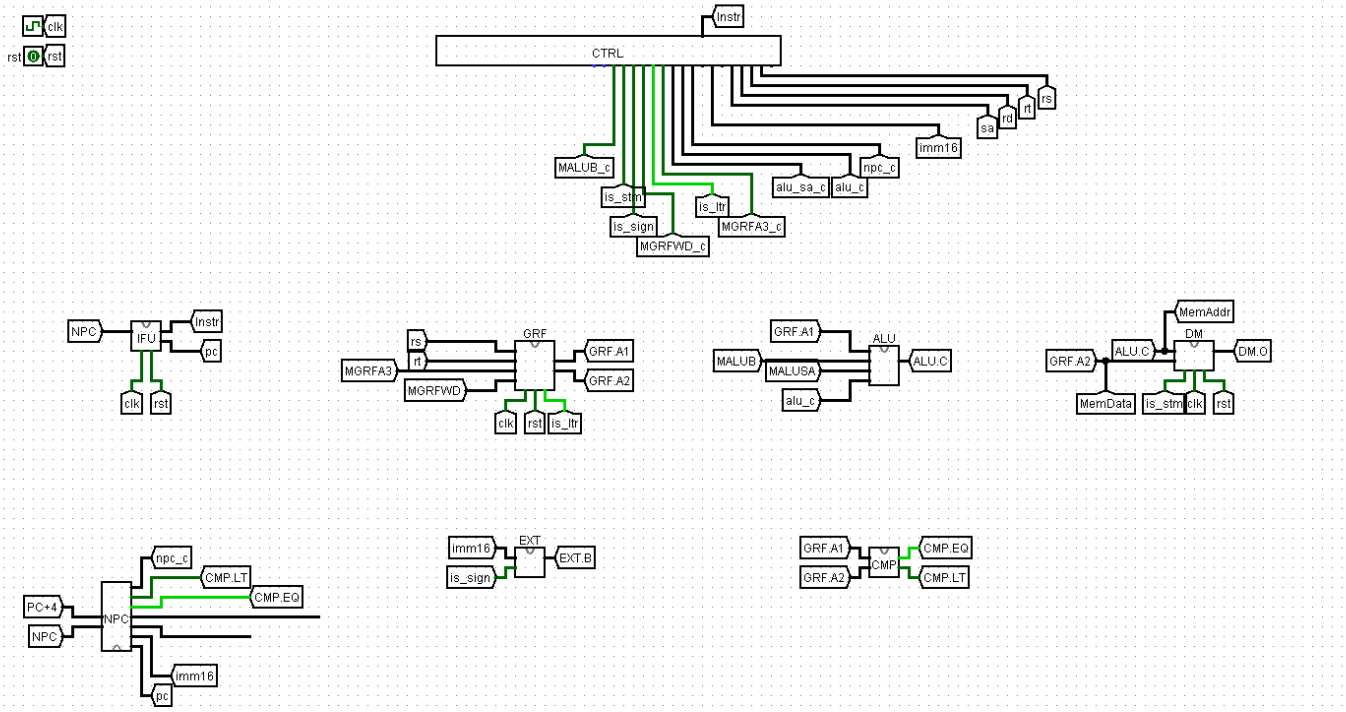
