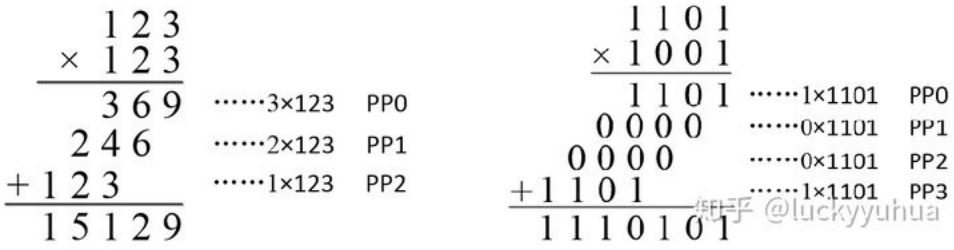
乘法器的布斯算法原理与VERILOG实现

luckyyu...
资不浅数字芯片工程师/闲适奶爸爱吃瓜
+ 关注他

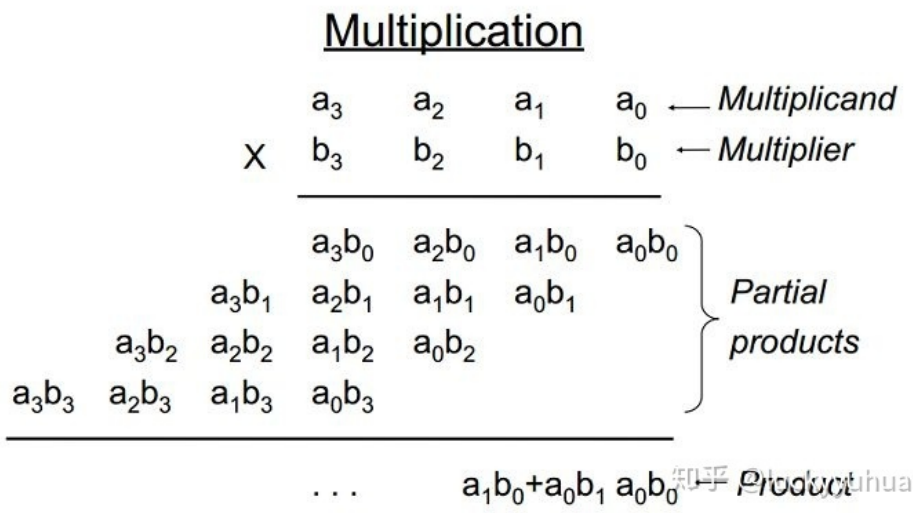
1、乘法器基本原理

乘法器是处理器设计过程中经常要面对的运算部件。一般情况下，乘法可以直接交由综合工具处理或者调用EDA厂商现成的IP，这种方式的好处是快捷和可靠，但也有它的不足之处，比如影响同一设计在不同工具平台之间的可移植性、时序面积可采取的优化手段有限、个性化设计需求无法满足等。所以，熟悉和掌握乘法器的底层实现原理还是有必要的，技多不压身，总有用得上的时候，同时也是一名IC设计工程师扎实基本功的体现。

不采用任何优化算法的乘法过程，可以用我们小学就学过的列竖式乘法来说明。从乘数的低位开始，每次取一位与被乘数相乘，其乘积作为部分积暂存，乘数的全部有效位都乘完后，再将所有部分积根据对应乘数数位的权值错位累加，得到最后的乘积。如下图，左边为十进制乘法过程，基数为10，右图为二进制乘法过程，基数为2。PP0～PP3分别表示每次相乘后的部分积。可见，二进制乘法与十制乘法本质上是没有差别的。



如果表示成通用形式，则如下图所示（以4位乘法器为例，其它位宽类似）



这样原始的乘法在设计上是可以实现的，但在工程应用上几乎不会采用，在时延与面积上都需要优化。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。

2、Booth变换

以上述 1101*1001为例，可以看到产生的四个部分积中，有两个是非零值（PP0, PP3），两个为0，分积的相加，实际

上可以简化成两个非零部分积相加，从而减少加法器的数目。这也就匹配了上节所述乘法优化的第一个方面，减少部分积个数，实际上是减少非零部分积的个数。

同时在该例中不难看出，非零部分积的个数等于乘数中1的个数，如果是1101*1011，那么将存在三个非零部分积，如果是1101*1111，则存在4个非零部分积。那么对于一个给定的乘数，有什么方法可以减少非零部分积个数呢？先以一个例子来具象地展示一下，比如 N*01111110, N为任意非零数。

