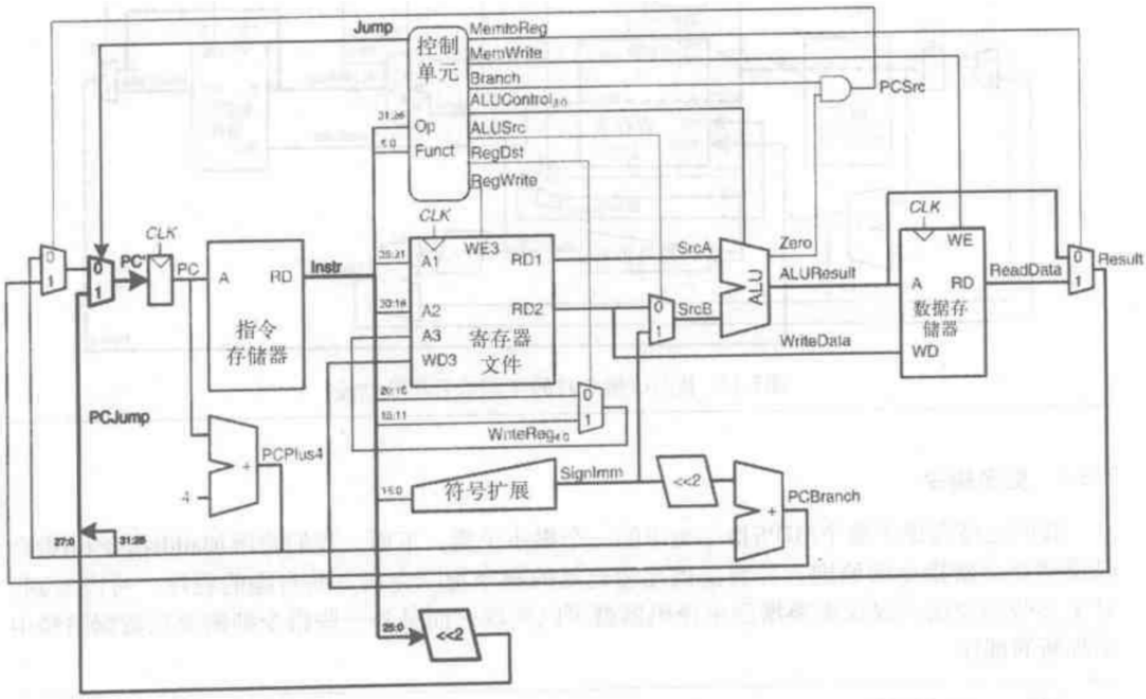
P3 单周期CPU设计文档

设计概述

- 设计的处理器为32位单周期处理器
- 处理器支持的指令集为`add, sub, ori, lw, sw, beq, lui, nop`等
- `nop`为空指令，其机器码为`0x00000000`，不进行任何有效行为，如修改寄存器等
- `add, sub`按无符号加减法处理，不考虑溢出

顶层设计

参考了《数字设计与计算机体系结构》图7-14，在实际设计上略有改动。



需求分析

R型指令

add：无符号加

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	add 100000
	6	5	5	5	5	6
格式	add rd, rs, rt					
描述	GPR[rd] ← GPR[rs]+GPR[rt]					

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) + (GPR[rt]$_{31}$‖GPR[rt])
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rd] ← temp$_{31..0}$
	endif					

示例	add $s1, $s2, $s3					
其他	temp$_{32}$≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 add 与 addu 等价。					

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	addu 100001
	6	5	5	5	5	6
格式	addu rd, rs, rt					
描述	GPR[rd] ← GPR[rs] + GPR[rt]					
操作	GPR[rd] ← GPR[rs] + GPR[rt]					
示例	addu $s1, $s2, $s3					
其他						

sub：无符号减

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	sub 100010
	6	5	5	5	5	6
格式	sub rd, rs, rt					
描述	GPR[rd] ← GPR[rs] - GPR[rt]					

操作	temp ← (GPR[rs]$_{31}$‖GPR[rs]) - (GPR[rt]$_{31}$‖GPR[rt])
	if temp$_{32}$ ≠ temp$_{31}$ then
	SignalException(IntegerOverflow)
	else
	GPR[rd] ← temp$_{31..0}$
	endif					

示例	sub $s1, $s2, $s3					
其他	temp$_{32}$ ≠ temp$_{31}$ 代表计算结果溢出。 如果不考虑溢出，则 sub 与 subu 等价。					

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	rt	rd	0 00000	subu 100011
	6	5	5	5	5	6
格式	subu rd, rs, rt					
描述	GPR[rd] ← GPR[rs] - GPR[rt]					
操作	GPR[rd] ← GPR[rs] - GPR[rt]					
示例	subu $s1, $s2, $s3					
其他	subu 不考虑减法溢出。例如 0x0000_0000 - 0xFFFF_FFFF = 0x0000_0001，即结果为非负值。					

> P3使用add和sub的funct字段，但实现addu和subu的功能，即不考虑溢出。

I型指令

ori：或立即数

编码	31 26	25 21	20 16	15 0
	ori 001101	rs	rt	immediate
	6	5	5	16
格式	ori rt, rs, immediate			
描述	GPR[rt] ← GPR[rs] OR immediate			
操作	GPR[rt] ← GPR[rs] OR zero_extend(immediate)			
示例	ori $s1, $s2, 0x55AA			
其他				

lw：加载字

编码	31 26	25 21	20 16	15 0
	lw 100011	base	rt	offset
	6	5	5	16
格式	lw rt, offset(base)			
描述	GPR[rt] ← memory[GPR[base]+offset]			
操作	Addr ← GPR[base] + sign_ext(offset)			

	GPR[rt] ← memory[Addr]
示例	lw $v1, 8($s0)
约束	Addr 必须是 4 的倍数(即 $Addr_{1..0}$ 必须为 00)，否则产生地址错误异常

sw：存储字

编码	31 26	25 21	20 16	15 0
	sw 101011	base	rt	offset
	6	5	5	16
格式	sw rt, offset(base)			
描述	memory[GPR[base]+offset] ← GPR[rt]			
操作	Addr ← GPR[base] + sign_ext(offset) memory[Addr] ← GPR[rt]			
示例	sw $v1, 8($s0)			
约束	Addr 必须是 4 的倍数(即 $Addr_{1..0}$ 必须为 00)，否则产生地址错误异常			

beq：相等时转移

编码	31 26	25 21	20 16	15 0
	beq 000100	rs	rt	offset
	6	5	5	16
格式	beq rs, rt, offset			
描述	if (GPR[rs] == GPR[rt]) then 转移			

