AUTOMATING FPGA-BASED SYSTEM IMPLEMENTATION WITH COMMON INTERFACING

John Bradley (The MITRE Corporation, Bedford MA USA, jbradley@mitre.org);
Karl Wagner (The MITRE Corporation, Bedford MA USA, kwagner@mitre.org)

ABSTRACT

A challenge in component-based design is reliably and efficiently instantiating the transport and any translation functions required to transfer data between each component comprising the entire system. While the components may involve complicated manipulation of the data stream, the transport between them is typically much simpler. It can be abstractly defined in terms of data flow characteristics qualified by various properties of the targeted hardware platform. If the interface between the components of the system is well defined, automated tools can easily create the required transport based on a high level structural view of the system. This paper will describe a complete solution for automating the process of interconnecting all components of a system. First, we defined a set of three component interfaces based on Open Core Protocol (a low-level hardware interface definition standard) that, as a group, we refer to as Common Interfaces. Next, we defined an XML representation and created a GUI for describing the high-level characteristics of the component interfaces and the interconnection of the components within the system. Lastly, we developed scripts to generate hardware description language source for the entire system besides the components. In so doing, we have freed the system designer to focus on developing the components, the heart of the system.

1. INTRODUCTION

The MITRE Programmable Radio Technology (PRT) Laboratory is a multidiscipline electronic system rapid prototyping team dedicated to performing risk reduction and requirements specification with the mission of assisting the government in the successful acquisition of state-of-the-art communication and networking systems. A particular focus is the development of strategies for the efficient reuse of FPGA-based software-defined radio (SDR) waveforms [1]. To vet these strategies, we developed the High Data Rate – Radio Frequency (HDR-RF) Test Waveform, the primary purpose of which is to ensure that the modem hardware platforms developed by multiple contractors for the HDR-RF program have adequate computing resources to implement the proposed operational waveforms. The Test Waveform includes basic elements common to many high-bandwidth, high-throughput (HBHT) waveforms, including acquisition and tracking, modulation, filtering, strong forward error correction, and several selectable modes of operation. The complexity of the Test Waveform requires deployment on hardware platforms using several FPGAs.

To aid in the rapid deployment of this system on multiple hardware platforms, the PRT Laboratory adopted and implemented the concept of designing processing components with a well-defined set of interfaces. This paper describes the details of this approach.

2. COMPONENT-BASED DESIGN

There are a number of advantages to component-based design, in which a complex system is broken down into less complex components, the components are designed separately, and then they are integrated to form the complex system [2]. A primary advantage is that appropriately-sized components of a system can be dispersed across multiple FPGAs on platforms with FPGAs that have a range of available resources. That is, generally a less complex design will require fewer hardware resources to implement than a more complex design. By breaking down the system into constituent components each of which consumes at most 50% of the resources of the least capable FPGA that the system might be targeted to, the designer has the freedom to deploy the system on any platform that has FPGAs meeting the minimum capability requirements.

A consequence of component-based design is that the components must be connected to each other in order to perform the overall function of the system. If all the interfaces of the components conform to some standard, several benefits arise and will be outlined throughout this paper.

3. COMMON INTERFACING

While the functions of any two components may be drastically different and the types of information that are transferred through the components' interfaces are dissimilar, these components can share a simple, well-defined set of interfaces. For our hardware designs, we defined a set of three component interfaces that provide for all interaction between components and their external environment based on our previous experience creating custom interfaces.

