编号_____

南京航空航天大学

实 验 报 告

题　目　流水线 MIPS CPU 的设计（36 条）

学生姓名 _____

学　　号 _____

学　　院　计算机科学与技术学院

专　　业　计算机科学与技术专业

班　　级 _____

指导教师 _____

二〇一九年七月

南京航空航天大学
实验报告诚信承诺书

本人郑重声明：所呈交的毕业设计（论文）（题目：流水线MIPS CPU 的设计（36 条））是本人在导师的指导下独立进行研究所取得的成果。尽本人所知，除了毕业设计（论文）中特别加以标注引用的内容外，本毕业设计（论文）不包含任何其他个人或集体已经发表或撰写的成果作品。

作者签名：　　2019 年 7 月　日

（学号）：

流水线 MIPS CPU 的设计（36 条）

摘　要

通过实验来学习流水线 CPU 的工作原理和基于 Verilog 的硬件描述语言来设计 CPU 的方法，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，同时获得 CPU 设计与仿真的实践和经验。通过对知识的综合应用，加深对 CPU 系统各模块的工作原理及相互联系。

研究的主要结果：实现了一个 MIPS 架构，能执行 addu,subu,slt,addiu,beq,jump 等共 36 条指令的流水线 CPU，支持除异常和中断以外的各类冒险处理，但不支持溢出处理。

关键词：流水线 cpu, MIPS, Verilog, 仿真

The Design of a Single Cycle CPU in MIPS Instruction Set

Abstract

Through experiments to learn the working principle of pipeline CPU and the method of designing CPU based on Verilog hardware description language, master the technology of debugging and simulation using Modelsim simulation technology, cultivate the independent working ability of scientific research and the ability of analyzing and solving problems, and acquire the practice and experience of CPU design and simulation. Through the comprehensive application of knowledge, the working principle of each module of CPU system is deepened and interrelated.

The main result of the study: The main result of the study: a pipeline CPU with MIPS architecture capable of executing a total of 36 instructions such as addu, subu, slt, addiu, beq, jump, etc., It supports various types of risk handling in addition to exceptions and interrupts, but does not support overflow handling.

Key Words：Pipeline CPU，MIPS，Verilog，simulation

目 录

第一章　流水线理论结构与信号取值

1.1 流水线 CPU 理论结构

流水线 CPU 的理论数据通路结构如图所示。（引自《计算机组成与设计：硬件/软件接口》David Patterson）

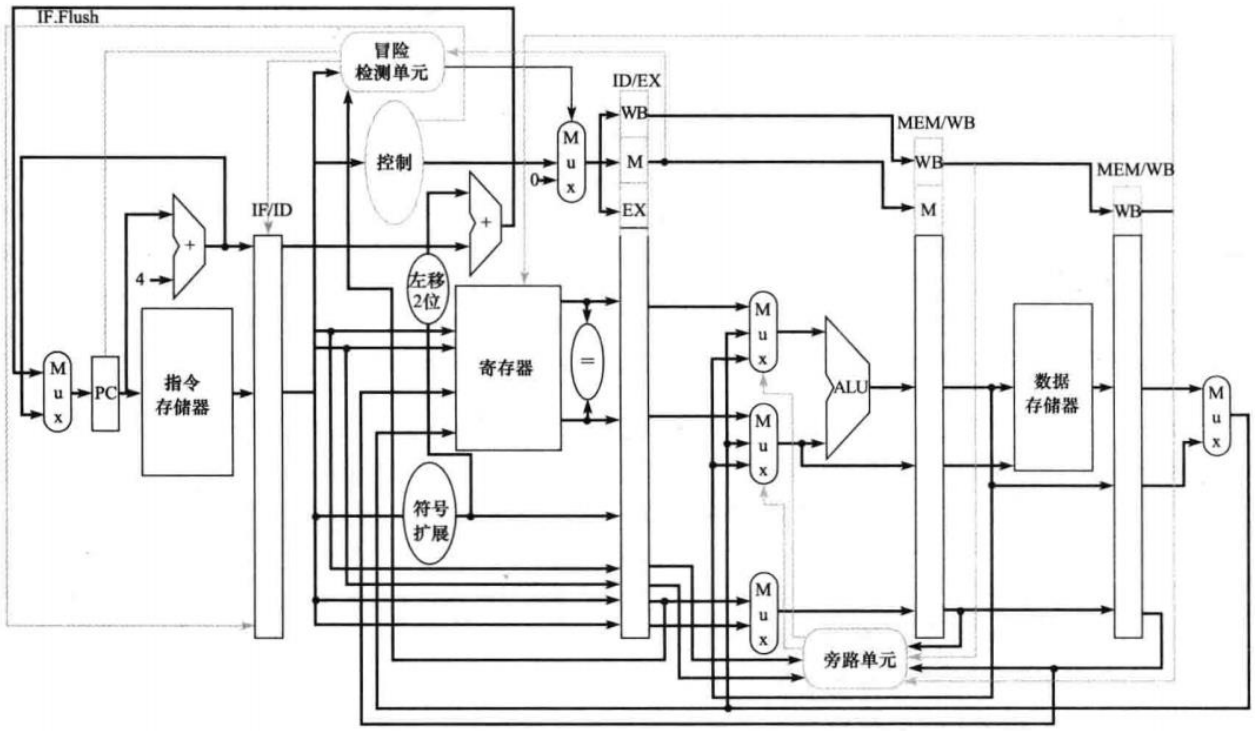

图 1.1 流水线处理器的数据通路

文件结构如下：

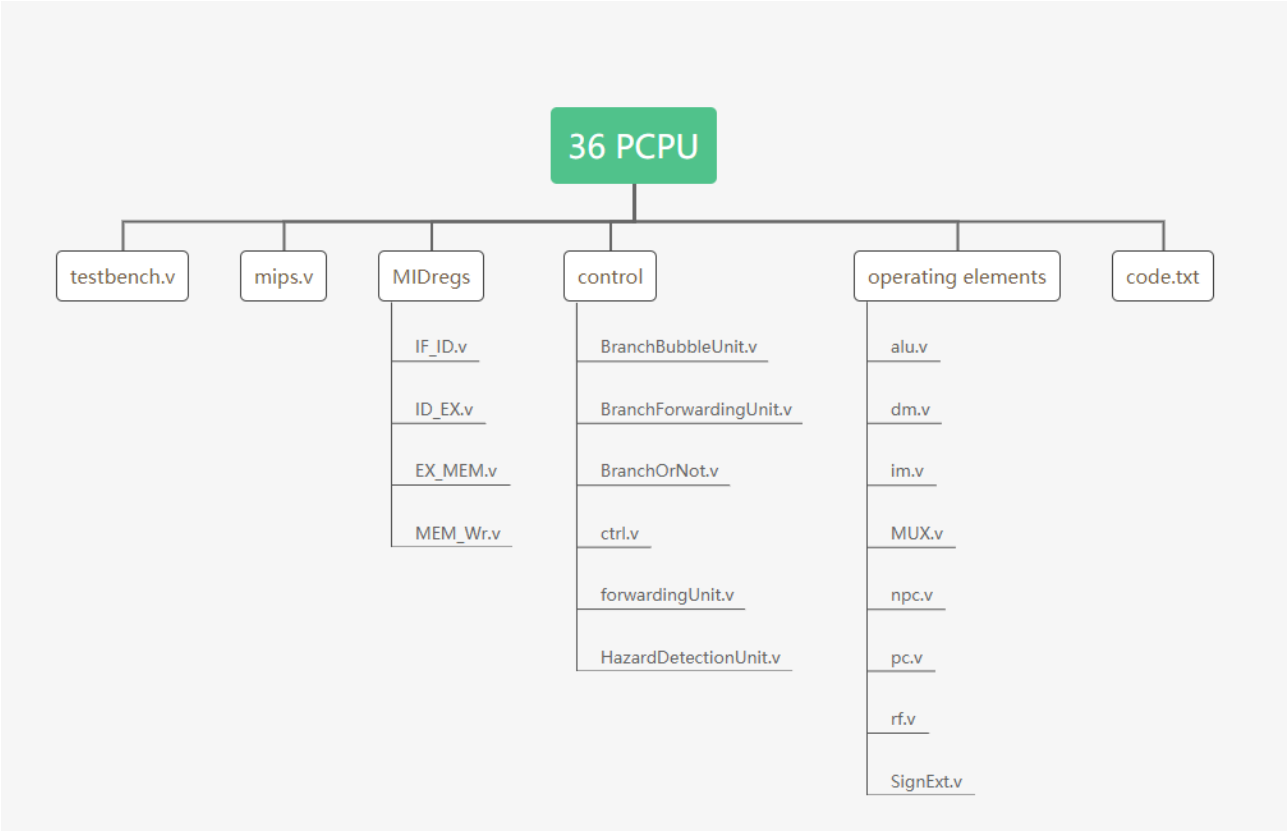

图 1.2 流水线处理器的文件结构

1.1.1 mips 模块定义

(1) 基本功能

作为整个 CPU 的最上级模块，整合了数据通路，操作元件和控制器，相当于单周期 CPU 中的 datapath。

（2）模块接口

表1.1　mips 接口的模块定义

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号 1：复位 0：无效

（3）功能定义

表1.2　mips 接口的功能定义

序号	功能名称	功能描述
1	复位	rst 有效时复位整个 CPU
2	连接	将各个操作元件和控制器连接起来成为完整数据通路

1.1.2 pc 模块定义

（1）基本功能

PC 模块输出当前指令在 im_4k 中的地址，在 rst 信号为真时复位，并且在发生冒险时进行相应操作。定义第一条指令地址为 0000_3000H 来对接 MARS 生成的机器码。

（2）模块接口

表 1.3 pc 模块接口定义

信号名	方向	描述
NPC[31:2]	I	下一条指令的地址
Reset	I	复位信号
Clk	I	时钟信号
hazard	I	load-use 冒险控制信号
BranchBubble	I	插入气泡控制信号
PC[31:2]	O	当前指令的地址

（3）功能定义

表 1.4 pc 接口的功能定义

序号	功能名称	功能描述
1	输出指令地址	根据 NPC 输出当前 PC 所指的指令地址
2	复位	将 PC 重置为 0000_3000H

1.1.3 im_4k 模块定义

（1）基本功能

存储测试指令并根据 PC 模块给出的地址将其取出。

（2）模块接口

表 1.5 im_4k 模块接口定义

信号名	方向	描述
addr[11:2]	I	PC 表示的指令地址
dout[31:0]	O	输出指令

（3）功能定义

表 1.6 im_4k 接口的功能定义

序号	功能名称	功能描述
1	存储指令	将测试指令存在内置的 im[1023:0]中
2	取出指令	根据 PC 值输出指令

1.1.4 rf 模块定义

（1）基本功能

作为寄存器存储函数运行时的数据，根据两个输入的地址输出寄存器的值，根据写信号和写入地址对寄存器进行写入。

（2）模块接口

表 1.7 rf 模块接口定义

信号名	方向	描述
addr[31:0]	I	数据存储器的写入地址，用于 lb,lbu 指令
WrEn	I	写入寄存器的控制信号
busW[31:0]	I	写入数据流
Rw[4:0]	I	数据写入地址
Ra[4:0]	I	输出数据地址 A
Rb[4:0]	I	输出数据地址 B
clk	I	时钟信号

