计算机组成原理P3实验报告

一、设计草稿

设计草稿部分主要分为模块化处理、数据通路设计以及控制器设计三个部分。本单周期cpu支持的指令集为{add,sub,sll,lw,lb,lui,ori,j,jal,jr}。

针对不同类型的指令，数据通路设计会有略微的变化，但R类型、I类型指令以及J类型指令分别有以下特征：

R类型

IFU读指令 -> Controller译码 -> 从GRF中取出需要的数据 -> a.送将GRF取出的数据输入至ALU　b.将GRF数据输入至NPC中**(jr)** -> 将ALU结果写入GRF中 -> PC + 4/NPC

I类型

IFU读指令 -> Controller译码 -> a.从GRF中取出需要的数据　b.对立即数进行对应的扩展操作 ->通过ALU计算出DM地址或者是否为0　->a. 将数据写入DM **（存储类指令）**b.从DM读数据 **（取数类指令）**　c.运算next_PC并跳转(条件跳转) -> （写结果入GRF） **（取数类指令）** -> PC + 4**（存取类指令）**

J类型

IFU读指令 -> Controller译码 -> 将PC+4存入GRF中**(JAL指令)**->计算NPC并跳转

无论是对于课下实现的指令，或者是课上要实现的指令，均可通过先判断指令类型，再按照RTL设计数据通路的方式进行实现。

（1）模块化处理

本CPU主要分为IFU、NPC、GRF、ALU、DM、EXT共6个部分（Controller会在控制器设计部分中单独分析），本节讲详细介绍各关键模块的信号端口及具体设计细节。

1.IFU

输入输出端口设计：

信号名	方向	描述
CLK	I	时钟信号
Reset	I	异步复位信号，将PC寄存器清零。1：复位；0：无效。
next_PC[31:0]	I	32位输入信号，下一条指令的地址
Instr[31:0]	O	32位输出信号，当前指令的机器码
PC[31:0]	O	32位输出信号，当前指令的地址

具体设计：

说明：

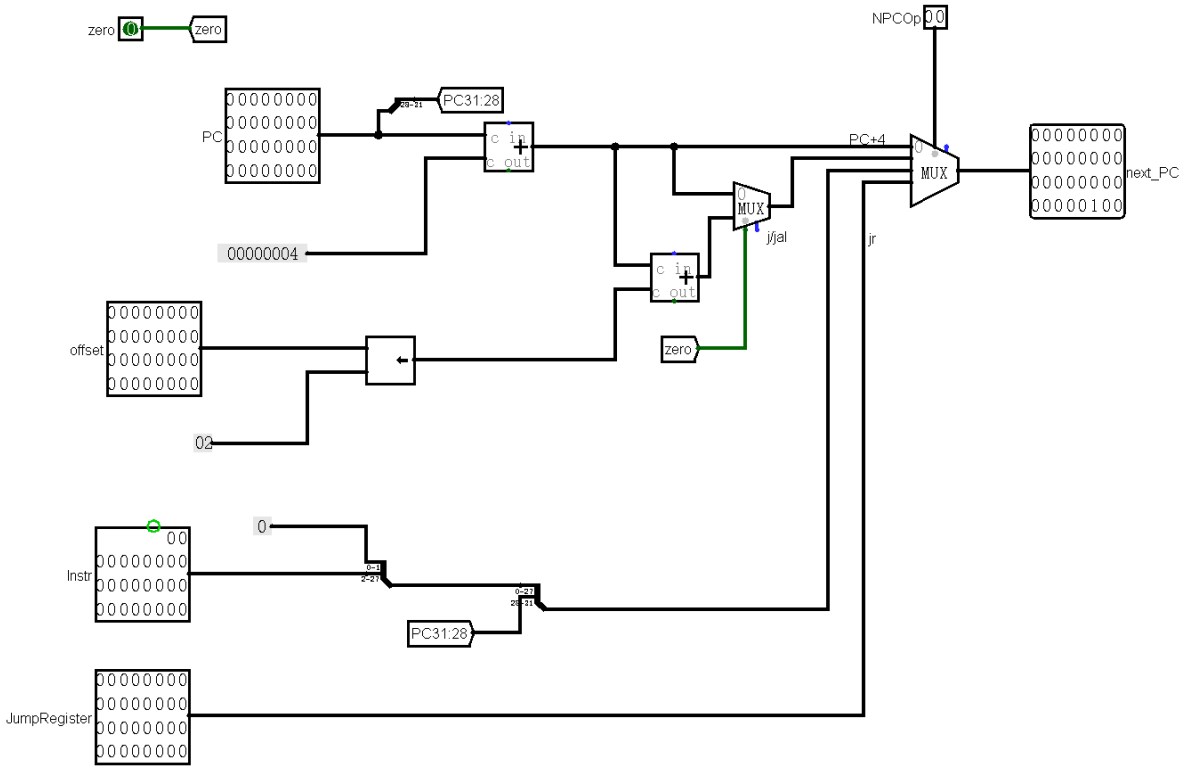
从PC寄存器取出32位数据后，由于ROM大小为32字，故A端口需要5位输入。又由于ROM采用字寻址的方式输入，故需要将PC寄存器输出数据的右移两位后再进行输入（即输入第2-6位）。

