

# AN FPGA FRAMEWORK SUPPORTING SOFTWARE PROGRAMMABLE RECONFIGURATION AND RAPID DEVELOPMENT OF SDR APPLICATIONS

David Rupe (BittWare, Concord, NH, USA; [drupe@bittware.com](mailto:drupe@bittware.com))

## ABSTRACT

The role of FPGAs in Software Defined Radio (SDR) applications has continued to increase in spite of significant development costs. Implementation practices are non-standard as developers work at low abstraction levels, treating FPGAs as a blank canvas. The majority of implementation cycles are spent building and testing external device interfaces and infrastructure to support the end application. The resulting long and drawn out schedules, increased complexity, and overall risk of FPGA inclusion in SDR applications is forcing developers to adopt new implementation practices. This paper introduces an FPGA framework leveraging concepts found in modern software applications. By utilizing software methodologies, this framework not only supports Software Programmable Reconfiguration (SPR) it also makes rapid development of FPGA-based SDR applications possible while decreasing costs and minimizing risk.

## 1. INTRODUCTION

The growing complexity, size, and performance requirements of today's SDR applications have driven industry wide utilization of FPGAs as primary processing devices. Significant improvements in FPGA technology provide developers with more gates, higher clock rates, and valuable processing resources.

However, implementation costs, which have always been difficult to measure, are a large limiting factor in FPGA development of SDR applications. Oftentimes, the majority of development cycles are wasted building and testing custom external interfaces, debugging component-to-component connections, and reconfiguring systems to meet changing requirements. Development efforts that should be reused, typically are not, as designers tend to be more comfortable with the 'roll-your-own' approach. As a result, schedules are drawn out while developers often focus on reinventing the wheel, instead of developing application specific IP.

FPGAs can continue to expand the capability and flexibility of SDR applications, but the traditional route of starting with a blank canvas is no longer suitable. By leveraging software like methodologies within a scalable

FPGA framework, developers will have easy access to resources that can facilitate Software Programmable Reconfiguration (SPR) and rapid development of FPGA-based SDR applications while significantly reducing risk.

## 2. THE SOFTWARE APPROACH

FPGA implementation methodology is very immature in comparison to the well defined nature of software development. Current FPGA implementation is most similar to micro-processor development before the mainstream use of high level languages like C++, Java, and widespread use of concepts like Object Oriented programming. Developers implemented their applications at a very low level, in assembly or C, writing custom device drivers and peripheral interconnects, very similar to today's gate level FPGA implementation. To combat the challenges in software development, developers began to abstract their application from the device they were targeting. Peripheral infrastructure like buses, UARTs, DMA, arbiters, and many others were developed to lower risk and increase development efficiency. As a result, peripheral and infrastructure reuse became mainstream and higher level language development more feasible.

By taking a software-like approach to FPGA development and looking at the FPGA as a System-On-a-Chip, with peripheral infrastructure in place, the goals of SPR and rapid development of FPGA-based SDR applications can be achieved. Figure 1 shows the similarities between this type of FPGA framework and the standard micro-processor framework with peripheral support. The application is separated into two distinct processing planes, each utilizing a common interface standard for component interconnect. The first is the control plane used for control, (re)configuration, status, and memory management. Routing of control/configuration and status is accomplished with a control fabric. The second plane is the streaming data plane. Each of the processing blocks are connected to a streaming data fabric that allows for point to point data transfer between waveform components.

In addition, applications built using a scalable FPGA framework promote hardware reuse at the component level as well as the application level. Multiple waveforms can be implemented on one device or across multiple devices and

component building blocks can be reused with each different

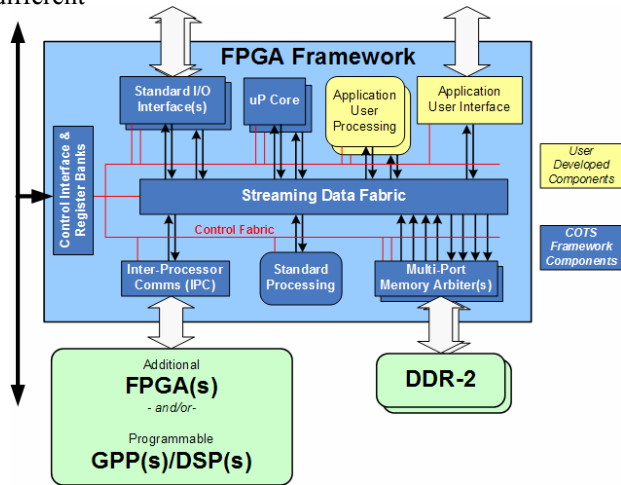
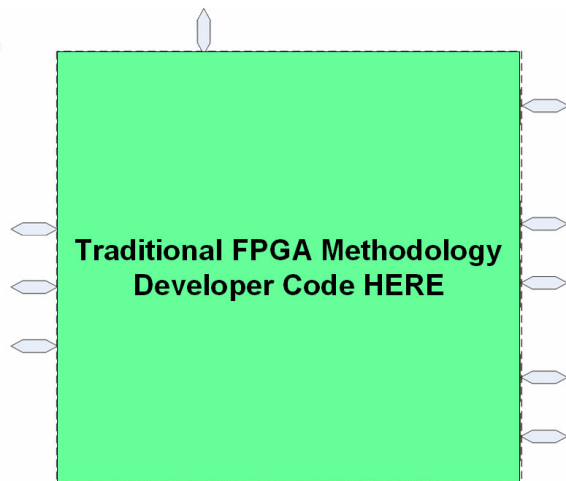


