_0000\_0000\_0000\_0100 |
| and $11,$2,$1 | 不执行 | 00415824 | 0000\_0000\_0100\_0001\_0101\_1000\_0010\_0100 |
| sw $11,#28($1) | 不执行 | AC2B001C | 1010\_1100\_0010\_1011\_0000\_0000\_0001\_1100 |
| sw $4, #16($1) | 不执行 | AC240010 | 1010\_1100\_0010\_0100\_0000\_0000\_0001\_0000 |
| jal #25 | 跳转到64H,  [$31] = 0000\_0054H | 0C000019 | 0000\_1100\_0000\_0000\_0000\_0000\_0001\_1001 |
| lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011\_1100\_0000\_1100\_0000\_0000\_0000\_1100 |
| srav $26,$12,$2 | [$26] = 0000\_000CH | 004CD007 | 0000\_0000\_0100\_1100\_1101\_0000\_0000\_0111 |
| sllv $27,$26,$1 | [$27] = 0000\_0018H | 003AD804 | 0000\_0000\_0011\_1010\_1101\_1000\_0000\_0100 |
| jalr $27 | 跳转到18H ,  [$31] = 0000\_0064H | 0360F809 | 0000\_0011\_0110\_0000\_1111\_1000\_0000\_1001 |
| sb $26,#5($3) | MEM[0000\_0016H] = 000C\_000DH | A07A0005 | 1010\_0000\_0111\_1010\_0000\_0000\_0000\_0101 |
| sltu $13,$3,$3 | [$13] = 0000\_0000H | 0063682B | 0000\_0000\_0110\_0011\_0110\_1000\_0010\_1011 |
| bgtz $13,#3 | 不跳转 | 1DA00003 | 0001\_1101\_1010\_0000\_0000\_0000\_0000\_0011 |
| sllv $14,$6,$4 | [$14] =FFFF\_FE20H | 00867004 | 0000\_0000\_1000\_0110\_0111\_0000\_0000\_0100 |
| sra $15,$14,#2 | [$15] =FFFF\_FF88H | 000E7883 | 0000\_0000\_0000\_1110\_0111\_1000\_1000\_0011 |
| srlv $16,$15,$1 | [$16] =00FF\_FFFFH | 002F8006 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0110 |
| blez $16,#8 | 不跳转 | 1A000008 | 0001\_1010\_0000\_0000\_0000\_0000\_0000\_1000 |
| srav $16,$15,$1 | [$16] =FFFF\_FFFFH | 002F8007 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0111 |
|  |  |  |  |
| addiu $11,$0,#140 | [$11] = 0000\_008CH | 240B008C | 0010\_0100\_0000\_1011\_0000\_0000\_1000\_1100 |
| bltz $16, #6 | 跳转到A0H | 06000006 | 0000\_0110\_0000\_0000\_0000\_0000\_0000\_0110 |
| lw $28,#3($10) | [$28] = 000C\_000DH /000C\_880DH | 8D5C0003 | 1000\_1101\_0101\_1100\_0000\_0000\_0000\_0011 |
| bne $28,$29,#7 | 不跳转/跳转ACH | 179D0007 | 0001\_0111\_1001\_1101\_0000\_0000\_0000\_0111 |
| sb $15,#8($5) | Mem[0000\_0015H] = 0000\_0088H | A0AF0008 | 1010\_0000\_1010\_1111\_0000\_0000\_0000\_1000 |
| lb $18,#8($5) | [$18] =FFFF\_FF88H | 80B20008 | 1000\_0000\_1011\_0010\_0000\_0000\_0000\_1000 |
| lbu $19,#8($5) | [$19] = 0000\_0088H | 90B30008 | 1001\_0000\_1011\_0011\_0000\_0000\_0000\_1000 |
| sltiu $24,$15,#0xFFFF | [$24] = 0000\_0001H | 2DF8FFFF | 0010\_1101\_1111\_1000\_1111\_1111\_1111\_1111 |
| or $29,$12,$5 | [$29] = 000C000DH | 0185E825 | 0000\_0001\_1000\_0101\_1110\_1000\_0010\_0101 |
| jr $11 | 跳转指令8CH | 01600008 | 0000\_0001\_0110\_0000\_0000\_0000\_0000\_1000 |
| andi $20,$15,#0xFFFF | [$20] = 0000\_FF88H | 31F4FFFF | 0011\_0001\_1111\_0100\_1111\_1111\_1111\_1111 |
| ori $21,$15,#0xFFFF | [$21] =FFFF\_FFFFH | 35F5FFFF | 0011\_0101\_1111\_0101\_1111\_1111\_1111\_1111 |
| xori $22,$15,#0xFFFF | [$22] = FFFF\_0077H | 39F6FFFF | 0011\_1001\_1111\_0110\_1111\_1111\_1111\_1111 |
| j #00H | 跳转指令00H | 08000000 | 0000\_1000\_0000\_0000\_0000\_0000\_0000\_0000 |

自写的测试代码：

24150064

02b5a821

20090000

21290064

214a0064

01494820

012a6020

3c1700ff

21290000

0182c823

01896022

01896824

318e00b4

01897825

352b01f4

018f8826

39b403e8

01897827

000c94c0

00149642

000c9143

030c9004

030c9006

030c9007

0189902a

29930066

016c782b

2d6d0190

ad490064

8d530064

118affe9

08000c00