Implementing FIR Filters
in FLEX Devices

Introduction

The finite impulse response (FIR) filter is used in many digital signal processing (DSP) systems to perform signal preconditioning, anti-aliasing, band selection, decimation/interpolation, low-pass filtering, and video convolution functions. Only a limited selection of off-the-shelf FIR filter circuits is available; these circuits often limit system performance. Therefore, programmable logic devices (PLDs) are an ideal choice for implementing FIR filters.

Altera FLEX devices, including the FLEX 10K and FLEX 8000 families, are flexible, high-performance devices that can easily implement FIR filters. For example, you can use a FLEX device for one or more critical filtering functions in a DSP microprocessor-based application, freeing the DSP processor to perform the lower-bit-rate, algorithmically complex operations. A DSP microprocessor can implement an 8-tap FIR filter at 5 million samples per second (MSPS), while an off-the-shelf FIR filter circuit can deliver 30 MSPS. In contrast, FLEX devices can implement the same filter at over 100 MSPS.

This application note describes how to map the mathematical operations of the FIR filter into the FLEX architecture and compares this implementation to a hard-wired design. Implementation details—including performance/device resource tradeoffs through serialization, pipelining, and precision—are also discussed.

Conventions

Parentheses are used to show which tap or coefficient is being referenced. The variable $x(n)$ refers to the nth input sample, e.g., $x(6)$ refers to tap number 6. The variable $h(n)$ refers to the nth coefficient of the FIR filter, e.g., $h(4)$ refers to coefficient 4. Subscripts are used to distinguish which bit in the binary word is being referenced, e.g., $x(6)_1$ denotes the least significant bit (LSB) in the sixth tap. All numbering begins with 1 rather than 0.
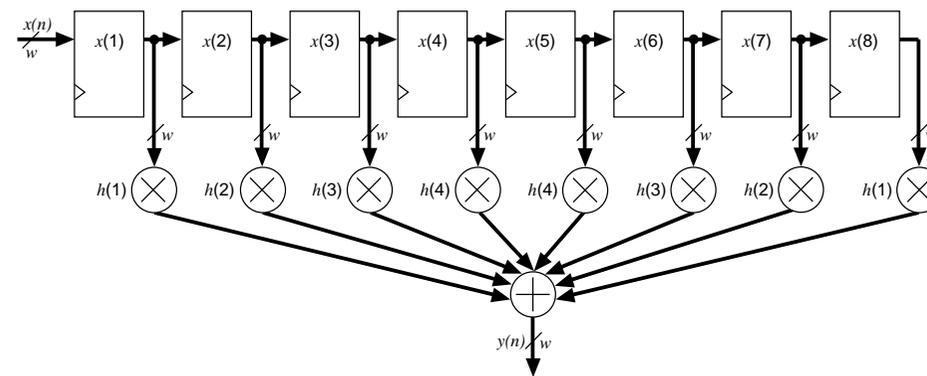
FIR Filter Architecture

This section describes a conventional FIR filter design and how the design can be optimized in FLEX devices.

Conventional FIR Filter Design

Figure 1 shows a conventional 8-tap FIR filter architecture. This filter has eight 8-bit registers arranged in a shift register configuration.

Figure 1. Conventional FIR Application

The output of each register is called a tap and is represented by $x(n)$, where n is the tap number. Each tap is multiplied by a coefficient $h(n)$ and then all the products are summed. The equation for this filter is:

$$y(n) \ = \ \sum_{n=1}^{8} x(n)h(n)$$

For a linear phase response FIR filter, the coefficients are symmetric around the center values. This symmetry allows the symmetric taps to be added together before they are multiplied by the coefficients. See Figure 2. Taking advantage of the symmetry lowers the number of multiplies from eight to four, which reduces the circuitry required to implement the filter.

Figure 2. Adding Symmetric Taps Before Multiplication
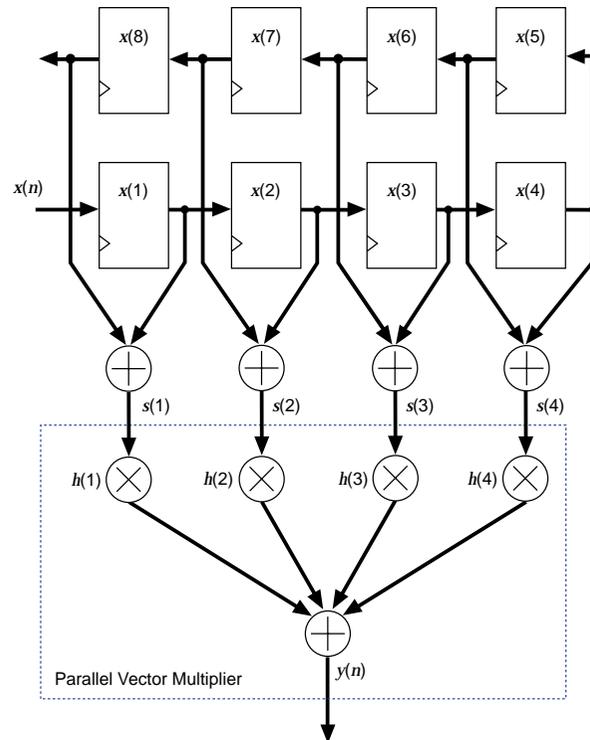
Optimizing the FIR Filter with FLEX Devices

The multiplication and addition for the vector multiplier shown in Figure 2 can be optimized by using look-up tables (LUTs) in a FLEX device. The equation for the vector multiplier is:

$$y = [s(1) \times h(1)] + [s(2) \times h(2)] + [s(3) \times h(3)] + [s(4) \times h(4)]$$

The multiplication and addition can be performed in parallel using LUTs. The following example uses 2-bit positive integers.

```
h(1) = 01, h(2) = 11, h(3) = 10, h(4) = 11
s(1) = 11, s(2) = 00, s(3) = 10, s(4) = 01
```

The multiplication and addition for the vector multiplier are shown in Figure 3.

