同济大学计算机系

计算机组成原理课程设计实验报告

题目：31 条 MIPS 指令 CPU 设计

学　　号 _____

姓　　名 _____

专　　业 _____

授课老师 _____

日期：

一、 实验环境部署与硬件配置说明

实验环境：vivado、Mars。

硬件配置：NEXYS A7 开发板。

本实验通过 vivado 实现 31 条 mips 指令 CPU 的 verilog 程序，然后在 FPGA 的 7 段数码管部分进行显示指令以及最终程序运行结果。

二、 实验的总体结构

1. 31 条 MIPS 指令 CPU 的总体结构

本实验设计的是一个支持 31 条 mips 指令的简化 CPU，CPU 的总体结构包括译码器、控制器、算数逻辑单元 ALU、程序计数器 PC 以及通用寄存器堆。同时，为了保持数据的通路，在 CPU 的上面还有顶层模块 sccomp_dataflow，这里面包括 CPU，以及指令存储器和数据存储器。

2. 指令的具体分析

对于本次实验要求实现的指令，可以分为 3 类：R 型指令、I 型指令和 J 型指令。

2.1 R 型指令

R 型指令的格式为：

Bit	31~26	25~21	20~16	15~11	10~6	5~0
R-type	op	rs	rt	rd	shamt	func

其中，R 型指令的前 6 位 op 部分相同，均为 000000，因此需要靠后 5 位的 func 来进行区分这些指令。

要求实现的指令有：

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	
R-type	op	rs	rt	rd	shamt	func	
add	000000	rs	rt	rd	0	100000	add $1,$2,$3
addu	000000	rs	rt	rd	0	100001	addu $1,$2,$3
sub	000000	rs	rt	rd	0	100010	sub $1,$2,$3
subu	000000	rs	rt	rd	0	100011	subu $1,$2,$3
and	000000	rs	rt	rd	0	100100	and $1,$2,$3
or	000000	rs	rt	rd	0	100101	or $1,$2,$3
xor	000000	rs	rt	rd	0	100110	xor $1,$2,$3
nor	000000	rs	rt	rd	0	100111	nor $1,$2,$3
slt	000000	rs	rt	rd	0	101010	slt $1,$2,$3
sltu	000000	rs	rt	rd	0	101011	sltu $1,$2,$3
sll	000000	0	rt	rd	shamt	000000	sll $1,$2,10
srl	000000	0	rt	rd	shamt	000010	srl $1,$2,10
sra	000000	0	rt	rd	shamt	000011	sra $1,$2,10
sllv	000000	rs	rt	rd	0	000100	sllv $1,$2,$3
srlv	000000	rs	rt	rd	0	000110	srlv $1,$2,$3
srav	000000	rs	rt	rd	0	000111	srav $1,$2,$3
jr	000000	rs	0	0	0	001000	jr $31

这里面大部分指令的过程为：

$$rd <- rs\ [操作符]\ rt$$

即将 rs 和 rt 所对应寄存器的值操作完后的结果，存到 rd 对应寄存器中。如：

add：将 rs 寄存器存的值和 rt 寄存器存的值相加，结果存入 rd 寄存器中。

而对于 sll、srl、sra 三条指令，则是：

$$rd <- rt\ [操作符]\ shamt$$

即将 rt 的值 shamt 操作完之后的结果存到 rd 中。如：

sll：将 rt 的值左移 shamt 位，结果存入 rd 中。

由于 shamt 在指令中只为 5 位，而 ALU 中操作位数为 32 位，因此这里需要对 shamt 进行位数扩展。

接着则是 jr 指令，jr 指令不经过 ALU，而是对 PC 进行操作，其指令为：

$$pc <- rs$$

即将 rs 存的值传递给 pc。

2.2 I 型指令

I 型指令的格式为：

Bit	31~26	25~21	20~16	15~0
I-type	op	rs	rt	immediate

I 型指令的 op 位各不相同，可以通过 op 位不同来区分。

要求实现的指令有：

Bit #	31..26	25..21	20..16	15..0	
I-type	op	rs	rt	immediate	
addi	001000	rs	rt	Immediate(- ~ +)	addi $1,$2,100
addiu	001001	rs	rt	Immediate(- ~ +)	addiu $1,$2,100
andi	001100	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
ori	001101	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
xori	001110	rs	rt	Immediate(0 ~ +)	andi $1,$2,10
lw	100011	rs	rt	Immediate(- ~ +)	lw $1,10($2)
sw	101011	rs	rt	Immediate(- ~ +)	sw $1,10($2)
beq	000100	rs	rt	Immediate(- ~ +)	beq $1,$2,10
bne	000101	rs	rt	Immediate(- ~ +)	bne $1,$2,10
slti	001010	rs	rt	Immediate(- ~ +)	slti $1,$2,10
sltiu	001011	rs	rt	Immediate(- ~ +)	sltiu $1,$2,10
lui	001111	00000	rt	Immediate(- ~ +)	Lui $1, 10

这其中大部分指令的过程为：

$$rt <- rs\ [操作符]\ immediate$$

即将 rs 和立即数 immediate 的操作结果存到 rt 寄存器中。如：

addi：将 rs 的值与立即数相加，结果存入 rt 中。

不过，lw、sw 指令稍微特殊，因为这两个指令涉及到从数据存储器中读出和写入数据，他们指令中的立即数是作为地址偏移量存在的。lw 指令为：

$$rt <- DMEM[rs + immediate]$$

即从 DMEM 数据存储器中读取 rs+immediate 作为地址对应的数据，加载到 rt 中。

而 sw 指令为：

$$DMEM[rs + immediate] <- rt$$

即将 rt 的值存入 DMEM 中 rs+immediate 地址位置中。

beq、bne 两条指令和 PC 有关，对应到条件是否跳转上面。

beq 指令是：

$$if(rs == rt)\ then\ \ pc <- pc + 4 + immediate << 2$$

bne 指令则是：

$$\text{if}(rs != rt)\ \text{then}\ \ pc <- pc + 4 + immediate << 2$$

beq 对应的是：如果 rs 等于 rt 的值，则进行跳转。bne 则相反，是 rs 不等于 rt 值时进行跳转。

此外，还有 lui 指令，这个指令是置高位立即数，即将 16 位立即数放到 rt 寄存器的高 16 位，低 16 位则置 0，指令为：

$$rt <- immediate * 65536$$

2.3 J 型指令

J 型指令的格式为：

Bit	31~26	25~0
J-type	op	index

同样，J 型指令的 op 位各不相同，通过 op 位即可区分。

要求实现的指令有：

Bit #	31..26	25..0	
J-type	op	Index	
j	000010	address	j 10000
jal	000011	address	jal 10000

j 和 jal 执行的跳转指令为：

$$pc <- \{(pc + 4)[31:28], index, 0, 0\}$$

这使得 pc 跳转到对应的地址处，然后执行该处的指令。同时，jal 相比 j，需要将跳转后的后一条指令的 pc 传递给 31 号寄存器，即：

$$\$31 <- pc + 4$$

分析完所有的指令后，我们就可以通过不同指令之间的 op 或 func 不同来区分这些指令，从而实现译码器的编写。同时，不同指令涉及到的寄存器不相同，有些需要改变 pc，有些需要从数据存储器中读取数据，这些都需要做出相应的判断，从而调用不同的模块来运行 CPU，这就需要译码器和控制器的相互配合了。

三、 总体架构部件的解释说明

1. CPU 总体结构

CPU 需要对传入的指令进行译码，并执行指令对应的操作。因此，CPU 作为整个管理模块，需要包含的其他模块有 Decoder 译码器、Controller 控制器、ALU 逻辑运算单元、PC 寄存器和 Regfiles 通用寄存器堆。同时，CPU 也需要获取 IMEM、DMEM 的数据和地址，同样也需要有相关的接口来控制这些操作。CPU31 的相关接口如下：

