# 计算机组成原理实验报告

16061007 王文珺

## 一、 CPU 设计文档

### （一）整体结构

本次设计的 CPU 为 32 位处理器，支持{addu,subu,ori,lw,sw,beq,lui,nop}指令集，其中 nop 指令为空指令，不进行任何有效行为。处理器将包含 Controller（控制器）、IFU（取指令单元）、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）、ALU（算术逻辑单元）、DM（数据存储器）、 EXT（位扩展器）等基本部件，通过 MUX、 Splitter 等内置器件组合连接成数据通路。顶层设计参考了下图 ：

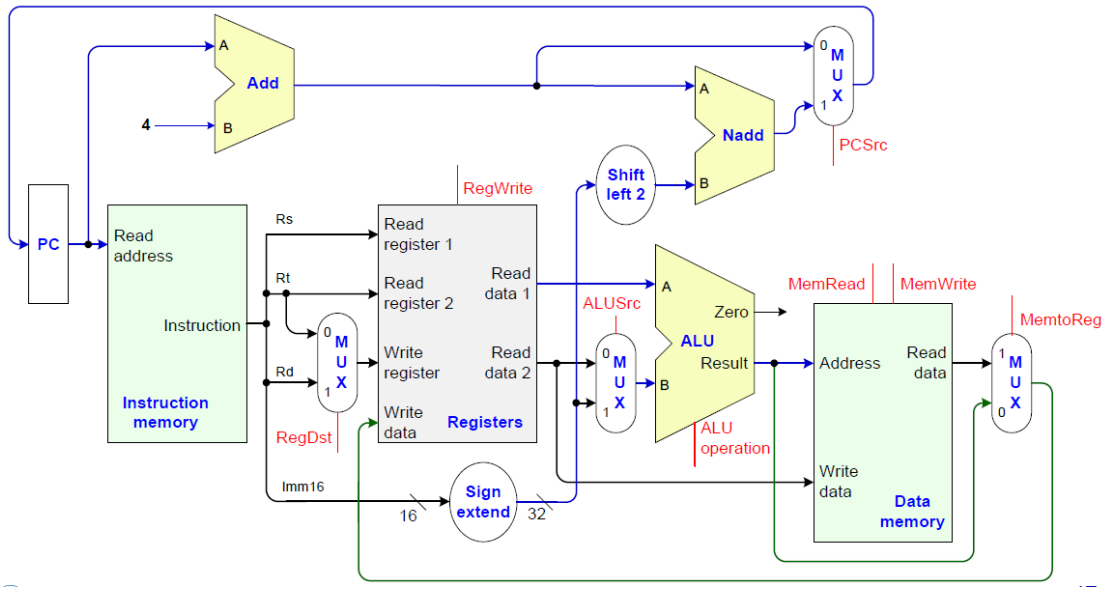



Figure 1 顶层结构设计

### （二）模块规格

#### 1、取指令单元（IFU）

IFU 主要包括 PC 寄存器和指令存储器 IM，利用加法器、多路选择器，以及对立即数进行处理的扩展器和移位器实现相关逻辑。其中 PC 存储器起始地址为

0x00000000，具有复位功能。IM 使用 ROM 实现，容量为 32bit*32，结构图如下：

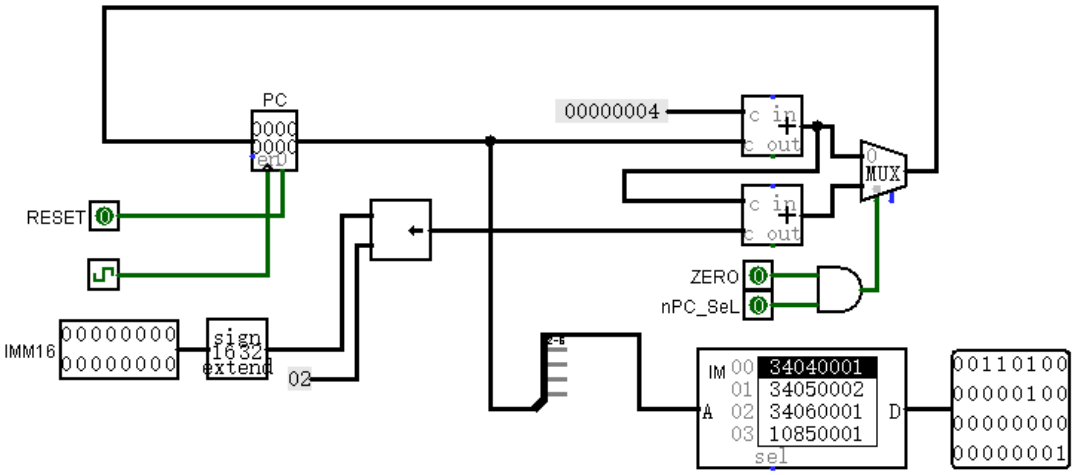



Figure 2 IFU 结构设计

接下来给出端口定义与功能说明：

Table 1 IFU 端口及功能说明

| 序号 | 端口名称 | I\O | 功能名称 | 功能描述 |
|---|---|---|---|---|
| 1 | RESET | I | 复位 | 当复位信号有效时，PC 被设置为 0x00000000 |
| 2 | ZERO | I | 相等判断 | 当 ZERO 信号有效时，表示 beq 指令判断相等 |
| 3 | nPC_Sel | I | 地址选择 | 当 nPC_Sel 信号有效时，表示正在执行 beq 指令 |
| 4 | IMM16 | I | 16 位立即数 | 输入 16 位立即数用于对地址进行操作 |
| 5 | Instr | O | 指令输出 | 输出当前指令编码，用于后续指令分析 |

