Constant Coefficient Multipliers for the XC4000E

XAPP 054 December 11, 1996 (Version 1.1)

Application Note by Ken Chapman

Summary

This paper identifies two points at which constant coefficient multipliers become the optimum choice in DSP, and implements constant (k) coefficient multipliers (KCMs) in the XC4000E. It also reveals the solution to an interesting design problem which emerges, and some additional enhancements since the original paper, introducing a hybrid technique, was first published in 1993.

Xilinx Family

XC4000E

Demonstrates

Distributed product arithmetic to avoid data bottlenecks and increase performance.

Introduction

The Xilinx XC4000E devices are becoming increasingly employed to implement functions associated with Digital Signal Processing (DSP). Often the key to such DSP functions is the ability to implement multipliers which are both efficient in their use of silicon, and of adequate performance.

During 1993, a hybrid technique combining on-chip memory and fast carry logic addition was published, and its use gained momentum.

In line with the requirements of the time, the technique offered a way to vary the multiplicand for a high speed data path. Although it was clear that a fixed multiplicand (or constant coefficient, ‘k’) could also be performed with the attractive advantage of occupying less silicon, it did not suit the applications targeted. Now, in 1996, there is a demand for DSP to be performed at the very highest of data rates. This naturally leads to a high degree of parallel processing where no longer is it always required to vary multiplicand values.

This paper identifies two points at which constant coefficient multipliers become the optimum choice in DSP, and implements constant (k) coefficient multipliers (KCMs) in the XC4000E. It also reveals the solution to an interesting design problem which emerges, and some additional enhancements since the original paper.

High Performance = Constant Coefficient

As always, the FIR filter forms such a perfect example by which to explain this strange “equation” (see Figure 1).

During the period between the capture of data samples, the DSP activity of an FIR filter must multiply and accumulate each of the previous data samples (taps), by an associated coefficient value. For a given filter characteristic, each of the coefficients are fixed. The method employed in standard “DSP” devices is akin to the operation of a microprocessor with a dedicated multiply and accumulate (MAC) instruction performed in hardware. When implementing a 16 tap FIR filter, the tap data and associated coefficients are applied to the MAC in turn. Not only does this require 16 MAC instructions to be performed with associated data handling, but also a fully operational high performance multiplier in which both input operands will be changing. Even provided with a MAC instruction rate of 20 ns (or 50 MIPS), the maximum sample period of such a filter will be restricted to 320 ns (16x20 ns), or 3.125 Msamples/s. When such a sample rate is adequate, resource sharing provides an optimum solution.

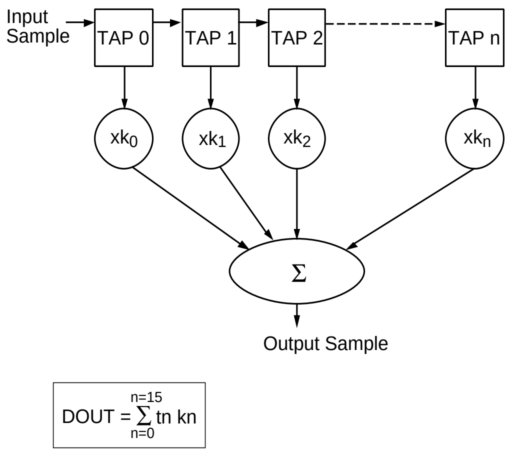


Figure 1: Principle of an FIR Filter

To increase the performance of this filter requires a degree of parallel processing. In simple terms the introduction of a second 50 MIP processor will double the sample rate to 6.25 Msamples/s. The additional overhead of resolving this system would in fact prevent a full doubling of performance. Multipliers in each processor still require full functionality as each still services 8 taps with a single MAC. To continue increasing performance, more and more processors can be introduced. In the limit, there are 16 processors, each servicing a single tap. Although in theory the performance would now have reached a full 50 Msamples/s, the system servicing overhead would now severely impair this figure or require several additional processors. This is by no means a practical implementation! However, the important observation to make is that at this extreme limit of performance, each multiplier is performing tap data multiplication by one fixed coefficient value. This means that the fully functional multiplier provided in the processor is no longer an optimum solution.

