

计算机组成原理

•实验报告•

专业: \_\_\_\_\_

班级: \_\_\_\_\_

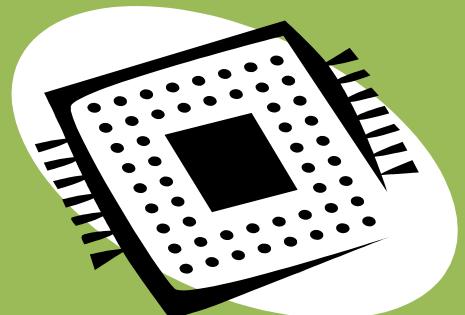
学号: \_\_\_\_\_

姓名: Dracula

电话: \_\_\_\_\_

邮件: \_\_\_\_\_

完成日期: 2016-12-25



计算机科学与技术学院

## 目 录

<b>1 运算器实验</b>	<b>2</b>
1.1 设计要求	2
1.2 方案设计	3
1.3 实验步骤	9
1.4 故障与调试	9
1.5 测试与分析	11
<b>2 存储器实验</b>	<b>14</b>
2.1 设计要求	14
2.2 方案设计	15
2.3 实验步骤	18
2.4 故障与调试	19
2.5 测试与分析	22
<b>3 CPU 实验</b>	<b>24</b>
3.1 设计要求	24
3.2 方案设计	26
3.3 实验步骤	33
3.4 故障与调试	34
3.5 测试与分析	37
<b>4 组成原理与编译原理</b>	<b>39</b>
<b>5 总结与心得</b>	<b>43</b>
5.1 实验总结	43
5.2 实验心得	43
<b>参考文献</b>	<b>45</b>

## 1 运算器实验

### 1.1 设计要求

利用 logisim 平台中现有运算部件构建一个 32 位运算器，可支持算数加、减、乘、除，逻辑与、或、非、异或运算、逻辑左移、逻辑右移，算术右移运算，支持常用程序状态标志（有符号溢出 OF、无符号溢出 CF，结果相等 Equal），运算器功能以及输入输出引脚见下表，在主电路中详细测试自己封装的运算器。

表 1.1 ALU 引脚与功能描述

引脚	输入/输出	位宽	功能描述
X	输入	32	操作数 X
Y	输入	32	操作数 Y
ALU_OP	输入	4	运算器功能码，具体功能见下表
Result	输出	32	ALU 运算结果
Result2	输出	32	ALU 结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零
OF	输出	1	有符号加减溢出标记，其他操作为零
CF	输出	1	无符号加减溢出标记，其他操作为零
Equal	输出	1	Equal=(x==y)?1:0, 对所有操作有效

表 1.2 运算符功能

ALU_OP	十进制	运算功能			
0000	0	Result = X << Y	逻辑左移	(Y 取低五位)	Result2=0
0001	1	Result = X >>> Y	算术右移	(Y 取低五位)	Result2=0
0010	2	Result = X >> Y	逻辑右移	(Y 取低五位)	Result2=0
0011	3	Result = (X * Y)[31:0];	Result2 = (X * Y)[63:32]	有符号	
0100	4	Result = X/Y;	Result2 = X%Y	无符号	
0101	5	Result = X + Y	Result2=0	(Set OF/CF)	

# 华中科技大学课程实验报告

0110	6	Result = X - Y Result2=0 (Set OF/CF)
0111	7	Result = X & Y Result2=0
1000	8	Result = X   Y Result2=0
1001	9	Result = X $\oplus$ Y Result2=0
1010	10	Result = $\sim(X   Y)$ Result2=0
1011	11	Result = (X < Y) ? 1 : 0 Signed Result2=0
1100	12	Result = (X < Y) ? 1 : 0 Unsigned Result2=0
1101	13	Result = Result2=0
1110	14	Result = Result2=0
1111	15	Result = Result2=0

## 1.2 方案设计

### 1.2.1 4 位先行进位加法器

所谓先行进位是指，当操作数给出时就能立刻确定各位的进位，而不再需要等待前一位计算完才能得到本位的进位。设计时，采用并行加法器进位链的方式实现，即通过下面的公式（具体推导就不推导了）实现先行进位。

$$G_i = X_i Y_i \quad P_i = X_i \oplus Y_i \quad C_i = G_i + P_i C_{i-1}$$

从而得到：

$$C_1 = G_1 + P_1 C_0$$

$$C_2 = G_2 + P_2 C_1 = G_2 + P_2 G_1 + P_2 P_1 C_0$$

$$C_3 = G_3 + P_3 C_2 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 C_0$$

$$C_4 = G_4 + P_4 C_3 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 C_0$$

通过上述公式推导，不难看出，每位的进位只与  $P_i$ 、 $G_i$  和  $C_0$  有关，而这三个量可在给出操作数后，经过最多一个门电路延迟得到，从而实现了先行进位（并行进位）的目的。

根据上述公式，很容易的设计出四位先行进位加法器电路 adder，如图 1.1 所示。

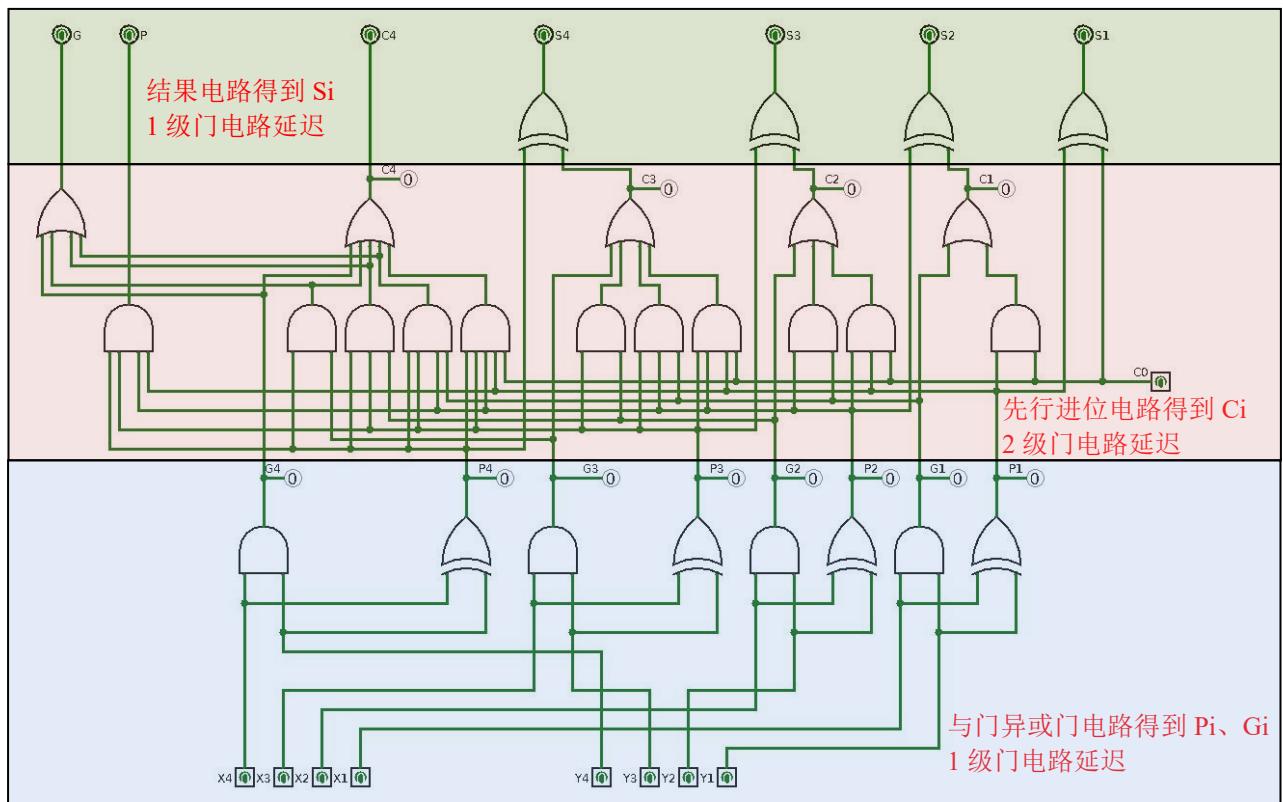


图 1.1 4 位先行进位加法器电路

根据图 1.1 中的延迟分析, 容易知道, 进位  $C_i$ 、 $P_i$ 、 $G_i$  需要  $1+2=3$  级门电路延迟就能得到, 而计算结果  $S_i$  则需要  $1+2+1=4$  级门电路延迟得到。

### 1.2.2 先行进位电路

1.2.1 只是得到了 4 位加法器, 并不是最终所需要的 32 位加法器, 因此需要将 8 个 4 位加法器进行级联得到 32 位加法器, 但若如果只是单纯的串联起来, 并不能达到组间并行进位的目的。为了达到组间并行的目的, 重新考虑公式:

$$C4 = G4 + P4C3 = G4 + P4G3 + P4P3G2 + P4P3P2G1 + P4P3P2P1C0$$

$$\text{如果设 } G4^* = G4 + P4C3 = G4 + P4G3 + P4P3G2 + P4P3P2G1, \quad P4^* = P4P3P2P1$$

则有  $C4 = G4^* + P4^*C0$ , 进一步可以得到:

$$C4 = G4^* + P4^*C0$$

$$C8 = G8^* + P8^*C4 = G8^* + P8^*G4^* + P8^*P4^*C0$$

$$C12 = G12^* + P12^*C8 = G12^* + P12^*G8^* + P12^*P8^*G4^* + P12^*P8^*P4^*C0$$

$$C16 = G4^* + P4^*C12$$

$$= G16^* + P16^*G12^* + P16^*P12^*G8^* + P16^*P12^*P8^*G4^* + P16^*P12^*P8^*P4^*C0$$

根据上述公式，很容易的设计出先行进位电路 carrier，如图 1.2 所示。

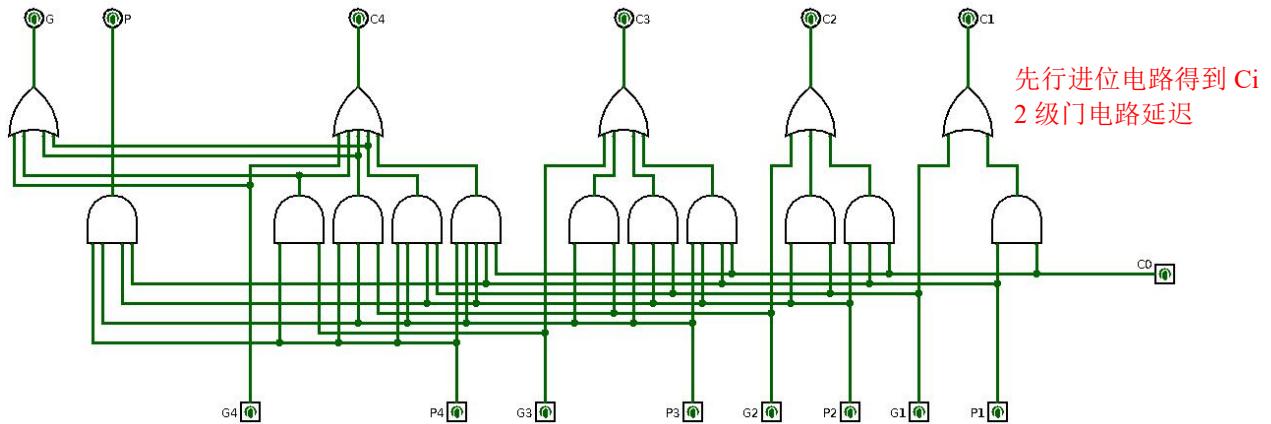


图 1.2 先行进位电路

显然，该先行进位电路需要两级门电路延迟，且通过 P 和 G 可作进一步级联。

### 1.2.3 32 位加法器电路

通过 1.2.1 设计的 4 位先行进位加法器电路和 1.2.2 设计的先行进位电路，可以很容易地得到一个组内先行进位、组间先行进位的 16 位加法器电路。32 位的加法器电路可由两个 16 位加法器电路连接而成，但有多种连接方式：

一种是，仍然使用先行进位电路连接两个 16 位加法器电路，但只使用先行进位电路的一部分，这样虽然有些浪费，但也是一种不错的解决方案，这种设计方案的电路图如图 1.3 所示；另一种方式则采用直接串联的方式，即直接将 16 位加法器电路 1 的进位作为 16 位加法器电路 2 的借位，这样这两组 16 位加法器电路之间便不再是组间先行进位，这种设计方案的电路图如图 1.4 所示。

接下来，对这两种 32 位加法器的延迟进行分析：

第一种 16 位加法器组间先行进位的设计方案里，考虑最长门电路延迟，即考虑第二个 16 位加法器的门电路延迟。其借位  $cin$  需要第一个 16 位加法器电路给出， $cin$  需要  $3(\text{adder 计算 } P*G^*)+2(\text{carrier 计算 } C1\sim4)+2(\text{carrier16 计算 } C1)=7$  级门电路延迟，则 32 位加法器总共需要  $7(cin)+2(\text{carrier 计算 } C1\sim4)+3(\text{adder 计算结果})=12$  级门电路延迟。这里需要注意的是，硬件电路是可以并行的，即两个 16 位加法电路里的 adder 是可以同时计算  $P*G^*$  的。

同理，第二种 16 位加法器组间直接串联的设计方案里， $cin$  需要  $3(\text{adder 计算}$

$P \cdot G^* + 2(\text{carrier 计算 } C1 \sim 4) = 5$  级门电路延迟，则 32 位加法器总共需要  $5(\text{cin}) + 2(\text{carrier 计算 } C1 \sim 4) + 3(\text{adder 计算结果}) = 10$  级门电路延迟。可见，这种设计方案不仅硬件成本较低，运行效率也更优。

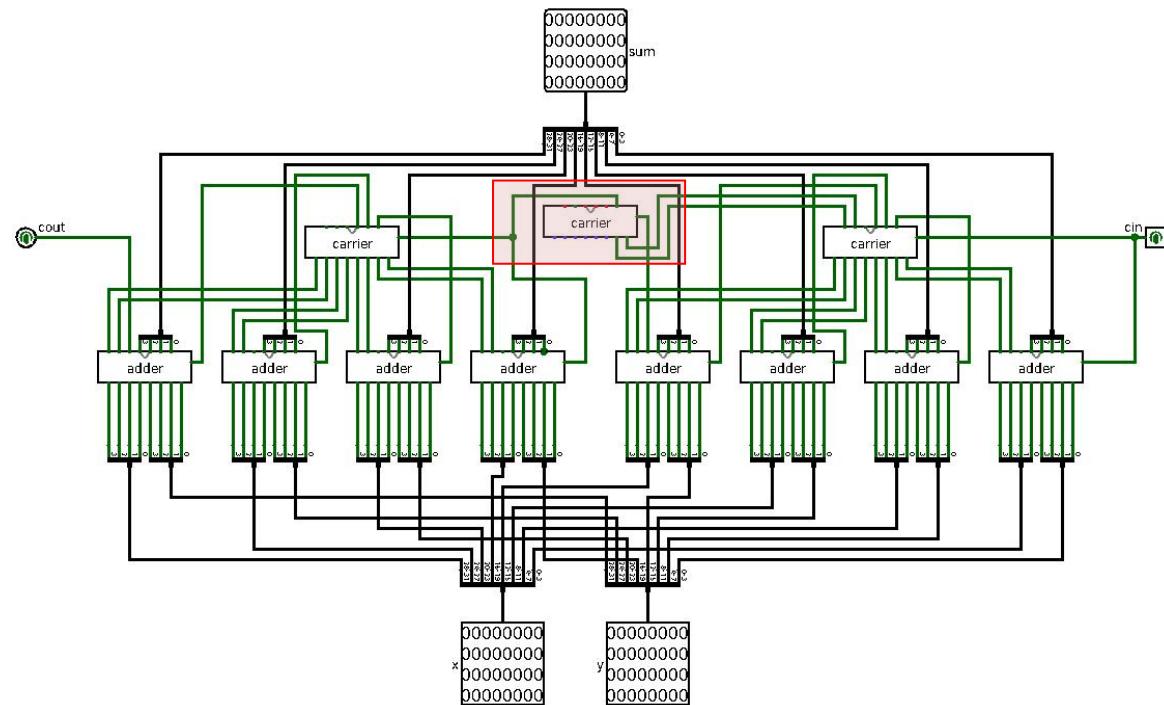


图 1.3 先行进位 32 位加法器（组件进位全部使用先行进位）电路

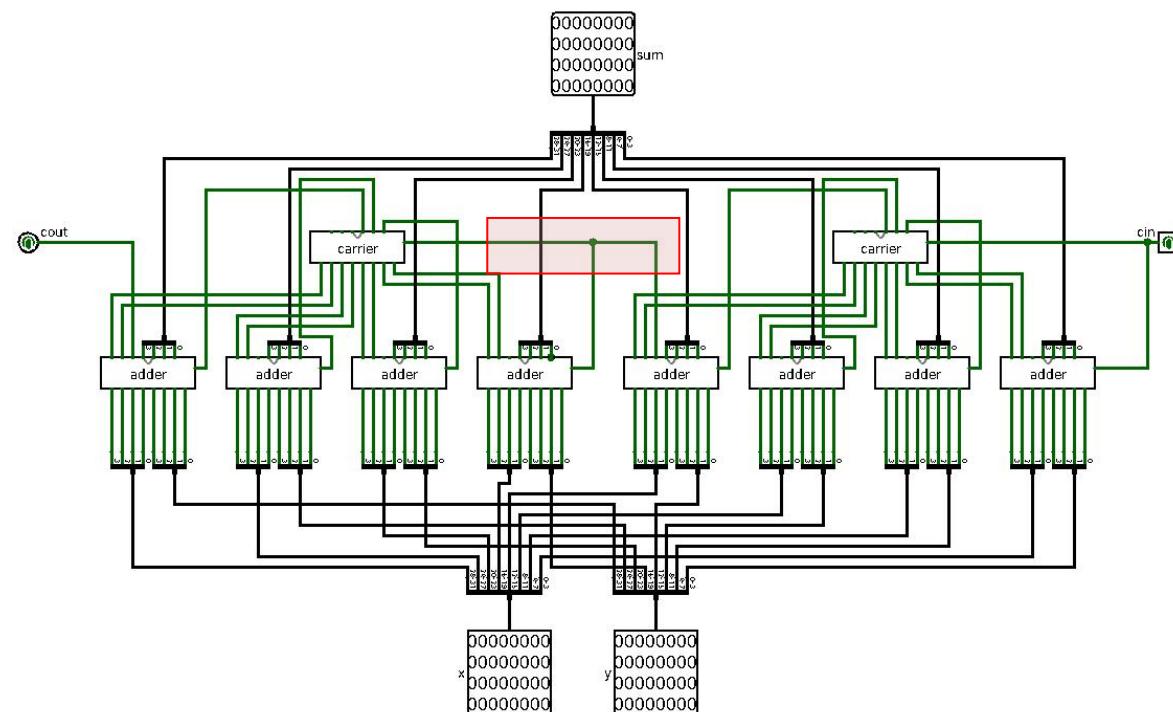


图 1.4 先行进位 32 位加法器（16 位之间直接串联）电路

## 1.2.4 ALU

ALU 的设计主要可以分为两个部分，第一部分是设计并实现各个操作（乘除加减等）的电路，第二部分则是使用一个多路选择器，实现给定操作码然后给出对应的操作结果的功能。