每个时钟周期内，PC 的值同时传入指令存储器 IM，因为地址每次+4，所以对应取其 2-6 位为 IM 的地址输入，取得当前指令编码输出。同时计算下一条指令地址，分别经过 PC+4 以及 PC+4+signext(offset||00)两个运算（其中 offset 数据即为 16 位立即数 IMM16），分别计算正常顺序执行的下一条指令地址以及 beq 指令的跳转地址，通过 ZERO 和 nPC_Sel 两个控制信号通过与门和多路选择器进行判断选择并存入 PC 寄存器。

### 2、通用寄存器组（GRF）

利用 32 个具有写使能的寄存器实现，且 0 号寄存器的值保持为 0，构造的过

程中需要注意的是为了保证在时钟上升沿写入寄存器，寄存器的 clk 端口需要直接连接时钟，使能端连接寄存器选择端口与 RegWrite 控制信号经过与门的信号。接下来给出端口定义与功能说明：

Table 2 GRF 端口及功能说明

| 序号 | 端口名称 | I\O | 功能名称 | 功能描述 |
|---|---|---|---|---|
| 1 | Rreg1 | I | 寄存器 1 选择 | 通过多路选择器选择寄存器 1 |
| 2 | Rreg2 | I | 寄存器 2 选择 | 通过多路选择器选择寄存器 2 |
| 3 | Wreg | I | 选择写入寄存器 | 当控制信号 RegWrite 有效时，在时钟上升沿将 Wdata 中的数据写入 Wreg 通过 Decoder 选择的寄存器中， |
| 4 | RegWrite | I | 写入控制 | |
| 5 | Wdata | I | 写入的数据 | |
| 6 | Rdata1 | O | 数据 1 输出 | 输出寄存器 1 的数据 |
| 7 | Rdata2 | O | 数据 2 输出 | 输出寄存器 2 的数据 |

### 3、算术逻辑单元（ALU）

利用 logisim 自带的加法器、减法器、比较器和基础门电路分别生成计算结果，通过 ALUOp 信号利用多路选择器选择适当结果输出，提供 32 位加、减、或运算及大小比较功能，但是暂时不检测溢出。

