摘 要

#### MIPS 的 CPU 流水线实验的实验目的是实现一个能流水线进行 MIPS 汇编指 令的 CPU 原理工程，并支持常见的 26 条指令。

#### 实验设计主要遵循从设计图到代码、验证几个步骤。 实验内容主要包括：首先进行单周期 CPU 代码的编写；然后通过 Modelsim

#### 验证代码是否正确；接着将单周期 CPU 的代码进行 IP 包装，生成替代 IM，上工

#### 程板查看运行情况；最后修改单周期 CPU 的代码和程序结构，同样使用 Modelsim

#### 进行检验流水线 CPU 代码是否正确。 实验结论为本实验中的流水线 CPU 代码能正常运行所有指令。

**关键词：**MIPS；流水线；CPU

目录

[1. 实验目的和意义 5](#_bookmark0)

[1.1. 实验目的 5](#_bookmark1)

[1.2. 实验意义 5](#_bookmark2)

[2. 实验设计 6](#_bookmark3)

[2.1. 概述 6](#_bookmark4)

[2.2. 实验环境 6](#_bookmark5)

* + 1. [Verilog HDL 简介 7](#_bookmark6)
    2. [ModelSim 简介 7](#_bookmark7)
    3. [MARS 简介 7](#_bookmark8)
    4. [Vivado 简介 8](#_bookmark9)

[2.3. 硬件设计 8](#_bookmark10)

[2.3.1. CPU 总体结构 8](#_bookmark11)

[2.3.2. 程序计数器（Next\_PC） 10](#_bookmark12)

[2.3.3. 指令存储器（Inetr\_Mem） 11](#_bookmark13)

[2.3.4. IF/ID 流水线寄存器（IF\_ID） 11](#_bookmark14)

[2.3.5. 寄存器组（RegFile） 12](#_bookmark15)

[2.3.6. 冒险判断单元（Hazard） 12](#_bookmark16)

[2.3.7. 控制单元（Ctrl） 13](#_bookmark17)

[2.3.8. 转发单元 C/D（ForwardC, ForwardD） 13](#_bookmark18)

[2.3.9. 判断相等单元（Equal） 14](#_bookmark19)

[2.3.10. 扩展单元（Ext） 14](#_bookmark20)

[2.3.11. ID/EX 流水线寄存器（ID\_EX） 15](#_bookmark21)

* + 1. [RegDst 选择器（U\_RegDst） 16](#_bookmark22)
    2. [转发单元 A/B（ForwardA, ForwardB） 17](#_bookmark23)

[2.3.14. ALU 输入选择器 1（ALUSrcA） 17](#_bookmark24)

[2.3.15. ALU 输入选择器 2（ALUSrcB） 18](#_bookmark25)

[2.3.16. 算术逻辑运算单元（ALU） 18](#_bookmark26)

[2.3.17. EX/ME 流水线寄存器（EX\_ME） 19](#_bookmark27)

[2.3.18. 数据存储单元（Mem） 20](#_bookmark28)

[2.3.19. ME/WB 流水线寄存器（ME\_WB） 20](#_bookmark29)

[2.3.20. 选择单元 Mem2Reg（U\_Mem2Reg） 21](#_bookmark30)

[2.3.21. 转发信号决定单元（Forward） 21](#_bookmark31)

[2.3.22. 模型机（MIPS） 22](#_bookmark32)

[2.4. 软件设计 22](#_bookmark33)

[2.4.1. 多路选择器单元（MUX） 22](#_bookmark34)

[2.4.2. 程序计数器（Next\_PC） 23](#_bookmark35)

[2.4.3. 指令存储器（Inetr\_Mem） 24](#_bookmark36)

[2.4.4. IF/ID 流水线寄存器（IF\_ID） 25](#_bookmark37)

[2.4.5. 寄存器组（RegFile） 25](#_bookmark38)

[2.4.6. 冒险判断单元（Hazard） 27](#_bookmark39)

[2.4.7. 控制单元（Ctrl） 28](#_bookmark40)

[2.4.8. 转发单元 C/D（ForwardC, ForwardD） 34](#_bookmark41)

[2.4.9. 判断相等单元（Equal） 34](#_bookmark42)

[2.4.10. 扩展单元（Ext） 34](#_bookmark43)

[2.4.11. ID/EX 流水线寄存器（ID\_EX） 35](#_bookmark44)

[2.4.12. RegDst 选择器，转发单元 A/B，ALUSrcA/B 37](#_bookmark45)

[2.4.13. 算术逻辑运算单元（ALU） 37](#_bookmark46)

[2.4.14. EX/ME 流水线寄存器（EX\_ME） 38](#_bookmark47)

[2.4.15. 数据存储单元（Mem） 39](#_bookmark48)

[2.4.16. ME/WB 流水线寄存器（ME\_WB） 40](#_bookmark49)

[2.4.17. 选择单元 Mem2Reg（U\_Mem2Reg） 41](#_bookmark50)

[2.4.18. 转发信号决定单元（Forward） 41](#_bookmark51)

[2.4.19. 模型机（MIPS） 42](#_bookmark52)

[2.5. 设计结果分析 48](#_bookmark53)

[2.5.1. 测试文件 48](#_bookmark54)

[2.5.2. 测试机器码 49](#_bookmark55)

[2.5.3. 测试结果分析 50](#_bookmark56)

[2.6. FPGA 开发板测试 57](#_bookmark58)

[2.6.1. FPGA 开发板介绍及照片 57](#_bookmark59)

[2.6.2. 测试显示方案 57](#_bookmark60)

[2.6.3. 连接开关、灯泡和七段显示管 58](#_bookmark61)

[2.6.3.1. 连接框架(Verilog 文件连接方式) 58](#_bookmark62)

* + - 1. [文件 TESTBENCH.v 58](#_bookmark63)
      2. [文件 clk\_div.v 59](#_bookmark64)

[2.6.3.4. 文件 seg7x16.v 60](#_bookmark65)

* + - 1. [文件 icf.xdc 62](#_bookmark66)
      2. [文件 Bubble\_sort.asm 64](#_bookmark67)
      3. [文件 Test\_6\_Instr.coe 65](#_bookmark68)

[2.6.4. 实验成果 66](#_bookmark69)

[3. 参考文献 70](#_bookmark70)

[教师评语评分 71](#_bookmark71)

# 1. 实验目的和意义

## 1.1. 实验目的

#### 本实验通过课前阅读材料，查阅资料，课中编写代码并进行调试，实现能对

#### MIPS 汇编指令进行处理的流水线 CPU。

## 1.2. 实验意义

#### 本实验通过实现能对 MIPS 汇编指令进行处理的流水线 CPU，提升自己对于 计算机组成原理中 CPU 部分的理解，提升自己的编程技巧和对于仅有数据变化 的调试能力，也对于各类软件有一个较浅的理解。

# 2. 实验设计

## 2.1. 概述

#### 本次实验实现 MIPS 指令集下的流水线 CPU，考虑数据转发和 ID 级控制冒 险。本 CPU 支持 MIPS 指令集下的 25 条指令。

#### 此处主要是一些硬件、软件配置的说明，以及 github 的项目地址。 项目分为三个文件夹：第一个文件夹是 Pipeline，这部分对应 CPU 流水线

#### 的部分，只有仿真，没有对于 Vivado 进行适配，自然也没有上板子。第二个 文件夹是 ProjectSingalCPU，这部分对应单周期 CPU 的代码编写，已经上了 板子。第三个文件夹是 Singal，这部分也对应 CPU 的代码编写，但是仅仅进 行了 modelsim 仿真，也作为仿真失败且无法复原代码时的备份文件。

#### 注意不保证 Singal 文件夹下和 ProjectSingalCPU 目录下的代码是一致 的。

#### 一些非实验环境信息（如 github 存储地址）如下：

|  |  |
| --- | --- |
| **表 2-1 本项目的非实验环境的信息** | |
| Github 地址 | <https://github.com/Ivy233/CSExperiment_MIPS> |
| 支持的指令 | 运算类：add(i)(u), sub(u), slt(i)(u),  and(i), or(i), xor(i), sll, srl, sra, lui, lw, sw  分支类：bne, beq  跳转类：j |

## 2.2. 实验环境

#### 本实验使用 VSCode 进行代码编写，在 ModelSim 下进行 Verilog 源码调试 汇编源码并监视波形图，对照 Mars 的结果。其中 FPGA 测试环节在 Win10 专业 版下采用 Vivado 对开发板进行封装 IP 和生成替换 IM，并上板子调试。

|  |  |
| --- | --- |
| **表 2-2 程序计数器模块接口定义** | |
| CPU | Core i5-7300HQ |
| Windows | ModelSim 在 Win10 17763.475 家庭版下。 |

|  |  |
| --- | --- |
|  | Vivado 在 Win10 17763.475 专业版下。 |
| ModelSim 版本 | ModelSim SE-64 10.4 |
| Vivado 版本 | Vivado 2017.1 |
| 代码编辑器 | VSCode |
| 开发板型号 | Artic-7 FPGA |
| Mars 版本 | 4.5 |

### Verilog HDL 简介

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

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

### ModelSim 简介

#### ModelSim 是 Mentor Graphics 的多语言 HDL 仿真环境，用于仿真硬件描 述语言，如 VHDL，Verilog 和 SystemC，并包含一个内置的 C 调试器。ModelSim 可以单独使用，也可以与 Intel Quartus Prime，Xilinx ISE 或 Xilinx Vivado 一起使用。使用图形用户界面（GUI）执行模拟，或使用脚本自动执行模 拟。

### MARS 简介

#### MARS（MIPS 汇编程序和运行时模拟器）是一个开发工具，为 MIPS 程序员 提供了一个用于创建和测试软件程序的直观环境。没有必要进行安装操作，因为 您只需双击下载的.jar 即可立即访问主应用程序窗口。 另一方面，您必须安装 Java Runtime Environment（JRE）。

### Vivado 简介

#### Vivado Design Suite 是 Xilinx 生产的用于 HDL 设计综合和分析的软件 套件，取代了 Xilinx ISE，具有片上系统和高级综合系统的附加功能。 Vivado 代表了对整个设计流程的重新思考和重新思考（与 ISE 相比），并且被评论者描 述为“精心构思，紧密集成，快速，可扩展，可维护和直观”。

