编号_____

南京航空航天大学

实 验 报 告

题　目　　单周期 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：_____）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：　　年　月　日

（学号）：

单周期 MIPS CPU 的设计（36 条）

摘　要

通过实验来学习单周期 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 条指令的单周期 CPU，但不支持溢出处理，也没有设置延迟槽。

关键词：单周期 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 single-cycle 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 single-cycle CPU with MIPS architecture capable of executing a total of 36 instructions such as addu, subu, slt, addiu, beq, jump, etc., but does not support overflow processing, and didn't set the Branch delay slot.

Key Words：single-cycle CPU，MIPS，Verilog，simulation

第一章 引 言

1.1 单周期理论结构

单周期处理器的数据通路结构如图所示。

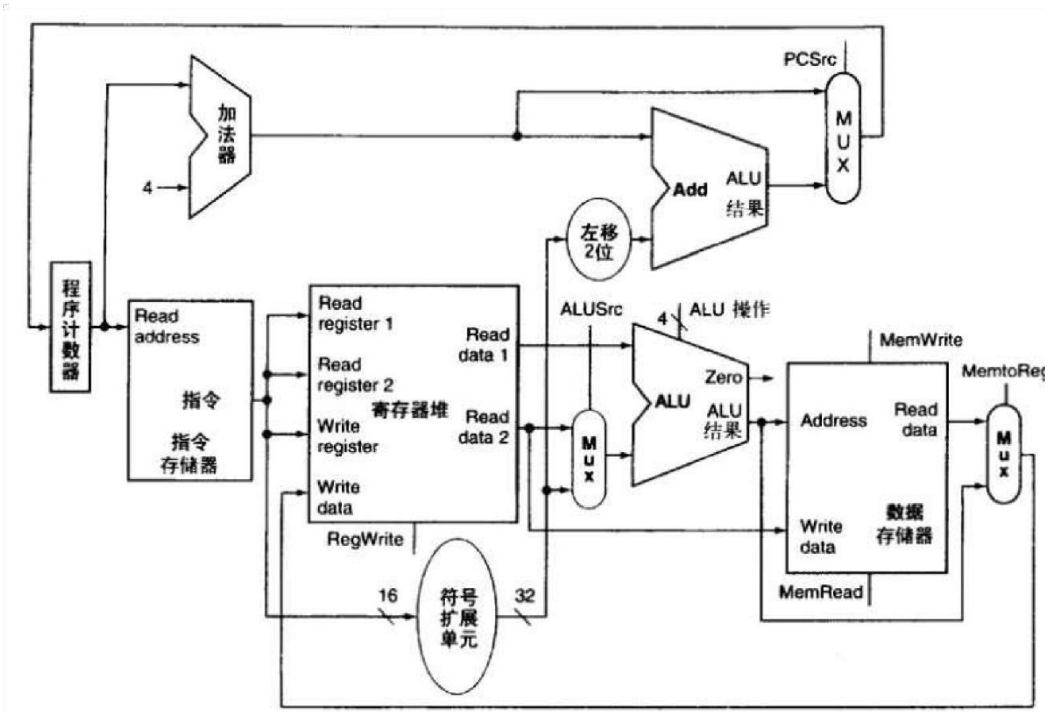

图 1.1 单周期处理器的数据通路

1.1.1 mips 模块定义

(1) 基本功能

作为整个 CPU 的最上级模块，将 controller 模块和 datapath 模块连接起来，并提供 clk 时钟信号和 rst 复位信号。

（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	链接	将 controller 和 datapath 连接起来

1.1.2 controller 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, ExtOp, Rtype, ALUctr, NPCop, DMop, REGSop 这些控制信号。

（2）模块接口

表 1.3　controller 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
Rt[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	0	用于控制数据存储器能否输出数据到寄存器
MemWr	0	用于控制数据存储器可否被写入
Branch	0	表示是否为分支指令
Jump	0	表示是否为无条件跳转指令
ExtOp	0	用于控制扩展器是否为符号扩展
Rtype	0	表示是否为 R 型指令
ALUctr[4:0]	0	控制 ALU 功能
NPCop[3:0]	0	控制 NPC 功能
DMop	0	控制数据存储器（dm）功能
REGSop[2:0]	0	控制寄存器功能

（3）功能模块

表 1.4　controller 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，Rt,生成各种控制信号并输出

1.1.3 datapath 模块定义

（1）基本功能

实现整个 CPU 的数据通路,将各个操作元件连接起来,并将输入的 clk、rst 信号和 controller 生成的控制信号传给它们，实现对操作元件的控制。并根据 instruction 模块的译码结果，给 controller 提供 op, func, Rt。

（2）模块接口

表 1.5 datapath 模块接口定义

信号名	方向	描述
clk	I	时钟信号
rst	I	复位信号
RegDst	I	用于控制寄存器写入地址
ALUsrc	I	用于控制 ALU 是否接收立即数
MemtoReg	I	用于控制数据存储器能否输出数据到寄存器
RegWr	I	用于控制寄存器可否被写入
MemWr	I	用于控制数据存储器可否被写入
Branch	I	表示是否为分支指令
Jump	I	表示是否为无条件跳转指令
ExtOp	I	用于控制扩展器是否为符号扩展
Rtype	I	表示是否为 R 型指令
ALUctr[4:0]	I	控制 ALU 功能
NPCop[3:0]	I	控制 NPC 功能
DMop	I	控制数据存储器（dm）功能
REGSop[2:0]	I	控制寄存器功能
op[5:0]	O	指令的 op 段，用于提供给 controller 生成控制信号
func[5:0]	O	指令的 func 段，用于提供给 controller 生成控制信号
Rt[4:0]	O	指令的 Rt 段，用于提供给 controller 生成控制信号

（3）功能定义

表 1.6 datapath 的功能定义

序号	功能名称	功能描述
1	连接元件	连接各个操作元件，完成数据通路
2	输出指令段落	提供 op，func，Rt 给 controller 生成控制信号

1.1.4 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 1.7 pc 模块接口定义

信号名	方向	描述
NPC[31:0]	I	下一条指令的地址
rst	I	复位信号
clk	I	时钟信号
PC[31:0]	O	当前指令的地址

（3）功能定义

表 1.8 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

1.1.5 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 1.9 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 1.10 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

1.1.6 instruction 模块定义

(1) 基本功能

将 pc 模块取出的指令进行译码并传回到 datapath 中，用于其他操作元件的操作和 controller 得出控制信号。

(2) 模块接口

表 1.11 instruction 模块接口定义

信号名	方向	描述
ins[31:0]	I	32 位指令输入
op[5:0]	O	op = ins[31:26]用于指令判断
func[5:0]	O	func = ins[5:0]用于 R 型指令判断
rs[4:0]	O	rs[4:0] = ins[25:21]
rt[4:0]	O	rt[4:0] = ins [20:16]
rd[4:0]	O	rd[4:0] = ins[15:11]
shamt[4:0]	O	shamt[4:0] = ins[10:6]做偏移量
imm16[15:0]	O	imm16[15:0] = ins[15:0]为立即数
target[25:0]	O	target[25:0] = ins[25:0]为跳转目标地址

（3）功能定义

表 1.12 instruction 接口的功能定义

序号	功能名称	功能描述
1	译码	根据输入的指令得出 op，func，rs，rt 等译码后信息

1.1.7 regfile 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 1.13 regfile 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
we	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
busA[31:0]	O	输出信息 A
busB[31:0]	O	输出信息 B

（3）功能定义

表 1.14 regfile 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

1.1.8 ext 模块定义

（1）基本功能

实现了将 WIDTH 位扩展到 extWIDTH 位的扩展器，由 ExtOp 控制是否为有符号扩展。

（2）模块接口

表 1.15 ext 模块接口定义

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

（3）功能定义

表 1.16 ext 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 width 位 a 无符号扩展为 extwidth 位 b
2	有符号扩展	将输入的 width 位 a 有符号扩展为 extwidth 位 b

1.1.9 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 1.17 alu 模块接口定义

信号名	方向	描述
ALUctr[4:0]	I	用于控制 alu 进行运算 00000:addu 00001:subu 00010:slt 00011:and 00100:nor 00101:or 00110:xor 00111:sll 01000:srl 01100:sllv 01101:sra 01110:srav 01111:srlv 10000:lui
busA[31:0]	I	数据输入通路 A
busB[31:0]	I	数据输入通路 B
shamt[4:0]	I	偏移量

Result[31:0]	O	运算结果
Zero	O	结果为 0 时 Zero 为 1

（3）功 I 能定义

表 1.18 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

1.1.10 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 1.19 dm_4k 模块接口定义

信号名	方向	描述
addr[31:0]	I	输入/输出地址
din[31:0]	I	输入数据流
we	I	写使能信号
clk	I	时钟信号
DMop	I	DMop = 1 时执行 sb 指令在指定位置存入字节
dout[31:0]	O	数据输出流