结构图如下：

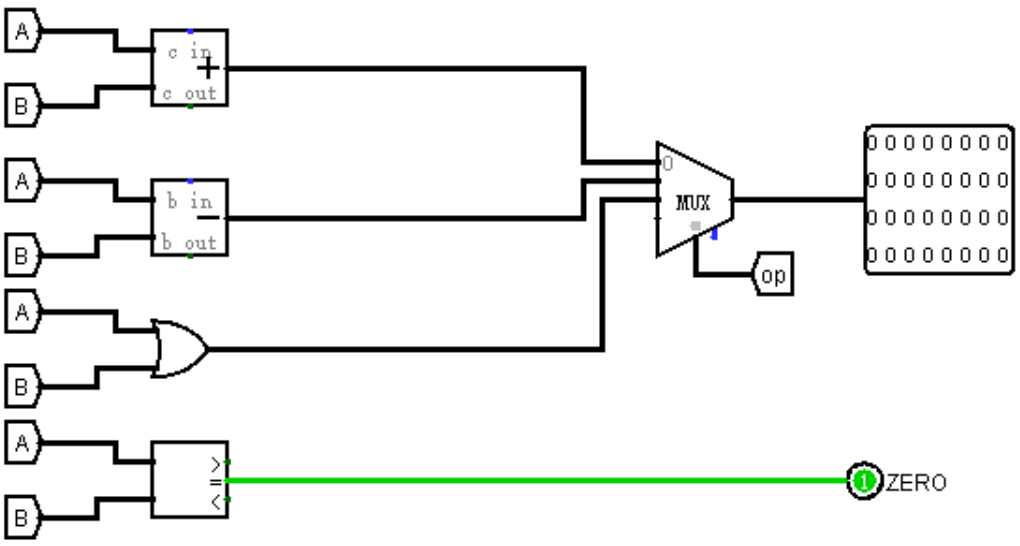



Figure 3 ALU 结构设计

接下来给出端口定义与功能说明：

Table 3 ALU 端口及功能说明

| 序号 | 端口名称 | I\O | 功能名称 | 功能描述 |
|---|---|---|---|---|
| 1 | A | I | 被处理数 A | 将 32 位数据 A 读入 ALU |
| 2 | B | I | 被处理数 B | 将 32 位数据 B 读入 ALU |
| 3 | ALUOp | I | 操作选择 | 00:加法对应 addu 指令<br>01:减法对应 subu 指令<br>10:按位或对应 ori 指令 |
| 4 | Result | O | 结果 | 根据 ALUOp 通过多路选择器选择的相应输出 |
| 5 | ZERO | O | 相等判断 | 当 A=B 时，ZERO=1 |

### 4、数据存储器（DM）

利用 RAM 实现数据存储器，容量为 32bit*32，起始地址为 0x00000000，主要负责数据的存储写入和读取输出，结构图如下：

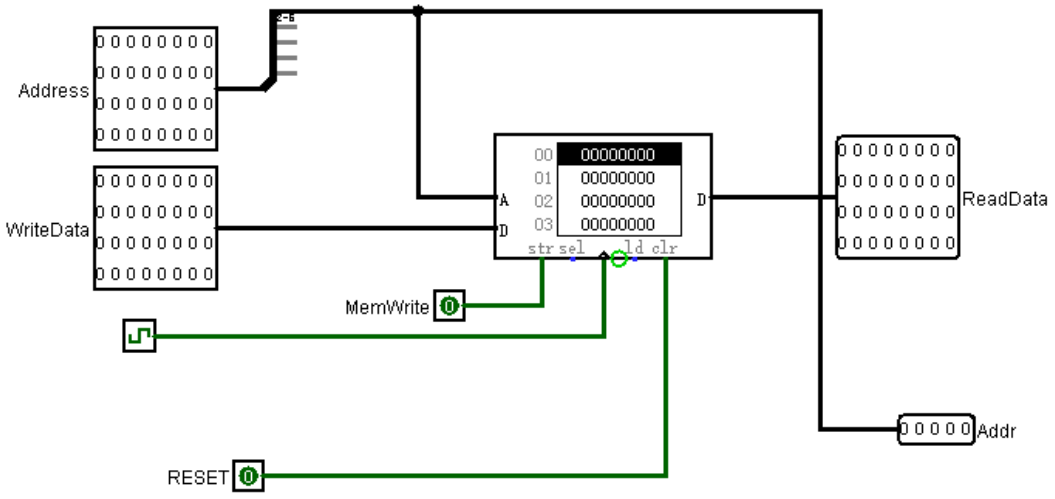



Figure 4 DM 结构设计

接下来给出端口定义与功能说明：

Table 4 DM 端口及功能说明

| 序号 | 端口名称 | I\O | 功能名称 | 功能描述 |
|---|---|---|---|---|
| 1 | RESET | I | 复位 | 当复位信号有效时，DM 中的数据被全部清零 |
| 2 | Address | I | 32 位地址 | 利用 Splitter 选择 DM 中存储的某条数据 |
| 3 | WriteData | I | 写入的数据 | 当 MemWrite 信号有效时，将 WriteData 写入 Address 所选择的地址在 DM 中的位置 |
| 4 | MemWrite | I | 数据写入 | |
| 5 | ReadData | O | 读取的数据 | 输出 DM 被选择的位置所保存的数据 |
| 6 | Addr | O | 5 位地址 | 输出 Address 选择 2-6 位后的数据，便于检查 |

