

华中科技大学

2017

计算机组成原理

· 实验报告 ·

专 业： 计算机科学与技术

班 级： ZY1501

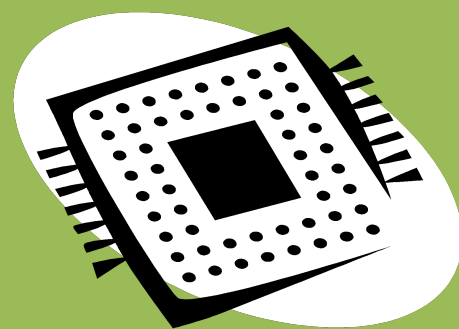
学 号： U201514898

姓 名： 胡思勳

电 话： 15727072561

邮 件： husixul@hotmail.com

完成日期： 2018-1-9



计算机科学与技术学院

华中科技大学课程实验报告

目 录

1	运算器实验	2
1.1	设计要求	2
1.2	方案设计	5
1.3	实验步骤	10
1.4	故障与调试	10
1.5	测试与分析	11
2	CPU 设计实验	13
2.1	设计要求	14
2.2	方案设计	14
2.3	实验步骤	19
2.4	故障与调试	20
2.5	测试与分析	21
3	总结与心得	24
3.1	实验总结	24
3.2	实验心得	24
	参考文献	25

1 运算器实验

1.1 设计要求

1.1.1 八位串行可控加减法电路设计

利用已经封装好的全加器（封装 1）设计 8 位串行可控加减法电路，其引脚电路如图 1.1 所示。

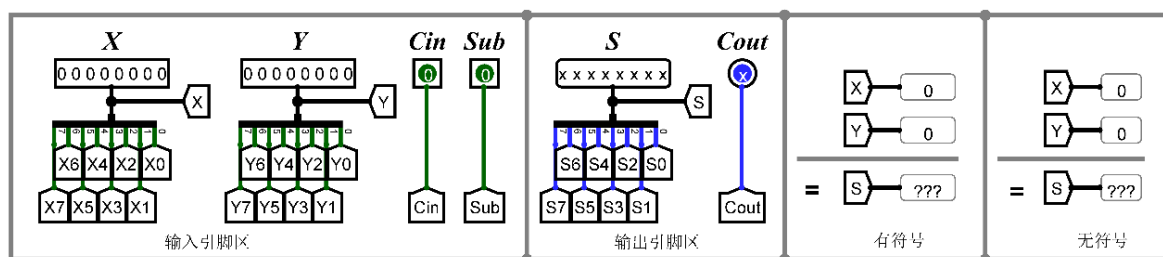
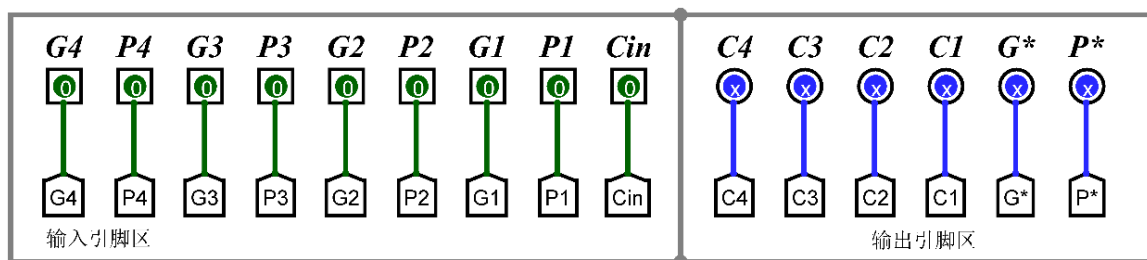


图 1.1 八位串行可控加减法电路引脚定义

1.1.2 四位先行进位电路

根据图 1.2 定义的输入输出引脚完成 4 位先行进位电路。



请根据以上引脚以及隧道信号设计完成74LS182先行进位电路

请勿增减引脚定义

图 1.2 四位先行进位电路引脚定义

1.1.3 四位快速加法器设计

利用已经前一步设计好的四位先行进位电路构造四位快速加法器，其引脚定义如图 1.3 所示。

华中科技大学课程实验报告

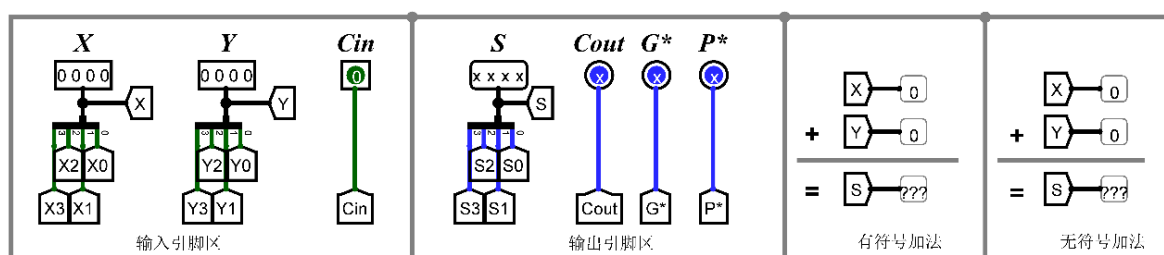


图 1.3 四位快速加法器引脚定义

1.1.4 十六位快速加法器设计

利用四位先行进位电路和四位快速加法器构造十六位组间先行进位，组内先行进位快速加法器，其引脚定义如图 1.4 所示。

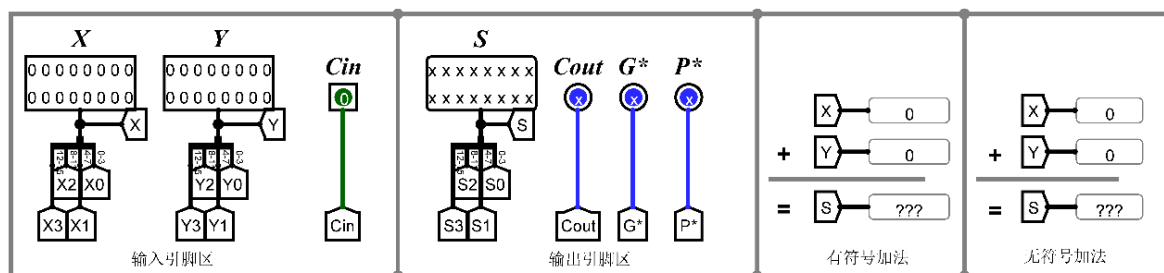


图 1.4 十六位快速加法器引脚定义

1.1.5 32 位快速加法器设计

利用前面部件完成 32 位快速加法器，并分析其时间延迟，其引脚定义如图 1.5 所示。

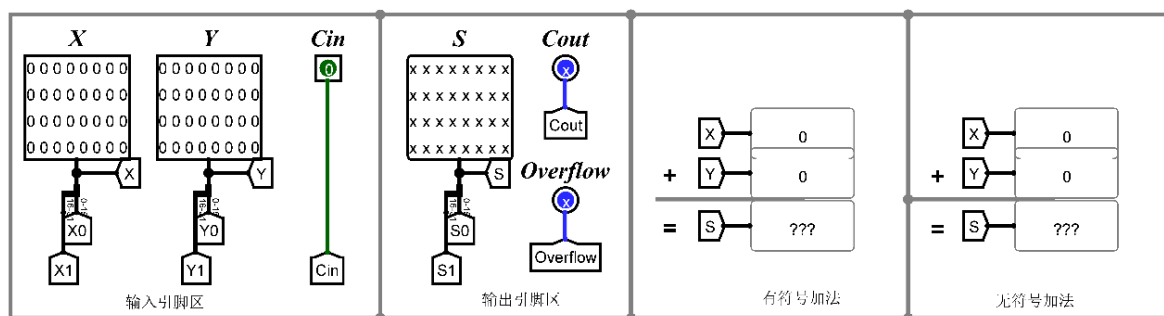


图 1.5 快速加法器引脚定义

1.1.6 32 位 MIPS 运算器设计

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，支持常程

华中科技大学课程实验报告

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 片引脚与功能描述

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 Result2=0	逻辑左移 (Y 取低五位)
0001	1	Result = X >>> Y Result2=0	算术右移 (Y 取低五位)
0010	2	Result = X >> Y Result2=0	逻辑右移 (Y 取低五位)
0011	3	Result = (X * Y)[31:0]; Result2 = (X * Y)[63:32]	有符号乘法
0100	4	Result = X/Y; Result2 = X%Y	无符号除法
0101	5	Result = X + Y Result2=0	(Set OF/CF)
0110	6	Result = X - Y Result2=0	(Set OF/CF)
0111	7	Result = X & Y Result2=0	按位与
1000	8	Result = X Y Result2=0	按位或
1001	9	Result = X ⊕ Y Result2=0	按位异或
1010	10	Result = ~(X Y) Result2=0	按位非