Figure 3. Vector Multiplier Multiply & Add

The partial products P1(n) and P2(n) can be added either horizontally or vertically without changing the result, which is 1010. Because each component of $h(n)$ is constant for any fixed-coefficient FIR filter, you can use a FLEX LUT to implement the vector multiplier.

In Figure 3, the four digits shown in bold, blue text are the LSBs of each $s(n)$, and are represented by $s(n)_1$. Each partial product P1(n)—highlighted with the blue bar—is either 00 or the corresponding value of the multiplicand's $h(n)$. The sum of all partial products P1(n) is P1 (in this case 100). Because $s(n)_1$ for the 4 multipliers uniquely determines the value for P1, there are only 16 possible values for P1. See Table 1.

Table 1. LSB of Each s(n) for Each Partial Product (P1)

| $s(n)_1$ *Note (1)*	P1	Result
0000	0	00 + 00 + 00 + 00 = 0000
0001	h(1)	00 + 00 + 00 + 01 = 0001
0010	h(2)	00 + 00 + 11 + 00 = 0011
0011	h(2) + h(1)	00 + 00 + 11 + 01 = 0100
0100	h(3)	00 + 10 + 00 + 00 = 0010
0101	h(3) + h(1)	00 + 10 + 00 + 01 = 0011
0110	h(3) + h(2)	00 + 10 + 11 + 00 = 0101
0111	h(3) + h(2) + h(1)	00 + 10 + 11 + 01 = 0110
1000	h(4)	11 + 00 + 00 + 00 = 0011
1001	h(4) + h(1)	11 + 00 + 00 + 01 = 0100
1010	h(4) + h(2)	11 + 00 + 11 + 00 = 0110
1011	h(4) + h(2) + h(1)	11 + 00 + 11 + 01 = 0111
1100	h(4) + h(3)	11 + 10 + 00 + 00 = 0101
1101	h(4) + h(3) + h(1)	11 + 10 + 00 + 01 = 0110
1110	h(4) + h(3) + h(2)	11 + 10 + 11 + 00 = 1000
1111	h(4) + h(3) + h(2) + h(1)	11 + 10 + 11 + 01 = 1001

Note:
(1) $s(n)_1$ refers to the LSB of each multiplier s(n). In Figure 3, the LSBs are denoted by the four digits shown in bold, blue text.

The partial product P2 can be calculated in the same manner, except the result must be shifted left by one bit before adding P1 and P2. In this example, the result is four bits wide. Therefore, the adders must be four bits wide.

The partial products (P1 and P2) can be calculated by 4-input LUTs in a FLEX device. All computations occur in parallel. The partial products can be fed into a tree of adders to calculate the final product called *y(n)* as shown in Figure 4.

Figure 4. Four 2-Bit Input Vector Multiplier

This multiplier performs the function [s(4) × h(4)] + [s(3) × h(3)] + [s(2) × h(2)] + [s(1) × h(1)]. Both LUTs consist of 4 FLEX 16 × 1 LUTs.
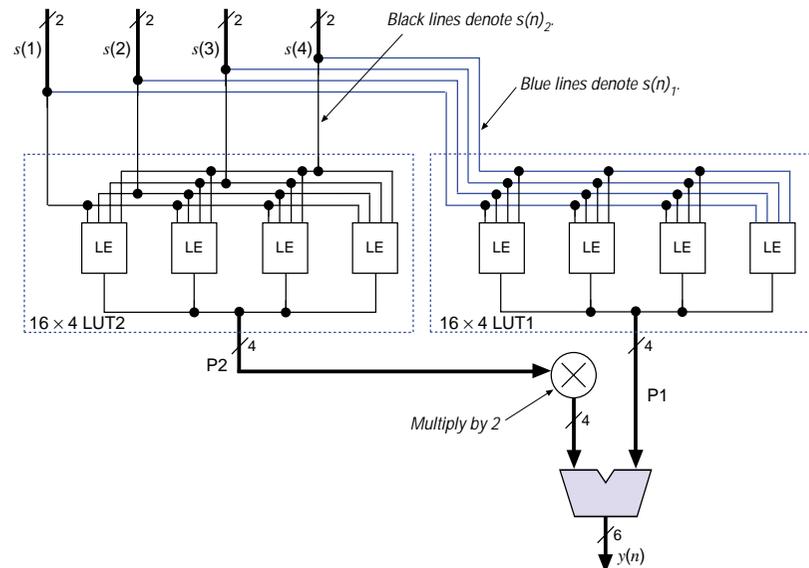
Only one adder is used in Figure 4 because the function has only two bits of precision. If more bits of precision are used, additional adders are required. See Figure 5. The vector multiplier can be used in a 7-bit input FIR filter. The eighth bit comes from adding the two 7-bit symmetric taps.