### 5、立即数处理单元（EXT）

EXT 本来是对立即数进行扩展（符号扩展及零扩展），但是我在这里为了统一对立即数的操作，把 lui 指令所需要的加载至高位合并了起来，并且合称该 EXT 为“立即数处理单元”，在之后的设计中，所有需要对立即数进行操作的指令 我将都合并进此立即数处理单元。结构图如下：

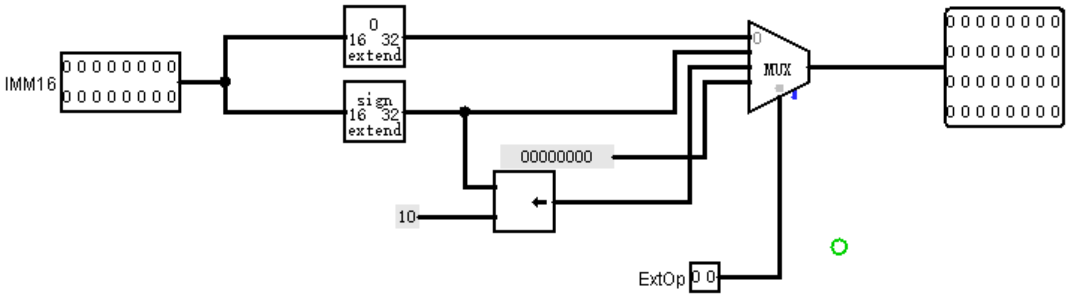



Figure 5 EXT 结构设计

接下来给出端口定义与功能说明：

Table 5 EXT 端口及功能说明

| 序号 | 端口名称 | I\O | 功能名称 | 功能描述 |
|---|---|---|---|---|
| 1 | IMM16 | I | 16 位立即数 | 输入被处理的 16 位立即数 |
| 2 | ExtOp | I | 操作选择 | 00:零扩展\01:符号扩展\10:立即数低位补 16 位 0 |
| 3 | IMM32 | O | 32 位立即数 | 输出被处理后的 32 位立即数 |