操作	if (GPR[rs] == GPR[rt])
	  PC ← PC + 4 + sign_extend(offset‖0^2)
	else
	  PC ← PC + 4			

示例	beq $s1, $s2, -2			
其他				

lui：立即数加载至高位

编码	31 26	25 21	20 16	15 0
	lui 001111	0 00000	rt	immediate
	6	5	5	16
格式	lui rt, immediate			
描述	GPR[rt] ← immediate‖0^{16}			
操作	GPR[rt] ← immediate‖0^{16}			
示例	lui $s1, 0x55AA			
其他				

其他

nop：空置零

略

模块定义

IFU(取指令单元)

内部包括 PC（程序计数器）、IM（指令存储器）及相关逻辑。

PC 用寄存器实现，应具有**异步复位**功能，复位值为起始地址。

起始地址：0x00003000。

地址范围：0x00003000 ~ 0x00006FFF。

IM用ROM实现，容量为4096 × 32bit。

IM实际地址宽度仅为12位，需要使用恰当的方法将PC中储存的地址同IM联系起来。

> PC的处理方法：
>
> - PC的变化范围为0x00003000 ~ 0x00006FFF，考虑使用PC' = PC - 0x00003000，则PC'的范围为0x00000000-0x00003FFF，不仅保证PC和PC'在数值上一一对应，而且在设计处理时更加方便。
> - 注意，输出是需要输出PC的值，而不是PC'。
> - IM的实际地址宽度为12位，而PC的有效位数（可能发生变化的位数）为低14位。因为ROM是按字寻址，在从IM读取指令时只需要用PC[13:2]作为地址，就可以正确读取数据。

端口定义

表2-1-1 IFU模块端口定义

信号名	方向	描述
clk	I	时钟信号
reset	I	**异步**复位信号，将PC置0
branch	I	是否执行beq指令 branch为0：不执行beq指令 branch为1：执行beq指令
offset[31:0]	I	符号扩展后的偏移量(来自beq指令)
Instr[31:0]	O	输出IM中PC地址上的指令
PC	O	输出当前PC的值

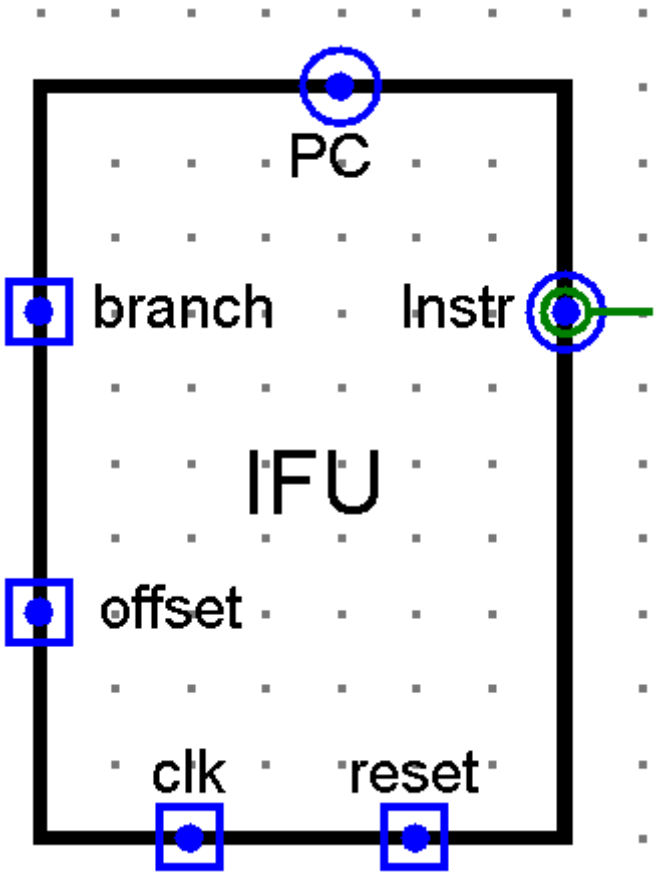
功能定义

表2-1-2 IFU模块功能定义

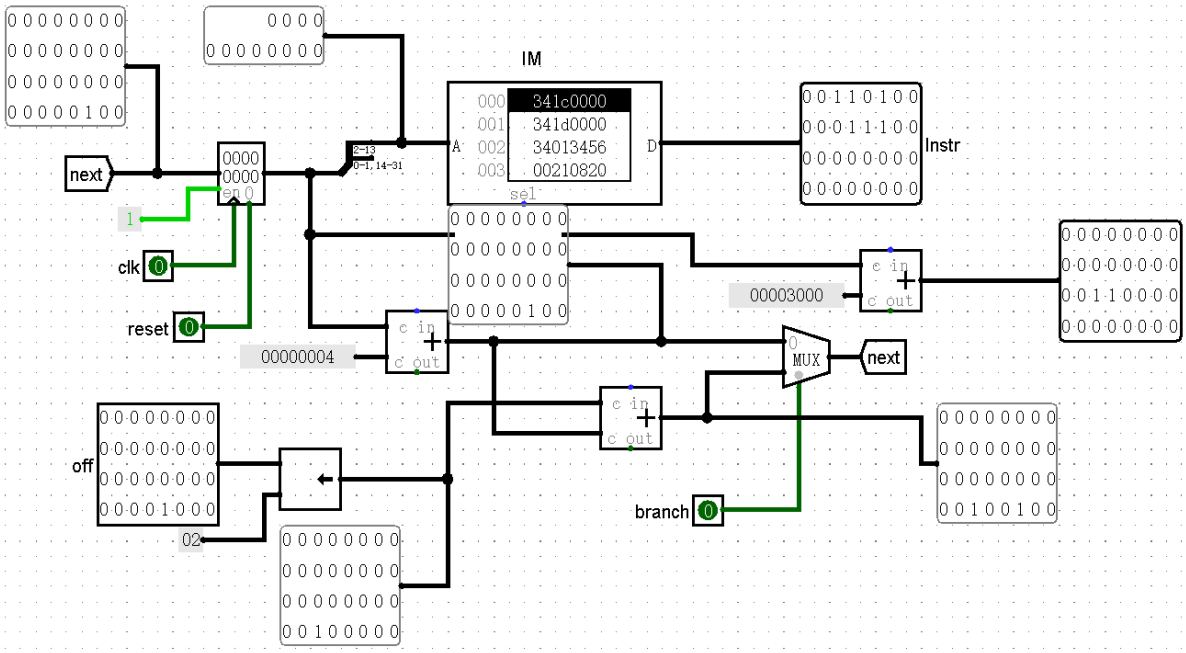
序号	功能	描述
1	异步复位	reset置1时，将PC'置为0x00000000
2	更新下一个PC的值	时钟上升沿来临时，将下一个PC的值存入寄存器 branch为0时，$PC' \rightarrow PC' + 4$ branch为1时，$PC' \rightarrow PC' + 4 + offset||0^2$

