计算机组成原理实验报告
LAB07

题目：MIPS 流水线 CPU

姓名：林祥

学号：PB16020923

实验目的

设计一个 MIPS 指令集的 CPU，

1、 基本要求：

a) 多周期

b) 包含 16 条指令

i. add addi addu sub subu

ii. and andi or nor xor

iii. bgtz bne j jr

iv. lw sw

2、 扩展要求

a) 在 16 条的基础上，增加其他指令

b) 实现中断功能

c) 实现流水

d) 实现下载

实验平台

EDA 工具为 ISE14.7，开发板型号为 Digilent 的 Nexys3，FPGA 型号为 Spartan-6 XC6SLX16-CS324。

使用部件

开发板上的开关，7 段数码管和按钮。

最终实现功能

1、实现五段流水线

2、实现完全转发和冒险检测

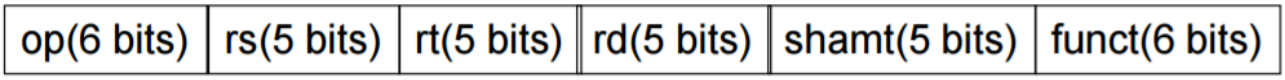
3、实现 36 条指令

4、实现下载并在数码管上动态显示内存

详细设计过程

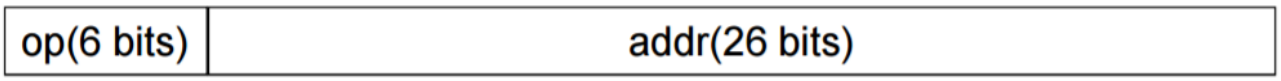
1、 MIPS 指令格式

MIPS-IV 指令集标准下的指令长度为 32 位，分为 R 型、I 型和 J 型指令



R-Tpye

I-Tpye



J-Tpye

它们的共同部分是 26-31 位的 Op 码，用来区分不同的指令，其中

（1）R-Type 的指令 Op=000000，再根据 0-5 位的 Funct 部分区分不同指令

（2）I-Type 的指令 0-15 位为立即数，位扩展后用于直接计算或者访存和跳转偏移地址。Op=000001 的 I-Type 指令，根据 Rt 的不同来区分。

（3）J-Type 的指令除了 Op 之外，0-25 位为立即数，位扩展后作为跳转地址偏移。

本次实现的 36 条指令，详细格式如下表

Instr	31-26	25-21	20-16	15-11	10-6	5-0	备注
	op	rs	rt	rd	shamt	func	
SLL	000000	-	$2	$1	shamt	000000	$1=$2<<shamt
SRL		-	$2	$1	shamt	000010	$1=$2>>shamt
SRA		-	$2	$1	shamt	000011	$1=$2>>>shamt
SLLV		$3	$2	$1	-	000100	$1=$2<<$3
SRLV		$3	$2	$1	-	000110	$1=$2>>$3
SRAV		$3	$2	$1	-	000111	$1=$2>>>$3
JR		$1	-	-	-	001000	PC=$1
MOVZ		$3	$2	$1		001010	Set $t1 = $t2 if $t3 is zero
MOVN		$3	$2	$1		001011	Set $t1 = $t2 if $t3 is not zero
ADD		$2	$3	$1	-	100000	$1=$2+$3
ADDU		$2	$3	$1	-	100001	$1=$2+$3(unsigned)
SUB		$2	$3	$1	-	100010	$1=$2-$3
SUBU		$2	$3	$1	-	100011	$1=$2-$3(unsigned)
AND		$2	$3	$1	-	100100	$1=$2 and $3
OR		$2	$3	$1	-	100101	$1=$2 or $3
XOR		$2	$3	$1	-	100110	$1=$2 xor $3
NOR		$2	$3	$1	-	100111	$1=$2 nor $3
SLT		$2	$3	$1	-	101010	if($2<$3) $1=1 else $1=0
SLTU		$2	$3	$1	-	101011	if($2<$3) $1=1 else $1=0(unsigned)
BLTZ	000001	$1	00000	offset			if($1<0) PC=PC+4+(sign-extend)offset<<2
BGEZ		$1	00001	offset			if($1>=0) PC=PC+4+(sign-extend)offset<<2
J	000010	immediate					PC=PC[31:28]+immediate<<2
BEQ	000100	$1	$2	offset			if($1==$2) PC=PC+4+(sign-extend)offset<<2
BNE	000101	$1	$2	offset			if($1!=$2) PC=PC+4+(sign-extend)offset<<2
BLEZ	000110	$1	-	offset			if($1<=0) PC=PC+4+(sign-extend)offset<<2
BGTZ	000111	$1	-	offset			if($1>0) PC=PC+4+(sign-extend)offset<<2

ADDI	001000	$2	$1	immediate	$1=$2+(sign-extend)immediate
ADDIU	001001	$2	$1	immediate	$1(unsigned)=$2+(sign-extend) immediate
SLTI	001010	$2	$1	immediate	if($2<(sign-extend)immediate) $1=1 else $1=0
SLTIU	001011	$2	$1	immediate	if($2<(sign-extend)immediate) $1=1 else $1=0
ANDI	001100	$2	$1	immediate	$1=$2 and (zero-extend)immediate
ORI	001101	$2	$1	immediate	$1=$2 or (zero-extend)immediate
XORI	001110	$2	$1	immediate	$1=$2 xor (zero-extend)immediate
LUI	001111	-	$1	immediate	$1=immediate*65536
LW	100011	$2	$1	immediate	$1=memory[$2+10]
SW	101011	$2	$1	immediate	memory[$2+10]=$1

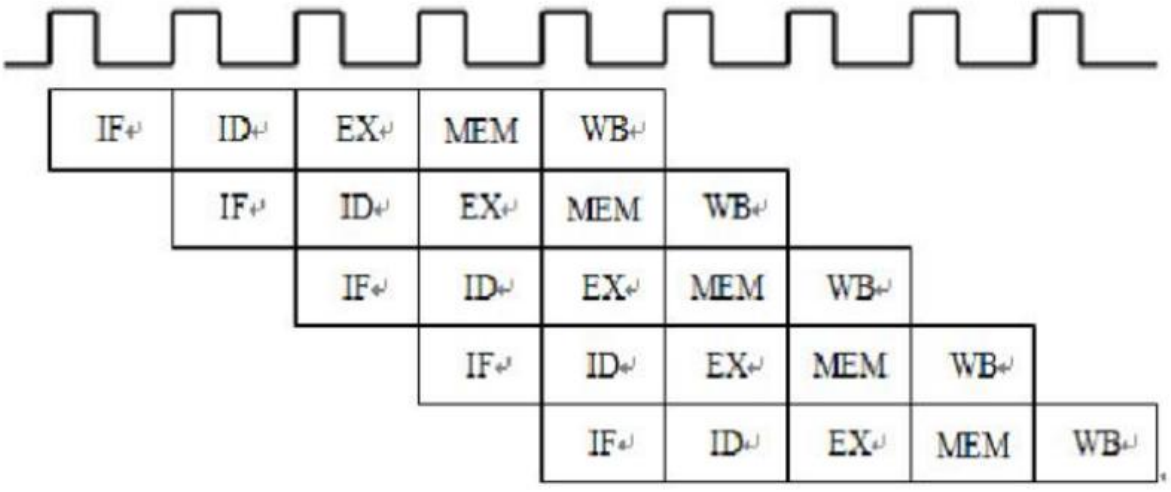
2、 整体设计

(1) 分为五个流水段 IF、ID、EX、MEM、WB。

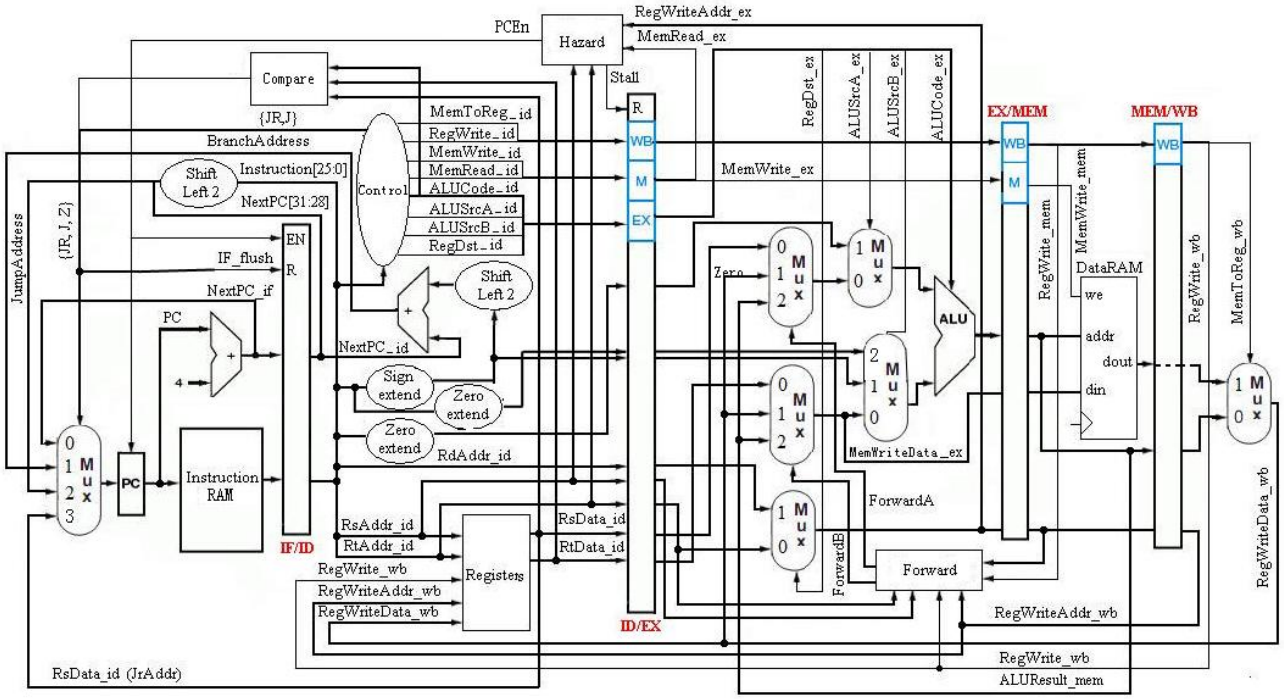
(2) clk 上升沿为每段的开始，每段长度为一个 clk。

(3) 每段的相关模块可以和 clk 上升沿同步也可以与 clk 下降沿同步。

(4) 甘特图如下



(5)详细数据通路图如下



3、 各模块设计，主要有以下几个模块

a) alu 模块——算术逻辑单元

alu.v			
输入/输出	宽度	信号名	说明
input	[31:0]	alu_a	无符号型的操作数 a，如果有负数，是以补码存储
input	[31:0]	alu_b,	无符号型的操作数 b，如果有负数，是以补码存储
input	[4:0]	alu_op,	运算类型
output	[31:0]	alu_out	无符号型的运算结果，如果有负数，是以补码存储

alu 模块完成大部分 R 型指令的计算，case 判断 alu_op 的值，使用使用 verilog 语言中的各运算符完成运算。

这里应该注意的是 add 和 addu 的实现是一样的，这是因为当操作数都是补码表示时，符号可以直接参与运算，两条指令的区别在于，add 视结果为有符号，因此如果要实现中断功能，就需要判断是否溢出。具体方法是 alu_a 与 alu_b 符号同号时，alu_out 与它们异号，则表示溢出。

