存档资料

成绩：

武汉大学计算机学院

实 验 报 告

课 程 名 称　计算机组成原理

题　　目　基于 FPGA 的 MIPS 流水线 CPU 设计

专 业 班 级　信息安全三班

学　　号　2015301500148

学 生 姓 名　张庭源

指 导 教 师　徐爱萍

2017 年 **4** 月

设计任务书

主要任务

使用硬件描述语言（Verilog HDL）设计一个基于 MIPS 指令集的流水线 CPU。该 CPU 需实现以下功能：

支持由{ addu , subu , ori , lui , lw , sw , beq , j }指令组成的简化 MIPS 指令集；

有效解决流水线 CPU 存在的旁路问题和数据冒险问题；

由仿真软件 Modelsim 对一系列汇编后的 Mips 指令进行功能波形仿真，这些指令间可能会存在数据冒险和指令冒险。

基本要求

1. 熟悉硬件描述语言（Verilog HDL）和仿真软件 Modelsim；
2. 了解基于 Mips 指令集的 CPU 的基本构架和设计原理；
3. 使用 Verilog 设计如下模块以实现前述功能：
   a) 二路选择器模块（mux2）
   b) 四路选择器模块（mux4）
   c) 程序计数器模块（PC）
   d) 指令寄存器模块（InsMem）
   e) 寄存器堆模块（RegFile）
   f) NPC 选择模块（NPC）
   g) NPC 数据处理模块（NPC_jump）
   h) 四个流水线寄存器模块（IF_ID、ID_EXE、EXE_MEM 、MEM_WB）
   i) 数据扩展模块（Ext）
   j) 控制器模块（Ctrl）
   k) 运算器模块（Alu）
   l) 运算器数据 A 选择模块（AluA_src）
   m) 运算器数据 B 选择模块（AluB_src）

n) 数据存储器模块（DataMem）

o) 写回数据选择模块（ChoseDataWrite）

p) 整机连接模块（Mips）

q) 数据初始化模块（Mips_tb）

4. 在 Modelsim 中完成汇编程序的仿真测试

参考资料

【1】计算机原理与设计：Verilog HDL 版，李亚民 著

【2】 Verilog 数字系统设计教程 夏宇闻编著 （第 17 章 简化的 RISC CPU 设计）

课程设计成绩评价表

<table>
<tr><td>课程名称</td><td colspan="7">计算机组成原理</td></tr>
<tr><td>题　　目</td><td colspan="7">MIPS 流水线 CPU 设计</td></tr>
<tr><td>学生姓名</td><td>张庭源</td><td>学号</td><td>2015301500148</td><td>指导教师姓名</td><td>徐爱萍</td><td>职称</td><td>教授</td></tr>
<tr><td>序 号</td><td colspan="2">评价项目</td><td colspan="3">指　　标</td><td>满分</td><td>评分</td></tr>
<tr><td>1</td><td colspan="2">工作量、工作态度和出勤率</td><td colspan="3">按期圆满的完成了规定的任务，难易程度和工作量符合教学要求，工作努力，遵守纪律，出勤率高，工作作风严谨，善于与他人合作。</td><td>20</td><td></td></tr>
<tr><td>2</td><td colspan="2">设计报告</td><td colspan="3">分析问题思路清晰，结构严谨，文理通顺，撰写规范，图表完备正确。</td><td>20</td><td></td></tr>
<tr><td>3</td><td colspan="2">设计结果</td><td colspan="3">能圆满调试出任务所规定的指令，仿真正确，数据冒险、数据阻塞、指令冒险处理正确。</td><td>30</td><td></td></tr>
<tr><td></td><td colspan="2">创新</td><td colspan="3">工作中有创新意识，对前人工作有一些改进或有一定应用价值。</td><td>15</td><td></td></tr>
<tr><td>4</td><td colspan="2">答辩</td><td colspan="3">能正确回答指导教师所提出的问题。</td><td>15</td><td></td></tr>
<tr><td>总
分</td><td colspan="7"></td></tr>
<tr><td colspan="8">评　语：</td></tr>
</table>

指导教师：　　年　　月　　日

目录

第一章　需求分析

本次实验是在系统学习了基于 Mips 指令集的计算机组成原理后进行的。计算机组成原理是计算机相关专业的的必修课程，在理论学习了该课程后，知识与技能结合在一起是必不可少的。为融会贯通计算机组成原理课程知识，通过对原理的综合运用和创新，加深对 CPU 系统各模块的工作原理和相互间的联系的认识。

CPU 是计算机最底层且最核心的组成部分，计算机专业的学生有必要充分地了解学习 CPU 的设计思想和运行原理，这在后续学习中起到至关重要的作用。同时也有必要学习 Verilog 语言的 EDA 设计方式，可以学习到硬件设计的理念和方式，对于一些业界常用的专业软件也可以有初步的了解。

我们有必要进行这样一次实验：通过工程设计来学习流水线 CPU 的工作原理和基于 Verilog 的硬件描述语言的设计方法，掌握采用 Modelsim 仿真技术进行调试和仿真的技术，培养科学研究的独立工作能力和分析解决问题的能力，取得 CPU 设计与仿真的实践和经验，巩固所学知识，为之后进一步的学习研究打下基础。

第二章　设计环境

2.1　Verilog HDL

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

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

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

2.2　ModelSim

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

2.3 MARS

MARS 是为 MIPS 汇编语言开发的一款轻量的集成开发环境（IDE）。

“MARS is a lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's Computer Organization and Design.”

第三章　设计概要

3.1　二路选择器模块（mux2）

1)　功能描述

mux2 提供规范化的二路选择器，用于在给定开关的情况下从两个输入的数据中选出一个输出，起到选择的作用。

2)　模块接口

信号名	方向	描述
d0 [31:0]	I	选择器输入数据 A
d1 [31:0]	I	选择器输入数据 B
s	I	0 选择数据 A，1 选择数据 B
y [31:0]	O	

3.2　四路选择器模块（mux4）

1)　功能描述

mux4 提供规范化的四路选择器，用于在给定开关的情况下从四个输入的数据中选出一个输出，起到选择的作用。

2)　模块接口

信号名	方向	描述
d0 [31:0]	I	选择器输入数据 A
d0 [31:0]	I	选择器输入数据 B
d0 [31:0]	I	选择器输入数据 C
d0 [31:0]	I	选择器输入数据 D

s [1:0]	I	00 选择输入数据 A 01 选择输入数据 B 10 选择输入数据 C 11 选择输入数据 D
y [31:0]	O	

3.3 程序计数器模块（PC）

1) 功能描述

位于 IF 级的模块。

PC 模块, 即程序计数器是用于存放下一条指令所在单元的地址的模块。它接受 NPC 所给出的下一条指令的地址，并在每次时钟上升沿将 NPC 置为 PC 输出。其中的 PCWr 信号可以阻止时钟上升沿更改 PC 值，用于完成流水线的阻塞。

信号名	方向	描述
clk	I	
rst	I	
PCWr	I	PC 写使能信号
NPC [31:0]	I	
PC [31:0]	O	

3.4 指令寄存器模块（InsMem）

1) 功能描述

位于 IF 级的模块。

输出指令内容, 提供给 IF/ID 寄存器。内含一个有 2^10 个 32 位寄存器的 InsMem, 存储有所有的指令内容，接受并使用 PC 的[11:2]10 位数据进行寻址。

2) 模块接口

信号名	方向	描述
addr [9:0]	I	10 位地址，可寻址范围为 0000000000 到 1111111111
dout [31:0]	O	32 位 MIPS 指令

3.5 NPC 选择模块（NPC）

1) 功能描述

位于 IF 级的模块。

NPC 模块接收三个 32 位宽的地址，分别是 IF 级的 PC+4，由 ID 级 NPC_jump 模块提供的 beq 指令目标地址、j 目标地址，接受一个由 Ctrl 模块提供的 2 位的 4 路选择信号，输出一个 NPC 值给 PC 模块。

2) 模块接口

信号名	方向	描述
NPC_4 [31:0]	I	PC+4
NPC_beq [31:0]	I	beq 目标地址
NPC_j [31:0]	I	j 指令目标地址
NPCOp [1:0]	I	00 选择 NPC_4 的值输出 01 选择 NPC_j 的值输出 10 选择 NPC_beq 的值输出
NPC [31:0]	O	

3.6 寄存器堆模块（RegFile）

1) 功能描述

位于 ID 级的模块。

接受 3 个 5 位寄存器号，前两个是当前指令的 rs 和 rt，第三个为 WB 级提供的目的寄存器。接受由 WB 级提供的 32 位宽的写回数据，接受一个使写能信号 RFWr。输入输出两个寄存器的值。

在控制信号 RFWr 的控制下，可以被 WD 信号写入目标寄存器的值。

2) 模块接口

信号名	方向	描述
A1 [5:0]	I	rs
A2 [5:0]	I	rt
WBdst [5:0]	I	由 MEM_WB 给出的目标寄存器
WD [31:0]	I	写回数据
RFWr	I	寄存器堆写使能
RD1 [31:0]	O	rs 地址指示寄存器中的数据
RD2 [31:0]	O	rt 地址指示寄存器中的数据

3.7 NPC 数据处理模块（NPC_jump）

1) 功能描述

位于 ID 级的模块。

将 ID 级的指令分别当作 beq 指令和 j 指令处理，接受当前指令的低 26 位，接受经 EXT 模块扩展后的 32 位有符号立即数，将其分别处理成 j 指令的地址和 beq 指令的地址，提交给 NPC 模块使用。

2) 模块接口

信号名	方向	描述
PC [31:0]	I	由 IF_ID 提供的 PC 值

Beq_offset [31:0]	I	由 EXT 提供的带符号为的 32 位立即数
im_out [25:0]	I	指令的低 26 位
NPC_beq [31:0]	O	交由 NPC 使用
NEC_j [31:0]	O	交由 NPC 使用

3.8 数据扩展模块（Ext）

1) 功能描述

位于 ID 级的模块。

此模块接受一个 16 位的输入，为当前指令的低 16 位立即数，接受一个 2 位操作数，决定数据扩展的方式。输出 32 位立即数，为扩展后的立即数。有三种扩展模式：无符号扩展、有符号扩展和低位置零扩展。

2) 模块接口

信号名	方向	描述
Imm16 [15:0]	I	16 位立即数
ExtOp [1:0]	I	00 使用无符号扩展 01 使用有符号扩展 10 使用低位置零扩展
Imm16 [31:0]	O	32 位立即数

3.9 控制器模块（Ctrl）

1) 功能描述

位于 ID 级的模块。

Ctrl 模块是 CPU 中的控制器，作用有二：为本条指令设置所有的控制信号；确定本条指令是否存在数据冒险和控制冒险，若存在，处理该冒险。具体处理方式将在第四

章中详细说明。

2) 模块接口

信号名	方向	描述
instr [31:0]	I	当前指令
EXE_stopThis	I	来自 ID_EXE 级，用于上一条是跳转指令时废除本条指令写权利
EXE_WBdst [4:0]	I	EXE 级目标寄存器
EXE_instrOp [5:0]	I	EXE 级指令代号
EXE_RegW	I	EXE 级寄存器写
MEM_WBdst [4:0]	I	MEM 级目标寄存器号
MEM_instrOp [5:0]	I	MEM 级指令代号
MEM_RegW	I	MEM 级寄存器写
RegFileA [31:0]	I	寄存器读数 A
RegFileB [31:0]	I	寄存器读数 B
MEM_out [31:0]	I	MEM 级数据存储器输出
EXE_Alu_C [31:0]	I	EXE 级 Alu 结果
MEM_Alu_C [31:0]	I	MEM 级 Alu 结果
MEM_WBsrc	I	MEM 级写回目标寄存器
instrOp [5:0]	O	当前指令代号
RegW	O	本指令寄存器写使能
RegW_Src	O	写回寄存器的数据选择（Alu 或 Mem）
NPCOp	O	NPC 操作数
MemW	O	MEM 写使能
AluAsrc	O	选择 Alu 操作数 A
AluBsrc	O	选择 Alu 操作数 B
ExtOp	O	扩展模块操作数
Aluctrl	O	Alu 控制

stopNext	O	废除下一条指令（本条指令是跳转指令）
WBdst[4:0]	O	本指令目标寄存器（rt 或 rd）
IF_IDWr	O	IF_IDWr 寄存器写使能（用于阻塞流水线）
MEMW_src	O	存储器写数据的源数据（寄存器值或旁路）
PCWr	O	PC 写使能

3.10 运算器模块（Alu）

1) 功能描述

位于 EXE 级的模块。

Alu 是 cpu 运算器模块，在本设计中只有三个共能，加、减、按位与，由 AluOp 控制。输出 32 位的计算结果 C。

2) 模块接口

信号名	方向	描述
A [31:0]	I	操作数 A
B [31:0]	I	操作数 B
ALUOp [3:0]	I	0000 进行加法运算 0100 进行减法运算 0101 进行按位或运算
C [31:0]	O	

3.11 运算器数据 A 选择模块（AluA_src）

1) 功能描述

位于 EXE 级的模块。

接受三个 32 位的数据，分别是寄存器输出值 A、Mem 级 AluC、WB 级写回数据，

接受一个 2 位控制信号，输出 Alu_A 的值。

2) 模块接口

信号名	方向	描述
RegFileA [31:0]	I	来自 ID_EXE 寄存器的 RegFileA
EXE_AluC [31:0]	I	来自 EXE_MEM 级的 AluC
MEM_WriteData [31:0]	I	来自 WB 级的写回数据
AluAsrc [1:0]	I	AluA 控制信号
AluA [31:0]	O	

3.12 运算器数据 B 选择模块（AluB_src）

1) 功能描述

位于 EXE 级的模块。

与 AluA_src 相似，确定 AluB 的值。不同之处在于此模块多一个 Ext 扩展后的立即数选项。

2) 模块接口

信号名	方向	描述
RegFileB [31:0]	I	来自 ID_EXE 寄存器的 RegFileB
EXT_Imm32 [31:0]	I	来自 ID_EXE 寄存器的扩展后立即数
EXE_AluC [31:0]	I	来自 EXE_MEM 级的 AluC
MEM_WriteData [31:0]	I	来自 WB 级的写回数据
AluBsrc [3:0]	I	AluB 控制信号
AluB [31:0]	O	