类型	接口	注释
input	clk	时钟信号
input	rst	复位
input	ena	使能
input[31:0]	IR	指令
input[31:0]	dmem_rdate	从 DMEM 读取的数据
output	dmem_ena	DMEM 使能

input	dmem_w	DMEM 写信号
output	dmem_r	DMEM 读信号
output[31:0]	dmem_addr	DMEM 地址
output[31:0]	dmem_wdata	写入 DMEM 的数据
output[31:0]	PC	PC

CPU31 中虽然有 Controller 用来传递控制信号，但是本身还需要对数据进行管理，进而决定数据传输给哪个模块。因此，除了包含几个元件的模块后，还需要对数据进行一些处理，这其中包括：对 shamt、immediate 等原本不到 32 位的数据进行数位扩展；从 Controller 获取选择信号后，使用 selector4 选择器，选择出传递给 ALU 的运算数据。因此，在 CPU31 的中间变量板块，还有 extend 和 selector 两部分。

1.1 Decoder

Decoder 完成的为：分析传入的指令 IR，给出对应执行哪一条指令，并且对应的寄存器 rs、rt、rd 以及相关的立即数等。相关接口如下：

类型	接口	注释
input[31:0]	IR	指令
output(31 个)	Add~Jal	31 个具体的指令
output[4:0]	rsc	rs 的地址
output[4:0]	rtc	rt 的地址
output[4:0]	rdc	rd 的地址（特指指令写入寄存器的地址）
output[4:0]	shamt	shamt
output[15:0]	immediate	immediate
output[25:0]	index	index

1.2 Controller

Controller 则是从 Decoder 中接收到具体执行哪条指令，从而给出相关的控制信号：寄存器写信号、ALU 输入选择信号、ALU 控制信号 ALUC、DMEM 的读写信号以及 PC 输入的选择信号。相关接口如下：

类型	接口	注释
input(31 个)	Add~Jal	31 个具体的指令
input	zero	0 标志位
output	we	寄存器写信号
output[1:0]	mux_A	ALU 输入 A 选择信号
output[1:0]	mux_B	ALU 输入 B 选择信号

output[3:0]	ALUC	ALU 控制信号
output	DM_w	DMEM 写信号
output	DM_r	DMEM 读信号
output[1:0]	mux_PC	PC 选择信号

1.3 ALU

ALU 是 CPU 的逻辑运算单元，需要执行 CPU 中基础的运算，相关的基础运算和控制信号的关系如下：

运算	ALUC[3:0]
ADD	0000
ADDU	0001
SUB	0010
SUBU	0011
AND	0100
OR	0101
XOR	0110
NOR	0111
SLT	1000
SLTU	1001
SLL	1010
SRL	1011
SRA	1100
LUI	1101

虽然说 mips 指令不止上述的运算，但是其他指令是在 ALU 的结果之后再进行判断，是否进行跳转和对 DMEM 进行读写，所以 ALU 需要满足的基础运算只需要上表这些。相关接口如下：

类型	接口	注释
input[31:0]	A	输入数 A
input[31:0]	B	输入数 B
input[3:0]	ALUC	ALU 控制信号
output[31:0]	C	输出结果 C
output	zero	0 标志位
output	carry	进位标志位
output	negative	负数标志位
output	overflow	溢出标志位

1.4 PCreg

PCreg 即 PC 寄存器，存的结果是 PC 的输出值，用于输出给 IMEM 读取指令。相关接口如下：

类型	接口	注释
input	clk	时钟信号
input	rst	复位
input	ena	使能
input[31:0]	PC_in	输入 PC 值
output[31:0]	PC_out	输出 PC 值

1.5 Regfiles

Regfiles 即通用寄存器堆，由于实现的 mips31 条指令需要 32 个通用寄存器，且数据为 32 位，所以给 Regfiles 的数组为[31:0] array_reg[31:0]。同时，寄存器堆需要接收对应寄存器的地址 rsc、rtc、rdc 以及相关的数据 rs、rt、rd。相关接口如下：

类型	接口	注释
input	clk	时钟信号
input	rst	复位
input	ena	使能
input	we	寄存器写信号
input[4:0]	rsc	rs 寄存器地址
input[4:0]	rtc	rt 寄存器地址
input[4:0]	rdc	rd 寄存器的地址（特指指令写入寄存器的地址）
output[31:0]	rs	rs 的值
output[31:0]	rt	rt 的值
input[31:0]	rd	写入 rd 的值

2. sccomp_dataflow 总体结构

sccomp_dataflow 则是大框架，包含了 CPU、IMEM、DMEM 三个大模块。sccomp_dataflow 的输入有 clk_in、reset，两个分别是时钟信号和复位信号，输出则为 inst 和 pc，用来输出当前时钟的 CPU 所对应的 PC 和指令。

这个模块主要是用于进行数据的流通，将 IMEM、DMEM 和 CPU 连接起来，这样 CPU 可以读取 IMEM 中的指令，同时可以从 DMEM 中存取数据。相关接口为：

类型	接口	注释
input	clk_in	时钟信号
input	reset	复位
output[31:0]	inst	指令
output[31:0]	pc	pc

2.1 IMEM

IMEM 模块主要是用于从 IP 核中调取指令，因此需要给出输入的地址以及输出的指令。由于这一模块不涉及到数据的改变，IP 核为 ROM，因此不需要设置时钟信号。相关接口为：

类型	接口	注释
input[10:0]	IM_addr	指令地址
output[31:0]	IM_data	指令（数据）

2.2 DMEM

DMEM 则是数据存储器，用来实现 CPU 从 DMEM 中读取或写入数据，因此需要时钟、使能、写入读取等信号。同时，读取的数据和写入的数据都有地址，所以还需要输入对应的地址。相关接口为：

类型	接口	注释
input	clk	时钟信号
input	ena	使能信号
input	DM_w	DMEM 写信号
input	DM_r	DMEM 读信号
input[10:0]	DM_addr	DMEM 对应的地址
output[31:0]	DM_wdata	写入 DMEM 的数据
output[31:0]	DM_rdata	从 DMEM 读取的数据

3. board_show 总体模块

上面的 sccomp_dataflow 已经可以完成数据的流通，实现模拟仿真，但是，如果要实现下板，则需要添加七段数码管的模块和时钟分频器。因此，在 sccomp_dataflow 模块之上还有 board_show 模块进行下板显示。board_show 相关接口如下：

类型	接口	注释
input	clk_in	时钟信号
input	reset	复位
input	result	选择输出结果
output[7:0]	o_seg	单个数码管显示结果
output[7:0]	o_sel	数码管选择

四、 实验仿真过程

·CPU 的仿真过程

CPU 的仿真过程主要是通过测试不同的汇编程序来实现的。在给出的相关文档中，有不同指令的测试文件，我们将这些汇编程序转化为 coe 文件，加载到 IP 核中，这样 CPU 程序即可以进行仿真测试。仿真过程可以对比 sccomp_dataflow 输出的 inst 和 pc，和汇编程序每一步的指令是否相同。同时，test_bench 也会将测试过程中每一步指令对应的寄存器结果打印出来，Mars 也可以看到程序运行过程中每一步的寄存器结果，因此通过对比这两个结果，

