Parallel processing in FPGAs rivals DSP speed

Steve Knapp

My previous column described a fast, efficient method to implement multipliers where one of the inputs is a constant value (Ref 1). Algorithms employing fixed coefficients abound, especially in image processing and digital signal processing. For example, the equation $Cr = 0.439R' - 0.368G' - 0.071B' + 128$ is part of a calculation to convert between the RGB and YCrCb color spaces in digital video. Another example is the equation $y = c_0x_0 + c_1x_1 + c_2x_2 + \ldots + c_nx_n$, which gives a generalized equation for many filtering algorithms.

The standard method to implement these algorithms is with a DSP chip or microprocessor. A typical device incorporates a high-performance multiply/accumulate (MAC) unit, which in most apps performs high-speed multiplication of a variable input and a constant and then accumulates (adds) the results from many operations to produce a final result.

The MAC unit necessarily needs to be very high performance because a DSP reuses a single shared resource to implement an inherently parallel algorithm such as a digital filter. It time-division multiplexes the algorithm in the MAC to conserve logic. Each separate multiply and accumulate step happens sequentially. The MAC must be fast to obtain reasonable performance.

Field-programmable gate arrays (FPGAs) generally can't perform a single multiply/accumulate step as fast as a modern DSP. However, with a little architectural magic, an FPGA outperforms a leading-edge DSP because instead of reusing one extremely fast MAC dozens of times per calculation, it can calculate dozens of moderately fast MAC steps in parallel.

This column describes general techniques for implementing multiply/accumulate functions in an FPGA, building on techniques described in earlier columns (Refs 1 and 2).

Mutliply/accumulate steps

Fig 1 shows a fully parallel implementation of the general algorithm in the following equation:

$$Y = k_0X_0 + k_1X_1 + k_2X_2 + k_3X_3$$

Each variable input X_0 through X_3 is multiplied by a constant coefficient. Summing the four products determines the final result, Y. Implemented in a typical DSP, this simple equation requires four distinct multiply/accumulate steps plus some overhead instructions to move data. Recall from

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.

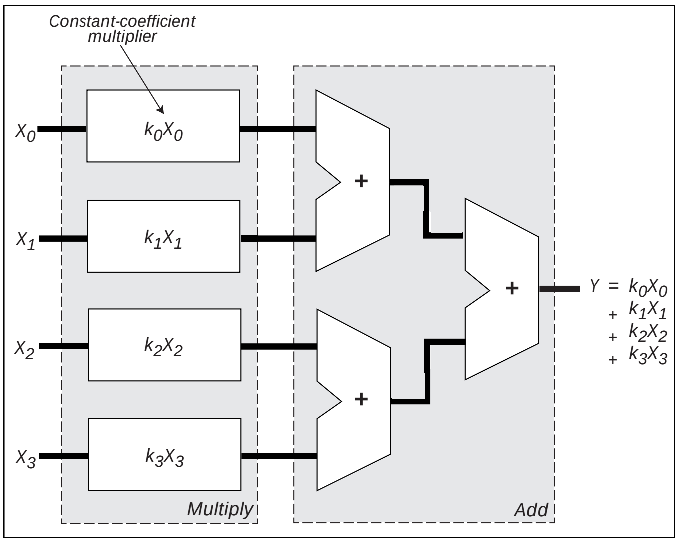
Constant-coefficient multiplier
X_0
X_1
X_2
X_3
k_0X_0
k_1X_1
k_2X_2
k_3X_3
+
+
+
$Y = k_0X_0 + k_1X_1 + k_2X_2 + k_3X_3$
Multiply
Add

Fig 1—An FPGA outperforms a programmable DSP processor by handling multiply/ accumulate operations in parallel.

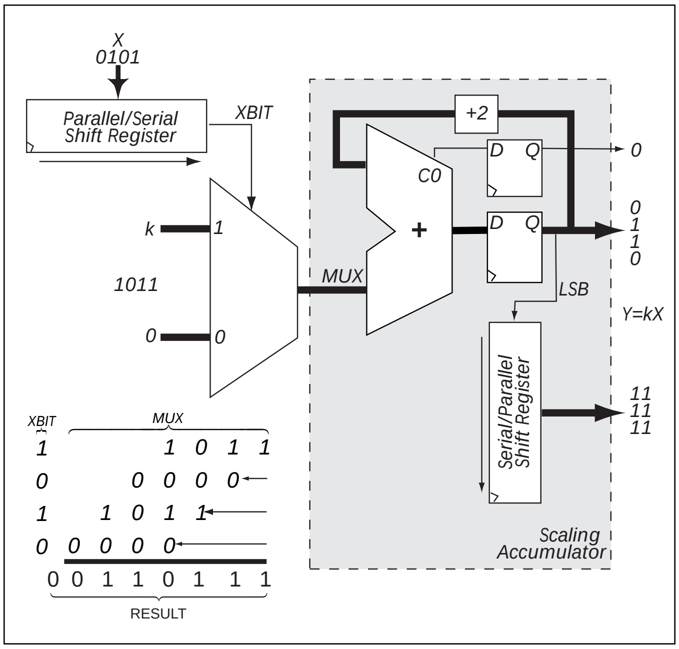

Fig 2—A shift-and-add approach simplifies the logic required to build a multiplier.

