第4章

常用电路、算法及电路实现

在CPU中，主要的算术运算和逻辑运算需要在算术逻辑单元ALU(arithmetic logic unit)中完成，例如加法、减法、逻辑与、逻辑或等。

本章将介绍CPU设计中常用的电路，介绍各种ALU算法，包括加、减、乘、除、逻辑运算、移位运算等，以及这些算法的逻辑电路实现方法。在本章中，各种算术运算(加、减、乘、除、算术移位等)使用的操作数都用补码表示。虽然补码数据表示方法不利于乘、除运算，但却比较适合加、减、算术移位等运算。

4.1 逻辑运算器

在逻辑运算中，操作数被视为逻辑数。所谓的逻辑数就是没有符号，每个位的地位都是相同的。“1”代表逻辑真，“0”代表逻辑假。在MIPS中，定义了逻辑与(AND)、逻辑或(OR)、逻辑或非(NOR)和逻辑异或(XOR)。

4.1.1 逻辑与

逻辑与操作为两个操作数的按位与。例如

```
          1100110111101110
AND       1010100110111100
-----------------------------
          1000100110101100
```

逻辑与的逻辑表达式为：Z＝A·B或者Z＝A B。图4.1给出32位逻辑与的逻辑电路图。

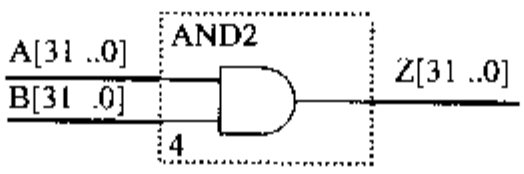


图4.1 逻辑与

4.1.2 逻辑或

逻辑或操作为两个操作数的按位或。例如

```
          1100110111101110
OR        1010100110111100
----------------------------
          1110110111111110
```

逻辑或的逻辑表达式为：Z＝A＋B。图 4.2 给出 32 位逻辑或的逻辑电路图。

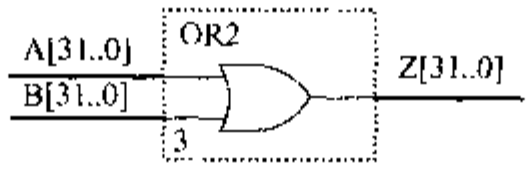


图 4.2 逻辑或

4.1.3 逻辑或非

逻辑或非操作为两个操作数的按位或然后非。例如

```
          1100110111101110
NOR       1010100110111100
----------------------------
          0001001000000001
```

逻辑或非的逻辑表达式为：$Z=\overline{A+B}$。图 4.3 给出 32 位逻辑或非的逻辑电路图。

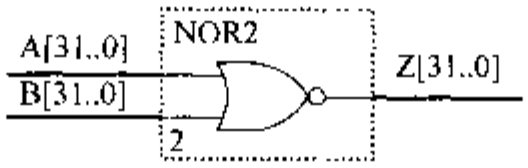


图 4.3 逻辑或非

4.1.4 逻辑异或

逻辑异或操作为两个操作数的按位异或。例如

```
          1100110111101110
XOR       1010100110111100
----------------------------
          0110010001010010
```

逻辑或非的逻辑表达式为：$Z=A\oplus B$。图 4.4 给出 32 位逻辑或非的逻辑电路图。

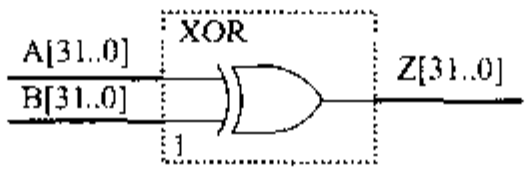


图 4.4　逻辑异或

4.2　常用电路

4.2.1　译码器

译码器的功能是接收一个二进制数值，然后将该数值进行译码，即把二进制代码转换成特定的信号输出。若译码器有输入为 n 位二进制数值，则有 2^n 个输出。每一个输出对应一个二进制编码。例如，对于一个 3-8 译码器，如果输入为 101，则第 5 个输出为高，其他输出都为低。下面以 3-8 译码器作为例子说明译码器的设计问题。3-8 译码器涉及以下逻辑变量。表 4.1 为 3-8 译码器的真值表。

(1) 二进制数值输入 A0、A1、A2；

(2) 使能输入 E；

(3) 译码输出 D0、D1、D2、D3、D4、D5、D6、D7。

表 4.1　译码器真值表

输入				输出							
A2	A1	A0	E	D7	D6	D5	D4	D3	D2	D1	D0
×	×	×	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0
1	0	1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

由表 4.1，可以得出 3-8 译码器的逻辑表达式为：

$$D0 = E\ \overline{A2}\ \overline{A1}\ \overline{A0} \qquad D1 = E\ \overline{A2}\ \overline{A1}\ A0$$

$$D2 = E\ \overline{A2}\ A1\ \overline{A0} \qquad D3 = E\ \overline{A2}\ A1\ A0$$

$$D4 = E\ A2\ \overline{A1}\ \overline{A0} \qquad D5 = E\ A2\ \overline{A1}\ A0$$

$$D6 = E\ A2\ A1\ \overline{A0} \qquad D7 = E\ A2\ A1\ A0$$

根据逻辑表达式，可以得出图 4.5 所示的逻辑电路图。

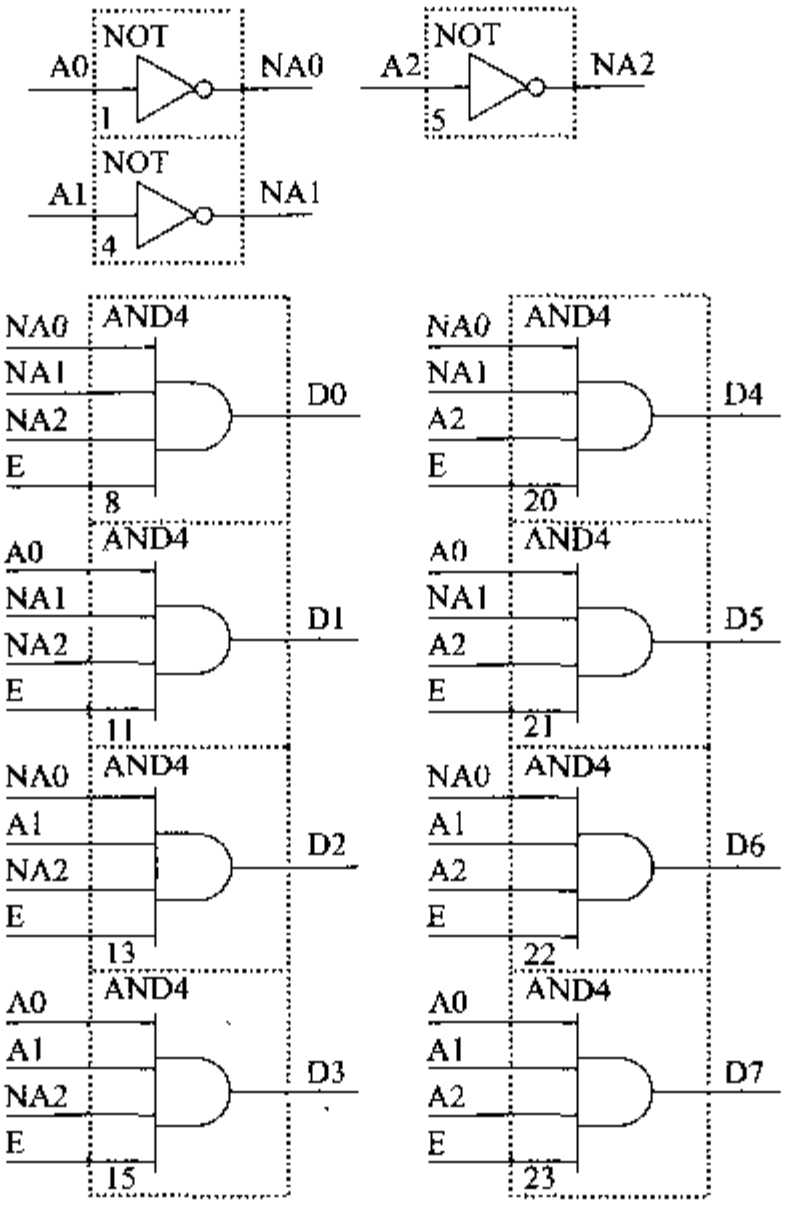


图 4.5　译码器逻辑电路图

4.2.2　数据选择器

数据选择器又称为多路选择器。数据选择器的功能是从多个输入数据中选择一个作为输出。下面以 32 位 2 选 1 数据选择器为例介绍数据选择器的设计。首先设计 1 位 2 选 1 数据选择器。1 位 2 选 1 数据选择器涉及的逻辑变量包括：

(1) 2 个 1 位输入 A0、A1；

(2) 选择子 S；

(3) 选择输出 Y。

表 4.2 为 1 位 2 选 1 数据选择器的真值表。

表 4.2　1 位 2 选 1 数据选择器真值表

S	Y
0	A0
1	A1

由真值表 4.2，可以得到 1 位 2 选 1 数据选择器的逻辑表达式为

$$Y=\overline{S}\,A0+S\,A1$$

根据逻辑表达式，可以得到图 4.6 所示的逻辑电路图。

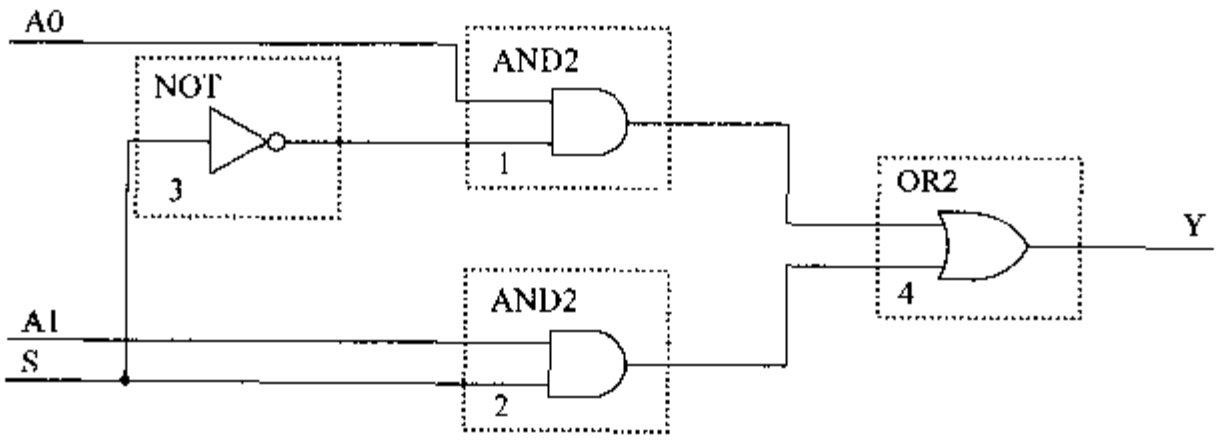


图 4.6　1 位 2 选 1 数据选择器逻辑电路图

可以使用 8 个 1 位 2 选 1 数据选择器来搭建 8 位 2 选 1 数据选择器。逻辑电路图如图 4.7 所示。

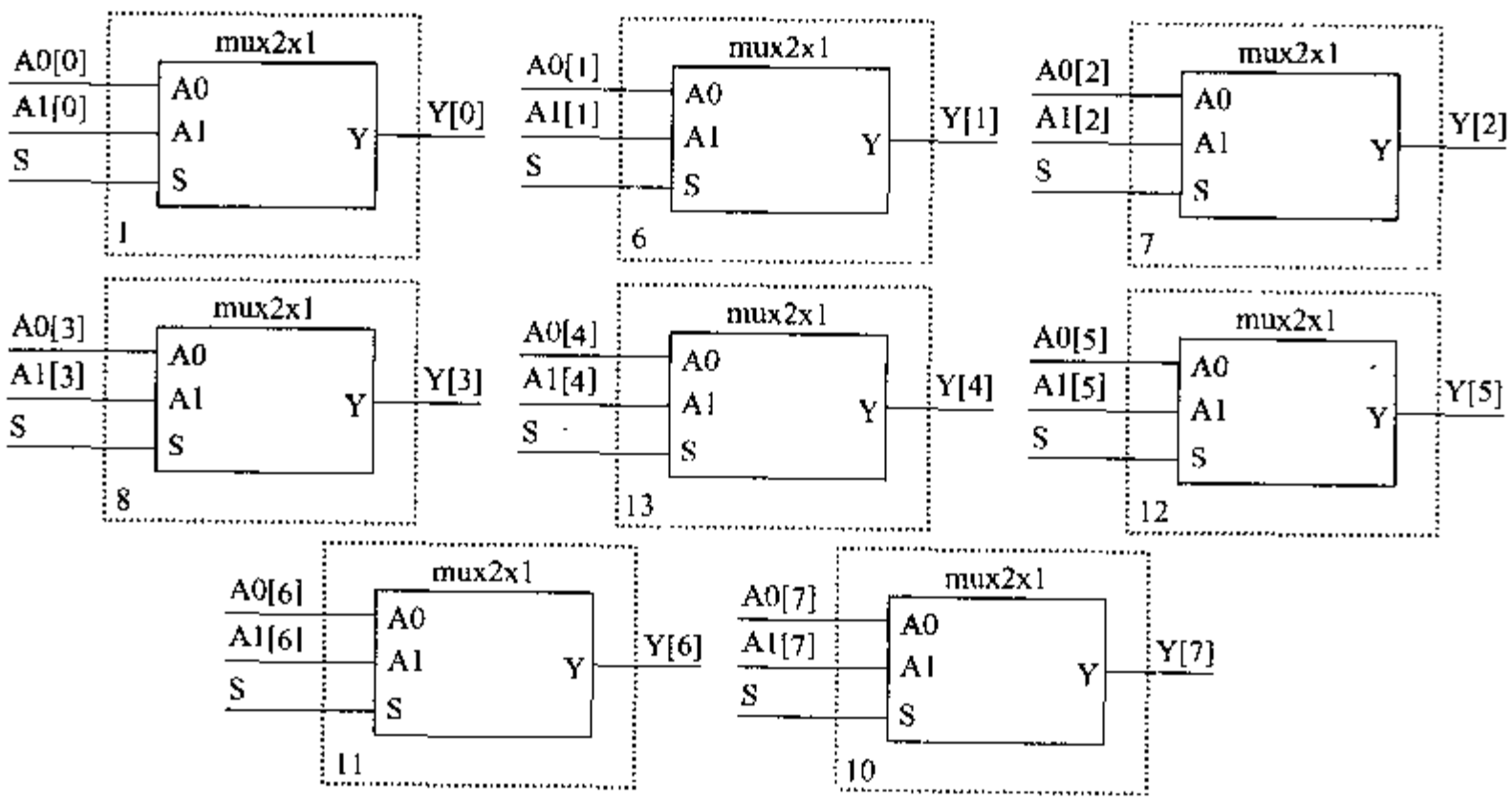


图 4.7　8 位 2 选 1 数据选择器逻辑电路图

最后，可以使用 4 个 8 位 2 选 1 数据选择器来搭建 32 位 2 选 1 数据选择器。逻辑电路图如图 4.8 所示。

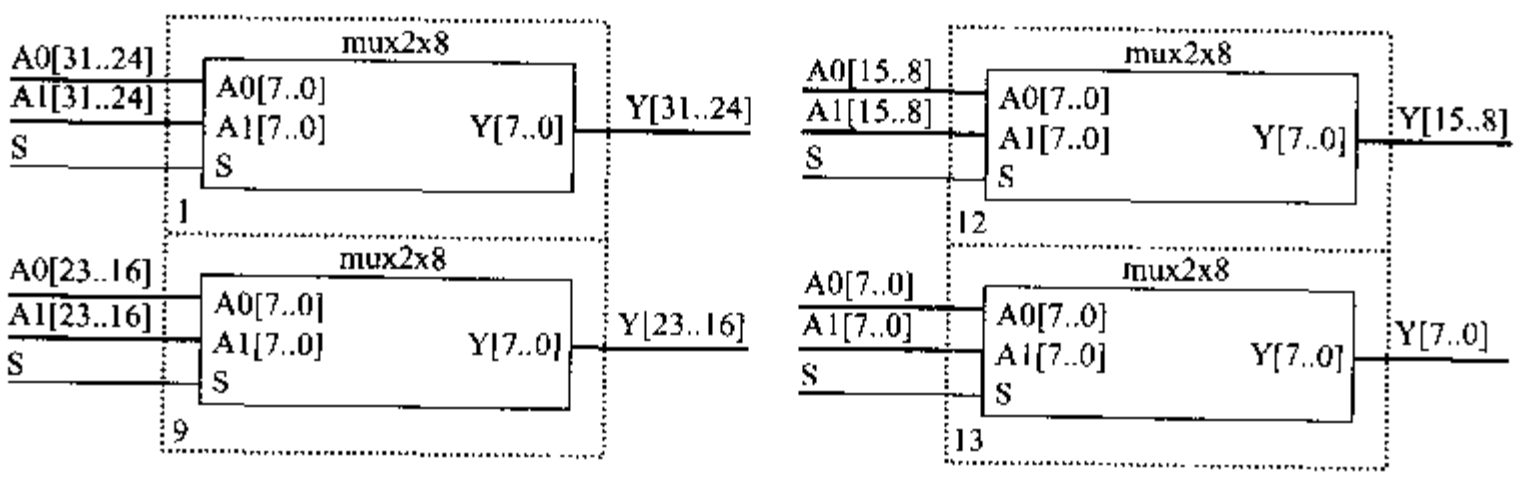


图 4.8　32 位 2 选 1 数据选择器逻辑电路图

相似地，可以使用 3 个 32 位 2 选 1 数据选择器来搭建 32 位 4 选 1 数据选择器。逻辑电路如图 4.9 所示。

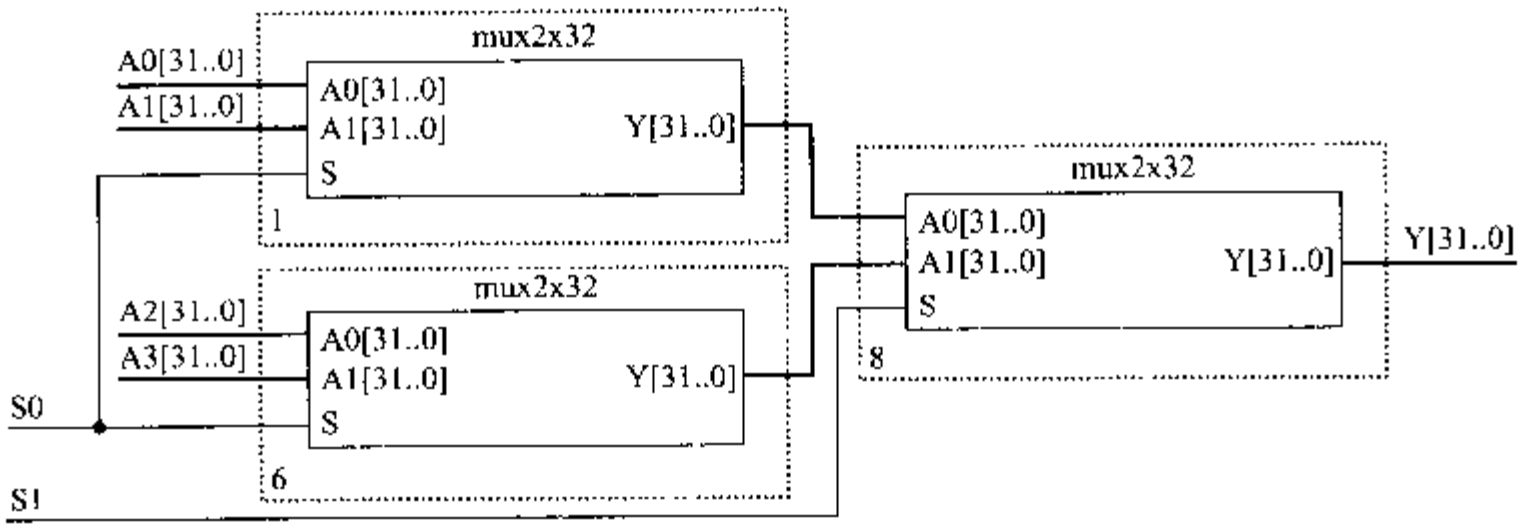


图 4.9　32 位 4 选 1 数据选择器逻辑电路图

当然，也可以搭建更大规模的数据选择器，例如 32 位 8 选 1 数据选择器、32 位 32 选 1 数据选择器等。

4.3　加减法器

加减法器是执行二进制加法运算和减法运算的逻辑部件。加减法器是 CPU 运算器中基本的逻辑部件。下面，将从简单到复杂地介绍加减法器的设计方法。

4.3.1　32 位加法器

32 位加法器的功能为执行两个 32 位操作数的加法操作：

$$S \leftarrow A + B$$

涉及加法器的 MIPS 指令有：ADD，ADDI，ADDU，ADDIU。先设计 1 位数的加法器。加法器有半加器和全加器之分。

1. 半加器

不考虑低位的进位，只对两个二进制 1 位数相加，得到两个加数之和以及向高位的进位的加法器，称为半加器。表 4.3 是半加器的真值表。

表 4.3　半加器真值表

a	b	co	s	说　明
0	0	0	0	0+0=00
0	1	0	1	0+1=01
1	0	0	1	1+0=01
1	1	1	0	1+1=10

表 4.3 中的 a,b 为 2 个 1 位的加数,s 为半加和,co 为向高位的进位。根据真值表,可以得到半加器的逻辑表达式。

$$s=\bar{a}\,b+a\,\bar{b}=a\oplus b$$

$$co=ab$$

由逻辑表达式,可以得出如图 4.10 所示的逻辑电路图。

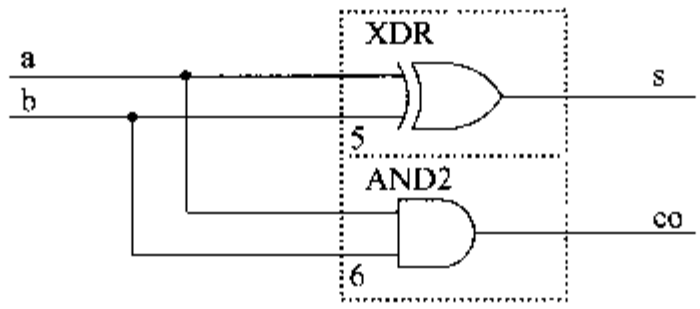


图 4.10　半加器逻辑电路图

2. 全加器

全加器是在半加器的基础上,考虑了低位向本位进位的加法器。表 4.4 是全加器的真值表。

表 4.4　全加器真值表

a	b	ci	co	s	说　明
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+0+1=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

其中 a,b 为 2 个 1 位的加数,ci 为来自低位的进位,s 为和,co 为向高位的进位。根据真值表,可以得出全加器的逻辑表达式。

$$s=a\,\bar{b}\,\bar{ci}+\bar{a}b\,\bar{ci}+\bar{a}\,\bar{b}\,ci+a\,b\,ci$$

$$co=a\,b+a\,ci+b\,ci$$

由逻辑表达式,可以得出如图 4.11 所示的逻辑电路图。

半加器和全加器都只能进行一位加法。但是,它们是组成多位加法器的基本单元。根据电路结构的不同,多位加法器可以分为串行加法器和并行加法器两种。

3. 32 位串行进位加法器

32 位串行进位加法器由 32 个全加器组成。低位全加器产生的进位要依次串行地向高位进位。32 位串行进位加法器的逻辑表达式为

$$s_i=a_i\overline{b_i}\,\overline{c_i}+\bar{a}_i\,b_i\overline{c_i}+\overline{a_i}\,\overline{b_i}\,c_i+a_i\,b_i\,c_i$$

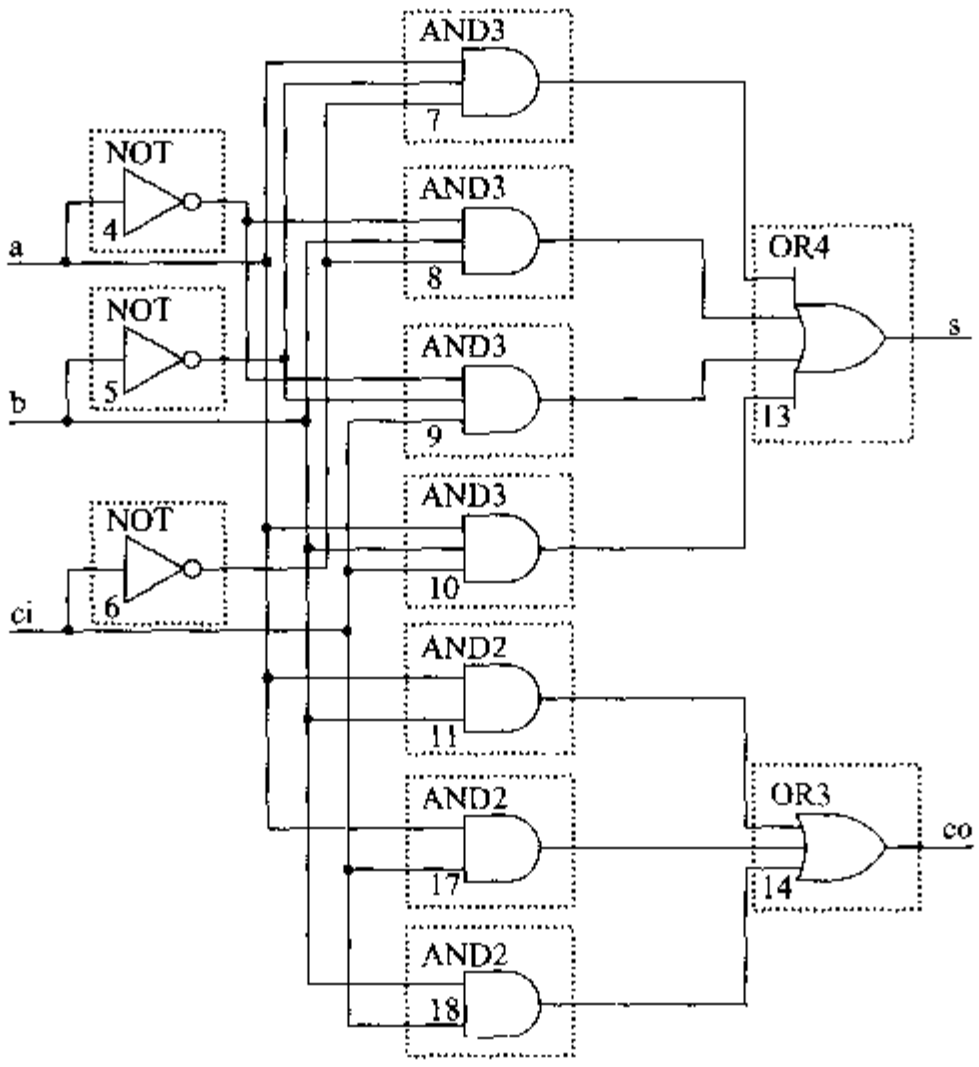


图 4.11 全加器逻辑电路图

$$c_0 = 0$$

$$c_{i+1} = a_i\, b_i + a_i\, c_i + b_i\, c_i; \quad i = 0, 1, \cdots, 31$$

其中，a_i 和 b_i 为两个加数的第 i 位，c_i 为第 i 位来自低位的进位，s_i 为和的第 i 位，c_{i+1} 为第 i 位向高位的进位。由逻辑表达式，可以得到如图 4.12 所示的 32 位串行进位加法器的逻辑电路图。

显然，串行进位加法器的电路比较简单，使用的逻辑器件比较少。但是，串行进位加法器每一位的和以及向高位进位，都要等到所有的低位加法完成后，得到低位进位的时候，才能够形成。这样，串行进位加法器完成加法运算的延迟时间比较长。并且，电路延迟与加法器的位数成正比关系。加法器位数越多，延迟就越大。在速度要求比较高的 CPU 中，这种串行进位加法器的实用性能不是很好。

4. 32 位超前进位加法器

串行进位加法器的延迟时间比较长，运算速度慢。要提高加法的运算速度，必须使用并行进位的方式。

在串行进位加法器中，影响各位“和”形成的关键是低位的进位。每一位的进位都要依赖于前一位进位的形成，即每一位的进位是串行形成的，从而影响了电路的速度。为了解决这个问题，可以设计专门的进位电路，使得每一位的进位能够并行形成，而与低位的运算情况无关。考虑如下逻辑表达式。

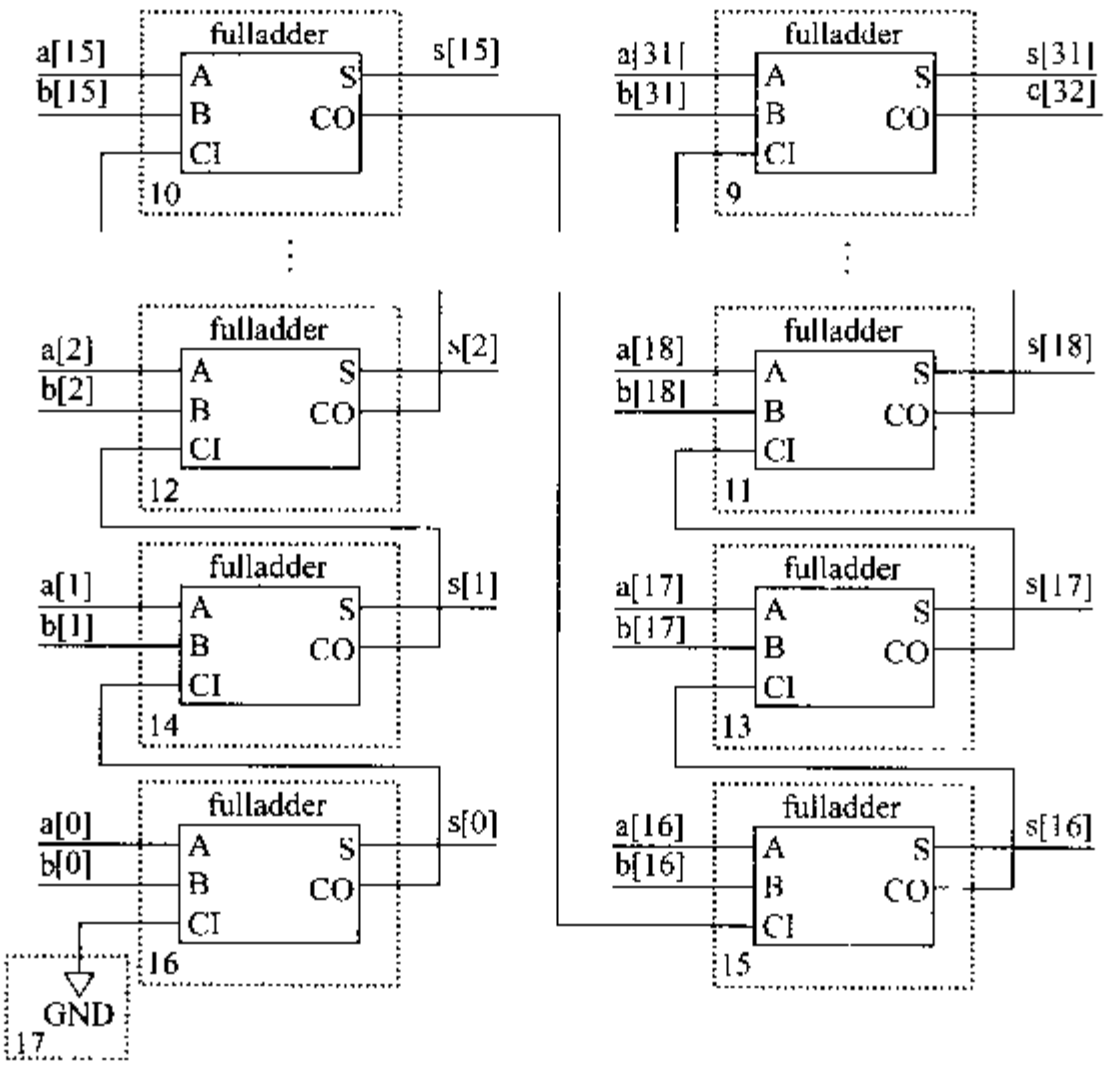


图 4.12　32 位串行进位加法器

$$c_0 = 0$$

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i = a_i b_i + (a_i + b_i) c_i$$

设

$$g_i = a_i b_i$$

$$p_i = a_i + b_i$$

则

$$\begin{aligned} c_{i+1} &= g_i + p_i c_i \\ &= g_i + p_i(g_{i-1} + p_{i-1} c_{i-1}) \\ &= g_i + p_i(g_{i-1} + p_{i-1}(g_{i-2} + p_{i-2} c_{i-2})) \\ &\vdots \\ &= g_i + p_i(g_{i-1} + p_{i-1}(g_{i-2} + p_{i-2}(\cdots(g_0 + p_0 c_0)\cdots))) \\ &= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_1 g_0 + p_i p_{i-1} \cdots p_1 p_0 c_0 \end{aligned}$$

由于 g_i，p_i 只和 a_i，b_i 有关，这样，c_{i+1} 就只和 $a_i, a_{i-1}, \cdots, a_0, b_i, b_{i-1}, \cdots, b_0$ 及 c_0 有关。所以，$c_{31}, c_{30}, \cdots, c_1$ 就可以并行地产生。这种进位称为超前进位(carry lookahead)。

通过逻辑表达式，可以得到图 4.13 所示的逻辑电路示意图。

图 4.13 的电路可以完成并行进位。但是，这种逻辑电路实现比较复杂，逻辑器件使用较多，VLSI 芯片布线困难，而且一个信号要驱动很多逻辑门。要提高逻辑器件的利用效率，需要做到某些基本逻辑单元能够复用，当然，同时要照顾到进位能够并行产生。为了达到这个目的，我们继续推导进位的逻辑表达式。

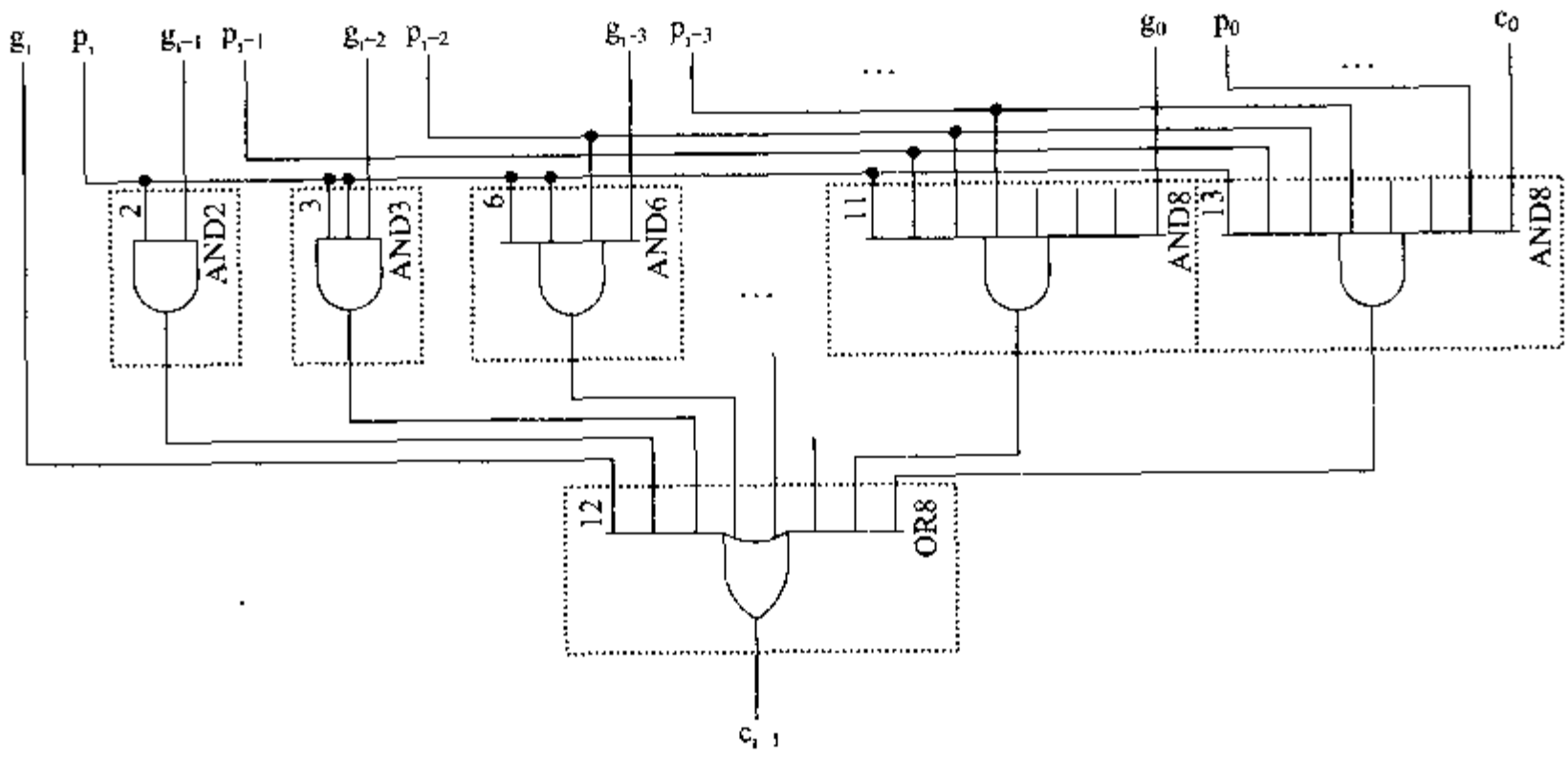


图 4.13 超前进位逻辑电路示意图

定义

$$G_{i,j} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \cdots + p_i p_{i-1} \cdots p_{j+1} g_j$$

$$P_{i,j} = p_i p_{i-1} \cdots p_{j+1} p_j$$

有

$$G_{i,i} = g_i$$

$$P_{i,i} = p_i$$

$$G_{i,j} = G_{i,k} + P_{i,k} G_{k-1,j}$$

$$P_{i,j} = P_{i,k} P_{k-1,j}$$

$$c_{i+1} = G_{i,j} + P_{i,j} c_j$$

从而能够得到如表 4.5 所示的算法。该算法为超前进位算法的扩展算法。为了方便起见，先考虑 8 位加法器的算法。

表 4.5 超前进位扩展算法

<table>
<tr><td>$G_{1,0} = g_1 + p_1 g_0$
$P_{1,0} = p_1 p_0$</td><td rowspan="2">$G_{3,0} = G_{3,2} + P_{3,2} G_{1,0}$
$P_{3,0} = P_{3,2} P_{1,0}$</td><td rowspan="4">$G_{7,0} = G_{7,4} + P_{7,4} G_{3,0}$
$P_{7,0} = P_{7,4} P_{3,0}$</td></tr>
<tr><td>$G_{3,2} = g_3 + p_3 g_2$
$P_{3,2} = p_3 p_2$</td></tr>
<tr><td>$G_{5,4} = g_5 + p_5 g_4$
$P_{5,4} = p_5 p_4$</td><td rowspan="2">$G_{7,4} = G_{7,6} + P_{7,6} G_{5,4}$
$P_{7,4} = P_{7,6} P_{5,4}$</td></tr>
<tr><td>$G_{7,6} = g_7 + p_7 g_6$
$P_{7,6} = p_7 p_6$</td></tr>
<tr><td colspan="3">$c_8 = G_{7,0} + P_{7,0} c_0$</td></tr>
</table>

