Using FPGAs to Solve Tough DSP Design Challenges

by Reg Zatrepalek
DSP/FPGA Design Specialist
Hardent Inc.
rzatrepalek@hardent.com

A short, practical review of DSP vs. FPGA technology and a specific comparison of both architectures in a FIR filter application.

DSPs are invaluable in electronic system design for their ability to quickly measure, filter or compress analog signals on the fly. In doing so, they help the digital world communicate with the real world, the analog world. But as electronic systems become more elaborate, incorporating multiple analog sources to process, engineers are forced to make tough decisions. Is it better to use multiple DSPs and synchronize that functionality with the rest of the system? Or is it better to have one high-performance DSP handling multiple functions with elaborate software?

In many cases, today's systems are so complex that single-DSP implementations have insufficient processing power. At the same time, system architects simply can't afford the costs, complexities and power requirements of multiple-chip systems.

FPGAs have now emerged as a great choice for systems requiring high-performance DSP functionality. In fact, FPGA technology can often provide a much simpler solution to difficult DSP challenges than a standalone digital signal processor. To understand why requires a look back at the origins and evolution of the DSP.

MICROPROCESSORS FOR A SPECIALIZED PURPOSE

Over the past two decades, traditional DSP architectures have struggled to keep pace with increasing performance demands. As video systems make greater strides toward high definition and 3D, and communications systems push the limits of current technology to achieve higher bandwidth, designers need alternative implementation strategies. Hardware used to implement digital signal-processing algorithms may be categorized into one of three basic types of device: microprocessors, logic and memory. Some designs may require additional hardware for analog-to-digital (A/D) and digital-to-analog (D/A) conversion, and for high-speed digital interfaces.

Traditional digital signal processors are microprocessors designed to perform a specialized purpose. They are well-suited to algorithmic-intensive tasks but are limited in performance by clock rate and the sequential nature of their internal design. This limits the maximum number of operations per second that they can carry out on the incoming data samples. Typically, three or four clock cycles are required per arithmetic logic unit (ALU) operation. Multicore architectures may increase performance, but these are still limited. Designing with traditional signal processors therefore necessitates the reuse of architectural elements for algorithm implementation. Every operation must cycle through the ALU, either fed back internally or externally, for each addition, multiplication, subtraction or any other fundamental operation performed.

Unfortunately, in dealing with many of today's high-performance applications, the classical DSP fails to satisfy system requirements. Several solutions have been proposed in the past, including using multiple ALUs within a device or multiple DSP devices on a board; however, such schemes often increase costs significantly and simply shift the problem to another arena. For example, to increase performance using multiple devices follows an exponential curve. In order to double the performance, two devices are required. To double the performance again takes four devices, and so on. In addition, the focus of programmers often shifts from signal-processing functions to task scheduling across the multiple processors and cores. This results in much additional code that functions as system overhead rather than attacking the digital signal-processing problem at hand.

A solution to the increasing complexity of DSP implementations came with the introduction of FPGA technology. The FPGA was initially developed as a means to consolidate and concentrate discrete memory and logic to enable higher integration, higher performance and increased flexibility. FPGA technology has become a significant part of virtually every high-performance system in use today. In contrast to the traditional DSP, FPGAs are massively parallel structures containing a uniform array of configurable logic blocks (CLBs), memory, DSP slices and some other elements. They may be programmed using high-level description languages like VHDL and Verilog, or at a block diagram level using System Generator. Many dedicated functions and IP cores are available for direct implementation in a highly optimized form within the FPGA.

The main advantage to digital signal processing within an FPGA is the ability to tailor the implementation to match system requirements. This means in a multiple-channel or high-speed system, you can take advantage of the parallelism within the device to maximize performance, while in a lower-rate system the implementation may have a more serial nature. Thus the designer can tailor the implementation to suit the algorithm and system requirements rather than compromising the desired ideal design to conform to the limitations of a purely sequential device. Very high-speed I/O further reduces cost and bottlenecks by maximizing data flow from capture right through the processing chain to final output.