## 2.3. 硬件设计

### 2.3.1. CPU 总体结构

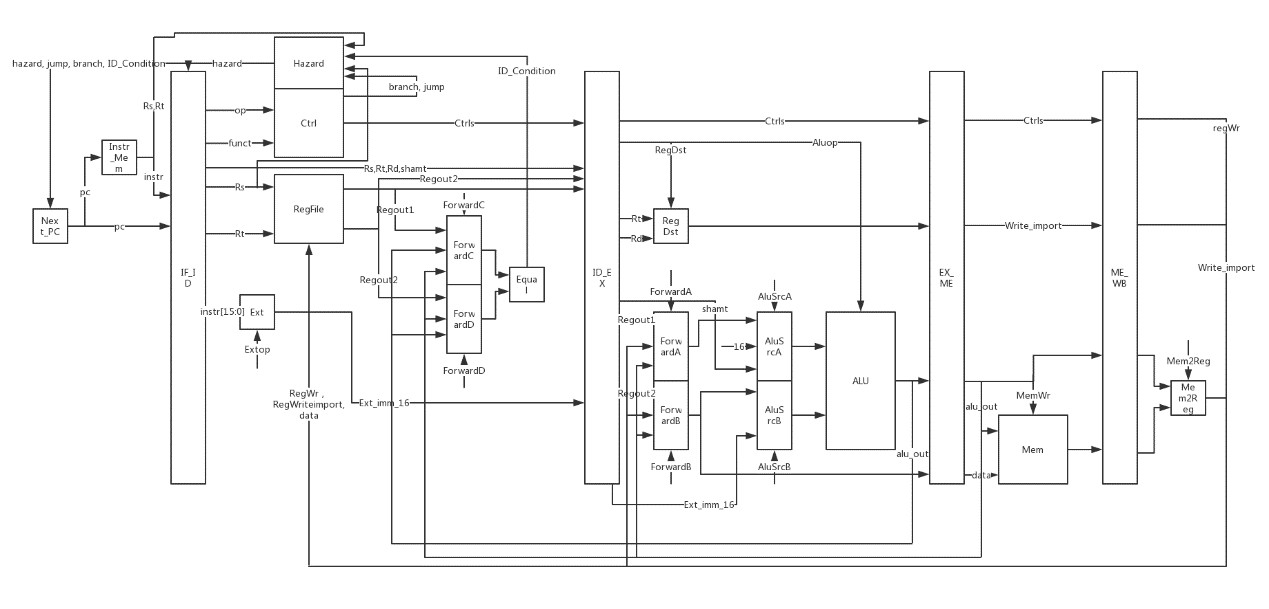


图 2-3-1 CPU 总体设计-总览图

#### 图片可能看不清楚，因此我分成了两部分以方便查看。图片中没有 Forward

#### 单元，主要原因有两点:

#### 1) Forward 单元不局限于某一个或者两个流水线阶段，因此不是很容易画出。

#### 2) Forward 单元加入以后，布线很难整理。 具体的内容可见[转发信号决定单元（Forward）](#_bookmark31)。

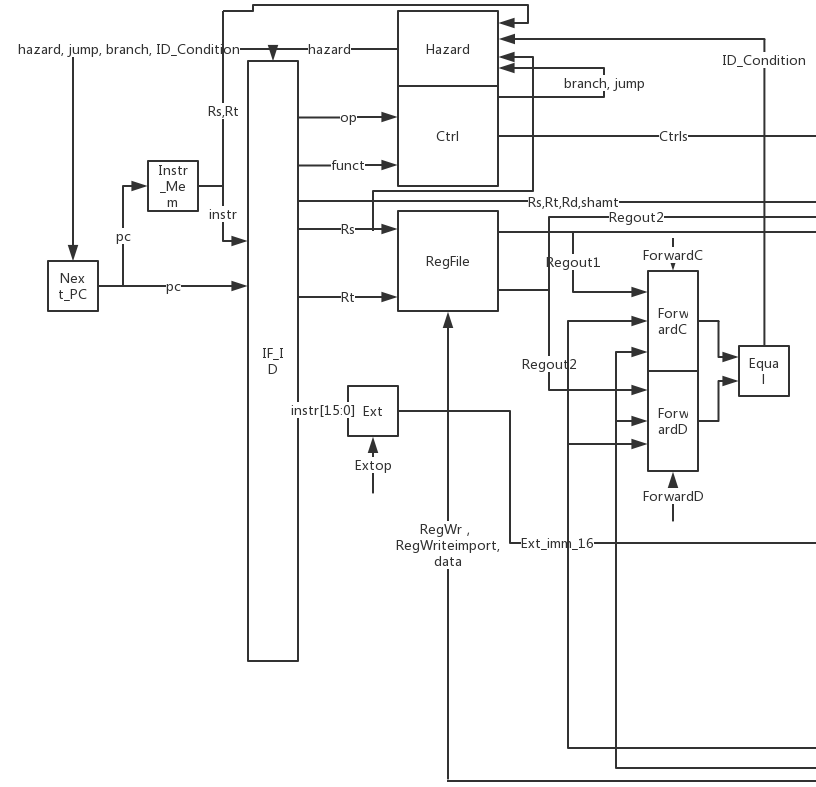


图 2-3-2 CPU 总体设计-左半边

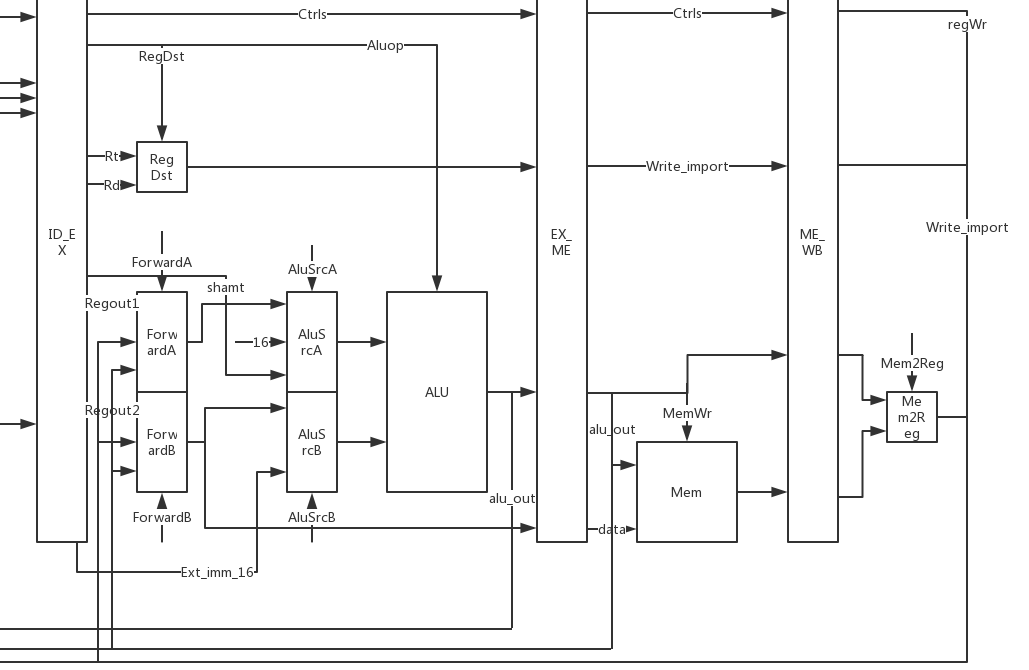


图 2-3-3 CPU 总体设计-右半边

### 2.3.2. 程序计数器（Next\_PC）

**1) 功能描述**

#### 程序计数器在 MIPS.v 对应的模块是 U\_Next\_PC，负责 PC 更新。除了 PC 的 正常+4 之外，还要根据跳转、分支、冒险等进行不同的处理，以实现转移和暂 停的功能。仅在 clk 或者 rst 为 posedge 时更新 PC。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-1 程序计数器模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| rst | Input | 重置信号 |
| hazard | Input | 冒险判断 |
| jump | Input | 是否跳转 |
| jump2where[31:0] | Input | 跳转到哪里 |
| branch | Input | 是否分支 |
| branch2where[31:0] | Input | 分支到哪里 |

|  |  |  |
| --- | --- | --- |
| pc[31:0] | Output | 需要更新的 pc |

### 2.3.3. 指令存储器（Inetr\_Mem）

**1) 功能描述**

#### 指令存储器在 MIPS.v 对应的模块是 U\_Instr\_Mem，负责从指令文件中读取 对应的指令。其中每个 PC 地址对应一条机器指令，每条指令长度为 32bit。在 PC 变化时，立即读取指令。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-2 指令存储器模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| pc[9:0] | Input | PC 地址 |
| instr[31:0] | Ouput | 读取到的指令 |

### 2.3.4. IF/ID 流水线寄存器（IF\_ID）

**1) 功能描述**

#### IF/ID 流水线寄存器在 MIPS.v 中的模块是 U\_IFID，负责将来自 IF 级的数 据传递到 ID 级。需要 clk 才能进行传递。同时由于分支预测在 hazard，需要 在冒险时清空此寄存器的内容以消除误读指令的影响。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-3 IF/ID 流水线寄存器模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| rst | Input | 重置信号 |
| hazard | Input | 冒险判断 |
| regin1[31:0] | Input | PC 读入（此处+4） |
| regin2[31:0] | Input | 指令读入 |
| regout1[31:0] | Output | PC 输出 |
| regout2[31:0] | Output | 指令输出 |

### 2.3.5. 寄存器组（RegFile）

**1) 功能描述**

#### 寄存器组在 MIPS.v 对应的模块是 U\_RegFile，主要存储、修改 32 个 32bit

#### 寄存器的内容，并同时能读取两个位置的寄存器 32bit 数据。 在任何时候都能读取数据。但是只有在 regWr 为 1，clk 为 negedge 时可以

#### 写入，如果为 posedge 会引发一个问题，具体会在[测试结果分析](#_bookmark56)中讲述。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-4 寄存器组模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| readimport1[4:0] | Input | 读端口 1 |
| readimport2[4:0] | Input | 读端口 2 |
| writeimport[4:0] | Input | 写端口 |
| Writedata[31:0] | Input | 写入数据 |
| regWr | Input | 能否写入 |
| regfile\_output1[31:0] | Output | 读出数据 1 |
| regfile\_output2[31:0] | Output | 读出数据 2 |

### 2.3.6. 冒险判断单元（Hazard）

**1) 功能描述**

#### 冒险判断单元在 MIPS.v 对应的模块是 U\_Hazard，会针对当前的跳转、分 支控制信号，寄存器写入，在 IF 的 Rs 和 Rt，ID 的 Rt 进行冒险判断。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-5 冒险判断模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| jump | Input | 是否跳转 |
| branch | Input | 是否分支 |
| mem2reg | Input | 写入寄存器的状况 |
| IF\_Rs[4:0] | Input | IF 级的 Rs |

|  |  |  |
| --- | --- | --- |
| IF\_Rt[4:0] | Input | IF 级的 Rt |
| ID\_Rt[4:0] | Input | ID 级的 Rt |
| hazard | Output | 是否冒险 |

### 2.3.7. 控制单元（Ctrl）

**1) 功能描述**

#### 控制单元在 MIPS.v 中对应的模块是 U\_Ctrl，会针对指令类型来推算各类 控制信号，具体的会在下表中列举。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-6 控制单元模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| op | Input | instr 的前 6 位 |
| funct | Output | instr 的后 6 位 |
| Ctrl\_alu[3:0] | Output | alu 运算类型 |
| Ctrl\_regDst | Output | 写回寄存器对于 rt 还  是 rd 的判断 |
| Ctrl\_aluSrcA[1:0] | Output | ALU 输入 1 选择端口 |
| Ctrl\_aluSrcB[1:0] | Output | ALU 输入 2 选择端口 |
| Ctrl\_Mem2Reg | Output | 是否为内存写入寄存器 |
| Ctrl\_ext | Output | 扩展类型 |
| Ctrl\_regWr | Output | 寄存器是否写入 |
| Ctrl\_MemWr | Output | 内存是否写入 |
| Ctrl\_branch[1:0] | Output | 分支类型 |
| Ctrl\_jump | Output | 是否跳转 |

**2.3.8. 转发单元 C/D（ForwardC, ForwardD） 1) 功能描述**

#### 转发单元 C/D 在 MIPS.v 的模块是 U\_ForwardC 和 U\_ForwardD， 基于 MUX 实现，针对寄存器组的输出进行优先转发，以防止数据不是最新导致的分支判断 错误。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-7 转发单元 C/D 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | 寄存器的输出 1 或者 2 |
| b[31:0] | Input | EX 级的 ALU 输出 |
| c[31:0] | Input | ME 级的 ALU 输出 |
| d[31:0] | Input | 写回寄存器的内容 |
| Ctrl[1:0] | Input | MUX 控制信号，0-3 分  别对应 a-d |
| out[31:0] | Output | 转发单元输出 |

### 2.3.9. 判断相等单元（Equal）

**1) 功能描述**

#### 判断相等单元在 MIPS.v 对应的模块是 U\_Equal，针对数据转发进行相等判 断，如果相等则输出 1。输出会进入 PC 处理，与 Ctrl 输出的 branch 信号一起 判断是否分支。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-8 判断模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| input1[31:0] | Input | ForwardC 的输出 |
| input2[31:0] | Input | ForwardD 的输出 |
| equal | Output | 比较结果 |

### 2.3.10. 扩展单元（Ext）

**1) 功能描述**

#### 扩展单元在 MIPS.v 对应的模块是 U\_Ext，针对 16 位数据进行扩展，包括 符号扩展和零扩展，仅用于 I 型指令，其他情况下会被 AluSrc 屏蔽。

**2) 模块接口**

**表 2-3-9 扩展单元接口定义**

|  |  |  |
| --- | --- | --- |
| 信号名 | 方向 | 描述 |
| input0[15:0] | Input | 16 位输入 |
| op | Input | 扩展类型：  0：零扩展  1：符号扩展 |
| out[31:0] | Output | 扩展输出 |

### 2.3.11. ID/EX 流水线寄存器（ID\_EX）

**1) 功能描述**

#### ID/EX 流水线寄存器在 MIPS.v 对应的是 U\_IDEX，负责将来自 ID 级的数据 传递到 EX 级。需要 clk 才能进行传递。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-10 ID/EX 流水线寄存器接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| rst | Input | 重置信号 |
| regin1[31:0] | Input | 寄存器输出 1 |
| regin2[31:0] | Input | 寄存器输出 2 |
| regin3[31:0] | Input | 扩展输出 |
| regin4[4:0] | Input | ID 级的 Rs |
| regin5[4:0] | Input | ID 级的 Rt |
| regin6[4:0] | Input | ID 级的 Rd |
| regin7[4:0] | Input | ID 级的 shamt |
| regdstin | Input | regdst 控制信号 |
| aluopin[3:0] | Input | ALU 控制信号 |
| alusrcain[1:0] | Input | ALU 输入 1 控制信号 |
| alusrcbin[1:0] | Input | ALU 输入 2 控制信号 |
| mem2regin | Input | Mem2reg 控制信号 |
| regwrin | Input | 寄存器写入控制信号 |

|  |  |  |
| --- | --- | --- |
| memwrin | Input | 内存写入控制信号 |
| regout1[31:0] | Output | 寄存器输出 1 |
| regout2[31:0] | Output | 寄存器输出 2 |
| regout3[31:0] | Output | 扩展输出 |
| regout4[4:0] | Output | EX 级的 Rs |
| regout5[4:0] | Output | EX 级的 Rt |
| regout6[4:0] | Output | EX 级的 Rd |
| regout7[4:0] | Output | EX 级的 shamt |
| regdstout | Output | regdst 控制信号 |
| aluopout[3:0] | Output | ALU 控制信号 |
| alusrcaout[1:0] | Output | ALU 输入 1 控制信号 |
| alusrcbout[1:0] | Output | ALU 输入 2 控制信号 |
| mem2regout | Output | Mem2reg 控制信号 |
| regwrout | Output | 寄存器写入控制信号 |
| memwrout | Output | 内存写入控制信号 |

### RegDst 选择器（U\_RegDst）

**1) 功能描述**

#### RegDst 选择器基于 MUX，会针对是否为 I 型指令进行寄存器写入端口的选 择。其中的控制信号原本只有一位，在前面补上前导 0 即可。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-11 选择单元 RegDst 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | EX 级的 Rt |
| b[31:0] | Input | EX 级的 Rd |
| c[31:0] | Input | 空 |
| d[31:0] | Input | 空 |
| Ctrl[1:0] | Input | MUX 控制信号，选择写  入寄存器的端口 |

|  |  |  |
| --- | --- | --- |
| out[31:0] | Output | 输出 |

### 转发单元 A/B（ForwardA, ForwardB）

**1) 功能描述**

#### 转发单元 A/B 在 MIPS.v 的模块是 U\_ForwardA 和 U\_ForwardB， 基于 MUX 实现，针对寄存器组的输出进行优先转发，以防止数据不是最新导致的 ALU 计算 错误。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-12 转发单元 A/B 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | EX 级寄存器输出 1/2 |
| b[31:0] | Input | 写入寄存器的数据 |
| c[31:0] | Input | ME 级 ALU 输出 |
| d[31:0] | Input | 空 |
| Ctrl[1:0] | Input | MUX 控制信号 |
| out[31:0] | Output | 转发单元输出 |

* + 1. **ALU 输入选择器 1（ALUSrcA） 1) 功能描述**

#### ALU 输入选择器 1 在 MIPS.v 的模块是 U\_ALUSrcA， 基于 MUX 实现，针对 位移、lui 指令和其他指令进行不同的处理。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-13 ALU 输入选择器 1 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | ForwardA 的输出 |
| b[31:0] | Input | 16，用于 lui |
| c[31:0] | Input | EX 级 shamt，零扩展  为 32 位 |
| d[31:0] | Input | 空 |

|  |  |  |
| --- | --- | --- |
| Ctrl[1:0] | Input | MUX 控制信号 |
| out[31:0] | Output | 选择单元 A 输出 |

### ALU 输入选择器 2（ALUSrcB）

**1) 功能描述**

#### ALU 输入选择器 2 在 MIPS.v 的模块是 U\_ALUSrcB， 基于 MUX 实现，针对

#### I 型指令和其他指令进行不同的处理。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-14 ALU 输入选择器 2 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | ForwardB 的输出 |
| b[31:0] | Input | EX 级扩展单元输出 |
| c[31:0] | Input | 空 |
| d[31:0] | Input | 空 |
| Ctrl[1:0] | Input | MUX 控制信号 |
| out[31:0] | Output | 选择单元 B 输出 |

### 2.3.16. 算术逻辑运算单元（ALU）

**1) 功能描述**

#### 算术逻辑运算单元在 MIPS.v 的模块是 U\_ALU，针对不同的 ALU 控制信号进 行不同的运算，输入两个数，输出一个数。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-15 算术逻辑运算单元模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| input1[31:0] | Input | ALUSrcA 的输出 |
| input2[31:0] | Input | ALUSrcB 的输出 |
| aluop[31:0] | Input | ALU 控制信号：  0: input1+input2  1: input1-input2  2: input2<<input1[4:0] |

|  |  |  |
| --- | --- | --- |
|  |  | 3: input2>>input1[4:0]  4: slt  5: and  6: or  7: xor  8: sltu(slt 需要注意符号)  9: input2>>>input1[4:0] |
| out[31:0] | Output | ALU 输出 |

### 2.3.17. EX/ME 流水线寄存器（EX\_ME）

**1) 功能描述**

#### EX/ME 流水线寄存器在 MIPS.v 对应的是 U\_EXME，负责将来自 EX 级的数据 传递到 ME 级。需要 clk 才能进行传递。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-16 EX/ME 流水线寄存器接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| rst | Input | 重置信号 |
| regin1[31:0] | Input | ALU 输出 |
| regin2[31:0] | Input | ForwardB 的输出 |
| regin3[4:0] | Input | 寄存器写入端口 |
| mem2regin | Input | Mem2reg 控制信号 |
| regwrin | Input | 寄存器写入控制信号 |
| memwrin | Input | 内存写入控制信号 |
| regout1[31:0] | Output | ME 级 ALU 输出 |
| regout2[31:0] | Output | ForwardB 的输出 |
| regout3[4:0] | Output | 寄存器写入端口 |
| mem2regout | Output | Mem2reg 控制信号 |
| regwrout | Output | 寄存器写入控制信号 |
| memwrout | Output | 内存写入控制信号 |

### 2.3.18. 数据存储单元（Mem）

**1) 功能描述**

#### 数据存储单元在 MIPS.v 的模块是 U\_Mem，负责存储足量的数据，写入，修 改数据，或者读取一个数据。写入和读取的端口是共用的。只有在 clk 为 negedge 并且 memWr 为 1 时才允许写入，但是在任何时候都可以读出。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-17 数据存储单元模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| import[31:0] | Input | 写入/读出端口 |
| write\_data[31:0] | Input | 写入数据 |
| memWr | Input | 内存写入信号 |
| read\_out[31:0] | Output | 读出数据 |