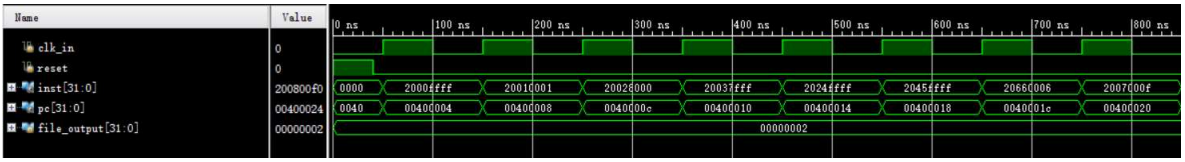
我们可以给出 CPU 程序是否正确，仿真结果是否准确。

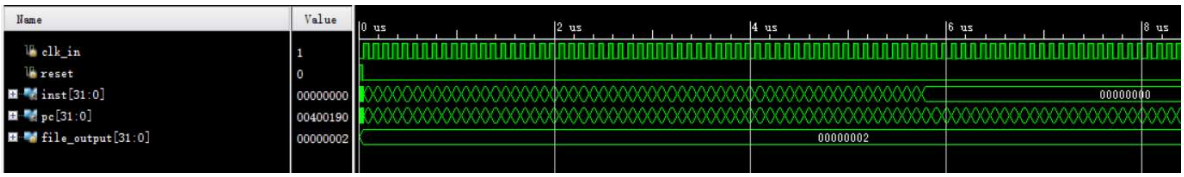
五、 实验仿真的波形图及某时刻寄存器值的物理意义

仿真部分，我们分别给出 addi、lwsw 以及 beq 的结果，用来举例分析。其他指令则在测试过程经过测试，限于篇幅不全部说明。

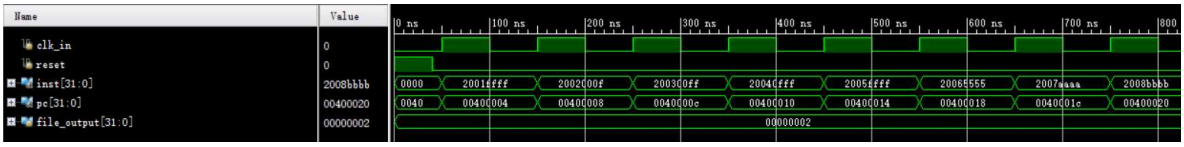
1. 仿真波形图

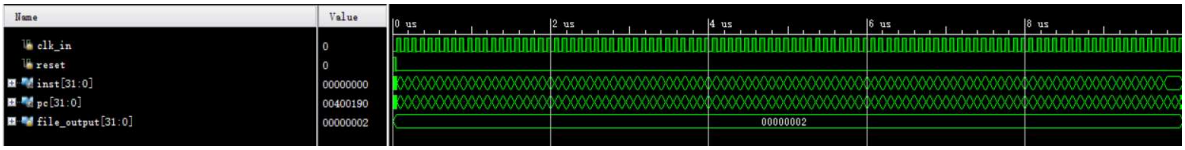
addi：



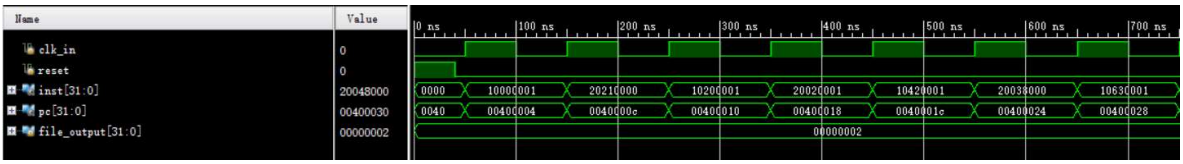


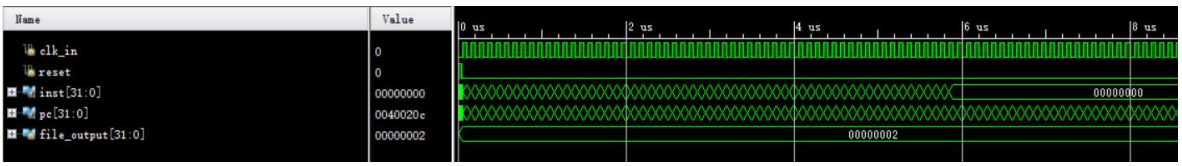
lwsw：





beq：





可以看到，在不同的指令下，sccomp_dataflow 均可以根据时钟的变化给出对应的 inst 和 pc，同时程序结束，inst 不会再变化，这与我们的预期是符合的。