PC[31:0]	I	当前指令，用于实现 jal，jalr
REGSop[2:0]	I	控制信号： 001：lb　010：lbu　011：jal　100：jalr
R31Wr	I	控制是否对 regs[31]进行写入
R31[31:2]	I	regs[31]写入数据流
busA[31:0]	O	输出信息 A
busB[31:0]	O	输出信息 B

（3）功能定义

表 1.8 rf 接口的功能定义

序号	功能名称	功能描述
1	存数	将 busW 的数据存入 Rw 指定的寄存器中，或将 busW 的特定字节扩展后存入 Rw 指定的寄存器中
2	取数	取出 Ra,Rb 指定的寄存器中的数

1.1.5 SignExt 模块定义

(1) 基本功能

分别实现了将 5 位，16 位立即数扩展到 WIDTH 位的扩展器，由 ExtOp 控制 16 位扩展器是否为有符号扩展。

(2) 模块接口

表 1.9 SignExt 模块接口定义

信号名	方向	描述
a[4/15:0]	I	待扩展数据
ExtOp	I	控制是否为有符号扩展
b[WIDTH - 1:0]	O	扩展后数据

（3）功能定义

表 1.10 SignExt 接口的功能定义

序号	功能名称	功能描述
1	无符号扩展	将输入的 a 无符号扩展为 width 位 b
2	有符号扩展	将输入的 a 有符号扩展为 width 位 b

1.1.6 alu 模块定义

（1）基本功能

通过 ALUctr 控制实现基本的运算操作，如 addu, subu, slt, and, nor, or, xor, sll 等。

（2）模块接口

表 1.11 alu 模块接口定义

信号名	方向	描述
ALUctr[3:0]	I	用于控制 alu 进行运算 0000:addu 0001:subu 0010:slt 0011:and 0100:nor 00101:or 0110:xor 0111:sll 1000:srl 1001:sltu 1010:sra 1011:lui
A[31:0]	I	数据输入通路 A
B[31:0]	I	数据输入通路 B
Result[31:0]	O	运算结果
Zero	O	结果为 0 时 Zero 为 1

（3）功能定义

表 1.12 alu 接口的功能定义

序号	功能名称	功能描述
1	输出运算结果	根据 ALUctr 来输出运算的结果
2	输出 Zero	Zero = Result == 0

1.1.7 dm_4k 模块定义

（1）基本功能

根据输入的地址读出数据，并根据写使能信号和地址写入数据。

(2) 模块接口

表 1.13 dm_4k 模块接口定义

信号名	方向	描述
addr[11:0]	I	输入/输出地址
din[31:0]	I	输入数据流
we[1:0]	I	写使能信号
clk	I	时钟信号
memRead[1:0]	I	根据 memRead 执行不同读取数据操作
dout[31:0]	0	数据输出流

（3）功能定义

表 1.14 dm_4k 接口的功能定义

序号	功能名称	功能描述
1	存入数据	将数据存入 dm_4k 中
2	取出数据	将数据从 dm_4k 中取出

1.1.8 NPC 模块定义

（1）基本功能

根据 PC 和各控制信号得出 nextPC 并送回 PC.

（2）模块接口

表 1.15 NPC 模块接口定义

信号名	方向	描述
PC_plus_4[31:2]	I	PC+4
PC_br[31:2]	I	branch 跳转地址
PC_jump01[31:2]	I	j， jal 跳转地址
PC_jump10[31:2]	I	jr, jalr 跳转地址

Branch_ok	I	是否进行 branch 跳转
Jump[1:0]	I	区别 J JAL JR JALR
NPC[31:2]	O	输出 NextPC

（3）功能定义

表 1.16 NPC 接口的功能定义

序号	功能名称	功能描述
1	输出下一个 pc	输出 NextPC

1.1.9 mux 模块定义

（1）基本功能

实现了 2 选 1 或 3 选 1 的默认 WIDTH 为 32 位的多选一模块

（2）模块接口

表 1.17 mux 模块接口定义

信号名	方向	描述
a[WIDTH-1:0]	I	输入 1
b[WIDTH-1:0]	I	输入 2
c[WIDTH-1:0]	I	输入 3
ctr[0/1:0]	I	选择信号，选择一个输入作为输出
y[WIDTH-1:0]	O	输出

（3）功能定义

表 1.18 mux 接口的功能定义

序号	功能名称	功能描述
1	多选 1	根据 flag 选择 a，b，c 中的一个输出

1.1.10 ctrl 模块定义

(1) 基本功能

实现整个CPU的控制器,接收指令中的op段、func段和Rt段,得到Branch, Jump, RegDst, ALUsrc, MemtoReg, RegWr, MemWr, MemRead, ExtOp, Rtype, ALUctr, ALUshf, R31Wr 这些

控制信号。

（2）模块接口

表 1.19　ctrl 接口的模块定义

信号名	方向	描述
op[5:0]	I	指令的 op 段，用于生成控制信号
func[5:0]	I	指令的 func 段，用于生成控制信号
Rb[4:0]	I	指令的 Rt 段，用于生成控制信号
RegWr	O	用于控制寄存器可否被写入
ALUsrc	O	用于控制 ALU 是否接收立即数
RegDst	O	用于控制寄存器写入地址
MemtoReg	O	用于控制数据存储器能否输出数据到寄存器
MemWr[1:0]	O	用于控制数据存储器可否被写入
Branch[2:0]	O	表示是否为分支指令
Jump[1:0]	O	表示是否为无条件跳转指令
ExtOp	O	用于控制扩展器是否为符号扩展
ALUctr[3:0]	O	控制 ALU 功能
ALUshf	O	判断 ALU 是否进行移位
MemRead[1:0]	O	控制数据存储器可否被读取
R31Wr	O	控制 31 号寄存器 Regs[31]可否被写入

（3）功能模块

表 1.20　ctrl 接口的功能定义

序号	功能名称	功能描述
1	输出控制信号	根据传入的 op，func，Rb,生成各种控制信号并输出

1.1.11 BranchBubbleUnit 模块定义

（1）基本功能

判断 branch 类型和 jr，jalr 指令是否与前面指令发生冒险，如果发生则在指令后插入气泡。

(2) 模块接口

表 1.21 BranchBubbleUnit 模块接口定义

信号名	方向	描述
ID_Ra[4:0]	I	ID 段读取目标地址 A
ID_Rb[4:0]	I	ID 段读取目标地址 B
EX_Rw[4:0]	I	Ex 段写入目标地址 Rw
MEM_Rw[4:0]	I	Mem 段写入目标地址 Rw
Ex_RegWr	I	Ex 段用于控制寄存器可否被写入
Mem_RegWr	I	Mem 段用于控制寄存器可否被写入
Mem_MemtoReg	I	Mem 段用于控制寄存器可否被读出到数据存储器
ID_Branch[2:0]	I	ID 段表示是否为分支指令
ID_Jump[1:0]	I	ID 段表示是否为无条件跳转指令
BranchBubble	O	分支指令冒险信号

（3）功能定义

表 1.22 BranchBubbleUnit 的功能定义

序号	功能名称	功能描述
1	插入气泡	判断是否出现冒险，出现则插入气泡

1.1.12 BranchForwardingUnit 模块定义

(1) 基本功能

是 Branch 语句的转发控制单元，通过接入数据来得到 BranchForwardA，BranchForwardB，让分支判断时能够正确选择数据来源以完成 Branch 指令的提前执行

(2) 模块接口

表 1.23 BranchForwardingUnit 模块接口定义

信号名	方向	描述
ID_Ra[4:0]	I	ID 段读取目标地址 Ra
ID_Rb[4:0]	I	ID 段读取目标地址 Rb
MEM_Rw[4:0]	I	Mem 段写入目标地址 Rw
Mem_MemtoReg	I	Mem 段用于控制数据存储器可否写回寄存器
Mem_RegWr	I	Mem 段用于控制寄存器可否被写入
BranchForwardA	O	输出 Branch 比较操作数 A 的选择信号
BranchForwardB	O	输出 Branch 比较操作数 B 的选择信号

（3）功能定义

表 1.24 BranchForwardingUnit 的功能定义

序号	功能名称	功能描述
1	输出选择信号	输出 BranchForwardA 和 BranchForwardB 来选择比较指令操作数

1.1.13 branchOrNot 模块定义

(1) 基本功能

用于判断 Branch 指令是否真的发生跳转，用以实现 Branch 指令的提前执行

(2) 模块接口

表 1.25 branchOrNot 模块接口定义

信号名	方向	描述
busA[31:0]	I	Branch 指令读取数据流 A
busB[31:0]	I	Branch 指令读取数据流 B
Branch[2:0]	I	Branch 指令控制信号

Branch_flag	0	输出该条 Branch 指令跳转与否

（3）功能定义

表 1.26 branchOrNot 的功能定义

序号	功能名称	功能描述
1	输出 flag	输出 Branch_flag 表示跳转与否

1.1.14 forwardingUnit 模块定义

(1) 基本功能

转发检测单元，用于向 EX 段输出正确的选择信号让 ALU 可以选择正确的数据进行计算。

(2) 模块接口

表 1.27 forwardingUnit 模块接口定义

信号名	方向	描述
EX_Ra[4:0]	I	EX 段读取目标地址 a
EX_Rb[4:0]	I	EX 段读取目标地址 b
MEM_Rw[4:0]	I	MEM 段写回地址 Rw
Wr_Rw[4:0]	I	Wr 段写回地址 Rw
MEM_RegWr	I	MEM 段控制是否写入寄存器
Wr_RegWr	I	Wr 段控制是否写回寄存器
forwardA[1:0]	O	ALU 第一操作数 A 的选择信号
forwardB[1:0]	O	ALU 第二操作数 B 的选择信号

（3）功能定义

表 1.28 forwardingUnit 的功能定义

序号	功能名称	功能描述

1	输出选择信号	输出 ALU 操作数选择信号 forwardA 和 forwardB

1.1.15 HazardDetectionUnit 模块定义

(1) 基本功能

检测是否发生 load-use 冒险

(2) 模块接口

表 1.29 HazardDetectionUnit 模块接口定义

信号名	方向	描述
Ex_MemtoReg	I	Ex 段控制数据存储器写入寄存器
Ex_Rw[4:0]	I	Ex 段寄存器写入地址
ID_Ra[4:0]	I	ID 段寄存器读取地址 A
ID_Rb[4:0]	I	ID 段寄存器读取地址 B
hazard	O	提示是否出现 load-use 冒险

（3）功能定义

表 1.30 HazardDetectionUnit 的功能定义

序号	功能名称	功能描述
1	输出 hazard	输出 hazard 表示出现 load-use 冒险与否

1.1.16 IF_ID 模块定义

(1) 基本功能

IF 至 ID 段流水寄存器

(2) 模块接口

表 1.31 IF_ID 模块接口定义

信号名	方向	描述
Clk	I	时钟信号
Reset	I	清零信号

ID_Jump[1:0]	I	ID 段跳转信号
Branch_ok	I	ID 段寄存器读取地址 B
hazard	I	load-use 冒险信号
IF_ins[31:0]	I	IF 段读出指令
B_PC[31:2]	I	IF 段 B_PC
PC[31:2]	I	IF 段 PC
BranchBubble	I	插入气泡信号
ID_ins[31:0]	O	ID 段指令
ID_B_PC[31:2]	O	ID 段 B_PC
ID_PC[31:2]	O	ID 段 PC