实现

整体



细节



GRF（寄存器文件）

使用**具有写使能功能**的寄存器实现，寄存器总数为**32个**，具有**异步复位**功能。

其中，**0号寄存器**（$zero）的值始终保持为0。其他的寄存器**初始值（复位后）均为0**，无需专门设置。

端口定义

表2-2-1 GRF模块端口定义

信号名	方向	描述
clk	I	时钟信号
reset	I	**异步**复位信号，将32个寄存器中的值全部清零 1:复位 0:无效
WE	I	写使能信号 1:可向GRF中写入数据 0:不能向GRF中写入数据
A1[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1
A2[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2
A3[4:0]	I	5位地址输入信号，指定32个寄存器中的一个将WD中的数据写入
WD[31:0]	I	32位数据输入信号
RD1[31:0]	O	输出A1指定的寄存器中的32位数据
RD2[31:0]	O	输出A2指定的寄存器中的32位数据

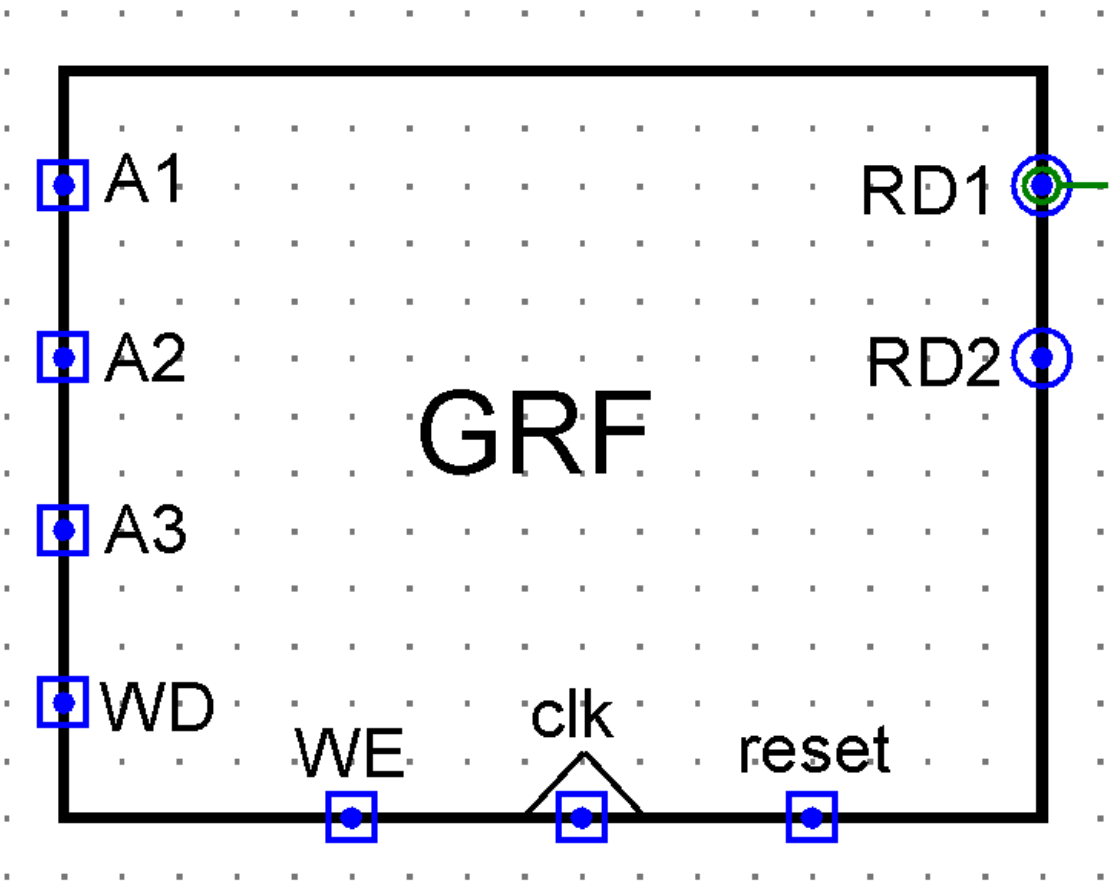
功能定义

表2-2-2 GRF模块功能定义

序号	功能	描述
1	异步复位	reset信号置1时，所有寄存器存储的数值清零，其行为与logisim自带部件register的reset接口完全相同
2	读数据	读出A1，A2地址对应寄存器中所存储的数据到对应的RD1，RD2
3	写数据	当**WE有效且时钟上升沿来临**时，将WD写入A3所对应的寄存器中