上式中，乘数为 01111110，存在6个1，如果不作任何变换，将产生6个非零部分积。但如果仔细观察，可以发现01111110可以表示成如下形式：

01111110 = 10000000 – 00000010

通过这个等式，N*01111110可以转化成

N*10000000 – N*00000010

在这个运算式中，相当于只有两个非零部分积相加（减法通过补码转变成加法），一下就省掉了4次累加，大大简化了运算。因为上述变换直观上看是两个数的减法算式，还是不够简练，我们将减法式计算完成，但将计算结果换一种表示方法：不够减时不借位，直接用负数表示该位相减的结果：

即01111110 = 100000-10，-1就是数学中的负1，可以验证这两个数是相等的：

再看下面的通用表达形式：

式1 可看成任意有符号二进制数的补码表达形式，多项式最高项为符号位, y[-1]为附加项，定义其值为0，不影响二进制数的实际值。 对 **式1**右边多项式等价变换：

由 **式1** 等价变换到 **式2**，观察多项式的每一项可知，相邻两bit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，套用上述等价变换，便不难得出上例 01111110 = 100000-10的结论。这种变换，就是**布斯变换**，或称**布斯编码**。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（**式2**）与原二进制数（**式1**）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。虽然[illegible]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。

依旧对 **式1** 进行等价变形：

通过 **式3** 的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 **式3** 所示的改进型布斯编码，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 **式3** 的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)。

由 **式3** 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。

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。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，通过此方法，解决了硬件乘法优化的第一个方面，简化和减少部分积的生成。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(Radix-4 Booth Encoding)。采用基4布斯编码的乘法相较于传统乘法运算，优化效果已经很明显且易于实现，可以满足大部分应用要求，32位乘法器，甚至64位乘法器都可以采用，是比较常用的一种方式。当然，更高阶的布斯编码可以更大程度地减少部分积个数，但因其部分积产生逻辑无法单纯通过移位实现，需要引入加法器等其它运算部件，从这方面来看又削弱了优化效果，需要综合考量选择。高阶布斯编码的实现可以用 式3 类似的方法导出，本文对此暂不作讨论，感兴趣的读者可自行推导。

3、进位保留加法器、3-2压缩与4-2压缩

布斯编码解决了乘法优化的第一个方面，通过减少部分积个数从而减少累加器个数，但累加器本身的进位传递延时对电路性能依然存在非常大的影响，所以优化的第二个方面，就是改进部分积累加结构，提升累加性能。如果采用部分积直接相加的方式，因为全加器进位的关系，当前bit的相加结果依赖于它前一bit的进位输出，整个计算过程相当于串行化，位宽越大，延时越大，所以优化的关键就是消除进位链，使运算并行化。

进位保留加法器（Carry Save Adder, CSA）是比较常用的一种优化方式，CSA实际上就是一位全加器，其逻辑表达示如下：

下面用一个例子对照来说明CSA怎样起到减少加法器延时的作用。

假设将三个4位数相加，分别是A[3:0], B[3:0] 和 C[3:0],先看第一种加法实现方式如下：

上述方式中，第一级加法存在4个CSA进位链带来的延时，如红色简头所示，显而易见如果计算位宽更大，则延时越大。

同样采用CSA，换一种组织方式如下，CSA的两个输入端和一个进位输入端分别接入三个待求和加数，CSA的进位输出作为下一级加法器的输入，如红色箭头所示。可以看出第一级4个CSA是全并行的，一共只消耗一个CAS的延时，并且该延时不会随着位宽的增长而增加。这种方式相当于一次接受3个输入产生2个输出，因此也称为3-2计数器或3-2压缩。运用到乘法器中，通过对若干个部分积按每3个为一组进行CSA计算，然后用同样的方法运用到每级CSA产生的输出上，直到最后一级CSA的两个输出，再用全加器得到最后的部分积累加结果。

3-2计数器由于砍断了行波加法进位链，从而可以实现部分积累加并行，极大削减了计算时延，同时由于存在输入到输出由3到2的压缩关系，也减少了累加级数，从而起到优化运算的效果。

相较于3-2压缩，还有一种压缩率更高的方式叫4-2压缩，或称为5-3计数器。顾名思义，就是有5个输入端，产生3个输出，运用到加法上，可以实现同时计算四个加数，生成对应的结果与进位值。设5-3计数器的5个输入分别为I[0], I[1], I[2], I[3], Ci，3个输出端分别为 D, C, Co，则它们之间满足如下代数关系：