Hence, in full parallel processing, constant (k) coefficient multipliers (KCMs) can often be considered.

A Non-Linear Progression

As the technique for implementing KCMs in the XC4000E is so efficient in its use of silicon, it becomes practical to opt for a fully parallel implementation earlier than expected.

In general, a KCM is one quarter of the size of a fully functional multiplier. Therefore, regardless of whether performance targets have been reached by resource sharing a single full functional multiplier up to four times, a full parallel implementation using several KCMs becomes a smaller solution, with increased performance a free bonus.

The Hybrid Technique

Back to School

Consider how decimal multiplication is performed by hand using the following 85x37 example:

```
     85
 x    7
 ------
     35
 +  560
 ------
    595  →  7x85       →        85
                               x    37
                               -------
            7x85       →           595
            3x85x10    →       + 2550
                               -------
                                  3145
```

We use each digit of the second number in turn to multiply all of the first number. Unfortunately, we do not naturally know our 85 times table, so we actually perform a localized single digit multiplication and add the value carried over from the previous column. By this time, we do know our 7 times table having been "programmed" at school. Finally, we add the two products.

Improving Speed

If we had to perform a lot of multiplications involving 85, it would save a great deal of time to build an 85 times table. This would reduce our task to looking up single digit products and then addition.

```
                 0 x 85 =   0
                 1 x 85 =  85
       85        2 x 85 = 170
  x   476        3 x 85 = 255
  -------
      510        4 x 85 = 340
     5950        5 x 85 = 425
  + 34000        6 x 85 = 510
  -------
   404600        7 x 85 = 595
                 8 x 85 = 680
                 9 x 85 = 765
```

Logic

The hybrid technique of multiplication is a hexadecimal equivalent of the long hand method. Look-up tables implemented in RAM or ROM are programmed with an appropriate times table. Since a single hex digit represents four bits, the table has entries for 0 to 15(F). The table for multiplication by 85 decimal (55 Hex) is shown in Table 1.

Table 1: Table for Multiplication by 85 Decimal (55 Hex)

```
       55
  x    2B
  -------
      3A7   →   B x 55
  +  0AA0   →   2 x 55
  -------
     0E47
```

0 x 55 = 000	8 x 55 = 2A8
1 x 55 = 055	9 x 55 = 2FD
2 x 55 = 0AA	A x 55 = 352
3 x 55 = 0FF	B x 55 = 3A7
4 x 55 = 154	C x 55 = 3FC
5 x 55 = 1A9	D x 55 = 451
6 x 55 = 1FE	E x 55 = 4A6
7 x 55 = 253	F x 55 = 4FB

Just as with manual multiplication, having the 85 times table on hand makes this part of the multiplication process very fast. It also eliminates the requirement to build logic that emulates either a calculator (slow) or the memory of a school child (huge!).

Just as before, the final result is obtained by simple addition, with appropriate offset. An 8-bit (2 hex digit) hybrid multiplier as shown in Figure 2.

