计算机组成原理实验报告

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 MIPS - CPU，支持的指令集包含{addu、subu、ori、lw、sw、beq、lui、sll、srl、sra、slt、j、jal、jr、jalr、nop}。为了实现这些功能，CPU 主要包含了 PC、NPC、IM、GRF、ALU、DM、EXT……，这些模块按照自顶向下的顶层设计逐级展开。

（二）关键模块定义

1. GRF

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。具体模块端口定义如下：

信号名	方向	描述
CLK	I	时钟信号
RST	I	异步复位信号
WE3	I	写使能信号
A1	I	读地址
A2	I	读地址
A3	I	写地址
WD	I	32 位写入数据
RD1	O	32 位读出数据
RD2	O	32 位读出数据

2. DM

信号名	方向	描述
CLK	I	时钟信号

A	I	读地址
WD	I	待写入 DM 的 32 位数据信号
WE	I	写数据使能信号
RST	I	异步复位信号
RD	O	DM 中 A 地址读出的数据

3. ALU

ALUControl[3:0]	功能	ALUControl[3:0]	功能
0000	A AND B	1000	\
0001	A OR B	1001	\
0010	A + B	1010	\
0011	A - B	1011	\
0100	A 逻辑左移 B	1100	\
0101	A 逻辑右移 B	1101	\
0110	A 算数右移 B	1110	\
0111	SLT	1111	\

4. Controller

控制器 Controller 端口声明如下：

控制器			
端口声明	方向	位宽	描述
Opcode	I	6	指令 Instr[31:26]
Funct	I	6	指令 Instrc[5:0]
REGorMEM	O	1	grf 回写控制信号之一： 0：ALUResult 回写； 1：DM 读出的值回写
MemWrite	O	1	DM 写使能信号： 0：不能写入；

			1：可以写入
NPCOp	O	3	NPC 计算方式选择信号： 000：PC = PC + 4; 001：PC = PC + 4 + ALUZero && Sign_ext(16imm || 00) (beq) 010：PC = PC[31:28] || 26imm || 00 (j、jal) 011：PC = GPR[rs] (jr $ra，注：$ra 不一定得是$31) 100：暂未定义 101：暂未定义 110：暂未定义 111：暂未定义
ZeroEXT	O	1	EXT 对 16 位立即数的扩展方式： 0：符号扩展 1：零扩展
A1op	O	2	GRF 中 A1 端口选择信号： 00：读取 rs（Instr[25:21]） 01：读取 rt（Instr[20:16]） 10：暂未定义 11：暂未定义
A2op	O	1	GRF 中 A2 端口选择信号： 0：读取 rt（Instr[20:16]） 1：暂未定义
A3op	O	2	GRF 中 A3 端口选择信号： 00：读取 rd（Instr[15:11]） 01：读取 rt（Instr[20:16]） 10：读取常数 31（即$r31 址） 11：暂未定义
REGop	O	2	GRF 写入数据端口选择信号： 00：回写 REGorMEM 选择后的信号

			01：回写 16 位立即数加载至高位后的 32 位数据 10：回写当前 PC 加 4 的值 11：暂未定义
RegWrite	O	1	GRF 写使能信号： 0：不可写 1：可写
ALU_Aop	O	1	ALU SrcA 端口选择信号： 0：读取 RD1 1：暂未定义
ALU_Bop	O	2	ALU SrcB 端口选择信号： 00：读取 RD2 01：读取 16 位立即数（根据 ALUZero 选择）扩展后的 32 位数据 10：读取指令 shamt 片段 0 扩展后的结果 11：暂未定义
ALUOp	O	4	ALU 功能选择信号： 0000：and 0001：or 0010：add（不考虑溢出） 0011：sub（不考虑溢出） 0100：sll 0101：srl 0110：sra 0111：slt（注：有符号!） 1000—1111：暂未定义