As an example of how the FPGA stacks up, let's consider a

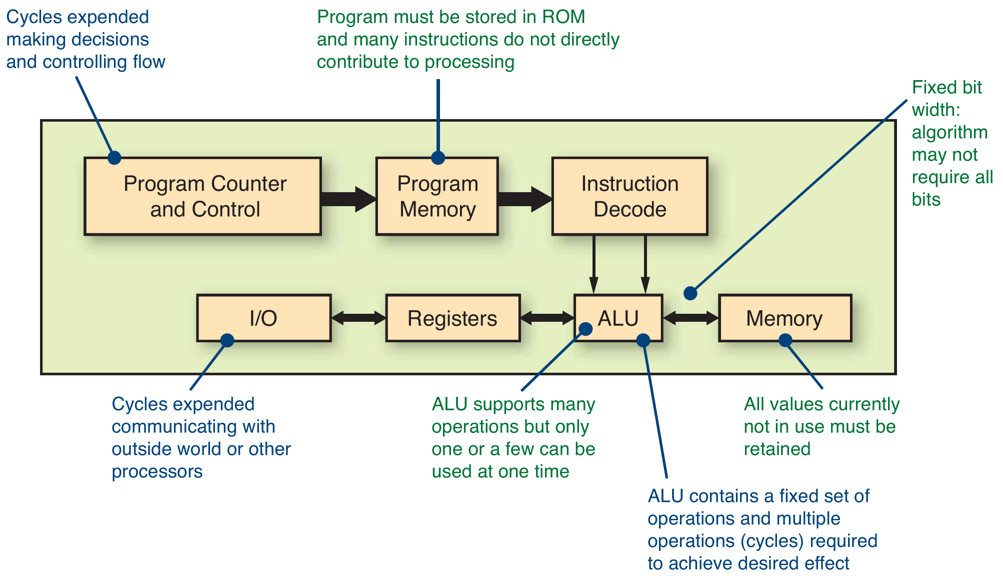
Cycles expended making decisions and controlling flow
Program must be stored in ROM and many instructions do not directly contribute to processing
Fixed bit width: algorithm may not require all bits
Program Counter and Control
Program Memory
Instruction Decode
I/O
Registers
ALU
Memory
Cycles expended communicating with outside world or other processors
ALU supports many operations but only one or a few can be used at one time
All values currently not in use must be retained
ALU contains a fixed set of operations and multiple operations (cycles) required to achieve desired effect

Figure 1 – Traditional DSP architecture

FIR filter implementation using both classical DSP and FPGA architectures to illustrate some of the strengths and weaknesses of each solution.

EXAMPLE: A DIGITAL FIR FILTER

One of the most widely used digital signal-processing elements is the finite impulse response, or FIR, filter. Designers use filters to alter the magnitude or frequency content of a data signal, usually to isolate or accentuate a particular region of interest within the sample data spectrum. In this regard, you can think of filters as a method of preconditioning a signal. In a typical filter application, incoming data samples combine with filter coefficients through carefully synchronized mathematical operations, which are dependent on the filter type and implementation strategy, and then move on to the next processing stage. If the data source and destination are analog signals, then the samples must first pass through an A/D converter, and the results fed through a D/A converter.

The simplest form of a FIR filter is implemented through a series of delay elements, multipliers and an adder tree or chain.

Mathematically, this equation describes the single-channel FIR filter:

$$Y_n = \sum_{i=0}^{N-1} k_{n-1} S_i$$

You can think of the terms in the equation as input samples, output samples and coefficients. If *S* is a continuous stream of input samples and *Y* is the resulting filtered stream of output samples, then *n* and *k* correspond to a particular instant in time. Thus, to compute the output sample *Y(n)* at time *n*, a group of samples at *N* different points in time, or *s(n)*, *s(n-1)*, *s(n-2)*, … *s(n-N+1)*, is required. The group of *N* input samples is multiplied by *N* coefficients and summed together to form the final result, *Y*.

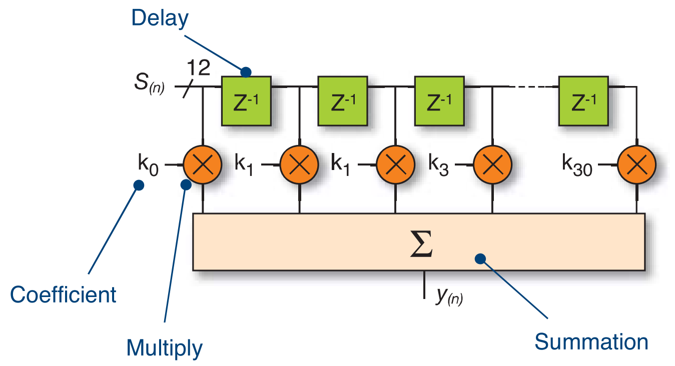

Figure 2 – FIR filter of length 31

Figure 2 is a block diagram for a simple 31-tap FIR filter (length N = 31).

