Implementation of Digital Filters in Programmable Logic Devices

Class #: 551 Embedded Systems Conference, San Francisco, Spring 2001

David J. Farrell, David M. Williams
Colorado Electronic Product Design, Inc.
http://www.cepdinc.com/

Abstract

Recent strides in programmable logic density, speed and hardware description languages (HDL's) have empowered the engineer with the ability to implement digital signal processing (DSP) functionality within programmable logic devices (PLD's or FPGA's). This paper, written for the intermediate DSP engineer or logic designer, begins with an overview of general DSP concepts. Filter design principles as well as specific DSP filter architectures are presented including serial and parallel architectures. Techniques for implementing DSP filters in FPGA's using VHDL are discussed. Methods of exploiting specific FPGA architectures in order to enhance performance in terms of speed, area and power consumption are presented. Some guidelines are given for evaluating different programmable logic device architectures for DSP designs. The paper culminates with a specific IIR filter design implemented in an FPGA using VHDL.

Introduction

When implementing a digital filter the embedded system designer is faced with two choices: using a dedicated DSP processor or using a hardware approach with either a programmable logic device (PLD) or ASIC. This paper provides an overview of how digital filters are implemented in programmable logic devices. Implementing digital filters in hardware can have distinct advantages over a dedicated processor approach. As IC processing improves and strives to meet Moore's Law, programmable logic has improved in both cost and speed, making it a viable alternative for implementing digital filters.

Not only have the devices improved but the method of designing with programmable logic has become more efficient with the advent of hardware description languages (HDL). HDL's offer an alternative to the schematic based approach to logic design, where logic gates or registers are connected in schematic fashion. HDL's are a text based design entry method that allows the designer to describe the logic function at a higher level of abstraction thereby increasing efficiency. The two primary HDL's used today are Verilog and VHDL, both IEEE standards. Both of these HDL's (Verilog /VHDL), can be used to implement digital filters and it is not our intention to argue the relative merits of each language. More specifically, this paper discusses how to implement digital filters in programmable logic using VHDL.

Why use Programmable Logic for Digital Filters?

Since PLD's are dedicated hardware, they can achieve significant performance increases over a DSP processor. Other advantages include reduced power consumption. Although dedicated DSP processors offer the most flexibility they can require extra clock cycles compared to a hardware implementation, and this can be power inefficient. Since many embedded systems already have some type of programmable logic on the system, the PLD may have the space available for a digital filter. If this extra logic space is not available, the embedded designer may be faced with porting all the firmware over to a dedicated DSP processor. A better alternative might be to increase the size of the programmable logic device to accommodate a digital filter. If the PLD is already in the data path of the embedded system, the latter approach may be easier than initiating a new hardware design with a dedicated DSP processor.

Many embedded systems have both a dedicated DSP processor with a PLD device. In over sampled DSP systems, where the data arrives at a high rate, a PLD can down sample the data prior to the DSP processor. Efficient techniques can be employed to simultaneously accomplish polyphase quadrature demodulation and down sampling [15]. When decimation is required, the PLD can off-load some of these processing tasks by performing both filtering and decimation before data is transferred to the DSP for further processing at a lower sample rate.

Some of the large programmable logic vendors include papers comparing both FPGA (a type of PLD) and DSP processors. We include two references that compare dedicated DSP processors with FPGA's [2, 9]. Although we have found that the PLD vendors will have strong arguments in favor of programmable logic for DSP functions, there are relative pros and cons with utilizing either a dedicated DSP processor, programmable logic or both in an embedded system.

Basic's of Programmable Logic

PLD's realize logic functions by interconnecting predefined hardware logic resources such as gates and registers on an integrated circuit. Basic combinatorial logic functions such as AND and OR gates are easily implemented and for sequential logic functions, registers are commonly supported as flip-flops.

