Designing FIR Filters with Actel FPGAs

Introduction

Many of the traditional users of HiRel silicon were early adopters of digital signal processing (DSP) applications. In the military-aerospace market, real-time DSP was needed for processing radar and sonar signals. Programmable DSP chips were not yet available, so the early adopters in this segment employed discrete components, or *building blocks* (ALUs, multipliers, registers, etc.), to build their DSP data paths. The development of programmable DSPs made building blocks obsolete, except in cases of sample rates that exceeded the capabilities of the programmable DSPs. In those cases, building blocks could still be employed to perform much of the required computation in parallel.

It is now possible to employ Actel's high-gate-count FPGAs to implement the high-sample-rate DSP applications required by many users of HiRel silicon. Furthermore, Actel's synthesis-friendly architecture makes designing these applications with HDLs (hardware description languages) straightforward and predictable. A big advantage of using HDLs is that the designs can be scalable or parameterizable and can therefore be reused in different applications.

In this application note, two architectures for FIR (finite impulse response) filters are described. The FIR filter is the most common DSP application, and the techniques illustrated in this application note should pertain to other DSP applications as well. A brief note about Two's Complement Arithmetic precedes the discussion about the filters. Schematics and block diagrams are used extensively to illustrate some of the techniques. Some basic knowledge of VHDL is assumed. The VHDL examples are used to illustrate some of the more advanced techniques that may be unique to DSP applications. Finally, some results for various implementations of the filters are presented along with some discussion about the use of the Actel tools.

Two's Complement Arithmetic

In DSP filtering applications, coefficients are made up of both positive and negative numbers. Depending on the application, the data is either all positive or both positive and negative. Two's Complement arithmetic efficiently handles the addition and multiplication of signed numbers.

In an n-bit Two's Complement binary number, the MSB (b_{n-1}) represents $-b_{n-1} * 2^{n-1}$.

The following are Two's Complement representations of various numbers:

-3 101 ($-2^2 + 2^0$) or 1101 ($-2^3 + 2^2 + 2^0$)

3 011 ($2^1 + 2^0$) or 0011 ($2^1 + 2^0$)

Notice that to go from a 3-bit representation to a 4-bit representation of a given number, the MSB is simply duplicated. This is called *sign extension* (Figure 1).

VHDL Example 1: Sign Extension Operator

```
-- The sxt operator takes two arguments. The
-- first is the vector being extended.
-- the second is the word length of the
-- result. <cr>
a(7 downto 0) <= sxt("1100", 8); <cr>
-- assigns "11111100" to signal <a>
```

The advantage of Two's Complement arithmetic is that you can use the same hardware to add negative numbers and positive numbers. The carry-out is discarded.

	0011	(3)
+	1110	(-2)
	0001	(1)

Note, in the following example, that you must sign extend the two addends by one bit more than the MSB to assure that the sum doesn't overflow.

	0011	(3)			011	(3)
+	0011	(3)	not	+	011	(3)
	0110	(6)			110	(-2)

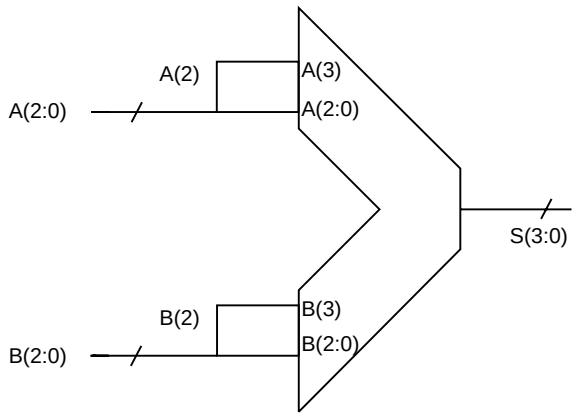

Figure 1 • *Hardware Implemention of Sign Extension*

Subtracting two numbers in Two's Complement arithmetic is carried out by taking the Two's Complement of the subtrahend and adding it to the minuend. In hardware, this can be accomplished by inverting all the bits of subtrahend and setting a carry-in to 1. A carry-in can be emulated by adding an LSB to both the subtrahend and the minuend, setting one of them to 1 and using the other as a carry-in (Figure 2).

	1101	(-3)	1101 1	(-3 and LSB set to 1)
–	0010	(2) is +	1101 1	(inverted 2 and carry-in (LSB) set to 1)
			1011	(–5 and discard the LSB)

In other words, the Two's Complement of a given number is equivalent to inverting all the bits and incrementing by 1. An empirical method of taking the Two's Complement of a number is to invert every bit after the least signficant 1.

