



Block Adaptive Filter

XAPP 055 January 9, 1997 (Version 1.1)

Application Note by Bill Allaire and Bud Fischer

Summary

This application note describes a specific design for implementing a high speed, full precision, adaptive filter in the XC4000E/EX family of FPGAs. The design may be easily modified, and demonstrates the suitability of using FPGAs in digital signal processing applications.

Xilinx Family

XC4000 Series

Demonstrates

Adaptive filters and FPGA-based DSP design techniques

General Description of Adaptive Filters

Adaptive filters are digital filters capable of self adjustment. These filters can change in accordance to their input signals. An adaptive filter is used in applications that require differing filter characteristics in response to variable signal conditions. Adaptive filters are typically used when noise occurs in the same band as the signal, or when the noise band is unknown or varies over time.

The adaptive filter requires two inputs: the signal and a noise or reference input. An adaptive filter has the ability to update its coefficients. New coefficients are sent to the filter from a coefficient generator. The coefficient generator is an adaptive algorithm that modifies the coefficients in response to an incoming signal. In most applications the goal of the coefficient generator is to match the filter coefficients to the noise so the adaptive filter can subtract the noise out from the signal. Since the noise signal changes the coefficients must vary to match it, hence the name adaptive filter. The digital filter is typically a special type of finite impulse response (FIR) filter, but it can be an infinite impulse response (IIR) or other type of filter.

Adaptive filters have uses in a number of applications including noise cancellation, linear prediction, adaptive signal enhancement, and adaptive control.

General Description of FPGA-based Signal Processing

Most digital signal processing done today uses a specialized microprocessor, called a digital signal processor, capable of very high speed multiplication. This traditional method of signal processing is bandwidth limited. There is a fixed number of operations that the processor can perform on a sample before the next sample arrives. This limits either the applications that can be performed on a signal or it limits the maximum frequency signal that the application can handle. This limitation stems from the sequential

nature of processors. DSPs using a single core can only perform one operation on one piece of data at a time. They can not perform operations in parallel. For example, in a 64 tap filter they can only calculate the value of one tap at a time, while the other 63 taps wait. Nor can they perform pipelined applications. In an application calling for a signal to be filtered and then correlated, the processor must first filter, then stop filtering, then correlate, then stop correlating, then filter, etc. If the applications could be pipelined, a filtered sample could be correlated while a new sample is simultaneously filtered. Digital Signal Processor manufacturers have tried to get around this problem by cramming additional processors on a chip. This helps, but it is still true that *in a digital signal processor most of your application is idle most of the time.*

FPGA-based digital signal processing is based on hardware logic and does not suffer from any of the software-based processor performance problems. FPGAs allow applications to run in parallel so that a 128 tap filter can run as fast as a 10 tap filter. Applications can also be pipelined in an FPGA, so that filtering, correlation, and many other applications can all run simultaneously. In an FPGA, most of your application is working most of the time. *An FPGA can offer 10 to 1000 times the performance of the most advanced digital signal processor at similar or even lower costs.*

Adaptive Filter Design Overview

This application note is based on a 12 bit data, 12 bit coefficient, full precision, block adaptive filter design. This design can be modified to accommodate different data and coefficient sizes, as well as lesser precision. The application note covers how to modify the design including the trade-offs involved. The filter is engineered for use in the XC4000E and XC4000EX families. The synchronous RAM and carry logic in these families make this design possible.