Figure 1: FPGA Framework

application. Integration of a soft core micro-processor into the application improves design exploration and test, providing a method of control, status, and flexible, real-time adaptive software reconfiguration. Instead of re-designing, re-writing HDL, simulating, re-synthesizing, and finally, reprogramming the FPGA every time a change is required, the whole range of application requirements are implemented and all the necessary adaptive software reconfigurable components are deployed to a single device or across multiple devices for easy software reconfiguration.



### 3. FPGA FRAMEWORK

Figure 2: Traditional FPGA Methodology

COTS FPGA systems are often perceived as difficult to use. When looking back on traditional FPGA implementation methodology, they are. Developers not only had to learn about the platform itself, they were also required to

implement all the external interfaces and internal infrastructure, in addition to application specific components. Figure 2 shows the clean slate FPGA developers used to start with when implementing applications on COTS boards.

The process of building and testing external interfaces and internal infrastructure is taxing, typically exhausting a significant amount of time and resources. Project managers find it difficult to cost projects and measure the amount of effort required to complete application development. Fortunately, COTS board vendors now supply some sort of FPGA developer kit, including the high risk external interface IP. However, they still do not offer an FPGA framework designed with the benefits of proven software constructs. This type of framework fully leverages a common interface, supplying more than just external interface IP. Focusing on just FPGA implementation, the FPGA framework should include:

1. Common Interface
2. External Interface IP
3. Processing Utility Libraries
4. Control, Configuration and Memory Management Facilities
5. Component Interconnect
6. Third Party IP
7. Integration Software
8. Capability for Software Programmable Reconfiguration

The infrastructure, supporting software and methodologies provided in an FPGA framework should be well-documented and tested, meeting specified performance criteria. In this manner, COTS board vendors can mitigate developer risk, allowing them to refocus on application specific IP.

#### 3.1. Common Interface

Similar to the software world, common interfaces abstract the functionality of a component from its interface. They provide a standard API for communication between a functional component, its sources, sinks, masters, and slaves. In doing so, component and platform reuse is promoted as there is no question to how components will communicate with each other. In addition, design verification is simplified. Simulation infrastructure like data generators, data checkers, and other verification facilities can also be made standard, and thus, reused. One of the most significant benefits to common interfaces that is so often overlooked, is that they enable automatic code generation. Because component interfaces are well-defined, it is an easy task for software tools to generate integration code.

The typical component has two types of common interfaces, as seen in Figure 3. Each component requires a control/memory interface and a streaming data interface. The control/memory interface is an addressable interface

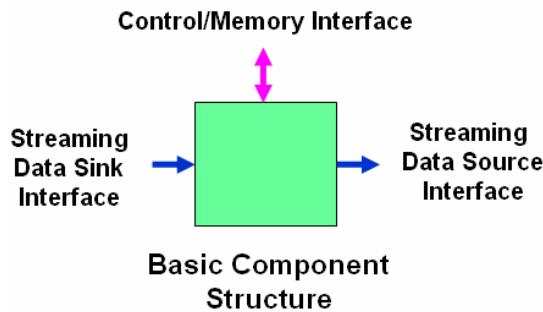


Figure 3: Basic Component Structure

that should support interrupts, burst read/write, flow control, wait states, variable latency, bi-directional or unidirectional bus interfaces, and other related functionality. It is a bidirectional interface supporting both reads and writes. Besides being used as a control interface, it is also used to interface register banks, flash, DDR, SRAM, and any other addressable interface. The streaming data interface is a point-to-point interface for all source/sink connections, including component-to-component, component-to-infrastructure, and component-to-external interfaces. It should support advanced features like: multiple channels, packets, burst and block transactions, flow control, wait states, variable latency, and other similar transactions.