### 2.3.19. ME/WB 流水线寄存器（ME\_WB）

**1) 功能描述**

#### ME/WB 流水线寄存器在 MIPS.v 对应的是 U\_MEWB，负责将来自 ME 级的数据 传递到 WB 级。需要 clk 才能进行传递。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-18 ME/WB 流水线寄存器接口定义** | | |
| 信号名 | 方向 | 描述 |
| clk | Input | 时钟信号 |
| rst | Input | 重置信号 |
| regin1[31:0] | Input | ME 级 ALU 输出 |
| regin2[31:0] | Input | Mem 输出 |
| regin3[4:0] | Input | 寄存器写入端口 |
| mem2regin | Input | Mem2reg 控制信号 |
| regwrin | Input | 寄存器写入控制信号 |
| regout1[31:0] | Output | WB 级 ALU 输出 |

|  |  |  |
| --- | --- | --- |
| regout2[31:0] | Output | WB 级 Mem 输出 |
| regout3[4:0] | Output | 寄存器写入端口 |
| mem2regout | Output | Mem2reg 控制信号 |
| regwrout | Output | 寄存器写入控制信号 |

### 2.3.20. 选择单元 Mem2Reg（U\_Mem2Reg）

**1) 功能描述**

#### 选择单元 Mem2Reg 基于 MUX，选择是否将内存读出的数据写入寄存器。由于 Mem2Reg 只有一位，因此补充前导 0 以避免高位错误。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-19 选择单元 Mem2Reg 模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| a[31:0] | Input | WB 级 ALU 输出 |
| b[31:0] | Input | WB 级 Mem 输出 |
| c[31:0] | Input | 空 |
| d[31:0] | Input | 空 |
| Ctrl[1:0] | Input | MUX 控制信号 |
| out[31:0] | Output | 输出 |

### 2.3.21. 转发信号决定单元（Forward）

**1) 功能描述**

#### 转发信号决定单元在 MIPS.v 中的模块是 U\_Forwrad，负责决定之前

#### ForwardA/B/C/D 的选择端口。

**2) 模块接口**

|  |  |  |
| --- | --- | --- |
| **表 2-3-20 转发信号决定单元模块接口定义** | | |
| 信号名 | 方向 | 描述 |
| ID\_Rs[4:0] | Input | ID 级 Rs |
| ID\_Rt[4:0] | Input | ID 级 Rt |
| EX\_Rs[4:0] | Input | EX 级 Rs |

|  |  |  |
| --- | --- | --- |
| EX\_Rt[4:0] | Input | EX 级 Rt |
| EX\_writeimport[4:0] | Input | EX 级写入寄存器端口 |
| ME\_writeimport[4:0] | Input | ME 级写入寄存器端口 |
| WB\_writeimport[4:0] | Input | WB 级写入寄存器端口 |
| EXRegWr | Input | EX 级寄存器写入信号 |
| MERegWr | Input | ME 级寄存器写入信号 |
| WBRegWr | Input | WB 级寄存器写入信号 |
| ForwardA[1:0] | Output | 输出 A |
| ForwardB[1:0] | Output | 输出 B |
| ForwardC[1:0] | Output | 输出 C |
| ForwardD[1:0] | Output | 输出 D |

### 2.3.22. 模型机（MIPS）

**1) 功能描述**

#### 模型机即 MIPS.v 本身的模组，负责将以上每一组件组装起来。

**2) 模块接口：无**

## 2.4. 软件设计

### 2.4.1. 多路选择器单元（MUX）

**1) 实现代码**

module Next\_PC( input clk, input rst, input hazard, input jump,

input[31:0] jump2where, input branch, input[31:0] branch2where, output reg[31:0] pc

);

always @(posedge rst) begin if(rst)begin

pc <= 32'h00003000;

end

end

always @(posedge clk) begin

if(jump == 1'b1) begin pc <= jump2where;

end*//j*

else if(branch)begin pc <= branch2where;

end

else if(hazard)begin pc <= pc;

end

else begin

pc <= pc + 3'b100; end*//other*

end

endmodule *// Next\_PC*

**2) 原理说明**

#### 在不同的选择控制下应当输出多种结果。

#### MUX 是流水线中很重要的组成部分，如 ForwardA/B/C/D, ALUSrcA/B, RegDst, Mem2Reg 都是以 MUX 实现的。如果有可选项不足 4 个的不需要过多担 心，因为只需要保证可选值不超过边界即可。

### 2.4.2. 程序计数器（Next\_PC）

**1) 实现代码**

module Next\_PC( input clk, input rst, input hazard, input jump,

input[31:0] jump2where, input branch, input[31:0] branch2where, output reg[31:0] pc

);

always @(posedge rst) begin if(rst)begin

pc <= 32'h00003000;

end

end

always @(posedge clk) begin if(jump == 1'b1) begin

pc <= jump2where; end*//j*

else if(branch)begin pc <= branch2where;

end

else if(hazard)begin pc <= pc;

end

else begin

pc <= pc + 3'b100; end*//other*

end

endmodule *// Next\_PC*

**2) 原理说明**

#### 在信号 rst 发生0 → 1的变化时，pc 应当重置。

#### 在 clk 发生0 → 1的变化时，应当按照jump → branch → hazard →其他的顺 序求值，实际上，前面三者不可能同时为一。注意这里的 branch 不是 Ctrl 直 接输出的 branch，而是与两个寄存器读出的值是否相等进行综合判断处理的， 就像：

.branch((ID\_Ctrl\_branch == 2'b01 && ID\_Condition == 1'b1)

|| (ID\_Ctrl\_branch == 2'b10 && ID\_Condition == 1'b0)),

#### 信号 jump 和 branch(从 ID 级 Ctrl 直接输出)可以通过控制 Ctrl 的输出方

#### 式让两者不同时为 1。此时再加入 hazard，如果 hazard 为 1，而且 jump 和 branch 都为 0，那么此时是读取 lw，确实需要阻塞一个周期；而两者有一个不 为 1 时，应当优先考虑前两者。

### 2.4.3. 指令存储器（Inetr\_Mem）

**1) 实现代码**

module Instr\_Mem( input[9:0] pc,

output reg[31:0] instr

);

reg[31:0] Instrs[1023:0]; always@(pc)begin

instr = Instrs[pc];

$display("pc=0x%8X, instr=0x%8X", pc, instr);

end

endmodule *// Instr\_Mem*

**2) 原理说明**

#### 只要 pc 更新，就应当立即从 Instrs 取出指令。Instrs 在 MIPS 运行之初 就应当从指令文件中读取并初始化。

#### 实际上，PC 的 00003000 中的 3 并没有很多用处，因为 3 刚好是在 10-11 位 为 1，而输入的 PC 刚好只读取到第 9 位，可能 3000 更多是为了和 mars 匹配。

#### 但是实际上如果 Instrs 的下标超过了 1024，那么 3 就会影响指令的读取。

### 2.4.4. IF/ID 流水线寄存器（IF\_ID）

**1) 实现代码**

module IF\_ID( input clk, input rst, input hazard,

input[31:0] regin1, input[31:0] regin2, output reg[31:0] regout1, output reg[31:0] regout2

);

initial begin regout1 <= 0;

regout2 <= 0;

end

always @(posedge rst) begin regout1 <= 0;

regout2 <= 0;

end

always @(posedge clk) begin if(hazard)begin

regout1 = 0;

regout2 = 0; end else begin

regout1 = regin1; regout2 = regin2;

end

end

endmodule *// IF\_ID\_Reg*

**2) 原理说明**

#### 如果不初始化为 0，那么会发生灾难性的后果。这里会沿着ID\_IF → RegFile → hazard → PC一路将 32’hxxxxxxxx 传递下去，从而导致指令无法读取。

#### 同时除此以外，还需要注意在冒险发生时需要将寄存器清空。

### 2.4.5. 寄存器组（RegFile）

**1) 实现代码**

module RegFile( input clk,

input[4:0] readimport1,

input[4:0] readimport2, input[4:0] writeimport,

input[31:0] Writedata, input regWr,

output[31:0] regfile\_out1, output[31:0] regfile\_out2

);

reg[31:0] register[0:31]; integer i;

initial begin

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

end

end

assign regfile\_out1 = (readimport1 != 0) ? register[readimport1] :

0;

assign regfile\_out2 = (readimport2 != 0) ? register[readimport2] :

0;

always @(negedge clk ) begin

if(regWr == 1'b1 && writeimport != 0)begin register[writeimport] = Writedata;

end

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

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

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

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

$display("R[16-23]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

register[16], register[17], register[18], register[19], register[20], register[21], register[22], register[23]);

$display("R[24-31]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

register[24], register[25], register[26], register[27], register[28], register[29], register[30], register[31]);

$display("writeimport=%8X Read\_import=%8X, %8X", writeimport, readimport1, readimport2);

$display("regWr =%8X", regWr);

end

endmodule *// RegFile*

**2) 原理说明**

#### 这里和 Next\_PC 一样需要有很多注意的地方。首先是寄存器必须清空，否则 会发生灾难性的后果，因为这会直接影响到 hazard 的判断， 于是会沿着 RegFile → hazard → PC的顺序出现 32’hxxxxxxxx，进而无法读取指令。

#### 除此以外，必须在 clk 为 negedge 时写入，否则调试时可能会发生数据跟不 上的情况，具体可[点击此处](#_bookmark57)。若 clk 改成 posedge 写入，那么会导致在写入的

#### 时候刚好读出数据，而且转发单元已经丢失了全部的最新数据，所以拿到的永远 是过时的数据。此处考虑到所有的非写入处理都在 posedge 进行处理，因此只 需要在 negedge 时，即卡在写入数据产生和从 ID/EX 输出的两个 posedge 之间 写入即可。

#### 其他的就是很简单的 0 号寄存器不能写入，必须保持为 0 之类的人为规定的 特性。

### 2.4.6. 冒险判断单元（Hazard）

**1) 实现代码**

module Hazard ( input jump,

input branch,

input mem2reg, input[4:0] IF\_Rs, input[4:0] IF\_Rt, input[4:0] ID\_Rt, output reg hazard

);

initial begin hazard <= 1'b0;

end

always @(\*) begin

if (mem2reg && (ID\_Rt != 0) && (ID\_Rt == IF\_Rs || ID\_Rt ==

IF\_Rt) ) begin

hazard <= 1; end else begin

hazard <= jump || branch;

end

end endmodule

**2) 原理说明**

#### 单元 hazard 是一个很重要的单元，因为它决定了跳转如何执行，如何在跳 转时抹去可能读取错误的指令的影响。再确认冒险发生之后，流水线阻塞一个周 期，之后重新获取上一次读到的指令。

#### 需要进行冒险的只有两类：一种是 lw 指令得到确认之后，下一条指令立即使 用了这个数；另一种是跳转和确认发生的分支（注意这里的 branch 也不是 Ctrl 单元直接获取的，而是 Next\_PC 中的 branch）。

### 2.4.7. 控制单元（Ctrl）

**1) 实现代码（此处代码较长，点击跳转到** [**2.4.8**](#_bookmark41)**）**

module Ctrl(

input[5:0] op, input[5:0] funct,

output reg[3:0] Ctrl\_alu, output reg Ctrl\_regDst, output reg[1:0] Ctrl\_aluSrcA, output reg[1:0] Ctrl\_aluSrcB, output reg Ctrl\_Mem2Reg, output reg Ctrl\_ext,

output reg Ctrl\_regWr, output reg Ctrl\_MemWr, output reg[1:0] Ctrl\_branch, output reg Ctrl\_jump

);

*// Operation code;*

parameter R ADDI

ADDIU SLTI SLTIU ANDI ORI XORI LUI LW

SW BEQ BNE J

*// Function code;*

= 6'b000000,

= 6'b001000,

= 6'b001001,

= 6'b001010,

= 6'b001011,

= 6'b001100,

= 6'b001101,

= 6'b001110,

= 6'b001111,

= 6'b100011,

= 6'b101011,

= 6'b000100,

= 6'b000101,

= 6'b000010;

parameter ADD = 6'b100000,

ADDU = 6'b100001, SUB = 6'b100010, SUBU = 6'b100011, AND = 6'b100100,

OR = 6'b100101,

XOR = 6'b100110,

NOR = 6'b100111,

SLT = 6'b101010, SLTU = 6'b101011, SLL = 6'b000000,

SRL = 6'b000010,

SRA = 6'b000011;

initial begin Ctrl\_regDst <= 1'b0;

Ctrl\_aluSrcB <= 2'b00; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_ext <= 1'b0;*//x* Ctrl\_MemWr <= 1'b0; Ctrl\_regWr <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0000;

end always@(\*)begin

case (op)

R:begin

Ctrl\_regDst <= 1'b1; Ctrl\_aluSrcB <= 2'b00; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_ext <= 1'b0;*//x* Ctrl\_MemWr <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0; case (funct)

ADD:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0000;

end ADDU:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0000;

end SUB:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0001;

end SUBU:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0001;

end

SLL:begin

Ctrl\_aluSrcA <= 2'b10; Ctrl\_alu <= 4'b0010;

end SRL:begin

Ctrl\_aluSrcA <= 2'b10; Ctrl\_alu <= 4'b0011;

end SRL:begin

Ctrl\_aluSrcA <= 2'b10; Ctrl\_alu <= 4'b0011;

end

AND:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0101;

end OR:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0110;

end XOR:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0111;

end NOR:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b1010;

end

SLT:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b0100;

end SLTU:begin

Ctrl\_aluSrcA <= 2'b00; Ctrl\_alu <= 4'b1000;

end

end endcase

ADDI:begin

Ctrl\_alu <= 4'b0000; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end ADDIU:begin

Ctrl\_alu <= 4'b0000; Ctrl\_regDst <= 1'b0;

end

Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

SLTI:begin

Ctrl\_alu <= 4'b0100; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end SLTIU:begin

Ctrl\_alu <= 4'b1000; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_branch <= 1'b0;

end ANDI:begin

Ctrl\_alu <= 4'b0101; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end ORI:begin

Ctrl\_alu <= 4'b0110; Ctrl\_regDst <= 1'b0;

end

Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

XORI:begin

Ctrl\_alu <= 4'b0111; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end LUI:begin

Ctrl\_alu <= 4'b0010; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b01; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end LW:begin

Ctrl\_alu <= 4'b0000; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b1; Ctrl\_regWr <= 1'b1; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

end SW:begin

Ctrl\_alu <= 4'b0000; Ctrl\_regDst <= 1'b0;

end

Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b01; Ctrl\_Mem2Reg <= 1'b0; Ctrl\_regWr <= 1'b0; Ctrl\_MemWr <= 1'b1; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b0;