### 6、控制器（Controller）

控制器使用与或门阵列构造控制信号，由于控制器较为特殊以及其在 CPU 中的核心地位，构造方法较为复杂，具体方法见下一节。

## （二）控制器设计

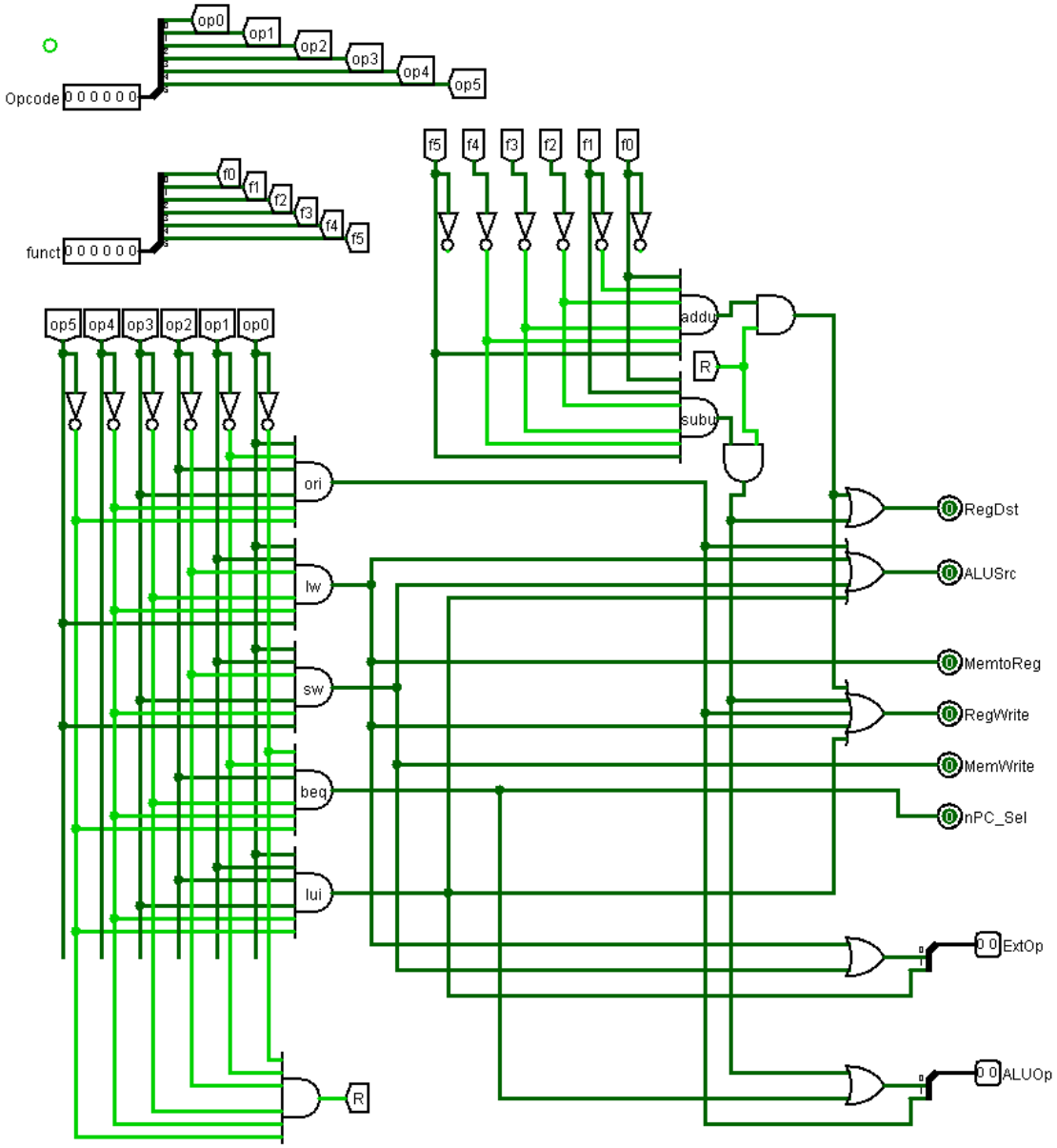
根据教程中提供的思路，将解码逻辑分为和逻辑和或逻辑两部分：和逻辑的功能是识别，将输入的机器码识别为相应的指令；或逻辑的功能是生成，根据输入的指令的不同，产生不同的控制信号。根据此 CPU 要求识别的指令集，并且分析 MIPS-C 指令集，可以得到如下表格：

Table 6 控制信号真值表

| func | 100001 | 100011 | N\A | | | | |
|---|---|---|---|---|---|---|---|
| op | 000000 | 000000 | 001101 | 100011 | 101011 | 000100 | 001111 |
| | addu | subu | ori | lw | sw | beq | lui |
| RegDust | 1 | 1 | 0 | 0 | x | x | 0 |
| ALUSrc | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| MemtoReg | 0 | 0 | 0 | 1 | x | x | 0 |
| RegWrite | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| MemWrite | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| nPC_Sel | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ExtOp[1:0] | x | x | 00 | 01 | 01 | x | 10 |
| | add | sub | or | add | add | sub | add |
| ALUOp[1:0] | 00 | 01 | 10 | 00 | 00 | 01 | 00 |

可以通过该表格得到每个控制信号的逻辑，其中 ALUOp 和 ExtOp 的值均为根据本 CPU 的处理单元所决定的，其中 lui 指令为加载立即数至高位，并不涉及另一个操作数，所以实际上它的 ALUOp 并不是必须为 00（add），这里只是暂时把它看做 32 位立即数与 0 相加。

其与或门逻辑实现如下：




接下来给出每个控制信号的意义：

Table 7 控制信号名称及功能描述

| 序号 | 控制信号名称 | 功能描述 |
|---|---|---|
| 1 | RegDust | 当前指令为 R 型指令时有效，选择 rd 寄存器进行写入，信号无效时将选择 rt 寄存器进行写入 |
| 2 | ALUSrc | 当其有效时为处理过的立即数与 rs 寄存器中的数据运算，信号无效时为 rt 寄存中的数据与 rs 寄存器中的数据运算 |
| 3 | MemtoReg | 当指令为 lw 时有效，从 DM 选择相应地址段的数据传入 GRF 的数据写入端，信号无效时为将 ALU 的计算结果传入数据写入端 |