根据上述代数关系式，可推算出真值表：

输出与输入之间的逻辑经化简最后可表达示为：

鉴于逻辑表达示看上去有一定复杂度，用更直观的逻辑图表示如下：

通过相邻两bit的 Co 与 Ci 级连，便可以实现多bit的加法，一次可以对四个加数进行运算。因为 Co 并不参与生成 C 的逻辑运算，所以这种级连不会造成行波进位链，每个5-3计数器之间依然是并行的。

举例：A[2:0], B[2:0], C[2:0], D[2:0]为待累加的四个数，采用如下方式进行4-2压缩后，最终结果为 (T[2:0]<<1) + S[2:0]。

4、设计实例与Verilog实现

下面通过一个 16*16的布斯乘法器，具体地说明上述方法在设计中的运用，更大位宽的乘法器本质上没有区别，按相同的方法扩展即可。

前面对算法原理的论述中，没有提及有符号数和无符号数，但在设计的时候，则需要考虑有符号数与无符号数的区别。实际上布斯编码是带符号位的，也就是它的编码方式是建立在有符号数基础之上，从多项式1的最高次项也可以看出来。所以，采用布斯编码的乘法器是一种有符号数乘法器，

够兼容无符号数计
数时，该位等于扩

展前的符号位。如此会导致增加一个部分积，但带来的便利是对两种不同性质的数可以采用完全一样的计算方式，下面的设计实例遵循此方式进行。

设 A[16:0], B[16:0]分别为乘法器的两个输入，以A为被乘数，B为乘数。A[16]和B[16]为兼容无符号数扩展的符号位。

首先，根据前文内容，对乘数B以基4的方式进行改进型布斯编码，即从LSB开始，以每3bit为一组进行分组，每相邻两组之间的最高位与最低位重合（红色位），LSB的右边还需增加一个附加位，定义为B[-1]，值为零。如下图所示：

B[1:-1]为第1组，B[3:1]为第2组，B[5:3]为第3组，依此类推，一共可划分为9组。其中第9组因为不够3个bit，故在最高位再扩展一位符号位（绿色位），符号位的扩展并不会改变补码数据的值。根据每组的值和**表1**的查找关系，可得出对应的部分积，一共产生9个部分积，定义为PP1~PP9。

从**表1**可以看出，基4布斯编码下，部分积最大为2Y，也就是被乘数的2倍，在此例中即为 A[16:0]<<1，为了保证当A[16:0]在左移后不产生溢出，故部分积 PPn 的位宽比被乘数A多1位，为 。根据式3的各项因子的指数关系，可对9个部分积作如下排列（因每次乘以乘数的两位，故相邻部分积之间偏移两位）。

部分积生成后，即可组建加法树。加法树的构成采用3-2压缩方式或4-2压缩方式均可以，每种压缩方式下也可以有不同的拓扑结构，考虑因素主要是计算效率以及后端布线难度两个方面。

4.1 3-2压缩器组建加法树

下图是采用3-2压缩器构建加法树的其种一种结构形式，数字表示bit位置，s表示扩展的符号位。前文已指出布斯乘法是补码乘法，故符号扩展是必要的，不能省略，否则计算结果不正确。

9个部分积分别为PP1～PP9，从PP1开始，每3个为一组，作为一个3-2压缩器的输入，则9个部分积共消耗3个3-2压缩器，这3个压缩器分别为A11、A12、A13，作为加法树第一级。因为相邻部分积之间的偏移造成3-2压缩器输入不足3个时，低位不够补0（蓝灰色填充区域），高位不够补**符号位（红色填充区域）**。16位乘法器正常积为32位，所以部分积超出32位以外的高位，以及运算

左侧灰色字体表示

每个3-2压缩器具有两个输出端，分别为数据输出与进位输出，因为加法进位权值为2，所以进位输出在运算时要相对数据输出左移一位。第一级3个压缩器共产生6个输出，依照第一级同样的方式，可组成第二级共2个3-2压缩器A21和A22。

因为A21和A22一共生成4个输出，如果继续使用3-2压缩器处理，则多出一个输出，这个输出可以保留到第四级处理；也可以在第三级的时候不使用3-2压缩改用4-2压缩；或者采用两个普通全加器，根据实际情况灵活处理。本文为简单起见，不考虑性能与物理实现等限制因素，采用第一种方式，则第四级压缩器的3个输入分别为A31的数据输出与进位输出，以及保留下来的A22的进位输出（**注意数位对齐不能变**）。