华中科技大学课程实验报告

表 1-3 运算符功能（续）

ALU OP	十进制	运算功能
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

1.1.7 一位补码乘法器设计

图 1.6 给出了一位补码乘法器的引脚定义以及主要部件，请增加控制电路和数据通路使得该电路能自动完成 8 位补码一位乘法运算，设置引脚值，然后驱动时钟仿真，电路自动完成运算，运算结束结果传输到输出引脚。

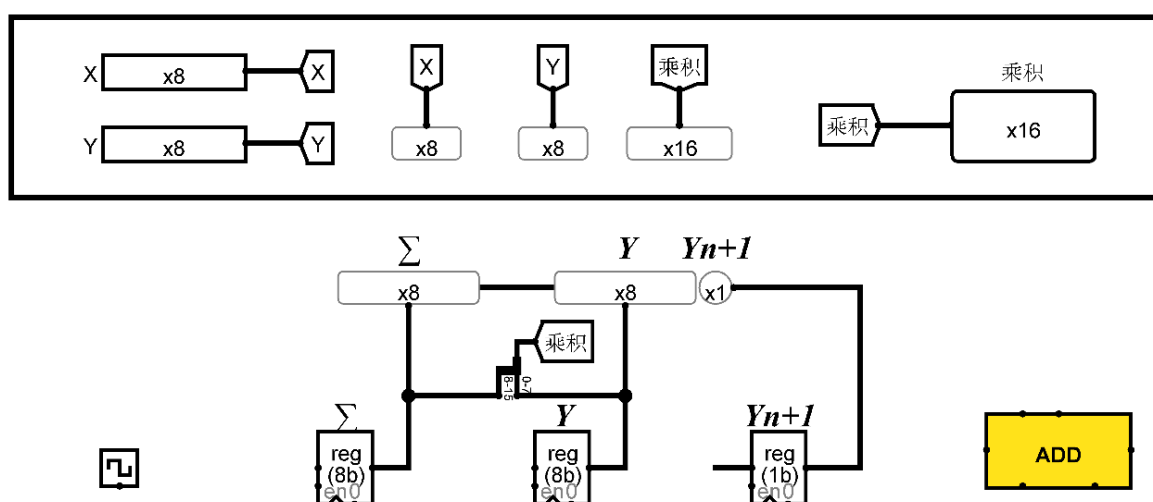


图 1.6 一位补码乘法器引脚定义及电路主要部件

1.2 方案设计

1.2.1 八位串行可控加减法电路方案设计

8 位串行可控加减法器由 8 个一位全加器串联实现，每一个全加器的进位输出作为下一个（更高位）的全加器的进位输入。假设两个操作数分别为 X 和 Y，当进行加

法时，直接将两个数的各位作为 8 个一位全加器的输入，而进行减法时，可采取加补码的方式进行，首先将 Y 按位取反，然后将最低位的进位输入置为 1，以达到将 Y 变为补码的效果。最终设计出的电路图如图 1.7 所示。在进行加法时，若有来自前一个

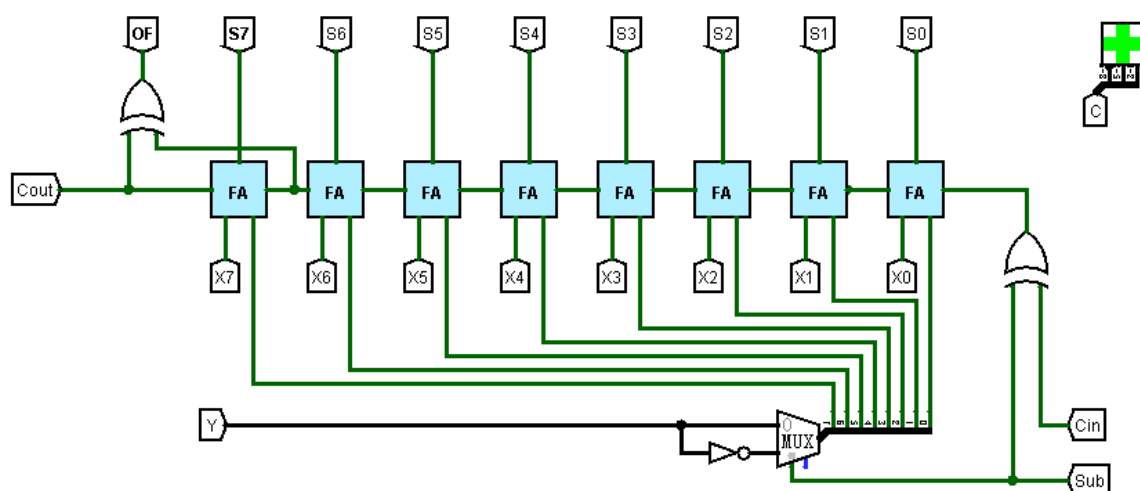


图 1.7 八位串行可控加减法器电路图

加法器的进位，则进位应该设置为 1，在进行减法时如果有来自前一位的借位则第一个一位加法器进位应该设置为 0，因此应该使用 Sub xor Cin 作为最低位一位加法器的进位输入。

1.2.2 四位先行进位电路

由于串行进位电路不能达到加法器的速度要求，因此需要四位先行进位电路，其原理是使用输入首先判断哪些位需要进位，而不是在计算时让每一位依赖于前一位的

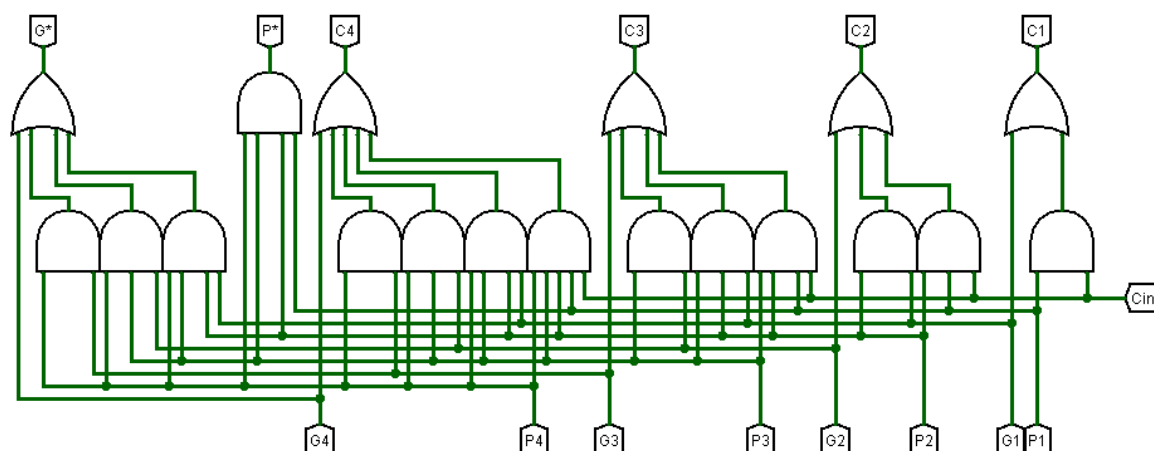
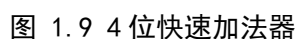
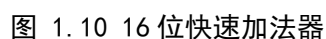


图 1.8 4 位先行进位电路

使用 1.2.2 中的四位加法器先行进位电路, 并将 $X_i Y_i$ 作为 G_i 的输入, $X_i \oplus Y_i$ 作为 P_i 的输入, 然后将 $C_i P_{i+1}$ 作为 S_i 进行输出, 这样就构成了 4 位快速加法器, 如图 1.9 所示。由于没有每一位之间的依赖关系, 其运算速度将比串行进位加减法器快。