（3）功能定义

表 1.32 IF_ID 的功能定义

序号	功能名称	功能描述
1	流水寄存	输出 ID 段所需信息

1.1.17 ID_EX 模块定义

(1) 基本功能

ID 至 EX 段流水寄存器

(2) 模块接口

表 1.33 ID_EX 模块接口定义

信号名	方向	描述
Clk	I	时钟信号
hazard	I	load-use 冒险信号
BranchBubble	I	插入气泡信号
ID_RegWr	I	ID 段寄存器写指令

ID_RegDst	I	ID 段寄存器选择输出指令
ID_ALUsrc	I	ID 段 ALU 选择输入指令
ID_MemtoReg	I	ID 段内存写入寄存器指令
ID_ALUshf	I	ID 段 ALU 移位指令
ID_busA[31:0]	I	ID 段寄存器输出 A
ID_busB[31:0]	I	ID 段寄存器输出 B
ID_imm16Ext[31:0]	I	ID 段立即数扩展
ID_ra[4:0]	I	ID 段寄存器读地址 A
ID_rb[4:0]	I	ID 段寄存器读地址 B
ID_rw[4:0]	I	ID 段寄存器写地址
ID_shf[4:0]	I	ID 段译码移位量
ID_rd[4:0]	I	ID 位指令写地址
ID_MemWr[1:0]	I	ID 段内存写指令
ID_MemRead[1:0]	I	ID 段内存读指令
ID_ALUctr[3:0]	I	ID 段 ALU 操作符
ID_PC[31：2]	I	ID 段 PC
EX_Ra[4:0]	O	EX 段 ra
EX_Rb[4:0]	O	EX 段 rb
EX_Rw[4:0]	O	EX 段寄存器写地址
Ex_Rd[4:0]	O	EX 段 rd
Ex_shf[4:0]	O	EX 段译码移位量
Ex_busA[31:0	O	EX 段 ALU 输入通路

]		
Ex_BusB[31:0]	0	EX 段 ALU 输入通路
Ex_imm16Ext[31:0]	0	EX 段立即数扩展
Ex_RegWr	0	EX 段寄存器写指令
Ex_RegDst	0	EX 段寄存器选择输出指令
Ex_ALUsrc	0	EX 段 ALU 选择输入指令
Ex_MemtoReg	0	EX 段内存写入寄存器指令
Ex_ALUshf	0	EX 段 ALU 移位指令
Ex_MemWr[1:0]	0	EX 段内存写入指令
Ex_MemRead[1:0]	0	EX 段内存读取指令
Ex_ALUctr[3:0]	0	EX 段 ALU 操作符
Ex_PC[31:2]	0	EX 段 PC

（3）功能定义

表 1.34 ID_EX 的功能定义

序号	功能名称	功能描述
1	流水寄存	输出 EX 段所需信息

1.1.18 EX_MEM 模块定义

(1) 基本功能

EX 至 MEM 段流水寄存器

(2) 模块接口

表 1.35 EX_MEM 模块接口定义

信号名	方向	描述
Clk	I	时钟信号
Ex_Zero	I	EX 段 ALU 结果为 0 信号
Ex_RegWr	I	EX 段寄存器写指令
EX_memtoReg	I	EX 段内存写入寄存器指令
Ex_alu_result[31:0]	I	EX 段 ALU 运算结果
EX_busB[31:0]	I	EX 段 ALU 输入 busB
EX_busA_mux3[31:0]	I	EX 段 ALU 输入 busA
Ex_Rw[4:0]	I	EX 段寄存器写入地址
Ex_cs[4:0]	I	EX 段目标地址 Rd
EX_MemWr[1:0]	I	EX 段内存写入指令
Ex_MemRead[1:0]	I	Ex 段内存读取指令
Ex_PC[31:2]	I	EX 段 PC
Mem_Zero	O	MEM 段 ALU 结果为 0 信号
Mem_RegWr	O	MEM 段寄存器写指令
Mem_MemtoReg	O	MEM 段内存写入寄存器指令
Mem_alu_result[31:0]	O	MEM 段 ALU 运算结果
Mem_busB[31:0]	O	MEM 段 ALU 输入 busB
Mem_busA[31:0]	O	MEM 段 ALU 输入 busA
Mem_Rw[4:0]	O	MEM 段寄存器写入地址

Mem_cs[4:0]	O	MEM 段目标地址 Rd
Mem_MemWr[1:0]	O	MEM 段内存写入指令
Mem_MemRead[1:0]	O	MEM 段内存读取指令
Mem_PC[31:2]	O	MEM 段 PC

（3）功能定义

表 1.36 EX_MEM 的功能定义

序号	功能名称	功能描述
1	流水寄存	输出 MEM 段所需信息

1.1.19 MEM_WR 模块定义

(1) 基本功能

MEM 至 WR 段流水寄存器

(2) 模块接口

表 1.37 MEM_WR 模块接口定义

信号名	方向	描述
Clk	I	时钟信号
Mem_RegWr	I	Mem 段寄存器写指令
Mem_MemtoReg	I	Mem 段内存写入寄存器指令
Mem_dout[31:0]	I	Mem 段内存输出
MEM_alu_result[31:0]	I	Mem 段 ALU 运算结果
MEM_busA[31:0]	I	Mem 段 ALU 输入 A

MEM_busB [31:0]	I	Mem 段 ALU 输入 B
MEM_Rw[4:0]	I	Mem 段寄存器写入地址
MEM_rd[4:0]	I	Mem 段目标地址
MEM_PC[31:2]	I	Mem 段 PC
Wr_RegWr	O	Wr 段寄存器写指令
Wr_MemtoReg	O	Wr 段内存写入寄存器指令
Wr_dout[31:0]	O	Wr 段内存输出
Wr_alu_result[31:0]	O	Wr 段 ALU 运算结果
Wr_busA[31:0]	O	Wr 段 ALU 输入 busA
Wr_busB[31:0]	O	Wr 段 ALU 输入 busB
Wr_Rw[4:0]	O	Wr 段寄存器写入地址
Wr_Rd[4:0]	O	Wr 段目标地址 Rd
Wr_PC[31:2]	O	Wr 段 PC

（3）功能定义

表 1.38 MEM_WR 的功能定义

序号	功能名称	功能描述
1	流水寄存	输出 WR 段所需信息

1.2 各个指令与控制信号之间的关系

ins name	Branch	Jump	RegDst	ALUsrc	MemtoReg	RegWr	MemWr	ExtOp	ins Type
addu	000	00	1	0	0	1	00	0	R
subu	000	00	1	0	0	1	00	0	R
slt	000	00	1	0	0	1	00	0	R
and	000	00	1	0	0	1	00	0	R
nor	000	00	1	0	0	1	00	0	R
or	000	00	1	0	0	1	00	0	R
xor	000	00	1	0	0	1	00	0	R
sll	000	00	1	0	0	1	00	0	R
srl	000	00	1	0	0	1	00	0	R
sltu	000	00	1	0	0	1	00	0	R
jalr	000	10	1	0	0	1	00	0	R
jr	000	10	1	0	0	1	00	0	R
sllv	000	00	1	0	0	1	00	0	R
sra	000	00	1	0	0	1	00	0	R
srav	000	00	1	0	0	1	00	0	R
srlv	000	00	1	0	0	1	00	0	R
addiu	000	00	0	1	0	1	00	1	I
beq	001	00	x	0	x	0	00	1	I
bne	010	00	x	0	x	0	00	1	I
lw	000	00	0	1	1	1	00	1	I
sw	000	00	x	1	x	0	01	1	I
lui	000	00	0	1	0	1	00	0	I
slti	000	00	0	1	0	1	00	1	I
sltiu	000	00	0	1	0	1	00	1	I
bgez	011	00	x	0	x	0	00	1	I
bgtz	100	00	x	0	x	0	00	1	I
blez	101	00	x	0	x	0	00	1	I
bltz	110	00	x	0	x	0	00	1	I
lb	000	00	0	1	1	1	00	1	I
lbu	000	00	0	1	1	1	00	1	I
sb	000	00	0	1	0	0	10	1	I
andi	000	00	0	1	0	1	00	0	I
ori	000	00	0	1	0	1	00	0	I
xori	000	00	0	1	0	1	00	0	I
j	000	01	x	x	x	0	00	0	J
jal	000	01	x	x	0	1	00	0	J

图 1.3 36CPU 控制信号真值表

ins name	(31-26)op	(25-21)rs	(20-16)rt	(15-11)rd	(10-6)shamt	func	ALUctr	R31Wr	ALUshf	MemRead
addu	000000	RS	RT	RD	00000	100001	0000	0	0	00
subu	000000	RS	RT	RD	00000	100011	0001	0	0	00
slt	000000	RS	RT	RD	00000	101010	0010	0	0	00
and	000000	RS	RT	RD	00000	100100	0011	0	0	00
nor	000000	RS	RT	RD	00000	100111	0100	0	0	00
or	000000	RS	RT	RD	00000	100101	0101	0	0	00
xor	000000	RS	RT	RD	00000	100110	0110	0	0	00
sll	000000	RS	RT	RD	shamt	000000	0111	0	1	00
srl	000000	RS	RT	RD	shamt	000010	1000	0	1	00
sltu	000000	RS	RT	RD	00000	101011	1001	0	0	00
jalr	000000	RS	RT	RD	00000	001001	0000	1	0	00
jr	000000	RS	RT	RD	00000	001000	0000	0	0	00
sllv	000000	RS	RT	RD	00000	000100	0111	0	0	00
sra	000000	RS	RT	RD	shamt	000011	1010	0	1	00
srav	000000	RS	RT	RD	00000	000111	1010	0	0	00
srlv	000000	RS	RT	RD	00000	000110	1000	0	0	00
addiu	001001	RS	RT	imm16			0000	0	0	00
beq	000100	RS	RT	offset			0000	0	0	00
bne	000101	RS	RT	offset			0000	0	0	00
lw	100011	BASE	RT	offset			0000	0	0	01
sw	101011	BASE	RT	offset			0000	0	0	00
lui	001111	00000	RT	imm16			1011	0	0	00
slti	001010	RS	RT	imm16			0010	0	0	00
sltiu	001011	RS	RT	imm16			1001	0	0	00
bgez	000001	RS	00001	offset			0000	0	0	00
bgtz	000111	RS	00000	offset			0000	0	0	00
blez	000110	RS	00000	offset			0000	0	0	00
bltz	000001	RS	00000	offset			0000	0	0	00
lb	100000	BASE	RT	offset			0000	0	0	10
lbu	100100	BASE	RT	offset			0000	0	0	11
sb	101000	BASE	RT	offset			0000	0	0	00
andi	001100	RS	RT	imm16			0011	0	0	00
ori	001101	RS	RT	imm16			0101	0	0	00
xori	001110	RS	RT	imm16			0110	0	0	00
j	000010	target					0000	0	0	00
jal	000011	target					0000	0	0	00