2. 某时刻寄存器值的物理意义

addi：

```
pc: 004000e8
instr: 00000000
regfile0: 00000000
regfile1: 00000001
regfile2: ffff8000
regfile3: 00007fff
regfile4: 00000000
regfile5: ffff7fff
regfile6: 00008005
regfile7: 0000000f
regfile8: 000000f0
regfile9: 00000f00
regfile10: fffff000
regfile11: 0000aaaa
regfile12: ffff5554
regfile13: 0000001e
regfile14: 000001fe
regfile15: 00001ffe
regfile16: 00000011
regfile17: fffff012
regfile18: ffff8013
regfile19: 00007014
regfile20: 00000015
regfile21: 00000226
regfile22: 00000ac7
regfile23: ffffdc18
regfile24: ffffa019
regfile25: ffffb01a
regfile26: ffffe01b
regfile27: 00007d1c
regfile28: ffffcbbd
regfile29: ffffdbde
regfile30: 0000788f
regfile31: 00007790
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0xffff8000
$v1	3	0x00007fff
$a0	4	0x00000000
$a1	5	0xffff7fff
$a2	6	0x00008005
$a3	7	0x0000000f
$t0	8	0x000000f0
$t1	9	0x00000f00
$t2	10	0xfffff000
$t3	11	0x0000aaaa
$t4	12	0xffff5554
$t5	13	0x0000001e
$t6	14	0x000001fe
$t7	15	0x00001ffe
$s0	16	0x00000011
$s1	17	0xfffff012
$s2	18	0xffff8013
$s3	19	0x00007014
$s4	20	0x00000015
$s5	21	0x00000226
$s6	22	0x00000ac7
$s7	23	0xffffdc18
$t8	24	0xffffa019
$t9	25	0xffffb01a
$k0	26	0xffffe01b
$k1	27	0x00007d1c
$gp	28	0xffffcbbd
$sp	29	0xffffdbde
$fp	30	0x0000788f
$ra	31	0x00007790
pc		0x004000e8

lwsw：

pc: 00400188
instr: 00000000
regfile0: 00000000
regfile1: 0000000f
regfile2: 000000ff
regfile3: 00000fff
regfile4: ffffffff
regfile5: 00005555
regfile6: ffffaaaa
regfile7: ffffbbbb
regfile8: ffffcccc
regfile9: ffffdddd
regfile10: ffffeeee
regfile11: ffffffff
regfile12: ffff0000
regfile13: 000f0000
regfile14: 00ff0000
regfile15: 0fff0000
regfile16: ffff0000
regfile17: 55550000
regfile18: aaaa0000
regfile19: bbbb0000
regfile20: cccc0000
regfile21: dddd0000
regfile22: eeee0000
regfile23: ffff0000
regfile24: ffff0000
regfile25: 000f0000
regfile26: 00ff0000
regfile27: 0fff0000
regfile28: ffff0000
regfile29: 55550000
regfile30: 00000000
regfile31: 0000000f

Name	Number	Value
$zero	0	0x00000000
$at	1	0x0000000f
$v0	2	0x000000ff
$v1	3	0x00000fff
$a0	4	0xffffffff
$a1	5	0x00005555
$a2	6	0xffffaaaa
$a3	7	0xffffbbbb
$t0	8	0xffffcccc
$t1	9	0xffffdddd
$t2	10	0xffffeeee
$t3	11	0xffffffff
$t4	12	0xffff0000
$t5	13	0x000f0000
$t6	14	0x00ff0000
$t7	15	0x0fff0000
$s0	16	0xffff0000
$s1	17	0x55550000
$s2	18	0xaaaa0000
$s3	19	0xbbbb0000
$s4	20	0xcccc0000
$s5	21	0xdddd0000
$s6	22	0xeeee0000
$s7	23	0xffff0000
$t8	24	0xffff0000
$t9	25	0x000f0000
$k0	26	0x00ff0000
$k1	27	0x0fff0000
$gp	28	0xffff0000
$sp	29	0x55550000
$fp	30	0x00000000
$ra	31	0x0000000f
pc		0x00400188

beq：

pc: 004000ec
instr: 2001ffff
regfile0: 00000000
regfile1: ffffffff
regfile2: 00000001
regfile3: ffff8000
regfile4: ffff8000
regfile5: 00007fff
regfile6: 00000551
regfile7: 00000552
regfile8: 00000006
regfile9: 00000004
regfile10: 00000002
regfile11: 00000002
regfile12: 00000000
regfile13: 00000000
regfile14: 00000000
regfile15: 00000000
regfile16: 00000000
regfile17: 00000000
regfile18: 00000000
regfile19: 00000000
regfile20: 00000000
regfile21: 00000000
regfile22: 00000000
regfile23: 00000000
regfile24: 00000000
regfile25: 00000000
regfile26: 00000000
regfile27: 00000000
regfile28: 00000000
regfile29: 00000000
regfile30: 00000000
regfile31: ffffffff

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000001
$v1	3	0xffff8000
$a0	4	0xffff8000
$a1	5	0x00007fff
$a2	6	0x00000551
$a3	7	0x00000552
$t0	8	0x00000006
$t1	9	0x00000004
$t2	10	0x00000002
$t3	11	0x00000002
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0xffffffff
pc		0x004000ec

每个指令上面是仿真程序 test_bench 打印的寄存器的结果，下面是对应 PC、inst 时刻 Mars 运行的结果。

我们在程序结束和程序中间对比仿真和 Mars，可以看到，对应的寄存器结果均相同，这也说明了我们 CPU 程序的正确性。

六、 实验验算数学模型及算法程序

·比萨塔摔鸡蛋游戏模型

两个同学在可变换层数的比萨塔上摔鸡蛋，一个同学秘密设定同一批鸡蛋耐摔值；另一个同学在指定层高的比萨塔拿着鸡蛋往下摔，用最少的摔次数和摔破的鸡蛋数求出鸡蛋的耐摔值。假定在耐摔值的楼层及其下面楼层，鸡蛋摔不破，可以重复使用，否则鸡蛋摔破。要求模型的算法输出包括：摔的总次数、摔的总鸡蛋数、最后摔的鸡蛋是否摔破。请使用 MIPS 指令汇编设计该验证模型的算法，并利用编译器生成 MIPS 指令集可执行目标程序。

我们在第一个实验中已经对摔鸡蛋这个程序编写了 c 程序和汇编。但是在第一个实验中，由于我们给了输入和输出，需要用到 31 条指令中不存在的指令，因此为了便于 31 条指令的 CPU 执行程序，我们将输入写定到对应寄存器中，同时将结果输出到寄存器中。这样当查看最终结果时，我们可以通过显示存程序结果的寄存器，从而显示摔鸡蛋程序的结果。相关汇编文件如下：

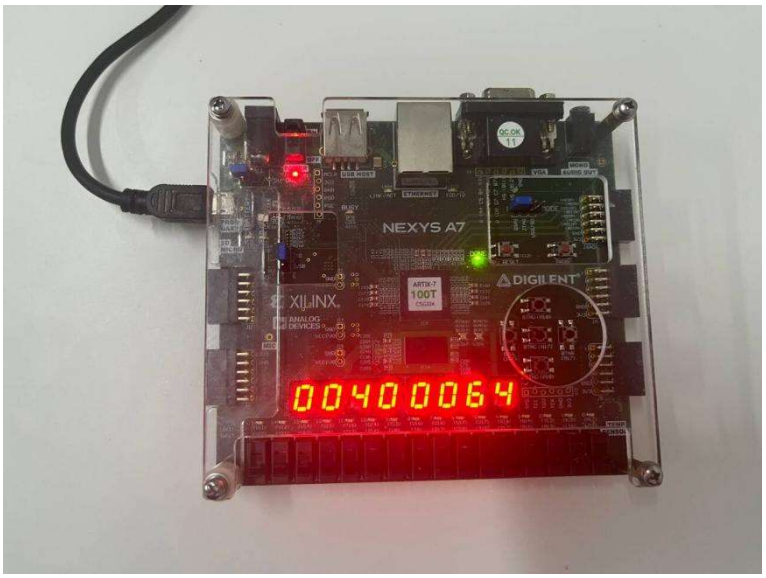
```
.text
main:
    sll $zero,$zero,0
    addi $s1,$zero,0x1     # L=1
    addi $s2,$zero,0x64  # R=100
    addi $s3,$zero,0x31    # value=49
    addi $s4,$zero,0     # break_time=0
    addi $s5,$zero,0     # trial_time=0
    addi $s6,$zero,0     # last_break=0
    addi $t5,$zero,1     # 临时变量，值为1
loop:
    sltu $t3,$s2,$s1     # $t3 = (right < left)
    bne $zero, $t3, loop_end  # 如果 right < left 则跳出循环
    add $s7,$s1,$s2      # mid = L + R
    sra $s7,$s7,1        # mid = mid / 2
    addi $s5,$s5,1       # trial_time++
    sltu $t1,$s3,$s7     # $t1 = (value < mid)
    beq $zero,$t1,not_equal  # 不相等 mid > value 跳转
    addi $s4,$s4,1       # break_time++
    sub $s2,$s7,$t5      # R = mid - 1
    addi $s6,$zero,1     # last_break = 1
    j loop
```

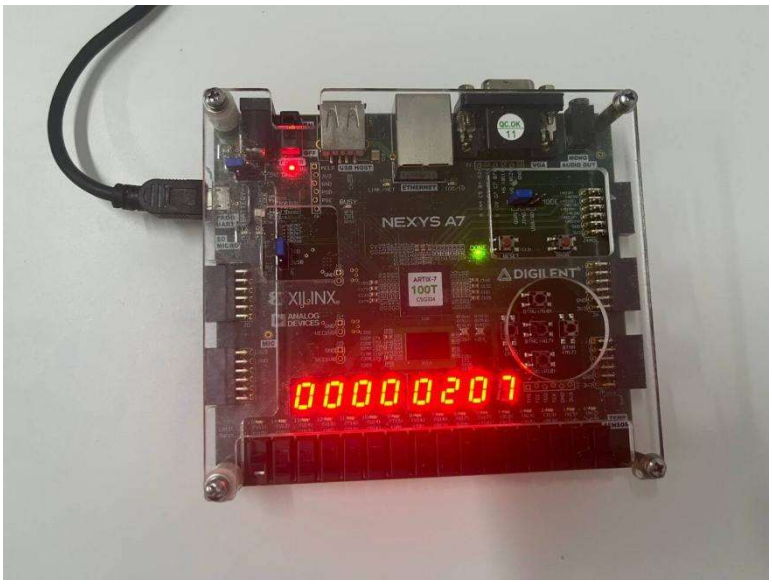
```
not_equal:
    addi $s1,$s7,1       # L = mid + 1
    addi $s6,$zero,0     # last_break = 0
    j loop
loop_end:
    addi $t2,$s4,0       # eggs = break_time
    bne $zero,$s6,not_break:
    addi $t2,$t2,1       # eggs = eggs + 1
not_break:
    j not_break:
    # 100层，耐摔49，
    # 最终的结果 trial_time $s5
    # 最终的结果 eggs $t2
    # 最终的结果 last_break $s6
```

可以看到，我们把初始的输入给到 s1、s2、s3 寄存器中，将最终实现的结果放入在 s5、t2 和 s6 寄存器中，这样修改初始寄存器的值可以实现输入，而查看输出结果则是增加接口，用于输出寄存器堆中存结果的寄存器，然后在显示结果时即将这几个寄存器的结果输出到七段数码管上，这样就可以完成程序的运行和输出。

七、 实验验算程序下板测试过程与实现

为了输出结果，我在 board_show 顶层模块中添加新的接口 result，用来选择是否输出结果：如果 result 为 0，则说明不输出结果，七段数码管显示 PC 的变化值；如果 result 为 1，则在七段数码管上分别显示：broken（最后是否摔碎）、eggs（摔的总鸡蛋数）和 fall_num（摔的总次数），显示数据的拼接为：{8'h0, broken, eggs, fall_num}，这样在七段数码管上除去开头两位，接下来每隔两位显示摔鸡蛋程序运行的结果。下板状况如图：





可以看到，程序最终运行结束的时候，PC 位于 00400064，这与我们汇编最后一条指令的位置是一致的。同时，摔鸡蛋程序的输出结果为：00、02、07，即：最后一次未摔破、摔了 2 个鸡蛋、一共摔了 7 次，这与程序开始给定的楼层为 100、耐摔值为 49 的结果是相吻合的，因此可以证明：CPU 的下板过程是成功的。

八、 总结与体会

本次 31 条 mips 指令 CPU 实验的实现过程比较难，主要是对 CPU 整体的逻辑进行梳理，对 CPU 内部各模块、CPU 和 IMEM、DMEM 之间的数据调度和传递都要有一定的整体规划，然后再进行整合。

实现 CPU 的过程中，最关键也是最困难的模块应该是 Controller 的实现。其他相关模块，如 ALU、Regfiles 都有专门的功能，可以方便的实现，但是如何控制 CPU 传递哪些值给这些模块就需要 Controller 来分析输出。我自己在实现的时候容易忽略一些信号的传递，比如 jal 指令需要将跳转后指令的 PC+4 传递给 31 号寄存器，这里不仅对 PCreg 有控制输入信号，对 Regfiles 也有控制输入信号，因此在实现的时候需要考虑更多的情况，相对而言容易犯错。最后我也是调整了很多次 Controller 的信号，才实现各种控制传值。

此外，由于不同模块之间都有时钟控制，如果采用相同时钟（如都为上升沿），容易出现本应该传递的值没有传递，而新的运算结果已经更新，这样就导致寄存器和 PC 的值与实际情况不符。因此，我也在测试的时候调整各模块的时钟触发，让不同模块之间错开，从而实现正确的传值。

下板模块和测试仿真的模块不同，需要添加七段数码管和分频器，因此为了保持之前程序的完整性，我在 sccomp_dataflow 模块上添加 board_show 顶层模块，调用之前的模块和显示输出相关的模块，从而更方便修改和显示结果。

31 条 mips 指令数量比较多，相对而言容易在编写的时候漏掉一些指令，或者是设计的时候出现小问题。通过这次程序的编写，我的分析问题——解决问题能力得到了很大的提高。我学会了使用 test_bench 打印寄存器的内容，比较寄存器和实际程序的结果，来判断程序是否出现问题、出现问题应该是什么指令出错……最终也是完成对整个项目的测试，实现了 31 条 mips 指令的 CPU，并完成了下板测试。

通过本次实验，我在硬件设计、编译运行程序、修改问题等领域的能力得到了提高。同时，我也明白，后续 54 条 CPU 的设计会更加复杂，出现的问题可能比现在的更加棘手，因此，我还需要不断改进，完善我的硬件知识水平，提高编程能力，实现对未来学习和应用的助力。

九、 附件（所有程序）

下板 CPU 整体模块文件

1. board_show.v