0011	(3)	->	1101	(-3)
0100	(4)	->	1100	(-4)

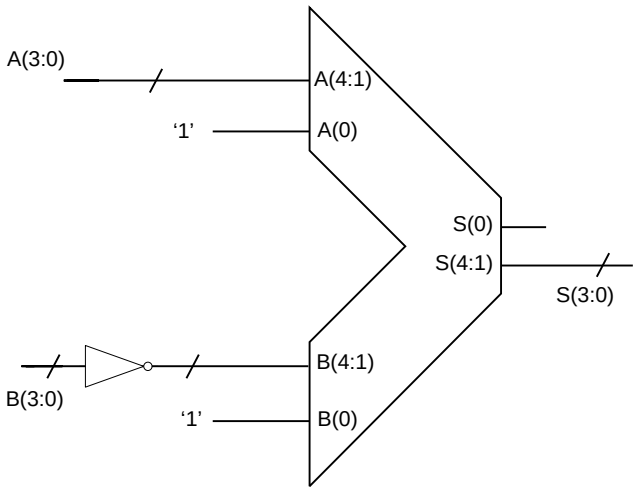

Figure 2 • *Schematic of a Two's Complement Subtractor*

Distributed Arithmetic

The following is the equation of a T-tap FIR filter is:

$$D_{out} = \sum_{i=1}^{T} c_i d_i \qquad (1)$$

where c_1, c_2, … c_T are the filter coefficients and d_1, d_2 … d_T are sequentially stored input data values, or taps (Figure 3).

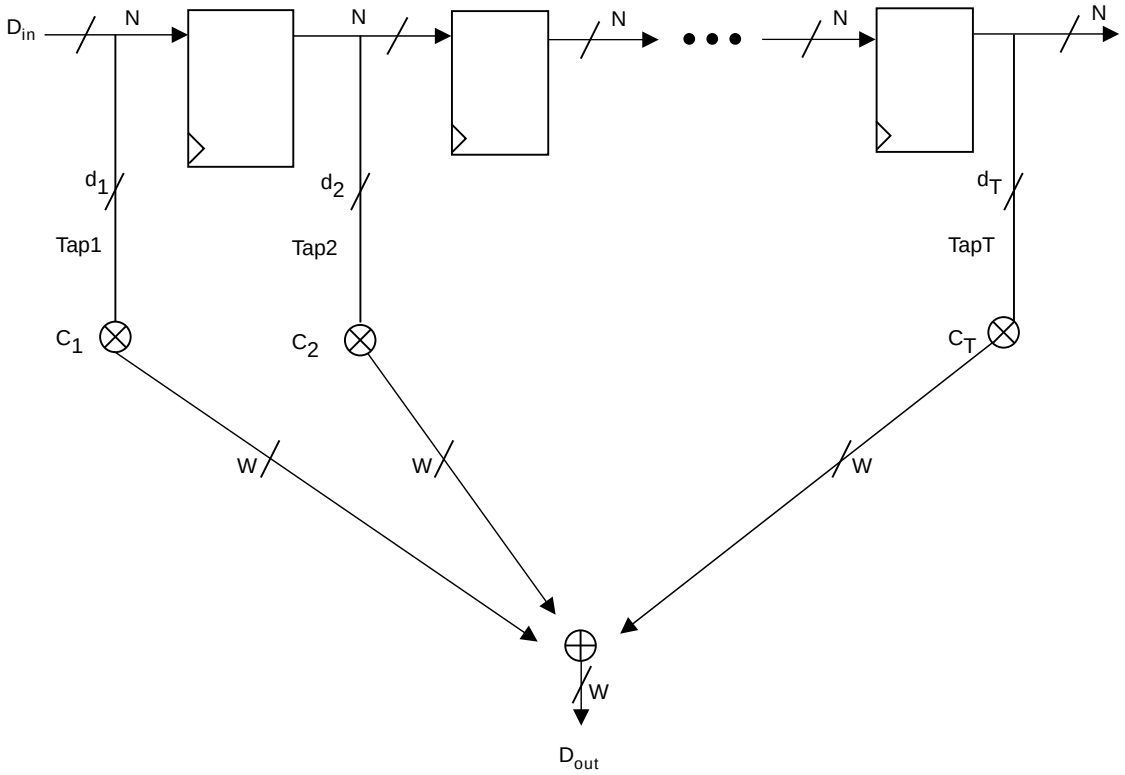

Figure 3 • *Block Diagram for a Generalized T-Tap FIR Filter*

As mentioned, adding some degree of parallelism to the FIR calculation enables filtering applications to be carried out with an FPGA at data rates higher than with a programmable DSP. In the case of a FIR filter, the hardware required for parallelism can be greatly minimized if the filter coefficients are known and constant. If this is the case, multipliers can be replaced by a combination of look-up tables and adders by using the distributed arithmetic technique.

Partial products for each data bit are evaluated for four-tap slices of the filter. If there are N data bits, there are N partial products:

$$p_0 = \sum_{n=1}^{4} c_n d_n(0) \qquad (2\text{-}1)$$

$$p_1 = \sum_{n=1}^{4} c_n d_n(1) \qquad (2\text{-}2)$$

$$p_{N-1} = \sum_{n=1}^{4} c_n d_n(N-1) \qquad (2\text{-}N)$$

where $d_1(n)$ is the nth bit of d_1, $d_2(n)$ is the nth bit of d_2, etc.

Each equation can be implemented by the same hardware in the form of a four-input look-up table (LUT). The input (D) consists of the nth bit each of the four taps in the slice (Figure 4).

D	LUT Output
0000	0
0001	c1
0010	c2
0011	c2 + c1
0100	c3
.	.
1111	c4 + c3 + c2 + c1

A LUT of this nature synthesizes very efficiently in Actel's mux-based architecture. Most often, only one module is required per bit of the result. At the most, two levels of logic will be required.