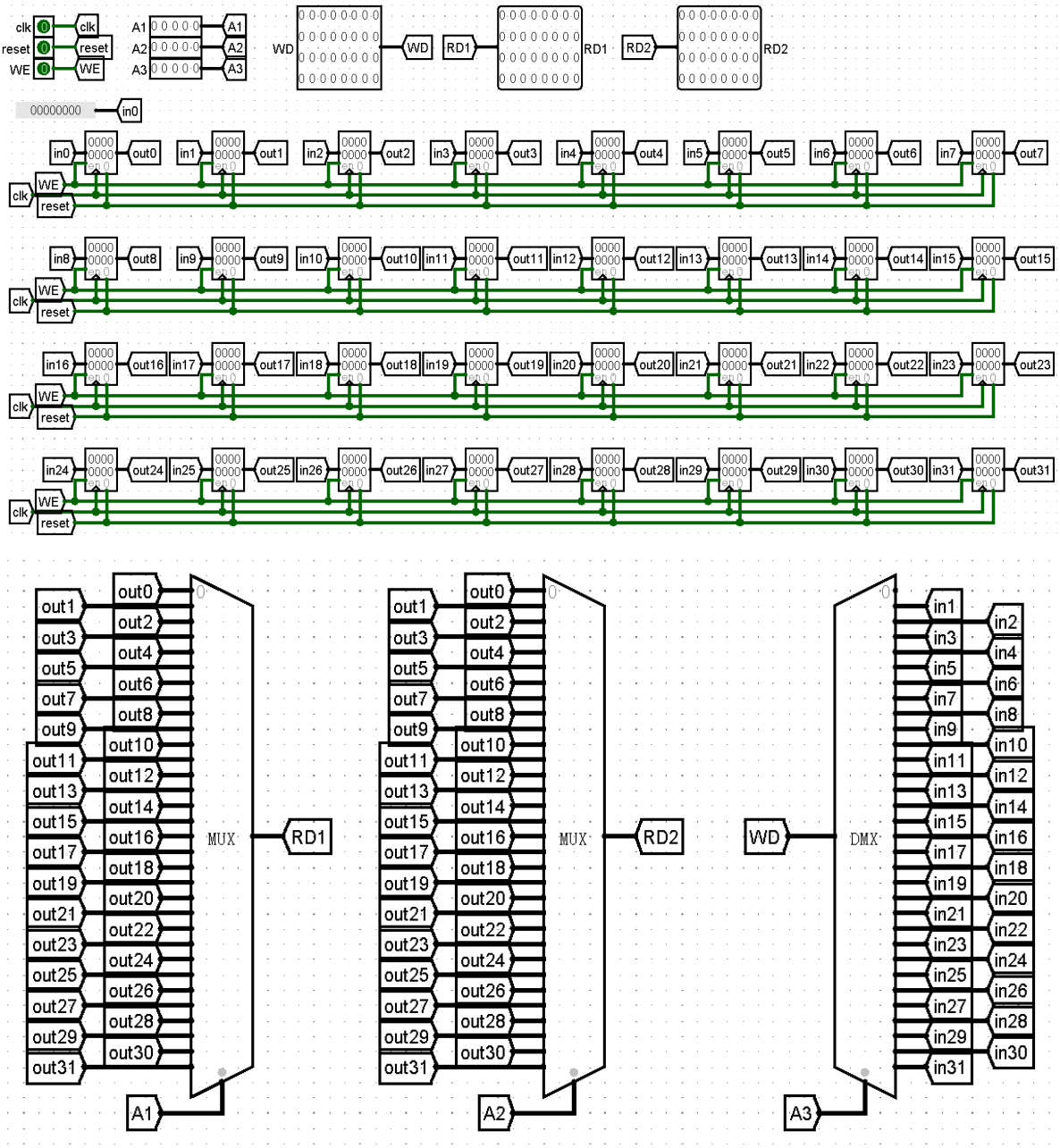
实现

整体



细节

参考 P0_L0_GRF 一题的实现。



ALU（算术逻辑单元）

提供 32 位加、减、或运算及大小比较功能。

加减法按无符号处理（不考虑溢出）。

端口定义

表2-3-1 ALU模块端口定义

信号名	方向	描述
A[31:0]	I	第一个32位计算数
B[31:0]	I	第二个32位计算数
ALUop[2:0]	I	指定ALU进行的计算
res[31:0]	O	运算结果
zero	O	标志A-B的结果是否为0 若A-B为0，zero置1，否则置0

功能定义

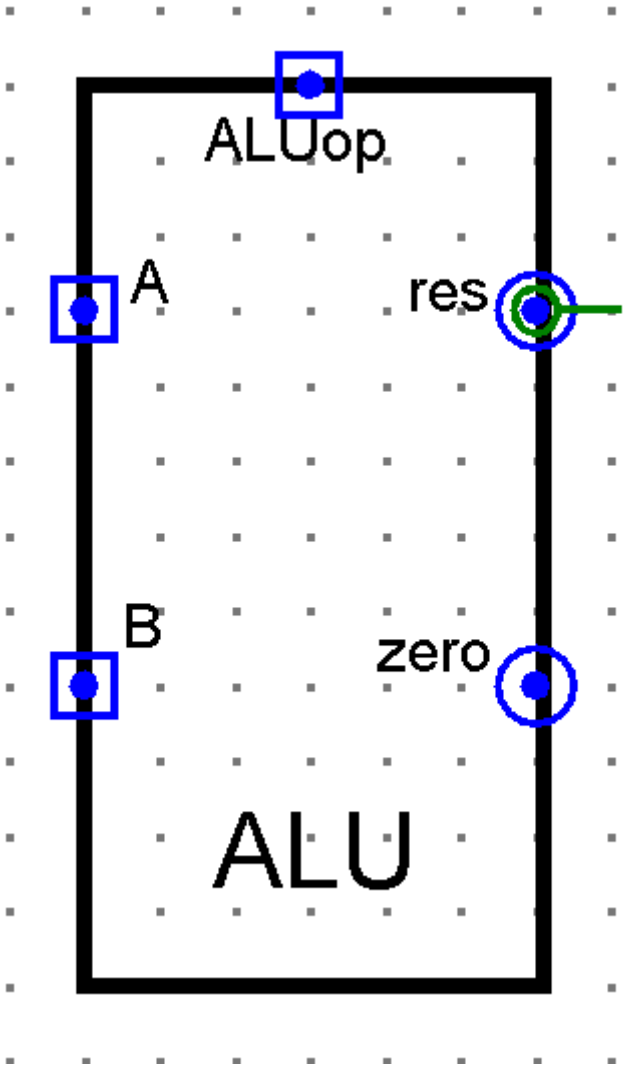
表2-3-2 ALU模块功能定义

ALUop	功能	描述
3'b000	加运算	res = A + B，不考虑溢出
3'b001	减运算	res = A - B，不考虑溢出 若res为0(A == B)，则zero置1，否则置0
3'b010	或运算	res = A | B
3'b011	B置高16位	res = B || 10^{16}