十六位快速加法器使用 4 个 4 位快速加法器实现，加法器与加法器之间的进位使用一个 4 位先行进位电路实现。X 与 Y 均按照每 4 位一段分为 4 段输入 4 个加法器中，输出的 4 个 S 进行合并得到 16 位的 S。其结构如图 1.10 所示，32 位快速加法器



则不使用先行进位电路，而是直接使用 2 个 16 位加法器串联，将 X 和 Y 按照 16 位一段分为 2 段后分别输入两个 16 位加法器，再将输出合并得到 S，其结构如图 1.11 所示。

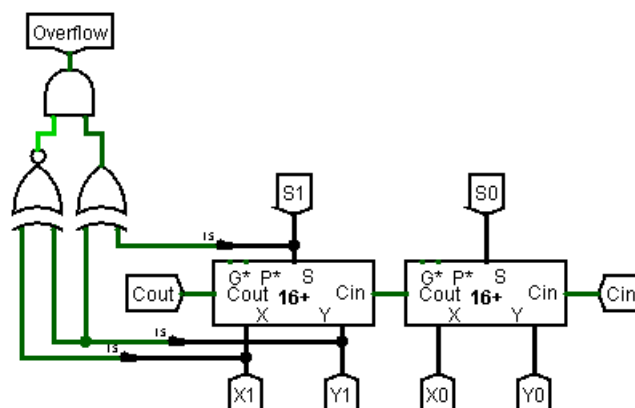


图 1.11 32 位加法器结构

1.2.5 32 位 MIPS 运算器方案设计

32 位 MIPS 运算器(ALU)需要实现的功能较多，但均由 ALUOP 进行控制，因此将各个功能实现后使用 ALUOP 控制多路选择器进行选择来控制输出哪一个功能的结果即可。其结构如图 1.12 所示。其中，除了加法与减法使用 32 位快速加法器实现之外，其余功能均使用 logisim 内置电路实现。图中的 4 个多路选择器从左到右依

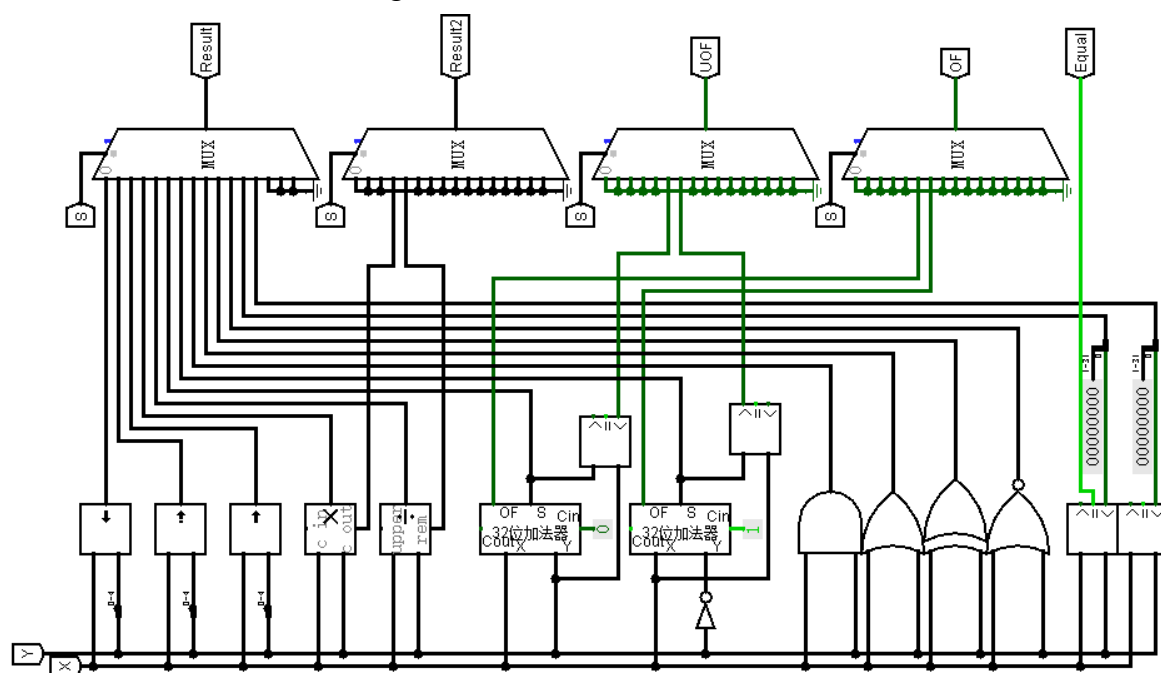


图 1.12 ALU 结构

次控制 Result、Result2、UOF 和 OF 的输出。Result2 用于乘法的高位以及除法的余数输出，因此除了连接乘法与除法模块之外，其余输入均接地。UOF 和 OF 同理，除了连接进行加法运算和减法运算的模块之外，其余的输入均接地。与 OF 不同的是，UOF 不能够直接使用加法器的进位输出判断是否溢出，而需要额外的判断机制，即加法的输出小于加数，减法的输出小于被减数即为溢出。由于 Equal 输出对于任意的输入均有效，因此不需要使用多路选择器进行选择。减法的实现同样使用加法器完成，直接使用将被加数转换为反码并将借位位设置为 1（等效为加补码）的方式即可完成。

1.2.6 补码一位乘法器方案设计

补码一位乘法依赖于公式：

$$\begin{aligned}
 [x \times y]_{\text{补}} &= [x]_{\text{补}} \times 0.y_1y_2 \dots y_n - y_0[x]_{\text{补}} \\
 &= [x]_{\text{补}} \times (-y_0 + y_12^{-1} + y_22^{-2} + \dots + y_n2^{-n}) \\
 &= [x]_{\text{补}} \times [-y_0 + (y_1 - y_12^{-1}) + (y_22^{-1} - y_22^{-2}) + \dots + (y_n2^{-n+1} - y_n2^{-n})] \\
 &= [x]_{\text{补}} \times [y_1 - y_0 + (y_2 - y_1)2^{-1} + (y_3 - y_2)2^{-2} + \dots + (0 - y_n)2^{-n}]
 \end{aligned}$$

然在电路中后对于公式中的每一个 $y_n y_{n+1}$ 进行计算，若其为 01 则 Σ 加上 $[x]_{\text{补}}$ ，若为 10 则 Σ 加上 $[-x]_{\text{补}}$ ，否则 $y_n - y_{n+1} = 0$ ， Σ 不进行任何变化。在每次判断并对于 Σ 进行操作后将 Σ 向右移动一位，且必须为算术右移。最终实现的电路如图 1.13。图