3.13 数据存储器模块（DataMem）

1) 功能描述

位于 MEM 级的模块。

DataMem 是一个有 4k 大小的存储器，由 1024 个 32 位寄存器构成，由 AluC 的[11:2]10 位寻址。接受两个 32 位写入数据信号分别是 RegfileB 和 WB 级的旁路数据，同时接受写使能信号和写入数据选择信号。

2) 模块接口

信号名	方向	描述
addr [9:0]	I	数据存储器寻址地址
din [31:0]	I	RegFileB 输入
DMWr	I	RAM 使写能信号
MEMW_src	I	0 选择 din，1 选择 WB_data 旁路
WB_data [31:0]	I	WB 级旁路
dout	O	

3.14 写回数据选择模块（ChoseDataWrite）

1) 功能描述

位于 WB 级的模块。

选择写回的数据是 AluC 还是 Mem_dout，并在时钟下降沿将此数据传回到寄存器的写数据中。

2) 模块接口

信号名	方向	描述
Alu_C [31:0]	I	Alu 结果
MemRead [31:0]	I	RAM 读取结果
RegW_Src	I	0 选择 AluC，1 选择 MemRead
clk	I	

WriteData [31:0]	O	在时钟下降沿写回

3.15 流水线寄存器模块

3.15.1 IF\ID 级寄存器模块（IF_ID）

1) 功能描述

IF 级和 ID 级间的流水线寄存器。

2) 模块接口

信号名	方向	描述
im_out [31:0]	I	
PC [31:0]	I	
clk	I	
rst	I	
IF_IDWr	I	
IF_im_out [31:0]	O	
IF_PC [31:0]	O	

3.15.2 ID\EXE 级寄存器模块（ID_EXE）

1) 功能描述

ID 级和 EXE 级间的流水线寄存器。

2) 模块接口

信号名	方向	描述
ID_RegW	I	
ID_RegW_Src	I	
ID_MemW	I	

ID_AluAsrc [1:0]	I	
ID_AluBsrc[1:0]	I	
ID_Aluctrl [3:0]	I	
ID_stopNext	I	
ID_WBdst [4:0]	I	
ID_MEMW_src	I	
[5:0] ID_instrOp	I	
ID_RegFileA[31:0]	I	
ID_RegFileB[31:0]	I	
ID_Imm32[31:0]	I	
clk	I	
rst	I	
EXE_RegW	O	
EXE_RegW_Src	O	
EXE_MemW	O	
EXE_AluAsrc [1:0]	O	
EXE_AluBsrc[1:0]	O	
EXE_Aluctrl [3:0]	O	
EXE_stopNext	O	
EXE_WBdst [4:0]	O	
EXE_instrOp[5:0]	O	
EXE_RegFileA[31:0]	O	
EXE_RegFileB[31:0]	O	
EXE_Imm32[31:0]	O	
EXE_MEMW_src	O	

3.15.3 EXE\MEM 级流水线寄存器模块（EXE_MEM）

1) 功能描述

EXE 级和 MEM 级间的流水线寄存器。

2) 模块接口

信号名	方向	描述
EXE_RegW	I	
EXE_RegW_Src	I	
EXE_MemW	I	
EXE_WBdst [4:0]	I	
EXE_instrOp [5:0]	I	
EXE_RegFileB [31:0]	I	
EXE_RegFileA [31:0]	I	
EXE_MEMW_src	I	
EXE_Alu_C [31:0]	I	
clk	I	
rst	I	
MEM_RegW	O	
MEM_RegW_Src	O	
MEM_MemW	O	
MEM_WBdst [4:0]	O	
MEM_instrOp [5:0]	O	
MEM_Alu_C [31:0]	O	
MEM_RegFileB [31:0]	O	
MEM_RegFileA [31:0]	O	
MEM_MEMW_src	O	

3.15.4 MEM\WB 级流水线寄存器模块（MEM_WB）

1) 功能描述

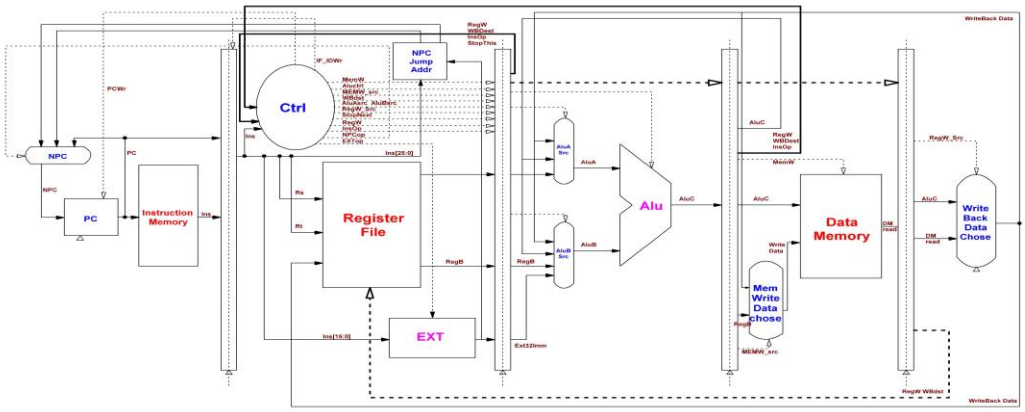
MEM 级和 WB 级间的流水线寄存器。

2) 模块接口

信号名	方向	描述
MEM_DataMem[31:0]	I	
MEM_RegW	I	
MEM_Reg_Src	I	
MEM_WBdst[4:0]	I	
MEM_Alu_C[31:0]	I	
clk	I	
rst	I	
WB_DataMem[31:0]	O	
WB_RegW	O	
WB_Reg_Src	O	
WB_WBdst[4:0]	O	
WB_Alu_C[31:0]	O	

第四章　详细设计

4.1　总设计电路图



（图见附件）

4.2　程序计数器模块（PC）

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

    input         clk;

    input         rst;

    input         PCWr;

    input  [31:0] NPC;

    output [31:0] PC;


    reg [31:0] PC;

    reg [1:0] tmp;

    always @(posedge clk or posedge rst)

    begin

        if ( rst )
```

```
            PC <= 32'h0000_3000;
        else if ( PCWr )
            PC <= NPC;
    end
endmodule
```

PC 模块，程序计数器，在每个时钟上沿将 NPC 的值赋给 PC。

同时，在每个 rst 脉冲信号上沿，将 PC 的值置为 32'h0000_3000，即测试代码的第一条指令在 InsMem 中的位置，实现了重置功能。

输入信号中，PCWr 是 PC 写使能的开关，若为 1，正常实现 PC 的功能；若为 0，使程序计数器不能在当前时钟上升沿将 NPC 的值，让上一条指令再次执行一遍，实现了一次流水线的阻塞。

4.3 寄存器堆模块（RegFile）

```
module RegFile ( A1 , A2 , WBdst , WD , RFWr , RD1 , RD2 );
    input  [4:0]    A1, A2, WBdst;
    input  [31:0]   WD;
    input          RFWr;
    output [31:0]   RD1, RD2;

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

    initial begin          //初始化
        for (i=0; i<32; i=i+1)
            regfile[i] = 0;
    end
```

```
    always @( * )
    begin
        if (RFWr)
        begin
            regfile[WBdst] = WD;
        end
    end

    assign RD1 = (A1 == 0) ? 32'd0 : regfile[A1];
    assign RD2 = (A2 == 0) ? 32'd0 : regfile[A2];
endmodule
```

基于 MIPS 的 32 位 CPU 有共有 32 个寄存器，可用 5 位数据寻址，对应于指令中的 rs、rt、rd 段。本次设计中，此寄存器堆接受两个寻址地址、一个写入地址、一个使写能信号、一个写入数据。输出两个寻址地址所对应的寄存器中的数据。

在此模块中，任何情况下（即使在 rs 或 rt 无意义的情况下），都将输出对应寄存器的值；写入寄存器则受到写使能信号的制约，该信号来自于 WB 级对应的指令内容和写寄存器的意愿，当该信号为 1 时，寄存器可以被写入，写入数据是 WB 级指令欲写入寄存器的数据，目标寄存器也由 WB 级提供。若 RFWr 为 0，则不能执行写寄存器操作。该信号统一由 Ctrl 控制器控制。

4.4 NPC 选择模块（NPC）

```
module NPC( NPC_4 , NPC_j , NPC_beq , NPCOp , NPC);
    input       [31:0]          NPC_4;
    input       [31:0]          NPC_j;
    input       [31:0]          NPC_beq;
    input       [1:0]           NPCOp;
```

```
    output      [31:0]          NPC;
    wire        [31:0]          temp;
    mux4 U_mux4(
        .d0(NPC_4),
        .d1(NPC_j),
        .d2(NPC_beq),
        .d3(temp),
        .s(NPCOp),
        .y(NPC)
    );
endmodule
```

NPC 模块接受三个地址输入，分别是当前 PC 自增 4 的地址、beq 跳转的地址、j 跳转的地址，由选择值 NPCop 选择。其中 beq 和 j 的地址来源与 ID 级的 jump 处理模块。输出正确的下一条指令并将之传递给 PC。

NPC 模块的内部实际上是一个四选一多选器。

可以看到的是，当发生跳转时，跳转指令的地址+4 已经进入流水线中了，故我们要阻止跳转指令的自增 4 地址对应的指令改变 CPU 的状态（包括两个存储器、一个寄存器堆、PC），这一点将在 4.6 节中详细讨论。

4.5 NPC 数据处理模块（NPC_jump）

```
module NPC_jump (PC , Beq_offset , im_out , NPC_beq , NPC_j);
    input   [31:0]      PC;
    input   [31:0]      Beq_offset;
    input   [25:0]      im_out;

    output  [31:0]      NPC_beq;
    output  [31:0]      NPC_j;
```

```
    wire    [31:0]      w_PC;
    wire    [31:0]      w_Beq_offset;
    //PC+4
    assign w_PC = PC + 3'b100;
    //jump
    assign NPC_j[31:28] = w_PC[31:28];
    assign NPC_j[27:2] = im_out;
    assign NPC_j[1:0] = 2'b00;
    //Beq
    assign w_Beq_offset[31:2] = Beq_offset[29:0];
    assign w_Beq_offset[1:0] = 2'b0;
    assign NPC_beq = w_PC + w_Beq_offset;
endmodule
```

本模块是 4.3 节中提到的跳转指令地址处理模块。本模块主要处理两种跳转：beq 指令和 j 指令。

beq 指令的处理方式是寄存器寻址：程序计数器 = 寄存器 + 分支地址。寄存器的值由 rs 提供，分支地址由被 ext 模块扩展过的有符号 32 位立即数提供。我们将计算程序计数器的过程提前到 ID 级，以减少阻塞的次数。对应于代码中的：

```
    assign w_Beq_offset[31:2] = Beq_offset[29:0];
    assign w_Beq_offset[1:0] = 2'b0;
    assign NPC_beq = w_PC + w_Beq_offset;
```

j 指令的处理方法是 PC 相对寻址：最终地址[31:0] = (PC+4)[31:28] + Ins[25:0] + 00。其中 PC+4 为当前指令地址自增 4，Ins[25:0]为本条指令除 op 段外低 26 位，低 2 位补 00。对应代码为：

```
    assign NPC_j[31:28] = w_PC[31:28];
    assign NPC_j[27:2] = im_out;
    assign NPC_j[1:0] = 2'b00;
```

4.6 数据扩展模块（Ext）

```
module Ext(Imm16, ExtOp, Imm32);

    input  [15:0] Imm16;

    input  [1:0]  ExtOp;

    output [31:0] Imm32;


    reg [31:0] Imm32;


    always @(*) begin

        case (ExtOp)

            2'b00:  Imm32 = {16'd0, Imm16}; //00

            2'b01:  Imm32 = {{16{Imm16[15]}}, Imm16}; //01

            2'b10:  Imm32 = {Imm16, 16'd0}; //10

            default: ;

        endcase

    end

endmodule
```

Ext 模块提供将 16 位立即数扩展为 32 位立即数的方法。有三种扩展方法 :无符号扩展、有符号扩展、低位置零扩展。三种扩展方式由 EXTOp 确定。

无符号扩展即将高 16 位置零，低 16 位填入 16 位立即数。

有符号扩展将高 16 位置为 16 位立即数的最高位，低 16 位填入 16 位立即数。这种扩展形式在本指令集中主要用于 beq 指令进行寄存器寻址。

低位置零扩展即将 32 位立即数的高 16 位填入 16 位立即数，低 16 位置零。这种扩展形式主要用于 lui 指令。

相关代码为：

```
            2'b00:  Imm32 = {16'd0, Imm16}; //00

            2'b01:  Imm32 = {{16{Imm16[15]}}, Imm16}; //01
```

```
        2'b10:  Imm32 = {Imm16, 16'd0}; //10
```

4.7 控制器模块（Ctrl）

```
//指令解析
    `define     SW_OP           6'b101011
    `define     LW_OP           6'b100011
    `define     ORI_OP          6'b001101
    `define     LUI_OP          6'b001111
    `define     BEQ_OP          6'b000100
    `define     J_OP            6'b000010
    `define     ADDU_OP         6'b100001
    `define     SUBU_OP         6'b100011
module Ctrl( instr ,EXE_stopThis , EXE_WBdst , EXE_instrOp ,
            EXE_RegW , MEM_WBdst , MEM_instrOp , MEM_RegW,
            RegFileA , RegFileB , instrOp , RegW ,
            RegW_Src , NPCOp , MemW , AluAsrc , AluBsrc ,
            ExtOp , Aluctrl , stopNext , WBdst , IF_IDWr , PCWr,
            MEM_out ,EXE_Alu_C , MEM_Alu_C ,MEM_WBsrc,MEMW_src);


    input       [31:0]          instr;
    input                       EXE_stopThis;   //来自 ID/EXE，确定是否废弃本条指
令，即上条是否跳转成功
    input       [4:0]           EXE_WBdst;
    input       [5:0]           EXE_instrOp;
    input                       EXE_RegW;
    input       [4:0]           MEM_WBdst;
    input       [5:0]           MEM_instrOp;
```