另外，还需要注意 slt 的实现，比较大小时，不能使用 alu_a-alu_b>0，因为它们都是无符号数，相减结果认为无符号数，而无符号数>0 是恒成立的。

b) regfile 模块——寄存器文件

regfile.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	rst_n	复位信号，低电平有效
input	[4:0]	rAddr1	读地址 1
output	[31:0]	rDout1	读数据 1
input	[4:0]	rAddr2	读地址 2
output	[31:0]	rDout2	读数据 2
input	[4:0]	wAddr	写地址
input	[31:0]	wDin	写数据
input	[0:0]	wEna	写使能，高电平有效

这个模块实现了一个 32 个 32 位宽的寄存器组，有两个读端口，一个写端口，读取数据为异步读，写入数据在时钟上升沿且写使能有效，使用非阻塞赋值。

c) mux 模块和 mux4 模块——2 路和 4 路选择器

mux.v			
输入/输出	宽度	信号名	说明
input	[0:0]	sel	选择信号
input	[WIDTH-1:0]	d0	选择数据 1
input	[WIDTH-1:0]	d1	选择数据 2
output	[WIDTH-1:0]	out	输出

d) IP 核生成的 DMem 模块和 IMem 模块——存放数据段和代码段

Mem 是同步读，同步写，且由指定 coe 文件初始化，coe 文件的内容是 16 进制文本，由 Mars 编译汇编代码生成。

e) IFID 模块——IF 段和 ID 段之间的寄存器

IFID.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	en	使能信号，高电平有效
input	[0:0]	flush	清空信号，高电平有效
input	[31:0]	PCPlus_in	IF 段的 PC+4
input	[31:0]	IMemout_in	IF 段的 Imem 输出
output	[31:0]	PCPlus_out	ID 段的 PC+4
output	[31:0]	IMemout_out	ID 段的 Imem 输出

f) IDEX 模块——ID 段和 EX 段之间的寄存器

IDEX.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	en	使能信号高电平有效
input	[0:0]	flush	清空信号高电平有效
input	[31:0]	PCPlus_in	ID 段的 PCPlus
input	[31:0]	RegRdout1_in	ID 段的 RegRdout1
input	[31:0]	RegRdout2_in	ID 段的 RegRdout2
input	[31:0]	IMMSignExtended_in	ID 段的 IMMSignExtended
input	[31:0]	IMMZeroExtended_in	ID 段的 IMMZeroExtended
input	[31:0]	ShamtZeroExtended_in	ID 段的 ShamtZeroExtended
input	[4:0]	Rs_in	ID 段的 Rs
input	[4:0]	Rt_in	ID 段的 Rt
input	[4:0]	RegWtaddr_in	ID 段的 RegWtaddr
output	[31:0]	PCPlus_out	EX 段的 PCPlus
output	[31:0]	RegRdout1_out	EX 段的 RegRdout1
output	[31:0]	RegRdout2_out	EX 段的 RegRdout2
output	[31:0]	IMMSignExtended_out	EX 段的 IMMSignExtended
output	[31:0]	IMMZeroExtended_out	EX 段的 IMMZeroExtended
output	[31:0]	ShamtZeroExtended_out	EX 段的 ShamtZeroExtended
output	[4:0]	Rs_out	EX 段的 Rs
output	[4:0]	Rt_out	EX 段的 Rt
output	[4:0]	RegWtaddr_out	EX 段的 RegWtaddr
input	[0:0]	RegDst_in	ID 段的 RegDst
input	[0:0]	ALUSrcASel_in	ID 段的 ALUSrcASel
input	[1:0]	ALUSrcBSel_in	ID 段的 ALUSrcBSel
input	[4:0]	ALUControl_in	ID 段的 ALUControl
input	[0:0]	DMemRead_in	ID 段的 DMemRead
input	[0:0]	DMemWrite_in	ID 段的 DMemWrite
input	[0:0]	DMemtoReg_in	ID 段的 DMemtoReg
input	[0:0]	RegWrite_in	ID 段的 RegWrite
output	[0:0]	RegDst_out	EX 段的 RegDst
output	[0:0]	ALUSrcASel_out	EX 段的 ALUSrcASel
output	[1:0]	ALUSrcBSel_out	EX 段的 ALUSrcBSel
output	[4:0]	ALUControl_out	EX 段的 ALUControl
output	[0:0]	DMemRead_out	EX 段的 DMemRead
output	[0:0]	DMemWrite_out	EX 段的 DMemWrite
output	[0:0]	DMemtoReg_out	EX 段的 DMemtoReg
output	[0:0]	RegWrite_out	EX 段的 RegWrite

g) EXMEM 模块——EX 段和 MEM 段之间的寄存器

EXMEM.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	en	使能信号高电平有效
input	[0:0]	flush	清空信号高电平有效
input	[31:0]	ALUResult_in	EX 段的 ALUResult
input	[31:0]	DMemin_in	EX 的 Dmemin
input	[4:0]	RegWtaddr_in	EX 的 RegWtaddr
output	[31:0]	ALUResult_out	MEM 段的 ALUResult
output	[31:0]	DMemin_out	MEM 的 Dmemin
output	[4:0]	RegWtaddr_out	MEM 的 RegWtaddr
input	[0:0]	DMemRead_in	EX 段的 DMemRead
input	[0:0]	DMemWrite_in	EX 段的 DMemWrite
input	[0:0]	DMemtoReg_in	EX 段的 DMemtoReg
input	[0:0]	RegWrite_in	EX 段的 RegWrite
output	[0:0]	DMemRead_out	MEM 段的 DMemRead
output	[0:0]	DMemWrite_out	MEM 段的 DMemWrite
output	[0:0]	DMemtoReg_out	MEM 段的 DMemtoReg
output	[0:0]	RegWrite_out	MEM 段的 RegWrite

h) MEMWB 模块——MEM 段和 WB 段之间的寄存器

MEMWB.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	en	使能信号高电平有效
input	[0:0]	flush	清空信号高电平有效
input	[31:0]	ALUResult_in	MEM 段的 ALUResult
input	[31:0]	DMemout_in	MEM 段的 Dmemout
input	[4:0]	RegWtaddr_in	MEM 段的 RegWtaddr
output	[31:0]	ALUResult_out	WB 段的 ALUResult
output	[31:0]	DMemout_out	WB 段的 DMemout
output	[4:0]	RegWtaddr_out	WB 段的 RegWtaddr
input	[0:0]	DMemtoReg_in	MEM 段的 DMemtoReg
input	[0:0]	RegWrite_in	MEM 段的 RegWrite
output	[0:0]	DMemtoReg_out	WB 段的 DMemtoReg
output	[0:0]	RegWrite_out	WB 段的 RegWrite

i) dff 模块——D 触发器，用于各个段寄存器

dff.v

输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	en	使能信号高电平有效
input	[0:0]	rst	复位信号高电平有效
input	[WIDTH-1:0]	datain	输入数据
output	[WIDTH-1:0]	dataout	输出数据

这是一个通用的 D 触发器，位数作为参数可根据需要变化，用一个 always 在时钟沿触发来实现非阻塞赋值。

j) compare 模块——用于分支信号的判断

dff.v			
输入/输出	宽度	信号名	说明
input	[31:0]	a,	有符号数 a，需加前缀 signed
input	[31:0]	b,	有符号数 b，需加前缀 signed
output	[1:0]	res	比较结果

a 等于 b 返回 2'b01，a 小于 b 返回 2'b00，a 大于 b 返回 2'b10，比较均是基于有符号数。

k) SignExtended 模块——立即数符号扩展

过于简单，直接在 top 中实现了。

l) ZeroExtended 模块——立即数无符号扩展

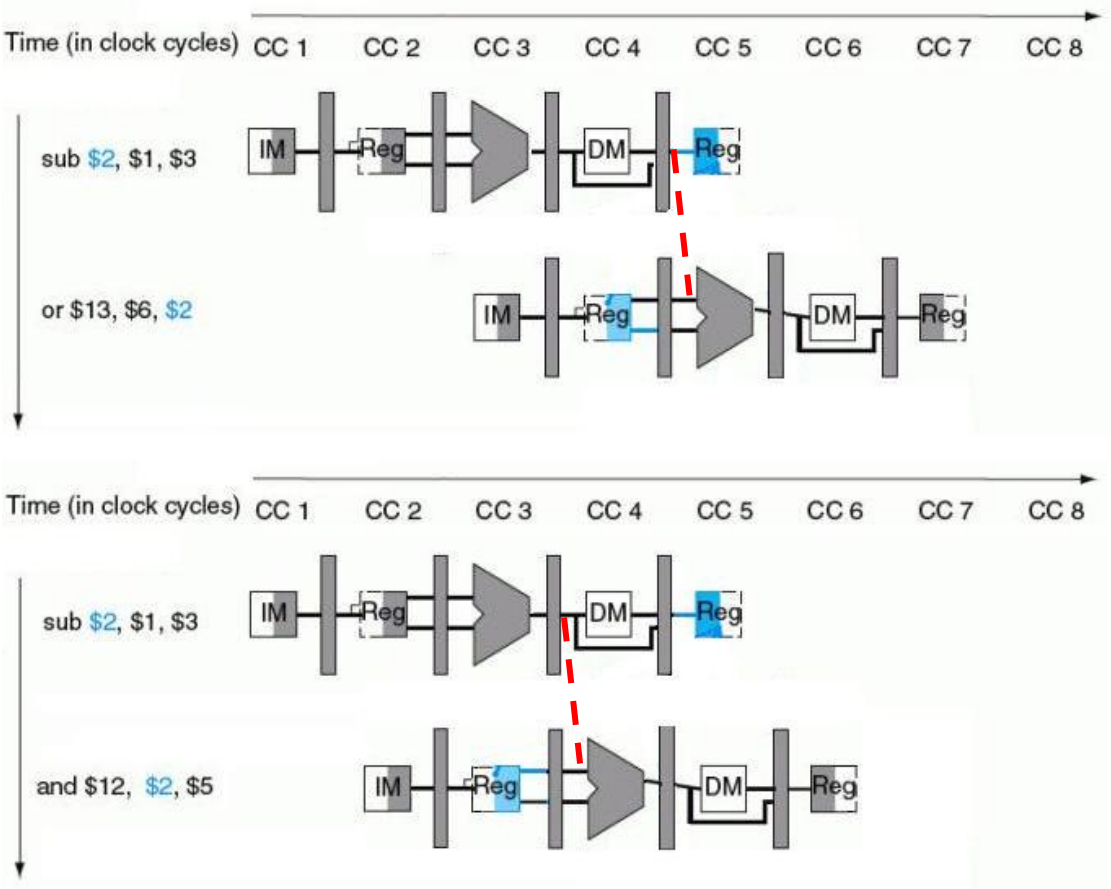
过于简单，直接在 top 中实现了。

m) forward 模块——旁路前推模块，实现转发

forward.v			
输入/输出	宽度	信号名	说明
input	[4:0]	Rs_EX	EX 段的 Rs
input	[4:0]	Rt_EX	EX 段的 Rt
input	[0:0]	RegWrite_MEM	MEM 段的 RegWrite
input	[0:0]	RegWrite_WB	WB 段的 RegWrite
input	[4:0]	RegWtaddr_MEM	MEM 段的 RegWtaddr
input	[4:0]	RegWtaddr_WB	WB 段的 RegWtaddr
output	[1:0]	RegRdout1Sel_Forward_EX	EX 段的 RegRdout1Sel_Forward
output	[1:0]	RegRdout2Sel_Forward_EX	EX 段的 RegRdout2Sel_Forward