2.NPC

输入输出端口设计：

信号名	方向	描述
PC [31:0]	I	32位输入信号，当前指令的地址
Offset [31:0]	I	32位输入信号，代表地址偏移量
Instr [25:0]	I	26位输入信号，代表j/jal指令的26位地址索引
RegData [31:0]	I	32位输入信号，代表jr指令目标寄存器的值
NPCOp[1:0]	I	下一个PC的选择信号
Zero	I	表示ALU_Result是否为0，用于执行beq操作
next_PC [31:0]	O	32位输出信号，下一个PC的地址

具体设计：

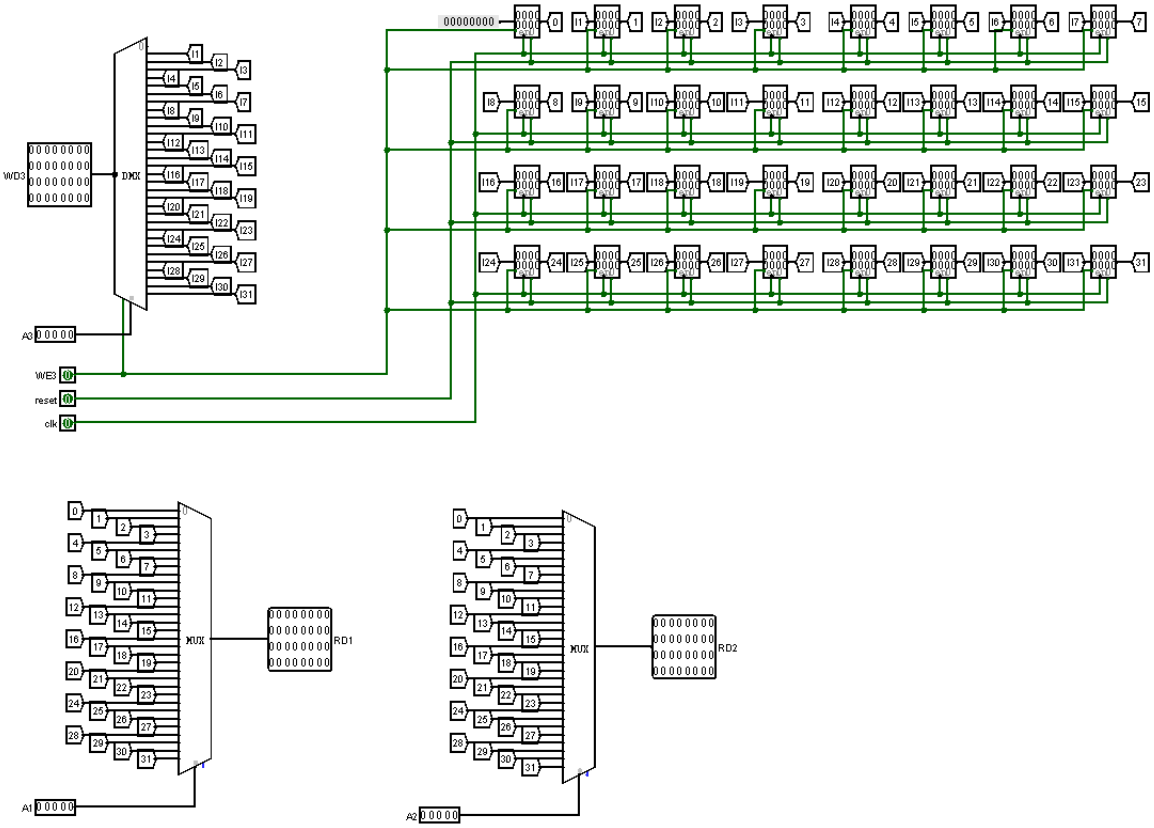


3.GRF

输入输出端口设计：

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号，将32个寄存器的值全部清零
WE3	I	写使能信号，高电平时向A3寄存器写入数据WD3；低电平时无效
A1 [4:0]	I	5位输入信号，选择32个寄存器中的一个并将其数据输出至RD1端口
A2 [4:0]	I	5位输入信号，选择32个寄存器中的一个并将其数据输出至RD2端口
A3 [4:0]	I	5位输入信号，选择32个寄存器中的一个作为数据写入的目标寄存器