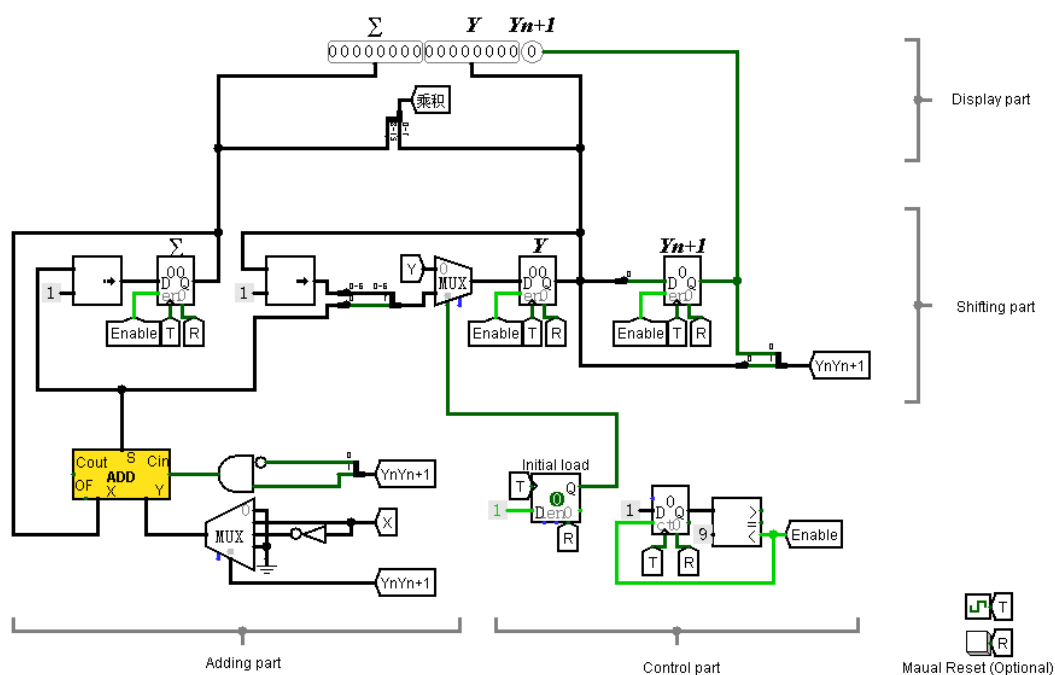


图 1.13 一位补码乘法电路

华中科技大学课程实验报告

中一共分为 4 个部分，最上面为显示部分，用于显示中间以及最终结果，中间为移位部分，用于进行算术右移，下面左边为加法部分，用于以 $y_n y_{n+1}$ 为依据进行判断并选择让 Σ 与 $[x]_{\text{补}}$ 或 $[-x]_{\text{补}}$ 进行相加。而下面右边则是整个时序电路的控制部分，用于控制补码一位乘法的启停。

1.3 实验步骤

- (1) 通过题目要求对于各个电路进行方案设计。
- (2) 对于设计出来的方案进行电路的搭建，其中：首先实现的是 8 位串行可控加减法器与 4 位先行进位电路，并在 4 位先行进位电路的基础上实现了 4 位快速加法器，然后基于 4 位快速加法器实现了 16 位快速加法器，并使用两个 16 位快速加法器实现了 32 位快速加法器，这一依赖链中的每一环都是至关重要的，任何一环出错会直接导致基于它的所有实现出现问题。在实现完加法器之后通过已经实现的加法器以及 logisim 自带的模块实现了 ALU，并对于 ALU 进行测试。最后，按照方案设计实现了一位补码移位乘法器。
- (3) 在实现每个电路完成后均对其进行功能上的测试，以保证依赖于这一电路的实现不会出现问题。

1.4 故障与调试

1.4.1 ALU 减法结果错误问题

故障现象：ALU 执行减法操作后结果比预计结果小 1，如图 1.14 所示。

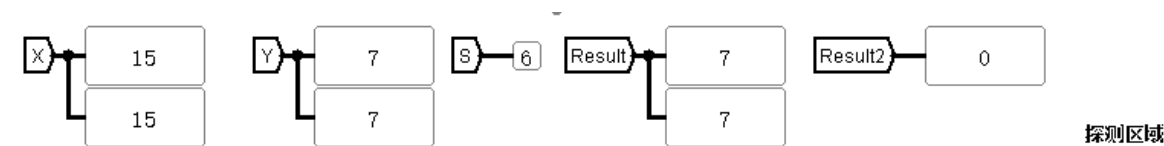


图 1.14 ALU 运算结果出错

原因分析：由于使用的是 32 位加法器实现的减法操作，在外部对于输入 Y 按位直接取反得到减数而忘记将借位位设置为 1 导致的。由于需要得到的是补码而不是反

码，因此需要将进位位取反以达到等效减数输入为反码的状态。

解决方案:将用于实现减法的 32 位快速加法器的进位位输入设置为 1(高电平)。

1.4.2 一位补码运算不能得到正确的结果

故障现象:在进行一位补码运算时，对两个给定的输入没有输出正确的结果。如图 1.15 所示，当输入 X 和 Y 分别为 22 和 5 时，期望输出的乘积是 110，而得到的是 366，与期望不符。

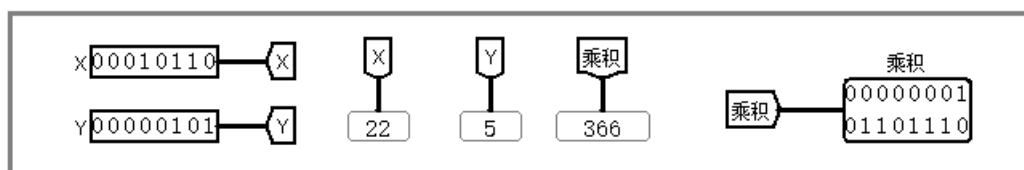


图 1.15 补码一位乘法结果错误

原因分析:通过一步步迭代并比较每一步的结果与输出结果，我发现是由于中间结果 Σ 为负数，而右移时进行了逻辑右移而不是算术右移导致了在进行下一次运算时其变成了正数，从而导致了整个结果的错误。

解决方案:将 Σ 寄存器的输入端连接的移位运算器的属性更改为算术右移。

1.5 测试与分析

1.5.1 8 位串行可控加减法器的测试

对于 8 位串行可控加减法器，分别对于加法以及减法、有进位/借位以及无进位/借位进行测试，测试的期望结果与实际结果如表 1-3 所示，其中所有数均以 8 位无符号整数表示。从表中可以看出，期望的结果与实际结果完全一致。

表 1-4 测试预期结果与实际结果

#	X	Y	运算	期望结果	期望进位	实际结果	实际进位
1	12	23	加	35	○	35	○
2	168	192	加	104	●	104	●
3	7	2	减	5	○	5	○
4	32	128	减	160	●	160	●

1.5.2 32 位快速加法器的测试

由于 32 位加法器依次向下依赖 16 位快速加法器、4 位快速加法器以及 CLA74182，而这些子模块在编写过程中就已经进行了测试，这些模块中唯一需要对于 ALU 以及对外提供服务的模块就是 32 位快速加法器，此处仅对于 32 位快速加法器进行测试，测试时对于加法运算结果的正确性以及溢出结果的正确性进行测试，测试的预期结果与实际结果如表 1-4 所示，其中所有的整数为 32 位无符号整数表示。从测试结果中可以看出，在有溢出和无溢出的情况下运算结果均与预期结果一致。

表 1-5 32 位快速加法器测试结果

#	X	Y	期望结果	期望溢出	实际结果	实际溢出
1	0	3	3	○	3	○
2	1040	0	1040	○	1040	○
3	32784	32768	65552	○	65552	○
4	2147483647	5	2147483642	●	2147483642	●

1.5.3 ALU 测试

ALU 的测试过程较为繁琐：首先对于每一个功能取 3 组不同的输入进行测试，分别观察得到的结果是否与期望的结果相同，由于测试项过多，此处不再给出表格，而对于功能测试的结果为：对于所有给定的输入，测试的结果均与期望的结果相同。对于算术右移运算特别测试了 X 为整数和负数的不同情况，对于乘法运算特别测试了运算结果少于 32 位和多余 32 位的情况，而对于除法则特别验证了除数为 0、余数为 0 和不为 0 的情况。需要说明的是，当除数为 0 时，logisim 自带的除法模块得到的商为被除数而余数为 0，预期结果采用了这一规则，在进行这些测试的过程中一并观察了 Equal 的输出结果，发现结果正确。此外，对于加减法运算的有符号溢出和无符号溢出进行了特别的测试。这一项的正确性是 CPU 实验能够顺利进行的基础，因此进行了特别的测试，测试结果如表 1-5 所示，其中所有整数为 32bit 无符号正整数表示 3。表中分别对于 32 位的加法和减法、有符号溢出以及无符号溢出的所有组合（共 $2 \times 2 \times 2 = 8$ 种）进行了测试。从测试的预期结果与得到的实际结果中可以看出，对于所有可能的组合，实际结果均与预期结果一致，因此对于溢出的判断是正确的。

华中科技大学课程实验报告

表 1-6 溢出信号测试用例

#	A	B	运算	有符号/期望	无符号/期望	有符号/实际	无符号/实际
1	3	3	加	○	○	○	○
2	3221225472	107371824	加	○	●	○	●
3	107371824	107371824	加	●	○	●	○
4	2147483648	2147483648	加	●	●	●	●
5	3	2	减	○	○	○	○
6	0	1	减	○	●	○	●
7	2147483648	1610612736	减	●	○	●	○
8	5	2147483648	减	●	●	●	●