图 1.4 36CPU 控制信号真值表（续）

注释：各个信号的取值过程如下：

```
if (op == 6'b000011 || (op == 6'b000000 && func == 6'b001001))
        R31Wr = 1;//000011jal,0010001jalr
    else R31Wr = 0;
    case (op)
      6'b000000:begin//jr and jalr
        if (func == 6'b001000 || func == 6'b001001)
          {ALUctr, Branch, RegDst, ALUsrc, MemtoReg, RegWr, MemWr, ExtOp, MemRead, ALUshf,
Jump}
          <= 19'b10;
        else //another R-type
```

```
    begin
      {Branch, Jump, ALUsrc, MemtoReg, MemWr, ExtOp, MemRead, RegDst, RegWr}
      <= 14'b11;
      ALUshf = (func == 6'b000000) || (func == 6'b000010) || (func == 6'b000011);
      case (func)
        6'b100001:ALUctr = 4'b0000; //addu
        6'b100011:ALUctr = 4'b0001; //subu
        6'b101010:ALUctr = 4'b0010; //slt
        6'b100100:ALUctr = 4'b0011; //and
        6'b100111:ALUctr = 4'b0100; //nor
        6'b100101:ALUctr = 4'b0101; //or
        6'b100110:ALUctr = 4'b0110; //xor
        6'b101011:ALUctr = 4'b1001; //sltu
        6'b000100:ALUctr = 4'b0111; //sllv
        6'b000111:ALUctr = 4'b1010; //srav
        6'b000110:ALUctr = 4'b1000; //srlv
        6'b000000:ALUctr = 4'b0111; //sll
        6'b000010:ALUctr = 4'b1000; //srl
        6'b000011:ALUctr = 4'b1010; //sra
        6'b001000:ALUctr = 4'b0000; //jr
        6'b001001:ALUctr = 4'b0000; //jalr
        default:ALUctr = 4'b0000;   //default
      endcase
    end//end another R-type
  end//end R-type
  6'b001001:begin//addiu
    {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, ExtOp, RegWr}
    <= 15'b111;
    ALUctr = 4'b0000;
```

```
end
6'b001111:begin//lui
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ExtOp, ALUsrc, RegWr}
   <= 15'b11;
   ALUctr = 4'b1011;
end
6'b001010:begin//slti
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, ExtOp}
   <= 15'b111;
   ALUctr = 4'b0010;
end
6'b001011:begin//sltiu
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, ExtOp}
   <= 15'b111;
   ALUctr = 4'b1001;
end
6'b001100:begin//andi
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ExtOp, ALUsrc, RegWr}
   <= 15'b11;
   ALUctr = 4'b0011;
end
6'b001101:begin//ori
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ExtOp, ALUsrc, RegWr}
   <= 15'b11;
   ALUctr = 4'b0101;
end
6'b001110:begin//xori
   {Branch, Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ExtOp, ALUsrc, RegWr}
   <= 15'b11;
```

```
    ALUctr = 4'b0110;
end
6'b000100:begin//beq
    {Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, Branch, ExtOp}
    <= 15'b11;
    ALUctr = 4'b0000;
end
6'b000101:begin//bne
    {Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, Branch, ExtOp}
    <= 15'b101;
    ALUctr = 4'b0000;
end
6'b000001:begin//bltz==bgez=6'b000001
    if (Rb == 5'b00001)          Branch = 3'b011;//bgez
    else if (Rb == 5'b00000) Branch = 3'b110;//blez
    {Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, ExtOp}
    <= 12'b101;
    ALUctr = 4'b0000;
end
6'b000111:begin//bgtz
    {Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, Branch, ExtOp}
    <= 15'b1001;
    ALUctr = 4'b0000;
end
6'b000110:begin//blez
    {Jump, RegDst, MemtoReg, MemWr, MemRead, ALUshf, ALUsrc, RegWr, Branch, ExtOp}
    <= 15'b1011;
    ALUctr = 4'b0000;
end
```

```
6'b100011:begin//lw
   {Jump, RegDst, MemWr, ALUshf, Branch, ALUsrc, MemtoReg, RegWr, ExtOp, MemRead}
   <= 15'b111101;
   ALUctr = 4'b0000;
end
6'b101011:begin//sw
   {Jump, RegDst, ALUshf, Branch, MemtoReg, RegWr, MemRead, ALUsrc, ExtOp, MemWr}
   <= 15'b1101;
   ALUctr = 4'b0000;
end
6'b100000:begin//lb
   {Jump, RegDst, MemWr, ALUshf, Branch, ALUsrc, MemtoReg, RegWr, ExtOp, MemRead}
   <= 15'b111110;
   ALUctr = 4'b0000;
end
6'b100100:begin//lbu
   {Jump, RegDst, MemWr, ALUshf, Branch, ALUsrc, MemtoReg, RegWr, ExtOp, MemRead}
   <= 15'b111111;
   ALUctr = 4'b0000;
end
6'b101000:begin//sb
   {Jump, RegDst, ALUshf, Branch, MemtoReg, RegWr, MemRead, ALUsrc, ExtOp, MemWr}
   <= 15'b1110;
   ALUctr = 4'b0000;
end
6'b000010:begin//j
   {RegDst, ALUshf, Branch, MemtoReg, RegWr, MemRead, ALUsrc, ExtOp, MemWr, Jump}
   <= 15'b1;
   ALUctr = 4'b0000;
```

```
        end
        6'b000011:begin//jal
            {RegDst, ALUshf, Branch, MemtoReg, RegWr, MemRead, ALUsrc, ExtOp, MemWr, Jump}
            <= 15'b1;
            ALUctr = 4'b0000;
        end
        default:begin
            {RegDst, ALUshf, Branch, MemtoReg, RegWr, MemRead, ALUsrc, ExtOp, MemWr, Jump}
            <= 15'b0;
            ALUctr = 4'b0000;
        end
    endcase
```

第二章 流水线 MIPS CPU 的具体设计与调试

2.1 具体源代码实现

由于源代码较多 仅选择部分粘贴！

2.1.1 Mips

```
module mips(clk,    rst);
  //DEFINE
  input   clk ;      // clock
  input   rst ;// reset
  ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
  //IF DEFINE

  wire[31:2] NPC, PC, PC_plus_4, B_PC;
  wire[31:0] IF_ins;
  wire Branch_ok, BranchBubble, R31Wr;
  ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
  //ID DEFINE

   wire[31:0] ID_ins, ID_busA,    ID_busA_mux2, ID_busB,    ID_busB_mux2;
     wire[31:2] ID_B_PC;
     wire[4:0] ID_Ra, ID_Rb, ID_Rw, ID_shf;
     wire[31:0] ID_imm16Ext;
     wire[15:0] ID_imm16;
  //CONTROLL SIGN
  wire ID_RegWr, ID_RegDst, ID_ExtOp, ID_ALUsrc;
  wire[2:0] ID_Branch;
  wire[1:0] ID_Jump, ID_MemWr, ID_MemRead;
  wire ID_MemtoReg, ID_ALUshf;
  wire[3:0] ID_ALUctr;

  wire[4:0] ID_rd;
  wire[31:2] ID_PC;
  ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
  //EX DEFINE

  wire[31:0] Ex_busB_mux3, Ex_alu_result, Ex_busA, Ex_busB;//Ex
  wire Ex_Zero;
  wire[4:0] Ex_Ra, Ex_Rb, Ex_Rw, Ex_Rw_mux2, Ex_shf, Ex_rd;
  wire[31:0] Ex_shfExt, Ex_imm16Ext, Ex_busA_mux3, Ex_alu_busA, Ex_alu_busB;
  wire Ex_RegWr, Ex_RegDst, Ex_ALUsrc, Ex_MemtoReg, Ex_ALUshf;
  wire[1:0] Ex_MemWr, Ex_MemRead;
```

```
   wire[3:0] Ex_ALUctr;
   wire[31:2] Ex_PC;
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
      //MEM DEFINE

   wire[31:0] Mem_dout, Mem_alu_result, Mem_busB, Mem_busA;
   wire[4:0] Mem_Rw, Mem_rd;
   wire Mem_RegWr, Mem_MemtoReg, Mem_Zero;
   wire[1:0] Mem_MemWr, Mem_MemRead;
   wire[31:2] Mem_PC;
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
      //WR DEFINE

   wire[31:0] Wr_dout, Wr_alu_result, Wr_busW, Wr_busA, Wr_busW_mux2_dout, Wr_busB;
   wire[4:0] Wr_Rw, Wr_rd;
   wire    Wr_RegWr, Wr_MemtoReg;
   wire[31:2] Wr_PC;
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
   //FORWARDING DEFINE

   wire[1:0] forwardA, forwardB;
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
   //HAZARD DEFINE

   wire hazard;
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
     //IF BLOCK

   pc pc(.NPC(NPC), .Clk(clk), .Reset(rst), .PC(PC), .hazard(hazard), .BranchBubble(BranchBubble));

   npc                    npc(.PC_plus_4(PC_plus_4),                    .PC_br(ID_B_PC[31:2]+
ID_imm16Ext[29:0]), .PC_jump01( {ID_B_PC[31:28], ID_ins[25:0]} ),
                  .PC_jump10(ID_busA_mux2[31:2]), .Branch_ok(Branch_ok), .Jump(ID_Jump), .NPC(
NPC));

    assign PC_plus_4 = PC + 1;
   assign B_PC = PC;
```

```
    im_4k im_4k( .addr(PC[31:2]), .dout(IF_ins));

    IF_ID IF_id(clk, rst, ID_Jump, Branch_ok, hazard, IF_ins, ID_ins, B_PC, ID_B_PC, BranchBubble, PC,
ID_PC);
    ////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
    //ID BLOCK

    assign ID_Ra = ID_ins[25:21];
    assign ID_Rb = ID_ins[20:16];
    assign ID_Rw = ID_ins[15:11];
    assign ID_imm16 = ID_ins[15:0];
    assign ID_shf = ID_ins[10:6];
    assign ID_rd = ID_ins[15:11];

    ctrl
ctrl(.clk(clk), .op(ID_ins[31:26]), .func(ID_ins[5:0]), .ALUctr(ID_ALUctr), .Branch(ID_Branch), .Jump(I
D_Jump), .RegDst(ID_RegDst),
                    .ALUsrc(ID_ALUsrc), .MemtoReg(ID_MemtoReg), .RegWr(ID_RegWr), .MemWr(ID
_MemWr), .ExtOp(ID_ExtOp), .MemRead(ID_MemRead), .ALUshf(ID_ALUshf),
                    .R31Wr(R31Wr), .Rb(ID_Rb));

    rf rf(.Clk(clk), .WrEn(Wr_RegWr), .Ra(ID_Ra), .Rb(ID_Rb), .Rw(Wr_Rw), .busW(Wr_busW),
            .busA(ID_busA), .busB(ID_busB), .R31Wr(R31Wr), .R31(ID_B_PC+30'b1)); //REGS[31] =
B_PC + 1

    SignExt16 SignExt16(.ExtOp(ID_ExtOp), .a(ID_imm16), .y(ID_imm16Ext));

    BranchForwardingUnit
BranchForwardingUnit(.ID_Ra(ID_Ra), .ID_Rb(ID_Rb), .Mem_Rw(Mem_Rw), .Mem_RegWr(Mem_R
egWr), .Mem_MemtoReg(Mem_MemtoReg),
                                        .BranchForwardA(BranchForwardA), .BranchForwardB(
BranchForwardB));

    MUX2 MUX2_ID_busA(ID_busA, Mem_alu_result, BranchForwardA, ID_busA_mux2);

    MUX2 MUX2_ID_busB(ID_busB, Mem_alu_result, BranchForwardB, ID_busB_mux2);

    branchOrNot BranchOrNot(ID_busA_mux2, ID_busB_mux2, ID_Branch, Branch_ok);

    BranchBubbleUnit
BranchBubbleUnit(.ID_Ra(ID_Ra), .ID_Rb(ID_Rb), .Ex_RegWr(Ex_RegWr), .Ex_Rw(Ex_Rw_mux2), .
Mem_RegWr(Mem_RegWr),
```

```
                                                .Mem_MemtoReg(Mem_MemtoReg), .Mem_Rw(M
em_Rw), .BranchBubble(BranchBubble),    .ID_Branch(ID_Branch),
                                                .ID_Jump(ID_Jump) );

  HazardDetectionUnit HazardDetectionUnit(Ex_MemtoReg & Ex_RegWr, Ex_Rw_mux2, ID_Ra,
ID_Rb, hazard);

  ID_Ex ID_ex(clk, hazard, BranchBubble, ID_busA, ID_busB, ID_Ra, ID_Rb, ID_Rw, ID_imm16Ext,
ID_RegWr, ID_RegDst, ID_ALUsrc,
                ID_MemWr, ID_MemtoReg, ID_ALUctr, ID_MemRead, ID_shf, ID_ALUshf,
Ex_busA, Ex_busB, Ex_Ra, Ex_Rb, Ex_Rw, Ex_imm16Ext,
                Ex_RegWr, Ex_RegDst, Ex_ALUsrc, Ex_MemWr, Ex_MemtoReg, Ex_ALUctr,
Ex_MemRead, Ex_shf, Ex_ALUshf, ID_rd, ID_PC, Ex_rd, Ex_PC);
  ////////////////////////////////////////////////////////////////////////////////////////////
////////////////
  //EX BLOCK

  MUX2 #(5) MUX2_RegDst(Ex_Rb, Ex_Rw, Ex_RegDst, Ex_Rw_mux2);

  forwardingUnit forwardingUnit(Ex_Ra, Ex_Rb, Mem_Rw, Mem_RegWr, Wr_Rw, Wr_RegWr,
forwardA, forwardB);

  SignExt5 SignExt5(.a(Ex_shf), .y(Ex_shfExt));

  MUX3   MUX3_alu_busA(Ex_busA, Wr_busW, Mem_alu_result, forwardA, Ex_busA_mux3);

  MUX2   MUX2_ALUshf_busA(Ex_busA_mux3, Ex_shfExt, Ex_ALUshf, Ex_alu_busA);

  MUX3   MUX3_alu_busB(Ex_busB, Wr_busW, Mem_alu_result, forwardB, Ex_busB_mux3);

  MUX2   MUX2_ALUsrc(Ex_busB_mux3, Ex_imm16Ext, Ex_ALUsrc, Ex_alu_busB);

  ALU ALU(Ex_alu_busA, Ex_alu_busB, Ex_ALUctr, Ex_Zero, Ex_alu_result);


  Ex_Mem Ex_mem(clk, Ex_Zero, Ex_alu_result, Ex_busB_mux3, Ex_Rw_mux2, Ex_RegWr,
Ex_MemWr, Ex_MemtoReg, Mem_Zero, Mem_alu_result,
                Mem_busB, Mem_Rw, Mem_RegWr, Mem_MemWr, Mem_MemtoReg,
Ex_MemRead, Mem_MemRead, Ex_busA_mux3,
                Mem_busA, Ex_rd, Ex_PC, Mem_rd, Mem_PC);
  ////////////////////////////////////////////////////////////////////////////////////////////
////////////////
  //MEM BLOCK
```

```
   dm_4k  dm_4k(.addr(Mem_alu_result[11:0]), .din(Mem_busB), .we(Mem_MemWr), .clk(clk), .dout(Mem_dout), .memRead(Mem_MemRead));

   Mem_Wr Mem_wr(clk, Mem_dout, Mem_alu_result, Mem_Rw, Mem_RegWr, Mem_MemtoReg,
Wr_dout, Wr_alu_result, Wr_Rw, Wr_RegWr, Wr_MemtoReg,
                              Mem_busA, Wr_busA,
                              Mem_rd, Mem_busB, Mem_PC, Wr_rd, Wr_busB, Wr_PC);
   ////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////
   //WR BLOCK

   MUX2   MUX2_Wr_dout(Wr_alu_result, Wr_dout, Wr_MemtoReg, Wr_busW_mux2_dout);

   assign Wr_busW = Wr_busW_mux2_dout;

endmodule
```