```
module board_show(
input clk_in,
input reset,

input result,

output [7:0] o_seg,
output [7:0] o_sel
    );
//---变量---
//Divider
wire clk_out;

//sccpu
wire [31:0] inst;
wire [31:0] pc;
wire [7:0] fall_num;
wire [7:0] eggs;
wire [7:0] broken;

//seg7x16
wire cs;
wire [31:0] i_data;

//---中间变量---
assign cs = 1;
assign i_data = result ? {8'h0, broken, eggs, fall_num} : pc;

//---模块---
Divider divider(
    .I_CLK(clk_in),
    .rst(reset),
    .O_CLK(clk_out)
);

//sccpu
sccomp_dataflow sccpu(
    .clk_in(clk_out),
    .reset(reset),
    .inst(inst),
```

```
    .pc(pc),
    .fall_num(fall_num),
    .eggs(eggs),
    .broken(broken)
);

//seg7x16
seg7x16 display7(
    .clk(clk_in),
    .reset(reset),
    .cs(cs),
    .i_data(i_data),
    .o_seg(o_seg),
    .o_sel(o_sel)
);
endmodule
```

2. Divider.v

```
module Divider(
    input I_CLK, //输入时钟信号，上升沿有效
    input rst, //复位信号，高电平有效
    output O_CLK //输出时钟
    );
parameter N = 100000000;

reg [30:0] count = 0;
reg out = 0;
always @(posedge I_CLK)
if(rst)
    begin
        out <= 0;
        count <= 0;
    end
else
    if(count < N / 2 - 1)
        begin
            count <= count + 1;
            out <= out;
        end
    else
        begin
            count <= 0;
            out <= ~out;
         end
```

```
assign O_CLK = out;
endmodule
```

3. seg7x16.v

```
module seg7x16(
    input clk,
     input reset,
     input cs,
     input [31:0] i_data,
     output [7:0] o_seg,
     output [7:0] o_sel
    );

    reg [14:0] cnt;
     always @ (posedge clk, posedge reset)
      if (reset)
        cnt <= 0;
      else
        cnt <= cnt + 1'b1;

    wire seg7_clk = cnt[14];

     reg [2:0] seg7_addr;

     always @ (posedge seg7_clk, posedge reset)
       if(reset)
          seg7_addr <= 0;
       else
          seg7_addr <= seg7_addr + 1'b1;

     reg [7:0] o_sel_r;

     always @ (*)
       case(seg7_addr)
          7 : o_sel_r = 8'b01111111;
          6 : o_sel_r = 8'b10111111;
          5 : o_sel_r = 8'b11011111;
          4 : o_sel_r = 8'b11101111;
          3 : o_sel_r = 8'b11110111;
          2 : o_sel_r = 8'b11111011;
          1 : o_sel_r = 8'b11111101;
          0 : o_sel_r = 8'b11111110;
       endcase
```

```
reg [31:0] i_data_store;
always @ (posedge clk, posedge reset)
  if(reset)
     i_data_store <= 0;
   else if(cs)
     i_data_store <= i_data;

reg [7:0] seg_data_r;
always @ (*)
  case(seg7_addr)
     0 : seg_data_r = i_data_store[3:0];
     1 : seg_data_r = i_data_store[7:4];
     2 : seg_data_r = i_data_store[11:8];
     3 : seg_data_r = i_data_store[15:12];
     4 : seg_data_r = i_data_store[19:16];
     5 : seg_data_r = i_data_store[23:20];
     6 : seg_data_r = i_data_store[27:24];
     7 : seg_data_r = i_data_store[31:28];
   endcase

reg [7:0] o_seg_r;
always @ (posedge clk, posedge reset)
  if(reset)
     o_seg_r <= 8'hff;
   else
     case(seg_data_r)
       4'h0 : o_seg_r <= 8'hC0;
     4'h1 : o_seg_r <= 8'hF9;
     4'h2 : o_seg_r <= 8'hA4;
     4'h3 : o_seg_r <= 8'hB0;
     4'h4 : o_seg_r <= 8'h99;
     4'h5 : o_seg_r <= 8'h92;
     4'h6 : o_seg_r <= 8'h82;
     4'h7 : o_seg_r <= 8'hF8;
     4'h8 : o_seg_r <= 8'h80;
     4'h9 : o_seg_r <= 8'h90;
     4'hA : o_seg_r <= 8'h88;
     4'hB : o_seg_r <= 8'h83;
     4'hC : o_seg_r <= 8'hC6;
     4'hD : o_seg_r <= 8'hA1;
     4'hE : o_seg_r <= 8'h86;
     4'hF : o_seg_r <= 8'h8E;
     endcase
```

```
     assign o_sel = o_sel_r;
     assign o_seg = o_seg_r;

endmodule
```

4. sccomp_dataflow.v

```
module sccomp_dataflow(
input clk_in,
input reset,
output [31:0] inst,
output [31:0] pc,

output [7:0] fall_num,
output [7:0] eggs,
output [7:0] broken
    );
//CPU
wire [31:0] pc_out;
wire [31:0] dmem_addr_out;

//IMEM
wire [31:0] imem_addr;
wire [31:0] imem_data;

//DMEM
wire dmem_ena;
wire dmem_w;
wire dmem_r;
wire [31:0] dmem_addr;
wire [31:0] dmem_wdata;
wire [31:0] dmem_rdata;

//---中间变量---
assign imem_addr = (pc_out - 32'h00400000) / 4;

assign dmem_addr = (dmem_addr_out - 32'h10010000) / 4;

//---模块---
//CPU31
CPU31 sccpu(
    .clk(clk_in),
    .rst(reset),
    .ena(1'b1),
    .IR(imem_data),
```

```
    .dmem_rdata(dmem_rdata),
    .dmem_ena(dmem_ena),
    .dmem_w(dmem_w),
    .dmem_r(dmem_r),
    .dmem_addr(dmem_addr_out),
    .dmem_wdata(dmem_wdata),
    .PC(pc_out),
    .fall_num(fall_num),
    .eggs(eggs),
    .broken(broken)
);