```
    input                       MEM_RegW;
    input       [31:0]          RegFileA;       //本级 rf 输出
    input       [31:0]          RegFileB;
    input       [31:0]          MEM_out;
    input       [31:0]          EXE_Alu_C;
    input       [31:0]          MEM_Alu_C;
    input                       MEM_WBsrc;


//指令分解
    output      [5:0]           instrOp;        //本条指令 Op，传给 ID_EXE
    wire        [4:0]           instrRs;
    wire        [4:0]           instrRt;
    wire        [4:0]           instrRd;
    wire        [5:0]           instrFunct;
//op 信号量
    output reg                  RegW;           //寄存器堆写入数据，为 1 写，否则不
写
    output reg                  RegW_Src;       //写入寄存器堆数据选择，1 写入 Mem
读数，否则 Alu 结果
    output reg  [1:0]           NPCOp;          //00 取 PC+4，01 取 jump 指令，10
取 beq 指令
    output reg                  MemW;           //写数据存储器
    output reg  [1:0]           AluAsrc;        //Alu_A 的选择，00 选择
RegFileA,01 选择 EXE 级 Alu_C，10 选择 MEM 级结果
    output reg  [1:0]           AluBsrc;        //运算器操作数选择,00 使用
RegFileB,01 使用立即数,10 使用 EXE 级 Alu_C，11 使用 MEM 级结果
    output reg  [1:0]           ExtOp;          //位扩展/符号扩展选择
    output reg  [3:0]           Aluctrl;        //Alu 运算选择
    output reg                  stopNext;       //是否废弃下一条指令
```

```
    output      [4:0]           WBdst;          //目标寄存器，传给 ID_EXE
    output reg                  IF_IDWr;        //IF/ID 寄存器写使能
    output reg                  MEMW_src;       //1 选 wb 级写回数据
    output reg                  PCWr;           //PC 写使能
    reg                         RegWDst;        //写目标寄存器号，为 1 选择 rd，0 为
rt
    reg         [31:0]          Num1;
    reg         [31:0]          Num2;           //用于 beq 检测
//分量赋值
    assign instrOp = instr[31:26];
    assign instrRs = instr[25:21];
    assign instrRt = instr[20:16];
    assign instrRd = instr[15:11];
    assign instrFunct = instr[5:0];
    assign WBdst = RegWDst ? instrRd : instrRt;

    initial
    begin
        RegW = 0;                               //寄存器不可写
        RegW_Src = 0;                           //从 Alu 写入寄存器，而非 MEM
        NPCOp = 2'b00;                          //取 PC+4 作为下条指令
        MemW = 0;                               //RAM 不可写
        AluAsrc = 2'b00;                        //AluB 数据源为 RegFileB
        AluBsrc = 2'b00;                        //AluA 数据源为 RegFileA
        ExtOp = 2'b00;                          //正常扩展
        Aluctrl = 4'b0000;                      //使用加法计算
        stopNext = 0;                           //下条不废弃
        IF_IDWr = 1;                            //IF/ID 可写
        PCWr = 1;                               //PC 可写
```

```
        RegWDst = 0;
        MEMW_src = 0;
    end
//op 赋值
    always @( * )
    begin
    //初始化
        RegW = 0;                                   //寄存器不可写
        RegW_Src = 0;                               //从 Alu 写入寄存器，而非 MEM
        NPCOp = 2'b00;                              //取 PC+4 作为下条指令
        MemW = 0;                                   //RAM 不可写
        AluAsrc = 2'b00;                            //AluB 数据源为 RegFileB
        AluBsrc = 2'b00;                            //AluA 数据源为 RegFileA
        ExtOp = 2'b00;                              //正常扩展
        Aluctrl = 4'b0000;                          //使用加法计算
        stopNext = 0;                               //下条不废弃
        IF_IDWr = 1;                                //IF/ID 可写
        PCWr = 1;                                   //PC 可写

        case ( instrOp )
            6'b0:
            begin
                case (instrFunct)
                    `ADDU_OP:
                    begin
                        RegW = 1;
                        RegWDst = 1;
                        Aluctrl = 4'b0000;
                    end
```

```
            `SUBU_OP:
            begin
                RegW = 1;
                Aluctrl = 4'b0100;
                RegWDst = 1;
                Aluctrl = 4'b0100;
            end
        endcase
    end
    `LUI_OP:
    begin
        RegW = 1;
        RegWDst = 0;                    //写到 rt
        AluBsrc = 2'b01;                //来自 EXT
        ExtOp = 2'b10;
    end
    `ORI_OP:
    begin
        RegW = 1;
        RegWDst = 0;
        AluBsrc = 2'b01;
        Aluctrl = 4'b0101;
    end
    `LW_OP:
    begin
        RegW = 1;
        RegWDst = 0;
        AluBsrc = 2'b01;
        RegW_Src = 1;
```

```
            end
            `SW_OP:
            begin
                MemW = 1;
                AluBsrc = 2'b01;
                RegWDst = 0;
                MEMW_src = 0;
            end
            `BEQ_OP:
            begin
                if( MEM_RegW)
                begin
                    if( instrRs == MEM_WBdst)
                    begin
                        Num1 = (MEM_WBsrc) ? MEM_out : MEM_Alu_C;
                    end
                    else
                    begin
                        Num1 = RegFileA;
                    end
                    if( instrRt == MEM_WBdst)
                    begin
                        Num2 = (MEM_WBsrc) ? MEM_out : MEM_Alu_C;
                    end
                    else
                    begin
                        Num2 = RegFileB;
                    end
                end
```

```
                if( EXE_RegW )
                begin
                    Num1 = (instrRs == EXE_WBdst) ? EXE_Alu_C : RegFileA;
                    Num2 = (instrRt == EXE_WBdst) ? EXE_Alu_C : RegFileB;
                end
                if(Num1 == Num2)        //跳转
                begin
                    stopNext = 1;
                    NPCOp = 2'b10;
                    ExtOp = 2'b01;
                end
                RegWDst = 0;
                MemW = 0;
            end
            `J_OP:
            begin
                stopNext = 1;
                NPCOp = 2'b01;
                RegWDst = 0;
                RegW = 0;
                MemW = 0;
            end
            default: $display("error!");
        endcase
//相关转发与阻塞
        if(EXE_stopThis)                        //上条跳转
        begin
            RegW = 0;
            MemW = 0;
```

```
            NPCOp = 2'b0;
        end
        else
        begin
            if(EXE_RegW)
            begin
                if(EXE_instrOp == `LW_OP)
                begin
                    RegW = 0;
                    MemW = 0;
                    IF_IDWr = 0;
                    PCWr = 0;
                end
                else if(instrOp == `SW_OP)
                begin
                    if(instrRt == EXE_WBdst)
                        begin
                            MEMW_src = 1;
                        end
                end
                else
                begin
                    if( instrRs == EXE_WBdst )
                    begin
                        AluAsrc = 01;
                    end
                    else if(RegWDst)
                    begin
                        if(instrRt == EXE_WBdst)
```

```
                        begin
                            AluBsrc = 10;
                        end
                    end
                end
            end
            if(MEM_RegW)
            begin
                if(instrRs == MEM_WBdst)
                begin
                    AluAsrc = 2'b10;
                end
                else if(RegWDst)
                begin
                    if(instrRt == MEM_WBdst)
                    begin
                        AluBsrc = 2'b11;
                    end
                end
            end
        end
    end
//
endmodule
```

控制器主要实现了两个功能：操作数控制和冒险控制，接下来将分别讨论两个内容在控制器中的实现。

4.7.1 控制信号处理

Ctrl 控制器输出的控制信号有 instrOp、RegW、RegW_Src、NPCOp、MemW、AluAsrc、AluBsrc、ExtOp、Aluctrl、stopNext、WBdst、IF_IDWr、MEMW_src、PCWr 共 14 个操作数，本节讨论按照无任何冒险发生简述这些信号的功能选择。

instrOp 是本条指令的高 6 位，是本条内容的代码。传递这条信号主要用于之后对于冒险的控制。

RegW，置 1 时，当本条指令执行到 WB 级时将写数据写回到寄存器中；置零则不写回。addu、subu、ori、lw、lui 指令需要将本指令置 1，其余置 0。

RegW_Src，置 0 时，选择 AluC 的值写回寄存器，置 1 时，选择 Mem_read 的值写回寄存器。需要写回寄存器的指令中，addu、subu、ori、lui 需要将 RegW_Src 置 0，lw 将 RegW_Src 置 1。

NPCOp，置 00 时使用 PC+4 作为下一条 PC，01 选择 j 指令的地址作为下一条 PC，10 选择 beq 作为下一条指令的地址。所有指令中，beq 置 NPCOp 为 10，j 置 NPCOp 为 01，其余均置 00。

MemW，决定本条指令是否能写 RAM。置零时不可写，置 1 时可写。所有指令中只有 sw 可置 MemW 为 1。

AluAsrc，置 00 时使用 RegFileA 的值，置 01 时使用当前 EXE 级的结果，置 10 时使用当前 MEM 级的结果。本条指令用于控制冒险，默认情况下均置 00。

AluBsrc，置 00 时使用 RegFileB 的值，置 01 时使用当前 EXE 级的结果，置 10 时使用当前 MEM 级的结果，置 11 时使用 EXT 的 32 位立即数。在无冒险的情况下，所有指令中，只有 ori 指令可置 AluBsrc 为 11，其余情况均置 00。

ExtOp，在 4.5 节中已有详细讨论，在此不再赘述。

Aluctrl，为 4 位信号，在本指令集中，0000 进行加法运算，0100 进行减法运算，0101 进行按位或运算。subu 指令置 Aluctrl 为 0100，ori 指令置 Aluctrl 为 0101，其余情况均置 Aluctrl 为 0000。

stopNext，置 1 时，使下一条指令不能改变 cpu 的状态。所有指令中，只有两条跳转指

令可置 stopNext 为 1。

WBdst，是写回寄存器的地址，可能来自于 rt 或 rd，取决于具体的指令内容。addu、subu 冒险控制部分选取 rd 作为 WBdst，ori、lw、lui 选取 rt 作为 WBdst。

IF_IDWr 和 PCWr 用于冒险控制，无冒险的情况置 0。

MEMW_src 用于冒险控制，无冒险的情况置 0。

4.7.2 冒险信号处理

4.7.2.1 控制冒险

控制冒险,即上条指令是 beq 和 j 跳转指令的情况。若这两条指令为上条指令成功跳转，会产生相应的 StopNext 信号，传递到本条指令则为 StopThis 信号。任何情况下，若 StopThis 的值为 1，将本条指令写 cpu 的能力去除，即置 RegW、MemW 为 0。这时本条指令无论进行任何操作，均不会对 cpu 产生影响。

4.7.2.2 数据冒险

数据冒险，即本条指令所需要的寄存器中的值还是前方指令的目标寄存器，故需传入之前指令的 Reg_dst、WBdata、InsOp 作为依据判断是否发生相关，若相关，则采取如下措施：

1) 若之数据相关的指令发生在 EXE 级：
   a) 相关的指令为 lw：
      i. 本条指令为 sw：

         将 MEMW_src 置为 1，即当本条指令进行到 MEM 级时，使用 WB 级的旁路进行数据存储操作。

      ii. 本条指令不为 sw：

         置本条指令的 RegW、MemW、IF_IDWr、PCWr 为 0，产生一次冒泡。

   b) 相关的指令不为 lw：
      i. 对于 addu , subu 指令：

         rs 与 EXE 级的 AluC 相关，将 AluAsrc 置为 01

         rt 与 EXE 级的 AluC 相关，将 AluBsrc 置为 01

      ii. 对于 ori , lui 指令：

         rs 与 EXE 级的 AluC 相关，将 AluAsrc 置为 01

      iii. 对于 sw 指令

         rs 与 EXE 级的 AluC 相关，将 AluAsrc 置为 01

         rt 与 EXE 级的 AluC 相关，将 MEMW_src 置为 1

      iv. 对于 lw 指令

         rs 与 EXE 级的 AluC 相关，将 AluAsrc 置为 01

v. 对于 beq 指令

rs 与 EXE 级的 AluC 相关，将 AluC 与 RegFileB 对比

rt 与 EXE 级的 AluC 相关，将 AluC 与 RegFileA 对比

2) 若之数据相关的指令发生在 MEM 级：

a) 对于 addu , subu 指令：

rs 与 MEM 级的 WBdata 相关，将 AluAsrc 置为 10

rt 与 MEM 级的 WBdata 相关，将 AluBsrc 置为 10

b) 对于 ori , lui , lw , sw , beq , j 指令

rs 与 MEM 级的 WBdata 相关，将 AluAsrc 置为 10

c) 对于 beq 指令

rs 与 MEM 级的 WBdata 相关，将 WBdata 与 RegFileB 对比

rt 与 MEM 级的 WBdata 相关，将 WBdata 与 RegFileA 对比

4.8 运算器模块（Alu）

```
module Alu (A, B, ALUOp, C);
    input  [31:0] A, B;
    input  [3:0]  ALUOp;
    output reg [31:0] C;

    always @( A or B or ALUOp )
    begin
        case ( ALUOp )
            4'b0000: C = A + B;           //加
            4'b0100: C = A - B;           //减
            4'b0101: C = A | B;           //或
        endcase
```

```
    end

endmodule
```

4.9 运算器数据 A 选择模块（AluA_src）

```
module AluA_src(RegFileA , EXE_AluC , MEM_WriteData , AluAsrc , AluA);

    input   [31:0]         RegFileA;

    input   [31:0]         EXE_AluC;

    input   [31:0]         MEM_WriteData;

    input   [1:0]          AluAsrc;

    output  [31:0]         AluA;


    wire    [31:0]         temp;


    mux4 U_mux4(

        .d0(RegFileA),

        .d1(EXE_AluC),

        .d2(MEM_WriteData),

        .d3(temp),

        .s(AluAsrc),

        .y(AluA)

