Applied Common Interfacing Techniques using OCP

Erich C. Whitney

The MITRE Corporation
Bedford, MA, USA

www.mitre.org

ABSTRACT

One of the most challenging problems in dealing with IP in large FPGA and SoC designs is the growing number of different interfaces required to connect components together. Using common interfacing leads to greater productivity and lends itself to more automation in design and verification. This paper demonstrates common interfacing techniques used on a large scale FPGA-based communications system developed at MITRE. This approach can be applied to 3rd party IP as well as internally developed IP as it leverages the Open Core Protocol International Partnership (OCP-IP) standard in addition to XML-based code generation techniques. A real-world design example is shown to demonstrate these concepts.

Table of Contents

Table of Figures

Table of Tables

1. **Acknowledgements**

I would like to give credit to John Bradley and Karl Wagner of MITRE for their contributions to this work. This paper presents their work as a foundation to work I have contributed to on the HDR-RF program. I have referenced their published work at the end of the paper and I highly encourage the reader to refer to these references for more perspective on the topic. I would also like to thank Jose Torres for his leadership and support.

2. Introduction

The MITRE Programmable Radio Technology (PRT) Laboratory is tasked with developing strategies for efficient reuse of FPGA-based software-defined radio (SDR) waveforms. The PRT laboratory has developed a High Data Rate-Radio Frequency (HDR-RF) Test Waveform for the purpose of ensuring that the modem hardware platforms developed by multiple contractors for the HDR-RF program have adequate computing resources to implement the operational waveform when it becomes available.

One of the concepts used to aid in the rapid deployment of this system on multiple hardware platforms is the common interfacing approach used for component development based on a set of well-defined Open Core Protocol International Partnership (OCP-IP) profiles. This paper describes this concept in detail.

3. Component-based Design

3.1. Introduction to Component-based Design

In component-based design, complex systems are broken down into simpler components. Each component can be designed in isolation and then integrated into a larger system. The platform-specific aspects of the system such as host interface and platform transport are factored out of the component design. This distinction between communication and function is an important attribute of component-based design. The communication infrastructure connects to the components using OCP interfaces so porting to a new hardware platform is a matter of creating gaskets or adapters which wrap the platform-specific interfaces with OCP.

3.2. Common Interfacing

One of the most important aspects of component-based design is common interfacing. Common interfacing allows components to be connected in a structured, well-defined manner which minimizes the number and types of interfaces supported. This may sound restrictive, but it is not unusual to find several different types of very similar interfaces in a large system. Standardizing on these interfaces makes component design and verification easier.

For designs that use third-party or legacy IP, there is a choice to be made with respect to their interfaces. If the set of common interfaces is chosen properly, then it should be fairly straightforward to create gaskets or adapters for these non-OCP interfaces. Adapting non-OCP interfaces to a common set of OCP interfaces is also beneficial for IP-reuse.

3.3. OCP-IP

The OCP specification is a standard which unambiguously defines interfaces and the associated protocol that operates on that interface. For any given interface, each signal is selected from the standard and assembled into an interface definition (or profile). The master/slave relationship is defined and the initiator/target functionality is understood because these are part of the specification. In addition to the interface definition, there is the issue of compliance. OCP compliance is a set of validation steps designed to make sure that the implementation meets the OCP specification such that there are no unintended consequences when two components are interconnected. Guaranteeing OCP compliance can be difficult, but no more so than standard verification practices, and there are commercial tools which automate some compliance checking.

4. Common Interfacing Using OCP

This section provides guidance to navigating the OCP specification with some examples.

4.1. Navigating the OCP-IP Specification

The Release 2.2 of the Open Core Protocol specification is an arduous 340 page document, therefore, it is recommended that a first time user should fully understand the sections in Part I which include theory of operation, signals and encoding, and protocol semantics as well as some details on configuration files and timing before attempting an implementation. This will provide the foundation necessary to understanding the finer details of OCP covered in Part II such as profiles and the associated timing diagrams.