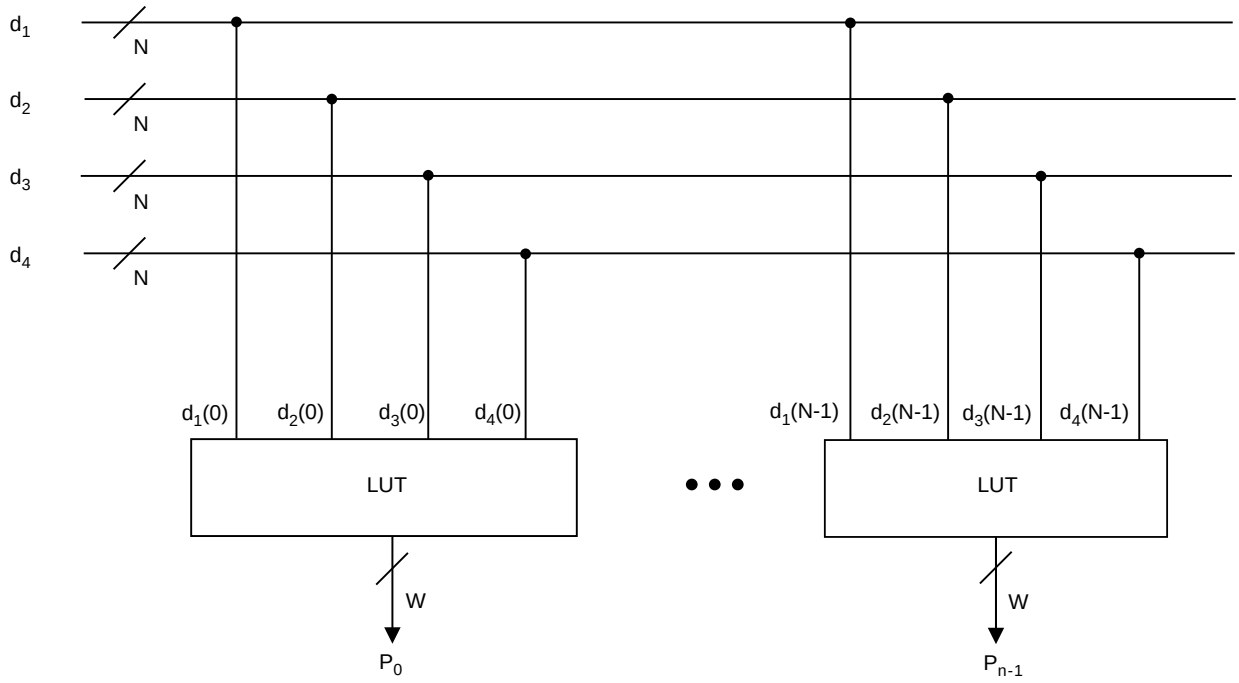

Figure 4 • *Block Diagram for Partial Product Generation for a FIR Filter*

VHDL Example 2: LUT to Evaluate Partial Products for Four Taps

```
Entity PartialProd

  c1 : std_logic_vector := “01001001”;        -- coefficient for tap 1
  c2 : std_logic_vector := “10010101”;        -- coefficient for tap 2
  c3 : std_logic_vector := “10010010”;        -- coefficient for tap 3
  c3 : std_logic_vector := “01101010”;        -- coefficient for tap 4

Port ( D : std_logic_vector(3 downto 0);
       P : std_logic_vector(9 downto 0));     -- 4 * 8 bit coeff => 10 bit product

End PartialProd;

Architecture Behave of PartialProd is

-- Compute all the partial products and store them as constants.

constant v0 : std_logic_vector := sxt(“0”, 10);
constant v1 : std_logic_vector := sxt(c1, 10);
constant v2 : std_logic_vector := sxt(c2, 10);
constant v3 : std_logic_vector := v1 + v2;
constant v4 : std_logic_vector := sxt(c3, 10);
constant v5 : std_logic_vector := v4 + v1;
constant v6 : std_logic_vector := v4 + v2;
constant v7 : std_logic_vector := v4 + v3;
constant v8 : std_logic_vector := sxt(c4, 10);
constant v9 : std_logic_vector := v8 + v1;
constant v10 : std_logic_vector := v8 + v2;
constant v11 : std_logic_vector := v8 + v3;
constant v12 : std_logic_vector := v8 + v4;
constant v13 : std_logic_vector := v8 + v5;
constant v14 : std_logic_vector := v8 + v6;
constant v15 : std_logic_vector := v8 + v7;

Begin

prodeval: process (D)
begin
case(d) is
  when “0000” => P <= v0;
  when “0001” => P <= v1;
  when “0010” => P <= v2;
  when “0011” => P <= v3;
  when “0100” => P <= v4;
  when “0101” => P <= v5;
  when “0110” => P <= v6;
  when “0111” => P <= v7;
  when “1000” => P <= v8;
  when “1001” => P <= v9;
  when “1010” => P <= v10;
  when “1011” => P <= v11;
  when “1100” => P <= v12;
  when “1101” => P <= v13;
  when “1110” => P <= v14;
  when “1111” => P <= v15;

end case;
end process;
end behave;
```

Once the partial products are evaluated, they need to be summed in a manner to produce the final filtered result for the four-tap slice, P_{out}.

$$p_{out} = \left[\sum_{n=0}^{N-2} 2^n \times p_n \right] - (2^{N-1} \times p_{N-1}) \qquad (3)$$

In hardware, the scaling (multiplication by a power of 2) can be accomplished in parallel or serially. In addition, recall that Pout represents the scaled and summed partial products of a four-tap slice. The partial product for the MSB (p_{N-1}) is subtracted from the total because it represents the partial product for the MSB of the data (d^{N-1}), which in turn corresponds to -2^{N-1}.