//IMEM
IMEM imem(
    .IM_addr(imem_addr[10:0]),
    .IM_data(imem_data)
);
 //DMEM
DMEM dmem(
    .clk(clk_in),
    .ena(dmem_ena),
    .DM_w(dmem_w),
    .DM_r(dmem_r),
    .DM_addr(dmem_addr),
    .DM_wdata(dmem_wdata),
    .DM_rdata(dmem_rdata)
);

//---输出---
assign inst = imem_data;

assign pc = pc_out;
endmodule
```

5. IMEM.v

```
module IMEM(
input [10:0] IM_addr,
output [31:0] IM_data
    );

dist_mem_gen_0 IMEM(
    .a(IM_addr),
    .spo(IM_data)
);
```

```
endmodule
```

6. DMEM.v

```
module DMEM(
input clk,
input ena,
input DM_w,
input DM_r,
input [31:0] DM_addr,
input [31:0] DM_wdata,
output [31:0] DM_rdata
    );
reg [31:0] DMEM[1023:0];

assign DM_rdata = (ena && DM_r) ? DMEM[DM_addr] : 32'bz;

always @(posedge clk)
begin
    if(ena && DM_w)
        DMEM[DM_addr] <= DM_wdata;
end
endmodule
```

7. CPU31.v

```
module CPU31(
input clk,                  //CPU 时钟
input rst,                  //复位
input ena,                  //使能
input [31:0] IR,            //指令
input [31:0] dmem_rdata,    //从 DMEM 读取的数据(MDR)

output dmem_ena,            //DMEM 使能
output dmem_w,              //DMEM 写信号
output dmem_r,              //DMEM 读信号
output [31:0] dmem_addr,    //DMEM 地址(MAR)
output [31:0] dmem_wdata,   //写进 DMEM 数据
output [31:0] PC,           //PC

output [7:0] fall_num,      //摔的总次数
output [7:0] eggs,          //摔的总鸡蛋数
output [7:0] broken         //最后是否摔碎
    );
//---变量---
//Decoder
```

```
wire Add,Addu,Sub,Subu,And,Or,Xor,Nor;
wire Slt,Sltu,Sll,Stl,Sta,Sllv,Srlv,Srav,Jr;
wire Addi,Addiu,Andi,Ori,Xori,Lw,Sw;
wire Beq,Bne,Slit,Sltiu,Lui;
wire J,Jal;
wire [4:0] shamt;
wire [15:0] immediate;
wire [25:0] index;

//extend
wire [31:0] ext5;
wire ext16_sign;
wire [31:0] ext16;
wire [31:0] ext18;
wire [31:0] ext18_add;
wire [31:0] concat;

//Controller
wire [1:0] mux_A;
wire [1:0] mux_B;
wire [1:0] mux_PC;

//ALU
wire [31:0] A;
wire [31:0] B;
wire [3:0] ALUC;
wire [31:0] C;
wire zero;
wire carry;
wire negative;
wire overflow;

//PC
wire PC_ena;
wire [31:0] PC_in;
wire [31:0] PC_out;

wire [31:0] NPC;

//Regfiles
wire Reg_ena;
wire we;
wire [4:0] rsc;
wire [4:0] rtc;
```

```
wire [4:0] rdc;
wire [31:0] rs;
wire [31:0] rt;
wire [31:0] rd;

//---中间变量---
//PC
assign NPC = PC_out + 32'd4;
assign PC_ena = ena;

//extend
assign ext5 = {27'b0, shamt};
assign ext16_sign  = (Andi || Ori || Xori || Lui) ? 1'b0 : immediate[15];
assign ext16 = {{16{ext16_sign}}, immediate};
assign ext18 = {{14{immediate[15]}}, immediate, 2'b0};
assign ext18_add = ext18 + NPC;
assign concat   = {NPC[31:28], index, 2'b0};

//selector
selector4 selector_A(
    .iC0(rs),
    .iC1(ext5),
    .iC2(PC_in),
    .iS(mux_A),
    .oZ(A)
);

selector4 selector_B(
    .iC0(rt),
    .iC1(ext16),
    .iC2(32'd4),
    .iS(mux_B),
    .oZ(B)
);

selector4 selector_PC(
    .iC0(NPC),
    .iC1(rs),
    .iC2(ext18_add),
    .iC3(concat),
    .iS(mux_PC),
    .oZ(PC_in)
);
```

```
//Regfiles
assign Reg_ena = !J;
assign rd = Lw ? dmem_rdata : C;

//---模块---
//Decoder
Decoder decoder(
    .IR(IR),
    .Add(Add),
    .Addu(Addu),
    .Sub(Sub),
    .Subu(Subu),
    .And(And),
    .Or(Or),
    .Xor(Xor),
    .Nor(Nor),
    .Slt(Slt),
    .Sltu(Sltu),
    .Sll(Sll),
    .Srl(Srl),
    .Sra(Sra),
    .Sllv(Sllv),
    .Srlv(Srlv),
    .Srav(Srav),
    .Jr(Jr),
    .Addi(Addi),
    .Addiu(Addiu),
    .Andi(Andi),
    .Ori(Ori),
    .Xori(Xori),
    .Lw(Lw),
    .Sw(Sw),
    .Beq(Beq),
    .Bne(Bne),
    .Slti(Slti),
    .Sltiu(Sltiu),
    .Lui(Lui),
    .J(J),
    .Jal(Jal),
    .rsc(rsc),
    .rtc(rtc),
    .rdc(rdc),
    .shamt(shamt),
    .immediate(immediate),
```

```
    .index(index)
);

//Controller
Controller controller(
    .Add(Add),
    .Addu(Addu),
    .Sub(Sub),
    .Subu(Subu),
    .And(And),
    .Or(Or),
    .Xor(Xor),
    .Nor(Nor),
    .Slt(Slt),
    .Sltu(Sltu),
    .Sll(Sll),
    .Srl(Srl),
    .Sra(Sra),
    .Sllv(Sllv),
    .Srlv(Srlv),
    .Srav(Srav),
    .Jr(Jr),
    .Addi(Addi),
    .Addiu(Addiu),
    .Andi(Andi),
    .Ori(Ori),
    .Xori(Xori),
    .Lw(Lw),
    .Sw(Sw),
    .Beq(Beq),
    .Bne(Bne),
    .Slti(Slti),
    .Sltiu(Sltiu),
    .Lui(Lui),
    .J(J),
    .Jal(Jal),
    .zero(zero),
    .we(we),
    .mux_A(mux_A),
    .mux_B(mux_B),
    .ALUC(ALUC),
    .DM_w(dmem_w),
    .DM_r(dmem_r),
    .mux_PC(mux_PC)
```

```
);

//ALU
ALU alu(
    .A(A),
    .B(B),
    .ALUC(ALUC),
    .C(C),
    .zero(zero),
    .carry(carry),
    .negative(negative),
    .overflow(overflow)
);

//PC
PCreg pcreg(
    .clk(clk),
    .rst(rst),
    .ena(PC_ena),
    .PC_in(PC_in),
    .PC_out(PC_out)
);

//Regfiles
Regfiles cpu_ref(
    .clk(clk),
    .rst(rst),
    .ena(Reg_ena),
    .we(we),
    .rsc(rsc),
    .rtc(rtc),
    .rdc(rdc),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .s5(fall_num),
    .t2(eggs),
    .s6(broken)
);

//---输出---
assign dmem_ena = (dmem_r || dmem_w);
assign dmem_addr = C;
assign dmem_wdata = rt;
```