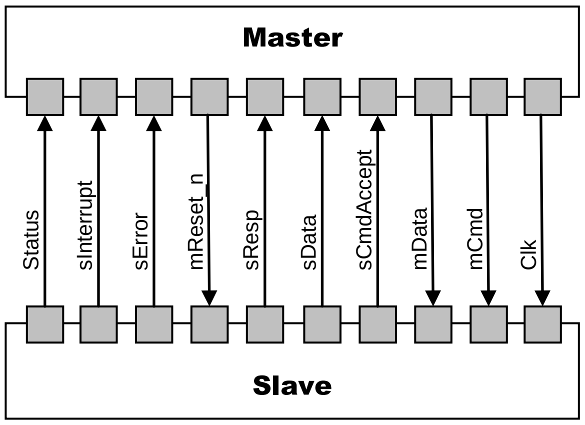

Figure 1: System OCP Profile

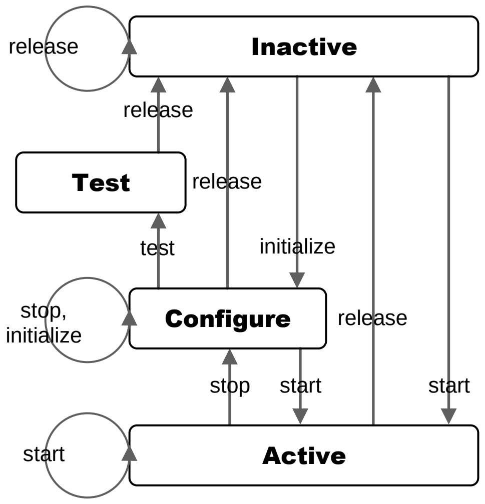

Figure 2: System OCP Profile States

The first is an interface that provides the means to manage the overall operation of the component. It allows control of the state of the component, the reception of error and interrupt indications, and the transfer of debugging information.

The second is a memory-type interface that provides straightforward, efficient read and write access to addressable memory locations. The core signals of this interface are address and data busses.

The last is an interface that provides a simple, unidirectional streaming data transfer mechanism. It is optimized for implementation efficiency and continuous data transfer with basic flow control.

With combinations of these three interfaces, a wide range of components can be interfaced to. If all component interfaces are restricted to this set, the interconnection of any compatible components is easily achieved. Furthermore, the system-level control of the collection of components is simple since a single controller can manage all of the components with a single interface type by simply replicating instantiations of that interface. Also, by ensuring that all components only use these interfaces, the designer will only need to create a single adapter for each combination of the three interfaces and the associated interface of the external environment resources specific to the hardware platform, such as onboard memory or high-speed data links.

We have termed adherence to this practice as Common Interfacing. For our set of Common Interfaces, we selected an open standard to which we would conform.

4. OCP-COMPLIANT INTERFACES

Adopting an open standard for the component interfaces may seem like a decision that would limit the flexibility of the interfaces or demand significant effort to implement. The Open Core Protocol (OCP), an openly-licensed, core-centric protocol developed by the OCP International Partnership (OCP-IP), addresses both of these concerns.

The OCP standard defines a set of fundamental signals that can be combined and configured to implement almost any conceivable interface. Rather than defining some number of standard interfaces from which designers can choose, the standard provides a structured means for describing the configuration of any interface that can be constructed from the OCP signals. The specification of a particular interface is referred to as a profile. The profile completely documents the interface, greatly simplifying the task of describing the component interfaces.

Since the features of an OCP-compliant interface are entirely determined by the designer, the interface can be as simple or as complex as the design warrants. By specifying the interface profile, the designer determines the effort and resources that will be required to implement the interface. Since the OCP signals are so fundamental and the minimum set of required signals is so basic, extremely simple interfaces can be created

We used the OCP to define three profiles to implement the interfaces described in Section 3. The profiles are named System, Memory, and Dataflow, respectively. The following three subsections describe the interfaces and the signals that comprise them. For an explanation of the function of the individual OCP signals in these profiles see the OCP Specification [3].

