Transposed Form FIR Filters

Author: Vikram Pasham, Andy Miller, and Ken Chapman

XAPP219 (v1.0) September 21, 2000

Summary

This application note describes a high-speed, reconfigurable, full-precision Transposed Form FIR filter design implemented in the Virtex™ series and Spartan®-II family of FPGAs. The VHDL reference design provided with this application note can be easily modified to change filter parameters including coefficients and the number of taps. By illustrating a design methodology for digital filters, the advantages of using FPGAs for digital signal processing applications (DSP) are emphasized. The Core Generator tool provides a preoptimized alternative solution to this reference design (**Core Generator Tool**).

Introduction

Digital filters are among the most significant components in digital signal processing applications. The function of a filter is to eliminate undesirable parts of the signal (random noise), or to extract signals in a particular frequency range. In other words, a filter selects, suppresses, or modifies certain frequency components of the signal, either to reduce noise or to shape the spectrum. This application note focuses on digital filters that are used widely in digital video broadcast, digital video effects, and digital wireless communication. Figure 1 is an example application of filters in a communication receiver.
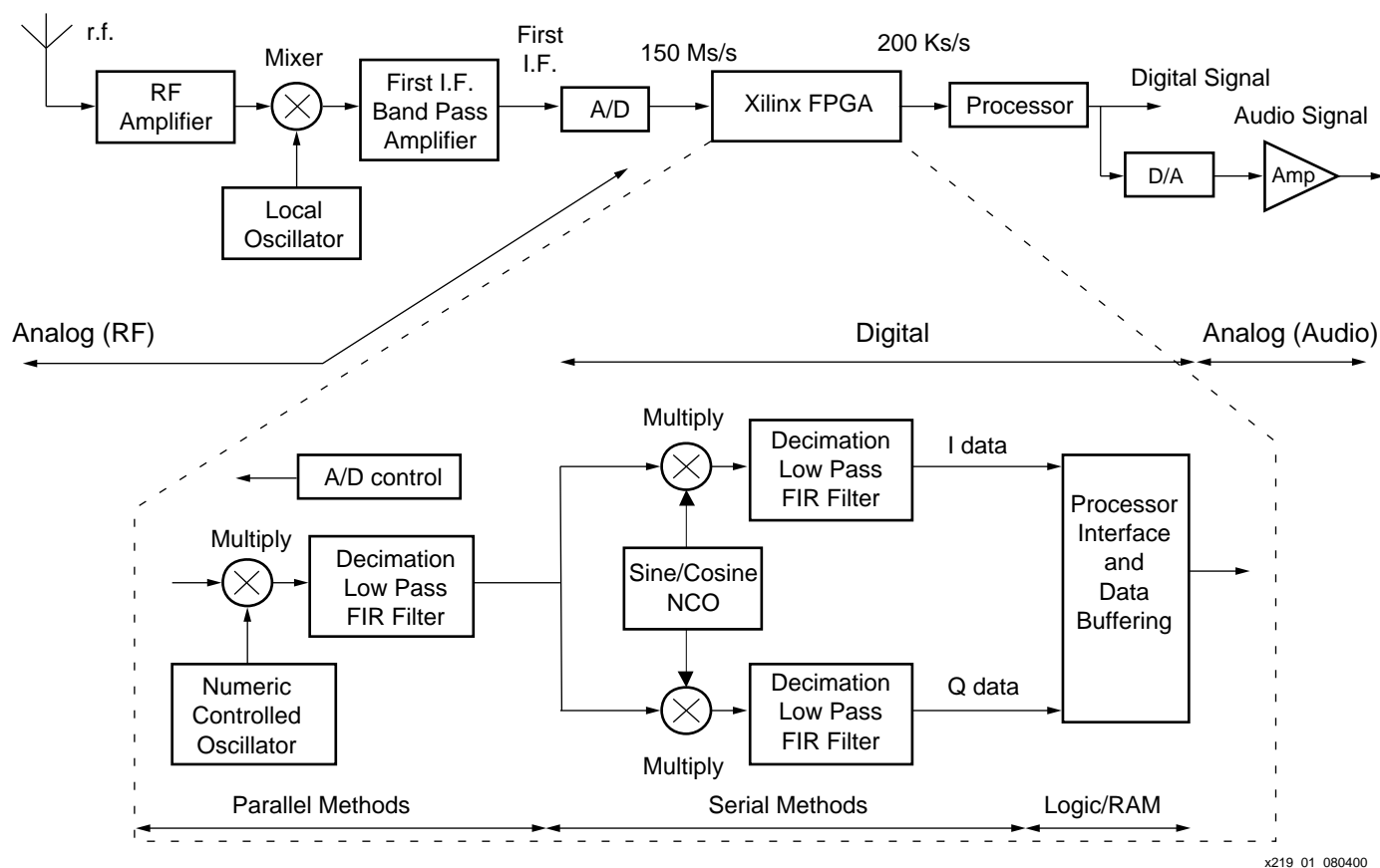

Figure 1: **Filter Applications: Communication Receiver**

Most of the traditional filters in DSP applications are implemented using highly specialized DSP processors. These DSP processors are capable of carrying out high-speed Multiply Accumulate (MAC) operations, but have bandwidth limitations. Only a fixed number of operations can be performed by these processors before the next sample arrives, thereby limiting the bandwidth. DSP processors are sequential in nature, and thus DSPs using a single processor can only perform one operation on a single set of data at a time. For example, in a 16-tap filter, they can only calculate the value of a single tap at a time, while the other 15 taps wait for their turn. This also limits the overall frequency of the application. Due to resource limitations, operations cannot be performed in parallel.

FPGA based filters are implemented with parallel-pipelined architecture, enhancing the overall performance. Thus, a 16-tap filter will run as fast as a 64- or 128-tap filter implemented in an FPGA. The FPGA implementation enables total access to the precision of the signal at each stage of the algorithm. This is a significant difference between an FPGA-based filter and an equivalent DSP processor solution. Implementations of digital filters with sample rates of a few MHz are generally difficult and expensive to realize using standard DSPs. The potential for parallel processing and reprogrammability makes Virtex FPGAs an ideal solution. The flexible architecture of FPGAs permits optimum use of the available gates in the form of Constant Coefficient Multipliers (KCM). The reprogrammability of FPGAs enables tuning of the filter at any time.

Structures for FIR Filters

Digital filter algorithms are primarily composed of multipliers, adders, and registers. The basic structure of a Finite Impulse Response (FIR) filter is shown in Figure 2. The multipliers and adders form the heart of a FIR filter. The input data passes to the multiplier and then to the adder with interleaving delay elements.
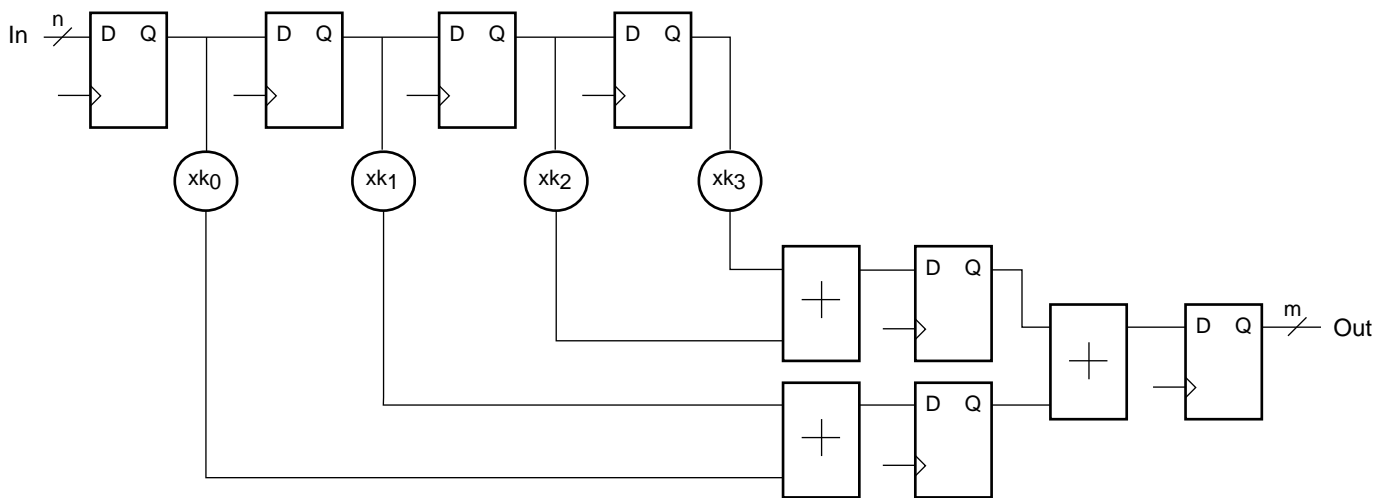
X219_02_091800