控制器真值表如下：（本电路实现中 X 全置 0）

指令	opcode	funct	REGor MEM	Mem Write	NPC op	Zero EXT	A1 op	A2 op	A3 op	REG op	RegW rite	ALU_ Aop	ALU_ Bop	ALU Op
addu	000000	100001	0	0	000	0	00	0	00	00	1	0	00	0010
subu	000000	100011	0	0	000	0	00	0	00	00	1	0	00	0011
ori	001101		0	0	000	1	00	0	01	00	1	0	01	0001
lw	100011		1	0	000	0	00	0	01	00	1	0	01	0010
sw	101011		0	1	000	0	00	0	00	00	0	0	01	0010
beq	000100		0	0	001	0	00	0	00	00	0	0	00	0000
lui	001111		0	0	000	0	00	0	01	01	1	0	00	0000
sll	000000	000000	0	0	000	0	01	0	00	00	1	0	10	0100
srl	000000	000010	0	0	000	0	01	0	00	00	1	0	10	0101
sra	000000	000011	0	0	000	0	01	0	00	00	1	0	10	0110
slt	000000	101010	0	0	000	0	00	0	00	00	1	0	00	0111
j	000010		0	0	010	0	00	0	00	00	0	0	00	0000
jal	000011		0	0	010	0	00	0	10	10	1	0	00	0000
jalr	000000	001001	0	0	011	0	00	0	00	10	1	0	00	0000
jr	000000	001000	0	0	011	0	00	0	00	00	0	0	00	0000

（三）重要机制实现方法

1. 跳转

该 CPU 涉及的跳转指令有：beq、j、jal、jr、jalr。

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				

对于 beq 指令，控制器解析 beq 指令，输出的 NPCOp = 001，同时 ALU 判断两个操作数 GPR[rs]、GPR[rt]是否相等，若相等将 ALUZero 置一。在 NPC 内，当 ALUZero = 1 并且 NPCOp = 001 时，才执行跳转，即 PC = PC + 4 + ALUZero && Sign_ext(16imm || 00)，若不满足该条件，则只执行 PC = PC + 4。

J：跳转

编码	31 26	25	0
	j 000010	instr_index	
	6	26	
格式	j target		
描述	j 指令是 PC 相关的转移指令。当把 4GB 划分为 16 个 256MB 区域，j 指令可以在当前 PC 所在的 256MB 区域内任意跳转。		
操作	PC ← PC31..28‖instr_index‖0^2		
示例	j Loop_End		
其他	如果需要跳转范围超出了当前 PC 所在的 256MB 区域内时，可以使用 JR 指令。		

对于 j 指令，控制器解析 j 指令，将 NPCOp 置为 010，同时 NPC 读入 26 位立即数，PC = {PC[31:28]，26imm，2{0}}（verilog 位拼接运算符），从而实现跳转。

JAL：跳转并链接

编码	31 26 25 0	
	jal 000011	instr_index
	6	26
格式	jal target	
描述	jal 指令是函数指令，PC 转向被调用函数，同时将当前 PC+4 保存在 GPR[31]中。当把 4GB 划分为 16 个 256MB 区域，jal 指令可以在当前 PC 所在的 256MB 区域内任意跳转。	
操作	PC ← PC31..28||instr_index||0^2 GPR[31] ← PC + 4	
示例	jal my_function_name	
其他	jal 与 jr 配套使用。jal 用于调用函数，jr 用于函数返回。当所调用的函数地址超出了当前 PC 所在的 256MB 区域内时，可以使用 jalr 指令。	

对于 jal 指令，其 NPC 相关控制信号与上述 j 指令相同。不同的是，对于 jal 指令，还需要通过 REGop 信号选择 10，将 PC + 4 的值存入$31 寄存器，为此，在 grf 写数据端口 WD3 放置了一个 4 选 1 多路选择器，其第三个选择端口为当前 PC 加 4 的值。