第一部分，对于除了加（op 操作码为 5）和减（op 操作码为 6）外，其余操作都可利用 logisim 自带运算器实现，因此实现起来相对简单。唯一需要注意的一点是，Equal 标志位可以通过 11 号操作用到的无符号比较器的“=”位得。这 11 种操作的运算电路如图 1.5 所示。

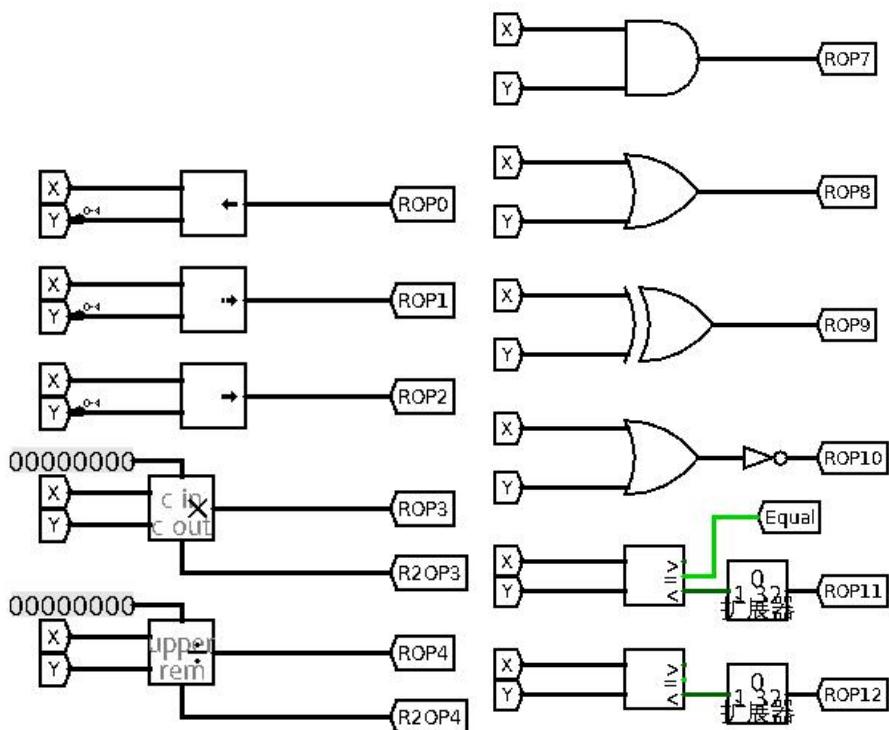


图 1.5 除加法减法外剩余 11 种操作运算电路

加法和减法均可通过一个加法器实现，其中减法采用补码运算。当执行减法运算时，可利用  $a-b=a+(-b)$  补，将 y 操作数转化为其相反数的补码即可。求一个数的相反数的补码的方法是其各位取反并加 1（加 1 可通过进位实现），这样在 y 操作数上加一个多路选择器控制是自身还是相反数的补码参与运算，同时控制加法器的进位，即可在一个加法器上同时实现加法和减法运算。有符号溢出标志位  $OF=XfYf\sim Sf + \sim Xf\sim YfSf$ ，其中  $Xf$  表示操作数  $x$  的符号位， $Yf$  表示操作数  $y$  的符号位， $Sf$  表示结果的符号位。无符号溢出标志位  $UOF=sub \oplus cout$ ，其中  $sub$  表示是否为减法（为

# 华中科技大学课程实验报告

1 表示减法, 为 0 表示加法), cout 表示最终的进位。加减法的运算电路如图 1.6 所示。

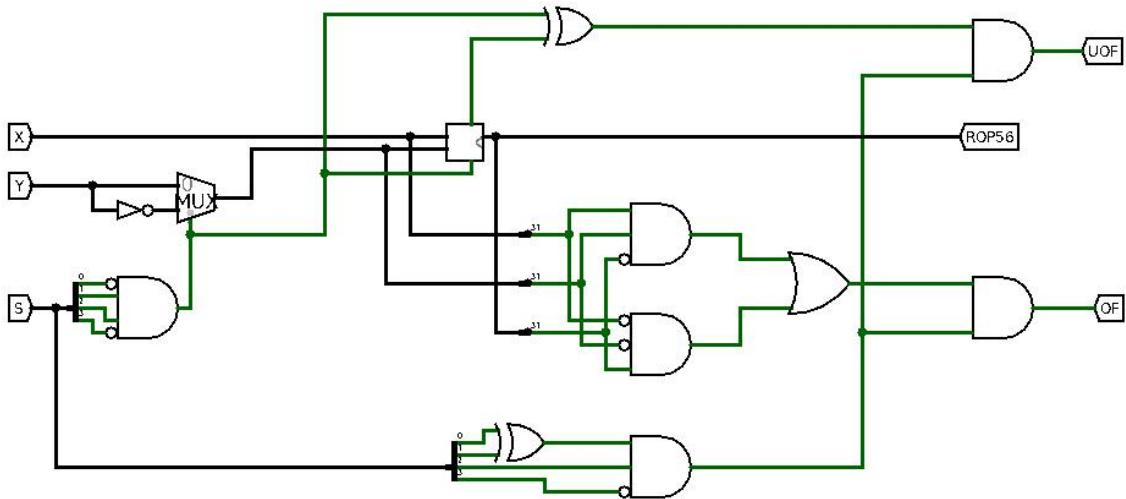


图 1.6 加法减法操作运算电路

第二部分主要是两个多路选择器用来选择 result 和 result2 的来源，其中隧道“ROPi”表示操作码为 i 的运算结果。result 和 result2 的多路选择电路如图 1.7 所示。

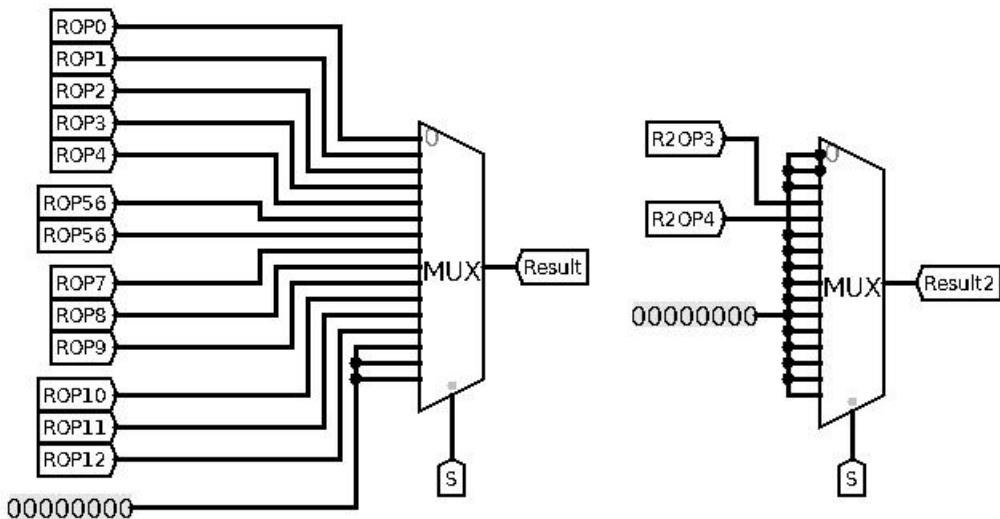


图 1.6 加法减法操作运算电路

对于 result2，只有当 op 操作码为 3 或 4 时，result2 才有有效值，其余情况均为 0；对于 result1，由于没有规定 13~15 号操作码，因此其对应的 result 也应该都为 0 才对。

## 1.3 实验步骤

- (1) 仔细阅读本次实验的对应的课本内容和上课所讲的 ppt, 理解实验原理。在利用 logisim 绘制实验电路之前, 仔细查看了课本第 3 章运算方法与运算器中关于先行进位加法器、先行进位电路、溢出检测和 ALU 设计的相关内容, 并结合谭老师 ppt 中的动画, 学习并掌握了先行进位的原理, 弄清楚了 P、G、P\*、G\*的具体含义, 还搞清楚了 ALU 的整体架构。
- (2) 利用 logisim 实现 32 位组内先行进位, 组间先行进位的加法器。首先, 应现实现一个具有先行进位特征的 4 位加法器, 然后实现一个类似 74182 的先行进位电路, 接着利用该先行进位电路连接 4 个 4 位组内先行进位加法器从而实现 16 位的组内先行进位、组间先行进位的加法器, 最后利用两种不同的连接方式连接 16 位加法器, 得到两种设计方案的 32 位的组内先行进位、组间先行进位的加法器。
- (3) 利用 logisim 和步骤 (2) 实现的加法器, 实现一个支持 13 种操作的 32 位 CPU。首先, 在步骤 (2) 实现的加法器的基础上, 实现一个加减法运算器, 该加减法运算器既可以进行加法运算也可以进行减法运算, 同时利用溢出检测的方法得到有符号溢出标志位和无符号溢出标志位; 然后, 利用 logisim 自带运算器实现剩余 11 种操作的运算电路, 并且利用比较器得到操作数相等标志位 Equal; 最后, 利用两个多路选择器根据操作码选择 result 和 result2, 最终得到 ALU。

## 1.4 故障与调试

### 1.4.1 接口处数据传输问题

**故障现象:** adder 电路连接完成后, 发现电路中存在三根红线 (错误) 和两根蓝线 (未连接), 导致最后的运算结果始终为一个莫名其妙的错误值。故障现象截图如图 1.7 所示。

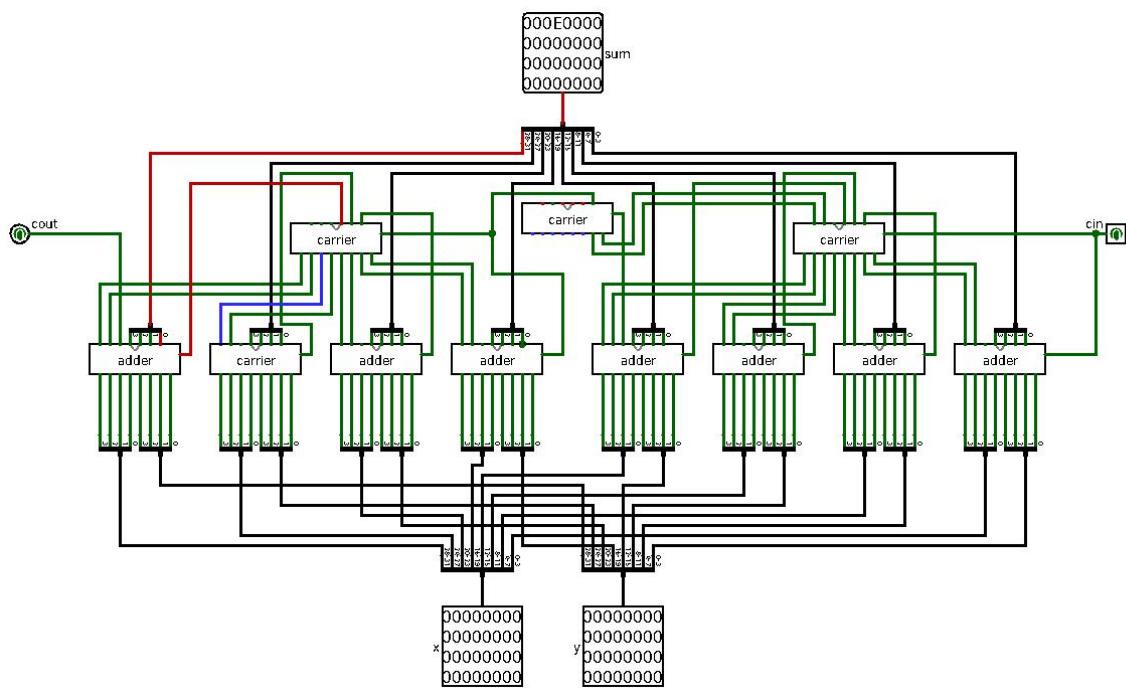


图 1.6 实验 1 故障 1 现象截图

**原因分析：**如图 1.6，仔细观察故障现象可以发现，红线（错误）的造成应该是蓝线（未连接）导致。蓝线说明此根电线没有输入端，进一步发现芯片在电线的连接位置确实没有端口，再看芯片的名字才发现，误把 carrier 芯片当作了 adder 芯片。

**解决方案：**将添加错的 carrier 芯片换成应有的 adder 芯片即可，解决问题（连线时一定要细心啊）。

## 1.4.2 故障 2

**故障现象：**在 logisim 里，大规模拖动电路会造成 logisim 死机或者拖动后的电路存在大量红线（错误连线）和蓝线（未连接）。

**原因分析：**logisim 的自身问题，logisim 基于 java 虚拟机实现，当大规模或者大范围拖动电路时，容易造成短时间内运算量过大，Java 虚拟机有可能因此崩溃，所以造成 logisim 死机。

**解决方案：**logisim 死机时“Ctrl+s”经测试还是有反应的，所以先按下“Ctrl+s”保存已经画好的电路，然后重启 logisim 即可。当大规模或者大范围拖动电路后出现

# 华中科技大学课程实验报告

大量的红线和蓝线也是一样的处理方案，先保存电路方案，然后重启 logisim，如果电路方案设计正确，重启 logisim 应该不再有红线或者蓝线。如果仍有红线和蓝线，说明应该是电路设计有问题。

## 1.5 测试与分析

ALU 测试用例见表 1.3。

表 1.3 ALU 测试用例

#	A	B	F	result	result2	OF	UOF	Equal
1	fffff0000	0ffff0000	5	0ffe0000	00000000	○	●	○
2	f00000000	900000000	5	800000000	000000000	○	●	○
3	70000000	400000000	5	b00000000	000000000	●	○	○
4	800000000	c00000000	5	400000000	000000000	●	●	○
5	fffffff	00000001	5	00000000	000000000	○	●	○
6	00000001	00000002	6	fffffff	000000000	○	●	○
7	0000000f	0000000f	6	00000000	000000000	○	○	●
8	efffffff	ffffffff	6	f0000000	000000000	○	●	○
9	7fffffff	ffffffff	6	80000000	000000000	●	●	○
10	7fffffff	00000000	6	7fffffff	000000000	○	○	○
11	00000000	80000000	6	80000000	000000000	●	●	○
12	80000000	00000001	6	7fffffff	000000000	●	○	○
13	5fffffff	6fffffff	6	f0000000	000000000	○	●	○
14	fffffff	80000000	6	7fffffff	000000000	○	○	○
15	00000005	00000f09	0	00000a00	000000000	○	○	○
16	f0000005	00000004	0	00000050	000000000	○	○	○
17	f0000009	00000004	1	ff000000	000000000	○	○	○
18	7fffffff	00000f1f	1	00000000	000000000	○	○	○
19	fffffff	0000001f	2	00000001	000000000	○	○	○
20	7fffffff	00000000	2	7fffffff	000000000	○	○	○

