Constant-coefficient multipliers save FPGA space, time

Steve Knapp

Multiplication is one of the more silicon-intensive functions, especially when implemented in programmable logic. This column demonstrates a technique that requires less silicon for multipliers, where one of the inputs is a constant. It leverages the techniques described in a previous column on lookup-table-based FPGAs (Ref 1).

Multiplying by a constant appears often in applications. Consider just a few examples (multiplication by a constant function appears in italics):

- **Gain-offset amplification**—The heart of most amplifier functions is to multiply an input (x) by a constant (A) with an offset (B) as in $y = Ax + B$.
- **Color-space conversion**—Translating among various color spaces in video applications involves constant-coefficient multiplication and offset calculations. The following equations show the relationship between the RGB space and the YCrCb color space, which is itself a scaled and offset version of the YUV color space (Refs 3 and 4):

$$Y = 0.257R' + 0.504G' + 0.98B' + 16$$
$$Cr = 0.439R' - 0.368G' - 0.071B' + 128$$
$$Cb = -0.148R' - 0.291G' + 0.439B' + 128$$

- **Digital Filtering**—Here's another application with many constant-coefficient multiplication functions. The equation below defines a generalized equation of a FIR filter, which multiplies each of its coefficients by a clock-delayed value of the input:

$$y = c_0x_0 + c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

Review the fundamentals

In cases where one of the factor inputs to a general-purpose multiplier is a constant, an obvious construct is to connect one input to a fixed value. One alternative, a special construct, greatly reduces the silicon area required (Ref 2).

Before delving into its details, though, a quick review of multiplication fundamentals helps demonstrate the overall approach. Consider the case where you want to multiply any integer from 0 to 31 by a constant (in this example, 47). Now take out paper and pencil and multiply 28 times 47—but no cheating, do it longhand the way you learned in elementary school. Most people multiply digit by digit and use the multiplication table stored in their brain.

A similar approach works with LUT-based FPGAs. Most of those devices support at least a 4-input lookup table, allowing as many as 16 possible input values. Think of a 4-input LUT as being a 16 x 1 ROM. With multiple LUTs you can represent 16 data words of practically any width.

Now represent the variable input as a 2-digit decimal value, using T to represent the tens column and U for its units, so now the multiplier's output becomes Product = TU x 47. Fur-

Steven K Knapp is the founder and president of OptiMagic Inc (Aptos, CA, www.optimagic.com), a firm that develops intellectual property and design software for programmable logic. Prior to founding this firm he held various applications, engineering and management positions at Xilinx and Intel's former programmable-logic division.

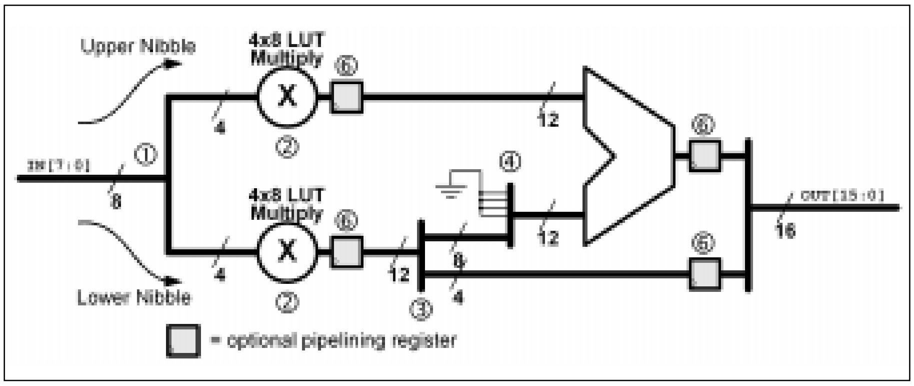


Fig 1—An 8-bit constant-coefficient multiplier function with 4-input LUTs requires two identical sets of coefficient tables and a 12-bit adder.

ther assume that the lookup table has just ten locations. How do you build a function with 32 different values in a lookup table with ten spaces? You must somehow simplify TU into values for T and U, each having just ten possible values.

Luckily, simple math transforms the previous equation into another one:

Product = (T x 47) x 10 + (U x 47)

where both T and U have values between 0 and 9. (In practice, it's possible to further reduce T in this example because it always lies between 0 and 3). Now a single lookup table (Table 1) works for the example. The implied multiplication by ten for T is something you can easily do in your head.