JR：跳转至寄存器

编码	31 26	25 21	20 11	10 6	5 0
	special 000000	rs	0 00 0000 0000	0 00000	jr 001000
	6	5	10	5	6
格式	jr rs				
描述	PC ← GPR[rs]				
操作	PC ← GPR[rs]				
示例	jr $31				
其他	jr 与 jal/jalr 配套使用。jal/jalr 用于调用函数，jr 用于函数返回。				

对于 jr 指令，其将 GPR[rs]寄存器存的 32 位值存入 PC，在数据通路上只需要在 NPC 的输入端口增加一个读入寄存器（32 位数据）的端口，同时设置 NPCOp = 011 时，将 pc 设置为输入的 32 位数据的值。

JALR：跳转并链接

编码	31 26	25 21	20 16	15 11	10 6	5 0
	special 000000	rs	0 00000	rd	0 00000	jalr 001001
	6	5	5	5	5	6
格式	`jalr rd, rs`					
描述	jalr 指令是函数指令，PC 转向被调用函数(函数入口地址保存在 GPR[rs]中)，同时将当前 PC+4 保存在 GPR[rd]中。					
操作	`PC ← GPR[rs]` `GPR[rd] ← PC + 4`					
示例	`jalr $s1, $31`					

对于 jalr 指令，其涉及的 PC 操作与 jr 相同，都是将 GPR[rs]存入 PC。同时，它与 jal 指令相似，都需要将当前 pc 加 4 的值存入寄存器。不同的是，jal 将 pc + 4 存入$31 寄存器，故 A3op 选择 10；而 jalr 将 pc + 4 存入$rd 寄存器，故 A3op 选择 00。

二、测试方案

（一）典型测试样例

尽管要求的 PC 从 0x0000 开始，与 Mars 中从 0x3000 开始不同，但由于指令存储器地址仅取 5 位，所以跳转指令的功能不受影响（但将当前 pc 值存入寄存器时，可能会比要求值多 0x3000）。故直接用 mars 中导出的 32 位 MIPS 机器码进行测试。

下面为对 cpu 中涉及的所有指令进行的一次相对全覆盖的测试：

（注意：在下面第 41 到 43 行对跳转指令 jalr 的测试中，我将$30 寄存器赋值为 0x3004 作为跳转的地址而非 0x0004，否则会导致其在 logisim 中能正常执行跳转并在第三行通过 jr $28 返回，而在 Mars 中会出现溢出错误。）

```
    addu $30, $30, $0
    beq $0, $30, start
    jr $28
start:
    lui $0, 0xffff
    ori $0, $0, 0xffff

    lui $1, 0xffff
    lui $2, 0xfff
    lui $3, 0xff
    ori $1, $1, 0xffff  # $1 = -1
    ori $2, $2, 0xfff
    ori $3, $3, 0xff
    ori $4, $0, 1
    ori $5, $0, 2
    ori $28, $0, 0x10

    nop  # see PC

    addu $6, $1, $4  # $6 = 0
    addu $7, $1, $5  # $7 = 1
    sll $7, $7, 2  # $7 = 4

    subu $8, $4, $1  # $8 = 2
    subu $9, $5, $4  # $9 = 1

    beq $6, $7, start

loop:  # $6 = 0
    beq $6, $28, end_loop
    sw $1, 0($6)
    sw $2, 4($6)
    addu $6, $6, $7  # $6 += 4
    addu $1, $1, $6  # $1 += 4
    addu $2, $2, $6  # $2 += 4
    j loop
end_loop:
    jal load_words
```

```
39
40      ori $30, $0, 0
41  test_jalr:
42      ori $30, $0, 0x3004
43      jalr $28, $30  # $28 = pc + 4; pc = $30
44
45      beq $0, $0, final_test
46
47
48  load_words:
49      lw $1, 0($0)
50      lw $2, 4($0)
51      lw $3, 8($0)
52      lw $4, 12($0)
53      lw $5, 16($0)
54      lw $6, 20($0)
55      lw $7, 24($0)
56      lw $8, 28($0)
57      jr $ra
58
59
60  final_test:
61      lui $1, 0xffff
62      ori $1, 0xf
63      lui $2, 0xffff
64      ori $2, $2, 213
65      lui $3, 0x6
66      ori $3, $3, 0xff32
67      lui $28, 0x7
68
69      sll $4, $1, 8
70      srl $5, $4, 4
71      sra $6, $4, 4
72      slt $7, $1, $3
73      slt $8, $3, $28
74      lui $28, 0x5
75      slt $7, $3, $28
76
```

