DSP design flows in FPGAs

Yankin Tanurhan, Vlad Dinkevich, Manish Dharod, and Shafi Syed, Actel Corp. - April 05, 2006

Introduction
Digital signal processing (DSP) occurs in communications, audio, and multimedia devices, imaging and medical equipment, smart antennas, automotive electronics, MP3 players, radar and sonar, and barcode readers, to name but a few. According to market research firm Forward Concepts' November 1, 2005 DSP/Wireless Market Bulletin, the estimated market for programmable DSP chips should exceed $8B in 2005.

DSP platforms
DSP algorithms can be implemented in many different ways. The most popular being as follows:

- General-purpose microprocessors (e.g. Pentium) and general-purpose microcontrollers (e.g. 8051) can run DSP algorithms of arbitrary complexity.
- Programmable DSP chips or DSP microprocessors (uP). The internal structure of DSP uPs is optimized to run many DSP algorithms much faster and more efficiently. For example, the DSP chips have one or more built-in fast hardware multiplier-accumulator (MAC) to perform MAC operations DSP algorithms make heavy use of.
- FPGA. An FPGA can be configured to run a particular DSP algorithm, thereby dedicating FPGA resources to particular DSP tasks. Also, the FPGA can run hundreds of MAC units in parallel. As a result, the performance may far exceed that of DSP uPs.
- ASIC. An ASIC offers even higher levels of "dedication" than the FPGA. ASICs are the champion when comparing performance per square millimeter of silicon. It is important to note, however, that the gap between the ASIC and the FPGA tends to narrow as the FPGA grows in size (e.g. larger than 1 million gates).

Parallel computing vs. Turing machines
There is a big performance difference between the various DSP platforms based on how the platform performs computations. Both general-purpose and specialized DSP processors belong to the class of Turing machines, which perform instructions one at a time. For example, in order to add two numbers A and B, the Turing machine would need to do something similar to the following:

- Fetch instruction 1 and decode it.
- Execute the instruction 1, i.e. fetch data A and put it in the accumulator.
- Fetch instruction 2 and decode it.
- Execute the instruction 2, i.e. fetch data B and add it to the accumulator.
- Fetch instruction 3 and decode it.
- Execute the instruction 3, i.e. put the accumulated result where it needs to be.

The FPGA and ASIC are 'deprived' of this limitation. In fact, there are few flexibility and performance limitations a modern FPGA puts on a system developer. The FPGA can run parallel
processing (i.e. execute multiple instructions at a time); implement Turing machine(s) as needed, including instantiation of soft microprocessor cores; and carry virtually any practical combination of parallel processors and Turing machines on the same silicon. The parallel processing dramatically improves performance of common DSP functions, such as FIR filter, FFT and correlator. A 4-tap FIR filter structure is shown in Fig 1. While a processor needs to run the computations one by one, the FPGA instantiates all the necessary components – four multipliers, four adders, and three delay elements – and enables them to work in parallel. As a result, such a structure can process a new input sample every clock period as compared with the 8 clock cycles per data sample required by the microprocessor.

Not every DSP algorithm can efficiently utilize parallel processing. An IIR filter is an example of such a category. On the other hand, there are several techniques, such as CORDIC or error-correction algorithms, where the FPGA technology, despite the limited application of parallel processing, has been proven to be more efficient than a DSP processor.

DSP and general-purpose processors are trying to catch up with parallel computation machines. In some cases, modern DSP processors can perform a few instructions at a time, such as certain HW co-processors or accelerators (e.g. Viterbi decoder, FFT engine). But the FPGA also does not sit quietly as it too can carry "soft" processor cores, thereby enjoying all the benefits Turing machines provide.

Practical Considerations
Why would anyone buy a processor to run a DSP application if the FPGA is so much better? In fact, there are a few reasons for this as follows:

First, processors have a longer history than FPGAs. Further, the necessary support infrastructure has been developed over that time, including compilers, assemblers, automatic converters from high-level language to assembly code, and extensive libraries. At some point in time, almost any practical DSP application was implemented on the microprocessor; so, theoretically, one could grab a ready off-the-shelf implementation. Additionally, many new DSP algorithms emerge as software routines so they are pretty much ready for the processor platform.

Second, dealing with the FPGA requires a different set of skills than those common in the DSP community. When deciding which platform to choose, today's rule of thumb is: if a general-purpose processor can keep up with the spec, stay there. If not, pick the DSP processor when it meets the necessary MAC rate. Finally, if one needs outstanding performance, the FPGA is the best choice. Again, the challenge is that the number of SW development experts far exceeds the number of DSP processor programmers, which in turn is larger than a number of the FPGA designers capable of implementing the DSP algorithms. Finally, even with appropriate FPGA expertise available, creating a good DSP design that capitalizes on the FPGA's major benefits is a time consuming and elaborate process.