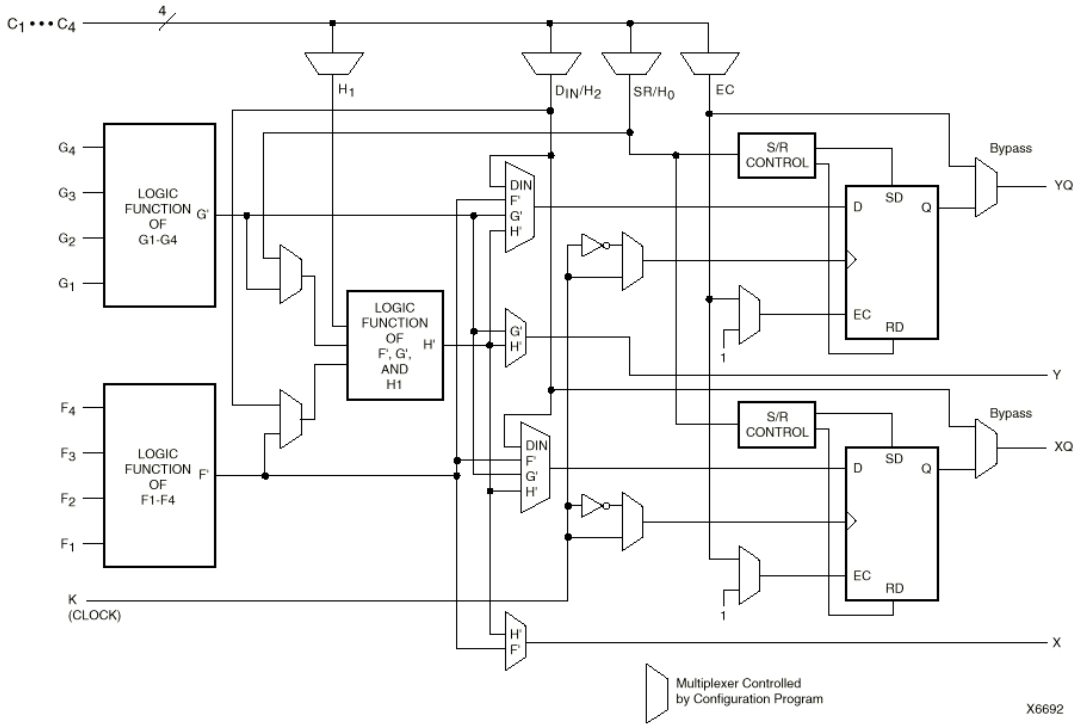
In some programmable logic devices, combinatorial logic functions are implemented in a small memory or look-up table (LUT). For example, a four input combinatorial function could be implemented in a 16-bit RAM memory or LUT. This is equivalent to a four input Karnaugh mapping. To generate large complex logic functions, many LUT's are distributed throughout the device. LUT's can be connected via global routing channels and large fan-in signals can be realized. For a counter or state machine logic function, the outputs of LUT's can drive a flip-flop to realize sequential logic functions.

Comparison of CPLD's and FPGA's

Programmable logic devices are usually segregated into two categories: Complex Programmable Logic Devices (CPLD) and Field Programmable Gate Arrays (FPGA). CPLD's are built in a non-volatile memory type technology and FPGA's are usually constructed in a volatile memory similar to a SRAM. CPLD's can be programmed once and maintain their configuration permanently but for most FPGA's, the configuration is loaded upon power up and is lost when the system is powered down. FPGA's are generally much denser and are more register rich than CPLD's. For this reason, FPGA's are better suited for implementing DSP design blocks than CPLD's. Although CPLD's are usually faster than FPGA's (less propagation delay), they are not available in large densities.

FPGA Architectural Structure

An example of a FPGA structure on a typical Xilinx FPGA is shown in Figure 1. We highlight both Xilinx and Altera FPGA architectures here since these companies are two of the larger FPGA suppliers. The four input LUT is a small memory and each LUT output can be steered (via MUX's) to the flip-flops if needed. In Xilinx FPGA's, the basic logic unit is called a CLB (configurable logic block).



Simplified Block Diagram of XC4000 Series CLB (RAM and Carry Logic functions not shown)

Figure 1. A typical Xilinx CLB from the Xilinx XC4000 data sheet [10]

On a typical Altera EPLD (similar to a FPGA), a similar structure is denoted a logic element or LE. Groups of LE's are referred to as a logic array block or LAB.