The most important chapter in Part II is Chapter 11, OCP Profiles. This chapter addresses the biggest benefit to the OCP specification. An OCP profile eliminates any ambiguity in the interface definition and isolates the design intent from the implementation details. Chapter 12 on OCP Core Performance addresses the common questions raised about shared IP.

4.2. OCP Signals

Part I, Chapter 3 (titled “Signals and Encoding”) of the OCP Specification is a detailed list of all the choices available for selecting interface signals. All of the profiles in the following discussion are comprised of a small subset of the entire list of signals available in the OCP Specification. While there are numerous features available by implementing the entire set of signals and capabilities (OCP Specification, Chapter 3, Tables 1-14), these features were not required for the targeted application, see Table 1-Table 4 below. The first set of signals fall into the dataflow category (Table 1). These are the signals that do the bulk of the work in the interface. The second set comes from the Sideband Signals section (Table 4) and comprise a few optional signals implementers may find useful.

4.2.1. Basic Signals[1]

Signal Name	Width	Driver	Function
Clk	1	Varies	Clock input
MCmd	3	Master	Encoding of the transaction action requested by the master
MData	Configurable	Master	Write Data
MAddr	Configurable	Master	Explicit transfer address—not used by all profiles
SCmdAccept	1	Slave	Slave accepts the transfer
SData	Configurable	Slave	Read Data
SResp	2	Slave	Transfer response

Table 1 Basic OCP Interface Signals

4.2.2. Basic Signal Encodings[2]

MCmd[2:0]	Command	Mnemonic	Request Type
0 0 0	Idle	IDLE	(none)
0 0 1	Write	WR	Write
0 1 0	Read	RD	Read
0 1 1	ReadEx	RDEX	Read
1 0 0	ReadLinked	RDL	Read
1 0 1	WriteNonPost	WRNP	Write
1 1 0	WriteConditional	WRC	Write
1 1 1	Broadcast	BCST	Write

Table 2 MCmd Signal Encoding

[1] Timing diagrams for these signals can be found in the OCP Specification[4].
[2] We only implemented IDLE, WR, and RD from this list.

SResp[1:0]	Response	Mnemonic
0 0	No response	NULL
0 1	Data valid/accept	DVA
1 0	Request failed	FAIL
1 1	Response error	ERR

Table 3 SResp Signal Encodings

4.2.3. Sideband Signals

Signal Name	Width	Driver	Function
MReset_n	1	Master	Master reset, active low
SInterrupt	1	Slave	Slave interrupt, typically to the system controller
SError	1	Slave	Slave error
Status	Configurable	Core	Core status information

Table 4 Sideband OCP Interface Signals

4.3. OCP Compliance

OCP compliance is the topic of Part III of the OCP Specification. Before an interface can be OCP compliant, its interface profile must be OCP compliant. Profile compliance is an important concept because it sets the foundation for what makes the interface hold together. Compliance issues are a side effect of a specification which is highly configurable and some subtle nuances of the specification can be difficult to appreciate on first reading.

Once the profile is correctly specified, the interface can be checked for compliance. In practice most of the issues of compliance will come out during verification and validation of the functionality, however, compliance checking is a good idea if you're going to publish or release your profile.

4.4. OCP Profiles

This section outlines examples of the three OCP profiles used on the HDR-RF program. The System Profile was chosen to provide a generalized method for controlling and coordinating components. The Memory Profile provides a generalized memory access method which can be used for simple memory-mapped control registers or connecting components to an external memory controller. And finally the Dataflow Profile is used to provide a streaming-type data interface for high-performance, low-latency, low overhead connection between components.

4.4.1. The System Profile

The System Profile is designed to provide the necessary control and status interface to a component. This is a generic interface that allows control over the state of the component in a generalized way to handle interrupts and error status. The Memory Profile, covered in the next section, is used for cases where the component requires programmable parameters and/or detailed status information.