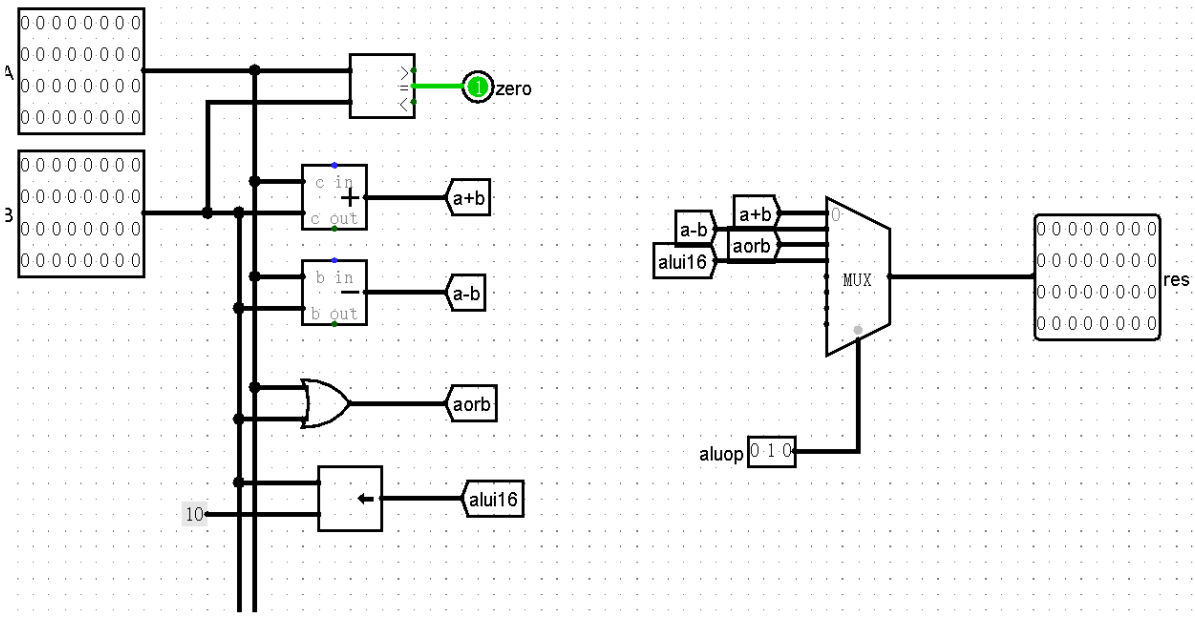
> 多余的ALUop为扩展指令预留。

实现

整体



细节



DM（数据存储器）

使用RAM实现，容量为3072 × 32bit，应具有**异步复位**功能，复位值为0x00000000。

起始地址：0x00000000。

地址范围：0x00000000 ~ 0x00002FFF。

RAM 应使用**双端口模式**，即设置 RAM 的 **Data Interface** 属性为 **Separate load and store ports**。

端口定义

表2-4-1 DM模块端口定义

信号名	方向	描述
clk	I	时钟信号
reset	I	**异步**复位信号，将DM内的RAM重置为0
WE	I	写使能信号，WE为1时，允许写入数据；WE为0时，禁止写入
A[31:0]	I	需要进行读/写操作的地址
WD[31:0]	I	写入RAM的32位输入数据
RD[31:0]	O	从RAM读出的32位输出数据

功能定义

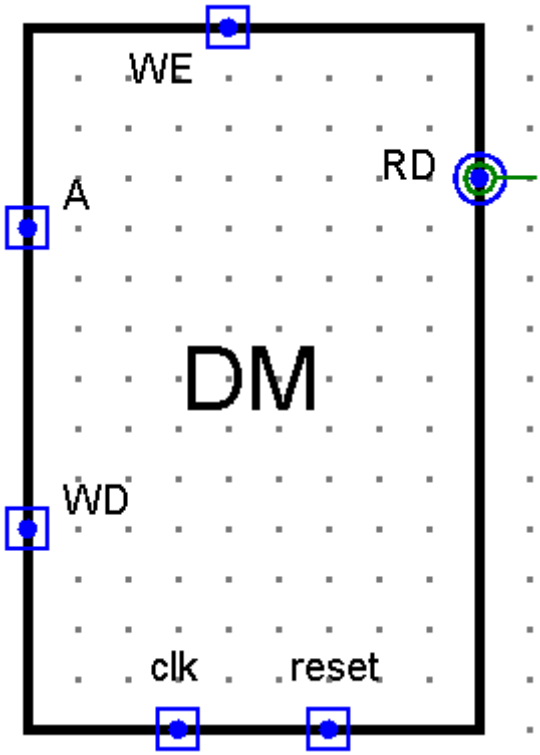
表2-4-2 DM模块功能定义

序号	功能	描述
1	异步复位	reset置1时，异步重置RAM内存为0
2	写数据	当**WE有效且时钟上升沿到来**时，将WD中的数据写入A对应的RAM地址中
3	读数据	读取A对应的RAM地址中存储的数据到RD

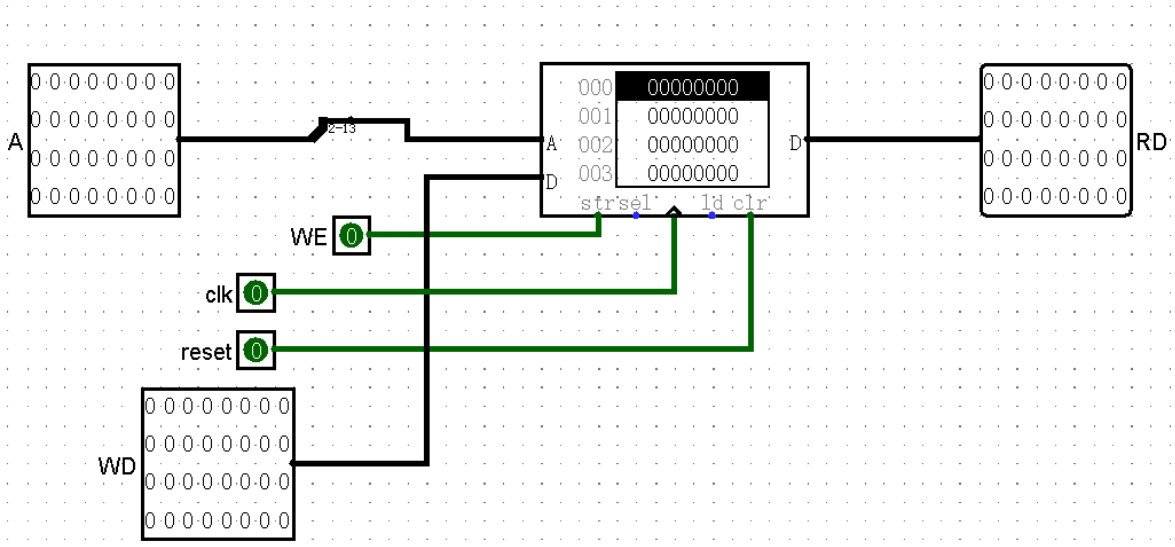
> 与处理IFU中地址的方法相同，使用A[13:2]即可从DM的RAM中正确读取数据。

实现

整体



细节



EXT(扩展单元)