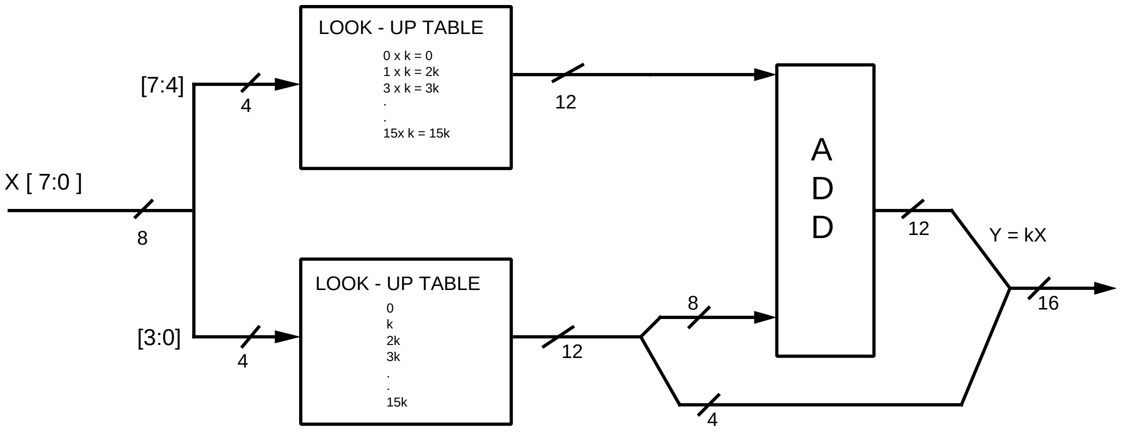


Figure 2: 8-bit Hybrid Multiplier with Optimum Adder Stage

In an optimum implementation, the addition logic can be reduced. During the addition of the two products obtained from the look-up tables, the least significant hex digit (4 bits) is always added to zero. These bits will therefore not be affected, or contribute a carry into the next column.

Engineering an Optimum Solution

FPGA Features

The XC4000 FPGA provides an excellent target architecture for this hybrid technique for the following reasons:

- The configurable Logic Blocks (CLBs) contain all the logic to perform 2-bit addition including a fast carry propagation to adjacent CLBs. This yields optimum adders of any size with the minimum of design effort.
- Any CLB can be used to represent RAM or ROM. In each case, a CLB can be configured for either a 32 x 1-bit, or 16 x 2-bit memory. RAM or ROM of any data width can then be made by combining these small elements, and is ideal for building the look-up tables in the multiplier.

Building a Look-up Table

In the 8-bit example, the look-up tables must provide 16 results ranging from 0 to 15 times the multiplicand value. Such products are up to 12 bits wide (4+8). This requires 6 CLBs to implement as shown in Figure 3.

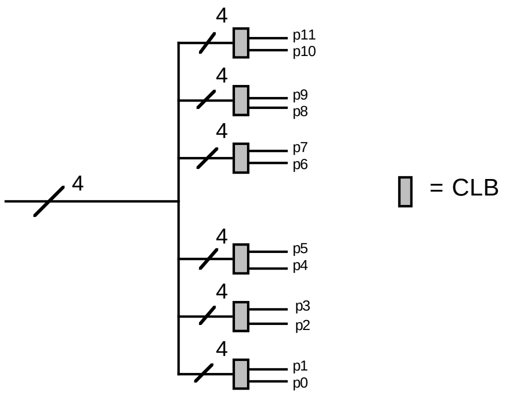


Figure 3: Forming a 12-bit Wide Look-Up Table Using Six XC4000 CLBs

Programming the Table

The data content of each CLB memory is a bit-slice across all the required products and is indicated in the 85 (decimal) times table below (see Table 2). It is necessary to program each memory element with its required pattern.