BEQ:begin

Ctrl\_alu <= 4'b0001; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b00; Ctrl\_Mem2Reg <= 1'b0;*//xx* Ctrl\_regWr <= 1'b0; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b01; Ctrl\_jump <= 1'b0;

end BNE:begin

Ctrl\_alu <= 4'b0001; Ctrl\_regDst <= 1'b0; Ctrl\_aluSrcA <= 2'b00; Ctrl\_aluSrcB <= 2'b00; Ctrl\_Mem2Reg <= 1'b0;*//xx* Ctrl\_regWr <= 1'b0; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b1; Ctrl\_branch <= 2'b10; Ctrl\_jump <= 1'b0;

end J:begin

Ctrl\_alu <= 4'b0000;*//xxxxx* Ctrl\_regDst <= 2'b00;*//xx* Ctrl\_aluSrcA <= 2'b00;*//xx* Ctrl\_aluSrcB <= 2'b00;*//xx* Ctrl\_Mem2Reg <= 2'b00;*//xx* Ctrl\_regWr <= 1'b0; Ctrl\_MemWr <= 1'b0; Ctrl\_ext <= 1'b0;*//x* Ctrl\_branch <= 2'b00; Ctrl\_jump <= 1'b1;

end

end endcase

endmodule *// Ctrl*

**2) 原理说明（此处跳转回** [**2.4.7**](#_bookmark40)**）**

#### 控制单元接受指令的前六位(op)和后六位(funct)，如果 op 为 0 则依据

#### funct 确定指令功能和控制信号，反之则由 op 决定。 此单元代码简单但是繁复，而且不能出错。

### 2.4.8. 转发单元 C/D（ForwardC, ForwardD）

#### 具体实现方法查看 [2.4.1](#_bookmark34)。

### 2.4.9. 判断相等单元（Equal）

**1) 实现代码**

module Equal ( input[31:0] input1, input[31:0] input2, output reg equal

);

initial begin equal <= 0;

end

always @(\*) begin

equal <= (input1 == input2);

end endmodule

**2) 原理说明**

#### 主要配合转发单元 C/D 进行使用。由于需要在 ID 级预测分支，因此需要多 出来这两个转发单元和这个 Equal 以判断是否符合分支要求。

### 2.4.10. 扩展单元（Ext）

**1) 实现代码**

module Ext(

input[15:0] input0, input op,

output reg[31:0] out

);

always@(\*)begin case (op)

1'b0: out = { {16{1'b0}} , input0};

1'b1: out = { {16{input0[15]}} , input0}; endcase

end

endmodule *// Ext*

**2) 原理说明**

#### 有可能你会问 lui 的处理去哪里了？实际上 lui 有几种处理方式：

#### a) 在这里添加一种移动到高位的位扩展，Ctrl 修改 lui 的 aluop 修改成加 法或者其他不影响结果的方式。

#### b) 这里作零扩展或者符号扩展都可以，在 Ctrl 中将 lui 实现为位运算，在

#### ALUSrcA 中新增一个 16 作为固定左移的第二输入。

#### c) 直接在 ALU 中新增一种 lui 的方式，不过我认为这并不值得推荐，因为 专门为了一条指令新增一种 ALU 运算不是很经济的行为。

#### 很显然在实现中我选择了 b)。在我实现的其他指令中（也包括整个 MIPS 指 令集的大部分），只需要零扩展或者符号扩展就可以应付。

### 2.4.11. ID/EX 流水线寄存器（ID\_EX）

**1) 代码实现（此处可能较长，点击跳转到** [**2.4.12**](#_bookmark45)**）**

module ID\_EX( input clk, input rst,

input[31:0] regin1, input[31:0] regin2, input[31:0] regin3, input[4:0] regin4, input[4:0] regin5, input[4:0] regin6, input[4:0] regin7, input regdstin, input[3:0] aluopin, input[1:0] alusrcain, input[1:0] alusrcbin, input mem2regin, input regwrin,

input memwrin,

output reg[31:0] regout1, output reg[31:0] regout2, output reg[31:0] regout3, output reg[4:0] regout4, output reg[4:0] regout5, output reg[4:0] regout6, output reg[4:0] regout7, output reg regdstout,

output reg[3:0] aluopout,

output reg[1:0] alusrcaout, output reg[1:0] alusrcbout, output reg mem2regout, output reg regwrout,

output reg memwrout

);

initial begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

regout4 <= 0;

regout5 <= 0;

regout6 <= 0;

regout7 <= 0;

regdstout <= 0;

aluopout <= 0;

alusrcaout <= 0;

alusrcbout <= 0;

mem2regout <= 0;

regwrout <= 0;

memwrout <= 0;

end

always @(posedge rst) begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

regout4 <= 0;

regout5 <= 0;

regout6 <= 0;

regout7 <= 0;

regdstout <= 0;

aluopout <= 0;

alusrcaout <= 0;

alusrcbout <= 0;

mem2regout <= 0;

regwrout <= 0;

memwrout <= 0;

end

always @(posedge clk) begin regout1 <= regin1; regout2 <= regin2; regout3 <= regin3; regout4 <= regin4; regout5 <= regin5; regout6 <= regin6; regout7 <= regin7; regdstout <= regdstin;

aluopout <= aluopin; alusrcaout <= alusrcain; alusrcbout <= alusrcbin; mem2regout <= mem2regin; regwrout <= regwrin; memwrout <= memwrin;

end

endmodule *// ID\_EX\_Reg*

**2) 原理说明（点击跳回** [**2.4.11**](#_bookmark47)**）**

#### 在第一版本的实现中，我新增了一个寄存器变量 TMP\_reg，可以在 git 历史 中查询到。它暂存了所有进入此流水线寄存器的变量一段时间，之后立即输出， 也只有在 clk 为 posedge 的时候才可以写入。

#### 这种方法原理和本代码差不多，而且非常简洁。但是在仿真时我发现每次出 问题都在这里的倒数第二个变量，可以写入但是读出永远为全 x 的变量。除此以 外后续寄存器也是如此，因此无奈只能换这种比较复杂的实现。

### 2.4.12. RegDst 选择器，转发单元 A/B，ALUSrcA/B

#### 具体实现请查看 [2.4.1](#_bookmark34)。

### 2.4.13. 算术逻辑运算单元（ALU）

**1) 实现代码**

module ALU(

input[31:0] input1, input[31:0] input2, input[3:0] aluop, output reg[31:0] out

);

always@(aluop or input1 or input2) begin case (aluop)

4'b0000: out = input1 + input2; 4'b0001: out = input1 - input2; 4'b0010: out = input2 << input1[4:0]; 4'b0011: out = input2 >> input1[4:0];

4'b0100: out = ($signed(input1) < $signed(input2)) ? 1 : 0; 4'b0101: out = input1 & input2;

4'b0110: out = input1 | input2; 4'b0111: out = input1 ^ input2;

4'b1000: out = (input1 < input2) ? 1 : 0; 4'b1001: out = $signed(input2) >>> input1[4:0];

endcase

end

endmodule *// ALU*

**2) 原理说明**

#### ALU 运算具体内容在代码中已经直观的显示出来。 这里针对是否会被符号位影响的运算进行了分别处理，比如右移有逻辑右移

#### 和算术右移两种，比较也有最高位是否为符号位的区别。但是加法和减法是不需 要区分的，因为两种运算不论是否带符号都不会影响运算方式，只会影响数据的 解读方式。

#### 可能有人会问 lui 还是没有出现，建议在 MIPS 部分进行搜索，ALUSrcA 中 有一个 16，这是专门为了 lui 而放置的，配合 ALU 的源码应该能理解。

### 2.4.14. EX/ME 流水线寄存器（EX\_ME）

**1) 实现代码**

module EX\_ME( input clk, input rst,

input[31:0] regin1, input[31:0] regin2, input[4:0] regin3, input mem2regin, input memwrin, input regwrin,

output reg[31:0] regout1, output reg[31:0] regout2, output reg[4:0] regout3, output reg mem2regout, output reg memwrout, output reg regwrout

);

initial begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

mem2regout <= 0;

memwrout <= 0;

regwrout <= 0;

end

always @(posedge rst) begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

mem2regout <= 0;

memwrout <= 0;

regwrout <= 0;

end

always @(posedge clk) begin regout1 <= regin1; regout2 <= regin2; regout3 <= regin3; mem2regout <= mem2regin; memwrout <= memwrin; regwrout <= regwrin;

end

endmodule *// EX\_ME\_Reg*

**2) 原理说明**

#### 这里的代码是从上一个流水线寄存器的模子里复刻出来的，所有内容极其相 似。同样需要注意清零和时钟。

### 2.4.15. 数据存储单元（Mem）

**1) 实现代码**

module Mem(

input clk, input[31:0] import,

input[31:0] write\_data, input memWr,

output reg[31:0] read\_out

);

reg[31:0] data\_memory[1023:0]; always@(\*)begin

read\_out = data\_memory[import[11:2]][31:0];

end

always@(negedge clk)begin if(memWr == 1'b1)

data\_memory[import[11:2]][31:0] <= write\_data[31:0];

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

data\_memory[0], data\_memory[1], data\_memory[2], data\_memory[3], data\_memory[4], data\_memory[5], data\_memory[6], data\_memory[7]);

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

data\_memory[8], data\_memory[9], data\_memory[10], data\_memory[11], data\_memory[12], data\_memory[13], data\_memory[14], data\_memory[15]);

$display("M[16-23]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

data\_memory[16], data\_memory[17], data\_memory[18], data\_memory[19], data\_memory[20], data\_memory[21], data\_memory[22], data\_memory[23]);

$display("M[24-31]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",

data\_memory[24], data\_memory[25], data\_memory[26], data\_memory[27], data\_memory[28], data\_memory[29], data\_memory[30], data\_memory[31]);

$display("Mem\_input =%8X, %8X", import, write\_data);

$display("memWr=%8X", memWr);

end endmodule *// Mem*

**2) 原理说明**

#### 这里和 RegFile 一样，需要进行 negedge 时写入的处理，不然会和[前面](#_bookmark38)一 样发生一样的问题。而且注意这里的内存是字节定址，但是写入和读取都是 4 个 byte，也就是 32bit，所以在 ALU 计算出来的地址中需要抹除后两位才是真正 写入地址的定位。

### 2.4.16. ME/WB 流水线寄存器（ME\_WB）

**1) 代码实现**

module ME\_WB( input clk, input rst,

input[31:0] regin1, input[31:0] regin2, input[4:0] regin3, input mem2regin, input regwrin,

output reg[31:0] regout1, output reg[31:0] regout2, output reg[4:0] regout3, output reg mem2regout, output reg regwrout

);

initial begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

mem2regout <= 0;

regwrout <= 0;

end

always @(posedge rst) begin regout1 <= 0;

regout2 <= 0;

regout3 <= 0;

mem2regout <= 0;

regwrout <= 0;

end

always @(posedge clk) begin

regout1 <= regin1; regout2 <= regin2;

regout3 <= regin3; mem2regout <= mem2regin; regwrout <= regwrin;

end

endmodule *// ME\_WB\_Reg*

**2) 原理说明**

#### 这里的代码是从上一个流水线寄存器的模子里复刻出来的，所有内容极其相 似。同样需要注意清零和时钟。

### 2.4.17. 选择单元 Mem2Reg（U\_Mem2Reg）

#### 具体实现请查阅 [2.4.1](#_bookmark34)。

### 2.4.18. 转发信号决定单元（Forward）

**1) 代码实现**

module Forward( input[4:0] ID\_Rs, input[4:0] ID\_Rt, input[4:0] EX\_Rs, input[4:0] EX\_Rt,

input[4:0] EX\_writeimport, input[4:0] ME\_writeimport, input[4:0] WB\_writeimport, input EXRegWr,

input MERegWr,

input WBRegWr, output reg[1:0] ForwardA, output reg[1:0] ForwardB, output reg[1:0] ForwardC, output reg[1:0] ForwardD

);

always @(\*) begin

if (MERegWr && (ME\_writeimport != 0) && (ME\_writeimport == EX\_Rs) )

ForwardA <= 2'b10;

else if(WBRegWr && (WB\_writeimport != 0) && (WB\_writeimport == EX\_Rs) )

ForwardA <= 2'b01; else ForwardA <= 2'b00;

if (MERegWr && (ME\_writeimport != 0) && (ME\_writeimport == EX\_Rt) )

ForwardB <= 2'b10;

else if (WBRegWr && (WB\_writeimport != 0) && (WB\_writeimport == EX\_Rt) )

ForwardB <= 2'b01; else ForwardB <= 2'b00;

if (WBRegWr && (WB\_writeimport != 0) && (WB\_writeimport == ID\_Rs) )

ForwardC <= 2'b11;

else if (MERegWr && (ME\_writeimport != 0) && (ME\_writeimport == ID\_Rs ) )

ForwardC <= 2'b10;

else if(EXRegWr && (EX\_writeimport !=0 ) && (ID\_Rs == EX\_writeimport) )

ForwardC <= 2'b01; else ForwardC <= 2'b00;

if (WBRegWr && (WB\_writeimport != 0) && (WB\_writeimport == ID\_Rt) )

ForwardD <= 2'b11;

else if (MERegWr && (ME\_writeimport != 0) && (ME\_writeimport == ID\_Rt ) )

ForwardD <= 2'b10;

else if(EXRegWr && (EX\_writeimport !=0 ) && (EX\_writeimport == ID\_Rt) )

ForwardD <= 2'b01; else ForwardD <= 2'b00;

end

endmodule

**2) 原理说明**

#### 转发决定单元分别针对 ABCD 四个单元进行设置。负责在寄存器的对应数值 没有更新的时候正确取出需要的值。如果不理解可以转到 ForwardA/B/C/D 的 接口定义配合理解即可。

#### ForwardA 的取值取决于后续使用的寄存器端口与 ID 级 Rs 是否存在冲突。

#### ForwardB 则取决于寄存器端口与 ID 级 Rt 是否存在冲突。

#### ForwardC 的取值取决于读取寄存器端口 1 是否和后续写入端口冲突，

#### Forward 则是读取寄存器端口 2。

### 2.4.19. 模型机（MIPS）

**1) 代码实现（此处代码较长，点击跳转** [**2.5**](#_bookmark53)**）**

module MIPS();

*//IF*

wire[31:0] IF\_instr, IF\_pc;

*//ID*

wire[31:0] ID\_pc, ID\_instr;

wire[31:0] ID\_Ext\_imm\_16, ID\_reg\_out1, ID\_reg\_out2;

*//ID:Ctrl*

wire[3:0] ID\_Ctrl\_alu;

wire[1:0] ID\_Ctrl\_aluSrcA, ID\_Ctrl\_aluSrcB; wire[1:0] ID\_Ctrl\_branch;

wire ID\_Ctrl\_Mem2Reg, ID\_Ctrl\_MemWr, ID\_Ctrl\_ext, ID\_Ctrl\_jump, ID\_Ctrl\_regDst, ID\_Ctrl\_regWr;

wire ID\_hazard, ID\_Condition;

*//EX*

wire[31:0] EX\_Ext\_imm\_16, EX\_alu\_out; wire[31:0] EX\_reg\_out1,EX\_reg\_out2; wire[4:0] EX\_shamt, EX\_Rs, EX\_Rt, EX\_Rd;

*//EX:Ctrl*

wire[3:0] EX\_Ctrl\_alu;

wire[1:0] EX\_Ctrl\_aluSrcA, EX\_Ctrl\_aluSrcB;

wire EX\_Ctrl\_Mem2Reg, EX\_Ctrl\_MemWr, EX\_Ctrl\_regDst, EX\_Ctrl\_regWr;

*//ME*

wire[31:0] ME\_Mem\_out, ME\_alu\_out, ME\_mem\_write\_data; wire[4:0] ME\_reg\_write\_import;

*//ME:Ctrl*

wire ME\_Ctrl\_Mem2Reg, ME\_Ctrl\_MemWr, ME\_Ctrl\_regWr;

*//WB*

wire[31:0] WB\_alu\_out, WB\_mem\_out; wire[4:0] WB\_reg\_write\_import;

*//WB:Ctrl*

wire WB\_Ctrl\_Mem2Reg, WB\_Ctrl\_regWr;

*//*

wire[1:0] ForwardA, ForwardB, ForwardC, ForwardD;

*//clk & rst* reg clk, rst; initial begin

$readmemh("others/Instr\_MIPS\_Pipeline.txt", U\_Instr\_Mem.Instrs); clk = 1;

rst = 0;

#5 rst = 1;

#20 rst = 0;

end

always #50 clk = ~clk;