2.1.2 pc

略

2.1.3 im_4k

略

2.1.4 rf

```
module rf(Clk, WrEn, Ra, Rb, Rw, busW, busA, busB, R31Wr, R31);

   input Clk, WrEn, R31Wr;
   input[4:0] Ra, Rb, Rw;
   input[31:0] busW;
   input[31:2] R31;

   output[31:0] busA, busB;
   reg[31:0] regs[0:31];

   integer i;
   initial begin
      for (i = 0; i < 32; i = i + 1) regs[i] = 0;
   end
```

```
    always @(negedge Clk)   begin
        if(WrEn)    regs[Rw] <= busW;
        if(R31Wr) regs[31] <= {R31, 2'b0};
    end
    assign busA = (Ra != 5'd0) ? regs[Ra]: 32'd0;
    assign busB = (Rb != 5'd0) ? regs[Rb]: 32'd0;

endmodule
```

2.1.5 SignExt

```
module SignExt16 #(parameter WIDTH = 32) (ExtOp, a, y);

    input ExtOp;
    input[15:0] a;

    output[WIDTH - 1:0] y;

    assign y = (ExtOp == 0) ? {16'b0, a} : {{WIDTH - 16{a[15]}}, a};

endmodule //16 bits

module SignExt5 #(parameter WIDTH = 32) (a, y);

    input[4:0] a;

    output[WIDTH - 1:0] y;

    assign y = {28'b0, a};

endmodule //5 bits
```

2.1.6 alu

```
module ALU (A, B, ALUctr, Zero, Result);

    input[31:0] A,B;
    input[3:0] ALUctr;

    output Zero;
    output reg[31:0] Result;
```

```
    wire [31:0] temp = {1'b1, {31{1'b0}}};
    //the first place of temp is 1, rest is 0
    //it XORs with a or b to get the framshift
    //this allows us to use < to compare the size directly
    //because < can only compare unsigned numbers normally
    wire [31:0] SLTA = A ^ temp;
    wire [31:0] SLTB = B ^ temp;

    assign Zero = ((Result == 0)? 1 : 0);

    always@(*)
    begin
        case (ALUctr)
          4'b0000 : Result = A + B;    //ADDU
          4'b0001 : Result = A - B;    //SUBU
          4'b0010 : Result = (SLTA < SLTB) ? 1 : 0;    //SLT
          4'b0011 : Result = A & B;    //AND
          4'b0100 : Result = ~ ( A | B); //NOR
          4'b0101 : Result = A | B;    //OR
          4'b0110 : Result = A ^ B;    //XOR
          4'b0111 : Result = B << A; //SLL
          4'b1000 : Result = B >> A; //SRL
          4'b1001 : Result = (A < B) ? 1 : 0;    //SLTU
          4'b1010 : Result = (B[31] == 0) ? B >> A : ~((~B) >> A); //SRA
          4'b1011 : Result = {B[15:0], 16'd0}; //LUI
       endcase
    end

endmodule
```

2.1.7 dm_4k

```
module dm_4k(addr, din, we, clk, dout, memRead) ;
    input[11:0] addr;    // address bus    ,[11:2] -> [11:0]
        input[31:0] din;      // 32-bit input data
        input[1:0] we;        // memory write enable
        input clk;      // clock
        input[1:0] memRead;

    output reg[31:0] dout;    // 32-bit memory output

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

```
integer i;

initial begin
   for (i = 0; i < 1024; i = i + 1)
      dm[i] = 32'h0000_0000;
end

always@(negedge clk)//second half write
begin
   if (we == 2'b01 && memRead == 2'b00) dm[ addr[11:2] ] <= din; //SW
   else if (we == 2'b10 && memRead == 2'b00)       //SB
   begin
      if (addr[1:0] == 2'b00) dm[ addr[11:2] ][7:0] <= din[7:0];    else
      if (addr[1:0] == 2'b01) dm[ addr[11:2] ][15:8] <= din[7:0]; else
      if (addr[1:0] == 2'b10) dm[ addr[11:2] ][23:16] <= din[7:0];else
      if (addr[1:0] == 2'b11) dm[ addr[11:2] ][31:24] <= din[7:0];
   end
end

wire [7:0] d0 = dm[ addr[11:2] ][7:0];    //dm read by byte
wire [7:0] d1 = dm[ addr[11:2] ][15:8];
wire [7:0] d2 = dm[ addr[11:2] ][23:16];
wire [7:0] d3 = dm[ addr[11:2] ][31:24];

always@(*) begin
   if (memRead == 2'b00 && we == 2'b00)
   begin
      assign dout = 32'b0;
   end
   else if (memRead == 2'b01 && we == 2'b00)   //lw
   begin
      assign dout = dm[ addr[11:2] ];
   end
   else if (memRead == 2'b10 && we == 2'b00)    //lb
   begin
      if (addr[1:0] == 2'b00)               //read 1st byte
      begin
         assign dout = { {24{d0[7]}}, d0};
      end
      else if (addr[1:0] == 2'b01)          //read 2nd byte
      begin
         assign dout = { {24{d1[7]}}, d1};
```

```
        end
        else if (addr[1:0] == 2'b10)          //read 3rd byte
        begin
          assign dout = { {24{d2[7]}}, d2};
        end
        else if (addr[1:0] == 2'b11)          //read 4th byte
        begin
          assign dout = { {24{d3[7]}}, d3};
        end
      end
      else if (memRead == 2'b11 && we == 2'b00)   //lbu
      begin
        if (addr[1:0] == 2'b00)               //read 1st byte unsigned
        begin
          assign dout = { 24'b0, d0};
        end
        else if (addr[1:0] == 2'b01)
        begin
          assign dout = { 24'b0, d1};
        end
        else if (addr[1:0] == 2'b10)
        begin
          assign dout = { 24'b0, d2};
        end
        else if (addr[1:0] == 2'b11)
        begin
          assign dout = { 24'b0, d3};
        end
      end
    end
endmodule
```

2.1.8 NPC