1.5.4 补码一位乘法器的测试

在补码一位乘法器的测试中，输入不同的 X 和 Y 的值，考虑 X 和 Y 分别为正数和负数的情况，并按下 Ctrl-K 进行模拟，并等待运算结束。然后对比运算结果与期望结果（由计算器得到）。期望结果与实际结果如表 1-6 所示，从表中可以看出，测试结果与实际结果在 X 和 Y 分别为正数和负数的各种组合下完全一致，因此补码一位乘法器功能完好。

表 1-7 补码一位乘法器测试结果

#	X	Y	期望结果	实际结果
1	3	11	33	33
2	4	-13	-52	-52
3	-124	19	-2356	-2356
4	-12	-12	144	144

2 CPU 设计实验

2.1 设计要求

利用运算器实验，存储系统实验中构建的运算器、寄存器文件、存储系统等部件以及 Logisim 中其他功能部件构建一个 32 位 MIPS CPU 单周期处理器，该处理器应支持基础指令集中列出的所有指令，包括 add, addi, addiu, addu, and, andi, sll, sra, srl, sub, or, ori, nor, lw, sw, beq, bne, slt, slti, sltu, j, jal, jr 和 syscall。其中 syscall 指令实现为：当 \$v0 为 10 时停机，否则数码管显示 \$a0 的值。另外还必须支持扩展指令集中的 2 条 C 类运算指令，1 条 M 类存储指令，1 条 B 类分支指令，分别为 srlv（逻辑可变右移），sltiu（小于立即数置 1），lbu（加载字节）以及 bltz（小于 0 转移）。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 应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）。

2.2 方案设计

经过初步的自顶而下的考虑，整个 CPU 从整体上来看应该划分为指令存储器、寄存器、运行内存、ALU、PC（程序计数器）控制、控制器、ALU 输入控制器。然后采取工程化的方法来构建 PC 控制、控制器、ALU 输入控制器以及各个部件之间的联系。

2.2.1 数据来源

首先确定各个主要部件的数据来源，其中：PC 控制器的数据来源有寄存器 \$rS 以及指令中的立即数，寄存器写入数据 \$rW 的数据来源有 ALU 的运算结果 R，以及运行内存的输出（M 型指令使用），ALU 数据输入的来源有寄存器 \$rS，寄存器 \$rT 以及程序计数器 PC、以及指令中的立即数，运行内存的数据来源有指令中的地址、寄存器 \$rS（作为基址）以及寄存器 \$rT（作为要存放的数据），syscall controller 的数据来源有寄存器 \$rS 和寄存器 \$rT，用于判断和显示。这些数据来源均由表 2-1 以及表 2-2 中的功能描述部分得出，详见 2.2.2 控制器设计章节。

华中科技大学课程实验报告

2.2.2 控制器设计

表 2-1 MIPS 指令与 ALU 指令转换表

#	指令	高 6 位	低 6 位	AluOp	AluF	AluX	AluY	功能描述
1	addu	000000	100000	0101	add	\$rs	\$rt	\$rd<-\$rs+\$rt
2	addi	001000	-	0101	add	\$rs	i	\$rt<-\$rs+i
3	addiu	001001	-	0101	add	\$rs	i	\$rt<-rs+i
4	addu	000000	100001	0101	add	\$rs	\$rt	\$rd<-\$rs+\$rt
5	and	000000	100100	0111	and	\$rs	\$rt	\$rd<-\$rs&\$rt
6	andi	001100	-	0111	and	\$rs	i	\$rt<-\$rs&i
7	sll	000000	000000	0000	sl	\$rt	sa	\$rd<-\$rt<<sa
8	sra	000000	000011	0001	sra	\$rt	sa	\$rd<-\$rt>>>sa
9	srl	000000	000010	0010	srl	\$rt	sa	\$rd<-\$rt>>sa
10	sub	000000	100010	0110	sub	\$rs	\$rt	\$rd<-\$rs-\$rt
11	or	000000	100101	1000	or	\$rs	\$rt	\$rd<-\$rs\$rt
12	ori	001101	-	1000	or	\$rs	i	\$rt<-\$rsi
13	nor	000000	100111	1010	nor	\$rs	\$rt	\$rd<-\$rs<nor>\$rt
14	lw	100011	-	0101	add	\$rs	i	\$rt<-offset(base),a.k.a:i+\$rs
15	sw	101011	-	0101	add	\$rs	i	offset(base)<-\$rt,a.k.a:i+\$rs
16	beq	000100	-	-	-	\$rs	\$rt	if:\$rs==\$rt,PC+=offset
17	bne	000101	-	-	-	\$rs	\$rt	if:\$rs!=\$rt,PC+=offset
18	slt	000000	101010	1011	lt	\$rs	\$rt	\$rd<-(\$rs<\$rt)
19	slti	001010	-	1011	lt	\$rs	i	\$rt<-(\$rs<i)
20	sltu	000000	101011	1100	ltu	\$rs	\$rt	\$rd<-(\$rs<\$rt)
21	j	000010	-	-	-	-	-	j->target
22	jal	000011	-	0101	add	PC	4	\$31<-PC+4,j->label
23	jr	000000	001000	-	-	-	-	PC<-\$rs
24	sys	000000	001100	-	-	-	-	syscall

华中科技大学课程实验报告

表 2-2 MIPS 指令与 ALU 指令转换表 (CCMB)

#	指令	高 6 位	低 6 位	AluOp	AluF	AluX	AluY	功能描述
C	srlv	000000	000110	0010	srl	\$rt	\$rs	\$rd<-\$rt>>\$rs
C	sltiu	001011	-	1011	lt	\$rs	i	\$rt<-(\$rs<i)
M	lbu	100100	-	0101	add	\$rs	i	\$rt<-offset(base),a.k.a:i+\$rs
B	bltz	000001	-	1011	lt	\$rs	0	if:\$rs<0,PC+=offset

整个控制器的设计与完成，乃至整个 CPU 设计的核心部分均是基于表 2-1 以及表 2-2 的。在这两张表中，描述了 MIPS 指令与 ALU 指令的对应关系，MIPS 指令所执行的动作以及 ALU 采取的运算符。表中，第一栏为指令的编号，接下来的三栏为 MIPS 指令最高的 6 位和最低的 6 位的值，所有需要完成的 MIPS 指令均由这 12 段控制语句的意义。AluOp 和 AluF 栏描述每一个 MIPS 语句对应的 ALU 控制码以及所需要使用的 ALU 功能，AluX 和 AluY 则描述了 Alu 两个输入的来源，其中 \$rs,\$rt,\$rd 从寄存器文件中取得，而 i 为指令中的立即数，从指令中取得。功能描述描述了 MIPS 指令所执行的功能，从中可以看出整个 CPU 完成这一条指令的数据通路。根据这两张表构建 CPU 的控制器 controller，其结构如图 2.1 所示。