这个模块实现从 DMem 输出到 ALU 输入以及 ALU 输出到 ALU 输入的转

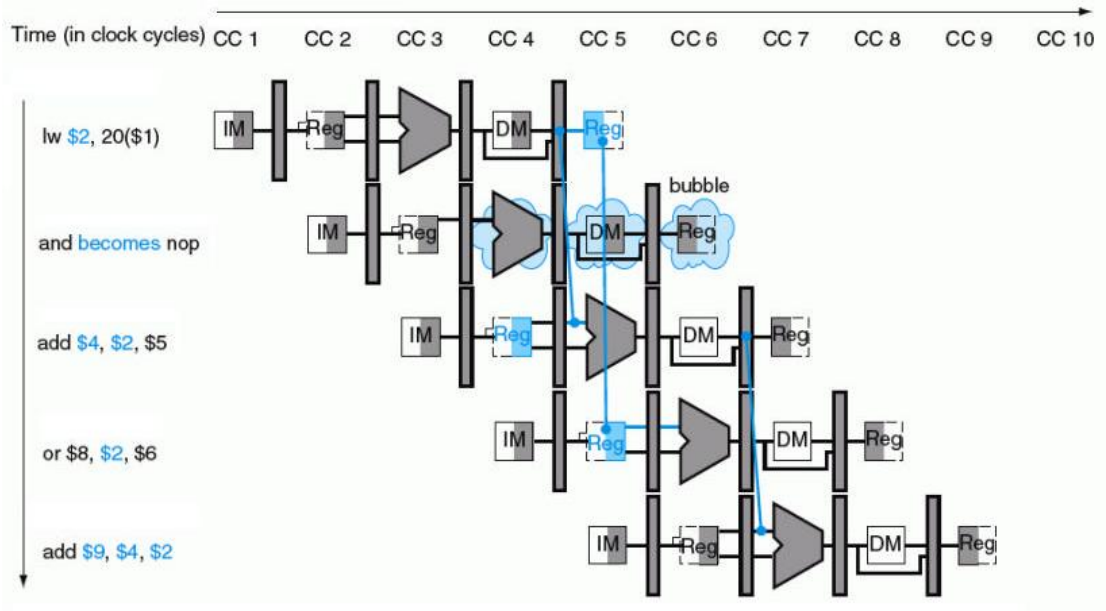
发。



n) hazard 模块——冒险检测模块，实现插入气泡

hazard.v			
输入/输出	宽度	信号名	说明
input	[4:0]	Rs_ID	ID 段的 Rs
input	[4:0]	Rt_ID	ID 段的 Rt
input	[4:0]	RegWtaddr_EX	EX 段的 RegWtaddr
input	[0:0]	DMemRead_EX	EX 段的 DMemRead
output	[0:0]	PCEn	允许 PC 更新，高电平有效
output	[0:0]	IF_ID_En	允许 IFID 更新，高电平有效
output	[0:0]	ID_EX_Flush	IDEX 清空，高电平有效

lw 之后 R-Type 需要使用访存结果，这在时间上是倒流的，不可能，只能插入一个气泡，清空 IDEX 寄存器，不更新 IFID 寄存器，不更新 PC。



o) control 模块——产生控制信号

control.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	rst	复位信号高电平有效
input	[5:0]	Op	
input	[4:0]	Rt	
input	[5:0]	Funct	
input	[1:0]	RsCMPRt	Rs 和 Rt 寄存器比较结果
input	[1:0]	RsCMPZero	Rs 寄存器和 0 比较结果
output	[1:0]	PCSrc	0:+4，1:Branch，2:J，3:JR
output	[0:0]	RegDst	0:RegWtaddr=rt，1:RegWtaddr=rd
output	[0:0]	ALUSrcASel	0:RegRdout1，1:ShamtZeroExtended
output	[1:0]	ALUSrcBSel	0:RegRdout2，1:IMMSignExtended，2:IMMZeroExtended
output	[4:0]	ALUControl	
output	[0:0]	DMemRead	1:En
output	[0:0]	DMemWrite	1:En
output	[0:0]	DMemtoReg	0:Aluout，1:DMemout
output	[0:0]	RegWrite	1:En

根据输入信号，判断指令类型，输出对应控制信号的取值。

p) debounce 模块——去抖动

debounce.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	in	输入信号，可能有抖动

output	[0:0]	out	输出信号，输入稳定后才变化

q) seg 模块——7 段数码管

seg.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[0:0]	rst_n	复位信号，低电平有效
input	[31:0]	data32	要显示的数据
output	[3:0]	sel	选择当前显示位，低电平有效
output	[6:0]	segments	选择显示的数码管，高电平有效

r) top 模块——实例化以上模块，连接各个信号

top.v			
输入/输出	宽度	信号名	说明
input	[0:0]	clk	时钟沿
input	[7:0]	sw	开关选择
output	[6:0]	seg7	7 段数码管选择
output	[3:0]	an	7 段数码管位选择
input	[0:0]	btns	按钮中间，按下复位

4、 信号命名规则

（1）信号名称由几个单词（或其简写）构成，每个单词首字母大写

（2）一般先写该信号对应的部件名，然后紧跟操作或接口名。

（3）如果该信号是另外一个信号的选择信号，则加上 Sel 后缀。

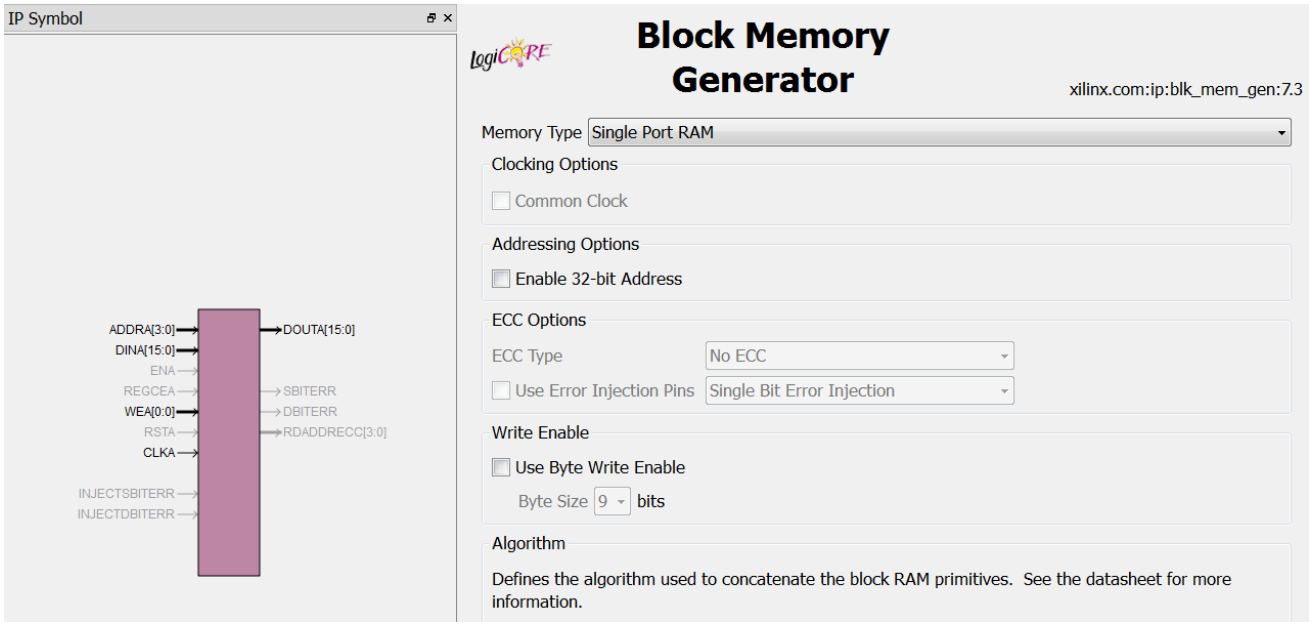
（4）如果该信号涉及在不同段之间传递，则在对应段的信号后加上所在段的名称。

（5）以上面方法命名基本不会引起混淆。

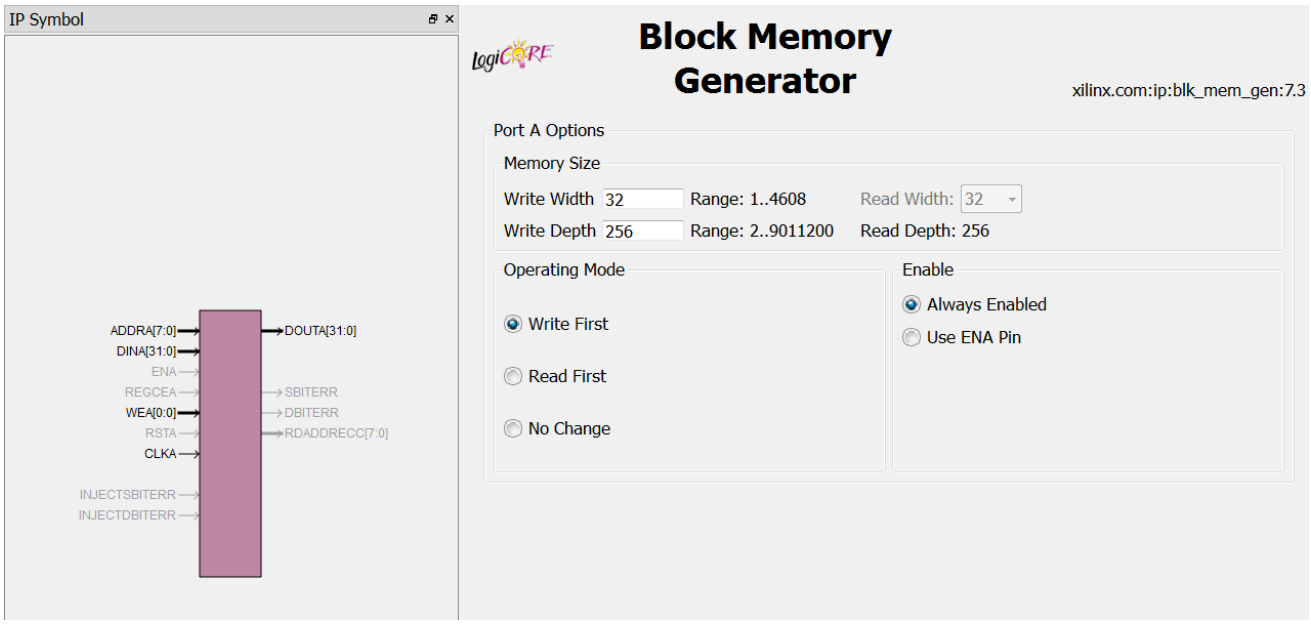
比如：RegRdout1_EX 表示寄存器文件读端口 1 读出的数据传递到 EX 段的信号。

5、 DMem 和 IMem 的初始化

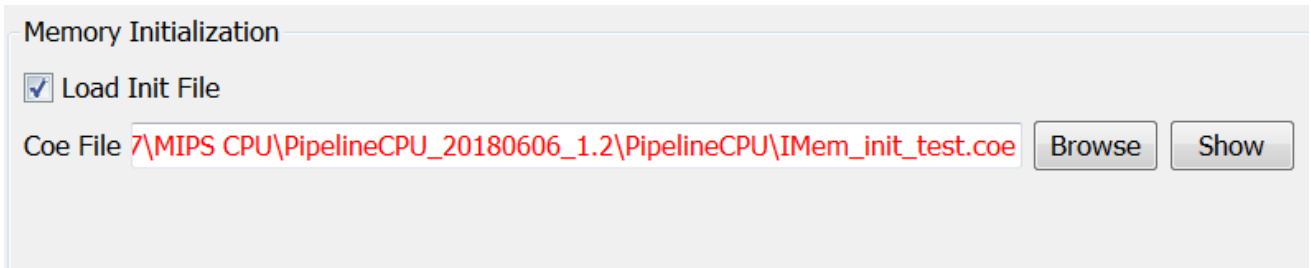
IP 核选择 Block Memory，设置为 Single Port RAM，