If the number of taps exceeds four, then those results must be summed as well to generate the final filter output.

$$D_{out} = \sum_{k=1}^{K} P_{out}(k) \qquad (4)$$

where $P_{out}(k)$ is the product of the kth four-tap slice.

Writing Parameterizable VHDL Code

FIR filters are inherently parameterizable. The basic equation (1) is the same for all FIR filters. What differs are parameters such as:

- Number of taps
- Word width of the data
- Word width of the coefficients
- Word width of the result

One can use VHDL Generics to Create the scalable or parameterizable code. One creates a Generic for each of the parameters in the top-level entity. These Generics are initialized at the top level to the value desired. If these parameters are needed in lower level entities, the initialized value can be passed to the lower level entity by using a Generic Map.

VHDL Example 3: Using Generics to Scale Buses

```
Entity FIR

Generic (O_Width : integer := 10;              -- word width of output
         D_Width : integer :=8);               -- word width of data

-- Note that the ports are scaled to the values initialized for the Generics.

Port (Din : in std_logic_vector(D_Width - 1 downto 0);        -- 7 downto 0
      Dout : out std_logic_vector(O_Width - 1 downto 0);      -- 9 downto 0
      Clk : in std_logic);

end FIR;
```

VHDL Example 4: Passing Parameters to Lower Level Entities Using Generics

```
Entity AddOrSubtract is

-- In this example, a Generic is used to pass a control value to a lower level entity.
-- The value of the Generic will control what the lower-level entity does.

Generic(Subract : integer := 0);-- 0 to add, 1 to subtract …

Architecture Behave of AddOrSubtract is …

Begin …

add_sub1 : Add_Sub-- instantiate component “Add_Sub”
  Generic Map (Subtract);-- passes value of Generic “Subtract” to component
  Port Map ( a => a, b => b, s => s); …

end Behave;

entity Add_Sub is

generic (Subtract : integer :=1);       -- initialized value is overridden
-- by value passed from top level
```

The *Generate Statement*, in combination with *Generics*, is used to create scalable hardware. In FIR filters, this is especially useful for creating a register chain that is as long as the number of taps.

In addition to components, the *Generate Statement* can be used in a similar manner to instantiate VHDL processes.

Finally, *Generics* can be used as loop controls for processes. This is a convenient way to generate scalable shift registers.

VHDL Example 5: Using Generics with a Generate Statement

```
Architecture Behave of FIR is …

Begin …

taps :                                  -- need a label for generate statement
for I in 1 to NumTap generate           -- generic “NumTap” is ending value of loop

tapregx : TapReg                        -- instantiating component TapReg

  Port Map ( in <= tap(I - 1),          -- assign signal tap(I-1) to input
       out <= tap(I);                   -- assign signal tap(I) to output
       clock <= clk);                   -- clock is assigned to clk in each instance
end generate;
```

VHDL Example 6: Using a Generic in a Loop

```
process(clk)
begin
if (clk’event and clk=’1’) then
  for I in RegLen downto 1 loop         -- Generic “RegLen” is starting value of loop
    regout(I) <= regout(I-1);
  end loop;
end process;
```

Parallel Summation Architecture for FIR Filters

This implementation of a FIR filter employs adder trees to evaluate the four-tap partial product slice in parallel.

Modifying our example to the case where N (the number of partial products) is 8, Equation 3 can be implemented in an adder tree with 3 ($\log_2 8$) levels (Figure 5).

Having scaled and summed all the partial products for all of the four-tap slices, a standard adder tree is used to produce the final result (Figure 6).

Writing HDL code to implement the adder trees is straightforward. One way to do this is to instantiate ACTgen adders, which are optimized for performance. One can also infer adders by using the HDL operator +. In some synthesis tools (ACTmap and Synopsys), the synthesized result is identical to the ACTgen result.

In many cases, the output of one level of adders must be transformed before input to the next level of adders. These cases include shifting the sum left to multiply by a power of 2, sign extending a sum that is to be added to a shifted sum and truncating a sum, to maintain a maximum internal precision level. The easiest way to transform the sums is with *Concurrent Signal Assignment* statements.