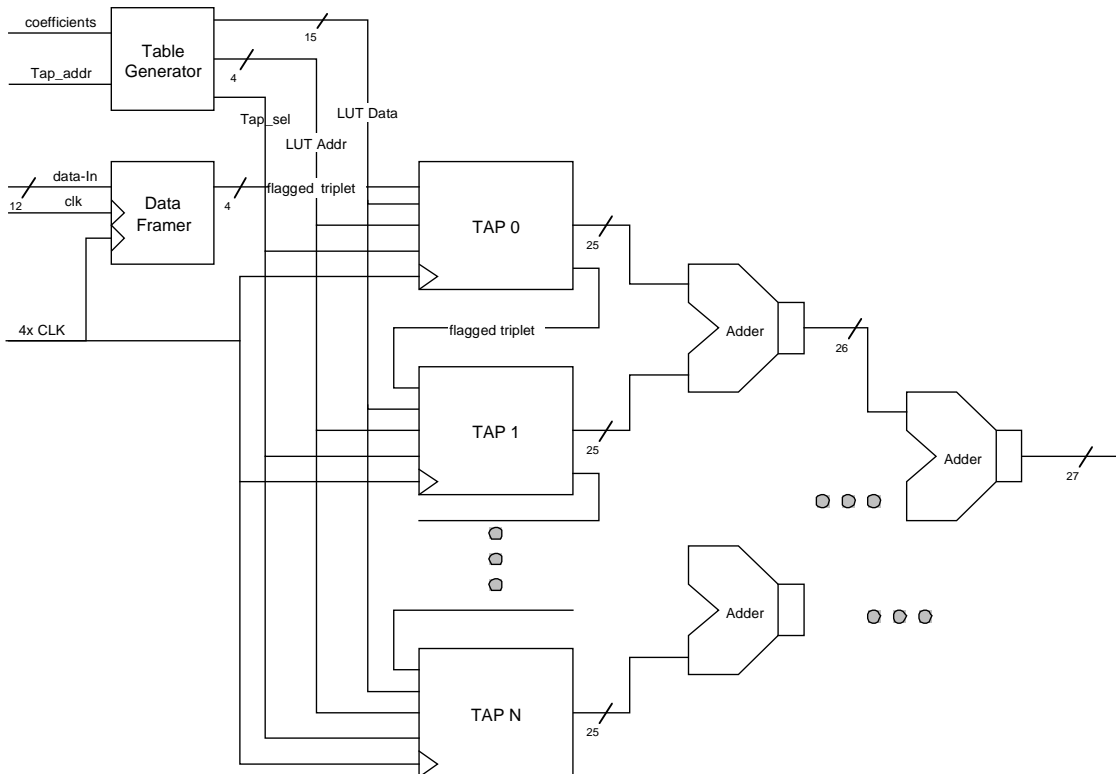


Figure 1: Block Diagram of the Block Adaptive Filter

There are a large number of designs that could fit this application. This design has a good balance between performance and density (gate count). This design can sustain a 15.5 MHz, 12 bit sample rate with an *unlimited* number of filter taps. Modified versions of this design can provide higher throughput at the expense of consuming more resources, or modified versions can provide better resource efficiency at a lower performance. Design modifications are discussed later in this application note.

Figure 1 is an overview of the entire adaptive filter. There are four basic components to the filter: the Table Generator, the Data Framer, the Filter Tap, and the Adder Tree.

The Filter Tap Design

The heart of any filter is the tap. The tap multiplies coefficient data (a constant) by sample data (variable input data) and outputs the result. A new result is calculated for every sample. The tap also forwards the sample data to the next

tap in the filter. In the case of the adaptive filter the coefficient can be changed periodically.

To conserve resources the filter tap described in this application note imports the sample data in four pieces. Each piece is multiplied by the coefficient to produce a partial product. The partial products are added together to produce the sample's final result for the tap. This tap has three main sections: Time Skew Buffer, Partial Product Multiplier, and Scaling Accumulator.

Implementing separate scaling accumulators in each tap increases the performance of the filter. Fewer resources could be used by sharing a single scaling accumulator at the output of the adder tree. However, adder trees are inherently slower than the tap. If they are used at the bit-rate they will limit the performance. By replicating the scaling accumulator in each tap, the adder tree operates at the word rate, and its delay is non-critical.

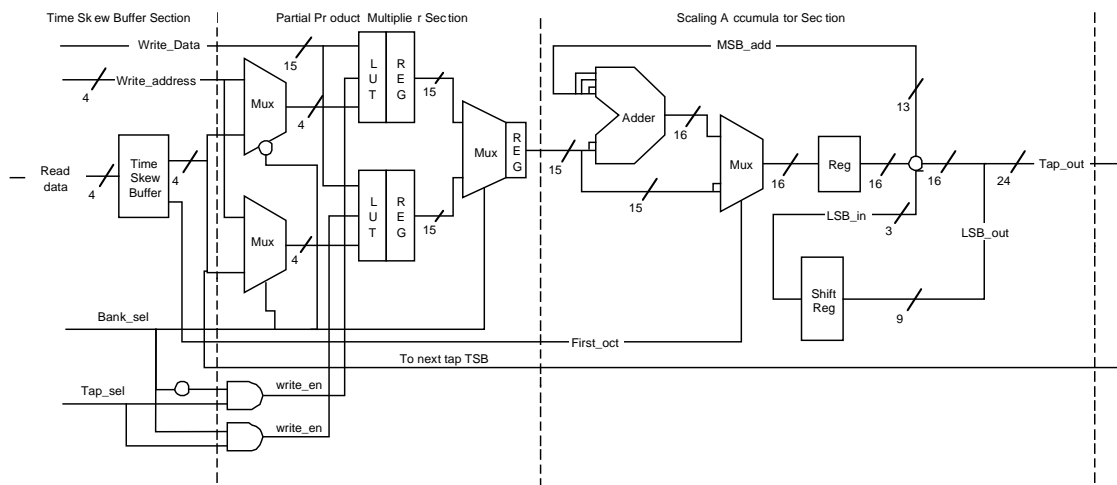


Figure 2: Block Diagram of the Filter Tap

The Tap Time Skew Buffer

The tap is depicted in [Figure 2](#). The incoming data has already been broken into 4 bit pieces (3 bits data, 1 control) by the Data Framer before it enters the tap. The data entering the tap is placed in the Time Skew Buffer (TSB). The TSB is simply a 4 x 4 bit shift register. The TSB is able to contain an entire sample (4 pieces of data, 4 bits each), storing the filter data. The TSB shifts 4 bit data out every clock cycle to both the Partial Product Multiplier and the next tap. The TSB is not strictly necessary for Tap 0, but can be included to make all the taps identical.