Ref 1 how to implement a constant-coefficient multiplier efficiently in lookup-table-based FPGAs, and then you can implement this simple function in parallel on even the smallest one.

Unfortunately, as word size or the number of products increases, hardware complexity usually increases dramatically. Is there a way to reduce the required logic? Luckily, parallel algorithms allow a broad range of speed vs area tradeoffs.

The easiest place to reduce logic is in the constant-coefficient multipliers. Instead of using a parallel approach, might it save logic to use the classic serial shift-and-add approach?

An old classic

One possible multiplier implementation is the classic shift-and-add approach (Fig 2). Here you load the parallel data input *X* into a parallel-to-serial shift register. On every clock edge, the LSB from the shift register, XBIT, selects a value from the 2:1 multiplexer. If the shifted bit is Zero the mux feeds a Zero to the scaling accumulator; if the shifted bit equals One, the constant value *k* appears on the accumulator input.

The scaling accumulator performs two functions. First, it sums the previously accumulated values with the partial products provided on a clock edge by the multiplexer output. Second, it shifts the accumulated value to the right on every clock edge, performing a divide-by-two and thereby adjusts the result by its proper binary weight. You perform the right-shift operation by wiring the proper feedback bits from the register back into the accumulator. The LSB from the adder feeds into the serial input on a serial-to-parallel shift register. During the calculation, the lower bits of the result appear as outputs from the shift register. The output of the accumulator—a registered adder with feedback—and its carry output provide the upper bits of the result. To demonstrate the function, annotations on Fig 2 show the constant value set to 11 (*k* = 1011 binary) and the variable input set to 5 (*X* = 0101 binary). The result after four clock cycles is 55 (00110111 binary). The lower three bits of the result shift from the accumulator into a serial-to-parallel shift register. The upper bit value is the carry output from the accumulator.

The bit-width of the value *X* determines the processing time. In this example, *X* is a 4-bit entity. Consequently, the shift-and-add multiplier supplies a new result every four clock cycles. The critical path limiting the clock performance is usually the carry propagation time through the adder.

The circuit in Fig 2 could effectively replace each constant-coefficient multiplier in Fig 1. Contrary to expectations, this shift-and-add approach actually increases the amount of logic required, especially for narrower word widths. Implementing constant-coefficient multipliers using LUTs in FPGAs is much more efficient. Should you therefore discard this entire approach or is there a way to restructure the problem and make it more efficient?

Serial-distributed arithmetic

Using a technique called distributed arithmetic (Refs 3, 4 and 5), you can reorganize the function in Fig 1 and fit it into a structure similar to the diagram in Fig 2. This approach has the added advantage that it absorbs the adder tree, conserving logic. In Fig 2, a bit shifted out from the *X* input determines whether the multiplexer selects the constant *k* or *Zero*. Fig 3 shows how this technique is expandable to process all four inputs simultaneously. Instead of selecting a single constant, this alternative selects various precomputed combinations of the four constants depending on the bits shifted out from the four parallel-to-serial shift registers.

For example, assume the outputs shifted from X_3, X_1 and X_0 are Zeroes, while the output shifted from X_2 is a One. As a result, the mux inputs S[3:0] = 0100 select a precomputed value, which is X_2's corresponding constant k_2. What happens if the values from the shift registers are mixed Ones and Zeroes? Assume that S[3:0] = 1100. Now the precomputed value $k_3 + k_2$ appears on the multiplexer output and into the scaling accumulator.

The new approach in Fig 3 behaves much like the one in Fig 2. The primary difference is that all four multiplications happen in parallel and share a larger multiplexer. It still produces a new result every four clock cycles. In fact, additional inputs and coefficients have no effect on the number of clock cycles, only the word width of the incoming values. For an n-bit word, the circuit requires n clock cycles to complete one calculation, regardless of the number of inputs or coefficients. The critical path is still the carry delay through the accumulator. However, combining all the precomputed constants into one multiplexer function reduces overall resource requirements.

At first glance, it might appear that the 16:1 mux in this distributed-arithmetic approach requires a significant amount of logic. However, its 16 inputs are constants, and it requires only the four select inputs S[3:0]. As described in Ref 2, if a function has four inputs and one output, then it fits into the 4-input lookup tables (LUTs) in some FPGAs. Here the 16:1 multiplexer reduces down to a few lookup tables. The maximum number of LUTs depends on the width of the widest coefficient value—the coefficient needn't match the parallel input width—plus the ceiling of the $\log_2$ of the number of coefficients:
number_of_LUTs = max(coefficient_width) + ceil(log2(number_of_coefficients)).