WD3 [31:0]	I	32位输入数据，当WE3有效时向GRF中写入数据
RD1 [31:0]	O	32位输出信号，输出A1中选择的寄存器中的数据
RD2 [31:0]	O	32位输出信号，输出A2中选择的寄存器中的数据

具体设计：



说明：

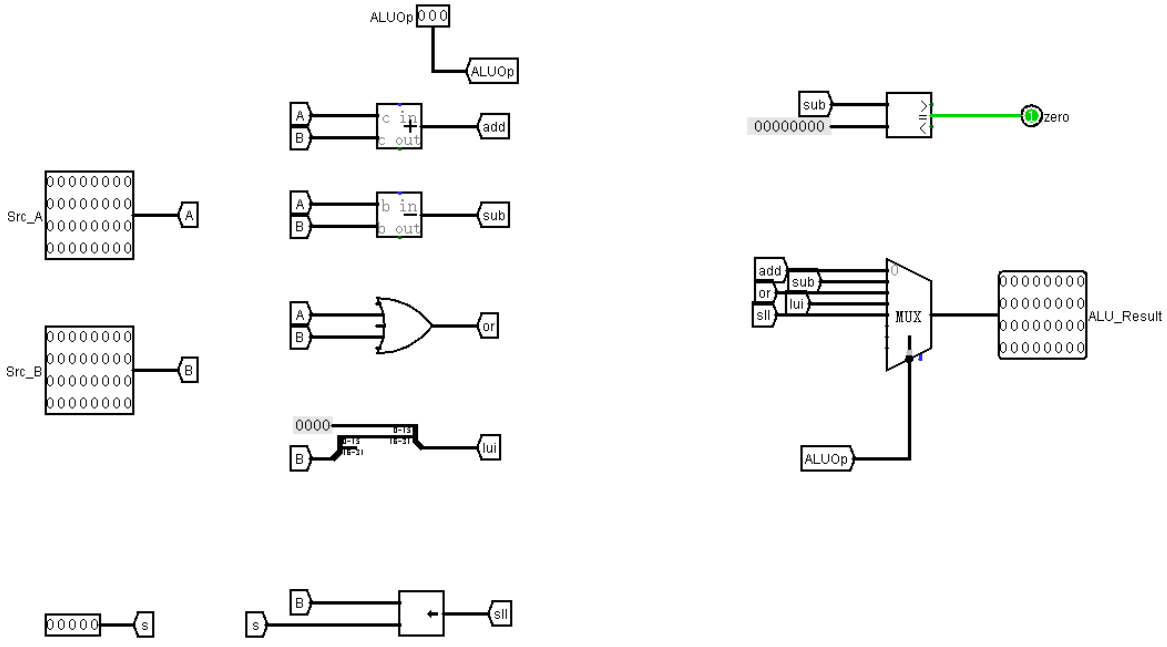
DMX器件部件将“Three_state?”选项勾选为“Yes”，保证之前输入至寄存器的数据不被抹除。同时，$0寄存器应当始终保持值为0，故DMX部件处无法对$0寄存器写入数据。

4.ALU

输入输出端口设计：

信号名	方向	描述
Src_A [31:0]	I	32位输入信号，待ALU处理的数据之一
Src_B [31:0]	I	32位输入信号，待ALU处理的数据之一
Shamt[4:0]	I	5位输入信号，代表数据的偏移量
ALUOp [2:0]	I	3位输入选择信号，选择ALU需要进行的操作
Zero	O	输出信号；高电平代表ALU_Result为0，低电平代表其不为0

具体设计：



说明：

当ALUOp为00时，ALU执行加运算操作；当ALUOp为01时，ALU执行减运算操作；当ALUOp为10时，ALU执行或运算操作；当ALUOp为11时，ALU执行lui操作。