The Tap Partial Product Multiplier

The Partial Product Multiplier is a RAM look-up table. The theory behind a look-up table (LUT) multiplier is straightforward. A 4-bit variable, for example, (the data) multiplied by a constant (the coefficient) has only 16 possible solutions. These solutions can be pre-calculated. If the solutions are accurately loaded into a 16 word RAM then the 4 bit data can be used as a read address to the RAM to select the correct partial product result for itself.

The filter is adaptive, meaning that the coefficients will change, therefore, the LUT must be able to change. To continue processing samples while new table results are calculated requires two look-up tables. One table can be providing answers while the other is being updated with results calculated using a new coefficient. Two multiplexers determine which LUT is reading data out and which is writing data in. The two multiplexers receive read address lines (Data) and write address lines for both LUTs. A Bank_sel

line is sent to each multiplexer to select either the read address or write address for each LUT. The Bank_sel line is inverted to one of the multiplexers so that a write address is always available to one LUT while a read address is being sent to the other LUT.

LUTs are a straightforward, high-performance method of generating filter tap solutions, but they are very resource intensive. A standard LUT for a 12 bit filter tap would require a RAM of 4K words (24 bit words) to provide every possible result. An adaptive filter requires two LUTs per tap, quickly making this method unfeasible. To be more resource efficient the sample data is partitioned into 4 segments.

Each sample is divided into 4 sections of 3 bit data which is called an octet. A control signal accompanies every octet. Each octet, with its control signal, acts as a four line read address bus for a 16 word RAM-based LUT. Each word is 15 bits (12 bit coefficient multiplied by a 3 bit octet). The results from the 4 octets in each sample are then accumulated to produce the tap's complete, full-precision solution.

There are 4 octets per sample, so the octets must be processed at a clock rate equal to 4 times the sample rate. Therefore the sample rate is one quarter of the device maximum throughput speed. A later section in this application note will discuss methods for running higher sample rates.

The octets, generated in the Data Framer, are arranged and presented to the filter tap from least significant octet to most significant octet. See [Figure 3 on page 4](#).

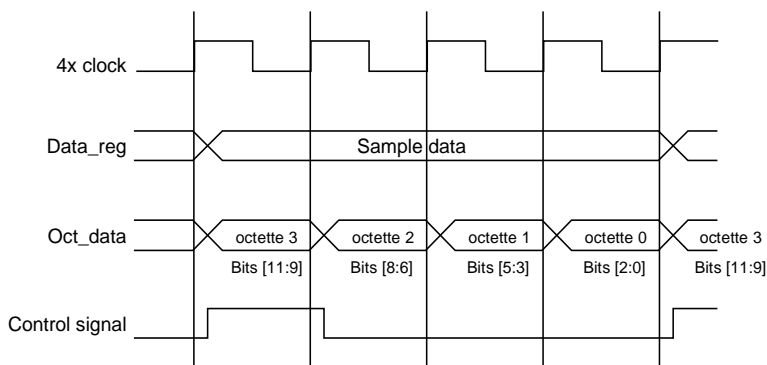


Figure 3: Octet Timing Diagram

The LUT values are shown in [Table 1](#). Normally the 3 bit octets would only require an eight word LUT to cover every possible result. The most significant octet (MSO), however, carries signed data and must be handled as an exception. The control signal accompanying every octet acts as an exception indicator. The added control signal occupies the highest address line in the now 4 bit data. The control signal is only asserted for the most significant octet so that the address of the MSO always points to the upper 8 words of the LUT. The upper 8 words are of the LUT are loaded with signed data in twos complement form. The most significant bit of the MSO is the signed bit. Zero is positive. A 1 is negative with a magnitude, equaling -4, by virtue of its position in the third bit. The second and first bit are positive numbers that are added to the -4. Therefore, 101 equals -3 (-4 + 1 = -3) in twos complement math.

The results for the 3 lower order octets (LOOs) are unsigned, use standard binary math, and are found in words 0 through 7 of the LUT. See [Table 1](#).

The output for the LUT is a 15 bit twos complement result. This accounts not only for potential negative data in the MSO, but also for the possibility of a negative coefficient.

The LUT output is the result of the Partial product Multiplier. The output of both LUTs are multiplexed. The Bank_sel line ensures that the result from the correct LUT is chosen, and sent on to the Scaling Accumulator Section.

Applications using magnitude only can eliminate the upper 8 words of the LUT.

Table 1: LUT Values

LUT Address		LUT Value	Sign Data
CS	Octet		
0	000	0 * coefficient	Unsigned Results: Octets 0, 1, 2
0	001	1 * coefficient	
0	010	2 * coefficient	
0	011	3 * coefficient	
0	100	4 * coefficient	
0	101	5 * coefficient	
0	110	6 * coefficient	
0	111	7 * coefficient	
1	000	0 * coefficient	Signed Results: Octet 3
1	001	1 * coefficient	
1	010	2 * coefficient	
1	011	3 * coefficient	
1	100	-4 * coefficient	
1	101	-3 * coefficient	
1	110	-2 * coefficient	
1	111	-1 * coefficient	

The Tap Scaling Accumulator

This job of the tap scaling accumulator is to take the 4 partial product results created for each sample and to add them together to produce the tap's full precision solution for the sample.