    );

endmodule
```

4.10 指令寄存器模块（InsMem）

```
module InsMem( addr, dout );
    input [9:0]    addr;
    output [31:0] dout;
    reg [31:0] imem[1023:0];
    reg [31:0]  dout;
    always @( addr )
    begin
        dout <= imem[addr];
    end
endmodule
```

4.11 运算器数据 B 选择模块（AluB_src）

```
module AluB_src(RegFileB , EXT_Imm32, EXE_AluC , MEM_WriteData , AluBsrc ,
AluB);
    input   [31:0]         RegFileB;
    input   [31:0]         EXT_Imm32;
    input   [31:0]         EXE_AluC;
    input   [31:0]         MEM_WriteData;
    input   [1:0]          AluBsrc;
    output  [31:0]         AluB;

    mux4 U_mux4(
        .d0(RegFileB),
        .d1(EXT_Imm32),
```

```
       .d2(EXE_AluC),
       .d3(MEM_WriteData),
       .s(AluBsrc),
       .y(AluB)
   );
endmodule
```

4.12 数据存储器模块（DataMem）

```
module DataMem( addr, din, DMWr, dout ,MEMW_src ,WB_data );
   input  [9:0] addr;
   input  [31:0] din;
   input        DMWr;
   input         MEMW_src;
   input  [31:0]  WB_data;
   output [31:0] dout;

   wire    [31:0] r_din;
   assign   r_din = MEMW_src ? WB_data : din;
   reg [31:0] dmem[1023:0];

   always @( * )
   begin
      if (DMWr)
         dmem[addr] <= r_din;
   end
   assign dout = dmem[addr];
endmodule
```

4.13 写回数据选择模块（ChoseDataWrite）

```
module ChoseDataWrite(Alu_C ,MemRead ,RegW_Src ,clk ,WriteData);

    input   [31:0]     Alu_C;

    input   [31:0]     MemRead;

    input              RegW_Src;

    input              clk;

    output  reg[31:0]   WriteData;


    wire    [31:0]     r_WriteData;


    mux2 U_mux2(

           .d0(Alu_C),

           .d1(MemRead),

           .s(RegW_Src),

           .y(r_WriteData)

    );


    always @(negedge clk)

    begin

        WriteData = r_WriteData;

    end

endmodule
```

4.14 整机连接模块（Mips）

```
module mips( clk, rst );

    input                   clk;

    input                   rst;

//IF

    //IM

    wire       [9:0]         IM_addr;

    wire       [31:0]        IM_dout;

    //PC

    wire                    PC_PCWr;

    wire       [31:0]        PC_NPC;

    wire       [31:0]        PC_PC;

    //NPC

    wire       [31:0]        NPC_NPC_4;

    wire       [31:0]        NPC_NPC_j;

    wire       [31:0]        NPC_NPC_beq;

    wire       [1:0]         NPC_NPCOp;

    wire       [31:0]        NPC_NPC;

    //regIF_ID

    wire       [31:0]        IF_ID_im_out;

    wire       [31:0]        IF_ID_PC;

    wire       [31:0]        IF_ID_IF_im_out;

    wire       [31:0]        IF_ID_IF_PC;

    wire                     IF_ID_IF_IDWr;

//ID

    //NPC_jump

    wire       [31:0]        NPC_jump_PC;

    wire       [31:0]        NPC_jump_Beq_offset;
```

```
    wire        [25:0]          NPC_jump_im_out;
    wire        [31:0]          NPC_jump_NPC_beq;
    wire        [31:0]          NPC_jump_NPC_j;
    //RegFile
    wire        [4:0]           RegFile_A1, RegFile_A2, RegFile_WBdst;
    wire        [31:0]          RegFile_WD;
    wire                       RegFile_RFWr;
    wire        [31:0]          RegFile_RD1, RegFile_RD2;
    //Ext
    wire        [15:0]          Ext_Imm16;
    wire        [1:0]           Ext_ExtOp;
    wire        [31:0]          Ext_Imm32;
    //Ctrl
     wire         [31:0]           Ctrl_instr;
     wire                          Ctrl_EXE_stopThis;  //来自 ID/EXE,确定是否废弃本
条指令，即上条是否跳转成功
     wire         [4:0]            Ctrl_EXE_WBdst;
     wire         [5:0]            Ctrl_EXE_instrOp;
     wire                          Ctrl_EXE_RegW;
     wire         [4:0]            Ctrl_MEM_WBdst;
     wire         [5:0]            Ctrl_MEM_instrOp;
     wire                          Ctrl_MEM_RegW;
     wire         [31:0]           Ctrl_RegFileA;      //本级 rf 输出
     wire         [31:0]           Ctrl_RegFileB;
     wire         [5:0]            Ctrl_instrOp;       //本条指令 Op，传给 ID_EXE
     wire                         Ctrl_RegW;          //寄存器堆写入数据，为 1 写，
否则不写
     wire                         Ctrl_RegW_Src;      //写入寄存器堆数据选择,1 写入
Mem 读数，否则 Alu 结果
```

```
    wire        [1:0]           Ctrl_NPCOp;         //00 取 PC+4,01 取 jump 指令,
10 取 beq 指令
    wire                        Ctrl_MemW;          //写数据存储器
    wire        [1:0]           Ctrl_AluAsrc;       //Alu_A 的选择，00 选择
RegFileA,01 选择 EXE 级 Alu_C，10 选择 MEM 级结果
    wire        [1:0]           Ctrl_AluBsrc;       //运算器操作数选择,00 使用
RegFileB,01 使用立即数,10 使用 EXE 级 Alu_C，11 使用 MEM 级结果
    wire        [1:0]           Ctrl_ExtOp;         //位扩展/符号扩展选择
    wire        [3:0]           Ctrl_Aluctrl;       //Alu 运算选择
    wire                        Ctrl_stopNext;      //是否废弃下一条指令
    wire         [4:0]          Ctrl_WBdst;         //目标寄存器，传给 ID_EXE
    wire                        Ctrl_IF_IDWr;       //IF/ID 寄存器写使能
    wire                        Ctrl_PCWr;          //PC 写使能
   wire       [31:0]         Ctrl_MEM_out;
   wire       [31:0]         Ctrl_EXE_Alu_C;
   wire       [31:0]         Ctrl_MEM_Alu_C;
   wire                      Ctrl_MEM_WBsrc;
   wire                      Ctrl_MEMW_src;
   //regID_EXE_
   wire                      ID_EXE_ID_RegW;         //寄存器堆写入数据，为 1 写，
否则不写
   wire                      ID_EXE_ID_RegW_Src;     //写入寄存器堆数据选择,1 写入
Mem 读数，否则 Alu 结果
   wire                      ID_EXE_ID_MemW;         //写数据存储器
   wire       [1:0]          ID_EXE_ID_AluAsrc;        //Alu_A 的选择，00 选择
RegFileA,01 选择 EXE 级 Alu_C，10 选择 MEM 级结果
   wire       [1:0]        ID_EXE_ID_AluBsrc;          //运算器操作数选择
   wire       [3:0]            ID_EXE_ID_Aluctrl;          //Alu 运算选择
   wire                      ID_EXE_ID_stopNext;     //是否废弃下一条指令
```

```
    wire        [4:0]        ID_EXE_ID_WBdst;          //目标寄存器，传给 ID_EXE
    wire        [5:0]        ID_EXE_ID_instrOp;          // 本 条 指 令 Op ， 传 给
ID_EXE
    wire        [31:0]         ID_EXE_ID_RegFileA;
    wire        [31:0]         ID_EXE_ID_RegFileB;
    wire        [31:0]         ID_EXE_ID_Imm32;
    wire                       ID_EXE_ID_MEMW_src;
    wire                       ID_EXE_EXE_RegW;            //寄存器堆写入数据，为 1
写，否则不写
    wire                       ID_EXE_EXE_RegW_Src;        //写入寄存器堆数据选择，
1 写入 Mem 读数，否则 Alu 结果
    wire                       ID_EXE_EXE_MemW;            //写数据存储器
    wire        [1:0]          ID_EXE_EXE_AluAsrc;         //Alu_A 的选择，00 选择
RegFileA,01 选择 EXE 级 Alu_C，10 选择 MEM 级结果
    wire        [1:0]         ID_EXE_EXE_AluBsrc;      //运算器操作数选择
    wire        [3:0]             ID_EXE_EXE_Aluctrl;      //Alu 运算选择
    wire                       ID_EXE_EXE_stopNext;        //是否废弃下一条指令
    wire        [4:0]        ID_EXE_EXE_WBdst;             // 目 标 寄 存 器 ， 传 给
EXE_EXE
    wire        [5:0]         ID_EXE_EXE_instrOp;      //本条指令 Op，传给 EXE_EXE
    wire        [31:0]         ID_EXE_EXE_RegFileA;
    wire        [31:0]         ID_EXE_EXE_RegFileB;
    wire        [31:0]         ID_EXE_EXE_Imm32;
    wire                       EXE_MEMW_src;
//EXE
    //Alu_
    wire        [31:0]         Alu_A, Alu_B;
    wire        [3:0]          Alu_ALUOp;
    wire        [31:0]         Alu_C;
```

```
    //AluA_src
    wire       [31:0]         AluA_src_RegFileA;
    wire       [31:0]         AluA_src_EXE_AluC;
    wire       [31:0]         AluA_src_MEM_WriteData;
    wire       [1:0]          AluA_src_AluAsrc;
    wire       [31:0]         AluA_src_AluA;
    //AluB_src_
    wire       [31:0]         AluB_src_RegFileB;
    wire       [31:0]         AluB_src_EXT_Imm32;
    wire       [31:0]         AluB_src_EXE_AluC;
    wire       [31:0]         AluB_src_MEM_WriteData;
    wire       [1:0]          AluB_src_AluBsrc;
    wire       [31:0]         AluB_src_AluB;
    //regEXE_MEM_
    wire                      EXE_MEM_EXE_RegW;          //寄存器堆写入数据，为 1
写，否则不写
    wire                      EXE_MEM_EXE_RegW_Src;      //写入寄存器堆数据选择，
1 写入 Mem 读数，否则 Alu 结果
    wire                      EXE_MEM_EXE_MemW;          //写数据存储器
    wire       [4:0]         EXE_MEM_EXE_WBdst;         //目标寄存器
    wire       [5:0]         EXE_MEM_EXE_instrOp;       //本条指令 Op
    wire       [31:0]         EXE_MEM_EXE_Alu_C;
    wire       [31:0]         EXE_MEM_EXE_RegFileB;
    wire       [31:0]         EXE_MEM_EXE_RegFileA;
    wire                      EXE_MEM_EXE_MEMW_src;
    wire       [31:0]         EXE_MEM_MEM_RegFileA;
    wire                      EXE_MEM_MEM_RegW;
    wire                      EXE_MEM_MEM_RegW_Src;
    wire                      EXE_MEM_MEM_MemW;
```

```
    wire       [4:0]       EXE_MEM_MEM_WBdst;
    wire       [5:0]       EXE_MEM_MEM_instrOp;
    wire       [31:0]        EXE_MEM_MEM_Alu_C;
    wire       [31:0]        EXE_MEM_MEM_RegFileB;
    wire                     EXE_MEM_MEM_MEMW_src;
//MEM
    //DataMem
    wire       [9:0]         DM_addr;
    wire       [31:0]        DM_din;
    wire                     DM_DMWr;
    wire       [31:0]        DM_dout;
    wire                     DM_MEMW_src;
    wire       [31:0]        DM_WB_data;
    //regMEM_WB_
    wire       [31:0]        MEM_WB_MEM_DataMem;
    wire                     MEM_WB_MEM_RegW;
    wire                     MEM_WB_MEM_Reg_Src;
    wire       [4:0]         MEM_WB_MEM_WBdst;
    wire       [31:0]        MEM_WB_MEM_Alu_C;
    wire       [31:0]        MEM_WB_WB_DataMem;
    wire                     MEM_WB_WB_RegW;
    wire                     MEM_WB_WB_Reg_Src;
    wire       [4:0]         MEM_WB_WB_WBdst;
    wire       [31:0]        MEM_WB_WB_Alu_C;
//WB_
    wire       [31:0]        WB_Alu_C;
    wire       [31:0]        WB_MemRead;
    wire                     WB_RegW_Src;
    wire       [31:0]        WB_WriteData;
```

```
//初始化
    //IF
    InsMem U_InsMem(
        .addr(IM_addr),
        .dout(IM_dout)
    );
    PC U_PC(
        .clk(clk),
        .rst(rst),
        .PCWr(PC_PCWr),
        .NPC(PC_NPC),
        .PC(PC_PC)
    );
    NPC U_NPC(
        .NPC_4(NPC_NPC_4) ,
        .NPC_j(NPC_NPC_j) ,
        .NPC_beq(NPC_NPC_beq) ,
        .NPCOp(NPC_NPCOp) ,
        .NPC(NPC_NPC)
    );
    regIF_ID U_regIF_ID(
        .clk(clk) ,
        .rst(rst) ,
        .im_out(IF_ID_im_out) ,
        .PC(IF_ID_PC) ,
        .IF_im_out(IF_ID_IF_im_out) ,
        .IF_PC(IF_ID_IF_PC),
        .IF_IDWr(IF_ID_IF_IDWr)
    );
```

```
//ID
NPC_jump U_NPC_jump(
    .PC(NPC_jump_PC) ,
    .Beq_offset(NPC_jump_Beq_offset) ,
    .im_out(NPC_jump_im_out) ,
    .NPC_beq(NPC_jump_NPC_beq) ,
    .NPC_j(NPC_jump_NPC_j)
);
RegFile U_RegFile(
     .A1(RegFile_A1) ,
     .A2(RegFile_A2) ,
     .WBdst(RegFile_WBdst) ,
     .WD(RegFile_WD) ,
     .RFWr(RegFile_RFWr) ,
     .RD1(RegFile_RD1) ,
     .RD2(RegFile_RD2)
);
Ext U_Ext(
    .Imm16(Ext_Imm16),
    .ExtOp(Ext_ExtOp),
    .Imm32(Ext_Imm32)
);


Ctrl U_Ctrl(
    .instr(Ctrl_instr) ,
    .EXE_stopThis(Ctrl_EXE_stopThis),
    .EXE_WBdst(Ctrl_EXE_WBdst) ,
    .EXE_instrOp(Ctrl_EXE_instrOp) ,
     .EXE_RegW(Ctrl_EXE_RegW) ,
```