Figure 5. Four 8-Bit Input Vector Multiplier

The multipliers increase by a power of 2 for each level, which maintains the correct precision for each 8-bit input.
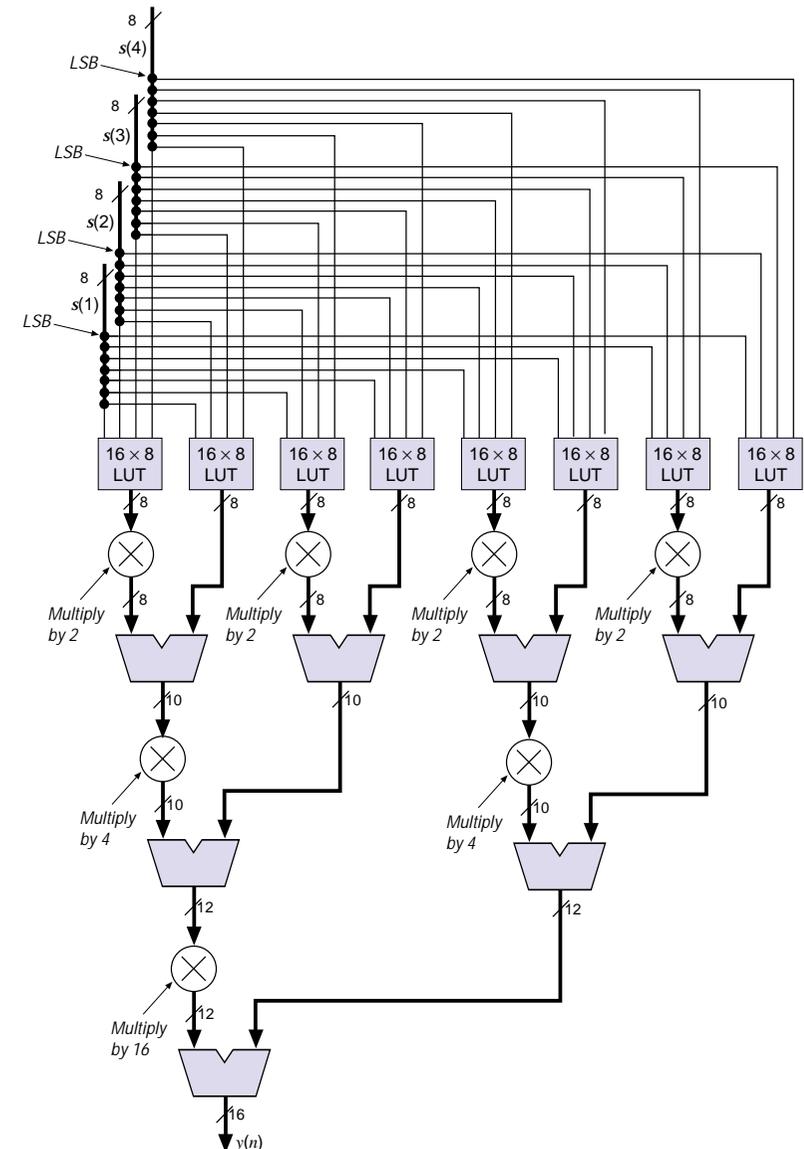
The filter in Figure 5 can be pipelined by placing registers at the outputs of each adder and LUT. Because FLEX devices have registers at the output of every logic cell, adding pipeline registers to the filter does not require additional device resources. Pipelining introduces a 5-Clock latency for this design.

Implementing FIR Filters

This section discusses the issues that must be resolved when implementing FIR filters in FLEX devices, including:

- Filter architecture
- Pipelining
- Scaling
- Symmetrical and antisymmetrical filters

Filter Architecture

The conventional FIR filter architecture described in the previous section is a fully parallel architecture, designed for maximum speed and size. However, you can create a smaller, slower filter by serializing the multiplications and additions. This section describes the size and speed tradeoffs of each architecture. Table 2 summarizes the FIR filter architectures.

Table 2. Comparison of FIR Filter Architectures in FLEX 8000A Devices

Filter Name	Input Precision (Bits)	Taps	Size (Logic Cells)	Device	Clock Rate (MHz)	Clock Cycles per Result	Speed (MSPS)	MIPS
Parallel	8	16	468	EPF8820A	101	1	101	1,616
Serial	8	16	272	EPF8452A	63	9	7.0	112

Go to the *FIR Filter Functional Specification* for more information on parallel and serial FIR filters.

Parallel FIR Filters

The conventional FIR filter described in "Conventional FIR Filter Design" is a fully parallel architecture. The output is a combinatorial function of the current and past data values. The LUT that corresponds to the MSB actually contains the two's complement of all the other LUTs to implement two's complement arithmetic. This filter can be pipelined for greater efficiency.

☞ For an explanation of two's complement arithmetic, refer to "Two's Complement Arithmetic" on page 22 of this application note.

Serial FIR Filters

Figure 6 shows a fully serial FIR filter. This architecture is similar to the fully parallel FIR filter, i.e., it uses the FLEX LUT to store the precomputed partial products P1, P2 ... Pn, where n = <*data width*> + 1. The serial filter in Figure 6 performs the same computation as the parallel filter, but it only processes one bit of the input data at a time. The serial filter first computes P1, which is a function of the four bits $s(1)_1$ through $s(4)_1$. On each successive cycle the serial filter computes the next partial product Pn from inputs $s(1)_n$ through $s(4)_n$. The partial products are summed in the scaling accumulator, which shifts the previous data right by one bit, i.e., during each cycle it divides the previous result by 2. This produces a result after <*data width*> + 1 Clock cycles because when the data passes though the symmetric tap adders (at the top of Figure 6) the data is <*data width*> + 1 bits wide (the fully parallel version has <*data width*> + 1 LUTs for the same reason). The serial FIR filter reuses the same LUT, rather than using extra circuitry.

