南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

快速乘法器的设计

刘存 林逸典 王禹衡

年级：2022 级

专业：计算机科学与技术

2024 年 4 月 10 日

摘要

本文旨在深入探讨快速乘法器的设计原理与实现方法，并通过与传统乘法器的对比分析，凸显其性能优势。传统乘法器尽管结构简单，但基于移位相加的运算方式导致其计算速度受限，难以满足高性能计算的需求。本文在充分探讨了其效率瓶颈的基础上引入布斯编码和 Wallace 树等技术，从而提升了乘法运算的效率。布斯编码通过优化乘法运算中的部分积生成，减少了移位操作次数，从而提高了运算速度。Wallace 树则通过并行处理的方式，将部分和的累计过程加速，进一步提升了乘法运算的整体性能。

本文详细阐述了快速乘法器的设计思路与实现细节，并通过仿真验证，证明了其计算速度的显著提升，同时确保了答案的正确性。此外，本文还进行了加速比和成本分析，对比了传统乘法器与快速乘法器在性能与资源消耗方面的差异。

综上所述，本文所设计的快速乘法器在性能上具有显著优势，适用于对计算速度有较高要求的应用场景。

关键词： 快速乘法器；布斯编码；Wallace 树；性能优化；并行处理

目录

一、 引言

乘法器在运算中扮演着至关重要的角色，它是现代计算体系中的基石之一。在数字信号处理、图像处理、科学计算以及人工智能等众多领域中，乘法运算的效率和精度直接影响到整个系统的性能。因此，研究和开发高性能的乘法器具有极其重要的意义。

随着科技的飞速发展，对于运算速度和精度的要求也在不断提升。传统的乘法器虽然能够满足基本的运算需求，但在处理大规模数据或复杂算法时，其性能往往受到限制。

本文旨在探讨快速乘法器的设计与实现方法，分析其性能优势，并通过实验验证其在实际应用中的效果。

二、 普通乘法器

(一) 基本原理

普通乘法器遵循与列竖式乘法相似的原理，即从乘数的最低位开始，逐位与被乘数相乘，并将每次得到的乘积作为部分和累加起来。

在二进制乘法中，这一过程与十进制乘法具有相同的本质。如图1所示，左图展示了十进制乘法的过程，而右图则展示了二进制乘法的过程。PP0 至 PP3 分别表示在乘法运算过程中逐次得到的部分积。可以观察到，二进制乘法与十进制乘法在基本原理上是相似的，均是通过逐位相乘和累加来得到最终结果。

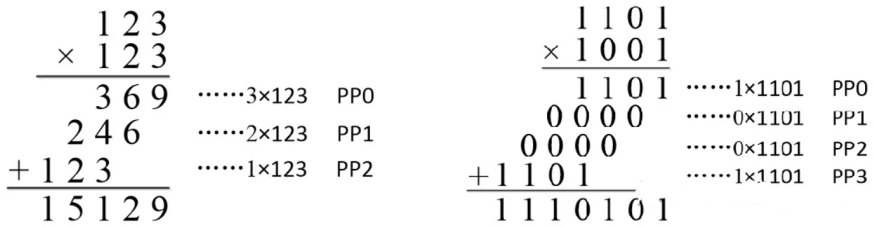

图 1: 基本乘法算法的原理图

(二) 数字电路中的乘法器实现

在数字电路中，乘法器的实现通常依赖于阵列乘法器（Array Multiplier）这一硬件结构。阵列乘法器的主要功能可以归结为三个部分：部分积的生成、部分积的累加以及最终的求和运算。

阵列乘法器的设计原理基于乘法的基本运算规则，通过特定的电路布局和逻辑控制，实现了乘法运算的高效执行。如图2所示，阵列乘法器电路图清晰地展示了其内部结构和工作原理。