（3）功能定义

表 1.20 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

1.1.11 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，内置 signext 模块。

（2）模块接口

表 1.21 NPC 模块接口定义

信号名	方向	描述
PC[31:0]	I	当前指令地址
target[25:0]	I	J 型指令跳转目标
imm16[15:0]	I	I 型指令立即数
Jump	I	J 型指令控制信号
Branch	I	分支指令控制信号
Zero	I	ALU 输出为 0 的控制信号
NPCop[3:0]	I	控制 NPC 操作 0000：j 0001:jal 0010:beq 0011:bne 0100:bgez 0101:bgtz 0110:blez 0111:bltz 1000:jr/jalr
busA[31:0]	I	用于实现 bgez，bgtz，blez，bltz，jr，jalr 时直接调用 REGS[rs]
NPC[31:0]	O	输出 NextPC

（3）功能定义

表 1.22 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

1.1.12 mux 模块定义

（1）基本功能

默认 WIDTH 为 32 位的 2 选 1 模块

（2）模块接口

表 1.23 mux 模块接口定义

信号名	方向	描述

in1[WIDTH-1:0]	I	输入 1
in2[WIDTH-1:0]	I	输入 2
flag	I	选择信号 flag = 1 选择 1 作为输出
out[WIDTH-1:0]	O	输出

（3）功能定义

表 1.24 mux 接口的功能定义

序号	功能名称	功能描述
1	2 选 1	根据 flag 选择 in1，in2 中的一个输出

1.1.13 signext 模块定义

（1）基本功能

实现了将 WIDTH 长度的变量有符号扩展到 extWIDTH 位变量的扩展器。

（2）模块接口

表 1.25 signext 模块接口定义

信号名	方向	描述
a[WIDTH-1:0]	I	WIDTH 长度输入
b[extWIDTH-1:0]	I	extWIDTH 长度输出

（3）功能定义

表 1.26 signext 接口的功能定义

序号	功能名称	功能描述
1	有符号扩展	将 a 有符号扩展为 extWIDTH 位

1.2 各个指令与控制信号之间的关系

1.2.1 R-type 指令信号控制

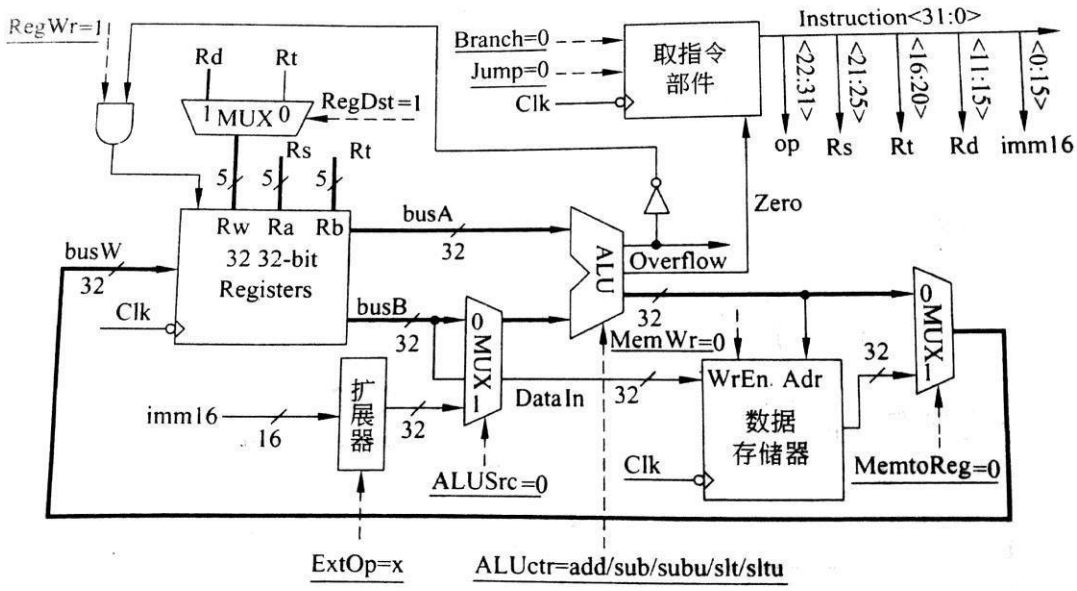

图 1.2 R-type 指令与控制信号

路径：RegFile(Rs, Rt)->busA, busB->ALU->RegFile(Rd)

1.2.2 I-type 指令与控制信号

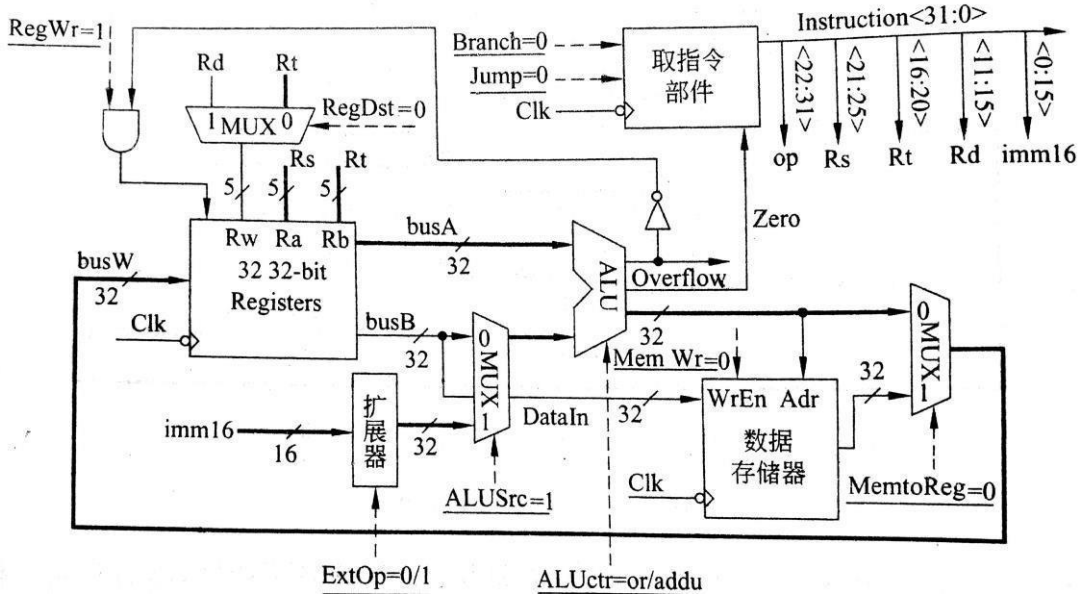

图 1.3 I-type 指令与控制信号

路径：RegFile(Rs)->busA, 扩展器(imm16)->ALU->RegFile(Rt)

1.2.3 Load 指令与控制信号

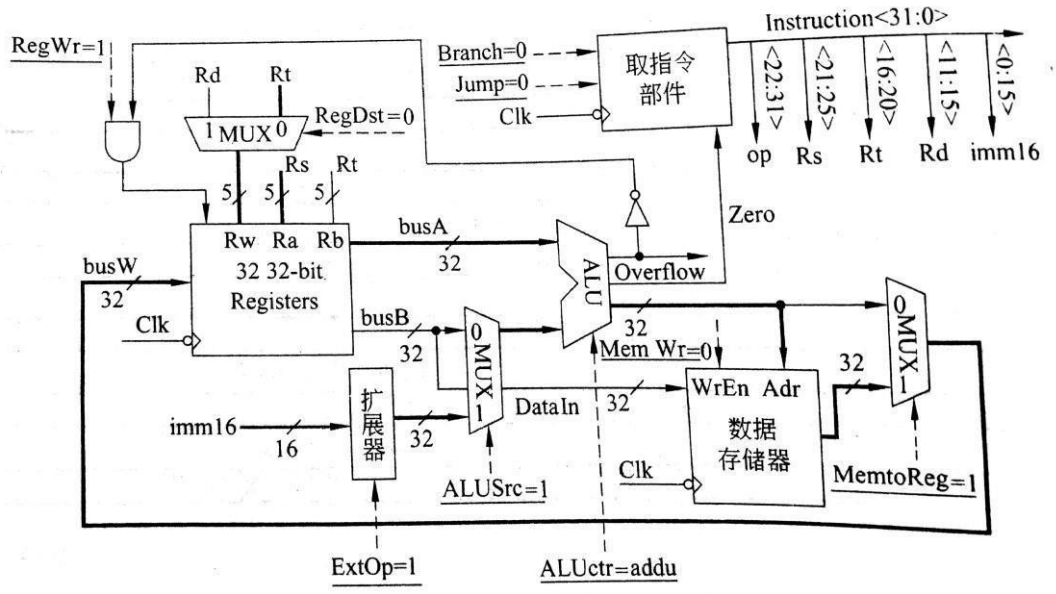

图 1.4 Load 指令与控制信号

路径：RegFile(Rs)->busA, 扩展器(imm16)->ALU(addu)->数据存储器->RegFile(Rt)