There are two important things to remember in regard to adding the octet results. The first is that the incoming data carries signed information and is therefore in twos complement form. The second is that each successive octet result is 3 bits more significant than the last octet result. This is due to the bit position of the octets in relation to the original sample data. For example octet 0 is comprised of sample bits [2:0], octet 1 is 3 bits (8 times) more significant because it is made of sample bits [5:3], etc. This relative

positioning must be recreated when adding the octet results.

To recreate the significance of each octet as they are added we need to perform the equivalent of the following. The result of octet 0 divided by 8, added to the result of octet 1, that result divided by 8, added to the result of octet 2, that result divided by 8 and added to the result of octet 3. This provides the tap's final result.

An efficient way to accomplish this is to shift the less significant result 3 places to the right before adding. Notice that when this occurs the least significant bits in the lower significance result have nothing to add, so we can remove and save them (they will be discussed later).

```
111100011010110
+ 010101101001011
```

Removing these gives:

```
111100011010
+ 010101101001011
```

These numbers, however, are both in twos complement form, and carrying sign data in their most significant bit. To maintain the sign value for both numbers it is necessary to sign extend the less significant result 3 places. To sign extend, the most significant bit (now bit 12) is replicated onto newly created bits 13, 14, and 15. This gives:

```
111111100011010
+ 010101101001011
```

The purpose for this is to have the sign bits line up. *Sign extending does not change the number's value.* This is obvious if the bit extended is zero, but is also true if the bit value is one. This is because the one carries both a magnitude and a negative sign. Recall the section on the LUT Multiplier. The example there showed that $101 = -3$ ($-4 + 1 = -3$). If 101 is sign extended to 1101 it is still equal to -3 ($-8 + 4 + 1 = -3$). This is true no matter how many places the sign is extended.

Using this method the least significant 3 bits of the result of octet 0 are removed and saved (these become bits [2:0] in the final tap result). The remaining data is sign extended 3 places and added to the result of octet 1. The least significant 3 bits of this result are removed and saved (these become bits [5:3] in the final tap result). The remaining data is sign extended 3 places and added to the result of octet 2. The least significant 3 bits of this result are removed and saved (these become bits [8:6] in the final tap result). The remaining data is again sign extended 3 places and added to the result of octet 3. This result becomes bits [23:9] of the final result. Combining this bus with the already derived bits [8:0] provides the final tap result.

The section designated Scaling Accumulator in [Figure 2 on page 3](#) shows how the design implements the method explained above. The 15 bit twos complement result from the Partial Product Multiplier is sign extended to 16 bits. This will account for overflow when the results of the octets are added. To sign extend, the value of bit 15, the most significant bit, is replicated onto a newly created line 16 (now the most significant bit). This does not change either the magnitude or sign of the data.

The 16 bit data is sent to both a multiplexer and a 16 bit adder. The output from the adder becomes the other input for the same multiplexer. This creates a bypass channel where the multiplexer can either send the data through directly or choose the accumulated result. The multiplexer output is then registered. The purpose of the multiplexer is to enable the first octet data (octet 0) to be sent through without being added to the accumulated result of the last sample. This effectively clears the adder after every sample.

The three least significant bits are sent to a shift register. The most significant 13 bits are fed back and logically mapped 3 bits down (bit 3 to bit 0, bit 4 to bit 1, etc.). Then they are sign extended 3 places and sent into the adder to be added to the result of the next octet.

On the addition of the MSO (octet 3) data, the 16 bit result is combined with the 9 bit data from the shift register to create a 24 bit full precision tap result. The lower 15 bits from the adder will be the most significant bits and the 9 bits from the shift register will be the least significant bits of the 24 bit Tap_out bus. Note that interim solutions (from octet 0-2 data) require the input data to be sign extended to prevent over flow. During the accumulation of the MSO (octet 3), there is no possible overflow allowing the MSB to be removed from the adder's final output. The three interim values will be sent out of Tap_out. The clock rate outside the tap, however, is only 1/4 of the tap clock. Therefore data valid outside the tap will only occur for the MSO (octet 3) accumulated result.

This completes the section for building a filter tap. It is suggested, however that the user create a relationally placed macro (RPM) for the design. An RPM provides partitioning and relative placement information to the design compiler. RPMs insure that the taps all maintain identical performance and do not vary with place and routing. It also guarantees that none of the taps will change when other logic is added or subtracted from the rest of the design. Placement of the logic for the RPM is conveniently handled by the Floorplanner tool. Combining the use of the RPM with the TimeSpec feature insures that a consistent minimum performance is met.

Combining the Tap Results

To derive the solution for the entire filter, it is necessary to add together all of the tap results for each sample cycle. To do this an adder tree is created.