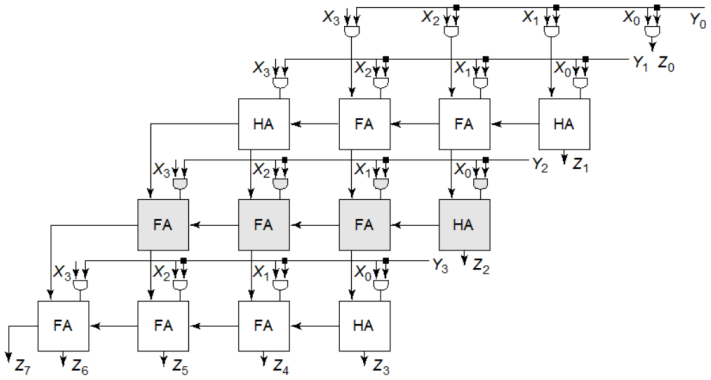

图 2: 阵列乘法器电路图

(三) 性能评估

在评估乘法器的性能时，我们通常关注其运算速度和资源消耗。假设乘数的位数为 N，被乘数的位数为 M。基于传统的乘法器实现，会产生 N 个部分积。每个部分积都是一个 M 位的数，因此需要 $N \times M$ 个二输入 AND 门以及 $N-1$ 个 M 位加法器来完成运算。

为了更精确地评估乘法器的性能，我们需要考虑其关键路径延时。关键路径延时是指在乘法运算过程中，从输入到输出所需的最长时间。根据乘法器的实现方式，关键路径延时可以表示为：$t_{\text{mult}} = [(M-1)+(N-2)]\,t_{\text{carry}} + (N-1)t_{\text{sum}} + t_{\text{and}}$ 其中，t_{carry} 表示进位传播的时间，t_{sum} 表示求和操作的时间，而 t_{and} 表示 AND 门操作的时间。这些时间参数取决于具体的硬件实现和工艺条件。通过分析关键路径延时，我们可以更准确地了解乘法器的性能瓶颈，并为后续的优化提供指导。

(四) 总结

通过上述的分析，我们可以清晰地认识到普通乘法器在运算性能上所面临的挑战。其时间瓶颈主要源自于两方面：一是与乘数位数紧密相关的移位运算和累加运算的次数，这直接影响了乘法器的整体运算速度；二是求和操作中进位链的存在，它往往成为限制乘法器性能的关键因素。

针对这些问题，我们有必要对乘法器的设计和实现进行深入研究，探索更加高效的算法和结构。通过优化乘法器的内部逻辑和运算流程，我们可以减少不必要的运算次数，缩短关键路径的延时，从而提高乘法器的运算速度和效率。

三、 快速乘法器

一个 N 位的乘法运算，需要产生 N 个部分积，并对它们进行全加处理，位宽越大，部分积个数越多，需要的加法器也越多，加法器延时也越大，那么针对乘法运算的优化，主要也就集中在两个方面：一是减少部分积的个数，二是减少加法器带来的延时 [1]。

(一) 实验原理

1. 布斯编码

布斯编码（Booth Encoding）是一种用于数字信号处理的编码方法，特别是在乘法运算中。通过布斯编码，可以减少乘法运算中的部分积的数量，从而加速乘法运算并提高计算速度。

任意有符号二进制数的补码形式都可以表示为:

$$y = -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + \ldots + 2^1y_1 + 2^0y_0 + y_{-1}$$

其中，y_i 是二进制数的第 i 位，n 是二进制数的位数，y_{-1} 是一个附加位，定义其值为 0，不影响二进制数的实际数值。

$$\begin{aligned}
y &= -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + 2^{n-3}y_{n-3} + \ldots + 2^1y_1 + 2^0y_0 + y_{-1} \\
&= -2^{n-1}y_{n-1} + 2\cdot 2^{n-2}y_{n-2} - 2^{n-2}y_{n-2} + 2\cdot 2^{n-3}y_{n-3} - 2^{n-3}y_{n-3} \\
&\quad + \ldots + 2\cdot 2^1y_1 - 2^1y_1 + 2\cdot 2^0y_0 - 2^0y_0 + y_{-1} \\
&= 2^{n-1}(-y_{n-1} + y_{n-2}) + 2^{n-2}(-y_{n-2} + y_{n-3}) + \ldots + 2(-y_1 + y_0) + (-y_0 + y_{-1})
\end{aligned}$$