# 华中科技大学课程实验报告

表 1.4 ALU 测试用例 (续)

#	A	B	F	result	result2	OF	UOF	Equal
21	7fffffff	7fffffff	3	00000001	3fffffff	○	○	●
22	fffffff	fffffff	3	00000001	00000000	○	○	●
23	00000000	fffffff	3	00000000	00000000	○	○	○
24	fffffff	fffffff	4	00000001	00000000	○	○	●
25	0000009f	00000008	4	00000013	00000007	○	○	○
26	fffffff	00000aab	4	0017ff40	0000003f	○	○	○
27	f0000005	2000000a	7	20000000	00000000	○	○	○
28	f0000001	0000000e	8	f000000f	00000000	○	○	○
29	b0000005	7000000a	9	c000000f	00000000	○	○	○
30	b0000005	7000000a	a	0fffff0	00000000	○	○	○
31	fffffff	7fffffff	b	00000001	00000000	○	○	○
32	00000001	00000001	b	00000000	00000000	○	○	●
33	fffffff	7fffffff	c	00000000	00000000	○	○	○

在进行实际测试时，需要对老师提供的测试程序“alu-test.circ”进行适当的改进。老师提供的测试电路并不易观察结果，所以额外添加 5 个结果 ROM，分别记录提前手动计算好的每个测试用例对应的 result、result1、OF、UOF 和 Equal。这样，在测试每个测试用例时，将 ALU 计算得到的 result、result1、OF、UOF 和 Equal 分别与对应 ROM 储存的结果进行比较（比较采用比较器），如果相同比较器的结果应该都为 1，否则为 0。这样，在进行测试时，就没有必要用人眼一个一个的比对 ALU 的输出与正确结果，只需观测比较器的输出结果是否都 1 即可。如果比较器的比较结果始终都为 1，说明上述测试用例在 ALU 上的执行结果都是正确的，一般情况下足够说明 ALU 的设计是正确的。否则，着重处理比较器结果输出为 0 时的测试用例和对应出错的地方即可。

本设计中的 ALU 针对上述测试用例在测试电路下的运行过程如图 1.7 所示，目前正在测试第 6 个测试用例。

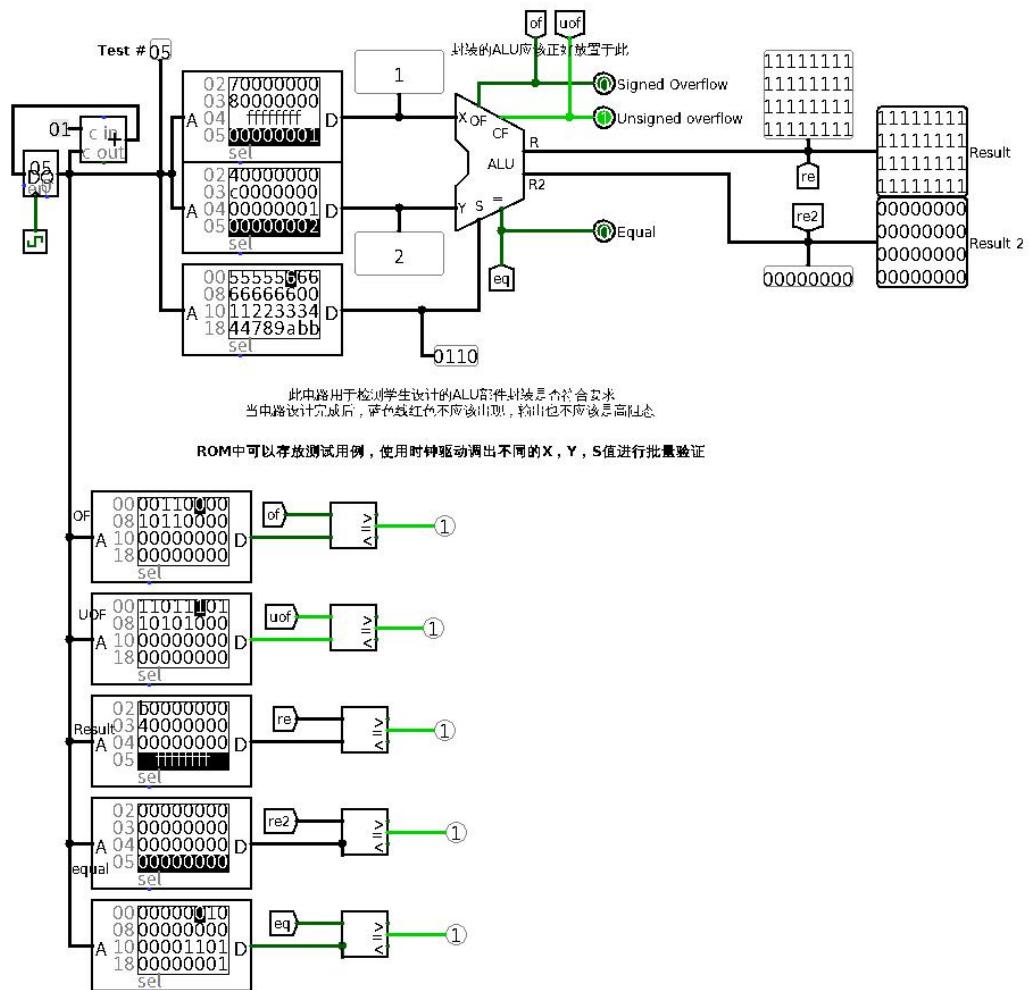


图 1.7 测试电路运行过程截图

经上述测试, 本设计的 ALU 在测试过程中各比较器的比较结果始终为 1, 因此基本可以确定本 ALU 的各个功能是正确的。

## 2 存储器实验

### 2.1 设计要求

(1) 利用 logisim 平台构建一个 MIPS 寄存器组，内部包含 32 个 32 位寄存器寄存器组输入输出引脚及功能描述见下表，在主电路中详细测试自己封装的寄存器组。注意零号寄存器值应该恒零。

表 2.1 寄存器组引脚与功能描述

引脚	输入/输出	位宽	功能描述
R1#	输入	5	读寄存器 1 编号
R2#	输入	5	读寄存器 2 编号
W#	输入	5	写入寄存器编号
Din	输入	32	写入数据
WE	输入	1	写入使能信号，为 1 时， CLK 上跳沿将 Din 数据写入 W#寄存器
CLK	输入	1	时钟信号，上跳沿有效
R1	输出	32	R1#寄存器的值
R2	输出	32	R2#寄存器的值
\$s0	输出	32	编号为 16 的寄存器的值

# 华中科技大学课程实验报告

\$s1	输出	32	编号为 17 的寄存器的值
\$s2	输出	32	编号为 18 的寄存器的值
\$ra	输出	32	编号为 31 的寄存器的值

(2) 利用 (1) 封装好的运算器, 以及 RAM 模块, 封装好的 MIPS 寄存器文件, 计数器等 logisim 模块构建一个自动运算电路, 该电路由时钟驱动, 可自动完成 RAM 模块 (32\*32 位) 0-31 号单元的累加, 并将累加的中间结果回存到同一 RAM 模块 32-63 号单元。主电路最上面一行请将所有关键点的值用探测和隧道方式结合引出, 用 10 进制方式显示, 便于检查, 运算器结果直接用 16 进制数码管显示。

## 2.2 方案设计

### 2.2.1 寄存器组 RegFile 电路

根据实验要求, 寄存器组可以分为两大部分:

第一部分就是 32 个 32 位的寄存器, 这 32 个寄存器按顺序依次编号, 从 0 号到 31 号寄存器。需要注意的是, 由于零号寄存器要求值恒为 0, 因此可以将 0 号寄存器用一个常量 “0” 代替, 但是在本方案设计中, 为了追求美观和对称性, 在 0 号寄存器的位置仍然使用了一个 32 位的寄存器, 只不过该寄存器的清空位恒为 1 来保证其值恒为 0。除此之外, 寄存器组应该为一个同步时序逻辑电路, 所以所有的 32 个寄存器应该共用同一个时钟信号。

第二部分是控制部分, 该部分控制选择写入哪个寄存器, 读出哪两个寄存器的内容。控制部分又可划分为两大部分, 写控制部分和读控制部分。

(1) 写控制部分, 应使用一个 5-32 译码器实现, 将写入的寄存器编号 “write\_reg1” 经过译码转为 32 位的寄存器写使能控制信号, 分别按顺序依次连接到 0 号~31 号寄存器的写使能段。寄存器组中的 32 个寄存器应该并发工作, 所以应将待写入的数据 “write\_data” 连接到所有寄存器的数据端。根据译码器的电路功能和特性, 显然这种设计方式是符合设计要求的。还需要注意的一点是, 寄存器组还有一个写使能端

“WE”，该端口应该连接到译码器的使能端，并设置译码器禁用（使能端为 0）时输出 0。

(2) 读控制部分，应使用两个（寄存器组支持同时读两个寄存器的值）多路选择器实现，将多路选择器的输出结果分别作为“reg1#”和“reg2”。多路选择器的数据位宽应选择 32 位，因为每个寄存器的输出（当前寄存器存储的内容）都是 32 位；多路选择器的选择位位宽应为 5 位，这样才能从  $2^5=32$  个数据输入端中根据选择位选择输出。显然，多路选择器的 32 个输入应按顺序依次连接 0~31 号寄存器，多路选择器的选择位应分别连接读出寄存器编号“reg1”和“reg2”。

此外，根据设计要求，16、17、18 和 31 号寄存器的值应该单独用引脚引出，分别记为“\$s0”、“\$s1”“\$s2”和“\$ra”。

综上所述，可以画出寄存器组电路 RegFile 的电路图如图 2.1 所示。

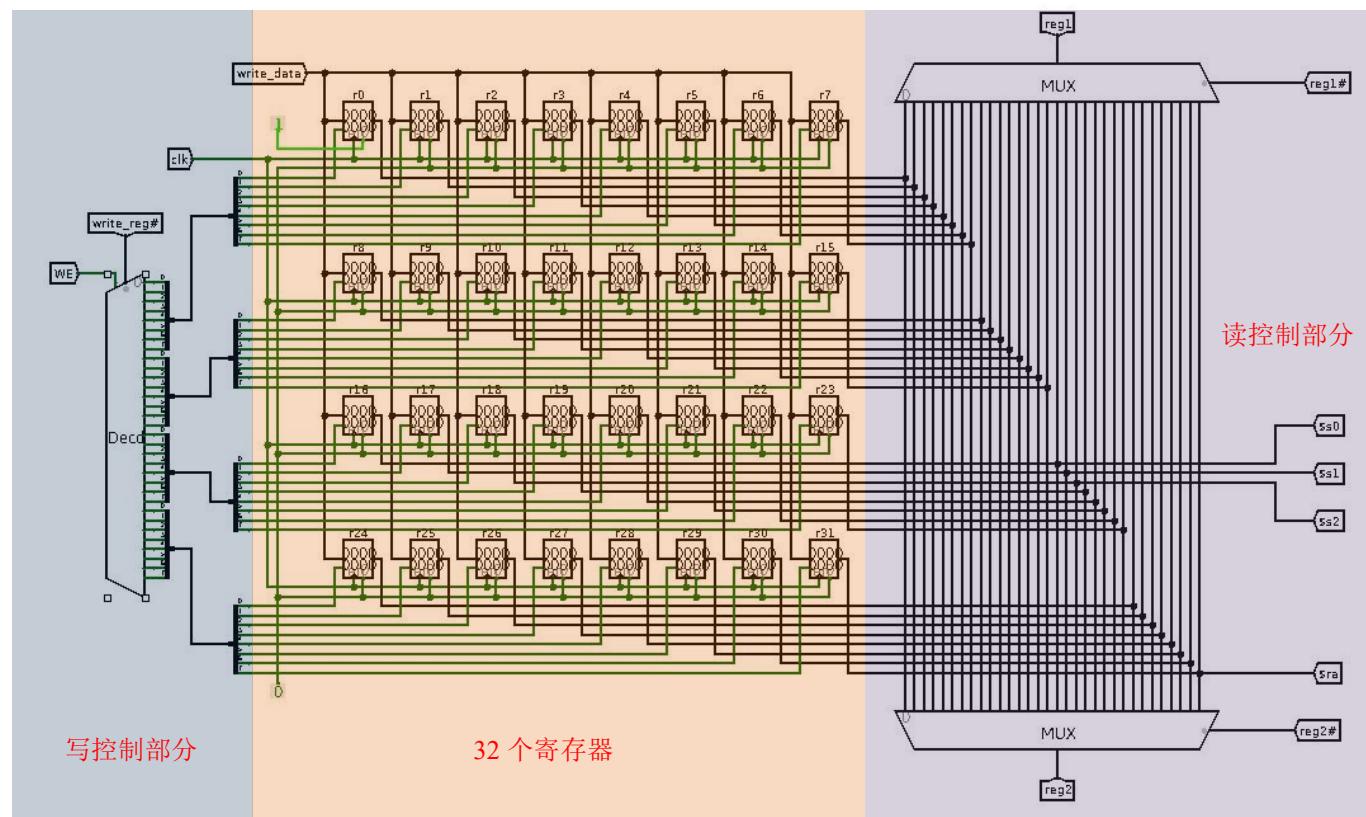


图 2.1 寄存器组 RegFile 电路

## 2.2.2 自动运算电路

根据题目要求，该自动运算电路可看作一简单的小型处理器，该处理器受时钟

# 华中科技大学课程实验报告

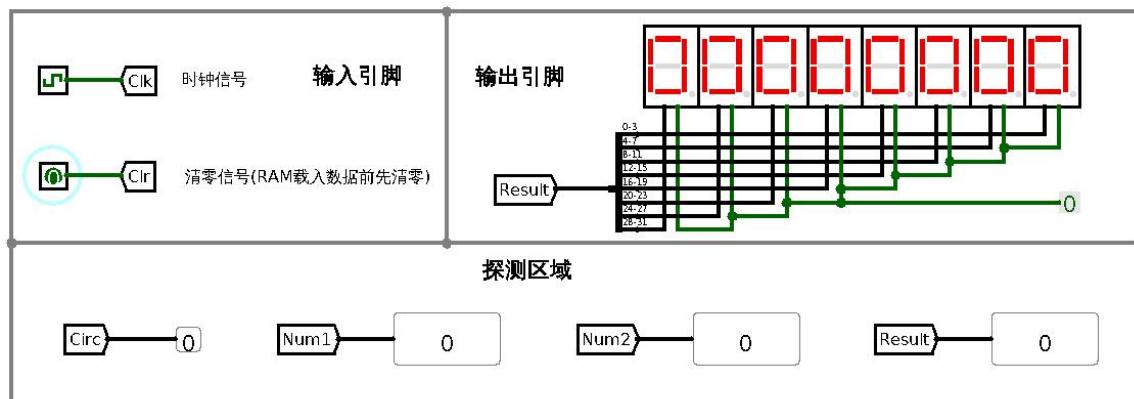
驱动，且能够自动完成从 RAM 中读数据、从 RegFile 中读数据、利用 ALU 运算、向 RAM 写入数据和向 RegFile 中写入数据的功能。

首先，考虑电路的主要部件。显然，根据题目要求，可以得到电路中一定有部件寄存器组 RegFile、运算器 ALU 和存储器 RAM。除此之外，考虑到访问 RAM 的地址在随时间变化，因此还需要有计数器；考虑到写入 RegFile 的多个数据来源（ALU 的运算结果和 RAM 中读出的数据），因此还需要二路选择器；当然还需要其他的的基本门电路。虽然本次设计用到了 RegFile，但实际上只需要使用其中的两个寄存器（分别存储 ALU 的两个操作数）即可，这里使用的是 1 号寄存器和 2 号寄存器。

然后，考虑电路的数据通路。结合题目要求与上一步的讨论，数据通路不难得出，即从 RAM 中读取的数据送 RegFile 的数据端（二路选择器），从 RegFile 中读出的两个寄存器的值分别送 ALU 的两个操作数，然后 ALU 的运算结果也送至 RegFile 的数据端（二路选择器），同时送至 RAM 的数据段。

接下来，考虑电路的控制部分。由于每次运算周期，需要向 RAM 中写入数据一次，向 RegFile 中写入数据两次，因此每个运算周期至少需要两个时钟周期。其中，第一个时钟周期把从 RAM 中读取的数据写入 RegFile 的 1 号寄存器，第二个时钟周期把 ALU 的运算结果写入 RAM 的高地址处和 RegFile 的 2 号寄存器。除此之外，还有一个需要注意的地方，那就是为了保证执行完 32 个运算周期后 ALU 的运算结果仍然是正确的，所以在第 32 个运算周期的第 2 个时钟周期里，ALU 的运算结果就不能再写入 RegFile 的 2 号寄存器了。