Figure 6. Fully Serial FIR Filter Schematic
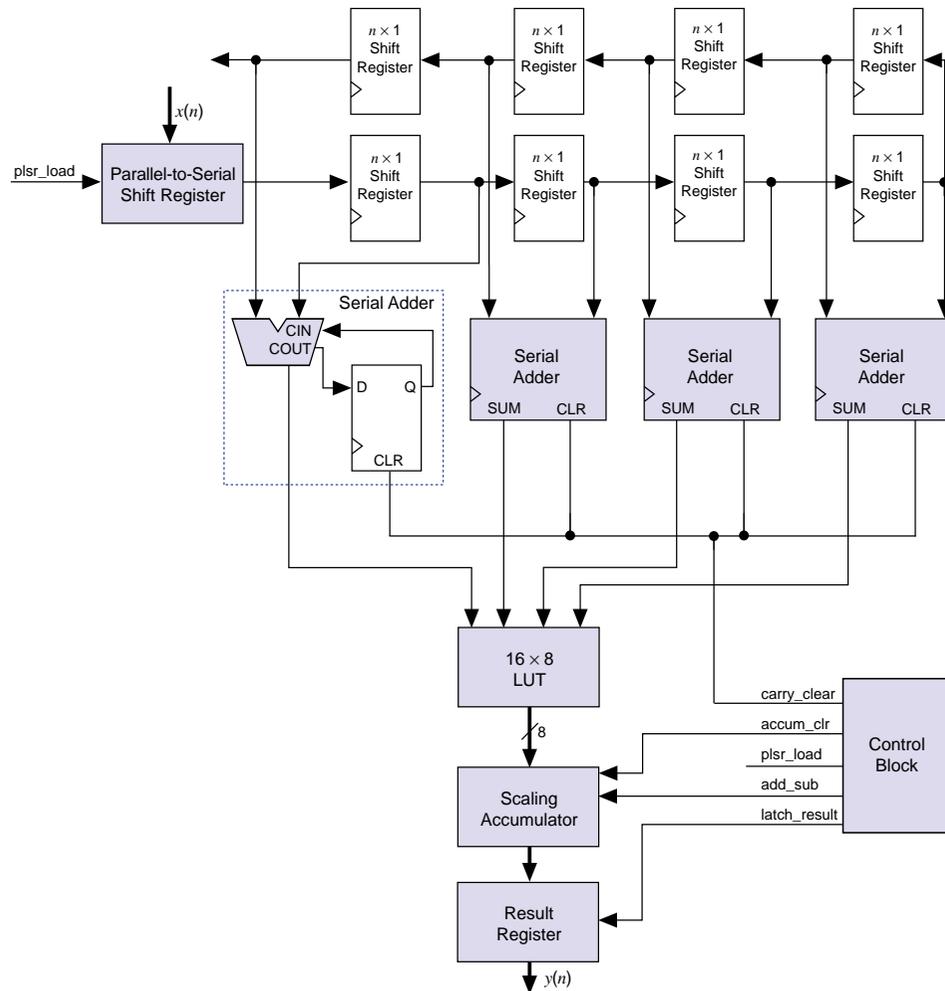
Because the serial filter contains one LUT, it can contain only one set of data. Therefore, the accumulator must perform a subtraction when calculating the MSB of the data, i.e., the accumulator must have an `add_sub` port. The controller deasserts the `add_sub` signal when the filter computes the MSB.

Serial/Parallel FIR Filters

The combined serial/parallel architecture has features from both the serial and parallel architectures. See Figure 7. The data shift registers at the top of Figure 7 are both serial and parallel. The parallelism in this case is two, i.e., two parallel computational chains exist. In contrast, the fully serial architecture has only one computational path. The serial/parallel filter requires only four Clock cycles to complete the computation. The filter in Figure 7 is a 7-bit filter because of possible carry into the eighth bit in the symmetric-tap adders. To obtain eight bits of precision on the incoming data, you must add another stage to each of the data shift registers.

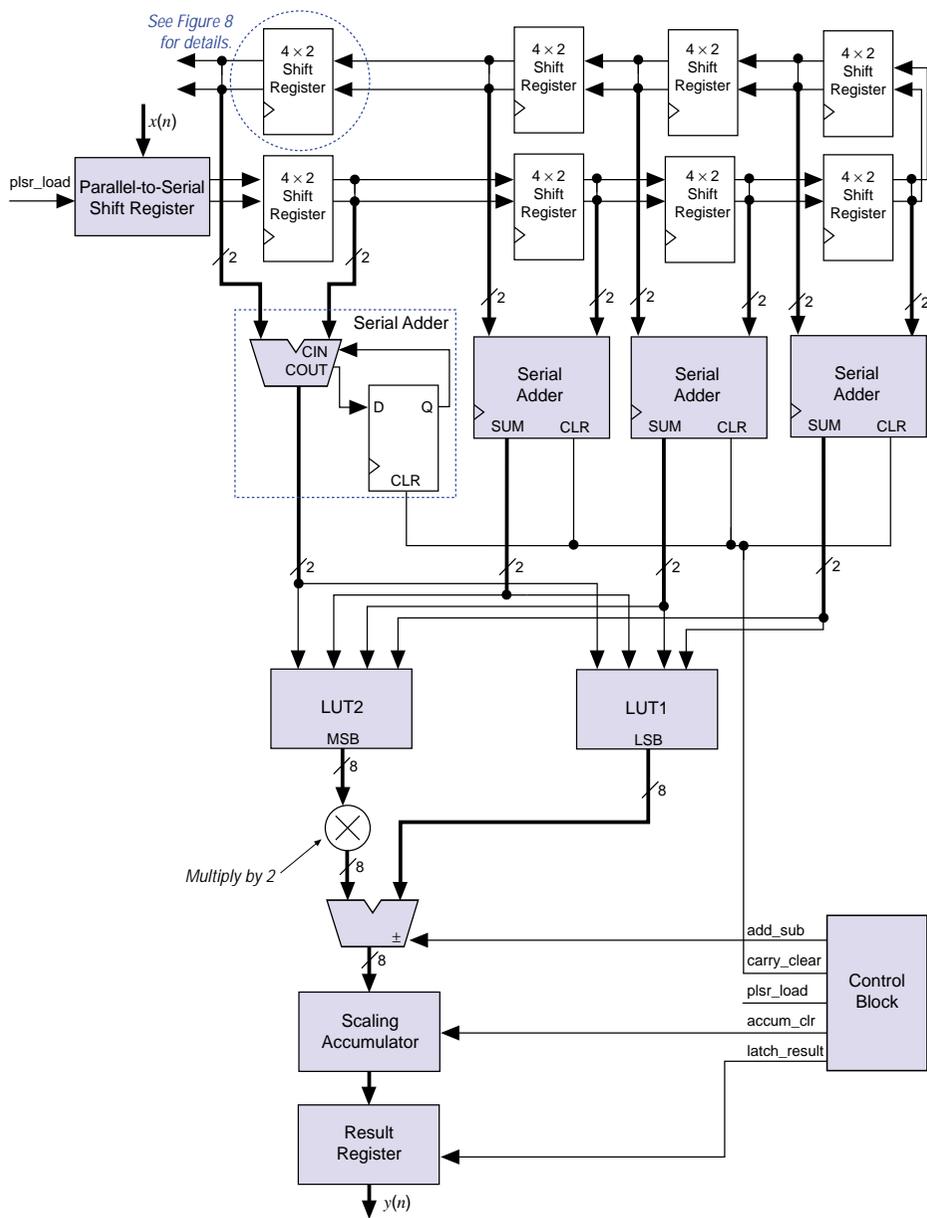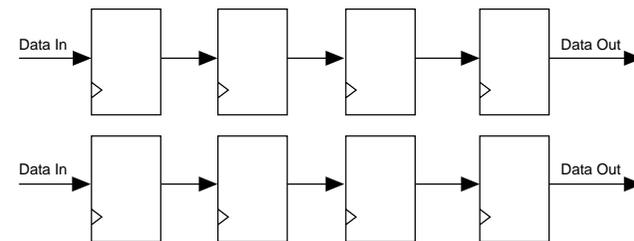*Figure 7. Combined Serial/Parallel FIR Filter Schematic*