Figure 2: **FIR Filter Structure Employing Tree of Pipelined Adders**

An alternate implementation structure called the Transposed Form FIR filter is shown in Figure 3. Utilizing the same resources, data samples are applied in parallel to all the tap multipliers through pipeline registers. The input registers are not required, because high fan-out input signals can be handled by the Virtex device architecture. The products are applied to a cascaded chain of registered adders, combining the effect of accumulators and registers. The order of tap coefficients must be reversed with the first tap closest to the output. This structure allows expansion of the number of taps required in a filter, since each "tap module" is identical. Since the structure is uniform, a single component can be designed and instantiated as many times as required by the number of taps.
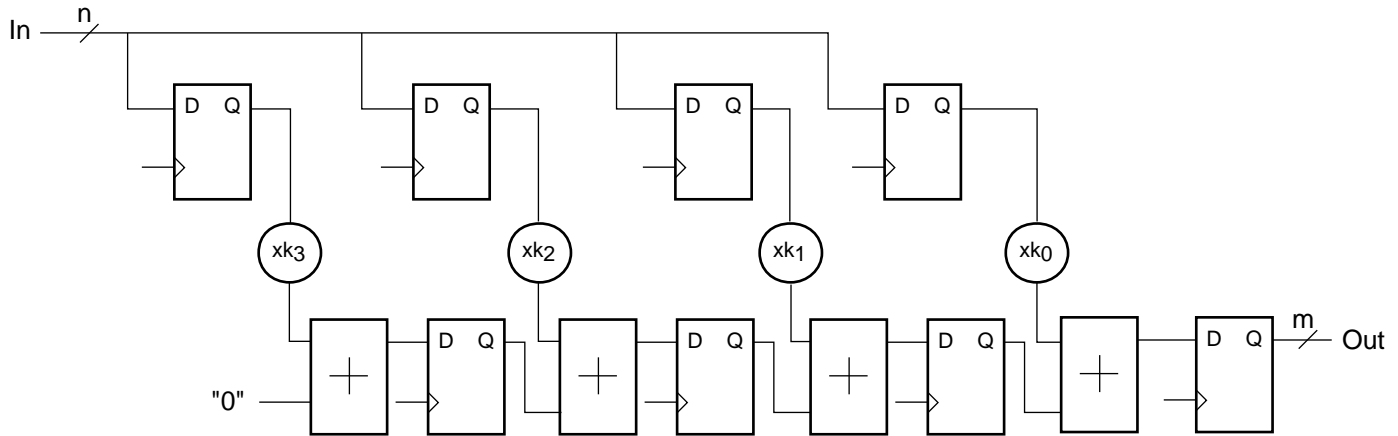
Figure 3: **Transposed Form FIR Filters Employing Cascaded Pipelined Adders**

FIR vs. Transposed Form FIR

Both FIR and Transposed Form FIR filters have trade-offs and limitations. It is up to the designer to choose the style most appropriate to the application. For an 8-tap, 16-bit filter, the device utilization and performance obtained were nearly identical. In general, a smaller filter profits from the traditional approach, while a larger filter benefits from the Transposed Form FIR approach. This argument becomes more obvious when very large filters are implemented across multiple devices. The cascadable nature of the tap-slice modules allows for easy interdevice connections. The input-to-output latency is reduced with fully pipelined Transposed Form FIR filters. The filter selection also depends on the type of coefficients (symmetric or asymmetric). In symmetric systems, coefficients occur in pairs.

In Transposed Form FIR filters, multipliers can be completely avoided if the coefficients can be tuned to powers of two (2^n) or values that are close to the powers of two ($2^3 + 1 = 9$.) In such cases, the multiplication can be achieved by shifting and adding.

Transposed Form Filter Design

In traditional DSPs, the FIR filters are implemented in dedicated hardware without any parallelism, thus limiting the sample rate. Virtex FPGAs have abundant hardware resources to facilitate full parallelism (each TAP has a dedicated multiplier and adder). The easiest way to implement a multiplier in HDL is to have it inferred automatically by the synthesis tools. The multiplier inferred by the synthesis tool is combinatorial logic. These multipliers occupy a large area, and resource sharing is not possible in this implementation. For multiplier performance improvement, the features of the filter have to be carefully studied. The efficiency of the multiplier determines the overall performance of the filter. Hence, the multiplier must be implemented for the best possible performance.

A comparative study of 14 x 16 full multipliers and multicycle KCM multipliers in a Virtex device reveals that a full multiplier takes 139 slices, while a KCM multiplier takes just 47 slices.

The reference design is an 8-tap filter based on 16-bit input samples and 14-bit signed coefficients. The basic building blocks of the filter are KCMs, Adders, Registers, and a delay-locked loop.

Constant Coefficient Multiplier (KCM)

In a fully parallel implementation of a filter, each tap has a dedicated multiplier. The tap data is an input of this multiplier, the other a constant coefficient. Since one input is a constant, these multipliers are called KCMs. KCMs are efficiently implemented by storing pre-computed partial products of the fixed coefficient, thereby reducing the logic required as compared to traditional two-variable multipliers. As a result, better performance can be achieved. In Xilinx FPGAs, these partial products can be stored in ROMs using the distributed memory.

The 16-bit input sample is separated into four 4-bit nibbles. Each nibble acts as an input to the ROM in different cycles. These ROMs store the product of the constant coefficient k, and a factor with variable values that change from 0 through 15. The ROM contents are 0 x k, 1 x k, 2 x k, 3 x k, ..., 15 x k. The word size in the ROM is:

(4-bit input nibble) x (14-bit coefficient) = 18 bits (ROM word size)