从表 4.5 中可以看出，本算法的核心思想是把 8 位加法器分成两个 4 位加法器，先求出低 4 位加法器的各个进位，特别是向高 4 位加法器的进位 c_4。然后，高 4 位加法器把 c_4 作为初始进位，使用低 4 位加法器相同的方法来完成计算。每一个 4 位加法器在计算时，又分成了两个 2 位的加法器。如此递归。所以，我们只需要 $\log_2 4+1=3$ 层这样的计算。而对于 32 位加法器来说，需要 $\log_2 32+1=6$ 层这样的计算。图 4.14 为该算法的示意图。

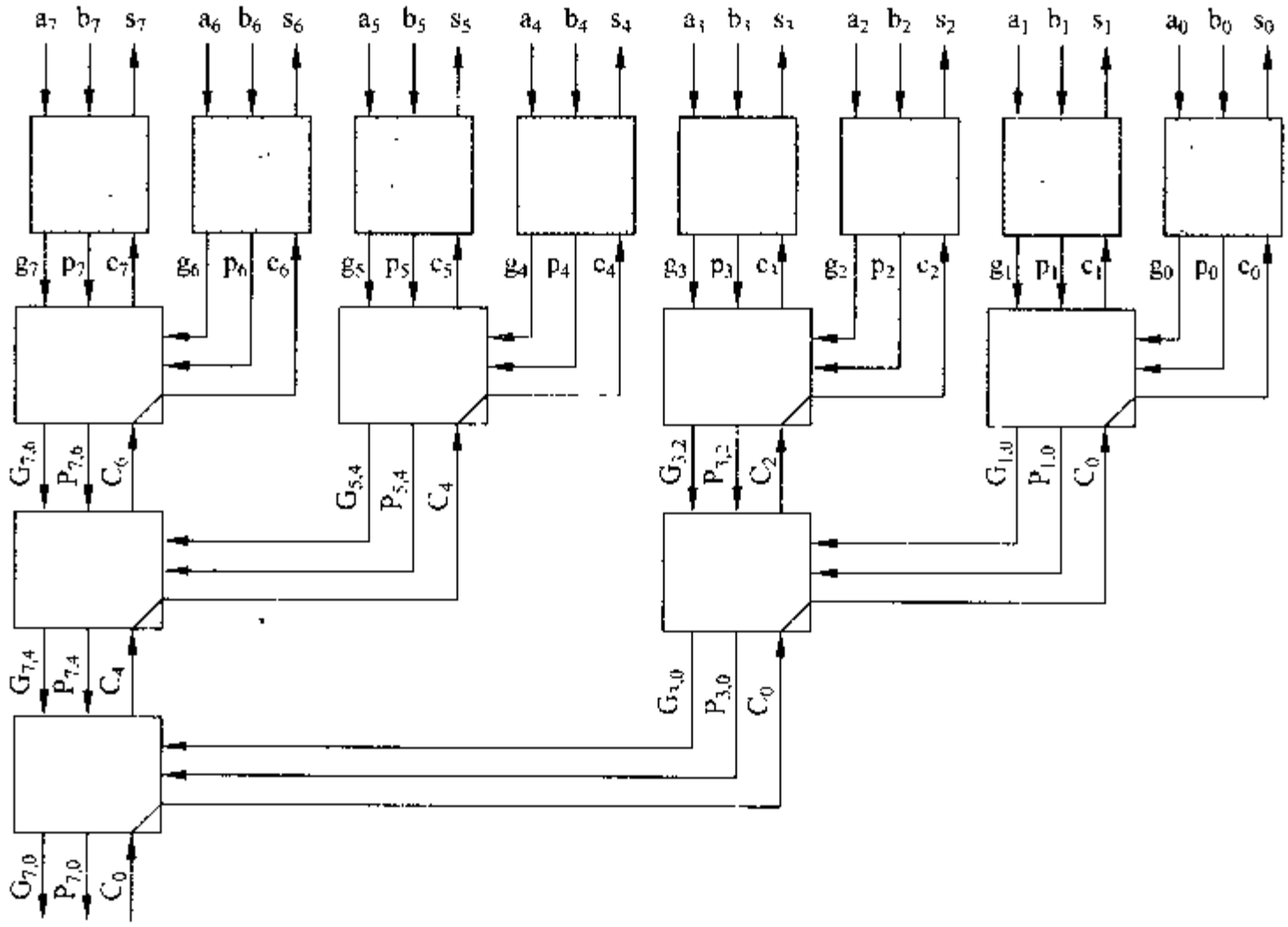


图 4.14　超前进位扩展算法示意图

这样，在超前进位扩展算法的逻辑电路实现中，需要设计两种电路。一种逻辑电路需要完成如下计算逻辑：

$$G_{i,j}=G_{i,k}+P_{i,k}G_{k-1,j}$$

$$P_{i,j}=P_{i,k}P_{k-1,j}$$

$$c_{i-1}=G_{i,j}+P_{i,j}c_j$$

另一种逻辑电路需要完成如下计算逻辑：

$$G_{i,i}=a_ib_i$$

$$P_{i,i}=a_i+b_i$$

$$s_i=a_i\overline{b_i}\ \overline{c_i}+\overline{a_i}\,b_i\overline{c_i}+\overline{a_i}\ \overline{b_i}\,c_i+a_ib_ic_i$$

这两种逻辑电路实现，如图 4.15 所示。这两种电路之间的连接如示意图 4.14 所示。

现在来估算一下超级进位扩展算法逻辑电路实现中的电路延迟。如前所述，32 位加法器中，在每个 16 位加法器上，需要经过 6 层如图 4.15 左边电路所示的电路延迟，所以，共需要经过 6+6−1=11 层这样的电路延迟。每个电路的延迟为 2 级延迟。所以，共为

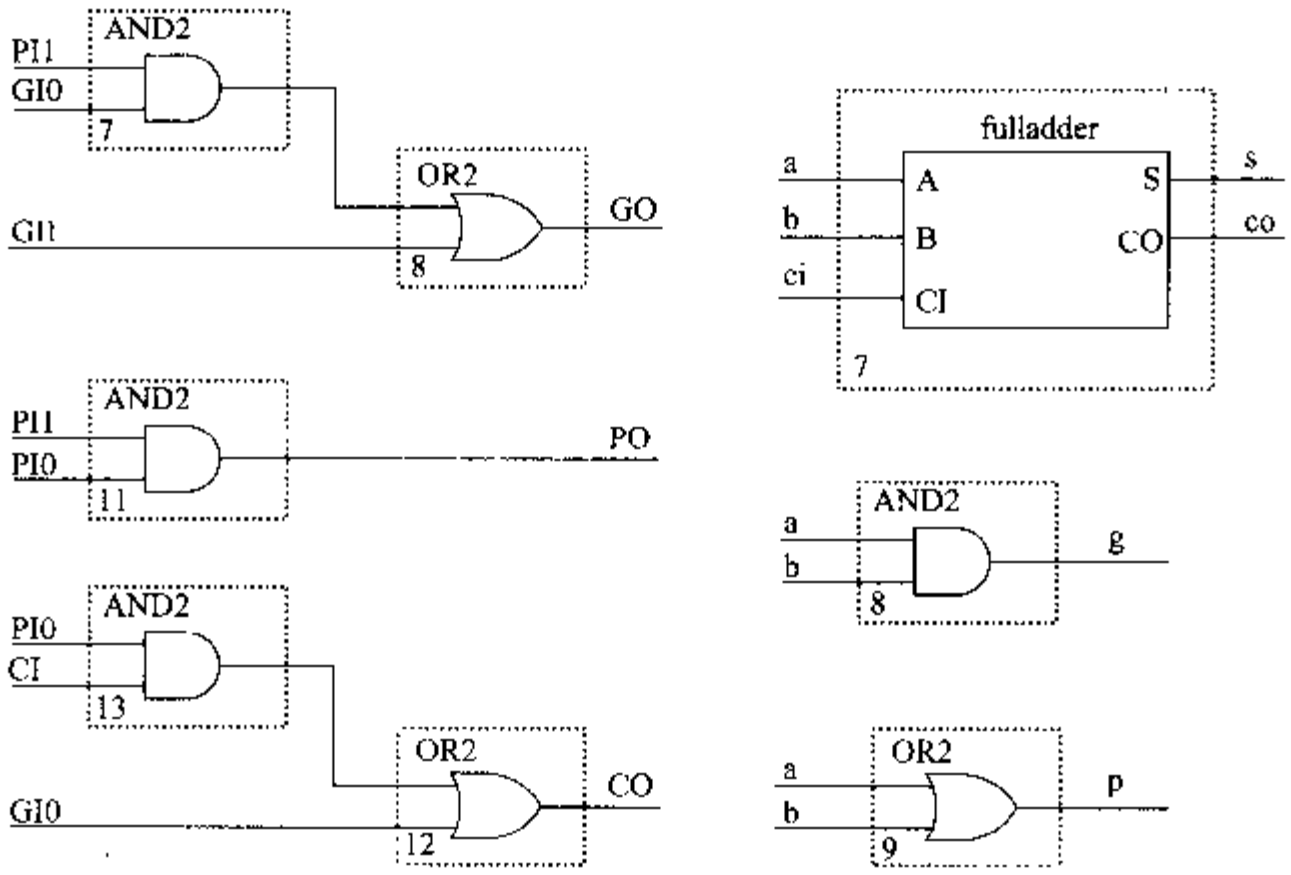


图 4.15　超前进位扩展算法逻辑电路实现

11×2=22 级延迟。同时，在计算 $G_{i,k}$ 和 $P_{i,k}$ 时，需要 1 级门延迟。在计算各位的和时，需要 3 级门延迟(即为一个全加器的门延迟)。所以，共需要 22+1+3=26 级门延迟。但是，从表 4.5 中可以看到，$G_{i,i}$ 和 $P_{i,i}$ 既参与了每位上进位的计算，又参与了下一级 $G_{i,i+1}$ 和 $P_{i,i+1}$ 的计算。这样，就复用了这些电路，使得需要的总逻辑电路数大大减少。这里体现出追求电路速度和追求逻辑电路精简这一对矛盾的一种平衡关系。

超前进位加法器的运算速度较快，但是，与串行进位加法器相比，逻辑电路比较复杂，使用的逻辑器件较多。这些是提高运算速度付出的代价。

4.3.2　32 位减法器

32 位减法器的功能为执行两个 32 位操作数的减法操作。操作：

$$S \leftarrow A - B$$

涉及减法器的 MIPS 指令有：SUB，SUBU。32 位减法器可以在 32 位加法器的基础上来设计。由于

$$a - b = a + (-b) = a + \bar{b} + 1$$

所以，可以用 32 位加法器来实现 32 位减法器。只需把减数 b 按位取反，再与被减数 a 相加，同时，低位的进位设为 1。这样，就在 32 位加法器的基础上实现了 32 位减法器。32 位减法器的逻辑电路图如图 4.16 所示。

4.3.3　32 位加减法器

32 位加减法器是 32 位加法器和 32 位减法器功能的综合。操作：

$$S \leftarrow A + (-1)^{sub} B$$

其中，sub 是进行加法操作还是进行减法操作的标志。与 32 位减法器相似，32 位加减法

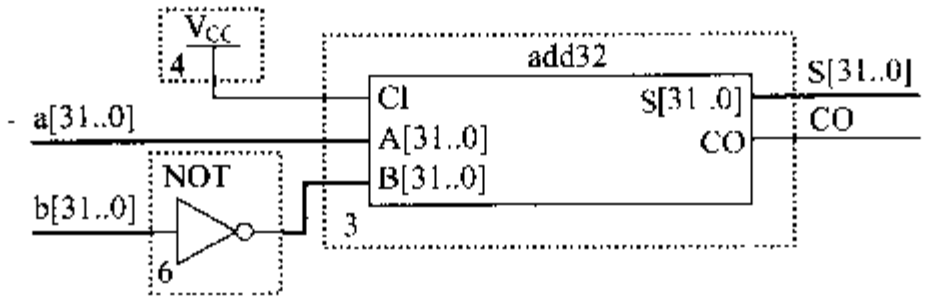


图 4.16　32 位减法器

器也可以在 32 位加法器的基础上设计。表 4.6 是 32 位加减法器的真值表。

表 4.6　位加减法真值表

sub	A	B	CO	说明
0	a	b	0	加法器
1	a	$\bar{b}$	1	减法器

表 4.6 中，A、B 分别为 32 位加法器输入的两个加数，CO 为 32 位加法器输入的低位进位。sub 为加法器、减法器的指示标志。由表 4.6，可以得到如下逻辑表达式。

$$A = a$$

$$B = \overline{sub}\, b + sub\, \bar{b} = sub \oplus b$$

$$CO = sub$$

从而，得到 32 位加减法器的逻辑电路图。如图 4.17 所示。

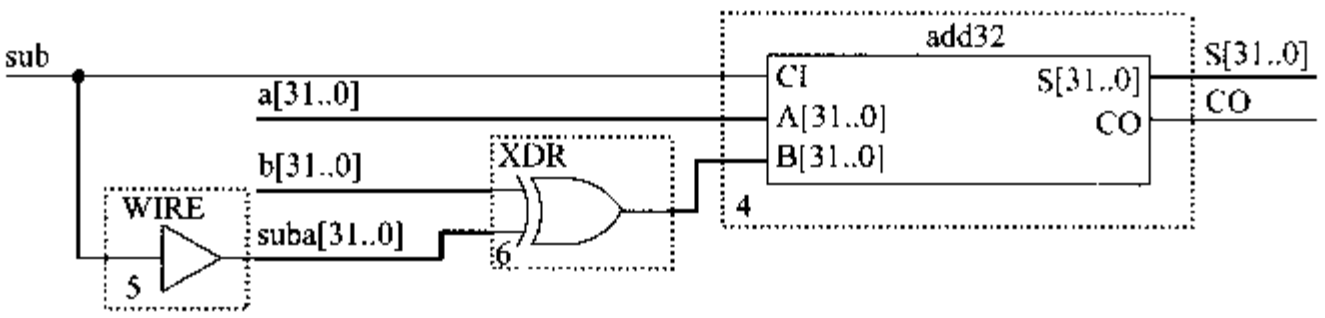


图 4.17　32 位加减法器逻辑电路图

4.4　乘法器

乘法器是实现两数相乘的运算部件。在 MIPS 中，定义了两种乘法运算。一种为无符号乘法，另一种为有符号乘法。

4.4.1　32 位无符号乘法器

32 位无符号乘法器实现两个无符号的整数乘法操作，并给出 64 位的计算结果。操作：

$$P \leftarrow A \times B$$

涉及 32 位无符号乘法器的 MIPS 指令有：MULTU。我们先观察 4 位乘法是怎么进

行的。图 4.18 中演示的是 1101 和 1011 的手工乘法。

```
                1  1  0  1
              × 1  0  1  1
-------------------------
                1  1  0  1
             1  1  0  1
          0  0  0  0
       1  1  0  1
-------------------------
1  0  0  0  1  1  1  1
```

图 4.18　4 位手工乘法

从图 4.18 的 4 位手工乘法可以看到，4 位乘法的运算结果可以由 4 次加法得出(第 1 次加 0)。每次加法完成以后，将得到的部分积右移一次，然后进行下一次的加。右侧移出的一位，就是结果乘积中的一个位。前 3 次每次加法可以得到乘积的一个位，最后一次加法得到乘积的高 5 位。这种乘法算法称为移位相加乘法。算法的运算过程如表 4.7 所示。其中乘积的高 4 位和低 4 位分别在 HI 和 LO 中。

表 4.7　4 位无符号乘法

加法次数	HI	LO	B
	0 0 0 0		
1	+ 1 1 0 1		1 0 1 1
	0 1 1 0 1		
	0 1 1 0		
2	+ 1 1 0 1	1	1 0 1
	1 0 0 1 1		
	1 0 0 1		
3	+ 0 0 0 0	1 1	1 0
	0 1 0 0 1		
	0 1 0 0		
4	+ 1 1 0 1	1 1 1	1
	1 0 0 0 1		
	1 0 0 0	1 1 1 1	

通过表 4.7，我们可以使用移位相加乘法算法来设计实现乘法器。由表 4.7 可以看出，随着计算的进行，乘积的低字节 LO 的位数越来越多，而乘数 B 的位数越来越少。这样，就可以把这两个部分合并在一起，由一个存储单元来表示。从而，可以节省一个存储单元。移位相加乘法器具体流程如图 4.19 所示。

在乘法器实现中，需要使用计数器、32 位寄存器、32 位移位寄存器等逻辑器件。下面将逐一设计这些器件。

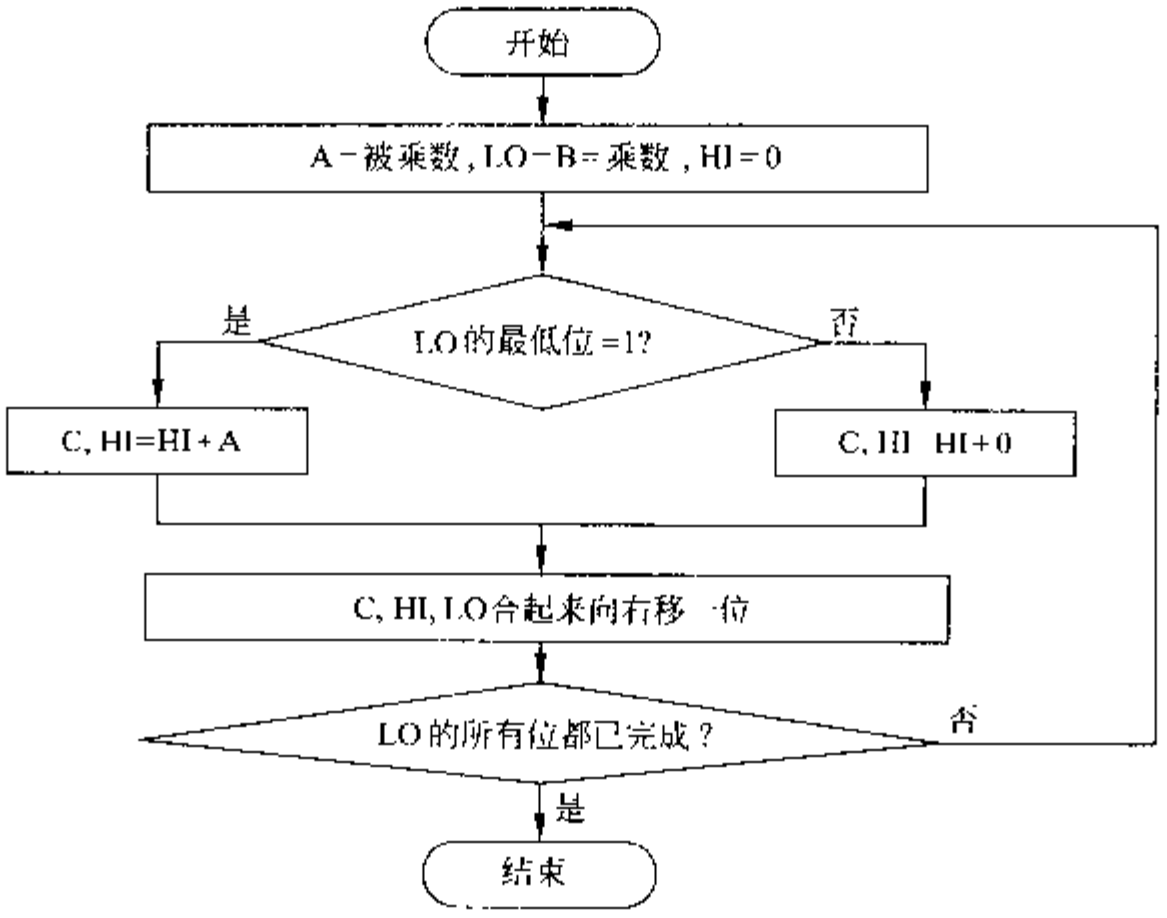


图 4.19　移位相加乘法器计算流程

1. 计数器

在该乘法器中，需要使用一个计数器来计算加法次数。因为 32 位乘法器中，需要进行 32 次循环加法，所以，计数器至少需要 5 位来满足计数要求。

现在来设计一个 5 位的递增计数器，同时，要求该计数器还能够方便地加载计数初始数字。加入控制位 LOAD。当 LOAD 为 1 时，计数器加载数据；当 LOAD 为 0 时，计数递增计数。该计数器为 5 位，有 32 种状态，分别设为 S0，S1，…，S31。根据上述对计数器的要求，得出计数器的原始状态表(表 4.8)。

表 4.8　5 位递增计数器原始状态表

S_n	LOAD	
	0	1
S0	S1	DI
SI	S2	DI
⋮	⋮	⋮
S30	S31	DI
S31	S0	DI

为计数器分配状态。该计数器共有 32 种状态，使用 5 位来表示。这 5 位分别定义为 Q0，Q1，…，Q4。各个状态分配如下：状态 S0 为 Q4Q3Q2Q1Q0＝00000，状态 S1 为 Q4Q3Q2Q1Q0＝00001，…，状态 S31 为 Q4Q3Q2Q1Q0＝11111。由此，得到状态分配表(表 4.9)。

表 4.9　5 位递增计数器状态分配表

Q4	Q3	Q2	Q1	Q0	LOAD				
					0				1
0	0	0	0	0	0	0	0	0	1	DI
0	0	0	0	1	0	0	0	1	0	DI
⋮					⋮				⋮
1	1	1	1	0	1	1	1	1	1	DI
1	1	1	1	1	0	0	0	0	0	DI

在电路实现中，我们使用 D 触发器。根据计数器的状态分配表，可以得出表 4.10 的各个 D 触发器的真值表。其中 Q[4..0]为计数器当前的输出，D[4..0]在时钟上升沿被存入 D 触发器中，即为下一个计数值。

表 4.10　D 触发器真值表

Q4	Q3	Q2	Q1	Q0	LOAD	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	0
⋮										
1	1	1	1	0	0	1	1	1	1	1
1	1	1	1	1	0	0	0	0	0	0
×	×	×	×	×	1	DI4	DI3	DI2	DI1	DI0

根据 D 触发器的真值表，可以得出如下的逻辑表达式：

$$D0 = \overline{LOAD}\ \overline{Q0} + LOAD\ DI0$$

$$D1 = \overline{LOAD}(Q1\ \overline{Q0} + \overline{Q1}Q0) + LOAD\ DI1$$
$$= \overline{LOAD}(Q1 \oplus Q0) + LOAD\ DI1$$

$$D2 = \overline{LOAD}(Q2(\overline{Q1} + \overline{Q0}) + \overline{Q2}Q1Q0) + LOAD\ DI2$$
$$= \overline{LOAD}(Q2 \oplus Q1Q0) + LOAD\ DI2$$

$$D3 = \overline{LOAD}(Q3(\overline{Q2} + \overline{Q1} + \overline{Q0}) + \overline{Q3}Q2Q1Q0) + LOAD\ DI3$$
$$= \overline{LOAD}(Q3 \oplus Q2Q1Q0) + LOAD\ DI3$$

$$D4 = \overline{LOAD}(Q4(\overline{Q3} + \overline{Q2} + \overline{Q1} + \overline{Q0}) + \overline{Q4}Q3Q2Q1Q0) + LOAD\ DI4$$
$$= \overline{LOAD}(Q4 \oplus Q3Q2Q1Q0) + LOAD\ DI4$$

通过逻辑表达式，可以得出如图 4.20 所示的逻辑电路图。至此，我们完成了 5 位递增计数器的设计。

在 MAX+PLUS Ⅱ中，也可以通过编辑 AHDL 文件来设计逻辑电路。下面将使用 AHDL 语言来设计一个 6 位的递减计数器。具体设计如下所示。

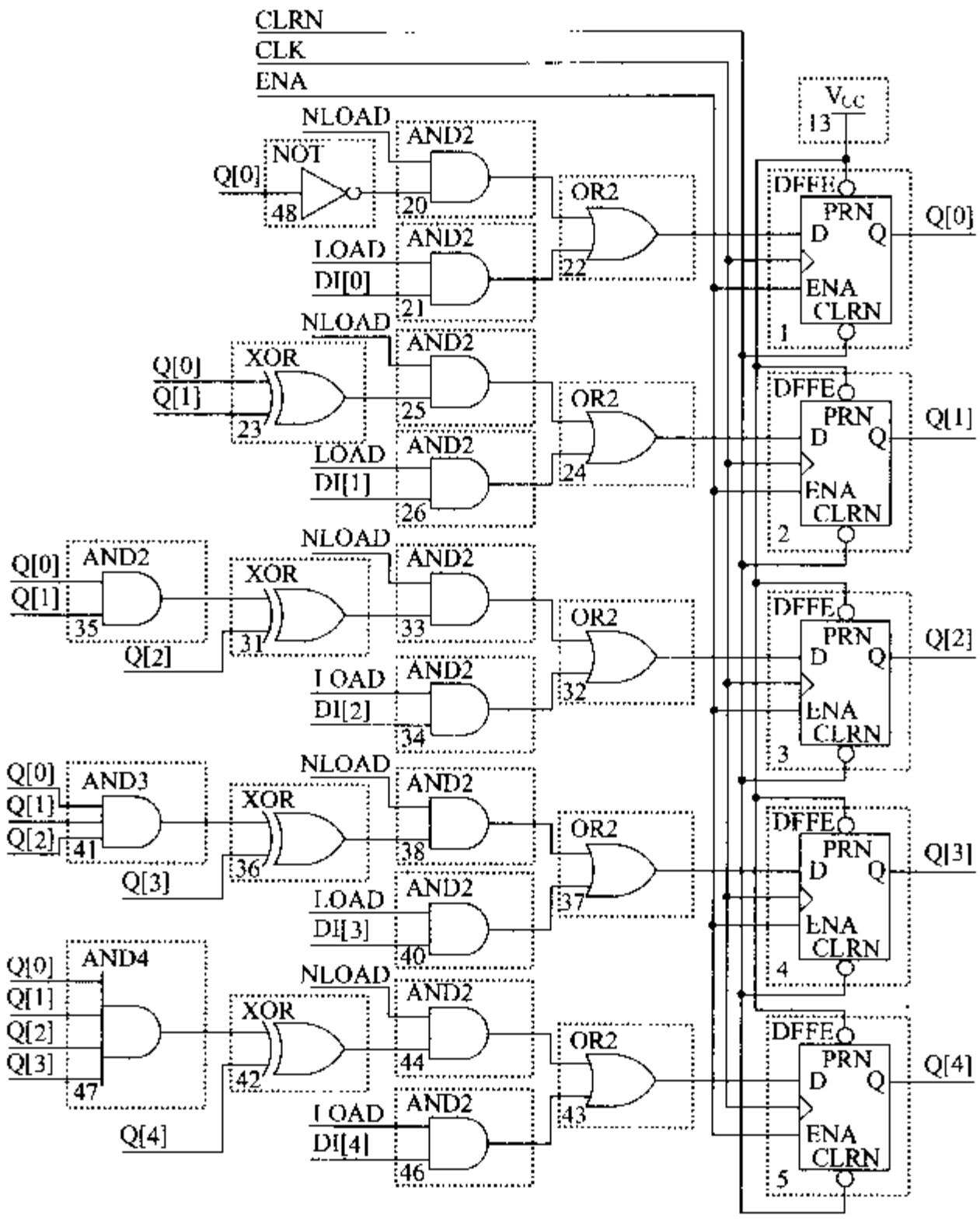


图 4.20　5 位递增计数器

```
SUBDESIGN 5counter
(
        clk,load,ena,clr,d[4..0]    :INPUT;
        q[4..0]                     :OUTPUT;
)

VARIABLE
        count[4..0]                 :DFF;
BEGIN
        count[].clk=clk;
        count[].clrn=! clr;
        IF load THEN
```

```
        count[].d=d[];
    ELSIF ena THEN
        count[].d=count[].q-1;
    ELSE
        count[].d=count[].q;
    END IF;
    q[]=count[];
END;
```

2. 32 位寄存器

我们使用 D 触发器来搭建 32 位寄存器。图 4.21 是使用 D 触发器来设计 8 位寄存器。然后,使用该 8 位寄存器来搭建 32 位寄存器,如图 4.22 所示。

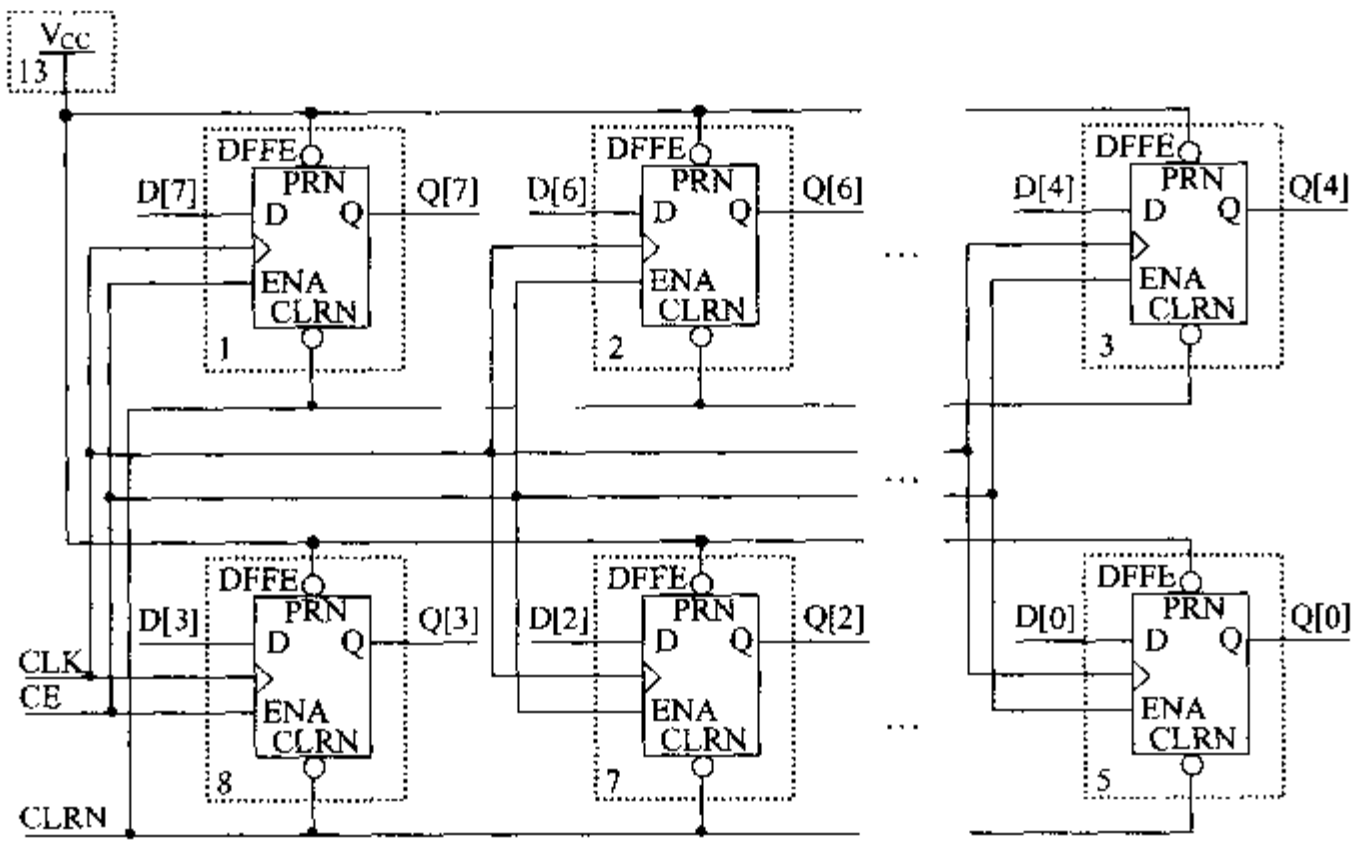


图 4.21 8 位寄存器

3. 32 位移位寄存器

32 位移位寄存器能够加载 32 位数字,保存在寄存器中。每个时钟周期右移寄存器内的数字,最高位由输入提供。可见,移位寄存器的功能是在寄存器功能的基础上加上了移位功能,所以可以使用寄存器来实现。

图 4.23 为使用 8 位寄存器来实现 8 位移位寄存器。在完成了 8 位移位寄存器设计之后,可以使用该 8 位移位寄存器来设计 32 位的移位寄存器,如图 4.24 所示。