最后,根据上述讨论和思考,不难在 logisim 下实现自动运算电路,其电路图如图 2.2 所示。



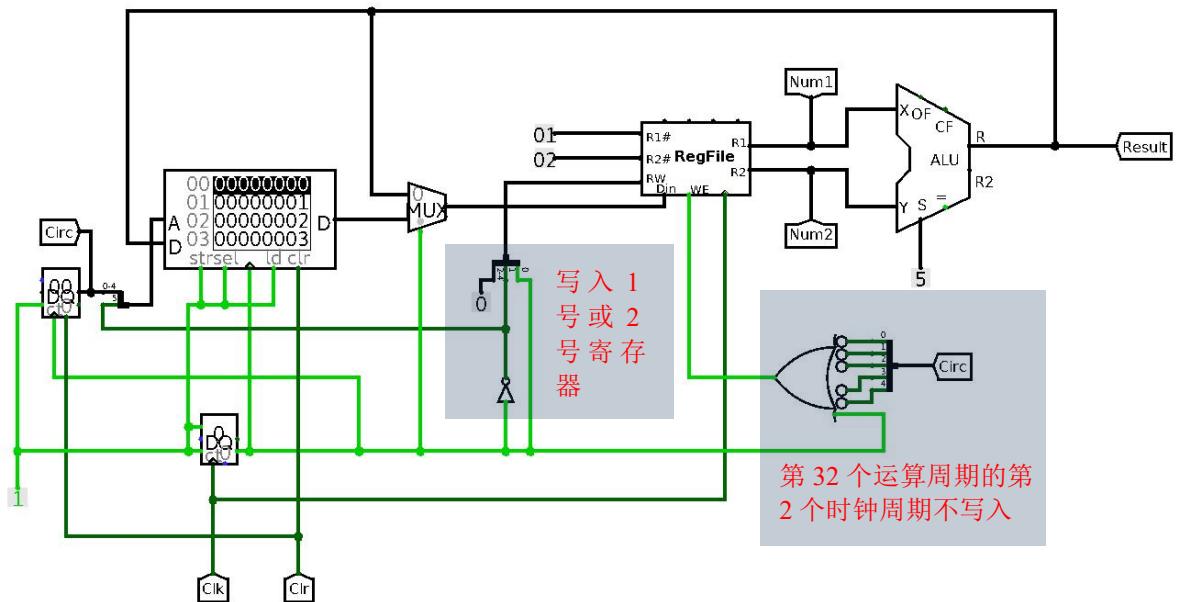


图 2.2 自动运算电路

### 2.3 实验步骤

- (1) 仔细阅读本次实验的对应的课本内容和上课所讲的 ppt, 理解实验原理。在利用 logisim 绘制实验电路之前, 仔细查看了此次的任务书, 并上网查阅了寄存器组的相关资料, 然后意识到这是在为第 3 次的 CPU 实验做准备。寄存器组是 CPU 内部的一个重要组成部分, 对于 MIPS 架构的 CPU 来说, 寄存器组中的 32 个寄存器每一个都有其特定的用途。
- (2) 利用 logisim 实现一个包含 32 个 32 位寄存器的寄存器组, 该寄存器组同时读取两个指定寄存器的值, 可在时钟驱动下向指定寄存器写入数据。总的来说, 寄存器组的实现逻辑较为简单, 稍微思考下, 就能大致想到寄存器组电路的整体结构, 只是寄存器组的连线相对复杂一些, 因为有 32 个寄存器需要连接, 它们的使能端、时钟端、数据段、输出端都需要连线。寄存器组的具体逻辑和电路在设计方案中已经详细阐述了, 故这里不再赘述。
- (3) 利用 logisim, 利用步骤 (2) 实现的寄存器组电路 RegFile、实验 1 实现的运算器电路 ALU 以及 logisim 自带存储器 RAM, 实现一个可以自动运算 RAM 的 0~31 号地址所存数据的累加和, 并能把计算的中间结果存储到

RAM 的 32~63 号地址区域中。首先，考虑该自动运算电路所需要的主要部件有哪些，除了题目涉及到的部件，还需要考虑为了实现所需功能所必须的其他部件；然后，考虑电路的数据通路，根据题目要求和读写关系，设计出合适的数据通路；最后考虑电路的控制部分如何实现，要考虑电路的每个运算周期需要几个时钟周期，考虑每个时钟周期电路需要完成什么任务。

## 2.4 故障与调试

### 2.4.1 接口处数据传输问题

**故障现象：**自动运算电路在前面的计算都是正确的，而且也能够把正确的中间结果写入到 RAM 的高地址区，但是 32 个运算周期后，显示出的计算结果错误。

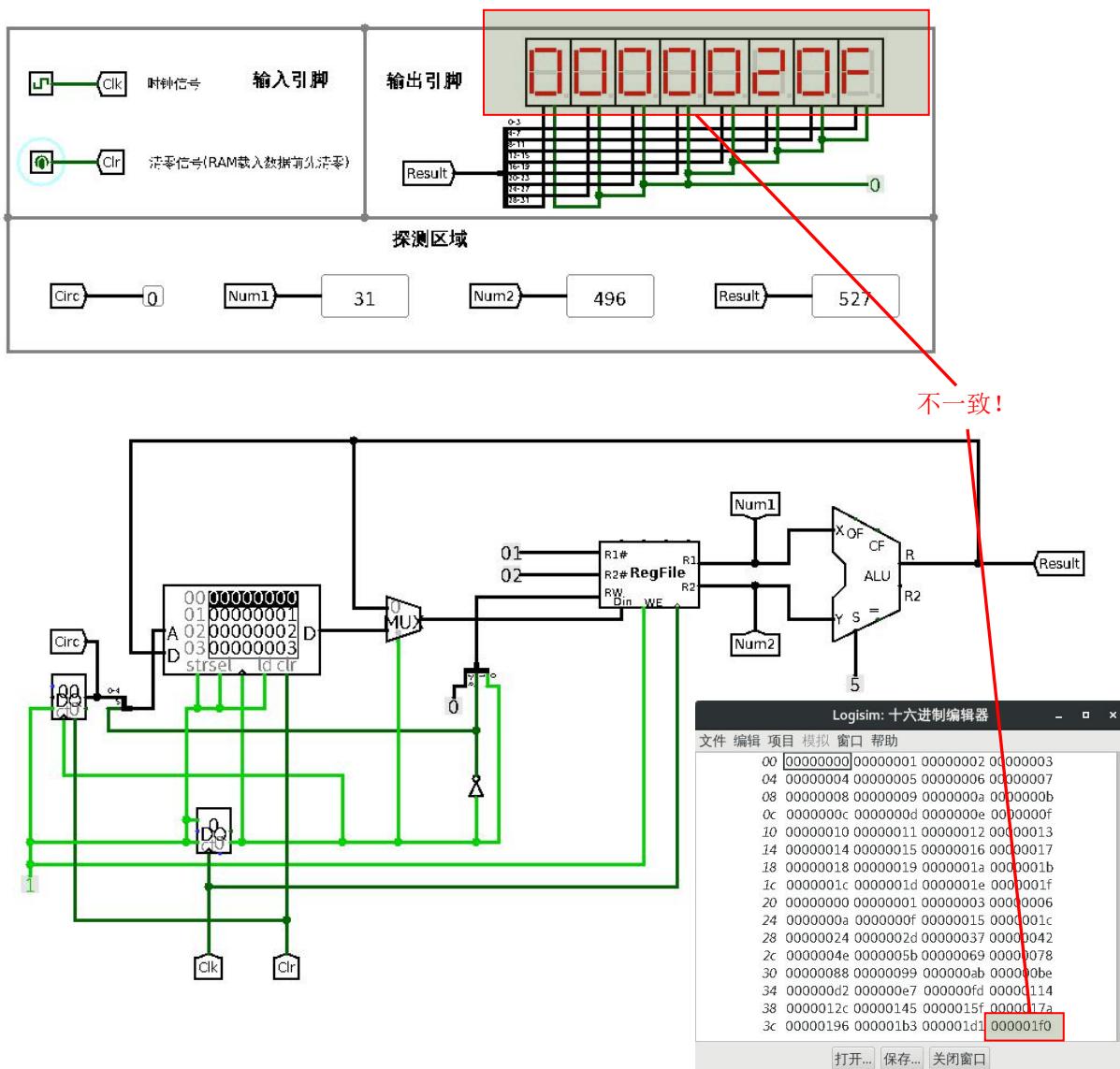


图 2.3 自动运算电路故障 1 运行截图

**原因分析：**如错误！未找到引用源。, 在 ROM 的高地址区, 计算的中间结果以此为“0、1、3...”, 最终的运算结果为  $0x1f=496$  也是正确的, 但是此时 ALU 的运算结果却显示为 527。进一步观察发现, 结束时的操作数 2 与结果相同为 496, 这说明最后一个运算周期里最终的计算结果也写入了 RegFile 的 2 号寄存器。这样, 便找到了问题, 在设计方案中, 每个运算周期的第 2 个时钟周期都会将 ALU 的运算结果写入到 RegFile 的 2 号寄存器, 但在第 32 个运算周期时, 因为不再继续运算, 所以不需要将 ALU 的计算结果写入 2 号寄存器。

**解决方案：**将 RegFile 原本的写使能信号与运算周期数进行逻辑运算, 保证第 32 个运算周期 (运算周期数为 31) 里 RegFile 的写使能信号为 0 即可。

## 2.4.2 故障 2

**故障现象：**按下“Ctrl+R”(logisim 的电路复位快捷键)，计数器的计数值为 1 不为 0，导致复位后从 RAM 的 1 号地址开始读写。故障运行截图如图 2.4 所示。

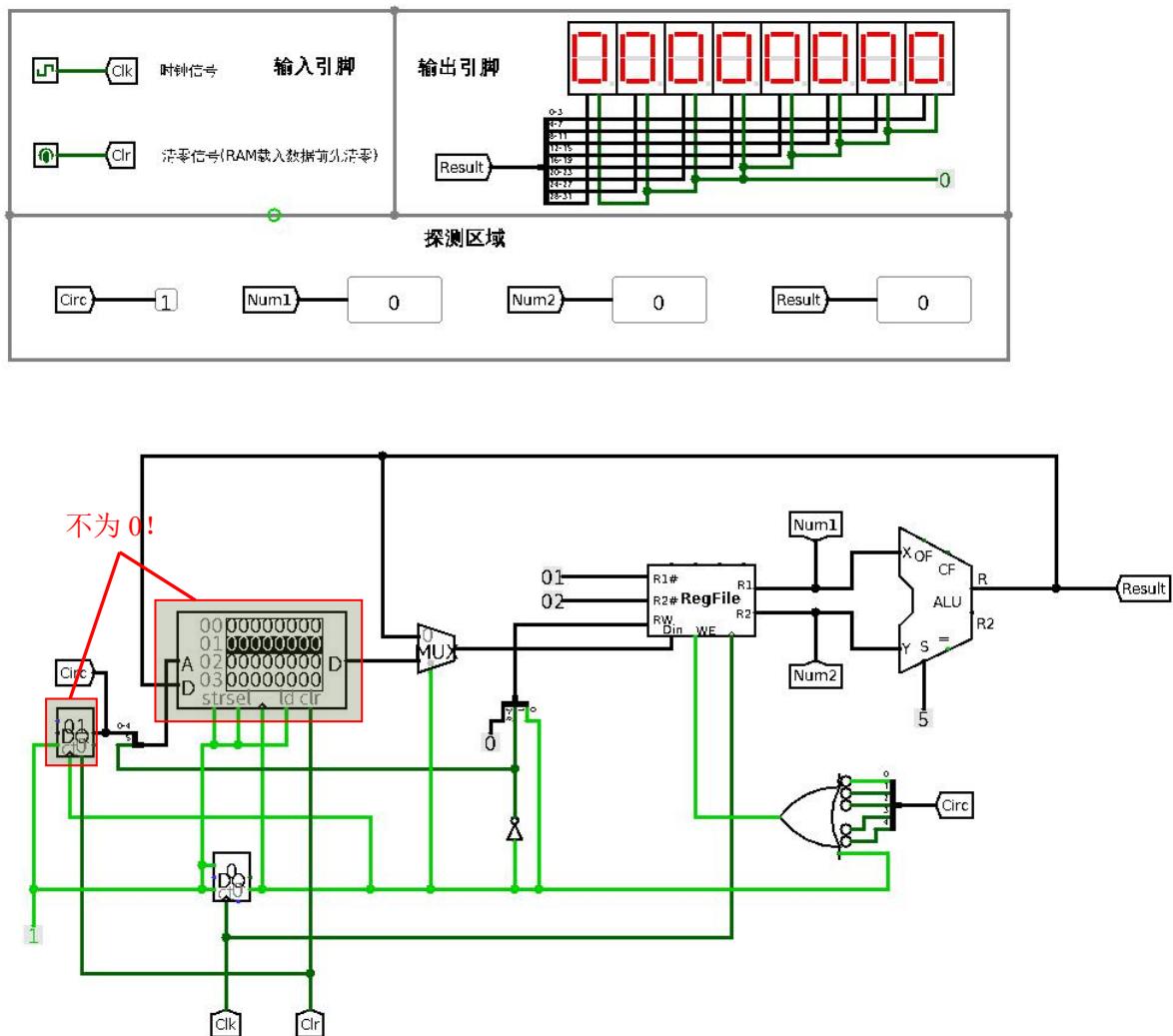


图 2.4 自动运算电路故障 2 运行截图

**原因分析：**如错误！未找到引用源。, 由于控制部分的设计有些不合理，导致一开始时计数器的时钟端值为 1，所以电路复位后，所有的存储器件中的数据均值为 0，但是此时计数器的时钟端会由 0 变成 1，这样就相当于一个时钟上升沿，从而计数器的计数值会加 1，导致了复位后计数器值不为 0，也导致了复位后 RAM 不是从 0 号地址开始读写。

**解决方案：**如错误！未找到引用源。, 添加一个手动清零信号“Clr”。每次向 ROM 中载入数据时，先按“Ctrl+R”进行电路复位，然后给一个 Clr 脉冲，将计数

器清零。

## 2.5 测试与分析

考虑到，自动运算电路是在寄存器组电路 RegFile 的基础上实现的，而且自动运算电路仅用到了 RegFile 的 32 个寄存器中的两个，所以可以按照如下步骤和策略进行测试：

(1) 将自动运算电路中用到的 RegFile 的两个寄存器的编号设为 1 和 2，向 RAM 中写入测试数据，然后启用时钟模拟，测试自动运算电路的正确性，同时也能验证 RegFile 中对 1 号、2 号寄存器的读和写都没有问题；

(2) 修改自动运算电路中用到的 RegFile 的两个寄存器的编号设为 3 和 4，重复步骤 (1)，以此类推，每次利用自动计算电路测试 RegFile 不同的两个寄存器。

在自动运算电路中用到的测试数据，即初始时 ROM 存储的数据如图 2.5 所示。

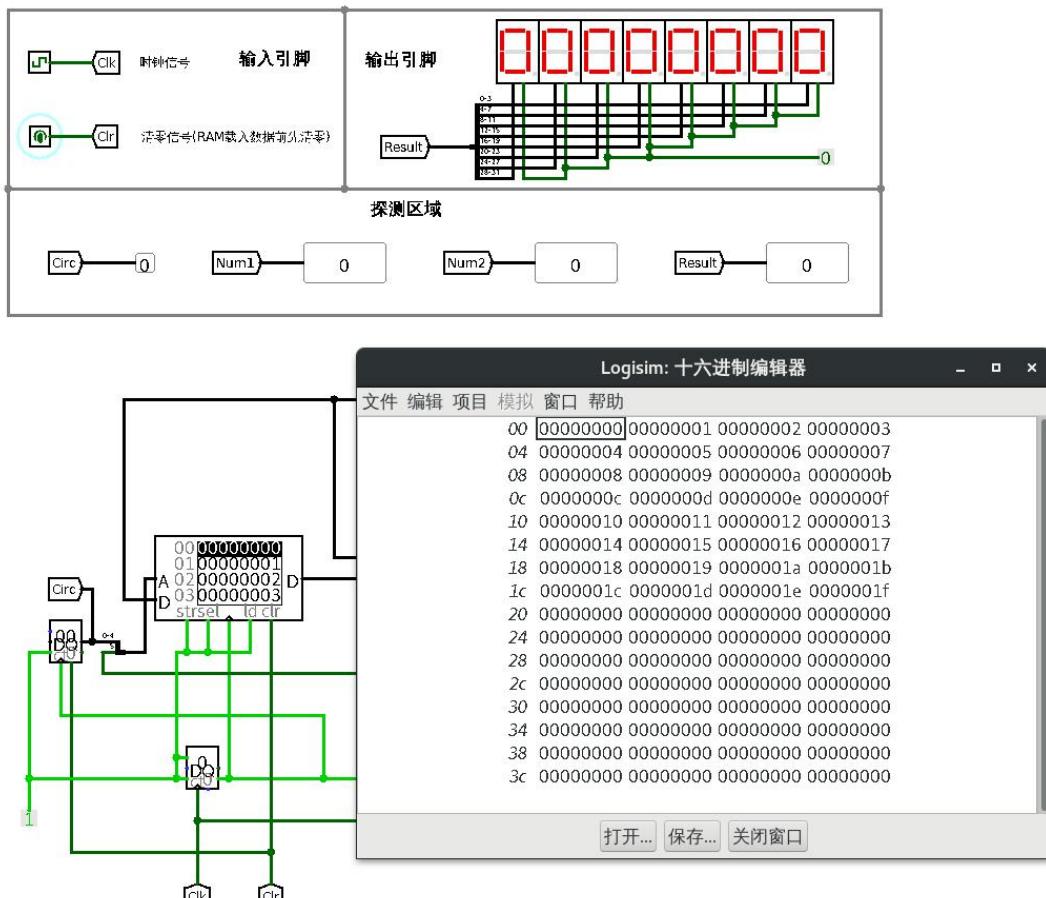


图 2.5 自动运算电路测试数据

不难看出，待测试的数据为一等差数列 $\{0, 1, 2, 3, \dots, 30, 31\}$ ，其累加的最终结果为 496，中间结果依次为 $\{0, 1, 3, 6, \dots, 465, 496\}$ 。