Table 2: Bit Slice Data for an 85 (Decimal) Look-Up Table

Address (hex) Data (hex)	p11	p10	p9	p8	p7	p6	p5	p4	p3	p2	p1	p0
0 x 55 = 000 =	0	0	0	0	0	0	0	0	0	0	0	0
1 x 55 = 055 =	0	0	0	0	0	1	0	1	0	1	0	1
2 x 55 = 0AA =	0	0	0	0	1	0	1	0	1	0	1	0
3 x 55 = 0FF =	0	0	0	0	1	1	1	1	1	1	1	1
4 x 55 = 154 =	0	0	0	1	0	1	0	1	0	1	0	0
5 x 55 = 1A9 =	0	0	0	1	1	0	1	0	1	0	0	1
6 x 55 = 1FE =	0	0	0	1	1	1	1	1	1	1	1	0
7 x 55 = 253 =	0	0	1	0	0	1	0	1	0	0	1	1
8 x 55 = 2A8 =	0	0	1	0	1	0	1	0	1	0	0	0
9 x 55 = 2FD =	0	0	1	0	1	1	1	1	1	1	0	1
A x 55 = 352 =	0	0	1	1	0	1	0	1	0	0	1	0
B x 55 = 3A7 =	0	0	1	1	1	0	1	0	0	1	1	1
C x 55 = 3FC =	0	0	1	1	1	1	1	1	1	1	0	0
D x 55 = 451 =	0	1	0	0	0	1	0	1	0	0	0	1
E x 55 = 4A6 =	0	1	0	0	1	0	1	0	0	1	1	0
F x 55 = 4FB =	0	1	0	0	1	1	1	1	1	0	1	1

1	1	0	1	1	0	1	1	0	1	1	0	1	1	0	0

p7 = DB6C

In a RAM based hybrid multiplier, as used so often in the past, a simple state machine is used to control a counter and accumulator which program the appropriate pattern into the memory elements. In effect, the designer never needs to work out what the bit-slice patterns really are. However, when designing a fixed coefficient multiplier using ROM tables, the designer must state the memory contents. See Table 2.

Standard Tools

In general, a designer can not be expected to manually perform the bit-slice process. Such a task is prone to errors at the very least. For this reason a memory generator utility is provided into which the designer specifies the size of ROM required and the data which it should hold. For example:

DATA 000, 055, 0AA, ... etc.

The memory generator (MemGen) rapidly produces a netlist with the required number of CLBs tagged with bit-slice data for programming.

Topology

In common with just about everything else in nature, the layout of logic gates in a structured and orderly manner yields improvement in several ways. First, local proximity of communicating elements not only reduces time delay for the propagation of signals, but also avoids congestion, which expedites the routing process. Secondly, an ordered placement (or floorplan) improves density, making optimum use of the target silicon. Finally, when provided with implementing a whole system, the more detail that is resolved and fixed in each module, the simpler it is for people and software tools to make rapid progress.

The hybrid method of producing KCM modules, while optimum in size and performance, are in themselves quite involved subsystems. A 16-bit version would in fact require eighty of the 16x1 ROM elements to build the four look-up tables, two 20-bit adders, and one 24-bit adder (where each bit of an adder requires sum and carry logic). In addition, 160 registers are required to provide full data path pipelining for maximum data rate should it be required. In total that is 368 discernible logic functions to be arranged in an optimum relationship to each other.

Jig-Saw Puzzle

Automatic placement software has been evolving for many years, not only in the area of Field Programmable Logic, but also in standard cell, gate array and PCB development. These tools are now very good, often producing results better than any novice user. However, just like any 400 piece jig-saw puzzle, it takes intense processing and time to complete. Implement an FIR filter using several KCMs and suddenly the puzzle grows to 3000 pieces! Unlike a jigsaw, there is not just one solution. Instead there are thousands of solutions ranging from very good (high performance), to very poor (low performance).

Since KCMs are fixed functions which are likely to be used repeatedly in a system, it makes sense to complete these smaller "jig-saws" once and for all, effectively making some "larger pieces."

Relationally Placed Macros

The Xilinx software tools (XACT*step*™) provide a method for tagging logic elements with placement information relative to other elements. This mechanism permits macros to be shaped while not defining an exact position for any one element on the die. A KCM can now be engineered into shape with consideration for performance, connectivity and overall dimensions. Tagging up to 368 elements in a macro in this way is not for the faint hearted, but it does only need to be done once, and it makes near perfection available for all subsequent uses of each KCM (see Figure 4). Manual floorplanning or automatic placement of a system also becomes a large piece jig-saw as opposed to a large jig-saw of many pieces!