```
assign PC = PC_out;

endmodule
```

8. Decoder.v

```
module Decoder(
input [31:0] IR,
//R-type
output Add,
output Addu,
output Sub,
output Subu,
output And,
output Or,
output Xor,
output Nor,
output Slt,
output Sltu,
output Sll,
output Srl,
output Sra,
output Sllv,
output Srlv,
output Srav,
output Jr,
//I-type
output Addi,
output Addiu,
output Andi,
output Ori,
output Xori,
output Lw,
output Sw,
output Beq,
output Bne,
output Slti,
output Sltiu,
output Lui,
//J-type
output J,
output Jal,

output [4:0] rsc,
```

```
output [4:0] rtc,
output [4:0] rdc,

output [4:0] shamt,
output [15:0] immediate,
output [25:0] index
    );
    //op 均为000000，靠func 区分
    parameter OP0        = 6'b000000;

    parameter ADD_FUNC  = 6'b100000;
    parameter ADDU_FUNC = 6'b100001;
    parameter SUB_FUNC  = 6'b100010;
    parameter SUBU_FUNC = 6'b100011;
    parameter AND_FUNC  = 6'b100100;
    parameter OR_FUNC   = 6'b100101;
    parameter XOR_FUNC  = 6'b100110;
    parameter NOR_FUNC  = 6'b100111;
    parameter SLT_FUNC  = 6'b101010;
    parameter SLTU_FUNC = 6'b101011;
    parameter SLL_FUNC  = 6'b000000;
    parameter SRL_FUNC  = 6'b000010;
    parameter SRA_FUNC  = 6'b000011;
    parameter SLLV_FUNC = 6'b000100;
    parameter SRLV_FUNC = 6'b000110;
    parameter SRAV_FUNC = 6'b000111;
    parameter JR_FUNC   = 6'b001000;
    //op 不同，靠op 来区分
    parameter ADDI_OP   = 6'b001000;
    parameter ADDIU_OP  = 6'b001001;
    parameter ANDI_OP   = 6'b001100;
    parameter ORI_OP    = 6'b001101;
    parameter XORI_OP   = 6'b001110;
    parameter LW_OP     = 6'b100011;
    parameter SW_OP     = 6'b101011;
    parameter BEQ_OP    = 6'b000100;
    parameter BNE_OP    = 6'b000101;
    parameter SLTI_OP   = 6'b001010;
    parameter SLTIU_OP  = 6'b001011;
    parameter LUI_OP    = 6'b001111;

    parameter J_OP      = 6'b000010;
    parameter JAL_OP    = 6'b000011;
```

```
    assign Add = ((IR[31:26] == OP0) && (IR[5:0] == ADD_FUNC)) ? 1'b1 :
1'b0;
    assign Addu = ((IR[31:26] == OP0) && (IR[5:0] == ADDU_FUNC)) ? 1'b1 :
1'b0;
    assign Sub = ((IR[31:26] == OP0) && (IR[5:0] == SUB_FUNC)) ? 1'b1 :
1'b0;
    assign Subu = ((IR[31:26] == OP0) && (IR[5:0] == SUBU_FUNC)) ? 1'b1 :
1'b0;
    assign And = ((IR[31:26] == OP0) && (IR[5:0] == AND_FUNC)) ? 1'b1 :
1'b0;
    assign Or = ((IR[31:26] == OP0) && (IR[5:0] == OR_FUNC)) ? 1'b1 : 1'b0;
    assign Xor = ((IR[31:26] == OP0) && (IR[5:0] == XOR_FUNC)) ? 1'b1 :
1'b0;
    assign Nor = ((IR[31:26] == OP0) && (IR[5:0] == NOR_FUNC)) ? 1'b1 :
1'b0;
    assign Slt = ((IR[31:26] == OP0) && (IR[5:0] == SLT_FUNC)) ? 1'b1 :
1'b0;
    assign Sltu = ((IR[31:26] == OP0) && (IR[5:0] == SLTU_FUNC)) ? 1'b1 :
1'b0;
    assign Sll = ((IR[31:26] == OP0) && (IR[5:0] == SLL_FUNC)) ? 1'b1 :
1'b0;
    assign Srl = ((IR[31:26] == OP0) && (IR[5:0] == SRL_FUNC)) ? 1'b1 :
1'b0;
    assign Sra = ((IR[31:26] == OP0) && (IR[5:0] == SRA_FUNC)) ? 1'b1 :
1'b0;
    assign Sllv = ((IR[31:26] == OP0) && (IR[5:0] == SLLV_FUNC)) ? 1'b1 :
1'b0;
    assign Srlv = ((IR[31:26] == OP0) && (IR[5:0] == SRLV_FUNC)) ? 1'b1 :
1'b0;
    assign Srav = ((IR[31:26] == OP0) && (IR[5:0] == SRAV_FUNC)) ? 1'b1 :
1'b0;
    assign Jr = ((IR[31:26] == OP0) && (IR[5:0] == JR_FUNC)) ? 1'b1 : 1'b0;

    assign Addi = (IR[31:26] == ADDI_OP) ? 1'b1 : 1'b0;
    assign Addiu = (IR[31:26] == ADDIU_OP) ? 1'b1 : 1'b0;
    assign Andi = (IR[31:26] == ANDI_OP) ? 1'b1 : 1'b0;
    assign Ori = ((IR[31:26] == ORI_OP)) ? 1'b1 : 1'b0;
    assign Xori = ((IR[31:26] == XORI_OP)) ? 1'b1 : 1'b0;
    assign Lw = (IR[31:26] == LW_OP) ? 1'b1 : 1'b0;
    assign Sw = (IR[31:26] == SW_OP) ? 1'b1 : 1'b0;
    assign Beq = (IR[31:26] == BEQ_OP) ? 1'b1 : 1'b0;
    assign Bne = ((IR[31:26] == BNE_OP)) ? 1'b1 : 1'b0;
    assign Slti = ((IR[31:26] ==  SLTI_OP)) ? 1'b1 : 1'b0;
    assign Sltiu = ((IR[31:26] == SLTIU_OP)) ? 1'b1 : 1'b0;
```

```
   assign Lui = (IR[31:26] == LUI_OP) ? 1'b1 : 1'b0;

    assign J = (IR[31:26] == J_OP) ? 1'b1 : 1'b0;
   assign Jal = (IR[31:26] == JAL_OP) ? 1'b1 : 1'b0;

    assign rsc = (Add || Addu || Sub || Subu || And || Or ||
                  Xor || Nor || Slt || Sltu || Sllv || Srlv ||
                  Srav || Jr || Addi || Addiu || Andi || Ori ||
                  Xori || Lw || Sw || Beq || Bne || Slti ||
                  Sltiu) ? IR[25:21] : 5'bz;

    assign rtc = (Add || Addu || Sub || Subu || And || Or ||
                  Xor || Nor || Slt || Sltu || Sll || Srl ||
                  Sra || Sllv || Srlv || Srav || Sw || Beq ||
                  Bne || Slti || Sltiu) ? IR[20:16] : 5'bz;

    assign rdc = (Add || Addu || Sub || Subu || And || Or ||
                  Xor || Nor || Slt || Sltu || Sll || Srl ||
                  Sra || Sllv || Srlv || Srav) ? IR[15:11] :
                  ((Addi || Addiu || Andi || Ori || Xori ||
                  Lw || Slti || Sltiu || Lui) ? IR[20:16] :
                  (Jal ? 5'd31 : 5'bz));

   assign shamt = (Sll || Srl || Sra) ? IR[10:6] : 5'bz;

    assign immediate = (Addi || Addiu || Andi || Ori || Xori ||
                        Lw || Sw || Beq || Bne || Slti || Sltiu ||
                        Lui) ? IR[15:0] : 16'bz;

    assign index = (J || Jal) ? IR[25:0] : 26'bz;
endmodule
```

9. Controller.v

```
module Controller(
input Add,
input Addu,
input Sub,
input Subu,
input And,
input Or,
input Xor,
input Nor,
input Slt,
input Sltu,
```