Next\_PC U\_Next\_PC(

.clk(clk),

.rst(rst),

.hazard(ID\_hazard),

*//jump*

.jump(ID\_Ctrl\_jump),

.jump2where({ID\_pc[31:28], ID\_instr[25:0], 2'b00}),

*//branch*

.branch((ID\_Ctrl\_branch == 2'b01 && ID\_Condition == 1'b1)

|| (ID\_Ctrl\_branch == 2'b10 && ID\_Condition == 1'b0)),*//beq || bne*

.branch2where(ID\_pc + {{14{ID\_instr[15]}}, ID\_instr[15:0], 2'b00}),

.pc(IF\_pc)

);

Instr\_Mem U\_Instr\_Mem(

.pc(IF\_pc[11:2]),

.instr(IF\_instr)

);

IF\_ID U\_IFID(

.clk(clk),

.rst(rst),

.hazard(ID\_hazard),

.regin1(IF\_pc + 3'b100),

.regin2(IF\_instr),

.regout1(ID\_pc),

.regout2(ID\_instr)

);

RegFile U\_RegFile(

.clk(clk),

.Writedata(U\_Mem2Reg.out),

.regWr(WB\_Ctrl\_regWr),

.writeimport(WB\_reg\_write\_import),

.readimport1(ID\_instr[25:21]),

.readimport2(ID\_instr[20:16]),

.regfile\_out1(ID\_reg\_out1),

.regfile\_out2(ID\_reg\_out2)

);

Hazard U\_Hazard(

.jump(ID\_Ctrl\_jump),

.branch(U\_Next\_PC.branch),

.mem2reg(ID\_Ctrl\_Mem2Reg),

.IF\_Rs(IF\_instr[25:21]),

.IF\_Rt(IF\_instr[20:16]),

.ID\_Rt(ID\_instr[20:16]),

.hazard(ID\_hazard)

);

Ctrl U\_Ctrl(

.op(ID\_instr[31:26]),

.funct(ID\_instr[5:0]),

*//output*

.Ctrl\_alu(ID\_Ctrl\_alu),

.Ctrl\_regDst(ID\_Ctrl\_regDst),

.Ctrl\_aluSrcA(ID\_Ctrl\_aluSrcA),

.Ctrl\_aluSrcB(ID\_Ctrl\_aluSrcB),

.Ctrl\_Mem2Reg(ID\_Ctrl\_Mem2Reg),

.Ctrl\_ext(ID\_Ctrl\_ext),

.Ctrl\_regWr(ID\_Ctrl\_regWr),

.Ctrl\_MemWr(ID\_Ctrl\_MemWr),

.Ctrl\_branch(ID\_Ctrl\_branch),

.Ctrl\_jump(ID\_Ctrl\_jump)

);

MUX #(32) U\_ForwardC(

.a(ID\_reg\_out1),

.b(EX\_alu\_out),

.c(ME\_alu\_out),

.d(U\_Mem2Reg.out),

.Ctrl(ForwardC)

);

MUX #(32) U\_ForwardD(

.a(ID\_reg\_out2),

.b(EX\_alu\_out),

.c(ME\_alu\_out),

.d(U\_Mem2Reg.out),

.Ctrl(ForwardD)

);

Equal U\_Equal(

.input1(U\_ForwardC.out),

.input2(U\_ForwardD.out),

.equal(ID\_Condition)

);

Ext U\_Ext(

.input0(ID\_instr[15:0]),

.op(ID\_Ctrl\_ext),

.out(ID\_Ext\_imm\_16)

);

ID\_EX U\_IDEX(

.clk(clk),

.rst(rst),

.regin1(ID\_reg\_out1),

.regin2(ID\_reg\_out2),

.regin3(ID\_Ext\_imm\_16),

.regin4(ID\_instr[25:21]),

.regin5(ID\_instr[20:16]),

.regin6(ID\_instr[15:11]),

.regin7(ID\_instr[10:6]),

.regdstin(ID\_Ctrl\_regDst),

.aluopin(ID\_Ctrl\_alu),

.alusrcain(ID\_Ctrl\_aluSrcA),

.alusrcbin(ID\_Ctrl\_aluSrcB),

.mem2regin(ID\_Ctrl\_Mem2Reg),

.regwrin(ID\_Ctrl\_regWr),

.memwrin(ID\_Ctrl\_MemWr),

.regout1(EX\_reg\_out1),

.regout2(EX\_reg\_out2),

.regout3(EX\_Ext\_imm\_16),

.regout4(EX\_Rs),

.regout5(EX\_Rt),

.regout6(EX\_Rd),

.regout7(EX\_shamt),

.regdstout(EX\_Ctrl\_regDst),

.aluopout(EX\_Ctrl\_alu),

.alusrcaout(EX\_Ctrl\_aluSrcA),

.alusrcbout(EX\_Ctrl\_aluSrcB),

.mem2regout(EX\_Ctrl\_Mem2Reg),

.regwrout(EX\_Ctrl\_regWr),

.memwrout(EX\_Ctrl\_MemWr)

);

MUX #(5) U\_RegDst(

.a(EX\_Rt),

.b(EX\_Rd),

.Ctrl({1'b0,EX\_Ctrl\_regDst})

);

MUX #(32) U\_ForwardA(

.a(EX\_reg\_out1),

.b(U\_Mem2Reg.out),

.c(ME\_alu\_out),

.Ctrl(ForwardA)

);

MUX #(32) U\_ALUSrcA(

.a(U\_ForwardA.out),

.b(32'h00000010),*//lui*

.c({{27{1'b0}}, EX\_shamt}),

.Ctrl(EX\_Ctrl\_aluSrcA)

);

MUX #(32) U\_ForwardB(

.a(EX\_reg\_out2),

.b(U\_Mem2Reg.out),

.c(ME\_alu\_out),

.Ctrl(ForwardB)

);

MUX #(32) U\_ALUSrcB(

.a(U\_ForwardB.out),

.b(EX\_Ext\_imm\_16),

.Ctrl(EX\_Ctrl\_aluSrcB)

);

ALU U\_ALU(

.input1(U\_ALUSrcA.out),

.input2(U\_ALUSrcB.out),

.aluop(EX\_Ctrl\_alu),

.out(EX\_alu\_out)

);

EX\_ME U\_EXME(

.clk(clk),

.rst(rst),

.regin1(EX\_alu\_out),

.regin2(U\_ForwardB.out),

.regin3(U\_RegDst.out),

.mem2regin(EX\_Ctrl\_Mem2Reg),

.memwrin(EX\_Ctrl\_MemWr),

.regwrin(EX\_Ctrl\_regWr),

.regout1(ME\_alu\_out),

.regout2(ME\_mem\_write\_data),

.regout3(ME\_reg\_write\_import),

.mem2regout(ME\_Ctrl\_Mem2Reg),

.memwrout(ME\_Ctrl\_MemWr),

.regwrout(ME\_Ctrl\_regWr)

);

Mem U\_Mem(

.clk(clk),

.import(ME\_alu\_out),

.write\_data(ME\_mem\_write\_data),

.memWr(ME\_Ctrl\_MemWr),

.read\_out(ME\_Mem\_out)

);

ME\_WB U\_MEWB(

.clk(clk),

.rst(rst),

.regin1(ME\_alu\_out),

.regin2(ME\_Mem\_out),

.regin3(ME\_reg\_write\_import),

.mem2regin(ME\_Ctrl\_Mem2Reg),

.regwrin(ME\_Ctrl\_regWr),

.regout1(WB\_alu\_out),

.regout2(WB\_mem\_out),

.regout3(WB\_reg\_write\_import),

.mem2regout(WB\_Ctrl\_Mem2Reg),

.regwrout(WB\_Ctrl\_regWr)

);

MUX #(32) U\_Mem2Reg(

.a(WB\_alu\_out),

.b(WB\_mem\_out),

.Ctrl({1'b0,WB\_Ctrl\_Mem2Reg})

);

Forward U\_Forward(

.ID\_Rs(ID\_instr[25:21]),

.ID\_Rt(ID\_instr[20:16]),

.EX\_Rs(EX\_Rs),

.EX\_Rt(EX\_Rt),

.EX\_writeimport(U\_RegDst.out),

.ME\_writeimport(ME\_reg\_write\_import),

.WB\_writeimport(WB\_reg\_write\_import),

.EXRegWr(EX\_Ctrl\_regWr),

.MERegWr(ME\_Ctrl\_regWr),

.WBRegWr(WB\_Ctrl\_regWr),

.ForwardA(ForwardA),

.ForwardB(ForwardB),

.ForwardC(ForwardC),

.ForwardD(ForwardD)

);

endmodule *//*

**2) 原理说明（此处**[**跳回**](#_bookmark52)**）**

#### MIPS 模型机负责的是模块之间的相互接线，除此之外基本没有运算，结合

#### [CPU 总体结构](#_bookmark11)可以更快地理解源码的结构。

## 2.5. 设计结果分析

### 2.5.1. 测试文件

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| L0: | lui | $1, | 0x1000 |  | | | |
|  | ori | $1, | $1, 0x2211 | # | $1 | = | 0x10002211 |
|  | lui | $2, | 0x1000 |  |  |  |  |
|  | ori | $2, | $2, 0x4433 | # | $2 | = | 0x10004433 |
|  | add | $3, | $2, $1 | # | $3 | = | 0x20006644 |
| sw $3, 0($0) # 0($0) = 0x20006644  lui $4, 0x3000  ori $4, $4, 0x5566 # $4 = 0x30005566  sub $5, $3, $4 # $5 = 0xf00010de  sw $5, 4($0) # 4($0) = 0xf00010de | | | | | | | |

L1:

L2:

L3:

lw $6, 0($0) # $s6 = 0x20006644

slti $7,$6,0x7fff # $7 = 0 $6>$5 beq $7, $0, L2 # goto L2

j L3

slt $8, $2, $3 # $2 = 0x10004433 < $3 = 0x20006644, $8 = 1

bne $8, $0, L1 # $8 = 1, goto L1 ori $9, $0, 0x10 # $9 = 0x00000010

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sll | $10, | $9, 5 | # | $10 | = | 0x00000200 |
| srl | $11, | $10, 4 | # | $11 | = | 0x00000020 |
| ori | $13, | $0, 0x1234 | # | $13 | = | 0x1234 |
| ori | $14, | $0, 0xff | # | $13 | = | 0xff |
| and | $15, | $13, $14 | # | $15 | = | 0x34 |
| lui | $16, | 0xffff | # | $17 | = | 0xffff0000 |
| or | $17, | $13, $16 | # | $17 | = | 0xffff1234 |

addu $18, $2, $1 # $18 = 10004433 + 10002211 = 0x20006644 subu $19, $3, $4 # $19 = 20006644 – 30005566 = 0xf00010de lw $20, 0($0) # $20 = 0x20006644

sw $20, 8($0) # 8($0) = 0x20006644 addi $21,$20, 0x1000 # $21 = 0x20007644 addi $22,$20, -0x1000 # $22 = 0x20005644

j L0

### 2.5.2. 测试机器码

3c011000

34212211

3c021000

34424433

00411820 ac030000 3c043000

34845566

00642822 ac050004 8c060000

28c77fff 10e00001

08000c10

0043402a

1500fffd 34090010

00095140

000a5902

340d1234

340e00ff 01ae7824

3c10ffff 01b08825

00419021

00649823

8c140000 ac140008 22951000

2296f000

08000c00

### 2.5.3. 测试结果分析

#### 测试结果分析主要由指令构成，因为需要分析的指令太多，所以并没有选择 所有的指令，而选择了具有难度的几块指令进行比较详细的解释，这样其他的指 令也不会有很多的问题，甚至可以说相当简单。

#### 此处更多会考虑数据转发和控制冒险，分别关注于 ALU 是否获得了正确的数 和是否进行了正确的 PC 修改。

**2.5.3.1. 简单的数据转发**

#### 指令如下：

L0:

lui $1, 0x1000

ori $1, $1, 0x2211

# $1 = 0x10002211

#### 对应的 modelsim 仿真图如下：

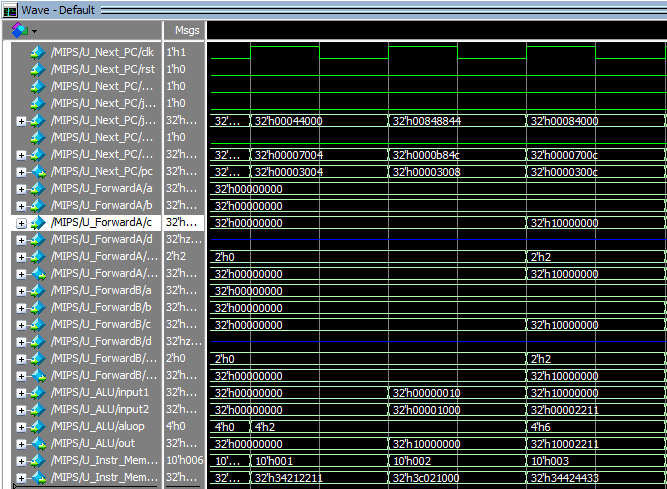


图 2-5-1 简单的数据转发的仿真图

#### 当 ALU 处在 ori $1, $1, 0x2211 指令时，lui 处理结果已经抵达 ME 级， 因此需要从数据转发单元直接从此处转发出数据。从仿真图可以看出，ALU 此处 输出正确，通过转发单元，input1 得到了正确的输入，因而进行了正确的运算。

#### 此处并没有对于其他的内容进行太多的解释，因为从 [CPU 总体结构](#_bookmark11)可以看出， 大部分数据流都是顺着IF → ID → EX → ME → WB进行流动的，因此只需要观察 需要回头的数据流动即可。

**2.5.3.2. 复杂的数据转发-negedge 的重要性**

#### 指令如下：

ori $1, $1, 0x2211

lui $2, 0x1000 ori $2, $2, 0x4433 add $3, $2, $1

# $1 = 0x10002211

# $2 = 0x10004433

# $3 = 0x20006644

#### 对应的 modelsim 仿真图如下：

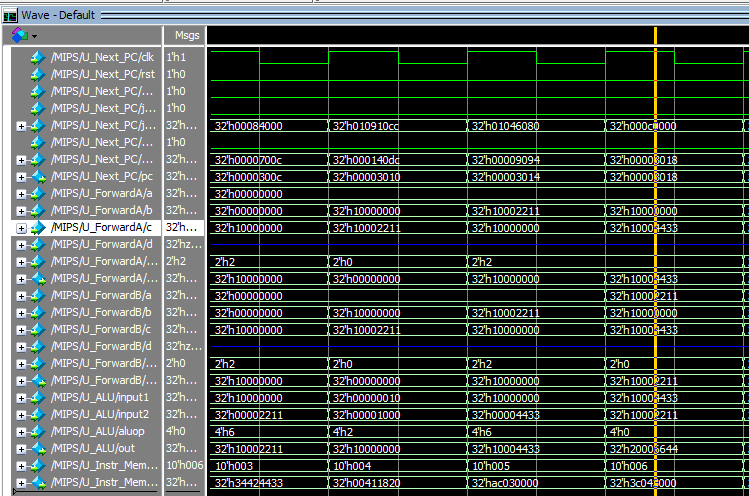


图 2-5-2 复杂的数据转发的仿真图

#### 这里非常容易出现问题，这里也会对于[寄存器组（RegFile）](#_bookmark15)里面提出的进行 详细的解释。也是 negedge 出现的最核心原因。

#### 理论上需要回头的数据是$1 和$2，在第一条 lui 指令只需要进行正常的处 理；在 ori 指令中需要将$2 从 ME 级转发回 EX 级；在 add 中需要将$2 从 ME 级 转发回 EX，并且将$1 从寄存器中重新读取。虽然$2 在 WB 级中也有数据，但是 这不是最新的，因此不需要过多考虑，而且这也不是核心问题。