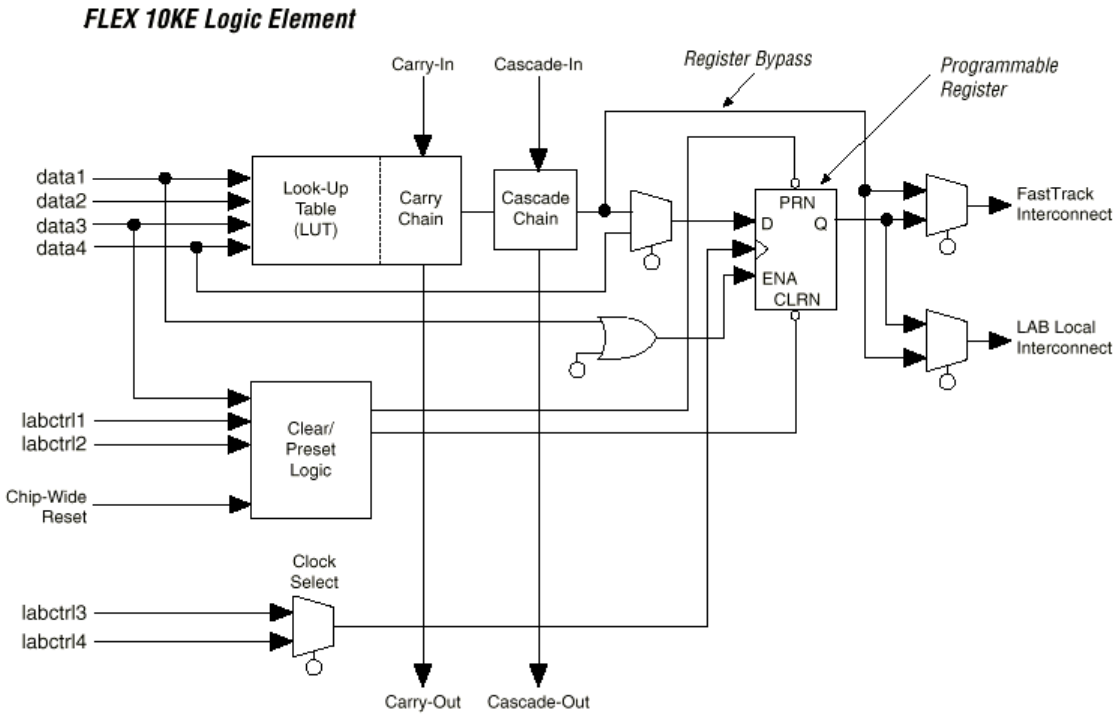


Figure 2. A typical Altera Logic Element from the Flex 10KE Data Sheet [11]

LUT's are used for combinatorial logic functions and flip-flops support the sequential logic functions. As in any embedded system, hardware resources are scarce and designing with FPGA's is no exception. CLB's or LE's are limited in number and it is the responsibility of the designer to utilize these resources effectively. Typical FPGA's can have from hundreds's to many thousands of CLB's or LE's.

Clock Integrity -An Important Design Issue in FPGA Design

In any sequential design that utilizes registers, the clock timing is critical. Any delay in the clock arrival time at a register can create glitch conditions on the register outputs. FPGA's attempt to solve this problem by offering global routing channels where the clock has equivalent access to each flip-flop to minimize any differences in clock arrival time. When designing with FPGA's it is crucial for the designer to exploit these global clock resources. External clock sources should be placed or routed onto these internal global routing channels.

Design Entry Methods with Programmable Logic

The design flow or method of designing with programmable logic is for the engineer to write the design equations in an HDL, make pin assignments, and the logic compiler to translate the HDL design description into a type of netlist. The complete process is referred to as synthesis. The netlist is then translated to the specific PLD by interconnecting all the available resources (gates and registers). Of course one of the main motivations for using an HDL is to free the engineer from the manual job of hand connecting the individual logic gates and registers on a schematic. Ideally, it is desirable to write the design in a higher level of abstraction and let the synthesis tool fit the design to the specific architecture of the FPGA. Although this is a worthy goal, it is not always achieved and various styles of HDL designs may be written to better utilize a specific FPGA architecture. Some designs may be written at a higher level of abstraction, usually referred to as behavioral style of modeling, where other HDL designs may be written to better

emulate the basic logic primitives and referred to as structural style of modeling. Often to get the best performance from an FPGA, the designer must check the connections made in the device to the intended design. Most synthesis tools output a variety of report files allowing the designer to check the physical placement of the design in the part. If the placement is not satisfactory, the design can be changed and the synthesis repeated and the routing or connection is verified for the intended design function. Designs can be optimized for area efficiency or for best speed performance. It would appear that a design that fits into the smallest area would also have the fastest performance, but this is not always the case.

At times, it may be necessary to control the placement of logic on the chip. Most synthesis tools can accept either timing constraints or physical placement constraints. As an example of a timing constraint, the synthesis tool could be constrained to synthesize logic for a particular logic path to have a propagation delay not to exceed a maximum time. Alternatively, another example would be for a logic function to be constrained to a specific location on the chip. These types of operations are referred to as "Floorplanning".

Once the synthesis is complete, the output from this process is a binary file (or configuration) that is to be loaded into the FPGA. Most FPGA's have a variety of configuration options including loading from a microprocessor bus, loading from a small configuration ROM or the configuration can be loaded from the PC's parallel or serial port into a dedicated port on the FPGA device (in-circuit programmable).

Design Verification

A common practice is for the design to be simulated before testing on the hardware. A new alternative to simulation is now available to the engineer. Recently some of the FPGA vendors have offered on-chip debugging of FPGA's, but this feature is only available on the larger devices. These on-chip tools emulate a logic analyzer embedded into the FPGA device, allowing the engineer to probe the actual timing of internal signals. Although this could prove to be very useful in the larger devices, simulation is still the predominate method of initial design verification.

Concurrent Operation-An Important Idea in Programmable Logic

Designers experienced with Ada/C/C++ languages will notice how HDL's look very similar. It can be very easy for an experienced embedded programmer to erroneously assume that a particular HDL design is always executing in a sequential fashion. HDL modules or statements can have concurrent execution (simultaneous operation). This is a very important distinction between programming languages like C and a HDL design executing in a FPGA. The ability to do many things in parallel is a distinct advantage that programmable logic has over a dedicated DSP. It is the logic designer's responsibility to write the VHDL logic to infer either concurrency or sequential logic.