4.1. System

The System profile provides overall control of the operation of the component. Figure 1 shows the OCP signals included in the profile. OCP connections are point-to-point only, having a single Master interface, which initiates all accesses that utilize the data busses using write or read commands,

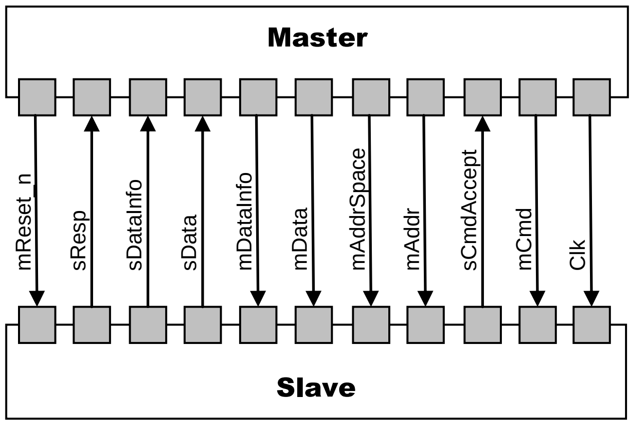

Figure 3: Memory OCP Profile

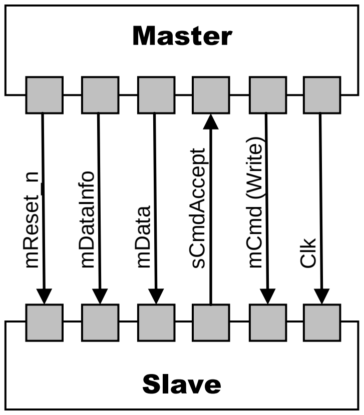

Figure 4: Dataflow OCP Profile

and one Slave interface. In the case of the System profile, the interface on the component is the Slave and a system controller would implement the corresponding Master interface.

To facilitate the platform concepts of resource allocation and power management, a set of operational states are defined. Figure 2 shows the typical state transition flow. These states are based on those used for software components in the Software Communications Architecture (SCA). The system controller directs the component into the various states by writes in which the initialize, start, stop, test, and release directives are encoded on the mData bus. The controller can query the component as to its current state by reads in which the Inactive, Configure, Active, and Test states are encoded on the sData bus.

The profile also provides several out-of-band Slave signals, Status, sInterrupt, and sError, with which the component can immediately inform the system controller of a condition without a query from the Master side of the connection.

4.2. Memory

The Memory profile was chosen to provide a full-featured interface while bounding overhead and implementation effort by excluding more complex features available in OCP. Figure 3 shows the signals included in the profile. Both read and write access is provided to an addressable memory space. This type of access is typical for memory-mapped peripherals. In our design, a Memory interface is typically used to set properties and query status for a component. In this case the component implements the Slave side of the interface and a system controller has the Master.

Any unused signals for a particular instance of a profile can be tied off to a benign default value. For instance, we set the mAddrSpace, mDataInfo, and sDataInfo signals all to default values for our design. However, if we develop new components that use these signals they will still be compatible with the original components.

4.3. Dataflow

The Dataflow profile was chosen to provide a high performance path with minimal overhead. The resources required for a Dataflow interface will be on the same order as those required for a basic custom interface implementation. Figure 4 shows the signals included in the profile. The inclusion of the sCmdAccept signal allows a Slave to throttle the rate at which data is transferred over a Dataflow connection, effectively enabling flow control.

In our design, the Dataflow interface is the most common means for components to interact. A component that generates data outputs it via a Master interface, and a component that consumes data receives it via a Slave interface. Many components are connected together in series with Dataflow connections and via the flow control provided by the interfaces the processing throughput rate is regulated by the slowest connection or component.

4.4. Usage Details

Each component will have one System interface and zero or more each of the Memory and Dataflow interfaces. Each interface defines a distinct interaction with the component. While the configuration of the signals, e.g. the number of bits in the mData bus, is fixed for the System profile, the signal configurations for the Memory and Dataflow could vary for each interface of each component. However, to simplify the process of connecting the interfaces between two components, we set a global OCP bus width for a design. For example mData is 256 bits wide for all Dataflow interfaces, therefore the mData master port on a component can be directly connected to any mData slave port. The width of the actual data to be transferred over the mData bus is determined by the components on either side of the bus.

While the full 256-bit bus is present in simulation, the unused bits are optimized away in synthesis.

