2023秋P3·CPU设计文档

Part 1 CPU设计草稿

1.1 总体概述

1.1.1 CPU类型

本CPU为**单周期MIPS - CPU**，目前仅由**Logisim**实现。

1.1.2 CPU结构

CPU包含**GRF、EXT、ALU、DM、Splitter、IFU、Controller**七个模块。

1.1.3 CPU支持的指令集

{**add, sub, ori, lw, sw, beq, lui, nop**}

> 要求中字面上是add和sub指令，是为了方便以后扩展功能。但是实际上本次开发只需考虑无符号数即可，更接近于addu和subu指令。

1.2 模块设计

1.2.1 GRF

这部分基本上参考**P0**的**grf**模块的设计。

1.2.1.1 端口定义

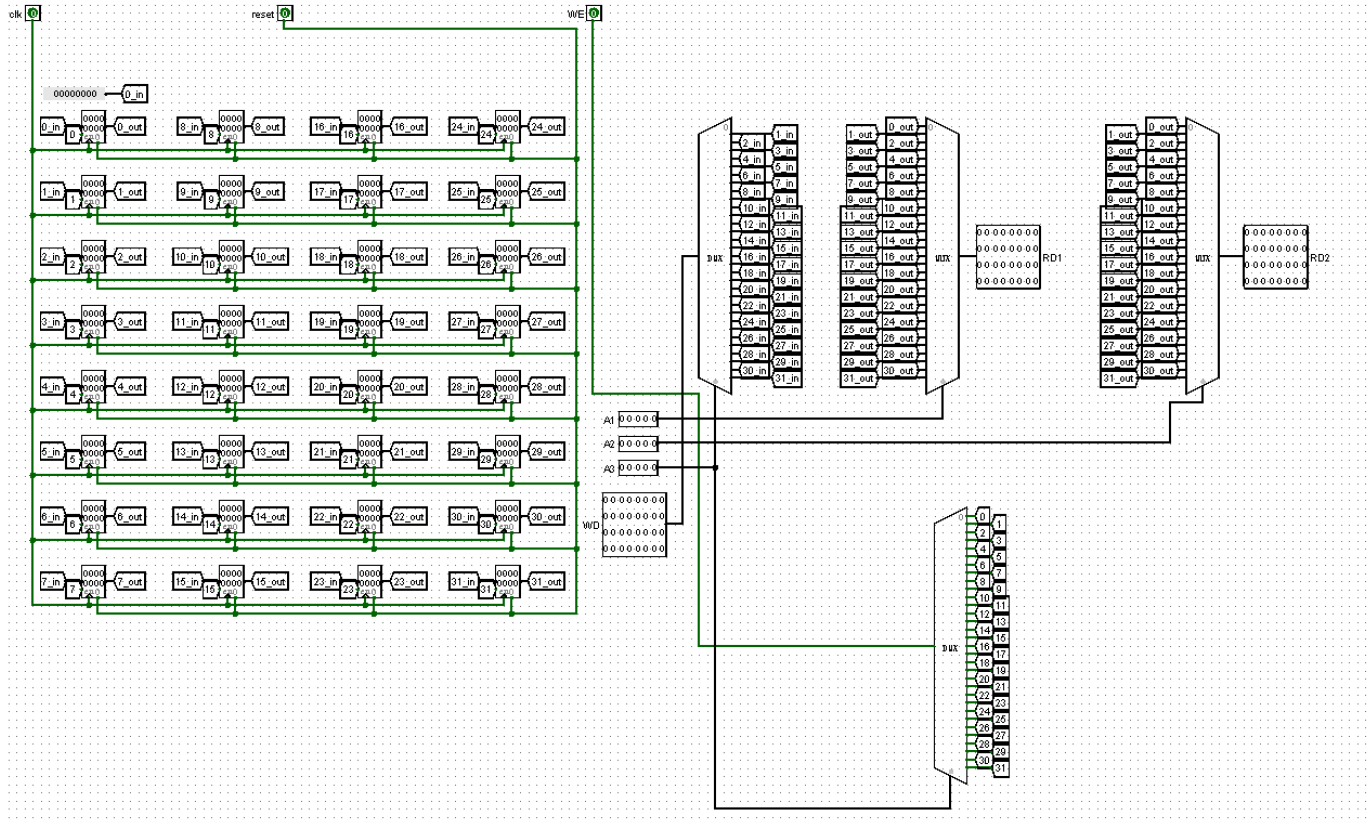
端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	异步复位信号，与logisim元件自带端口功能相同
WE	I	1	写使能信号：1，允许写入；0，不可写入
A1	I	5	读出数据的地址1
A2	I	5	读出数据的地址2
A3	I	5	写入数据的地址
WD	I	32	写入数据
RD1	O	32	读出数据1
RD2	O	32	读出数据2

1.2.1.2 功能定义

- 读数据
  - 读数据1：根据地址A1，读出对应的数据RD1
  - 读数据2：根据地址A2，读出对应的数据RD2
- 写数据
  - 写数据：WE为1时且时钟上升来临时，根据地址A3，写入数据WD
- 异步复位