Figure 8 shows the 4×2 shift register in Figure 7.
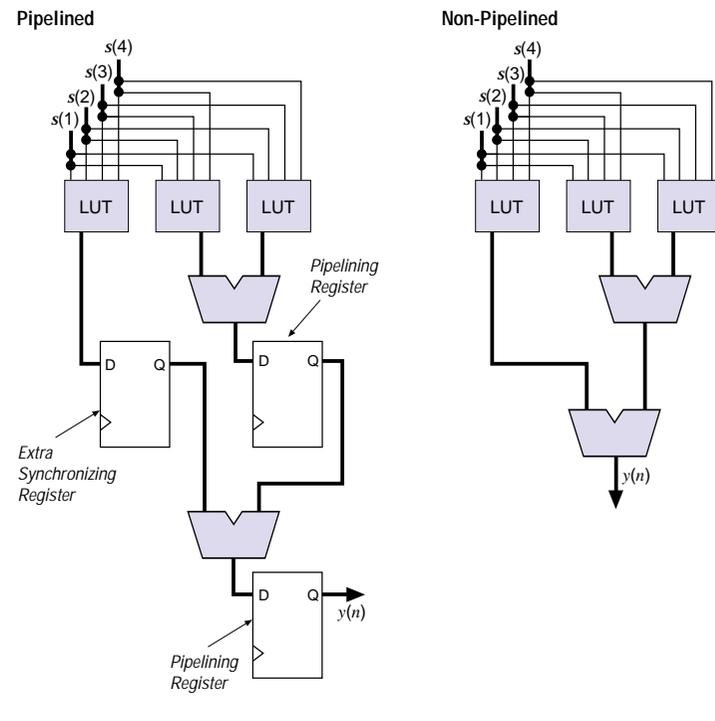
Figure 8. 4×2 Shift Register

To implement two's complement arithmetic, LUT2 for the MSB of the computation must contain the two's complement of the value. The adder at the bottom of Figure 7, which can be an adder or subtractor depending on the control signal add_sub, performs this operation. Every fourth cycle add_sub is de-asserted, complementing the output of LUT2. The accumulator at the bottom of Figure 7 adds and shifts the data by two bits every cycle.

Pipelining

Pipelining increases the performance of a FIR filter with little or no impact on the size of the design. The FLEX architecture has a flipflop in each logic cell. Therefore, an adder and a register require only one logic cell per bit. If the width of $s(n)$ is not a power of two, extra pipeline registers are required to maintain synchronization, which makes the pipelined filter less efficient. See Figure 9.

Figure 9. Pipelined vs. Non-Pipelined FIR Filter

Figure 10. 16-Tap Parallel FIR Filter

To create a serial filter with more than 8 taps, implement 2 or more 8-tap filters with 1 accumulator at the bottom of all the filters. Figure 11 shows a 32-tap filter.