Essentially this ROM functions as a times table of the constant coefficient, k. In this reference design, the value read from this ROM based on its 4-bit input is added to another partial product stored in an adjacent ROM. As a result, KCMs are less than one-third the size of full multipliers. A KCM block diagram is shown in Figure 4.
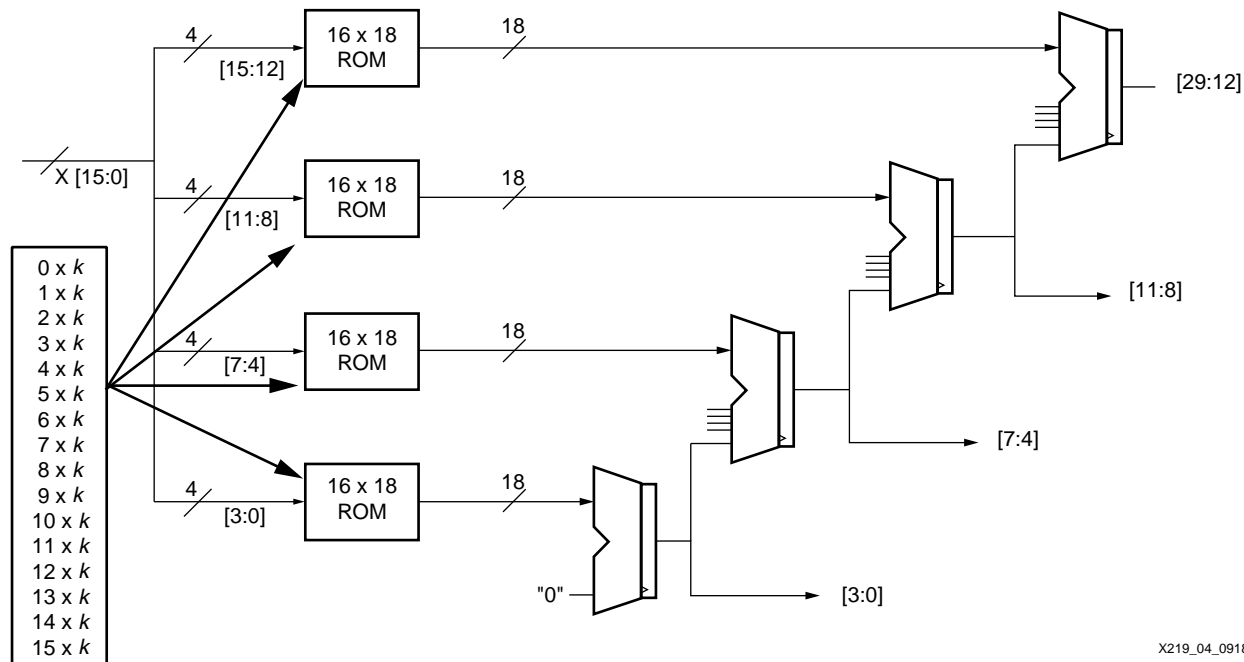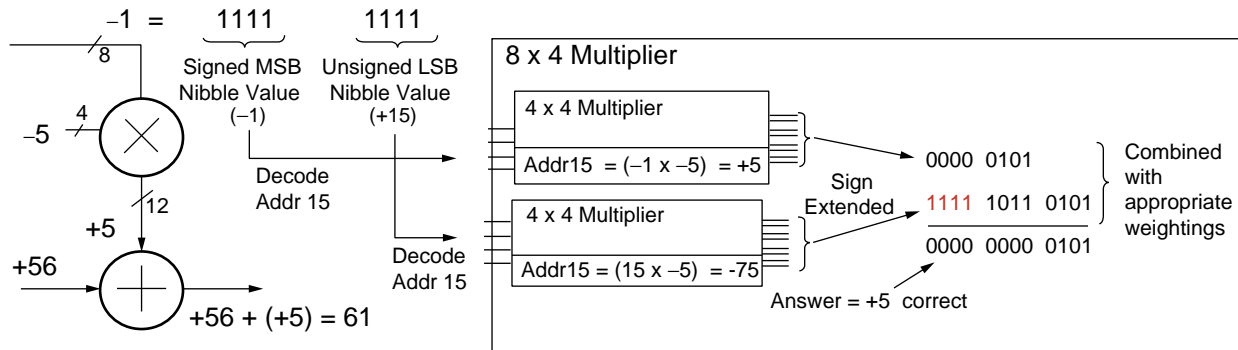

Figure 4: **KCM Block Diagram**

A KCM would be ideal for unsigned inputs and coefficients. There are a couple of options for handling signed numbers. The first approach is to implement two ROM tables, one for the signed MSB nibble and the other for the LSB nibbles. This approach requires two separate ROM tables per tap, as shown in Figure 5. This is not an optimal solution.
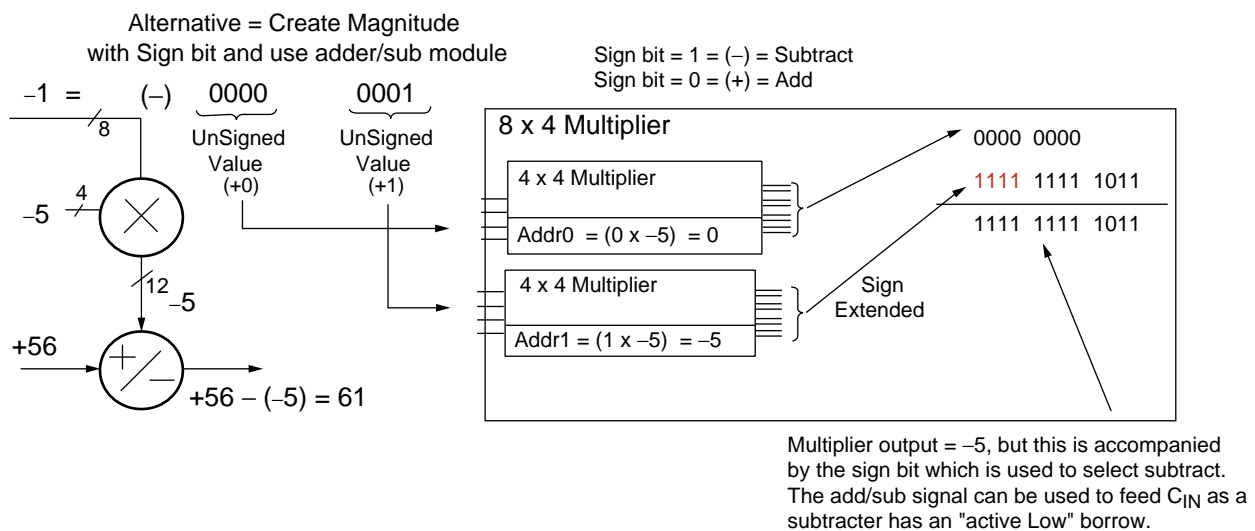
The second approach, shown in Figure 6, is to convert the signed sample input data into an unsigned magnitude word and a sign bit, using a 2's-complement module. When a negative word is detected, it is complemented, and the magnitude decodes a value from the same ROM table that a non-negative data would use. The multiplier output is a negative value, which is incorrect; however, the accompanying sign bit causes a subtract operation in the ADD/SUB module resulting in the correct sign and magnitude.

In order to handle signed inputs and coefficients, a 2's-complement component is used to convert negative numbers to positive. After all the operations, the final result is made positive or negative depending on the sign of the input and coefficient.

Figure 5: **KCM Implementation with Two ROM Tables**

X219_05_091800

Multiplier output = −5, but this is accompanied by the sign bit which is used to select subtract. The add/sub signal can be used to feed C_{IN} as a subtracter has an "active Low" borrow.

x219_06_091800

Figure 6: **KCM Implementation with Add/Subtract Module**

The third approach, as implemented in the reference design, uses three 2's-complement modules to handle both signed inputs and coefficients. This is used to avoid signed multiplication and addition.

The operation of a KCM multiplier implemented using a ROM is explained with the following example:

16-bit input:	`0001 0010 0000 0100`	(Decimal equivalent 4612)
14-bit coefficient:	`00 0000 0000 0010`	(Decimal equivalent 2)

The 16-bit input is separated into four 4-bit nibbles: "0001", "0010", "0000", and "0100". All fifteen coefficient factors, 0 x 2, 1 x 2, 2 x 2, …15 x 2 are stored with an 18-bit (14-bit x 4-bit) word size in the ROM. Each 4-bit nibble of the 16-bit input acts as an address to the ROM. The corresponding ROM content at this address is read.

First partial product	=	00	0000	0000	0000	1000 (ROM contents at address "0100")
Second partial product	=	00	0000	0000	0000	0000 (ROM contents at address "0000")
Third partial product	=	00	0000	0000	0000	0100 (ROM contents at address "0010")
Fourth partial product	=	00	0000	0000	0000	0010 (ROM contents at address "0001")

All the partial products are then added after shifting them appropriately (shown below):

```
                      00  0000  0000  0000  1000      First partial product
                 00   0000  0000  0000  0000  0000    Second partial product
            00   0000  0000  0000  0100  0000  0000   Third partial product
+  00  0000  0000  0000  0010  0000  0000  0000        Fourth partial product
    00  0000  0000  0000  0010  0100  0000  1000       (Decimal equivalent 9224)
```

Pipelining and resource sharing of adders can further enhance the performance of KCM multipliers. An enhanced multicycle KCM schematic of the reference design is shown in Figure 7. The input sample arrives at clock frequency f_1, while all the internal operations of the KCM can be performed at a much higher frequency of f_2 (4 x f_1). Four muxes are used to select 4-bit input nibbles. A 2-bit counter clock operating at f_2 frequency acts as the select signal for these muxes. For every 4-bit input nibble, a corresponding value is read from the ROM and corresponding partial products are added after taking care of the required shift operations.