设置使用的 RegFile 中两个寄存器的编号为 1 和 2，启用时钟模拟，执行 32 个运算周期后，运算结果和 ROM 中存储的数据情况如图 2.6 所示。

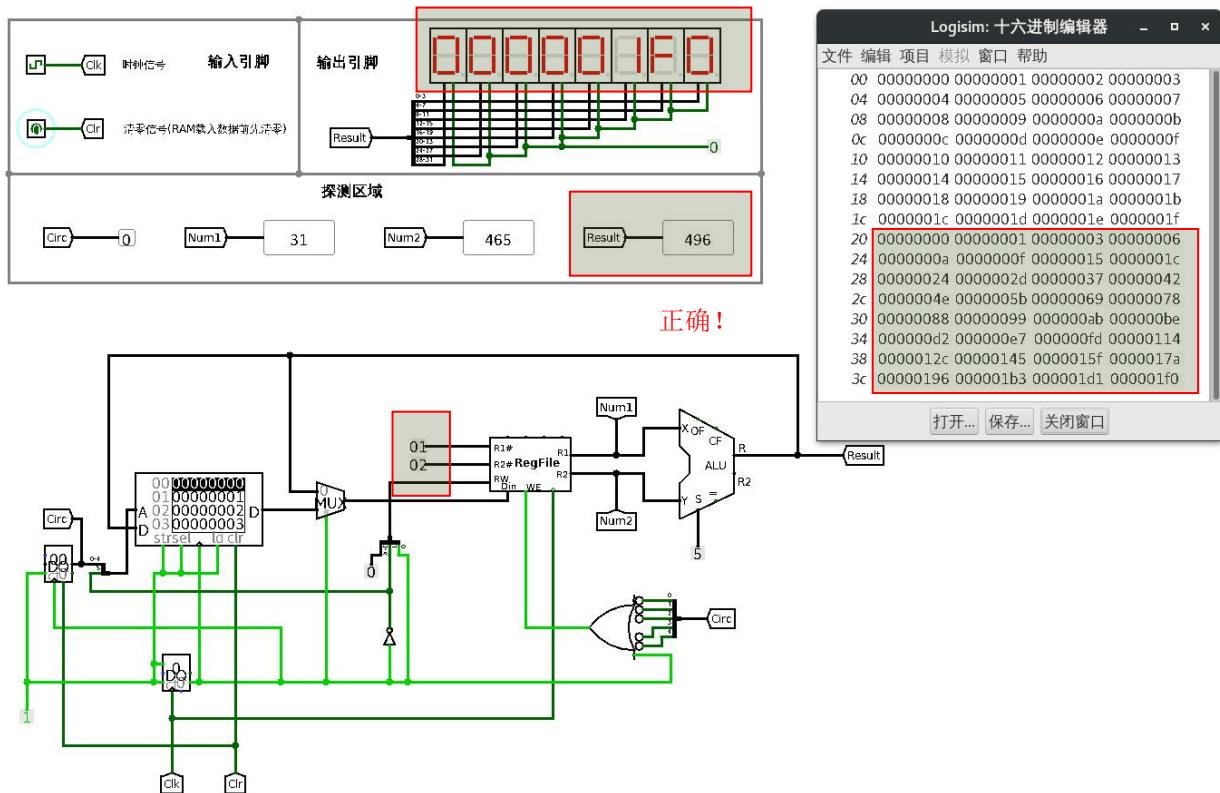


图 2.6 自动运算电路测试结果

通过图 2.6 不难看出，在选择 1 号寄存器和 2 号寄存器的前提下，测试结果正确。这说明，本设计中的自动选择电路是正确的，RegFile 对 1 号和 2 寄存器的读写也是正确的。

接下来，就是改变寄存器编号重复上述测试，由于结果均正确，所以这里不一一给出测试结果截图。

综上，RegFile 电路和自动运算电路的设计和各功能实现均正确无误。

### 3 CPU 实验

#### 3.1 设计要求

利用 logisim 平台, 利用实验 1、2 中构建的运算器, 寄存器文件等部件以及 logisim 中其他功能部件构建一个 32 位 MIPS CPU 处理器, 该处理器应支持表 3.1 中所述指令, 具体指令功能参见附件中的 MIPS 标准文档。最终设计完成的 CPU 能运行教师提供的标准测试程序, 程序存储在 logisim ROM 模块中 (指令存储器、数据存储器分开)。

表 3.1 指令格式

#	指令	格式	备注
1	Add	add \$rd, \$rs, \$rt	
2	Add Immediate	addi \$rt, \$rs, immediate	
3	Add Immediate Unsigned	addiu \$rt, \$rs, immediate	
4	Add Unsigned	addu \$rd, \$rs, \$rt	指令功能及指
5	And	and \$rd, \$rs, \$rt	令格式
6	And Immediate	andi \$rt, \$rs, immediate	参考 MIPS32 指令集
7	Shift Left Logical	sll \$rd, \$rt, shamt	
8	Shift Right Arithmetic	sra \$rd, \$rt, shamt	
9	Shift Right Logical	srl \$rd, \$rt, shamt	

# 华中科技大学课程实验报告

---

10	Sub	sub \$rd, \$rs, \$rt
11	Or	or \$rd, \$rs, \$rt
12	Or Immediate	ori \$rt, \$rs, immediate
13	Nor	nor \$rd, \$rs, \$rt
14	Load Word	lw \$rt, offset(\$rs)
15	Store Word	sw \$rt, offset(\$rs)
16	Branch on Equal	beq \$rs, \$rt, label

表 3.2 指令格式 (续)

#	指令	格式	备注
17	Branch on Not Equal	bne \$rs, \$rt, label	
18	Set Less Than	slt \$rd, \$rs, \$rt	
19	Set Less Than Immediate	slti \$rt, \$rs, immediate	
20	Set Less Than Unsigned	sltu \$rd, \$rs, \$rt	
21	Jump	j label	
22	Jump and Link	jal label	
23	Jump Register	jr \$rs	
24	syscall (display or exit)	syscall	If \$v0==10 halt(停机指令) else 数码管显示\$a0 值

## 3.2 方案设计

### 3.2.1 取指令通路

首先需要注意的是，这里给出的取指令通路并不是最终 CPU 中的取指令通路，只是一个简化版，PC 每次只会加 1，从而取出程序存储器中的下一条指令。该简化版的取指令通路如图 3.1 所示。

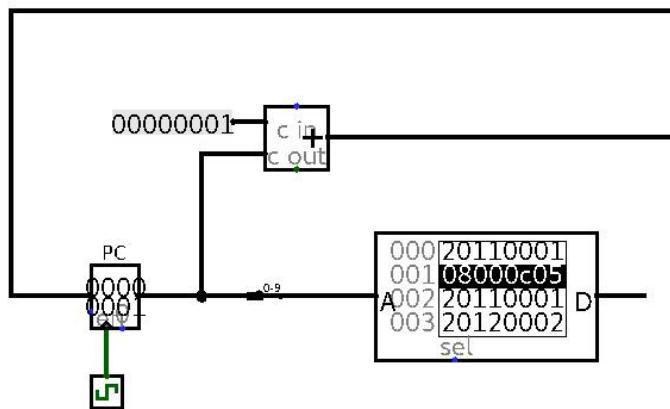


图 3.1 取指令通路简化版

通过该简化版指令通路，设计者需要考虑几个问题。第一，程序存储器采用的是只读 ROM，其数据位宽为 32 位，但是地址为宽仅为 10 位，另一方面 PC 寄存器却是一个 32 位寄存器，所以如何处理 ROM 的 10 位地址线与 PC 寄存器的 32 位数据？本设计方案采取了任务书中的建议，即访问 ROM 时只取 PC 的低 10 位作为 10 位地址（访问数据 ROM 也采用同样的策略）；第二，程序存储 ROM 是按字（4 字节）访问，所以不像一般的 MIPS 架构的 CPU 那样 PC 每次加 4，PC 在这里需要每次加 1，而且除此之外，许多跟地址跳转有关的指令的具体操作也跟传统 MIPS 体系 CPU 的操作有所不同。

### 3.2.2 指令解析电路

指令解析电路，顾名思义，是对指令进行解析，提取有效信息并传递给后续电路作进一步处理，相当于流水线中的译码阶段。

指令解析电路的输入应该是取指令通路的输出，即取指令电路将指令从指令存储器中取出交给指令解析电路进行解析和译码。指令解析电路的输出则有多方面的用途：一方面，控制器需要指令解析的结果中的 funct 和 op 来判断当前指令类型，

从而生成对应的控点信息；另一方面，访问寄存器组中某个寄存器也需要指令解析结果中的源寄存器或目的寄存器编号；除此之外，对于跳转指令而言，跳转的目标地址也需要指令解析结果里的一部分字段才能计算出来。

MIPS 指令（仅考虑此次实验所涉及的）可以分为三大类：R 型指令、I 型指令和 J 型指令，这三类指令的指令格式各不相同，但每类指令所包含的指令的指令格式却是相近的。对于 R 型指令的机器码而言，从高到低依次为——6 位 OP 码、5 位 Rs 寄存器序号、5 位 Rt 寄存器序号、5 位 Rd 寄存器序号、5 位 Shamt 和 6 位 funct 编码。对于 I 型指令而言，从高到低依次为——6 位 OP 码、5 位 Rs 寄存器序号、5 位 Rt 寄存器序号和 16 位立即数。对于 J 型指令而言，从高到低依次为——6 位 OP 码和 26 位立即数。

根据 MIPS 指令的特点，不难看出，所谓的指令解析电路，实际上就是利用分离器将指令中的一部分按照一定的含义提取出来，并不太需要什么太复杂的逻辑关系。

综上所述，本设计方案的指令解析电路如图 3.2 所示。

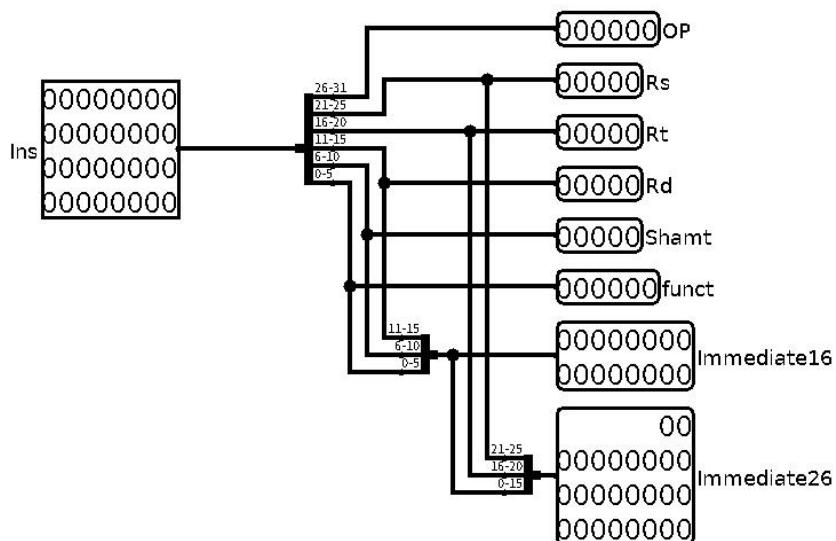


图 3.2 指令解析电路

### 3.2.3 数据通路

首先，构建一个数据通路的第一步是分析在数据通路上要用到哪些部件。如果从宏观向微观考虑，那么大的方面，需要有 3.2.1 的取指令通路，需要有 3.2.2 的指

# 华中科技大学课程实验报告

令解析模块，需要有控制器控制数据通路的选通，需要有存储模块存储各种数据，需要有执行模块来执行指令与访存。往细了思考，取指令通路需要用到 PC 寄存器、程序存储器 ROM、加法器，指令解析模块作为一个芯片直接使用，控制器在 3.2.4 中会详细介绍故这里也是作为一个封装好的芯片直接使用，存储模块需要用到寄存器组和数据存储器 RAM，执行模块需要用到运算器 ALU、有符号扩展、无符号扩展等器件。除此之外，考虑到很多器件的输入源可能有多个，因此还需要用多路选择器来选择输入源。当然，一些基本的门电路像与门、或门、与非门等肯定也是会用到的。

然后，需要考虑具体是哪些器件有多个数据来源，而这些数据来源又具体是什么，因为这决定了哪些器件需要使用多路选择器，而多路选择器的输入来源又如何连接。显然，这些都与指令有关。某些器件之所以需要多个数据来源，是因为在不同的指令执行时，该器件所起到的功能不同。所以，为了解决这一问题，应该事先作出一张表，该表格需要记录执行不同指令时，数据通路上各个器件的数据来源是什么（要弄清楚这一点，必须对指令的作用有正确理解，而且要对指令执行的动作十分熟悉）。在设计中作出的表格如表 3.2 和表 3.3 所示（版面原因分成两张表）

表 3.2 器件数据来源表 1

序号	指令	指令类型	PC	PCAdd		InsMem	RegiFile			
				A	B		R1#	R2#	W#	Din
1	Add	R	PC+1			PC	Rs	Rt	Rd	ALU
2	Add Immediate	I	PC+1			PC	Rs		Rt	ALU
3	Add Immediate Unsigned	I	PC+1			PC	Rs		Rt	ALU
4	Add Unsigned	R	PC+1			PC	Rs	Rt	Rd	ALU
5	And	R	PC+1			PC	Rs	Rt	Rd	ALU
6	And Immediate	I	PC+1			PC	Rs		Rt	ALU
7	Shift Left Logical	R	PC+1			PC	Rt		Rd	ALU
8	Shift Right Arithmetic	R	PC+1			PC	Rt		Rd	ALU
9	Shift Right Logical	R	PC+1			PC	Rt		Rd	ALU
10	Sub	R	PC+1			PC	Rs	Rt	Rd	ALU
11	Or	R	PC+1			PC	Rs	Rt	Rd	ALU
12	Or Immediate	I	PC+1			PC	Rs		Rt	ALU
13	Nor	R	PC+1			PC	Rs	Rt	Rd	ALU
14	Load Word	I	PC+1			PC	Rs		Rt	DM Dout
15	Store Word	I	PC+1			PC	Rs	Rt		
16	Branch On Equal	I	PC+1+extsign/PC+1	PC+1	extsign	PC	Rs	Rt		
17	Branch on Not Equal	I	PC+1/PC+1+extsign	PC+1	extsign	PC	Rs	Rt		
18	Set Less Than	R	PC+1			PC	Rs	Rt	Rd	ALU
19	Set Less Than Immediate	I	PC+1			PC	Rs		Rt	ALU
20	Set Less Than Unsigned	R	PC+1			PC	Rs	Rt	Rd	ALU
21	Jump	J	extzero26			PC				
22	Jump and Link	J	extzero26	PC+1	#0	PC			#31	PCAdd
23	Jump Register	R	RF.D1				Rs			
24	Sys Call	R	PC & Clock							

# 华中科技大学课程实验报告

表 3.3 器件数据来源表 2

序号	指令	SignExt	ZeroExt5	ZeroExt16	ZeroExt26	ALU		DataMem	
						A	B	Addr	Din
1	Add					RF.D1	RF.D2		
2	Add Immediate	IM16				RF.D1	extsign		
3	Add Immediate Unsigned	IM16				RF.D1	extsign		
4	Add Unsigned					RF.D1	RF.D2		
5	And					RF.D1	RF.D2		
6	And Immediate			IM16		RF.D1	extzero16		
7	Shift Left Logical		shamt			RF.D1	extzero5		
8	Shift Right Arithmetic		shamt			RF.D1	extzero5		
9	Shift Right Logical		shamt			RF.D1	extzero5		
10	Sub					RF.D1	RF.D2		
11	Or					RF.D1	RF.D2		
12	Or Immediate			IM16		RF.D1	extzero16		
13	Nor					RF.D1	RF.D2		
14	Load Word	IM16				RF.D1	extsign	ALU	
15	Store Word	IM16				RF.D1	extsign	ALU	RF.D2
16	Bransh On Equal	IM16				RF.D1	RF.D2		
17	Brach on Not Equal	IM16				RF.D1	RF.D2		
18	Set Less Than					RF.D1	RF.D2		
19	Set Less Than Immediate	IM16				RF.D1	extsign		
20	Set Less Than Unsigned					RF.D1	RF.D2		
21	Jump				IM26				
22	Jump and Link				IM26				
23	Jump Register								
24	Sys Call								