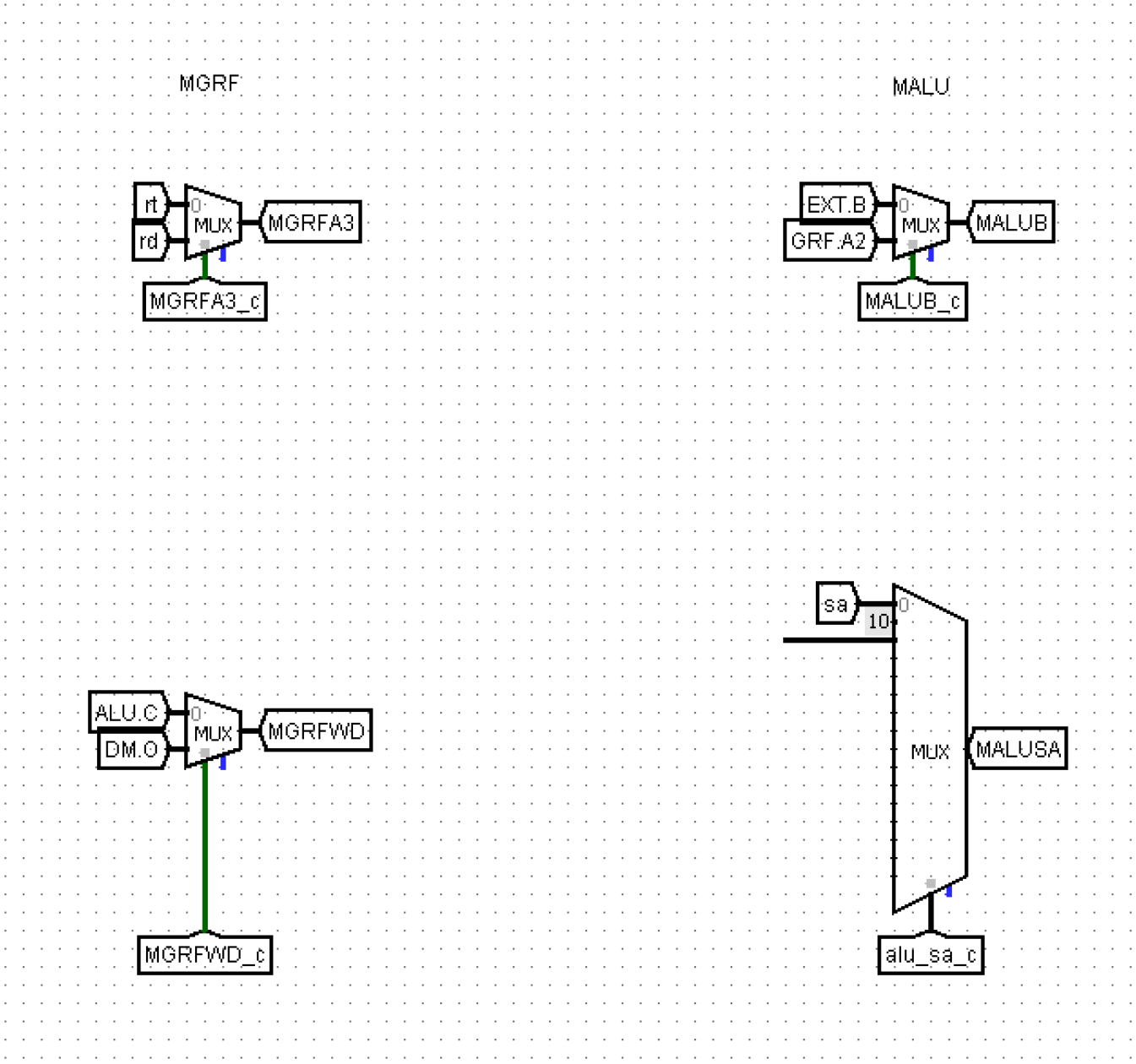
P3 - Logisim单周期CPU - 设计文档