Various design tools are available to help select the ideal length of a filter and the coefficient values. The goal is to select the appropriate parameters to achieve the required filter performance. The most popular design tool for choosing these parameters is MATLAB®. Once you have selected the filter parameters, the implementation follows the mathematical equation.

The basic steps for implementation of an FIR filter are:

1. Sample the incoming data stream.
2. Organize the input samples in a buffer so that each captured sample may be multiplied by each filter coefficient.
3. Multiply each data sample by each coefficient and accumulate the result.
4. Output filtered result.

A typical C program for implementing this FIR filter on a processor using a multiply–accumulate approach is shown in the code below.

```
/*
* Capture the incoming data samples
*/
datasample = input();
/*
* Push the new data sample onto the buffer
*/
S[n] = datasample;
/*
* Multiply each data sample by each coef-
  ficient and accumulate the result
*/
y = 0;
for (i = 0; i<N; i++)
{
y += k[i] * S[(n + i) %N];
}
n = (n+1) %N;
/*
* Output filtered result
*/
output(y);
```

The implementation illustrated in Figure 3 is known as a multiply-and-accumulate or MAC-type implementation. This is almost certainly the way a filter would be implemented in a classical DSP processor. The maximum performance of a 31-tap FIR filter implemented in this fashion in a typical DSP processor with a core clock rate of 1.2 GHz is about 9.68 MHz, or a maximum incoming data rate of 9.68 Megasamples per second

An FPGA, on the other hand, offers many different implementation and optimization options. If a very resource-efficient implementation is desired, the MAC engine technique may prove ideal. Using a 31-tap filter as an example illustrates the impact of filter specifications on required logic resources. A block diagram of the implementation is shown in Figure 4.

Memory is required for data and coefficient storage. This may be a mixture of RAM and ROM internal to the FPGA. RAM is used for the data samples and is implemented using a cyclic RAM buffer. The number of words is equal to the number of filter taps and the bit width is set by sample size. ROM is required for the coefficients. In the worst case, the number of words will be the same as the number of filter taps, but if symmetry exists, this may be reduced. The bit width must be large enough to support the largest coefficient. A full multiplier is required since both the data sample and coefficient data change on every cycle. The accumulator adds the results as they are produced. The capture register is needed because the accumulator output changes on every clock cycle as the filter is sampling data. Once a full set of *N* samples has been accumulated, the output register captures the final result.

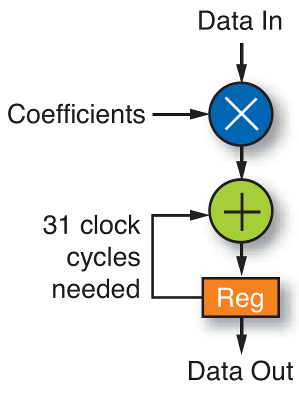

Figure 3 – MAC implementation in a classical DSP

When used in MAC mode, the DSP48 is a perfect fit. The input registers, output registers and adder unit are present in the DSP48 slice. The resources required for this 31-tap MAC engine implementation are one DSP48, one 18-kbit block RAM and nine logic slices. There are a few additional slices required for sample and coefficient address generation and control. If a 600-MHz clock were available in the FPGA, this filter could run at an input sample rate of 19.35 MHz, or 19.35 Msamples/s in a -3 speed grade Xilinx® 7 series device.

If the system specification required a higher-performance FIR filter, a parallel structure could be implemented. Figure 5 shows a block diagram of a Direct Form Type I implementation.

The Direct Form I filter structure provides the highest-performance implementation within an FPGA. This structure, which is also commonly referred to as a systolic FIR filter, uses pipelining and adder chains to exploit maximum performance from the DSP48 slice. The input is fed into a cascade of registers that acts as the data sample buffer. Each register delivers a sample to a DSP48 which is then multiplied by the respective coefficient. The adder chain stores the partial products that are then successively combined to form the final result.