```
    .MEM_WBdst(Ctrl_MEM_WBdst) ,
    .MEM_instrOp(Ctrl_MEM_instrOp) ,
    .MEM_RegW(Ctrl_MEM_RegW),
     .RegFileA(Ctrl_RegFileA) ,
    .RegFileB(Ctrl_RegFileB) ,
    .instrOp(Ctrl_instrOp) ,
    .RegW(Ctrl_RegW) ,
     .RegW_Src(Ctrl_RegW_Src) ,
    .NPCOp(Ctrl_NPCOp) ,
    .MemW(Ctrl_MemW) ,
    .AluAsrc(Ctrl_AluAsrc) ,
    .AluBsrc(Ctrl_AluBsrc) ,
     .ExtOp(Ctrl_ExtOp) ,
    .Aluctrl(Ctrl_Aluctrl) ,
    .stopNext(Ctrl_stopNext) ,
    .WBdst(Ctrl_WBdst) ,
    .IF_IDWr(Ctrl_IF_IDWr) ,
    .PCWr(Ctrl_PCWr),
    .MEM_out(Ctrl_MEM_out),
    .EXE_Alu_C(Ctrl_EXE_Alu_C),
    .MEM_Alu_C(Ctrl_MEM_Alu_C),
    .MEM_WBsrc(Ctrl_MEM_WBsrc),
    .MEMW_src(Ctrl_MEMW_src)
);


regID_EXE U_regID_EXE(
    .ID_RegW(ID_EXE_ID_RegW) ,
    .ID_RegW_Src(ID_EXE_ID_RegW_Src) ,
    .ID_MemW(ID_EXE_ID_MemW) ,
```

```
    .ID_AluAsrc(ID_EXE_ID_AluAsrc) ,
    .ID_AluBsrc(ID_EXE_ID_AluBsrc) ,
    .ID_Aluctrl(ID_EXE_ID_Aluctrl) ,
    .ID_stopNext(ID_EXE_ID_stopNext) ,
    .ID_WBdst(ID_EXE_ID_WBdst) ,
    .ID_instrOp(ID_EXE_ID_instrOp) ,
    .ID_RegFileA(ID_EXE_ID_RegFileA) ,
    .ID_RegFileB(ID_EXE_ID_RegFileB) ,
    .ID_Imm32(ID_EXE_ID_Imm32),
    .ID_MEMW_src(ID_EXE_ID_MEMW_src),
    .clk(clk),
    .rst(rst),
    .EXE_RegW(ID_EXE_EXE_RegW) ,
    .EXE_RegW_Src(ID_EXE_EXE_RegW_Src) ,
    .EXE_MemW(ID_EXE_EXE_MemW) ,
    .EXE_AluAsrc(ID_EXE_EXE_AluAsrc) ,
    .EXE_AluBsrc(ID_EXE_EXE_AluBsrc) ,
    .EXE_Aluctrl(ID_EXE_EXE_Aluctrl) ,
    .EXE_stopNext(ID_EXE_EXE_stopNext) ,
    .EXE_WBdst(ID_EXE_EXE_WBdst) ,
    .EXE_instrOp(ID_EXE_EXE_instrOp) ,
    .EXE_RegFileA(ID_EXE_EXE_RegFileA) ,
    .EXE_RegFileB(ID_EXE_EXE_RegFileB) ,
    .EXE_Imm32(ID_EXE_EXE_Imm32),
    .EXE_MEMW_src(ID_EXE_EXE_MEMW_src)
);
//EXE
Alu U_Alu(
    .A(Alu_A),
```

```
    .B(Alu_B),
    .ALUOp(Alu_ALUOp) ,
    .C(Alu_C)
);
AluA_src U_AluA_src(
    .RegFileA(AluA_src_RegFileA) ,
    .EXE_AluC(AluA_src_EXE_AluC) ,
    .MEM_WriteData(AluA_src_MEM_WriteData) ,
    .AluAsrc(AluA_src_AluAsrc) ,
    .AluA(AluA_src_AluA)
);
AluB_src U_AluB_src(
    .RegFileB(AluB_src_RegFileB) ,
    .EXT_Imm32(AluB_src_EXT_Imm32),
    .EXE_AluC(AluB_src_EXE_AluC) ,
    .MEM_WriteData(AluB_src_MEM_WriteData) ,
    .AluBsrc(AluB_src_AluBsrc) ,
    .AluB(AluB_src_AluB)
);
regEXE_MEM U_regEXE_MEM(
    .MEM_RegW(EXE_MEM_MEM_RegW) ,
    .MEM_RegW_Src(EXE_MEM_MEM_RegW_Src) ,
    .MEM_MemW(EXE_MEM_MEM_MemW) ,
    .MEM_WBdst(EXE_MEM_MEM_WBdst),
    .MEM_instrOp(EXE_MEM_MEM_instrOp) ,
    .EXE_RegFileB(EXE_MEM_EXE_RegFileB),
    .MEM_MEMW_src(EXE_MEM_MEM_MEMW_src),
    .clk(clk) ,
    .rst(rst) ,
```

```
    .EXE_RegW(EXE_MEM_EXE_RegW) ,
    .EXE_RegW_Src(EXE_MEM_EXE_RegW_Src),
    .EXE_Alu_C(EXE_MEM_EXE_Alu_C),
    .EXE_MemW(EXE_MEM_EXE_MemW) ,
    .MEM_Alu_C(EXE_MEM_MEM_Alu_C),
    .EXE_WBdst(EXE_MEM_EXE_WBdst) ,
    .EXE_instrOp(EXE_MEM_EXE_instrOp),
    .MEM_RegFileB(EXE_MEM_MEM_RegFileB),
    .EXE_MEMW_src(EXE_MEM_EXE_MEMW_src)
);
//MEM
DataMem U_DataMem(
    .addr(DM_addr),
    .din(DM_din),
    .DMWr(DM_DMWr),
    .dout(DM_dout) ,
    .MEMW_src(DM_MEMW_src),
    .WB_data(DM_WB_data)
);
regMEM_WB U_regMEM_WB(
    .MEM_DataMem(MEM_WB_MEM_DataMem) ,
    .MEM_RegW(MEM_WB_MEM_RegW) ,
    .rst(rst) ,
    .MEM_Reg_Src(MEM_WB_MEM_Reg_Src) ,
    .MEM_WBdst(MEM_WB_MEM_WBdst) ,
    .MEM_Alu_C(MEM_WB_MEM_Alu_C) ,
    .clk(clk) ,
    .WB_DataMem(MEM_WB_WB_DataMem) ,
    .WB_RegW(MEM_WB_WB_RegW) ,
```

```
        .WB_Reg_Src(MEM_WB_WB_Reg_Src) ,
        .WB_WBdst(MEM_WB_WB_WBdst) ,
        .WB_Alu_C(MEM_WB_WB_Alu_C)
    );
    //WB
    ChoseDataWrite U_ChoseDataWrite(
        .Alu_C(WB_Alu_C) ,
        .MemRead(WB_MemRead) ,
        .RegW_Src(WB_RegW_Src) ,
        .clk(clk),
        .WriteData(WB_WriteData)
    );
//IF连线
    assign IM_addr = PC_PC[11:2];

    assign PC_PCWr = Ctrl_PCWr;
    assign PC_NPC = NPC_NPC;

    assign NPC_NPC_4 = PC_PC + 3'b100;
    assign NPC_NPC_j = NPC_jump_NPC_j;
    assign NPC_NPC_beq = NPC_jump_NPC_beq;
    assign NPC_NPCOp = Ctrl_NPCOp;

    assign IF_ID_im_out = IM_dout;
    assign IF_ID_PC = PC_PC;

//ID连线
    assign NPC_jump_PC = IF_ID_IF_PC;
    assign NPC_jump_Beq_offset = Ext_Imm32;
```

```
assign NPC_jump_im_out = IF_ID_IF_im_out;

assign IF_ID_IF_IDWr = Ctrl_IF_IDWr;


assign RegFile_A1 = IF_ID_IF_im_out [25:21];

assign RegFile_A2 = IF_ID_IF_im_out [20:16];

assign RegFile_WBdst = MEM_WB_WB_WBdst;

assign RegFile_RFWr = MEM_WB_WB_RegW;

assign RegFile_WD = WB_WriteData;


assign Ext_Imm16 = IF_ID_IF_im_out;

assign Ext_ExtOp = Ctrl_ExtOp;


assign Ctrl_instr = IF_ID_IF_im_out;

assign Ctrl_EXE_stopThis = ID_EXE_EXE_stopNext;

assign Ctrl_EXE_WBdst = ID_EXE_EXE_WBdst;

assign Ctrl_EXE_instrOp = ID_EXE_EXE_instrOp;

assign Ctrl_EXE_instrOp = ID_EXE_EXE_instrOp;

assign Ctrl_MEM_WBdst = EXE_MEM_MEM_WBdst;

assign Ctrl_MEM_instrOp = EXE_MEM_MEM_instrOp;

assign Ctrl_MEM_RegW = EXE_MEM_MEM_RegW;

assign Ctrl_RegFileA = RegFile_RD1;

assign Ctrl_RegFileB = RegFile_RD2;

assign Ctrl_EXE_RegW = ID_EXE_EXE_RegW;

assign Ctrl_MEM_out = DM_dout;

assign Ctrl_MEM_Alu_C = EXE_MEM_MEM_Alu_C;

assign Ctrl_MEM_WBsrc = EXE_MEM_MEM_WBdst;

assign Ctrl_EXE_Alu_C = Alu_C;


assign ID_EXE_ID_RegW = Ctrl_RegW;
```

```
    assign ID_EXE_ID_RegW_Src = Ctrl_RegW_Src;
    assign ID_EXE_ID_MemW = Ctrl_MemW;
    assign ID_EXE_ID_AluAsrc = Ctrl_AluAsrc;
    assign ID_EXE_ID_AluBsrc = Ctrl_AluBsrc;
    assign ID_EXE_ID_Aluctrl = Ctrl_Aluctrl;
    assign ID_EXE_ID_stopNext = Ctrl_stopNext;
    assign ID_EXE_ID_WBdst = Ctrl_WBdst;
    assign ID_EXE_ID_instrOp = Ctrl_instrOp;
    assign ID_EXE_ID_RegFileA = RegFile_RD1;
    assign ID_EXE_ID_RegFileB = RegFile_RD2;
    assign ID_EXE_ID_Imm32 = Ext_Imm32;
    assign ID_EXE_ID_MEMW_src = Ctrl_MEMW_src;


//EXE 连线
    assign Alu_A = AluA_src_AluA;
    assign Alu_B = AluB_src_AluB;
    assign Alu_ALUOp = ID_EXE_EXE_Aluctrl;


    assign AluA_src_RegFileA = ID_EXE_EXE_RegFileA;
    assign AluA_src_EXE_AluC = EXE_MEM_MEM_Alu_C;
    assign AluA_src_MEM_WriteData = WB_WriteData;
    assign AluA_src_AluAsrc = ID_EXE_EXE_AluAsrc;


    assign AluB_src_RegFileB = ID_EXE_EXE_RegFileB;
    assign AluB_src_EXT_Imm32 = ID_EXE_EXE_Imm32;
    assign AluB_src_EXE_AluC = EXE_MEM_MEM_Alu_C;
    assign AluB_src_MEM_WriteData = WB_WriteData;
    assign AluB_src_AluBsrc = ID_EXE_EXE_AluBsrc;
```

```
    assign EXE_MEM_EXE_RegW = ID_EXE_EXE_RegW;
    assign EXE_MEM_EXE_RegW_Src = ID_EXE_EXE_RegW_Src;
    assign EXE_MEM_EXE_MemW = ID_EXE_EXE_MemW;
    assign EXE_MEM_EXE_instrOp = ID_EXE_EXE_MemW;
    assign EXE_MEM_EXE_Alu_C = Alu_C;
    assign EXE_MEM_EXE_RegFileB = ID_EXE_EXE_RegFileB;
    assign EXE_MEM_EXE_WBdst = ID_EXE_EXE_WBdst;
    assign EXE_MEM_EXE_RegFileA = ID_EXE_EXE_RegFileA;
    assign EXE_MEM_EXE_MEMW_src = ID_EXE_EXE_MEMW_src;

//MEM 连线
    assign DM_addr = EXE_MEM_MEM_Alu_C[11:2];
    assign DM_din = EXE_MEM_MEM_RegFileB;
    assign DM_DMWr = EXE_MEM_MEM_MemW;
    assign DM_MEMW_src = EXE_MEM_MEM_MEMW_src;
    assign DM_WB_data = WB_WriteData;

    assign MEM_WB_MEM_DataMem = DM_dout;
    assign MEM_WB_MEM_RegW = EXE_MEM_MEM_RegW;
    assign MEM_WB_MEM_Reg_Src = EXE_MEM_MEM_RegW_Src;
    assign MEM_WB_MEM_WBdst = EXE_MEM_MEM_WBdst;
    assign MEM_WB_MEM_Alu_C = EXE_MEM_MEM_Alu_C;
//WB 连线
    assign WB_Alu_C = MEM_WB_WB_Alu_C;
    assign WB_MemRead = MEM_WB_WB_DataMem;
    assign WB_RegW_Src = MEM_WB_WB_Reg_Src;
endmodule
```

4.15 数据初始化模块（Mips_tb）

```
module mips_tb();


   reg clk, rst;


   mips U_MIPS(
      .clk(clk), .rst(rst)
   );


   initial begin
      $readmemh( "code1.txt" , U_MIPS.U_InsMem.imem ) ;
      $monitor("PC = 0x%8X, IR = 0x%8X", U_MIPS.U_PC.PC, U_MIPS.IM_dout );
      clk = 1 ;
      rst = 0 ;
      #5 ;
      rst = 1 ;
      #20 ;
      rst = 0 ;
   end


   always
       #(50) clk = ~clk;


endmodule
```

MIps_tb 实现了数据初始化的功能，将生成的 16 进制代码载入到 InsMem 中。具体实现代码中调用了一个 verilog 系统调用：$readmemh()