由上述等价变换，观察多项式的每一项可知，相邻两位的 1 会被抵消为零，如果原二进制数中存在一连串的 1，则最低位 1 变换为-1（0 到 1 跳变位置），最高位 1 的上一位 0 变换为 1（1 到 0 跳变位置），二者中间的 1 全部变换为 0。这种变换，就是布斯变换，或称布斯编码 [3]。布斯变换可以对连续 1 的位数大于等于 3 的二进制数起到化简作用，连续 1 的位数越多，化简效果越好。当用于乘法计算时，对乘式中连续 1 较多的乘数进行布斯变换后再相乘，则会减少非 0 部分积的个数，从而对部分积的累加过程起到优化作用。

但上述变换并不能在硬件乘法器电路中起到真正的优化作用，经过变换后的二进制数与原二进制数相比，多项表达式的项数并没有变化，也就是说实际部分积个数不会减少 [4]。虽然从理论上来讲减少非 0 部分积个数就简化了累加操作，但硬件电路的结构是相对固定的，加法器个数只与部分积个数相关，与部分积的值是否全 0 无关。所以在电路设计中，一般采用改进的布斯编码方式，达到真正减少部分积个数，从而减少累加器个数的目的。

$$\begin{aligned}
y &= -2^{n-1}y_{n-1} + 2^{n-2}y_{n-2} + 2^{n-3}y_{n-3} + \ldots + 2^1y_1 + 2^0y_0 + y_{-1} \\
&= 2^{n-2}(-2\cdot y_{n-1} + y_{n-2} + y_{n-3}) + 2^{n-4}(-2\cdot y_{n-3} + y_{n-4} + y_{n-5}) + \ldots + (-2\cdot y_1 + y_0 + y_{-1})
\end{aligned}$$

通过等效变形后可以发现，多项表达式的项数变成了原来的一半。原二进制数从 LSB 开始，以三位为一组（第一组最低位需补一个附加位即 y_{-1}，附加值为 0），相邻组之间重叠一位（低位组的最高位与高位组的最低位重叠），构成新的多项式因子，这就是改进的布斯编码方式。

两个二进制数乘，如果对乘数进行改进型布斯编码，则得出的部分积个数相较直接相乘可以减半。比如，两个 32 位数相乘，不做布斯编码直接相乘，则有 32 个部分积需要累加，而采用布斯编码方式对其中一个因数进行变换后，将只有 16 个部分积需要累加 (如果兼容无符号数，则需要累加 17 个部分积，后文将具体描述)。

$$A \times B = \sum_{i=0}^{\frac{n}{2}-1} \left[2^i \cdot (-2\cdot B_{2i+1} + B_{2i} + B_{2i-1})\right]$$

由上式可知，组成多项式因子的每连续三位之间的关系是完全相同的，二进制中每一位的数值非 0 即 1，由此可以列出相邻三位所有取值组合下，其对应多项式因子的值。设某乘法运算中，被乘数为 Y，乘数为 X，X_{2i+1},X_{2i},X_{2i-1} 分别为 X 的连续三位，其中 i 为自然数 N，PP_i 为 i 不同取值下对应的部分积，对 X 进行改进的布斯变换后再与 Y 相乘，则可以有如下推算关系:

X_{2i+1}	X_{2i}	X_{2i-1}	PP_i
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	2Y
1	0	0	-2Y
1	0	1	-Y
1	1	0	-Y
1	1	1	0