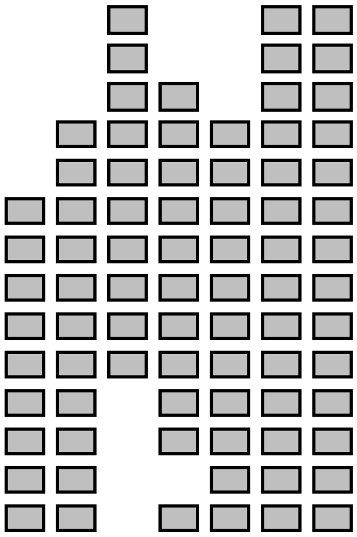


Figure 4: RPM of a 16-Bit Pipelined KCM

There's Always a Problem

In order to tag logic elements with these relative placements (RLOC parameters), these elements must be accessible. Unfortunately the memory generation utility can be considered as a specialized synthesis program. The memory elements which form the look-up tables of a KCM do not exist until synthesized by the program and consequently RLOC tags would need to be added to every individual element after each new look-up table was synthesized. That would be eighty tags for every 16-bit KCM!

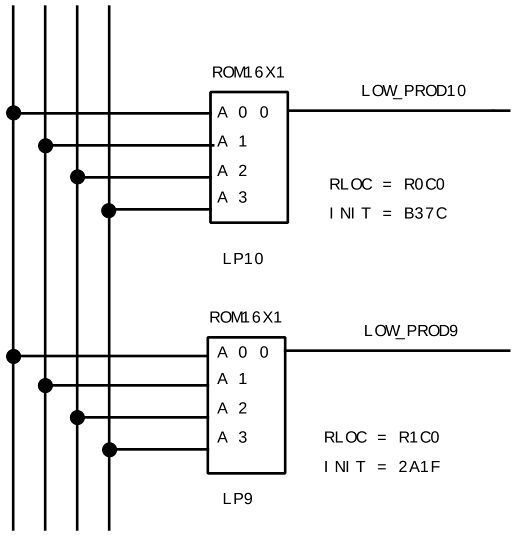
To gain access to memory elements earlier, the memory primitives must be directly specified in the macro. This solves the issue of applying RLOC tags, but brings back the issue of how to specify the bit-slice data for each memory element forming the look-up tables.

Where There's a Will There's a Way

Faced with having to use memory primitives in order to define the placement, a new user-friendly way to program the look-up tables is required. The solution is a new software utility.

The program is required to perform the same bit-slice task as the memory generation utility, but instead of synthesizing the memory elements required, it needs to identify existing memory elements and change their programming data (INIT property).

Regardless of the method used for design entry (schematic, VHDL, etc.), the processing of a design results in some form of "Netlist" describing the types of components and their connectivity. The Xilinx Netlist Format (XNF), in common with many other netlists, is a fairly simple ASCII file in which locating logic elements is easy. Substituting the programming data for these elements is then a straight forward process of string manipulation and file handling; see Figure 5.



```
SYM, LP9, ROM, SCHNM=ROM16X1, LIBVER=2.0.0, DEF=ROM, RLOC=R1C0, INIT=2A1F
PIN, A0, I, X0
PIN, A1, I, X1
PIN, A2, I, X2
PIN, A3, I, X3
PIN, O, O, LOW_PROD9
END
SYM, LP10, ROM, SCHNM=ROM16X1,LIBVER=2.0.0, DEF=ROM, RLOC=R0C0, INIT=B37C
PIN, A0, I, X0
PIN, A1, I, X1
PIN, A2, I, X2
PIN, A3, I, X3
PIN, O, O, LOW_PROD10
END
```

Figure 5: Schematic and Netlist for Two ROM Elements

Capitalizing on Software

The primary purpose of this software utility is to generate memory bit-slice data, and then perform string substitution. However, having covered such basic requirements, a software utility can add value of its own.

The Calculator

Although very obvious, the first value the program can give is the fundamental data for the look-up table. The user should only need to specify the constant (k) for the multiplier and subsequently all details of the table (as in Table 2) can be performed.