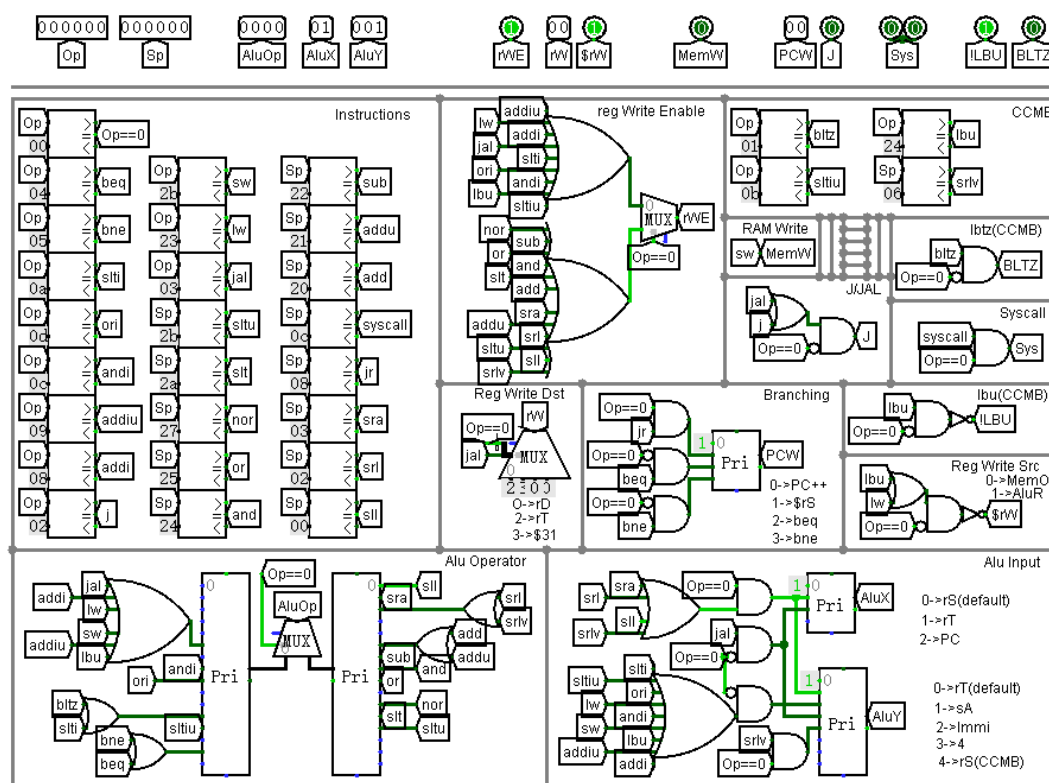


图 2.1 控制器结构

整个控制器有 2 个输入，12 个输出，其功能即为通过对 MIPS 指令的分析产生对于其它所有部件所需要的控制信号。输入 Op 和 Sp 即为 MIPS 指令中的高六位和低六位，首先在左上角标有 instruction 和右上角的 CCMB 区域内对于这两个值进行比较，以生成各个指令信号，需要注意的是这时生成的指令信号并未对于高 6 位为 0 和不不为 0 的情况进行区分，因此可能产生冲突(同时两个指令信号为高电平)，因此加入 Op==0 信号，便于在接下来的部分中对这两个部分分开进行处理。

图中标有 Alu Operator 的区域用于生成 Alu 的操作码。其原理是通过优先编码器将 MIPS 的指令按照表 2-1 和表 2-2 映射到对应的 Alu 指令上去。Alu Input 区域做的操作与之类似，只不过其输出的为 Alu 的两个输入的值的来源控制信号。余下的部分均为对于外部其它设备的控制信号，其中 Reg Write Enable 和 Reg Write Src 控制寄存器文件的写入使能以及数据来源，Reg Write Dst 控制写入哪一个寄存器，RAM Write 控制运行内存的写入，J/JAL, Branching 共同控制程序计数器 PC 的数据来源，只有 Ibu 较为特殊，需要对于运行内存的输出在外部进行处理，因此单独为其设立一个输出信号。这些信号的产生原理与 AluOp 的产生原理均相同，只不过部分简单的信号未使用优先编码器而是直接通过多路选择器和门电路进行映射。

2.2.3 PC 控制器、ALU 控制器以及 Syscall 控制器设计

为了便于维护与更改，对于 PC 的控制，ALU 输入的控制以及 Syscall 的控制均分别封装成为了一个规模较小的模块。图 2.2、图 2.3、图 2.4 分别展示了 PC 控制模块、ALU 控制模块以及 Syscall 控制模块的结构。

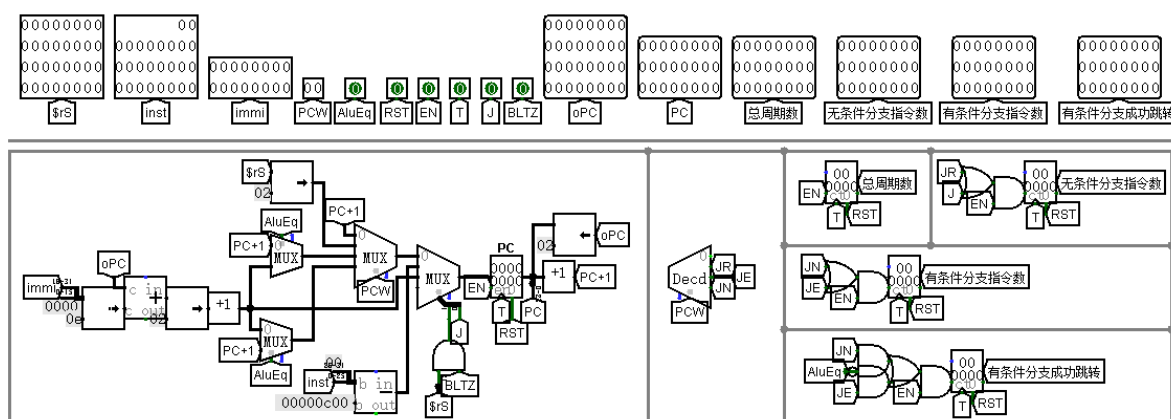


图 2.2 PC 控制模块

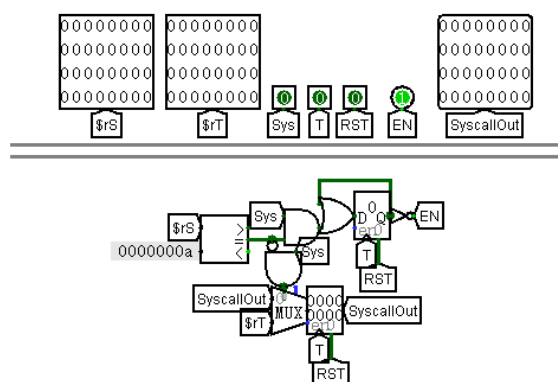


图 2.4 Syscall 控制模块

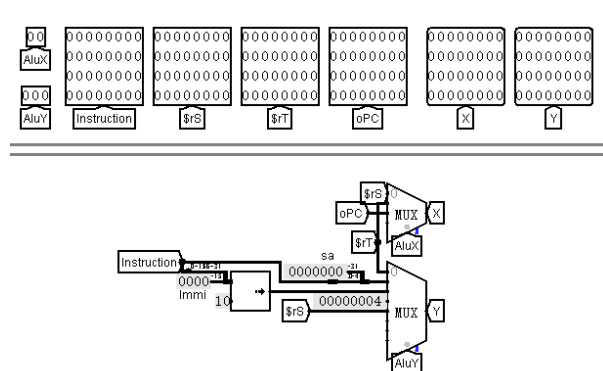


图 2.3 Alu 输入控制模块

其中，PC 控制模块通过输入的 \$rS 以及 inst、immi 来计算可能的跳转地址，其中 inst 以及 immi 直接从 MIPS 指令中取得，而 PC 的写入控制则由 PCW 与 J 来控制，具体的控制码以及含义在图 2.1 中已经给出。通过多路选择器来选择数据的来源并存入 PC。

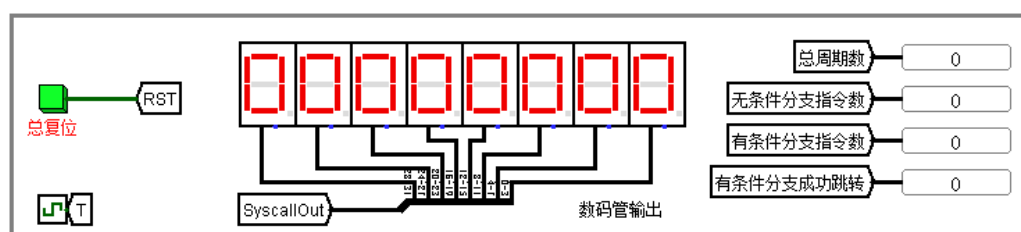
Syscall 控制模块则通过输入的 Sys 信号以及 \$rS, \$rT 的值来控制执行的功能。当 Sys 信号为高电平时，CPU 的实现将保证 \$rS 为 \$v0 而 \$rT 为 \$a0。如果 \$v0 的值为 10 则将 \$a0 的值存入寄存器中，从而刷新显示输出，否则将 EN 信号永久置为 0（只有 RST 信号能将其重新置 1）EN 信号为整个 CPU 其它部件的使能信号，正常情况下为 1，为 0 时 CPU 停止工作，从而完成 halt 指令。

Alu 输入模块则是简单的通过控制器给出的 AluX 以及 AluY 控制码控制 ALU 的输入来源并进行显示输出。值得数的是当需要输出立即数时需要进行位扩展，这里通过在原立即数后先补 0 至 32 位然后算是右移实现。