要求实现一个支持`add, sub, ori, lw, sw, beq, lui, nop`的单周期CPU。

整体结构


clk
rst
rst
Instr
CTRL
rs
rt
rd
sa
imm16
npc_c
alu_c
alu_sa_c
MGRFA3_c
MGRFWD_c
is_ltr
is_sign
is_stm
MALUB_c
NPC
IFU
Instr
pc
clk
rst
rs
rt
MGRFA3
MGRFWD
GRF
GRF.A1
GRF.A2
clk
rst
is_ltr
GRF.A1
MALUB
MALUSA
alu_c
ALU
ALU.C
MemAddr
GRF.A2
ALU.C
DM
DM.O
MemData
is_stm
clk
rst
npc_c
CMP.LT
CMP.EQ
PC+4
NPC
NPC
imm16
pc
imm16
is_sign
EXT
EXT.B
GRF.A1
GRF.A2
CMP
CMP.EQ
CMP.LT



一个标准的单周期CPU模型罢了，并没有什么好说的。

考虑了MIPS指令集中`rs`，`rt`，`rd`等信号域位置恒定的设计，此处`GRF`的`A1`，`A2`恒接入`CTRL.RS`,`CTRL.RT`， `ALU`的`A`恒接入`GRF.RD1`。

但在后续阅读往年题目时，发现要求添加的指令出于微妙的原因，不完全符合MIPS指令集规范，更有强行将多周期指令塞入单周期指令者，ALU部分的恒接入有待商榷。此外，CMP的输入、DM部分的输入也可能需要加入MUX。

在设计的时候，为了预先考虑一下后续的改动，我们对模块分了一下层，相信由图里模块的位置关系也能略知一二。

- 层次1：`IFU`，`NPC`
- 层次2：`CTRL`，`GRF`，`EXT`

- 层次3：`ALU`，`CMP`
- 层次4：`DM`
- （实际上的）层次5: `GRF的A3, WE, WD部分`； 显然我们现在还没有分这个层的必要，故直接接上层次2里的GRF了。

思考题

- 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。

  `NPC` 以及所有逻辑电路模块（`ALU` 等）进行状态转移， `DM` 与 `GRF` 进行状态存储。

  `IM` 在初始化便不变，我们姑且认为其不属于状态存储或转移部分。硬要说的话，它也发挥状态存储。

模块规格

控制器的设计单独给出。

IFU

分析

由于Logisim自带库的限制，我们只能整字读取ROM内的数据，故需要对输入的PC信号进行一些处理。

此外，由于复位时要回到的是 `0x00003000` 而非 `32'b0`，我们决定采取如下处理办法：先将 `NPC` 减去 `0x00003000` 得相对地址；再取相对地址的 `[13:2]` 位，接入一12位寄存器。这一寄存器将存储**以字而非字节为单位，且相对初始地址**的 `IM` 地址。

取 `[13:2]` 的原因在于：字对齐，低2位必为0；内存地址为 `0x00003000 – 0x00006FFF`，大小为 `0x00003fff`，相对地址的高18位亦必为0。截取下来，就是 `[13:2]`。

端口定义

端口/位宽	方向	说明
NPC[31:0]	I	输入NPC单元计算得出的NPC值
INS[31:0]	O	读出PC指向的IM地址之指令
PC[31:0]	O	传出当前PC值

NPC

分析

计算下一PC值。

我们这里已有了 `branch` 类指令，故需要控制信号信号指示NPC模块进行正常运算，还是进行 `branch` 类对应的立即数操作。

由于 `branch` 类的立即数操作较为特殊，使用 `EXT` 操作过于困难，遂直接传入NPC进行特殊计算。同时接入来自 `CMP` 的比较信号，判断是否跳转。

端口定义

端口名	方向	说明
PC[31:0]	I	读入当前PC值
IMM16[15:0]	I	传入 `branch` 类指令的立即数
EQ	I	来自CMP的比较结果
LT	I	来自CMP的比较结果
NPCOp[3:0]	I	控制信号
NPC[31:0]	O	计算出的NPC值
clk	I	时钟信号
rst	I	异步复位信号

控制信号

NPCOp	4'b0000	4‘b0001
功能	PC+4	branch
说明	正常操作	执行 `branch` 类指令

EXT

分析

在 `ori` 中，我们需将16位立即数进行无符号扩展；而在 `beq` 中，我们需要对16位立即数进行符号拓展。

因此，我们的符号拓展应支持这两种拓展方式。

端口定义

端口/位宽	方向	说明
A[15:0]	I	输入的16位立即数
EXTOp	I	指示拓展方式
B[31:0]	O	扩展结果

控制信号

EXTOp	1'b0	1'b1
功能	ext_ze	ext_se
说明	零扩展	符号扩展

GRF

分析

没什么好说的，直接搓即可。

之前P0课下做过了，直接复用。

端口定义

端口/位宽	方向	说明
A1[4:0]	I	要读取的$[rs]之地址
A2[4:0]	I	要读取的$[rt]之地址
A3[4:0]	I	要写入的$[rd]之地址
WD[31:0]	I	写入的数据
WE	I	写使能信号
RD1[31:0]	O	读取的$[rs]
RD2[31:0]	O	读取的$[rt]
rst	I	异步复位信号
clk	I	时钟信号