```
module npc(PC_plus_4, PC_br, PC_jump01, PC_jump10, Branch_ok, Jump, NPC);

  input [31:2]    PC_plus_4,PC_br,PC_jump01, PC_jump10;
  input Branch_ok;
  input [1:0] Jump;

  output[31:2] NPC;
```

```
  assign NPC = ((Branch_ok == 1) ? PC_br :
              ((Jump == 2'b01) ? PC_jump01 :
              ((Jump == 2'b10) ? PC_jump10 : PC_plus_4)));

endmodule
```

2.1.9 mux

略

2.1.10 ctrl

见 1.2 注释，略

2.1.11 BranchBubbleUnit

```
module BranchBubbleUnit(ID_Ra, ID_Rb, Ex_RegWr, Ex_Rw, Mem_RegWr, Mem_MemtoReg, Mem_Rw,
                  BranchBubble, ID_Branch, ID_Jump);

  input[4:0] ID_Ra, ID_Rb, Ex_Rw, Mem_Rw;
  input Ex_RegWr, Mem_RegWr, Mem_MemtoReg;
  input[2:0]  ID_Branch;
  input[1:0] ID_Jump;

  output reg BranchBubble;

  initial begin
    BranchBubble = 0;
  end

  always @(*) begin
    if (ID_Branch == 3'b000) begin //no Branch
      if (ID_Jump == 2'b10) begin //JR JALR
        if (((Ex_RegWr == 1) && (Ex_Rw != 0 && Ex_Rw == ID_Ra)) ||
            ((Mem_MemtoReg == 1) && (Mem_Rw != 0 && Mem_Rw == ID_Ra))) begin
          BranchBubble <= 1;
        end else begin
          BranchBubble <= 0;
```

```
            end
        end
        else begin
            BranchBubble <= 0;
        end
    end else if (ID_Branch == 3'b001 || ID_Branch == 3'b010) begin //beq,bne
        if (((Ex_RegWr == 1) && (Ex_Rw != 0 && (Ex_Rw == ID_Ra || Ex_Rw == ID_Rb))) ||
            ((Mem_MemtoReg == 1) && (Mem_Rw != 0 && (Mem_Rw == ID_Ra || Mem_Rw ==
ID_Rb)))) begin
            BranchBubble <= 1;
        end else begin
            BranchBubble <= 0;
        end
    end else begin //bgez, bltz, bgtz, blez
        if (((Ex_RegWr == 1) && (Ex_Rw != 0 && Ex_Rw == ID_Ra)) ||
            ((Mem_MemtoReg == 1) && (Mem_Rw != 0 && Mem_Rw == ID_Ra))) begin
            BranchBubble <= 1;
        end else begin
            BranchBubble <= 0;
        end
    end
  end

endmodule
```

2.1.12 BranchForwardingUnit

```
module BranchForwardingUnit(ID_Ra, ID_Rb, Mem_Rw, Mem_RegWr, Mem_MemtoReg,
BranchForwardA, BranchForwardB);

  input[4:0] ID_Ra, ID_Rb, Mem_Rw;
  input Mem_MemtoReg, Mem_RegWr;

  output reg BranchForwardA, BranchForwardB;

  initial begin
    BranchForwardA = 0;
    BranchForwardB = 0;
  end

  always @(*) begin
```

```
    if (Mem_RegWr == 1 && Mem_Rw != 0 && Mem_Rw == ID_Ra)
        BranchForwardA <= 1;
    else
        BranchForwardA <= 0;
    if (Mem_RegWr == 1 && Mem_Rw != 0 && Mem_Rw == ID_Rb)
        BranchForwardB <= 1;
    else
            BranchForwardB <= 0;
  end

endmodule
```

2.1.13 branchOrNot

```
module branchOrNot(busA, busB, Branch, Branch_flag);

  input[31:0] busA;
  input[31:0] busB;
  input[2:0] Branch;

  output reg Branch_flag;

  initial begin
    Branch_flag = 0;
  end

  always @(*) begin
    case(Branch)
      3'b000:Branch_flag = 0;
      3'b001:Branch_flag = (busA == busB);
      3'b010:Branch_flag = (busA != busB);
      3'b011:Branch_flag = (busA[31] == 1'b0);
      3'b100:Branch_flag = (busA[31] == 1'b0 && busA != 32'b0);
      3'b101:Branch_flag = (busA[31] == 1'b1 || busA == 32'b0);
      3'b110:Branch_flag = (busA[31] == 1'b1);
    endcase
  end

endmodule
```

2.1.14 forwardingUnit

```
module forwardingUnit(ex_Ra, ex_Rb, mem_Rw, mem_RegWr, wr_Rw, wr_RegWr, forwardA, forwardB);

  input[4:0] ex_Ra, ex_Rb, mem_Rw, wr_Rw;
  input mem_RegWr, wr_RegWr;

  output reg[1:0] forwardA;
  output reg[1:0] forwardB;

  initial begin
    forwardA = 2'd0;
    forwardB = 2'd0;
  end

  always @(*) begin
    forwardA <= 2'b00;
    forwardB <= 2'b00;

    if(mem_RegWr != 0 && mem_Rw != 0 && mem_Rw == ex_Ra)
    begin
      forwardA <= 2'b10;
    end
    else if(wr_RegWr != 0 && wr_Rw != 0 && wr_Rw == ex_Ra)
    begin
      forwardA<=2'b01;
    end

    if(mem_RegWr != 0 && mem_Rw != 0 && mem_Rw == ex_Rb)
    begin
      forwardB <= 2'b10;
    end
    else if(wr_RegWr != 0 && wr_Rw != 0 && wr_Rw == ex_Rb)
    begin
      forwardB <= 2'b01;
    end
  end

endmodule
```

2.1.15 HazardDetectionUnit

```
module HazardDetectionUnit(Ex_MemtoReg, Ex_Rw, ID_Ra, ID_Rb, hazard);
```

```
    input Ex_MemtoReg;
    input[4:0] Ex_Rw, ID_Ra, ID_Rb;

    output reg hazard;

    initial hazard = 0;

    always@(*) begin
        if (Ex_MemtoReg != 0 && Ex_Rw != 0 && (Ex_Rw == ID_Rb || Ex_Rw == ID_Ra))
            hazard <= 1;
        else    hazard <= 0;
    end