2.2.4 模块整合

在设计完成所有的模块之后需要进行最后的整合工作，也就是将模块的输入与输出按照事先规划好的控制信号以及模块间关系连接起来，连接完成后如图 2.5 所示。对于寄存器输入输出的选择、运行内存输入的地址计算以及运行内存输出的选择由于过于简单所以没有封装为独立的模块，这些选择信号均由多路选择器实现，多路选择器的控制信号来源均为控制器 controller。最后，向指令存储器中导入预先由 Mars 编译号的程序即可运行。

华中科技大学课程实验报告



可以通过项目增加Logisim库的方式将前几次实验完成的电路作为子电路引入到本电路中使用
数码管输出应该用寄存器锁存，否则显示数据只能一闪而过 总复位信号应将除ROM以外所有存储组件全部清零

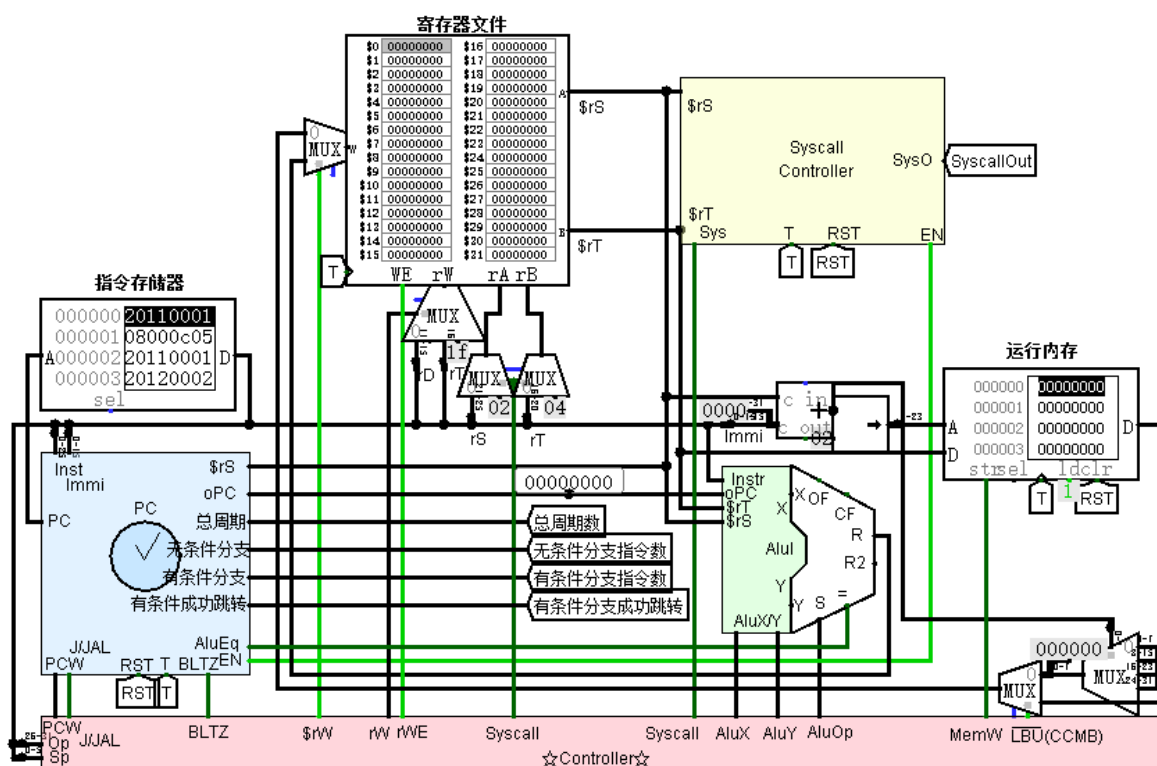


图 2.5 CPU 结构

2.3 实验步骤

- (1) 首先按照题目要求对于整个系统的架构进行分析，并得出表 2-1 以及表 2-2，作为整个实验构建 CPU 的核心与基础。
- (2) 根据所得出的数据映射表构建各个模块，并在每个模块构建完成后对其进行单元测试，以保证各个模块独自工作时功能是正常的。
- (3) 将各个模块按照规划好的模块间关系以及各个信号的走向进行连接，以构建一个完整的 CPU。
- (4) 在构建好的 CPU 上加载程序并运行，对于执行错误的地方进行单步调试并分析错误原因，然后对于电路进行修改直到没有任何错误出现。

2.4 故障与调试

2.4.1 程序运行不能停止问题

故障现象：程序执行完成后不能停止，而是一直执行后面的指令。

原因分析：修改前的 Syscall 模块如图 2.6 所示。在 \$v0 的值为 10(0xa)的时候确实将 EN 所连接的寄存器置为了 0，但是在下一时钟周期由于程序已经不再为 Syscall 语句，因此寄存器中的 1 又重新变为了 0，要解决这一方案必须将寄存器变为一次触发永久锁定的模式。

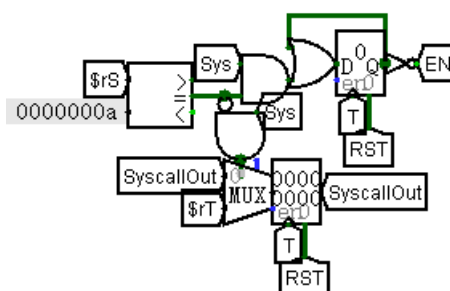
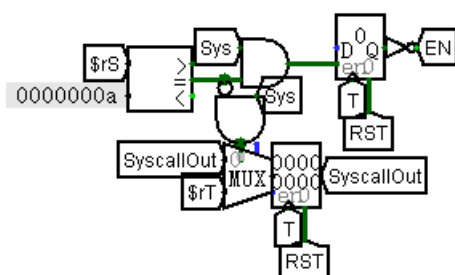


图 2.6 修改前的 Syscall 模块

图 2.7 修改后的 Syscall 模块

解决方案：如图 2.7 所示，在 EN 所连接的寄存器的输出段引出一个信号并与原输入进行与运算后重新接入原输入位置，这样在此寄存器值变为 1 后会将自身锁定，不会变回 0。

2.4.2 程序分支指令数统计错误问题

故障现象：程序运行完成后总周期数正确，但是无条件分支指令数统计错误。

原因分析：修改前的统计电路如图 2.8 所示，由于将无条件分支信号直接接在了计数器的时钟端，导致若出现两次连续的跳转时，无条件分支计数器只进行了一次计数，因此统计结果比正确结果少。