控制信号

本栏目中控制信号是对于同一模块的不同单元而言的。此处只有读/写功能，故没有这个意义下的“控制信号”。

时钟，写使能等控制信号已在上表中给出。

ALU

分析

经分析，我们得知其至少满足**加、减、或三个操作，且不考虑溢出**

考虑扩展，我们ALUOp给到4位之多。

在MIPS指令集中，`rs`，`rt`，`rd`三者的位置在机器码中是固定的；因此，我们希望ALU的输入亦严格与`rs`、`rt（或立即数IMM）`对应。

一番分析之后，假设要加入移位操作，其运算层级应该是与ALU一致的，遂将其一并划入ALU中。

这又引起了问题：我们此时的IMM是由`A2`输入的，**对于`beq`等需要比较的指令，ALU无法同时进行地址计算与比较**，故将比较功能单独交给一名为`CMP`的比较单元进行。

值得注意的是，为保持各模块间的层次，对于`branch`与`jump`类指令的立即数移位等操作，我们交由NPC进行。

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
ALUOp[3:0]	I	ALU控制信号
B[31:0]	I	输入RT或IMM
SA[4:0]	I	输入移位位数
C[31:0]	O	输出结果

控制信号

ALUOp	4'b0000	4'b0001	4'b0010	4'b0011
功能	alu_add	alu_sub	alu_ori	alu_sll

ALUOp	4'b0000	4'b0001	4'b0010	4'b0011
说明	加运算	减运算	或立即数	左移运算

CMP

分析

进行比较操作。

考虑到`bgt`，`bge`，`blt`等指令的比较情况，由于我们此时的输入顺序固定，故我们需要有关比较结果的两个输出，分别对应等于和小于。大于、大于等于，小于等于的综合则交给要使用比较结果的其他单元去做。

端口定义

名称/位宽	方向	说明
A[31:0]	I	输入RS
B[31:0]	I	输入RT或IMM
EQ	O	(A==B)
LT	O	(A < B)

DM

分析

与IFU相似，RAM还是只能整字读取，我们仍然需要对输入的信号进行处理。

DM的地址范围是`0x00000000 ~ 0x00002FFF`，字对齐，故高16+2位，低2位均视作0，截取`[13:2]`作为RAM的地址信号。

注意到题目所给的RAM规格应为`3072 × 32bit`，但很明显3072不是2的整数次幂，故我们的ROM还是给到`4096 × 32bit`。

端口定义

端口/位宽	方向	说明
ADDR[31:0]	I	DM地址
WD[31:0]	I	待写入的数据

端口/位宽	方向	说明
WE	I	写使能信号
clk	I	时钟信号
rst	I	异步复位信号
O	O	读出指定地址的数据

思考题

- 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。

  从功能的角度来看是合理的。

  - 对于IM，我们的写入操作仅在初始化执行指令的时候进行，此后我们对其操作只有读操作，没有写操作，这符合ROM（烧录后）只读的特性
  - 对于RAM，在执行`lw`、`sw`等指令时，我们需要对**指定内存地址**进行**读、写**操作，与RAM的特性相符。**而且内存实际也是RAM。**
  - 对于GRF，GRF本身即是**寄存器**堆，用寄存器是自然的。对指定编号的寄存器进行访问，看起来RAM似乎更适合些...吗？答案是否定的，因为寄存器的读写速度实际比RAM快，我们需对此进行区分。

  从实际情况而言，**IM、DM部分的实现欠妥。**

  - 因为IM、DM均是**内存的一部分**，不应将其分开。

  如果我们远视一点，不小心多看了点流水线有关的知识，我们会发现**我们只能这么做。**

  - 对于`lw`指令，我们执行到M级时，我们要对DM进行读操作；**与此同时，我们要从IM中取出接下来的指令，若DM、IM未分开的话，会在仅有的一个读取地址端口处产生不可避免的冲突。** 为了避免冲突，我们只能这么办。
  - 实际是一体的，那怎么办？见改进意见

  改进意见：我们保持IM、DM一体不变，转而将结构中的`IM`、`DM`改成两个缓存：`IM_Cache`、`DM_Cache`。这两者将对对应内存区段的数据进行读写。当然，我们现在还没学到缓存...
- 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。