- 当reset为1时，将所有寄存器清零

1.2.1.3 模块截图展示



1.2.2 EXT

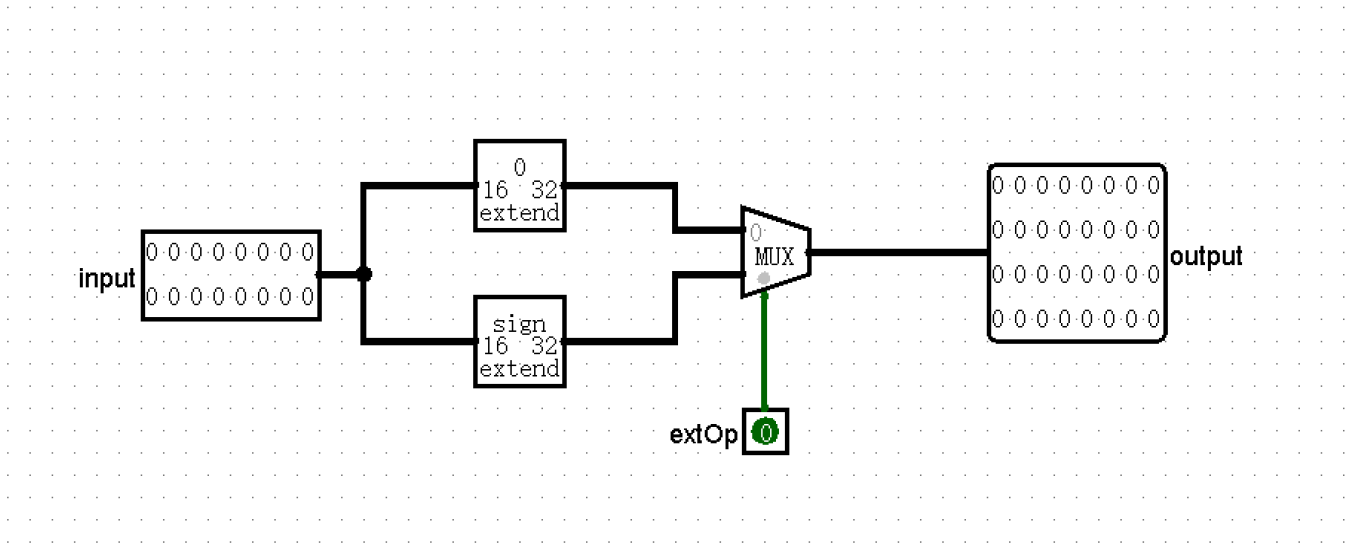
1.2.2.1 端口定义

端口名	方向	位宽	信号描述
input	I	16	输入数据,16位立即数
extOp	I	1	扩展操作类型：1，符号扩展；0，无符号扩展
output	O	32	输出数据,32位立即数

1.2.2.2 功能定义

- 位扩展
  - 将16位的立即数扩展为32位的立即数，提供符号扩展和无符号扩展两种方式

1.2.2.3 模块截图展示



1.2.3 ALU

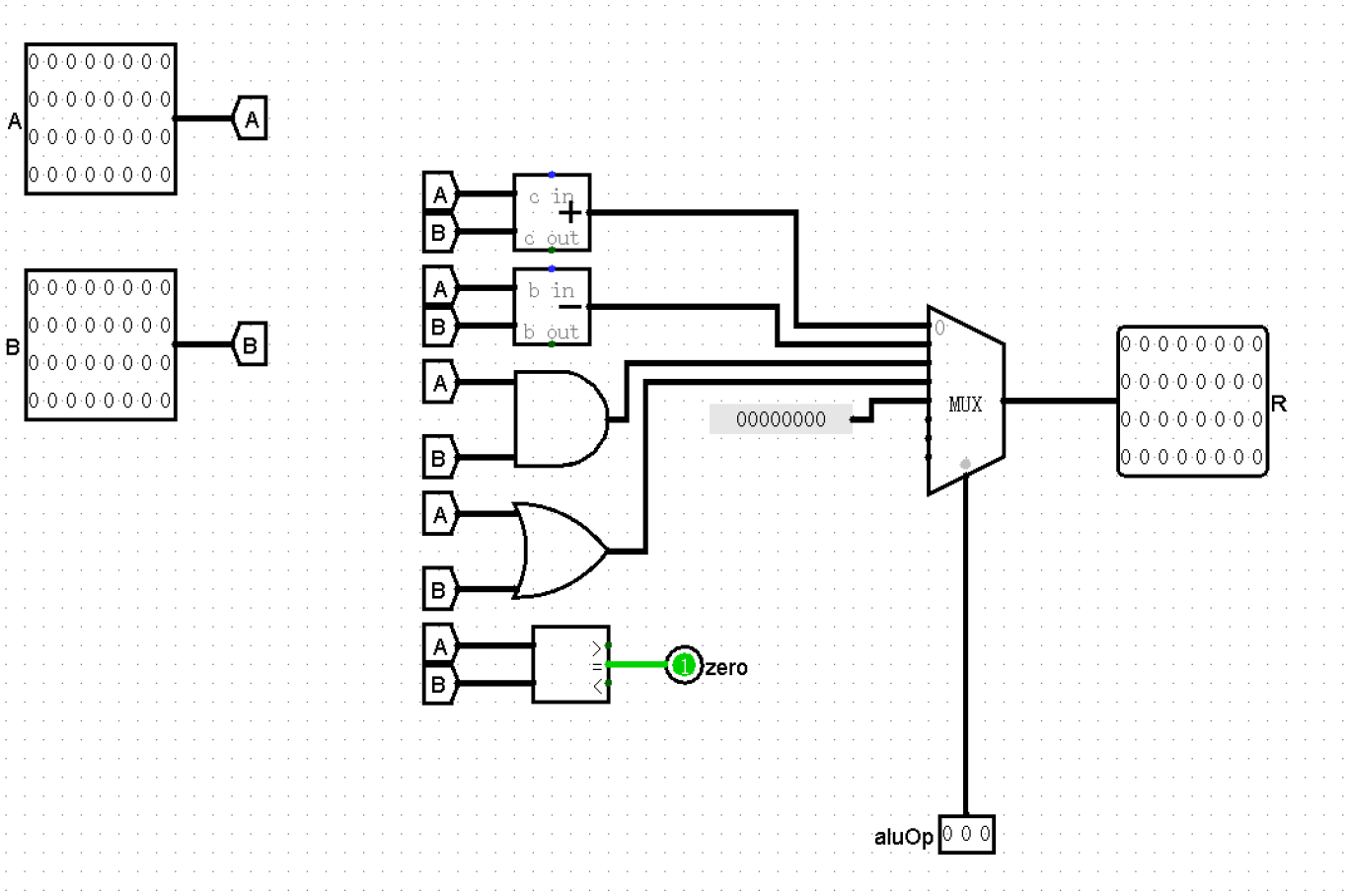
1.2.3.1 端口定义

端口名	方向	位宽	信号描述
A	I	32	第一个操作数
B	I	32	第二个操作数
aluOp	I	3	ALU操作类型：000，加；001，减；010，按位与；011，按位或；100，判断相等
R	O	32	运算结果
zero	O	1	相等状态码：1，相等；0，不相等