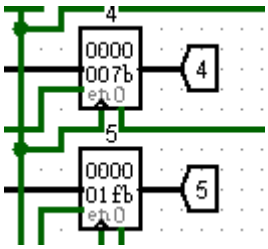
| | | |
|---|---|---|
| 4 | RegWrite | 当需要向寄存器中写入数据时有效，将数据写入端的数据写入选择的寄存器，信号无效时不执行该操作 |
| 5 | MemWrite | 当指令为 sw 时有效，将 GRF 传出的 Rdata2 数据写入 DM 通过 ALU 计算出的相应地址段，信号无效时不执行该操作 |
| 6 | nPC_Sel | 当指令为跳转指令时有效，例如 beq，bne，j 等，判断是否生效后将选择经过计算后的跳转地址存入 PC 寄存器以便选择下一条指令，无效时即为顺序向下执行一条指令（PC+4） |
| 7 | ExtOp[1:0] | 00：零扩展 01：符号扩展 10：立即数低位补 16 位 0 |
| 8 | ALUOp[1:0] | 00：加 01：减 10：或 |

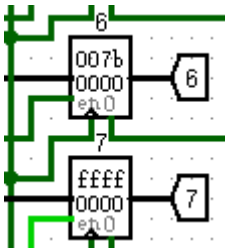
考虑到之后的扩展性，这里准备将对立即数进行的操作放入 ExtOp 中，例如左移右移等。对两个操作数进行的运算放入 ALUOp，例如与非，或非，乘法除法等。通过增加 ExtOp 和 ALUOp 的位数来实现更多的操作，其与或门的连接逻辑仍然采用前文提到的方法。

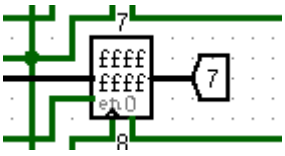
## （三）测试程序

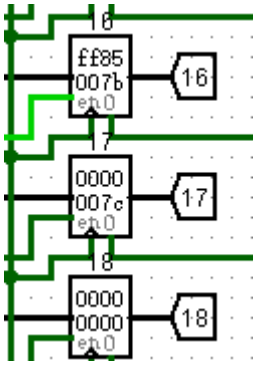
为了保证测试所有的指令，这里编写了如下的测试程序并给出测试期望

```
.text
#ori 指令测试
ori $a0,$0,123  # $4 被写入 0x0000007b
ori $a1,$a0,456  # $5 被写入 0x000001fb
#lui 指令测试
lui $a2,123  # $6 被写入 0x007b0000
lui $a3,0xffff  # $7 被写入 0xffff0000
ori $a3,$a3,0xffff  # $7 被写入 0xffffffff
#subu 指令测试
subu $s0,$a0,$a2  # $16 被写入 0xff85007b
subu $s1,$a0,$a3  # $17 被写入 0x0000007c
```

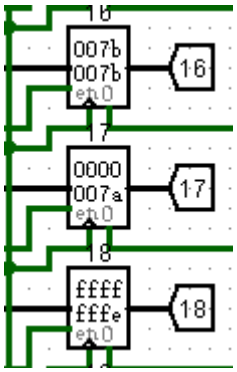


```
subu $s2,$a3,$a3  # $18 被写入 0x00000000
```

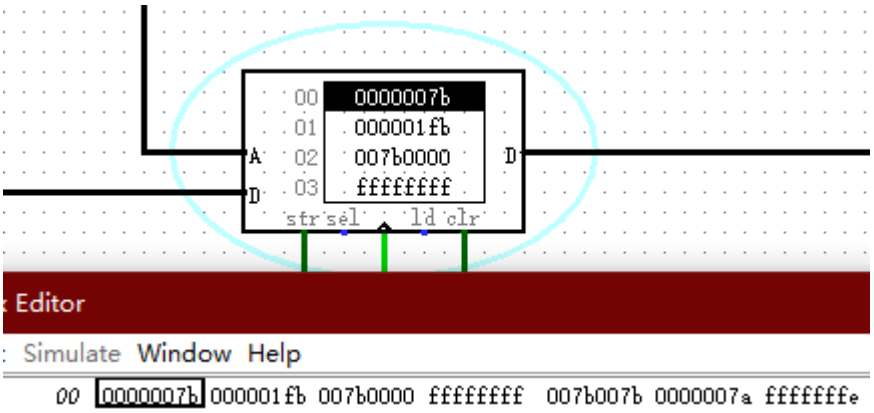


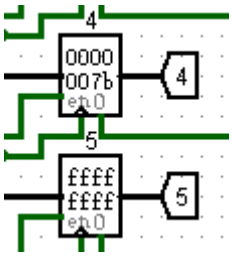

#addu 指令测试

```
addu $s0,$a0,$a2  # $16 被写入 0x007b007b
addu $s1,$a0,$a3  # $17 被写入 0x0000007a
addu $s2,$a3,$a3  # $18 被写入 0xfffffffe
```