No external logic is required to support the filter and the structure is extendable to support any number of coefficients. This is the structure that can achieve maximum performance, because there is no high-fanout input signal. The resources required to implement a 31-tap FIR filter are only 31 DSP48 slices. If a 600-MHz clock were available in the FPGA, this fil-

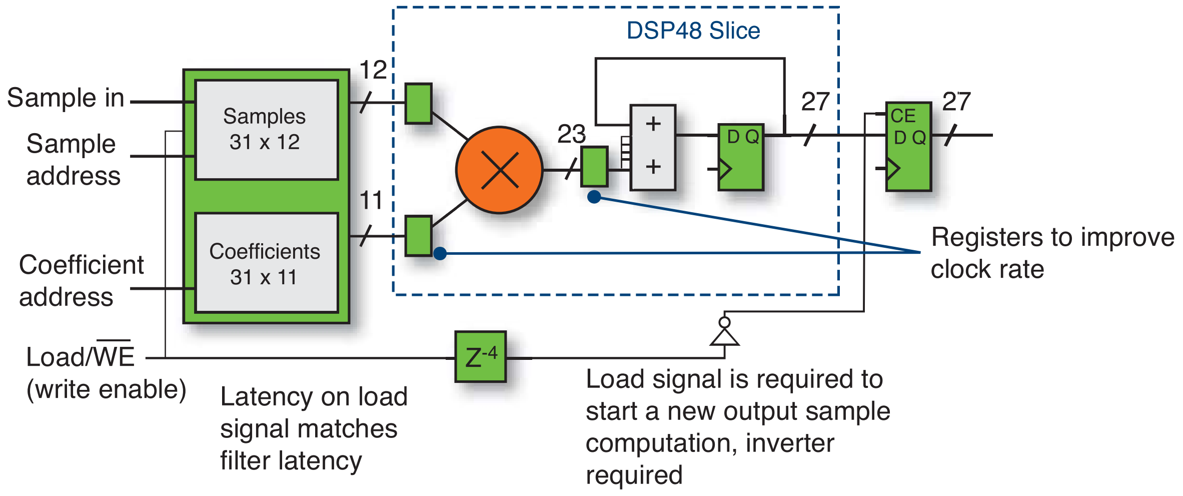
DSP48 Slice
Sample in
Sample address
Samples 31 x 12
Coefficients 31 x 11
Coefficient address
12
11
23
27
27
D Q
CE D Q
+
+
Registers to improve clock rate
Load/WE (write enable)
Z-4
Latency on load signal matches filter latency
Load signal is required to start a new output sample computation, inverter required

Figure 4 – MAC engine FIR filter in an FPGA

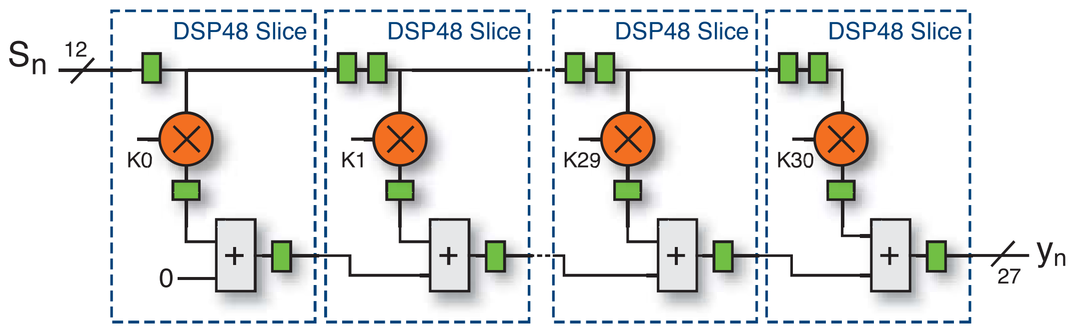

Figure 5 – Direct Form I FIR filter in an FPGA

ter could perform at an input sample rate of 600 MHz, or 600 Msamples/s, in a -3 speed grade 7 series device.

From this example, you can clearly see that the FPGA not only significantly outperforms a classic digital signal processor, but it does so with much lower clock rates (and therefore lower power consumption).

This example illustrates only a couple of implementation techniques for FIR filters in FPGA. The device may be further tailored to take advantage of data sample rate specifications that may fall in between the extremes of sequential MAC operation and full parallel operation. You may also consider additional trade-offs between performance and resource utilization involving symmetric coefficients, interpolation, decimation, multiple channels or multirate. The Xilinx CORE Generator™ or System Generator utilities will help you exploit all of these design variables and techniques.

DECIDING BETWEEN TRADITIONAL DSP AND FPGA

Conventional digital signal processors have been around for many years, and there are certainly instances where they present the best solution to a particular problem. If the system sample rate is below a few kilohertz and is a single-channel implementation, the DSP may be the obvious choice. However, as sample rates increase beyond a couple of megahertz, or if the system requires more than a single channel, FPGAs become more and more attractive. At high data rates the DSP may struggle to capture, process and output the data without any loss. This is due to the many shared resources, buses and even the core within the processor. The FPGA, however, can dedicate resources to each of these functions.