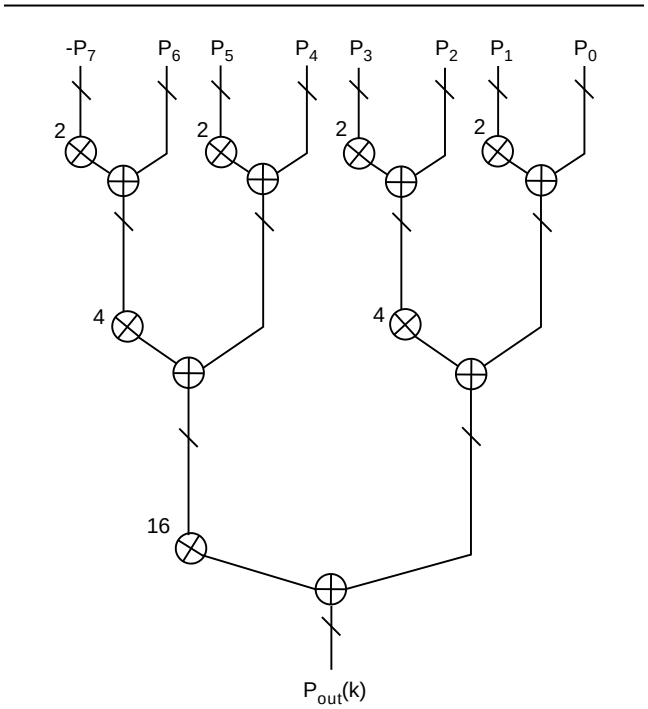

Figure 5 • *Schematic of an Adder Tree Used to Implement Equation 3 for N=8*

VHDL Example 7: Multiplying a Sum by a Power of 2 and Sign Extending a Sum

```
-- multiply by 4 (shift left 2)
a3(7 downto 0) <= s1(5 downto 0), “00”;

-- sign extend
b3(7 downto 0) <= sxt(s2(5 downto 0), 8);

-- truncate by 1 bit
b4(MaxPrec downto 0) <= sxt(s3(MaxPrec downto 1),MaxPrec+1)
```

Note that the size of the adder used in the truncate example is (MaxPrec + 1). This is because once the maximum precision has been reached, the output of the adder must be one bit longer than the inputs to guarantee that there is no overflow. The LSB of the sum is then truncated to maintain the precision to MaxPrec.

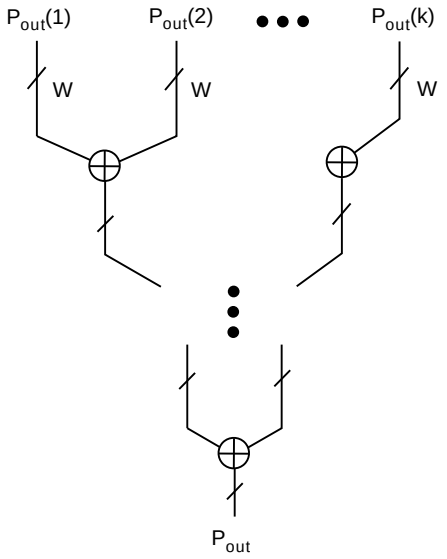

Figure 6 • *Adder Tree to Sum the Partial Products of the four-tap Slices*

Pipelining for Maximum Performance

The structure of a parallelized FIR filter lends itself naturally to pipelining. Pipelining enables one to run the circuit at a higher sample rate, at the cost of delaying the result by a number of clock cycles. In most DSP applications, this delay is not a problem. The natural pipelining stages include every stage of the adder trees and the output of the look-up tables used to generate the partial product terms. Creating these pipeline stages using HDL code is trivial.

VHDL Example 8: Creating Pipeline Stages

```
-- for an adder tree stage
process(clk)
begin
if (clk’event and clk=’1’) then
    s <= a + b;
end if;
end process;
```

Serial Summation Architecture for FIR Filters

At the expense of a lower sample rate, the hardware required to implement a FIR filter in an Actel part can be greatly reduced by performing the partial product summation serially. In the Parallel Summation architecture, the result for a four-tap slice requires N look-up tables where N is the word length of the data, and an adder tree to sum and scale the N partial products. The Serial Summation architecture takes advantage of the fact that each of the N look-up tables is identical.

The first step to implement the Serial Summation architecture is to serialize the data stream by using a simple parallel-to-serial shift register. This is easily created by using either ACTgen and instantiating it or by just coding it directly.

The serialized data is then fed to a serial chain of T (number of taps) N*1 serial shift registers (Figure 7). The LSB of each of the T registers is then fed to the look-up tables used to evaluate the partial products. In this manner, the partial products (p_n) for all of the taps are evaluated at one time for a single data bit, starting with the LSB.

Next, all the partial products of the four-tap slices for bit n of the data are summed in an adder tree (Figure 8). Equations 3 and 4 are thus combined to become the following:

$$D_{out} = \sum_{n=0}^{N-1} 2^n \times P_n \qquad (5)$$

where $P_n = \sum_{k=1}^{K} p_n(k)$ and, in turn,

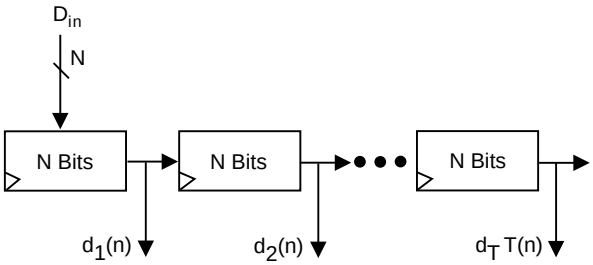

Figure 7 • *Register Chain for Serial Summation Architecture*

where $p_n(k)$ is the partial product for the nth data pit of the kth four-tap slice.