#### 核心问题在于，$1 此时刚好从 WB 出来写入寄存器，如果写入信号是和 ID/EX 一样的 posedge，那么会导致 ID/EX 获得的寄存器值不是最新（时序电路的特 点），从而导致计算错误。因此写入信号必须在 ID/EX 获得数据之前写入，而一 个巧妙地的方法是 negedge 时写入，正好卡在 ME/WB 的 posedge 和 ID/EX 的 posedge 之间。

#### 从仿真结果可以看出，ALU 在最后的 add 指令中获得了正确的$1 值，这个值 从 ForwardB 的 0 号选择端口输出，也是我们期望的数据流动。

**2.5.3.3. 指令 lw 的冒险**

#### 指令如下：

sw $5, 4($0)

lw $6, 0($0)

# 4($0) = 0xf00010de

# $s6 = 0x20006644

slti $7,$6,0x7fff # $7 = 0 $6>$5

#### 对应的仿真图如下：

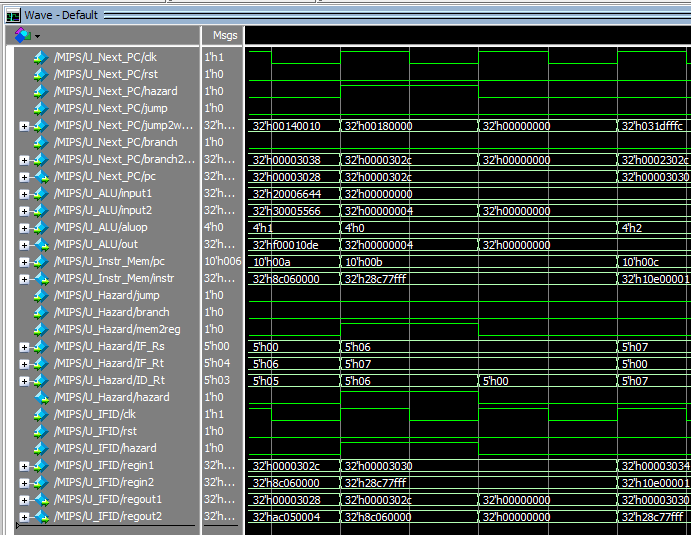


图 2-5-3 lw 冒险的仿真图

#### 注意到这里的仿真和之前的输出信息不一样，因为这里不主要关注获得的数 据是否正确，而更需要关注 PC 的跳转是否有问题。

#### 理论上这里需要进行一次冒险，让 PC 暂停+4 一次。在 ALU 进行 sw 运算时， ID 级正在处理 lw 指令，由于除了寄存器写入都不需要 clk，因此 hazard 立即 变成高电平；经过一个 clk 之后 ID 的内容清空，EX 继续处理 lw 指令，还需要 ME 和 WB 才能从寄存器中获得正确的值。

#### 实际上在上一部分指令中我们将写入信号修改成了 negedge，因此在 slti 在刚进入 ID 级时，此时刚经过 posedge，读取了错误的值，但是在下一个 posedge 之前寄存器已经写入了新的内容，因此可以将正确的值送到 EX。

#### 从仿真图来看，lw 产生了正确的冒险，这是因为 ID\_Rt==IF\_Rs，并且因为 mem2reg==1。在产生 hazard 之后一个 clk，寄存器确实清空了，此时 EX 级正 在处理 lw。之后也确实如同分析一样获得了正确的值，程序给出了正确的结果。

**2.5.3.4. 指令 j/beq/bne 的冒险-连续跳转**

#### 指令如下：

slti $7,$6,0x7fff

beq $7, $0, L2

# $7 = 0 $6>$5

# goto L2

L1:

j L3

L2:

slt $8, $2, $3

bne $8, $0, L1

# $2 = 0x10004433 < $3 = 0x20006644, $8 = 1

# $8 = 1, goto L1

L3:

ori $9, $0, 0x10

# $9 = 0x00000010

#### 对应的仿真图如下（此处图比较多）。

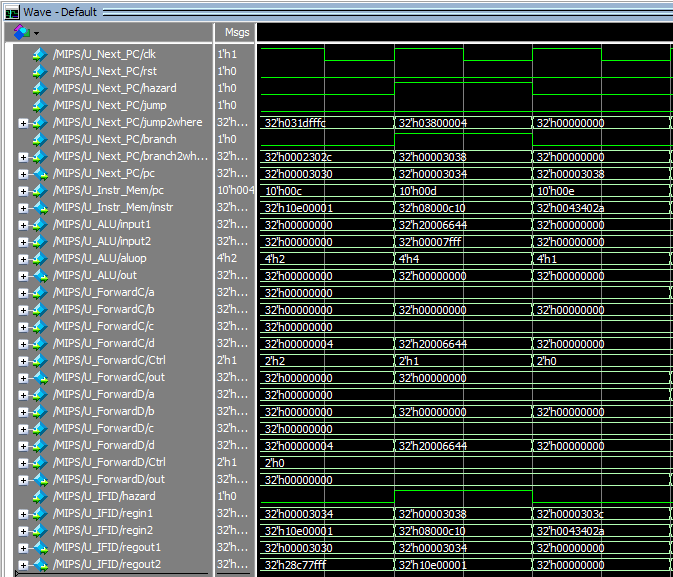


图 2-5-4 指令 j/beq/bne 的冒险的仿真图-1

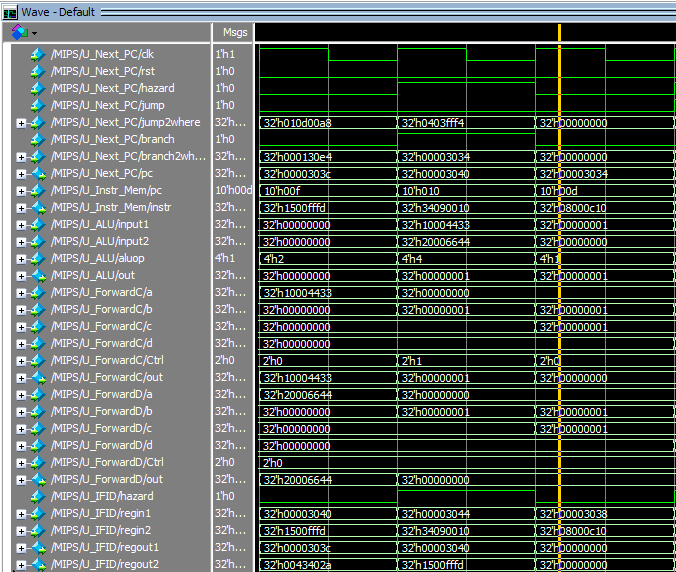


图 2-5-4 指令 j/beq/bne 的冒险的仿真图-2

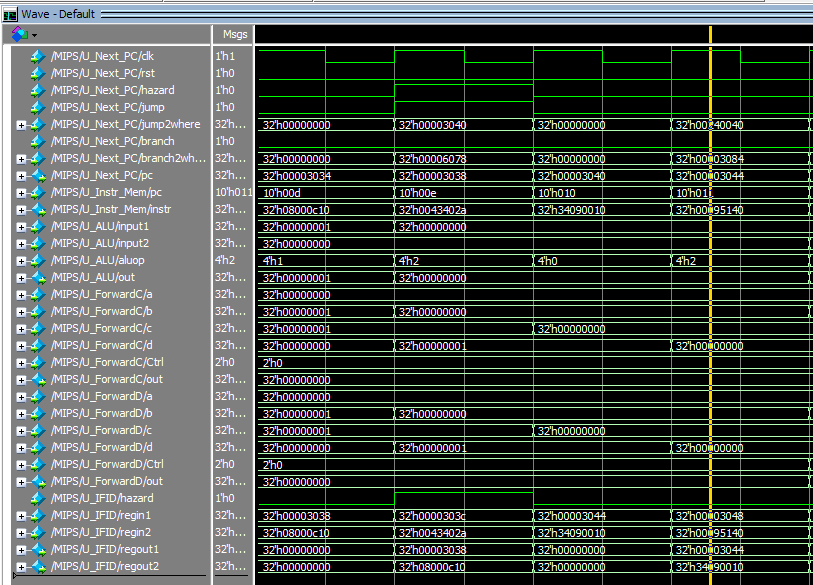


图 2-5-4 指令 j/beq/bne 的冒险的仿真图-3

#### 此处设计程序的连续跳转，也是一个相当难得点，我在调试的时候发现了一 个 IF\_ID 寄存器的错误才得以输出正确的结果，这个错误就是漏了

.regin1(IF\_pc + 3'b100),

#### 的+4。具体原因会沿着指令分析一一解释清楚。

#### 在 ALU 的 slti 运算的同时，beq 需要 EX 级的$7 才能获得正确的结果。此 处转发需要转发到 ForwardC，之后和$0 读取出的 0 进入 Equal 进行比较，结 果使 hazard 为 1 从而产生阻塞。阻塞时发生跳转，指令跳转到 0x3038 处， ID\_IF 清空，之后正常读取。

#### 之后 ALU 进行了一次 slt 运算，同时 ID 发现了 bne 需要跳转，于是进行了 同样的操作。最后等正常恢复读取之后立即读取到 j，在 IF 级什么都不会发生， 但是等到 ID 级就会产生和之前类似的效果。

#### 由于在处理 beq 时，所有的写入开关都没有打开，因此 beq 在抵达 EX，ME， WB 之后虽然进行了很多的运算的，但是实际上什么都没有发生。

#### 以上只是大致的分析应该发生什么，接下来结合仿真图进行更具体的分析。 从第一张仿真图(beq)可以看出，确实和大致分析的一样，但是有个细节确实 值得注意：PC 从 0x3030 跳转到 0x3034 之后才会跳转到 0x3038。实际上，在 从 0x3030 变成 0x3034 的时候，hazard=1，这意味着清空指令还没有生效，并 且依照 PC 需要 clk 的特点，在需要分支/跳转/阻塞发生的时候，需要延后一个

#### clk 才能将变化实施，这才有了似乎没有跳转的假象。后续将会证明这个假设是 正确的。

#### 在阻塞发生之后(hazard=1 之后的一个 clk)，ID 指令清空一次，此时 IF 已 经读取到新的指令 slt，在下个周期进入 ID，这确实做到了阻塞一次。在 slt 进入 EX 级 的 时 候 也 发 生 了 类 似 的 跳 转 ， 观 察 PC 的 变 化 ： 0x303c->0x3040->0x3034，对照 mars 可以很明显的发现前面的解释是正确的，

#### 0x3034 对应了 j 指令。指令 j 发生的时候也是类似，0x3034->0x3038->0x3040， 这也是我们期望发生的。

#### 整个过程中，产生了$7, $8 两个寄存器需要写入，实际上由于我们将跳转放 在 ID 级，之后的运算完全不受影响，该写入就写入。

## 2.6. FPGA 开发板测试

### 2.6.1. FPGA 开发板介绍及照片

#### Artix®-7 器件在单个成本优化的 FPGA 中提供了最高性能功耗比结构、 收 发器线速、 DSP 处理能力以及 AMS 集成。 包含 MicroBlaze™软处理器和 1,066Mb/s DDR3 技术支持，此系列为各类成本功耗敏感型应用提供最大价值， 包括软件定义无线电、机器视觉照相以及低端无线回传。



图 2-6-1 artix-7 FPGA 的图片介绍(xc7a100t)

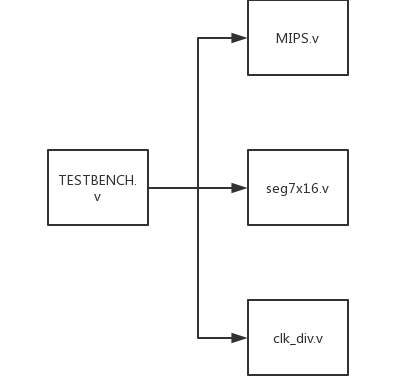
### 2.6.2. 测试显示方案

#### 由于是单周期的实验，所以我们只需要证明 PC 流动完全没有问题，之后证 明最后结果正确即可。

#### 为了方便计算，我将输出的 PC 更改为第几条指令，也为了方便证明结果正 确，最后的输出数据是按照内存单元 0-4(words)循环滚动，之后的 asm 文件也 显示了这一点：0-4(words)是存放数据的内存地址号。

### 2.6.3. 连接开关、灯泡和七段显示管

### . 连接框架(Verilog 文件连接方式)



### . 文件 TESTBENCH.v

`include "seg7x16.v"

`include "clk\_div.v"

`include "MIPS.v" module TESTBENCH( input clk,

input rst, input[15:0] sw\_i, output[15:0] led\_o, output[7:0] o\_seg, output[7:0] o\_sel

);

wire clk\_display = clk; wire clk\_CPU;

wire rst\_CPU = ~rst; reg[31:0] clk\_cnt; wire[31:0] display\_data;

initial begin clk\_cnt = 0;

end

always @(posedge rst\_CPU) begin

clk\_cnt = 0;

end

always @(posedge clk\_CPU) begin clk\_cnt = clk\_cnt + 1; if(clk\_cnt >= 3072)begin

$finish;

end

end

MIPS U\_MIPS(

.clk(clk\_CPU),

.rst(rst\_CPU),

.Ctrl\_out(led\_o),

.display\_data(display\_data)

);

clk\_div U\_clk\_div(

.clk\_dis(clk\_display),

.rst(rst\_CPU),

.sw(sw\_i),

.clk\_CPU(clk\_CPU)

);

seg7x16 U\_seg7x16(

.clk(clk\_display),

.rst(rst\_CPU),

.i\_data(display\_data),

.o\_seg(o\_seg),

.o\_sel(o\_sel)

);

endmodule *// TESTBENCH*

### . 文件 clk\_div.v

*//`timescale 1ns / 1ps*

module clk\_div ( input clk\_dis, input rst, input[15:0] sw, output clk\_CPU

);

reg[31:0] clkdiv;

always @ (posedge clk\_dis or posedge rst) begin

clkdiv <= (rst) ? 0 : clkdiv + 1'b1;

*/\* if (rst)*

*clkdiv <= 0; else*

*clkdiv <= clkdiv + 1'b1;\*/*

end

*//sw\_15:29->5.3687*

*//sw\_14:28->2.6844*

*//sw\_13:27->1.3422*

*//sw\_12:26->0.6711*

*//sw\_11:25->0.3355*

*//sw\_10:24->0.1678*

assign clk\_CPU=(sw[15]) ? clkdiv[29] :

(sw[14]) ? clkdiv[28] :

(sw[13]) ? clkdiv[27] :

(sw[12]) ? clkdiv[26] :

(sw[11]) ? clkdiv[25] :

(sw[10]) ? clkdiv[24] : clkdiv[19];

endmodule

### . 文件 seg7x16.v

*//`timescale 1ns / 1ps*

module seg7x16( input clk, input rst,

input [31:0] i\_data,

output [7:0] o\_seg,

output [7:0] o\_sel

);

reg [14:0] cnt;

always @ (posedge clk, posedge rst) if (rst)

cnt <= 0; else

cnt <= cnt + 1'b1; wire seg7\_clk = cnt[14]; reg [2:0] seg7\_addr;

always @ (posedge seg7\_clk, posedge rst) if(rst)

seg7\_addr <= 0; else

seg7\_addr <= seg7\_addr + 1'b1; reg [7:0] o\_sel\_r;

always @ (\*)

case(seg7\_addr)

7 : o\_sel\_r = 8'b01111111;