宽度为 32，深度为 256



选中 Load Init File 设置，选择初始化 coe 文件的路径，



其中 IMem_init_test.coe 文件内容为

```
memory_initialization_radix = 16;
memory_initialization_vector =
20110008
01d17020
21ceffff
11c00001
02200008
20100001
200f2000
20080010
00084040
00084082
ade80000
21ef0004
8de9fffc
01295004
3c090001
ade90000
21ef0004
adea0000
21ef0004
01505022
adea0000
21ef0004
010a4025
01004024
01084027
ade80000
21ef0004
01485826
000b5843
020b5807
adeb0000
21ef0004
000b5842
020b5806
000b5dc0
016b5821
adeb0000
21ef0004
01685823
adeb0000
21ef0004
256b0001
```

```
adeb0000
21ef0004
356b0001
316b0003
390b0001
adeb0000
21ef0004
020b602a
adec0000
21ef0004
020b602b
adec0000
21ef0004
296c0001
adec0000
21ef0004
2d6c0001
adec0000
21ef0004
018c680a
aded0000
21ef0004
0210680b
aded0000
21ef0004
200e0000
08000062
08000066
21ce0001
adee0000
21ef0004
1560fffb
21ce0001
adee0000
21ef0004
1000fff8
21ce0001
adee0000
21ef0004
1800fff8
21ce0001
adee0000
21ef0004
0560fff8
21ce0001
adee0000
```

```
21ef0004
0401fff8
21ce0001
adee0000
21ef0004
1e00fff8
21ce0001
adee0000
21ef0004
1560fff8
21ce0001
adee0000
21ef0004
116bfff8
21ce0001
adee0000
08000068;
```

DMem 的初始化类似该过程，其 coe 文件初始全为 0。

6、 测试的汇编代码

```
.text
    addi $17, $0, 8 #const $17=8
    add $14, $14, $17 #$14=8
    addi $14, $14, -1 #$14--
    beq $14, $0, begin
    jr $17 #jump to IMemAddr 8
begin:
    addi $16, $0, 1 #const 1
    addi $15, $0, 8192 #MemAddr

    addi $8, $0, 16 #$8=16
    sll $8, $8, 1 #$8=32
    srl $8, $8, 2 #$8=8

    sw $8, 0($15) #-------Save Value: 8------
    addi $15, $15, 4 #MemAddr++

    lw $9, -4($15) #$9=8(32'b00000000_00000001_00000000_00001000)
    sllv $10, $9, $9 #$10=2048(32'b00000000_00000000_00001000_00000000)
    lui $9, 1 #$9=32'b00000000_00000001_00000000_00000000

    sw $9, 0($15) #-------Save Value: 32'b00000000_00000001_00000000_00001000------
    addi $15, $15, 4 #MemAddr++

    sw $10, 0($15) #-------Save Value: 2048------
```

```
addi $15, $15, 4 #MemAddr++

sub $10, $10, $16 #$10=2047(32'b00000000_00000000_00000111_11111111)

sw $10, 0($15) #-------Save Value: 2047------
addi $15, $15, 4 #MemAddr++

or $8, $8, $10 #$8=2047(32'b111_11111111)
and $8, $8, $0 #$8=0
nor $8, $8, $8 #$8=2^32-1

sw $8, 0($15) #-------Save Value: 2^32-1------
addi $15, $15, 4 #MemAddr++

xor $11, $10, $8 #$11=32'b11111111_11111111_11111000_00000000
sra $11, $11, 1 #$11=32'b11111111_11111111_11111100_00000000
srav $11, $11, $16 #$11=32'b11111111_11111111_11111110_00000000

sw $11, 0($15) #-------Save Value: 32'b11111111_11111111_11111110_00000000------
addi $15, $15, 4 #MemAddr++

srl $11, $11, 1 #$11=32'b01111111_11111111_11111111_00000000
srlv $11, $11, $16 #$11=32'b00111111_11111111_11111111_10000000

sll $11, $11, 23 #$11=32'b01000000_00000000_00000000_00000000

addu $11, $11, $11 #$11=32'b10000000_00000000_00000000_00000000

sw $11, 0($15) #-------Save Value: 32'b10000000_00000000_00000000_00000000------
addi $15, $15, 4 #MemAddr++

subu $11, $11, $8 #$11=32'b10000000_00000000_00000000_00000001

sw $11, 0($15) #-------Save Value: 32'b10000000_00000000_00000000_00000001------
addi $15, $15, 4 #MemAddr++

addiu $11, $11, 1 #$11=32'b10000000_00000000_00000000_00000010

sw $11, 0($15) #-------Save Value: 32'b10000000_00000000_00000000_00000010------
addi $15, $15, 4 #MemAddr++

ori $11, $11, 1 #$11=32'b10000000_00000000_00000000_00000011
andi $11, $11, 3 #$11=32'b00000000_00000000_00000000_00000011
xori $11, $8, 1 #$11=32'b11111111_11111111_11111111_11111110

sw $11, 0($15) #-------Save Value: 32'b11111111_11111111_11111111_11111100------
```

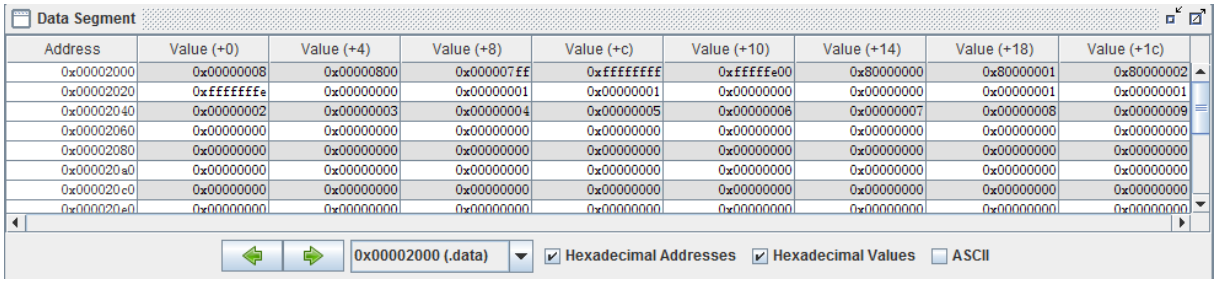
```
    addi $15, $15, 4 #MemAddr++

    slt $12, $16, $11 #$12=0

    sw $12, 0($15) #-------Save Value: 0------
    addi $15, $15, 4 #MemAddr++

    sltu $12, $16, $11 #$12=1

    sw $12, 0($15) #-------Save Value: 1------
    addi $15, $15, 4 #MemAddr++

    slti $12, $11, 1 #$12=1

    sw $12, 0($15) #-------Save Value: 1------
    addi $15, $15, 4 #MemAddr++

    sltiu $12, $11, 1 #$12=0

    sw $12, 0($15) #-------Save Value: 0------
    addi $15, $15, 4 #MemAddr++

    movz $13, $12, $12 #because $12=0, set $13=$12,

    sw $13, 0($15) #-------Save Value: 0------
    addi $15, $15, 4 #MemAddr++

    movn $13, $16, $16 #because $16=1(!=0), set $13=$16,

    sw $13, 0($15) #-------Save Value: 1------
    addi $15, $15, 4 #MemAddr++


    addi $14, $0, 0 #count
    j a
i:  j end


h:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    bne $11, $0, i #to i

g:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
```

```
    beq $0, $0, h #to h

f:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    blez $0, g #to g


e:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    bltz $11, f #to f

d:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    bgez $0, e #to e

c:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    bgtz $16, d #to d

b:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    bne $11,$0, c #to c

a:  addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------
    addi $15, $15, 4 #MemAddr++
    beq $11, $11, b #to b

end:addi $14, $14, 1 #count++
    sw $14, 0($15) #-------Save Value: $14------

out:j out
```

完整汇编代码如上，36 条指令均测试到了，每条语句的执行结果在该语句后面给出了注释，每测试几条指令就将结果写入内存，方便查看，并且 3 种相关有包括在上面的代码中。

实验结果

（1）汇编代码在 Mars 编译运行得到 Data 段为

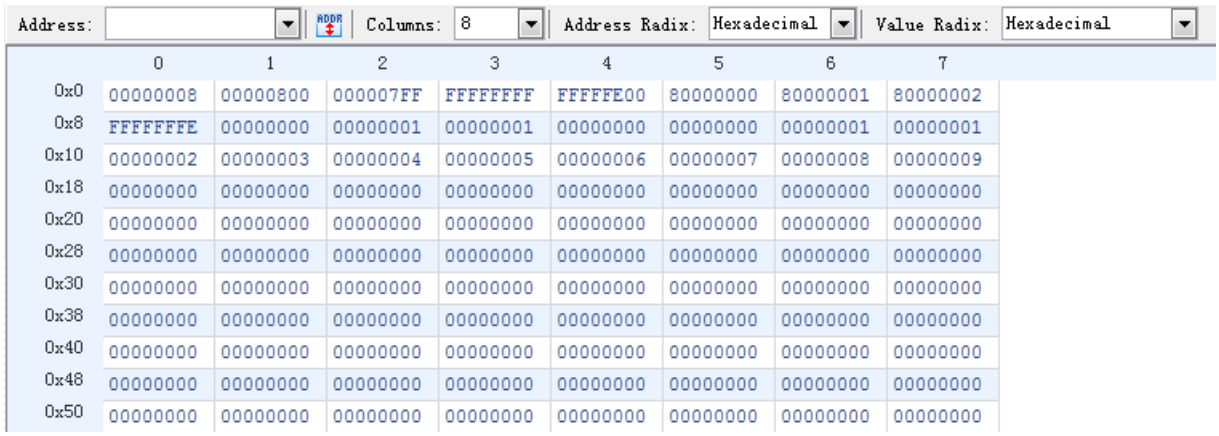


Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00002000	0x00000008	0x00000800	0x000007ff	0xffffffff	0xffffffe00	0x80000000	0x80000001	0x80000002
0x00002020	0xfffffffe	0x00000000	0x00000001	0x00000001	0x00000000	0x00000000	0x00000001	0x00000001
0x00002040	0x00000002	0x00000003	0x00000004	0x00000005	0x00000006	0x00000007	0x00000008	0x00000009
0x00002060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00002080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000020e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x00002000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