1.2.4 Store 指令与控制信号

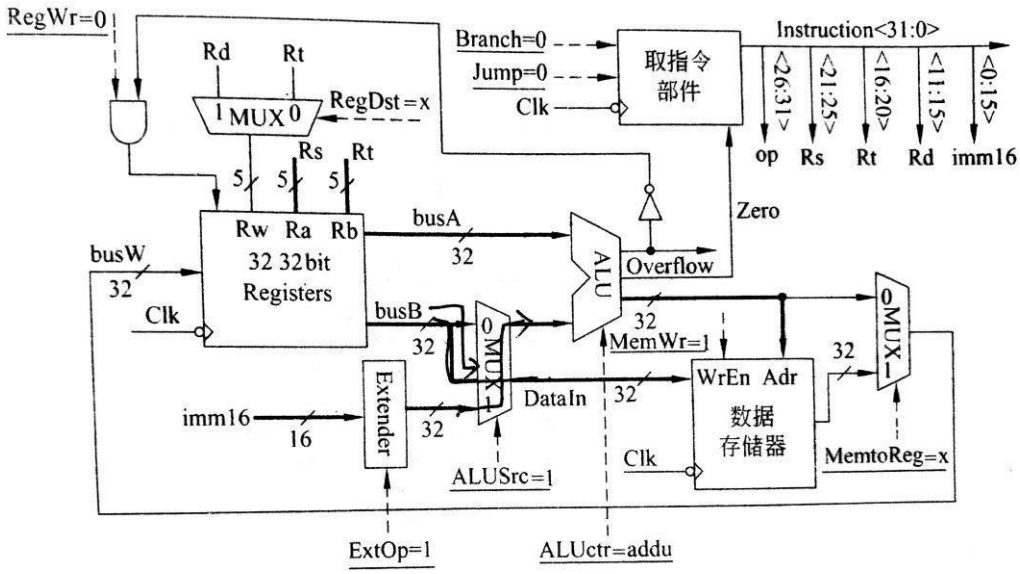

图 1.5 Store 指令与控制信号

路径：RegFile(Rs, Rt)->busA, 扩展器(imm16), busB->ALU(addu), busB->数据存储器

1.2.5 分支指令与控制信号

路径：RegFile(Rs, Rt)->busA, busB->ALU(subu)->Zero->取指令部件

1.2.6 各指令控制信号的取值

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

图 1.6 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	NPCop	DMop	REGSop
addu	000000	RS	RT	RD	00000	100001	00000			
subu	000000	RS	RT	RD	00000	100011	00001			
slt	000000	RS	RT	RD	00000	101010	00010			
and	000000	RS	RT	RD	00000	100100	00011			
nor	000000	RS	RT	RD	00000	100111	00100			
or	000000	RS	RT	RD	00000	100101	00101			
xor	000000	RS	RT	RD	00000	100110	00110			
sll	000000	RS	RT	RD	shamt	000000	00111			
srl	000000	RS	RT	RD	shamt	000010	01000			
sltu	000000	RS	RT	RD	00000	101011	01001			
jalr	000000	RS	RT	RD	00000	001001	01010	1000		11
jr	000000	RS	RT	RD	00000	001000	01011			
sllv	000000	RS	RT	RD	00000	000100	01100			
sra	000000	RS	RT	RD	shamt	000011	01101			
srav	000000	RS	RT	RD	00000	000111	01110			
srlv	000000	RS	RT	RD	00000	000110	01111			
addiu	001001	RS	RT	imm16			00000			
beq	000100	RS	RT	offset			00001	0010		
bne	000101	RS	RT	offset			00001	0011		
lw	100011	BASE	RT	offset			00000			
sw	101011	BASE	RT	offset			00000			
lui	001111	00000	RT	imm16			10000			
slti	001010	RS	RT	imm16			00010			
sltiu	001011	RS	RT	imm16			01001			
bgez	000001	RS	00001	offset				0100		
bgtz	000111	RS	00000	offset				0101		
blez	000110	RS	00000	offset				0110		
bltz	000001	RS	00000	offset				0111		
lb	100000	BASE	RT	offset						00
lbu	100100	BASE	RT	offset						01
sb	101000	BASE	RT	offset					1	
andi	001100	RS	RT	imm16			00011			
ori	001101	RS	RT	imm16			00101			
xori	001110	RS	RT	imm16			00110			
j	000010	target						0000		
jal	000011	target					01010	0001		10

图 1.7 36CPU 控制信号真值表（续）

注释：各个信号的逻辑表达式如下：

```
//------------------------------------------------------------------------
//Instruction Sign
wire r_addu = func[5] & !func[4] & !func[3] & !func[2] & !func[1] & func[0];
wire r_subu = func[5] & !func[4] & !func[3] & !func[2] & func[1] & func[0];
wire r_slt = func[5] & !func[4] & func[3] & !func[2] & func[1] & !func[0];
wire r_and = func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0];
wire r_nor = func[5] & !func[4] & !func[3] & func[2] & func[1] & func[0];
wire r_or = func[5] & !func[4] & !func[3] & func[2] & !func[1] & func[0];
wire r_xor = func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0];
wire r_sll = !func[5] & !func[4] & !func[3] & !func[2] & !func[1] & !func[0];
wire r_srl = !func[5] & !func[4] & !func[3] & !func[2] & func[1] & !func[0];
wire r_sltu = func[5] & !func[4] & func[3] & !func[2] & func[1] & func[0];
```

```
wire r_jalr = !func[5] & !func[4] & func[3] & !func[2] & !func[1] & func[0];
wire r_jr = !func[5] & !func[4] & func[3] & !func[2] & !func[1] & !func[0];
wire r_sllv = !func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0];
wire r_sra = !func[5] & !func[4] & !func[3] & !func[2] & func[1] & func[0];
wire r_srav = !func[5] & !func[4] & !func[3] & func[2] & func[1] & func[0];
wire r_srlv = !func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0];

wire i_addiu = !op[5] & !op[4] & op[3] & !op[2] & !op[1] & op[0];
wire i_beq = !op[5] & !op[4] & !op[3] & op[2] & !op[1] & !op[0];
wire i_bne = !op[5] & !op[4] & !op[3] & op[2] & !op[1] & op[0];
wire i_lw = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];
wire i_sw = op[5] & !op[4] & op[3] & !op[2] & op[1] & op[0];
wire i_lui = !op[5] & !op[4] & op[3] & op[2] & op[1] & op[0];
wire i_slti = !op[5] & !op[4] & op[3] & !op[2] & op[1] & !op[0];
wire i_sltiu = !op[5] & !op[4] & op[3] & !op[2] & op[1] & op[0];
wire i_bgez = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & op[0];
wire i_bgtz = !op[5] & !op[4] & !op[3] & op[2] & op[1] & op[0];
wire i_blez = !op[5] & !op[4] & !op[3] & op[2] & op[1] & !op[0];
wire i_bltz = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & op[0];
wire i_lb = op[5] & !op[4] & !op[3] & !op[2] & !op[1] & !op[0];
wire i_lbu = op[5] & !op[4] & !op[3] & op[2] & !op[1] & !op[0];
wire i_sb = op[5] & !op[4] & op[3] & !op[2] & !op[1] & !op[0];
wire i_andi = !op[5] & !op[4] & op[3] & op[2] & !op[1] & !op[0];
wire i_ori = !op[5] & !op[4] & op[3] & op[2] & !op[1] & op[0];
wire i_xori = !op[5] & !op[4] & op[3] & op[2] & op[1] & !op[0];

wire j_j = !op[5] & !op[4] & !op[3] & !op[2] & op[1] & !op[0];
wire j_jal = !op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];
```

```
    //------------------------------------------------------------------------
    //SIGNctrl
    assign Rtype = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & !op[0];
    assign Branch = i_beq | i_bne | i_bgez | i_bgtz | i_blez | i_bltz;
    assign Jump = j_j | j_jal;
    assign RegDst = Rtype;
    assign ALUSrc = i_addiu | i_lw | i_sw | i_lui | i_slti | i_sltiu | i_lb | i_lbu | i_sb | i_andi | i_ori
| i_xori;
    assign MemtoReg = i_lw | i_lb | i_lbu;
    assign RegWr = Rtype | i_addiu | i_lw | i_lui | i_slti | i_sltiu | i_lb | i_lbu | i_andi | i_ori |
i_xori | j_jal;
    assign MemWr = i_sw | i_sb;
    assign ExtOp = i_addiu | i_lw | i_sw | i_slti | i_sltiu | i_lb | i_lbu | i_sb;

    //------------------------------------------------------------------------
    //ALUctrl
    assign FuncOp[4] = 1'b0;
    assign FuncOp[3] = r_srl | r_sltu | r_jalr | r_jr | r_sllv | r_sra | r_srav | r_srlv;
    assign FuncOp[2] = r_nor | r_or | r_xor | r_sll | r_sllv | r_sra | r_srav | r_srlv;
    assign FuncOp[1] = r_slt | r_and | r_xor | r_sll | r_jalr | r_jr | r_srav | r_srlv;
    assign FuncOp[0] = r_subu | r_and | r_or | r_sll | r_sltu | r_jr | r_sra | r_srlv;

    assign ALUop[4] = i_lui;
    assign ALUop[3] = i_sltiu | j_jal;
    assign ALUop[2] = i_ori | i_xori;
    assign ALUop[1] = i_slti | i_andi | i_xori | j_jal;
    assign ALUop[0] = i_beq | i_bne | i_sltiu | i_andi | i_ori;

    assign ALUctr = Rtype ? FuncOp : ALUop;
```

```
//------------------------------------------------------------------------
//NPCctrl
assign NPCop[3] = Rtype & (r_jr | r_jalr);
assign NPCop[2] = (i_bgez && Rt == 5'b1) | i_bgtz | i_blez | (i_bltz && Rt == 5'b0);
assign NPCop[1] =    i_beq | i_bne | i_blez | (i_bltz && Rt == 5'b0);
assign NPCop[0] = j_jal | i_bne | i_bgtz | (i_bltz && Rt == 5'b0);

//------------------------------------------------------------------------
//DMctrl
assign DMop = i_sb;

//------------------------------------------------------------------------
//REGSctrl
assign REGSop[2] = r_jalr;
assign REGSop[1] = i_lbu | j_jal;
assign REGSop[0] = i_lb | j_jal;
```