What about the ASIC? The FPGAs offer many of the same advantages as ASICs, such as reduction in size, weight and power dissipation; higher throughput; better design security against unauthorized copies; reduced device and inventory cost; and reduced board test cost.

ASICs lose to FPGAs when it comes to reduction in development time by a factor of three to four; ability to modify the configuration, including remote in-circuit programmability; and lower NRE costs that a customer pays prior to obtaining an actual ASIC device.

Development Flows
Typical DSP development flow
Atypical DSP development flow chart is shown in Fig 2. Usually, an algorithm makes its first appearance as a floating-point software model.
The algorithm gets tested, evaluated and verified using an appropriate test bench. It is worth noting the test bench development often takes the same or even more effort as the algorithm design. Floating-point representation lets algorithm creators take advantage of high-precision computations while not caring about dynamic range. At this stage, the algorithm is implementation independent.

As soon as the algorithm is found to be useful, it needs to be converted into the fixed-point representation. Implementing floating-point calculations directly, while possible in principle, takes a huge amount of silicon resources, and/or the computation rate is painfully slow. This stage is implementation dependent. In many cases, the conversion is not a straightforward process. On the contrary, it may take several iterations and experiments to obtain acceptable results. Whoever does the conversion, the person or group needs to possess good knowledge of the algorithm and reasonable familiarity with the implementation platform.

After the fixed-point algorithm gets verified, the actual implementation may start. Until that point the DSP architects use one of the common programming languages like C/C++, or an environment supporting a higher level of abstraction, like the MATLAB-Simulink package from Mathworks. In order to implement the algorithm on the FPGA, it needs to be handed over to the FPGA designers. An FPGA design flow is depicted in Fig. 3. It starts with a design capture stage, which is usually the most time-consuming and skills-demanding portion of the design. The way design engineers typically envision their domain is a collection of blocks described in Verilog or VHDL, captured at a register transfer level (RTL) [1]. An even more important fact is that simulation and synthesis tools used in the FPGA design flow expect the RTL design entry. Therefore, the DSP algorithm needs to be converted into the HDL RTL. Along the way the designer has to "unroll" the algorithm to make it suitable for parallel synchronous processing. Often this is not an easy task and requires familiarity with efficient VLSI DSP structures [4].

Obviously, manual conversion is quite a time consuming process and is prone to errors. One key concern is that there is no clear handoff between the DSP architect and the hardware design engineer working in the implementation domain. In fact, this worst-case scenario requires an engineer who is an expert in both domains, and such people are "few and far between" [1].

Fortunately, there is a growing variety of ways to mitigate the problem.

**Intellectual property (IP)**

FPGA and third-party vendors create highly parameterized HDL models that implement some popular DSP functions, such as FIR filter, FFT, CORDIC, etc. Thus, when the HW engineer encounters a "standard" function, or core in the algorithm, he/she simply parameterizes (configures) the one and instantiates it in the overall design. Needless to say, the IP cores have already been tested and verified by the vendors. In addition, these are often supplied with appropriate test benches that may serve as foundation for the larger test bench covering the entire design. IP also provides other valuable benefits:

- An IP core is a hardware module designed to be used by a HW engineer. Because they are hardware engineers, a majority of the FPGA designers favor IP cores.
- The IP-based FPGA design makes the algorithm handoff much easier since the FPGA designer does not have to get to the bottom of the DSP function. This allows for some detachment of the algorithm development and its FPGA implementation. From the business organization standpoint that might be beneficial.
- IP usually offers "fine tuned" implementation options, e.g. fixed-coefficient FIR filter, distributed arithmetic (DA) FIR filter implementation, MAC-based filter, etc. These differ significantly by
amount and type of resources utilized and performance. For example, the MAC-based FIR filter utilizes multipliers and is good for multiplier-rich parts. The DA FIR filter, to the contrary, is a clever "multiplierless" structure, primarily utilizing RAM blocks; therefore, the designer can select a solution that fits the design needs the best.

- IP developed for a particular platform takes advantage of the FPGA type architecture.