三、思考题

（一）题目描述：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。

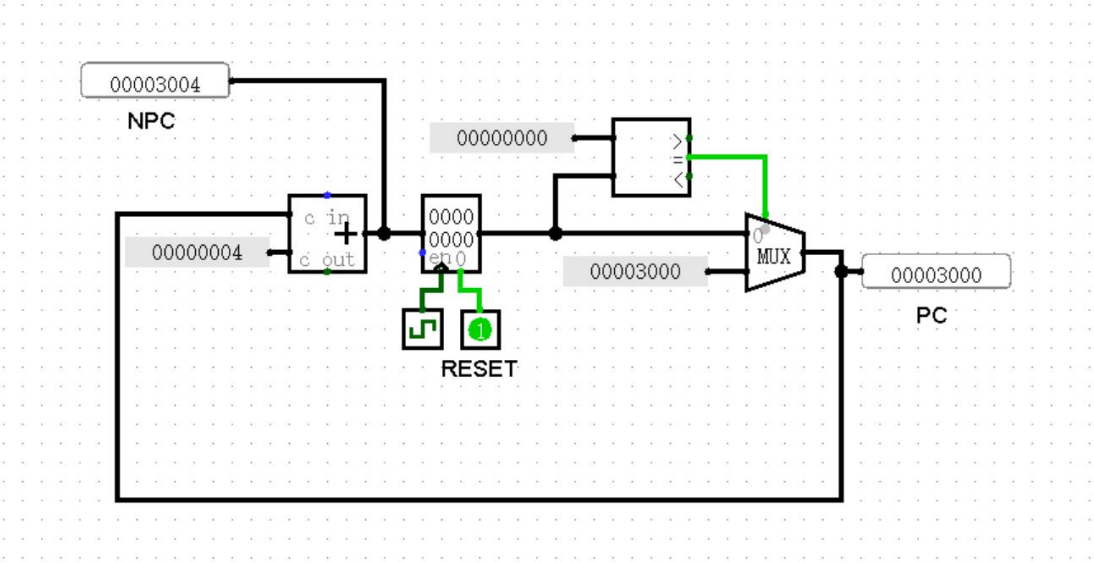
合理。首先，IM、DM、GRF 都具有储存数据的功能。其中，IM 为指令存储器，一般情况下不需要写入，只需要读出，故选用 ROM；DM 为数据存储器，既要支持读功能，也要支持写功能，故选用随机访问存储器 RAM；而 GRF 为通用寄存器堆，显然要采用 Register。

（二）题目描述：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。

因为空指令不执行任何操作，只进行 PC 自增。其对应 16 进制机器码为 0x00000000。而本 cpu 中支持 sll 指令。Sll 为 R 型指令，其 opcode 和 funct 字段均为 000000，故 sll $0, $0, 0 这条汇编指令翻译成机器码后正是 0x00000000，而这条指令的实际效果只有 PC + 4。故不需要将 nop 指令加入控制信号真值表。

（三）题目描述：上文提到，MARS 不能导出 PC 与 DM 起始地址均为 00 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。

在测试时将 logisim 异步复位到 0x3000，保持与 Mars 一致即可，如下图（这里的 NPC 仅涉及 NPC = PC + 4 这种情况）：



（四）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。

根据维基百科的定义：在计算机硬件（特别是集成电路）和软件系统的设计过程中，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。

其实，形式验证就是通过数学的方法严格证明我们的 CPU 没有 bug。相较于编写程序进行测试，其可以做到 100%的清除错误，但是其相当耗费时间和精力，几乎难以完成。