The Adder Tree is very straightforward. The results of all the taps are paired and added. Those results are paired and added, and so on, until only one result remains. This result is the solution for the entire filter. Please note that the inputs for all the adders, on all levels, is in twos complement form and must be sign extended to account for possible overflow.

The number of adders required is equal to the number of taps minus one.

Preparing the Sample Data

The Data Framer is located before the first tap. It breaks the 12 bit incoming data into four 3 bit octets. The data is arranged in octets to save resources in the filter tap. The Data Framer also adds the control signal to each octet that identifies the most significant octet. See [Figure 4](#).

The 12 bit sample data is registered and then sent on to one of three 4 to 1 multiplexers. The three multiplexers

each output one bit of the three bit octet. The multiplexers are used to select the most significant bit, the middle bit, and the least significant bit for each octet. The input sample data is alternated between the multiplexers so that line 0 goes to the first multiplexer, line 1 goes to the second multiplexer, and line 2 goes to the third multiplexer. The pattern then repeats until all 12 lines are connected to multiplexers. The multiplexer selections are controlled by a 2 bit counter: 00 selects octet 0, 01 selects octet 1, 10 selects octet 2, and 11 selects octet 3. The results from the counter are also sent to an AND gate to create the control signal. This makes the control signal high when octet 3 is selected. The output of the multiplexers (the octet) and the control signal are registered and sent to the tap.

Partial Products Generation

This filter is an adaptive filter, therefore, it must change to accommodate changing signal conditions. The filter changes when it is sent a new coefficient. When the coefficient changes the LUT values in the Partial Product Multiplier become obsolete. The Partial Products Table Generator uses the new coefficient to create new values to be sent to the tap LUTs. It does this for every tap in the filter. See [Figure 5](#) and refer to [Table 1](#).