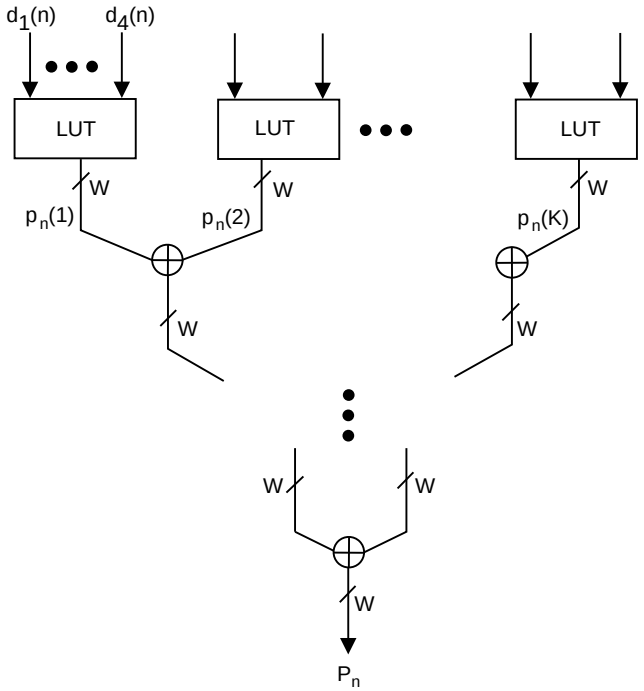

Figure 8 • *Scaling Accumulator*

The sum P_n is then fed to a scaling accumulator.

The scaling accumulator shifts its output right by one bit before feeding back, effectively multiplying the input by 2 (Figure 9). The structure here requires additional circuitry to subtract P_{N-1} and to reset the accumulator without losing a clock cycle. The LSB of the result can be stored in an overflow register to increase the precision of the output.

Therefore, in the Serial Summation architecture, N clock cycles are required per sample. As with the Parallel Summation architecture, it lends itself to pipelining.

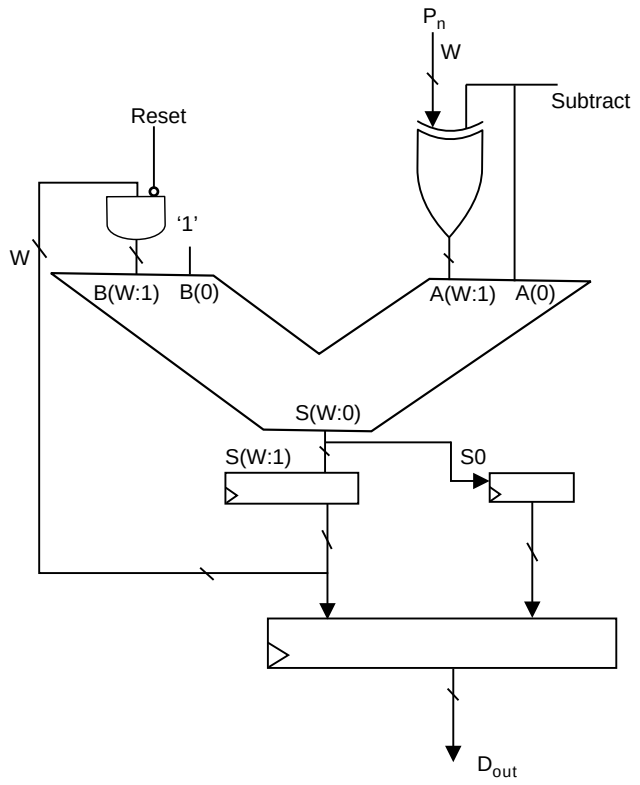

Figure 9 • *Scaling Accumulator*

Using the DX Dual-Port SRAM as a Serial Register

FIR filters consume a large number of registers (N bits * T taps). This is particularly onerous in any FPGA architecture because the logic in front of the flip-flops is wasted. The dual-port RAM in the 3200DX family can be used in place of registers for shifting the data, freeing up the other flip-flops for pipelining and other applications.

A DX RAM block is 32 * 8 . If it is used as a serial register in a FIR filter, it can store eight taps, which can contain word lengths up to 32 bits. After the n^{th} word of the t^{th} tap is read, it is written to the n^{th} word of the $(t+1)^{th}$ tap. The read address can therefore be controlled by a counter that counts to N-1. To generate the write address, the read address is simply delayed by one clock cycle.

The basic synchronous DX RAM block can be generated using ACTgen and then instantiated in the HDL code.

Symmetric and Antisymmetric Filters

In most cases, filter coefficients have symmetry (Figure 11 and Figure 12), and if they do, hardware requirements can be greatly reduced by adding (subtracting in the antisymmetric case) the two symmetric taps together before evaluating the partial product, reducing by half the number of look-up tables and adder trees.