第二章 单周期 MIPS CPU 的具体设计与调试

2.1 具体源代码实现

Mips

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

    wire[5:0] op, func;
    wire[4:0] Rt;
    wire[4:0] ALUctr;
    wire[3:0] NPCop;
    wire DMop;
    wire[2:0] REGSop;
    wire RegWr, ALUSrc, RegDst, MemtoReg, MemWr, Branch, Jump, ExtOp, Rtype;

    controller controller(op, func, Rt, RegWr, ALUSrc, RegDst, MemtoReg, MemWr, Branch, Jump,
ExtOp, Rtype, ALUctr, NPCop, DMop, REGSop);
    datapath datapath(clk, rst, RegDst, ALUSrc, MemtoReg, RegWr, MemWr, Branch, Jump, ExtOp,
Rtype, ALUctr, NPCop, DMop, REGSop, op, func, Rt);
endmodule
```

Controller

```
module controller(op, func, Rt, RegWr, ALUSrc, RegDst, MemtoReg, MemWr, Branch, Jump, ExtOp,
Rtype, ALUctr, NPCop, DMop, REGSop);
    input[5:0] op, func;
    input[4:0] Rt;

    output RegWr, ALUSrc, RegDst, MemtoReg, MemWr, Branch, Jump, ExtOp, Rtype;
    output[4:0] ALUctr;
```

```
output[3:0] NPCop;
output DMop;
output[2:0] REGSop;

wire[4:0] FuncOp, ALUop;

//------------------------------------------------------------------------
//Instruction Sign
wire r_addu = func[5] & !func[4] & !func[3] & !func[2] & !func[1] & func[0];
wire r_subu = func[5] & !func[4] & !func[3] & !func[2] & func[1] & func[0];
wire r_slt = func[5] & !func[4] & func[3] & !func[2] & func[1] & !func[0];
wire r_and = func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0];
wire r_nor = func[5] & !func[4] & !func[3] & func[2] & func[1] & func[0];
wire r_or = func[5] & !func[4] & !func[3] & func[2] & !func[1] & func[0];
wire r_xor = func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0];
wire r_sll = !func[5] & !func[4] & !func[3] & !func[2] & !func[1] & !func[0];
wire r_srl = !func[5] & !func[4] & !func[3] & !func[2] & func[1] & !func[0];
wire r_sltu = func[5] & !func[4] & func[3] & !func[2] & func[1] & func[0];
wire r_jalr = !func[5] & !func[4] & func[3] & !func[2] & !func[1] & func[0];
wire r_jr = !func[5] & !func[4] & func[3] & !func[2] & !func[1] & !func[0];
wire r_sllv = !func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0];
wire r_sra = !func[5] & !func[4] & !func[3] & !func[2] & func[1] & func[0];
wire r_srav = !func[5] & !func[4] & !func[3] & func[2] & func[1] & func[0];
wire r_srlv = !func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0];

wire i_addiu = !op[5] & !op[4] & op[3] & !op[2] & !op[1] & op[0];
wire i_beq = !op[5] & !op[4] & !op[3] & op[2] & !op[1] & !op[0];
wire i_bne = !op[5] & !op[4] & !op[3] & op[2] & !op[1] & op[0];
wire i_lw = op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];
```

```
    wire i_sw = op[5] & !op[4] & op[3] & !op[2] & op[1] & op[0];
    wire i_lui = !op[5] & !op[4] & op[3] & op[2] & op[1] & op[0];
    wire i_slti = !op[5] & !op[4] & op[3] & !op[2] & op[1] & !op[0];
    wire i_sltiu = !op[5] & !op[4] & op[3] & !op[2] & op[1] & op[0];
    wire i_bgez = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & op[0];
    wire i_bgtz = !op[5] & !op[4] & !op[3] & op[2] & op[1] & op[0];
    wire i_blez = !op[5] & !op[4] & !op[3] & op[2] & op[1] & !op[0];
    wire i_bltz = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & op[0];
    wire i_lb = op[5] & !op[4] & !op[3] & !op[2] & !op[1] & !op[0];
    wire i_lbu = op[5] & !op[4] & !op[3] & op[2] & !op[1] & !op[0];
    wire i_sb = op[5] & !op[4] & op[3] & !op[2] & !op[1] & !op[0];
    wire i_andi = !op[5] & !op[4] & op[3] & op[2] & !op[1] & !op[0];
    wire i_ori = !op[5] & !op[4] & op[3] & op[2] & !op[1] & op[0];
    wire i_xori = !op[5] & !op[4] & op[3] & op[2] & op[1] & !op[0];

    wire j_j = !op[5] & !op[4] & !op[3] & !op[2] & op[1] & !op[0];
    wire j_jal = !op[5] & !op[4] & !op[3] & !op[2] & op[1] & op[0];

    //--------------------------------------------------------------------
    //SIGNctrl
    assign Rtype = !op[5] & !op[4] & !op[3] & !op[2] & !op[1] & !op[0];
    assign Branch = i_beq | i_bne | i_bgez | i_bgtz | i_blez | i_bltz;
    assign Jump = j_j | j_jal;
    assign RegDst = Rtype;
    assign ALUSrc = i_addiu | i_lw | i_sw | i_lui | i_slti | i_sltiu | i_lb | i_lbu | i_sb | i_andi | i_ori
| i_xori;
    assign MemtoReg = i_lw | i_lb | i_lbu;
    assign RegWr = Rtype | i_addiu | i_lw | i_lui | i_slti | i_sltiu | i_lb | i_lbu | i_andi | i_ori |
i_xori | j_jal;
```

```
assign MemWr = i_sw | i_sb;
assign ExtOp = i_addiu | i_lw | i_sw | i_slti | i_sltiu | i_lb | i_lbu | i_sb;

//------------------------------------------------------------------------
//ALUctrl
assign FuncOp[4] = 1'b0;
assign FuncOp[3] = r_srl | r_sltu | r_jalr | r_jr | r_sllv | r_sra | r_srav | r_srlv;
assign FuncOp[2] = r_nor | r_or | r_xor | r_sll | r_sllv | r_sra | r_srav | r_srlv;
assign FuncOp[1] = r_slt | r_and | r_xor | r_sll | r_jalr | r_jr | r_srav | r_srlv;
assign FuncOp[0] = r_subu | r_and | r_or | r_sll | r_sltu | r_jr | r_sra | r_srlv;

assign ALUop[4] = i_lui;
assign ALUop[3] = i_sltiu | j_jal;
assign ALUop[2] = i_ori | i_xori;
assign ALUop[1] = i_slti | i_andi | i_xori | j_jal;
assign ALUop[0] = i_beq | i_bne | i_sltiu | i_andi | i_ori;

assign ALUctr = Rtype ? FuncOp : ALUop;

//------------------------------------------------------------------------
//NPCctrl
assign NPCop[3] = Rtype & (r_jr | r_jalr);
assign NPCop[2] = (i_bgez && Rt == 5'b1) | i_bgtz | i_blez | (i_bltz && Rt == 5'b0);
assign NPCop[1] =    i_beq | i_bne | i_blez | (i_bltz && Rt == 5'b0);
assign NPCop[0] = j_jal | i_bne | i_bgtz | (i_bltz && Rt == 5'b0);

//------------------------------------------------------------------------
//DMctrl
assign DMop = i_sb;
```

```
    //------------------------------------------------------------------------
    //REGSctrl
    assign REGSop[2] = r_jalr;
    assign REGSop[1] = i_lbu | j_jal;
    assign REGSop[0] = i_lb | j_jal;

endmodule
```

Datapath

```
module datapath(clk, rst, RegDst, ALUsrc, MemtoReg, RegWr, MemWr, Branch, Jump, ExtOp, Rtype,
ALUctr, NPCop, DMop, REGSop, op, func, Rt);

    input clk, rst, RegDst, ALUsrc, MemtoReg, RegWr, MemWr, Branch, Jump, ExtOp, Rtype;
    input[4:0] ALUctr;
    input[3:0] NPCop;
    input DMop;
    input[2:0] REGSop;

    output[5:0] op, func;
    output[4:0] Rt;

    wire[31:0] PC, NPC;
    wire[31:0] instructions;
    pc pc(NPC, PC, rst, clk);
    im_4k im(PC[11:2], instructions);
```