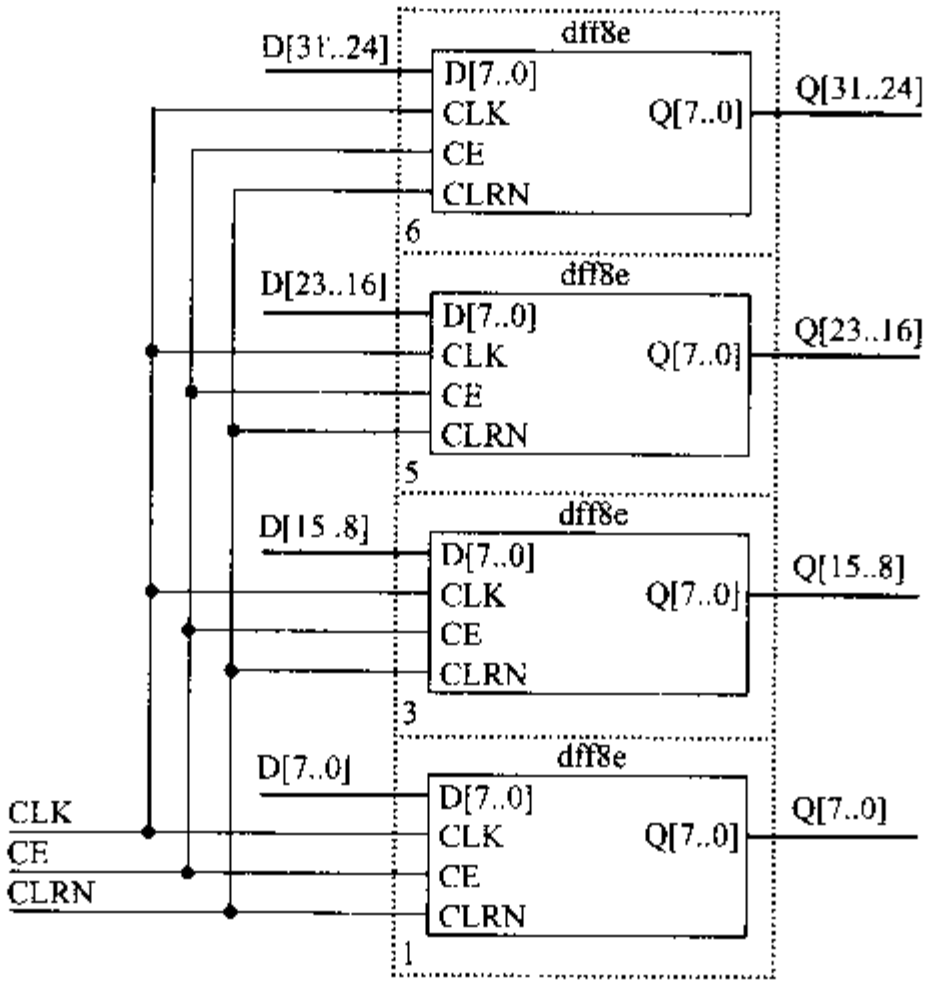


图 4.22　32 位寄存器

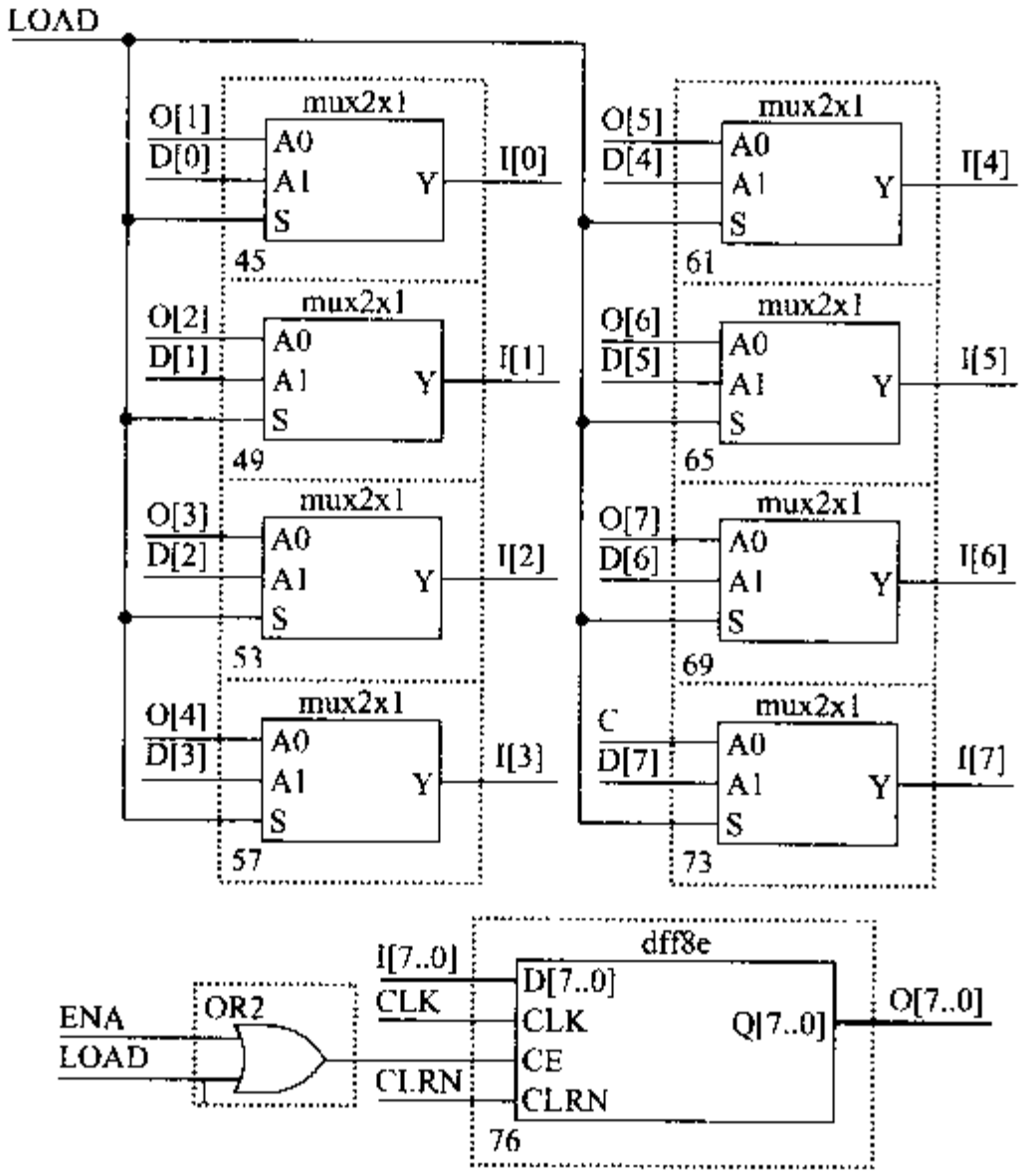


图 4.23　8 位移位寄存器

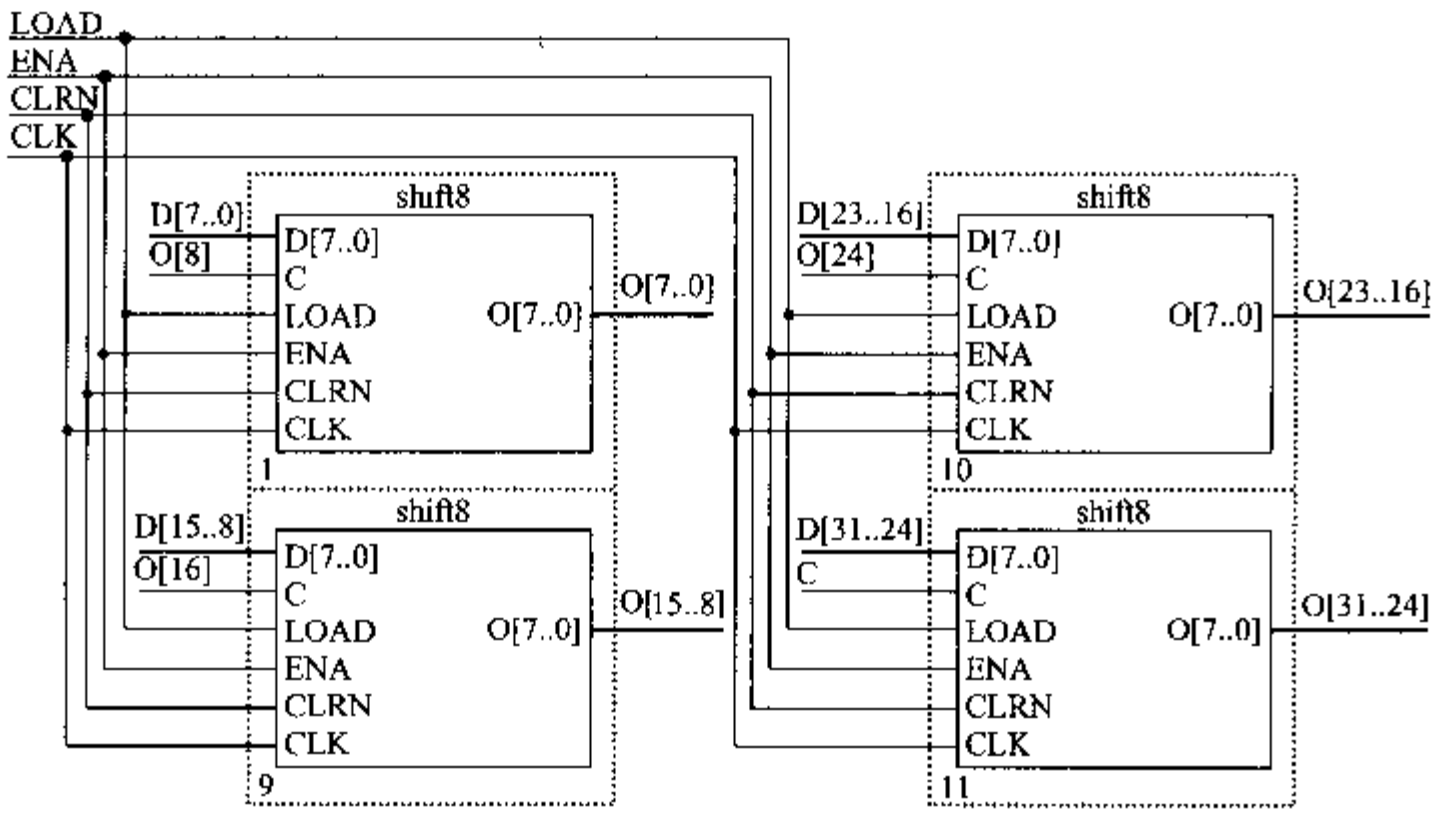


图 4.24　32 位移位寄存器

4.4.2　32 位乘法器

下面，我们来设计 32 位无符号乘法器，如图 4.25 所示。在该逻辑电路设计中，使用一个 5 位递减计数器来为乘法器控制加法次数。

完成了 32 位无符号乘法器的设计，现在我们来考虑 32 位有符号乘法器的设计。表 4.11 为有符号乘法与无符号乘法的比较。

表 4.11　有符号乘法与无符号乘法比较

被乘数×乘数	无符号乘法结果	有符号乘法结果
0101×0011	00001111(3×5=15)	00001111(3×5=15)
1101×0011	00100111(13×3=39)	11110111((−3)×3=−9)
0011×1101	00100111(3×13=39)	11110111(3×(−3)=−9)
1101×1011	10001111(13×11=143)	00001111((−3)×(−5)=15)

从表 4.11 可以看到，除了被乘数和乘数都是正数之外，有符号乘法和无符号乘法的算法很不相同。下面，我们推导有符号乘法的算法。根据被乘数和乘数的正负关系，可以分成“正数乘以正数”、“负数乘以正数”、“正数乘以负数”、“负数乘以负数”四种情况。

1. 正数乘以正数

在“正数乘以正数”的情况下，有符号乘法和无符号乘法的算法基本相同。

2. 负数乘以正数

当被乘数为负数、乘数为正数的时候，乘法的意义为乘数个被乘数(负数)相加得到的结果为乘积。这样，与“正数乘以正数”的算法类似，在做循环加法的时候，需要带着符号

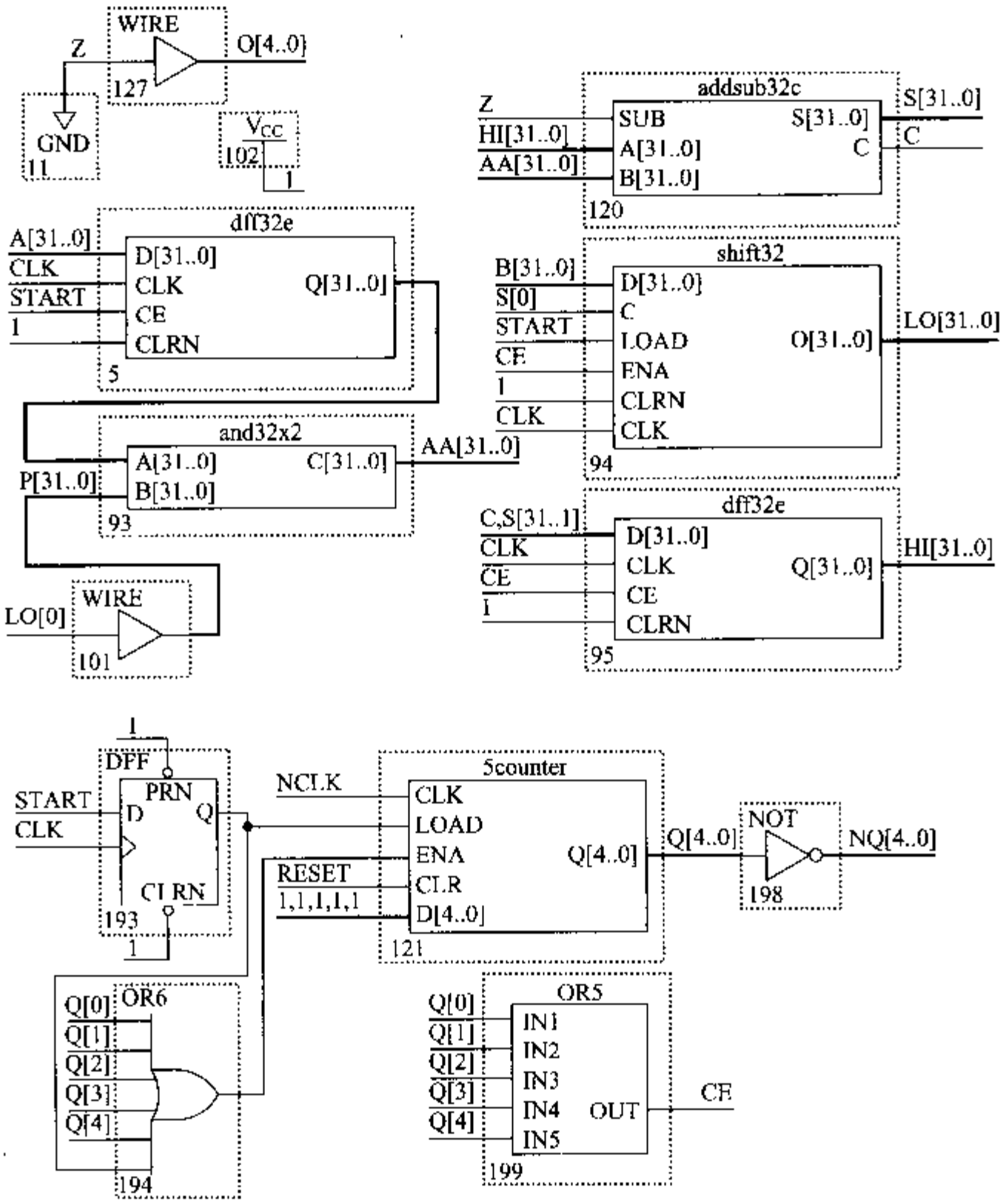


图 4.25　32 位无符号乘法器

位来相加。图 4.26 显示了“负数乘以正数”有符号乘法与无符号乘法的区别。表 4.12 演示了“负数乘以正数”有符号乘法的运算过程，表中为 1101×0101。

无符号乘法								
					1	1	0	1
×					0	1	0	1
	0	0	0	0	1	1	0	1
	0	0	0	0	0	0	0	
	0	0	1	1	0	1		
+	0	0	0	0	0			
	0	1	0	0	0	0	0	1

有符号乘法								
					1	1	0	1
×					0	1	0	1
	1	1	1	1	1	1	0	1
	0	0	0	0	0	0	0	
	1	1	1	1	0	1		
+	0	0	0	0	0			
	1	1	1	1	0	0	0	1

图 4.26　“负数乘以正数”有符号乘法

表 4.12 4 位有符号乘法

加法次数	HI	LO	B
1	0 0 0 0 + 1 1 0 1 1 1 1 0 1		0 1 0 1
2	1 1 1 0 + 0 0 0 0 1 1 1 1 0	1	0 1 0
3	1 1 1 1 + 1 1 0 1 1 1 1 0 0	0 1	0 1
4	1 1 1 0 + 0 0 0 0 1 1 1 1 0	0 0 1	0
	1 1 1 1	0 0 0 1	

需要注意的是由于是有符号乘法，同时，被乘数为负数，而乘数为正数，所以在表 4.13 中，HI 中的整数将只能为 0 或者为负数。可以通过下面的真值表来确定 HI 移入位。表中的 E 为最后一次加减法运算标志量，SUB 为每次加减法运算循环中加减法的选择，0 表示加法，1 表示减法。

由真值表 4.13 可以得出下面的逻辑表达式。

$$C2HI = \overline{E}(HI[31] + LO[0]) + E$$

$$SUB = E$$

表 4.13 “负数乘以正数”时 HI 移入位真值表

HI[31]	LO[0]	E	C2HI	SUB
0	0	0	0	0
0	1	0	1	0
1	×	0	1	0
×	×	1	1	1

3. 正数乘以负数

若 a 为正数，b 为负数，则 a×b 应为

$$2^{64} - a \times (2^{32} - b) = 2^{64} - 2^{32}a + a \times b = 2^{32}(2^{32} - a) + a \times b$$

$$= (a \times b|_{63..32} - a) \| (a \times b|_{31..0})$$

其中“‖”表示连接运算。上述等式中的乘法为两个正数的乘法。从等式可以看出，乘数为负数的乘法，只需按照无符号整数乘法来计算，然后，在高 32 中减去被乘数。表 4.14

演示了“正数乘以负数”情况下的乘法运算，表中为 0101×1101。

表 4.14　4 位有符号乘法

加减法次数	HI	LO	B
1	0 0 0 0 +0 1 0 1 0 0 1 0 1		1 1 0 1
2	0 0 1 0 +0 0 0 0 0 0 0 1 0	1	1 1 0
3	0 0 0 1 +0 1 0 1 0 0 1 1 0	0 1	1 1
4	0 0 1 1 +0 1 0 1 0 1 0 0 0	0 0 1	1
5	0 1 0 0 −0 1 0 1 1 1 1 1	0 0 0 1	

表 4.14 中，在最后一次加法完成并完成右移后，要从 HI 中减去 a。由于乘数 b 为负数，最高位为 1，所以，最后一次加法运算一定是加上被乘数 a。然后右移，再减去 a，相当于先减去 2a，再右移。这样，就有 HI+a−2a=HI−a，然后右移。这样做，就减少了一次加法运算。表 4.15 演示了该算法。

表 4.15　4 位有符号乘法

加减法次数	HI	LO	B
1	0 0 0 0 +0 1 0 1 0 0 1 0 1		1 1 0 1
2	0 0 1 0 +0 0 0 0 0 0 0 1 0	1	1 1 0
3	0 0 0 1 +0 1 0 1 0 0 1 1 0	0 1	1 1
4	0 0 1 1 −0 1 0 1 1 1 1 1 0	0 0 1	1
	1 1 1 1	0 0 0 1	

在“正数乘以负数”的情况下，HI 的移入位即为 HI 加减法运算的高位进位。同时，最后一次运算为减法运算。

4. 负数乘以负数

“负数乘以负数”乘法运算和“正数乘以负数”乘法运算基本相同。表 4.16 演示了这种情况下的乘法运算，表中为 1101×1101。

表 4.16 4 位有符号乘法

加减法次数	HI	LO	B
1	0 0 0 0 + 1 1 0 1 1 1 1 0 1		1 1 0 1
2	1 1 1 0 + 0 0 0 0 1 1 1 1 0	1	1 1 0
3	1 1 1 1 + 1 1 0 1 1 1 1 0 0	0 1	1 1
4	1 1 1 0 − 1 1 0 1 0 0 0 1	0 0 1	1
	0 0 0 0	1 0 0 1	

可以通过真值表 4.17 来确定 HI 移入位。

表 4.17 “负数乘以正数”时 HI 移入位真值表

HI[31]	LO[0]	E	C2HI	SUB
0	0	0	0	0
0	1	0	1	0
1	×	0	1	0
×	×	1	0	1

由真值表 4.17 可以得出下面的逻辑表达式：

$$C2HI = \overline{E}(HI[31] + LO[0])$$

$$SUB = E$$

综合上述情况，可以得出下面的逻辑表达式：

$$\begin{aligned} C2HI = {} & \overline{A[31]}\ \overline{B[31]}C + A[31]\ \overline{B[31]}(\overline{E}(HI[31] + LO[0]) + E) \\ & + \overline{A[31]}\ \overline{B[31]}C[31] + A[31]B[31]\overline{E}(HI[31] + LO[0]) \\ = {} & \overline{A[31]}C + A[31]\overline{E}(HI[31] + LO[0]) + A[31]\ \overline{B[31]}E \end{aligned}$$

$$= \overline{A[31]}C + A[31](HI[31] + LO[0])\overline{E} + A[31]\,\overline{LO[0]}E$$

$$= \overline{A[31]}C + A[31](LO[0] \oplus E) + A[31]HI[31]\overline{E}$$

$$SUB = \overline{A[31]}B[31]E + A[31]B[31]E = B[31]E = LO[0]E$$

通过上述逻辑表达式，可以得出如图 4.27 所示的逻辑电路图。

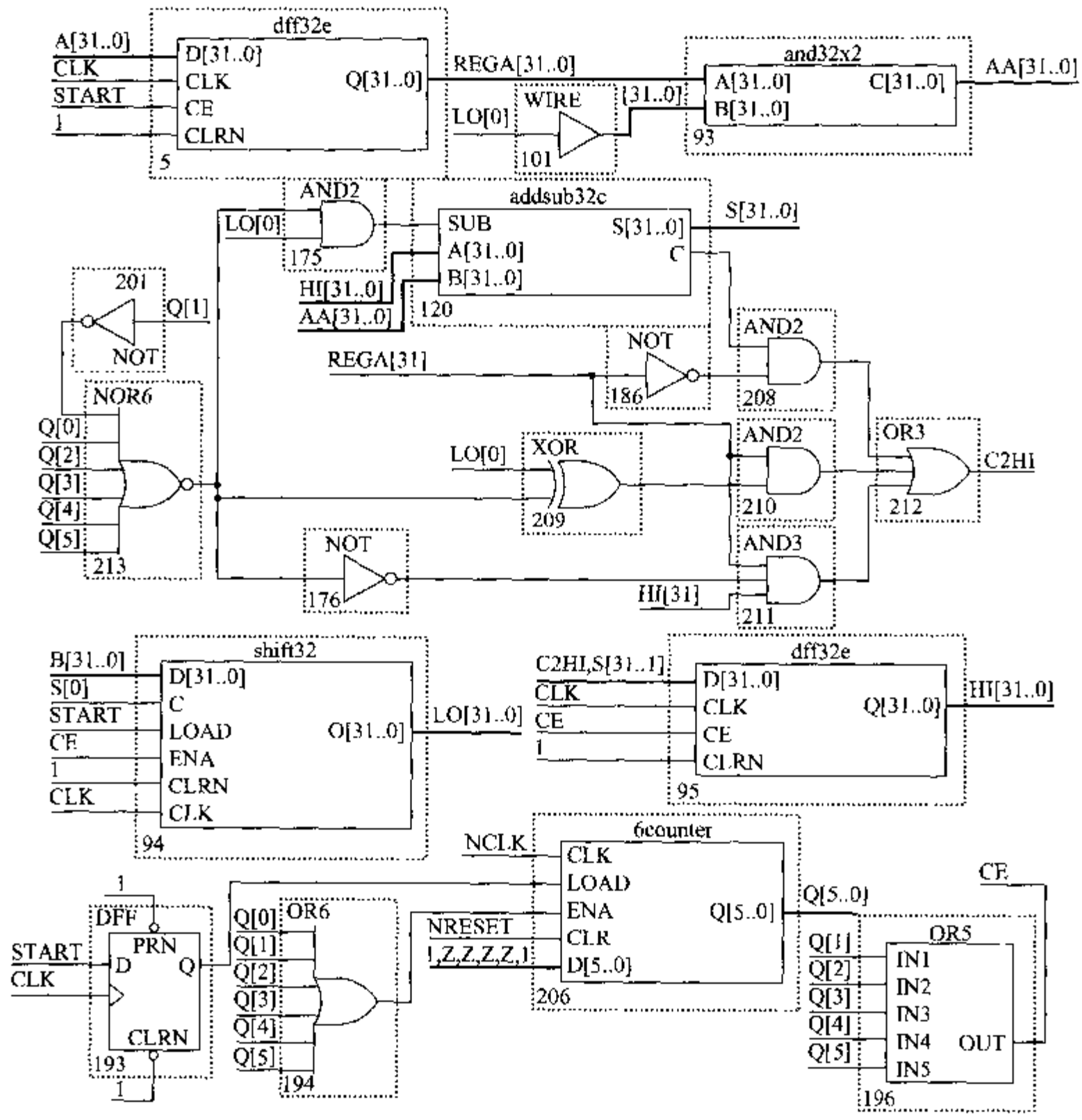


图 4.27　有符号乘法器逻辑电路图

下面考虑将无符号乘法器和有符号乘法器合并在一起，设计完整的乘法器。引入一个新的逻辑量 SIGN，表示当前乘法操作是否为有符号乘法操作。如 SIGN 为 1，则表示当前操作为有符号乘法；否则，为无符号乘法。由此，易得如下逻辑表达式：

$$C2HI = SIGN(\overline{A[31]}C + A[31](LO[0] \oplus E) + A[31]\overline{E}) + \overline{SIGN}\,C$$

$$= \overline{(SIGN\ A[31])}C + SIGN(A[31](LO \oplus E) + A[31]HI[31]\overline{E})$$

$$SUB = SIGN\ LO[0]E$$

有这些逻辑表达式，可以得出图 4.28 所示的逻辑电路图。

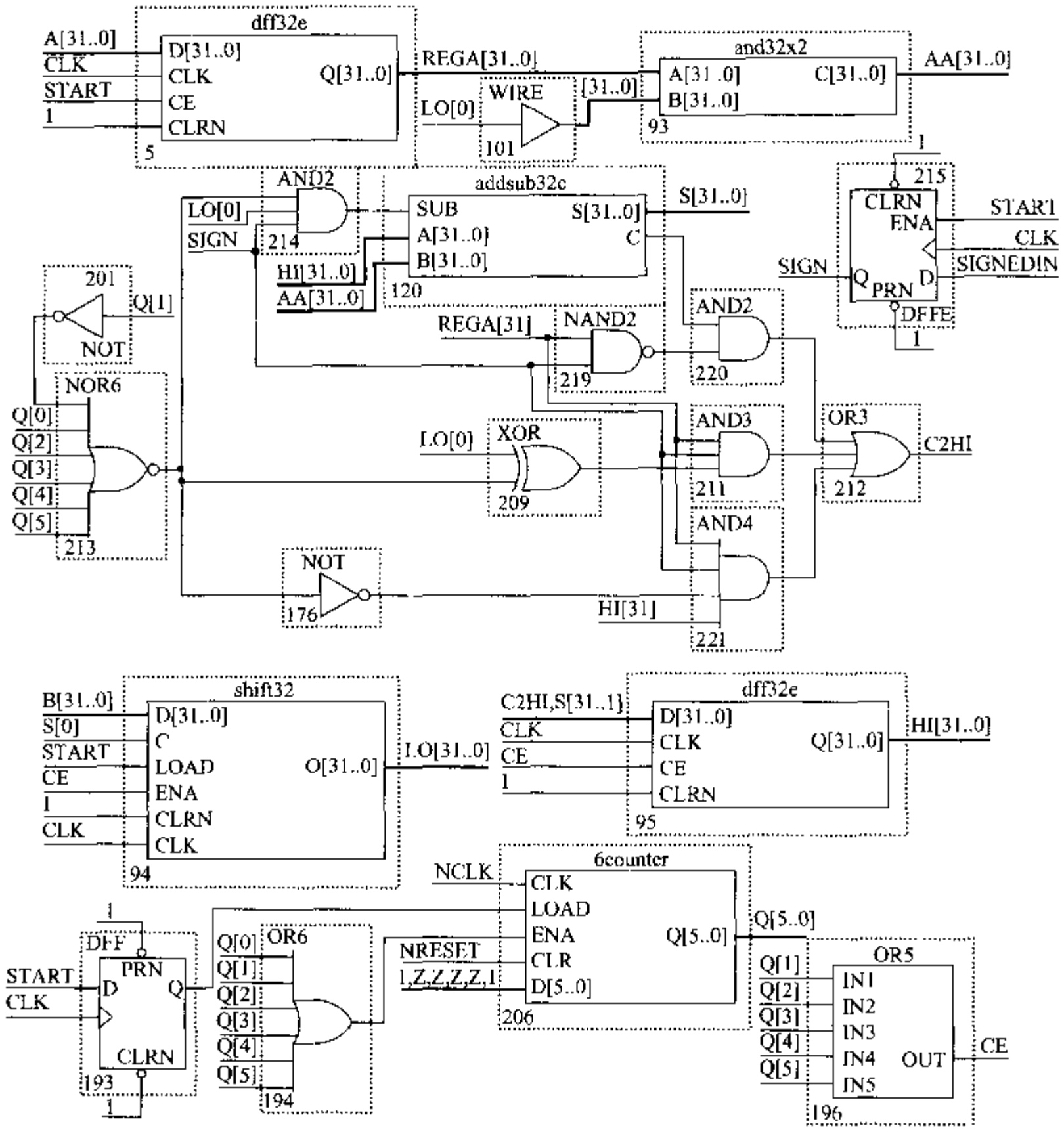


图 4.28　32 位乘法器逻辑电路图

4.4.3　乘法并行阵列

上面介绍了使用迭代加法实现的乘法器。这种方式简单易行，但是延迟时间比较长，需要 32 个时钟周期才能够完成计算。为了缩短时间延迟，可以考虑使用更快速的方式来实现乘法器。下面，介绍一种使用并行阵列的方式实现的乘法器。

如图 4.29 所示，在进行乘法运算时，只需要把每位乘出来的结果错位加起来。根据乘法的形式，可以设计一种加法阵列来实现乘法运算，如图 4.30 所示。

				a3	a2	a1	a0
			×	b3	b2	b1	b0
				a3 b0	a2 b0	a1 b0	a0 b0
			a3 b1	a2 b1	a1 b1	a0 b1	
		a3 b2	a2 b2	a1 b2	a0 b2		
+	a3 b3	a2 b3	a1 b3	a0 b3			
p7	p6	p5	p4	p3	p2	p1	p0

图 4.29　4 位乘法示意图

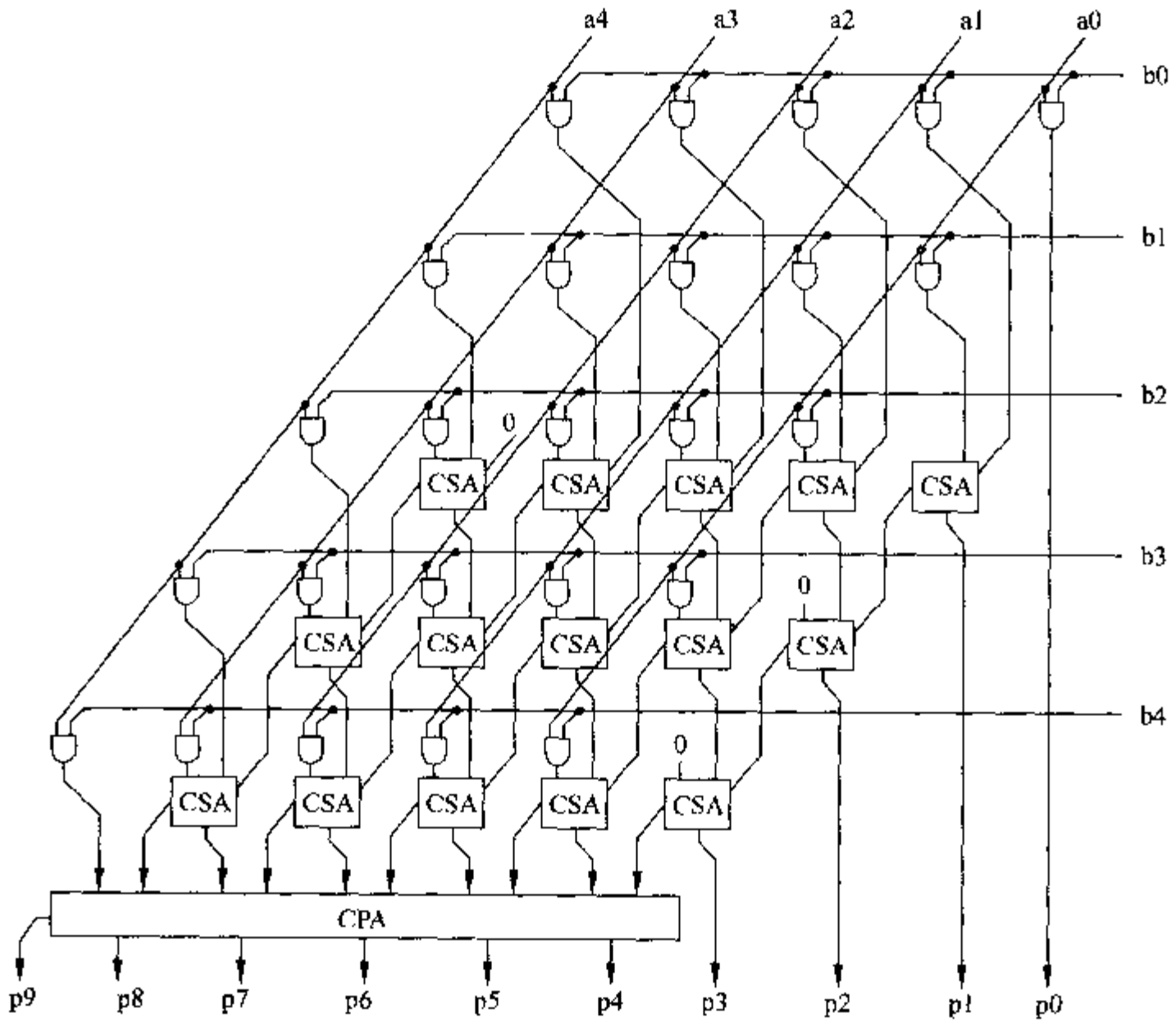


图 4.30　乘法并行阵列示意图

图 4.30 中的 CSA 为进位保存加法器(carry save adder)。CSA 的作用是能同时执行三个数相加，但不是产生这三个数之和的最终结果。进位保存加法器将每一位的进位保存下来，作为一个输出结果输出，以便于后面的运算能够处理。同时，CSA 输出每一位相加的部分和。

由于进位保存加法器中不存在每个位之间的串行进位，所以，进位保存加法器能够以更快的方式得到进位和部分和。如图 4.30 所示，进位保存加法器产生的进位和部分和将传递给下一级的进位保存加法器来处理。

当计算进行到最后一级时，需要使用 CPA(carry propagate adder)来使用这些进位和

部分和产生最后的乘积结果。CPA 为进位传递加法器，即为前面介绍的先行进位加法器。根据上面的“乘法并行阵列示意图”，可以得出如图 4.31 所示的逻辑电路图。

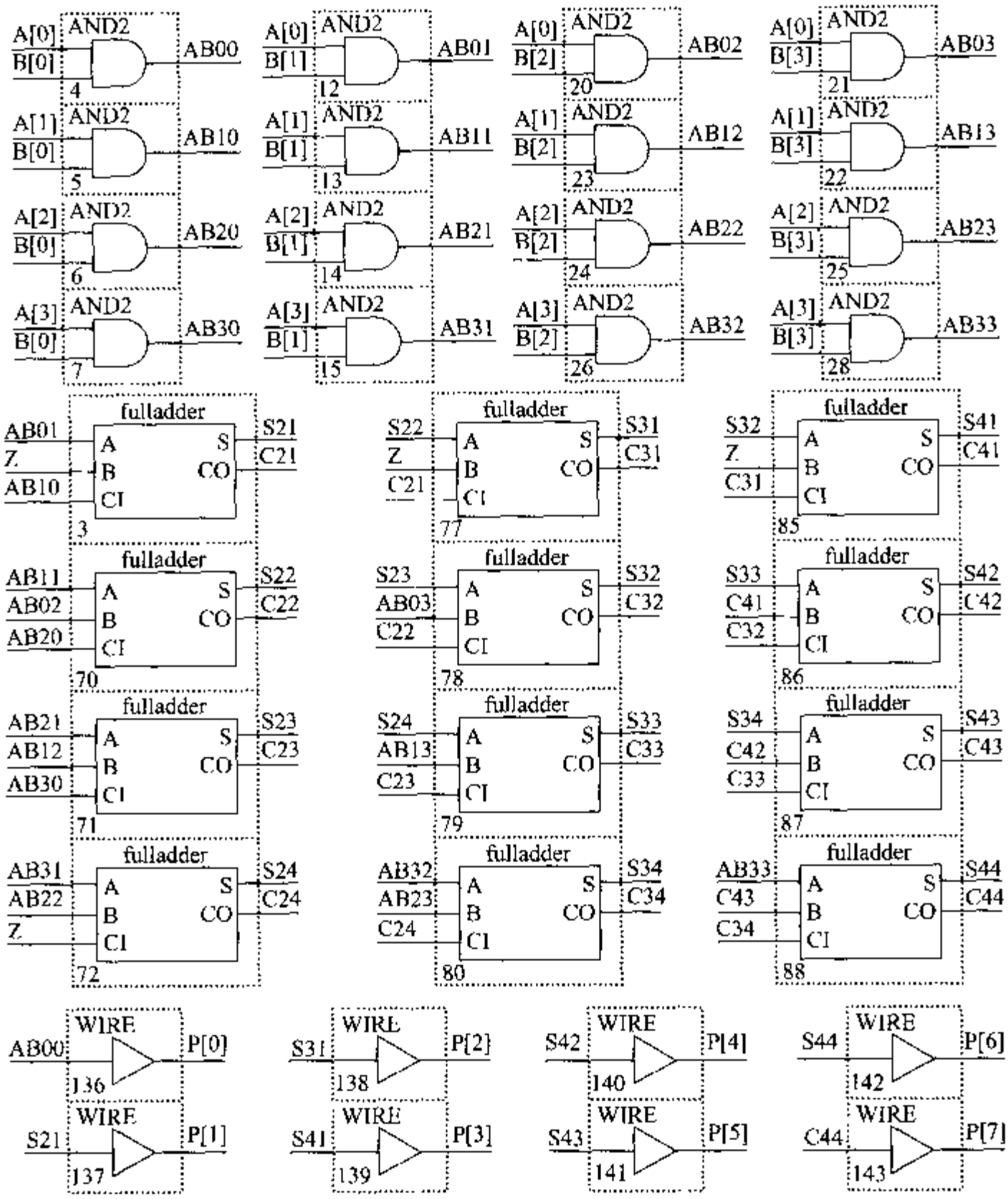


图 4.31 乘法并行阵列逻辑电路图

上面演示了使用乘法并行阵列实现的 4 位乘法器。32 位乘法器也可以使用相同的方法来设计。当使用的加法器位数较多时，可以使用 Wallace 树的方式来搭建一种树形的运算结构来完成计算。

4.4.4 Booth 乘法算法

在上面设计 32 位有符号乘法器时，最重要也最麻烦的是确定 HI 移入位的值。Booth 乘法算法可以简化 HI 移入位的确定。我们先来看图 4.32 所示例子。

普通算法									Booth 算法								
7×6									7×6=7×(8−2)								
					0	1	1	1					0	1	1	1	
×					0	1	1	0	×				0	1	1	0	
+					0	0	0	0	+				0	0	0	0	
+				0	1	1	1		−			0	1	1	1		
+			0	1	1	1			+		0	0	0	0			
+		0	0	0	0				+	0	1	1	1				
	0	0	1	0	1	0	1	0		0	0	1	0	1	0	1	0

图 4.32　Booth 算法

考虑 32 位整数 a 与 b 的乘法运算。正如我们在第 1 章所描述的，用补码表示一个整数时，例如 $b=b_{31}b_{30}\cdots b_1b_0$，不管是正数还是负数，其值的大小的运算公式为 $b_{31}b_{30}\cdots b_1b_0=-b_{31}\times2^{31}+b_{30}\times2^{30}+\cdots+b_1\times2^1+b_0\times2^0$。则

$$a\times b=a\times(-b_{31}\times2^{31}+b_{30}\times2^{30}+\cdots+b_1\times2^1+b_0\times2^0)$$

$$=a\times(2^{31}(b_{30}-b_{31})+2^{30}(b_{29}-b_{30})+\cdots+2^1(b_0-b_1)+2^0(b_{-1}-b_0))$$

其中 $b_{-1}=0$。根据上述表达式，在 Booth 算法中，每次迭代步的操作如表 4.18 所示。

表 4.18　Booth 算法

b		$b_{i-1}-b_i$	操　作
b_i	b_{i-1}		
0	0	0	原部分积右移一次
0	1	+1	加 a，然后右移一次
1	0	−1	减 a，然后右移一次
1	1	0	原部分积右移一次

下面，演示一下 1101×1101，如表 4.19 所示。

表 4.19　Booth 算法演示

运算次数	HI	LO	B	操作说明
1	0 0 0 0 −1 1 0 1 0 0 1 1		1 1 0 1 0	10：减 a，然后右移一次
2	0 0 0 1 +1 1 0 1 1 1 1 0	1	1 1 0 1	01：加 a，然后右移一次
3	1 1 1 1 −1 1 0 1 0 0 1 0	0 1	1 1 0	10：减 a，然后右移一次
4	0 0 0 1	0 0 1	1 1	11：原部分积右移一次
	0 0 0 0	1 0 0 1		

在 Booth 算法中，加 a 和减 a 是交替进行的，不可能有连续的两个加法和连续的两个减法存在，所以，在运算过程中，不存在整数溢出的问题。这样，在每次右移部分积的时候，只需要对部分积进行符号扩展，即 HI 的移入位为部分积的最高位。根据表 4.19，可以得出真值表 4.20。