As an example, multiply 28 by 47. First look up the multiplicand 2 in Table 1, which yields 94 for T x 47. Next look up the multiplicand 8, and the table gives 376 for U x 47. Return to the previous equation and plug in the intermediate results for T x 47 and U x 47 to get the result 94 x 10 + 376 = 1316.

Table 1—A lookup table for multiplying by 47 (decimal) is just one step above the multiplication tables that elementary school students memorize.

Multiplicand (decimal)	Product = Multiplicand x 47
0	0
1	47
2	94
3	141
4	188
5	235
6	282
7	329
8	376
9	423

Adapting the approach

Is this method a good technique to use in FPGA designs? Not really, but a similar approach works. We humans deal in decimal numbers most of the time. Unfortunately, multiplying by ten in the binary world costs lots of silicon area. However, you can modify the approach for FPGAs by remembering two facts: First, a 4-input LUT holds as many as 16 different values. Second, multiplying a value by 16 is easy in a binary system—just shift it to the left four times. In an FPGA, you can accomplish this multiply by properly connecting wires in the design.

Instead of working in decimal, an FPGA design becomes more efficient if you work in hexadecimal (base 16). Now the task becomes multiplying the variable input, which ranges from 0 to 1F, by a constant = 2F. The values for the lookup table appear in Table 2, and the modified equation becomes Product = (T x 2F) x 10_{16} + (U x2F). Note that you multiply T x 2F by 10_{16}, which means 16 decimal. Further, you can reduce the lookup table for T x 2F to two entries because in the example T has a limited range from 0 to 1.

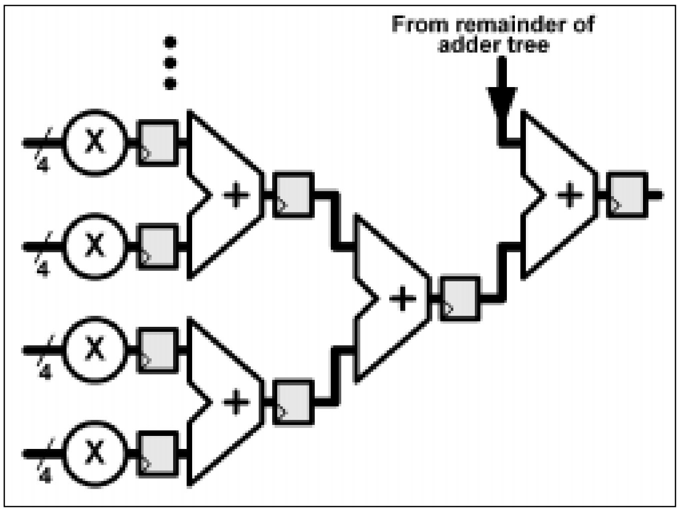


Fig 2—A large-width multiplier requires a large adder tree. While a multiplier isn't limited to any size conceptually, adder delay places a practical limit on the word size a multiplier can handle unless the FPGA need only calculate a few thousand multiplies per second.

Structural implementation

With the algorithm in hand, how do you actually build such a beast? Fig 1 shows a block diagram of the structure for an 8-bit input multiplied by an 8-bit constant. FPGA lookup tables hold the 8-bit constant as the product of the input value, similar to the values in Table 2. Creating this table is probably the most difficult part of the process, but some vendors like Xilinx and Altera provide tools to make the job easier.

The dataflow through the multiplier involves the following six parts:

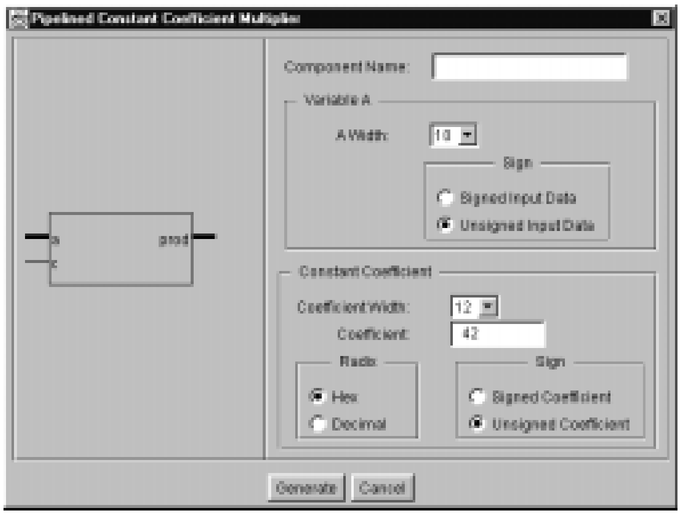
1. This example uses 4-input LUTs to build the multiplier lookup table. Because its value is eight bits wide, the input signal splits into two 4-bit nibbles.

2. Each nibble connects to its own copy of the coefficient lookup table. The 4-bit input nibble, multiplied by an 8-bit constant, results in a 12-bit output. Each nibble multiplier consumes 12 LUTs.

3. Before adding the products from the nibble multipliers, the design must multiply the upper result by 16 (decimal). However, instead of shifting the upper nibble by four places, strip off the bottom four places from the lower nibble multiplier because the adder doesn't need them. Consequently, the size of the final adder drops from 16 bits down to 12 bits. Luckily, the adder carry delay is also the critical path, so this approach minimizes logic requirements and reduces overall delay (Ref 2).

4. The input to the 12-bit adder derives from the upper eight bits of the lower nibble multiplier, appended with four "stuff" bits at the top. This example assumes that all the values are positive and the stuff bits are all Low. For negative values, the stuff bits should all be High. To support both positive and negative data, the "stuff" bits are the sign extension of the upper eight bits from the lower nibble multiplier.

Fig 3—The Xilinx Core Generator software produces nearly optimal implementations of constant coefficient multipliers.



Pipelined Constant Coefficient Multiplier

Component Name:

Variable A
A Width: 10
Sign: Signed Input Data [✓ Unsigned Input Data]

Constant Coefficient
Coefficient Width: 12
Coefficient: 42
Radix: [✓ Hex] Decimal
Sign: Signed Coefficient [✓ Unsigned Coefficient]

5. The 12-bit adder recombines the results from the upper nibble multiplier and the extended upper eight bits from the lower nibble multiplier. The addition generates no carry output. The final multiplier output equals the output of the adder and the lower four bits from the lower nibble multiplier, producing a 16-bit result. The adder is usually the critical path in these apps.

Multiplicand	*Product = Multiplicand x 2F*
0	000
1	02F
2	05E
3	08D
4	0BC
5	0EB
6	11A
7	149
8	178
9	1A7
A	1D6
B	205
C	234
D	263
E	292
F	2C1

Table 2—A lookup table for multiplying by 2F (hex) requires the use of base 16 instead of base 10 math. Still, the table reduces the multiplication process into a series of additions.

6. Optional pipelining registers help boost performance through the multiplier. Clock rates of 100 MHz or more are possible if the design tolerates the extra clock latency. A pipelined adder provides the greatest increase in performance but at the cost of additional silicon and clock latency.

The structure in Fig 1 shows how to create a multiplier for an 8-bit input, but what about wider inputs? The width of the LUT used to build the multiplier lookup table determines the overall structure. Using a 4-input LUT, an FPGA processes the input in 4-bit chucks. The logic scales results from each nibble multiplier to its proper binary weighting and then sums them together through an adder tree (Fig 2). Again, note the optional pipelining registers to boost performance.

Is this trip necessary?

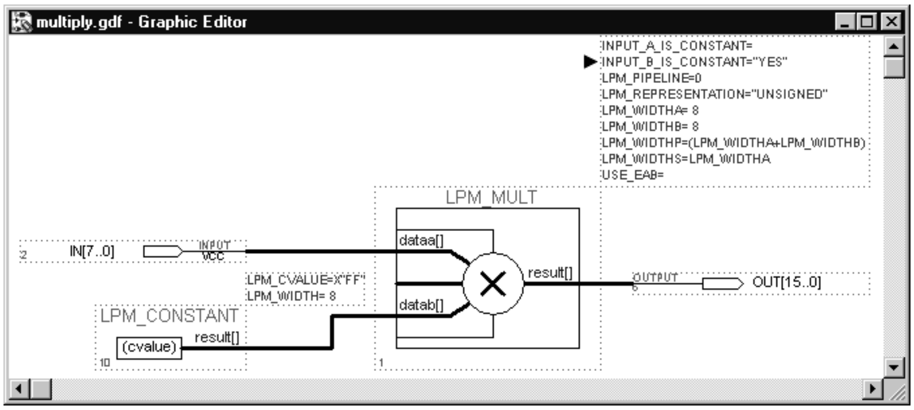
After running through this exercise, you might be asking "Is this effort really necessary?" A few statistics might convince you of the answer. A design built with Xilinx XC4000E-1 devices forms the basis for the data in Table 3. The multiplier style directly affects the number of XC4000E logic blocks required to build the design and the resulting performance. Connecting a constant value to an input on a 8 x 8-variable multiplier reduces its size, but not to the extent of the technique described here. In general, a constant-coefficient multiplier is about a quarter to a third the size of an area-optimized multiplier and as fast or faster than a performance-optimized multiplier. Results for other FPGA architectures with 4-input LUTs are comparable, assuming optimal implementation.

Luckily, a few FPGA vendors offer direct support to build constant-coefficient multipliers. One of the better tools is the Xilinx Core Generator (Ref 6). It produces nearly optimal solutions for constant-coefficient multipliers including relative placement information for maximum performance. The bit width of the input and of the coefficient is user selectable (Fig 3). Additionally, the core generator automatically creates a symbol for various schematic editors, an HDL template file for instantiating the core in VHDL or Verilog as well as HDL simulation models.

Table 3—The logic and time required for a multiplier with one constant and one variable input are much lower than for multipliers that handle two variable inputs.

Multiplier	XC4000E Logic Blocks	XC4000E-1 Performance
8x8 parallel, area optimized (Ref 5)	54	83 MHz
8x8 parallel, performance optimized (Ref 5)	70	97 MHz
8-bit constant, 8-bit variable	19	> 100 MHz

Altera also offers some support through the LPM_MULT multiplier library primitive included with Altera's Max+Plus II development package. This general-purpose multiplier function allows users to specify bit widths and pipelining. To generate a constant-coefficient multiplier, you must indicate to the LPM_MULT function that one of the inputs is a constant. Fig 4 demonstrates this indication via the INPUT_B_IS_CONSTANT parameter. You create the constant value in this example using the Altera LPM_CONSTANT function. **PE&IN**



```
INPUT_A_IS_CONSTANT=
INPUT_B_IS_CONSTANT="YES"
LPM_PIPELINE=0
LPM_REPRESENTATION="UNSIGNED"
LPM_WIDTHA= 8
LPM_WIDTHB= 8
LPM_WIDTHP=(LPM_WIDTHA+LPM_WIDTHB)
LPM_WIDTHS=LPM_WIDTHA
USE_EAB=
```

```
LPM_CVALUE=X"FF"
LPM_WIDTH= 8
```

Fig 4—The LPM_MULT function provides support for constant-coefficient multipliers in Altera devices.

Acknowledgment

The author thanks Ken Chapman from Xilinx for introducing the above concept and pioneering this work in FPGAs.

References

1. Knapp, S, "FPGA lookup tables build flexible pattern matchers," *PE&IN*, May 1998, pgs 56-60.
2. Chapman K, "Constant Coefficient Multipliers for the XC4000E," *Application Note XAPP 054*, Dec 11, 1996, Xilinx Inc (www.xilinx.com/xapp/xapp054.pdf).
3. "RGB2YcrCb & YcrCb2RGB Color Space Converters", *Intellectual Property Products Data Sheet*, Feb 1997, Altera Corp (www.altera.com/document/ds/rgb.pdf).
4. Jack, K, *Video Demystified: A Handbook for the Digital Engineer*, Ch 3, 1993. HighText Publications ISBN 1-878707-09-4 (www.optimagic.com/books.html#Video).
5. Core Solutions Data Book; Xilinx Inc, Feb 1998 (www.xilinx.com/products/logicore/core_sol.htm).
6."Xilinx CORE Generator", Xilinx Inc (www.xilinx.com/products/logicore/coregen/index.htm).

Editorial Feedback
This article's value to me was:
High—263 Average—264 Low—265