```
    wire[4:0] Rs, Rt, Rd, shamt;
    wire[15:0] imm16;
    wire[25:0] target;
    instruction instruction(instructions, op, Rs, Rt, Rd, shamt, func, imm16, target);

    wire[31:0] Result;
    wire[31:0] busW, busA, busB;
    wire[4:0] Rw, Ra, Rb;
    assign Rw = RegDst ? Rd : Rt;
    assign Ra = Rs;
    assign Rb = Rt;
    regfile regfile(Result, RegWr, busW, Rw, Ra, Rb, clk, PC, REGSop, busA, busB);

    wire[31:0] imm32;
    ext ext(imm16, imm32, ExtOp);

    wire[31:0] tempBus;
    assign tempBus = ALUsrc ? imm32 : busB;
    wire Zero;
    alu alu(busA, tempBus, ALUctr, shamt, Result, Zero);

    wire[31:0] dout;
    dm_4k dm(Result, busB, MemWr, clk, DMop, dout);
    assign busW = MemtoReg ? dout : Result;

    NPC npc(PC, target, imm16, Jump, Branch, Zero, NPCop, busA, NPC);
endmodule
```

PC

```
//pc get ins
module pc(NPC, PC, rst, clk);
	input[31:0] NPC;
	input rst, clk;

	output[31:0] PC;

	reg[31:0] PC;

	initial
	begin
		PC <= 32'h0000_3000;
	end

	always@(posedge clk)
	begin
		PC <= rst ? 32'h0000_3000 : NPC;
	end

endmodule
```

im_4k

```
module im_4k(addr, dout);
	input[11:2] addr;
```

```
    output[31:0] dout;

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

    initial
    begin
    //    $readmemh("..\\code.txt", im);
          $readmemh("code.txt", im);
    end
    assign dout = im[addr[11:2]];
endmodule
```

instruction

```
module instruction(ins, op, rs, rt, rd, shamt, func, imm16, target);
    input[31:0] ins;

    output[5:0] op, func;
    output[4:0] rs, rt, rd, shamt;
    output[15:0] imm16;
    output[25:0] target;

    assign op = ins[31:26];
    assign rs = ins[25:21];
    assign rt = ins[20:16];
    assign rd = ins[15:11];
    assign shamt = ins[10:6];
```

```
    assign func = ins[5:0];
    assign imm16 = ins[15:0];
    assign target = ins[25:0];
endmodule
```

regfile

```
//without OverFlow
module regfile(addr, we, busW, Rw, Ra, Rb, clk, PC, REGSop, busA, busB);
    input[31:0] busW, addr, PC;
    input[4:0] Rw, Ra, Rb;
    input clk, we;
    input[2:0] REGSop;

    output[31:0] busA, busB;

    reg[31:0] regs[31:0];

    integer i;

    initial
    begin
        for(i = 0; i < 32 ; i = i + 1)
       regs[i] = 32'b0;
    end

    always@(posedge clk)
    begin
```

```
if(we)
begin
    case(REGSop)
    //lb
    3'b001:
    begin
if(Rw != 0)
begin
        case(addr[1:0])
            2'b00: regs[Rw] <= {{24{busW[7]}}, busW[7:0]};
            2'b01: regs[Rw] <= {{24{busW[15]}}, busW[15:8]};
            2'b10: regs[Rw] <= {{24{busW[23]}}, busW[23:16]};
            2'b11: regs[Rw] <= {{24{busW[31]}}, busW[31:24]};
        endcase
end
    end
//lbu
3'b010:
    begin
if(Rw != 0)
begin
        case(addr[1:0])
        2'b00: regs[Rw] <= {{24'b0}, busW[7:0]};
        2'b01: regs[Rw] <= {{24'b0}, busW[15:8]};
        2'b10: regs[Rw] <= {{24'b0}, busW[23:16]};
        2'b11: regs[Rw] <= {{24'b0}, busW[31:24]};
        endcase
end
    end
```

```
            //jal
            3'b011: regs[31] <= PC + 4;
            //jalr
            3'b100: regs[31] <= PC + 4;
            default:
        if(Rw != 0)
        begin
              regs[Rw] <= busW; //normal
        end
            endcase
            end
    end

    assign busA = (Ra == 0) ? 0 : regs[Ra];
    assign busB = (Rb == 0) ? 0 : regs[Rb];

endmodule
```

ext

```
module ext #(parameter WIDTH = 16, extWIDTH = 32)(a, b, ExtOp);
    input[WIDTH - 1:0] a;
    input ExtOp;

    output[extWIDTH - 1:0] b;

    assign b = ExtOp ? {{extWIDTH - WIDTH{a[WIDTH - 1]}}, a} :
                    {{extWIDTH - WIDTH{1'b0}}, a};
```

endmodule

alu

```
module alu(busA, busB, ALUctr, shamt, Result, Zero);
    input[4:0] ALUctr;
    input[31:0] busA, busB;
    input[4:0] shamt;

    output[31:0] Result;
    output Zero;

    reg[31:0] Result;

    always@(busA or busB or ALUctr)
    begin
        case (ALUctr)
            5'b00000:Result <= busA + busB;      //addu
            5'b00001:Result <= busA - busB;      //subu
            5'b00010:Result <= busA < busB;      //slt
            5'b00011:Result <= busA & busB;      //and
            5'b00100:Result <= ~(busA | busB); //nor
            5'b00101:Result <= busA | busB;    //or
            5'b00110:Result <= busA ^ busB;      //xor
            5'b00111:Result <= busB << shamt; //sll
            5'b01000:Result <= busB >> shamt; //srl
            5'b01001:Result <= (busA < busB) ? 1 : 0; //sltu
```

```
        //      5'b01010: jalr
        //      5'b01011: jr
              5'b01100:Result <= busB << busA;       //sllv
              5'b01101:Result <= ($signed(busB)) >>> shamt; //sra
              5'b01110:Result <= ($signed(busB)) >>> busA;    //srav
              5'b01111:Result <= busB >> busA;       //srlv
              5'b10000:Result <= busB << 16;    //lui
        endcase
    end

    assign Zero = (Result == 0);

endmodule
```

dm_4k

```
module dm_4k(addr, din, we, clk, DMop, dout) ;

    input [31:0] addr ; // address bus
    input [31:0] din ; // 32-bit input data
    input we ; // memory write enable
    input clk ; // clock
    input DMop;

    output [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'b0;
    end

    always@(posedge clk)
    begin
        if(we)
        begin
            if(DMop == 1'b1)
            begin
                if(addr[1:0] == 2'b00)
                   dm[addr[11:2]][7:0] <= din[7:0];
          if(addr[1:0] == 2'b01)
                dm[addr[11:2]][15:8] <= din[7:0];
          if(addr[1:0] == 2'b10)
                dm[addr[11:2]][23:16] <= din[7:0];
          if(addr[1:0] == 2'b11)
                dm[addr[11:2]][31:24] <= din[7:0];
            end
            else
                 dm[addr[11:2]] <= din;
        end
    end

    assign dout = dm[addr[11:2]];
```

```
endmodule
```

NPC

```
//next pc
module NPC(PC, target, imm16, Jump, Branch, Zero, NPCop, busA, NPC);
    input[31:0] PC, busA;
    input[25:0] target;
    input[15:0] imm16;
    input[3:0] NPCop;
    input Jump, Branch, Zero;

    output[31:0] NPC;

    wire[29:0] imm30;
    signext #(16,30) signext(imm16, imm30);
    wire[31:0] imm32;
    assign imm32 = {imm30, 2'b00};

    //focus!
    //wire[31:0] PCadd4 = PC + 4;              //Branch delay slot
    //focus!

    wire[31:0] nAddr = PC + 4;                 //no-J
    //wire[31:0] bAddr = PCadd4 + imm32;    //Branch
    wire[31:0] bAddr = PC + imm32;    //Branch
    //wire[31:0] jAddr = {PCadd4[31:28], target, 2'b00};    //Jump
    wire[31:0] jAddr = {PC[31:28], target, 2'b00};    //Jump
```

```
    reg[31:0] temp;

    always@(*)
    begin
        case(NPCop)
        4'b0000: temp = Jump ? jAddr : nAddr;              //j
        4'b0001: temp = Jump ? jAddr : nAddr;              //jal
        4'b0010: temp = (Zero && Branch) ? bAddr : nAddr;              //beq
        4'b0011: temp = (!Zero && Branch) ? bAddr : nAddr;             //bne
        4'b0100: temp = (busA[31] == 0) ? bAddr : nAddr;      //bgez
        4'b0101: temp = (Branch && busA != 0 && busA[31] == 0) ? bAddr : nAddr;     //bgtz
        // temp = (Branch && busA != 0 && busA[31] == 0) ? bAddr : nAddr;
        4'b0110: temp = (Branch && (busA == 0 || busA[31] == 1)) ? bAddr : nAddr;      //blez
        // temp = (Branch && (busA == 0 || busA[31] == 1)) ? B_NPC : N_NPC;
        4'b0111: temp = (busA[31] == 1) ? bAddr : nAddr;         //bltz
        4'b1000: temp = busA;        //jr jalr
        default: temp = nAddr;      //nextPC
        endcase
    end

    assign NPC = temp;
    // PC = PC + 1
    // PC = PC + 1 + signExt[imm16]
    // PC = PC[31:28] + target
    //assign NPC = Jump ? jAddr :
    //       Branch & Zero ? bAddr : nAddr;
endmodule
```

mux