  有的。见上文的`CMP`模块。至于为何需要加入这一模块及设计思路，详见`ALU`的分析部分。

数据通路与MUX

数据通路

对于IFU输入必来自NPC此类的显然连接，我们在表中略去,在下单独给出。

IFU.NPC = NPC.NPC
CTRL.INS = IFU.INS
NPC.PC = IFU.PC
ALU.A = GRF.A1
CMP.A1 = GRF.A1，CMP.A2=A2

输入/指令	**add**	**sub**	**sll(nop)**	**ori**	**lw**	**sw**	**beq**
NPC.NPCOp	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	npc_norm	npc_
GRF.A3	RD	RD	RD	RT	RT	/	/
GRF.WE	grf_we_e	grf_we_e	grf_we_e	grf_we_e	grf_we_e	grf_we_d	grf_
GRF.WD	ALU.C	ALU.C	ALU.C	ALU.C	DM.O	/	/
EXT.EXTOp	/	/	/	ext_ze	ext_se	ext_se	/
ALU.ALUOp	alu_add	alu_sub	alu_sll	alu_ori	alu_add	alu_add	/
ALU.B	GRF.A2	GRF.A2	GRF.A2	EXT.B	EXT.B	EXT.B	/
ALU.SA	/	/	SA	/	/	/	/
DM.ADDR	/	/	/	/	ALU.C	ALU.C	/
DM.WE	dm_we_d	dm_we_d	dm_we_d	dm_we_d	dm_we_d	dm_we_e	dm_
DM.WD	/	/	/	/	/	GRF.A2	/

(1):此处的移位方式并不重要，用sign_extend也是可以的...
(2):

MUX

MALUB_c	**1'b0**	**1'b1**
MALUB	EXT.B	GRF.A2

alu_sa_c	**4'b0000**	**4'b0001**	**4'b0010**
MALUSA	CTRL.SA	5'b16	(reserved)

MGRFA3_c	1'b0	1'b1
MGRFA3	CTRL.RT	CTRL.RD

MGRFWD_c	1'b0	1'b1
MGRFWD	ALU.C	DM.O

MUX控制信号及其它控制信号的取值，详见控制器-或阵列部分。

控制器设计

端口/位宽	方向	说明
INS[31:0]	I	指令机器码
opcode[5:0]	O（下同）	输出机器码对应信号域（下同）
rs[4:0]		
rt[4:0]		
rd[4:0]		
sa[4:0]		
funct[5:0]		
imm16[15:0]		
imm26[25:0]		
npc_c[3:0]		`NPC` 控制信号
alu_c[3:0]		`ALU` 控制信号
alu_sa_c[3:0]		`MALUSA` 控制信号；以下控制信号对应MUX，详见上文数据通路分析。
MGRFA3_c		`MGRFA3控制信号` 。 IS_R-type_INStruction?
is_ltr		IS_Load_To_Register? 控制GRF.WE
MGRFWD_c		`MGRFWD` 控制信号。IS_Load_From_Memory?
is_sign		IS_SIGNed? 控制EXT.Op
is_stm		IS_Store_To_Memory? 控制DM.WE
MALUB_c		`MALUB` 控制信号。

其中 `opcode` 和 `funct` 的输出是多余的；为了可能的debug需要，还是将其保留了下来。

我们分成**解码、与阵列、或阵列**三个部分进行设计。

解码（CTRL.DEC）

将32位机器码拆分成 `opcode` 等信号域。

各信号域具体取值参见MIPS指令集，此处不多赘述。

端口/位宽	方向
INS[31:0]	I
opcode[5:0]	O
rs[4:0]	O
rt[4:0]	O
rd[4:0]	O
sa[4:0]	O
funct[5:0]	O
imm16[15:0]	O
imm26[25:0]	O

与阵列（CTRL_AND）

根据解码部分传入的 `opcode` 与 `funct`，进行指令的翻译。

考虑到后续添加指令的扩展需求，应留出扩展空间。

注意到部分控制信号（如MALUB）仅与指令格式类型有关，故给出 `r_ins` 输出，指示指令是否为R型指令。

端口/位宽	方向
opcode[5:0]	I
funct[5:0]	I
r_ins	O（下同）
add	
sub	
sll	
beq	

端口/位宽	方向
lui	
lw	
sw	
ori	
(tbc.)	