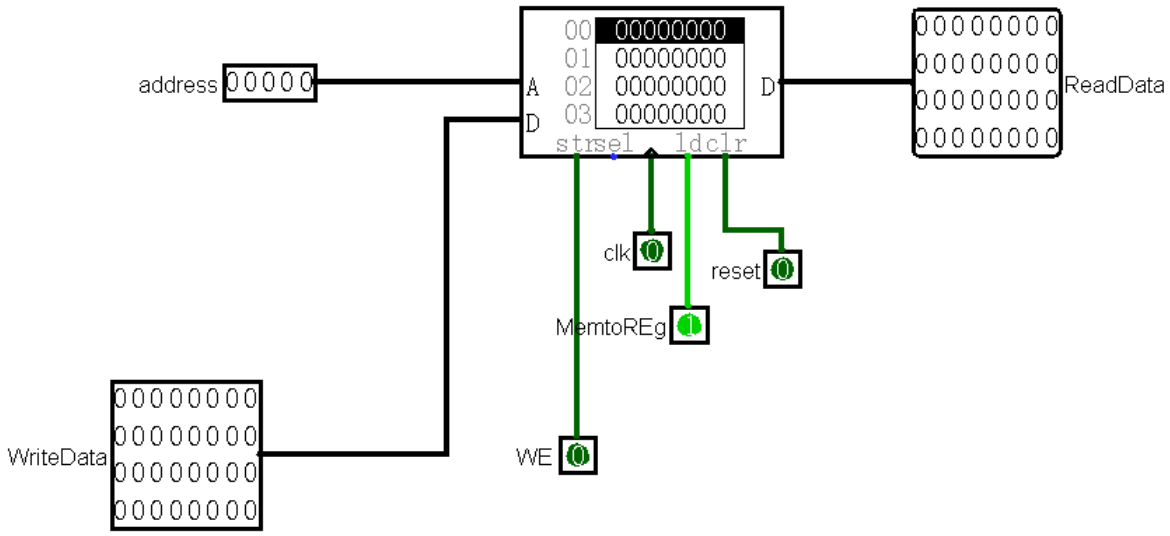
5.DM

输入输出端口设计：

信号名	方向	描述

信号名	方向	描述
clk	I	时钟信号
reset	I	异步复位信号；高电平时将RAM复位，低电平时无效
WE	I	写使能信号，高电平时将WD中数据写入DM中，低电平时无效
MemtoReg	I	读使能信号，高电平时将DM中的数据输出，低电平时无效
A [4:0]	I	5位输入信号，选择RAM中一个地址单元进行读写操作
WD [31:0]	I	32位输入信号，待输入至DM中的数据
D [31:0]	O	32位输出信号，从DM中读出的数据

具体设计：



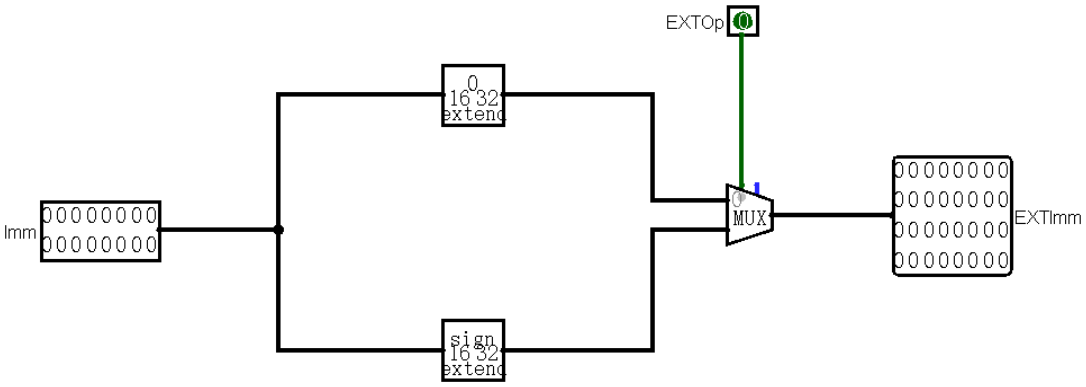
说明：此处引入的五位地址信号同IFU中操作，需要引入32位数据的第2-6位。

6.EXT

输入输出端口设计：

信号名	方向	描述
Imm [15:0]	I	16位输入信号，表示待进行拓展的立即数
EXTOp	I	EXT选择信号，1：进行符号拓展至32位；0：进行零拓展至32位
EXTImm [31:0]	O	32位输出信号，表示拓展后的立即数