经过第四级压缩器后，只剩A41的数据与进位两个输出值，采用全加器将这两个值相加，即得以最终的乘积。

综上，全部采用3-2压缩方式，一共需要4级压缩和1级全加器，如果设计需要流水线处理，则可根据实际要求在级与级之间插入流水线寄存器，下图是上述计算过程的拓扑结构图，更为合理高效的结构可以上述基础上自行拓展思考。

4.2 4-2压缩器组建加法树

下图是采用4-2压缩器为主组建加法树结构形式中的一种，与3-2结构形式大同小异，仅对几个细节区别加以说明。

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

出如果在最终积的位（标注为co），

实际上是A11最高位的进位输出co与其它四个部分积的符号扩展位相加的结果。。无法加入到4-2压缩器输入的部份积保留到下一级，直到可以参与运算为止。

第三级只有3个数据需要运算，故采用3-2压缩器，也可以采用一个全加器先算出两个数的和，再与第三个数相加，但实际电路速度会相较第一种略差一点（这个位宽下实际差别并不大）。

采用4-2压缩方式，加法树级数相比采用3-2压缩方式更少（位宽越大越明显），但因为4-2压缩器的逻辑结构相比3-2压缩器复杂，电路延时要大，最终综合效率并不一定比3-2压缩高，需要根据实际情况合理设计优化，在速度、面积上取得平衡。

同样给出上述实现方式的拓扑图。

5、代码示例

纸上得来终觉浅，绝知此事要躬行。很多人学东西从书本上一看，好像是那么回事，便觉得理解了，等要真正动手的时候，才发现脑子里一片浆糊，茫茫然不知所措；也有很多人写东西点到即止，总不愿意说透，不知道是他认为读者都懂，还是他自己其实也没有太懂。为了防止这种作者自说自话、读者不知所以的情况出现，我以一份实际代码作为结尾，理论结合实践，对照着看相信更容易理解前文所述内容。

代码是一个16位乘法器，支持有符号运算与无符号运算。代码结构与上述第4节的两个示例完全吻合，未作任何特殊的优化处理，通过一个宏定义可以选择其中的一种实现方式，也可以通过宏定义实现不同的流水线结构。代码中附带了一个简单的可运行testbench和仿真环境，以Synopsys的DesignWare乘法器IP作为对照参考，默认是5级流水，如果改变了流水线级别，则自己去替换对应的DW IP。testbench没有写自动check代码，拉波形看一下就完了，很简单的几个信号，run目录下的Makefile可能需要根据你自己的仿真工具进行相应修改。

代码中的syn目录有一个简单的可运行的DC综合脚本，并附带了一个SMIC 55nm工艺的标准单元库，仅供参考。

你可以将此示例代码用到自己的设计中，但作者不对因使用这份代码可能带来的项目风险承担任何责任，虽然我觉得它是可行的。

链接：pan.baidu.com/s/1XEToRN...

提取码：9nqf

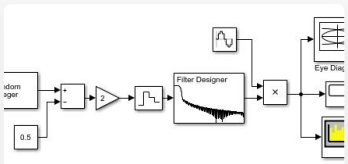
编辑于 2020-04-09

数字电路 Verilog HDL 芯片（集成电路）

推荐阅读

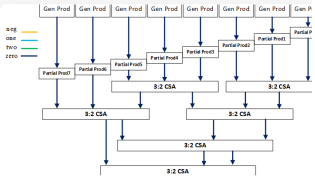
FPGA从入门到精通(5) - 进位链

JAspe... 发表于FPGA从...



（学习Verilog）5. FPGA定点数截位基本准则

万物皆可卷... 发表于BUG记录



【HDL系列】乘法器(6)——Radix-4 Booth乘法器

纸上谈芯 发表于纸上谈芯

6 条评论

切换为时间排序

只有关注了作者的人才可以评论

Trustintruth 2020-04-08

感谢分享，讲的真好必须三联

赞

ljgibbs 2020-04-20

读过之后，收获很大

赞

ljgibbs 2020-04-20

非常感谢

赞

Xduan 2020-05-14

感谢分享，很强大

赞

jeketai 2020-11-21

我最近在研究乘法器。基于booth4编码+csa32压缩+cla加法器实现无符号乘，发现DC综合面积和Timing都比直接a * b综合的差。请问无符号设计上有什么优化方法?

赞

1 条评论被折叠（为什么？）