6 : o\_sel\_r = 8'b10111111;

5 : o\_sel\_r = 8'b11011111;

4 : o\_sel\_r = 8'b11101111;

3 : o\_sel\_r = 8'b11110111;

2 : o\_sel\_r = 8'b11111011;

1 : o\_sel\_r = 8'b11111101;

0 : o\_sel\_r = 8'b11111110; endcase

reg [31:0] i\_data\_store;

always @ (posedge clk, posedge rst) if(rst)

i\_data\_store <= 0; else

i\_data\_store <= i\_data;

reg [7:0] seg\_data\_r; always @ (\*)

case(seg7\_addr)

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

1. : seg\_data\_r = i\_data\_store[7:4];
2. : seg\_data\_r = i\_data\_store[11:8];
3. : seg\_data\_r = i\_data\_store[15:12];
4. : seg\_data\_r = i\_data\_store[19:16];
5. : seg\_data\_r = i\_data\_store[23:20];
6. : seg\_data\_r = i\_data\_store[27:24];
7. : seg\_data\_r = i\_data\_store[31:28]; endcase

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

always @ (posedge clk, posedge rst) if(rst)

o\_seg\_r <= 8'hff; else

case(seg\_data\_r)

4'h0 : o\_seg\_r <= 8'hC0; 4'h1 : o\_seg\_r <= 8'hF9; 4'h2 : o\_seg\_r <= 8'hA4; 4'h3 : o\_seg\_r <= 8'hB0; 4'h4 : o\_seg\_r <= 8'h99; 4'h5 : o\_seg\_r <= 8'h92; 4'h6 : o\_seg\_r <= 8'h82; 4'h7 : o\_seg\_r <= 8'hF8; 4'h8 : o\_seg\_r <= 8'h80; 4'h9 : o\_seg\_r <= 8'h90; 4'hA : o\_seg\_r <= 8'h88; 4'hB : o\_seg\_r <= 8'h83;

4'hC : o\_seg\_r <= 8'hC6; 4'hD : o\_seg\_r <= 8'hA1; 4'hE : o\_seg\_r <= 8'h86; 4'hF : o\_seg\_r <= 8'h8E;

endcase

assign o\_sel = o\_sel\_r; assign o\_seg = o\_seg\_r;

endmodule

### . 文件 icf.xdc

# Clock signal

set\_property -dict { PACKAGE\_PIN E3 IOSTANDARD LVCMOS33 } [get\_ports

{ clk }]; #IO\_L12P\_T1\_MRCC\_35 Sch=clk100mhz

create\_clock -add -name sys\_clk\_pin -period 100.00 -waveform {0 50} [get\_ports {clk}];

set\_property -dict { PACKAGE\_PIN C12 IOSTANDARD LVCMOS33 } [get\_ports

{ rst }]; #IO\_L3P\_T0\_DQS\_AD1P\_15 Sch=cpu\_resetn

# 7seg

set\_property -dict { PACKAGE\_PIN T10 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[0] }]; #IO\_L24N\_T3\_A00\_D16\_14 Sch=ca

set\_property -dict { PACKAGE\_PIN R10 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[1] }]; #IO\_25\_14 Sch=cb

set\_property -dict { PACKAGE\_PIN K16 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[2] }]; #IO\_25\_15 Sch=cc

set\_property -dict { PACKAGE\_PIN K13 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[3] }]; #IO\_L7P\_T2\_A26\_15 Sch=cd

set\_property -dict { PACKAGE\_PIN P15 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[4] }]; #IO\_L13P\_T2\_MRCC\_14 Sch=ce

set\_property -dict { PACKAGE\_PIN T11 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[5] }]; #IO\_L19P\_T3\_A10\_D26\_14 Sch=cf

set\_property -dict { PACKAGE\_PIN L18 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[6] }]; #IO\_L4P\_T0\_D04\_14 Sch=cg

set\_property -dict { PACKAGE\_PIN H15 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_seg[7] }]; #IO\_L19N\_T3\_A21\_VREF\_15 Sch=dp

set\_property -dict { PACKAGE\_PIN J17 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[0] }]; #IO\_L23P\_T3\_FOE\_B\_15 Sch=an[0]

set\_property -dict { PACKAGE\_PIN J18 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[1] }]; #IO\_L23N\_T3\_FWE\_B\_15 Sch=an[1]

set\_property -dict { PACKAGE\_PIN T9 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[2] }]; #IO\_L24P\_T3\_A01\_D17\_14 Sch=an[2]

set\_property -dict { PACKAGE\_PIN J14 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[3] }]; #IO\_L19P\_T3\_A22\_15 Sch=an[3]

set\_property -dict { PACKAGE\_PIN P14 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[4] }]; #IO\_L8N\_T1\_D12\_14 Sch=an[4]

set\_property -dict { PACKAGE\_PIN T14 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[5] }]; #IO\_L14P\_T2\_SRCC\_14 Sch=an[5]

set\_property -dict { PACKAGE\_PIN K2 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[6] }]; #IO\_L23P\_T3\_35 Sch=an[6]

set\_property -dict { PACKAGE\_PIN U13 IOSTANDARD LVCMOS33 } [get\_ports

{ o\_sel[7] }]; #IO\_L23N\_T3\_A02\_D18\_14 Sch=an[7]

# Switches

set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[0] }]; #IO\_L24N\_T3\_RS0\_15 Sch=sw[0]

set\_property -dict { PACKAGE\_PIN L16 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[1] }]; #IO\_L3N\_T0\_DQS\_EMCCLK\_14 Sch=sw[1]

set\_property -dict { PACKAGE\_PIN M13 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[2] }]; #IO\_L6N\_T0\_D08\_VREF\_14 Sch=sw[2]

set\_property -dict { PACKAGE\_PIN R15 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[3] }]; #IO\_L13N\_T2\_MRCC\_14 Sch=sw[3]

set\_property -dict { PACKAGE\_PIN R17 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[4] }]; #IO\_L12N\_T1\_MRCC\_14 Sch=sw[4]

set\_property -dict { PACKAGE\_PIN T18 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[5] }]; #IO\_L7N\_T1\_D10\_14 Sch=sw[5]

set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[6] }]; #IO\_L17N\_T2\_A13\_D29\_14 Sch=sw[6]

set\_property -dict { PACKAGE\_PIN R13 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[7] }]; #IO\_L5N\_T0\_D07\_14 Sch=sw[7]

set\_property -dict { PACKAGE\_PIN T8 IOSTANDARD LVCMOS18 } [get\_ports

{ sw\_i[8] }]; #IO\_L24N\_T3\_34 Sch=sw[8]

set\_property -dict { PACKAGE\_PIN U8 IOSTANDARD LVCMOS18 } [get\_ports

{ sw\_i[9] }]; #IO\_25\_34 Sch=sw[9]

set\_property -dict { PACKAGE\_PIN R16 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[10] }]; #IO\_L15P\_T2\_DQS\_RDWR\_B\_14 Sch=sw[10]

set\_property -dict { PACKAGE\_PIN T13 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[11] }]; #IO\_L23P\_T3\_A03\_D19\_14 Sch=sw[11]

set\_property -dict { PACKAGE\_PIN H6 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[12] }]; #IO\_L24P\_T3\_35 Sch=sw[12]

set\_property -dict { PACKAGE\_PIN U12 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[13] }]; #IO\_L20P\_T3\_A08\_D24\_14 Sch=sw[13]

set\_property -dict { PACKAGE\_PIN U11 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[14] }]; #IO\_L19N\_T3\_A09\_D25\_VREF\_14 Sch=sw[14]

set\_property -dict { PACKAGE\_PIN V10 IOSTANDARD LVCMOS33 } [get\_ports

{ sw\_i[15] }]; #IO\_L21P\_T3\_DQS\_14 Sch=sw[15]

# LEDs

set\_property -dict { PACKAGE\_PIN H17 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[0] }]; #IO\_L18P\_T2\_A24\_15 Sch=led[0]

set\_property -dict { PACKAGE\_PIN K15 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[1] }]; #IO\_L24P\_T3\_RS1\_15 Sch=led[1]

set\_property -dict { PACKAGE\_PIN J13 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[2] }]; #IO\_L17N\_T2\_A25\_15 Sch=led[2]

set\_property -dict { PACKAGE\_PIN N14 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[3] }]; #IO\_L8P\_T1\_D11\_14 Sch=led[3]

set\_property -dict { PACKAGE\_PIN R18 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[4] }]; #IO\_L7P\_T1\_D09\_14 Sch=led[4]

set\_property -dict { PACKAGE\_PIN V17 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[5] }]; #IO\_L18N\_T2\_A11\_D27\_14 Sch=led[5]

set\_property -dict { PACKAGE\_PIN U17 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[6] }]; #IO\_L17P\_T2\_A14\_D30\_14 Sch=led[6]

set\_property -dict { PACKAGE\_PIN U16 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[7] }]; #IO\_L18P\_T2\_A12\_D28\_14 Sch=led[7]

set\_property -dict { PACKAGE\_PIN V16 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[8] }]; #IO\_L16N\_T2\_A15\_D31\_14 Sch=led[8]

set\_property -dict { PACKAGE\_PIN T15 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[9] }]; #IO\_L14N\_T2\_SRCC\_14 Sch=led[9]

set\_property -dict { PACKAGE\_PIN U14 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[10] }]; #IO\_L22P\_T3\_A05\_D21\_14 Sch=led[10]

set\_property -dict { PACKAGE\_PIN T16 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[11] }]; #IO\_L15N\_T2\_DQS\_DOUT\_CSO\_B\_14 Sch=led[11]

set\_property -dict { PACKAGE\_PIN V15 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[12] }]; #IO\_L16P\_T2\_CSI\_B\_14 Sch=led[12]

set\_property -dict { PACKAGE\_PIN V14 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[13] }]; #IO\_L22N\_T3\_A04\_D20\_14 Sch=led[13]

set\_property -dict { PACKAGE\_PIN V12 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[14] }]; #IO\_L20N\_T3\_A07\_D23\_14 Sch=led[14]

set\_property -dict { PACKAGE\_PIN V11 IOSTANDARD LVCMOS33 } [get\_ports

{ led\_o[15] }]; #IO\_L21N\_T3\_DQS\_A06\_D22\_14 Sch=led[15]

set\_property CLOCK\_DEDICATED\_ROUTE FALSE [get\_nets rst\_IBUF]

#### 注意最后一行是我自行添加的，因为在编译时 vivado 产生了一个错误，错

#### 误提示添加一行，并且提示了内容为最后一行的内容。建议编译时优先去掉最后 一行编译，针对报错情况单独处理。

#### 如果替换了 TESTBENCH.v 的变量名，那么这里的变量名也会替换，建议

#### Ctrl+H 全局替换。

### . 文件 Bubble\_sort.asm

LOAD: lui $1,0x1000

ori $1,$1,0x1008 sw $1,0($0) xori $1,$1,0xA sw $1,4($0) xori $1,$1,0x7 sw $1,12($0)

lui $1,0x8000 ori $1,$1,0x1001

sw $1,8($0)

xori $1,$1,0x1 sw $1,16($0)

#xor $9,$9,$9 andi $9,$9,0 ori $9,$9,0x14

#xor $10,$10,$10

andi $10,$10,0 ori $10,$10,0x10

# length=max\_I=$9=20

#max\_J=$10=20-i-4

andi $11,$11,0

#xor $11,$11,$11 I\_LOOP: slt $1,$11,$9

beq $1,$0,END andi $12,$12,0

#xor $12,$12,$12

# i=$11

# i<length

# j=$12

J\_LOOP: slt $1,$12,$10

beq $1,$0,END\_I

#for-loop

lw $2,0($12) lw $3,4($12) slt $1,$2,$3 beq $1,$0,SWAP sw $2,0($12) sw $3,4($12)

j END\_J

# if a[i]<a[j]

SWAP: sw $2,4($12)

sw $3,0($12)

END\_J: addi $12,$12,4 j J\_LOOP

END\_I: addi $11,$11,4

addi $10,$10,-4 j I\_LOOP

END:

### . 文件 Test\_6\_Instr.coe

#### 本文件虽然是最初实验的文件名，但是内容是排序代码。如果有问题可以参 照 [2.6.3.6](#_bookmark67) 的 asm 代码进行查验。

memory\_initialization\_radix=16;

memory\_initialization\_vector= 3c011000,

34211008, ac010000, 3821000a, ac010004, 38210007,

ac01000c, 3c018000,

34211001, ac010008, 38210001, ac010010, 31290000,

35290014,

314a0000,

354a0010,

316b0000,

0169082a,

10200011,

318c0000,

018a082a,

1020000b,

8d820000,

8d830004,

0043082a,

10200003, ad820000, ad830004, 08000c1f, ad820004, ad830000, 218c0004,

08000c14,

216b0004,

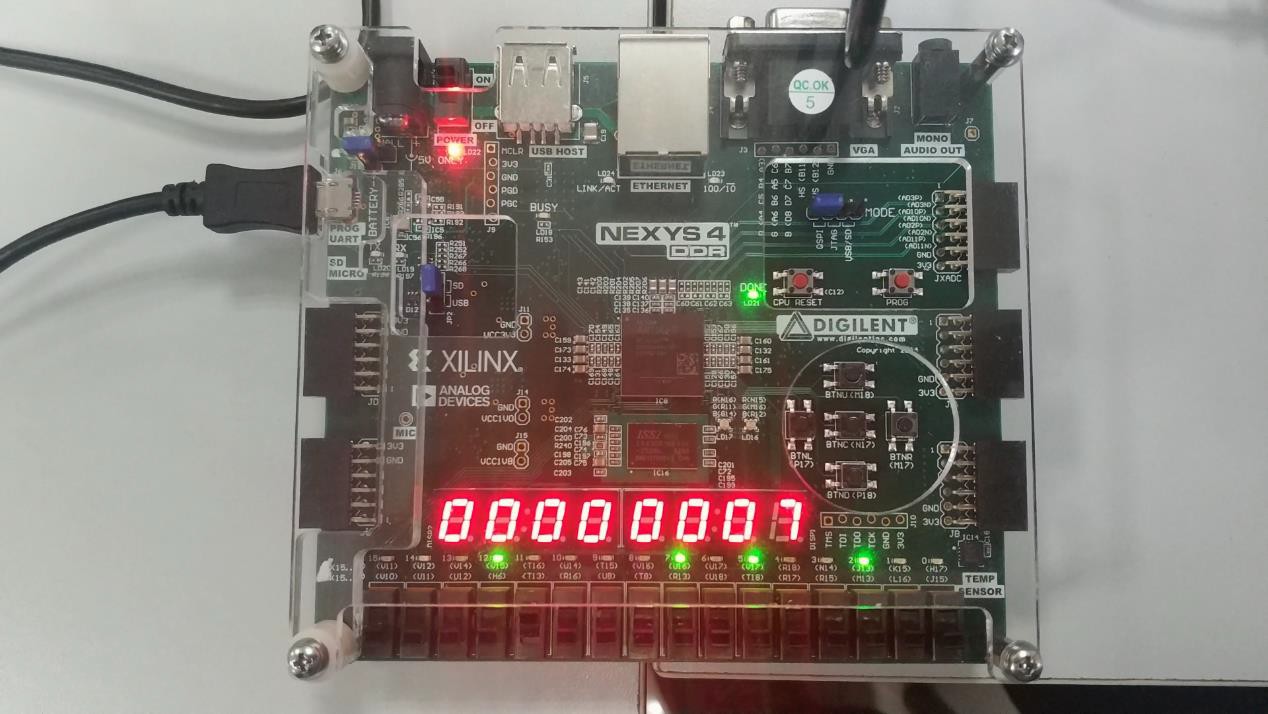
214afffc, 08000c11;