具体设计：



（2）数据通路设计

针对不同指令的数据通路进行分析后，得到如下结果：

指令	IFU	NPC.PC	NPC.Offset	NPC.Instr	NPC.RegData	NPC.Zero	GRF.A1	GRF.A2	GRF.A3	GRF.WD3	ALU.Src_A	ALU.Src_B	DM.A	Dm.WD	EXT.Imm
add	next_PC	PC					Rs/Base	Rt	Rd	ALU.Result	GRF.RD1	GRF.RD2			
sub	next_PC	PC					Rs/Base	Rt	Rd	ALU.Result	GRF.RD1	GRF.RD2			
lw	next_PC	PC					Rs/Base		Rt	DM.RD	GRF.RD1	EXT	ALU.Result		imm16
sw	next_PC	PC					Rs/Base	Rt			GRF.RD1	EXT	ALU.Result	GRF.RD2	imm16
beq	next_PC	PC	EXT			ALU.Zero	Rs/Base	Rt			GRF.RD1	GRF.RD2			imm16
ori	next_PC	PC					Rs/Base		Rt	ALU.Result	GRF.RD1	EXT			imm16
lui	next_PC	PC					Rs/Base		Rt	ALU.Result		EXT			imm16
nop	next_PC	PC													
j	next_PC			IFU.Instr											
jal	next_PC			IFU.Instr					0x1f	IFU.PC+4					
jr	next_PC				GRF.RD1		Rs/Base								
sll	next_PC	PC						Rt	Rd	ALU.Result		GRF.RD2			
整体	next_PC	PC	EXT	PC.Instr	GRF.RD1		Rs/Base	Rt	Rd|Rt|0x1f	ALU.Result|DM.RD|IFU.PC+4	GRF.RD1	GRF.RD2|EXT	ALU.Result	GRF.RD2	imm16

说明：由表格可知，在GRF.A3(GRF写入寄存器)、GRF.WD3(GRF写入数据)以及在ALU.Src_B三列中均有不同的输入来源，故在连接电路时需要三个MUX部件实现数据的导通。

（3）控制器设计

控制信号设计分析：

针对如上数据通路进行分析，我们可知针对三个MUX部件，ALU.Src_B需要1位选择信号ALUSrc，GRF.WD3需要2位选择信号MemtoReg

GRF.A3需要2位选择信号RegDst。同时，控制信号仍应包括各模块的写入/读出信号（如MemWrite）、选择信号（如ALUOp）以及特别地，beq指令的Branch信号。

首先列出控制信号真值表：

指令	add	sub	sll	lw	sw	ori	lui	beq	j	jal	jr
MemWrite	0	0	0	0	1	0	0	0	0	0	0
WDSel1	0	0	0	0	x	0	0	x	x	1	x
WDSel0	0	0	0	1	x	0	0	x	x	0	x
NPCOp1	0	0	0	0	0	0	0	0	1	1	1
NPCOp0	0	0	0	0	0	0	0	1	0	0	1

指令	add	sub	sll	lw	sw	ori	lui	beq	j	jal	jr
ALUOP2	0	0	1	0	0	0	0	0	x	x	x
ALUOP1	0	0	0	0	0	1	1	0	x	x	x
ALUOP0	0	1	0	0	0	0	1	1	x	x	x
ALUSrc	0	0	0	1	1	1	1	0	x	x	x
RegDst1	0	0	0	0	x	0	0	x	x	1	x
RegDst0	1	1	1	0	x	0	0	x	x	0	x
RegWrite	1	1	1	1	0	1	1	0	0	1	0
EXTOp	x	x	x	1	1	0	0	1	x	x	x