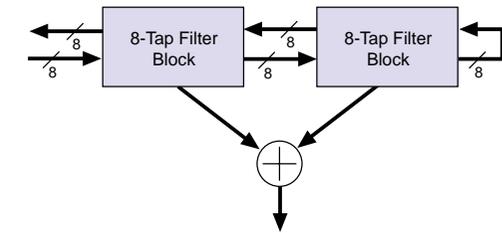Figure 11. 32-Tap Serial FIR Filter

Scaling

This section discusses how to adjust the number of taps or the bits of precision for a FIR filter.
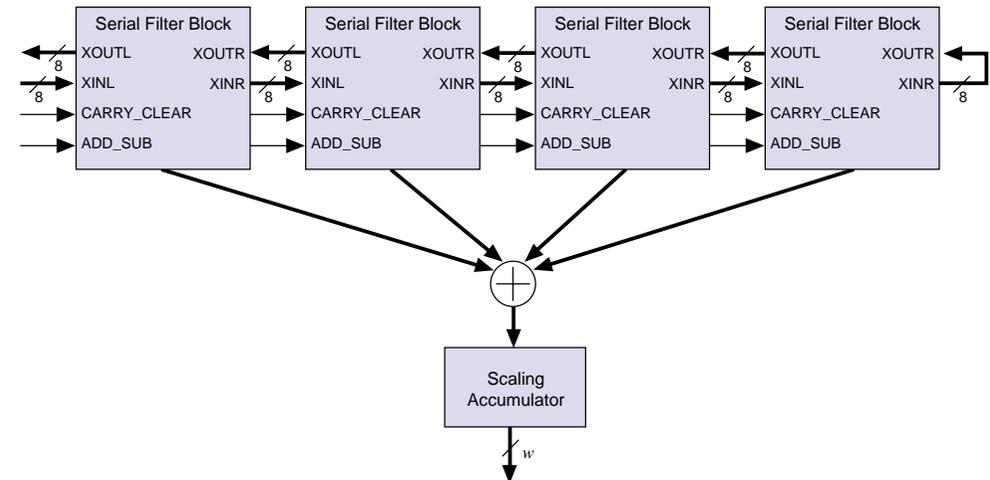
Adjusting the Number of Taps

You can easily create a parallel filter with more than 8 taps by implementing 2 or more parallel 8-tap filters—described previously in this application note—and assigning different coefficients to each filter. Then, add the outputs of each 8-tap filter. See Figure 10.

Adjusting the Bits of Precision

To add more bits of precision on the input, add additional LUTs to each block. For parallel designs, each extra bit of precision requires one additional LUT. In serial designs, the computation will take one more Clock cycle.
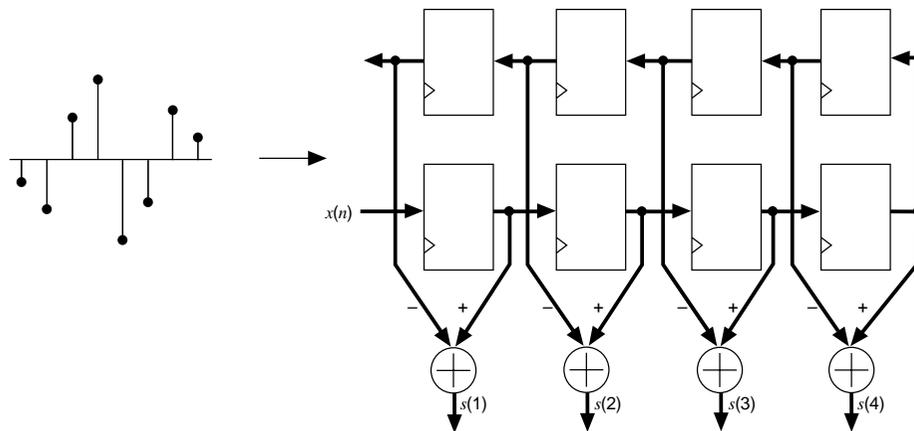
Symmetric & Antisymmetric Filters

Symmetric filters that implement even functions are created with symmetric tap adders. You can create an antisymmetric FIR filter that implements an odd function by using symmetric tap subtractors. See Figure 12.

Figure 12. Symmetric vs. Antisymmetric Filters
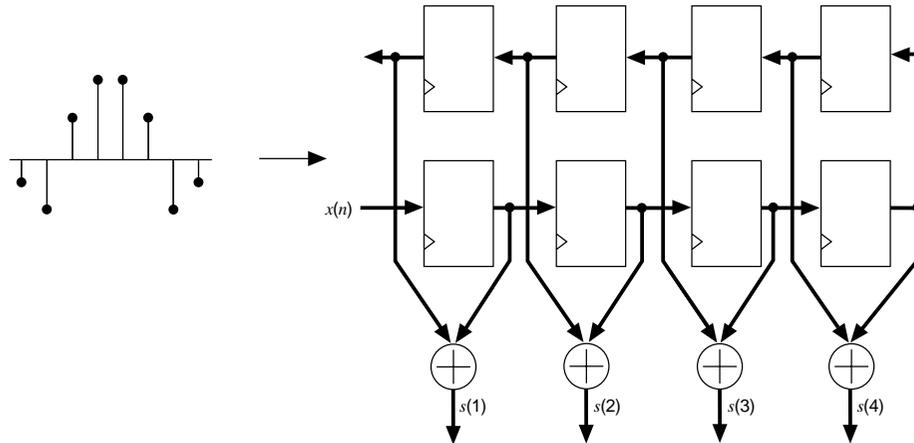
Truncation & Rounding

Truncation and rounding can be important in a design. Many existing FIR filters are not flexible enough to allow you to specify the amount of truncation or rounding, potentially wasting device resources.

In contrast, FLEX devices permit you to specify how many bits of precision to use when calculating the result of a FIR filter. For an 8-bit 32-tap filter, keeping every possible bit of precision requires an output word 19 bits wide. Most applications do not require 19 bits of precision for a function with an 8-bit wide input. You should keep only the bits of precision necessary for your design to minimize the design size.

Odd-Length Filters

To implement an odd-length filter, simply remove one of the shift register stages. See Figure 13.

Figure 13. 7-Tap Odd-Length Filter

Decimating Filter for Sample Rate Reduction

A decimating filter only computes every nth result, where n is the decimation factor. You can create a decimating filter by simply discarding any unwanted results from a regular FIR filter. For example, if the input data rate is 100 MHz, and you want to decimate the data rate by 2, the output data rate is only 50 MHz. See Figure 14. However, this implementation is not very efficient. The low-pass FIR filter must run at the XIN data rate (i.e., 100 MHz), which wastes half of the computations performed by the FIR filter.