The hybrid technique is ultimately an integer multiplier. Even so, this does not prevent fractional multiplication from being performed since integers can be used to represent fixed point values. In this example, the same binary patterns illustrate true integer and fixed point multiplication (note the position of the binary point in the result).

$$83 \quad x \quad 29 = 2407_{10}$$

$$01010011 \text{ x } 00011101 = 0000100101100111$$

$$10.375 \text{ x } 3.625 = 37.609375_{10}$$

$$01010.011 \text{ x } 00011.101 = 0000100101.100111$$

In this case the software utility would help the user by permitting constants to be entered in their true form, but then allowing the hybrid technique to formulate the best position for the binary point to both minimize rounding errors and maximize result resolution. Some additional research is required before making this automatic.

It is important that designers today realize that “integer” does not exclude fractions; they are just hiding!

Negative Thoughts

Representing negative values can be performed in two ways:

- Signed Binary – A bit is used as a flag to indicate that the value is negative. The value is then represented in standard unsigned binary:

  -12 $\longrightarrow$ 10001100

- Twos-complement – This format still uses the leading bit to represent the polarity, but negative values now require interpretation:

  -12 $\longrightarrow$ 11110100

Signed Binary has an initial simplicity, but during arithmetic operations the sign bits require separate handling. Twos complement representation inherently processes using standard arithmetic logic for addition and hence has become the generally adopted format in most microprocessors and digital systems.

The hybrid technique described thus far has only considered positive values for coefficient and incoming data. It is possible for the technique to also handle twos complement.

The partial products emerging from the look-up tables are applied to adders. As long as the partial products are themselves in twos complement format then this section requires no special logic. It is important to sign extend one input to each adder to compensate for the offset (see Figure 6)

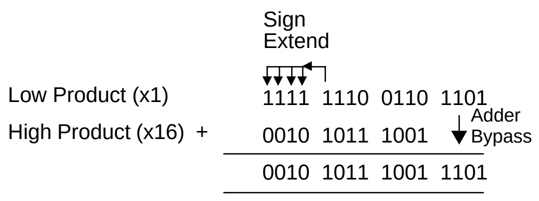


Figure 6: Sign Extension of Partial Products During 4-bit Offset Addition

So the mechanism for implementing signed multiplication is contained in the look-up table data. For negative coefficients (k) the look-up table becomes filled with negative values, i.e. if k = -3 then each look-up table is programmed with; 0, -3, -6, -9, -12, etc. This will then produce correct results providing the input data (x) is still unsigned binary. When the input (x) is also twos complement there would be a failure; consider X = -1, now the bit pattern of "1111 1111" will attempt to access 15k (F x k) from each look-up table and produce totally the wrong result.

The art is to analyze a twos complement number in a particular way. The leading bit not only indicates the polarity of the data, but can be considered to be the only negative contribution to the value. For example, the most significant bit of an 8 bit number normally represents 12810 (i.e., 27). However, in a twos complement 8 bit number, this one bit now represents -128, but with all other bits still representing positive contributions to the total value (see Table 3)

Table 3: Bit Values for an 8-bit Twos Complement Number

Bit	7	6	5	4	3	2	1	0
Value	-128	+64	+32	+16	+8	+4	+2	+1

Looking now at X = -1, it can be seen to be made up as follows; -128 +64 +32 +16 +8 +4 +2 +1 = -1. It can be seen that the lower four bits (bits 3 down to 0) still form an unsigned binary number, and hence the look-up table which these lower bits address remains the products 0k to 15k.

The upper four bits (bits 7 down to 4) actually form a 4-bit twos complement number which can represent values -8 up to +7. The look-up table which these address must be programmed with products which reflect these values. These are 0k to 7k, followed by -8k to -1k in ascending address order. For larger operands, such as 16 bits where four look-up tables are employed, only the look-up table associated with the sign bit of "X" requires this special programming consideration. Such considerations are best removed from the designer and implemented in the software utility.