IP Core Considerations

In order to save time in the development of FPGA based digital filters, project managers may choose to purchase or license an existing filter design in the form of an intellectual property core. Other alternatives include contracting out the design work, or completing the design in-house. There are cost, time, and support considerations to be addressed when making such a decision. This paper presents FPGA filter design principles to allow engineers and managers to gain insight into what circuitry may be hidden inside an IP core and the scope of such a development.

Fundamentals of Digital Filters

Many books have been written describing the fundamentals of digital signal processing, the sampling theorem and the Z transform [8, 15]. We provide only a brief overview here. Digital filters are classified as finite impulse response or moving average (FIR, MA) and infinite impulse response or autoregressive moving average (IIR, ARMA).

The difference equations for both types of digital filters are given in equations (1) and (2),

$$(1) \qquad y_n = \sum_{k=0}^{N} b_k \bullet x_{n-k} - \sum_{k=0}^{N} y_{n-k} \bullet a_k \qquad \textbf{IIR Filter}$$

$$(2) \qquad y_n = \sum_{k=0}^{N} b_k \bullet x_{n-k} \qquad \textbf{FIR Filter}$$

with Z transformed transfer functions given in equations (3) and (4).

(3)

$$\frac{Y}{X} = \frac{\sum_{k=0}^{N} b_k \bullet z^{-k}}{\sum_{k=0}^{N} a_k \bullet z^{-k}} \qquad \textbf{IIR}$$

(4)

$$\frac{Y}{X} = \sum_{k=0}^{N} b_k \bullet z^{-k} \qquad \textbf{FIR}$$

Where ak and bk are the filter coefficients, integer n is the discrete-time increment, y[n] the output and x[n] the input of the filter.

Some of the more significant differences between IIR and FIR filters include:

1. FIR filters require substantially more filter coefficients or taps than IIR for equivalent responses.
2. An FIR filter can be designed to have a linear phase response.
3. Certain types of FIR filters can be configured in a symmetrical structure saving computational resources. Symmetrical even order FIR filters will require half the number of multipliers and consume less logic space. See Figure 5.
4. An IIR filter contains poles. Stability is determined by insuring that the magnitudes of the roots of the denominator are less than unity, i.e. the poles are within the unit circle.
5. FIR filters implemented directly as in equation (2) are stable.
6. IIR filters can directly emulate analog filters. Standard transformations can be used to convert Butterworth, Bessel, and other conventional analog filters into digital filters.

Some General Guidelines for Using Either FIR or IIR Filters

Jervis [8] recommends using IIR filters when high throughput and a sharp cutoff are required. FIR filters are preferred when linear phase is a requirement and the filter order is a relatively small number.

Steps in Digital Filter Design

The basic digital filter design steps [8] are provided below and many of these steps are automated with a good filter design software package.

1) Specification of filter requirements
2) Calculation of filter coefficients
3) Representation of the filter by a suitable structure
4) Analysis of stability and finite wordlength effects
5) Evaluate the frequency response with the Z transform
6) Implementation of the filter in either hardware or software.

Digital Filter Structures

Most digital filters can be broken down into three building blocks. These fundamental blocks are the unit delay, the summing node and the multiplication node (multiplier). The combination of a multiplier and summing node is referred to as a multiply and accumulate (MAC) block.

The unit delay (also called a filter tap) is implemented by recalling an old value from memory storing the new value into memory and waiting T seconds, where T is the sampling period. The Z transform of the unit delay is z^{-1} which can also be expressed as $e^{-j 2\pi f T} = \cos(2\pi fT) - j \sin(2\pi fT)$, where $j^2 = -1$ (Euler's identity).

The summing node is just simple addition. The multiplier can be used to multiply two signals or to multiply a signal by a coefficient. These blocks are represented graphically (Figure 3.) with signal flow diagrams (SFD) or signal flow graphs (SFG) [8].

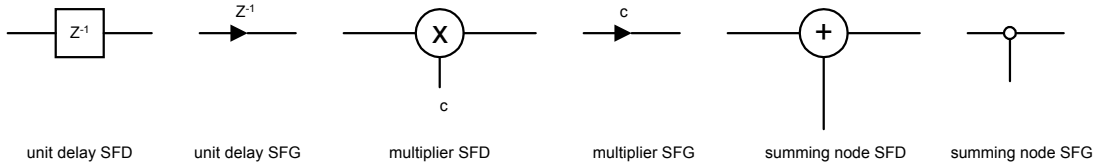


Figure 3. Graphical DSP building blocks

The signal flow diagram of a FIR filter is shown below:

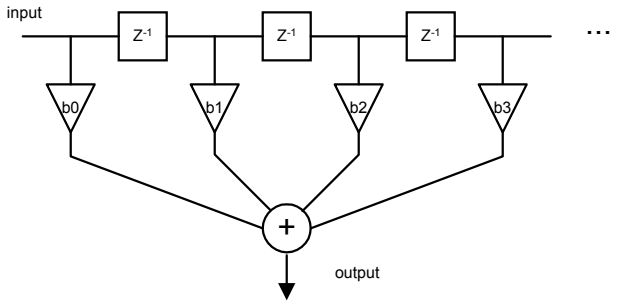


Figure 4. FIR SFD

An FIR filter can be optimally designed to be symmetrical [2]. The summing node is separated to yield the regular repeated structure shown in Figure 5.