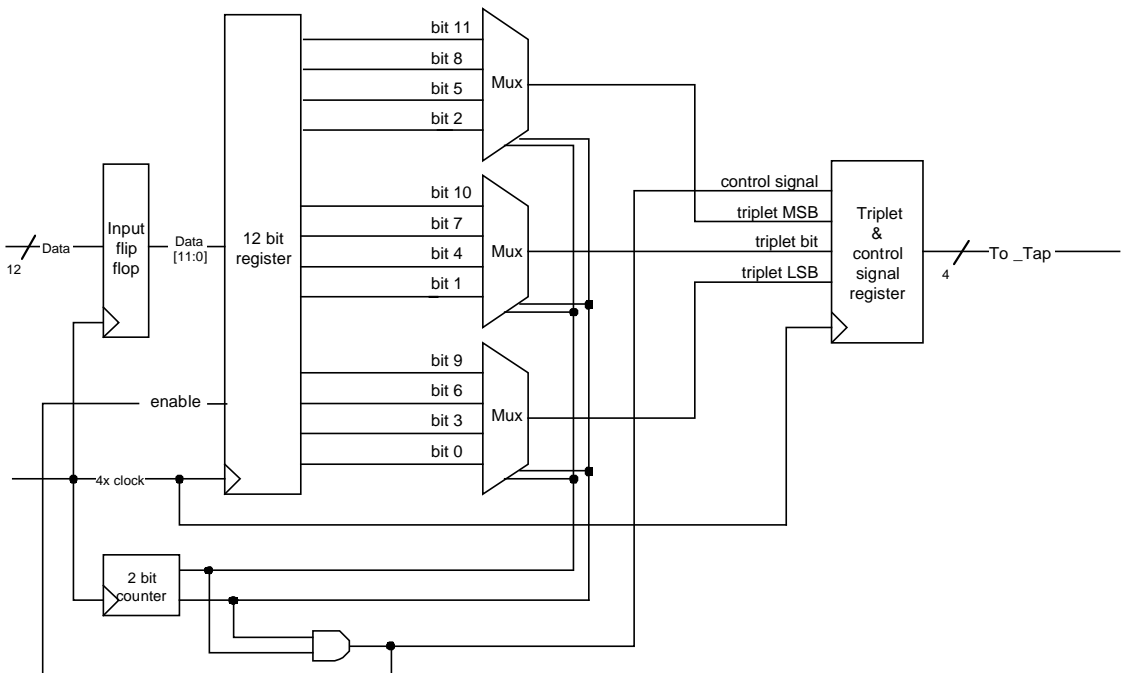


Figure 4: Block Diagram of the Data Framer

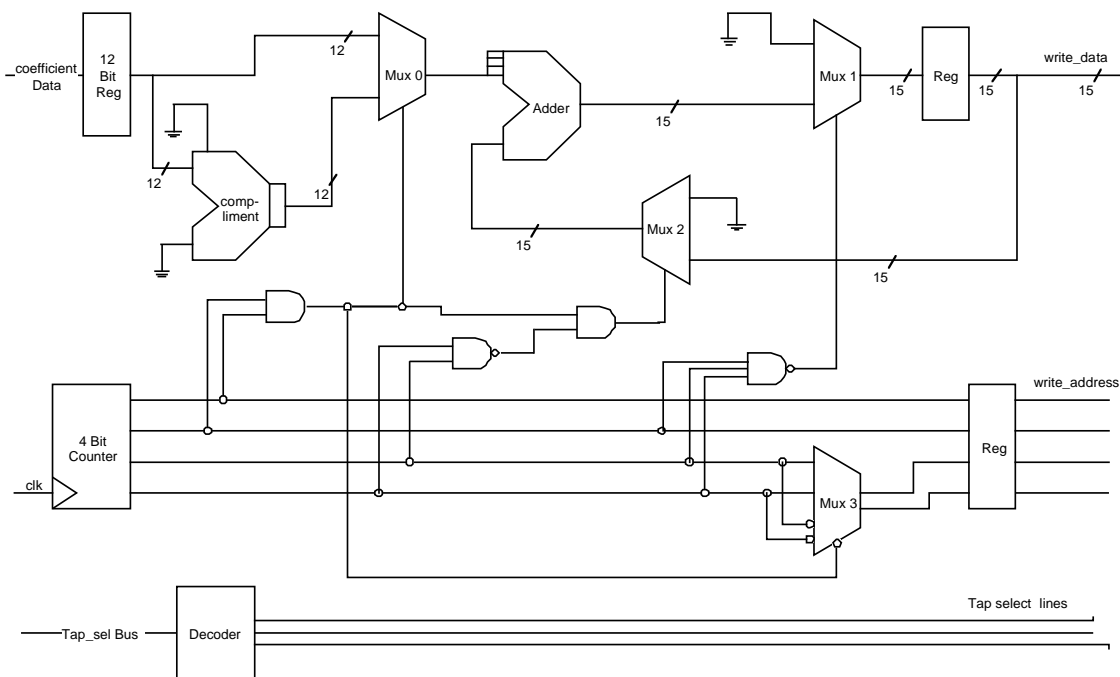


Figure 5: Block Diagram of the Partial Product Table Generator

The new coefficient and Tap_address are sent to the filter and received by the Partial Products Table Generator. A new tap address accompanies each new coefficient. This is a block adaptive filter so the coefficients are not updated after every sample. The coefficient and tap address can be updated after every fourth sample, which is 16 clock cycles. This means that it takes 4 sample periods (16 clock cycles) to reload the LUT in each tap. The number of samples the entire filter takes to reload is 4 times the number of taps in the filter.

To determine which tap is to be updated with the new coefficient a tap address is sent to the filter and received by the Partial Products Table Generator. The Partial Products Table Generator decodes the address and selects the appropriate tap. The tap_address, decode, and tap select lines are minimally sketched in Figure 5, because the number of address lines, decode logic, and tap select lines will vary based on the number of taps used in the application. Only a standard address decode is required and this should be straightforward for the user.

The actual partial product generator creates the LUT values and is somewhat complex. It is more resource efficient, however, than using a more standard multiplier. A 4 bit counter is used to create the LUT write address. These address lines can be logically combined to create control signals for the LUT generator.

Look at Table 1 on page 4. It shows that an accumulator will work very well for words 0-7. If an initial value of zero is used the coefficient need only be added to the last result to achieve the next result. The same is true for words 8-11. Multiplexer 1 chooses between ground (zero) and the output of the adder. The control signal for multiplexer 1 is the NAND of the 3 lower bits of the write address. This ensures that a zero is sent to the LUT for addresses 0 and 8. It also effectively clears the adder. The adder takes its own output from a register and feeds it back to itself to be added to the coefficient. This provides exactly the results needed for addresses 1-7 and 9-11. For example the scaling accumulator output for address 1 is the coefficient, address 2 is coefficient + coefficient (which is equal to $2 * \text{coefficient}$), address 3 is coefficient + coefficient + coefficient (which is equal to $3 * \text{coefficient}$), etc. Address 9 is equal to the coefficient again because address 8 had cleared the adder.

The upper 8 words of the LUT are strictly used for finding the partial product of the MSO. This is because the MSO contains sign data. The lower 4 words (addresses 8:11) account for positive data in the MSO and their handling has been described in the paragraph above. The upper 4 words (addresses 12-15) need to account for negative answers and this makes things interesting. Inverting the coefficient and adding 1 yields the complement (negative) of the coefficient. Multiplexer 0 chooses between the coefficient and

its complement. The multiplexer 0 select line is the AND of address lines 3 and 2. This means that the complement is always chosen for addresses 12-15.

Clock cycle 13 is when the table results transition from positive to negative. Starting at address 12 (clock cycle 13) the coefficient input to the adder is complemented, but the adder must also be cleared during clock cycle 13 to send through negative data. Multiplexer 2 selects zero to be fed back during clock cycle 13 (AND address lines 3 and 2, NAND address lines 1 and 0, and AND the results of both operations). The result from the adder is then 0 + -coefficient. Looking at [Table 1 on page 4](#) shows that this is not the correct result for address 12, but it is correct for address 15. Continuing in this manner the output of the adder during clock cycle 14 would be -coefficient + -coefficient. Again this is incorrect for address 13 but correct for address 14. The adder result for address 14 is incorrect, but correct for address 13, and the adder result for address 15 is incorrect, but correct for address 12. Notice, however, if the write address lines [1:0] are inverted during clock cycles 13 - 16 the LUT will be loaded correctly. Multiplexer 3 chooses between address lines [1:0] and their inverse. The inverse is selected by the AND of address lines [3:2]; the same line that chooses the complemented coefficient for multiplexer 0.