ROM Implementation

In HDL, there are two approaches to infer ROMs using the function generators or Look-Up Tables (LUTs) in Xilinx FPGAs. One approach is to use the case statement. With this approach, the code would require as many case statements as the number of ROMs required in the filter design, and each case statement would have to specify all 2^n possibilities, n being the number of address bits. Although this can make the code lengthy and tedious, an advantage is the fact that the coefficients can be changed without an impact to the utilization or performance of the filter design.

The reference design **xapp219.zip** uses array declarations. This second approach results in a concise code that is easily editable, as well as a more optimal use of resources compared to the first approach. As a result, any changes in the coefficient values would cause the utilization, and thereby the performance, to be slightly changed.
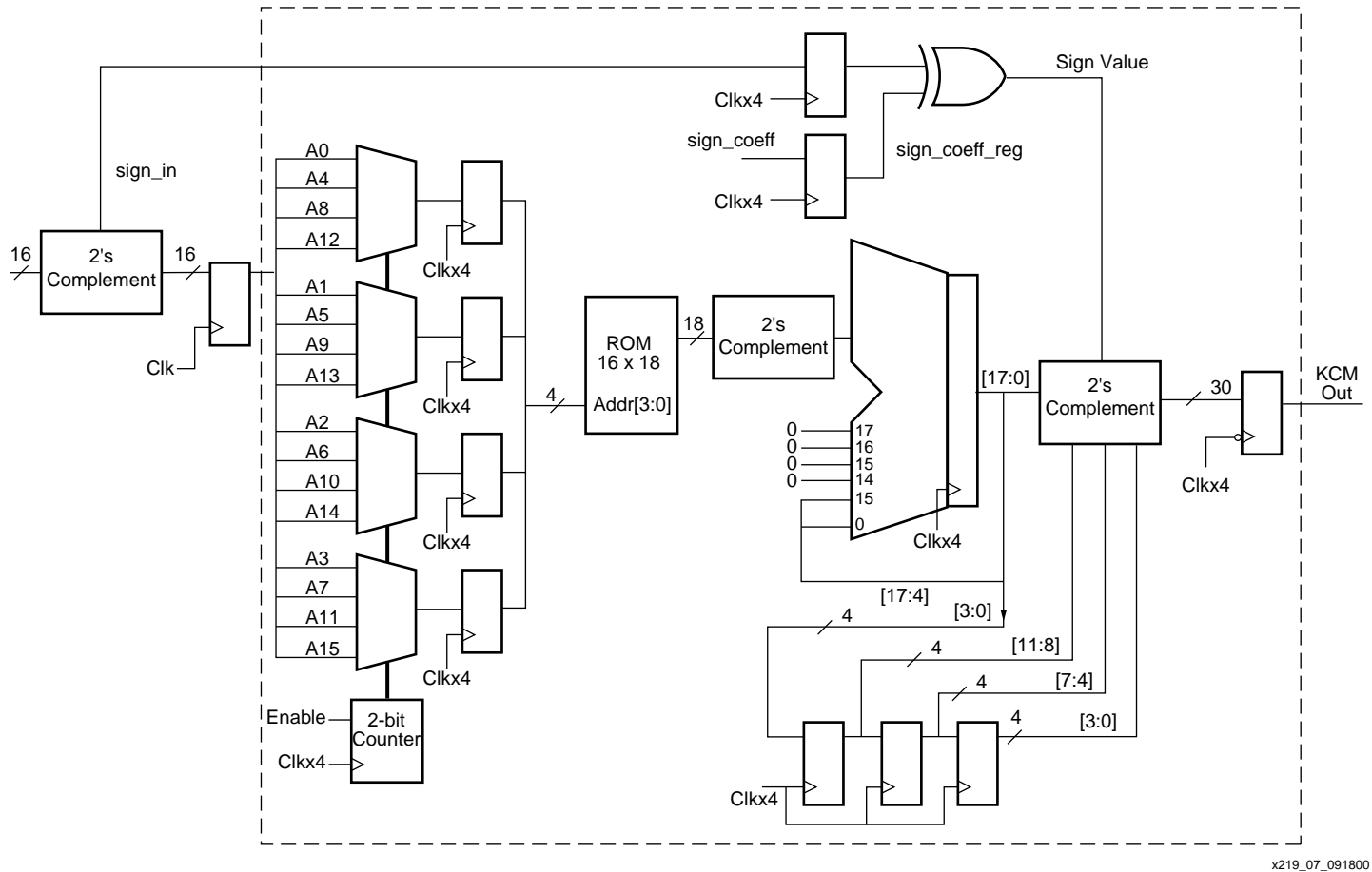
Figure 7: **Multicycle KCM implementation**

x219_07_091800

Delay-Locked Loops (DLL)

Virtex devices have built-in DLLs which are highly efficient for clock deskew and clock manipulation. As discussed earlier, multicycle KCM uses two clocks of frequency f_1 and f_2, where $f_1 = f_2 / 4$ or $f_2 = 4 \times f_1$.
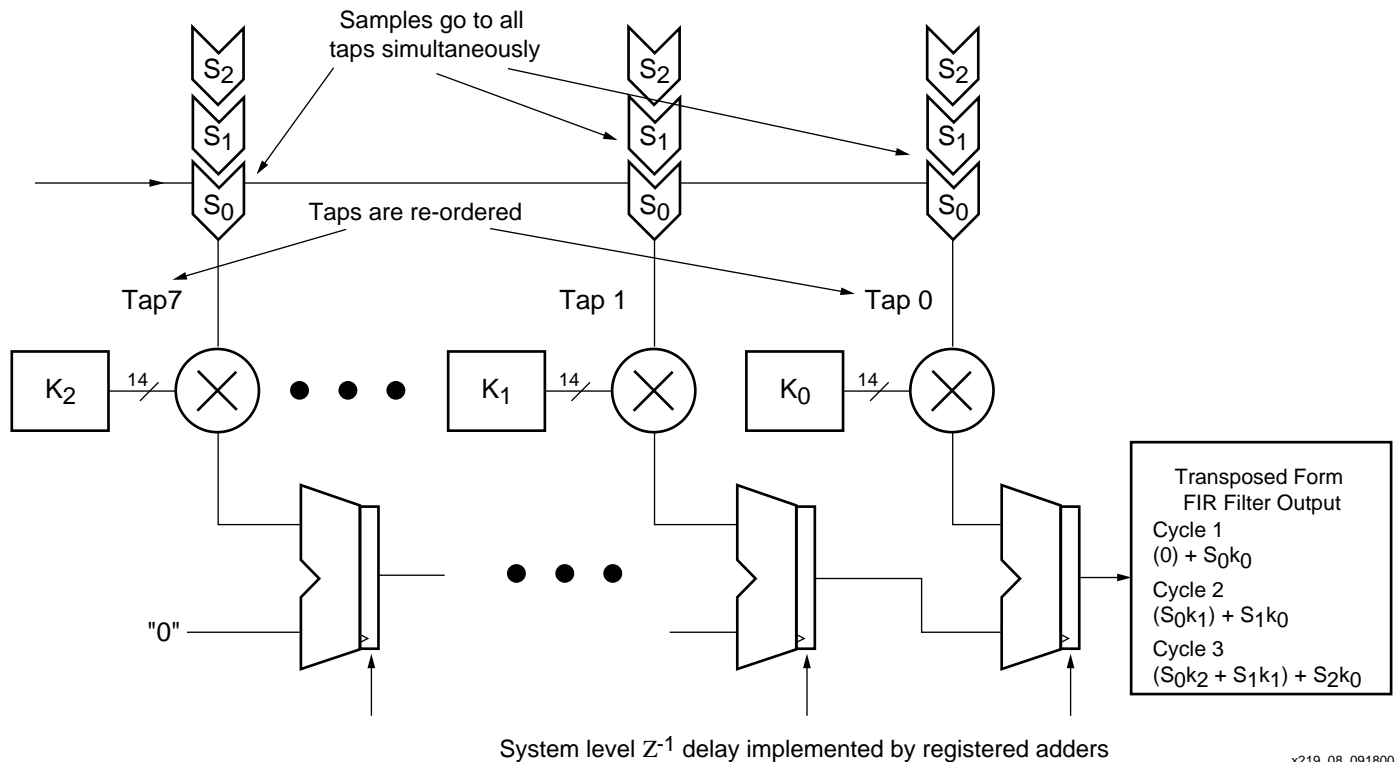
There are two approaches to generate these two clocks using DLLs:

1. One DLL with an input frequency of f_2 can be used to generate the frequency $f_1 = f_2 / 4$, using the clock division capability of the DLL.

