Digital Logic Design

A Combinational Multiplier
Using the Xilinx Spartan II FPGA

Objective and Introduction:

A combinational multiplier is a good example of how simple logic functions (gates, half adders and full adders) can be combined to construct a much more complex function. In particular, it is possible to construct a 4x4 combinational multiplier from an array of AND gates, half-adders and full-adders, taking what you have learned recently and extending it to a more complex circuit. The purpose of this document is to introduce how a relatively complex arithmetic function, such as binary multiplication, can be realized using simple logic building blocks.

It is useful to consider how binary multiplication can be performed. Consider the following binary multiplication of two positive 4-bit integer values.

```
multiplicand      1101      (13)
multiplier      * 1011      (11)
                  ----
                  1101
                 1101
                0000
               1101
               --------
               10001111     (143)
              /   / | \ \
          128 + 8 + 4 + 2 + 1 = 143
```

In the course of multiplying two binary numbers, each bit in the multiplier is multiplied with the multiplicand. Each of the four products is aligned (shifted left) according to the position of the bit in the multiplier that is being multiplied with the multiplicand. The four resulting products are added to form the final product.

Since we are dealing with *binary* numbers, forming the products is particularly easy. If the multiplier bit is a 1, then the corresponding product is simply an appropriately shifted copy of the multiplicand. If the multiplier bit is a zero, then the product is zero. 1-bit binary multiplication is thus just an AND operation (look at the truth table to convince yourself).

For an N-bit multiplier and multiplicand (an NxN bit product), the result is 2N bits wide. The result of our desired 4x4 bit multiplication is thus an 8-bit result.

The question then is how can we construct a combinational circuit that can be used to form the necessary products and the final sum? The method used here is sometimes referred to as *partial product accumulation* and is described in detail in many digital logic textbooks; and others provide more in-depth information if you want to learn more), but all you really need to know is summarized from these two sources right here in this handout.

METHOD ONE

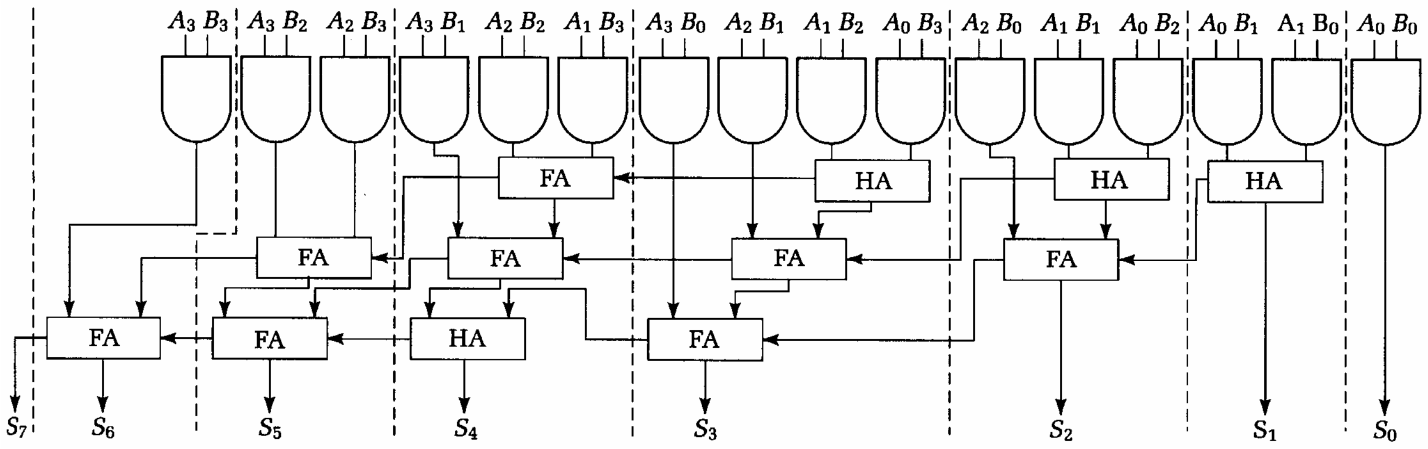
Consider computing the product of two 4-bit integer numbers given by $A_3A_2A_1A_0$ (multiplicand) and $B_3B_2B_1B_0$ (multiplier). The product of these two numbers can be formed as shown below.

$$
\begin{array}{ccccccc}
 & & & A_3 & A_2 & A_1 & A_0 \\
 & & & B_3 & B_2 & B_1 & B_0 \\
\cline{4-7}
 & & & A_3 \bullet B_0 & A_2 \bullet B_0 & A_1 \bullet B_0 & A_0 \bullet B_0 \\
 & & A_3 \bullet B_1 & A_2 \bullet B_1 & A_1 \bullet B_1 & A_0 \bullet B_1 & \\
 & A_3 \bullet B_2 & A_2 \bullet B_2 & A_1 \bullet B_2 & A_0 \bullet B_2 & & \\
A_3 \bullet B_3 & A_2 \bullet B_3 & A_1 \bullet B_3 & A_0 \bullet B_3 & & & \\
\hline
S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0
\end{array}
$$

Each of the ANDed terms is referred to as a partial product. The final product (the result) is formed by accumulating (summing) down each column of partial products. Any carries must be propagated from the right to the left across the columns.

Since we are dealing with binary numbers, the partial products reduce to simple AND operations between the corresponding bits in the multiplier and multiplicand. The sums down each column can be implemented using one or more 1-bit binary adders. Any adder that may need to accept a carry from the right must be a full adder. If there is no possibility of a carry propagating in from the right, then a half adder can be used instead, if desired (a full adder can always be used to implement a half adder if the carry-in is tied low). The diagram below illustrates a combinational circuit for performing the 4x4 binary multiplication.

The initial layer of AND gates forms the sixteen partial products that result from ANDing all combinations of the four multiplier bits with the four multiplicand bits. The column sums are formed using a combination of half and full adders. Look again at the first two illustrations of the binary multiplication process above, and make a careful comparison with the figure below. Make sure you understand the correspondence before you proceed. We will refer to the realization shown below as **METHOD ONE**.



The adder blocks (indicated by FA and HA) in the figure above are drawn in such a way that the two bits to be added enter from the top, any carry in from the right enters from the right, and any carry out exits from the left of each block. The output from the bottom of a block is the sum.

The least significant output bit, S_0 (the first column), involves only two input bits and is computed as the simple output of an AND gate. This operation cannot generate a carry out.

The next output bit, S_1, involves the sum of two partial products. A half adder is used to form the sum since there can be no carry in from the first column; however, a carry out can be produced.

The third output bit, S_2, is formed from the sum of three (1-bit) partial products plus a possible carry in from the previous bit. This operation requires two cascaded adders (one half adder and one full adder) to sum the four possible input bits (three partial products and one possible carry in from the right).

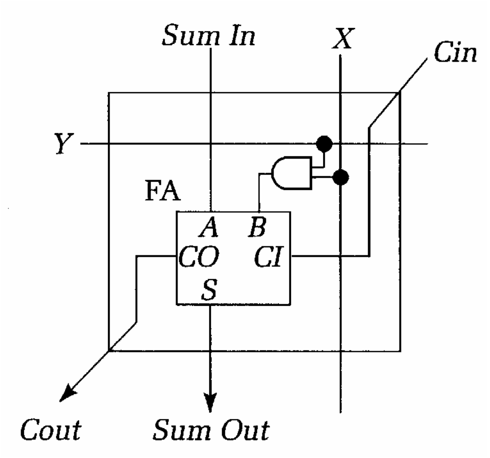
The remaining output bits are formed similarly. Because in some columns we must add more than two binary numbers, there may be more than one carry out generated to the left.

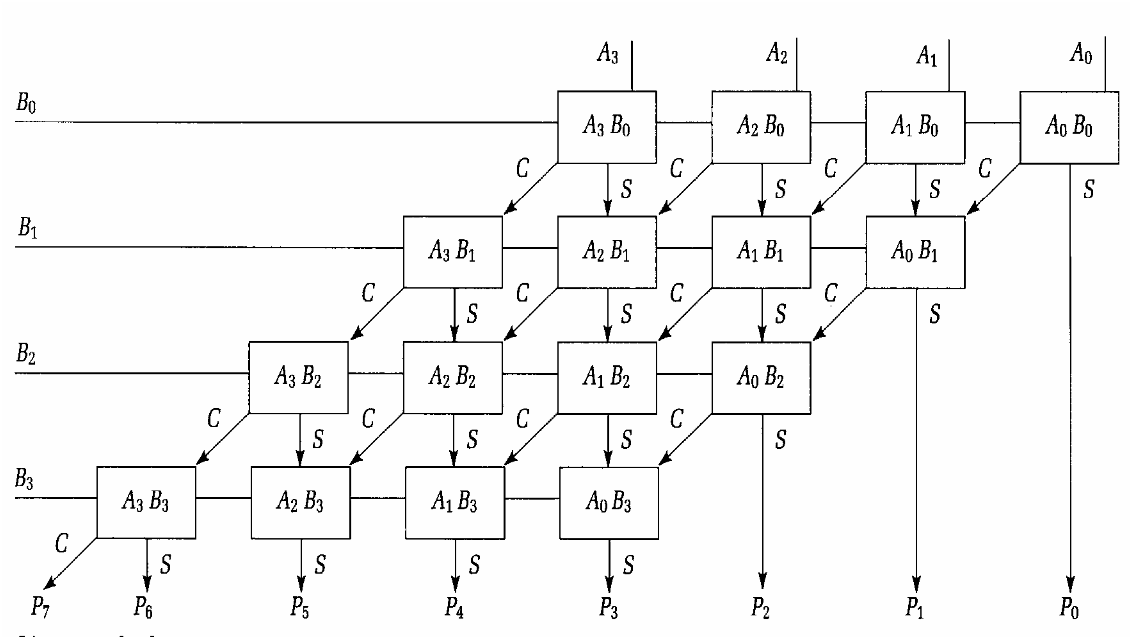
The speed of an adder constructed in this manner is a function of how long it takes the carries from the less significant bits on the right to propagate through to the left. The scheme that is shown above is often referred to as *ripple carry* since each more significant column in the sum must wait for the carries to be computed in the less significant columns before its corresponding sum bit can be computed. It's possible to include *carry lookahead* circuitry to compute the carries in parallel, thus greatly speeding up the computation at an increase in gate count. We won't consider this aspect here.

As noted above, it's not necessary to use both full adders and half adders. For simplicity, all the adders can be chosen to be full adders. The carry in on a full adder can be set to logic-0 wherever the half adder function is needed. This adds slightly to the overall complexity of the circuit but makes the design somewhat more convenient since only one type of adder block is required. If we count six gates per adder (12 adders) and sixteen AND gates on the input, this circuit would require 88 gates to implement.

METHOD TWO

A slightly different implementation scheme is shown in the figure below. This multiplier is constructed from an array of *building blocks*, shown in the first figure below. Each building block consists of an AND gate for computing locally the corresponding partial product ($X g Y$), an input passed into the block from above (*Sum In*), and a carry (*Cin*) passed from a block diagonally above. It generates a carry out bit (*COUT*) and a new sum out bit (*Sum Out*). The second figure illustrates the interconnection of these building blocks to construct a 4x4 combinational multiplier. The A_i values are distributed along block diagonals (look at the products to see the correspondence), and the B_i values are passed along the rows. We will refer to this realization as **METHOD TWO**.





(b) 4 × 4 multiplier structure

<u>**Conclusion:**</u>

Of the two methods, METHOD ONE is probably the easier to understand, but METHOD TWO is more modular and regular. You can see from the two example realizations above that the 4x4 combinational multiplier is not a circuit that you would want to build from discrete gates – there are lots of gates and lots of wires. It's certainly not a circuit that you could complete during a single lab period. It is, however, a circuit that is easy to implement using a programmable logic device such as the Xilinx XC2S50. In fact, we could design and implement much larger circuits than these in a short amount of time.