As noted earlier this Partial Products Table Generator takes 4 times the number of taps to reload all the LUTs in the filter. To converge more quickly multiple Table Generators can be used. Choosing initial coefficients that are near the conversion point, if possible, will also speed convergence. If ultimate convergence time is required, however, a block adaptive filter is not the best solution.

Performance and Sizing Estimates

All of the performance and sizing estimates are based on the XC4000E family using the -3 speed grade, which is the fastest available at the time of this writing. The XC4000EX family offers the potential for slightly better performance, in the same speed grade, for very large filters due to its more abundant routing resources. *Note that the sample rate performance is for a 12 bit sample.*

Data Framer Size and Performance

Only one data framer is needed for the adaptive filter so it uses a relatively small portion of the overall resources. The data framer is not a bottleneck for the filter so any performance increase will not improve the filter throughput until other sections are enhanced.

# CLBs	CLK/ Octet Rate	Sample Rate
16	Performance is tap limited	

The Adder Tree Size and Performance

The size and performance of the adder tree will vary according to the number of taps. Not only does the number of adders change in accordance with the number of taps, but the size of the adders will change according to the number of levels in the tree, because the number of bits to be added will change (to handle overflow). In general the number of CLBs needed per adder will be half the number of bits plus one. Minor changes in the adder tree will not affect overall filter performance.

Single 24 Bit Adder		
# CLBs	CLK/ Octet Rate	Sample Rate
16	Performance is tap limited	

Table Generator Size and Performance

Only one table generator is needed for the adaptive filter so its impact on the overall size of the filter is not great. The table generator is not a bottleneck for the filter so any performance increase will not improve the filter throughput.

# CLBs	CLK/ Write Cycle	Sample Rate
56	Performance is tap limited	

Filter Tap Size and Performance

The tap is the most important factor in the size and performance of the adaptive filter. Since the tap is used many times its size has the greatest impact on the filter density. The tap also contains the filter performance bottleneck. The Adder in the Scaling Accumulator Section takes a maximum of 14 nanoseconds. This limits the total performance of both the tap and the entire filter.

Time Skew Buffer		
# CLBs	CLK/ Octet Rate	Sample Rate
10	Scaling Accumulator	limited

Partial Product Multiplier		
# CLBs	CLK/ Octet Rate	Sample Rate
27	Scaling Accumulator	limited

Scaling Accumulator		
# CLBs	CLK	Sample Rate
14	62MHz	15.5 MSPS

Tap Total		
# CLBs	CLK	Sample Rate
51	62MHz	15.5 MSPS

Modifying the Design

There are four basic ways to lower the amount of resources used by this filter: Reduced Precision, Smaller Sample Size, Smaller Coefficient Size, and Reduced Performance.

Reduced Precision

Reduced precision will provide a small, but significant amount of resource abatement. For example changing from full precision to 16 bit precision will reduce every adder in the adder tree by 4 CLBs. It will also eliminate the shift register from the filter tap, saving 5 CLBs per tap. Changing the precision will not affect the filter throughput.

Sample or Coefficient Size in Regard to the Adder Tree

Sample or coefficient size alteration will change the number of CLBs needed by the adders in the Adder Tree by one CLB for every two bits of modification. For example; 8 bit data and 8 bit coefficient require a 16 bit adder which takes 9 CLBs. A filter using a 12 bit sample and 12 bit coefficient needs a 13 CLB adder. The adder sizes do not have any affect on the performance of the filter, because the performance bottleneck is in the tap.

Performance, Sample Size, and Coefficient Size in the Tap

Performance, Sample Size, and Coefficient Size are all very interrelated inside the tap. The most important factor, however, is the data frame size. The frame size is the size of the segments into which the sample is broken. In this application note the sample has been broken into octets, but the design could be modified to use a single bit, paired bits, nibbles, etc. The frame size has the most impact on both performance and resources. [Table 3 on page 10](#) shows some examples of the relationship between frame sizes, coefficient sizes, sample sizes, and performance.

The frame size plus the control signal governs the depth of the LUTs. The LUTs are organized so that the partial product determines the width, and the frame size plus the control signal (address lines) determine the depth of the RAM. RAM in the XC4000E/EX families is organized as 16 x 1 bits. This requires the depth (addresses) of both LUTs together to be a multiple of 16. Since the frame size determines the number of address lines, the frame size dictates the RAM depth. The LUT width is the partial product size, which is equal to the coefficient bit width plus the frame bit width.