1.2.3.2 功能定义

- 算术运算
  - 加法：R = A+B
  - 减法：R = A-B
- 位运算
  - 按位与：R = A&B
  - 按位或：R = A|B
- 判断相等
  - 判断A和B是否相等，相等时zero为1，不相等时zero为0；**此时R输出端置为0**

1.2.3.3 模块截图展示



1.2.4 DM

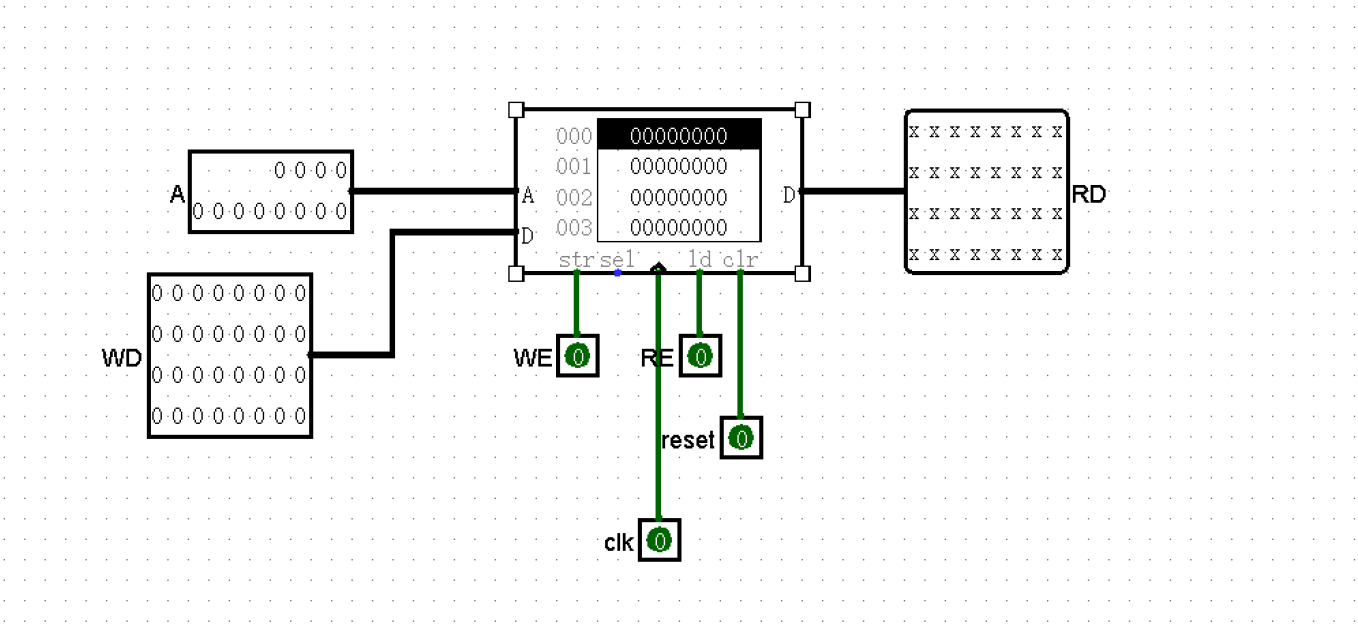
1.2.4.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	异步复位信号，与logisim元件自带端口功能相同
WE	I	1	写使能信号：1，允许写入；0，不可写入
RE	I	1	读使能信号：1，允许读出；0，不可读出
A	I	12	读写地址：范围（以字节为单位）是0x0000-0x2fff，共3072字
WD	I	32	写入数据
RD	O	32	读出数据

1.2.4.2 功能定义

- 读数据
  - 读数据：RE为1时且时钟上升来临时，根据地址A，读出数据RD
  - 读数据时，若地址超出范围，则读出0
- 写数据
  - 写数据：WE为1时且时钟上升来临时，根据地址A，写入数据WD
- 异步复位
  - 当reset为1时，将所有内容清零