表 1: 基 4 布斯编码查找表

由此可见，根据乘数每连续三位的值，可以快速推算出其对应的部分积 [2]。且在二进制中，乘 2 操作可以通过左移一位来实现，不需要复杂的计算，电路实现非常简单。作为改进型布斯编码中最基础的一种，它被称之为基 4 布斯编码。基 4 编码相当于每次用乘数的两位与被乘数相乘产生部分积，从而使部分积个数减少一半，也可以看成是将乘数转化为 4 进制表达，故称为基 4 布斯编码 (Radix-4 Booth Encoding)。

我们对乘法公式进行逻辑代数化简，利用卡诺图等方法，最终得到乘法表达式如下：

$PP_{ij} = (b_{2i} \oplus b_{2i-1})(b_{2i+1} \oplus a_j) + \overline{(b_{2i} \oplus b_{2i-1})}(b_{2i} \oplus b_{2i+1})(b_{2i+1} \oplus a_{j-1})$

因此，我们作出对应的电路图为：

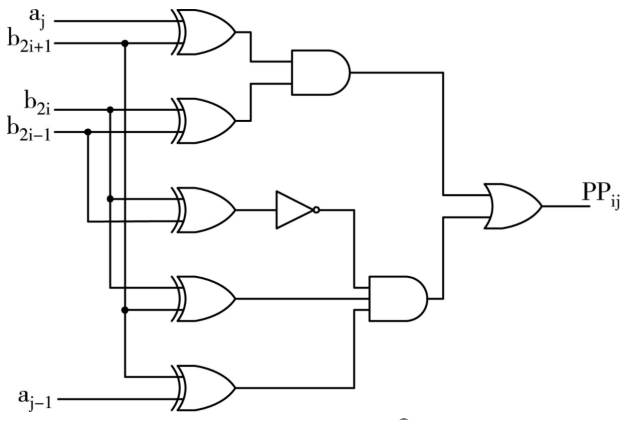

图 3: 基 4 布斯编码电路

2. Wallace 树

Wallace 树是一种高效快速的加法树结构，即许多个加数求和，每 3 个加数分为一组，压缩至 2 个加数，循环往复。

我们以有符号 16 位数乘法为例介绍 Wallace 树，当处理 16 位有符号数时，我们需要考虑符号位（通常是最高位）以及数值部分。有符号数乘法通常通过补码形式实现，这样可以将减法转化为加法，从而简化硬件设计。

下面简单介绍下 Wallace 树的压缩构建过程：

- 部分积生成：首先，根据乘法原理，将 16 位被乘数与乘数的每一位相乘，生成一系列的部分积。

- Wallace 树构建：接下来，开始构建 Wallace 树。在树的每一层，将相邻的部分积进行相加。由于加法可能产生进位，因此需要仔细处理这些进位，以确保正确的结果。Wallace 树的每一层都减少了部分积的数量。
- 压缩过程：随着 Wallace 树的深入，部分积的数量逐渐减少，位宽逐渐增加。这个过程实际上是对部分积进行压缩，以便后续的处理更加高效。在树的底层，通常只剩下少数几个较宽的部分积。
- 最终加法：最后，将 Wallace 树底层剩余的部分积进行加法运算，得到最终的乘积结果。

Wallace 压缩模块，其核心组成部分为压缩器。压缩器依据其功能差异被细分为不同型号，如 3-2 压缩器和 4-2 压缩器等。在此，我们以 4-2 压缩器为例，深入剖析其结构特点，以揭示其工作原理。

4-2 压缩器由两个 CSA（Carry Save Adder，进位保存加法器）组合而成，通过精心设计的结构，实现了高效的压缩功能。如图4所示，该压缩器巧妙地将输入信号进行处理，并通过内部逻辑运算，最终输出压缩后的结果。