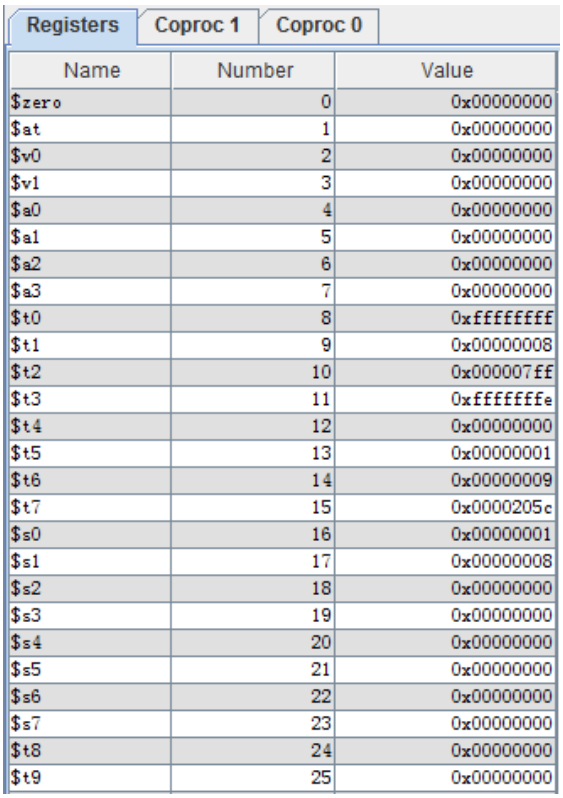
而 ISE 仿真结果显示 DMem 为



Address: Columns: 8 Address Radix: Hexadecimal Value Radix: Hexadecimal

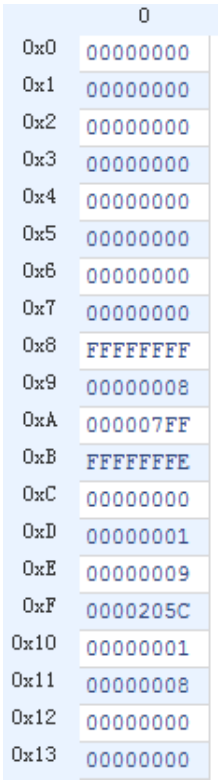
	0	1	2	3	4	5	6	7
0x0	00000008	00000800	000007FF	FFFFFFFF	FFFFFE00	80000000	80000001	80000002
0x8	FFFFFFFE	00000000	00000001	00000001	00000000	00000000	00000001	00000001
0x10	00000002	00000003	00000004	00000005	00000006	00000007	00000008	00000009
0x18	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x20	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x28	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x30	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x38	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x40	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x48	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x50	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

汇编代码在 Mars 编译运行得到结束时寄存器数据和仿真结果的 regfile 数据



Registers Coproc 1 Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0xffffffff
$t1	9	0x00000008
$t2	10	0x000007ff
$t3	11	0xfffffffe
$t4	12	0x00000000
$t5	13	0x00000001
$t6	14	0x00000009
$t7	15	0x0000205c
$s0	16	0x00000001
$s1	17	0x00000008
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000

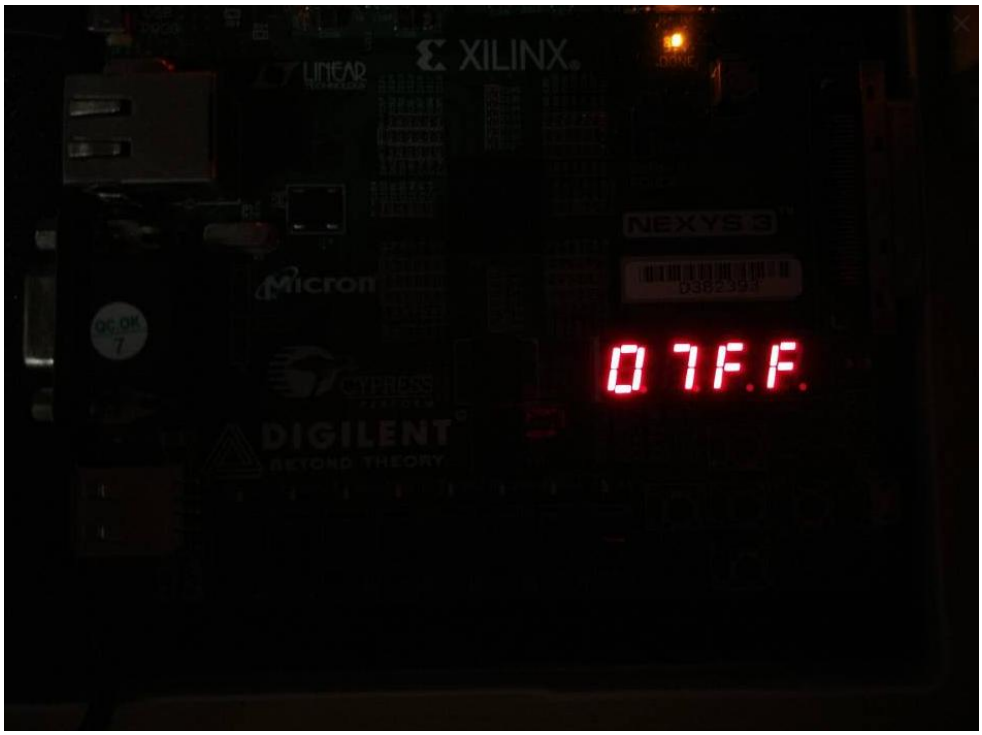


	0
0x0	00000000
0x1	00000000
0x2	00000000
0x3	00000000
0x4	00000000
0x5	00000000
0x6	00000000
0x7	00000000
0x8	FFFFFFFF
0x9	00000008
0xA	000007FF
0xB	FFFFFFFE
0xC	00000000
0xD	00000001
0xE	00000009
0xF	0000205C
0x10	00000001
0x11	00000008
0x12	00000000
0x13	00000000

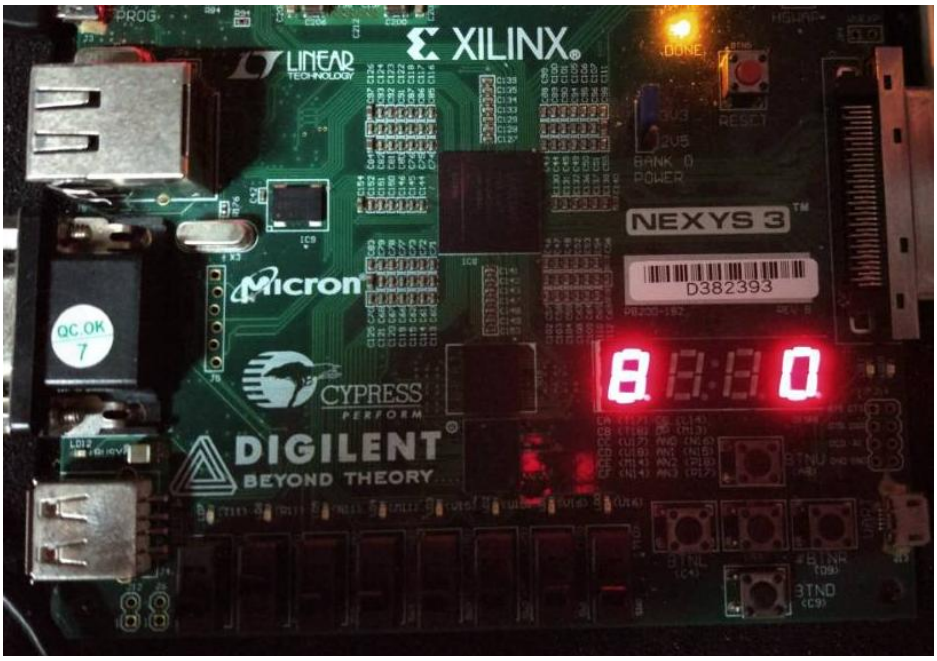
可见测试程序在编写的 MIPS CPU 上运行结果与 Mars 运行结果一致，并且符合上面注释的的预期结果。

（2）下载结果

按下按钮中键复位，拨动开关选择地址，数码管显示对应地址内存。



由于数据为 32 位，4 个数码管只能显示 16 位，为此还实现了动态显示，数字会向左移动循环显示。



附录:

一、 编写过程中遇到问题和一些发现

1、如果位拼接作为左值，单个变量也要花括号，如

{{RegDst},{ALUSrc},{MemtoReg},{RegWrite},{MemWrite},{Branch},{ALUOp[1:0]},{Jump}}=9'b100100100;

2、"=="会严格匹配每一位，比如 module 内 5 位数字比较（一个常量，一个 input），但外部传入的是 3 位的，高位自动拓展好像是 x，就导致“==”不成立，这是一个不好发现的问题。

3、仿真遇到高阻态，多半是这个信号不存在，经常是拼写错了，有时候 ise 检查不出来。

4、在 top 里的 wire 信号，如果通过 control 模块（或其他模块）进行控制，那么这个信号不能直接在 top 里赋初值，要到仿真模块里赋值，不然 control 里改变这个信号的值时，会变成不确定。

6、在 clk 边缘用<=对某信号进行赋值时，如果同时刻需要对该信号进行判断（如状态机，根据状态以完成对应操作），则判断时信号的值是本次改变前的。（非阻塞赋值的特性体现）

7、 srl 是逻辑右移，高位直接补零，而 sra 是算术右移，高位补符号，在 Verilog 中，既可以自行实现，也可以这样实现

```
$signed(alu_a) >>> alu_b
```

应该注意， >>> 是算术右移，但对于无符号数，高位仍然补 0，所以需要用 $signed()转为有符号数

```
$signed(alu_a) >> alu_b
```

这样写也不行，>>不具有算术右移的功能

8、reset 的时候应该把各段寄存器的各信号置 0，否则将导致 forward，hazard 模块输出的值不确定进而使相关的选择信号不确定（比如 PCEn 不确定，这样一开始就不能更新 PC）。