endmodule
```

2.1.16 IF_ID

略

2.1.17 ID_EX

略

2.1.18 EX_MEM

略

2.1.19 MEM_Wr

略

2.2 Modelsim 测试与仿真

2.2.1 综合测试用代码

测试用汇编代码以及十六进制，二进制表示

表 2.1 36CPU 测试代码

指令地址	汇编指令	结果描述	机器指令的机器码	
			16 进制	二进制
00H	addiu $1, $0,#1	[$1] = 0000_0001H	24010001	0010_0100_0000_0001_0000_0000_0000_0001
04H	sll $2, $1,#4	[$2] = 0000_0010H	00011100	0000_0000_0000_0001_0001_0001_0000_0000
08H	addu $3, $2,$1	[$3] = 0000_0011H	00411821	0000_0000_0100_0001_0001_1000_0010_0001
0CH	srl $4, $2,#2	[$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010
10H	slti $25,$4,#5	[$25] = 0000_0001H	28990005	0010_1000_1001_1001_0000_0000_0000_0101
14H	bgez $25,#16	跳转到 54H	07210010	0000_0111_0010_0001_0000_0000_0001_0000
18H	subu $5, $3,$4	[$5] = 0000_000DH	00642823	0000_0000_0110_0100_0010_1000_0010_0011
1CH	sw $5, #20($0)	Mem[0000_0014H] = 0000_000DH	AC050014	1010_1100_0000_0101_0000_0000_0001_0100
20H	nor $6, $5,$2	[$6] = FFFF_FFE2H	00A23027	0000_0000_1010_0010_0011_0000_0010_0111
24H	or $7, $6,$3	[$7] = FFFF_FFF3H	00C33825	0000_0000_1100_0011_0011_1000_0010_0101
28H	xor $8, $7,$6	[$8] = 0000_0011H	00E64026	0000_0000_1110_0110_0100_0000_0010_0110
2CH	sw $8, #28($0)	Mem[0000_001CH] = 0000_0011H	AC08001C	1010_1100_0000_1000_0000_0000_0001_1100
30H	beq $8, $3,#2	跳转到 38H	11030002	0001_0001_0000_0011_0000_0000_0000_0010
34H	slt $9, $6,$7	不执行	00C7482A	0000_0000_1100_0111_0100_1000_0010_1010
38H	addiu $1, $0,#8	[$1] = 0000_0008H	24010008	0010_0100_0000_0001_0000_0000_0000_1000
3CH	lw $10,#20($1)	[$10] = 0000_0011H	8C2A0014	1000_1100_0010_1010_0000_0000_0001_0100
40H	bne $10,$5,#4	跳转到 50H	15450004	0001_0101_0100_0101_0000_0000_0000_0100
44H	and $11,$2,$1	不执行	00415824	0000_0000_0100_0001_0101_1000_0010_0100
48H	sw $11,#28($1)	不执行	AC2B001C	1010_1100_0010_1011_0000_0000_0001_1100
4CH	sw $4, #16($1)	不执行	AC240010	1010_1100_0010_0100_0000_0000_0001_0000
50H	jal #25	跳转到 64H, [$31] = 0000_0054H	0C000019	0000_1100_0000_0000_0000_0000_0001_1001
54H	lui $12,#12	[$12] = 000C_0000H	3C0C000C	0011_1100_0000_1100_0000_0000_0000_1100
58H	srav $26,$12,$2	[$26] = 0000_000CH	004CD007	0000_0000_0100_1100_1101_0000_0000_0111
5CH	sllv $27,$26,$1	[$27] = 0000_0018H	003AD804	0000_0000_0011_1010_1101_1000_0000_0100
60H	jalr $27	跳转到 18H , [$31] = 0000_0064H	0360F809	0000_0011_0110_0000_1111_1000_0000_1001
64H	sb $26,#5($3)	MEM[0000_0016H] = 000C_000DH	A07A0005	1010_0000_0111_1010_0000_0000_0000_0101
68H	sltu $13,$3,$3	[$13] = 0000_0000H	0063682B	0000_0000_0110_0011_0110_1000_0010_1011
6CH	bgtz $13,#3	不跳转	1DA00003	0001_1101_1010_0000_0000_0000_0000_0011

70H	sllv $14,$6,$4	[$14] =FFFF_FE20H	00867004	0000_0000_1000_0110_0111_0000_0000_0100
74H	sra $15,$14,#2	[$15] =FFFF_FF88H	000E7883	0000_0000_0000_1110_0111_1000_1000_0011
78H	srlv $16,$15,$1	[$16] =00FF_FFFFH	002F8006	0000_0000_0010_1111_1000_0000_0000_0110
7CH	blez $16,#8	不跳转	1A000008	0001_1010_0000_0000_0000_0000_0000_1000
80H	srav $16,$15,$1	[$16] =FFFF_FFFFH	002F8007	0000_0000_0010_1111_1000_0000_0000_0111
84H	addiu $11,$0,#140	[$11] = 0000_008CH	240B008C	0010_0100_0000_1011_0000_0000_1000_1100
88H	bltz $16, #6	跳转到 A0H	06000006	0000_0110_0000_0000_0000_0000_0000_0110
8CH	lw $28,#3($10)	[$28] = 000C_000DH /000C_880DH	8D5C0003	1000_1101_0101_1100_0000_0000_0000_0011
90H	bne $28,$29,#7	不跳转/跳转 ACH	179D0007	0001_0111_1001_1101_0000_0000_0000_0111
94H	sb $15,#8($5)	Mem[0000_0015H] = 0000_0088H	A0AF0008	1010_0000_1010_1111_0000_0000_0000_1000
98H	lb $18,#8($5)	[$18] =FFFF_FF88H	80B20008	1000_0000_1011_0010_0000_0000_0000_1000