```
input Sll,
input Srl,
input Sra,
input Sllv,
input Srlv,
input Srav,
input Jr,
//I-type
input Addi,
input Addiu,
input Andi,
input Ori,
input Xori,
input Lw,
input Sw,
input Beq,
input Bne,
input Slti,
input Sltiu,
input Lui,
//J-type
input J,
input Jal,

input zero,

output we,
output [1:0] mux_A,
output [1:0] mux_B,
output [3:0] ALUC,
output DM_w,
output DM_r,
output [1:0] mux_PC
    );
assign we = (!(Jr || Sw || Beq || Bne || J)) ? 1'b1 : 1'b0;

assign mux_A[0] = Sll || Srl || Sra;
assign mux_A[1] = Jal;

assign mux_B[0] = Addi || Addiu || Andi || Ori || Xori ||
                Lw || Sw || Slti || Sltiu || Lui;
assign mux_B[1] = Jal;

assign ALUC[3] = Slt || Sltu || Sll || Srl || Sra || Lui ||
```

```
                Slti || Sltiu || Sllv || Srlv || Srav;
assign ALUC[2] = And || Or || Xor || Nor || Sra || Lui ||
                Andi || Ori || Xori || Srav;
assign ALUC[1] = Sub || Subu || Xor || Nor || Sll || Srl ||
                Xori || Sllv || Srlv || Beq || Bne;
assign ALUC[0] = Addu || Subu || Or || Nor || Sltu || Srl || Lui ||
                Addiu || Ori || Sltiu || Srlv || Beq || Bne;

assign DM_r = Lw;
assign DM_w = Sw;

assign mux_PC[0] = Jr || J || Jal;
assign mux_PC[1] = (Beq && zero) || (Bne && ~zero) || J || Jal;

endmodule
```

10. ALU.v

```
module ALU(
input [31:0] A,
input [31:0] B,
input [3:0] ALUC,

output [31:0] C,
output zero,
output carry,
output negative,
output overflow
    );
    parameter ADD = 4'b0000;
    parameter ADDU = 4'b0001;
    parameter SUB = 4'b0010;
    parameter SUBU = 4'b0011;
    parameter AND = 4'b0100;
    parameter OR = 4'b0101;
    parameter XOR = 4'b0110;
    parameter NOR = 4'b0111;
    parameter SLT = 4'b1000;
    parameter SLTU = 4'b1001;
    parameter SLL = 4'b1010;
    parameter SRL = 4'b1011;
    parameter SRA = 4'b1100;
   parameter LUI = 4'b1101;

    reg [32:0] result;
```

```
    wire signed [31:0] signed_A;
    wire signed [31:0] signed_B;

    assign signed_A = A;
    assign signed_B = B;
    assign C = result[31:0];
    assign zero = (result == 32'b0) ? 1 : 0;
    assign carry = result[32];
    assign negative = (ALUC == SLT ? (signed_A < signed_B) : ((ALUC ==
SLTU) ? (A < B) : 1'b0));
   assign overflow = result[32] ^ result[31];

    always @(*)
    begin
        case(ALUC)
            ADD: result <= signed_A + signed_B;
            ADDU: result <= A + B;
            SUB: result <= signed_A - signed_B;
            SUBU: result <= A - B;
            AND: result <= A & B;
            OR: result <= A | B;
            XOR: result <= A ^ B;
            NOR: result <= ~(A | B);
            SLT: result <= (signed_A < signed_B);
            SLTU: result <= (A < B);
            SLL: result <= B << A;
            SRL: result <= B >> A;
            SRA: result <= signed_B >>> A;
            LUI: result <= {B[15:0], 16'b0};
            default: result <= 32'b0;
        endcase
   end

endmodule
```

11. Pcreg.v

```
module PCreg(
input clk,
input rst,
input ena,
input [31:0] PC_in,
output reg [31:0] PC_out
    );
always @(posedge clk or posedge rst)
```

```
begin
    if(rst)
        PC_out <= 32'h00400000;
    else if(ena)
        PC_out <= PC_in;
end
endmodule
```

12. Regfiles.v

```
module Regfiles(
input clk,
input rst,
input ena,
input we,
input [4:0] rsc,
input [4:0] rtc,
input [4:0] rdc,

output [31:0] rs,
output [31:0] rt,
input [31:0] rd,

output [7:0] s5,
output [7:0] t2,
output [7:0] s6
    );
reg [31:0] array_reg[31:0];
integer i;

assign rs = ena ? array_reg[rsc] : 32'bz;
assign rt = ena ? array_reg[rtc] : 32'bz;

assign s5 = array_reg[21];
assign t2 = array_reg[10];
assign s6 = array_reg[22];

always @(negedge clk or posedge rst)
begin
    if(rst)
    begin
        for(i = 0; i < 32; i = i + 1)
            array_reg[i] <= 32'b0;
    end
    else if(ena && we && (rdc != 5'b0)) //0 号寄存器只能为0，不能修改
```

```
        array_reg[rdc] <= rd;
end

endmodule
```

13. selector4.v

```
module selector4(
input [31:0] iC0,
input [31:0] iC1,
input [31:0] iC2,
input [31:0] iC3,
input [1:0] iS,
output reg [31:0] oZ
    );
always @(*)
begin
    case(iS)
        2'b00:oZ <= iC0;
        2'b01:oZ <= iC1;
        2'b10:oZ <= iC2;
        2'b11:oZ <= iC3;
        default: oZ <= oZ;
    endcase
end
endmodule
```

test_bench 文件

14. CPU31_single_tb.v

```
module _246tb_ex10_tb;

    // Inputs
    reg clk_in;
    reg reset;

    // Outputs
    wire [31:0] inst;
    wire [31:0] pc;
    // Instantiate the Unit Under Test (UUT)
    sccomp_dataflow uut (
        .clk_in(clk_in),
        .reset(reset),
        .inst(inst),
        .pc(pc)
    );
```

```
    integer file_output;


initial begin
        //$dumpfile("mydump.txt");
        //$dumpvars(0,cpu_tb.uut.pcreg.data_out);
        file_output = $fopen("_246tb_ex10_result.txt");
        // Initialize Inputs
        clk_in = 0;
        reset = 1;

        // Wait 100 ns for global reset to finish
        #40;
      reset = 0;
        // Add stimulus here

        //#100;
        //$fclose(file_output);
    end

    always begin
    #50;
    clk_in = ~clk_in;
    if(clk_in == 1'b1) begin
            $fdisplay(file_output, "pc: %h", pc);
            $fdisplay(file_output, "instr: %h", inst);
            $fdisplay(file_output, "regfile0: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[0]);
            $fdisplay(file_output, "regfile1: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[1]);
            $fdisplay(file_output, "regfile2: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[2]);
            $fdisplay(file_output, "regfile3: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[3]);
            $fdisplay(file_output, "regfile4: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[4]);
            $fdisplay(file_output, "regfile5: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[5]);
            $fdisplay(file_output, "regfile6: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[6]);
            $fdisplay(file_output, "regfile7: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[7]);
```

```
            $fdisplay(file_output, "regfile8: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[8]);
            $fdisplay(file_output, "regfile9: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[9]);
            $fdisplay(file_output, "regfile10: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[10]);
            $fdisplay(file_output, "regfile11: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[11]);
            $fdisplay(file_output, "regfile12: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[12]);
            $fdisplay(file_output, "regfile13: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[13]);
            $fdisplay(file_output, "regfile14: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[14]);
            $fdisplay(file_output, "regfile15: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[15]);
            $fdisplay(file_output, "regfile16: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[16]);
            $fdisplay(file_output, "regfile17: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[17]);
            $fdisplay(file_output, "regfile18: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[18]);
            $fdisplay(file_output, "regfile19: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[19]);
            $fdisplay(file_output, "regfile20: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[20]);
            $fdisplay(file_output, "regfile21: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[21]);
            $fdisplay(file_output, "regfile22: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[22]);
            $fdisplay(file_output, "regfile23: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[23]);
            $fdisplay(file_output, "regfile24: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[24]);
            $fdisplay(file_output, "regfile25: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[25]);
            $fdisplay(file_output, "regfile26: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[26]);
            $fdisplay(file_output, "regfile27: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[27]);
            $fdisplay(file_output, "regfile28: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[28]);
            $fdisplay(file_output, "regfile29: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[29]);
```

```
            $fdisplay(file_output, "regfile30: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[30]);
            $fdisplay(file_output, "regfile31: %h",
_246tb_ex10_tb.uut.sccpu.cpu_ref.array_reg[31]);
        end
    end
endmodule
```