Two common interface standards are currently gaining industry attention. The first is OCP ([www.ocpip.org](http://www.ocpip.org)) and the second is Avalon, defined by Altera. Both interface specifications define a set of signals, the behavior of the signals, and the types of transfers supported while

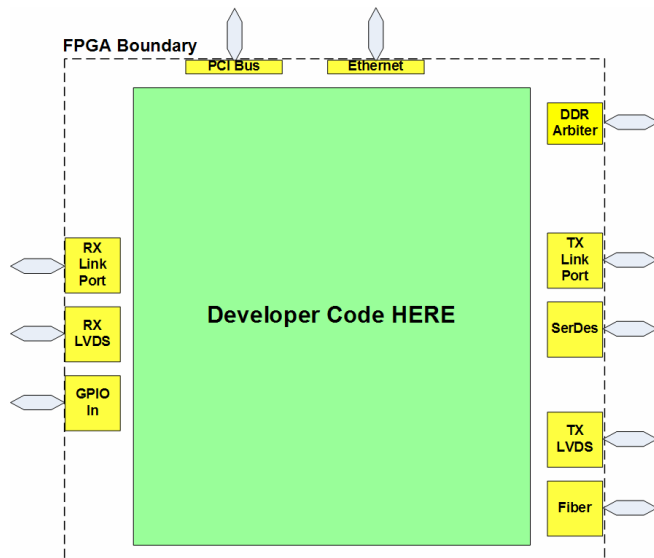


Figure 4: External IP

maintaining low resource overhead.

### 3.2. External Interface IP

COTS board vendors add significant value to application development by providing external interface IP to developers. In doing so, hardware specifics are abstracted away from the implementer. Similar to a CPU with peripherals, external interface IP is well-defined, fully tested, and reusable. By providing external IP with common interfaces, implementation efforts can be reduced significantly. Figure 4 shows a block diagram of an FPGA device with yellow blocks representing the external interface IP. Fiber, SerDes, LVDS, and other external interfaces are already implemented, thus reducing the work the developer has to worry about, which is arguably the most difficult and time-consuming aspect of FPGA implementation.

### 3.3. Processing Utility Libraries

The software world has printf, memcpy, string manipulation and many other helpful functions. These library functions are provided to speed up implementation as developers can reuse common, existing, and verified functionality. FPGA development should be no different. A complete developer framework should supply HDL utility libraries to cut down on implementation efforts. Some of the most commonly used functions that should be provided are:

- Signal processing functions
  - Scale, round, saturate
  - Mag est., magnitude squared
  - Min/max
- Common interface helper functions
  - Initialization
  - Scaling, resize, reshape
  - Array interfaces

In addition to the functions described above, the library should include other items to help minimize the developer's learning curve. Things like common interface How-To and design and component templates should also be made available. Figure 5 depicts an application specific component leveraging HDL utility functions.

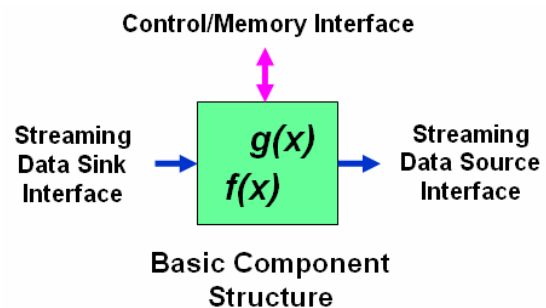


Figure 5: Processing Libraries

### 3.4. Control/Configuration and Memory Management

Similar to software applications, an FPGA framework should include infrastructure to facilitate high level control/(re)configuration and status facilities. Control capabilities, like interrupts and register banks for component reconfiguration, are examples of common infrastructure that can be reused across components and applications. In addition, components require access to different memory structures through arbiters and other types of memory controllers. This commonly used and reusable memory infrastructure should be developed and ready for integration into the developer's application.

### 3.5. Component Interconnect

Leveraging common interfaces for component implementation allows developers to reuse control and streaming data fabrics like muxes, FIFOs, dual port memories, arbiters, and other component to component interconnect. The FPGA framework should not only provide the most commonly used interconnect infrastructure, it should also provide some less commonly used, like common interface adaptors to translate between different common interfaces standards, data serializers, deserializers, and other data reshape, and multi-channel interconnect. Further enhancements can be made to the component interconnect by providing a control interface allowing real-time reconfiguration and switching. This infrastructure is well-tested, documented, and proven to meet specified implementation requirements.