DSPs are instruction based, not clock based. Typically, three to four instructions are required for any mathematical operation on a single sample. The data must first be captured at the input, then forwarded to the processing core, cycled through that core for each operation and then released through the output. In contrast, the FPGA is clock based, so every clock cycle has the potential ability to perform a mathematical operation on the incoming data stream.

Since the DSP operates on instructions or code, the programming mechanism is standard C or, for higher performance, low-level assembly. This code may have high-level decision trees or branching operations, which may prove difficult to implement in an FPGA. A wide variety of legacy code exists to perform predefined functions or standards like audio and telephony codecs, for example.

FPGA vendors and third-party partners have realized the advantages of using FPGAs for high-performance DSP systems, and today many IP cores are available across most vertical markets including video, image-processing, communications, automotive, medical and military applications. Often it is simpler to break a high-level system block diagram into FPGA modules and IP cores than it is to map it into C code for DSP implementation.

MOVING FROM DSP TO FPGA

Examining a few key criteria may facilitate the decision between conventional DSP and FPGA (see Table 1).

It is widely accepted that software programmers outnumber hardware designers by a significant margin. The same is true for DSP programmers vs. FPGA designers. However, the transition for system architects or DSP designers to FPGA is not as difficult as software-to-hardware migration. Many resources are available to dramatically decrease the learning curve for DSP algorithm development and implementation in FPGAs.

The main hurdle is a paradigm shift from a sample- and event-based approach toward a clock-based problem description and solution. This transition is much easier to comprehend and apply if it is made at the system architecture and definition stage of the design process. It is not unusual for different engineers and mathematicians to be defining system architecture, DSP algorithms and FPGA implementation somewhat isolated from one another. This process is, of course, much smoother if each member has some knowledge of the challenges the other team members face.

In order to appreciate FPGA implementations, an architect need not be highly proficient at FPGA design. A fundamental understanding of the devices, resources and tools is

	FPGA	DSP
System sample rate	If the sample rate is more than a few MHz, or if multiple channels are required, FPGAs are the best choice.	For single-channel applications at sample rates of a few kHz, standalone DSPs will provide a more cost-effective solution.
System specification	A specification developed from or including a block diagram may map more easily into modules in the FPGA.	If the system spec is based on a C model, DSP may be easier to code into similar functional modules.
Data rate	If the I/O data rates are greater than a few Mbytes/second, the FPGA will be easier and more efficient to implement.	For low data rates extra cycles are usually available, so DSPs will not be bandwidth limited.
Data width or precision	Because of its highly uniform parallel structure, the FPGA is the clear choice for high-data-width or high-precision systems.	There may be no significant limitations if the data width is identical to the processor bus width and no potential for bit growth exists within the algorithm.
Decision trees or conditional branches	State machines may provide an easy alternative to conditional branches. If the required structure is a decision tree, embedded processor cores or MicroBlaze™ may provide the optimum solution.	DSPs are usually easier to implement with branches.
Floating point	FPGAs can have floating-point cores. Often high-precision fixed-point implementations can supplant floating-point requirements, and may in fact provide a higher-performance solution.	Conventional DSPs may have floating-point cores.
Legacy code	Mechanisms to adapt C code for use in FPGAs are now available.	If much C code already exists, a traditional DSP may be quicker to code but may struggle to increase performance from the legacy system specification.
Cores, libraries and third-party IP	Extensive parameterizable IP cores and libraries optimized for FPGAs are available from Xilinx and third-party vendors. Licensing agreements are usually simpler and easier to negotiate for FPGA implementations.	Many cores and libraries exist for conventional DSPs. May involve per-use royalty fees or complex and restrictive revenue formulas.

Table 1 – Key criteria in DSP vs. FPGA choice

all that is required. These steps can be reduced through the many focused courses that are available.

The exact steps will vary depending on an engineer's background and expertise. Specifically in the DSP category are classes in algorithm development, efficient implementation and System Generator design. If you wish to become proficient with DSP design in FPGA, a great starting point would be three courses offered by Hardent and other Xilinx Authorized Training Partners: DSP Primer, Essential DSP Implementation Techniques for Xilinx FPGAs and DSP Design Using System Generator.

Hardent also offers general courses describing Xilinx devices, HDL design entry languages, optimization techniques, and design and debug strategies. Specialized courses and workshops focus on high-speed I/O considerations, embedded processing and DSP design techniques.

Consult *www.hardent.com/training* for more on the Hardent training schedule.