#sw 指令测试

```
ori $t0,$0,0x0000  # $8 被写入 0x00000000
sw $a0,0($t0) # DM 中地址 0x00000000 被写入 0x0000007b
sw $a1,4($t0) # DM 中地址 0x00000004 被写入 0x000001fb
sw $a2,8($t0) # DM 中地址 0x00000008 被写入 0x007b0000
sw $a3,12($t0) # DM 中地址 0x00000012 被写入 0xffffffff
sw $s0,16($t0) # DM 中地址 0x00000016 被写入 0x007b007b
sw $s1,20($t0) # DM 中地址 0x00000020 被写入 0x0000007a
sw $s2,24($t0) # DM 中地址 0x00000024 被写入 0xfffffffe
```



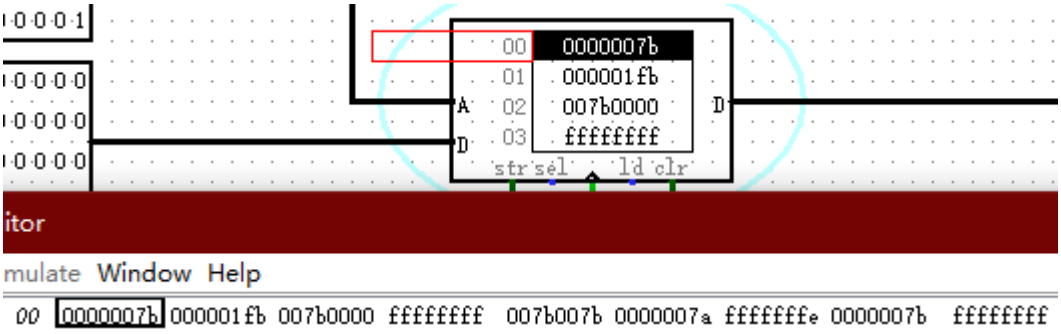




#lw 指令测试

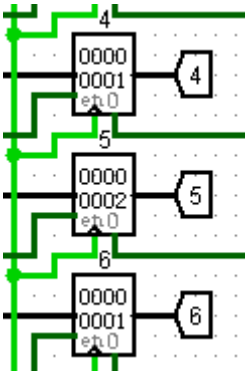
```
lw $a0,0($t0) # DM 中地址 0x00000000 的数据 0x0000007b 被写入$4
lw $a1,12($t0) # DM 中地址 0x00000012 的数据 0xffffffff 被写入$5
```

**sw $a0,28($t0)** # DM 中地址 0x00000028 被写入 0x0000007b

**sw $a1,32($t0)** # DM 中地址 0x00000032 被写入 0xffffffff




**#beq 指令测试**



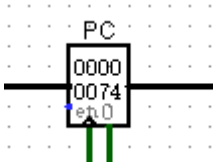

**ori $a0,$0,1** # $4 被写入 0x00000001

**ori $a1,$0,2** # $5 被写入 0x00000002

**ori $a2,$0,1** # $6 被写入 0x00000001

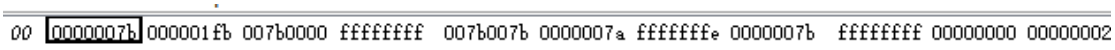
**beq $a0,$a1,loop1** # $4 与$5 中数据不相等，不跳转至 loop1，PC+4，执行下一条命令，PC=0x0000306c