The partial product size, which is a function of the coefficient size and frame size, controls the resources used by everything after the LUT, except the shift register. Every two bits in the partial product size changes the number of CLBs per tap by about four; one each for the two multiplexers, the adder, and the register.

Performance is also a function of the frame size. The size of the adder in the scaling accumulator section determines the maximum clock rate for the filter, but the frame size determines the actual throughput. The number of segments into which the sample is divided is the main factor in performance. The segments are processed sequentially in the tap, so the sample throughput is the adder speed divided by the number of segments per sample.

Example Using a 2 Bit Frame Size

Employing the paired data frame size (2 bits) for a 12 bit sample and 12 bit coefficient filter saves about 10 CLBs per tap (versus octets), at the expense of a 33% degradation in performance.

The performance is slower for paired 12 bit sample and 12 bit coefficient data because it takes 6 clock cycles per sample instead of the four clock cycles needed for octets.

In using paired data the majority of the CLB savings come from the use of smaller LUTs. The two LUTs change from 16 words of 15 bits each, to 8 words of 14 bits each. This is an 8 CLB reduction. The TSB will need to become a 6 by 3 bit shift register. The scaling accumulator section also changes. The partial product is smaller so all the functions are now smaller (15 bits), but the shift register now must hold 10 LSBs.

Note that the Data Framer and Partial Products Table Generator will need modification to accommodate different frame, sample and coefficient sizes. The Data Framer for the 12 coefficient, 12 bit sample, paired data filter must now have two 6 to 1 multiplexers. The counter must be a 3 bit counter, the control signal needs to be set to 1 when the counter reaches 5 and the counter must be reset after 5. The Partial Products Table Generator must generate [Table 2](#).

Summary

There are literally hundreds of ways to modify the design in this application note. Digital signal processing in the XC4000E/EX allows the designer tremendous flexibility to achieve the user's exact goals.

Table 2: Partial Products Table Generator

LUT Address		LUT Value	Sign Data
CS	Pair		
0	00	0 * coefficient	Unsigned Results: Pairs 0, 1, 2, 3, 4
0	01	1 * coefficient	
0	10	2 * coefficient	
0	11	3 * coefficient	
1	00	0 * coefficient	Signed Results: Pair 5
1	01	1 * coefficient	
1	10	-2 * coefficient	
1	11	-1 * coefficient	

Table 3: Relationship Between Frame Sizes, Coefficient Sizes, Sample Sizes, and Performance

Sample Size X Coefficient Size	Data Frame Size					
	Pair		Octet		Nibble	
	Speed*	CLBs*	Speed*	CLBs*	Speed*	CLBs*
8 X 8	19 MSPS	34	24 MSPS	39	35 MSPS	52
10 X 10	14 MSPS	39	16 MSPS	46	21 MSPS	61
12 X 12	10 MSPS	44	15 MSPS	51	19 MSPS	68
14 X 14	8 MSPS	49	11 MSPS	57	13 MSPS	76
16 X 16	7 MSPS	54	11 MSPS	63	13 MSPS	84

Note: * All figures are approximate



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>

North America

Irvine, California
(714) 727-0780

Englewood, Colorado
(303) 220-7541

Sunnyvale, California
(408) 245-9850

Schaumburg, Illinois
(847) 605-1972

Nashua, New Hampshire
(603) 891-1098

Raleigh, North Carolina
(919) 846-3922

West Chester, Pennsylvania
(610) 430-3300

Dallas, Texas
(214) 960-1043

Europe

Xilinx Sarl
Jouy en Josas, France
Tel: (33) 1-34-63-01-01
Net: frhelp@xilinx.com

Xilinx GmbH
Aschheim, Germany
Tel: (49) 89-99-1549-01
Net: dlhelp@xilinx.com

Xilinx, Ltd.
Byfleet, United Kingdom
Tel: (44) 1-932-349401
Net: ukhelp@xilinx.com

Japan

Xilinx, K.K.
Tokyo, Japan
Tel: (03) 3297-9191

Asia Pacific

Xilinx Asia Pacific
Hong Kong
Tel: (852) 2424-5200
Net: hongkong@xilinx.com

© 1996 Xilinx, Inc. All rights reserved. The Xilinx name and the Xilinx logo are registered trademarks, all XC-designated products are trademarks, and the Programmable Logic Company is a service mark of Xilinx, Inc. All other trademarks and registered trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described herein; nor does it convey any license under its patent, copyright or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. cannot assume responsibility for the use of any circuitry described other than circuitry entirely embodied in its products. Products are manufactured under one or more of the following U.S. Patents: (4,847,612; 5,012,135; 4,967,107; 5,023,606; 4,940,909; 5,028,821; 4,870,302; 4,706,216; 4,758,985; 4,642,487; 4,695,740; 4,713,557; 4,750,155; 4,821,233; 4,746,822; 4,820,937; 4,783,607; 4,855,669; 5,047,710; 5,068,603; 4,855,619; 4,835,418; and 4,902,910. Xilinx, Inc. cannot assume responsibility for any circuits shown nor represent that they are free from patent infringement or of any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made.