二、 模块源代码

top.v

```
module top(
    input clk,
    input [7:0] sw,
    output [6:0] seg7,
    output [3:0] an,
    input btnr,btnl,btns,btnd,btnu
    );

    //_后缀表示该信号所在的流水段

    wire ALUSrcASel_ID;
    wire ALUSrcASel_EX;
    wire [1:0] ALUSrcBSel_ID;//alu B 在 regout2 和 imm 之间选择
    wire [1:0] ALUSrcBSel_EX;
    wire [31:0] ALUSrcA_EX;
    wire [31:0] ALUSrcB_EX;
    wire [4:0] ALUControl_ID;
    wire [4:0] ALUControl_EX;
    wire [31:0] ALUResult_EX;
    wire [31:0] ALUResult_MEM;
    wire [31:0] ALUResult_WB;

    wire [1:0] RsCMPZero;
    wire [1:0] RsCMPRt;

    wire [31:0] IMMSignExtended_ID;
    wire [31:0] IMMSignExtended_EX;
    wire [31:0] IMMZeroExtended_ID;
    wire [31:0] IMMZeroExtended_EX;
    wire [31:0] ShamtZeroExtended_ID;
    wire [31:0] ShamtZeroExtended_EX;
```

```
wire [1:0] RegRdout1Sel_Forward_EX;//旁路单元产生的选择信号
wire [1:0] RegRdout2Sel_Forward_EX;
wire [31:0] RegRdout1_Forward_EX;//旁路数据
wire [31:0] RegRdout2_Forward_EX;

wire [4:0] RegRdaddr1_ID;
wire [31:0] RegRdout1_ID;
wire [31:0] RegRdout1_EX;
wire [4:0] RegRdaddr2_ID;
wire [31:0] RegRdout2_ID;
wire [31:0] RegRdout2_EX;
wire [4:0] RegWtaddr_ID;
wire [4:0] RegWtaddr_EX;
wire [4:0] RegWtaddr_MEM;
wire [4:0] RegWtaddr_WB;
wire [31:0] RegWtin_WB;
wire RegWrite_ID;
wire RegWrite_EX;
wire RegWrite_MEM;
wire RegWrite_WB;
wire RegDst_ID;

wire [31:0] IMemaddr;
wire [31:0] IMemout;

wire [31:0] DMemaddr_MEM;
wire [31:0] DMemin_MEM;
wire DMemRead_MEM;
wire [31:0] DMemout_MEM;
wire [31:0] DMemout_WB;
wire DMemWrite_MEM;
wire DMemtoReg_EX;
wire DMemtoReg_MEM;
wire DMemtoReg_WB;


wire [31:0] PC;
wire [31:0] PCPlus_IF;
wire [31:0] PCPlus_ID;
wire [31:0] PCPlus_EX;
wire [31:0] EPC;
wire [31:0] nextPC;
wire PCEn;
wire [1:0] PCSrc_ID;//Control 输出的，0:+4,1:Branch,2:J,3:JR
wire IF_ID_En;
```

```
    wire IF_ID_Flush;
    wire ID_EX_Flush;


    wire [31:0] PCJump_ID;
    wire [31:0] PCJR_ID;
    wire [31:0] PCBranch_ID;

    wire [31:0] Instr;
    wire [5:0] Funct;
    wire [4:0] Shamt;
    wire [15:0] IMM16;
    wire [4:0] Rd;
    wire [4:0] Rt;
    wire [4:0] Rs;
    wire [5:0] Op;
    wire [4:0] Rt_EX;//为了旁路判断
    wire [4:0] Rs_EX;//为了旁路判断

    wire [25:0] JumpIMM;
    wire [31:0] IMMSignExtendedShiftLeft2;


    wire btns_d;
    debounce debounce(clk,btns,btns_d);//中键去抖动
    reg rst;
    assign Led = rst;
    always @(posedge btns_d) rst=~rst;
    //--------------regfile_copy_DMem 是 DMem 的一份复制--------------
    wire [4:0] addr_show;
    wire [31:0] out_show;
    assign addr_show = sw;//开关输输入要显示的地址
    seg seg1(clk,1'b1,out_show,an,seg7);//显示数据
    regfile_copy_DMem regfile_copy_DMem(clk,~rst,addr_show,out_show,DMemaddr_MEM >>
2,DMemin_MEM,DMemWrite_MEM);
    //---------------方便显示内存-------------------------


    //======================IF=======================

    mux4 MUXPC(
        .sel(PCSrc_ID),
        .d0(PCPlus_IF),//+4 直接用 IF 的
        .d1(PCBranch_ID),
        .d2(PCJump_ID),
        .d3(PCJR_ID),
```

```
        .out(nextPC)
    );
    dff DFFPC(
        .clk(~clk),//下降沿更新 PC
        .en(PCEn),
        .rst(rst),
        .datain(nextPC),
        .dataout(PC)
    );

    alu ALUPCPlus(PC,4,5'd01,PCPlus_IF);

    assign IMemaddr = PC >> 2;//>>2 是因为这里 IMem 是每个地址存储 4 字节，和实际上的（一地址一字节）不一样
    IMem IMem(clk,1'b0,IMemaddr,32'b0,IMemout);//上升沿读指令


    //======================IFID=======================
    IFID IFID(
        .clk(~clk),
        .en(IF_ID_En),
        .flush(IF_ID_Flush || rst),
        .PCPlus_in(PCPlus_IF),
        .IMemout_in(IMemout),
        .PCPlus_out(PCPlus_ID),
        .IMemout_out(Instr)
    );
    //======================ID=========================
    assign JumpIMM = Instr[25:0];
    assign Funct = Instr[5:0];
    assign Shamt = Instr[10:6];
    assign IMM16 = Instr[15:0];
    assign Rd = Instr[15:11];
    assign Rt = Instr[20:16];
    assign Rs = Instr[25:21];
    assign Op = Instr[31:26];

    //*******Control*******
    control control(
        //in
        .clk(clk),
        .rst(rst),
        .Op(Op),
        .Rt(Rt),
        .Funct(Funct),
        .RsCMPRt(RsCMPRt),
```

```
    .RsCMPZero(RsCMPZero),
    //out
    .PCSrc(PCSrc_ID),
    //ID
    .RegDst(RegDst_ID),
    //EX
    .ALUSrcASel(ALUSrcASel_ID),
    .ALUSrcBSel(ALUSrcBSel_ID),
    .ALUControl(ALUControl_ID),
    //MEM
    .DMemRead(DMemRead_ID),
    .DMemWrite(DMemWrite_ID),
    //WB
    .DMemtoReg(DMemtoReg_ID),
    .RegWrite(RegWrite_ID)
);

//*******Control*******

assign RegRdaddr1_ID = Rs;
assign RegRdaddr2_ID = Rt;
mux #(5) MUXRegWtaddr(RegDst_ID,Rt,Rd,RegWtaddr_ID);


assign ShamtZeroExtended_ID = {{27{1'b0}},Shamt};
assign IMMSignExtended_ID = {{16{IMM16[15]}},IMM16};
assign IMMZeroExtended_ID = {{16{1'b0}},IMM16};

assign IMMSignExtendedShiftLeft2 = IMMSignExtended_ID << 2;
alu BranchALU(PCPlus_ID,IMMSignExtendedShiftLeft2,5'd01,PCBranch_ID);
assign PCJump_ID = {{PCPlus_ID[31:28]},{{2'b00,JumpIMM}<<2}};
assign PCJR_ID = RegRdout1_ID;


assign IF_ID_Flush = (PCSrc_ID != 2'b00);//有跳转则清空 IF_ID 寄存器

regfile regfile(clk,~rst,RegRdaddr1_ID,RegRdout1_ID,RegRdaddr2_ID,
        RegRdout2_ID,RegWtaddr_WB,RegWtin_WB,RegWrite_WB);

compare compare1(RegRdout1_ID,RegRdout2_ID,RsCMPRt);//for beq,bne
compare compare2(RegRdout1_ID,0,RsCMPZero);//for movz,movn,blez,bgtz,bltz,bgez


hazard hazard(Rs,Rt,RegWtaddr_EX,DMemRead_EX,PCEn,IF_ID_En,ID_EX_Flush);
//======================IDEX=======================
IDEX IDEX(
```

```
    .clk(~clk),
    .en(1'b1),
    .flush(ID_EX_Flush || rst),
    //data
    //in
    .PCPlus_in(PCPlus_ID),
    .RegRdout1_in(RegRdout1_ID),
    .RegRdout2_in(RegRdout2_ID),
    .IMMSignExtended_in(IMMSignExtended_ID),
    .IMMZeroExtended_in(IMMZeroExtended_ID),
    .ShamtZeroExtended_in(ShamtZeroExtended_ID),
    .Rs_in(Rs),
    .Rt_in(Rt),
    .RegWtaddr_in(RegWtaddr_ID),
    //out
    .PCPlus_out(PCPlus_EX),
    .RegRdout1_out(RegRdout1_EX),
    .RegRdout2_out(RegRdout2_EX),
    .IMMSignExtended_out(IMMSignExtended_EX),
    .IMMZeroExtended_out(IMMZeroExtended_EX),
    .ShamtZeroExtended_out(ShamtZeroExtended_EX),
    .Rs_out(Rs_EX),
    .Rt_out(Rt_EX),
    .RegWtaddr_out(RegWtaddr_EX),

    //control sign
    //in
    .RegDst_in(RegDst_ID),
    .ALUSrcASel_in(ALUSrcASel_ID),
    .ALUSrcBSel_in(ALUSrcBSel_ID),
    .ALUControl_in(ALUControl_ID),
    .DMemRead_in(DMemRead_ID),
    .DMemWrite_in(DMemWrite_ID),
    .DMemtoReg_in(DMemtoReg_ID),
    .RegWrite_in(RegWrite_ID),
    //out
    .RegDst_out(RegDst_EX),
    .ALUSrcASel_out(ALUSrcASel_EX),
    .ALUSrcBSel_out(ALUSrcBSel_EX),
    .ALUControl_out(ALUControl_EX),
    .DMemRead_out(DMemRead_EX),
    .DMemWrite_out(DMemWrite_EX),
    .DMemtoReg_out(DMemtoReg_EX),
    .RegWrite_out(RegWrite_EX)
);
//=======================EX=======================
```

```
forward forward(Rs_EX,Rt_EX,RegWrite_MEM,RegWrite_WB,RegWtaddr_MEM,
       RegWtaddr_WB,RegRdout1Sel_Forward_EX,RegRdout2Sel_Forward_EX);
mux4 MUXRegRdout1FW(RegRdout1Sel_Forward_EX,RegRdout1_EX,RegWtin_WB,
       ALUResult_MEM,0,RegRdout1_Forward_EX);//forward
mux4 MUXRegRdout2FW(RegRdout2Sel_Forward_EX,RegRdout2_EX,RegWtin_WB,
       ALUResult_MEM,0,RegRdout2_Forward_EX);//forward
mux MUXALUSrcA(ALUSrcASel_EX,RegRdout1_Forward_EX,ShamtZeroExtended_EX,
       ALUSrcA_EX);
mux4 MUXALUSrcB(ALUSrcBSel_EX,RegRdout2_Forward_EX,IMMSignExtended_EX,
       IMMZeroExtended_EX,0,ALUSrcB_EX);
alu alu(ALUSrcA_EX,ALUSrcB_EX,ALUControl_EX,ALUResult_EX);
//======================EXMEM=======================
EXMEM EXMEM(
    .clk(~clk),
    .en(1'b1),
    .flush(rst),
    //data
    //in
    .ALUResult_in(ALUResult_EX),
    .DMemin_in(RegRdout2_Forward_EX),
    .RegWtaddr_in(RegWtaddr_EX),
    //out
    .ALUResult_out(ALUResult_MEM),
    .DMemin_out(DMemin_MEM),
    .RegWtaddr_out(RegWtaddr_MEM),
    //control sign
    //in
    .DMemRead_in(DMemRead_EX),
    .DMemWrite_in(DMemWrite_EX),
    .DMemtoReg_in(DMemtoReg_EX),
    .RegWrite_in(RegWrite_EX),
    //out
    .DMemRead_out(DMemRead_MEM),
    .DMemWrite_out(DMemWrite_MEM),
    .DMemtoReg_out(DMemtoReg_MEM),
    .RegWrite_out(RegWrite_MEM)
);
//======================MEM=======================
assign DMemaddr_MEM = ALUResult_MEM;
DMem DMem(clk,DMemWrite_MEM,DMemaddr_MEM >> 2,DMemin_MEM,DMemout_MEM);
//======================MEMWB=======================
MEMWB MEMWB(
    .clk(~clk),
    .en(1'b1),
    .flush(rst),
    //data
```

```
        //in
        .ALUResult_in(ALUResult_MEM),
        .DMemout_in(DMemout_MEM),
        .RegWtaddr_in(RegWtaddr_MEM),
        //out
        .ALUResult_out(ALUResult_WB),
        .DMemout_out(DMemout_WB),
        .RegWtaddr_out(RegWtaddr_WB),
        //control sign
        //in
        .DMemtoReg_in(DMemtoReg_MEM),
        .RegWrite_in(RegWrite_MEM),
        //out
        .DMemtoReg_out(DMemtoReg_WB),
        .RegWrite_out(RegWrite_WB)
    );
    //=======================WB=======================

    mux MUXDMemtoReg(DMemtoReg_WB,ALUResult_WB,DMemout_WB,RegWtin_WB);

endmodule
```