其中，`r_ins = !|(opcode)`

或阵列（CTRL_OR）

根据与阵列的指令翻译结果，进行MUX与其它控制信号的编码。

输入的指令信号同CTRL_AND的输出，不再单独列出，此处主要描述各输出信号的输出逻辑。

输出信号:	npc_c[3:0]	alu_c[3:0]	alu_sa_c[3:0]	MGRFA3_c	is_ltr	MGRFWD_c	is_sig

```
MGRFA3_C = MALUB_C = r_ins
is_ltr = ! (sw | beq)
MGRFWD_c = (lw)
is_sign = (lw | sw)
is_stm = (sw)
npc_c = (b)? 4'b0001 :
                (j)? 4'b0010 :
                (i)? 4'b0011 :
                4'b0000
        b = (beq)
        j = (reserved for future development)
        i = (reserved for future development)
alu_c = (alu_add)? 4'b0000 :
                (alu_sub)? 4'b0001 :
                (alu_ori)? 4'b0010 :
                (alu_sll)? 4'b0011 :
                XXXX
        alu_add = (add | lw | sw)
        alu_sub = (sub)
```

```
        alu_ori = (ori)
        alu_sll = (sll | lui)
alu_sa_c = (shamt)? 4'b0000 :
                    (dec16)? 4'b0001 :
                    (v)? 4'b0010 :
                    XXXX
        shamt = (sll)
        dec16 = (lui)
        v = (reserved for future development)
```

思考题

- 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？
  我们先看高六位：`000000`，说明其为R型指令

  继续看低六位：还是`000000`，说明其为逻辑左移命令

  剩下几位均为0，翻译结果：`sll $0,$0,0`

  不难发现，这段指令执行之后并不会对GRF、DM产生任何影响，故不需要特判，直接利用已有的`sll`编码即可。

  还有一种想法，就是rs、rt、rd域的地址均为0；对`$0`做什么操作，都不会有影响

测试CPU

测试程序如下：

```
.text
# 操控$5为0xffff_ffff,$6为0x8000_0000
# 测试$0
ori $0,$0,65535
# 若通过，测试ori；zero_ext
ori $3,$0,4096 # 高第三位为1
ori $3,$3,65534 #最后一位0，高第三位为1
# 若通过，测试lui
lui $3,65535
lui $0,65535
# 若通过，测试减
ori $4,$0,5 #init
ori $2,$0,1
sub $3,$4,$2 #正常减法
```

```
ori $2,$0,6
sub $3,$3,$2 #跨到-1
ori $2,$0,1
sub $3,$6,$2 # -2137483648 - 1，溢出到2137483647
sub $0,$3,$1 # $0
# 若通过，测试加
ori $4,$0,5 #init
ori $2,$0,1
add $3,$4,$2 #正常+法
ori $2,$0,1
sub $3,$0,$2 #$3跨到-1
add $3,$3,$2 #-1跨到0
ori $2,$0,1
add $3,$5,$2 # 2137483647 + 1，溢出到-2137483648
add $0,$3,$2 # $0
# sll
ori $3,$0,65535
sll $3,$3,16
sll $0,$3,1
sll $0,$0,0 #nop，无事发生
# 通过，测试sw
ori $3,$0,1
ori $2,$0,4
sw $3,0($2)
sw $3,4($2)
sw $3,-4($2)
# lw
# 完成上述测试后，0，4，8应存有1
ori $4,$0,1
ori $2,$0,4
lw $3,4($2)
lw $3,0($2)
lw $3,-4($2)
lw $0,0($2) # $0
# beq
ori $2,1
ori $3,2
ori $4,1
that:
beq $2,$3,that
```

```
beq $2,$2,next
ori $2,5
next:
beq $2,$4,this
this:
beq $2,$2,this
```

在目力观察+借用讨论区各路仙人的测试程序验证后，确定该设计可以通过上述样例。

思考题

- 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。

  测试一般情况是够了，但是部分特例没有考虑：
  - 写入寄存器类指令（add,sub等）均没有测试目标写入寄存器为$0的情况。
  - 加减操作没测溢出。
  - 跳转操作只有向后跳，没有向前跳。（无法测试立即数符号扩展是否正确）
  - 跳转操作，若比较为真且偏移为0，相当于没有跳转；偏移为-1，则相当于陷入循环，也没有测。
  - lw/sw可能存在溢出问题。