**beq $a0,$a2,loop2** # $4 与$6 中数据相等，跳转至 loop2，PC=0x00003074




**loop1:sw $a0,36($t0)** #地址为 0x00003070

**loop2:sw $a1,40($t0)** #地址为 0x00003074 DM 中地址 0x00000040 被写入 0x00000002




所以经过测试，搭建的 CPU 结果与预期相同

## 二、思考题

### （一）模块规格部分

1、若 PC（程序计数器）位数为 30 位，试分析其与 32 位 PC 的优劣。

30 位 PC 只能寻址 256M 个字的位置，而 32 位 PC 能寻址 1GB 字的位置，因此在同样的 PC+4 操作中，32 位 PC 的寻址范围更广，可以从指令存储器中独处的指令数是 30 位 PC 的四倍。倘若把 32 位 PC+4 的操作用 30 位 PC+1 的操作替换，在 logisim 中可以实现相同的功能，但是由于本 CPU 的其他单元，例如寄存器，IM，DM 中的数据都是 32 位的，所有相对应的操作，包括偏移量，立即数处理等便都需要更改，但是在取地址时就不需要取 2-6 位，直接取低 5 位便是当前指令的地址。所以我认为 32 位 PC 的优势大于 30 位 PC。

2、现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

我认为很合理。IM 在 CPU 运行过程中保存了执行的指令，不应该在过程中被改变，所以采用只读存储器 ROM 实现。而 DM 在 CPU 运行过程中其储存的值可以利用 sw 指令更改或者覆盖重写，因此采用随机访问存储器 RAM 实现。GRF 为了满足快速读写操作的需要，采用寄存器，可以更快的读写数据以便后续单元处理使用。

### （二）控制器设计部分

1、结合上文给出的样例真值表，给出 RegDst， ALUSrc， MemtoReg，RegWrite, nPC_Sel, ExtOp 与 op 和 func 有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。

RegDst=$\overline{o5o4o3o2o1o0}f5\overline{f4f3f2f1}f0$+$\overline{o5o4o3o2o1o0}f5\overline{f4f3f2}f1f0$

ALUSrc=$\overline{o5}\overline{o4}o3o2\overline{o1}o0 + o5\overline{o4o3o2}o1o0 + o5\overline{o4}o3\overline{o2}o1o0 + \overline{o5o4}o3o2o1o0$

MemtoReg=$\overline{o5}o4o3o2\overline{o1o0}$

RegWrite= $\overline{o5o4o3o2o1o0}f5\overline{f4f3f2f1}f0 + \overline{o5o4o3o2o1o0}f5\overline{f4f3f2}f1f0 + \overline{o5o4}o3o2\overline{o1}o0 + o5\overline{o4o3o2}o1o0 + \overline{o5o4}o3o2o1o0$

MemWrite=$o5\overline{o4}o3\overline{o2}o1o0$

nPC_Sel=$\overline{o5o4o3}o2\overline{o1o0}$

ExtOp[1]= $\overline{o5o4}o3o2o1o0$

ExtOp[0]= $o5\overline{o4o3o2}o1o0 + o5\overline{o4}o3\overline{o2}o1o0$

2、充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式，请给出化简后的形式。

RegDst=$\overline{o5o4o3o2o1o0}f5\overline{f4f3f2}f0$

ALUSrc=$\overline{o5o4}o3o2o0 + o5\overline{o4o2}o1o0$

MemtoReg=$\overline{o5}o4o2\overline{o1o0}$

RegWrite=$\overline{o5o4o3o2o1o0}f5\overline{f4f3f2}f0$+$\overline{o5o4}o3o2o0 + o5\overline{o4o3o2}o1o0$

nPC_Sel=$\overline{o5o4o3}o2\overline{o1o0}$

ExtOp[1]= $\overline{o5o4}o3o2o1o0$

ExtOp[0]= $o5\overline{o4o2}o1o0$

3、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。

Nop 指令的机器码为 0x00000000 为对 0 号寄存器进行操作，但是 0 号寄存器是不能被修改的，换而言之 nop 指令并没有任何操作，所以不必将 nop 指令加入控制信号真值表。

## （三）测试程序部分

1、前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM 片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。

Data base address[31:0]-0x00003000 因此，data base address[13]=1，data base address[12]=1，其余为都为零，将这两位的与作为 DM 中 RAM 的控制信号之一，与 MemWrite 控制信号通过与门控制 str 端口，只有当这两位均为 1 时才可以对 DM 进行写操作。

2、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，简要阐述相比与测试，形式验证的优劣。

形式验证优点在于不必考虑输入具体数据进行仿真验证，避免了穷举所不能完成的对所有情况进行验证，耗时短，减少设计周期，易于发现错误并且进行修改。形式验证可以进行从系统级到门级的验证，缺点在于不能有效的验证电路的性能，如电路的时延和功耗，且对抽象思维的要求很高。