Digital Design for Multiplication

Norman Matloff

October 15, 2003

1 Overview

A cottage industry exists in developing fast digital logic to perform arithmetic computations. Fast addition, for example, can be accomplished via **carry lookahead adders**. Multiplication, which presents much more of a challenge than addition, is our focus here. A plethora of different digital algorithms for multiplication have been developed. As a sampling of them, we present two such methods here.

2 Example

Consider for example the multiplication of two 4-bit strings using the “pencil and paper" method:

```
     1011
    x1101
    -----
     1011
    0000
   1011
  1011
  -------
 10001111
```

The four rows

```
     1011
    0000
   1011
  1011
```

are called **partial products**. To sum them, we could use three 8-bit adders. The first adder would compute

```
     1011
   +0000
   ------
    01011
```

and the second adder would compute

```
  1011
+1011
------
100001
```

(The inputs to these adders would simply copy 1011 or 0000 to the appropriate places.) The third adder would compute the sum of the outputs of the first two adders. The first two additions would be done simultaneously with each other. If we using registers or some other method, we could even dispense with the third adder, reusing the first adder for the third addition.

If we wish to multiply two 8-bit strings, we would have eight rows of partial products, and would again sum them simultaneously in pairs, using four adders. We would then have four sums, and would break them into two pairs to be summed, etc. In general, to multiply two n-bit strings, we would have an $O(log_2 n)$-stage process.

This, however, would be slow, even using carry lookahead adders, and even accounting for the fact that we are doing a number of things in parallel. So, how can we speed up the process?

3 Direct Simultaneous Computation

This method is similar to the idea of carry lookahead. In the latter, we anticipate the carries, instead of waiting for them to propagate. The same is true here. Instead of waiting for sums and carries, we use a formula which will allow us to compute things ahead of time.

We could actually try to derive a formula, as in the case of carry lookahead, but it is easier simply to write down the truth table and then simplify the sum-of-products expression implied by the table, using Karnaugh or other methods.[1] Here is how to do it for 2-bit strings.

Denote the multiplicand, multiplier and product by (A_1, A_0), (B_1, B_0) and (P_3, P_2, P_1, P_0), respectively. Then the truth table is shown in Table 1. For example, the row in Table 2 corresponds to $3 \times 2 = 6$.

Now remember, the P_i here are the output functions, each of which will need to be implemented in digital logic. We can see, for instance, that

$$P_2 = A_1 \bar{A}_0 B_1 \bar{B}_0 + A_1 \bar{A}_0 B_1 B_0 + A_1 A_0 B_1 \bar{B}_0$$

Table 3 shows the corresponding Karnaugh map. From that, you can see that the simplification for P_2 is

$$P_2 = A_1 B_1 \bar{B}_0 + A_1 \bar{A}_0 B_1$$

The complete set of simplified expressions for the product bits is

[1] Our example here will be small enough to be able to use Karnaugh, but for larger bit strings we would need to use more advanced circuit simplification methods.

A_1	A_0	B_1	B_0	P_3	P_2	P_1	P_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Table 1: Truth Table for 2-Bit Multiplication

1	1	1	0	0	1	1	0

Table 2: Table Entry for $3 \times 2 = 6$

A_1, A_0 \ B_1, B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	1
10	0	0	1	1

Table 3: Karnaugh Map for P_2

$$
\begin{aligned}
P_0 &= A_0 B_0, \\
P_1 &= \bar{A}_1 A_0 B_1 + A_1 A_0 \bar{B}_0 + A_1 \bar{B}_1 B_0 + A_1 \bar{A}_0 B_0, \\
P_2 &= A_1 B_1 \bar{B}_0 + A_1 \bar{A}_0 B_1, \\
P_3 &= A_1 A_0 B_1 B_0
\end{aligned}
$$

4 The Carry-Save Method

In the example in Section 2, one problem is that we were only adding two partial products at a time. One way to speed things up would be to add three partial products—or at least three items of some kind—at a time. This is the goal of the **carry-save method** of multiplication.[2]

Where will we get an adder capable of adding three summands? Actually, we already have one, in the form of Full Adder (FA) components. If you recall, an FA has three inputs, rather than two: It not only has inputs for the two bits to be added together, but also a third input which is meant for the carry-in from the previous bit. Then as long as we can deal with carries in some alternative way, we can use the carry-in input for our third summand. Then for example we could use a block of eight FAs as a device for summing three summands—as long as we find some way to deal with carries, since we're using the carry-in inputs for one of the summands instead of for carries.

Well, then, how *do* we handle the carries? The answer is to postpone adding them. To see how this is done, consider the following example, done using the ordinary "pencil and paper" way:

```
 11
 0111
+1010
-----
10001
```

where the 11 row consists of carries. Now, here is another way to do it, in which we generate carries but postpone using them:

```
 0111
+1010
-----
 1101  bitwise sums
0010   bitwise carries
-----
10001
```

Note the line labeled "bitwise carries." In it, each bit shows the carry that would result from the addition of the two bits in the previous bit position. For example, the rightmost 0 in the 0010 bitwise carries string above is the carry from the rightmost bit position in the original sum, as seen underscored by asterisks:

[2] It can also be used for addition.

```
 0111
   ^*
+1010
   ^*
-----
0010   bitwise carries
  ^*
```

Similarly, the 1 in the bitwise carries string is the carry resulting from the second-to-the-right bit position, as underscored with carats.