2. Two DLLs can be cascaded together to obtain $4 \times f_1 = f_2$. The clock with frequency f_1 would be the input to the first DLL, and its output $2 \times f_1$ would be the input to the second DLL. Please refer to **XAPP132** for DLL details.

The given reference design is based on the first option using a single DLL. In this case, the data streams at $f_2/4$ and the KCM operates at f_2. Alternatively, the second option can be used. The selection must be based on the external clock and the input sample rate. The clock output from the DLL is only valid after its lock signal is enabled. This lock signal is also used in this design to enable the 2-bit counter in the multicycle KCM.

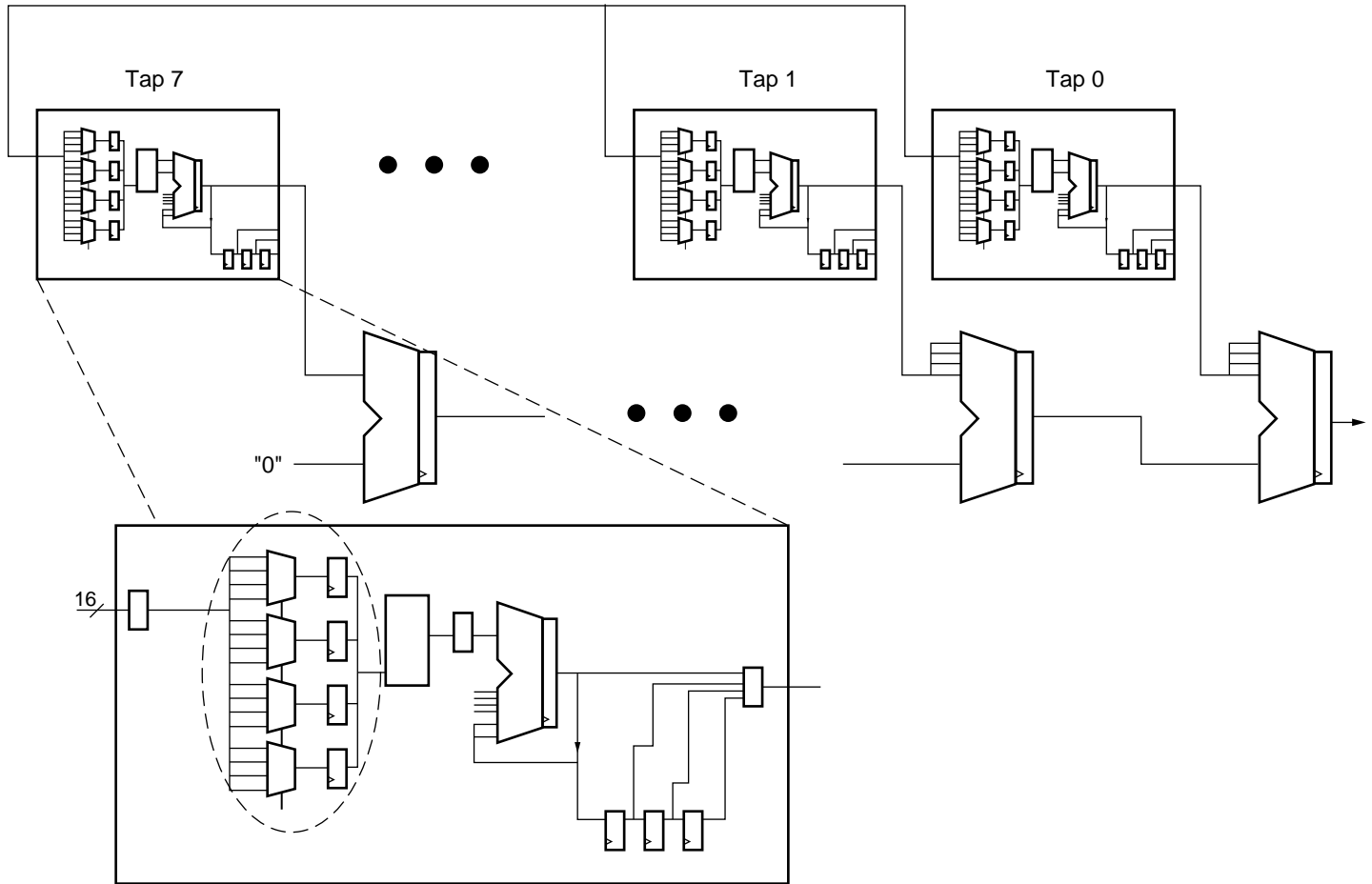# Transposed Form FIR Filter Implementation

The complete filter is built by integrating the KCM multipliers, delay elements, and adders. The transposed form FIR filter block diagram is shown in Figure 8, and a more detailed schematic design with eight taps is shown in Figure 9. The precision of the filter is preserved at every tap of the filter. The MSB bit from the corresponding KCM multiplier is sign-extended by one bit to accommodate any sign overflow.

Samples go to all taps simultaneously

Taps are re-ordered

Transposed Form FIR Filter Output

Cycle 1
$(0) + S_0 k_0$

Cycle 2
$(S_0 k_1) + S_1 k_0$

Cycle 3
$(S_0 k_2 + S_1 k_1) + S_2 k_0$

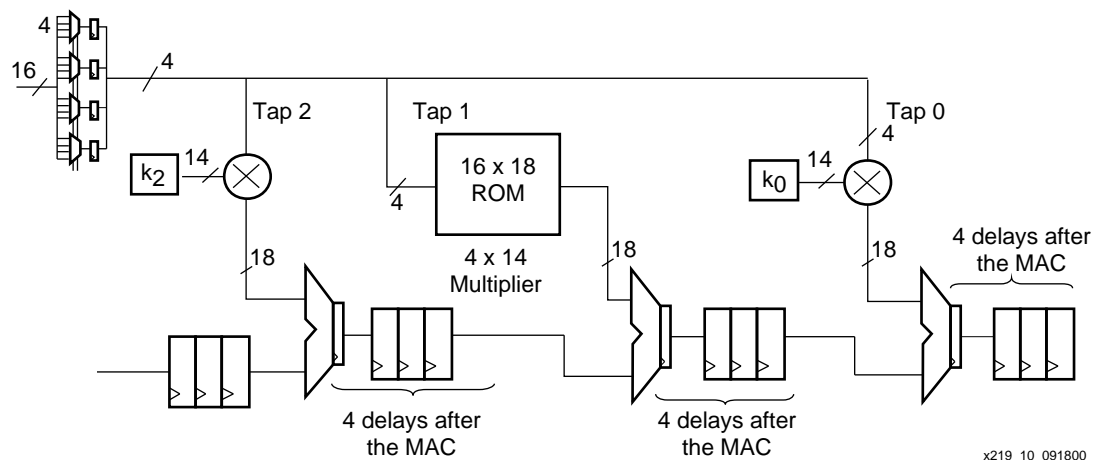System level Z^{-1} delay implemented by registered adders

x219_08_091800

Figure 8: **Transposed Form FIR Filter Block Diagram**

The reference design implements the structural design shown in Figure 9. This design can be further optimized by sharing the common resource of all the KCM multipliers. The 4-to-1 muxes in the KCM multipliers are extracted and the adders are merged to optimize resources, as shown in Figure 10. As before, each tap multiplier is implemented by a 16 x 18 ROM. Each tap produces four 18-bit partial products at 4x clock frequency, rather than one 30-bit result in one clock frequency. Four partial products need to be stored between the adder chain taps to guarantee that only partial products with the same weighting are added together.