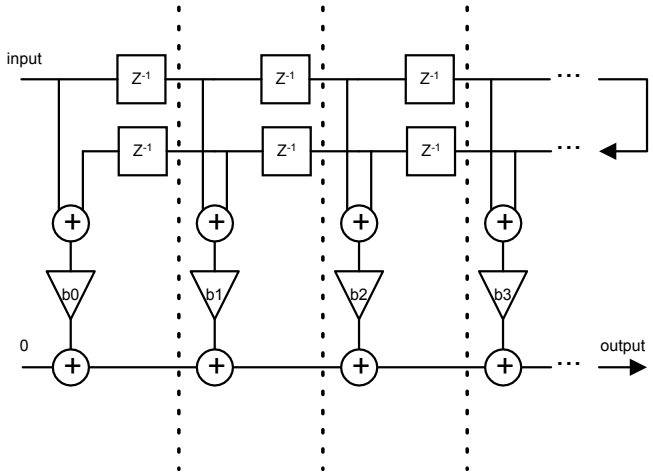


Figure 5. Symmetrical FIR

IIR filters are typically implemented using cascaded second order stages called biquads. Biquads can be implemented in many forms [8]. The direct form II (Figure 6.) is commonly used because its implementation requires one tap per filter order.

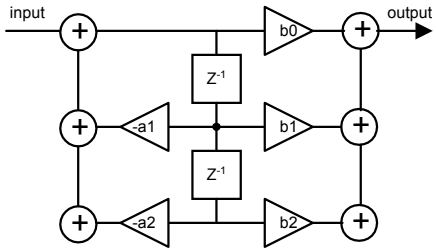


Figure 6. Direct Form II IIR SFD

Fixed-Point Arithmetic

Implementations of fixed-point math functions require less space in an FPGA compared to floating point math. For this reason, fixed-point is used almost exclusively when implementing DSP functionality in FPGA's. Digital signal processors are available in both floating-point and fixed-point. The fixed-point DSP's are usually less expensive and faster than their floating-point counterparts so this situation is not unique to FPGA signal processing designs.

There are two commonly used fixed-point formats, 2s complement and signed-magnitude [12]. In either case, an additional bit is required to represent negative numbers (due to the range including twice as many numbers as the positive set). In 2s complement, negative binary numbers are represented as the inversion or complement of the positive representation plus 1. For example, a 12 bit representation of the number –6 is 1111 1111 1010. As a check, if you add 6 (0110) to this number you get 0000 0000 0000 plus a carry, which is thrown away. The advantage to using 2s complement is that addition and subtraction operations are implemented with a standard adder or ALU.

In signed magnitude format, a sign bit or bits is used to indicate that the number is negative. For example, a 12 bit representation of the number -6 is 1000 0000 0110. The advantage to signed magnitude format is that the signed bit can be removed easily and multiplication can be implemented utilizing one less bit for each of the two arguments. This results in reduced logic, even though the sign must be computed and restored in the product.

For a DSP implementation, it is desirable to represent numbers that have both a whole part and a fractional part. Typically, we view the signal s as a whole number and the coefficient c as having a whole part and a fractional part. A commonly used representation of the coefficients is called the Qn format [14]. In Qn format, the number n represents the number of binary digits to the right of the binary point. In other words, the size in bits of the fractional part of the number. The 12 bit number 6.1 in Q0 format is 0000 0000 0110. The 12 bit number 6.1 in Q8 format is 0110 0001 1001, which actually equals 6.09765625 due to coefficient quantization. Notice that to arrive at this representation, we simply multiplied $6.1 * 2^8 = 6.1 * 256$. In general, it is wise to use the largest number of fractional bits to represent your coefficients, while being careful that the whole part of the number has enough bits to handle the expected range of the coefficients. For example, the 12 bit number 1.918 could be represented best in Q10. $1.918 * 2^{10} =$ 0111 1010 1100. In fixed-point arithmetic, one trades accuracy and range as shown in Table 1.