根据表 3.2 和表 3.3 可以统计出，每个器件的所有可能数据来源，从而也可以确定在每个器件前需要加一个多大的多路选择器。统计的结果如表 3.4 所示。

表 3.4 器件数据来源汇总表

PC	PCAdd		InsMem	RegiFile				SignExt	ZeroExt5	ZeroExt16	ZeroExt26	ALU	
	A	B		R1#	R2#	W#	Din					A	B
4Mux: (0)RF.D1 (1)PC+1 (2)PCAdd (3)extzero26	PC+1	2Mux: (0)#0 (1)extsign	PC	2Mux: (0)Rs (1)Rt	Rt	3Mux: (0)Rt (1)Rd (2)#31	3Mux: (0)PCA dd (1)DM Dout (2)ALU	IM16	shamt	IM16	IM26	RF.D1	4Mux: (0)RF.D2 (1)extzero5 (2)extsign (3)extzero16

最后，根据上一步作出的器件数据来源统计表，再加上对电路的布局进行合理安排和设计，便可以得到一个简洁、正确、无误的数据通路。实际上，布局的调整是极其必要的，因为整洁的电路布局让人心情愉悦、成就感爆棚，而且也会更容易贴到报告里。本设计中调整布局前数据通路如图 3.3。

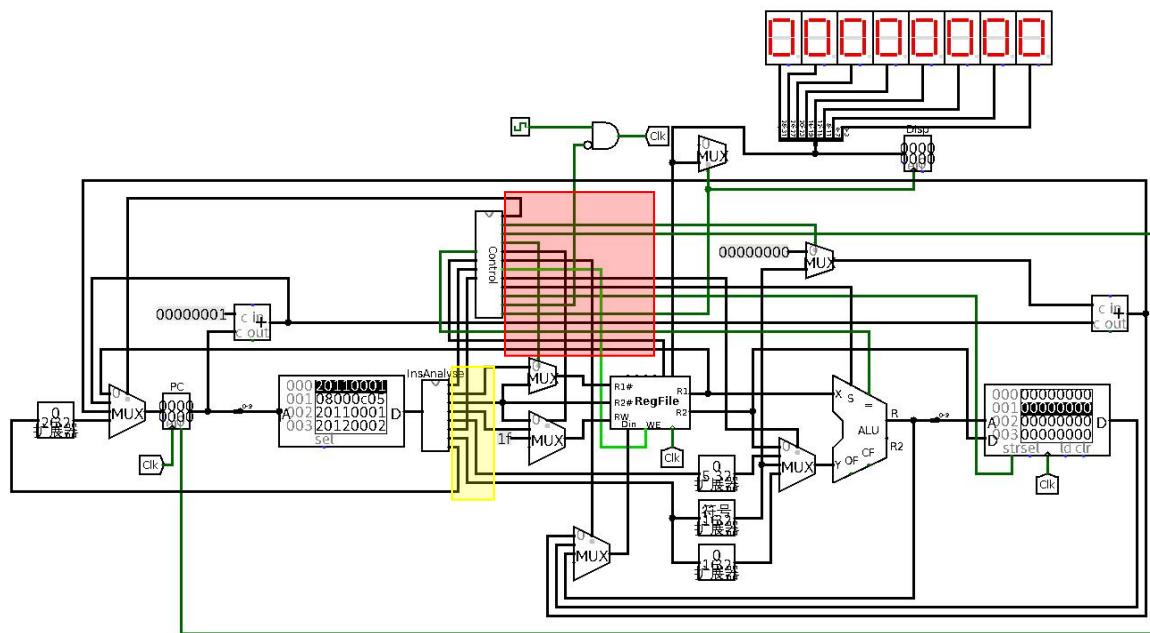


图 3.3 调整布局前数据通路

该数据通路有很多可以优化的地方，其中，红色框区域内线路来回交错，所以使得电路显得十分混乱，该问题可以通过调整控制器和指令解析模块的引脚布局来解决，使线路尽量不交错。

调整布局后，数据通路如图 3.4 所示。

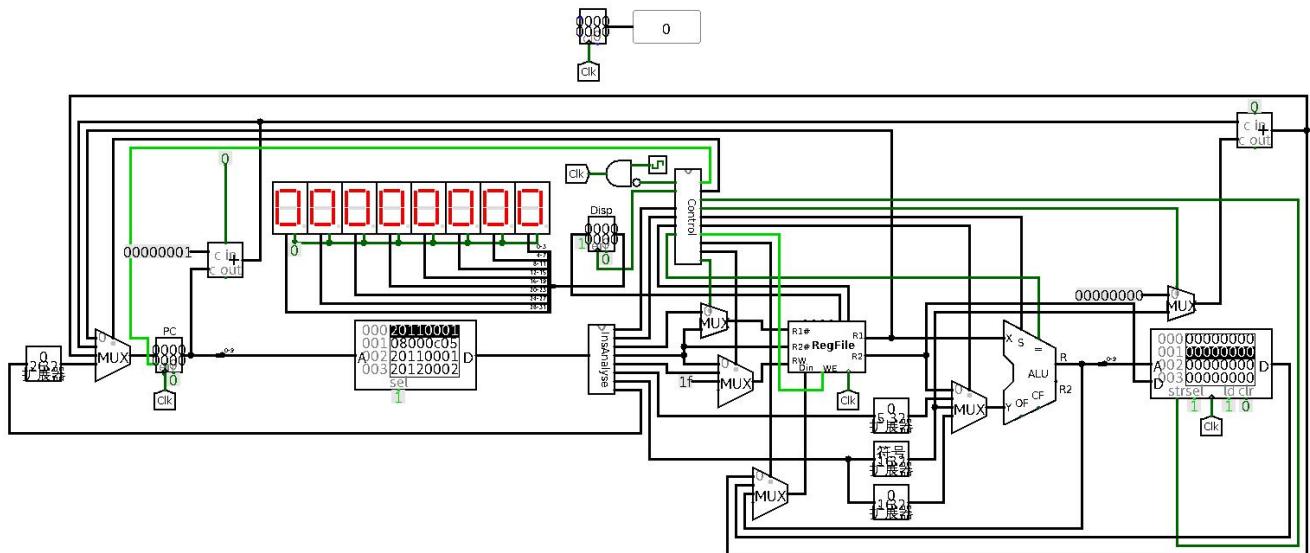


图 3.4 调整布局后数据通路

## 3.2.4 控制器电路

当数据通路设计好后，最后也是最重要的一部分就是要设计控制器。控制器的功能就是要生成数据通路中多路选择器的选择信号、寄存器的写使能端、数据存储器的写使能端、运算器 ALU 的操作码等控制信号，也成为控点信息。显然，控点信息决定了某条指令的数据通路具体是什么样的，而数据通路又是指令正确执行的重要基础，所以控制器的重要地位不言而喻。

关于控制器的设计方法其实有很多种，但都要事先作好表格用以明确控点值与控制器输入的关系，然后根据这张关系表利用 logisim 自动生成或连接电路。在设计控制器时，由于直接考虑的指令和各控点的信息，而不是具体的 OP 码和 func 值，所以在作表时，是直接作的指令和控点信息的关系表。那么在设计电路时，根据所作关系表，控制器需要知道指令信号（当前正在执行的指令的指令信号为 1，其余指令的指令信号均为 0）才能得到控点信息。但是，控制器的输入是指令解析结果的一部分字段，即 op 和 funct 字段然后再加上 ALU 的 Equal 标志位（beq 和 bne 需要用到）、RegFile 中寄存器\$v0 的值（syscall 用到），如何通过这些信息得到对应的指令信号呢？答案是通过译码器，将 op 和 funct 作为译码器的选择信号，就可以得到一系列指令信号，这里需要注意的是 logisim 最大支持 5-32 译码器，但 op 和 funct 都是 6 位，所以需要用 1-2 译码器和 5-32 译码器结合使用。

根据前述分析，可以在 logisim 下设计并实现——将 op 和 funct 字段转化为对应指令信号的电路，该转化电路如图 3.5 所示。

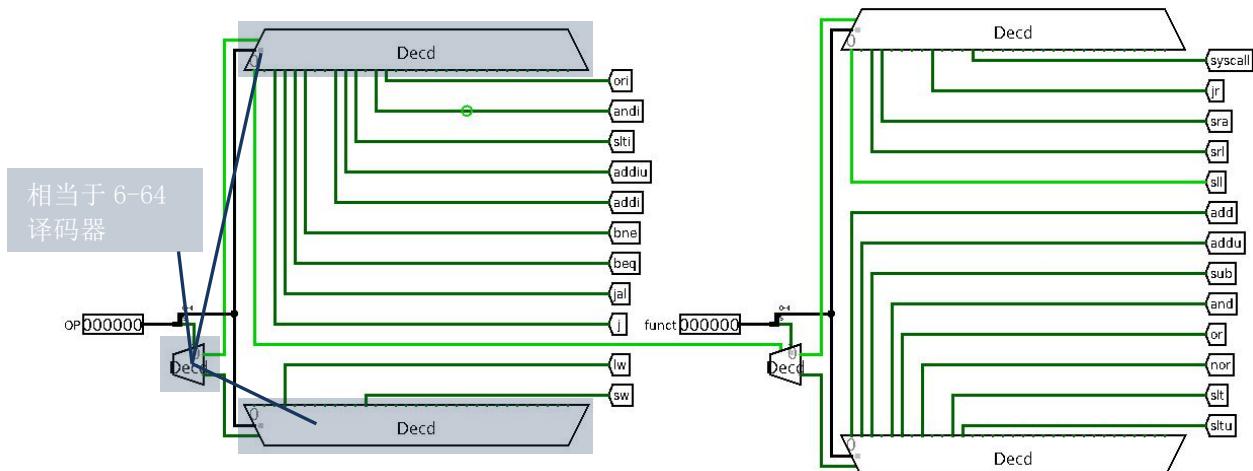


图 3.5 op、funct 转指令信号

从该电路中可知，当 op 为 0、funct 为 0 时，表明当前指令为 sll，这是正确的。

# 华中科技大学课程实验报告

显然，该电路可以将 op、funct 值转化为对应的指令信号。

指令和控点信息的关系表，可以通过表 3.4 和数据通路得到，如表 3.5 所示。

表 3.5 器件数据来源汇总表

序号	指令	PCSrc		PCA ddrc	PCEn	RFR1 Src	RFRWSrc		RFDinSrc		RFWE	ALUYSrc		ALUOP				DMWE
		M1	M0				M1	M0	M1	M0		M1	M0	OP3	OP2	OP1	OP0	
1	add	0	1	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0
2	addi	0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	1	0
3	addiu	0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	1	0
4	addu	0	1	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0
5	and	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	1	0
6	andi	0	1	0	1	0	0	0	1	0	1	1	1	0	1	1	1	0
7	sll	0	1	0	1	1	0	1	1	0	1	0	1	0	0	0	0	0
8	sra	0	1	0	1	1	0	1	1	0	1	0	1	0	0	0	1	0
9	srl	0	1	0	1	1	0	1	1	0	1	0	1	0	0	1	0	0
10	sub	0	1	0	1	0	0	1	1	0	1	0	0	0	1	1	0	0
11	or	0	1	0	1	0	0	1	1	0	1	0	0	1	0	0	0	0
12	ori	0	1	0	1	0	0	0	1	0	1	1	1	1	0	0	0	0
13	nor	0	1	0	1	0	0	1	1	0	1	0	0	1	0	1	0	0
14	lw	0	1	0	1	0	0	0	0	1	1	1	0	0	1	0	1	0
15	sw	0	1	0	1	0	0	0	1	0	0	1	0	0	1	0	1	1
16	beq	equal	~equal	1	1	0	0	0	1	0	0	0	0	0	0	1	1	0
17	bne	~equal	equal	1	1	0	0	0	1	0	0	0	0	0	0	1	1	0
18	slt	0	1	0	1	0	0	1	1	0	1	0	0	1	0	1	1	0
19	slti	0	1	0	1	0	0	0	1	0	1	1	0	1	0	1	1	0
20	sltu	0	1	0	1	0	0	1	1	0	1	0	0	1	1	0	0	0
21	j	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
22	jal	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0
23	jr	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
24	syscall	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0

该表给出了执行某一指定指令时，各控点信息是为 0 还是为 1。

前述对表 3.5 的分析，实质上都是在以行为单位分析表格。如果换个思路，以列为单位分析表，那么该表格还可以表明：对于某一指定控点，其在哪些指令时为 1，在哪些指令时为 0。根据这个信息，很容易写出控点信息关于指令信号的逻辑表达式，进而可以用组合逻辑电路实现该逻辑表达式，从而得到控点信息的逻辑电路。由于每个控点都对应一个逻辑电路，而控点又有十几个，因此不一一给出每个控点的具体电路，而是只以 ALUYSrc 控点作为示例。

ALUYSrc 是一个四路选择器（用来选择运算器 ALU 第二个操作数的数据来源），故选择端包含两位 M1 和 M0。根据表 3.5，很容易就可以得到，M1 只有在当前指令为 addi、addiu、andi、ori、lw、sw 或 slti 时才为 1，故立刻可以写出 M1 的逻辑表达式为： $ALUYSrc1 = addi + addiu + andi + ori + lw + sw + slti$ 。同理，也可以写出 M0 的逻辑表达式为： $ALUYSrc0 = andi + sll + sra + srl + ori$ 。最后，通过一个分离器汇总 ALUYSrc1 和 ALUYSrc0 便可以得到 ALUYSrc 控点的逻辑电路，如图 3.6 所示。

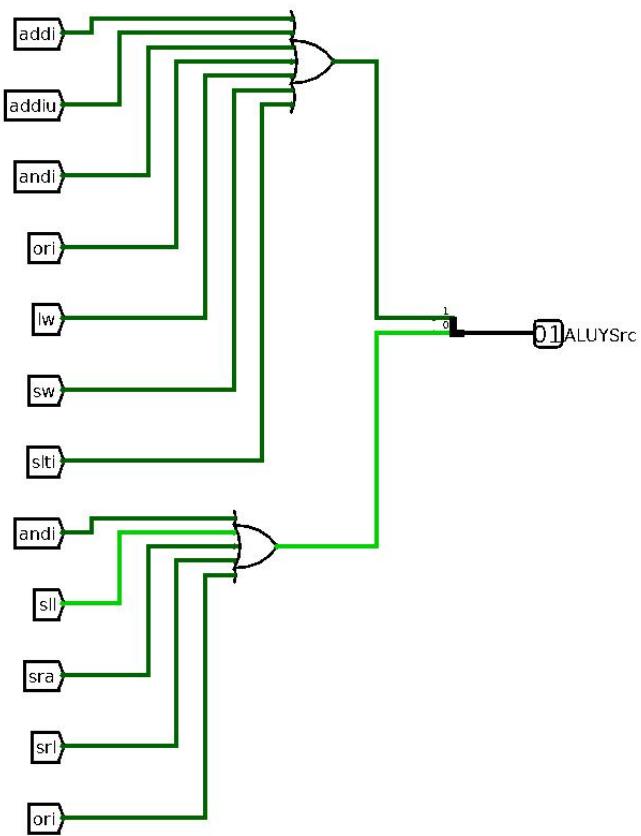


图 3.6 ALUYSrc 控点的逻辑电路

其余控点均可以仿照上述分析和设计过程得到对应的逻辑电路，完成了所有控点的逻辑电路，也就完成了控制器的设计。

### 3.3 实验步骤

- (1) 仔细阅读本次实验的对应的课本内容和上课所讲的 ppt，理解实验原理。在利用 logisim 绘制实验电路之前，仔细查看了谭老师 ppt 里关于单周期 MIPS 架构 CPU 设计的相关内容，ppt 里提到了两种设计方案，一种设计方案是一类一类指令的设计，先实现一类指令，然后修改电路兼容另一类型指令，依此类推直到电路支持 R、I 和 J 型指令；另一种方案是北航的利用表格简化设计方案，并且一次得到所有控点信息从而实现所有指令。经考虑后，决定采用了第二种方案，因为感觉这样虽然前期分析和作表比较麻烦，但一旦完成表格，电路设计就变得十分简单。
- (2) 利用 logisim 平台中现有部件构建取指令通路 (PC 寄存器 32 位)，其

中 PC 由时钟驱动, 每个时钟自动完成取值以及  $PC=PC+4$  的功能, 控制存储器在后续步骤中实现。此处指令存储器选用 10 位地址线, 32 位数据线的 ROM 部件, MIPS32s 位地址为字节地址, ROM 地址线宽度有限, 建议将 CPU 32 位地址高位部分和字节偏移部分直接屏蔽, 数据存储器也参照类似方法处理。

- (3) 利用综合实验一中封装的 32 位运算器以及 RAM 模块构建能满足上述指令系统的简单运算通路, 具体指令运算通路可以利用 MARS 仿真器中的 MIPS-XRAY 功能查看, 如图 3.7 所示。