```
      $readmemh( "code1.txt" , U_MIPS.U_InsMem.imem ) ;
```

第五章　测试和结果分析

5.1　测试文件

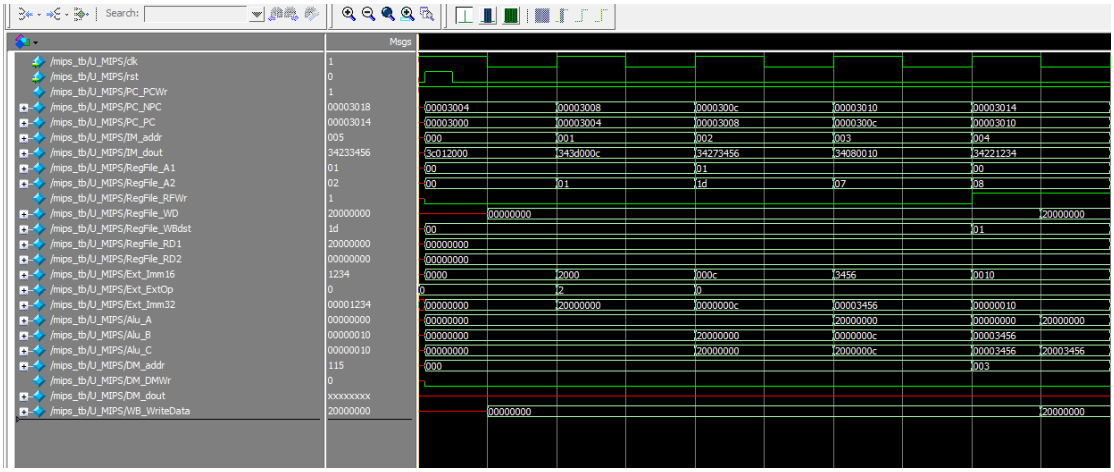
```
lui  $1,0x2000

ori $29, $1, 12

ori $7, $1, 0x3456

ori $8, $0, 0x10

ori $2, $1, 0x1234

ori $3, $1, 0x3456

f0:

addu $4, $2, $3

sw $2, 0($0)

sw $3, 4($0)

sw $4, 8($0)

lw $5, 0($0)

beq $2, $5, _lb2

subu $6, $3, $4

subu $6, $3, $4

_lb1:

lw $3, 4($0)

_lb2:

lw $5, 8($0)

subu $6, $3, $5

sw $6, -4($8)