```
module mux #(parameter WIDTH = 32)(in1, in2, flag, out);

    input[WIDTH - 1:0] in1, in2;
    input flag;

    output[WIDTH - 1:0] out;

    assign out = flag ? in1 : in2;

endmodule
```

signext

```
module signext #(parameter WIDTH = 16, extWIDTH = 32)(a, b);
    input[WIDTH - 1:0] a;

    output[extWIDTH - 1:0] b;

    assign b = {{extWIDTH - WIDTH{a[WIDTH - 1]}}, a};

endmodule
```

2.2 Modelsim 测试与仿真

2.2.1 综合测试用代码

（1）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）测试代码执行过程

代码仿真结果如下:

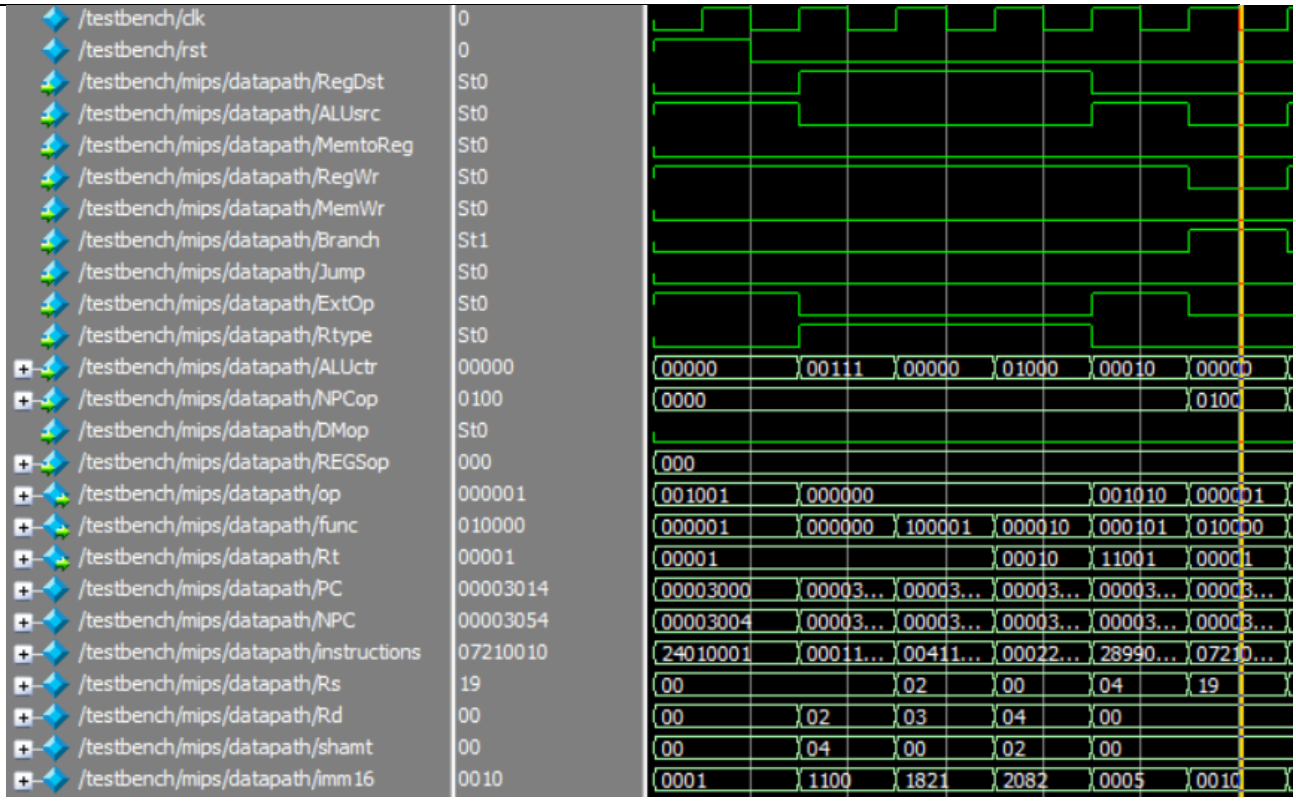

图 2.1 执行到 14H bgez $25, 16 的波形情况

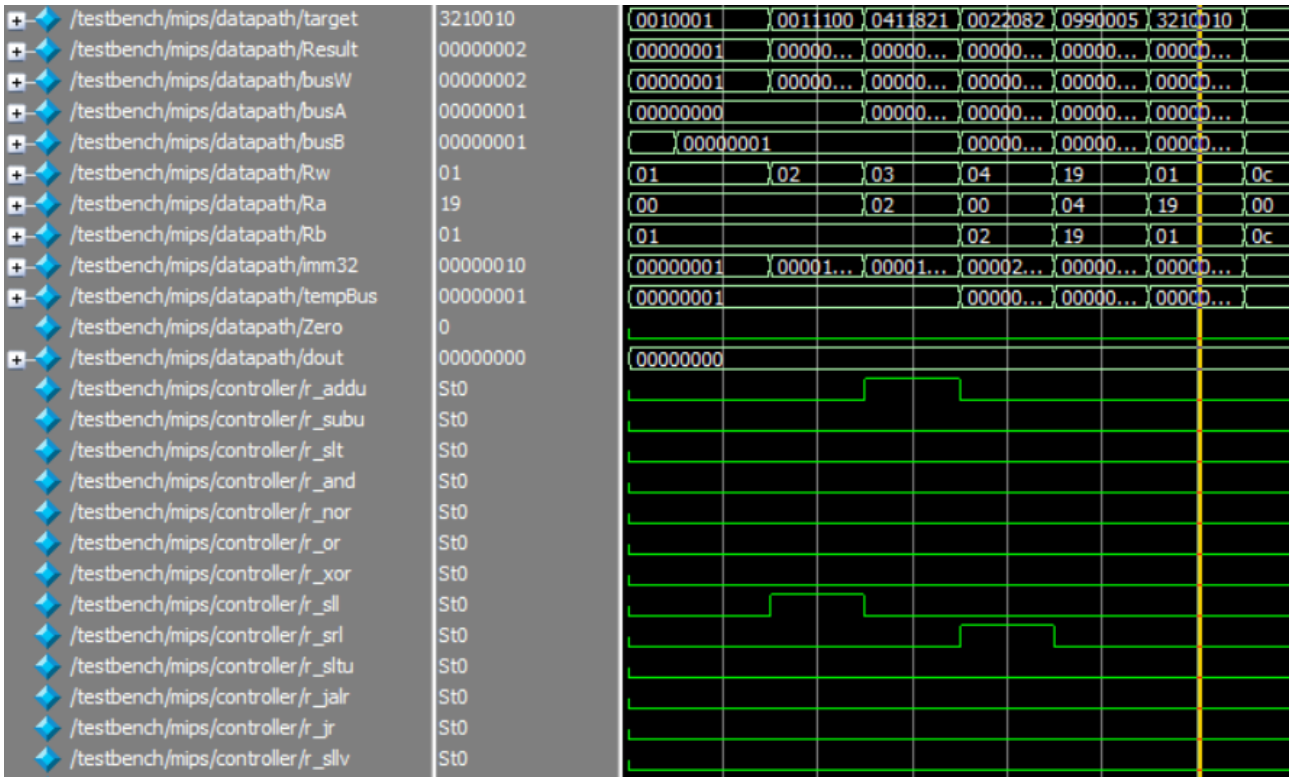

图 2.2 执行到 14H bgez $25, 16 的波形情况（续 1）

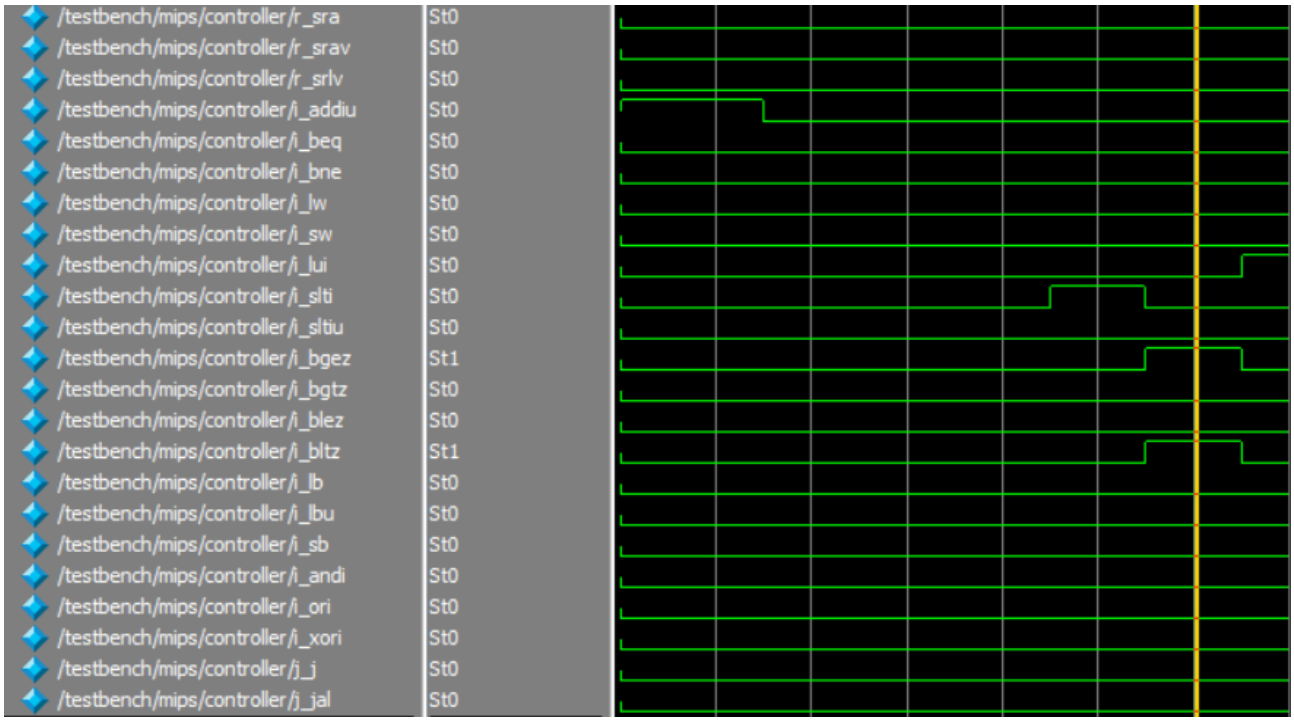