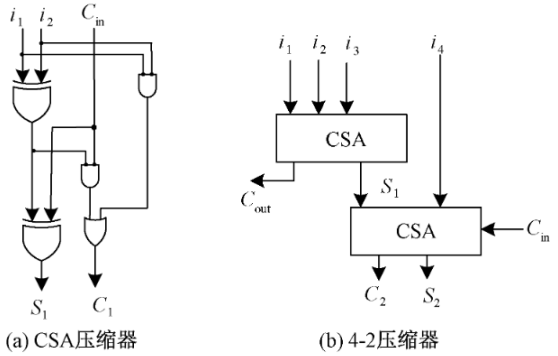

图 4: 传统的 CSA 和 4-2 压缩器

4-2 压缩器有 4 个数据输入端和 1 个进位输入端，数据输入端接 4 个部分积相应的 bit 位，每个 bit 对应的进位输入端接其前 1 位的 co 输出，第 0 位的进位输入端接零值，最高位的 co 输出如果在最终积的有效范围内（未超出 32 位），一定要加进来不可忽略。无法加入到 4-2 压缩器输入的部分积保留到下一级，直到可以参与运算为止。

CSA 使用了 2 个异或门和 3 个与门, 最长门传输延时为 2 级异或门。4-2 压缩器的工作原理是:i_1、i_2、i_3 进入第一级 CSA, 得到 C_{out} 和 S_1, 然后将 S_1 和 i_4 和 C_{in} 送入第二级 CSA, 得到 C_2 和 S_2。这种 4-2 压缩器使用门的数量是 CSA 结构的两倍, 最长门传输延时为 4 级异或门。CSA 结构的 C_1 和 S_1 的逻辑表达式:

$$\begin{cases} C_1 = i_1 i_2 + C_{in}(i_1 \oplus i_2) \\ S_1 = i_1 \oplus i_2 \oplus C_{in} \end{cases}$$

由 CSA 级联得到的 4-2 压缩器的 C_{out}、C_2 和 S_2 的逻辑表达式:

$$\begin{cases} C_{out} = i_1 i_2 + i_3(i_1 \oplus i_2) \\ S_1 = i_1 \oplus i_2 \oplus i_3 \\ C_2 = i_4 S_1 + C_{in}(i_4 \oplus S_1) \\ S_2 = i_4 \oplus C_{in} \oplus S_1 \end{cases}$$

以 16 位有符号数乘法为例，我们研究一个 Wallace 树的压缩结构:

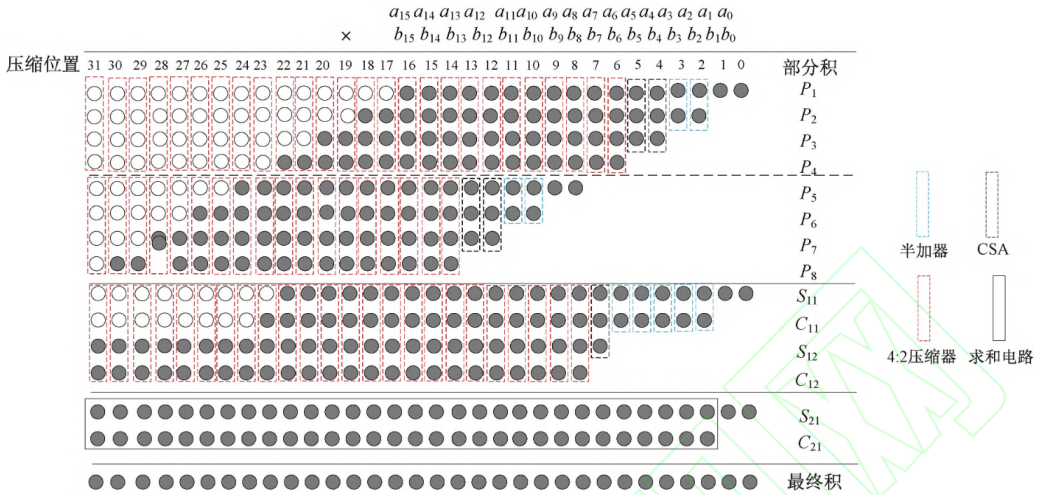

图 5: 4-2 压缩 Wallace 树

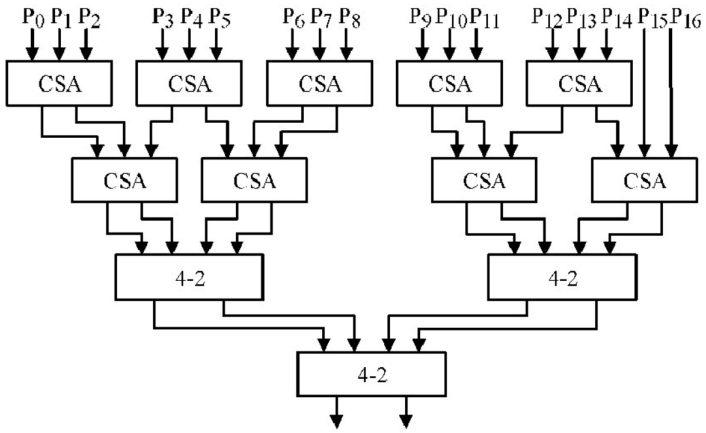

图 6: 4-2 压缩 Wallace 树电路逻辑图

(二) 仿真验证

在本次研究中，为了验证布斯编码与 Wallace 树在有符号 32 位二进制数快速乘法器设计中的有效性，我们设计了一个具体的乘法器示例，并通过编程模拟其运行过程，以进行详尽的仿真验证。

1. 3-2 压缩器

在本次实验中，我们设计了一种 64 位的 3-2 压缩器，多个压缩器并行运行，以将多组相互独立的 3 个数压缩成 2 个数。每一个 64 位的 3-2 压缩器均由 64 个一位全加器构成，与一个普通的 64 位加法器相同。

Algorithm 1 一位全加器:add1

1: **function** ADD1$(a, b, cin, cout, so)$
2: $cout \leftarrow (a + b + cin)/2$
3: $so \leftarrow (a + b + cin) \mod 2$

4: **end function**

Algorithm 2 3-2 压缩器：merge

1: **function** MERGE($num1[64], num2[64], num3[64], cout[64], so[64]$)
2: **for** $i \leftarrow 0$ to 63 **do**
3: $ADD1(num1[i], num2[i], num3[i], cout[i+1], so[i])$
4: **注意**：64 位的计算过程相互独立，并行执行，cout 的最低位置为 0，最高位予以舍弃。
5: **end for**
6: **end function**