Figure 14. Typical Decimating Filter

Figure 15 shows the shift register section of a 16-tap, decimate-by-2 FIR filter. This filter only computes every other result (i.e., the data skips every other register). Therefore, most of the filter runs at the output rate rather than the input rate, saving roughly 50% of the power consumed by the full-speed filter. The filter in Figure 15 also permits a faster input sample rate. For example, if the input data rate is 100 MHz, only the T flipflop must run at the 100-MHz rate—the rest of the FIR filter runs at 50 MHz. To create a filter that decimates by more than two, simply reroute the data through the shift register.

Figure 15. FLEX Decimating Filter

Interpolating Filter

An interpolating filter (i.e., an up-sampling filter) performs the opposite function of a decimating filter—it increases the sample rate by a factor of n. One way to perform interpolation is to add extra samples between each input data sample, where the added samples are zero. See Figure 16. The data stream (with the zeros) is sent through a low-pass filter; the output data from the low-pass filter is at a higher sample rate than the input data.

Figure 16. Stuffing Zeros into the Data Stream

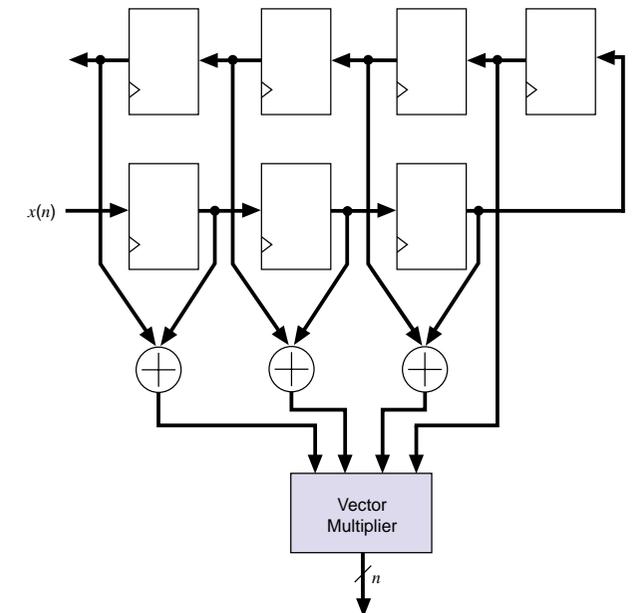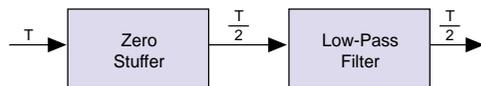Figure 17 shows the interpolating filter block diagram.

Figure 17. Interpolating Filter

Because much of the input data to the low-pass filter is zero, the shift register is always sparsely populated. Figure 18 shows an example of an interpolating filter where the data is up-sampled by two. One input to each of the symmetric tap adders is always zero. Therefore, the adder is not necessary.
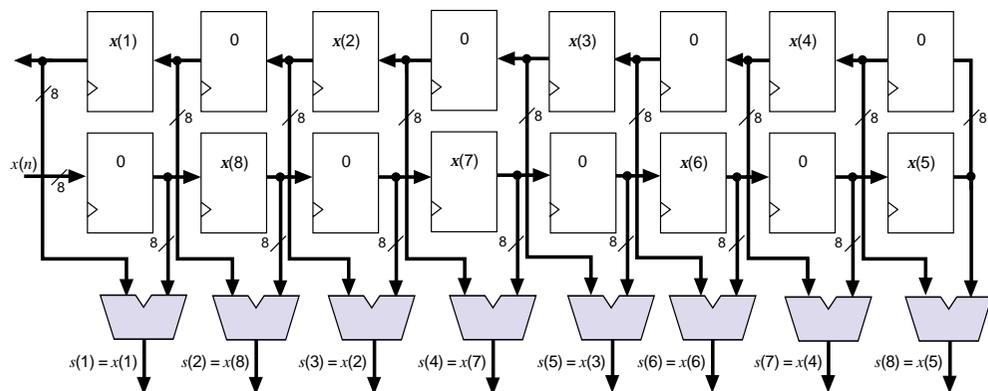
Figure 18. Up-Sampling Data by Two

Table 3 shows how s(*t*) evolves over time when the input data *x(n)* is *x*(10)*x*(9)*x*(8)*x*(7)*x*(6)*x*(5)*x*(4)*x*(3)*x*(2)*x*(1). Each time snapshot represents the value of *s(t)* at a given time.

Table 3. Values of s(t) over Time

s(t)								Time
X(1)	X(8)	X(2)	X(7)	X(3)	X(6)	X(4)	x(5)	T = 1
X(9)	X(2)	X(8)	X(3)	X(7)	X(4)	X(6)	x(5)	T = 2
X(2)	X(9)	X(3)	X(8)	X(4)	X(7)	X(5)	x(6)	T = 3
X(10)	X(3)	X(9)	X(4)	X(8)	X(5)	X(7)	x(6)	T = 4

Figure 19 shows a graphical representation of how s(*t*) evolves over time.

Figure 19. s(t) over Time
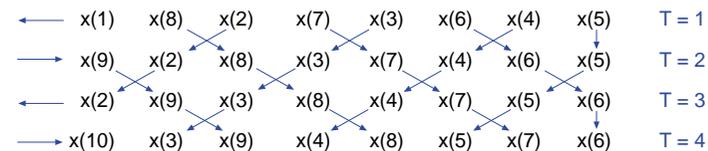
The data flow switches every other Clock cycle. Figure 20 shows the structure that would implement this shift register. You can implement the shift register in Figure 20 in a FLEX device using one logic cell per bit. Using a regular FIR filter with a zero-stuffed data stream requires three logic cells per bit: one to store the data, one to store a zero, and one to implement the adder. In a regular 16-tap, 8-bit FIR filter, the shift register section requires the following number of logic cells:

$$(\text{Taps} \times \text{Width}) + \left(\frac{\text{Taps}}{2} \times (\text{Width} + 1)\right) = (16 \times 8) + (8 \times 9) = 200$$

The more efficient shift register structure shown in Figure 20 would require only 64 logic cells, which saves significant resources.