12 bit Format	Represents	Range	Approximate resolution
Q0	Whole numbers	-2048 to 2047	1
Q1	Fractional Numbers	-1024 to 1023.5	0.5
Q10	Fractional Numbers	-2 to 1.999	0.001
Q12	Fractional Numbers	-0.5 to 0.4999	0.00024

Table 1. Fixed-point Representation Examples

In fixed-point arithmetic, we must make sure numbers are in the same Q format prior to addition or subtraction. For DSP applications, we can think of multiplication as being a Q0 number (the signal) times a Qn (a coefficient) number, yielding a Q0 result (scaled signal). For a 12 bit Qn multiplication process, we first multiply the signal by the coefficient, yielding a 24 bit result. Only 12 bits of the product are returned. The 12 bits, which are returned, are the ones produced if the product was shifted right by n bits. As an example, let us take the Q10 coefficient 1.414 multiplied by a signal of 46. We expect a result of 65. 1.414 in Q10 is 1448. 1448 * 46 = 66608 = 0000 0001 0000 0100 0011 0000. Shifting right by 10, we have 0000 0100 0001 which is 65.

Bit Serial and Digit Serial Multiplication

The fastest method for performing multiplication is to implement full parallel multipliers. These multipliers are straightforward to implement in VHDL, since the function is represented by a single asterisk. Unfortunately, this simple operator causes a large amount of FPGA logic to be consumed for each multiplier. There are ways to trade speed for space by utilizing serial multipliers. This topic has been the result of much study [1, 6, 7]. There are many forms of bit serial and digit serial multipliers. One example of each is presented to demonstrate the principles of these designs. The examples presented here are for instructional purposes and are not claimed to be optimal.

In one type of a bit serial multiplier, each bit of the coefficient is multiplied by the entire signal. The signal is shifted left one bit and the coefficient is shifted right one bit. When the coefficient bit is 1, the shifted signal is added to the accumulator. If the coefficient bit is 0, the accumulator is unchanged. After all of the bits have been shifted, multiplied by the signal and accumulated, the product is complete. The process takes n clock cycles, where n is the number of bits in the coefficient. The process computes the equation (5).

$$p = \sum_{i=0}^{n} c_i \bullet s \bullet 2^i \qquad (5)$$

A block diagram of the circuit is shown in Figure 7.

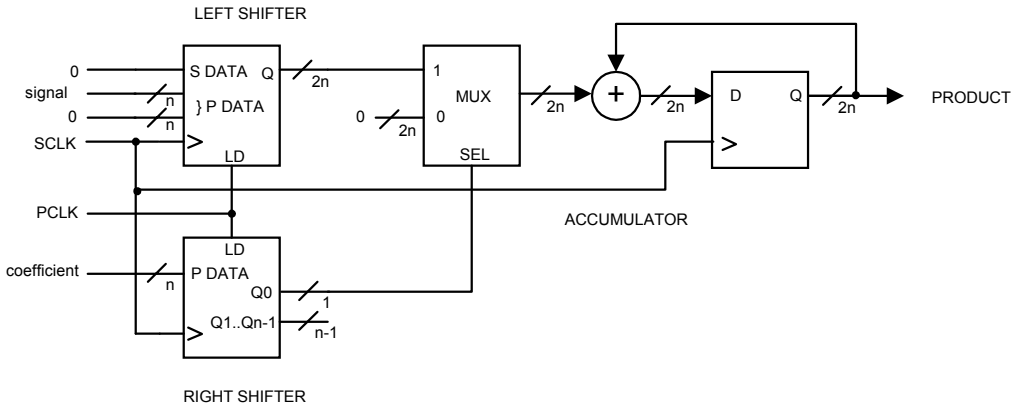


Figure 7. A Bit-Serial Multiplier Form

In Figure 7, the coefficient and signal can be swapped. In many DSP applications, the coefficients are constant. The logic required to implement bit serial multipliers can be reduced when the coefficients are constant and known in advance [4].

The implementation of digit serial multiplication is different from bit serial, in that the signal and coefficients are each broken down into multi-bit digits. For our example, we'll present a 12 by 12 multiplier using 4 bit digits. The signal is broken into 3 digits and the coefficient is broken down the same way. The 4 bit digits of the signal are multiplied by the 4 bits of the coefficient and the distributed property of multiplication is used to combine the 4 by 4 products. The 4 by 4 products can be implemented using a look up table or with a standard Wallace tree multiplier. Much study has been directed toward utilizing CLB's/LAB's and embedded RAM for these look up tables. In VHDL, a 12 bit vector , is designated by the nomenclature [11 downto 0]. We represent the signal s[11 downto 0] as s0 = s[3 downto 0], s1 = s[7 downto 4] and s2 = s[11 downto 8] and similarly break the coefficient c into 3 parts, c0, c1 and c2. Using the distributed property of multiplication, we can represent the product of s • c as