Tap 7

Tap 1

Tap 0

• • •

• • •

"0"

16

x219_09_091800

Figure 9: **Transposed Form FIR Filter with Eight Taps**

4

16

4

Tap 2

Tap 1

Tap 0

k₂

14

16 x 18
ROM

k₀

14

4

4 x 14
Multiplier

18

18

18

4 delays after
the MAC

4 delays after
the MAC

4 delays after
the MAC

x219_10_091800

Figure 10: **Optimized Transposed Form FIR Filter**

VHDL Reference Design

The reference design provided with this application note is ideal for asymmetric coefficients. This design was implemented structurally by instantiating KCMs and a DLL. All the KCMs are identical in the filter, with different ROM contents for each tap. Instead of defining four KCMs, a single KCM is defined with an option of selecting different ROMs for each tap. The constant coefficients for eight taps are declared in the package. This makes it easier to change the constants. Figure 11 shows simulation waveforms of the reference design. The input is registered at the slower clock edge. The KCM output is obtained after six clock cycles of the faster clocks (f_2), and the final filter output is obtained after a two-clock cycle latency of the slower clock, ($f_1 = f_2/4$.)
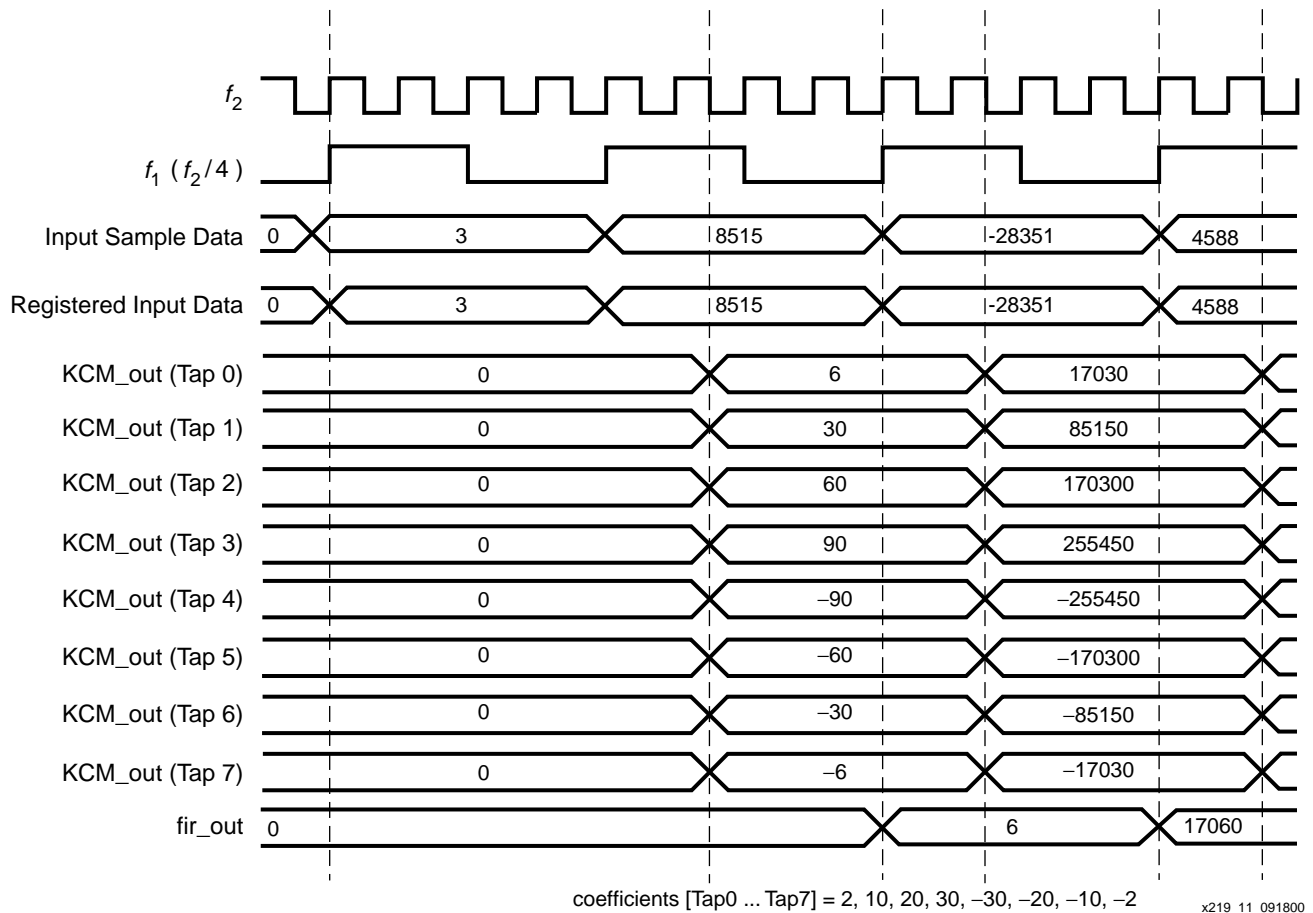

Figure 11: **Simulation Waveform**

Synthesis Tool Results

The reference design was synthesized using different commercial synthesis tools. The results are presented in Table 1. The filter has 8 taps, 16-bit inputs, 14-bit signed coefficients, and was targeted to one of the smaller members of the Virtex family, XCV100-TQ144. The input data samples at one-quarter of the clock frequency in Table 1.

Table 1: **Transposed Form FIR Filter**

Synthesis Tool	Number of Slices	Number of 4-input LUTs	Number of Slice Registers	Clock Frequency (Timing Report)
Synplify 6.0	584	931	755	70.78 MHz
FPGA Express 3.4	654	977	807	72.15 MHz
Exemplar 2000.1a	703	792	677	56.0 MHz

MATLAB FIR Filter Implementation

Three different FIR filters were implemented using the MATLAB tool to prove that the impulse responses for the traditional-form FIR and the Transposed Form FIR filter were identical. Note that the coefficients for these filters are symmetric. The reference design provided with this application note does not realize an optimal implementation for symmetric coefficients.

Ideal FIR Filter

The Ideal FIR filter implemented in MATLAB is a full-precision floating point implementation using the Equiripple FIR (Remez Algorithm).

Ideal FIR filter coefficients:

$H(z) = [\ 0.0112\ -0.1308\ 0.0390\ 0.5236\ 0.5236\ 0.0390\ -0.1308\ 0.0112\ ]$

where $F_{3dB} = 4$ MHz, $F_{20dB} = 6$ MHz, $F_S = 16$ MHz

The impulse response for this filter is shown in Figure 12.
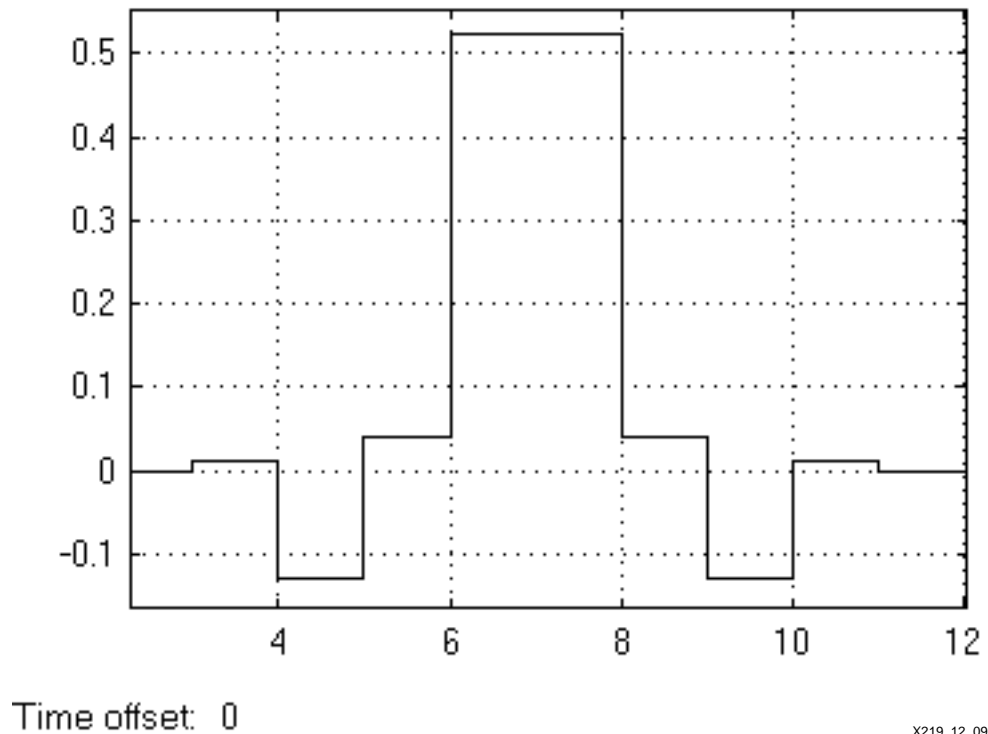
X219_12_091800

Figure 12: **Ideal FIR Filter Impulse Response**
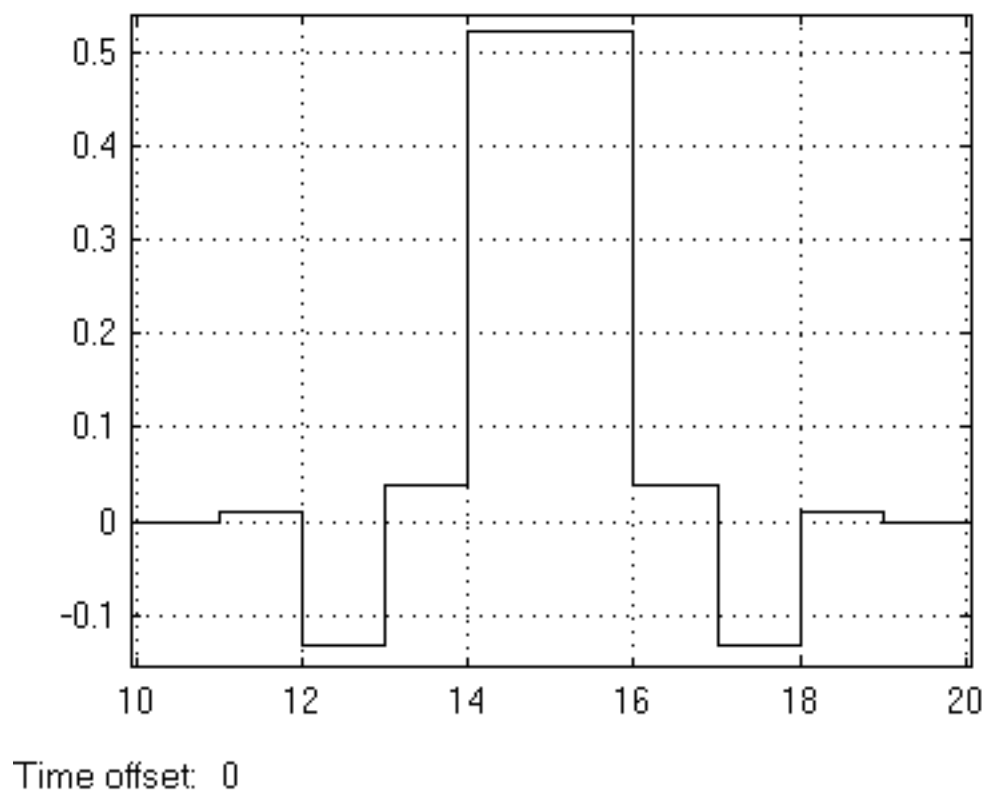
Traditional FIR Filter

The fixed-point, traditional-form FIR filter was implemented using the Xilinx System Generator tool, which is a simulink blockset. It is a signed, single-rate CoreGen filter.
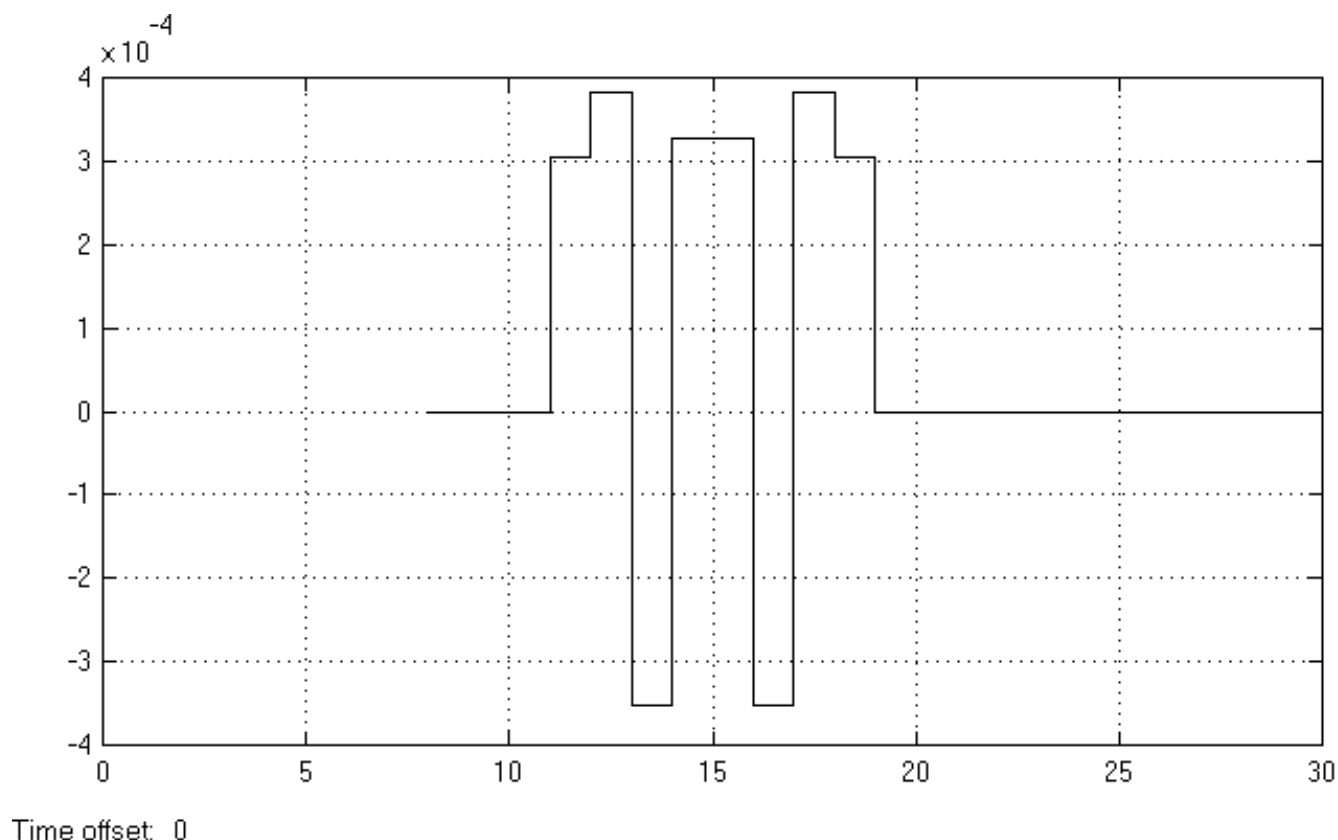
The fixed-point FIR filter coefficients:

$$H(z) = [\ 0.112\ \ -1.308\ \ 0.390\ \ 5.236\ \ 5.236\ \ 0.390\ \ -1.308\ \ 0.112\ ]$$

Quantization is 14 bits with the binary point at the 10th bit. The impulse response for this filter is shown in Figure 13. The quantization error for this filter is shown in Figure 14.