Figure 20. 64-Logic Cell Shift Register Structure
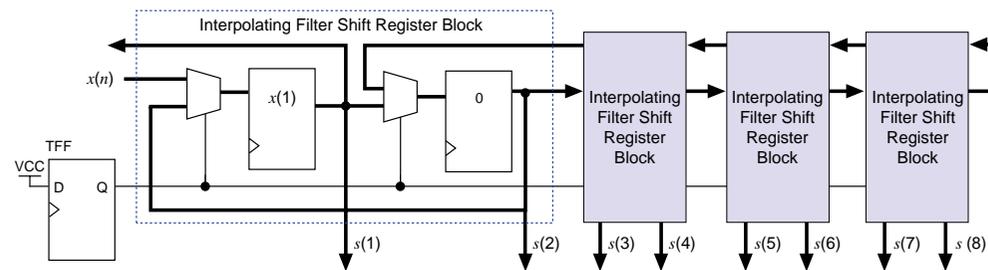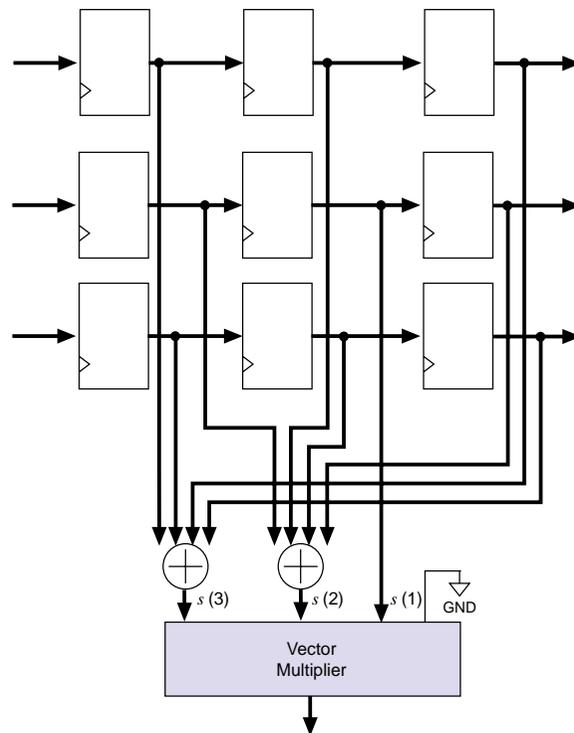

Two-Dimensional Video Convolution Filter

You can create a two-dimentional convolution filter using the techniques described above. Using a symmetric convolution window, you can extend the one-dimensional filter to a two-dimensional filter. Figure 21 shows a symmetric two-dimensional convolution window.

Figure 21. Symmetric Two-Dimensional Convolution Window

C3	C2	C3
C2	C1	C2
C3	C2	C3

The coefficients C1, C2, and C3 are constants. Figure 22 shows the architecture that implements this convolution filter. The two-dimensional filter uses the same vector multiplier as the one-dimensional filter, except only three inputs are used instead of four.

Figure 22. Convolution FIR Filter

You can implement the two-dimensional filter using the same procedure as the one-dimensional filter, but this implementation creates an inefficiency because one input from each LUT is not used. The two-dimensional video convolution filter can run at about 100 MSPS in a FLEX 8000A (-2 speed grade) device.

Two's Complement Arithmetic

Two's complement arithmetic is a system of representing numbers in which the negative of a number is equal to its inverse plus one. The most significant bit (MSB) requires special treatment in two's complement arithmetic. One option involves sign-extending each number by one bit, adding these sign-extended numbers with a regular adder, and discarding the carry-out. For example, for 2-bit two's complement numbers, the range of possible values are –2 through 1. The binary equivalents are summarized in Table 4.

Table 4. Binary Equivalents for 2-Bit Two's Complement Numbers

Binary	Decimal
00	0
01	1
10	–2
11	–1

When adding positive and negative numbers in two's complement arithmetic, a special procedure must be used. For example, –1 + 1 should equal 0 in decimal arithmetic. However, using a regular adder, the binary equivalent is 11 + 01 = 100 (i.e., –4). To obtain the correct result, the binary numbers must be sign-extended to 3-bit numbers and added, then the carry out must be discarded: 111 + 001 = 1000 (i.e., –1 + 1 = 0).

All adders in this application note use two's complement arithmetic, with an adder that is $n + 1$ bits wide, where n is the width of the input. The FLEX architecture implements very fast adders—e.g., carry-chains for the MSB—and does not require more logic to implement an adder than any other 4-input logic function. Therefore, using a full adder in a FLEX device to manipulate the MSB is faster than using an XOR gate.

Conclusion

There are several ways to implement FIR filters in FLEX devices, depending on speed and device resource constraints. The fastest, largest filter is a fully parallel pipelined version, which can operate as fast as 105 MSPS. The fully serial filter is the slowest and smallest and can sample at rates of 5.7 MSPS for an 8-bit input width, which is adequate for many medium speed applications. The combined serial/parallel filter is capable of rates between these two extremes, depending on how much of the design is in parallel. For high-speed FIR filter applications, Altera FLEX devices provide the most flexibility and best performance.

Revision History

The information contained in *Application Note 73 (Implementing FIR Filters in FLEX Devices)* version 1.01 supersedes information published in *Application Note 73 (Implementing FIR Filters in FLEX Devices)* version 1.0. Version 1.01 includes a corrected multiplier equation in Figure 4.

---

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(888) 3-ALTERA
lit_req@altera.com

I.S. EN ISO 9001

Printed on Recycled Paper.