2. 布斯编码

根据布斯编码的原理，计算 16 个 64 位部分积, 作为后续 Wallace 树的输入。

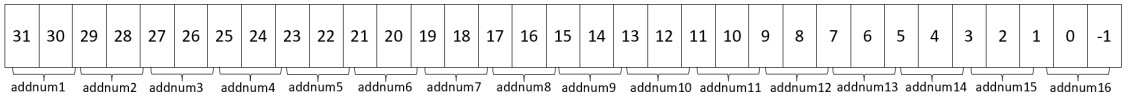

图 7: 16 个 64 位部分积

3. Wallace 树

根据 Wallace 树的原理，综合考虑输入数的个数和 3-2 压缩器的功能，我们设计了如下的 Wallace 树。

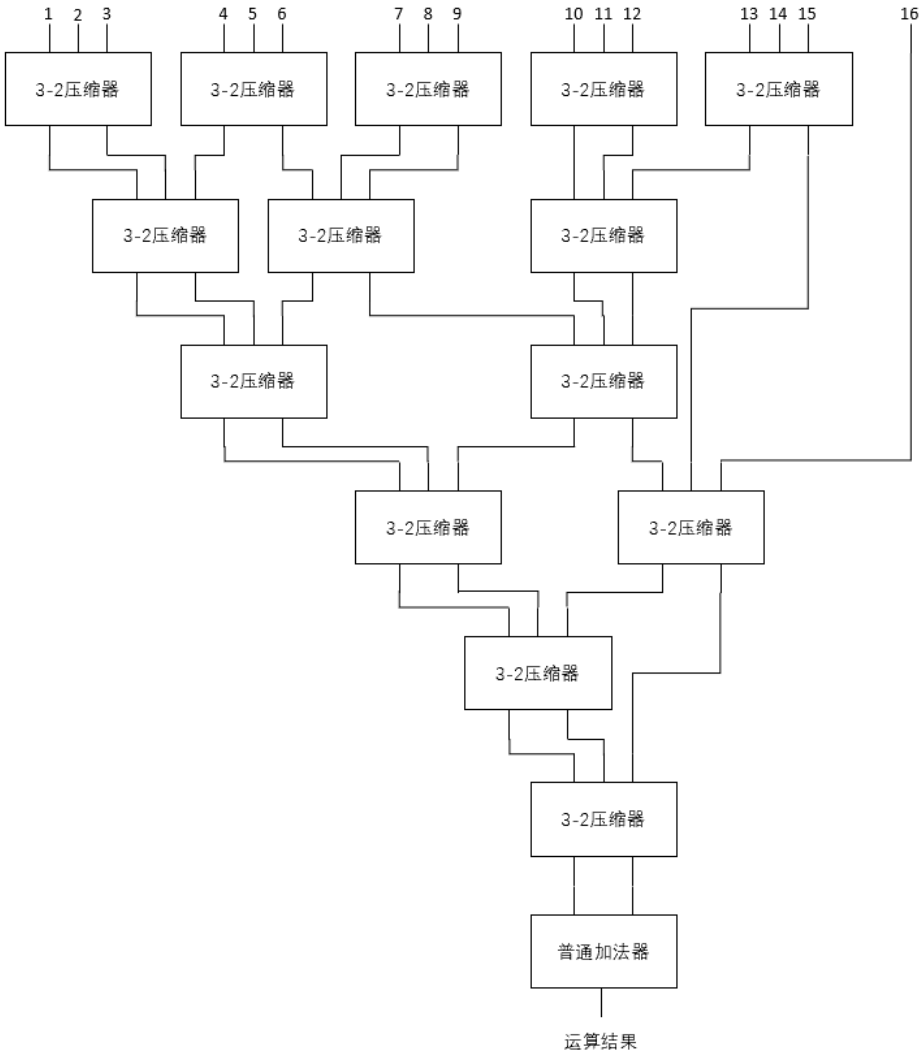

图 8: Wallace 树

4. 仿真结果

根据布斯编码的原理计算 16 个 64 位部分积, 然后根据设计的 Wallace 树结构，我们将部分积不断通过 64 位 3-2 压缩器进行压缩，最终使用 64 位普通全加器来计算最终结果。

```
本轮计算的被乘数为:0001001000010101001101010010010 对应的有符号整数为: 303379748
本轮计算的乘数为  :1100000010001001010111101000001 对应的有符号整数为: -1082744449
现在将输入转化为补码形式，所有操作数不相互依赖，消耗1个时钟周期:
现在生成16个加数，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第一层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第二层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第三层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第四层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第五层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行第六层华莱士树计算，所有操作数不相互依赖，消耗1个时钟周期:
现在进行最后的合并计算，采用普通全加器，消耗32个时钟周期:
答案为:0000100100011110000000101001111010101100111011011111111100100100 对应的有符号整数为: -328482738086018852
现在进行正确性验证:结果正确
```

图 9: 仿真结果

(三) 加速比分析

在对比普通乘法器与本文所设计的快速乘法器的性能时，我们详细分析了两者的运算周期数，并据此计算了加速比。

对于普通乘法器而言，其运算过程包括两部分：32 次乘数与被乘数的移位操作，以获得 32 个部分和，这共需要 32 个时钟周期；部分和的累加，每次累加需要 64 个时钟周期，共需进行 31 次。因此，普通乘法器的总时间周期数为 32+31×64=2016 个。

而对于本文设计的快速乘法器，其优势在于采用了布斯编码和 Wallace 树结构。乘数与被乘数的移位操作可以并行进行，仅需 1 个时钟周期即可获得 16 个部分和。根据 6 层 Wallace 树的特性，部分和的压缩需要 6 个时钟周期。最后，使用 64 个时钟周期来计算最终答案。因此，快速乘法器的总时间周期数为 1+6+64=71 个。