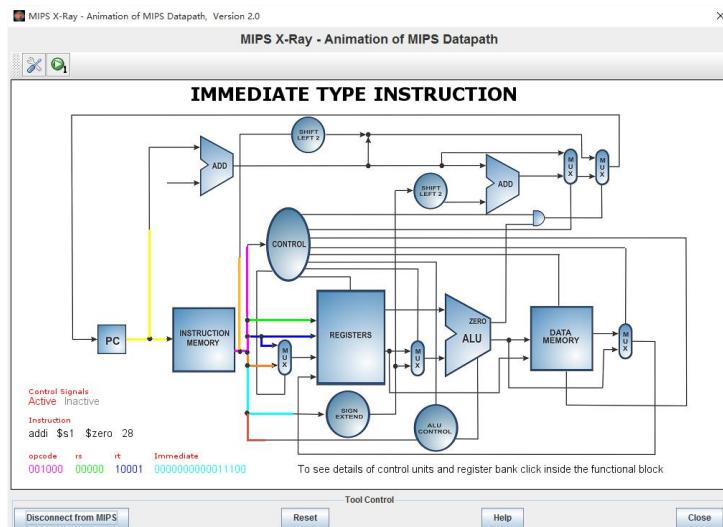


图 3.7 Mars 的 MIPS-XRAY 功能

- (4) 对步骤 (3) 构建的不同指令的数据通路进行综合, 生成控点 (操作控制信号) 设计一览表, 并且根据控点信息表设计控制器。
- (5) 利用 Mars 汇编老师提供的测试用例, 并将机器码以 16 进制导出然后加载到 CPU 的指令存储器中, 然后启用时钟模拟进行测试。

## 3.4 故障与调试

### 3.4.1 接口处数据传输问题

**故障现象:** 开启时钟模拟后, 时钟在跑, 但是 PC 的计数寄存器却不会增加, 即 CPU 一直停在第 1 条指令处而不会往下运行。

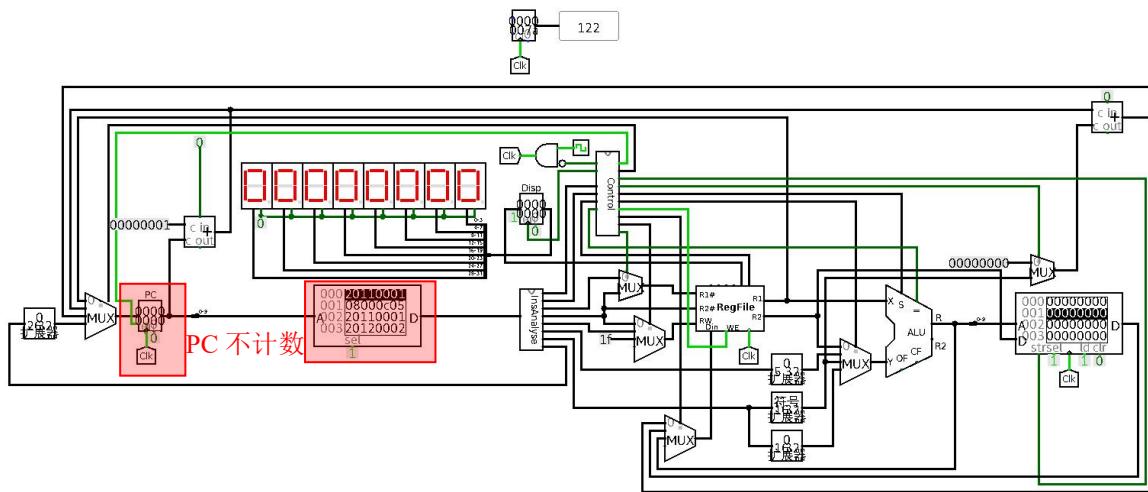


图 3.8 CPU 电路故障 1 运行截图

原因分析：利用 logisim 中的探测器，将探测器放到多路选择器的四个输入源，其中 3 号输入源对应的是  $PC+1$ ，此时其值为 1，说明是多路选择器的输入源没有问题。然后将探测器连接到多路选择器的输出端，发现此时探测器的值却为 0，表明多路选择器并未选择 3 号输入源。再将探测器连接到多路选择器的选择端，发现此时探测器的值却为 0，表明控制器给出的  $PCSrc$  的控点信息是错误的。

**解决方案：**对照控制器中 PCSrc 控点的逻辑电路与从表 3.8 中得到的 PCSrc 控点的逻辑表达式，可发现 PCSrc 控点的逻辑电路有一个地方不小心连接错误，将之改正即可。

### 3.4.2 故障 2

**故障现象：**用设计的 CPU (调整布局前) 对排序测试用例进行测试时, 程序无法正常结束。程序运行截图如图 3.9 所示, 运行的时钟周期数大的离谱。

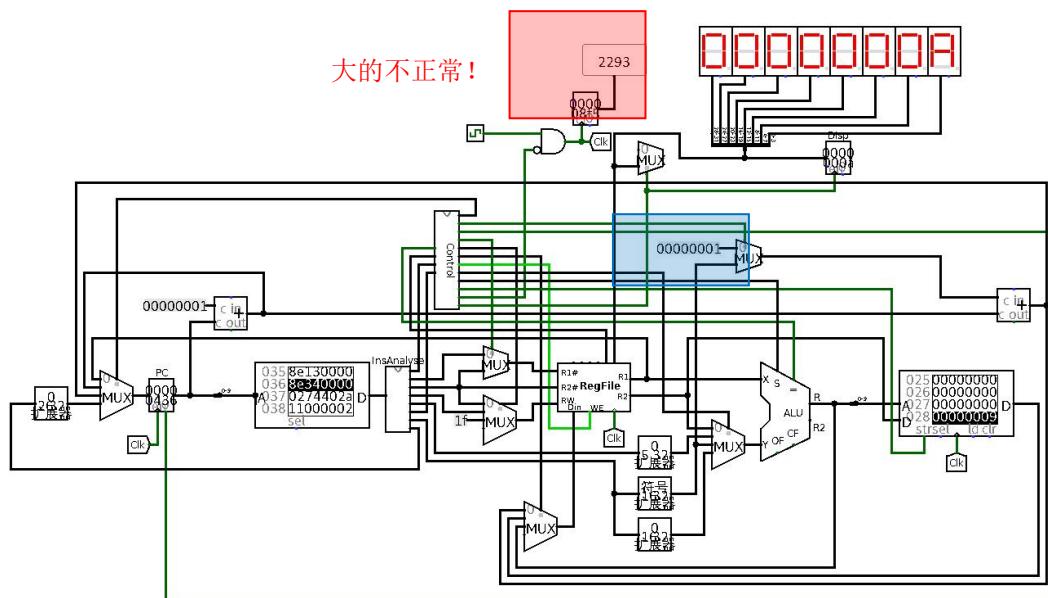


图 3.8 CPU 电路故障 2 运行截图

**原因分析:** 测试用例运行无法结束, 最大的可能原因就是跳转指令跳转不正确。所以将老师提供的测试用例中的 J 型指令测试用例用 Mars 编译后载入程序存储器, 一个周期一个周期的执行, 同时也用 Mars 单步运行测试用例, 比对每条执行的运行结果。最后发现, 是 jr 时跳转不正确, CPU 会跳转到本该跳转的那条指令的下一条指令。这里的主要原因就是, 程序在执行 jal 时, 正确执行的话会把 PC+1 存储到 \$ra 寄存器中, 但是当时关于 jal 指令的功能是询问的同学, 他说是把 PC+2 存到 \$ra 中, 所以 jal 所存的 PC 地址是错误的。

**解决方案:** 将蓝色方框内的加法器的一个操作数, 由常数 1 改为常数 0 即可, 这样 jal 就会把 PC+1 而不是 PC+2 存到寄存器 \$ra 中了。

### 3.4.3 故障 2

**故障现象:** 调用 syscall 将 \$a0 打印到显像管时, 由于指令执行的频率很快, 因此 \$a0 的值只是在显像管上一闪而过。当不调用 syscall 时, 显像管上显示的是不合法数据, 运行截图如图 3.9 所示。

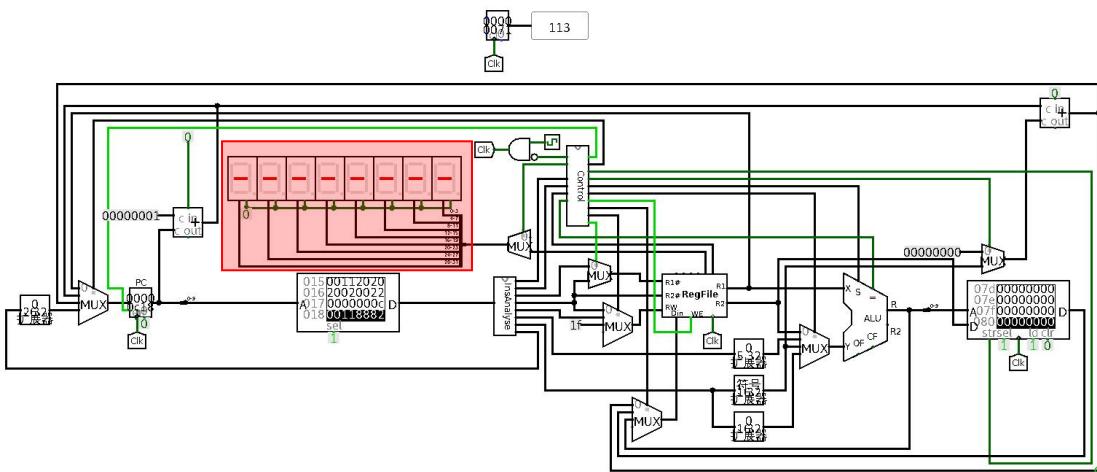


图 3.9 CPU 电路故障 3 运行截图

**原因分析：**设计时，不清楚在不调用 `syscall` 将 \$a0 显示在显像管上时，显像管应该显示什么数据。询问老师后，老师说应该显示上一个 \$a 值。

**解决方案：**在显像管和 \$a0 之间加一个寄存器，其输出接显像管，输入接 \$a0 值，时钟接控制器给出的 Disp 信号。当调用 `syscall` 将 \$a0 显示在显像管上时，便把此时 \$a0 的值写入寄存器。

## 3.5 测试与分析

在排除故障后，CPU 可以正确地执行各条指令。

对于测试用例，选用的老师提供的“benchmark”（benchmark.asm），因为 benchmark 中用到了所有要求的指令，且显示结果容易观察。benchmark 主要包含几个部分：第一部分是跳转指令测试，这一部分虽然不会在显像管上显示什么，但显然如果跳转指令错误，接下来的测试是肯定会执行错误的；第二部分是移位测试，包括逻辑右移测试、逻辑左移测试和算数右移测试；第三部分是走马灯测试，该部分将综合测试多条指令，在显像管上实现“走马灯”的效果；第四部分是降序排序测试，该部分会在显像管上依次显示一个等差数列，同时，该部分执行结束后，数据存储器中的数据会降序排列。

首先，利用 Mars 将 benchmark.asm 汇编成机器码（16 进制）并导出，稍作修改后（在导出文件头添加“v2.0 raw”），然后将其载入 CPU 的程序存储器 ROM 中，接着开启时钟模拟。运行过程如下：

# 华中科技大学课程实验报告

移位测试运行截图如图 3.10 所示。

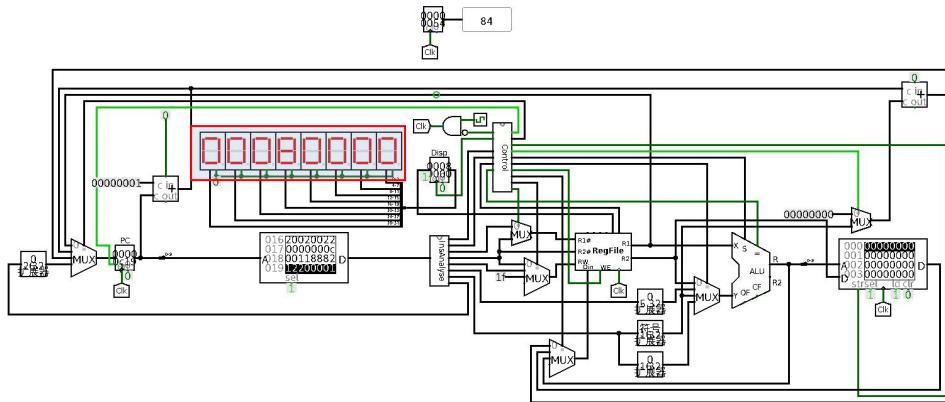


图 3.10 benchmark 移位测试运行截图

走马灯测试运行截图如图 3.11 所示。

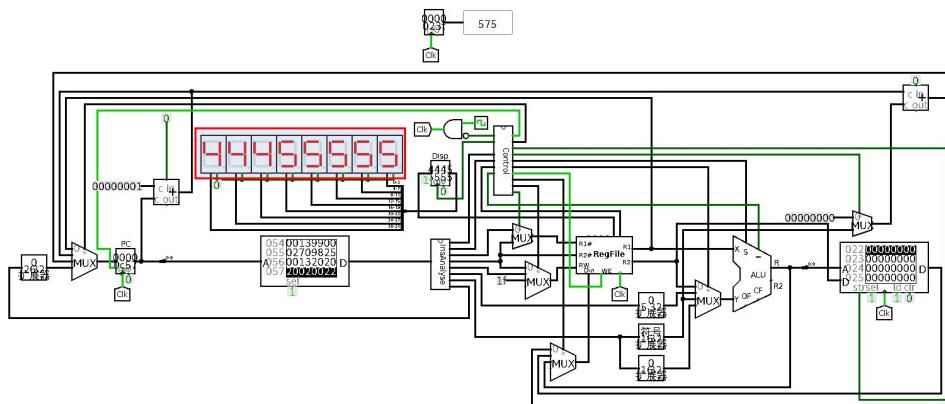


图 3.11 benchmark 走马灯测试运行截图

降序排序测试结果 (RAM 按字访问故数据间隔为 4 字节) 截图如图 3.12 所示。

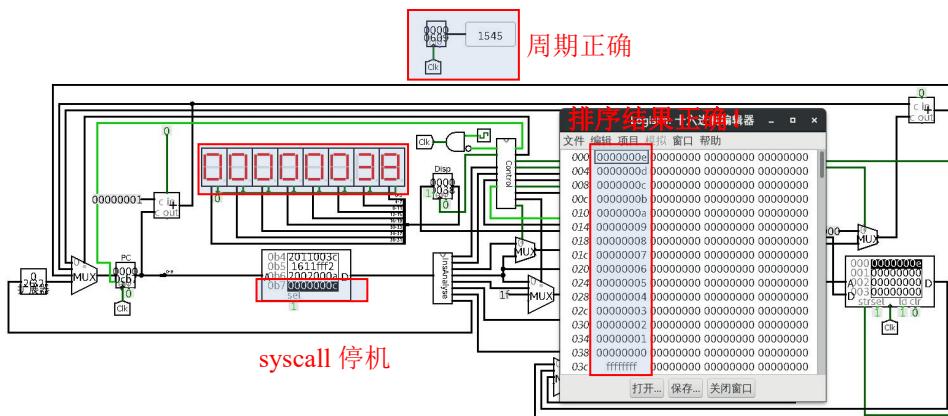


图 3.12 benchmark 排序测试运行截图

通过测试结果，可以说 CPU 正确运行了 benchmark，基本说明 CPU 设计正确。

## 4 组成原理与编译原理

本章节应该算作此次组成原理一大实验的亮点，即成功地将组成原理的实验与编译原理的实验成功地结合了起来，使得在组原的 CPU 上可以运行编译实验实现的编译器所编译出来的程序。为了实现这一点，组原的 CPU 在原有实验（实验 3）的基础上进行了一定的改进（稍后提到），编译的编译器也在生成汇编代码阶段做了特殊处理。

首先，简单说明编译器所做的特殊处理。第一，是关于一些条件跳转指令的处理，由于此次组原实验支持的条件跳转指令只有“beq”和“bne”，即相等跳转和不相等，故需要编译器时对其余条件跳转指令作一些转化，例如对于小于等于跳转指令“ble \$rs, \$rt, label”，利用四条等价 MIPS 指令（已经实现了的）和 1 号编译器使用寄存器去翻译，使用“beq \$rs, \$rt, label”、“slt \$1, \$rs, \$rt”、“addi \$1, \$1, -1”、“beq \$1, \$0, label”四条指令便可等效地实现小于等于跳转的功能；第二，是关于乘除法的处理，由于此次组原实验并不支持乘法指令和除法指令，因此编译器需要在生成汇编代码时，将乘法利用 booth-乘法算法（该算法只用到移位、加法等已有指令）实现，至于除法，没有找到合适的算法实现；第三，由于 CPU 中用到的数据存储器 RAM 的大小仅有 1024 个字节，地址是从 0~1023，因此为了避免可能的错误，编译器生成的汇编代码在开始处便将“\$sp”置为 1023（数据存储器尾）；第四，编译器在编译打印字符串时，针对组原 CPU 用到的“哑终端”，生成汇编代码时作了一定处理。