1.2.4.3 模块截图展示



1.2.5 Splitter

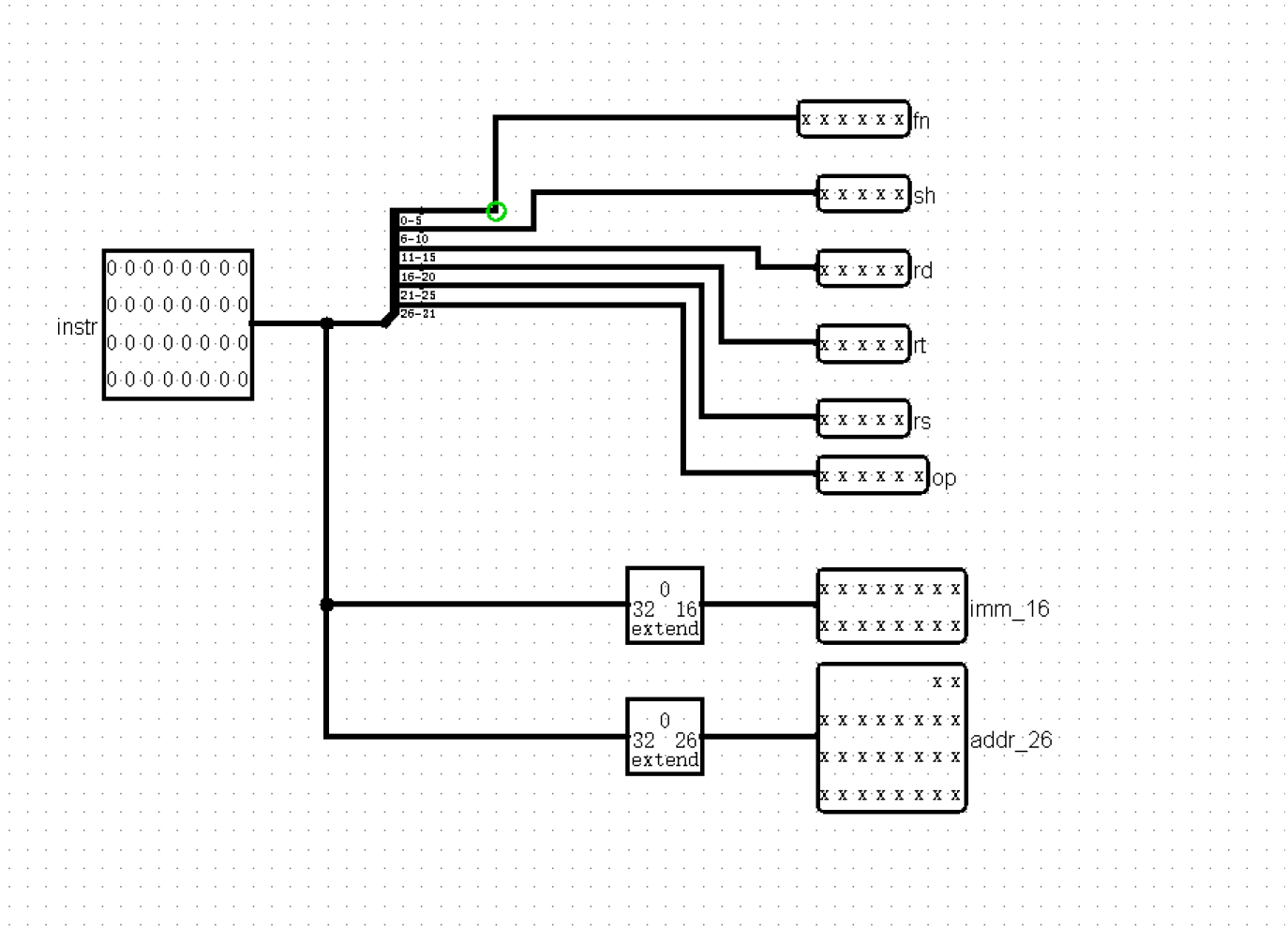
1.2.5.1 端口定义

端口名	方向	位宽	信号描述
instr	I	32	输入32位指令码
op	O	6	操作码
rs	O	5	源寄存器
rt	O	5	目标寄存器
rd	O	5	目的地寄存器
sh	O	5	移位位数
fn	O	6	功能码
imm_16	O	16	16位立即数
addr_26	O	26	26位地址

1.2.5.2 功能定义

- 指令解析
  - 将32位指令码解析为6位操作码、5位源寄存器、5位目标寄存器、5位目的地寄存器、5位移位位数、6位功能码、16位立即数或是26位地址码。目前支持**1.1.3中的8条指令。**

1.2.5.3 模块截图展示



1.2.6 IFU

1.2.6.1 端口定义

端口名	方向	位宽	信号描述
clk	I	1	时钟信号
reset	I	1	异步复位信号，复位使PC的值为0x3000（起始地址）
zero	I	1	相等状态码：1，相等；0，不相等
jumpOp	I	3	跳转操作类型：000，无跳转；001，beq（目前只有一种，待扩展）
offset	I	16	跳转偏移量
nInstr	O	32	下一条32位指令数据
pc	O	32	PC值，用于显示当前指令的地址(按字节)