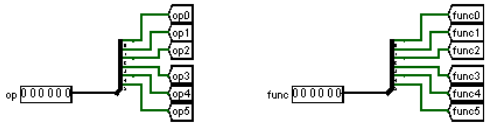
基于上述分析，给出控制器中输入输出端口设计：

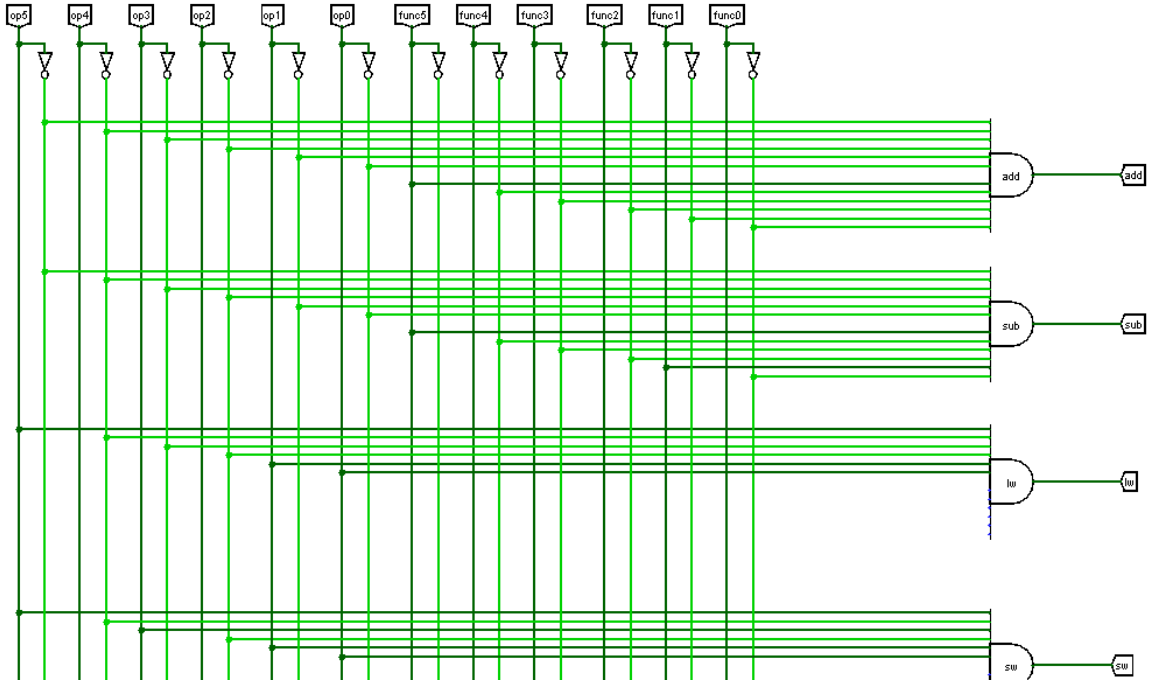
信号名	方向	描述
Op [5:0]	I	6位输入信号，表示指令的基本操作
Funct [5:0]	I	6位输入信号，R类型指令的功能码
MemWrite	O	高电平时，将数据写入DM；低电平时无效
NPCOp[2:0]	O	000:PC执行+4操作 001：beq条件被触发，需要进一步判断Zero 010:执行j/jal 011:执行jr操作
ALUOp [2:0]	O	000：加运算 001：减运算 010：或运算 011：逻辑左移16位 100：逻辑左移运算
ALUSrc	O	0：选择RD2作为输入源 1：选择EXTImm作为输入源
RegDst[1:0]	O	00：选择Rt作为输入源 01:选择Rd作为输入源 10:选择0x1f($31)作为输入源
RegWrite	O	高电平时，将数据写入GRF中；低电平时无效
EXTOp	O	0：对立即数进行零拓展 1：对立即数进行符号拓展

具体设计（不完全展示）：

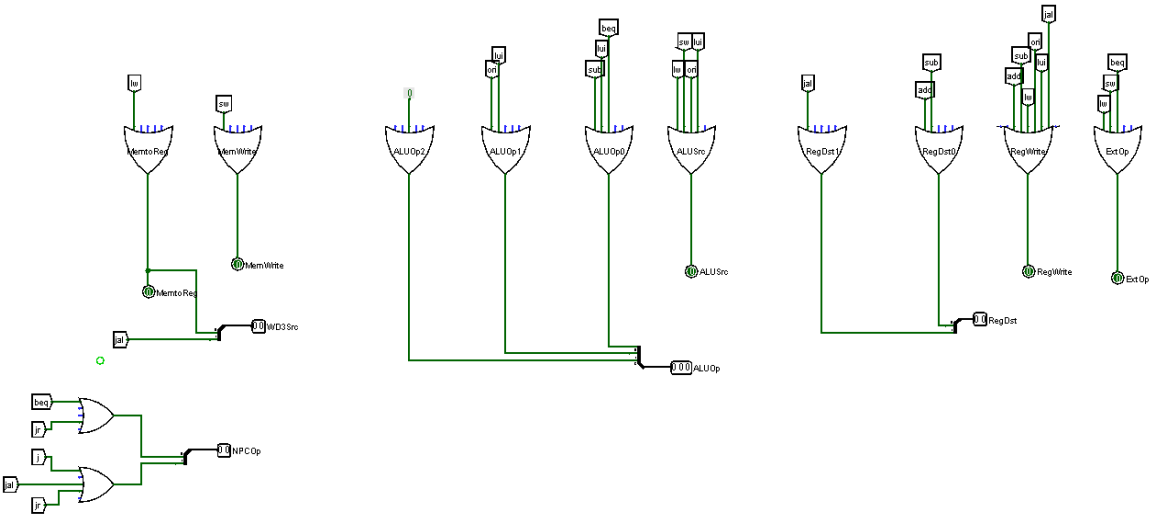
本控制器设计是基于cscore网站上**和逻辑**和**或逻辑**的设计思路进行实现的，该方案具有较强的可拓展性，便于后续进行更深入的实验，故采用本方法搭建控制器。在和逻辑中，只有当op位000000时（R类型指令），funct部分才会对该组合逻辑电路起作用。若op不为000000，代表该指令并非R类型指令，故不需要考虑funct部分的输入。