5. BEYOND PROTOCOL

The OCP profiles describe the signals present in an interface and the protocol they use to transfer data. This is all that is required for transport of the data between components. However, for components to manipulate the data they must have a common interpretation of what the data is. For the System profile, the data signal uses a specific encoding to define the various transitions and states, but the other profiles define the data signal based on the requirements of the component.

Working from the practices used for SCA software components, the data encoding used by each interface is defined using Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL). Each interface defines some number of functions each taking some number of arguments. These arguments are concatenated with an ordinal value specifying the function to form the data signal of the profile. The generated code for the component provides a routine to convert between a structure containing each argument and the flattened representation used by the data signal. Although any set of functions can be defined, to maximize the flexibility in connecting components, the interface definition should be kept as general as possible. Only those components which use the same profile and same interface definition can be directly connected.

Since parallelism of component algorithms is of special importance to achieve high throughput while allowing the proper balance of clock rate to resources, multiple function calls can also be mapped into the same transaction. This does not require a new interface and components with different levels of parallelism can be connected using a simple generic adapter.

6. XML SYSTEM DESCRIPTION

The use of the Common Interfacing concept alone is helpful in the process of instantiating and interconnecting components into a system by hand. However, greater benefit can be gained by leveraging the consistent, structured nature of the interfaces. We created a set of tools to automate the instantiation and interconnection process of components based on a structured description of their interfaces and connections within the system.

We created a set of XML schema to describe the interfaces and other characteristics, e.g. VHDL generics, of each component and to describe the connection of multiple components to form a system. Concurrently, The SPIRIT Consortium was undertaking a similar yet much larger-scoped effort. This consortium of companies has released a specification, IP-XACT, which provides the definition of XML schema to neutrally describe intellectual property (IP) [4]. This IP can be a component or a system of components. A common goal we have is to completely describe the interfaces of a component in a standard way so that tools can be used to automatically integrate components into a verification environment or into a system that can be deployed to hardware. We have not yet determined the effort required to transition to the IP-XACT schema, but we are interested in exploring that option.

In the mean time, we are using our schema with success. In addition to describing the component interfaces, the XML specifies build-time variable parameters of the components, for instance, selecting the method of modulation that a modulator component will perform. Also, the XML specifies how components are interconnected to form a complete system that can be synthesized. This function is extended to serve as the basis for an automatically-generated verification environment.

To make the task of creating and modifying the XML description of the components and the system we leveraged a graphical schematic entry tool. This GUI presents a means for the designer to create components, interfaces, subsystems, and systems and set properties for each. Components can be connected to form a system, and the system can be partitioned into subsystems. The designer can create libraries of these components and systems to facilitate the rapid creation of new systems.

Additionally, in the system-level view the designer can include various models of a particular component and connect them to verification objects. This information is also captured in the XML system description and utilized by downstream tools for the automatic generation of source code.

7. AUTOMATIC GENERATION

7.1. Simulation Environment Generation

A significant focus of our system implementation process is the functional verification of the system via simulation. To aid in the execution of this stage of the process, we automated the task of creating a simulation environment that incorporates various models of the components. The system-level XML specifies which models of a component, e.g. high-level SystemC or synthesizable VHDL are included in the simulation. Each component can use a different level of model or multiple levels for integrated runtime verification. A collection of scripts processes the XML, pulls in the appropriate component worker models, and creates a SystemC-based, mixed-language simulation. Figure 5 depicts the inputs and a high-level view of the output simulation environment. A set of configuration files specify Dataflow interfaces that should be compared and