图 2.3 执行到 14H bgez $25, 16 的波形情况（续 2）

可以看到此时 CPU 各信号输出正常，此时应该观察到 regs[1] = 0000_0001H regs[2] = 0000_0010H regs[3] = 00000011H regs[4] = 0000_0004H，现记录寄存器数据如下：

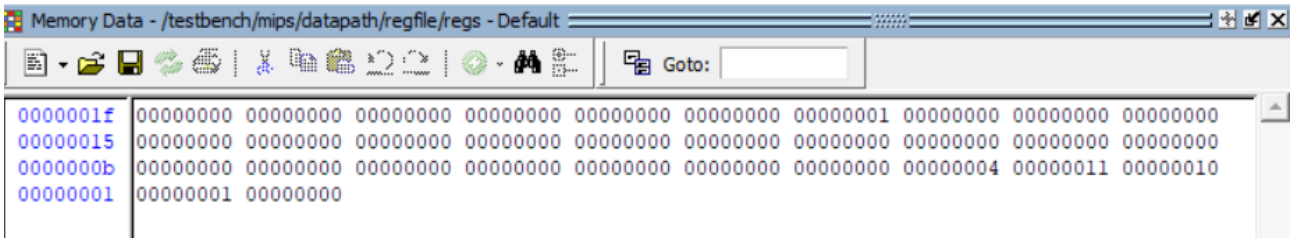

图 2.4 执行到 14H bgez $25, 16 的寄存器情况

可以看到执行无误。

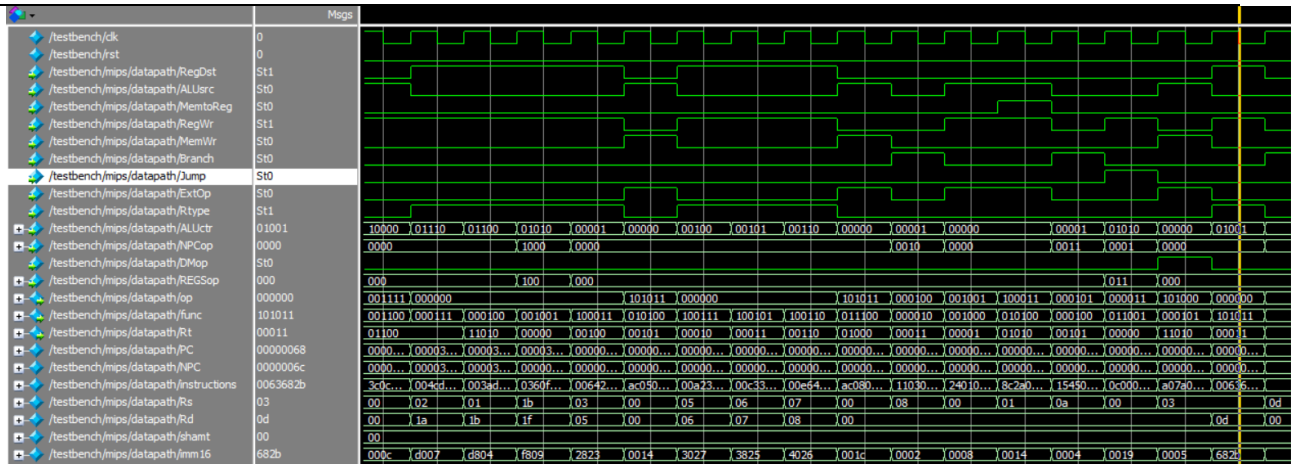

图 2.5 执行到 68H sltu $13, $3, $3 的波形情况

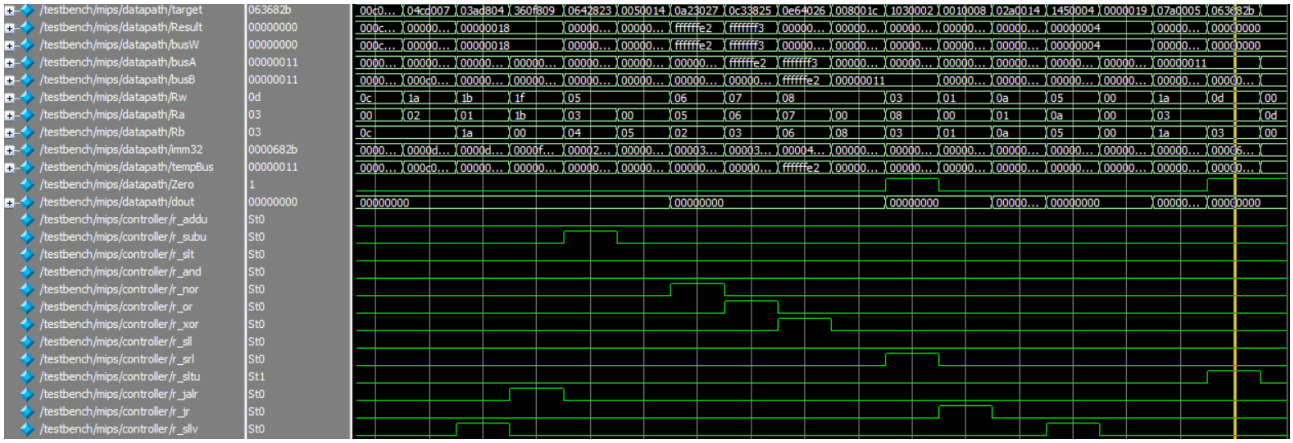

图 2.6 执行到 68H sltu $13, $3, $3 的波形情况（续 1）

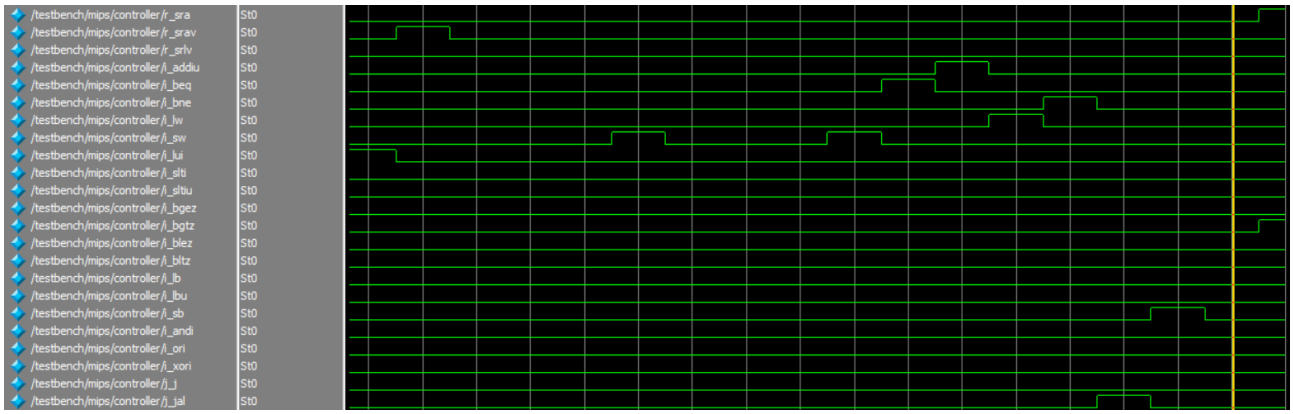

图 2.7 执行到 68H sltu $13, $3, $3 的波形情况（续 2）

此时寄存器情况应该为：

$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	$12 = 000C_0000H	$13 = 0000_0000H	$25 = 0000_0001H
$26 = 0000_000CH	$27 = 0000_0018H	$31 = 0000_0054H	

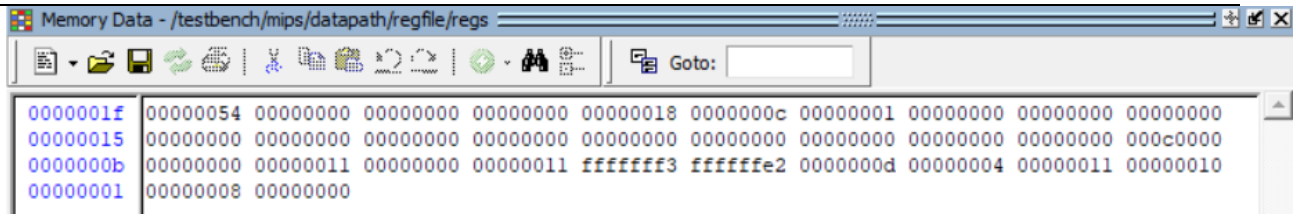

图 2.7 执行到 68H sltu $13, $3, $3 的寄存器情况