j f0
```

5.2 测试机器码

```
3c012000

343d000c

34273456

34080010

34221234

34233456

00432021

ac020000

ac030004

ac040008

8c050000

10450003

00643023

00643023

8c030004

8c050008

00653023

ad06fffc

08000c06
```

5.3 测试结果分析

5.3.1 lui $1,0x2000



图示的五个周期为第 1 条指令 lui $1,0x2000 执行的周期。

在第一周期，PC 读到这条指令的 16 进制地址为$(00003000)_{16}$，指令寄存器读到该地址的指令为$(3c012000)_{16}$，保存到第一级流水线寄存器当中；

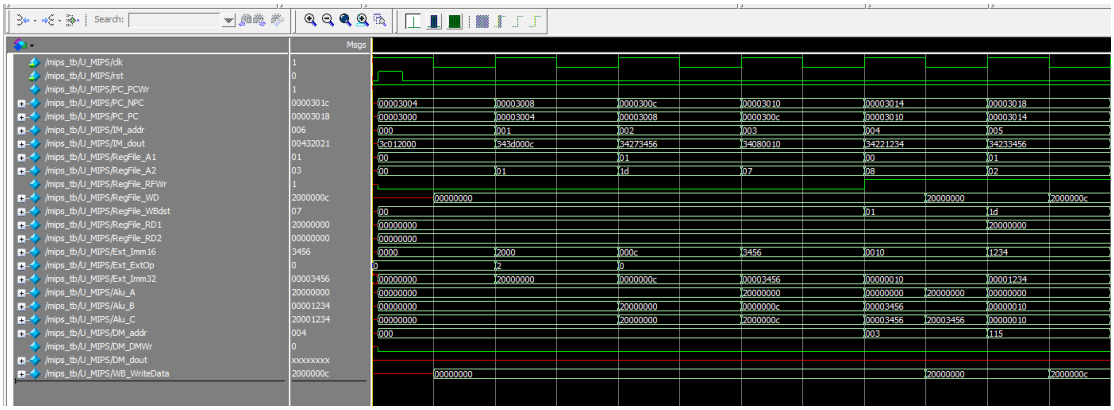
在第二周期，该指令进入 RegFile 和 EXT 中，在 RegFile 中，该指令读取了零号寄存器的值 0，在 EXT 中，该指令采用低位置零的扩展，可以看到 16 位立即数$(2000)_{16}$被扩展为了 32 为立即数$(20000000)_{16}$，保存到第二级流水线中。

在第三周期指令处于 EXE 级，指令在 Alu 中将 AluA 设置为 0 号寄存器的值 0，将 AluB 设置为扩展后的立即数$(20000000)_{16}$，执行加法运算得到 AluC 的值为$(20000000)_{16}$，传入第三级流水线寄存器；

在第四个周期 MEM 级，指令不执行读写操作；

在第五周期 WB 级的时钟下降沿，指令选择了$(1d)_{16}$作为目标寄存器将$(20000000)_{16}$写入寄存器堆中，这样就完成了 lui $1,0x2000 的功能。

5.3.2 ori $29, $1, 12



上图中后五个周期为第 2 条指令 ori $29, $1, 12 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003004)_{16}$，同时指令存储器读取到该地址的指令为$(343d000c)_{16}$，存入第一级流水线寄存器当中。

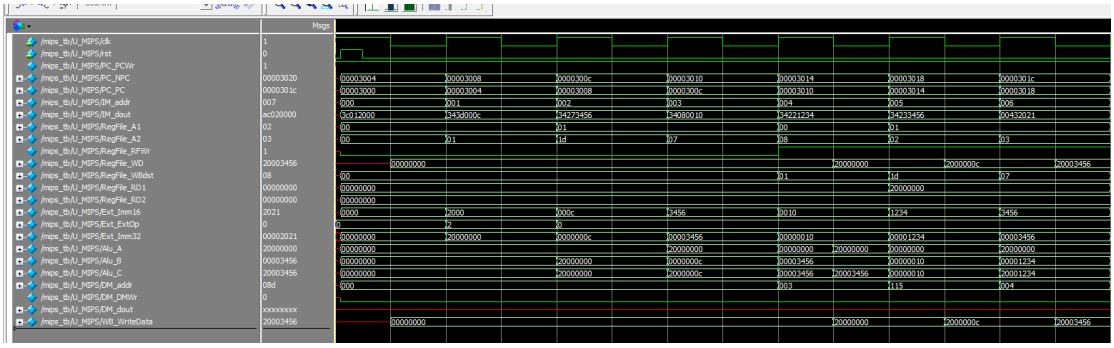
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 1 号寄存器的值为$(00000000)_{16}$，在 EXT 中将立即数扩展为$(0000000c)_{16}$，同时在 Ctrl 中进行相关性检测，发现相关发生（1 号寄存器虽在上一条指令被使用，并在本条指令作为数据寄存器使用），故设置相关信号以使用旁路运算。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 MEM 级旁路的值$(20000000)_{16}$作为操作数 A，选取立即数扩展后的值(0000000c)作为操作数 B，执行按位或运算后，得出结果$(2000000c)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值写回寄存器堆，并使用 29 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.3 ori $7, $1, 0x3456



上图中后五个周期为第 3 条指令 ori $7, $1, 0x3456 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003008)_{16}$，同时指令存储器读取到该地址的指令为$(34273456)_{16}$，存入第一级流水线寄存器当中。

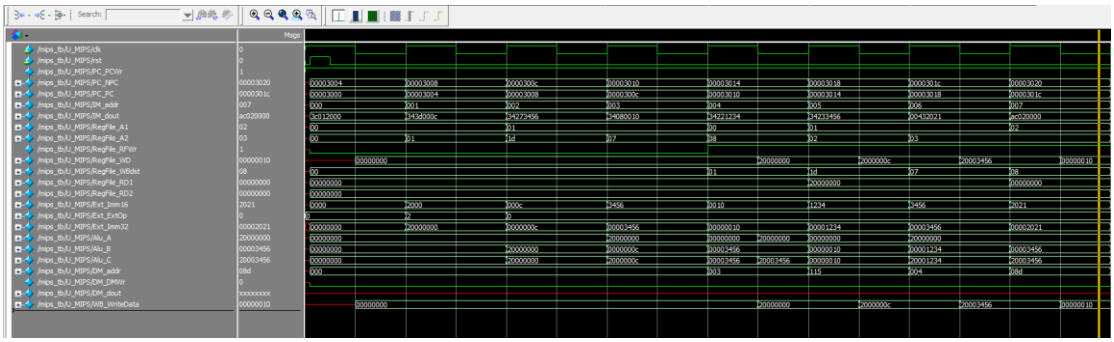
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 1 号寄存器的值为$(00000000)_{16}$，在 EXT 中将立即数扩展为$(00003456)_{16}$，同时在 Ctrl 中进行相关性检测，发现相关发生（1 号寄存器虽在上上条指令被使用，并在本条指令作为数据寄存器使用），故设置相关信号以使用旁路运算。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 WB 级旁路的值$(20000000)_{16}$作为操作数 A，选取立即数扩展后的值(00003456 作为操作数 B，执行按位或运算后，得出结果$(20003456)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值$(20003456)_{16}$写回寄存器堆，并使用 7 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.4 ori $8, $0, 0x10



上图中后五个周期为第 4 条指令 ori $8, $0, 0x10 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(0000300c)_{16}$，同时指令存储器读取到该地址的指令为$(34080010)_{16}$，存入第一级流水线寄存器当中。

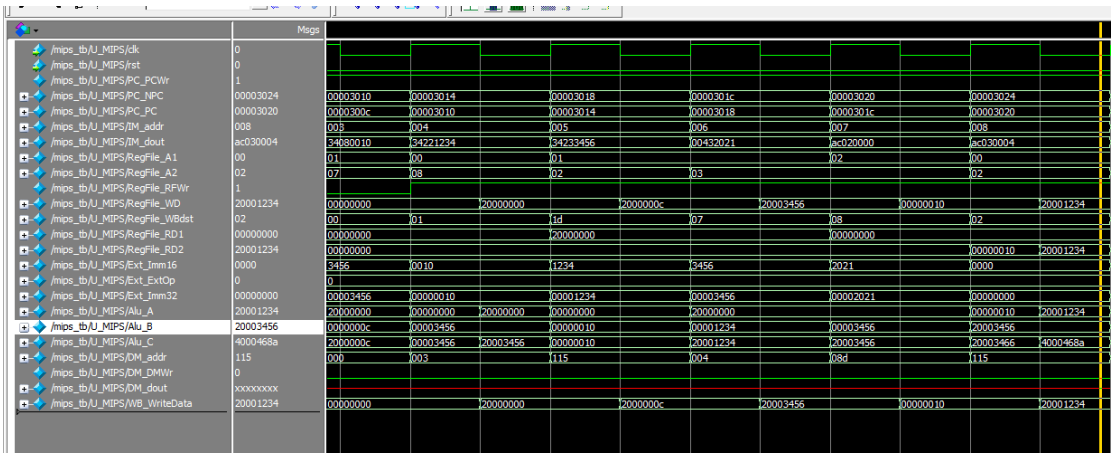
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为$(00000000)_{16}$，在 EXT 中将立即数扩展为$(00000010)_{16}$，同时在 Ctrl 中进行相关性检测，没有发现数据冒险发生。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA 的值$(00000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00000010)_{16}$作为操作数 B，执行按位或运算后，得出结果$(00000010)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值$(00000010)_{16}$写回寄存器堆，并使用 8 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.5 ori $2, $1, 0x1234



上图中后五个周期为第 5 条指令 ori $2, $1, 0x1234 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003010)_{16}$，同时指令存储器读取到该地址的指令为$(34221234)_{16}$，存入第一级流水线寄存器当中。

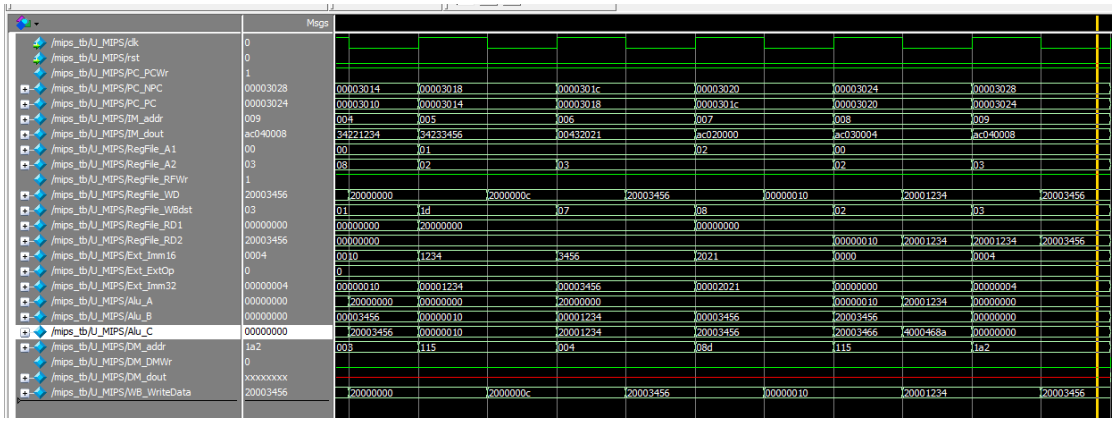
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 1 号寄存器的值为$(20000000)_{16}$，在 EXT 中将立即数扩展为$(00001234)_{16}$，同时在 Ctrl 中进行相关性检测，未发现相关发生。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA 的值$(20000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00001234)_{16}$作为操作数 B，执行按位或运算后，得出结果$(20001234)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值$(20001234)_{16}$写回寄存器堆，并使用 2 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.6 ori $3, $1, 0x3456



上图中后五个周期为第 6 条指令 ori $3, $1, 0x3456 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003014)_{16}$，同时指令存储器读取到该地址的指令为$(34233456)_{16}$，存入第一级流水线寄存器当中。

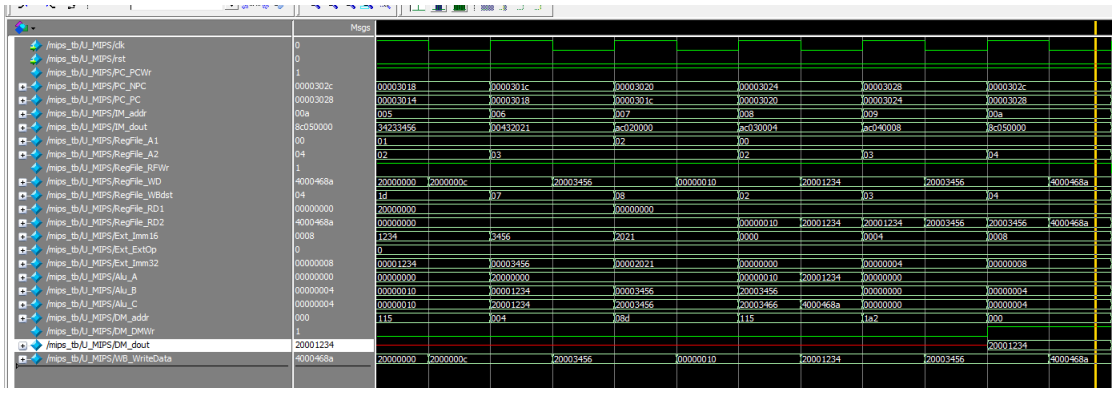
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 1 号寄存器的值为$(20000000)_{16}$，在 EXT 中将立即数扩展为$(00003456)_{16}$，同时在 Ctrl 中进行相关性检测，未发现相关发生。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA 的值$(20000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00003456)_{16}$作为操作数 B，执行按位或运算后，得出结果$(20001234)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值$(20003456)_{16}$写回寄存器堆，并使用 2 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.7 addu $4, $2, $3



上图中后五个周期为第 7 条指令 addu $4, $2, $3 的执行过程。

在第一个周期中，PC 指向下一条指令地址为(00003018)$_{16}$，同时指令存储器读取到该地址的指令为(00432021)$_{16}$，存入第一级流水线寄存器当中。

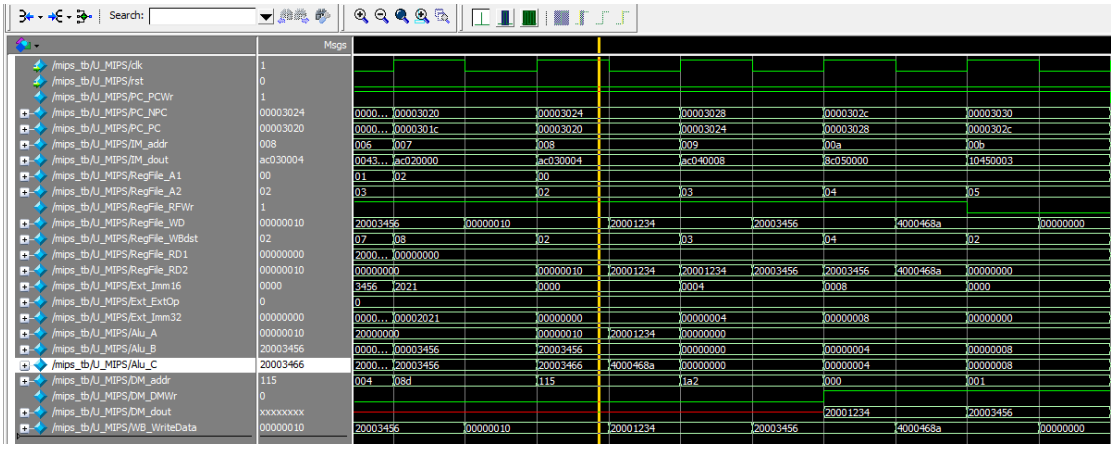
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 2 号寄存器的值为(00000000)$_{16}$，读取 3 号寄存器的值为(00000000)$_{16}$，同时在 Ctrl 中进行相关性检测，发现 rs 与 MEM 级的 ori $2, $1, 0x1234 指令的目的寄存器数据相关，rt 与 EXE 级的 ori $3, $1, 0x3456 指令的目的寄存器数据相关，设置相关信号设置旁路。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 WB 级的旁路(20001234)$_{16}$作为操作数 A，选取立即数扩展后的值(20003456)$_{16}$作为操作数 B，执行按位或运算后，得出结果(4000468a)$_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值(4000468a)$_{16}$写回寄存器堆，并使用 4 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.8 sw $2, 0($0)



上图中后五个周期为第 8 条指令 sw $2, 0($0)的执行过程。

在第一个周期中，PC 指向下一条指令地址为(0000301c)$_{16}$，同时指令存储器读取到该地址的指令为(ac020000)$_{16}$，存入第一级流水线寄存器当中。

在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为(00000000)$_{16}$，读取 2 号寄存器的值为(20001234)$_{16}$，在 EXT 中将 16 位立即数(0000)$_{16}$扩展为 32 位立即数(00000000)$_{16}$，同时在 Ctrl 中进行相关性检测，未发现产生数据相关。

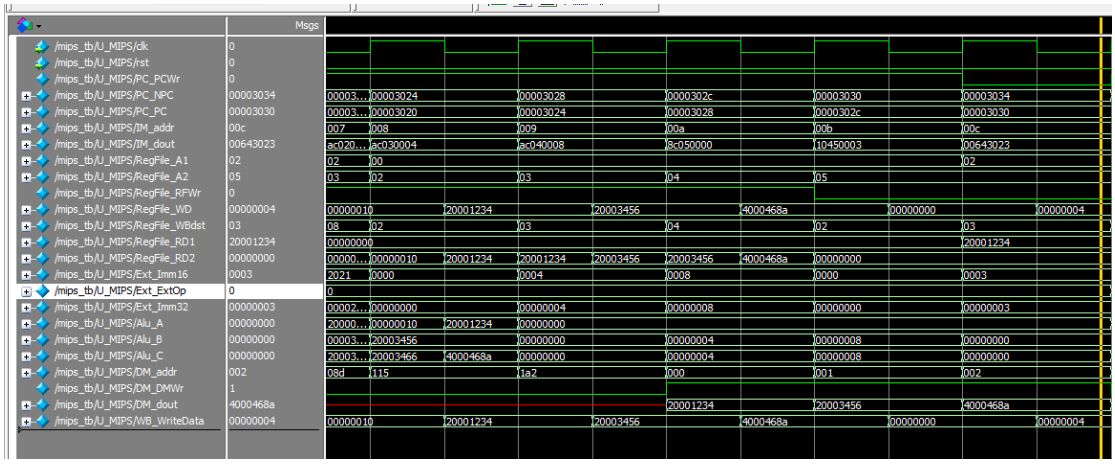
在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA(00000000)$_{16}$作为操作数 A，选取立即数扩展后的值(00000000)$_{16}$作为操作数 B，执行加法运算后，得出了结果

$(00000000)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果$(00000000)_{16}$作为 Mem 的地址，将 RegFileB 的值$(20001234)_{16}$作为写入数据，观察到 MemW 值为 1，成功写入 RAM。

在最后一个周期中，指令在 WB 级，观察到 RegW 值为 0，不写入寄存器堆。这样就完成了本条指令。

5.3.9 sw $3, 4($0)



上图中后五个周期为第 9 条指令 sw $3, 4($0)的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003020)_{16}$，同时指令存储器读取到该地址的指令为$(ac030004)_{16}$，存入第一级流水线寄存器当中。

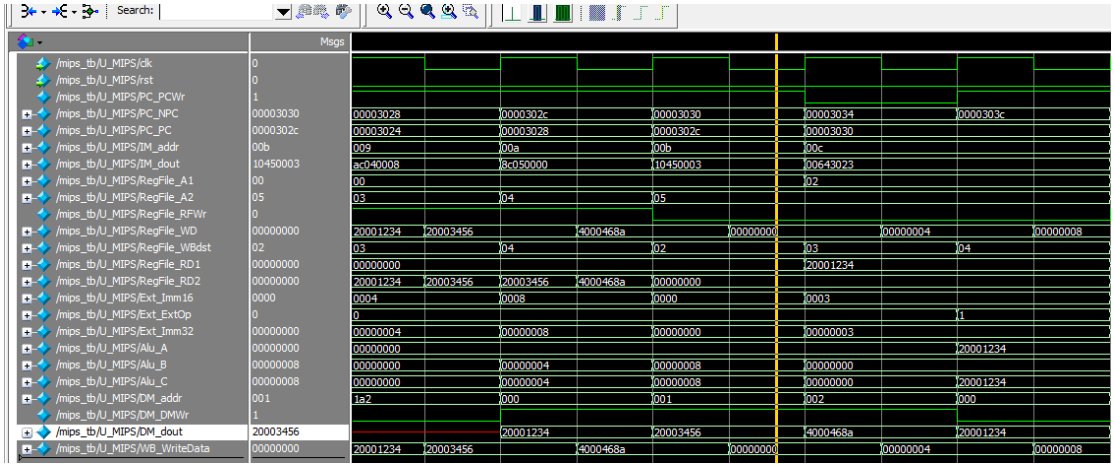