表 4.20 Booth 乘法器真值表

b_i	b_{i-1}	SUB
0	0	×
0	1	0
1	0	1
1	1	×

根据真值表，可以得出下面的逻辑表达式：

$$SUB = b_i \overline{b_{i-1}}$$

根据上面的论述，可以得出图 4.33 所示的逻辑电路图。

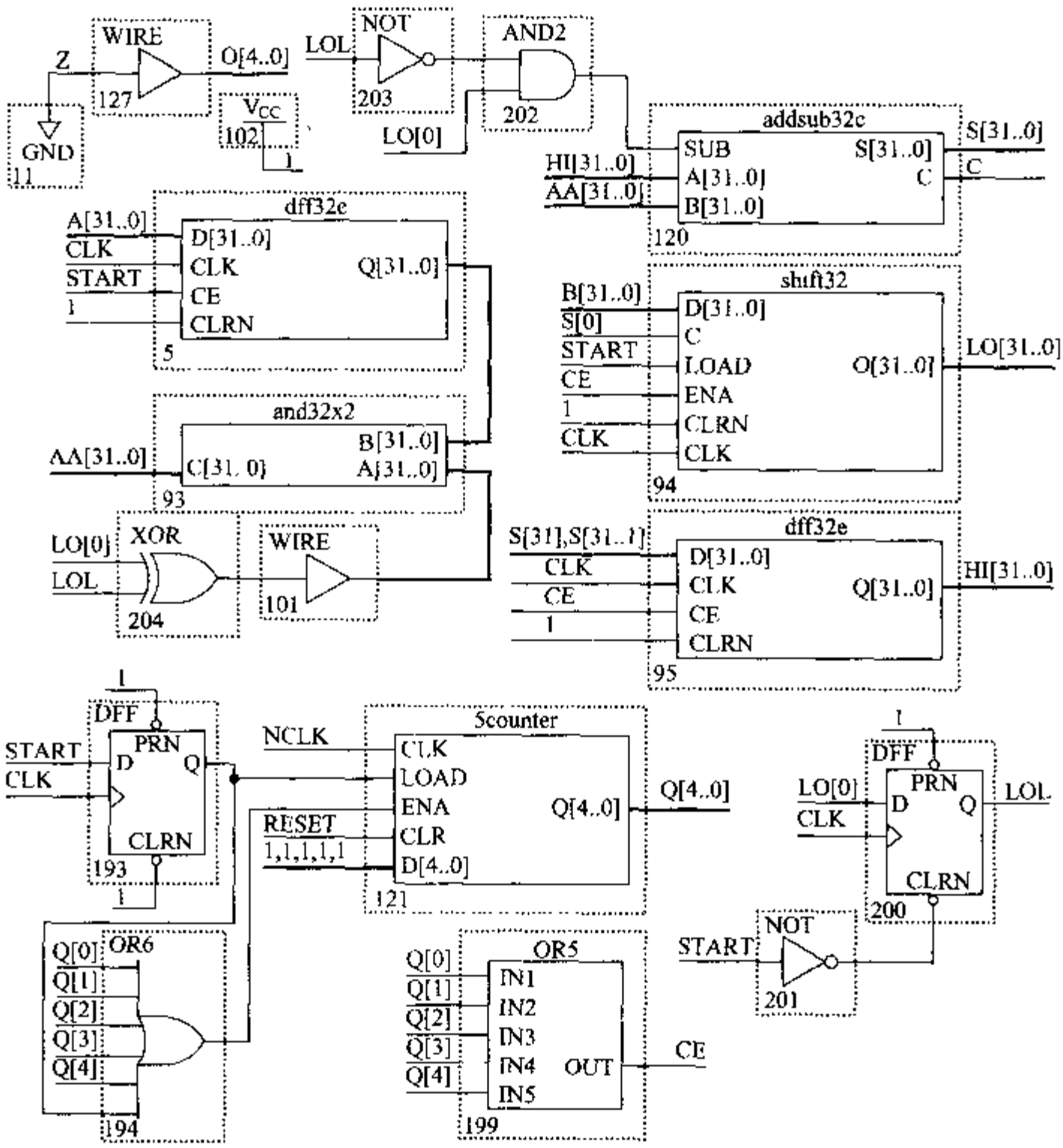


图 4.33 Booth 乘法器逻辑电路图

在上述各种乘法器设计之外，还有很多乘法器的设计方法，这里就不赘述了。

4.5　除法器

首先设计无符号的32位除法器。来看一看二进制除法的运算过程，如图4.34所示。

```
          0011
      ------
0011 ) 1011
       11
      ----
        101
         11
        ---
         10
```

图4.34　4位除法

与乘法运算中的移位相加类似，在除法中是移位相减。乘法可以由多个加法来实现，而除法则可以由多个减法来实现。从左到右依次取被除数的各位，组成部分被除数。判断该部分被除数是否够除数减。如果不够，则在该位上商0，同时补充被除数的下一位，组成更大的数；如果够减，则该位商1，在部分被除数中减去除数，然后，补充被除数的下一位。直至被除数的每一个位都使用了为止。

由此可以看出在除法进行过程中，存在着一个怎样判断部分被除数是否够除数减的问题。有两种方式来解决这个问题。第一种方式是使用专门的数据比较器来比较部分被除数是否比除数大。然后，根据比较的结果采取相应的操作。第二种方式是不管部分被除数是否够除数减，先用部分被除数减去除数，然后，判断产生的差是否为负数。如果为负数，则表明不够减，否则，表明够减。当够减时，这种方式就同时判断了是否够减和完成了减法操作，一举两得。与之相比，第一种方式则浪费了专门的数据比较电路，同时，也加大了运算的延迟，降低了运算的速度。在第二种方式中，当完成了减法操作之后，如果发现不够减，就存在是否要把除数加回到产生的差，以恢复原来的部分被除数上。这个问题也有两种方式来解决。一种方式是加回，这种方式称为恢复余数法。另一种方式是不加回，把产生的差留给下一次运算时再处理，这种方式称为不恢复余数法。下面介绍这两种算法。

4.5.1　恢复余数法

恢复余数法的运算过程如图4.35的运算流程所示。

表4.21为恢复余数法的算法演示，表中为1101/0011。

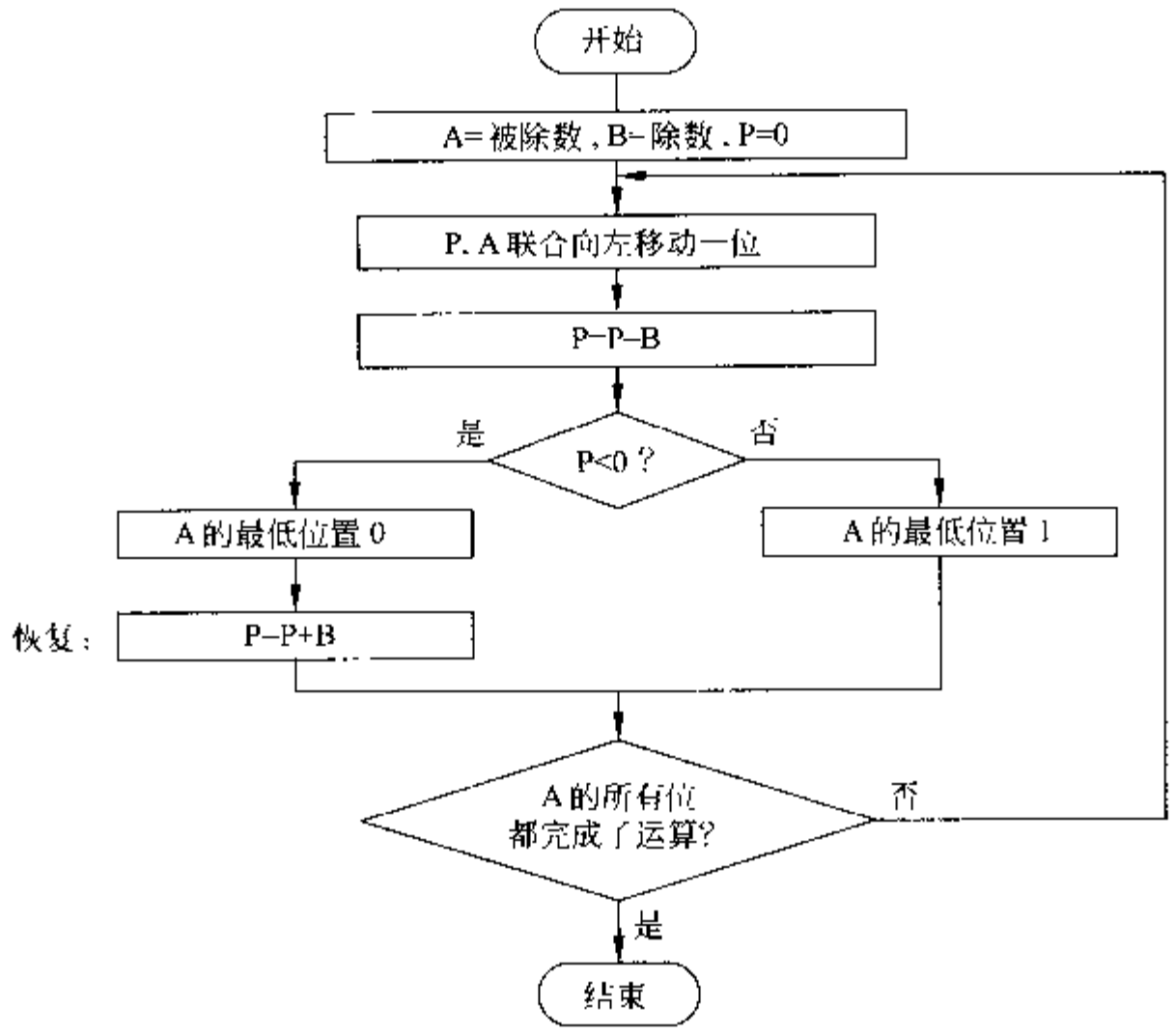


图 4.35 恢复余数法运算流程图

表 4.21 恢复余数法算法演示

运算次数	余数 P	被除数 A	商 Q	操作说明
1	0 0 0 0	1 0 1 1		
	0 0 0 1	0 1 1		PA 左移一位
	0 0 1 1			P=P-B
	1 1 1 0		0	P<0，商 0
	+0 0 1 1			恢复余数
	0 0 0 1			
2	0 0 1 0	1 1		PA 左移一位
	−0 0 1 1			P=P−B
	1 1 1 1		0 0	P<0，商 0
	+0 0 1 1			恢复余数
	0 0 1 0			
3	0 1 0 1	1		PA 左移一位
	−0 0 1 1			P=P−B
	0 0 1 0		0 0 1	P≥0，商 1
4	0 1 0 1			PA 左移一位
	−0 0 1 1			P=P−B
	0 0 1 0		0 0 1 1	P≥0，商 1

4.5.2　不恢复余数法

在恢复余数法中，如果余数 P 为负数，即不够减，则需要加上除数 B，恢复成原来的余数，即 P+B。然后，余数 P 和被除数 A 联合左移 1 位。此时余数为 $2(P+B)+a_i$，其中 a_i 为 A 中的一位，被移入 P 中。随后，进入下一步运算。即从余数中减去除数 B：$(2(P+B)+a_i)-B$。有

$$(2(P+B)+a_i)-B=2P+a_i+B$$

这样，就提供一种方法，当在某一步中，判断余数 P 为负数，无需在该步骤中加上除数 B 恢复成原来的余数，只需在下一步中把恢复余数算法中的减去除数 B 改成加上除数 B，就可以了，这种算法称为不恢复余数法。不恢复余数法也带来一个问题。当计算进行到最后一步时，此时的余数不是中间结果，而是最后结果的余数。所以，如果最后一步中，余数 P 为负数，则需要把余数恢复成原来的余数。

不恢复余数法具体的运算流程如图 4.36 所示。

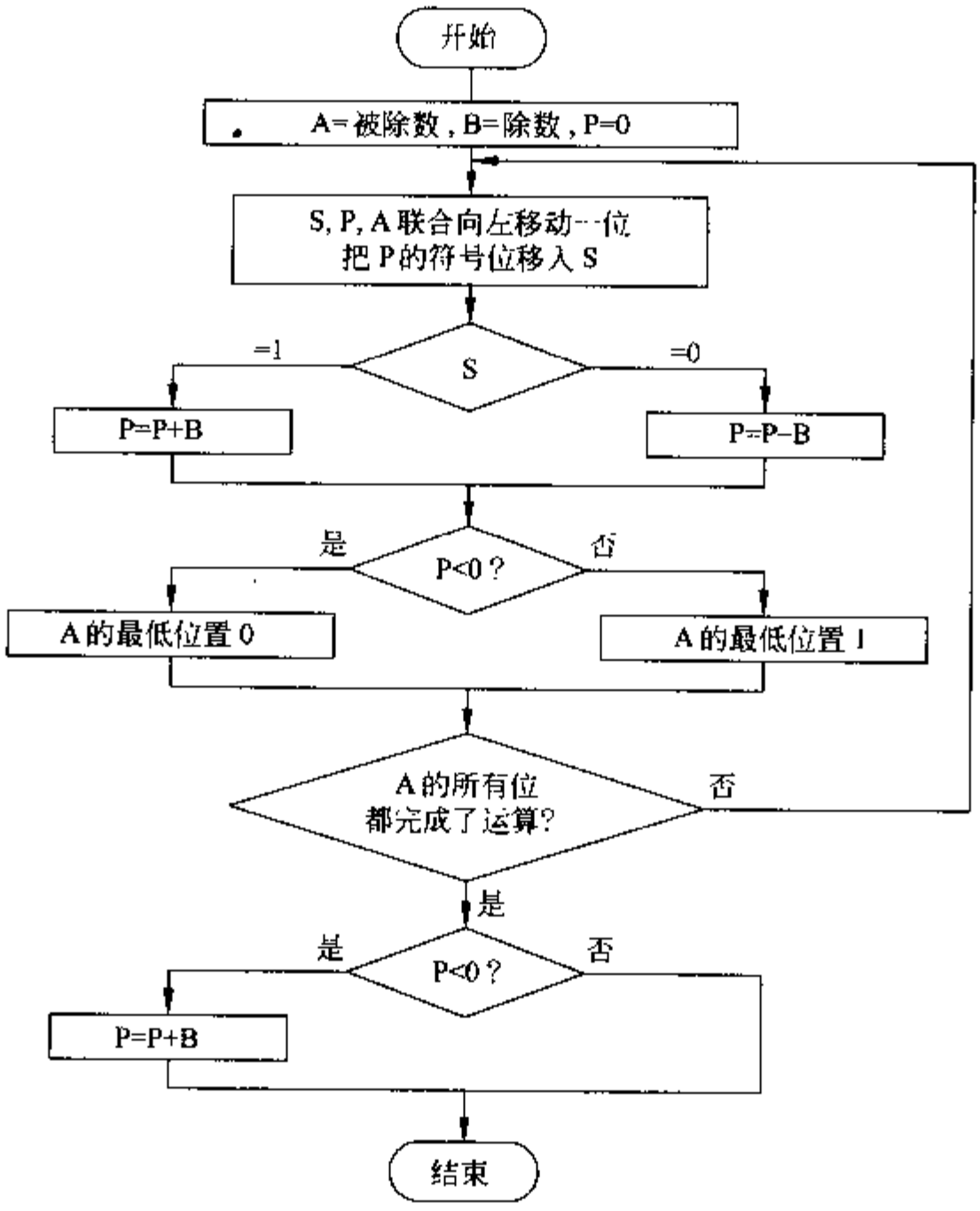


图 4.36　不恢复余数法运算流程图

表 4.22 为不恢复余数法的算法演示，表中为 1101/0011。

表 4.22 不恢复余数法算法演示

运算次数	余数 P	被除数 A	商 Q	操作说明
1	0 0 0 0	1 0 1 1		
	0 0 0 1	0 1 1		PA 左移一位
	−0 0 1 1			P=P−B
	1 1 1 0		0	P<0,商 0
2	1 1 0 0	1 1		PA 左移一位
	+0 0 1 1			S=1,P=P+B
	1 1 1 1		0 0	P<0,商 0
3	1 1 1 1	1		PA 左移一位
	+0 0 1 1			S=1,P=P+B
	0 0 1 0		0 0 1	P≥0,商 1
4	0 1 0 1			PA 左移一位
	−0 0 1 1			S=0,P=P−B
	0 0 1 0		0 0 1 1	P≥0,商 1

根据上述论述,可以得到如表 4.23 所示的真值表。

表 4.23 不恢复余数除法器真值表

R[31]			AS
0			1
1			0
R[31]	S[31]	ASEND	HI
×	×	0	S[30..0],LO[31]
×	0	1	S[31..0]
0	1	1	HI
1	1	1	HI+2×B

表 4.23 中涉及的符号意义如下。

(1) R[31]:上一次加减法器计算结果的符号位,即表 4.22 中的 S 位。

(2) S[31]:当前周期中加减法器计算结果的符号位。

(3) AS:加减法器的运算选择。AS 为 0,表示做加法运算;AS 为 1,做减法运算。

(4) ASEND:表示当前周期为最后一个运算周期。

如表 4.22 所示,若 S 位为 1,则当前周期做加法运算;若 S 位为 0,则当前周期做减法运算。所以,若 R[31]为 0,则 AS 为 1;若 R[31]为 1,则 AS 为 0。

当计算未结束(即 ASEND 为 0)时,寄存器 HI 写入 S[30..0],LO[31]。当计算结束(即 ASEND 为 1)时,需要根据情况恢复最后的余数:

(1) 若当前周期加减法器的计算结果为正数或者为0，即S[31]为0，则当前的计算结果即为最后的余数。

(2) 若当前周期加减法器的计算结果为负数，并且上一个计算周期的计算结果为正数，即S[31]为1且R[31]为0，则表明是当前周期的减法运算造成了余数为负，所以，需要恢复当前周期运算之前的余数，即为减法运算前的HI。

(3) 若当前周期加减法器的计算结果为负数，并且上一个计算周期的计算结果也为负数，即S[31]为1且R[31]为1，则表明当前周期中为余数加上一个B之后，余数仍然为负，所以，必须再为余数加上一个B，从而才能够恢复原来的余数，即余数为HI + 2×B。

对于HI的取值，可以使用一个4通道数据选择器来实现。所以，需要对HI取值的4种情况进行编码，如表4.24所示。

表4.24　HI取值编码真值表

R[31]	S[31]	ASEND	S0	S1
×	×	0	0	0
×	0	1	1	0
0	1	1	0	1
1	1	1	1	1

由上面的真值表，可以得出逻辑表达式：

$$AS = \overline{R[31]}$$

$$S0 = \overline{S[31]}\ ASEND + R[31]\ S[31]\ ASEND$$
$$= (\overline{S[31]} + R[31])\ ASEND$$
$$= (\overline{S[31]} + \overline{AS})\ ASEND$$
$$= \overline{(S[31]\ AS)}\ ASEND$$

$$S1 = \overline{R[31]}\ S[31]\ ASEND + R[31]\ S[31]\ ASEND$$
$$= S[31]ASEND$$

$$HI = \overline{S0}\ \overline{S1}(S[31..0], LO[31]) + S0\ \overline{S1}\ S[31..0] + \overline{S0}\ S1\ HI + S0\ S1\ (HI + 2 \times B)$$

由上述逻辑表达式，可以得出图4.37所示的不恢复余数除法器逻辑电路图。

4.5.3　有符号除法器

下面设计32位有符号除法器。在有符号除法中，有两种方式可以完成运算。第一种方式是先算出被除数和除数的绝对值。然后，使用无符号除法的方式把这两个绝对值相除。这时，得到的商表示被除数可以分成几个完整的除数，得到的余数表示被除数中把所有的完整除数分出去之后剩余的部分。最后，对得到的结果进行符号修正。这种方法比较繁琐，使用的逻辑器件比较多，同时，运算延迟比较长，运算速度不高。第二种方式是直接使用补码方式表示的被除数和除数进行带符号运算。在这种情况下，需要小心地处理运算过程中的加减、符号问题。

有符号整数除法分为以下几种情况。

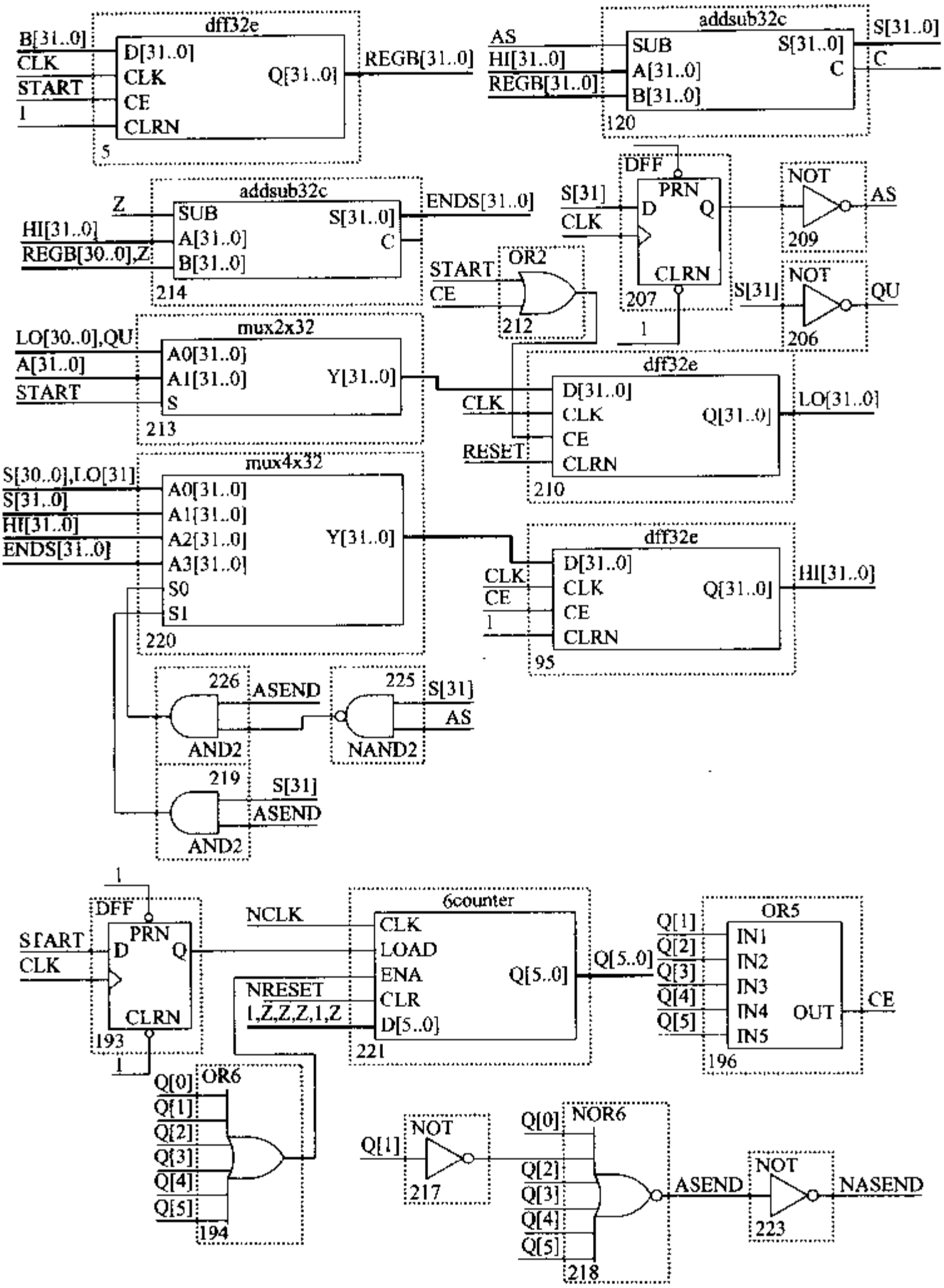


图 4.37 不恢复余数除法器

1. 正数除以正数

这种情况下，最后的运算结果为“正数/正数＝正数……正数或 0”。

在运算过程中，使用“正数/正数＝正数……正数或 0”形式。每一个迭代步中，基本

的运算为“减法”。如果发生不够减的情况，即余数为负数时，则在下一个迭代步中用余数加上除数。

2. 正数除以负数

这种情况下，最后的运算结果为“正数/负数＝负数……正数或 0”。

在运算过程中，使用“正数/负数＝正数……正数或 0”形式。每一个迭代步中，基本的运算为“加法”。如果发生不够加的情况，即余数为正数时，则在下一个迭代步中用余数减去除数。

3. 负数除以正数

这种情况下，最后的运算结果为“负数/正数＝负数……负数或 0”。

在运算过程中，使用“负数/正数＝正数……负数或 0”形式。每一个迭代步中，基本的运算为“加法”。如果发生不够加的情况，即余数为正数时，则在下一个迭代步中用余数减去除数。

4. 负数除以负数

这种情况下，最后的运算结果为“负数/负数＝正数……负数或 0”。

在运算过程中，使用“负数/负数＝正数……负数或 0”。每一个迭代步中，基本的运算为“减法”。如果发生不够减的情况，即余数为负数时，则在下一个迭代步中用余数加上除数。

在上述第 2、3 种情况中，最后结果需要得到负数形式的商，但在运算过程中则使用正数形式的商。这个正数形式的商就表示被除数绝对值中能够分出多少个除数绝对值大小的数。这正是无符号整数除法中商的意义。为了解决商的运算中间结果和最终解决不一致的问题，需要在运算完成后把商的中间结果取反，得到最终结果形式的商。

综合上述各种情况，可以得出表 4.25 所示的运算规则。

表 4.25　有符号除法运算规则

A 符号	B 符号	中间余数符号	商符号	上商	下一步运算
0	0	0	0	余数非负(够减)：1	余数为负(不够减)：0
				余数非负(够减)：减	余数为负(不够减)：加
0	1	0	1*	余数非负(够加)：1	余数为负(不够加)：0
				余数非负(够加)：加	余数为负(不够加)：减
1	0	1	1*	余数非正(够加)：1	余数为正(不够加)：0
				余数非负(够加)：加	余数为负(不够加)：减
1	1	1	0	余数非正(够减)：1	余数为正(不够减)：0
				余数非负(够减)：减	余数为负(不够减)：加

* 商为 0 时，符号位为 0。

在有符号除法中，主要有如下控制信号：

(1) 在除法运算过程中，很重要的一点是判断够不够减(加)，故引入逻辑变量 NEN。若 NEN 为 0，表示够减(加)；若 NEN 为 1，表示不够减(加)。

(2) 保存被除数 A 的符号位 ASIGN。

(3) 保存除数 B[31..0]为 REGB[31..0]。

(4) 在每一个运算步中，加法或者减法执行的结果为 S[31..0]。SZ 表示 S[31..0]是否为 0。若 SZ 为 1，则表示 S[31..0]为 0；若 SZ 为 0，则表示 S[31..0]不为 0。

(5) 加减法的控制信号为 AS。若 AS 为 0，表示执行加法；若 AS 为 1，表示执行减法。

(6) 每一位的上商用 QU 表示。

(7) 余数部分保存在 HI[31..0]中。

(8) ASEND 表示最后一个运算步。

根据有符号除法的运算规则，可以得出各种控制信号的真值表(表 4.26～表 4.28)。

表 4.26 有符号除法器控制信号真值表

ASIGN	REGB[31]	NEN_{i-1}	AS
0	0	0	1
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

表 4.27 有符号除法器控制信号真值表

ASIGN	REGB[31]	S[31]	SZ	QU	NEN_i
0	0	0	×	1	0
0	1	0	×	1	0
1	0	0	0	0	1
1	1	0	0	0	1
0	0	1	×	0	1
0	1	1	×	0	1
1	0	1	×	1	0
1	0	×	1	1	0
1	1	1	×	1	0
1	1	×	1	1	0

表 4.28　有符号除法器控制信号真值表

ASIGN	REGB[31]	S[31]	SZ	NEN_{i-1}	ASEND	HI
×	×	×	×	×	0	S[31..0]
0	0	0	×	×	1	S[31..0]
0	0	1	×	0	1	HI[30..0],LO[31]
0	0	1	×	1	1	(HI[30..0],LO[31])+2×B
0	1	0	×	×	1	S[31..0]
0	1	1	×	0	1	HI[30..0],LO[31]
0	1	1	×	1	1	(HI[30..0],LO[31])−2×B
1	0	1	×	×	1	S[31..0]
1	0	×	1	×	1	S[31..0]
1	0	0	0	0	1	HI[30..0],LO[31]
1	0	0	0	1	1	(HI[30..0],LO[31])−2×B
1	1	1	×	×	1	S[31..0]
1	1	×	1	×	1	S[31..0]
1	1	0	0	0	1	HI[30..0],LO[31]
1	1	0	0	1	1	(HI[30..0],LO[31])+2×B

根据表 4.26～表 4.28，可以得出下面的逻辑表达式。

$$AS = \overline{NEN_{i-1}} \cdot \overline{ASIGN \oplus REGB[31]} + NEN_{i-1} \cdot (ASIGN \oplus REGB[31])$$

$$= \overline{NEN_{i-1} \oplus ASIGN \oplus REGB[31]}$$

$$QU = \overline{ASIGN} \cdot \overline{S[31]} + ASIGN \cdot S[31] + ASIGN \cdot SZ$$

$$= \overline{ASIGN \oplus S[31]} + ASIGN \cdot SZ$$

$$NEN_i = \overline{QU} = \overline{\overline{ASIGN \oplus S[31]} + ASIGN \cdot SZ}$$

$$HI = \overline{ASEND} \cdot S[31]$$

$$+ ASEND \cdot ((\overline{ASIGN} \cdot \overline{S[31]} + ASIGN \cdot S[31] + ASIGN \cdot SZ) \cdot S[31..0]$$

$$+ (\overline{ASIGN} \cdot S[31] \cdot \overline{NEN_{i-1}} + ASIGN \cdot \overline{S[31]} \cdot \overline{SZ} \cdot \overline{NEN_{i-1}})$$

$$\cdot (HI[31..0],LO[31])$$

$$+ ASIGN \cdot S[31] \cdot NEN_{i-1} \cdot (\overline{REGB[31]} \cdot ((HI[31..0],LO[31]) - 2B)$$

$$+ REGB[31] \cdot ((HI[31..0],LO[31]) + 2B))$$

$$+ ASIGN \cdot \overline{S[31]} \cdot \overline{SZ} \cdot NEN_{i-1} \cdot (\overline{REGB[31]} \cdot ((HI[31..0],LO[31]) - 2B)$$

$$+ \mathrm{REGB}[31] \cdot ((\mathrm{HI}[31..0], \mathrm{LO}[31]) + 2\mathrm{B})))$$

$$= (\overline{\mathrm{ASEND}} + \mathrm{ASEND} \cdot (\overline{\mathrm{ASIGN}} \cdot \overline{\mathrm{S}[31]}$$

$$+ \mathrm{ASIGN} \cdot \mathrm{S}[31] + \mathrm{ASIGN} \cdot \mathrm{SZ})) \cdot \mathrm{S}[31..0]$$

$$+ \mathrm{ASEND} \cdot (\overline{\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31]} \cdot \overline{\mathrm{NEND}_{i-1}} + \mathrm{ASIGN} \cdot \overline{\mathrm{S}[31]} \cdot \overline{\mathrm{SZ}} \cdot \overline{\mathrm{NEN}_{-1}})$$

$$\cdot (\mathrm{HI}[31..0], \mathrm{LO}[31])$$

$$+ \mathrm{ASEND} \cdot (\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31] \cdot \mathrm{NEN}_{i-1} + \mathrm{ASIGN} \cdot \overline{\mathrm{S}[31]} \cdot \overline{\mathrm{SZ}} \cdot \mathrm{NEN}_{i-1})$$

$$\cdot (\overline{\mathrm{ASIGN} \oplus \mathrm{REGB}[31]} \cdot ((\mathrm{HI}[31..0], \mathrm{LO}[31]) - 2\mathrm{B})$$

$$+ (\mathrm{ASIGN} \oplus \mathrm{REGB}[31]) \cdot ((\mathrm{HI}[31..0], \mathrm{LO}[31]) + 2\mathrm{B}))$$

设

$$a = \overline{\mathrm{ASEND}} + \mathrm{ASEND} \cdot (\overline{\mathrm{ASIGN}} \cdot \overline{\mathrm{S}[31]} + \mathrm{ASIGN} \cdot \mathrm{S}[31] + \mathrm{ASIGN} \cdot \mathrm{SZ})$$

$$b = \mathrm{ASEND} \cdot (\overline{\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31]} \cdot \overline{\mathrm{NEN}_{i-1}} + \mathrm{ASIGN} \cdot \overline{\mathrm{S}[31]} \cdot \overline{\mathrm{SZ}} \cdot \overline{\mathrm{NEN}_{-1}})$$

$$c = \mathrm{ASEND} \cdot (\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31] \cdot \mathrm{NEN}_{i-1} + \mathrm{ASIGN} \cdot \overline{\overline{\mathrm{S}[31]}} \cdot \overline{\mathrm{SZ}} \cdot \mathrm{NEN}_{-1})$$

我们使用 4 通道的数据选择器来选择 HI 的值。这就需要对 HI 逻辑表达式中的 a，b，c 进行编码，如表 4.29 所示。

表 4.29 选取编码真值表

a	b	c	S0	S1
1	0	0	0	0
0	1	0	1	0
0	0	1	0	1

由真值表 4.29 可以得出下面的逻辑表达式。

$$S0 = b = \mathrm{ASEND} \cdot (\overline{\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31]} \cdot \overline{\mathrm{NEN}_{i-1}} + \mathrm{ASIGN} \cdot \overline{\mathrm{S}[31]} \cdot \overline{\mathrm{SZ}} \cdot \overline{\mathrm{NEN}_{-1}})$$

$$S1 = c = \mathrm{ASEND} \cdot (\overline{\mathrm{ASIGN}} \cdot \mathrm{S}[31] \cdot \mathrm{NEN}_{i-1} + \mathrm{ASIGN} \cdot \overline{\mathrm{S}[31]} \cdot \overline{\mathrm{SZ}} \cdot \mathrm{NEN}_{i-1})$$

由上面的逻辑表达式，可以得出图 4.38 所示的逻辑电路图。

下面，将无符号除法器和有符号除法器结合起来，设计成完整的 32 位除法器。引入逻辑变量 SIGNED，表示是否为无符号整数除法。同时，把有符号除法器中除数 B 的符号位单独保存成 BSIGN。无符号整数除法和有符号除法中“正数/正数”的计算过程完全相同。所以，当进行无符号整数除法时，只需要把 ASIGN 和 BSIGN 设置成 0，就可以了。32 位除法器的逻辑电路图如图 4.39 所示。

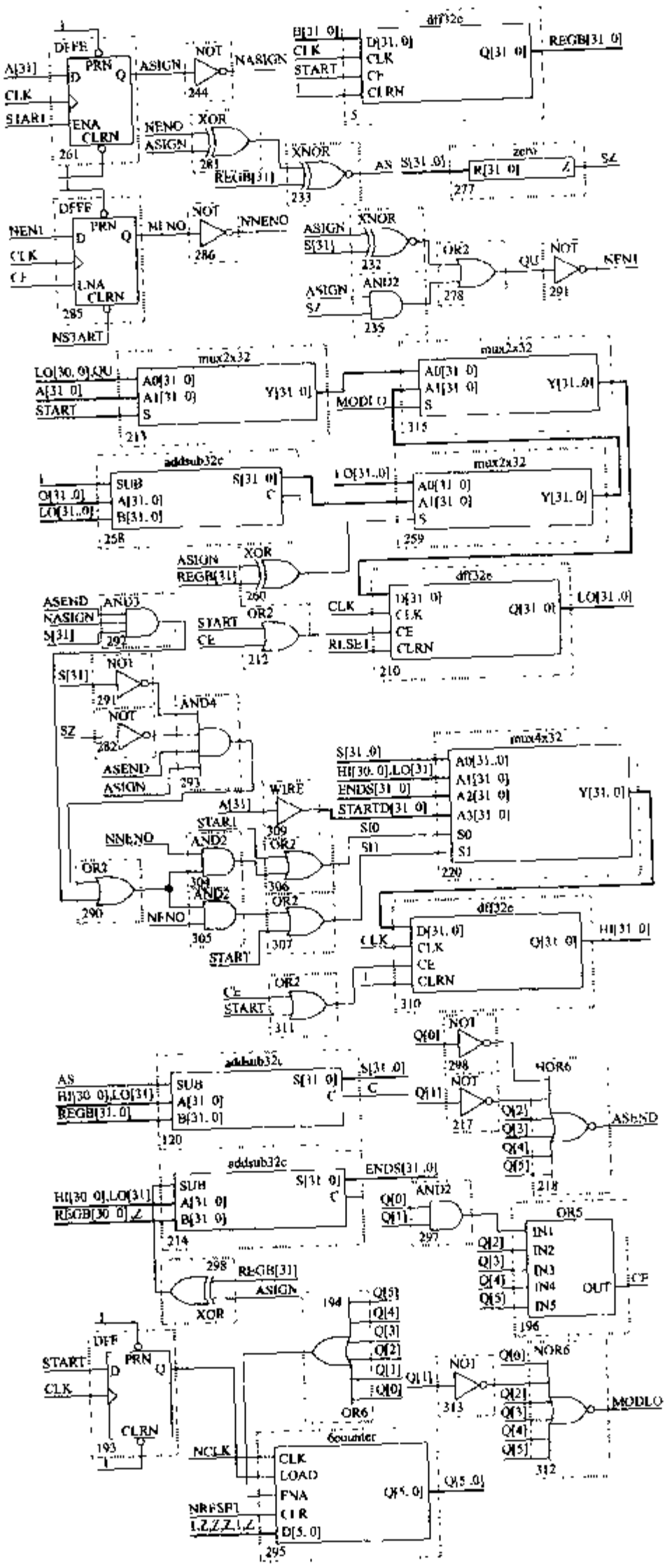


图 4.38　有符号除法逻辑电路图

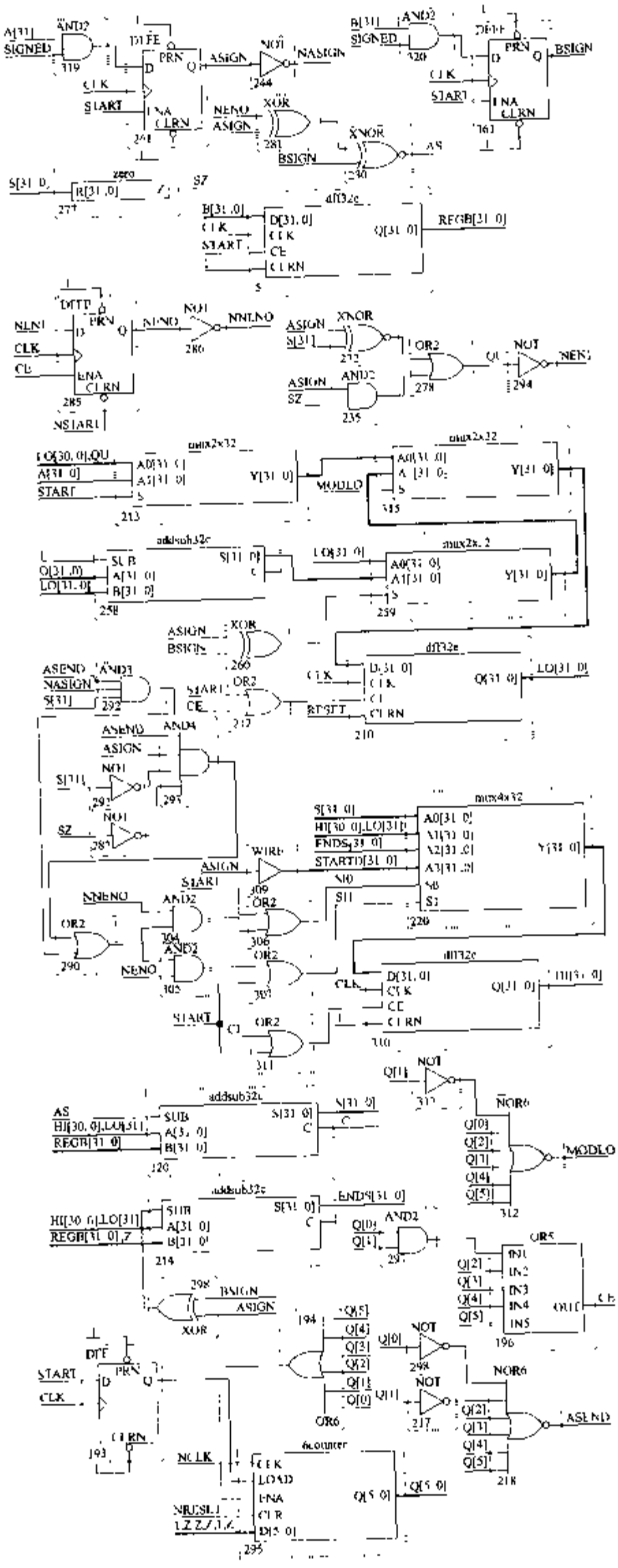


图 4.39　32 位除法器逻辑电路图

4.6　移位器

在 MIPS 中，有以下 3 种移位操作：(1)逻辑移位，(2)算术移位，(3)循环移位。逻辑移位中又分为逻辑右移和逻辑左移两种。

4.6.1　逻辑移位

逻辑移位是把操作数看作逻辑数。逻辑数没有符号位，也就是说，在进行操作时，每个位的地位都是一样的，都要参与操作。下面分别介绍逻辑右移和逻辑左移。

1. 逻辑右移

逻辑右移是把操作数按照给出的移位次数右移，左侧移空的数位上移入 0，抛弃右侧移出的位，如下所示。

```
                    1101101011101011
逻辑右移 6 位  →    0000001101101011
                    移入位补 0
```