9CH	lbu $19,#8($5)	[$19] = 0000_0088H	90B30008	1001_0000_1011_0011_0000_0000_0000_1000
A0H	sltiu $24,$15,#0xFFFF	[$24] = 0000_0001H	2DF8FFFF	0010_1101_1111_1000_1111_1111_1111_1111
A4H	or $29,$12,$5	[$29] = 000C000DH	0185E825	0000_0001_1000_0101_1110_1000_0010_0101
A8H	jr $11	跳转指令 8CH	01600008	0000_0001_0110_0000_0000_0000_0000_1000
ACH	andi $20,$15,#0xFFFF	[$20] = 0000_FF88H	31F4FFFF	0011_0001_1111_0100_1111_1111_1111_1111
B0H	ori $21,$15,#0xFFFF	[$21] =FFFF_FFFFH	35F5FFFF	0011_0101_1111_0101_1111_1111_1111_1111
B4H	xori $22,$15,#0xFFFF	[$22] = FFFF_0077H	39F6FFFF	0011_1001_1111_0110_1111_1111_1111_1111
B8H	j #00H	跳转指令 00H	08000000	0000_1000_0000_0000_0000_0000_0000_0000

2. 2. 2 测试代码执行过程

代码仿真结果如下：（注意！该波形中 PC 的值省略最低两位，请补 0 后再参考上表进行对照！）

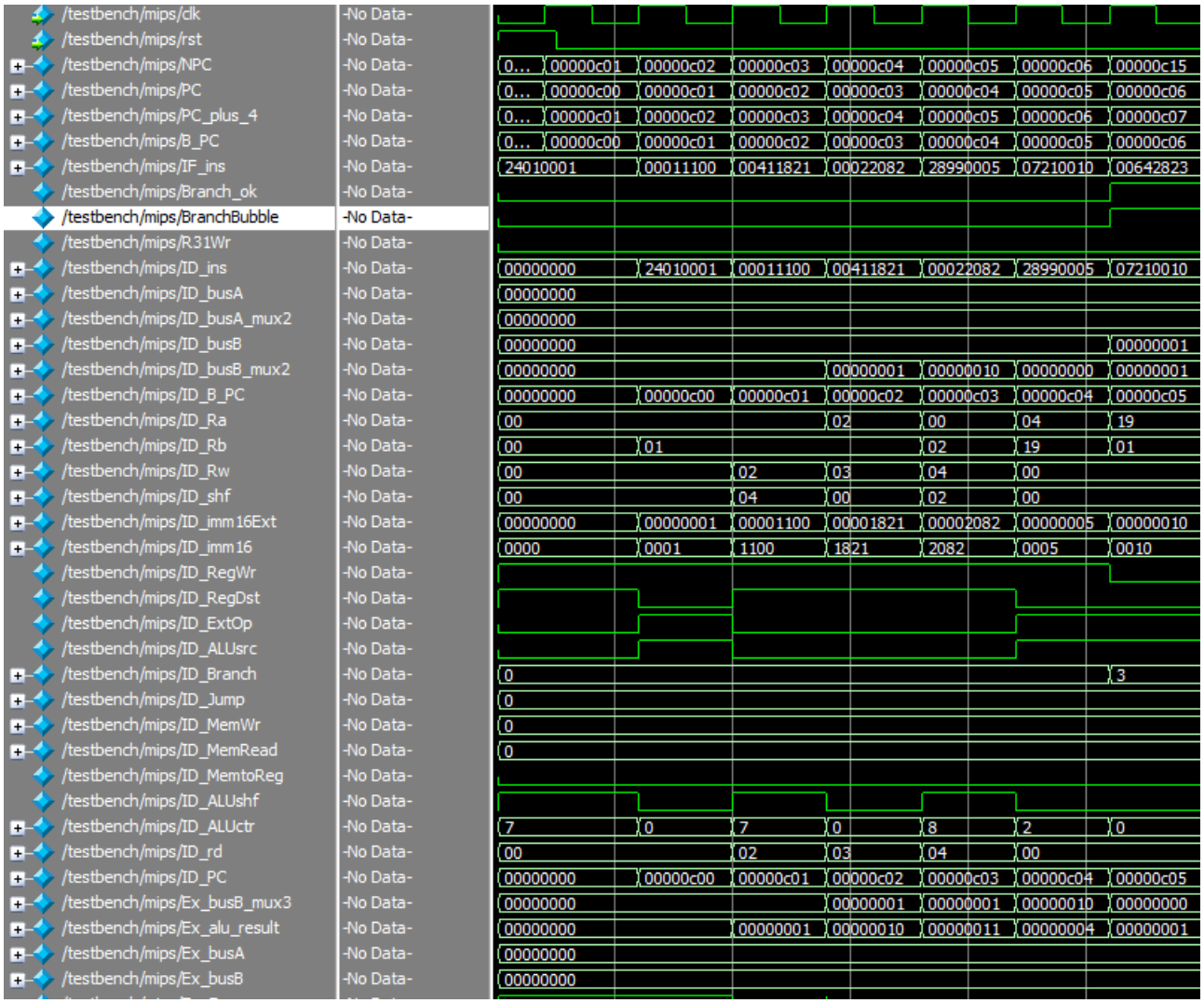

图 2.1 流水线 CPU 执行时波形

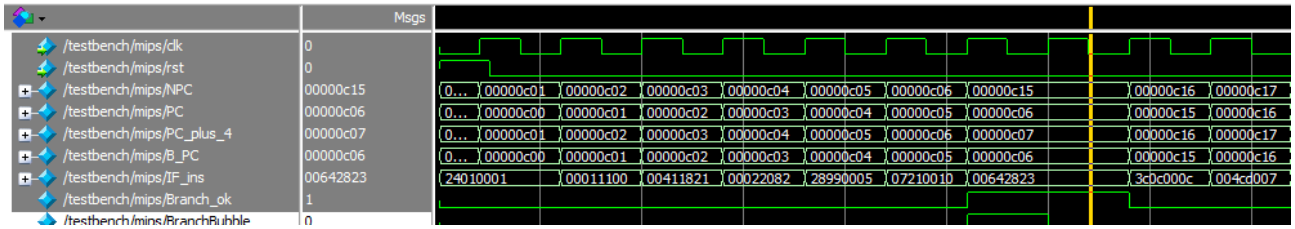

图 2.2 延迟槽

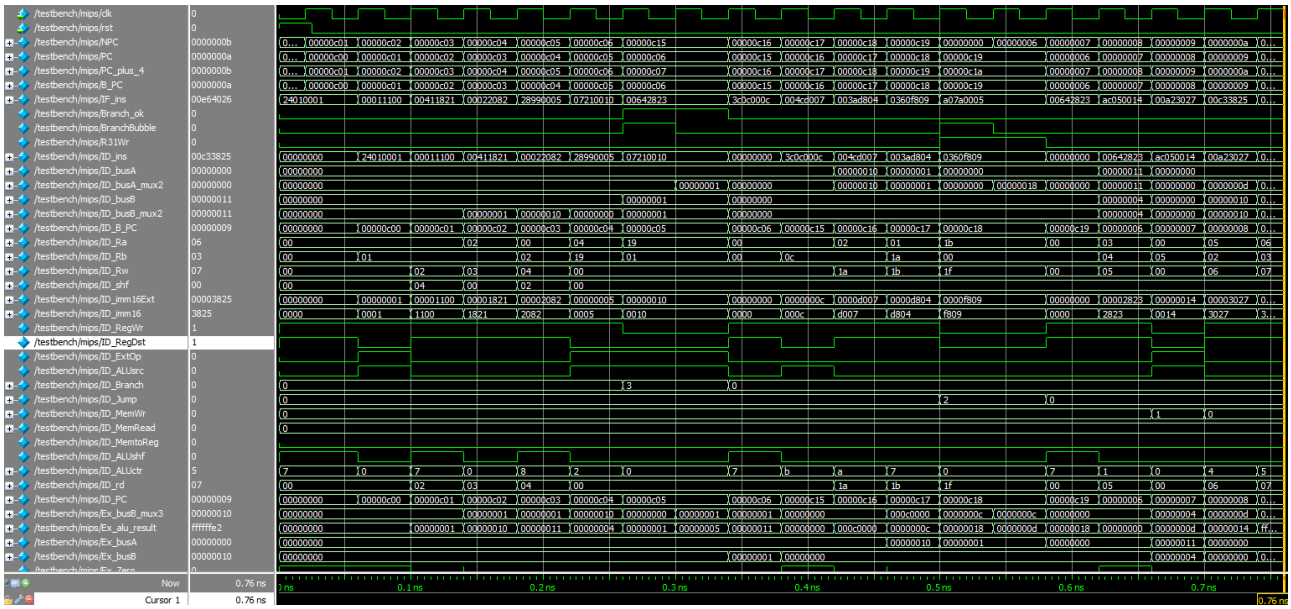

图 2.3 CPU 执行情况

可以看到此时 CPU 各信号输出正常，此时应该观察到

$1 = 0000_0001H　　$2 = 0000_0010H　　$3 = 0000_0011H　　$4 = 0000_0004H
$5 = 0000_000DH　　$12 = 000C_0000H　　$25 = 0000_0001H　　$26 = 0000_000CH
$27 = 0000_0018H　　$31 = 0000_3064H

现记录寄存器数据如下：（大端）

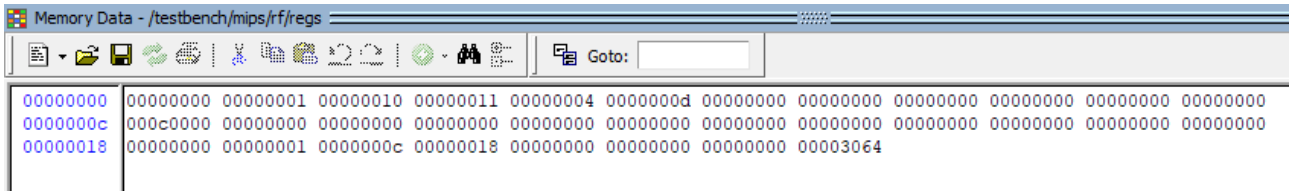

图 2.4 执行到 1CH sw $5, 20($0)的寄存器情况

还应该观察到内存数据

Mem[0000_0014H] = 0000_000DH

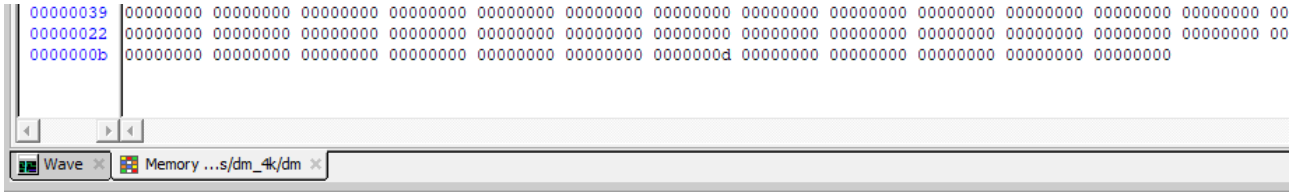

图 2.5 执行到 1CH sw $5, 20($0)的内存情况

可以看到执行无误。

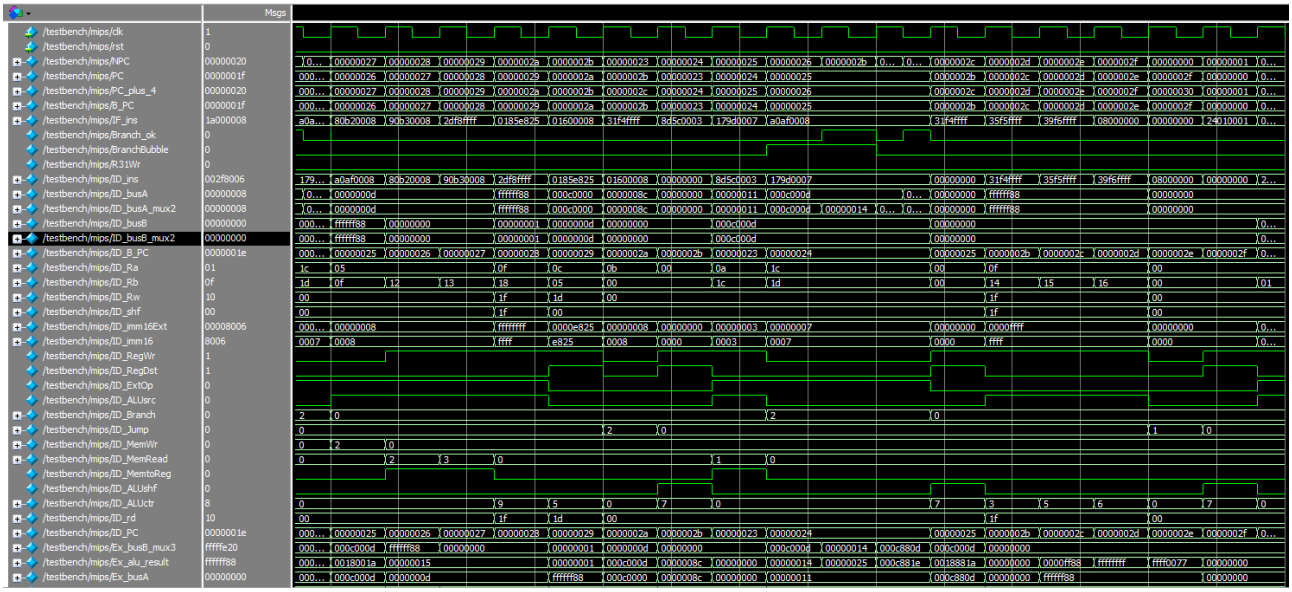

图 2.6 执行到 B8H j 00H 的波形情况

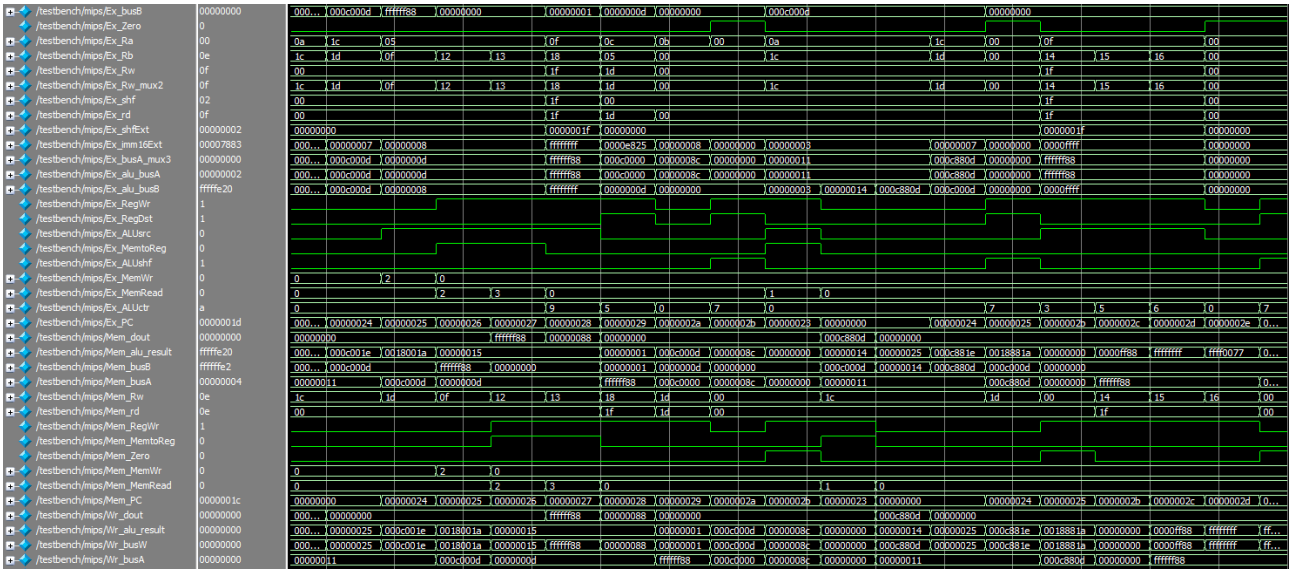

图 2.7 执行到 B8H j 00H 的波形情况（续 1）

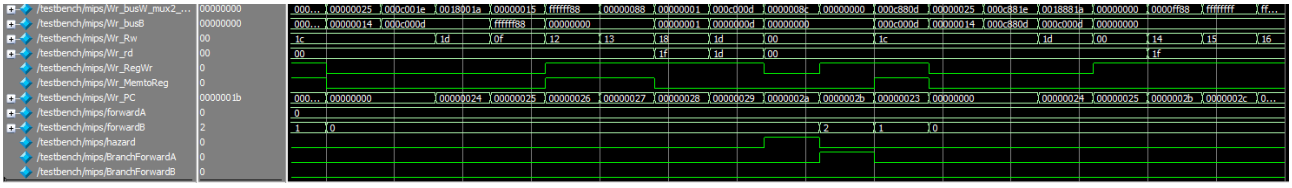

图 2.8 执行到 B8H j 00H 的波形情况（续 2）

此时寄存器情况应该为：（大端）

$1 = 0000_0008H　　$2 = 0000_0010H　　$3 = 0000_0011H　　$4 = 0000_0004H

$5 = 0000_000DH	$6 = FFFF_FFE2H	$7 = FFFF_FFF3H	$8 = 0000_0011H
$10 = 0000_0011H	$11 = 0000_008CH	$12 = 000C_0000H	$13 = 0000_0000H
$14 = FFFF_FE20H	$15 = FFFF_FF88H	$16 = FFFF_FFFFH	$18 = FFFF_FF88H
$19 = 0000_0088H	$20 = 0000_FF88H	$21 = FFFF_FFFFH	$22 = FFFF_0077H
$24 = 0000_0001H	$25 = 0000_0001H	$26 = 0000_000CH	$27 = 0000_0018H
$28 = 000C_880DH	$29 = 000C_000DH	$31 = 0000_0054H	

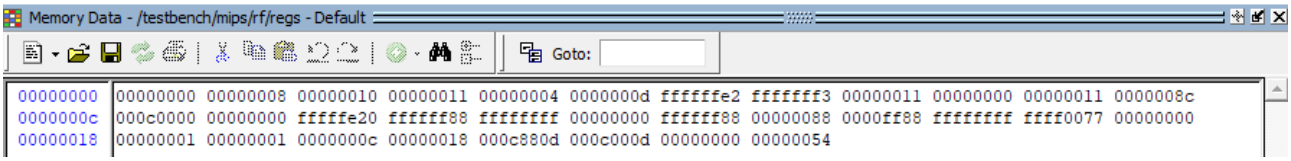

图 2.9 执行到 B8H j 00H 的寄存器情况

此时内存情况应该为：

MEM[0000_0014H] = 000C_880DH　　MEM[0000_001CH] = 0000_0011H

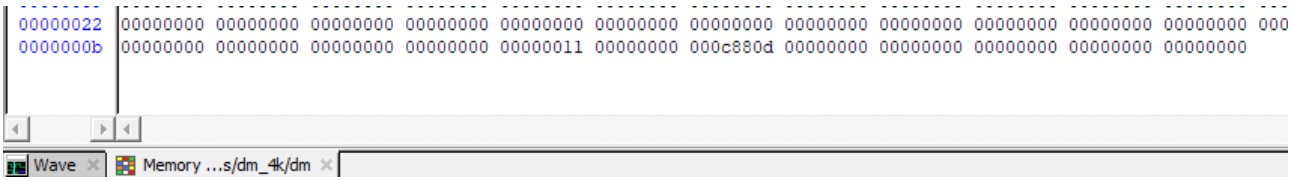

图 2.10 执行到 B8H j 00H 的内存情况

执行无误，总体测试结束。

参 考 文 献

[1] 袁春风，杨若瑜，王帅，唐杰，等. 计算机组成与系统结构[M]. 第 2 版，北京：清华大学出版社, 2015.

[2] 雷思磊. 自己动手写 CPU[M]. 北京：电子工业出版社, 2014.

[3] MIPS technologies Inc. MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set[M]. Revision 2.50, Mountain View, CA: MIS technologies Inc., 2005.

[4] David A. Patterson, John L. Hennessy. 计算机组成与设计 硬件/软件接口[M]. 第五版，北京：机械工业出版社, 2015.

心得体会

本次实验我完成了一个能够运行 36 条 MIPS 指令的五级流水线 CPU，支持除了异常中断以外的冒险，并实现了 Branch 指令提前执行的功能。构建 36 条流水线 CPU 的时候，我首先从 36 条单周期 CPU 着手，对已有的数据通路做修改，实现了一个能够完成 R 型和 I 型运算指令（如 addiu）的流水线通路，然后对于其他需要加入的指令再对 CPU 做数据通路和结构上的修改，并新增了转发模块和阻塞模块用于处理冒险，增加新功能的路是困难的，但是课本和参考资料中有比较清晰的解决方法，依照方法去解决问题的话，虽然困难，但还是可以完成的。

与单周期 CPU 不同的是，我重构了 CPU 的控制指令集来让我更好地划分整个流水线和处理冒险，但我深知我的方法可能还不是最优解，我放弃了原先使用的逻辑运算进行控制指令赋值而采用 switch-case 模块进行控制指令赋值。但我也有所突破，比如在给许多个变量同时赋值的时候采用了将它们级联之后再统一赋值的做法，这使得我的代码简化了许多。

最后一天的晚上我还尝试了把我的代码烧进 FPGA 开发板内运行，但是 ISE 出现了莫名其妙的报错，当我想要把一个外部文件加入到项目内时，ISE 提示我该文件已经在项目内存在，我猜测可能是因为项目内已经定义了同名模块导致的，但由于当时已经是半夜 1.30 分，时间太晚只能放弃调试，上板子的计划就此作罢。

我深知凭我的水平不可能完成一份完美的流水线 CPU，在我紧张的两天时间内（写完单周期之后只剩 2 天就要交流水线了）我受到了许多人的鼓励，也得到了老师的宽容和理解，这些都是我能够完成这份课设的原因所在。

十分感谢老师一学期来的教导，这次课程设计让我了解了 Verilog 编程的一些小技巧和小

思路，比如先做出各个元件再进行连接，先实现简单指令集再进一步复杂功能等等……也感谢***同学催我吃饭睡觉，不然我可能在调试流水线的电脑桌前就再起不来了（没有的事）。调bug 是一件很艰巨的任务，尤其是使用 modelsim 的时候，还是共勉吧！