alu.v

```
`define A_NOP 5'd00 //nop
`define A_ADD 5'd01 //signed_add
`define A_SUB 5'd02 //signed_sub
`define A_AND 5'd03 //and
`define A_OR  5'd04 //or
`define A_XOR 5'd05 //xor
`define A_NOR 5'd06 //nor
`define A_ADDU 5'd07 //unsigned_add
`define A_SUBU 5'd08 //unsigned_sub
`define A_SLT 5'd09 //slt
`define A_SLTU 5'd10 //unsigned_slt
`define A_SLL 5'd11 //sll
`define A_SRL 5'd12 //srl
`define A_SRA 5'd13 //sra
`define A_MOV 5'd14 //movz,movn
`define A_LUI 5'd15 //lui
module alu(
    input [31:0] alu_a,//无符号型的，如果有负数，是以补码存储
    input [31:0] alu_b,
    input [4:0] alu_op,
    output reg [31:0] alu_out
```

```
    );
    always@(*)
        case (alu_op)
            `A_NOP: alu_out = 0;
            `A_ADD: alu_out = alu_a + alu_b;
            `A_SUB: alu_out = alu_a - alu_b;
            `A_AND: alu_out = alu_a & alu_b;
            `A_OR : alu_out = alu_a | alu_b;
            `A_XOR: alu_out = alu_a ^ alu_b;
            `A_NOR: alu_out = ~(alu_a | alu_b);
            `A_ADDU: alu_out = alu_a + alu_b;
            `A_SUBU: alu_out = alu_a - alu_b;
            `A_SLT: //a<b(signed) return 1 else return 0;
               begin
                   if(alu_a[31] == alu_b[31]) alu_out = (alu_a < alu_b) ? 32'b1 : 32'b0;
//对于不加 signed 的变量类型，运算和比较视为无符号，但依然可以存储有符号数，这里相当于自行根据首位判断
//首位相等，即同号情况，直接比较，如果同正，后面 31 位大的，原数就大，如果同负，后面 31 位（补码）大的，依然是原数大
                   else alu_out = (alu_a[31] < alu_b[31]) ? 32'b0 : 32'b1;
//异号情况，直接比较符号
               end
            `A_SLTU: alu_out = (alu_a < alu_b) ? 32'b1 : 32'b0;
            `A_SLL: alu_out = alu_b << alu_a;
            `A_SRL: alu_out = alu_b >> alu_a;
            `A_SRA: alu_out = $signed(alu_b) >>> alu_a;
//使用>>>为算术右移，高位补符号，应该注意，如果是无符号数，>>>仍是逻辑右移，故应该$signed
            `A_MOV: alu_out = alu_b;
//原样输出，相当于 reg[rt],mov 本不需要通过 alu，但因为是 RType 格式，故统一
            `A_LUI: alu_out = alu_b << 16;
           default: ;
        endcase
endmodule
```

regfile.v

```
module regfile(
    input clk,
    input rst_n,
    input [4:0] rAddr1,//读地址 1
    output [31:0] rDout1,//读数据 1
    input [4:0] rAddr2,//读地址 2
    output [31:0] rDout2,//读数据 2
    input [4:0] wAddr,//写地址
    input [31:0] wDin,//写数据
```

```
    input wEna//写使能

);
    reg [31:0] data [0:31];
    integer i;
    assign rDout1=data[rAddr1];//读 1
    assign rDout2=data[rAddr2];//读 2
    always@(posedge clk or negedge rst_n)//写和复位
        if(~rst_n)
        begin
            for(i=0; i<32; i=i+1) data[i]<=0;
        end
        else
        begin
            if(wEna)
                data[wAddr]<=wDin;
        end
endmodule
```

dff.v

```
module dff #(parameter WIDTH = 32) ( //Data Flip-Flop
    input clk,
    input en,
    input rst,
    input [WIDTH-1:0] datain,
    output reg [WIDTH-1:0] dataout
    );
    always@(posedge clk)
    begin
        if(rst)
            dataout <= 0;
        else if(en)
            dataout <= datain;
    end
endmodule
```

mux.v

```
module mux #(parameter WIDTH = 32)( //2 路选择器
    input sel,
    input [WIDTH-1:0] d0,
    input [WIDTH-1:0] d1,
    output [WIDTH-1:0] out
```

```
    );
    assign out = (sel == 1'b1 ? d1 : d0);
endmodule
```

mux4.v

```
module mux4 #(parameter WIDTH = 32)( //4路选择器
    input [1:0] sel,
    input [WIDTH-1:0] d0,
    input [WIDTH-1:0] d1,
     input [WIDTH-1:0] d2,
     input [WIDTH-1:0] d3,
    output reg [WIDTH-1:0] out
    );
    always@(*)
        case(sel)
            2'b00: out=d0;
            2'b01: out=d1;
            2'b10: out=d2;
            2'b11: out=d3;
            default:;
        endcase
endmodule
```

IFID.v

```
module IFID(
    input clk,
    input en,
    input flush,
    input [31:0] PCPlus_in,
    input [31:0] IMemout_in,
    output [31:0] PCPlus_out,
    output [31:0] IMemout_out
    );
     dff dff1(clk,en,flush,PCPlus_in,PCPlus_out);
     dff dff2(clk,en,flush,IMemout_in,IMemout_out);
endmodule
```

IDEX.v

```
module IDEX(
    input clk,
```

```
input en,
input flush,//flush for stall or start
input [31:0] PCPlus_in,
input [31:0] RegRdout1_in,
input [31:0] RegRdout2_in,
input [31:0] IMMSignExtended_in,
input [31:0] IMMZeroExtended_in,
input [31:0] ShamtZeroExtended_in,
input [4:0] Rs_in,
input [4:0] Rt_in,
input [4:0] RegWtaddr_in,
output [31:0] PCPlus_out,
output [31:0] RegRdout1_out,
output [31:0] RegRdout2_out,
output [31:0] IMMSignExtended_out,
output [31:0] IMMZeroExtended_out,
output [31:0] ShamtZeroExtended_out,
output [4:0] Rs_out,
output [4:0] Rt_out,
output [4:0] RegWtaddr_out,
//control
input RegDst_in,
input ALUSrcASel_in,
input [1:0] ALUSrcBSel_in,
input [4:0] ALUControl_in,
input DMemRead_in,
input DMemWrite_in,
input DMemtoReg_in,
input RegWrite_in,
output RegDst_out,
output ALUSrcASel_out,
output [1:0] ALUSrcBSel_out,
output [4:0] ALUControl_out,
output DMemRead_out,
output DMemWrite_out,
output DMemtoReg_out,
output RegWrite_out
);
dff dff1(clk,en,flush,PCPlus_in,PCPlus_out);
dff dff2(clk,en,flush,RegRdout1_in,RegRdout1_out);
dff dff3(clk,en,flush,RegRdout2_in,RegRdout2_out);
dff dff4(clk,en,flush,IMMSignExtended_in,IMMSignExtended_out);
dff dff5(clk,en,flush,IMMZeroExtended_in,IMMZeroExtended_out);
dff dff6(clk,en,flush,ShamtZeroExtended_in,ShamtZeroExtended_out);
dff #(5) dff7(clk,en,flush,Rs_in,Rs_out);
dff #(5) dff8(clk,en,flush,Rt_in,Rt_out);
```

```
    dff #(5) dff9(clk,en,flush,RegWtaddr_in,RegWtaddr_out);

    dff #(1) dff10(clk,en,flush,RegDst_in,RegDst_out);
    dff #(1) dff11(clk,en,flush,ALUSrcASel_in,ALUSrcASel_out);
    dff #(2) dff12(clk,en,flush,ALUSrcBSel_in,ALUSrcBSel_out);
    dff #(5) dff13(clk,en,flush,ALUControl_in,ALUControl_out);
    dff #(1) dff14(clk,en,flush,DMemRead_in,DMemRead_out);
    dff #(1) dff15(clk,en,flush,DMemWrite_in,DMemWrite_out);
    dff #(1) dff16(clk,en,flush,DMemtoReg_in,DMemtoReg_out);
    dff #(1) dff17(clk,en,flush,RegWrite_in,RegWrite_out);
endmodule
```

EXMEM.v

```
module EXMEM(
    input clk,
     input en,
     input flush,
     input [31:0] ALUResult_in,
     input [31:0] DMemin_in,
     input [4:0] RegWtaddr_in,
     output [31:0] ALUResult_out,
     output [31:0] DMemin_out,
     output [4:0] RegWtaddr_out,
     //control
     input DMemRead_in,
     input DMemWrite_in,
     input DMemtoReg_in,
     input RegWrite_in,
     output DMemRead_out,
     output DMemWrite_out,
     output DMemtoReg_out,
     output RegWrite_out
    );
     dff dff1(clk,en,flush,ALUResult_in,ALUResult_out);
     dff dff2(clk,en,flush,DMemin_in,DMemin_out);
     dff #(5) dff3(clk,en,flush,RegWtaddr_in,RegWtaddr_out);

     dff #(1) dff14(clk,en,flush,DMemRead_in,DMemRead_out);
     dff #(1) dff15(clk,en,flush,DMemWrite_in,DMemWrite_out);
     dff #(1) dff16(clk,en,flush,DMemtoReg_in,DMemtoReg_out);
     dff #(1) dff17(clk,en,flush,RegWrite_in,RegWrite_out);

endmodule
```

MEMWB.v

```
module MEMWB(
    input clk,
    input en,
    input flush,
    input [31:0] ALUResult_in,
    input [31:0] DMemout_in,
    input [4:0] RegWtaddr_in,
    output [31:0] ALUResult_out,
    output [31:0] DMemout_out,
    output [4:0] RegWtaddr_out,
    //control
    input DMemtoReg_in,
    input RegWrite_in,
    output DMemtoReg_out,
    output RegWrite_out
   );
    dff dff1(clk,en,flush,ALUResult_in,ALUResult_out);
    dff dff2(clk,en,flush,DMemout_in,DMemout_out);
    dff #(5) dff3(clk,en,flush,RegWtaddr_in,RegWtaddr_out);
    dff #(1) dff16(clk,en,flush,DMemtoReg_in,DMemtoReg_out);
    dff #(1) dff17(clk,en,flush,RegWrite_in,RegWrite_out);
endmodule
```

compare.v

```
`define LESS 2'b00
`define EQUAL 2'b01
`define GREATER 2'b10
module compare(//为了判断分支
    input signed [31:0] a,
    input signed [31:0] b,
    output reg [1:0] res
   );
    always @(*)
      if(a == b) res = 2'b01;
      else if(a < b) res = 2'b00;
      else if(a > b) res = 2'b10;
endmodule
```

forward.v

```
module forward(//前推
    input [4:0] Rs_EX,
    input [4:0] Rt_EX,
    input RegWrite_MEM,
    input RegWrite_WB,
    input [4:0] RegWtaddr_MEM,
    input [4:0] RegWtaddr_WB,
    output reg [1:0] RegRdout1Sel_Forward_EX,
    output reg [1:0] RegRdout2Sel_Forward_EX
    );
    always @(*) begin
        RegRdout1Sel_Forward_EX[0] = RegWrite_WB && (RegWtaddr_WB != 0) &&
(RegWtaddr_MEM != Rs_EX) && (RegWtaddr_WB == Rs_EX);
        RegRdout1Sel_Forward_EX[1] = RegWrite_MEM && (RegWtaddr_MEM != 0) &&
(RegWtaddr_MEM == Rs_EX);
        RegRdout2Sel_Forward_EX[0] = RegWrite_WB && (RegWtaddr_WB != 0) &&
(RegWtaddr_MEM != Rt_EX) && (RegWtaddr_WB == Rt_EX);
        RegRdout2Sel_Forward_EX[1] = RegWrite_MEM && (RegWtaddr_MEM != 0) &&
(RegWtaddr_MEM == Rt_EX);
    end
endmodule
```

hazard.v

```
module hazard(//上一条指令是 LW 指令且当前指令 ID 级读的是同一个寄存器,则插入 bubble
     input [4:0] Rs_ID,
     input [4:0] Rt_ID,
     input [4:0] RegWtaddr_EX,
     input DMemRead_EX,
     output PCEn,
     output IF_ID_En,
     output ID_EX_Flush
    );
    assign ID_EX_Flush = ((RegWtaddr_EX == Rs_ID) || (RegWtaddr_EX == Rt_ID)) &&
DMemRead_EX;//条件成立则为 1，清空
    assign IF_ID_En = ~ID_EX_Flush;//条件成立则为 0，保持
    assign PCEn = ~ID_EX_Flush;//条件成立则为 0，保持
endmodule
```