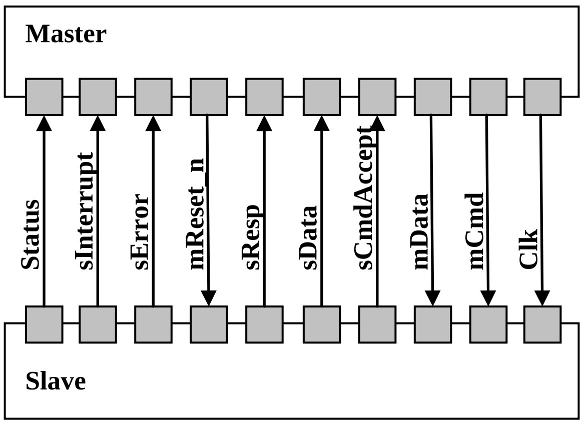

Figure 1 System OCP Profile

The System Profile is designed such that each system component implements the slave interface and a system controller implements a master interface for each instantiated component. This is a simple point-to-point connection and requires very few actual wires to implement.

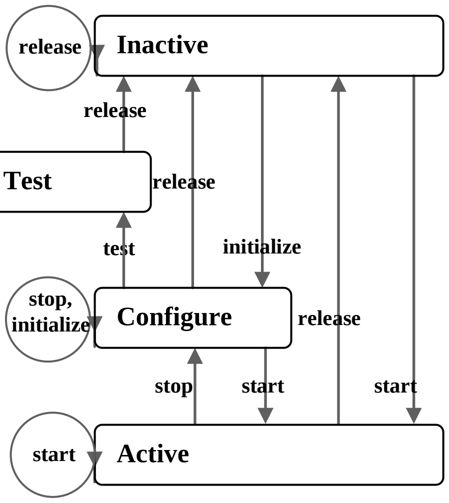

Figure 2 System OCP Profile States

The System Profile states are based on those used for the software components in the Software Communications Architecture (SCA [8]). This association between OCP and SCA is not strictly necessary however, it proved to be useful. The system controller instructs each component to transition to its next state via encodings on the mData bus and it can query the status by reading the state back on the sData bus. For an embedded system with exception handling capability, the out-of-band signals Status, sInterrupt, and sError are provided and are optional.

A typical transaction on this interface would start with the master asserting mData and mCmd. The slave would then assert sCmdAccept, sResp, and possibly sData. The transaction is complete as soon as the master receives sCmdAccept.

4.4.2. The Memory Profile

The Memory Profile provides a more full-featured interface designed to provide the signals necessary to access a component which has some type of memory. Both read and write accesses are provided and the memory address has its own signal. The component could implement either master or slave interfaces, or both. The memory could be internal to the component, external to the component, or even external to the device.

A typical transaction on this interface starts with mCmd, mAddr, and optionally mAddrSpace, mDataInfo, and mData. The slave responds with sCmdAccept, sResp, and optionally sData and sDataInfo. Note there is no out-of-band handshake with this profile. Unused optional signals may be tied off.

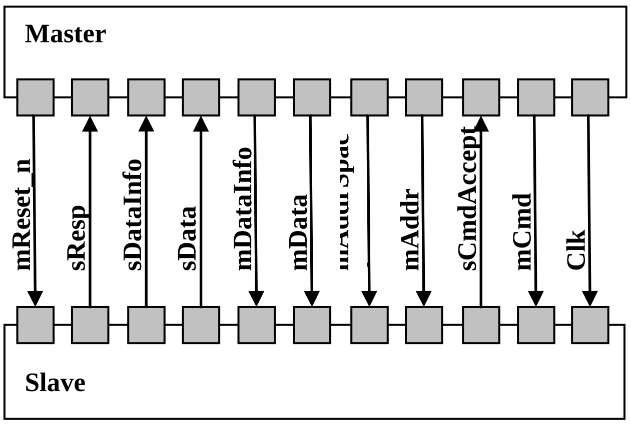

Figure 3 Memory OCP Profile

4.4.3. The Dataflow Profile

The Dataflow Profile is a high performance data path with minimal overhead.