2. 逻辑左移

逻辑左移是把操作数按照给出的移位次数左移，右侧移空的数位上移入 0，抛弃左侧移出的位，如下所示。

```
                    1101101011101011
逻辑左移 6 位  →    1011101011000000
                                移入位补 0
```

4.6.2　算术移位

与逻辑移位不同，算术移位把操作数看作是有符号的数。操作数的最高位为符号位，与其他的位不同。

算术移位是把操作数按照给出的移位次数右移，左侧移空的数位上移入原来操作数的符号位，抛弃右侧移出的位，如下所示。

```
                    1101101011101011
算术右移 6 位  →    1111111101101011
                    移入位符号扩展原来的高位
```

4.6.3 循环移位

循环移位是把操作数按照给出的移位次数向某个方向循环移位，即一侧移出的位作为另一次移入的位。

在 MIPS 中，只定义了循环右移，即操作数右移，右侧移出的位作为左侧移入的位，如下所示。

1101101011101011

循环右移 6 位 → 1010111101101011

移入位为原来的低位

为了把这些移位操作集成在一个移位器中，需要对这些运算做一个编码，如表 4.30 所示。

表 4.30 移位操作编码

操 作	CONTROL[1..0]
逻辑右移	00
算术右移	01
逻辑左移	10
循环移位	11

根据以上各种移位操作的介绍，在移位操作中，有以下几个变量：

(1) A[31..0]，操作数。

(2) CONTROL[1..0]，控制字，如表 4.30 所示。

(3) S[4..0]，移位次数，如表 4.31 所示。

表 4.31 移位次数

S4＝1	A[31..0]移 2^4 位
S3＝1	A[31..0]移 2^3 位
S2＝1	A[31..0]移 2^2 位
S1＝1	A[31..0]移 2^1 位
S0＝1	A[31..0]移 2^0 位

图 4.40 示出的是移位操作的总体逻辑电路图。

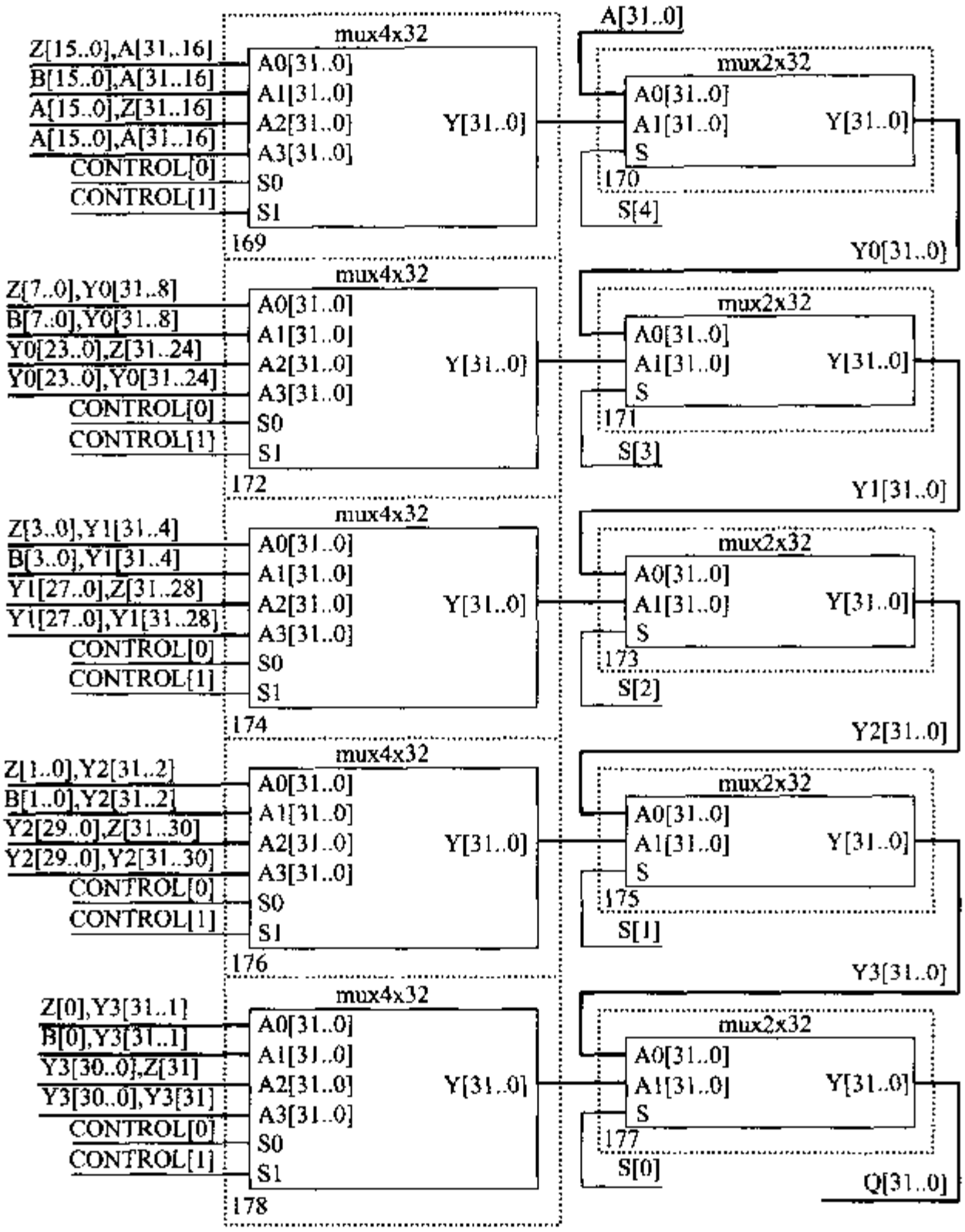


图 4.40　各种移位操作集成在一起的逻辑电路图

4.7　首 0/1 计数器

4.7.1　首 1 计数器

首 1 计数器(count leading ones,CLO)的功能是从高位向低位扫描操作数,计算操作数字首有多少个连续的 1。如果操作数为 0,则扫描结果置为 32。首 1 计数器逻辑中有如下逻辑变量:

(1) 操作数 A[31..0];

(2) 扫描结果 P[5..0]。

表 4.32 为首 1 计数器的真值表。

表 4.32 32 位首 1 计数器真值表

A[31]	A[30]	A[29]	A[28]	…	A[1]	A[0]	P[5]	P[4]	P[3]	P[2]	P[1]	P[0]
0	×	×	×	…	×	×	0	0	0	0	0	0
1	0	×	×	…	×	×	0	0	0	0	0	1
1	1	0	×	…	×	×	0	0	0	0	1	0
1	1	1	0	…	×	×	0	0	0	0	1	1
				⋮						⋮		
1	1	1	1	…	1	0	0	1	1	1	1	1
1	1	1	1	…	1	1	1	0	0	0	0	0

根据真值表 4.32，可以得出如下的逻辑表达式：

$$\begin{aligned}
P[0] = {} & A[31] \cdot \overline{A[30]} \\
& + A[31] \cdot A[30] \cdot A[29] \cdot \overline{A[28]} \\
& + \cdots \\
& + A[31] \cdot A[30] \cdot \oplus \cdot A[1] \cdot \overline{A[0]} \\
P[1] = {} & A[31] \cdot A[30] \cdot \overline{A[29]} \\
& + A[31] \cdot A[30] \cdot A[29] \cdot \overline{A[28]} \\
& + \cdots \\
& + A[31] \cdot A[30] \cdot \cdots \cdot A[1] \cdot \overline{A[0]} \\
& \vdots \\
P[5] = {} & A[31] \cdot A[30] \cdot \cdots \cdot A[1] \cdot A[0]
\end{aligned}$$

可见，首 1 计数器的逻辑表达式比较复杂，直接使用电路来实现这些逻辑表达式比较困难。所以，首先实现一个 8 位的首 1 计数器。然后，使用这个 8 位的首 1 计数器来实现 32 位的首 1 计数器。8 位首 1 计数器逻辑中有如下逻辑变量：

(1) 操作数 A[7..0]；

(2) 扫描结果 P[3..0]。

8 位首 1 计数器的真值表如表 4.33 所示。

表 4.33 8 位首 1 计数器真值表

A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	P[3]	P[2]	P[1]	P[0]
0	×	×	×	×	×	×	×	0	0	0	0
1	0	×	×	×	×	×	×	0	0	0	1
1	1	0	×	×	×	×	×	0	0	1	0
1	1	1	0	×	×	×	×	0	0	1	1
1	1	1	1	0	×	×	×	0	1	0	0
1	1	1	1	1	0	×	×	0	1	0	1
1	1	1	1	1	1	0	×	0	1	1	0
1	1	1	1	1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0

根据真值表 4.33，可以得出下面的逻辑表达式。

$$P[0]= A[7]\overline{A[6]}$$
$$+ A[7]A[6]A[5]\overline{A[4]}$$
$$+ A[7]A[6]A[5]A[4]A[3]\overline{A[2]}$$
$$+ A[7]A[6]A[5]A[4]A[3]A[2]A[1]\overline{A[0]}$$
$$P[1]= A[7]A[6]\overline{A[5]}$$
$$+ A[7]A[6]A[5]\overline{A[4]}$$
$$+ A[7]A[6]A[5]A[4]A[3]A[2]\overline{A[1]}$$
$$+ A[7]A[6]A[5]A[4]A[3]A[2]A[1]\overline{A[0]}$$
$$P[2]= A[7]A[6]A[5]A[4]\overline{A[3]}$$
$$+ A[7]A[6]A[5]A[4]A[3]\overline{A[2]}$$
$$+ A[7]A[6]A[5]A[4]A[3]A[2]\overline{A[1]}$$
$$+ A[7]A[6]A[5]A[4]A[3]A[2]A[1]\overline{A[0]}$$
$$P[3]= A[7]A[6]A[5]A[4]A[3]A[2]A[1]A[0]$$

根据逻辑表达式，可以得出图 4.41 所示的逻辑电路图。

要实现 32 位首 1 计数器需要使用 4 个 8 位首 1 计数器。32 位首 1 计数器涉及下面的逻辑变量：

(1) 操作数 A[31..0]；

(2) 4 个 8 位首 1 计数器输入 A0[7..0]，A1[7..0]，A2[7..0]，A3[7..0]；

(3) 4 个 8 位首 1 计数器输出 P0[3..0]，P1[3..0]，P2[3..0]，P3[3..0]；

(4) 扫描结果 P[5..0]。

令 A3[7..0]=A[31..24]，A2[7..0]=A[23..16]，A1[7..0]=A[15..8]，A0[7..0]=A[7..0]。表 4.34 为 32 位首 1 计数器真值表。

由真值表 4.34 可得到下面的逻辑表达式：

$$P[5..0]= \overline{P3[3]} \cdot (000,P3[2..0])$$
$$+ P3[3] \cdot \overline{P[2]} \cdot (001,P2[2..0])$$
$$+ P3[3] \cdot P[2] \cdot \overline{P[1]} \cdot (010,P1[2..0])$$
$$+ P3[3] \cdot P[2] \cdot P[1] \cdot \overline{P[0]} \cdot (011,P0[2..0])$$
$$+ P3[3] \cdot P[2] \cdot P[1] \cdot P[0] \cdot (100000)$$

由逻辑表达式，可以得到图 4.42 所示的逻辑电路图。

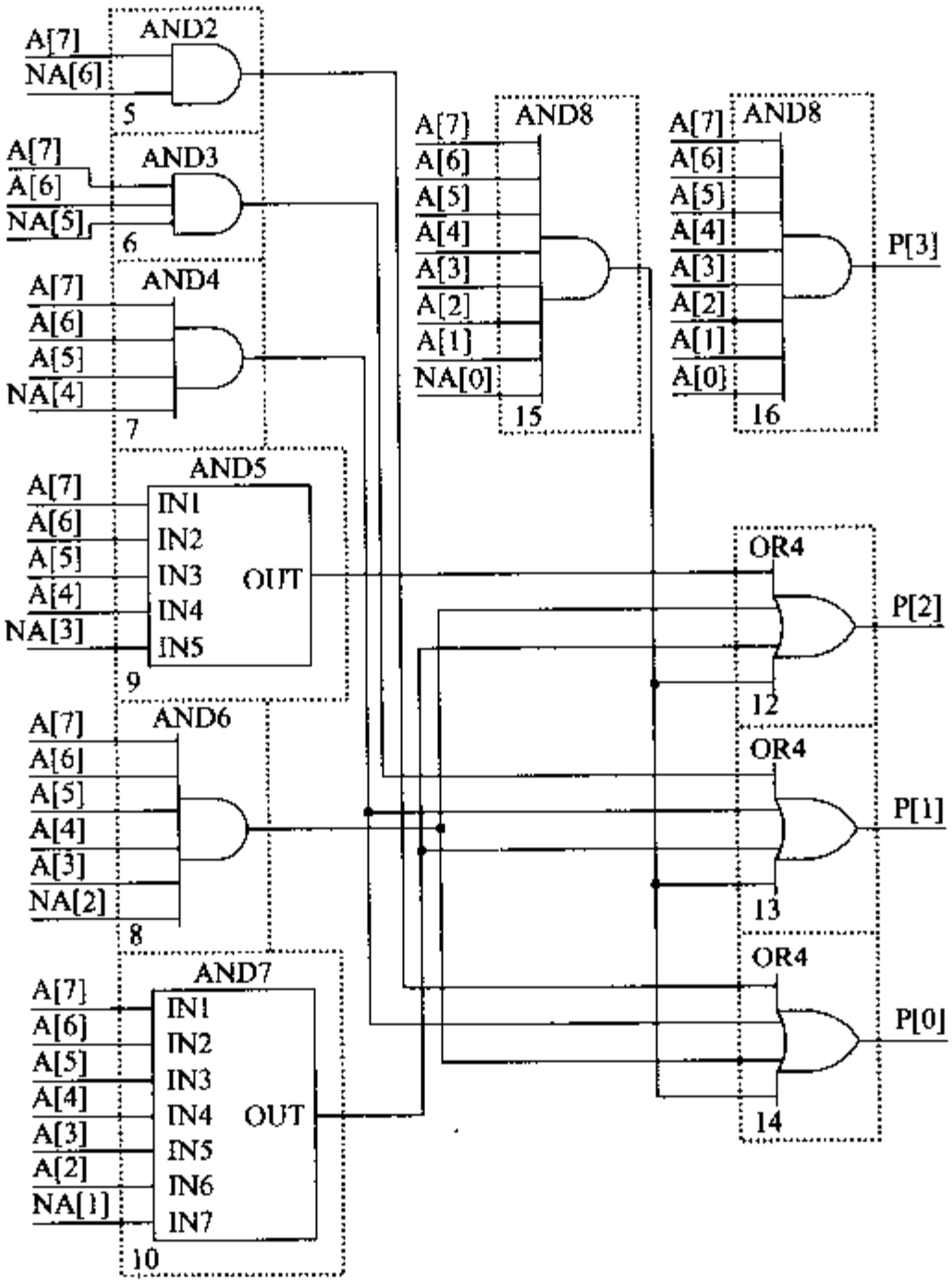


图 4.41　8 位首 1 计数器逻辑电路图

表 4.34　32 位首 1 计数器真值表

P3[3]	P2[3]	P1[3]	P0[3]	P[5..0]
0	×	×	×	000,P3[2..0]
1	0	×	×	001,P2[2..0]
1	1	0	×	010,P1[2..0]
1	1	1	0	011,P0[2..0]
1	1	1	1	100000

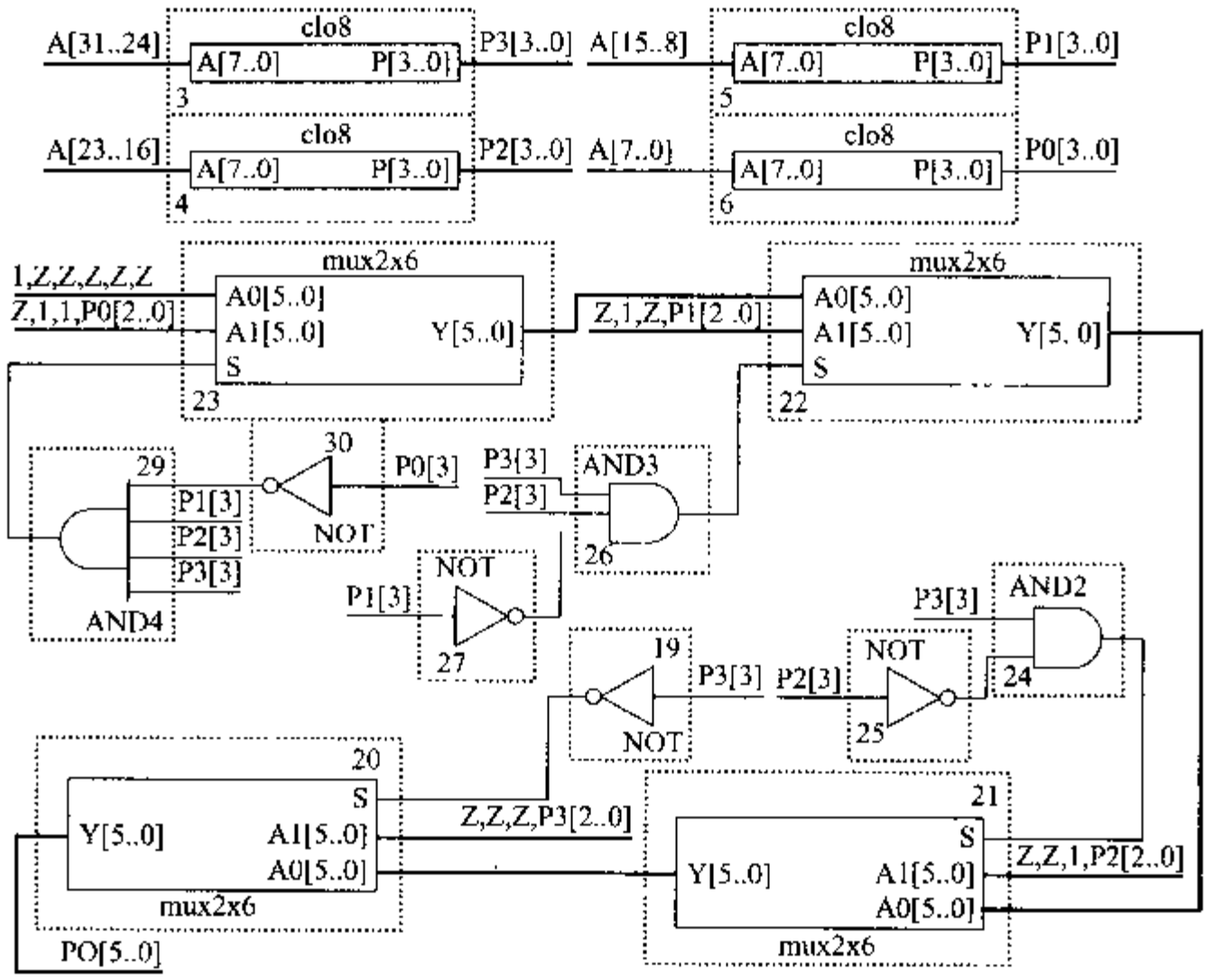


图 4.42　32 位首 1 计数器逻辑电路图

4.7.2　首 0 计数器

首 0 计数器(count leading zeros,CLZ)的功能是从高位向低位扫描操作数,计算操作数字首有多少个连续的 0。如果操作数中的每个位都为 1,则扫描结果置为 32。

由此可见,如果把首 0 计数器的操作数按位取反,然后对结果做首 1 计数,则得到结果就是原来操作数首 0 计数的结果。所以,首 0 计数器的逻辑电路图如图 4.43 所示。

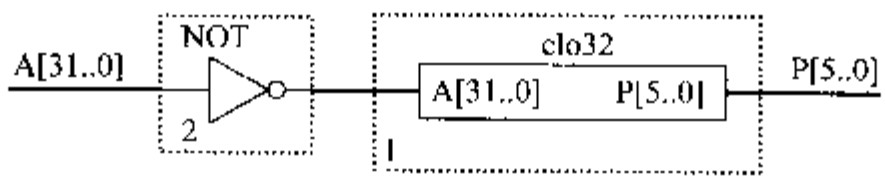


图 4.43　首 0 计数器逻辑电路图

4.8　比较器

比较器的功能是比较两个有符号整数的大小,然后给出比较的结果。涉及比较器的指令有:SLT。操作:SLT←(A<B)比较器的真值表如表 4.35 所示,其中 S=A−B。

表 4.35　比较器真值表

A[31]	B[31]	S[31]	SLT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

由真值表 4.35，可以得到比较器的逻辑表达式：

$$SLT = \overline{(A[31] \oplus B[31])} \cdot S[31] + (A[31] \oplus B[31]) \cdot A[31]$$

根据比较器的逻辑表达式，可以得出如图 4.44 所示的逻辑电路图。

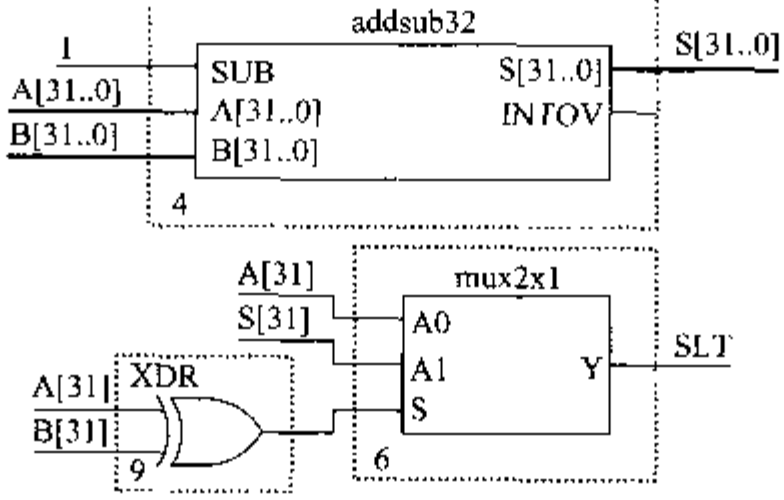


图 4.44　比较器逻辑电路图

4.9　ALU 设计

在完成了各种算术运算部件和逻辑运算部件的设计之后，就可以设计算术逻辑单元 ALU 了。

算术逻辑单元 ALU 集成了各种算术运算和逻辑运算部件的功能，包括加、减、乘、除、逻辑运算、移位运算等。把这些功能集成在一个逻辑部件 ALU 之中，使得 ALU 同时具有算术运算和逻辑运算功能。这种设计方法可以使功能比较紧凑，简化对逻辑运算部件和算术运算部件的使用。同时，还能够最大限度地复用某些逻辑部件，从而减少逻辑电

路的使用。

设计 ALU 首先要做的就是对各种算术、逻辑运算进行编码。表 4.36 为 ALU 功能编码表。

表 4.36　ALU 功能编码表

指令类型	CONTROL[4..0]	功　能
逻辑指令	00 000	A 与 B
	01 000	A 或 B
	10 000	A 或非 B
	11 000	A 异或 B
加减运算指令	×0 001	A 加 B
	×1 001	A 减 B
比较指令	×1 010	A<B
加载指令	×× 011	把 B 的低 16 位加载到高 16 位上
移位指令	00 100	B 逻辑右移 A 位
	01 100	B 算术右移 A 位
	10 100	B 逻辑左移 A 位
	11 100	B 循环移位 A 位
首 0、1 计数指令	×0 101	对 A 首 1 计数
	×1 101	对 A 首 0 计数
乘法指令	00 110	A 乘 B,无符号乘
	01 110	A 乘 B,有符号乘
除法指令	00 111	A 除 B,无符号除
	01 111	A 除 B,有符号除

根据上面的 ALU 功能编码表,可以得到如图 4.45 所示的逻辑电路图。

4.10　小结

本章介绍了算术逻辑单元 ALU 的设计,介绍了各种常用电路的算法和电路实现。ALU 将具体执行 CPU 指令中涉及的算术操作和逻辑操作。

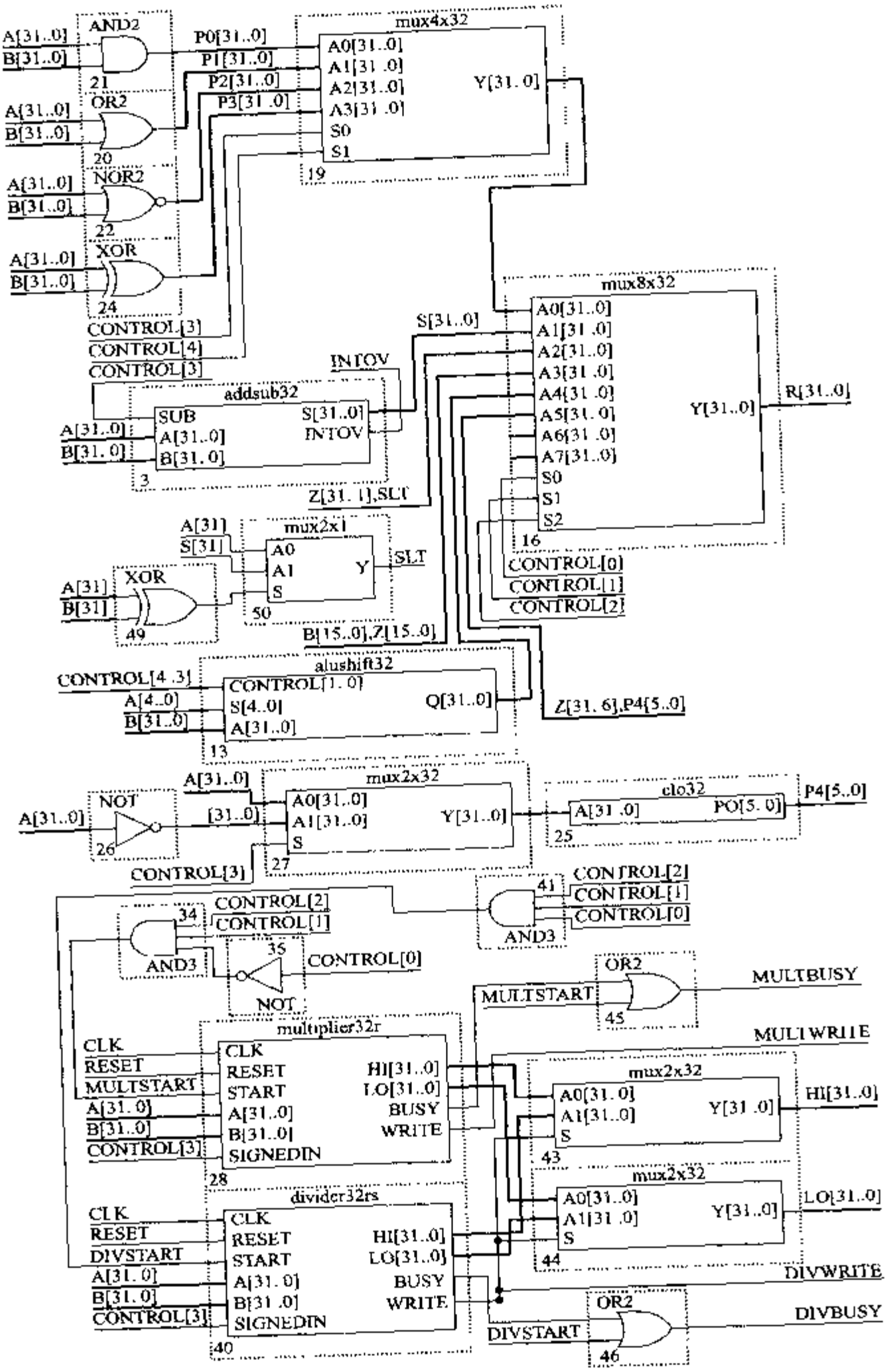


图 4.45 ALU 逻辑电路图

第5章

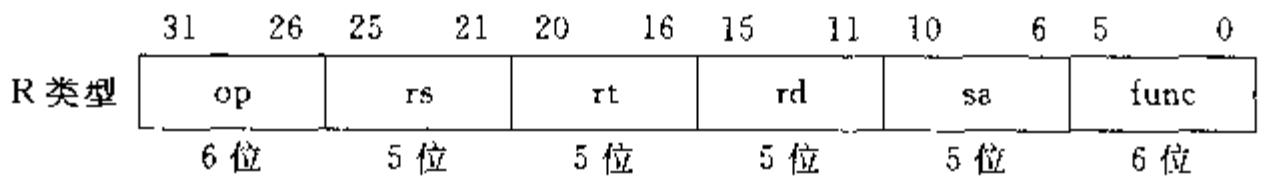
单周期 CPU 设计

在第 2 章和第 3 章中，我们介绍了 MIPS CPU 的指令集系统和 ALU 的设计。本章将给出具体的 MIPS 指令集实现。本章中设计的 CPU 只是实现 MIPS 指令集中的一个子集。

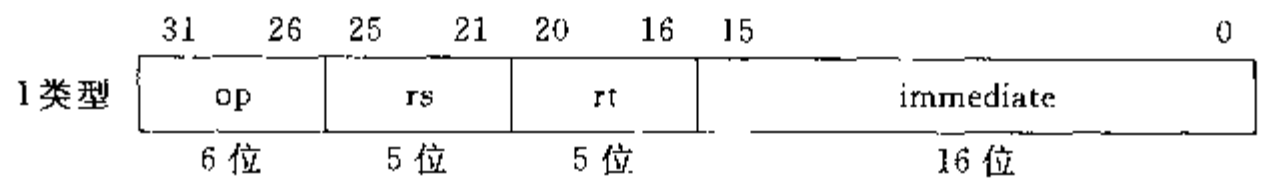
5.1 指令描述

本章中设计的 CPU 仅实现从 MIPS 指令集中挑选的有代表性的 13 条指令。这 13 条指令为 add、sub、and、or、slt、lw、sw、addi、andi、ori、bne、beq 和 j。根据 MIPS 指令的分类方法，这 13 条指令分别属于 3 类 MIPS 指令类型。

(1) 指令 add、sub、and、or、slt 属于 R 类型。格式为：



(2) 指令 lw、sw、addi、andi、ori、bne、beq 属于 I 类型。格式为：



(3) 指令 j 属于 J 类型。格式为：

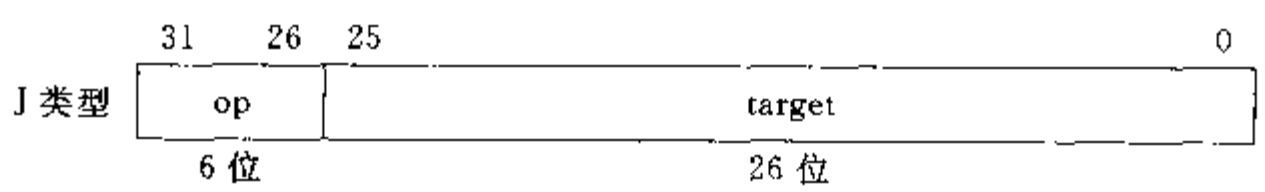


表 5.1 为这 13 条指令的格式编码。表 5.2 为这 13 条指令的详细说明。

表 5.1 13 条指令的格式

	31 26	25 21	20 16	15 11	10 6	5 0
add	000000	rs	rt	rd	00000	100000
sub	000000	rs	rt	rd	00000	100010
and	000000	rs	rt	rd	00000	100100
or	000000	rs	rt	rd	00000	100101
slt	000000	rs	rt	rd	00000	101010
lw	100011	rs	rt	offset		
sw	101011	rs	rt	offset		
addi	001000	rs	rt	immediate		
andi	001100	rs	rt	immediate		
ori	001101	rs	rt	immediate		
bne	000100	rs	rt	offset		
beq	000101	rs	rt	offset		
j	000010	target				

表 5.2 指令说明

指令	格 式	说 明
add	ADD rd,rs,rt	rd← rs + rt;当结果产生整数溢出时,将产生整数溢出例外
sub	SUB rd,rs,rt	rd←rs−rt;当结果产生整数溢出时,将产生整数溢出例外
and	AND rd,rs,rt	rd←rs AND rt
or	OR rd,rs,rt	rd←rs OR rt
slt	SLT rd,rs,rt	rd←(rs<rt)
lw	LW rt,offset(base)	rt←memory[base+offset]
sw	SW rt,offset(base)	memory[base + offset]←rt
addi	ADDI rt,rs,immediate	rt← rs + immediate;当结果产生整数溢出时,将产生整数溢出例外
andi	ANDI rt,rs,immediate	rt←rs AND immediate
ori	ORI rt,rs,immediate	rt←rs OR immediate
beq	BEQ rs,rt,offset	if rs=rt then branch;16 位有符号数 offset 向左位移 2 位,与分支延迟槽中指令的 PC 相加,得到转移目标地址

续表

指令	格　式	说　明
bne	BNE rs,rt,offset	if rs≠rt then branch;16 位有符号数 offset 向左位移 2 位,与分支延迟槽中指令的 PC 相加,得到转移目标地址
j	J target	jump:26 位目标地址 target 向左位移 2 位,并使高 2 位与分支延迟槽中指令的地址高 2 位一致,形成目标地址

从表 5.2 中可以看出以下几个问题:

(1) R 类型指令的操作码 op 为 000000,通过功能码 func 来区分不同的操作。而 I 类型和 J 类型指令的 op 各不相同,通过 op 来区分不同的指令。

(2) 对于 R 类型指令,源数据一般为寄存器 rs 和寄存器 rt,目的寄存器为 rd。而对于除了分支指令的 I 类型指令,源数据一般为寄存器 rs 和立即数 immediate,目的寄存器为 rt。所以,需要注意寄存器 rt 的用法。分支指令的源数据为寄存器 rs 和寄存器 rt。

在本章实现的指令子集中没有包含乘法、除法等一些算术指令,也没有包含浮点运算等指令。然而,通过实现上述指令,将展示如何设计一个 CPU。其他指令的实现方式和本章中介绍的指令实现相似。

5.2　设计思路

单周期 CPU 的特点是每条指令的执行需要一个时钟周期,一条指令执行完再执行下一条指令。由于每个时钟周期的时间长短都是一样的,因此在确定时钟周期的时间长度时,要考虑指令集中最复杂的指令执行时所需时间。

假设现在要设计一个简单的 CPU,使它能够执行在 2.3 节最后部分给出的程序例子。通过对这个程序中所有指令的分析,我们知道其中最复杂的指令是 lw。以 lw $4,80($1)为例,它从存储器中取来数据,放入寄存器堆的寄存器 $4 中;存储器的地址由两个数相加得到,其中的一个数是寄存器 $1 中的内容,另一个是指令中的立即数 80。它的指令格式及二进制码如下:

lw $4,80($1)　　# $4=Memory[$1+80]

6 位	5 位	5 位	16 位
op	rs	rt	immediate
35	1	4	80
100011	00001	00100	0000 0000 0101 0000

执行时需要以下 5 个步骤:

(1) 使用 PC 作为存储器地址,从存储器中把指令取来。

(2) 从寄存器堆中寄存器 $1 读出 32 位数据;把立即数 80 符号扩展成 32 位。

(3) 把上述两个数相加。

(4) 使用相加的结果作为地址,从存储器中取来数据。

(5) 把取来的数据写入寄存器 $4 中。

假设以上的每个步骤需要 1ns(1ns=10^{-9}s),则共需 5ns。由此可以确定 CPU 的时钟(clock)频率为

$$F = \frac{1}{T} = \frac{1}{5 \times 10^{-9}} = 200\text{MHz}$$

有些指令,比如 j 指令,比 lw 简单得多,并不需要 5ns。但是在单周期 CPU 中,简单指令也使用一个时钟周期。

由于 lw 指令执行的 5 个步骤均在一个时钟周期内完成,并且有两种存储器访问(取指令和取数据),我们需要有两个存储器模块:指令存储器和数据存储器。地址相加由算术逻辑单元 ALU(arithmetic logic unit)完成。ALU 是 CPU 中负责计算工作的器件,除了能够做加法之外,还要能够完成指令所需的其他操作。