(6)

$$
\begin{aligned}
p = \quad & s0 \bullet c0 + \\
& (s1 \bullet c0 + s0 \bullet c1)\ 2^4 + \\
& (s2 \bullet c0 + s1 \bullet c1 + s0 * c2)\ 2^8 + \\
& (s2 \bullet c1 + s1 \bullet c2)\ 2^{12} + \\
& (s2 \bullet c2)\ 2^{16}
\end{aligned}
$$

One way to implement this multiplier is to create three 4 bit registers for the signal. Each of these digit registers is paired with a fixed coefficient digit. A shift clock is used to shift the 4-bit signal digit from one 4-bit register to the next. Three partial products result from the alignment of the signal register with each of the coefficient digits for each clock cycle. These partial products are summed and passed into a barrel shifter, which applies the appropriate power of two shift before the sum is added to the accumulator. This digit serial process requires 5 clocks cycles to complete the multiplication. In this implementation, the multiplication is simplified if the signal and coefficients are positive, using signed-magnitude format. The signs of the factors are saved and multiplied separately to determine the sign of the product. A block diagram of the circuit is provided in Figure 8.

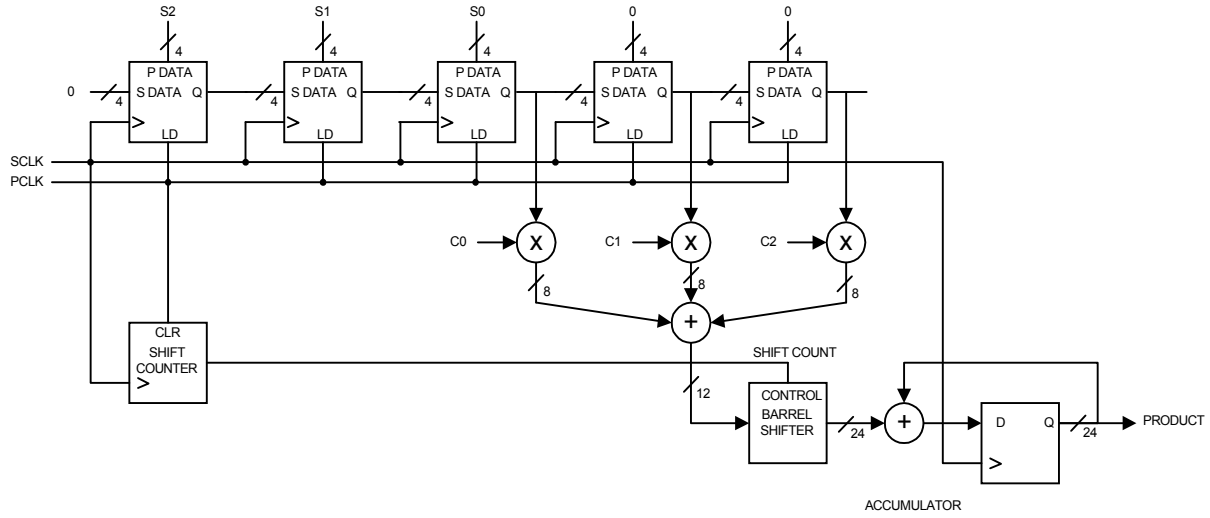


Figure 8. Digit Serial Multiplier

Filter Design Verification

Digital filters can be designed and analyzed easily with off the shelf programs. Simulation tools are readily available for digital signal processors to evaluate frequency response of the DSP code. However, when implementing a design in an FPGA, the ability to sweep or analyze the frequency response is very cumbersome. This is due to the limitations in FPGA simulation tools and due to the difficulty in creating and playing out the large sequences required for the stimulation waveforms. Another method of verification, which has proven very useful, is to compare the unit pulse response of the filter with a simulation of the unit pulse response. The unit pulse response is the sequence of output samples created by first clearing the filter taps, forcing a single input sample to 1, and forcing subsequent input samples to 0. The simulation is run for a time interval equal to the total number of delay elements in the filter plus a couple of intervals to allow for latency between stages and other miscellaneous delays in the design. The output sequence from the FPGA timing simulation is compared to the unit pulse response computed by the design tool or by a small program written for this purpose. It is important that the analysis tool or program is capable of using the same number of quantized bits in the signal and coefficients are used in the actual FPGA implementation. For an FIR filter, the unit pulse response simply "reads" out the coefficient sequence, so an analysis tool is only necessary to verify IIR filters. Finally, after the design implementation has functionally debugged by simulation, the filter can be tested. The filter programming file is loaded into a prototype system, where the frequency response can be swept, the signal to noise ratio measured, overflow/underflow limits checked, etc.

Implementation of a 6th order elliptic bandpass filter using cascaded biquads

An IIR filter example is instructive due its increased design complexity as compared to a FIR. A filter design program was used to design the filter with the specifications from Table 2.