### 3.6. Third Party IP

Reusing existing functional IP is invaluable for rapid application development. Existing IP is design ready. It has been fully tested and validated. It should be well documented with performance metrics and resource utilization stated clearly. Third party IP should be implemented using a common interface so the developer does not have to adapt the IP to his application. In addition, it is very valuable to have a tool that aids in the definition and configuration of IP similar to Altera's Mega Wizard. Pretty much any type of IP can be purchased from many

different companies at varying levels of quality. Some of these may include soft micro-processor cores, such as the NIOS, peripheral interfaces such as DDR and PCI, signal processing IP such as filters and FFT functions, and communications IP such as modulators and encoders.

### 3.7. Integration Software

Once a component has been developed and unit tested, it is necessary to integrate it into the application. When using components implementing a common interface, the tedious, error prone task of integration can easily be generated using design integration software. The interconnect fabric for control and memory, and the point-to-point streaming data connections should also be generated. This software should allow developers to import custom components for quicker application integration.

### 3.8. Software Programmable Reconfiguration

Figure 6 represents an FPGA application based on the framework discussed previously. Common interfaces are utilized throughout the application making up the API for communication across both control/memory and data planes. Supplied infrastructure provides the component interconnect and supporting control/configuration and memory management capabilities. External interface IP connects the waveform application to the outside world, while processing utility libraries and third party IP reduce component development efforts. All in all, implementation effort, cost, and risk are substantially reduced so developers can focus on implementing waveform specific components, leaving even the integration process to special integration software.

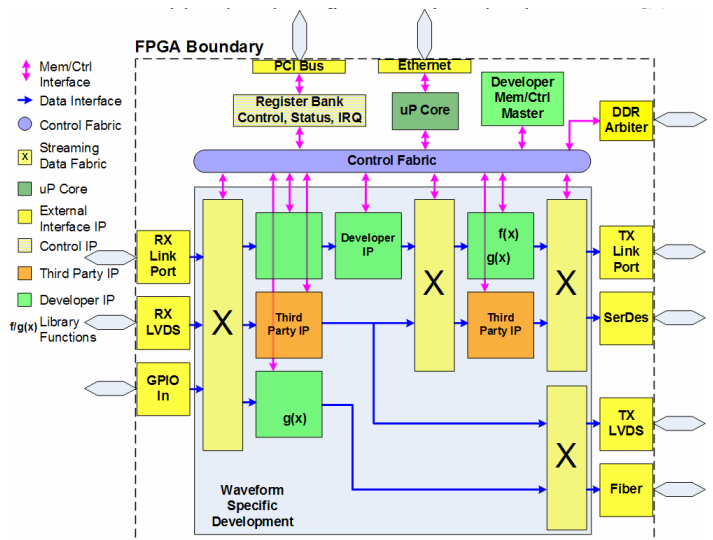


Figure 6: Typical Design Diagram

Integrating a soft core micro-processor, like the NIOS, provides reconfiguration to aid in field testing and measurement. Reconfiguration only requires a simple register write or other simple software switch. In addition, applications supporting SPR with an integrated soft core micro-processor can utilize ethernet, offering remote software control, reconfiguration, status, and visualization.

#### 4. BITTWARE SPR DEMO

In an effort to explore the benefits of an FPGA framework supporting SPR, BittWare and Altera have invested in the implementation of a reference application. This task required the utilization of common interfaces in the development of portable infrastructure IP, and waveform specific IP. Combined with third party IP, an existing NIOS soft core implementation, and beta integration software, an SPR reference design was created.

Figure 7 provides a high level view of the application running on a BittWare B2-AMC board connected via LVDS to Altera's Cyclone III Starter Kit. A GUI was developed for reconfiguration and data visualization. Commands can be sent over a wireless ethernet connection from the GUI to a NIOS soft core micro-processor which then directs it to

the correct destination. A TigerSharc DSP was used to source and sink data through the system, simulating the ADC/DAC interface, and a Stratix II FPGA acts as the bridge between the B2-AMC and the Altera Cyclone III Starter Kit (shown in Figure 8). The most interesting part of the demo is the software reconfigurable waveform application built using the Atlantis FPGA Framework developed by BittWare. The application is simply three waveform components, a mixer, filter, and an FFT. Each of the components is fully reconfigurable from the GUI. The mixer has an adjustable LO frequency, the filter has three different filter banks that can be chosen, and the FFT can operate at five different FFT block sizes. In addition, the streaming data fabric is a fully reconfigurable interconnect, allowing the streaming data path to change, bypassing or including any of the waveform components.

The complete FPGA implementation effort took one engineer a total of 18 weeks, full time. A listing of task distribution follows.