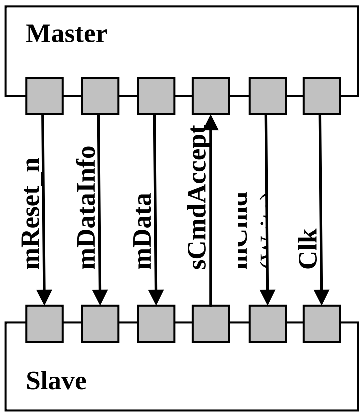

Figure 4 Dataflow OCP Profile

The master asserts mCmd and mData (mDataInfo is optional) and the slave throttles the transactions with sCmdAccept. In the case where a component has no need for flow control, its sCmdAccept may be tied high. With this profile, there are only writes and no explicit address information is provided. Typical uses for this type of interface include connecting processing components in a complex DSP pipeline. The mCmd signal is essentially a 'data valid' indicator from the master.

4.4.4. Profile Choices

In the HDR-RF implementation, the maximum width of MData and SData is set using the constants OCP_M_DATA_WIDTH and OCP_S_DATA_WIDTH , respectively (defined in a VHDL package). A generic is used to define how many of these bits are valid on each MData or SData for each specific interface. It is left up to the implementer to make sure these data widths are matched properly. With this approach, the infrastructure remains agnostic to the actual width used so each interface is truly common. Components are either written to adapt to the data width given or they impose a constraint on the number of valid data bits. In the latter case generic infrastructure components have been written to handle the resizing of the data. Synthesis typically removes any unconnected bits so there's no real penalty for defining an arbitrarily large maximum data bus width.

4.5. VHDL Interface Definitions

4.5.1. Dataflow Source Interface

Dataflow Source Master Interface

```
type DataFlowSourceMasterInterfaceType is record
  mCmd          : std_logic_vector( 2 downto 0 );
  mData         : std_logic_vector( OCP_M_DATA_WIDTH-1 downto 0 );
  mDataInfo[3]    : std_logic_vector( OCP_M_DATA_INFO_WIDTH-1 downto 0 );
  mBurstLength[4] : std_logic_vector( OCP_BURST_LENGTH_WIDTH-1 downto 0);
  mReqLast[5]     : std_logic;
  mReset_n      : std_logic;
end record;
```

[3] This field is optional.
[4] This field is reserved for future use.
[5] This field is currently unused.

Dataflow Source Slave Interface

```
type DataFlowSourceSlaveInterfaceType is record
  sCmdAccept    : std_logic;
  sThreadBusy   : std_logic;
end record;
```

4.5.2. Memory Interface

Memory Master Interface

```
type MemoryMasterInterfaceType is record
  mCmd          : std_logic_vector( 2 downto 0 );
  mAddr         : std_logic_vector( OCP_M_ADDR_WIDTH-1 downto 0 );
  mAddrSpace[6]    : std_logic_vector( OCP_M_ADDR_SPACE_WIDTH-1 downto 0 );
  mData         : std_logic_vector( OCP_M_DATA_WIDTH-1 downto 0 );
  mDataInfo[6]     : std_logic_vector( OCP_M_DATA_INFO_WIDTH-1 downto 0 );
  mReset_n      : std_logic;
end record;
```

Memory Slave Interface

```
type MemorySlaveInterfaceType is record
  sData         : std_logic_vector( OCP_S_DATA_WIDTH-1 downto 0 );
  sDataInfo[6]     : std_logic_vector( OCP_S_DATA_INFO_WIDTH-1 downto 0 );
  sResp         : std_logic_vector( 1 downto 0);
  sCmdAccept    : std_logic;
end record;
```

4.5.1. System Interface

System Master Interface

```
type SystemMasterInterfaceType is record
  mCmd          : std_logic_vector( 2 downto 0);
  mData         : std_logic_vector( OCP_SYS_DATA_WIDTH-1 downto 0);
  mReset_n      : std_logic;
end record;
```

System Slave Interface

```
type SystemSlaveInterfaceType is record
  sCmdAccept    : std_logic;
  sData         : std_logic_vector( OCP_SYS_DATA_WIDTH-1 downto 0);
  sResp         : std_logic_vector(1 downto 0);
  sError        : std_logic;
  sInterrupt    : std_logic;
  status        : std_logic_vector( OCP_SYS_STATUS_WIDTH-1 downto 0 );
end record;
```

[6] This field is optional

5. Digging Deeper into OCP

5.1. Protocol vs. Interface

While OCP provides a description of each interface signal and the protocol each component must use to transfer data, it does not provide an interpretation for what that data means. The OCP specification is there to provide a convenient means to define both the interface definition and the protocol used for communication on that interface. The OCP specification does not attempt to address what meaning of the data itself. Once the interface and protocol issues are out of the way, the discussion about what the component is doing can move up a level of abstraction making the design easier to understand.