接下来考虑组原方面的 CPU 所做的改进。第一，增加了“输入一个整数的”syscall 功能，该 syscall 对应的 \$v0 值为 5，且会将读入的整数存到 \$v0 寄存器中，为此需要做的修改有：添加一个输入端以及一个输入确认按钮（当调用该 syscall 时，CPU 会处在停机状态等待用户输入，用户输入完毕后按下输入确认按钮程序才会继续执行），为 RegFile 的 RW 端、Write\_data 端（多路选择器）添加新的数据来源，并在控制器中进行适当修改；第二，增加了“在哑终端输出一个字符”的 syscall 功能，该 syscall 对应的 \$v0 值为 11，会将 \$a0 中 ascii 码对应的字符打印到哑终端中，为此需要添加一个哑终端，并对控制器作适当修改和改进；第三，对原有的打印 \$a0 到

# 华中科技大学课程实验报告

显像管的 syscall 作了修改，原 CPU 下只要 \$v0 != 10 即是调用该功能，现在规定其 syscall 对应的 \$v0 值为 1，同样需要对控制器作适当修改和改进。

待编译的源代码如图 4.1 所示，这是一种名为 Decaf 的语言，是编译原理所要求采用的一种面向对象的语言。该院代码的功能较为简单，用户输入一个整数 n，然后调用 calculate 类的不同方法，分别计算 n 的阶乘和  $(1!+2!+\dots+n!)$ 、计算 n 的累加和  $(1+2+\dots+n)$  和计算 n 的阶乘  $(n!)$  并分别打印。

```
1 class Claculate
2 {
3     int getSum(int n)
4     {
5         int i;
6         int sum;
7         sum = 0;
8         for (i = 1; i <= n; i = i + 1)
9             sum = sum + i;
10        return sum;
11    }
12
13    int getMul(int n)
14    {
15        int i;
16        int mul;
17        mul = 1;
18        for (i = 1; i <= n; i = i + 1)
19            mul = mul * i;
20        return mul;
21    }
22
23    int getMulSum(int n)
24    {
25        int i;
26        int mul;
27        int sum;
28        mul = 1;
29        for (i = 1; i <= n; i = i + 1)
30        {
31            mul = mul * i;
32            sum = sum + mul;
33        }
34        return sum;
35    }
36 }
38 class Main
39 {
40     int main()
41     {
42         int n;
43         int result1;
44         int result2;
45         int result3;
46         class Claculate calc;
47         calc = new Claculate();
48         Print("Input n now");
49         n = ReadInteger();
50         Print("n is on digits now");
51         result1 = calc.getMulSum(n);
52         Print(result1);
53         Print("The total factorial is on digits");
54         result2 = calc.getSum(n);
55         Print(result2);
56         Print("The sum is on digits");
57         result3 = calc.getMul(n);
58         Print(result3);
59         Print("The factorial is on digits");
60     }
61 }
62 }
63
64
65
66
67
68
69
70
71
72
73
```

图 4.1 测试程序 decaf 源码

经过编译实验实现的编译器编译后，生成的汇编代码的片段如图 4.2 所示。

```
1 ori $sp, $0, 1023
2 ori $fp, $0, 1023
3 jal main
4 ori $v0, $0, 10
5 syscall
6 ##### Execute from "main" #####
7
8 getSum:
9 sw $ra, 0($sp)
10 sw $fp, -4($sp)
11 addu $fp, $sp, $0
12 ori $1, $0, 48
13 sw $1, -44($sp)
14 sub $sp, $sp, $1
15 ##### Correspond to source file line 3 #####
16 ##### Correspond to source file line 3 #####
17 lw $t0, -36($fp)
18 ori $t0, $0, 0
19 sw $t0, -36($fp)
803 booth mul:
804 srl $a0, $a0, 16
805 addi $t2, $0, 16
806 add $v0, $0, $a0
807 addi $a3, $0, 0
808 booth_loop:
809 andi $t2, $a0, 1
810 beq $a3, $t2, booth_l1
811 slt $t3, $t2, $a3
812 beq $t3, $0, booth_l2
813 add $v0, $v0, $a1
814 j booth_l1
815 booth_l2:
816 sub $v0, $v0, $a1
817 booth_l1:
818 sra $v0, $v0, 1
819 add $a3, $t2, $0
820 sra $a0, $a0, 1
821 addi $a2, $a2, -1
822 bne $a2, $0, booth_loop
823 add $s0, $v0, $0
824 jr $ra
825 ##### Use booth multiply to replace instruction "mul" #####

```

图 4.2 测试程序汇编代码片段

通过编译得到汇编代码后，还需要借助汇编器 Mars 将汇编代码转化机器代码

并以十六制形式导出到文件中，然后将在该文件的文件头处添加“v2.0 raw”便可以将其加载进 logisim 的程序存储器 ROM 中。开始时，CPU 截图如图 4.3 所示。

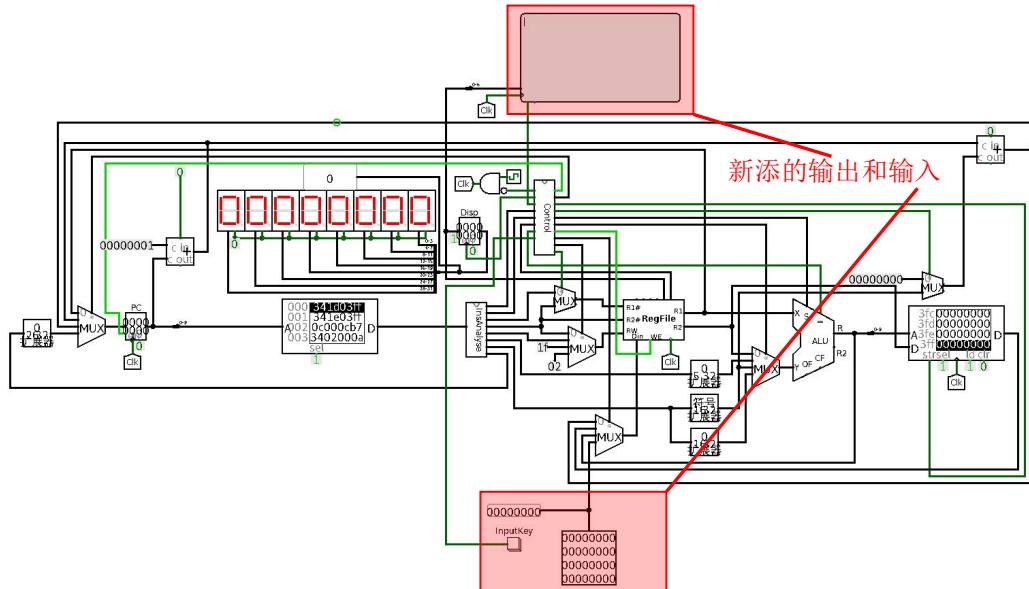


图 4.3 适应编译的 CPU

开启时钟模拟，程序将会开始执行，并在需要输入 n 时等待用户输入。等待输入时的运行截图如图 4.4 所示。

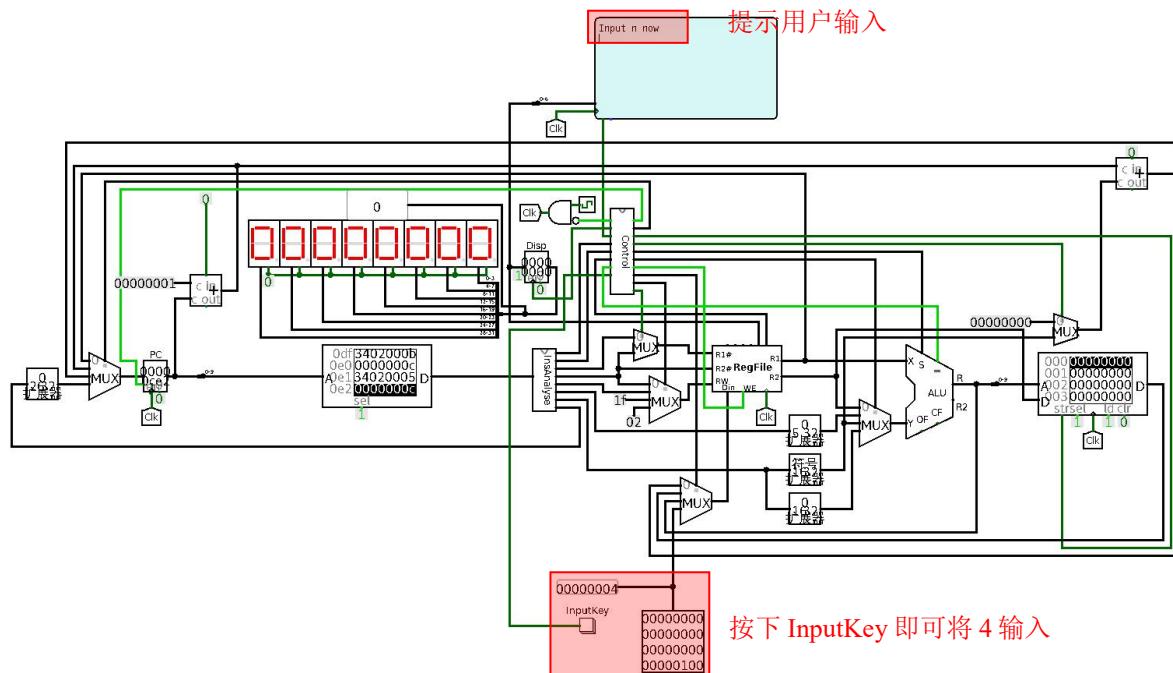


图 4.4 CPU 等待输入

计算并打印出阶乘和的运行截图如图 4.5 所示，因为版面原因，只截出显像管

和哑终端。(显像管显示的是 16 进制, 10 进制数值由探测器给出)

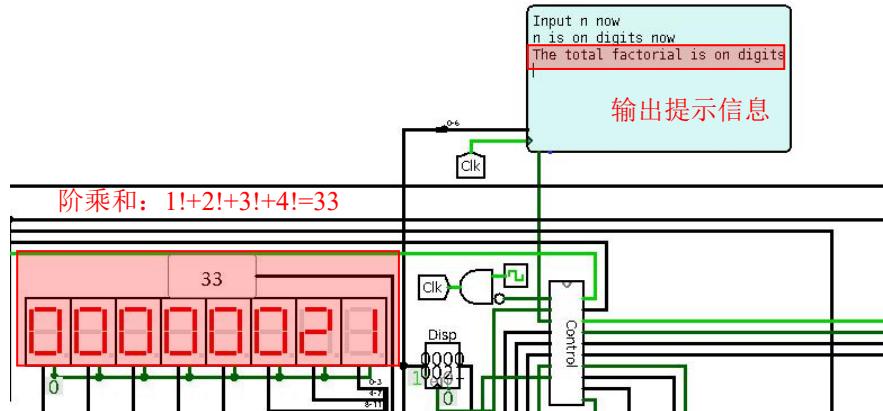


图 4.5 CPU 计算并打印阶乘和

计算并打印累加和的运行截图如图 4.6 所示。

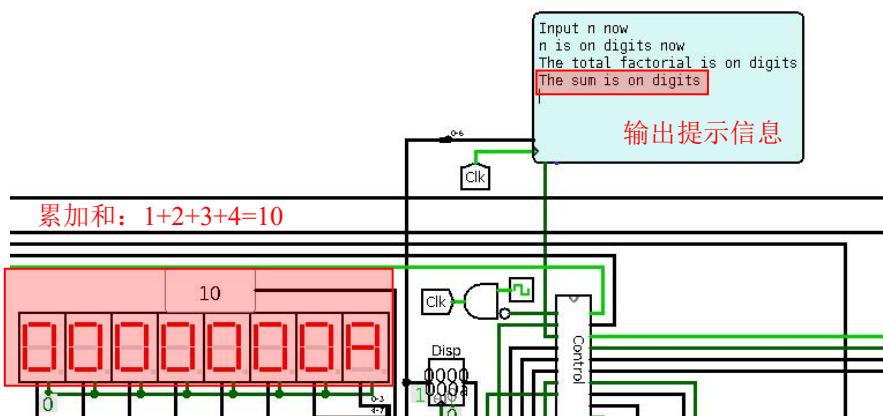


图 4.6 CPU 计算并打印累加和

计算并打印阶乘的运行截图如图 4.7 所示。

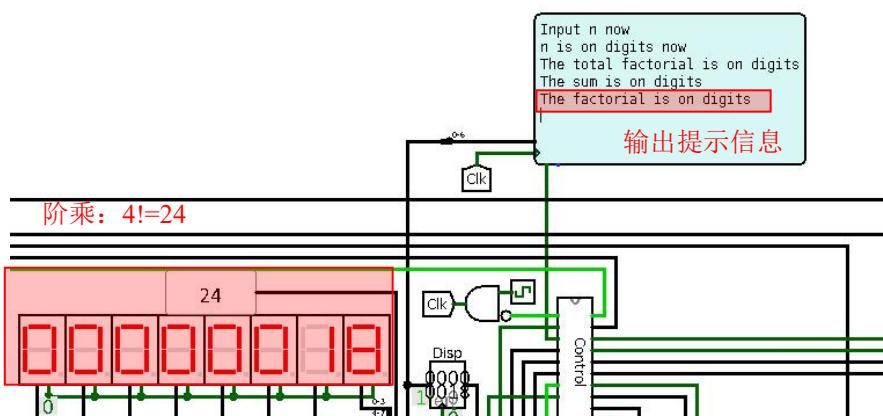


图 4.6 CPU 计算并打印累加和

通过上述运行测试, 所设计的改进后的 CPU 可支持运行自己写的编译器所编译出来的一部分程序。

## 5 总结与心得

### 5.1 实验总结

本次实验主要完成了如下几点工作：

- 1) 思考并设计了 4 位先行进位加法器，完成了先行进位电路的设计方案，学习并设计了 32 位具有组间先行进位、组内先行进位特征的加法器，完成了运算器 ALU 的方案设计；思考并设计了包含 32 个 32 位寄存器的寄存器组电路，完成了一个计算存储器中部分数据累加和并储存中间结果的自动运算电路的设计方案；思考并设计了取指令通路，完成了数据通路的设计方案，学习并设计了支持 24 条指令的单周期 CPU；并改进了任务要求中的 CPU 使其可以运行编译原理实验实现的编译器所编译出来的程序。
- 2) 实现并封装了支持 13 种运算、提供有符号溢出和无符号溢出标志位的 32 位运算器 ALU 单元，实现并封装了包含 32 个 32 位寄存器、可同时读出两个寄存器和可同时受时钟驱动可同时写入一个寄存器的寄存器组电路 RegFile，实现了一个计算存储器中部分数据累加和并储存中间结果的自动运算电路，实现了支持 24 条指令的单周期 CPU。
- 3) 有效地结合了组成原理的实验和编译原理的实验，使得它们互相支持，即改进版的组成原理实验 CPU，可以运行编译原理实验编译器编译出的汇编程序（借助汇编器翻译成机器代码）。

### 5.2 实验心得

- 1) 这次组成原理的实验收获颇丰，因为自己相对于软件更喜欢硬件的东西，更喜欢做一些底层的东西，所以组原的实验是我非常愿意投入心思和精力去完成的。通过这次组原实验，对先行进位加法器、运算单元 ALU、寄存器组 RegFile、单周期 CPU 都有了很深刻的理解，也对 logisim 的运用有了些心得。尤其是在设计 CPU 时，对数据通路和控制器进行了深入的学习和思考，对它们的认识也不仅仅再局限于之前 Verilog 课程上所学的东

西。除此之外，通过自己设计并实现简单版的 CPU 也让我对 CPU 的设计流程有了一定了解：先是选择指令集，然后构造取指令通路，接着构造数据通路，最后完成控制器。在初步实现了 CPU 后，我花了大约半上午的时间去调整 CPU 的布局，使其看起来更为整洁，同时，也花了一定的时间对 CPU 进行改造使其支持最基本的编译生成的目标代码。我十分喜欢和享受这样的设计过程。

- 2) 个人感觉组成原理实验的安排比较合理，由浅及深，由单部件到最后的 CPU，层层递进，环环相扣。但总的来说，个人感觉组原的实验难度较低，实验 3 所实现的 CPU 有点过于简单，支持的指令条数还是有点少，对编译的支持有点欠缺。

## 参考文献

- [1] DAVID A. PATTERSON(美). 计算机组成与设计硬件/软件接口(原书第4版). 北京: 机械工业出版社.
- [2] David Money Harris(美). 数字设计和计算机体系结构(第二版). 机械工业出版社
- [3] 秦磊华, 吴非, 莫正坤. 计算机组成原理. 北京: 清华大学出版社, 2011年.
- [4] 袁春凤编著. 计算机组成与系统结构. 北京: 清华大学出版社, 2011年.
- [5] 张晨曦, 王志英. 计算机系统结构. 高等教育出版社, 2008年.

• 指导教师评定意见 •

---

---

### 一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字: 嵌入签名图片

### 二、对课程实验的学术评语（教师填写）

### 三、对课程设计的评分（教师填写）

评分项目 (分值)	报告撰写 (30分)	课设过程 (70分)	最终评定 (100分)
得分			

指导教师签字: \_\_\_\_\_ 2017-09-27

---

---