此时内存情况应该为：

MEM[0000_0014H] = 000C_000DH　　MEM[0000_001C] = 0000_0011H

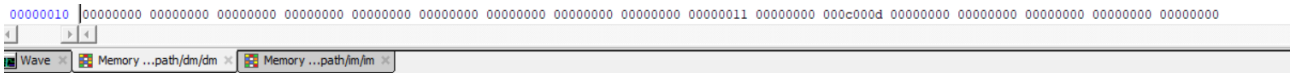

图 2.8 执行到 68H sltu $13, $3, $3 的内存情况

执行无误。

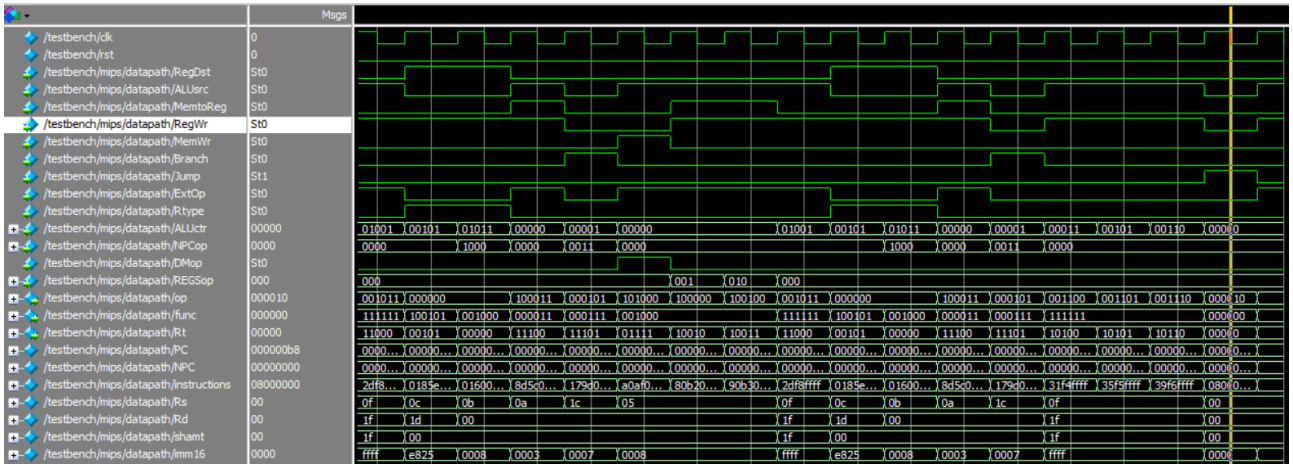

图 2.9 执行到 B8H j 00H 的波形情况

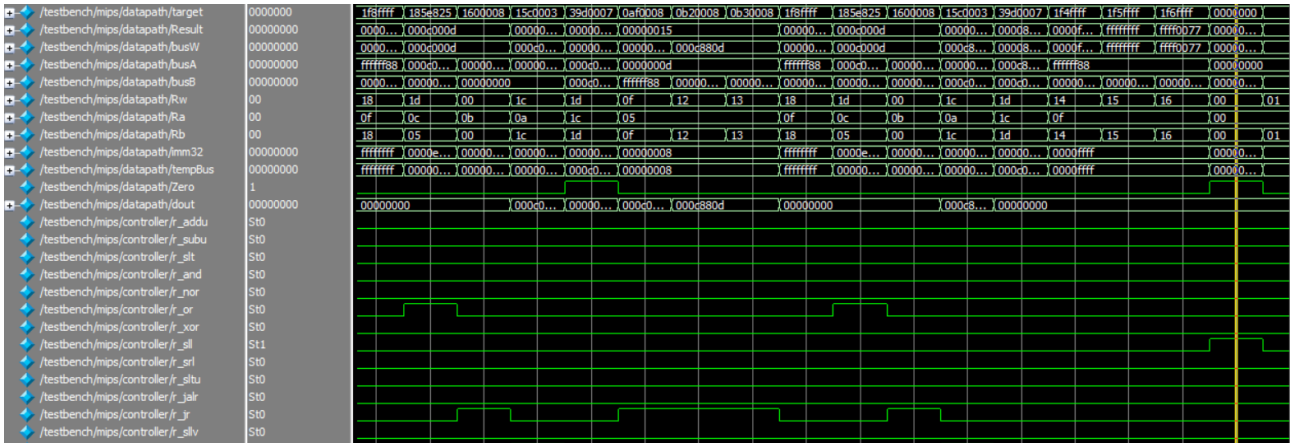

图 2.10 执行到 B8H j 00H 的波形情况（续 1）

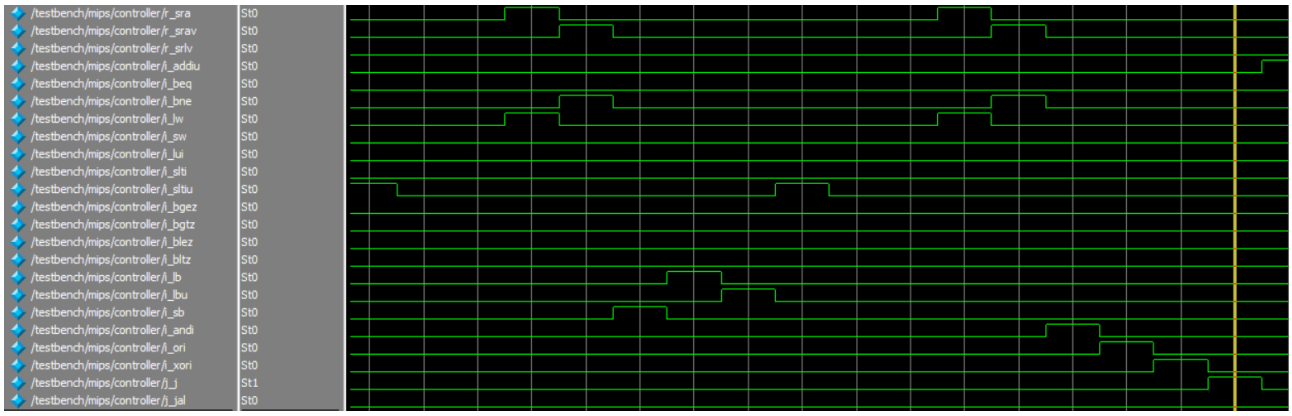

图 2.11 执行到 B8H j 00H 的波形情况（续 2）

此时寄存器情况应该为：

$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	

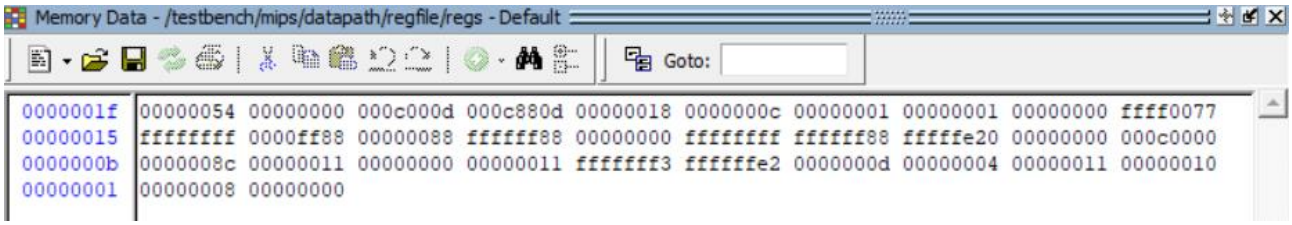

图 2.12 执行到 B8H j 00H 的寄存器情况

此时内存情况应该为：

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

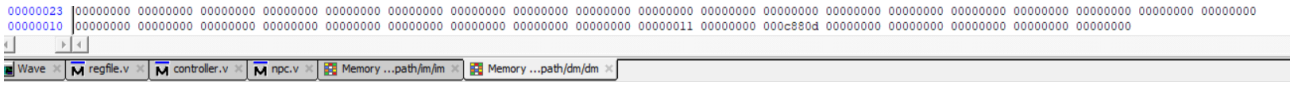

图 2.13 执行到 B8H j 00H 的内存情况

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