Time offset: 0

X219_13_091400

Figure 13: **Traditional FIR Filter Impulse Response**

X219_14_091400

Figure 14: **Traditional FIR Filter Quantization Error**
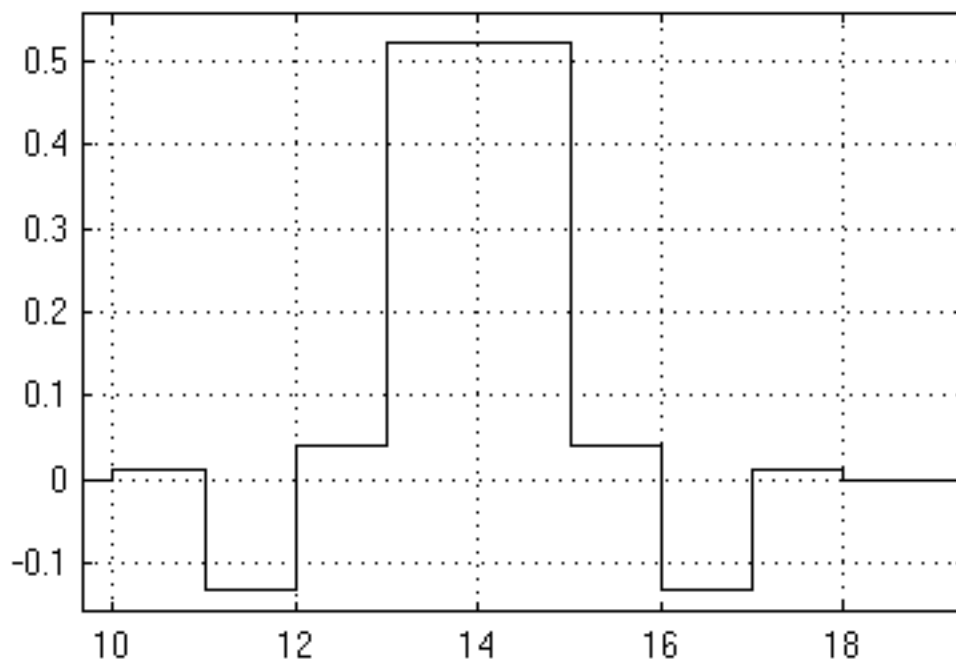
Transposed Form FIR Filter

The fixed-point Transposed Form FIR filter was also built with the Xilinx System Generator tool using the math primitives in the Xilinx blockset. It is a signed, single-rate filter.

Fixed-point Transposed Form FIR filter coefficients:

$$H(z) \ = \ [\ 0.1123 \ -1.308 \ 0.3896 \ 5.236 \ 5.236 \ 0.3896 \ -1.308 \ 0.1123 \ ]$$

The slight discrepancy in the coefficients is due to quantizing these coefficients. As can be seen, this has no impact on the quantization error or impulse response for this filter. Quantization is 14 bits with the binary point at the 10th bit. The impulse response for this filter is shown in Figure 15. The quantization error for this filter is shown in Figure 16.

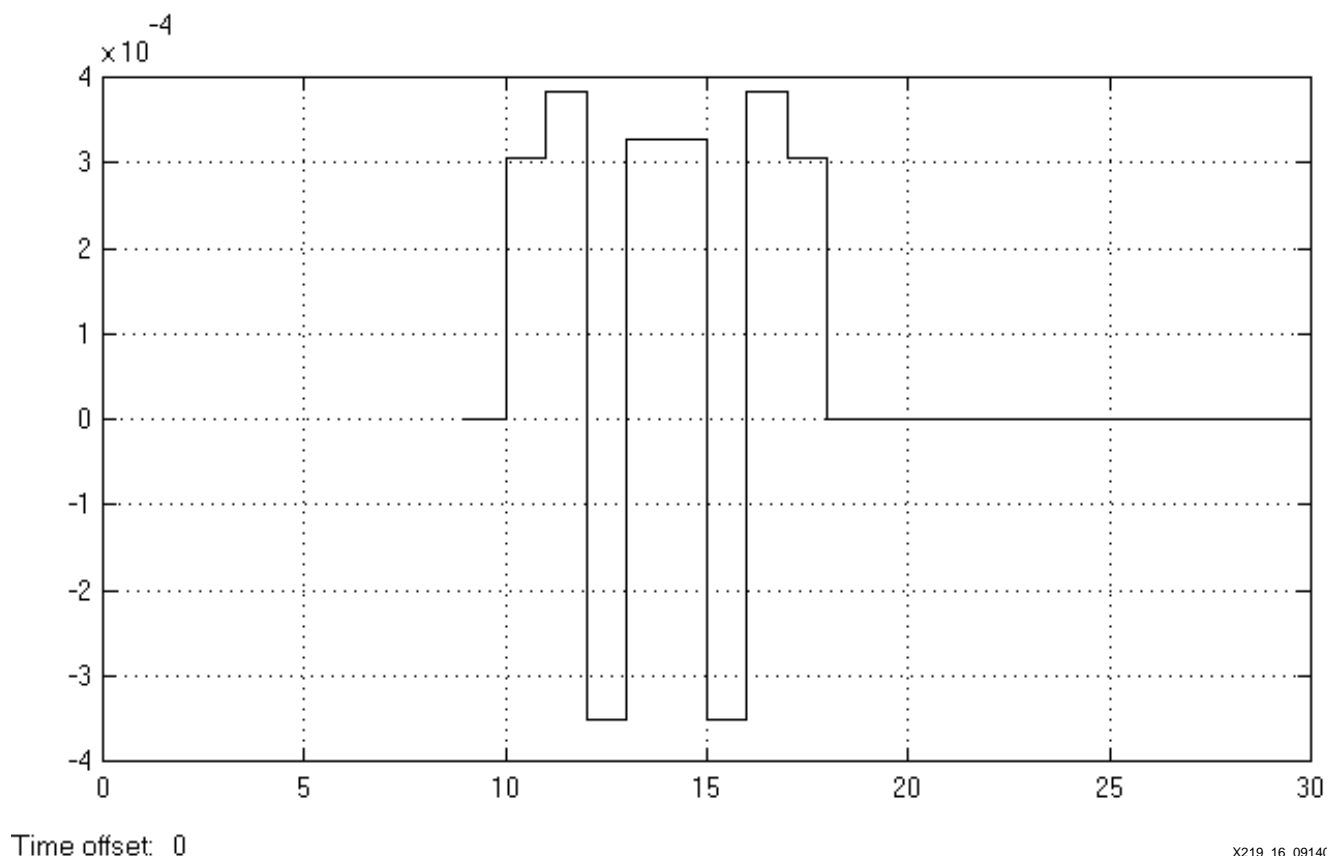*Figure 15:* **Transposed Form FIR Filter Impulse Response**

X219_16_091400

Figure 16: **Transposed Form FIR Filter Quantization Error**

Conclusion

FIR filters are commonly used in DSP applications. The FIR filters implemented in Virtex and Spartan-II FPGAs provide the designer tremendous flexibility in terms of the number of filter taps and changes in existing coefficients. It may be necessary to "tune" a filter in an existing system, or to have multiple filter settings. The reconfigurability of FPGAs is exploited by making necessary coefficient changes in the synthesizable HDL code. In a KCM, the coefficients are constant; therefore, they are stored as partial products in ROM elements that are implemented in function generators or LUTs. This implementation permits any coefficient values to be programmed into the same logic, thereby reducing the impact on place and route or performance. The HDL reference design provided with this application note is easily modified to achieve specific requirements.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
9/21/00	1.0	Initial Xilinx release.

www.xilinx.com XAPP219 (v1.0) September 21, 2000
1-800-255-7778