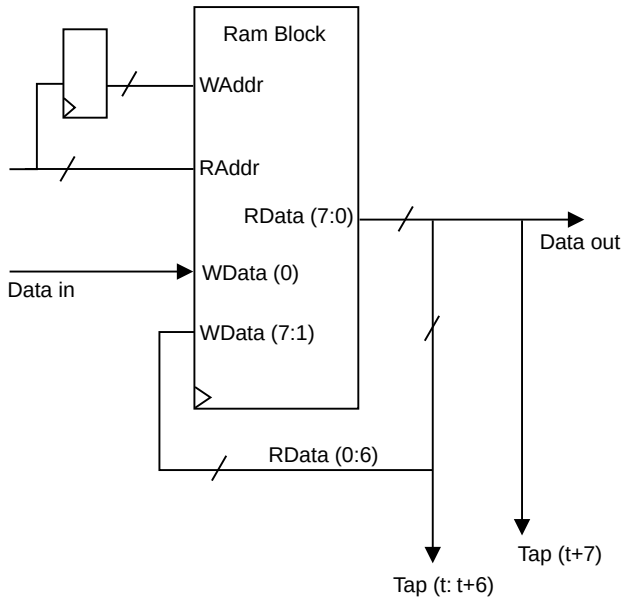

Figure 10 • *The Dual-Port RAM Used as a Serial Register*

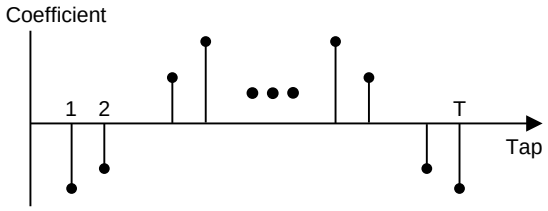

Figure 11 • *Symmetric Coefficients*

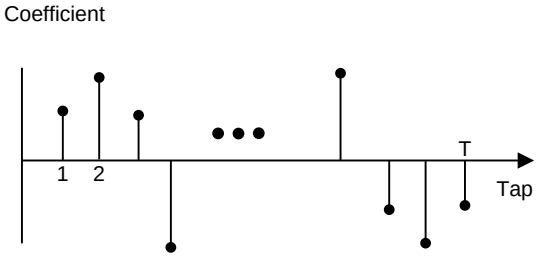

Figure 12 • *Anti-symmetric Coefficients*

In the Parallel Summation architecture, the addition is straightforward for the symmetric case. One way to handle the anti-symmetric case is to pass the data through a Two's Complementer at the point of symmetry (after tap T/2 if T is even, and after tap (T+1)/2 if T is odd).

In the Serial Summation architecture, a serial adder is fairly straightforward to implement (Figure 13) but it has the effect of adding an extra clock cycle per sample because adding two N-bit numbers yields an (N+1) bit result. (Note in Figure 13 that the Actel macro FA1A is instantiated. This is to guarantee a minimum delay.) The data register must therefore halt for a cycle when the MSB is being process by the serial adder, effectively sign extending the data. For the antisymmetric case, a serial Two's Complementer can be used at the point of symmetry. This can apply the empirical Two's Complement algorithm, which is to invert every bit after the least significant 1 (Figure 14).

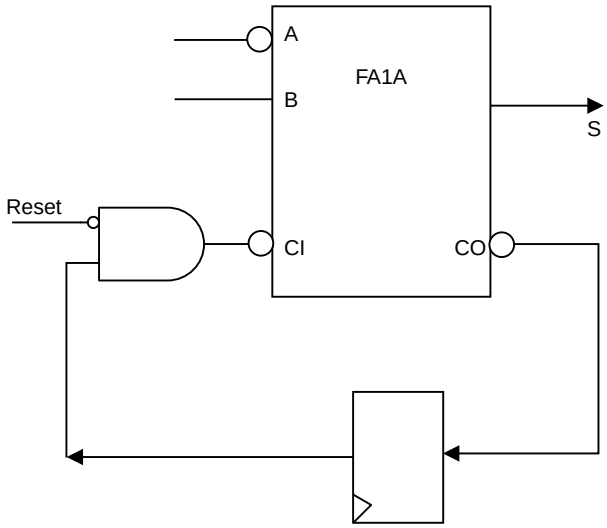

Figure 13 • *Serial Adder*

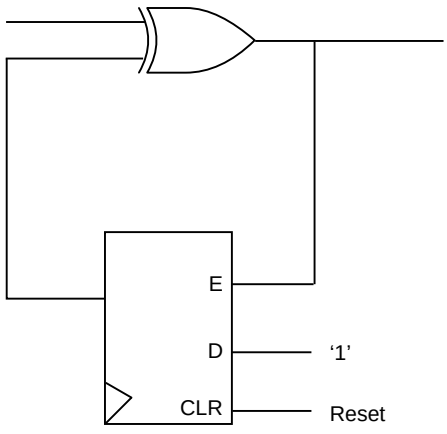

Figure 14 • *Serial Two's Complementer*

Controlling the Serial Summation Architecture

There are several control signals required for the Serial Summation architecture. They include the Raddr and Waddr of the RAM, the reset for the serial adder, and the reset and subtract for the scaling accumulator. Designing the data path for the filter is fairly straightforward. The control logic is usually the most time consuming.