control.v

```
module control(
```

```
    input clk,rst,
    input [5:0] Op, //instr[31:26]
    input [4:0] Rt, //instr[20:16]

    input [5:0] Funct,//instr[5:0]
    input [1:0] RsCMPRt,
    input [1:0] RsCMPZero,
    output reg [1:0] PCSrc, //0:+4,1:Branch,2:J,3:JR
    //ID
    output reg RegDst,//0:RegWtaddr=rt,1:RegWtaddr=rd
    //EX
    output reg ALUSrcASel,//0:RegRdout1,1:ShamtZeroExtended
    output reg [1:0] ALUSrcBSel, //0:RegRdout2,1:IMMSignExtended,2:IMMZeroExtended
    output reg [4:0] ALUControl,
    //MEM
    output reg DMemRead,//1:En
    output reg DMemWrite,//1:En
    //WB
    output reg DMemtoReg,//0:Aluout,1:DMemout
    output reg RegWrite//1:En
    );

    reg [1:0] tmpsrc;

    always @(*)
    begin
    if(rst)
        begin

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_00},{`A_NOP},{4'b0001}};
        end
    else
        case(Op)
            6'b000000: //R-Type
                case(Funct)
                    //SLL 的 rs rt rd shamt 全 0 时是 nop，本来 nop 没有对应的
                    6'b000000: //SLL,注意 Alu_a 来自 Shamt 的无符号拓展,即 ALUSrcASel=1，下面两个同理

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_1_00},{`A_SLL},{4'b0001}};
                    //6'b000001: //MOVCI
                    6'b000010: //SRL

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_1_00},{`A_SRL},{4'b0001}};
```

```
                6'b000011: //SRA

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_1_00},{`A_SRA},{4'b0001}};
                6'b000100: //SLLV，注意 Alu_a 来自 reg[rs],即 ALUSrcASel=0，下面两个同理

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SLL},{4'b0001}};
                //6'b000101: //*
                6'b000110: //SRLV

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SRL},{4'b0001}};
                6'b000111: //SRAV

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SRA},{4'b0001}};
                6'b001000: //JR

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b11_z_z_zz},{`A_NOP},{4'b00z0}};
                //6'b001001: //JALR

                6'b001010: //MOVZ,如果 reg[rs]=0 则 reg[rd]=reg[rt],此时 RsCMPZero=01,所以
RegWrite=RsCMPZero[0]

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_MOV},{3'b000},{RsCMPZero[0]}};
                6'b001011: //MOVN,如果 reg[rs]!=0 则 reg[rd]=reg[rt],此时 RsCMPZero=00 或 10,
所以 RegWrite=~RsCMPZero[0]

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_MOV},{3'b000},{~RsCMPZero[0]}};

                6'b100000: //ADD

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_ADD},{4'b0001}};
                6'b100001: //ADDU

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_ADDU},{4'b0001}};
                6'b100010: //SUB

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SUB},{4'b0001}};
```

```
                6'b100011: //SUBU

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SUB},{4'b0001}};
                6'b100100: //AND

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_AND},{4'b0001}};
                6'b100101: //OR

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_OR},{4'b0001}};
                6'b100110: //XOR

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_XOR},{4'b0001}};
                6'b100111: //NOR

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_NOR},{4'b0001}};

                6'b101010: //SLT

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SLT},{4'b0001}};
                6'b101011: //SLTU

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_1_0_00},{`A_SLTU},{4'b0001}};
            endcase
          6'b000001:
            case(Rt)
                6'b00000: //BLTZ,Reg[rs]<0 则跳转
                begin
                    if(RsCMPZero == `LESS) tmpsrc = 2'b01; else tmpsrc = 2'b00;

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{tmpsrc},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
                end
                6'b00001: //BGEZ,Reg[rs]>=0 则跳转
                begin
                    if(RsCMPZero == `GREATER || RsCMPZero == `EQUAL) tmpsrc = 2'b01; else tmpsrc
= 2'b00;

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{tmpsrc},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
```

```
                end
            endcase
        6'b000010: //J,无条件跳转

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b10_z_z_zz},{`A_NOP},{4'b00z0}};
        //6'b000011: //JAL
        6'b000100: //BEQ,Reg[rs]==Reg[rt]则跳转,RsCMPRt=01(==),则 PCSrc=01，否则 PCSrc=00(不
跳转)

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{{1'b0},{RsCMPRt[0]}},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
        6'b000101: //BNE,Reg[rs]!=Reg[rt]则跳转,RsCMPRt=00(<)或 10(>),则 PCSrc=01，否则
PCSrc=00(不跳转)

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{{1'b0},{~RsCMPRt[0]}},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
        6'b000110: //BLEZ,Reg[rs]<=0 则跳转
            begin
            if(RsCMPZero == `LESS || RsCMPZero == `EQUAL) tmpsrc = 2'b01; else tmpsrc = 2'b00;

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{tmpsrc},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
            end
        6'b000111: //BGTZ,Reg[rs]>0 则跳转
            begin
            if(RsCMPZero == `GREATER) tmpsrc = 2'b01; else tmpsrc = 2'b00;

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{tmpsrc},{4'bz_z_zz},{`A_NOP},{4'b00z0}};
            end
        6'b001000: //ADDI,注意 RegDst=0,AluBSrcSel=01(IMMSignExtended),下面三个同理

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_01},{`A_ADD},{4'b0001}};
        6'b001001: //ADDIU

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_01},{`A_ADDU},{4'b0001}};
        6'b001010: //SLTI

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_01},{`A_SLT},{4'b0001}};
        6'b001011: //SLTIU
```

```
{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_01},{`A_SLTU},{4'b0001}};
          6'b001100: //ANDI,注意 RegDst=0,AluBSrcSel=10(IMMZeroExtended),下面三个同理

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_10},{`A_AND},{4'b0001}};
          6'b001101: //ORI

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_10},{`A_OR},{4'b0001}};
          6'b001110: //XORI

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_10},{`A_XOR},{4'b0001}};
          6'b001111: //LUI

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_10},{`A_LUI},{4'b0001}};

          6'b100011: //LW,注意 RegDst=0(写到
Reg[rt]),AluBSrcSel=01(IMMSignExtended),DMemtoReg=1(来自 DMem),

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_0_0_01},{`A_ADD},{4'b1011}};

          6'b101011: //SW,注意 RegDst=x(不写 Reg),AluBSrcSel=01(IMMSignExtended)

{{PCSrc},{RegDst},{ALUSrcASel},{ALUSrcBSel},{ALUControl},{DMemRead},{DMemWrite},{DMemtoReg
},{RegWrite}}={{6'b00_z_0_01},{`A_ADD},{4'b01z0}};

          default: ;
       endcase
    end
endmodule
```

debounce.v

```
module debounce(//去抖动
     input clk,
    input in,
    output reg out=0

    );
    reg [31:0] cnt=0;
```

```
    always@(posedge clk)
        begin
            if(in!=out)
                begin
                    cnt=cnt+1;

                    if(cnt==100000)
                    begin
                        out=~out;
                        cnt=0;
                    end

                end
            else cnt=0;

        end
endmodule
```

seg.v

```
module seg(
    input clk,
    input rst_n,
    input [31:0] data32,
    output reg [3:0] sel,
    output reg [6:0] segments
    );
    integer clk_25=0;//4 位数码管循环显示用
    integer clk_50000000=0;//2hz，移动显示用
    reg [1:0] cnt;
    reg [3:0] cnt2;
    reg [15:0] data16;//data32 的 16bit
    reg [3:0] data4;//data16 的 4bit
    reg [3:0] empty;//空白位
    always@(*)//组合逻辑，控制七段数码管
        begin
            if(!rst_n)
                segments = 7'b000_0000;
            else
                case(data4)
                    0: segments =      ~7'b011_1111;//0
                    1: segments =      ~7'b000_0110;//1
                    2: segments =      ~7'b101_1011;//2
                    3: segments =      ~7'b100_1111;//3
                    4: segments =      ~7'b110_0110;//4
```

```
                5: segments =      ~7'b110_1101;//5
                6: segments =      ~7'b111_1101;//6
                7: segments =      ~7'b000_0111;//7
                8: segments =      ~7'b111_1111;//8
                9: segments =      ~7'b110_1111;//9
                10:segments =      ~7'b111_0111;//A
                11:segments =      ~7'b111_1100;//b
                12:segments =      ~7'b011_1001;//C
                13:segments =      ~7'b101_1110;//d
                14:segments =      ~7'b111_1001;//E
                15:segments =      ~7'b111_0001;//F
                default: segments = 7'b000_0000; // required
            endcase
    end

always@(posedge clk)//时序逻辑，产生位选择信号段选择信号
    begin
        //if(!rst_n)
            //cnt = 2'b00;
        //else
            if(clk_25==400000)
                begin
                    clk_25=0;
                    cnt = cnt + 2'b01;
                end
            else
                clk_25=clk_25+1;

            if(clk_50000000==50000000)//
                begin
                    clk_50000000=0;
                    cnt2=cnt2+1;
                    if(cnt2==4'b1010) cnt2=4'b0000;
                end
            else
                clk_50000000=clk_50000000+1;
        end


always@(*)//组合逻辑，选择当前显示段
    begin
        case(cnt2)
            4'b0000:begin data16={8'bzzzzzzzz,data32[31:24]}; empty=4'b1100; end
            4'b0001:begin data16={4'bzzzz,data32[31:20]}; empty=4'b1000; end
            4'b0010:begin data16=data32[31:16]; empty=4'b0000; end
            4'b0011:begin data16=data32[27:12]; empty=4'b0000; end
```

```
            4'b0100:begin data16=data32[23:8]; empty=4'b0000; end
            4'b0101:begin data16=data32[19:4]; empty=4'b0000; end
            4'b0110:begin data16=data32[15:0]; empty=4'b0000; end
            4'b0111:begin data16={data32[11:0],4'bzzzz}; empty=4'b0001; end
            4'b1000:begin data16={data32[7:0],8'bzzzzzzzz}; empty=4'b0011; end
            4'b1001:begin data16={data32[3:0],8'bzzzzzzzz,data32[31:28]};
empty=4'b0110; end
            default:;
        endcase
    end

    always@(*)//组合逻辑，选择当前显示位
    begin
        case(cnt)
            2'b00:sel=4'b1110 | empty;
            2'b01:sel=4'b1101 | empty;
            2'b10:sel=4'b1011 | empty;
            2'b11:sel=4'b0111 | empty;
            default:sel=4'b1110;
        endcase
    end

    always@(*)//组合逻辑，选择当前显示位的数据
    begin
        case(cnt)
            2'b00:data4=data16[3:0];
            2'b01:data4=data16[7:4];
            2'b10:data4=data16[11:8];
            2'b11:data4=data16[15:12];
            default:data4=16'b0;
        endcase
    end
endmodule
```

test.v

```
module test;
    // Inputs
    reg clk;
    reg rst;
    // Instantiate the Unit Under Test (UUT)
    top uut (
        .clk(clk),
        .rst(rst),
    );
```

```
    initial begin
        // Initialize Inputs
        clk = 1;

        rst = 0;
        #100;

        rst = 1;

        // Wait 100 ns for global reset to finish
        #100;
        clk=~clk;
        #10;
        clk=~clk;
      rst = 0;

        forever begin
            #10;
            clk=~clk;
        end

        // Add stimulus here
    end

endmodule
```