**MATLAB interface**

Looking at Fig 2, a keen mind may figure out the fact that Step 4 yields a comprehensive algorithm description in some form, e.g. C-program, MATLAB code or a state diagram. Is there a way to convert that description into HDL automatically? Or, is it possible to bypass the design capture stage of the FPGA design flow (Fig 3), and bridge the latter with the DSP development flow? The answer is yes in many cases. Further, the test bench can also be converted into the HDL description or another form the synthesis tool can accept.

It is quite common for the today’s FPGA industry to provide one or another form of the MATLAB–FPGA interface [2]. These can differ significantly by the implementation and even philosophy but from a user standpoint they look somewhat similar. The design flow follows the steps 1 to 4 shown on Fig 2. The DSP architect evaluates the algorithm at a high level of abstraction using the MATLAB-Simulink package. The environment offers advanced features, such as a smart and convenient visualization facility, extended DSP libraries, automated test bench facilities and a nice user interface.

In many cases, the DSP architects are not familiar with the RTL based design flow. It’s a challenge for FPGA vendors to conceive a flow that enables the DSP architects to leverage familiar MATLAB and Simulink environments, while transparently exercising the FPGA design flow. There has been significant progress made by Mathworks, FPGA vendors, along with a few EDA vendors, to put together such a flow. In this new generation of DSP-FPGA tools, significant advantages are offered to users through a what-if scenario analysis capability. Users can analyze multiple options and make trade-offs at every stage of the design flow. The four stages of iterative design flow enabling users to evaluate the different possibilities on hand are illustrated in Fig 4.

**Algorithm Design:** From a concept, a DSP architect translates the ideas into a design in the MATLAB and Simulink environment. At this stage, DSP blocks supplied by the FPGA or the EDA tool vendors are used. For instance, when targeting Actel FPGA devices, the designer uses block sets supplied by SynplifyDSP and/or vendor IP.

An example of how to create a design in Simulink and use the Filter Design Analysis tool from MATLAB is illustrated in Fig 5. The design capture can start by dragging in the desired blocks and connecting them up to realize the desired function. Once the design capture is done, the Filter tool provides a convenient facility to analyze the functional behavior of the filter.

**Conversion to Fixed-Point:** Simulink provides an environment to simulate the design and analyze its behavior using floating point and fixed point accuracies. Simulation can be performed using the built-in stimuli and scope block sets in Simulink. Floating point format provides a baseline performance of the algorithm that helps in the analysis of the fixed-point behavior of the design. As shown in Fig 6, the fixed point tool in SynplifyDSP helps to automatically change the accuracy of the data and the effects of the tradeoffs can be easily viewed in the scope.

**RTL Design Generation:** The Synplify DSP software offers the following optimization strategies as shown in Fig 7. Users can try different design tradeoffs to meet system performance easily by selecting the right options:
1. **Folding:** This optimization strategy helps in reducing the area utilization by reusing the same area hardware components (like multipliers) for multiple streams of data. This results in a very compact design. This optimization is a tradeoff between the area utilization of the hardware and the higher clock rate to maintain similar data rates.

2. **Retiming:** Retiming optimization is similar to the register balancing optimization done at the RTL Synthesis level; in this case though the optimization is done at the system level.

3. **Multi-Channelization:** Once a DSP algorithm has been developed and verified in Simulink, the design can then be replicated over multiple channels. This optimization automatically creates the mux logic required for passing the input data stream over the corresponding channels.

**Physical Design Implementation:** The FPGA implementation environment where the RTL design generated from Synplify DSP is synthesized, optionally simulated and mapped to the FPGA device is shown in Fig 8. There are several tools that participate in this process the design flow has been made easy for users through the flowchart shown in the tool. The user must push the relevant buttons in the flow to complete the task.

**DSP-centric FPGA design flow**
The Actel Libero tool is not limited to performing the physical design only and supports the overall standard FPGA design flow. In conjunction with vendor-optimized DSP tools and IP libraries, it creates a unified flow depicted on Fig 9.

Following the flow, the DSP architect can create a DSP design in the MATLAB and Simulink environment, convert the floating-point representation into the fixed-point, optimize the latter and verify the result. To evaluate the filter, the architect creates a test bench using the available pre-developed test modules, signal generators, scopes, signal analyzers, etc. Once the filter has been verified a user is given an opportunity to enter implementation specific configuration parameters, such as desirable clock rate, format of the input and output data, and precision of the results. Finally, after pushing a button, either the RTL model or a netlist gets generated. Thus, the systems architect who primarily develops the DSP algorithm can implement the one on the FPGA with no or little help from an FPGA designer. If not a final implementation, then a physical model that can be tested in real time gets delivered much sooner.