使用Logisim内置的Bit Extender。

端口定义

表2-5-1 EXT模块端口定义

信号名	方向	描述
num[15:0]	I	需要扩展的16位立即数
sel	I	指定进行扩展的方式
result[31:0]	O	扩展完成的32位数

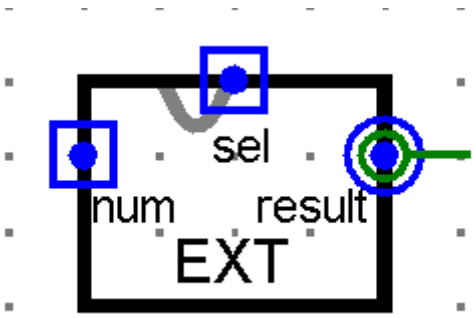
功能定义

表2-5-2 EXT模块功能定义

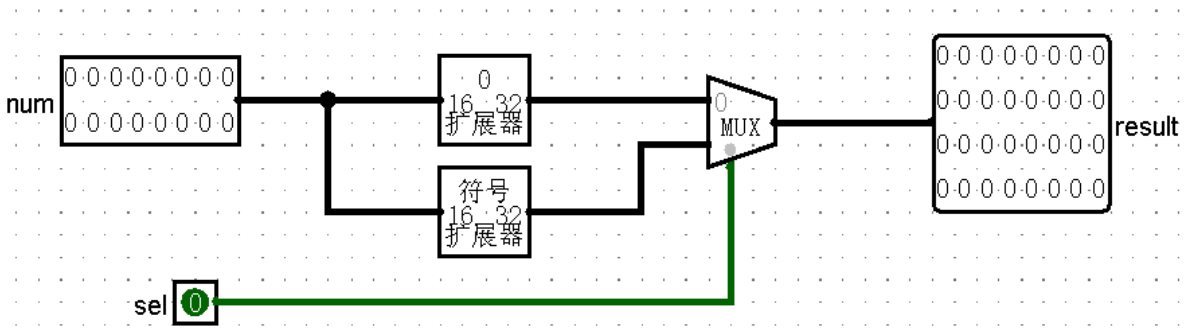
sel	功能	描述
1'b0	零扩展	result = zero_extend(num)
1'b1	符号扩展	result = sign_extend(num)

实现

整体



细节



Controller(控制器)

使用与或门阵列构造控制信号。

和逻辑的功能是**识别**，将输入的机器码识别为相应的指令；或逻辑的功能是**生成**，根据输入的指令的不同，产生不同的控制信号。

端口定义

表2-6-1 Controller模块端口定义

信号名	方向	描述
op[5:0]	I	32位指令Instr[31:26]
funct[5:0]	I	32位指令Instr[5:0]
RegDst	O	指定数据写入的寄存器序号 RegDst为0时，写入的寄存器序号来自Instr[20:16]，对应I型指令 RegDst为1时，写入的寄存器序号来自Instr[15:11]，对应R型指令
ALUSrc	O	指定ALU第二个运算数是否是立即数 ALUSrc为0时，运算数来自GRF ALUSrc为1时，运算数为立即数
MemToReg	O	指定写入GRF的数据的来源 MemToReg为0时，数据为ALU的输出res MemToReg为1时，数据为DM的输出RD
RegWrite	O	是否向GRF中写入数据
MemWrite	O	是否向DM中写入数据
beq	O	是否为beq指令
ExtOp	O	指定EXT进行立即数扩展的方式 ExtOp为0时，EXT进行零扩展 ExtOp为1时，EXT进行符号扩展