由以上的分析可以知道,能够执行 lw 指令的 CPU 需要的器件大致有:

(1) 一个 PC 寄存器,用于保存程序计数器 PC 值。

(2) 一个指令存储器模块,用于存放程序(指令)。

(3) 一个寄存器堆,其中有 32 个寄存器。

(4) 一个 ALU,用于完成指令所需的操作。

(5) 一个数据存储器模块,用于存放数据。

(6) 一个控制部件,用于产生控制信号。

(7) 一个加法器,用于产生下一条顺序指令的地址(PC+4)。

把这些器件的输入和输出信号适当地连接(见图 5.1),一个能够执行 lw 指令的 CPU 的逻辑电路就设计完成了。图中的 ALUOP 是告诉 CPU 执行何种操作的控制信号,执行 lw 指令时,ALU 做加法。WRITEREG 为 1 时,在时钟的上升沿处,把从数据存储器取来的数据写入寄存器堆。PC+4 写入 PC 也在时钟的上升沿处完成。

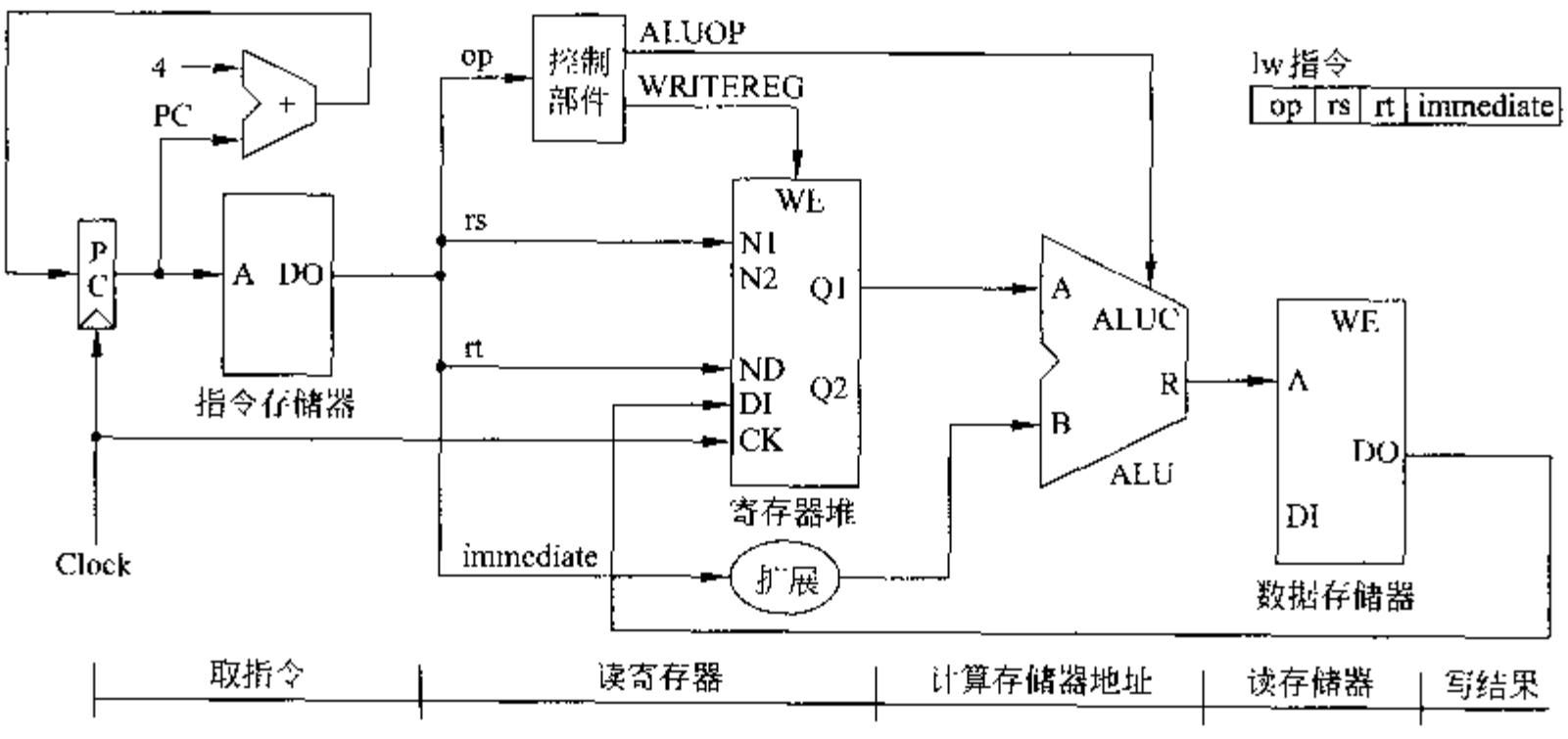


图 5.1 执行 lw 指令的 CPU 的逻辑电路

寄存器堆有两个读出端口：两个寄存器号分别从 N1 和 N2 送入，相应寄存器的内容分别从 Q1 和 Q2 送出。注意 lw 指令只需读一个寄存器数，源寄存器号是指令中的 rs。另外还有一个写端口：被写入的寄存器由 ND 指定，被写入的数据从 DI 送入。lw 指令中的 rt 是目的寄存器号。

下面考虑 add 指令。以 add $3，$3，$4 为例。它把寄存器 3 和寄存器 4 的内容相加，结果放回寄存器 3 中。指令格式及二进制码如下：

and $3，$3，$4　　# $3=$3+$4

6 位	5 位	5 位	5 位	5 位	6 位
op	rs	rt	rd	sa	funct
0	3	4	3	0	32
000000	00011	00100	00011	00000	100000

图 5.2 给出的是 CPU 执行 add 指令时所需的电路。它比 lw 电路要简单，因为 add 指令不访问数据存储器。add 指令的执行次序是：首先取指令，然后读两个源操作数，接下来由 ALU 做加法，最后把相加结果写入寄存器堆。这里有四个地方需要特别注意：第一个是两个寄存器数要同时读出，寄存器号分别是指令中的 rs 和 rt；第二个是寄存器输出端口 2 接 ALU B 输入端，而 lw 是扩展的立即数。再一个是目的寄存器号是 rd，而 lw 指令是 rt；最后一个地方是写入寄存器堆的数据来自于 ALU 的输出，而 lw 来自于存储器的输出。

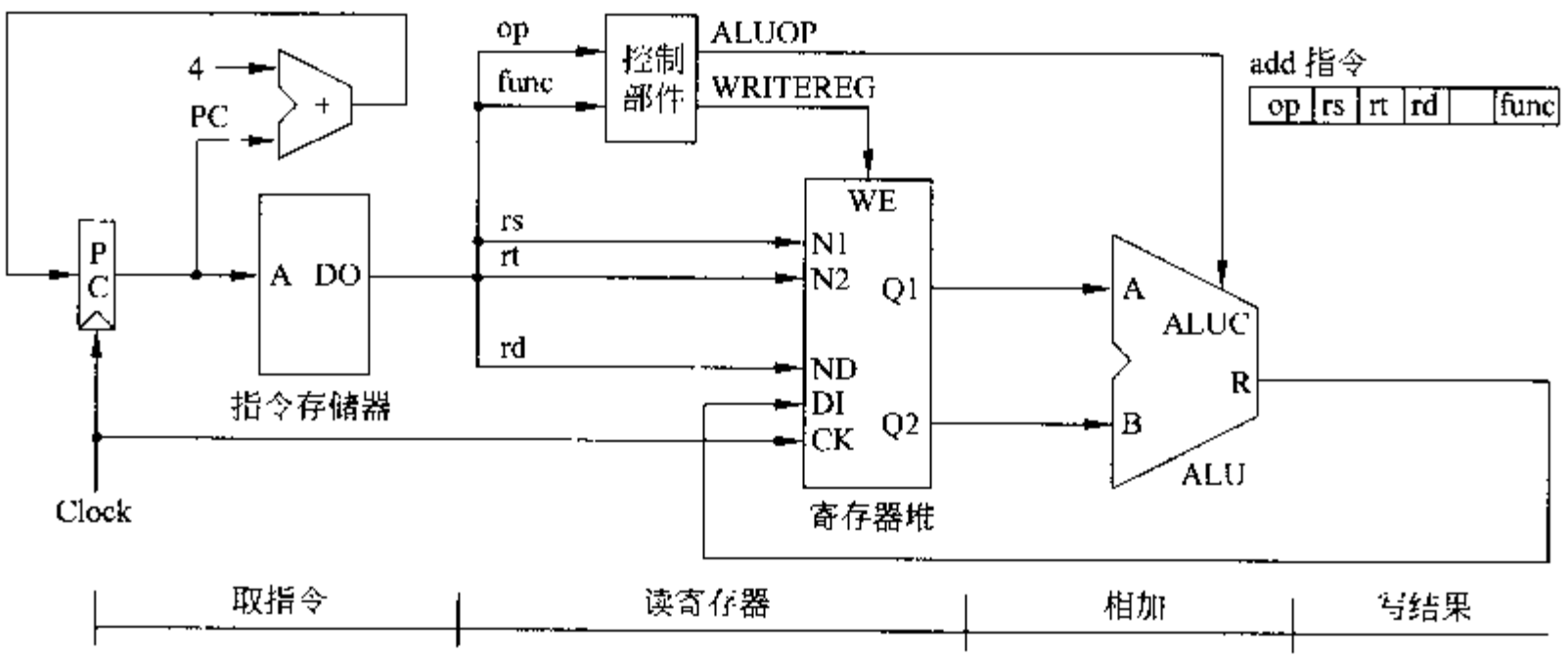


图 5.2　执行 add 指令的 CPU 的逻辑电路

在这种一个输入端有多个数据来源的情况下，可以使用一个多路选择器(multiplexer)的电路，从多个来源中选择合适的一个。图 5.3 所示的 CPU 逻辑电路能执行 lw 和 add 指令(实际上也能执行其他指令，例如 sub，and，or，addi，andi 和 ori 等，前提是 ALU 提供了这些运算功能)。图中使用了三个多路选择器。与此相对应，控制部件提供了这些多路选择器的选择信号。这些选择信号的意义如下所述：

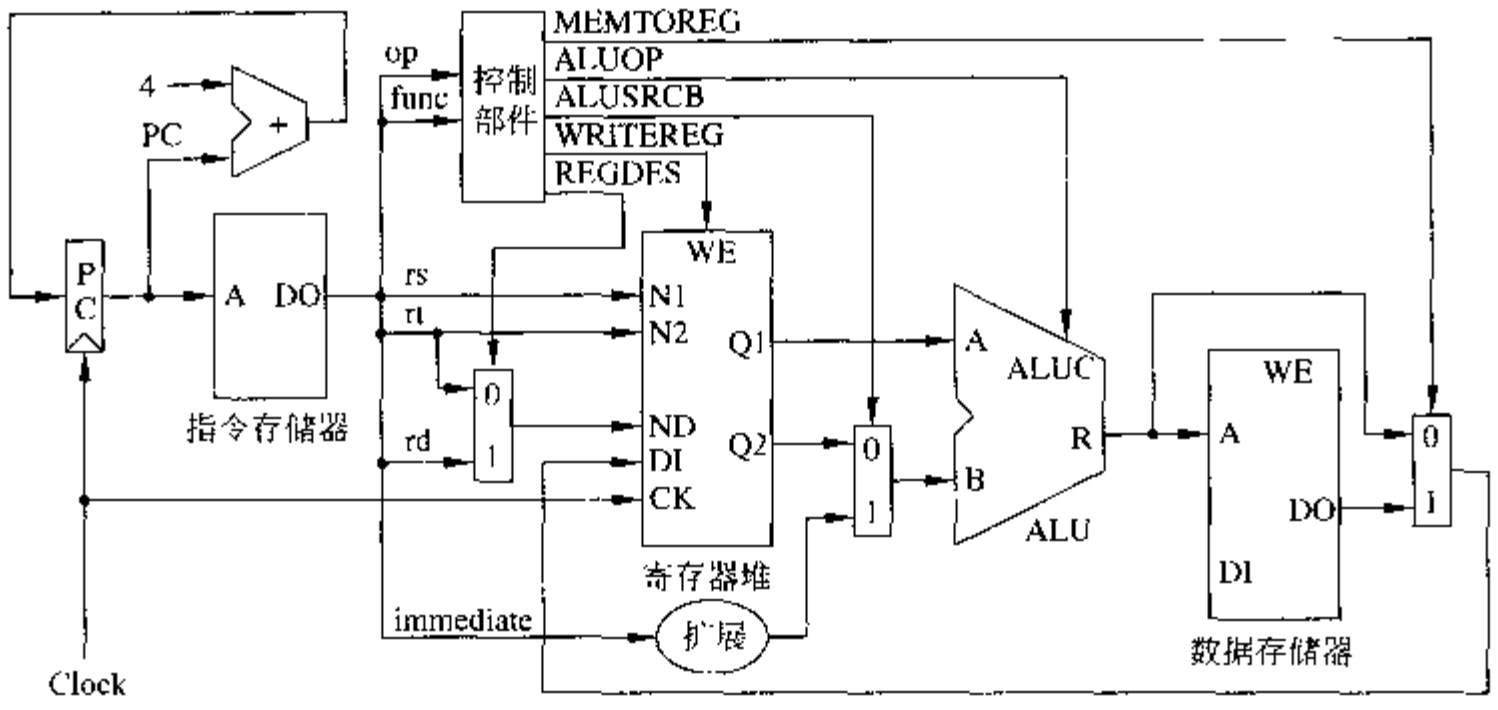


图 5.3 同时执行 lw 和 add 指令的 CPU 的逻辑电路

(1) ALUSRCB:为 1 时,选择扩展的立即数,为 0 时选择寄存器数据。

(2) MEMTOREG:为 1 时,选择存储器数据,为 0 时选择 ALU 输出的数据。

(3) REGDES:为 1 时,选择 rd,为 0 时选择 rt。

下面以 beq 为例讨论条件转移指令。其指令的格式为:beq rs,rt,label,属于 I 类型的指令。它首先比较寄存器 rs 和 rt 中的内容是否相等。若相等,则转移至 label 处,称其为目标地址。如果 PC 是这条指令的地址,则 label=PC+4+(符号扩展)immediate×4,其中 immediate 是指令中的 16 位立即数。若不等,则执行 beq 的下面一条指令。比较两个数是否相等可由 ALU 做减法来实现。若相减结果为 0,则表示两个数相等。为此,我们在设计 ALU 时,可以输出一位 z:等于 1 时表示 ALU 的输出结果为 0。因此,转移的条件是:当前指令为 beq 并且 z=1。写成逻辑表达式为:BRANCH=beq·z。如果把 bne 指令也考虑进去,则

$$\mathrm{BRANCH}=\mathrm{beq}\cdot z+\mathrm{bne}\cdot \bar{z}$$

转移目标地址的计算由一个专用加法器完成。一个多路选择器选择 PC+4 或目标地址,它的选择信号为 BRANCH。即,BRANCH 为 1 时,选择目标地址,为 0 时选择 PC+4。CPU 的这部分电路如图 5.4 所示。

综合以上的电路,并加上 sw 和 j 指令,我们可以给出能够执行三种类型指令的单周期 CPU 的电路图(图 5.5)。所有的控制信号说明如下。

(1) ALUSRCB:为 1 时,选择扩展的立即数;为 0 时选择寄存器数据。

(2) ALUOP:ALU 操作的控制码。

(3) BRANCH:为 1 时,选择转移目标地址;为 0 时选择 PC+4(图中的 NPC)。

(4) JUMP:为 1 时,选择跳转目标地址;为 0 时选择由 BRANCH 选出的地址。

(5) WRITEMEM:为 1 时写入存储器;存储器地址由 ALU 的输出决定,写入数据为寄存器 rt 的内容。

(6) MEMTOREG:为 1 时,选择存储器数据;为 0 时选择 ALU 输出的数据。

(7) REGDES:为 1 时,选择 rd;为 0 时选择 rt。

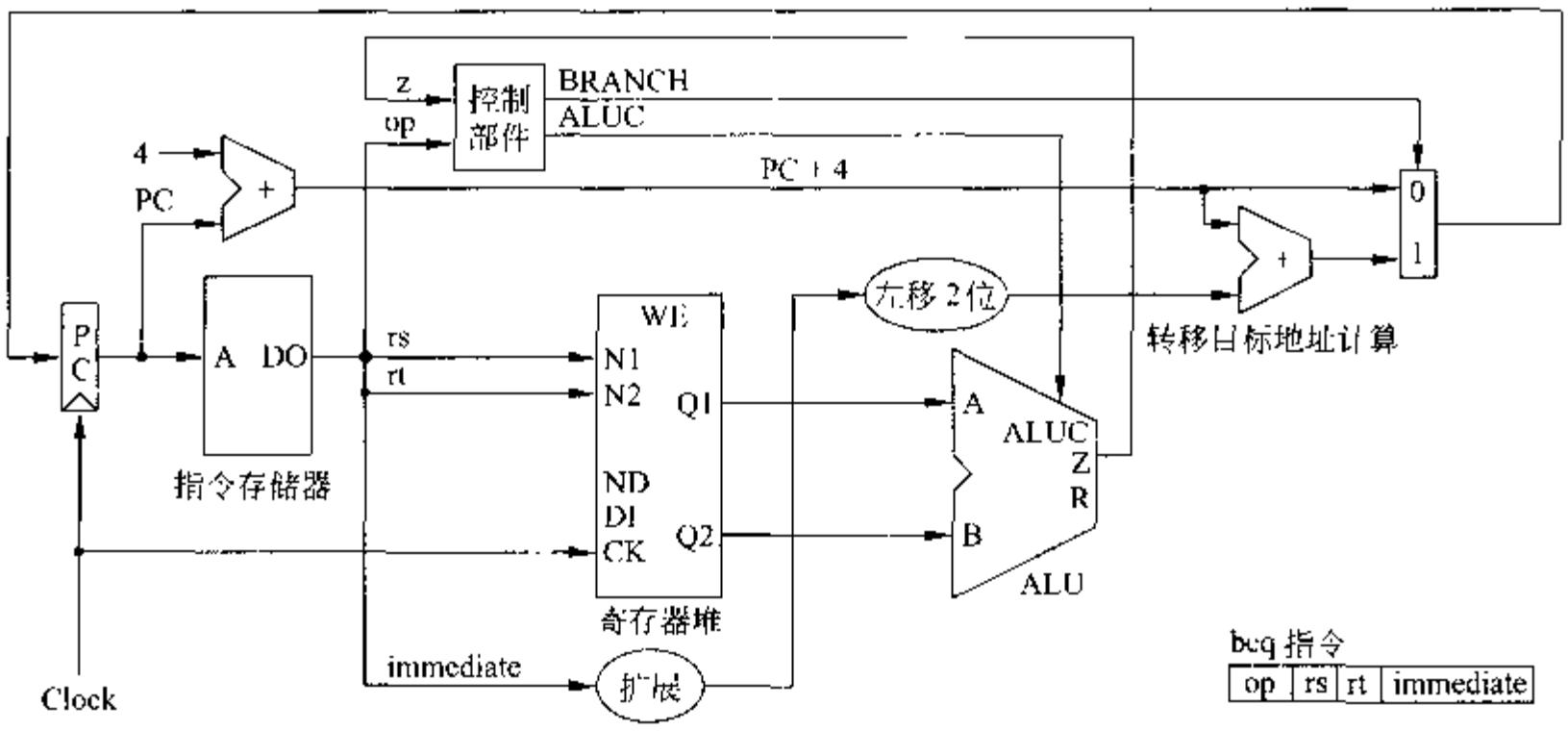


图 5.4　执行 beq 指令的 CPU 的逻辑电路

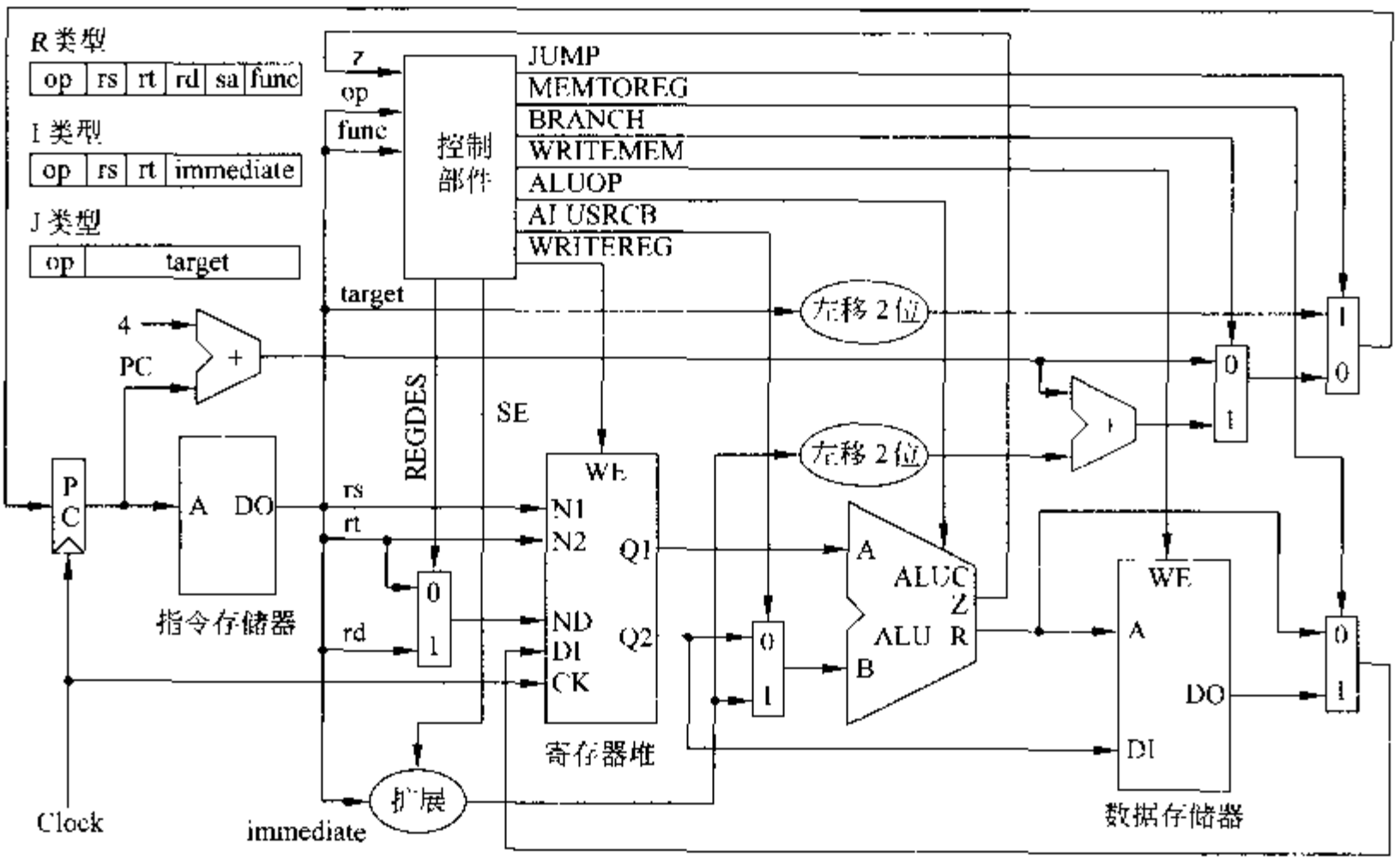


图 5.5　单周期 CPU 的逻辑电路

(8) WRITEREG：为 1 时写入寄存器堆，目的寄存器号是由 REGDES 选出的 rt 或 rd，写入数据是由 MEMTOREG 选出的存储器数据或 ALU 的输出结果。

(9) SE：符号扩展。为 1 时，符号扩展；为 0 时，0 扩展。

从上面的介绍可以看出，设计 CPU 时一个重要的工作是理顺数据在 CPU 各个部分之间的数据通道，例如，寄存器堆和 ALU 之间的数据通道，寄存器堆和存储器之间的数据通道，ALU 和存储器之间的数据通道等。下面，再根据不同类型指令的处理过程回顾

一下单周期 CPU 的设计，然后给出详细的电路设计图。

5.2.1 R 类型指令

R 类型指令使用寄存器 rs 和寄存器 rt 作为源数据寄存器，使用寄存器 rd 作为目的寄存器。该类型指令使用 ALU 对两个源数据进行某种运算之后，把 ALU 生成的结果写入到目的寄存器中。所以，在 R 类型指令处理过程中，存在如下几条数据通道，图 5.6 显示了这些数据通道的建立：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 寄存器堆中的寄存器 rt 到 ALU 的输入数据 B；

(3) ALU 的计算结果 R 到寄存器堆中的寄存器 rd。

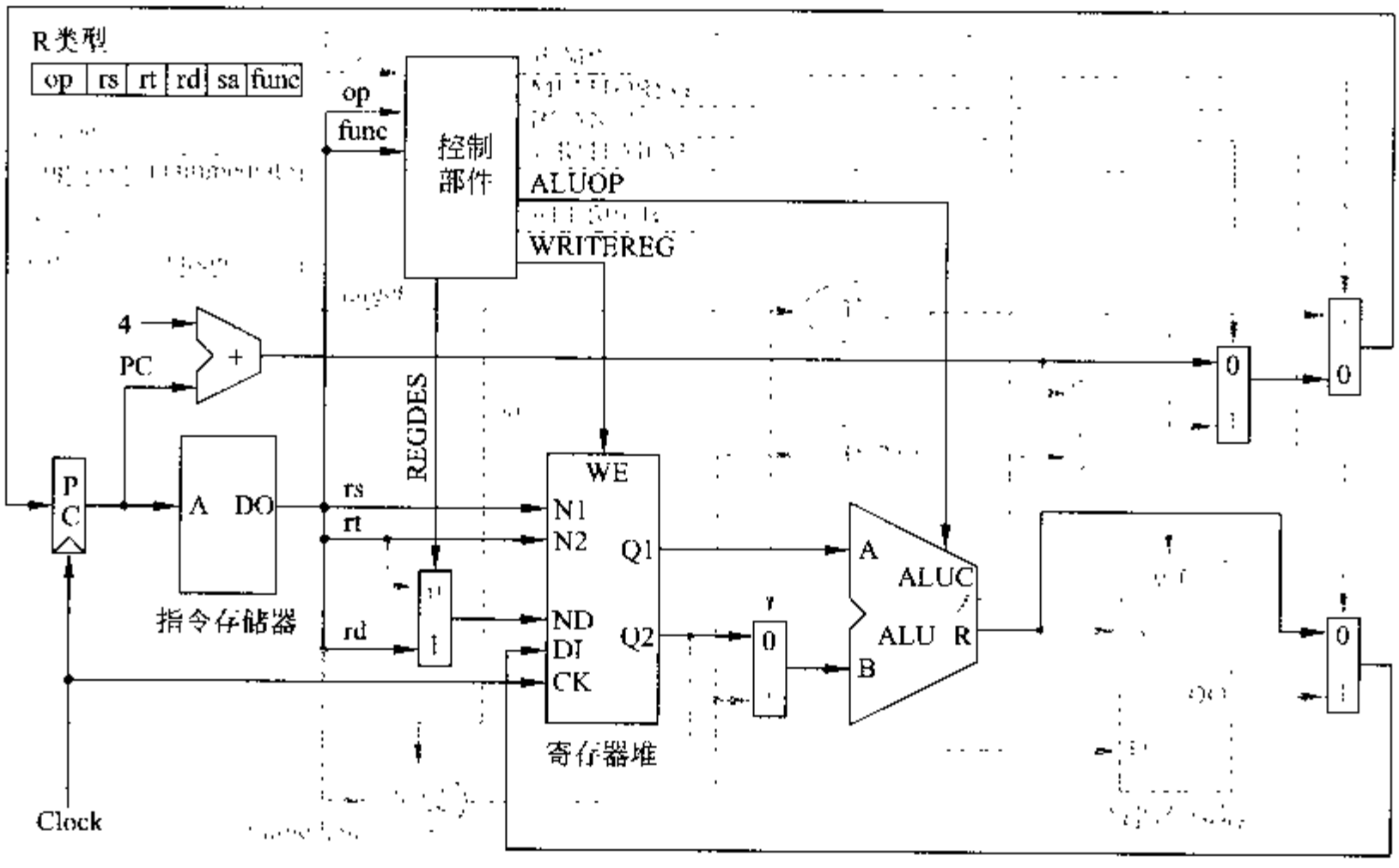


图 5.6 R 类型指令处理过程图

5.2.2 I 类型指令

I 类型指令存在着多种情况。包括运算指令、Load 指令、Store 指令、分支指令等。下面逐一介绍。

1. 运算指令

在 I 类型的运算指令中，使用寄存器 rs 和指令字中的立即数为源数据，使用寄存器 rt 作为目的寄存器。该类型指令使用 ALU 对两个源数据进行某种操作之后，把 ALU 产

生的结果保存到目的寄存器中。所以，在 I 类型的运算指令中，需要建立如下几条数据通道：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的运算结果 R 到寄存器堆中的寄存器 rt。

2. Load 指令

在 Load 指令中，使用寄存器 base(即寄存器 rs)和指令字中的立即数为源数据。使用 ALU 把这两个源数据相加，得到的结果作为输出给存储器的数据地址。然后，读入存储器按照数据地址给出的数据，把这个数据输入到目的寄存器 rt 中。所以，在 Load 指令中，需要建立如下几条数据通道，图 5.7 显示了这些数据通道的建立情况：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的运算结果 R 到通往存储器的地址总线；

(4) 来自存储器的数据总线到寄存器堆的寄存器 rt。

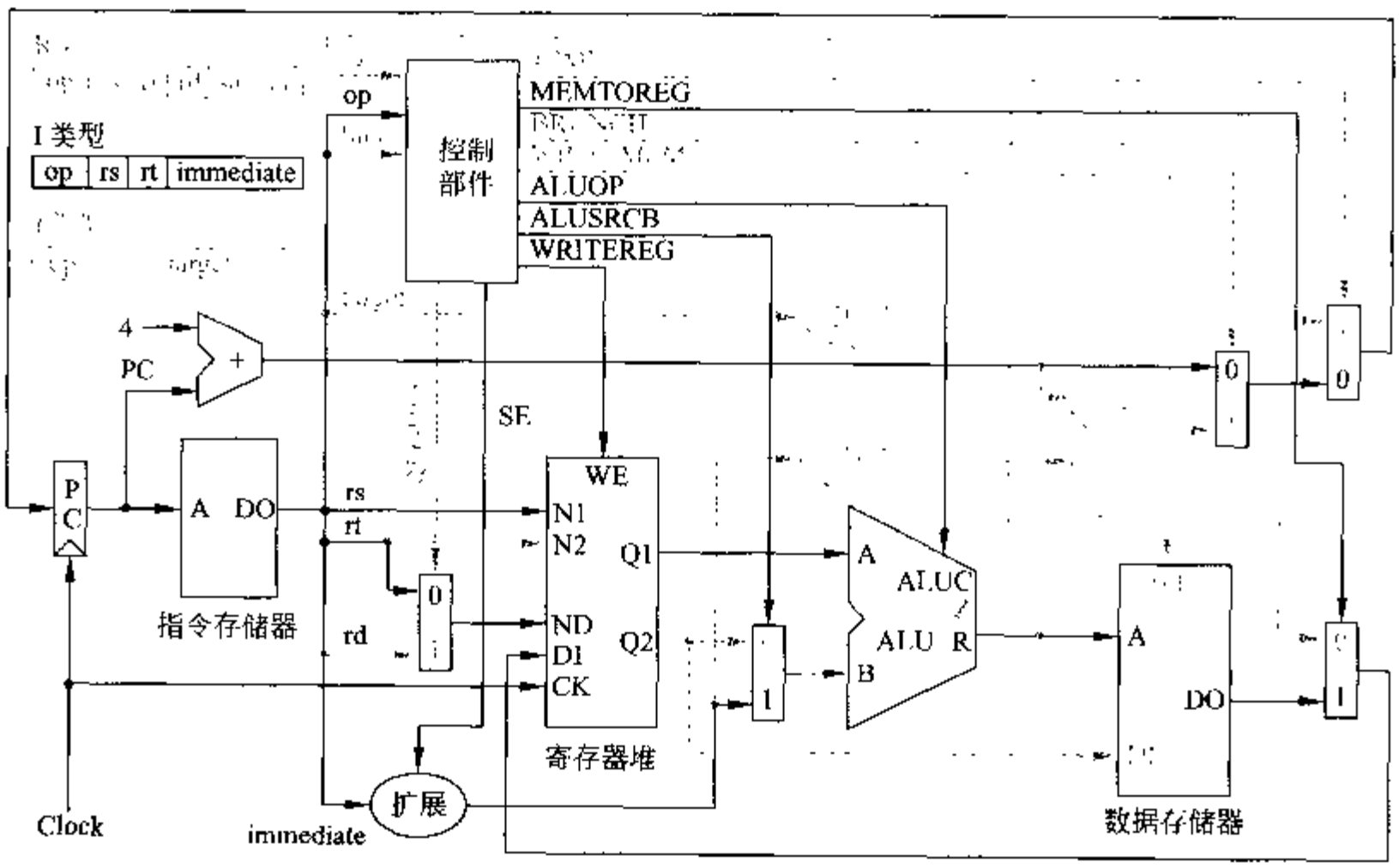


图 5.7　I 类型指令 lw 的处理过程图

3. Store 指令

在 Store 指令中，使用寄存器 base(即寄存器 rs)和指令字中的立即数为源数据。使

用 ALU 把这两个源数据相加，得到的结果作为输出给存储器的数据地址。把寄存器 rt 中的数据作为输出给存储器的写入数据。在给存储器输出写入信号之后，使存储器按照数据地址把寄存器 rt 中的数据写入。所以，在 Store 指令中，需要建立如下几条数据通道，图 5.8 显示了这些数据通道的建立情况：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的运算结果 R 到通往存储器的地址总线；

(4) 寄存器堆的寄存器 rt 到通往存储器的数据总线。

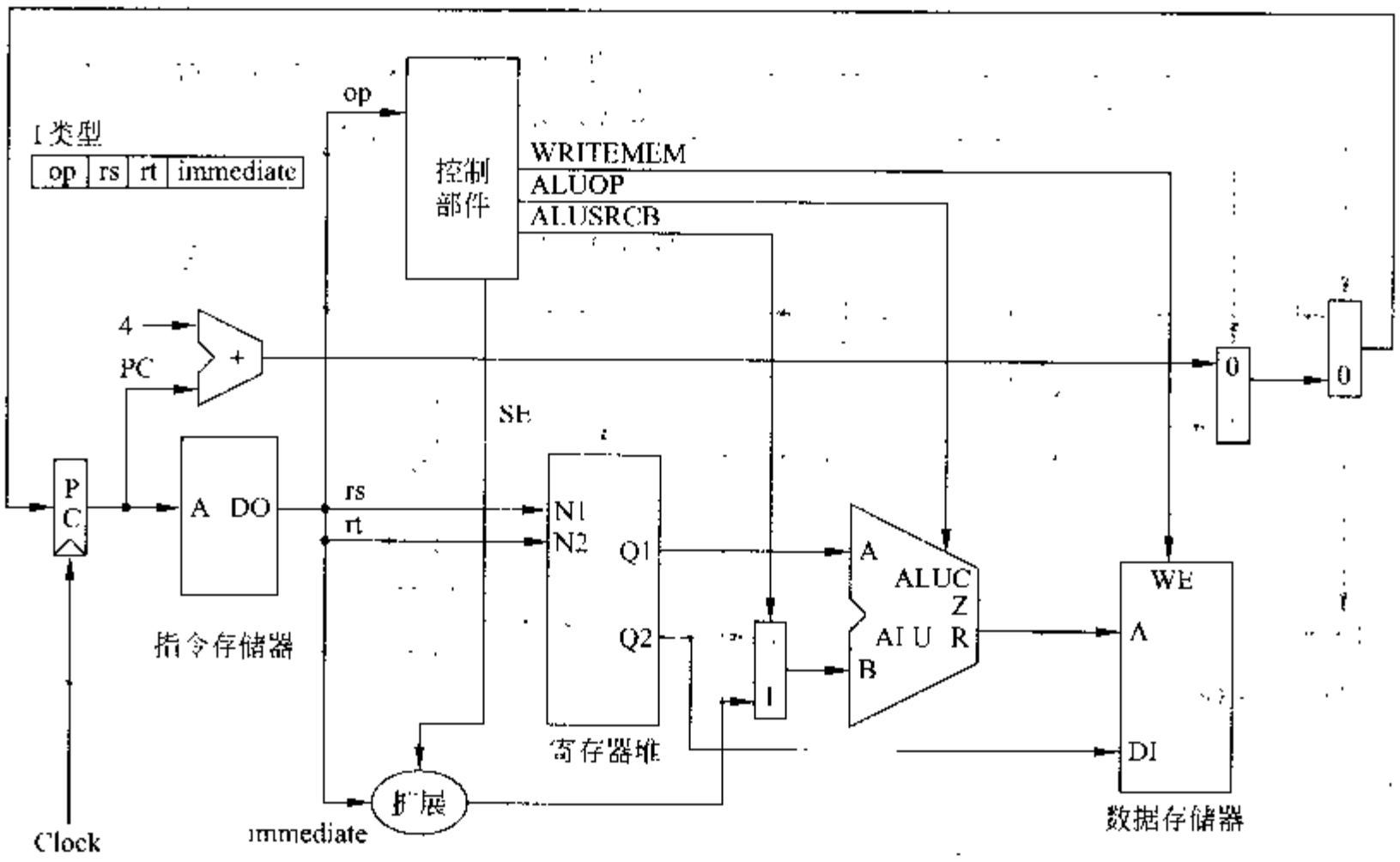


图 5.8 I 类型指令 sw 的处理过程图

4. 分支指令

分支指令 beq 和 bne 使用寄存器 rs 和寄存器 rt 作为源数据寄存器，使用 ALU 对这两个源数据进行减法操作。然后，判断结果是否为 0，把判断的结果传送给控制部件 CU。CU 通过指令类别和是否为 0 的判断结果来确定是否需要进行指令跳转，给出跳转标志。同时，分支指令需要把程序计数器 PC 和 4 相加，得到一个新的 PC 值 A，作为下一个 PC 值的候选值。然后，再把 PC 值 A 和指令字中的立即数相加，得到一个新的 PC 值 B，作为另一个下一个 PC 值的候选值。最后，根据 CU 给出的跳转标志来确定是采用 PC 值 A 还是 PC 值 B 作为下一个 PC 的值。这样，在分支指令的处理过程中，需要建立下面几条数据通道，图 5.9 显示了这些数据通道的建立情况：