Results

The KCMs which have been produced at time of publication are 8, 10, and 16-bit versions. The 8 and 16-bit versions often form a reference point for many users as well as interfacing well with microprocessors. The 10-bit version is more practical for several of today's video applications and definitely proves to be fast enough.

Since all details of the KCM size and shape are fixed, the number of CLBs (configurable logic blocks) occupied is known before implementing the macro on silicon. Table 4 indicates this CLB count for each KCM. Often in the combinatorial KCMs, several of the CLBs are not fully utilized and may implement additional logic in a system design. Also, unused CLBs will be removed by the software should they not be required, such as when using only 16 of the 32 bit result from a 16-bit KCM.

Performance in system depends on three factors:

- Speed grade of device.
- Quality of interconnection within macro.
- Connectivity of macro to surrounding system.

The -3 speed grade was selected. This grade is the fastest XC4000 available today (January 1996), for which the figures are fully defined. Faster silicon is available but preliminary figures would introduce uncertainty. The static timing analyzer (XDelay) identifies the critical path in a design independent of test vectors. The performance figure it specifies is for highest temperature and lowest voltage (worst-worst-case).

While the KCMs are defined with regard to placement (shape), their connectivity internally and externally is still dependent on automatic routing tools. Analyzing several designs containing KCMs indicated that so long as the KCM was communicating with logic in relatively close proximity, the critical path was actually within the KCM macro (more specifically, associated with the final adder stage). The close placement of these internal logic elements then leads to very little variation in results. Table 4 presents performance allowing for connection to other logic in the FPGA.

Table 4: Size and Performance for 8, 10 and 16-bit KCMs (XC4000E-3).

Operand Size	Combinatorial Delay (ns)	CLBs	Pipelined Performance (MHz)	No. of Stages	CLBs
8	19	19	66.5	2	20
10	29	39	58.2	2	40
16	41	75	50.0	3	80

Analysis

Fully functional multipliers (both inputs changing simultaneously) have been the subject of investigation within Xilinx for the past year. There are several methods with various advantages and disadvantages. An overview of all techniques in the -3 speed grade is given in Table 5.

Although in some cases these techniques may have even higher performance than a KCM, such speed results from the highest number of CLBs. In general a KCM is 3 to 3.8 times smaller than a fully functional multiplier while offering comparable performance.

Table 5: Overview of Size and Performance for Fully Functional Multipliers

Operand Size	CLBs	Combinatorial Delay (ns)
8	57 to 73	23 to 36
16	230 to 270	38 to 84

Conclusions

KCM System Advantage

Where fixed coefficients can be employed in a system, employing 3 to 4 KCM macros in place of one fully functional multiplier will return higher system performance. This distributed product arithmetic will also avoid data “bottlenecks,” further increasing performance. Returning to the example of an FIR filter, it has been very easy to obtain performance greater than 50 Msamples/s on a single FPGA using KCMs.

Engineering Solutions

This paper has attempted to describe not only the results of a project, but also the engineering involved in arriving at a solution. The technique of forming “near perfect” logic templates and then developing a simple software utility to modify the parameters would appear to be suitable for many applications. Even at a system level, it is common for much logic to remain unchanged, i.e. a UART could have a logic template, and the baud rate could be modified to the exact application requirement.

FPGAs Are Not Constant

It should always be remembered that FPGAs such as the XC4000 are SRAM based devices and can be reconfigured. This means that a KCM is only constant for a given configuration and can easily be modified. This proves useful during the tuning of a system or for future field upgrades.

XILINX® The Programmable Logic Company^{SM}

Headquarters	Xilinx, Inc. 2100 Logic Drive San Jose, CA 95124 U.S.A.	Tel: 1 (800) 255-7778 or 1 (408) 559-7778 Fax: 1 (800) 559-7114	Net: hotline@xilinx.com Web: http://www.xilinx.com