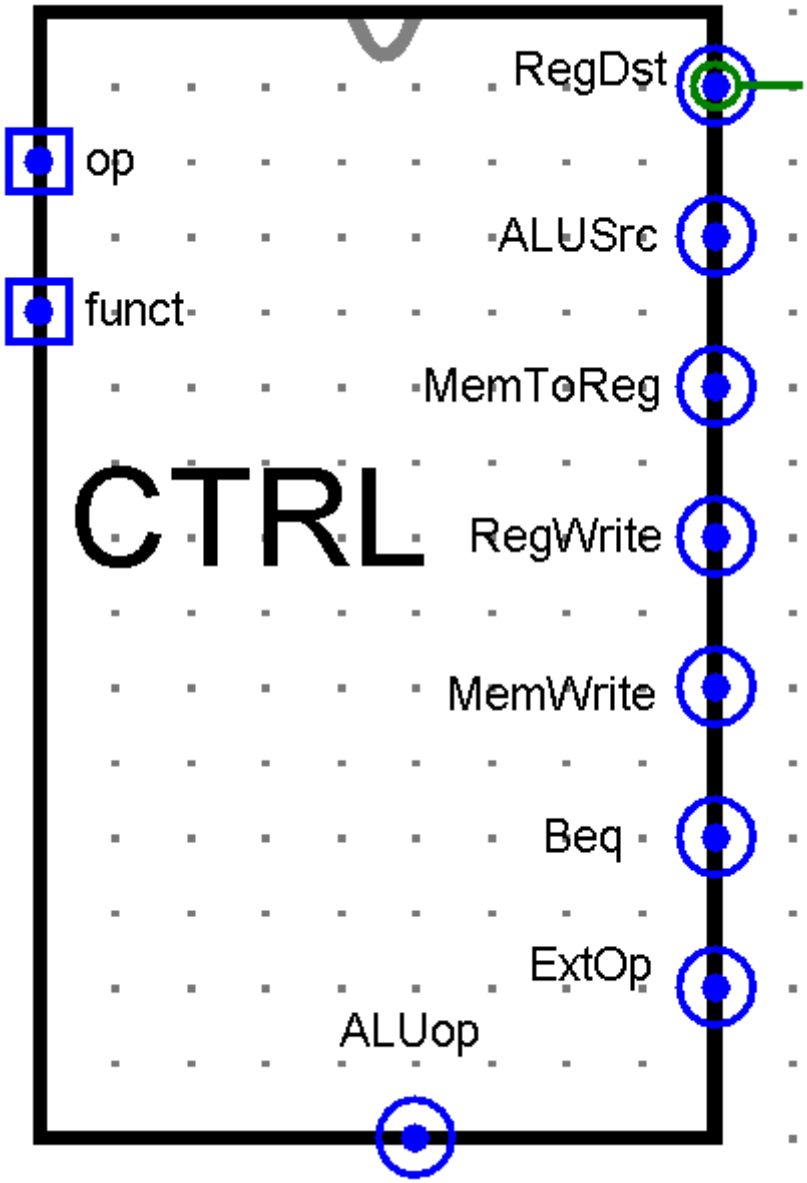
功能定义

表2-6-2 Controller模块功能定义

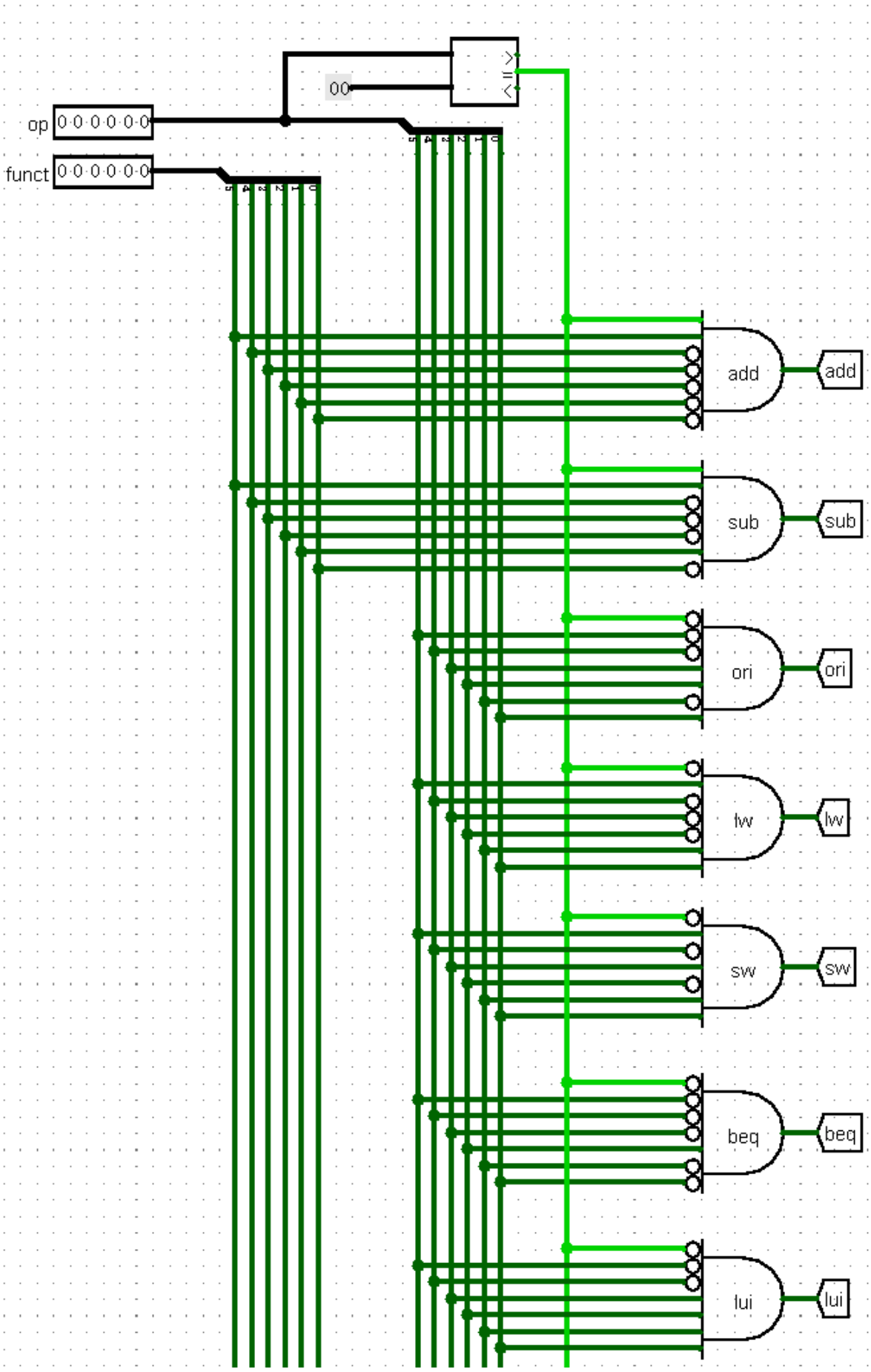
序号	功能	描述
1	生成控制信号	生成控制信号

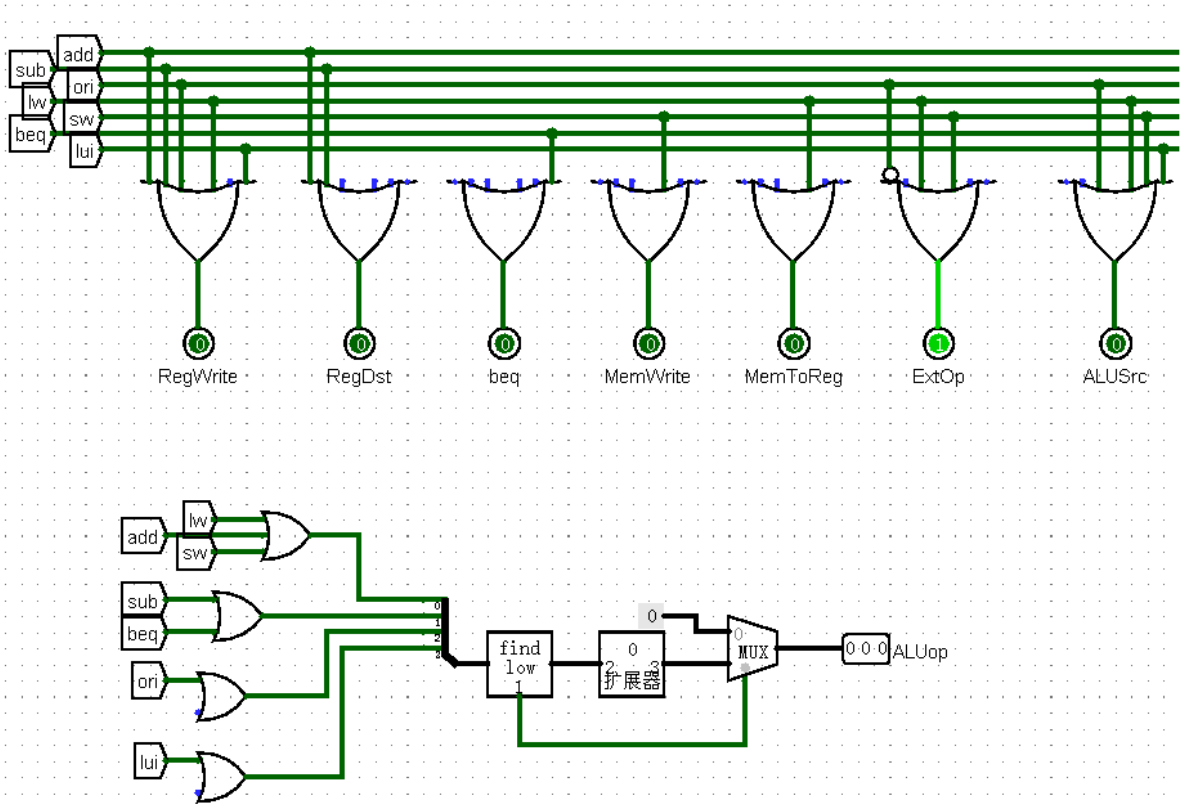
实现

整体


RegDst
op
ALUSrc
funct
MemToReg
CTRL
RegWrite
MemWrite
Beq
ExtOp
ALUop

细节


op
0 0 0 0 0 0
funct
0 0 0 0 0 0
00
add
sub
ori
lw
sw
beq
lui



重要机制实现方法

控制信号的生成

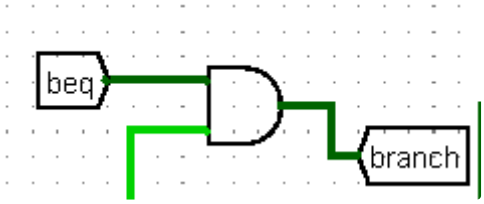
表3-1-1 控制信号生成真值表

	add	sub	ori	lw	sw	beq	lui
funct	100000	100010	undefined				
op	000000	000000	001101	100011	101011	000100	001111
RegDst	1	1	0	0	X	X	0
ALUSrc	0	0	1	1	1	0	0
MemToReg	0	0	0	1	X	X	0
RegWrite	1	1	1	1	0	0	1
MemWrite	0	0	0	0	1	0	0
beq	0	0	0	0	0	1	0
Extop	X	X	0	1	1	X	X
ALUop[2:0]	3'b000	3'b001	3'b010	3'b000	3'b000	3'b001	3'b011

beq指令的执行判断

beq指令执行的两个条件：

- Controller判断为beq指令；
- GPR[rs] == GPR[rt]，即ALU的zero信号为1。



其中亮绿色为zero信号。

测试方案

测试代码1

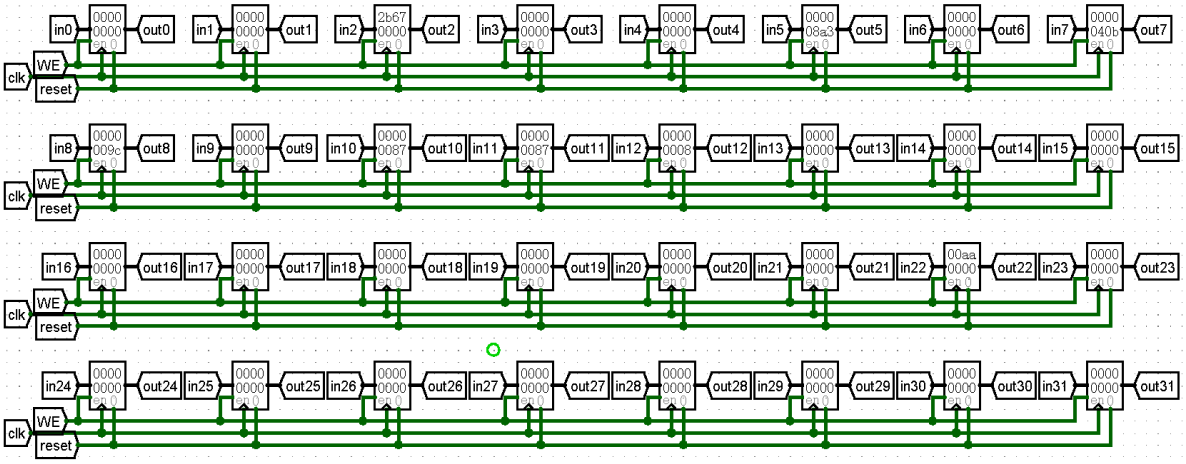
```
ori $t0,$0,156
ori $t2,$0,135
ori $a3,$a3,1035
lui $a1,101
ori $a1,$0,2211
nop
loop:
beq $t3,$t2,end
ori $t4,8
lui $s6,170
add $t3,$t3,$t4
add $t3,$t2,$0
lw $s0,4($t1)
out:
add $t2,$t2,$t1
sub $t3,$t2,$0
beq $t3,$t2,loop
end:
lui $v0,11111
```

导出为

```
v2.0 raw
3408009c
340a0087
34e7040b
3c050065
340508a3
00000000
116a0008
358c0008
3c1600aa
016c5820
01405820