和逻辑部分：





或逻辑部分：



二、测试方案

在初步设计完成CPU后，主要分为计算类（ori,lui,add,sub）、存取类（sw,lw）以及跳转类（beq, j, jal, jr）指令进行测试。

（1）计算类

```
.text

####先进行ori\lui的测试
#0附近的数字
ori $a0, $0, 0
ori $a1, $0, 1
lui $t1, 0
lui $t2, 1

#16位无符号边界
ori $t3, $0, 65534
ori $t4, $0,65535
lui $t5, 65534
lui $t6,65535

#随机数
ori $t7,$0,34322
lui $t8,5346

#目标寄存器为0
ori $0,$0,2131
lui $0,5435

####测试add\sub
add $s1,$t0,$t2
add $s2,$t4,$t6
sub $s3,$s2,$t4
sub $s4,$s3,$s3

####测试寄存器的极限值
lui $s5,32767
ori $s5,65535   #t1=0x01111111_11111111_11111111_11111111
lui $s6,65535
ori $s6,65535   #t2=0x11111111_11111111_11111111_11111111
```

（2）存取类

```
.text
####测试存取指令
lui $t0,65535
ori $t0,$t0,65516   #t0 is -20
##base<0
    #offset>0
    ori $s1,$s1,0x00000001
    sw $s1,24($t0)  #4 is s1
    lw $s2,24($t0)
##base=0
  #offset=0
```

```
  ori $s1,$s1,0x00000010    #0 is s1
  sw $s1,0($0)
  lw $s3,0($0)
  #offset>0
  ori $s1,$s1,0x00000100
  sw $s1,8($0)  #8 is s1
  lw $s4,8($0)
##base>0
  ori $t1,$t1,20
  #offset<0
  ori $s1,$s1,0x00001000
  sw $s1,-8($t1)
  lw $s5,-8($t1)
  #offset=0
  lui $s1,0x0001
  sw $s1,0($t1)
  lw $s6,0($t1)
  #offset>0
  lui $s1,0x0010
  sw $s1,4($t1)
  lw $s7,4($t1)
```

(3) 跳转类

```
##j指令测试
ori $t1,$t1,4369
j test1

ori $t2,$t2,4369
test1:          #j:向前跳转测试

ori $t3,$t3,4369
j test2

test3:          #j: 向后跳转测试
ori $t5,$t5,4369
j end1

test2:
ori $t4,$t4,4369
j test3

end1:
ori $t6,$t6,4369

##jal指令测试
jal test4
ori $s7,$0,0x1111
j beq_test
##jr指令测试
test4:
```

```
add $t7,$ra,$0  #将ra的值存到t7之中
jr $ra

beq_test:
ori $t8,$t8,0x1111
ori $t9,$t9,0x1111
j branch1
##beq指令测试

#跳转且目标在指令之前
branch1_test:
ori $s0,$s0,0x1010
j branch3

branch1:

beq $t8,$t9,branch1_test
ori $s0,$s0,0x1111
j branch3

#跳转且目标就是这条指令
#branch2:
#beq $t8,$t9,branch2
#ori $s1,0x1010
#j branch3

branch3:
#跳转且目标在指令之后
beq $t8,$t9,branch3_test
ori $s2,$s2,0x1111
j branch4
branch3_test:
ori $s2,$s2,0x1010
j branch4

#不跳转且目标在指令之前
branch4_test:
ori $s3,$s3,0x1010
j branch5

branch4:
add $t8,$0,$0
ori $t8,$t8,0x111
beq $t8,$t9,branch4_test
ori $s3,$s3,0x1111
j branch5

branch5:
#不跳转且目标就是这条指令
beq $t8,$t9,branch5
ori $s4,$s4,0x1111
j branch6
```

```
branch6:
#不跳转且目标在指令之后
beq $t8,$t9,branch6_test
ori $s5,$s5,0x1111
j end

branch6_test:
ori $s5,$s5,0x1010
j end

end:
```