通过对比两者的总时间周期数，我们可以计算出加速比，即普通乘法器所需周期数与快速乘法器所需周期数之比，即 2016/71 28.4。这一结果表明，采用布斯编码和 Wallace 树的快速乘法器在运算速度上较普通乘法器有显著提升。为便于查看，我们将上述分析整理成表格如下：

	普通乘法器	快速乘法器	加速比
运算数移位	32	1	
部分和运算	31×64	6+64	
总时间周期数	2016	71	28.4

表 2: 加速比分析

通过此表格，我们可以清晰地看到普通乘法器与快速乘法器在运算周期数上的显著差异，以及由此带来的加速效果。这一优化分析不仅有助于我们理解快速乘法器的性能优势，也为后续的优化工作提供了参考。

(四) 成本分析

在进行硬件设计时，成本分析是一个至关重要的环节。我们分别针对普通乘法器和快速乘法器进行了成本分析，以比较两者在硬件资源使用上的差异。

对于普通乘法器，其设计相对简单，主要使用了 2 个 64 位寄存器来记录部分和以及最终结果，并使用 1 个 64 位加法器来进行部分和的累加。

而对于快速乘法器，我们采用了布斯编码和 Wallace 树的设计，这导致了硬件资源的增加。具体来说，快速乘法器需要 17 个 64 位寄存器来记录部分和以及最终结果，以支持更多的并行运算。此外，为了实现部分和的压缩，我们使用了 15 个 64 位 3-2 压缩器和 1 个 64 位加法器。

为了更直观地比较两者的成本差异，我们整理成如下表格：

	普通乘法器	快速乘法器
寄存器	2	17
运算器	1	15+1
总器件数	3	33

表 3: 成本分析

四、 总结

在本文中，我们深入探讨了普通乘法器和快速乘法器的设计原理、性能评估以及成本分析。通过对比这两种乘法器的实现方式和性能特点，我们得以全面理解它们在不同应用场景下的优势和局限性。

对于普通乘法器，我们介绍了其基于移位和累加的基本原理，以及在数字电路中的常见实现方式。通过性能评估，我们发现普通乘法器虽然结构简单，但在处理大数乘法时运算周期较长，难以满足一些对实时性要求较高的应用场景。

为了克服普通乘法器的不足，我们进一步研究了快速乘法器的设计。快速乘法器采用了布斯编码和 Wallace 树等高级技术，通过减少部分积的数量和并行压缩部分积，显著提高了乘法运算的速度。通过仿真验证，我们证明了快速乘法器在运算周期上的优势，并计算了与普通乘法器相比的加速比。

然而，快速乘法器在性能提升的同时，也带来了硬件成本的增加。通过成本分析，我们量化了快速乘法器在寄存器数量和运算器数量上的增加，以及处理大位宽数时所需的额外硬件资源。这些成本考虑因素对于在实际应用中选择乘法器设计方案至关重要。

综上所述，普通乘法器和快速乘法器各有其优缺点，适用于不同的应用场景。在选择乘法器设计方案时，需要根据实际需求权衡性能、成本和实时性等因素。未来，随着技术的不断进步和应用需求的不断变化，我们期待乘法器设计领域能够涌现出更多创新性的解决方案。

参考文献

[1] 周婉婷 and 李磊. 基 4booth 编码的高速 32×32 乘法器的设计与实现. **电子科技大学学报**, (S1):4, 2008.

[2] 朱鑫标 and 施隆照. 一种高压缩 wallace 树的快速乘法器设计. **微电子学与计算机**, 30(2):4, 2013.

[3] 胡薇. **高性能冗余二进制乘法器的研究与设计**. PhD thesis, 南京航空航天大学.

[4] 蔡永祺, 李振涛, and 万江华. 基于新型部分积生成器和提前压缩器的乘法器设计. **电子与封装**, (11):91–96, 2023.