（1）寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

（2）寄存器堆中的寄存器 rt 到 ALU 的输入数据 B；

（3）ALU 的输出结果 R 经判断是否为 0 后输出给控制部件 CU；

（4）程序计数器 PC 加 4 后到数据选择器的一个输入；

（5）程序计数器 PC 加 4 后再与指令字中的立即数相加，输出到数据选择器的另一个输入；

（6）控制部件 CU 给出跳转标志到数据选择器的选择端；

（7）数据选择器把选择结果输出给程序计数器 PC。

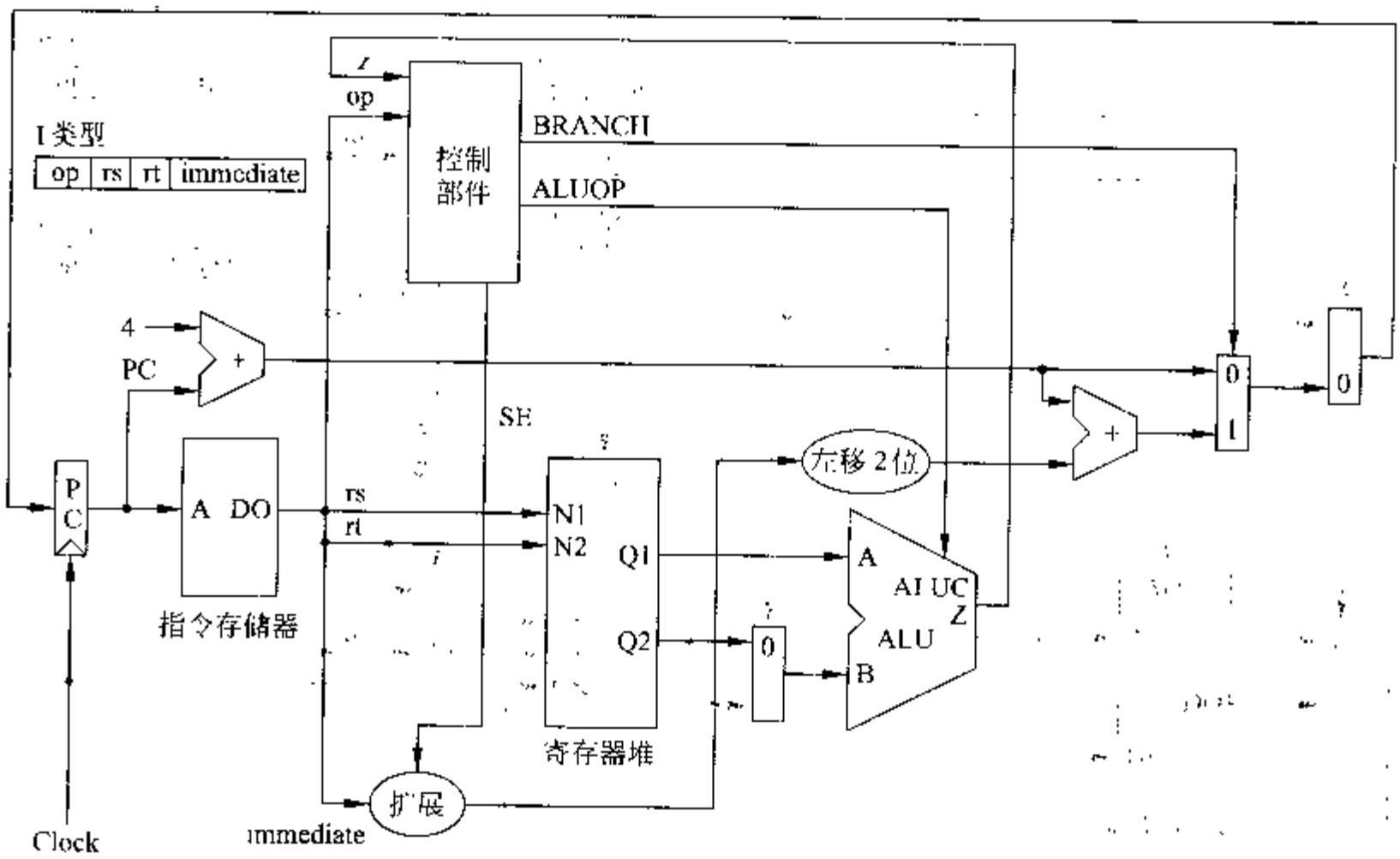


图 5.9　I 类型指令 beq 的处理过程图

5.2.3　J 类型指令

跳转指令会引起程序控制流的无条件跳转。在跳转指令的处理过程中，需要把程序计数器 PC 加 4 后的高 4 位和指令字中的跳转地址合并。然后，把合并的结果作为下一个 PC 值，输出到程序计数器 PC 中。所以 jump 指令需要建立如下的数据通道，图 5.10 显示了这些数据通道的建立情况：

（1）程序计数器 PC 加 4 后的高 4 位和指令字中的跳转地址合并；

（2）合并结果输出到程序计数器 PC。

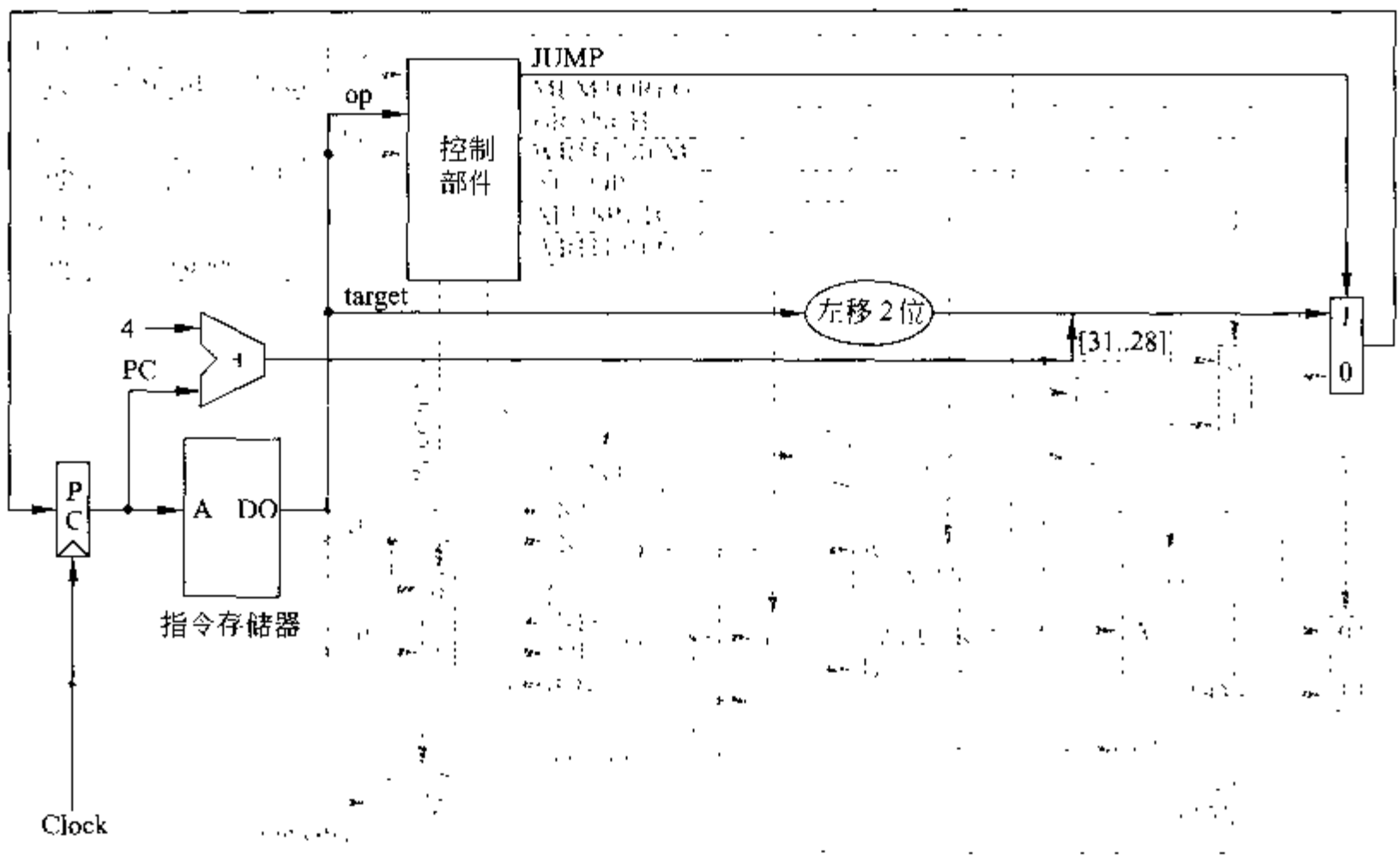


图 5.10 J类型指令处理过程图

5.3 寄存器堆设计

寄存器堆由一组寄存器组成。CPU 在运算过程中，将一些运算数据保存在寄存器中。寄存器堆中的寄存器分为两种，一种为通用寄存器 GPR，另一种为专用寄存器。在 MIPS 中，定义了 32 个 32 位的通用寄存器和 3 个专用寄存器。如图 5.11 所示。

32 个通用寄存器中有两个被赋予了特殊的功能：

(1) r0 被硬接线到了 0。当读 r0 时，将始终得到 0；当写 r0 时，将没有任何动作，没有任何影响。这样，如果某些指令需要丢弃运算结果，就可以把 r0 作为保存结果的目的寄存器。如果某些指令需要以 0 作为操作数，就可以把 r0 作为源数据寄存器。

(2) r31 为指令 JAL、BLTZAL、BLTZALL、BGEZAL、BGEZALL 在没有指定目的寄存器时的默认目的寄存器。r31 也可以被用作通用寄存器。

3 个专用寄存器为：

(1) PC，程序计数寄存器。

(2) HI，乘法指令的高位结果寄存器、除法指令的余数寄存器。

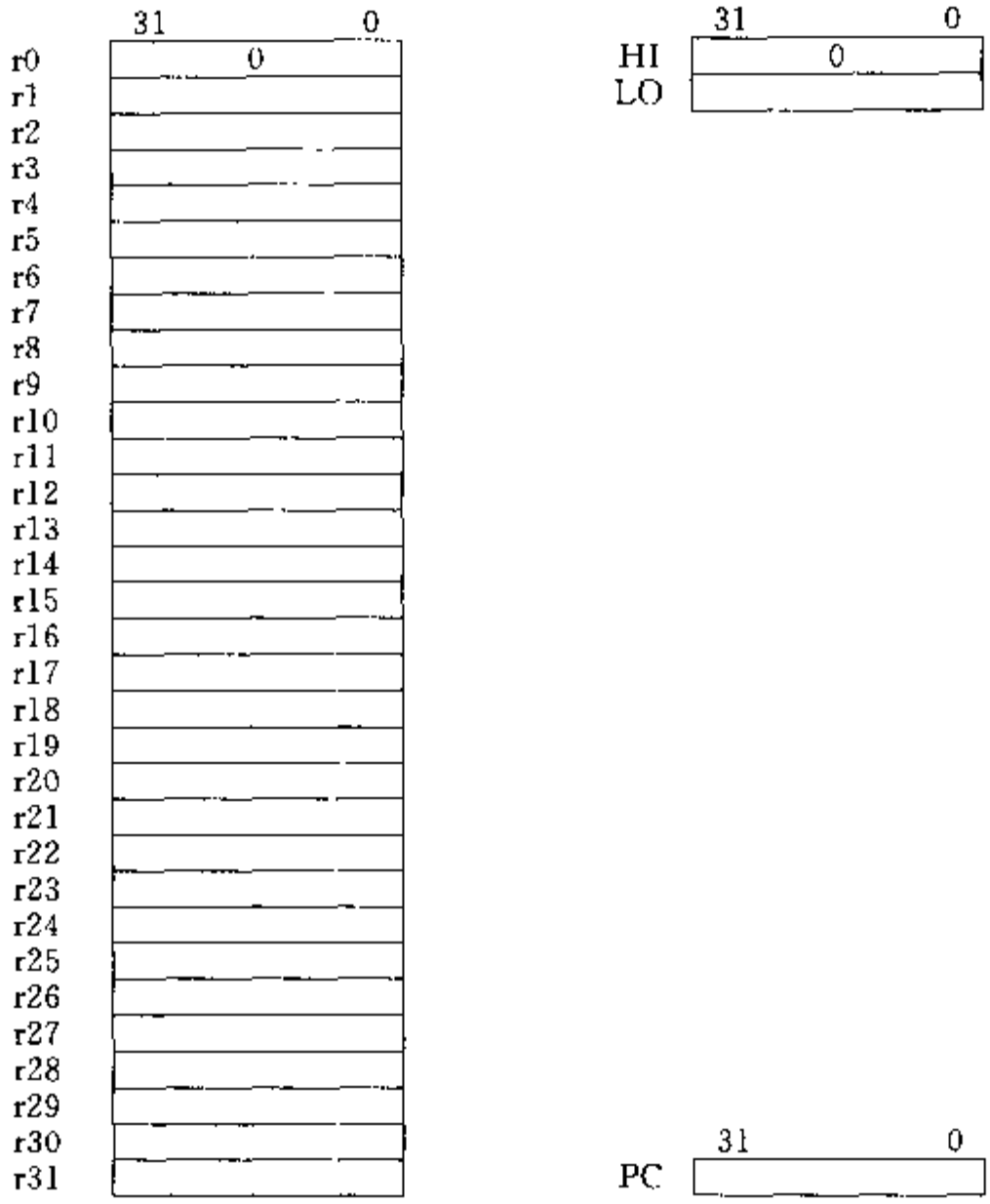


图 5.11　MIPS CPU 寄存器

(3) LO,乘法指令的低位结果寄存器、除法指令的商寄存器。

算术逻辑单元 ALU 执行各种算术逻辑运算,例如加、减、乘、除、逻辑运算、位移运算等。在本章 CPU 设计中,使用上一章设计的算术逻辑单元 ALU。控制部件 CU、寄存器堆、算术逻辑单元 ALU 三个部分协同工作,共同完成 CPU 指令集中每条指令规定的操作。

MIPS 定义了 32 个 32 位的通用寄存器和 3 个专用寄存器。由于 3 个专用寄存器的用途比较特殊,所以,不把它们放在寄存器堆中。同时,为了简单起见,这里只设计一个 8 个 32 位寄存器的寄存器堆。当然,32 个寄存器的寄存器堆的设计方法也是相同的,只是规模更大了一些。

MIPS 体系结构中 R 类型指令有 3 个操作数,其中两个操作数需要从寄存器堆中读出,作为算术逻辑单元 ALU 的输入,另一个操作数是算术逻辑单元 ALU 的输出,需要写入到寄存器堆中。这样,寄存器堆就需要 1 个写通道、2 个读通道。在寄存器堆读操作中,需要给出要读出寄存器的寄存器号,然后,寄存器堆将该寄存器号索引的寄存器中的

内容输出。在寄存器堆写操作中，需要给出要写入寄存器的寄存器号和要写入的数据，然后，寄存器堆将把数据写入到该寄存器号索引的寄存器中。

寄存器堆输入信号：

(1) N0[4..0]：读通道 0 的寄存器号；

(2) N1[4..0]：读通道 1 的寄存器号；

(3) ND[31..0]：写通道的寄存器号；

(4) DI[31..0]：写通道的输入数据；

(5) CE：写使能。

寄存器堆输出信号：

(1) RESULT0[31..0]：读通道 0 的输出结果；

(2) RESULT1[31..0]：读通道 1 的输出结果。

图 5.12 是寄存器堆整体示意图，详细的逻辑电路图如图 5.13(a)和图 5.13(b)所示。

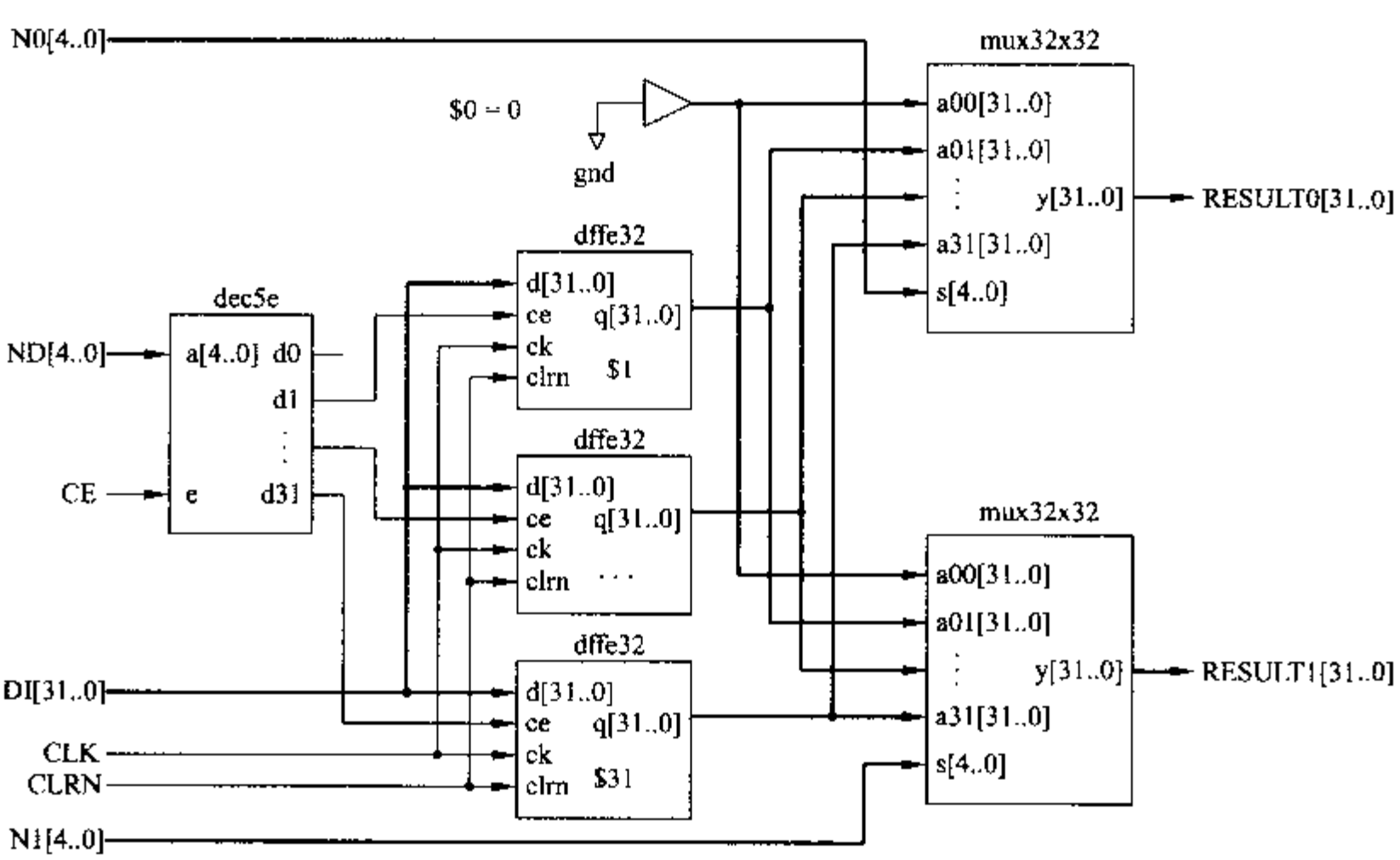
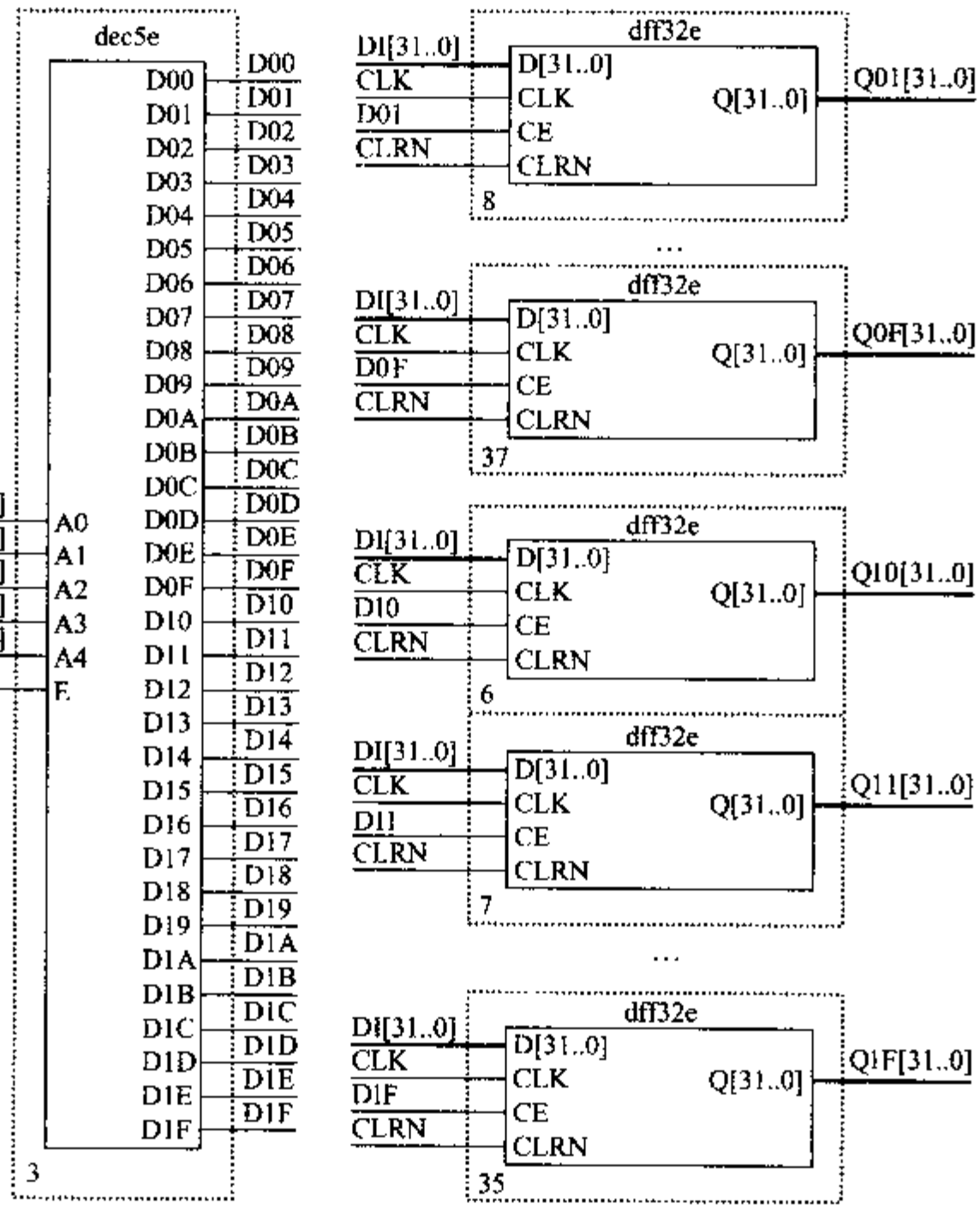
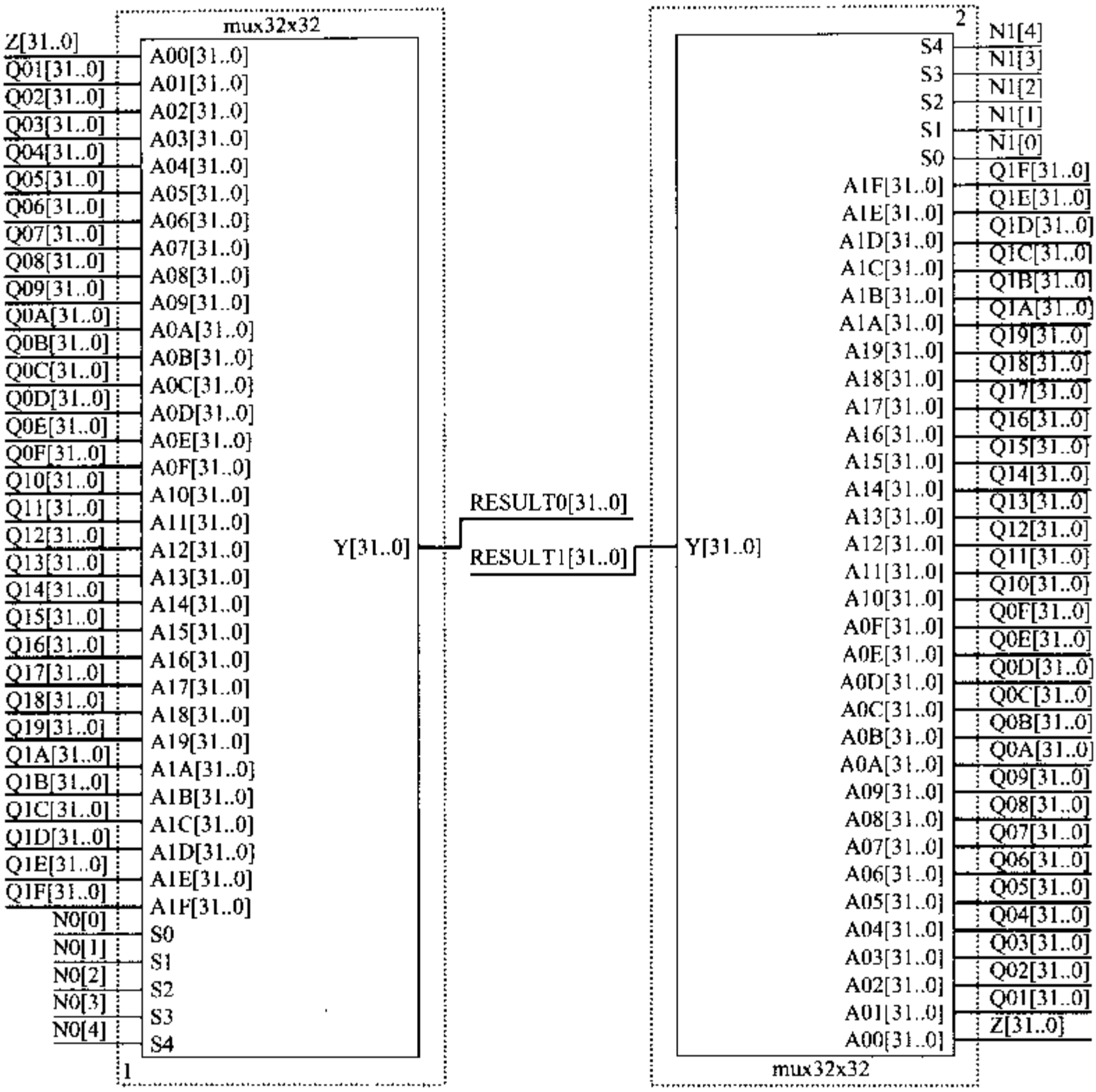


图 5.12 寄存器堆电路示意图



(a) 寄存器和写端口

图 5.13　寄存器堆逻辑电路图



(b) 寄存器读端口

图 5.13(续)

5.4 单周期 CPU 详细逻辑电路设计

一般来说，一个 CPU 在处理指令时需要经过以下几个步骤：

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，然后转到译码状态。同时，在 PC 中产生取下一条指令需要的指令地址。

(2) 指令译码(ID)：对取指令操作中得到的指令进行译码，确定这条指令需要完成的操作，从而产生相应的控制信号，驱动执行状态中的各种动作。

(3) 指令执行(EXE)：根据指令译码得到的控制信号，具体地执行指令动作，然后，转移到结果写回状态。

(4) 存储器访问(MEM):所有需要访问存储器的操作都将在这个步骤中执行,该步骤给出访问存储器的数据地址,把数据写入到存储器中数据地址所指示的位置或者从存储器中得到数据地址所指示的数据。

(5) 结果写回(WB):该步骤负责把指令执行的结果或者访问存储器中得到的数据写回到相应的目的寄存器中。

图 5.14 显示了单周期 CPU 中 5 个指令处理步骤的执行顺序。

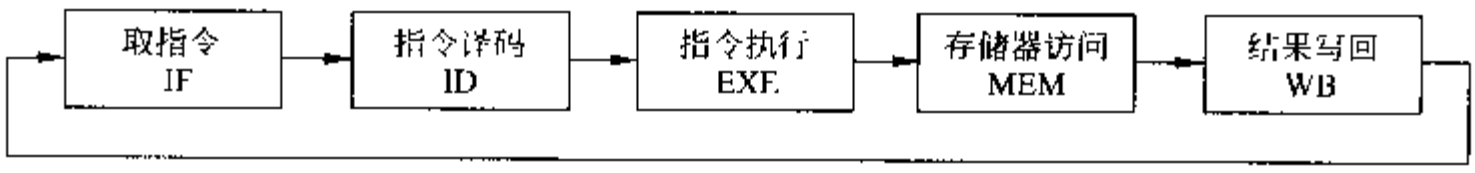


图 5.14　单周期 CPU 指令处理过程

5.4.1　取指令逻辑

CPU 执行指令时,第一步要做的就是把指令从存储器中取出来。这个动作由取指令(IF)步骤完成,如图 5.15 所示。

图 5.15　取指令操作

要为 CPU 取得需要执行的指令,首先要保证的是存在器件能够保存计算机指令。通常情况下,计算机指令保存在存储器中。除了保存指令的存储器之外,还需要有能够保存计算机指令地址的器件。在 MIPS 体系结构中,计算机指令地址保存在一个 32 位的专用寄存器中,该寄存器称为程序计数器 PC。在取出了当前指令之后,还需要为下一个周期的指令执行计算指令地址。所以,除了保存指令的存储器和保存指令地址的程序计数器 PC 之外,还需要有计算下一条指令地址的逻辑,如图 5.16 所示。

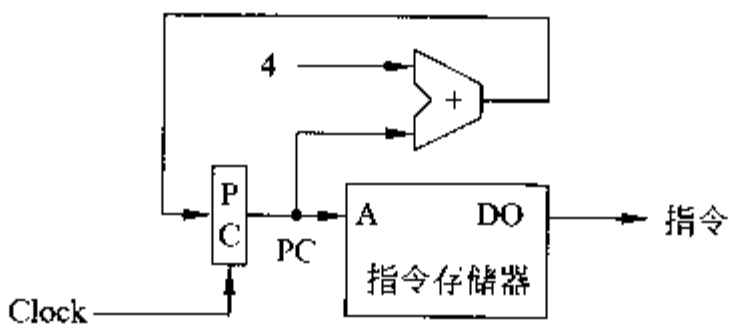


图 5.16　取指令操作逻辑图

这样,在步骤 IF 中,需要完成如下几个操作:

(1) 给出需要取出指令的存储器地址。因为指令地址保存在程序计数器 PC 中,所以,只需要把 PC 中的地址输出到地址总线上即可。

(2) 从存储器处读取指令字。

(3) 计算下一条指令的地址。

1. 指令地址

在 MIPS 体系结构中，计算机指令地址保存在一个 32 位的专用寄存器——程序计数器 PC 中。可以使用一个 32 位的寄存器来实现程序寄存器 PC，如图 5.17 所示。

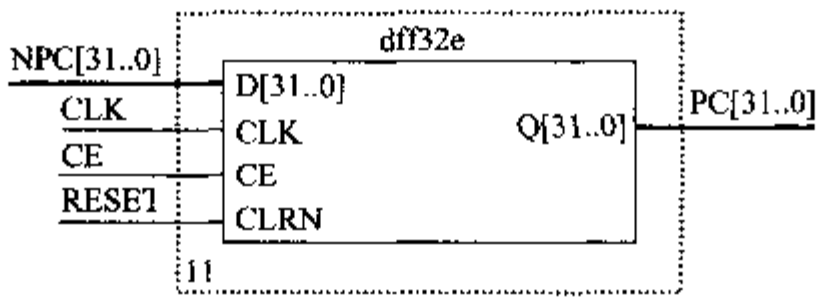


图 5.17 程序计数器 PC

2. 指令存储器

计算机指令保存在指令存储器中。在本章设计中，使用 ROM 来实现指令存储器。本章中实现的指令存储器能够保存 32 条指令字。如图 5.18 所示。指令存储器输入：A[31..0]：指令地址。实际使用的地址为 A[6..0]（注：A[6..0]是字节地址，图中使用字地址 A[6..2]）。指令存储器输出信号：Q[31..0]：指令字。

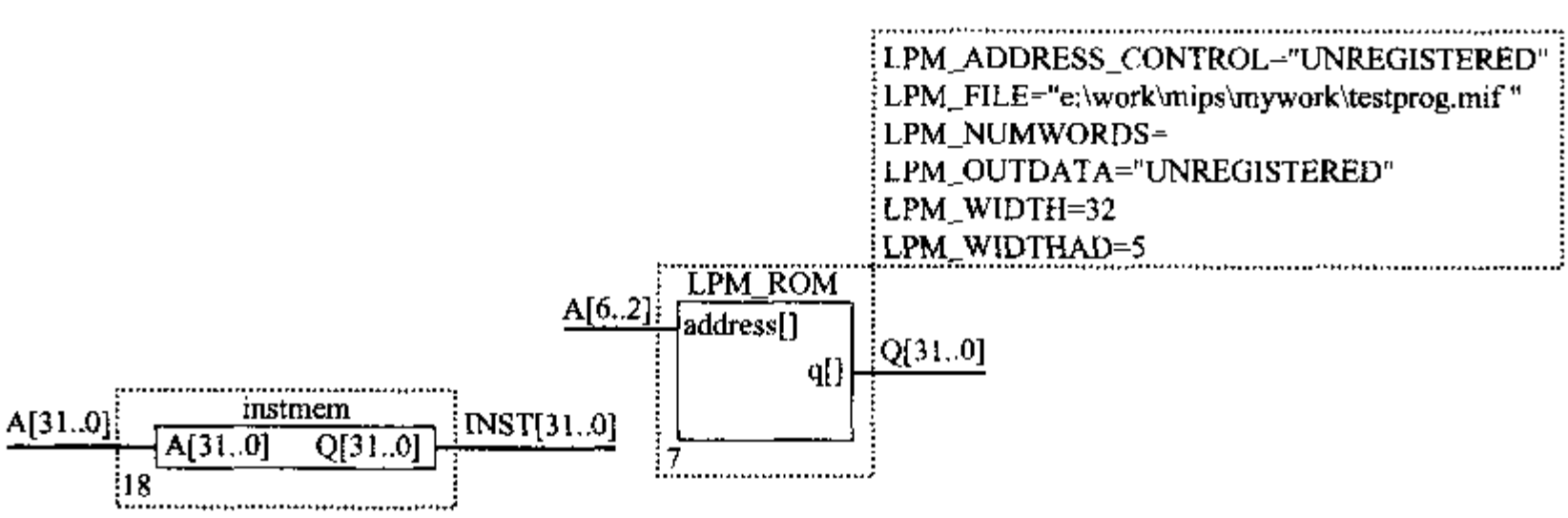


图 5.18 指令存储器及指令存储器的实现

3. 下一条指令地址

在计算下一条指令的指令地址时，有如下几种情况：

（1）如果当前指令不是分支或者跳转指令，则可以简单地在当前指令地址的基础上加 4，就得到了下一条指令的地址。

（2）当前指令为分支或者跳转指令时，则需要根据当前指令来计算跳转地址，并把产生的跳转指令作为下一条指令保存到程序计数器 PC 中。根据指令的不同，计算跳转地址的方法也不一样，计算方法见表 5.2 中有关分支和跳转指令的说明。

如表 5.2 中说明，在计算分支指令时，需要把分支指令中的 16 位立即数进行符号扩展。然后，把符号扩展后的立即数向左位移 2 位，把结果加到当前指令的指令地址上，从而产生跳转地址。在分支指令中，16 位的立即数为 INST[15..0]。该立即数的符号位为