Sample rate	500 kHz	nom
Stop band edge 1	90 kHz	min
Pass band edge 1	100 kHz	max
Pass band edge 2	150 kHz	min
Stop band edge 2	165 kHz	max
Pass band ripple	1 dB	max
Stop band attenuation	20 dB	min

Table 2. Example Filter Specifications

These specifications resulted in 3 biquad stages with the following floating-point coefficients (quantized with 12 bit Q10) given in Table 3.

Section 1			
b0	0.406250	a0	N/a
b1	0.376953	a1	0.585938
b2	0.406250	a2	0.898438
Section 2			
b0	0.406250	a0	N/a
b1	-0.376953	a1	-0.585938
b2	0.406250	a2	0.898438
Section 3			
b0	0.406250	a0	N/a
b1	0.000000	a1	0.000000
b2	-0.406250	a2	0.672852

Table 3. Example Filter Specifications

With the expected frequency response plotted in Figure 9.

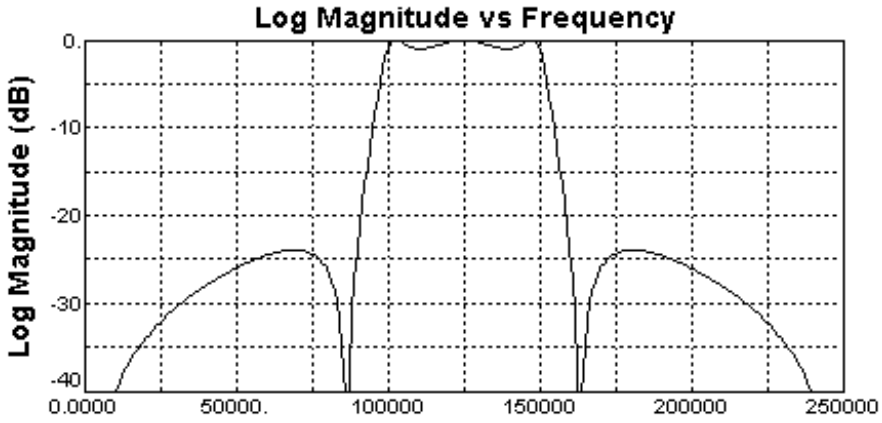


Figure 9. Magnitude Response of Example Filter Design

The design was implemented using VHDL and targeted to an Altera FPGA. The 12 bit multipliers were implemented using 4 bit, 3 digit, serial distributed arithmetic. A simulation test bench was used to verify the FPGA and produce the unit pulse response for comparison with the expected sequence. VHDL source code files, and an Excel spreadsheet used to analyze the design, are available from the authors upon request.

Acknowledgements

We wish to thank Aaron Oakley for his efforts in researching this topic for us. In addition, we thank Jim Gilbert for his help co-authoring a previous paper with us [13], which introduced us to some of the challenges in implementing DSP functionality in an FPGA.

References

1. Andraka, Raymond J.; "FIR Filter Fits in an FPGA Using a Bit Serial Approach," *3rd Conference on Programmable Logic Design*; March 1993.

2. Goslin, G R; "A Guide to Using FPGAs for Application-Specific Digital Signal Processing," *Application Note, Xilinx Inc*, 1995.

3. "Implementing FIR Filters in Flex Devices", *Application Note 73, Altera Corporation*, February 1998.

4. Knapp, S; "Constant-coefficient Multipliers Save FPGA space, time," *Personal Engineering and Instrumentation News*; July 1998; pp. 45-48.

5. Knapp, S; "Parallel Processing in FPGAs Rivals DSP Speed," *Personal Engineering and Instrumentation News*; October 1998; pp. 60-65.

6. Lee, H; Sobelman, G E; "FPGA-Base FIR Filters Using Digit Serial Arithmetic," *Tenth Annual IEEE ASIC Conference*; September 1997; pp.225-228.

7. New, Bernie; "A Distributed Arithmetic Approach to Designing Scaleable DSP Chips," *EDN Magazine*; August 17, 1995; pp. 107-114.

8. Jervis, Barrie W. and Ifeachor, Emmanuel C., *Digital Signal Processing A Practical Approach*, Addison-Wesley, 1993, ISBN 020154413X.

9. *FPGAs and DSP*, *Application Note, Xilinx Inc*, 1996.

10. *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Data Sheet v1.6, May 1999

11. *Altera Data Sheet for the FLEX 10KE,* Altera Digital Library 2000 version 6 CD-ROM, 2000.

12. Koren, Israel; *Computer Arithmetic Algorithms*, Prentice Hall, 1993, ISBN 0131519522

13. Gilbert, J. A. and Farrell, D. J.; "An FPGA Implementation of a Digital QRS Detector, " *ICSPAT Proceedings,* Miller Freeman, 1999

14. First-Generation TMS320 User Guide, Texas Instruments, Inc., 1987

15. Vaidyanathan, P. P., *Multirate Systems and Filter Banks*, Prentice-Hall, 1993, ISBN 0136057187.