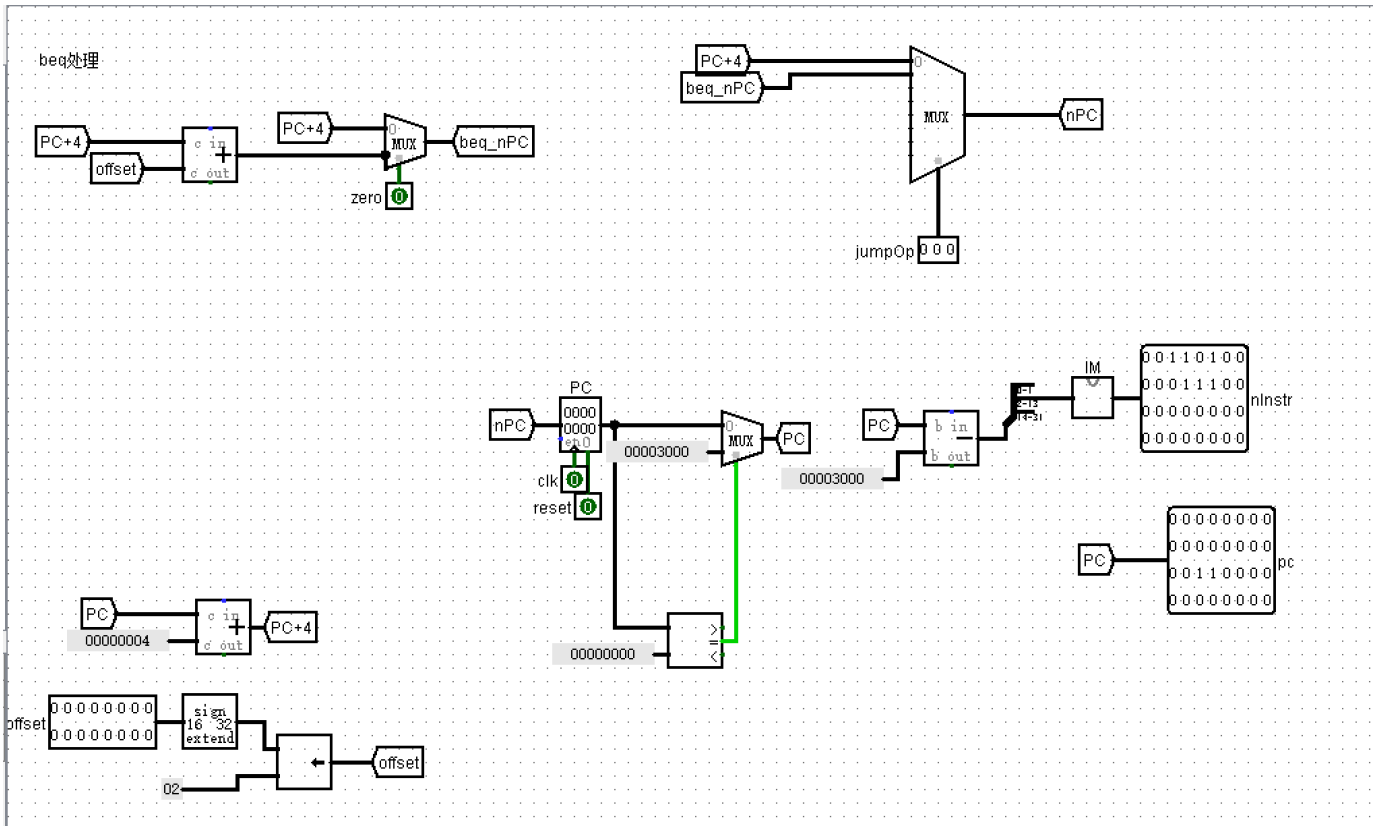
附： IFM子模块IM的端口定义：

端口名	方向	位宽	信号描述
A	I	12	指令地址：范围（以字节为单位）是0x3000 ~ 0x6fff，共4096字，但在ROM中为节省空间从0x0000处存储
nInstr	O	32	取出的32位指令

1.2.6.2 功能定义

- 异步复位
  - 当reset为1时，将PC的值置为0x3000（起始地址）
- 计算下一条指令地址（调整PC值）
  - 当beq指令有效时（zero == 1 && jumpOp == 001），PC = PC + 4 + offset * 4
  - 否则，PC = PC + 4
- 取指令
  - 根据PC的值，从IM中读出下一条指令nInstr

1.2.6.3 模块截图展示



1.2.7 Controller

按照教程的思路，分为**指令识别（和）逻辑**以及**控制信号生成（或）逻辑**两部分。

1.2.7.1 端口定义

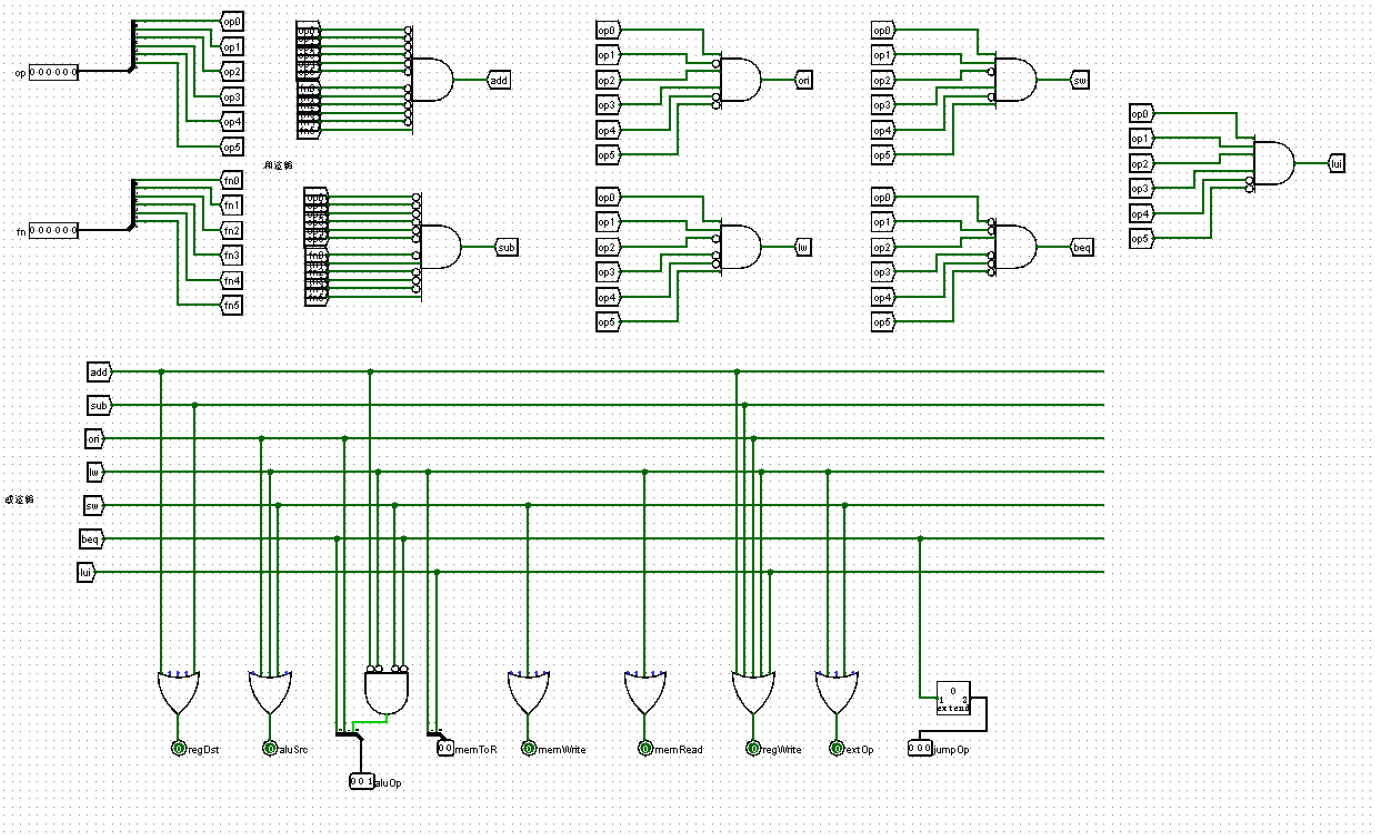
端口名	方向	位宽	信号描述
op	I	6	操作码
fn	I	6	功能码
regDst	O	1	寄存器堆写入地址选通信号：1，选用rd为地址写入；0，选用rt为地址写入
aluSrc	O	1	ALU第二个操作数（B）选通信号：1，选用立即数；0，选用寄存器堆读出的数据
aluOp	O	3	ALU操作类型：000，加；001，减；010，按位与；011，按位或；100，判断相等
memToR	O	2	数据写入寄存器堆时的数据来源：00，来自ALU运算结果；01，来自立即数；10，来自DM
memWrite	O	1	DM写使能信号：1，允许写入；0，不可写入

端口名	方向	位宽	信号描述
memRead	O	1	DM读使能信号：1，允许读出；0，不可读出
regWrite	O	1	寄存器堆写使能信号：1，允许写入；0，不可写入
extOp	O	1	扩展操作类型：1，符号扩展；0，无符号扩展。仅负责ALU两个操作数的符号扩展
jumpOp	O	3	跳转操作类型：000，无跳转；001，beq（目前只有一种，待扩展）

1.2.7.2 功能定义

- 指令识别
  - 识别指令的操作码和功能码，根据指令的不同，生成不同的控制信号
- 控制信号生成
  - 根据指令识别的结果，生成控制信号

1.2.7.3 模块截图展示



1.2.7.4 指令识别子模块

op[5:0]、fn[5:0]与七种指令（除nop外）的对应关系如下表所示：

op[5:0]	fn[5:0]	add	sub	ori	lw	sw	beq	lui
000000	100000	1	0	0	0	0	0	0
000000	100010	0	1	0	0	0	0	0
001101	x	0	0	1	0	0	0	0
100011	x	0	0	0	1	0	0	0
101011	x	0	0	0	0	1	0	0

op[5:0]	fn[5:0]	add	sub	ori	lw	sw	beq	lui
000100	x	0	0	0	0	0	1	0
001111	x	0	0	0	0	0	0	1

> 对于nop，指令码为0x00000000，op[5:0]为000000，fn[5:0]为000000，均为0，故不在表中列出。

1.2.7.5 控制信号生成子模块

所有控制信号与指令识别子模块的输出信号的对应关系如下表所示（展示的是每个信号为1时所需要的控制信号情况）：

信号名	regDst	aluSrc	aluOp[2:0]	memToR[1:0]	memWrite	memRead	regWrite	extOp	jumpOp[2:0]
add	1	0	000	00	0	0	1	x	000
sub	1	0	001	00	0	0	1	x	000
ori	0	1	011	00	0	0	1	0	000
lw	0	1	000	10	0	1	1	1	000
sw	x	1	000	x	1	0	0	1	000
beq	x	0	100	x	0	0	0	x	001
lui	0	x	x	01	0	0	1	x	000

Part 2 测试方案

2.1 自动化测试

> 只是想在此保留一下我的尝试记录。第一次实践中遇到诸多问题，功能不完善请见谅。
> 目前最主要的问题是**通过命令行让Logisim运行测试电路时无法自行停止**，或者说**在下面的**`python`**脚本中**`os.system("java -jar logisim-generic-2.7.1.jar test.circ -tty table,halt >my-text.txt")`**命令会一直处于执行状态。**
> 目前暂行的办法在终端是人为输入`ctrl+C`停止运行，得到输出的结果。

主要进行的步骤如下：

- 编写测试电路
  - 在自己已成型的cpu电路文件（`.circ`）基础上拷贝一份，加入最顶层模块`test`（命名可自定义）。将原来的`main`模块做为子模块，接入`test`模块中。具体的连接方式可以模仿**评测机提交窗口的测试电路图。**
  - 将这个修改后的电路文件（`.circ`）做为用来测试的电路，这里命名为`p3-cpu-test.circ`。
- 编写自动化测试`python`脚本
  - 参考代码如下，其中**部分框架参考了已有的python模块。具体内容由本人填充。**

```
import os
import re
import random
path=os.path.dirname(os.path.realpath(__file__))
os.chdir(path)
# 加载测试用的数据（用mips汇编语言给出，随机生成），写到test.asm文件中
with open("test.asm","w") as file:
    # 随机生成各15条ori和lui指令
    for i in range(15):
        x=random.randint(0,31)
```

```
            y=random.randint(0,31)
            num=random.randint(0,10000)
            file.write("ori $%d,$%d,%d\n"%(x,y,num))
            file.write("lui $%d,%d\n"%(x,num))
        # 随机生成各15条sw和lw指令
        for i in range(15):
            x=random.randint(0,31)
            y=random.randint(0,31)
            num=random.randint(0,100000)
            file.write("sw $%d,%d($%d)\n"%(x,num,y))
            file.write("lw $%d,%d($%d)\n"%(x,num,y))
        # 随机生成各15条add和sub指令
        for i in range(15):
            x=random.randint(0,31)
            y=random.randint(0,31)
            z=random.randint(0,31)
            file.write("add $%d,$%d,$%d\n"%(x,y,z))
            file.write("sub $%d,$%d,$%d\n"%(x,y,z))
    # 利用test.asm文件中的内容，调用Mars将其翻译成机器码，写入test.txt
    rom_name="test.txt"
    command="java -jar Mars4_5.jar test.asm nc mc CompactDataAtZero a dump .text HexText
    "+rom_name
    os.system(command)
    content=open(rom_name).read()

    # 将这些指令机器码写入测试电路IM模块的ROM中，并将写入后的电路另存为test.circ
    circmy=open("p3-cpu-test.circ",encoding='UTF-8').read()
    # addr/data: 12 32 这一部分是正则字符串匹配，请按照自己的ROM规格修改
    circmy=re.sub(r'addr/data: 12 32([\s\S]*)</a>',"addr/data: 12 32\n"+content+"
    </a>",circmy)
    with open("test.circ","w",encoding='UTF-8') as file:
            file.write(circmy)

    # 调用Logisim对test.circ电路进行测试，将测试得到的输出写到mytest.txt
    os.system("java -jar logisim-generic-2.7.1.jar test.circ -tty table,halt
    >mytest.txt")
    print("generate is successful!")
```

- 运行并比对结果
  - 在Windows命令行脚本文件目录下，运行脚本：python xxx.py。请确保目录下有 logisim-generic-2.7.1.jar和Mars4_5.jar
  - 得到自己的输出后，可以同样用其他人的标准的cpu进行测试，并将两个结果进行对比

2.2 Mars辅助测试

构造一段MIPS程序，输入Mars中和自己的cpu中运行，比对DM和GRF中的内容。
注意:

- 请在Mars导出的机器码文件第一行加入v2.0 raw后导入电路中IM的ROM
- 如果DM数据太多，可以选择将其导出进行比较

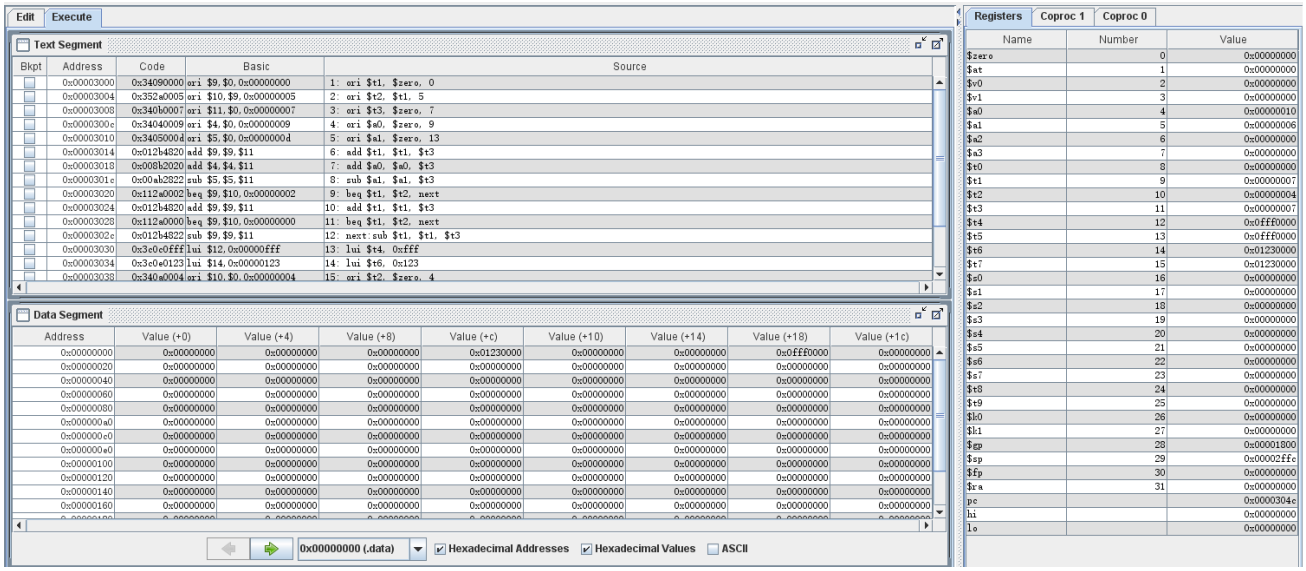
2.2.1 测试代码与结果

- 测试代码:

```
ori $t1, $zero, 0
ori $t2, $t1, 5
```

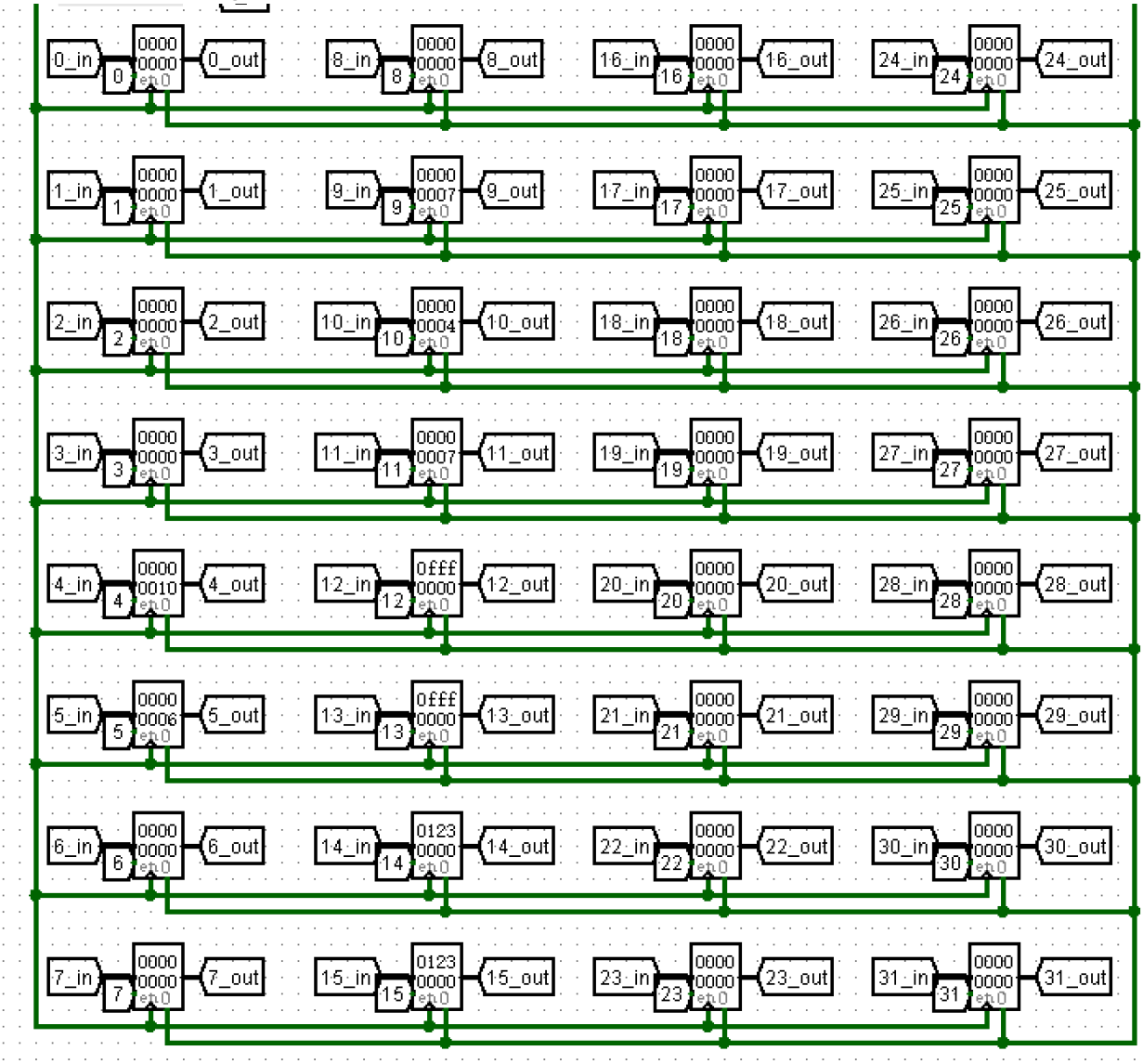
```
ori $t3, $zero, 7
ori $a0, $zero, 9
ori $a1, $zero, 13
add $t1, $t1, $t3
add $a0, $a0, $t3
sub $a1, $a1, $t3
beq $t1, $t2, next
add $t1, $t1, $t3
beq $t1, $t2, next
next:sub $t1, $t1, $t3
lui $t4, 0xfff
lui $t6, 0x123
ori $t2, $zero, 4
sw $t6, 8($t2)
sw $t4, 0x14($t2)
lw $t5, 0x14($t2)
lw $t7, 8($t2)
```

- Mars运行结果：



- CPU运行结果
  - DM中的数据（来自Logisim中直接导出）： 0 0 0 1230000 0 0 fff0000

- GRF中的数据（来自Logisim中截图）：



2.2.2 结论

经对比内容一致，本测试通过，说明CPU的基本功能实现正确。

Part 3 思考题解答

- **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 , EXT , ALU , Controller , Splitter

- **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。**

  合理。
  IM起到存储指令的作用。指令显然不需要也不能在执行过程中修改。需要只有读取功能的存储器。
  DM是起到存储数据的作用。在程序的执行过程中需要不断地向DM读取数据，需要读取和写入功能的存储器。 DM需要大量的存储空间，使用Regsiter成本过高。
  GRF为寄存器堆，选择32个寄存器去实现32个寄存器，符合逻辑。
  很抱歉，没有改进意见。

- **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。**

很抱歉，我只实现了教程要求的7个模块，**但已经能够完成要求的基本功能。**模块过多也不利于测试管理。

- **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?**

  nop是空指令，什么都不做，我们自然也可以什么都不做。全0的输入也保证了电路不会出现输出e或x的情况。

- **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。**
  Pre的测试样例为：

```
ori $a0, $0, 123
ori $a1, $a0, 456
lui $a2, 123             # 符号位为 0
lui $a3, 0xffff          # 符号位为 1
ori $a3, $a3, 0xffff     # $a3 = -1
add $s0, $a0, $a2       # 正正
add $s1, $a0, $a3       # 正负
add $s2, $a3, $a3       # 负负
ori $t0, $0, 0x0000
sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $s0, 16($t0)
sw $s1, 20($t0)
sw $s2, 24($t0)
lw $a0, 0($t0)
lw $a1, 12($t0)
sw $a0, 28($t0)
sw $a1, 32($t0)
ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1
beq $a0, $a1, loop1      # 不相等
beq $a0, $a2, loop2      # 相等
loop1:sw $a0, 36($t0)
loop2:sw $a1, 40($t0)
```

首先,基本上指令至少都出现了一次进行了测试，有一定的概率测出bug。
但观察汇编指令发现，全程并没有nop和sub指令出现；正数或者负数在临界处不够多，数据覆盖范围比较窄；同时发现检测的寄存器覆盖范围不够广；对于beq则只是检测了向后跳。跳转范围检测比较窄。