This works because at a backplane of the MATLAB interface there are two libraries: one used by the MATLAB-Simulink (called a "block set"), and another containing parameterized HW IP for all the components of the first library. Pushing the button, a user transfers algorithm configuration and parameters to the matching IP. Keep in mind, the final implementation is only as good as the pre-designed IP cores and how considerate a user was when ordering these or other implementation parameters.

**Design flow for mixed-signal FPGAs**
FPGA vendors have been adding SOC blocks on the FPGA fabric to deliver better value to their customers. The emergence of mixed-signal FPGAs, such as the Fusion Programmable System Chip (PSC) from Actel, opens the door for new design methods. The versatility offered by on-chip flash memory and a large number of analog input channels leverage new techniques in DSP system design. Some of the methods are outlined below.

**Nonvolatile on-chip storage:** The majority of common DSP applications make some use of fixed data sets. For example, the folded and polyphase FIR filters store filter coefficients; FFT utilizes sine/cosine tables. Some sophisticated DSP algorithms can be efficiently implemented as the look-up tables with relatively simple logic around them. For a single-chip DSP solution on the FPGA, such applications require that data sets can be loaded from a predetermined location. Fusion offers a large amount of on-chip flash memory to store the data. Each set can be considered as a context
based on the condition in which the application or the core is used. The appropriate context containing the necessary data set can then be loaded. Multiple tables can also be loaded from a single Flash memory block Fig 10. The IP block to load information from the table is supplied by the vendor.

The entire design can be realized with a single chip, thus saving board space and overall design cost.

**Handling multiple channels of data stream:** Designs that involve processing data from multiple data streams can create architectures on the fly leveraging multichannelization tools. Users can create a filter and once satisfied by its performance, can quickly invoke the special option to create a design that handles multiple channels. Fig 11 shows how the data streams coming in from multiple on-chip analog channels in the PSC can be processed through a filter, which has been multichannelized in Synplicity's SynpifyDSP tool.

**Context management for adaptive coefficients:** In adaptive filters, the coefficients that are tuned towards specific scenarios can be safely stored and retrieved from the on-chip flash memory available. Fig 12 shows one such case where the signals are subjected to different scenarios and the corresponding adapted coefficients are stored for later use. Extending this concept with clever use of multiple contexts adds significant value for this one-chip DSP solution.

It's clear that the SOC features can be leveraged to design new techniques in DSP processing as demonstrated by the above examples. These device features, when supported with tools that are friendly towards DSP architects, make the design space exploration an easy task. Even DSP architects, who are not skilled in RTL design methods can create compact and high-performance DSP designs on FPGAs, through these new generation of DSP design generators.

**Types of core generated**

It's likely an engineer who struggles for the density of a design may want to cut off some unnecessary stuff from a generated core. He or she may also want to access a few core internal signals to use them elsewhere in the entire design. But the core generated by the IP, either a direct one or using MATLAB interface, often does not provide a designer with full control over its contents. Driven by the desire to protect their IP, some (but not all) vendors limit the designer's access to the internal workings of the core.

In general, a core is delivered in three different ways - netlist or encrypted netlist, automatic RTL code or "Human" RTL code.

The first category prohibits any kind of access to the core. The core itself is not portable, i.e. it applies to the specific vendor products and most likely to some particular parts. The second group does not expect a user to look inside the core because the automatically generated code is really hard to read. The code generated though is portable. Finally the third group that generates hand-made readable RTL code gives a user the full access to the internal workings of the core. Consider this example: Many, if not all, popular FFT cores utilize sine/cosine LUTs. Upon power-on, the sine/cosine table gets loaded in on-chip RAM. Obviously, prior to that, a pre-calculated table sits on an external PROM. For those vendors with one-chip solutions, the table has to be initialized internally (i.e. the design carries an internal sine/cosine table generator). The speed is not an issue here since the initialization takes place upon power-on. The CORDIC algorithm provides a good solution for a small and slow (in HW terms) table generator. If a designer has access to both cores, FFT and CORDIC, he/she can easily combine the two if both cores are generated in 'human' RTL.

There is always a trade-off between size/speed and time-to-market. Clearly, the direct IP and MATLAB interface aim at shortening the design cycle. Such rapid prototype may serve as a final
solution in many cases. But, in other cases, more or less significant improvement might prove to be necessary. The worst case scenario is that a user needs to redesign some or all the "black box" cores, meaning he needs to start designing from scratch. However, if the cores are open to a qualified user, they can be used as a platform for further improvement rather than dumping the whole prototype.