- 1 week – learning Avalon common interface and working out kinks in understanding use cases
- 4 weeks – implementing and testing component interconnect infrastructure IP
- 2 weeks – implementing and testing control/config and memory management infrastructure IP
- 1 week – implementing processing utility libraries
- 5 weeks – implementing and testing external interface IP
- 1 week – reading 3rd party IP specs and properly generating IP and implementing application specific components using 3rd party IP
- 2 weeks – writing common interface wrappers including configuration and status interface for 3rd party IP
- 1 week – writing and verifying component integration since immature integration software proved to have limitations
- 1 week – performing test and measurement of the application for data path scaling and optimization purposes

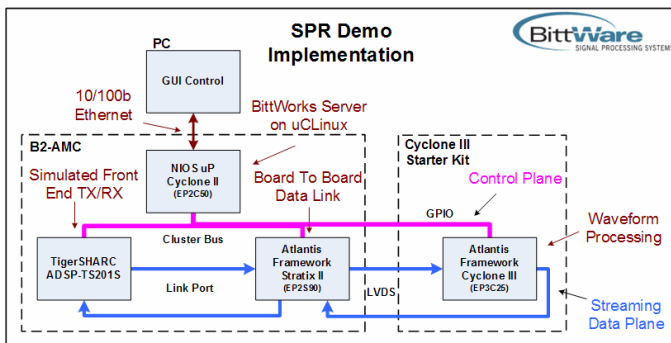


Figure 7: SPR Demo Implementation

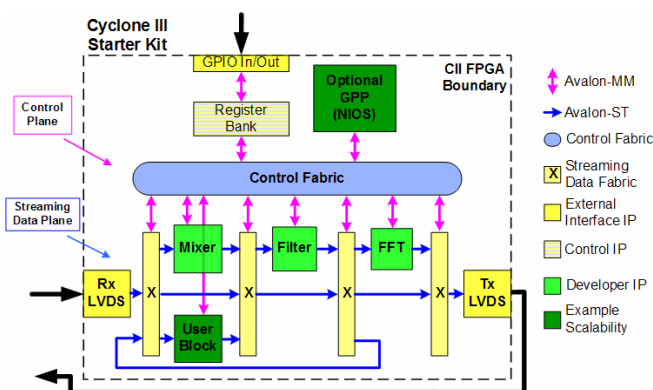


Figure 8: Cyclone III Design Diagram

With a developer framework in place, this 18 week task would have been reduced to 6 weeks. By taking out the one week learning curve for common interfaces, as it is only required once per engineer, using more mature integration software, and obtaining 3rd party IP fully supporting common interface streaming data and reconfiguration, the schedule is reduced by another four weeks. That leaves just two weeks of actual implementation effort focused on the waveform specific components. Adding and removing components from this application, now that the developer framework is in place would take nominal effort beyond the development of the component itself.

## 5. FUTURE WORK

The FPGA framework, design methodology and toolset discussed in this paper are not limited to Software Defined Radio applications. In fact, European companies are now discussing what they have also called, SDR, Software Defined Radar. Image processing and sensor networks are also struggling with the same difficulties in FPGA development as the Software Defined Radio community. By adopting an FPGA framework supporting software programmable reconfiguration as described in this paper, any application requiring FPGAs will benefit significantly. As the use of FPGAs continues to grow, BittWare will adapt this methodology and supporting tools to meet industry's varied needs.

BittWare is committed to the goal of adding significant value to SDR applications beyond top notch FPGA compute platforms. By providing a stable, reusable, well-defined FPGA framework, supporting SPR, implementation engineers can once again focus on their application specific IP.

## 6. CONCLUSION

As FPGA technology continues to improve, gates and computational resources will become cheaper and cheaper. Developers can be well-positioned for the future by utilizing FPGAs intelligently. With an FPGA framework supporting the concept of SPR, the abstraction level of FPGA development can be raised, similar to that of a software application running on a micro-processor with peripheral support. This scalable FPGA framework allows for an increase in application complexity, even mapping directly to an ASIC flow, and most importantly, enables rapid development of SDR applications within budget constraints.



[www.bittware.com](http://www.bittware.com)

Copyright © 2007, BittWare, Inc. All Rights Reserved

The information in this whitepaper has been carefully checked and is believed to be accurate and reliable. However, BittWare assumes no responsibility for any inaccuracies, errors, or omissions that may be contained in this manual. In no event will BittWare be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect or omission in this document. BittWare reserves the right to revise this document and to make changes from time to time in the content hereof without obligation of BittWare to notify any person or persons of such revision or changes.