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为$(00000000)_{16}$，读取 3 号寄存器的值为$(20003456)_{16}$，在 EXT 中将 16 位立即数$(0004)_{16}$扩展为 32 位立即数$(00000004)_{16}$，同时在 Ctrl 中进行相关性检测，未发现产生数据相关。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA$(00000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00000004)_{16}$作为操作数 B，执行加法运算后，得出了结果$(00000004)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果$(00000004)_{16}$作为 Mem 的地址，将 RegFileB 的值$(20003456)_{16}$作为写入数据，观察到 MemW 值为 1，成功写入 RAM。

在最后一个周期中，指令在 WB 级，观察到 RegW 值为 0，不写入寄存器堆。这样就完成了本条指令。

5.3.10 sw $4, 8($0)



上图中后五个周期为第 10 条指令 sw $4, 8($0)的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003024)_{16}$，同时指令存储器读取到该地址的指令为$(ac04008)_{16}$，存入第一级流水线寄存器当中。

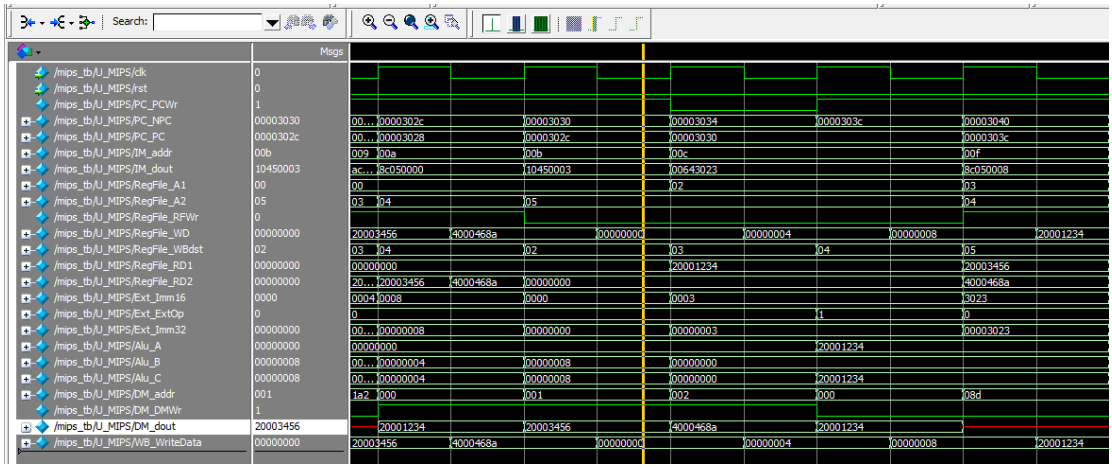
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为$(00000000)_{16}$，读取 4 号寄存器的值为$(4000468a)_{16}$，在 EXT 中将 16 位立即数$(0008)_{16}$扩展为 32 位立即数$(00000008)_{16}$，同时在 Ctrl 中进行相关性检测，未发现产生数据相关。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA$(00000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00000008)_{16}$作为操作数 B，执行加法运算后，得出了结果$(00000008)_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果$(00000008)_{16}$作为 Mem 的地址，将 RegFileB 的值$(4000468a)_{16}$作为写入数据，观察到 MemW 值为 1，成功写入 RAM。

在最后一个周期中，指令在 WB 级，观察到 RegW 值为 0，不写入寄存器堆。这样就完成了本条指令。

5.3.11 lw $5, 0($0)



上图中后五个周期为第 11 条指令 lw $5, 0($0)的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003028)_{16}$，同时指令存储器读取到该地址的指令为$(8c050000)_{16}$，存入第一级流水线寄存器当中。

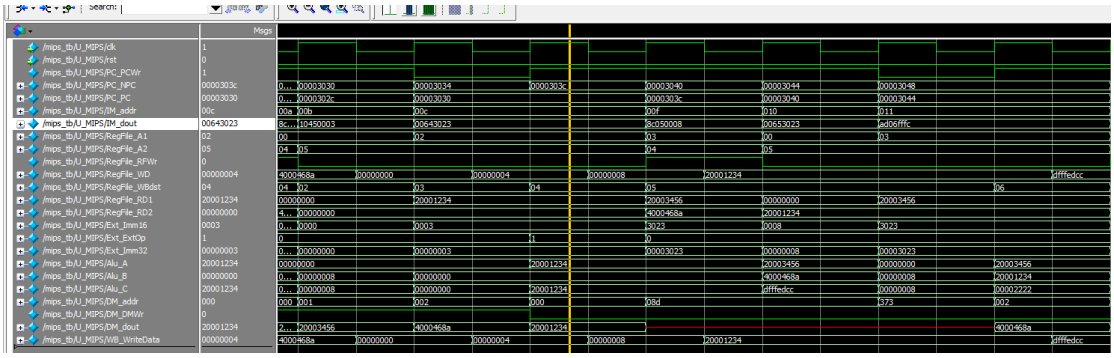
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为$(00000000)_{16}$，在 EXT 中将 16 位立即数$(0000)_{16}$扩展为 32 位立即数$(00000000)_{16}$，同时在 Ctrl 中进行相关性检测，未发现产生数据相关。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA$(00000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00000000)_{16}$作为操作数 B，得出了结果$(00000000)_{16}$。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果$(000000000)_{16}$作为 Mem 的地址，0 观察到 MemW 值为 0，dout 值为$(20001234)_{16}$，读出数据到流水线寄存器。

在最后一个周期中，指令在 WB 级，RegW 值为 1，选择 RAM 结果 $(20001234)_{16}$作为结果存入目标寄存器 5 号寄存器，这样就完成了本条 lw 指令。

5.3.12 beq $2, $5, _lb2



上图中后七个周期为行第 12 条指令 beq $2, $5, _lb2 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(0000302c)_{16}$，同时指令存储器读取到该地址的指令为$(10450003)_{16}$，存入第一级流水线寄存器当中。

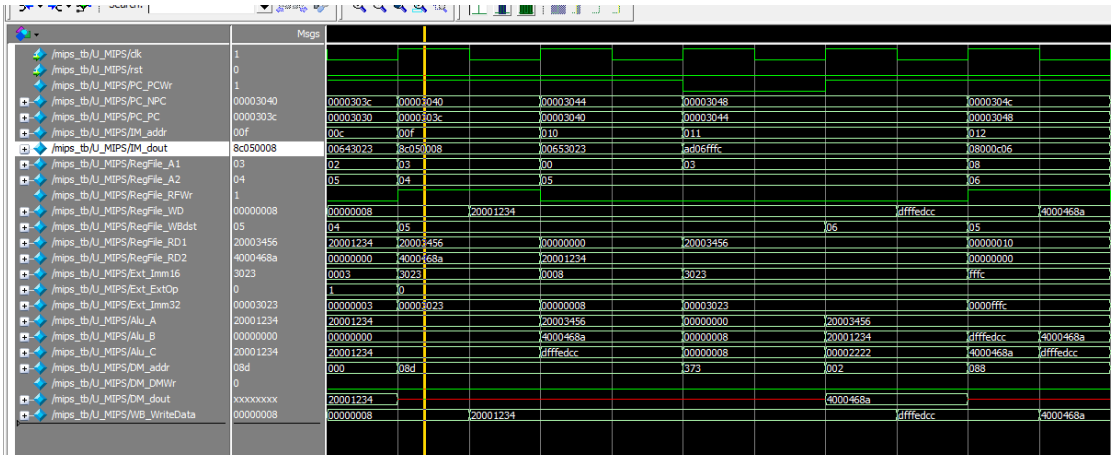
在第二个周期中，beq 指令流入 ID 级，在 RegFile 中，读取 2 号寄存器的值为$(00000000)_{16}$，读取 5 号寄存器的值为$(00000000)_{16}$，在 EXT 中将 16 位立即数$(0000)_{16}$扩展为 32 位立即数$(00000000)_{16}$，同时在 Ctrl 中进行相关性检测，发现上一条指令的目的寄存器与 rt 数据相关，且上条指令为 lw 指令，故设置相关信号，使得流水线阻塞一个周期。

在第三周期中，beq 指令被阻塞（重新执行），在 Ctrl 中从 WB 级拿到 rt 寄存器的值为$(20001234)_{16}$，与 rs 寄存器的值$(20001234)_{16}$ 对比发现需要跳转，将下条指令的 StopNext 信号设为 1，并将 NPC 的值设为跳转后的地址$(0000303c)_{16}$。同时，subu $6,$3,$4 进入流水线。

在第四周期中，subu $6,$3,$4 进入 Ctrl 控制器，发现上条指令为跳转指令且 stopthis 信号为 1，故设置改动 cpu 的相关信号全部为 0。

在第五、六、七周期中，cpu 状态均为改变，不做详述。这样就完成了一次跳转指令。

5.3.13 lw $5, 8($0)



上图中后五个周期为第 13 条指令 lw $5, 0($0)的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(0000303c)_{16}$，同时指令存储器读取到该地址的指令为$(8c050008)_{16}$，存入第一级流水线寄存器当中。

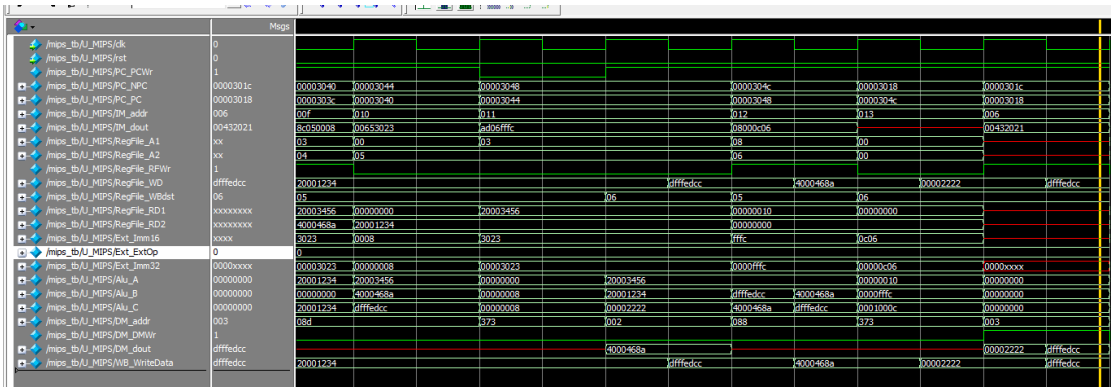
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 0 号寄存器的值为$(00000000)_{16}$，在 EXT 中将 16 位立即数$(0008)_{16}$扩展为 32 位立即数$(00000008)_{16}$，同时在 Ctrl 中进行相关性检测，未发现产生数据相关。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA$(00000000)_{16}$作为操作数 A，选取立即数扩展后的值$(00000008)_{16}$作为操作数 B，得出了结果$(00000008)_{16}$。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果$(000000008)_{16}$作为 Mem 的地址，0 观察到 MemW 值为 0，dout 值为$(4000468a)_{16}$，读出数据到流水线寄存器。

在最后一个周期中，指令在 WB 级，RegW 值为 1，选择 RAM 结果 $(4000468a)_{16}$作为结果存入目标寄存器 5 号寄存器，这样就完成了本条 lw 指令。

5.3.14 subu $6, $3, $5



上图中后五个周期为第 14 条指令 subu $6, $3, $5 的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003040)_{16}$，同时指令存储器读取到该地址的指令为$(00653023)_{16}$，存入第一级流水线寄存器当中。

在第二个周期中，指令流入 ID 级，在 RegFile 中，在 Ctrl 中进行相关性检测，发现 rt 与 EXE 级的 lw $5, 0($0)指令的目的寄存器数据相关，故阻塞一次流水线，设置了相关信号。

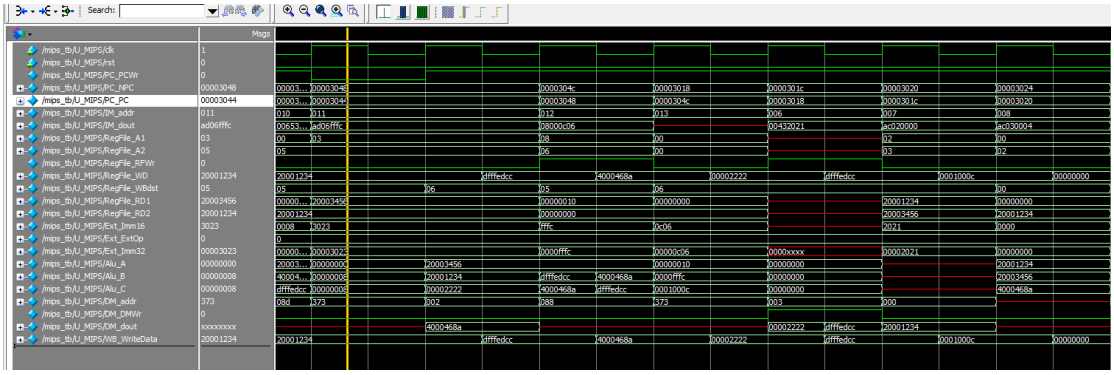
在第三个周期中，因流水线被阻塞，指令再次进入 ID 级，读取 2 号寄存器的值为$(20003456)_{16}$，读取 3 号寄存器的值为$(20001234)_{16}$，同时在 Ctrl 控制器中检测相关性，发现 MEM 级的 lw $5, 0($0)指令的目的寄存器数据相关，可以使用旁路解决，即设置相关信号。

在第四个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA 的值$(20001234)_{16}$作为操作数 A，选取 WB 级旁路的值$(4000468a)_{16}$作为操作数 B，执行减法运算后，得出结果$(dfffedcc)_{16}$，保存在流水线寄存器中。

在第五个周期中，指令流入 MEM 级，不产生操作。

在最后一个周期中，指令在 WB 级，选择 AluC 的值$(dfffedcc)_{16}$写回寄存器堆，并使用 6 号寄存器作为目标寄存器。这样就完成了这条指令的功能。

5.3.15 sw $6, -4($8)



上图中后五个周期为第 15 条指令 sw $6, -4($8)的执行过程。

在第一个周期中，PC 指向下一条指令地址为$(00003044)_{16}$，同时指令存储器读取到该地址的指令为$(ac020000)_{16}$，存入第一级流水线寄存器当中。

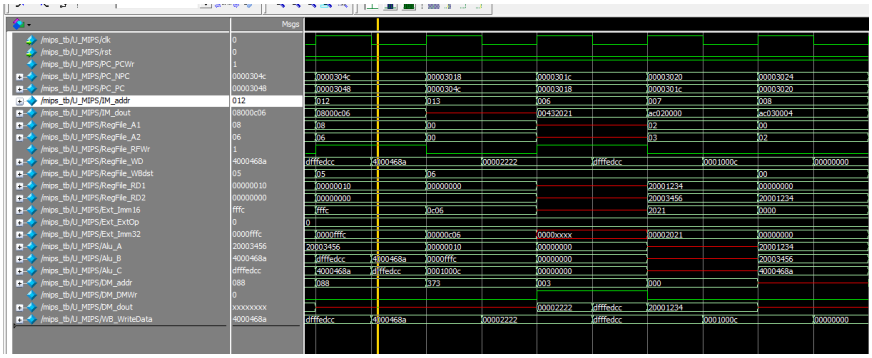
在第二个周期中，指令流入 ID 级，在 RegFile 中，读取 8 号寄存器的值为$(20003456)_{16}$，读取 6 号寄存器的值为$(20001234)_{16}$，在 EXT 中将 16 位立即数$(fffc)_{16}$扩展为 32 位立即数$(fffffffc)_{16}$，同时在 Ctrl 中进行相关性检测，发现与 EXE 级的目的寄存器产生数据相关，设置相关信号。

在第三个周期中，指令流入 EXE 级，在 Alu 中，选取 RegFileA(20003456)$_{16}$作为操作数 A，选取立即数扩展后的值(fffffffc)$_{16}$作为操作数 B，执行加法运算后，得出了地址结果(20003452)$_{16}$，保存在流水线寄存器中。

在第四个周期中，指令流入 MEM 级，将 AluC 的结果(20003452$_{16}$作为 Mem 的地址，将 WB 级旁路的值(dfffedcc)$_{16}$作为写入数据，观察到 MemW 值为 1，成功写入 RAM。

在最后一个周期中，指令在 WB 级，观察到 RegW 值为 0，不写入寄存器堆。这样就完成了本条指令。

5.3.16 j f0



上图中后五个周期为行第 16 条指令 j f0 的执行过程。

在第一个周期中，PC 指向下一条指令地址为(00003048)$_{16}$，同时指令存储器读取到该地址的指令为(08000c06)$_{16}$，存入第一级流水线寄存器当中。

在第二个周期中，j 指令流入 ID 级，在 Ctrl 中，设置 NPC 的值为(00003018)$_{16}$，同时废除下一条指令改变 cpu 的功能。

在第三、四、五周期中，cpu 状态均为改变，不做详述。完成了一次跳转指令。

第六章　课程设计总结

本次设计实验中，设计者根据实验资料设计了基于 MIPS 指令集的流水线 CPU。设计过程中，采用了模块化的设计方法，首先着眼于单个模块的功能实现，接着考虑各个模块间的协同工作，再考虑流水线工作中可能遇到的问题，最终通过一系列的测试初步确定了一个较为完整的设计方案。

本次实验也有部分创新：在传统的 MIPS CPU 中，冒险只分为数据冒险和控制冒险，则在处理 lw-sw 型冒险时，将产生一次冒泡，但事实上 sw 指令可以直接使用 WB 级的旁路数据进行寸数操作而无需冒泡。本次设计在这一点上做出改进，设计将额外增加 lw-sw 型冒险的判断，以提高 CPU 的性能。

当然，本次设计还存在以下问题：

1. 测试文件太少，导致部分可能的数据冒险没有被检测处来；再进一步的检测中，我们可以人为增加更多可能的冒险，保证设计的稳定性；
2. 本次实验的设计只包含了精简后的 MIPS 指令集，不能完成一个可以正常工作的 CPU 所需要的所有指令，在以后更进一步的设计中可以进行补充；
3. 模块化的代码可维护型不高。由于再设计的一开始就设计了功能模块，导致模块不能很好的配合工作，部分模块冗余，部分模块过分庞杂，在实际设计中应使用更成熟的设计方法；
4. 由于对传统的 MIPS CPU 设计了解不够，在一些旁路选择的位置设计上不完善，导致冒险控制单元较为复杂，在进一步改进中应予以更正。
5. ModelSim 只完成了 Verilog 设计的功能仿真而非时序仿真，也就是说在真正的时序仿真环境或物理环境下，这次设计将完全无法使用或是下载到电路中。但本次设计的初衷是学习 CPU 的设计理念，故这个错误在许可范围内。

通过本次课程设计，进一步学习了 CPU 的部分原理，为以后进一步的学习打下基础。