**More development flows**

**High-level flows**
The MATLAB-Simulink interface is not the only direction EDA vendors are looking at. Another approach, called electronic system-level (ESL) design, is said to specify a system in an implementation-neutral language with the push of a button, and out would emerge full, detailed hardware design and corresponding software (see also the Design Reuse website).

SystemC also includes both software and hardware concepts. SystemC supports designing both the hardware and software components together as these components would exist on the final system, but at a high level of abstraction. SystemVerilog extends the original Verilog towards system-level description and modeling.

**'Manual' implementation or back to FPGA design**
Still, many DSP designs are fully or partially handcrafted. Both business and technical reasons may push a user towards that decision. Not all people use the MATLAB-Simulink solution nor can every developer afford buying an IP license. Finally, pre-designed IP cores don't always fit nicely in highly optimized designs. After all, there is a competition between users that makes them constantly look for better, more optimal solutions. If every consumer utilized only standard cores then the competition would be limited by nomenclature of the cores and the way they are placed to implement an application.

Perhaps not many people today remember the time when there was a serious discussion between Assembly language users and the high-level language supporters, such as ALGOL or C. (Yes, C language was considered to be high-level). It took some time to create efficient compilers and other tools. Today, not many SW gurus write Assembly. The situation looks a bit different when it comes to embedded SW, particularly DSP uP software. Though compilers and libraries keep improving, it is not unusual for a DSP uP programmer to use Assembly language. The conversion from a SW language, such as C/C++ or MATLAB m-language, to the RTL is even more problematic and non-trivial a task. As a result, manual design will not be abandoned any time soon.

Manual DSP design faces all the traditional design challenges as well as a few specific to DSPs. We discussed some issues above, and here are some more.

DSP-rich designs are often much larger than regular ones. Obviously, it is much easier to create designs out of a variety of building blocks. To help a designer meet these challenges, all kinds of highly efficient, well-optimized libraries should be at his/her disposal -- from large IP-supporting DSP functions, peripherals and soft processors to relatively small components, like counters or adders.

DSP designs require plenty of multipliers. Many FPGA parts, in addition to regular fabric and RAM blocks, carry multiple hard multipliers, or more sophisticated MAC units. The issue here is that the multiplier instantiation is not always straightforward. For example, not every multiplication sign on a Verilog code has to call the hard multiplier instance. Instead, it may call for a constant coefficient multiplier, which implements nicely as one or a couple of adders. Another example is a barrel shifter. The one implemented on a regular fabric may cause a speed bottleneck Being built out of a spare hard multiplier it exhibits excellent speed characteristics and saves a lot of fabric resources. A smarter synthesis tool could mitigate the problem.
Sometimes design density is less important or a designer is ready to trade it for the time-to-market reduction. In other cases, it presents a non-trivial, elaborate and time-consuming problem. Imagine a wireless base station design that uses only off-the-shelf IP. A given FPGA can accommodate say, 10 of those communications channels. After some manual optimization as a result of an efficient resource sharing, the same part can accommodate 20 channels. Now, after manual intrusion in place and route, or floorplanning, 40 channels can fit in the part. The final optimized design, though a time consuming one, reduces the overall number of FPGA parts required, the power consumption of the whole station, its weight and size, and sometimes eliminates the need for a cooling subsystem, etc.

Independent of a target application IP, designers constantly face the density problem as well. The IP designer community includes the FPGA vendors, independent IP core providers, as well as FPGA consumers who want to maintain their own libraries.

Power efficiency is extremely important in large designs. While contemporary FPGAs provide enough MAC power to support a wide variety of the DSP techniques, in many cases the power consumption is becoming a limiting factor. The FPGA vendors offer an ever-expanding set of HW and SW features and facilities, improving power consumption.

Summary
Among many existent DSP platforms, the FPGA – alone or in combination with other devices – provides unmatched computational power and flexibility. FPGA and EDA vendors, as well as "open source" groups offer a few design flows that speed up the FPGA implementation and make it less error prone. It is the work in progress that is going to take more efforts to convert the DSP application design into a painless task available to many HW engineers as well as the DSP architects. In the mean time, a variety of well-designed and tested IP/libraries, primarily the RTL generators, can significantly improve productivity of the FPGA design flow.

References

This article, which was written by Yankin Tanurhan, Vlad Dinkevich, Manish Dharod and Shafi Syed of Actel Corporation, is excerpted from a paper of the same name presented at the Embedded Systems Conference Silicon Valley 2006. Used with permission of the Embedded Systems Conference. Please visit www.embedded.com/esc/sv.