In general, a counter will be needed to generate the addresses. The rest of the control signals can be generated by a state machine that uses the counter as input. Here are a few guidelines that may ease the design of the control circuitry:

- Put all control circuitry, except for the counter in a lower-level entity. Use the counter as an input to the control block and all the control signals as the output (Figure 15).
- Inside the control block, feed all of the control signal into a scalable pipeline register. The length of the register should be passed into the block as a *Generic* and will be dependent on the number of pipeline stages in the filter, which is generally dependent on the number of levels in the adder trees. Reference each control signal to the *Generic* that is passed. In this way, if the number of pipeline stages changes, one only has to change the value of the *Generic* in the top level, and not the control logic.

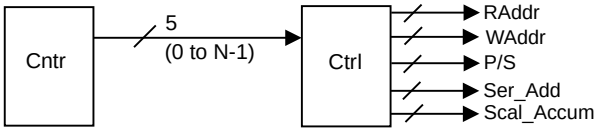

Figure 15 • *Block Diagram of the Control Logic*

VHDL Example 9: Pipelining Control Signals

```
-- Concatenate all the control signals together to create a control bus.
ctrl(0) <= acc_sub & acc_rst & ser_add_rst & cntr_en & par_ser_ld;

process(clk)
begin
if (clk'event and clk='1') then
  for I in stages downto 1 loop          -- Generic "stages" starts loop
    ctrl(I) <= ctrl(I+1);
  end loop;
end process;

acc_sub_out <= ctrl(stages)(4);          -- accumulator signals depend on Generic "stages"
acc_rst_out <= ctrl(stages)(3);
ser_add_out <= ctrl(3)(2);               -- last 3 signals have fixed location wrt to beginning
cntr_en_out <= ctrl(2)(1);
par_ser_ld <= ctrl(1)(0);
```

Generating Results

Once the scalable code has been created, producing FIR filters with different parameters is straightforward. One simply needs to import the synthesized netlist into the Actel tools and run Place & Route. If the filter sample rate is known, just enter the period in the DirectTime Editor and run Place & Route using the DirectTime option. Timing problems are also straightforward to sort out for the filter using the DirectTime Analyze tool. In general, the Serial Summation architecture is limited by the scaling accumulator, which requires one additional level of logic more than an adder of the same size. At times, some of the control signals will also be limiting because they have to be distributed throughout the array. If necessary, results can be improved for the control signals by adjusting the synthesis timing parameters. The Parallel Summation architecture is balanced and will probably be limited by one of the adders.

Table 1 shows some of the results that can be achieved for several embodiments of both the Parallel Summation architecture and the Serial Summation architecture. As a point of reference, today's state-of-the-art programmable DSPs run at about 10ns per tap (i.e., 10 Tap filter => 10 MHz sample rate).

Conclusions

Using the techniques described in this application note, one can efficiently design parameterizable FIR filters that can perform at sample rates greatly exceeding those of a state-of-the-art programmable DSP. These techniques include the following:

- Employing Two's Complement arithmetic for any application using both positive and negative integers

- If coefficients are known, synthesizing LUTs and adder trees in place of multipliers to conserve valuable logic resources
- Trading off performance of a DSP algorithm with utilization by altering the degree of parallelism in the data path
- Pipelining a DSP datapath
- Parameterizing FIR filters in VHDL by using Generics and the Generate statement
- Employing the Dual Port RAM in the 3200DX devices in a Serial Summation architecture as the data registers
- Further reducing logic resources by taking advantage of symmetry in the coefficients
- Implementing with the straightforward and predictable Actel tools.

With these capabilities, a DSP application that, in the past, may have been implemented with building blocks can now be integrated onto a single Actel FPGA. The FPGA solution offers complete flexibility in the design and, by reducing chip count, improves the overall reliability of the system.

Table 1 • *FIR Filter Results*

Taps (T)	DataWidth (N)	Max Precision (W)	Architecture	Part	Utilization		Sample Rate (MHz)	
					S Mods	Total Mods	–3 WC Comm	–1 WC Mil
32	8	12	Serial	A32200DX	138	275	5.6	3.9
32	12	12	Serial	A32200DX	222	486	3.7	2.6
64	8	16	Serial	A32200DX	450	968	5.6	3.9
64	12	12	Serial	A32200DX	408	872	3.5	2.4
8	8	10	Parallel	A14100A	227	505	66	46.5
8	8	17	Parallel	A14100A	268	565	67.6	47.6
16	8	10	Parallel	A14100A	451	997	66.2	46.6

Take it to a higher level.

http://www.actel.com

Actel Europe Ltd.
Daneshill House, Lutyens Close
Basingstoke, Hampshire RG24 8AG
United Kingdom
Tel: +44(0).1256.305600
Fax: +44(0).1256.355420

Actel Corporation
955 East Arques Avenue
Sunnyvale, California 94086
USA
Tel: 408.739.1010
Fax: 408.739.1540

Actel Japan
EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ka
Tokyo 150 Japan
Tel: +81.(0)3445.7671
Fax: +81.(0)3445.7668

5102645-0