This space left blank intentionally.

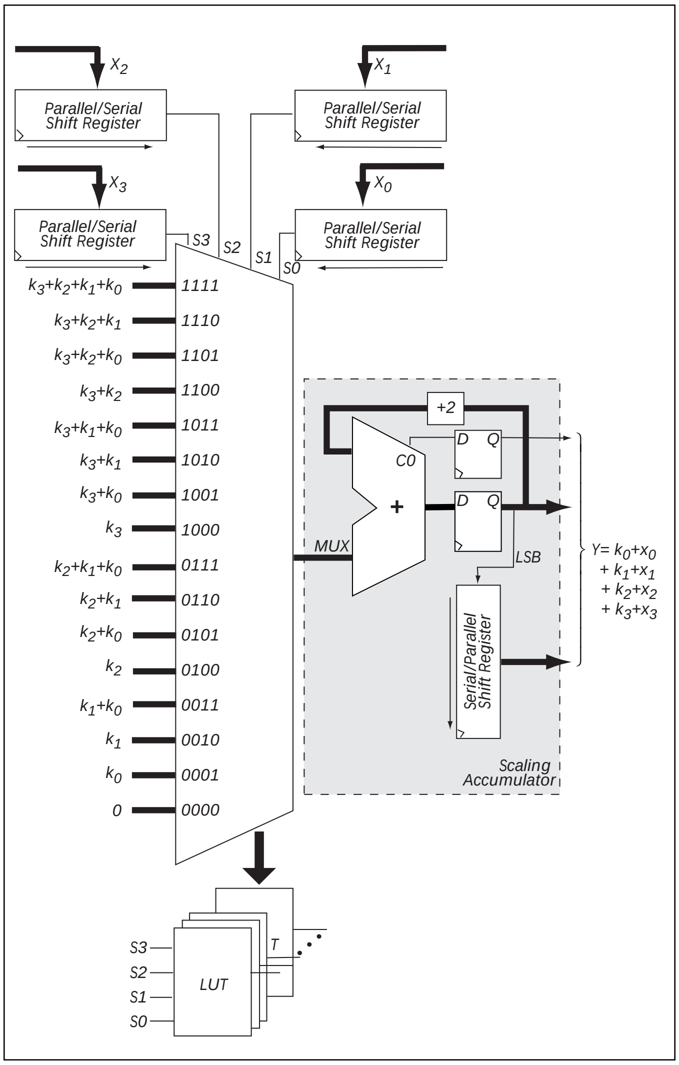

Fig 3—A serial-distributed arithmetic approach reduces the constant-coefficient multipliers and the adder tree to a lookup table.

adder has six inputs from the multiplexer or LUTs, and the scheme holds the six outputs in flip-flops. It shifts the three lesser significant bits (*n-1*) into the serial-to-parallel shift register. The carry output from the adder is the result's MSB.

The architecture in Fig 3 works fine given only four *X* inputs and corresponding coefficients. Fig 4 shows how to extend the architecture to practically any number of inputs. Groups of four inputs fit nicely into a 4-bit chunk, whose coefficients you precompute and place in a LUT. The scheme sums outputs from multiple chunks in an adder tree before arriving at the scaling accumulator. While each adder contributes to the critical path, optional pipelining registers—abundant in LUT-based architec-

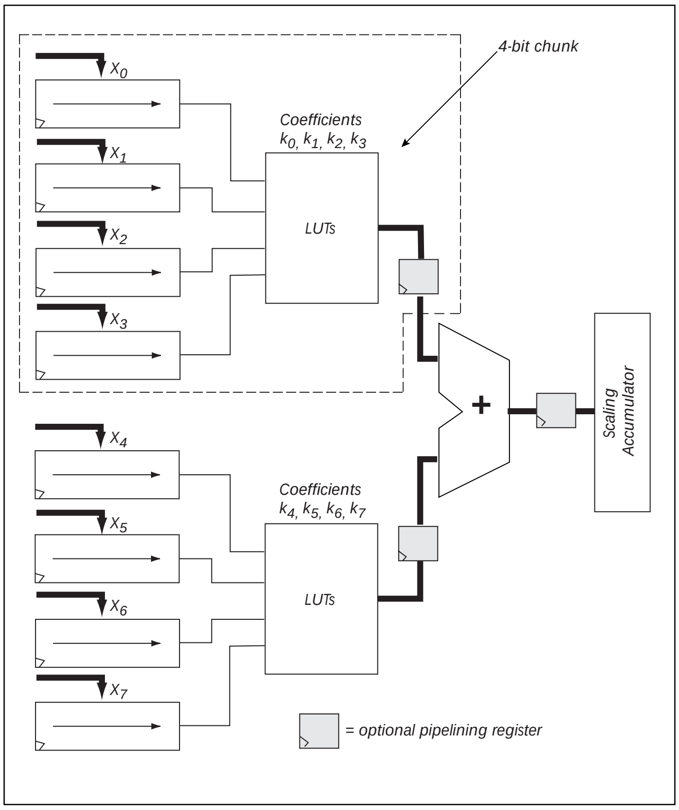
4-bit chunk
X0
X1
X2
X3
Coefficients k0, k1, k2, k3
LUTs
X4
X5
X6
X7
Coefficients k4, k5, k6, k7
LUTs
+
Scaling Accumulator
= optional pipelining register