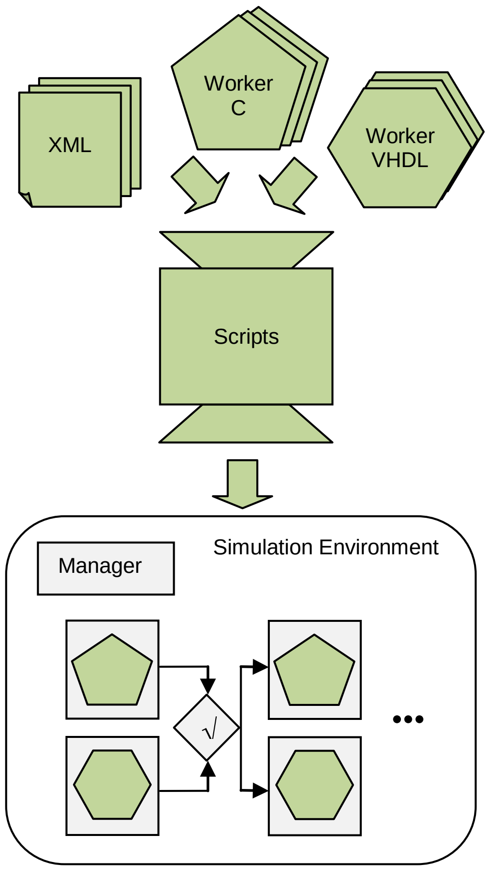

Figure 5: Simulation Environment Generation

Dataflow interfaces that should be sourced from or sunk to a file. The component Common Interfaces allowed for a standard set of adapters to be created to integrate the component worker models into the simulation environment.

7.2. Synthesizable System Generation

The same system-level XML that is used to construct the simulation environment is also used to generate an OCP-compliant system in synthesizable VHDL. Figure 6 shows the flow. The automatic generation scripts pull in component worker VHDL, and, based on the system-level XML, they completely construct an OCP-compliant VHDL entity that instantiates and interconnects all the components.

The XML system description is used to collect components into subsystems. These subsystems can correspond, for instance, to a single FPGA in a multi-FPGA platform. The scripts will generate OCP-compliant VHDL entities that implement the subsystems. Each of these entities is ready for instantiation in the application area of an FPGA on the target hardware platform. Components can be rapidly redistributed amongst a number of subsystems to target the design to platforms possessing different FPGA resources or simply to adjust the distribution of components

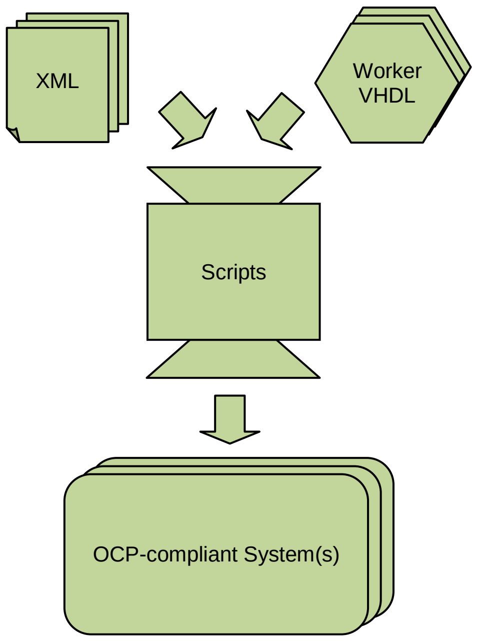

Figure 6: Synthesizable System Generation

on the same hardware platform as components are added to or removed from the design.

8. CONCLUSIONS

Adhering to a set of common interfaces for components has several advantages. Interconnecting these components becomes trivial, even to the point that scripts can automate the process. Component interfaces are easily adapted to platform-specific interfaces once per platform rather than once per platform per component per interface. Documenting a specific component's interfaces is reduced to providing a description of the data types or memory maps on its interfaces, since the details of the signaling on the three Common Interfaces can be specified once for all components. Furthermore, Common Interfacing encourages component reuse. By making a small sacrifice in flexibility, significant benefits are realized.

9. REFERENCES

[1] K. Skey, J. Bradley, and K. Wagner, "A Reuse Approach for FPGA-Based SDR Waveforms," Milcom 2006, October 2006.
[2] J. Hogg and F. Bordeleau, "Optimizing Portable SDR Software," SDR Forum Technical Conference, 2007.
[3] OCP International Partnership, "Open Core Protocol Specification, Release 2.2", January 2007.
[4] The SPIRIT Consortium, "IP-XACT v1.4: A Specification for XML Meta-data and Tool Interfaces", March 2008.