说明：经测试发现，beq强测试点仍不够强，未能检测除beq的bug。在本新测试数据中，beq在跳转的我时候发现立即数并未进行符号拓展而是进行了零拓展，导致NPC计算结果出错。

三、思考题回答

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。

状态存储功能：IFU、DM、GRF

状态转移功能：Controller、EXT、NPC、ALU

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。

合理：

1.ROM为只读存储器，一旦载入数据后便无法对其进行更改，且掉电不会丢失ROM中的数据，能够满足CPU取指令的需求。

2.RAM为随机存储器，用于DM中主要有以下三方面考量：1)ROM无法修改其内部的数据，无法满足DM可读写的功能。2)Register价格较高，而DM存储空间相对较大，若用Register则会导致CPU成本过高。3)RAM既具有较快的读写速度，又具有相对较低的成本，能够更均衡地满足DM地需求。

3.Register为高速寄存器，CPU在执行指令的过程中需要大量地对GRF进行读写数据操作，采用Register能够满足这种高速的需求。同时，许多指令均为R类型指令，需要直接对寄存器进行操作，采用Register为更合理的选择。

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。

我还设计了NPC（Next Program Counter）模块，具体功能为由输入端的PC，功能选择信号NPCOp、偏移Offset、指令Instr_Index，32位寄存器数据以及zero选择信号计算下一个时钟周期PC的值，并将其返回至IFU之中。

设计思路如下：本CPU支持beq、j、jal、jr四种跳转指令，需要设计一个专门的模块来处理计算下一个PC的值。针对四种不同的指令，可以设计如下功能表：NPCOp：00：PC+4，01:beq，10:j/jal，11:jr。在控制器生成对应的NPCOp信号的同时，将j/jal指令需要的指令[25:0]位，jr指令需要的32位输入寄存器数据输入至模块当中。同时，需要将ALU的输出zero也接到NPC模块之中，用以判别beq指令是否需要执行。

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?

nop空指令地RegWrite、MemWrite信号均为0，其余信号为x，故nop不会对GRF、DM等存储单元进行任何操作，因此无需加入控制信号真值表中。

5.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。

在设置为compact, data at address 0的情况下，MARS在导出PC时，起始地址为0x3000,DM的起始地址为0x0000，为解决此问题，可能的方案为在IIFU中采用同步/异步复位的方法，将PC寄存器的值恢复为0x3000,同时在ROM前增加片选信号，仅选择符合要求的值取出指令。

我最终采用的方案：在运行的过程中将Memory Configuration设置为compact, data at address 0，进行运行结果的比较。在需要导出机器码的时候，先将Memory Configuration设置为compact, text at address 0，再导出机器码并加载至logisim之中。

6.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。

各个指令的覆盖情况：

总体来看，该测试样例覆盖了CPU所支持的全部指令{ `ori`，`lui`，`add`，`sw`，`lw`，`beq`}，在指令测试的完整性上达到了最基本的要求。在进行该测试样例的测试与比较后，可以保证CPU在执行最弱的数据时大概率可以不出错。

单一指令各种行为的覆盖情况：

但具体来看，该测试样例在单一指令的测试覆盖仍有较大的改进空间。

在{ori,lui}指令的测试中，该测试样例仅简单构造了正负的测试，而并未分为对16位立即数边缘、寄存器32位边缘以及随机数三个方面进行测试，并未测试CPU在极端数据下的正确性。同时，这两条指令也并未覆盖将$0作为目标寄存器的情况，$0的值是否可修改并未被测试。

在{add}指令的测试中，测试样例覆盖了正正、正负、负负三种情况，基本覆盖数据的正负性，但还可以利用add指令测试寄存器的边缘值，进一步利用极端数据测试CPU的正确性。

在{sw,lw}指令的测试中，测试样例的base与offset均为正数，测试强度较弱。实际上，base与offset均有正、0、负三种可能，在具体测试的过程中应当对这几种情况均进行覆盖。

在{beq}指令的测试中，测试样例仅测试了跳转目标为当前指令之后的情况，仍可以测试跳转目标在当前指令之前以及就是当前指令这两种情况；除此之外，原测试样例在`beq $a0, $a1, loop1`成立的前提下跳转到loop1，在执行完loop1后仍会顺序执行loop2的操作，本人认为在测试beq的过程中应尽可能减少各条指令对互相的干扰。