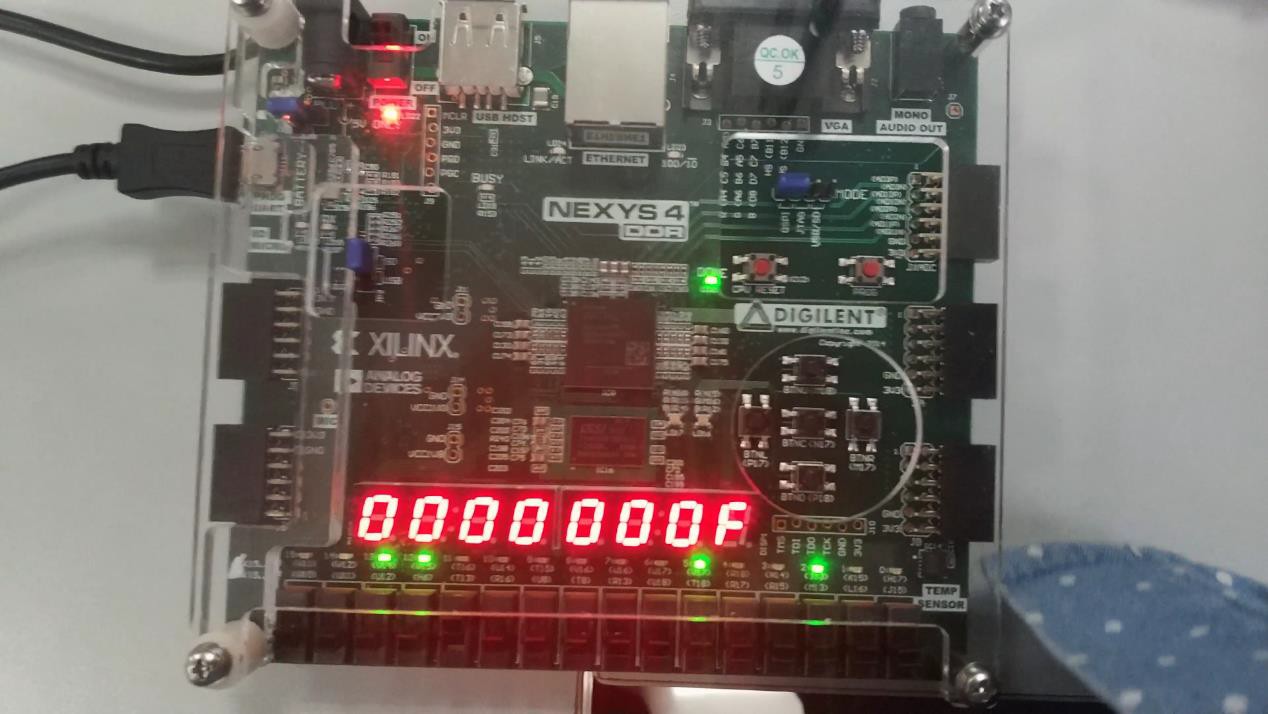
### 2.6.4. 实验成果

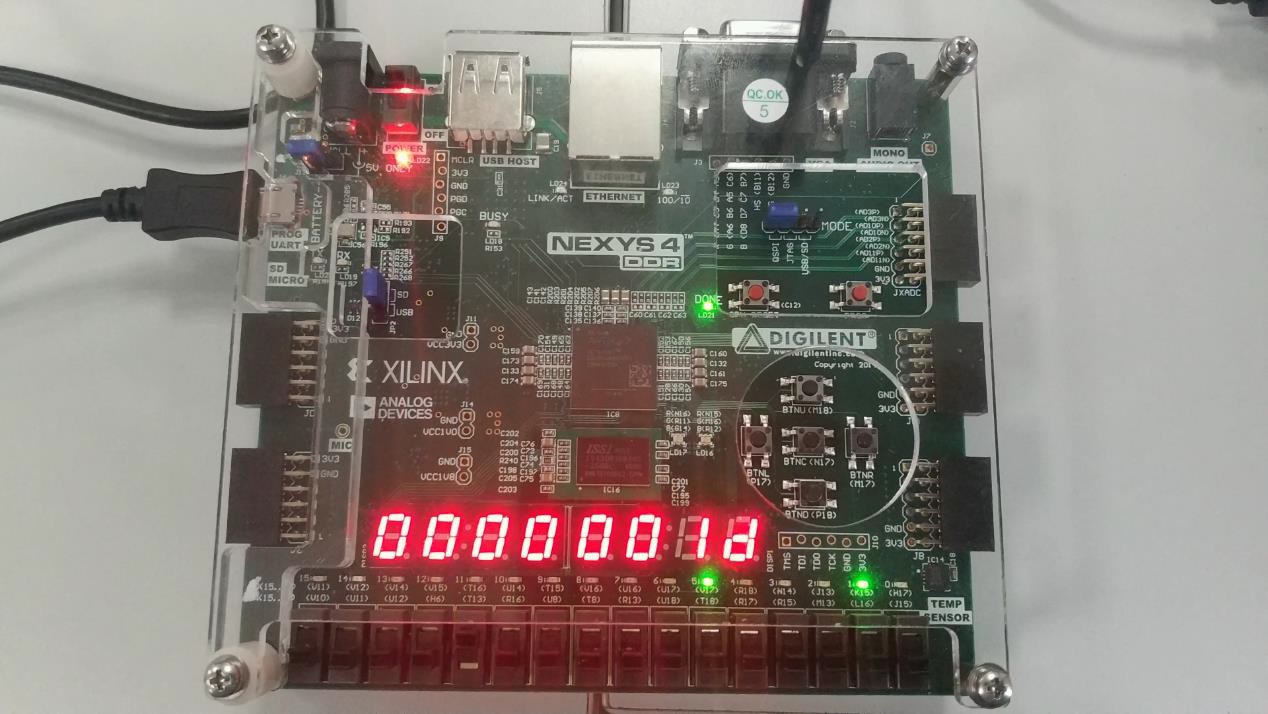
#### 如果替换了 TESTBENCH.v 的变量名，那么这里的变量名也会替换，建议

#### Ctrl+H 全局替换。 实验成果见如下照片，照片仅是从视频中截取下来的，具体视频详见：

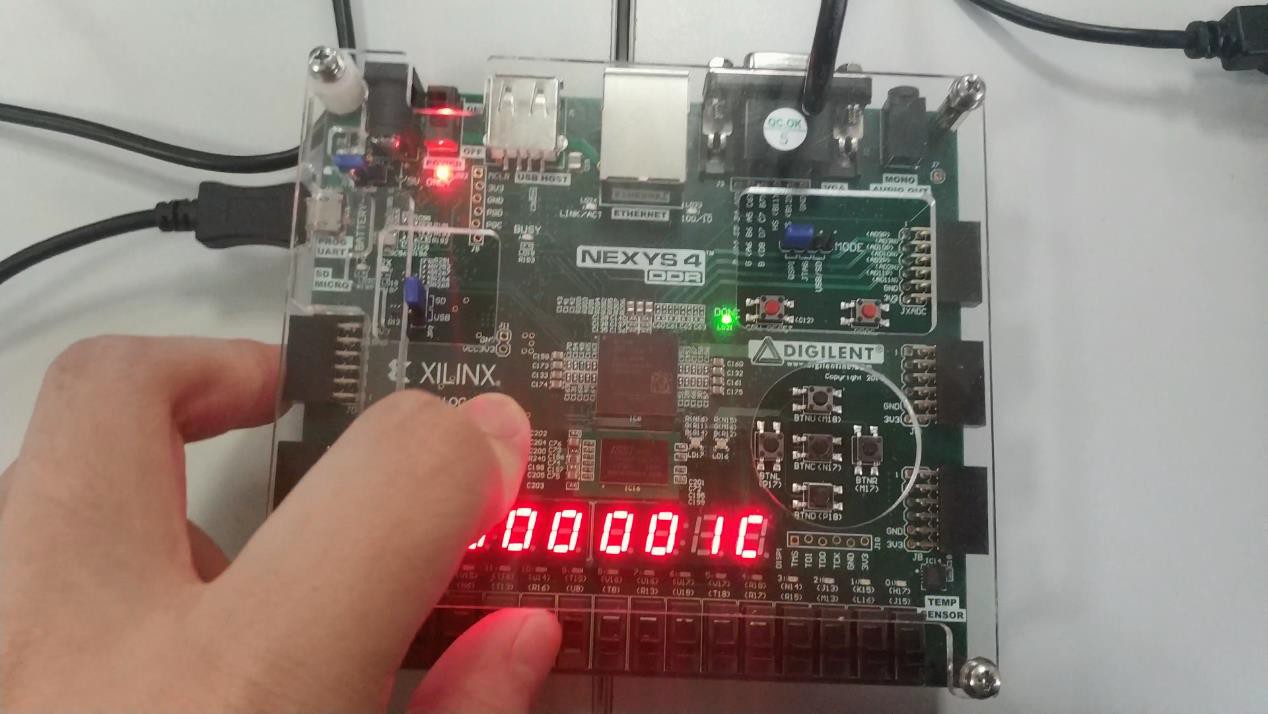
#### <https://1drv.ms/v/s!AgfTt2-0u-8j6hhjbC69aOFlR1xd>。 1) PC 被稍作修改成了第几条指令。

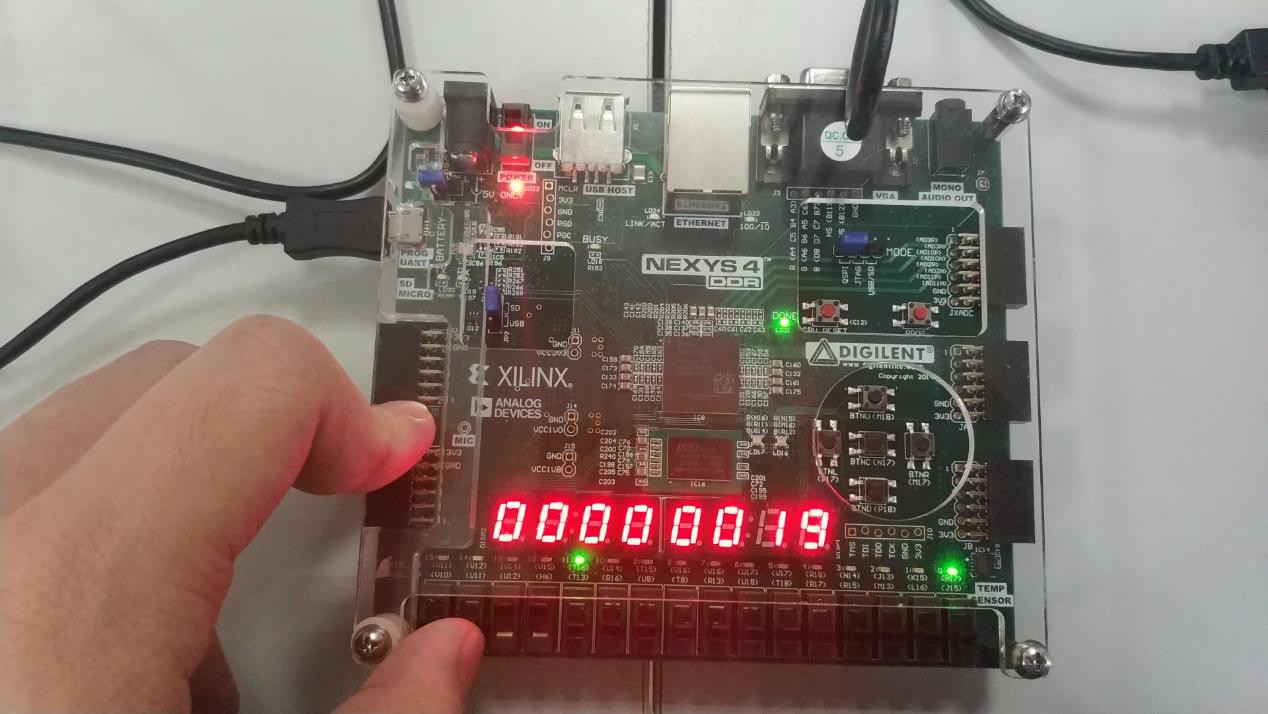




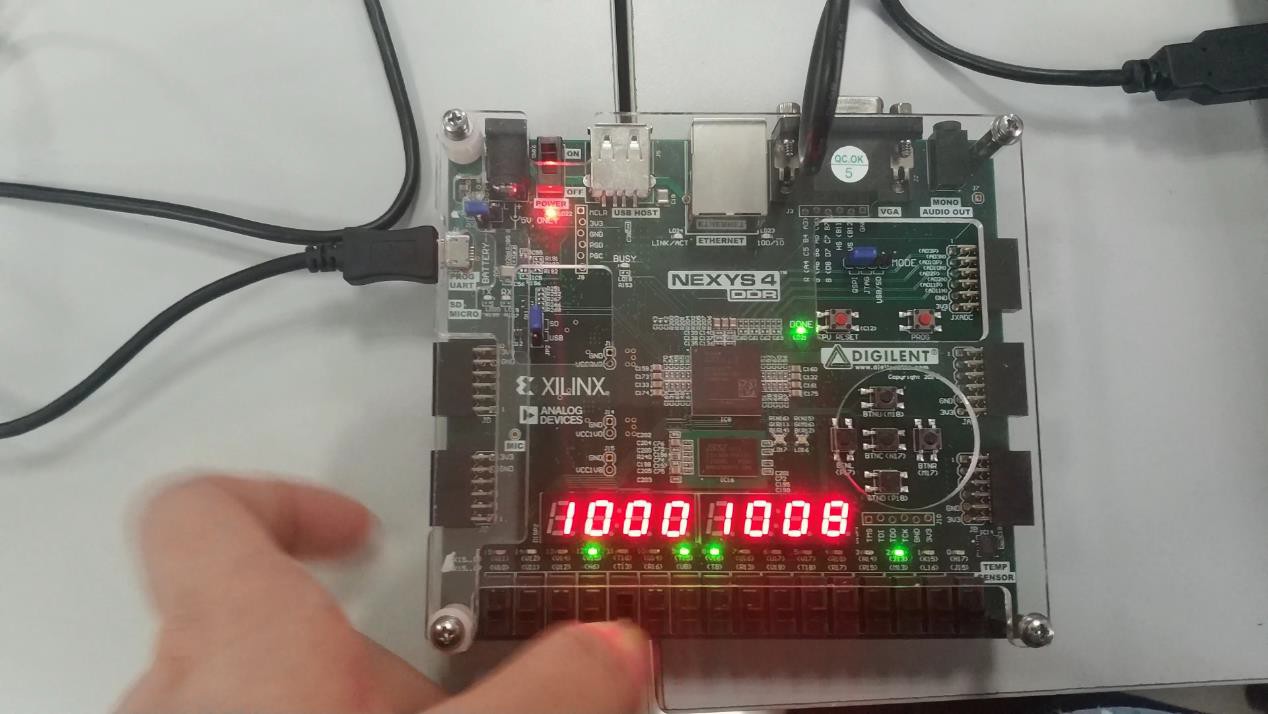


#### 2) 调整速度：





#### 3) 排序完成之后循环显示排序结果。



# 3. 参考文献

#### [1] David A. Patterson, John L. Hennessy. Computer Organization & Design: The Hardware/Software Interface. fifth edition, Morgan Kaufmann, 2013

#### [2] Missouri State University, MARS <http://courses.missouristate.edu/KenVollmar/> MARS/

# 教师评语评分

评语：

评分：

评阅人：

年 月 日