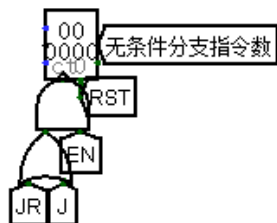


图 2.8 更改前的无条件分支计数器

图 2.9 更改后的无条件分支计数器

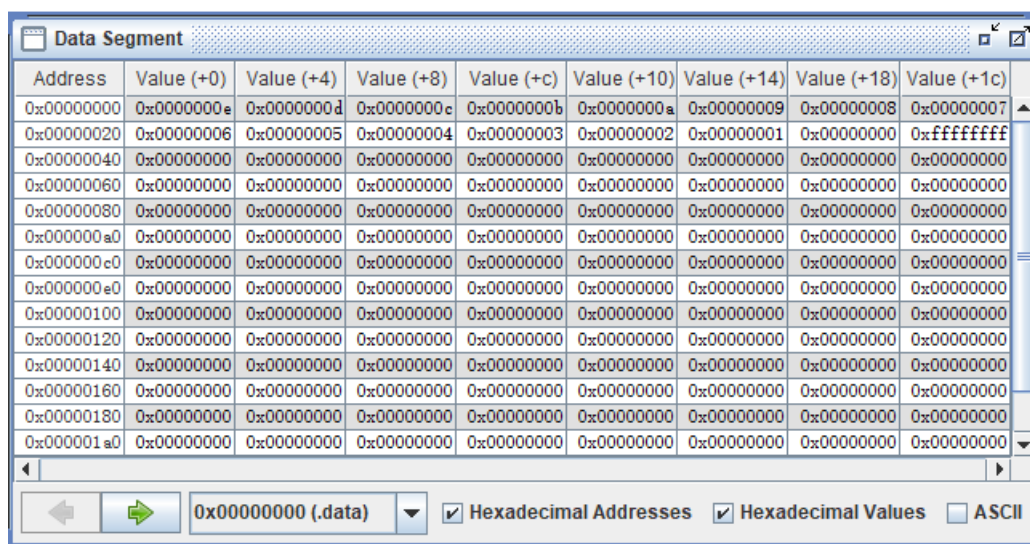
华中科技大学课程实验报告

解决方案：使用时钟驱动计数器，无条件分支计数器信号只用于计数器的使能端，更改后的电路如图 2.9 所示。

2.5 测试与分析

2.5.1 基础指令的测试

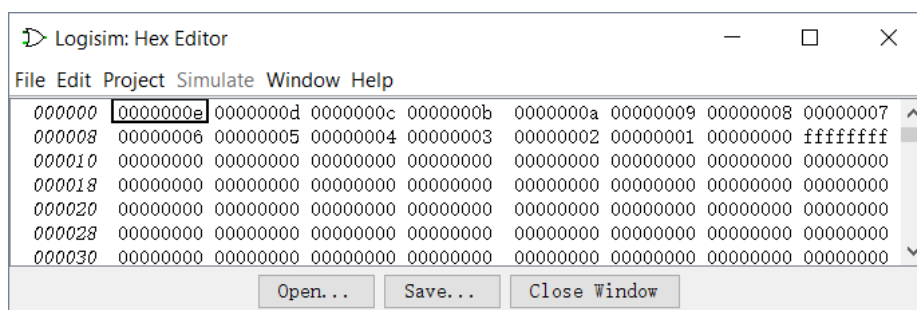
对于基础指令而言。首先在 Mars 中对于 benchmark.asm 进行仿真并观察程序的结果，然后使用自己的 CPU 进行模拟，并将模拟的结果与 Mars 中的结果进行比对。执行完成后，Mars 中程序数据段的值如图 2.10 所示，可以看出，内存中地址从 0x00 到 0x38 中分别为 0x00 到 0x0E 的倒序排列。0x3C 中存储的为 0xffffffff (-1)。



Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x0000000e	0x0000000d	0x0000000c	0x0000000b	0x0000000a	0x00000009	0x00000008	0x00000007
0x00000020	0x00000006	0x00000005	0x00000004	0x00000003	0x00000002	0x00000001	0x00000000	0xffffffff
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000120	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000001a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 2.10 程序执行完成后数据段的值

在 logisim 中进行仿真，在程序结束之后直接对于运行内存中的值进行查看，结果如图 2.11 所示。可以看出，内存的分布与 Mars 仿真的结果完全相同。



Address	Value	Value	Value	Value	Value	Value	Value	Value
000000	0000000e	0000000d	0000000c	0000000b	0000000a	00000009	00000008	00000007
000008	00000006	00000005	00000004	00000003	00000002	00000001	00000000	ffffff
000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000018	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000028	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
000030	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

图 2.11 logisim 仿真完成后内存中的值

华中科技大学课程实验报告

此外，观察数码管的输出并将其与 Mars 的 RUN I/O 进行对比，输出的值与 Mars 中完全一致。

最后，观察程序指令统计，logisim 仿真执行完成指令统计结果如图 2.12 所示，从图中可以看出此 CPU 实现统计的总周期数为 1546 条，无条件分支指令数为 38 条，有条件分支指令数为 338 条，有条件分支成功跳转为 276 条。

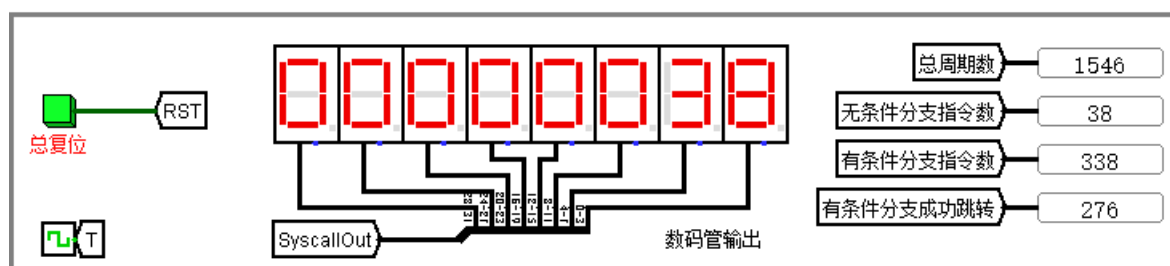


图 2.12 logisim 指令统计结果

打开 Mars 中的 Instruction Statistics 工具并连接到 Mars，重新执行程序。程序执行完成后指令统计的结果如图 2.13 所示，可以看出统计结果与 logisim 中的统计结果相同。对于有条件分支成功跳转的统计，打开 BHT Simulator，连接到 Mars 并重新执行程序，执行后将 Correct 栏的数字加起来，与 logisim 吗相同，均为 276，说明 CPU 工作正常。

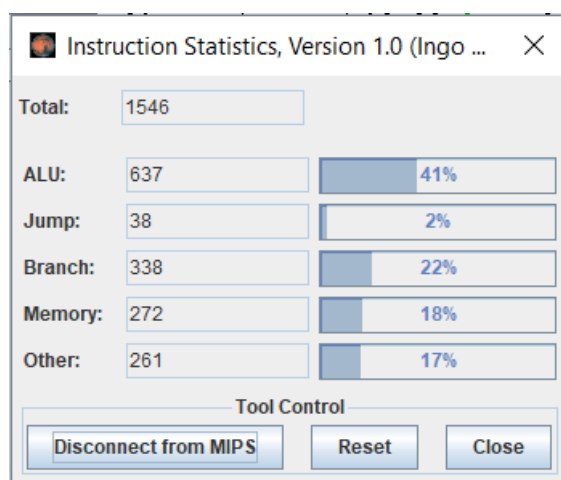


图 2.13 Mars 中指令统计结果

2.5.2 CCMB 指令的测试

对于 CCMB 指令而言，首先编写一个简短的测试程序，分别对于 srlv, sltiu, lbu 和 bltz 进行测试，然后在 Mars 中进行模拟，记录下每一语句执行后寄存器的变化、内存

华中科技大学课程实验报告

的变化以及显示输出的变化。然后在 logisim 中通过控制时钟信号进行单步模拟，同时观察每一个语句执行完成后寄存器、内存以及显示输出的变化，发现所有的变化与 Mars 中模拟的完全一致，此时可以确信对于 CCMB 指令本 CPU 进行了正确的实现。

3 总结与心得

3.1 实验总结

- 1) 设计并实现了八位串行可控加减法器。
- 2) 设计并实现了 4 位先行进位电路。
- 3) 设计并实现了 4 位快速加法器。
- 4) 设计并实现了 16 位快速加法器。
- 5) 设计并实现了 32 位快速加法器。
- 6) 利用上述组件设计了并实现了 ALU。
- 7) 设计并实现了补码一位乘法器。
- 8) 实现了阵列乘法器。
- 9) 设计并实现了单周期支持四条额外 CCMB 指令的 CPU。
- 10) 完成了所有电路的错误调试。
- 11) 修正了所有可能出现的错误。
- 12) 完成了令 CPU 的实际模拟频率能够达到 800Hz 的性能调优。
- 13) 完成了 CPU 各模块的封装以及单元测试。
- 14) 完成了给定程序以及自编程序在实现的 CPU 上的模拟。

3.2 实验心得

- 1) 在电路设计与实现的过程中需要极度细心, 否则将会导致不能预料的各种错误, 尤其是边界情况错误。
- 2) 需要熟练 logisim 等工具并将之运用于课堂学习的内容上, 熟练使用各种快捷键以及脚本, 否则完成实验会遇到少许困难。
- 3) 在使用 logisim 时使用触摸板而不是鼠标将给自己带来不必要的麻烦。
- 4) 尽量使用已有的模块、已被验证的工具解决问题而不是凡事都自己造工具想办法解决问题效率更高。
- 5) 进行任何实践前都需要有良好的规划, 对于自己、实验过程以及实验目标的良好认知, 不要盲目自大, 同时需要按照计划行事。

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] 袁春风编著. 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: _____

二、对课程实验的学术评语（教师填写）

三、对课程实验的评分（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: _____ 2018-01-12