8d300004
01495020
01405822
116afff7
3c022b67
```

结果如下：

GRF中的寄存器



MARS上的运行结果

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x2b670000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x000008a3
$a2	6	0x00000000
$a3	7	0x0000040b
$t0	8	0x0000009c
$t1	9	0x00000000
$t2	10	0x00000087
$t3	11	0x00000087
$t4	12	0x00000008
$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	0x00aa0000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00003040
hi		0x00000000
lo		0x00000000

发现是完全一致的。

思考题

1. 上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。

状态存储：GRF、DM

状态转移：IFU、ALU、EXT、Controller

2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗？请给出分析，若有改进意见也请一并给出。

我认为是合理的。

IM只需要被读取，而ROM是只读的，下次打开文件时内存依然存在，且运行过程中不会被篡改；

DM需要支持读、写功能，一个时钟周期内只会进行读、写的其中一种操作。RAM即可支持读写操作，又在占用空间上优于寄存器文件。

GRF需要支持读、写功能，且与ALU直接相连，对读、写速度要求较高，故使用寄存器文件。

3. 在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。

并未设计新的模块。

4.事实上，实现 nop 空指令，我们并不需要将它加入控制信号真值表，为什么？

Controller采用与或门阵列实现，读入nop指令时所有的控制信号均保持在低电平，只进行了PC ⟹ PC + 4，而不会产生其他任何操作。

5. 阅读 Pre 的“MIPS 指令集及汇编语言”一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。

我认为该样例覆盖了该CPU中支持的所有指令，且先由最基本的可独立判断正误的指令进行验证，之后再对更高层的指令的结果正误进行验证，能对CPU的设计起到较为准确的反馈。

可以考虑加入一些32位数、16位无符号数的边界情况，多增加一些目标寄存器为 $0 的指令，达到更好的测试效果。