OCP is a point to point interface definition language. OCP interfaces reduce the problem of connecting two like components to what can be thought of as a simple function call. The meaning of the data that goes across the interface, however, is left up to each component. OCP does not attempt to define how the data should be interpreted by each component.

Another aspect of the interface is performance. In communications systems, parallelism is an important method used to achieve high throughput. Properly balancing clock rate and resources to achieve the desired performance is necessary. Therefore, components with different levels of parallelism will likely need to be connected together. This typically requires a straightforward gasket component.

5.2. Using XML

Writing a large amount of code by hand is both tedious and error-prone. Common Interfacing provides opportunities for automating more of the design process to reduce errors and increase productivity. The HDR-RF program has developed a set of tools to automate this instantiation and interconnection process based on a convenient XML schema. Since most of the designs start in MATLAB, a MATLAB script is used to generate an XML design description. A PERL script is then used to generate netlists, source code, and testbenches from this XML. The scripts are configured to interconnect either VHDL or SystemC models for the components in various configurations for different testing scenarios. The SPIRIT Consortium has a very similar goal with their IP-XACT specification, however, their undertaking is considerably more comprehensive than ours.

5.3. Automatic generation for synthesis and simulation

A working XML schema provides the necessary information from which netlists and environments for simulation and synthesis are generated. Since modelling is such an important part of the development process, either high-level SystemC models or synthesizable VHDL may be included when running system simulations. This not only provides flexibility when components may not have a working implementation, but it allows for architectural exploration and a speedup in large scale simulations by allowing the appropriate level of detail for a given simulation.

The code generator can also construct a completely specified component declaration and instantiation template as a starting point for the component implementer. Device partitioning is a simple a matter of grouping components in the GUI and the scripts generate the resulting netlist. This also allows for more platform independence since the components can be easily rearranged for a given target platform and its associated resources.

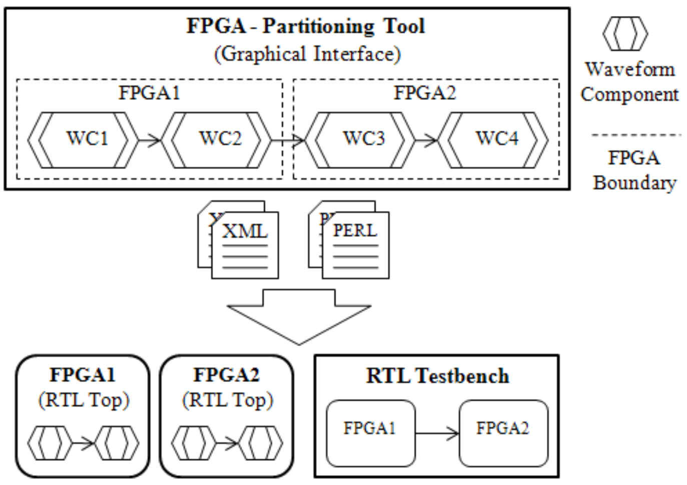

Figure 5 FPGA Partitioning

Figure 5 shows the tool flow from a graphical representation of the components (in this case using MATLAB). The user selects the component grouping that makes sense for the target FPGAs. From this, the component boundary for each FPGA is defined and an XML representation is generated by the graphical tool. From this XML, PERL scripts parse the XML and produce an FPGA RTL top level design for each FPGA as well as an RTL testbench. All of this code is easily generated because the interfaces are well defined. The one piece of manual effort required at this point is to take the RTL top level and stitch it into the specific FPGAs chosen for the deployment. It is possible to use templates for this part of the process, however, each target FPGA device needs an appropriate template created by hand.

6. **Test Waveform Description**

6.1. