计算机组成原理　P3 实验报告

一、CPU 设计方案综述

（一）总体设计概述

本 CPU 为 Logisim 实现的单周期 MIPS32 - CPU，支持的指令集包含{addu、subu、ori、lw、sw、beq、lui、nop}。为了实现这些功能，CPU 主要包含了 PC、IM、GRF、DM、CU、cut、EXT、ALU、NPC 这些关键模块，采用模块化和层次化设计，顶层有效的驱动信号仅有内置的 clk 时钟信号和异步复位 reset 信号。

（二）关键模块定义

1. PC：程序计数器

用 logisim 内置的 32 位 register 寄存器实现，应具有**异步复位**功能，复位值为起始地址 0x00000000。

2. IM：指令存储器

使用 ROM 实现，容量为 32bit * 32。因为一条指令宽度为 32 位，故直接取 PC 提供的指令地址的 2-6 位作为 ROM 的地址。IM 输出满足 CPU 指令规范的 32 位机器码指令。

端口定义

信号名	方向	描述
PC	I	待读取指令的 32 位存储地址
R_I	O	读取的 32 位指令

3. NPC：更新程序计数器 PC 的相关逻辑

端口定义

信号名	方向	描述
PC	I	当前执行指令的 32 位地址
Br	I	3 位控制信号，根据当前指令的类型、控制取下一条指令地址的方法 000：其余使程序顺序执行的指令

		001：beq
		其他：暂时未定义
15:0imm	I	I 型指令 15-0 位中存储的 16 位立即数
		1 位控制信号，控制 beq 分支指令是否满足条件跳转
		0：条件不成立，不跳转，按照 PC+4 顺序执行指令
PCSrc	I	1：条件成立，跳转到 beq 指令立即数中立即数所指向的地址
PC+4	O	当前执行指令的下一条指令的 32 位地址
PC'	O	下一周期待执行指令的 32 位地址

4. cut：针对 32 位机器码指令的分线器

端口定义

信号名	方向	描述
Instr	I	当前执行的 32 位机器码指令
shamt	O	指令 10-6 位
25:21	O	指令 25-21 位
20:16	O	指令 20-16 位
15:11	O	指令 15-11 位
15:0imm	O	指令 15-0 位
25:0	O	指令 25-0 位

5. CU：根据 32 位机器码指令，判断指令内容，得到控制信号

端口定义

信号名	方向	描述
Instr	I	当前执行的 32 位机器码指令
		3 位控制信号，根据当前指令的类型、控制取下一条指令地址的方法
		000：其余使程序顺序执行的指令
		001：beq
Br	O	其他：暂时未定义

		2 位控制信号，选择写入寄存器地址的信号 00：20-16 位作为写入寄存器的编号 01：15-11 位作为写入寄存器的编号
A3Sel	O	其他：未定义
		2 位控制信号，选择写入寄存器的数据的信号 00：写入 ALU 计算结果 01：写入 DM 取数据的结果 10：写入下一条指令的地址 PC+4
WDSel	O	11：未定义
		1 位控制信号，GRF 的写使能信号
RegWrite	O	1：可向 GRF 中写入数据　　0：不可向 GRF 中写入数据
		1 位控制信号，控制 EXT 对 16 位立即数的位扩展方式
EXTop	O	0：零扩展　　1：符号扩展
		1 位控制信号，控制 ALU 的第二个操作数的来源 0：GRF 中 a2 指定的寄存器中的 32 位数据作为 ALU 的第二个操作数
Bsel	O	1：EXT 扩展结果的 32 位数据作为 ALU 的第二个操作数
		1 位控制信号，DM 的写使能信号
MemWrite	O	1：可向 DM 中写入数据　　0：不可向 DM 中写入数据
		4 位控制信号，控制 ALU 的计算行为 0000：ScrA + ScrB = ALULout 0001：ScrA - ScrB = ALULout 0010：ScrA & ScrB = ALULout 0011：ScrA | ScrB = ALULout
ALUControl	O	0100：ScrB << 16 = ALULout
		1 位信号，标识指令是否为分支指令 0：不是分支指令
BRANCH	O	1：是分支指令

具体实现方式

① 取指令 31-26 位的 opcode 部分；取指令 5-0 的 func 部分。

② A. 若 opcode = 0x00，则指令为 R 型指令

B．若指令为 R 型指令，且 func = 0x21，则指令为 addu；

若指令为 R 型指令，且 func = 0x23，则指令为 subu。

③ 若 opcode = 0x0d，则指令为 ori

④ 若 opcode = 0x04，则指令为 beq

⑤ 若 opcode = 0x2b，则指令为 sw

⑥ 若 opcode = 0x23，则指令为 lw

⑦ 若 opcode = 0x0f，则指令为 lui

⑧ 标志指令类型：若指令为 beq，则指令为分支指令，BRANCH = 1；

若指令为 sw，则指令为存储指令，STORE = 1；

若指令为 lw，则指令为加载指令，LOAD = 1；

⑨ 利用②~⑦得到的具体指令、和⑧得到的指令类型，通过 Priority Encoder + Pull Resistor 和 OR + NOR ，得到 32 位机器码指令对应的控制信号。

Br	000：其余使程序顺序执行的指令 001：beq 其他：暂时未定义
A3Sel	00：其他指令，写入 20-16 位 01：R 型指令，写入 15-11 位 其他：暂时未定义
WDSel	00：其他指令，写入 ALUout 01：LOAD 指令，写入 DMout 其他：暂时未定义
RegWrite	STORE、BRANCH 指令，不用写入 GRF
EXTop	STORE、LOAD 指令，使用符号扩展方法
Bsel	R 型、BRANCH 类指令，不使用扩展 imm 的 32 位数据作为 ALU 的第二个操作数
MemWrite	STORE 指令，需要写入 DM
ALUControl	0000：执行加法 0001：subu，执行减法 0010：and，执行且运算

0011：ori、or 等，执行或运算

0100：lui，执行移位运算

其他：暂时未定义

6. GRF：通用寄存器组

GRF 中包含 32 个 32 位寄存器，分别对应 0~31 号寄存器，其中 0 号寄存器读取的结果恒为 0

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：不可向 GRF 中写入数据
a1	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	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	I	5 位地址输入信号，指定 32 个寄存器中的一个，作为写入的目标寄存器
WD	I	32 位数据输入信号
RD1	O	输出 a1 指定的寄存器中的 32 位数据
RD2	O	输出 a2 指定的寄存器中的 32 位数据

模块功能定义

序号	功能名称	描述
1	复位	reset 信号有效时，所有寄存器存储的数值清零，其行为与 logisim 自带部件 register 得 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得数据到 RD1,RD2
3	写数据	当 WE 有效且时钟上升沿来临时，将 WD 写入 A3 所对应得寄存器中。

7. ALU 算数逻辑单元

提供 32 位加、减、或运算及大小比较功能。

端口定义

信号名	方向	描述
ScrA	I	32 位计算数 a
ScrB	I	32 位计算数 b
		4 位控制信号，控制 ALU 的计算行为
		0000：ScrA + ScrB = ALULout
		0001：ScrA - ScrB = ALULout
		0010：ScrA & ScrB = ALULout
		0011：ScrA | ScrB = ALULout
ALUControl	I	0100：ScrB << 16 = ALULout
shamt	I	5 位移位数值，用于移位指令
ALUout	O	32 位 ALU 计算结果
		标识 ALU 的两个计算结果是否相等
ALU=	O	0：ScrA != ScrB　　1: ScrA = ScrB

8.DM：数据存储器

使用 RAM 实现，容量为 32bit * 32，具有**异步复位**功能，复位值和起始地址都为 0x00000000。

RAM 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

信号名	方向	描述
WE	I	1 位，数据存储器的写使能信号
clk	I	0：不能写入　　1：可以写入
reset	I	异步复位信号
A	I	5 位地址输入信号，指示数据在 DM 中存储的地址
WD	I	32 位写入的数据
RD	O	32 位读出的数据

（三）重要机制实现方法

1. 跳转

NPC 模块和 ALU 模块协同工作支持指令 beq 的跳转机制。

二、测试方案

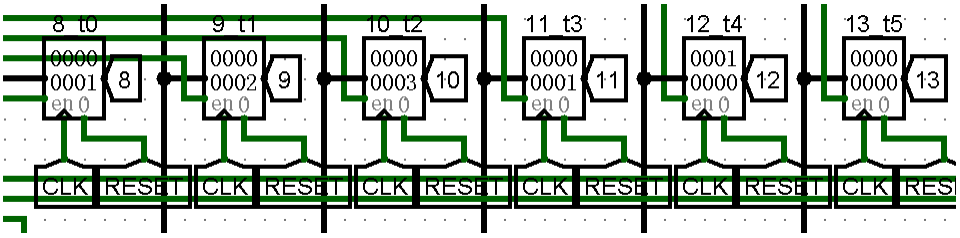
（一）典型测试样例

1. ALU 功能测试(ori，addu，subu，lui 指令)

测试数据：

```
0x0000000000000000:  35 08 00 01      ori   $t0, $t0, 1
0x0000000000000004:  35 29 00 02      ori   $t1, $t1, 2
0x0000000000000008:  01 09 50 21      addu   $t2, $t0, $t1
0x000000000000000c:  01 28 58 23      subu   $t3, $t1, $t0
0x0000000000000010:  3C 0C 00 01       lui   $t4, 1
```

测试结果：

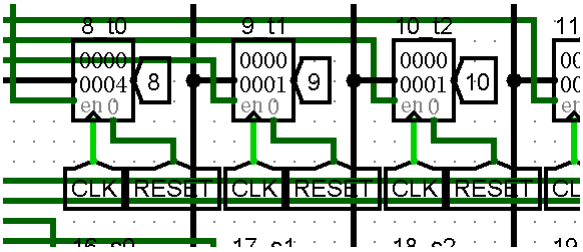


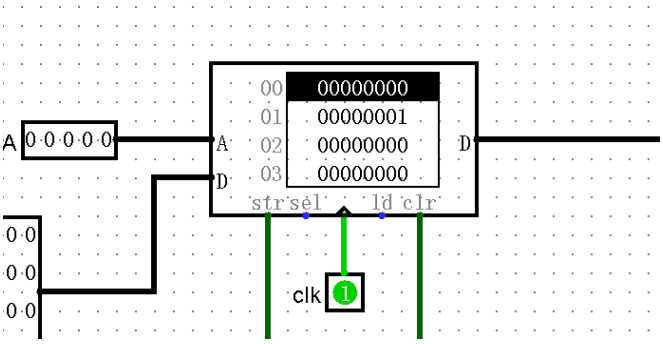
2. DM 功能测试（lw，sw 指令）

测试数据：

```
0x0000000000000000:  35 08 00 04      ori $t0, $t0, 4
0x0000000000000004:  35 29 00 01      ori $t1, $t1, 1
0x0000000000000008:  AD 09 00 00       sw   $t1, ($t0)
0x000000000000000c:  8D 0A 00 00       lw   $t2, ($t0)
```

测试结果：



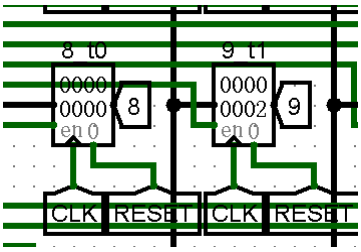


3. 跳转指令测试（beq 指令）

测试数据：

0x0000000000000000:	11 09 00 01	beq $t0, $t1, 8
0x0000000000000004:	35 08 00 01	ori $t0, $t0, 1
0x0000000000000008:	35 29 00 02	ori $t1, $t1, 2

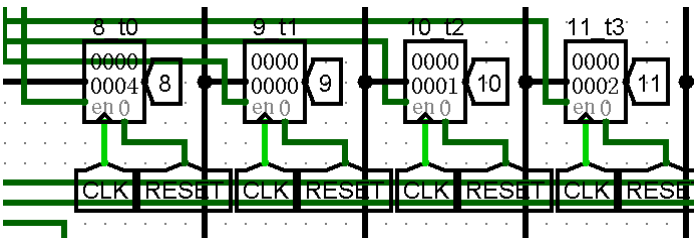
测试结果：



测试数据：

0x0000000000000000:	35 08 00 04	ori $t0, $t0, 4
0x0000000000000004:	11 09 00 01	beq $t0, $t1, 8
0x0000000000000008:	35 4a 00 01	ori $t2, $t2, 1
0x000000000000000c:	35 6b 00 02	ori $t3, $t3, 2

测试结果：



三、思考题

1. **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。**

答案：合理。

IM 使用 ROM：电路可以访问 ROM 中的当前值，但是不能更改他们；而用户可以通过 Poke 工具或者菜单工具改变 ROM 里面的值；ROM 可以设置数据位宽为 32 位（与指令等长）。ROM 的特性与实际 CPU 中 IM 的行为相符合，不可更改性同时提高了电路的安全性。（改进意见：可以使用多个并行的 ROM 模块，以增加可存储指令的数量）

DM 使用 RAM：RAM 可以设置数据位宽为 32 位，与 MIPS-CPU 的数据长度等长。RAM 自带的 clk 上升沿+存储使能端口，使 RAM 写入行为与实际 CPU 中的 DM 行为一致。RAM 自带的读取地址输入端口+读取数据输出端口，使 RAM 读取行为与实际 CPU 中的 DM 行为一致。（改进意见：可以使用多个并行的 RAM 模块，以增加可存储数据的容量）

GRF 使用 register：32 个 register 均可以设置数据位宽为 32 位，与 MIPS-CPU 的数据长度等长。32 个 register 自带的 clk 上升沿+使能端口，使 register 写入行为与实际 CPU 中的 GRF 行为一致。为板块化的 32 个 register 设置读取地址输入端口+读取数据输出端口，使 register 读取行为与实际 CPU 中的 GRF 行为一致。（改进意见：参照 MIPS-CPU 复位和初始值的实际情况，为 32 个 register 设定不同的初值，而不是全部设置为 0）

2. **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？请给出你的理由**

答案：nop 空指令的 opcode 和 funct 部分，在控制器的“AND”和逻辑中不能匹配任意一条 CPU 中的指令，从而不能影响“OR”或逻辑的结果。同时，尽管执行 nop 指令时有数据在 CPU 结构中流动，但是在控制信号无效的情况下，这些数据不能写入 DM 和 GRF，是无效的数据

3. **上文提到，MARS 不能导出 PC 与 DM 起始地址均为 0 的机器码。实际上，可以通过为 DM 增添片选信号，来避免手工修改的麻烦，请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法**

答案：在 PC 输出前判断，若指令地址大于 0x3000，则输出 pc-0x3000 的值，实现地址的平移。保险起见，还需要将指令存储器的存储范围扩宽，这样可以存储更多的指令，避免执行 n 条指令加（32-n）条指令后进入与 MARS 行为不符的循环，这样的循环与 0x3000 的平移映射一起，会给 CPU 带来错误。

4. **除了编写程序进行测试外，还有一种验证 CPU 设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)”了解相关内容后，简要阐述相比于测试，形式验证的优劣之处。**

答案：形式验证的主要思想是通过使用数学证明的方式来验证一个设计的功能是否正确。形式验证使用严格的数学推理来证明待测试设计的正确性，由于其静态、数学的特性，避免了对所有可能测试向量的枚举，而且能够达到 100%无死角的检测。

模拟仿真	形式验证
使用编写的“样例”“测试向量”	使用数学推理
对整个模型的实现正确性进行验证,	对模型的小部分进行正确性验证 在不缺漏分支的情况下、分布验证每个模块的正确性，可以确定整个模型的正确性； 但是问题就是在于难以确定检验的分支（或者问题的属性是否完备）
只能排除错误，不能证明整个模型的正确性	