INST[15]。这样，在对 16 位立即数 INST[15..0]进行符号扩展时，需要把 INST 位[15]扩展到高 16 位。如图 5.19 所示。

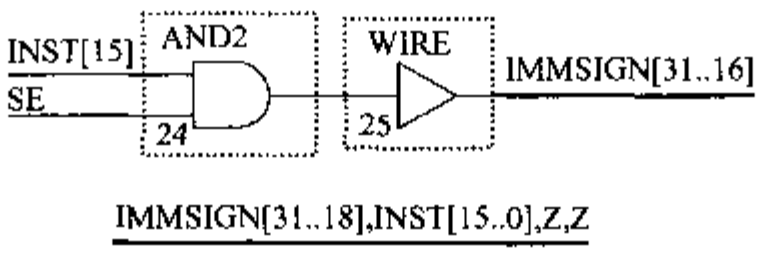


图 5.19　符号扩展

计算跳转指令地址和下一条指令地址的电路分别在图 5.20 和图 5.21 中给出。

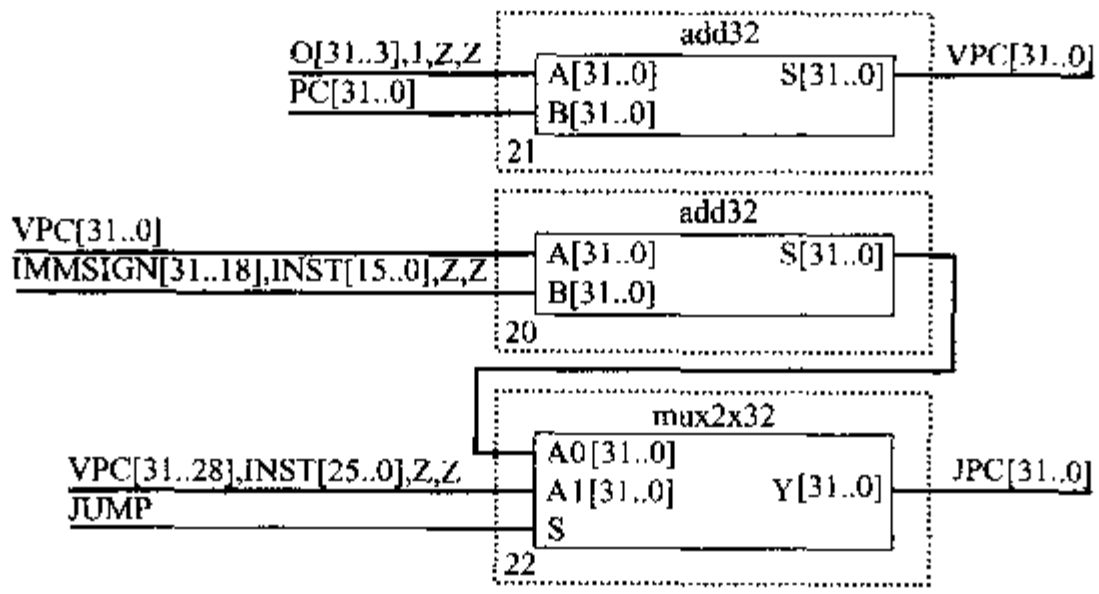


图 5.20　计算跳转指令地址

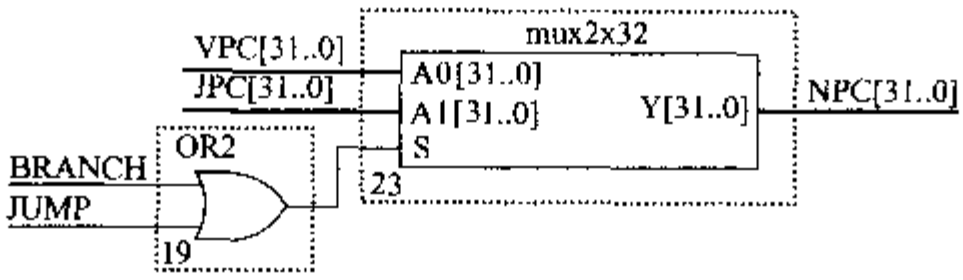


图 5.21　计算下一条指令地址

5.4.2　指令译码逻辑

在存储器得到了指令之后，将转入到指令译码阶段，如图 5.22 所示。

图 5.22　指令译码操作

指令译码操作步骤完成如下功能：

(1) 识别指令；

(2) 根据不同的指令给出各种控制信号；

(3) 根据指令从相应的源数据寄存器中取出操作数，为下一步的指令执行做好准备。指令译码动作由前面介绍的控制部件 CU 完成。

1. 识别指令

在 MIPS 指令集中，根据操作码 op 和功能码 func 来区别指令。根据表 5.2 中列出的 13 条指令编码，可以得出这 13 条指令的逻辑表达式如下，其中，OP[5..0]为操作码 op，FUNC[5..0]为功能码 func：

$$\text{SPECIAL}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\text{OP}[3]\cdot\text{OP}[2]\cdot\text{OP}[1]\cdot\text{OP}[0]$$

$$\text{add}=\text{SPECIAL}\cdot\text{FUNC}[5]\cdot\overline{\text{FUNC}[4]}\cdot\overline{\text{FUNC}[3]}\cdot\overline{\text{FUNC}[2]}\cdot\overline{\text{FUNC}[1]}\cdot\overline{\text{FUNC}[0]}$$

$$\text{sub}=\text{SPECIAL}\cdot\text{FUNC}[5]\cdot\overline{\text{FUNC}[4]}\cdot\overline{\text{FUNC}[3]}\cdot\overline{\text{FUNC}[2]}\cdot\text{FUNC}[1]\cdot\overline{\text{FUNC}[0]}$$

$$\text{and}=\text{SPECIAL}\cdot\text{FUNC}[5]\cdot\overline{\text{FUNC}[4]}\cdot\overline{\text{FUNC}[3]}\cdot\text{FUNC}[2]\cdot\overline{\text{FUNC}[1]}\cdot\overline{\text{FUNC}[0]}$$

$$\text{or}=\text{SPECIAL}\cdot\text{FUNC}[5]\cdot\overline{\text{FUNC}[4]}\cdot\overline{\text{FUNC}[3]}\cdot\text{FUNC}[2]\cdot\overline{\text{FUNC}[1]}\cdot\text{FUNC}[0]$$

$$\text{slt}=\text{SPECIAL}\cdot\text{FUNC}[5]\cdot\overline{\text{FUNC}[4]}\cdot\text{FUNC}[3]\cdot\overline{\text{FUNC}[2]}\cdot\text{FUNC}[1]\cdot\overline{\text{FUNC}[0]}$$

$$\text{lw}=\text{OP}[5]\cdot\overline{\text{OP}[4]}\cdot\overline{\text{OP}[3]}\cdot\overline{\text{OP}[2]}\cdot\text{OP}[1]\cdot\text{OP}[0]$$

$$\text{sw}=\text{OP}[5]\cdot\overline{\text{OP}[4]}\cdot\text{OP}[3]\cdot\overline{\text{OP}[2]}\cdot\text{OP}[1]\cdot\text{OP}[0]$$

$$\text{addi}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\text{OP}[3]\cdot\overline{\text{OP}[2]}\cdot\overline{\text{OP}[1]}\cdot\overline{\text{OP}[0]}$$

$$\text{andi}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\text{OP}[3]\cdot\text{OP}[2]\cdot\overline{\text{OP}[1]}\cdot\overline{\text{OP}[0]}$$

$$\text{ori}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\text{OP}[3]\cdot\text{OP}[2]\cdot\overline{\text{OP}[1]}\cdot\text{OP}[0]$$

$$\text{beq}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\overline{\text{OP}[3]}\cdot\text{OP}[2]\cdot\overline{\text{OP}[1]}\cdot\overline{\text{OP}[0]}$$

$$\text{bne}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\overline{\text{OP}[3]}\cdot\text{OP}[2]\cdot\overline{\text{OP}[1]}\cdot\text{OP}[0]$$

$$\text{j}=\overline{\text{OP}[5]}\cdot\overline{\text{OP}[4]}\cdot\overline{\text{OP}[3]}\cdot\overline{\text{OP}[2]}\cdot\text{OP}[1]\cdot\overline{\text{OP}[0]}$$

2. 产生控制信号

在本章设计的简单 CPU 中，存在以下几个控制信号。

(1) JUMP

表明该指令为跳转指令 j 引起的指令跳转。该信号影响取指令操作中下一条指令地址的产生。由于跳转指令 j 无条件地引起指令跳转，所以，信号 JUMP 的逻辑表达式为：

$$\text{JUMP}=\text{j}$$

注意，在 MIPS 的分支指令(BRANCH)和跳转指令(JUMP)中，规定在分支延迟槽中的指令执行完成后，再产生指令跳转。并且，计算跳转指令地址时，都是基于分支延迟槽中指令的指令地址。本章为了方便，不执行分支延迟槽中的指令，而是直接产生跳转。但是，在计算跳转指令地址时，仍然基于分支延迟槽中指令的指令地址。对分支延迟槽支持的实现，请参见本章 5.6 节。

(2) BRANCH

表明该指令为分支指令 bne、beq 引起的指令跳转。该信号影响取指令操作中下一条指令地址的产生。分支指令 bne 和 beq 能否引起跳转，取决于寄存器 rs 和寄存器 rt 相减的结果是否为 0。引入逻辑变量 Z 来表示相减结果是否为 0。信号 BRANCH 的真值表如表 5.3 所示。

表 5.3　BRANCH 信号真值表

Z	BRANCH
0	bne
1	beq

由真值表 5.3，可以得到信号 BRANCH 的逻辑表达式为

$$BRANCH = Z \cdot beq + \overline{Z} \cdot bne$$

(3) WRITEREG

该信号控制将 ALU 产生的计算结果或者从存储器读出的数据是否写入到目的寄存器中。从表 5.2 中可以看出，指令 add、sub、and、or、slt、lw、addi、andi、ori 都要把数据写入到目的寄存器中。所以，信号 WRITEREG 的逻辑表达式为

$$WRITEREG = add + sub + and + or + slt + lw + addi + andi + ori$$

(4) REGDES

如上所述，R 类型指令和 I 类型指令使用的目的寄存器不相同，前者为寄存器 rd，而后者为寄存器 rt。该信号指示使用哪一个寄存器作为目的寄存器。若 REGDES 为 1，则表示目的寄存器使用 rd；若 REGDES 为 0，则表示目的寄存器使用 rt。从表 5.2 中可以看出，R 类型的指令 add、sub、and、or、slt 的目的寄存器使用 rd，其他类型的指令使用寄存器 rt。所以，信号 REGDES 的逻辑表达式为

$$REGDES = add + sub + and + or + slt = SPECIAL$$

(5) WRITEMEM

该信号表示要把数据写入到存储器中。在 13 条指令中，只有指令 sw 需要把数据写入到存储器中。所以，信号 WRITEMEM 的逻辑表达式为

$$WRITEMEM = sw$$

(6) MEMTOREG

在把数据保存到寄存器中的时候，寄存器数据的来源有两种，一种是 ALU 的计算结果，另一种是从存储器或者其他地方读入的数据。该信号表明寄存器写入数据的来源。

在13条指令中，只有指令lw的目的寄存器数据来自存储器，其他指令目的寄存器的数据都来自ALU的计算结果。所以，信号MEMTOREG的逻辑表达式为

$$\text{MEMTOREG} = \text{lw}$$

(7) ALUOP[4..0]

该信号控制ALU做何种运算。具体编码见ALU设计。13条指令引起的ALU操作见表5.4。

表5.4 13条指令引起的ALU操作

指令	ALU操作	ALUOP[4..0]
add	加	×0001
sub	减	×1001
and	与	00000
or	或	01000
slt	比较	×1010
lw	加	×0001
sw	加	×0001
addi	加	×0001
andi	与	00000
ori	或	01000
beq	减	×1001
bne	减	×1001
j	无	×××××

表5.4中存在下面两个问题：

① 指令lw和sw需要使用ALU进行加法操作，是因为需要把寄存器base(即寄存器rs)和立即数offset相加得到存储器地址。

② 指令beq和bne需要使用ALU进行减法操作，是因为需要把寄存器rs和寄存器rt相减，然后检验得到的差，从而得到rs和rt的大小关系。

由表5.4，可以得到信号ALUOP[4..0]的逻辑表达式为

$$\text{ALUOP}[4] = 0$$

$$\text{ALUOP}[3] = \text{sub} + \text{or} + \text{slt} + \text{ori} + \text{beq} + \text{bne}$$

$$\text{ALUOP}[2] = 0$$

$$\text{ALUOP}[1] = \text{slt}$$

$$\text{ALUOP}[0] = \text{add} + \text{sub} + \text{lw} + \text{sw} + \text{addi} + \text{beq} + \text{bne}$$

(8) ALUSRCB

R类型指令和I类型指令使用的源操作数中，第一个操作数的来源是相同的，都是来自寄存器rs，但第二个操作数的来源却有所区别。在使用ALU进行计算的R类型指令

的第二个操作数来源于寄存器 rt，而 I 类型指令的第二个操作数则来自指令字中的立即数。信号 ALUSRB 标识了 ALU 运算的第二个操作数的来源。若 ALUSRB 为 0，则表示第二个操作数来自寄存器 rt；如 ALUSRB 为 1，则表示第二个操作数来自立即数。

第二个操作数来自立即数的指令有 addi、andi、ori、lw、sw。所以，信号 ALUSRB 的逻辑表达式为

$$ALUSRCB = addi + andi + ori + lw + sw$$

(9) SE

在 I 类型的运算指令以及 lw 和 sw 等指令中，使用算术逻辑单元 ALU 进行计算的第二个操作数来自于指令字中的立即数。通常，指令字中的立即数为 16 位。在进行计算时，需要把 16 位的立即数扩展到 32 位。在进行立即数的位数扩展时，有两种方式。一种方式是符号扩展，即把 16 位立即数的符号位填充到扩展出来的高 16 位上，形成有符号的 32 位整数。另一种方式为零扩展，即在扩展出来的高 16 位上，填充上 0，形成无符号的 32 位整数。

在本章实现的 13 条指令中，指令 addi、lw、sw 中的立即数需要符号扩展，而指令 andi、ori 中的立即数需要零扩展。同时，指令 bne 和 beq 在进行下一条指令地址计算时，也需要对指令中的立即数进行符号扩展。所以，信号 SE 的逻辑表达式为

$$SE = addi + lw + sw + beq + bne$$

3. 控制部件 CU

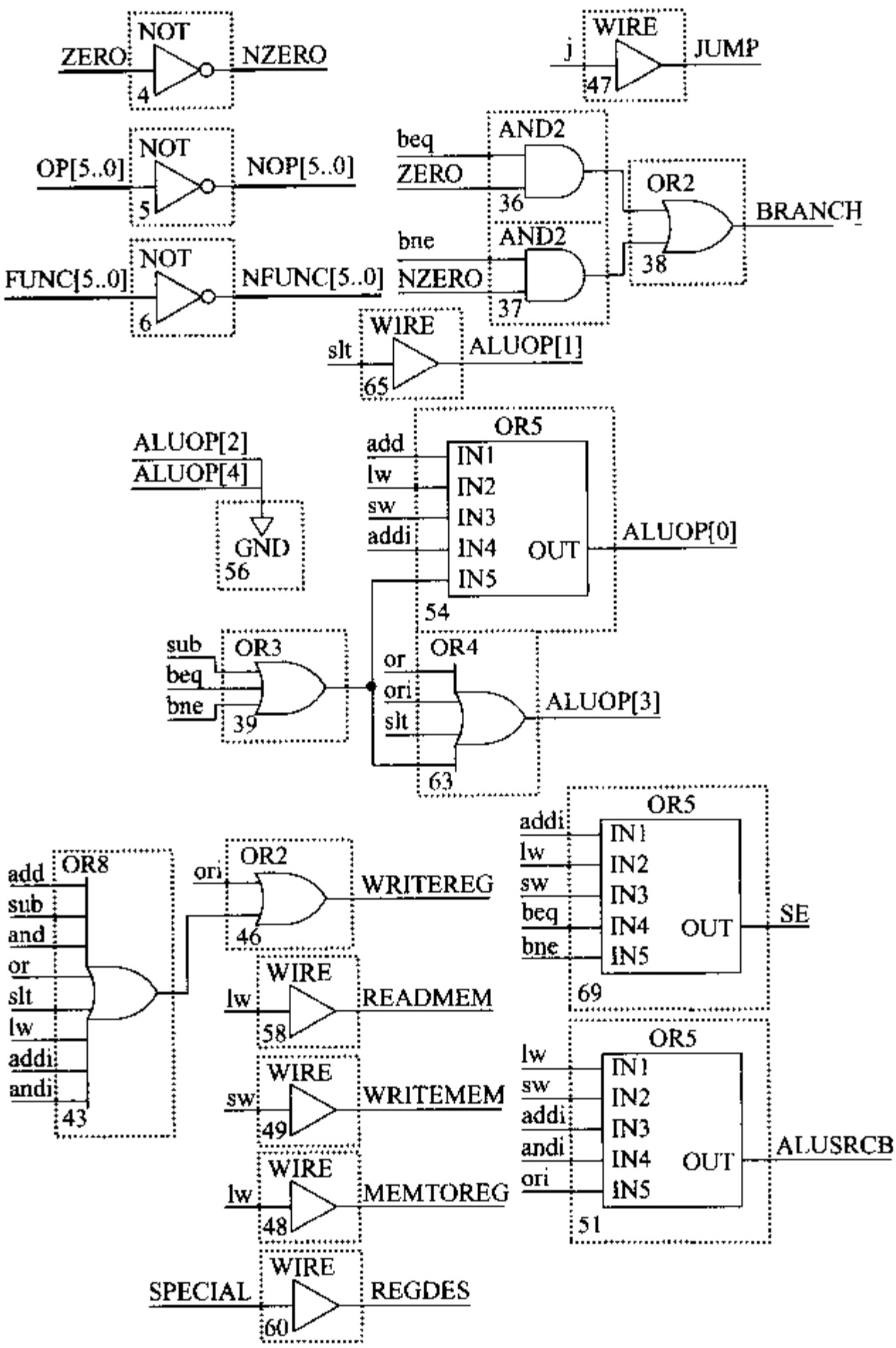
在得到了各条指令和各种控制信号的逻辑表达式之后，可以根据这些逻辑表达式得出控制部件 CU 的逻辑电路图，如图 5.23 所示。

4. 取操作数

控制部件 CU 完成了指令译码，给出了各种控制信号之后，需要从源数据寄存器中取出操作数，为下一步的指令执行准备好数据。从寄存器堆中读取的源操作数为寄存器 rs 和寄存器 rt，它们分别对应指令字 INST[31..0]中的 INST[25..21]和 INST[20..16]。本章设计的寄存器堆有 2 个读数据通道，所以，可以同时读出寄存器 rs 和寄存器 rt 中的数据。图 5.24 为取操作数步骤的逻辑电路图。

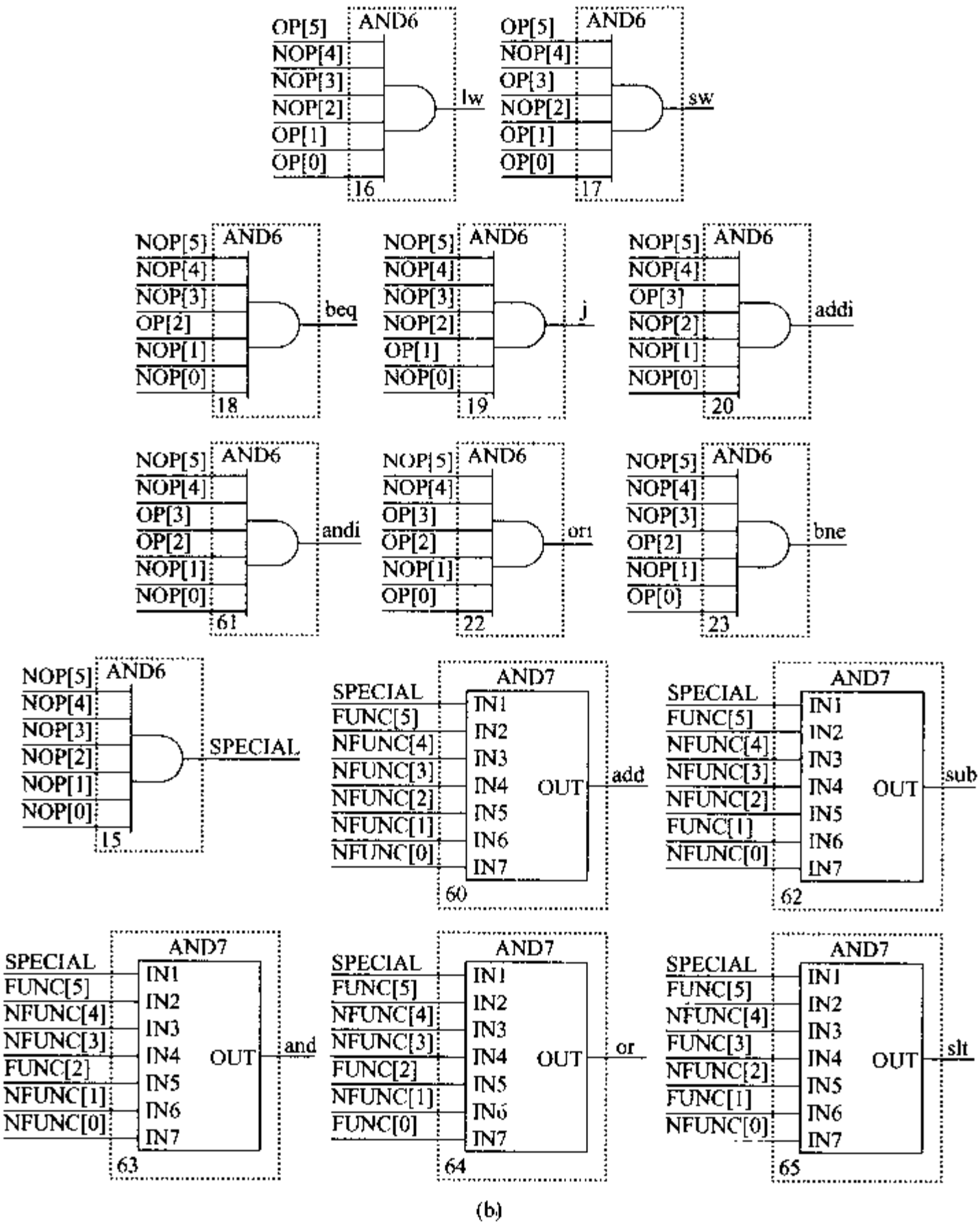
5.4.3　指令执行逻辑

经过指令译码操作，正在处理的指令需要使用算术逻辑单元 ALU 执行何种操作已经由信号 ALUOP[4..0]确定了下来。经过取操作数操作，ALU 需要使用的操作数也已经准备好了。这样，就可以进行到指令执行操作步骤，如图 5.25 所示。



(a)

图 5.23 控制部件 CU 逻辑电路图



(b)

图 5.23(续)

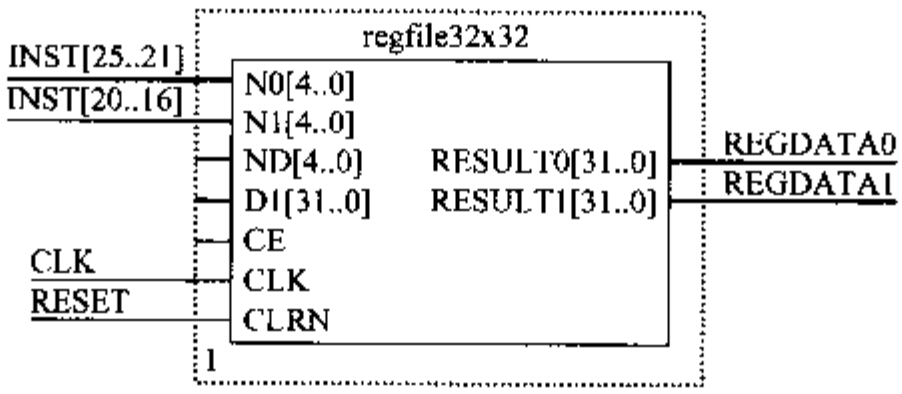


图 5.24　取操作数逻辑电路图

图 5.25 指令执行操作

ALU 的第一个操作数为寄存器 rs 中的数据。而 ALU 的第二个操作数有两个选择，一个是使用寄存器 rt 中的数据，另一个是使用指令字中的立即数。ALU 使用哪一个操作数由信号 ALUSRCB 来确定。指令执行步骤的逻辑电路图如图 5.26 所示。

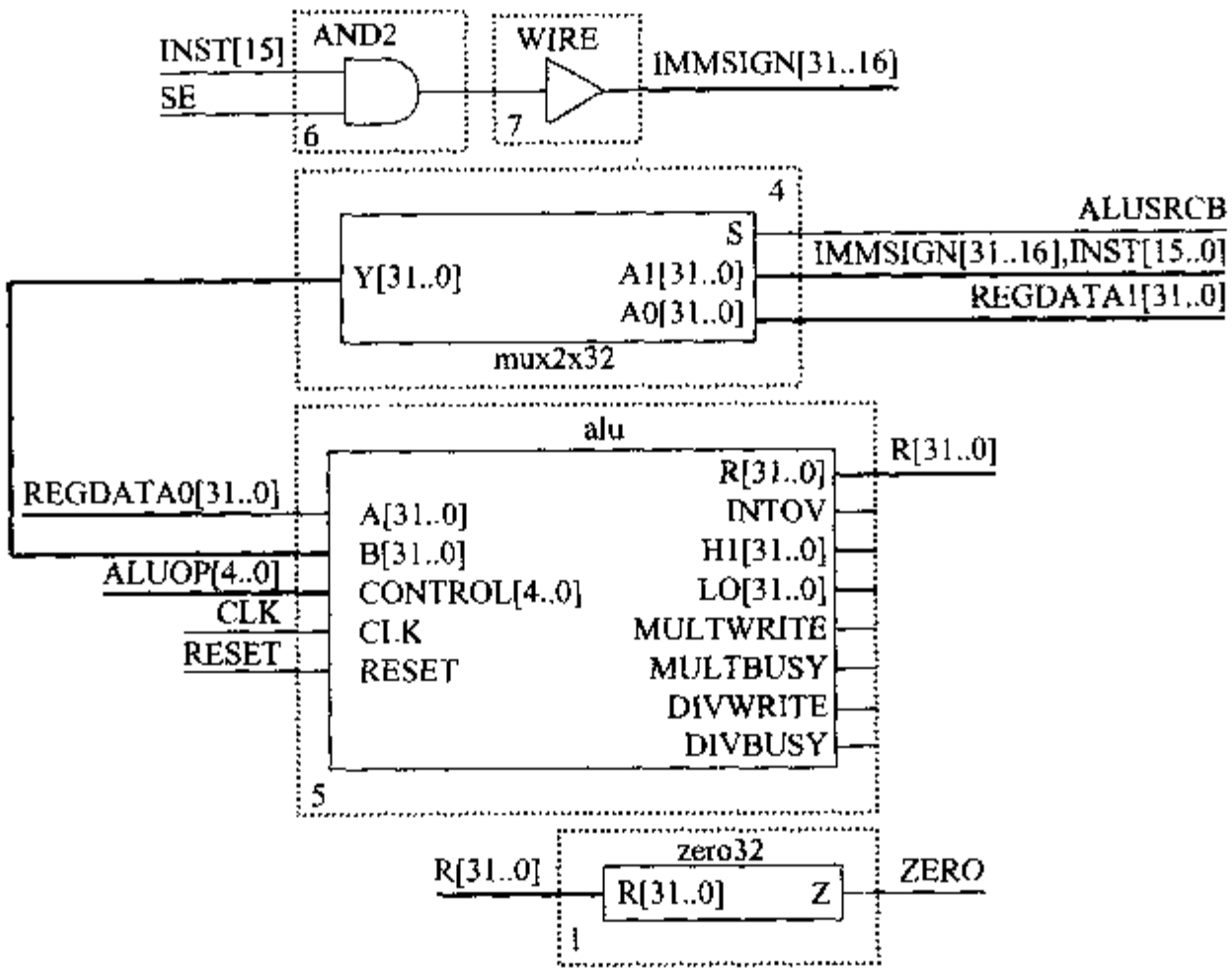


图 5.26 指令执行逻辑电路图

5.4.4 存储器访问逻辑

下面，CPU 指令处理将进入到存储器访问操作步骤，如图 5.27 所示。

图 5.27 存储器访问操作

在 MIPS 结构中定义了两种访问存储器的指令，即 Load 指令和 Store 指令。CPU 通过并且只能通过这两种指令来访问存储器。这是 RISC 结构 CPU 通用的做法。在本章设计的 CPU 中，Load 指令为 lw，Store 指令为 sw。在指令执行操作(EXE)完成之后，将要进行存储器访问操作。在访问存储器的时候，CPU 需要给出以下几种控制信号和

数据：

(1) 访问存储器的方式——读存储器还是写存储器。访问存储器的方式由信号 WRITEMEM 来确定。

(2) 访问存储器的数据地址。无论是指令 lw 还是指令 sw，都需要使用 ALU 把寄存器 rs 和指令字中的立即数相加得到数据地址。所以，ALU 的计算结果 R[31..0]即为需要访问数据的数据地址。

(3) 如果是写存储器，则还需要给出要写入的数据。写入存储器的数据为寄存器 rt 中的数据。在指令译码步骤中，寄存器 rt 从寄存器堆读出的内容为 REGDATA1[31..0]，该数据即为存储器的写入数据。

存储器访问操作的逻辑电路如图 5.28 所示。

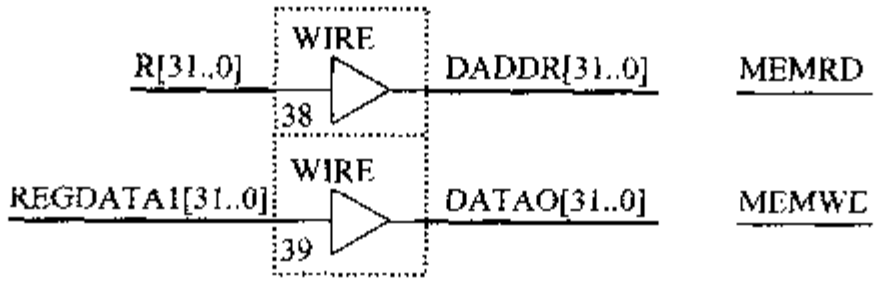


图 5.28　存储器访问逻辑电路图

5.4.5　结果写回逻辑

完成了指令执行操作和存储器访问操作后，下面要执行的将是结果写回操作，如图 5.29 所示。结果写回操作将 ALU 计算的结果或者从存储器等设备读入的数据写入寄存器堆中。

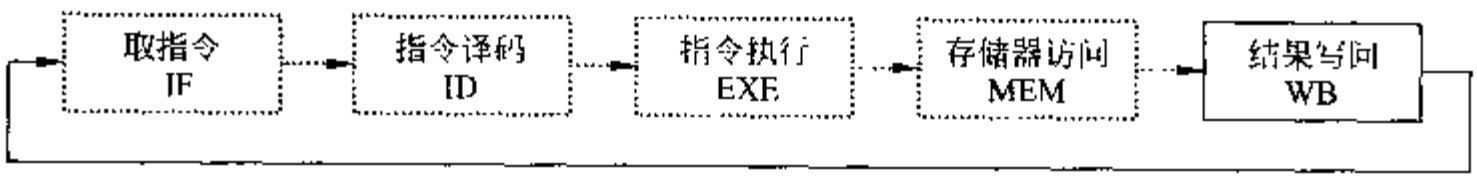


图 5.29　结果写回操作

这个过程涉及到的数据和信号如下：

(1) 要写入的数据。寄存器堆要写入的数据来源有两种，一种为 ALU 计算的结果 R[31..0]，另一种为来自存储器等其他设备的数据 DATAI[31..0]，该数据由外界输入到 CPU 中。在指令译码过程中，控制部件 CU 给出了控制信号 MEMTOREG 来区分这两种不同的数据来源。

(2) 寄存器堆写信号 WRITEREG。该信号控制寄存器堆是否执行写入操作。图 5.30为结果写回逻辑电路图。

在完成了上面 CPU 指令处理的各个步骤的设计之后，就可以把这些部分集成在一起，组成一个单周期的 CPU 了，如图 5.31 所示。当然该单周期 CPU 还比较简单，只能够简单地执行 13 条 MIPS 指令。

现在，把上面设计的 CPU 和指令存储器及数据存储器集成在一起，组成一个系统。如图 5.32 所示。

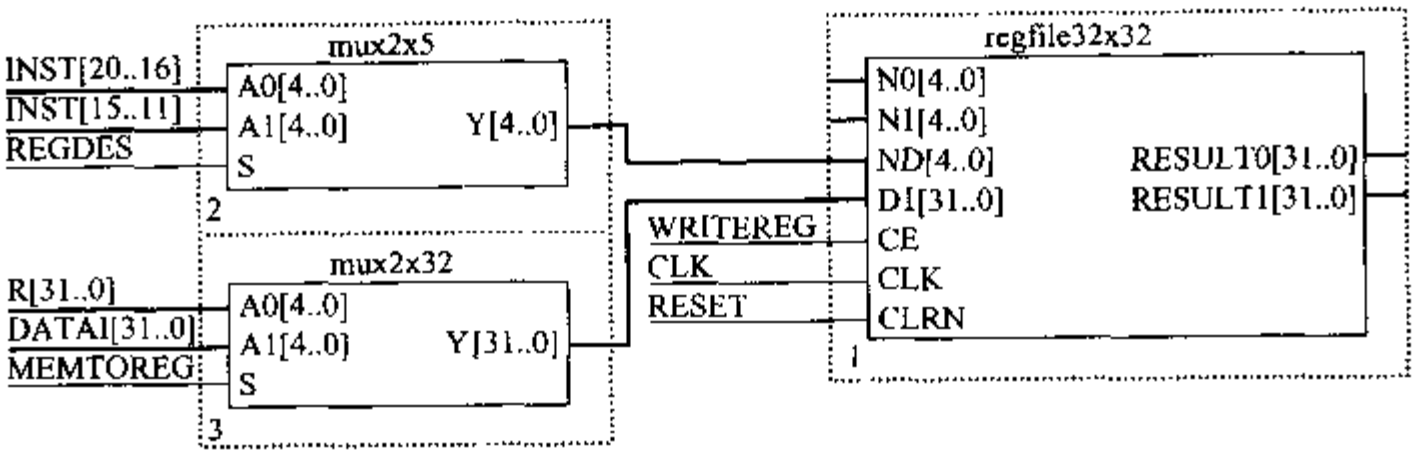


图 5.30 结果写回逻辑电路图

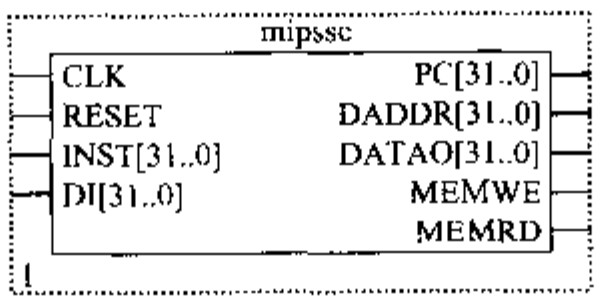


图 5.31 单周期 CPU

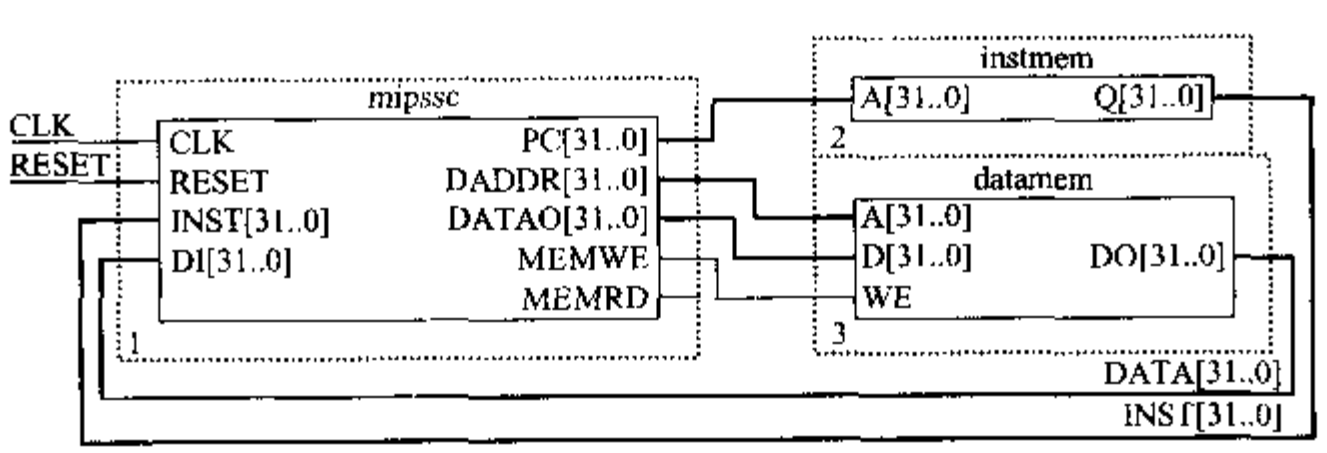


图 5.32 系统逻辑电路图

5.5 测试波形图

下面，对本章设计的 CPU 进行测试。在指令存储器中装载下面的程序。

```
DEPTH = 32;                 % Memory depth and width are required %
WIDTH = 32;                 % Enter a decimal number %
ADDRESS_RADIX = HEX;        % Address and value radixes are optional %
DATA_RADIX = HEX;           % Enter BIN, DEC, HEX, or OCT; unless %
                            % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..1F] : 0;    % Range--Every address from 0 to 1F = 00000000          %
0 : 00000820;   % 000000 00000 00000 00001 00000 100000 (00)           %
```

```
                %           add   $1, $0, $0        ; address          %
1 : 20020004;   % 001000 00000 00010 00000 00000 000100 (04)          %
                %           addi  $2, $0, 4         ; counter          %
2 : 00001820;   % 000000 00000 00000 00011 00000 100000 (08)          %
                %           add   $3, $0, $0        ; sum              %
3 : 8C240000;   % 100011 00001 00100 00000 00000 000000 (0C)          %
                % loop:    lw    $4, 0($1)         ; load data        %
4 : 20210004;   % 001000 00001 00001 00000 00000 000100 (10)          %
                %           addi  $1, $1, 4         ; address + 4      %
5 : 00641820;   % 000000 00011 00100 00011 00000 100000 (14)          %
                %           add   $3, $3, $4        ; sum              %
6 : 2042FFFF;   % 001000 00010 00010 11111 11111 111111 (18)          %
                %           addi  $2, $2, -1        ; counter - 1      %
7 : 10400001;   % 000100 00010 00000 00000 00000 000001 (1C)          %
                %           beq   $2, $0, finish    ; finish?          %
8 : 08000003;   % 000010 00000 00000 00000 00000 000011 (20)          %
                %           j     loop              ; no, goto loop    %
9 : AC230000;   % 101011 00001 00011 00000 00000 000000 (24)          %
                % finish:  sw    $3, 0($1)         ; yes, store sum   %
A : 0800000A;   % 000010 00000 00000 00000 00000 001010 (28)          %
                % here:    j     here              ; dead loop        %
END ;
```

在数据存储器中装载下面的数据。

```
DEPTH = 16;                 % Memory depth and width are required %
WIDTH = 32;                 % Enter a decimal number %
ADDRESS_RADIX = HEX;        % Address and value radixes are optional %
DATA_RADIX = HEX;           % Enter BIN, DEC, HEX, or OCT; unless %
                            % otherwise specified, radixes = HEX %
CONTENT
BEGIN
[0..0F] : 00000000;
% Range--Every address from 0 to 1F = 00000000 %
0 : 000000A3; % %
1 : 00000027; % %
2 : 00000079; % %
3 : 00000115; % %
END ;           % %
```

在经过逻辑仿真之后，可以得出如图 5.33 所示的波形图。

从图 5.33 可以看出，本章设计的 CPU 正确地完成了程序所要求的任务。

在单周期 CPU 中，每条指令都是在一个时钟周期内运行完成的。实际上，每条指令运行需要的时间并不相同。有的指令运行很快，例如 j 指令，有的指令运行却很慢，例如