Fig 4—Multiply/accumulate (MAC) functions with more than four inputs must be cascaded with adders before the final scaling accumulator.

Because the coefficient values are precomputed, the additions might generate a carry, thus requiring a wider word width. For instance, adding four 4-bit values might necessitate a 6-bit result. Thus if each coefficient is four bits wide and there are four coefficients, then the design requires six LUTs and the input to the scaling accumulator is six bits wide.

With four 4-bit inputs and four 4-bit coefficients, the scaling accumulator produces a 10-bit result. The tures—improve performance but at the cost of additional clock latency.

Next steps

The concepts described here can extend to other application. Digital filtering involves significant amounts of multiply / accumulate operations. Finite-impulse response (FIR) filters, for example, accept one data input, and intermediate values are time-delayed versions of the original input. Fig 5 shows how modifying the dataflow within the 4-bit chunk creates a structure more suitable for building FIR filters. An initial parallel-to-serial shift register captures a new *n*-bit data word every *n* clock cycles. The shift-register's output feeds the LUT containing precomputed coefficients, and the same output feeds a cascaded chain of serial-in / serial-out shift registers. Each shift register is part of a single tap in the filter.

My next column demonstrates how to build efficient high-performance FIR filters in FPGAs using the techniques described here. Incredible performance levels are possible by exploiting the FPGA's ability to implement parallel operations. **PE&IN**

References

1. Knapp, S, "Constant-coefficient multipliers save FPGA space, time," *PE&IN*, July 1998, pgs 45-48 (*www.pein.com/1998/PEIN0798/0798ecs.pdf*).
2. Knapp, S, "FPGA lookup tables build flexible pattern matchers," *PE&IN*, May 1998, pgs 56-60 (*www.pein.com/1998/PEIN0398/0598ecs.pdf*).
3. Goslin, G R, "A Guide to Using FPGAs for Application-Specific Digital Signal Processing," Xilinx Inc (*www.xilinx.com/appnotes/dspguide.pdf*).
4. Minster, L, "The Role of Distributed Arithmetic in FPGA-based Signal Processing," (*http://home.att.net/~pcuenin/theory1.PDF*).
5. New, B, "A distributed arithmetic approach to designing scalable DSP chips," *EDN*, Aug 17, 1995 (*www.ednmag.com/reg/1995/081795/17df5.cfm*).

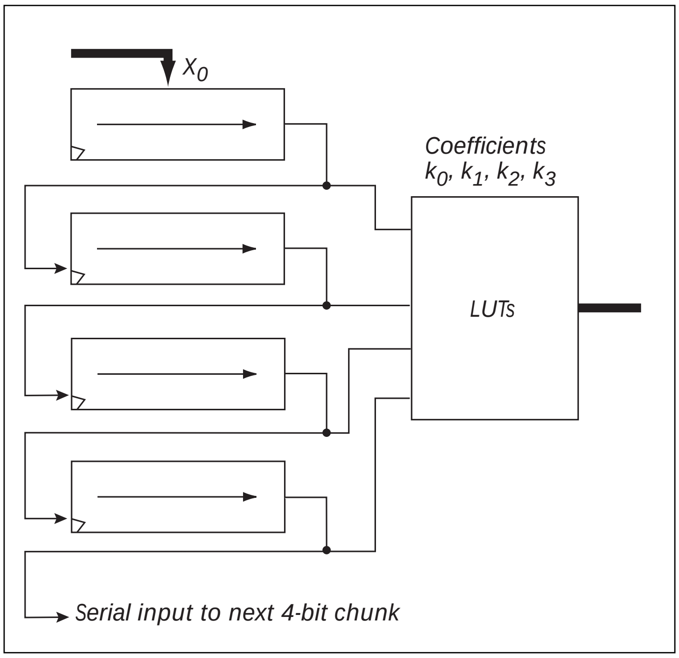

Fig 5—Modifying the data flow through the 4-bit chunk makes it more efficient for digital filtering.

Editorial Feedback
This article's value to me was:
High—266 Average—267 Low—268

This space left blank intentionally.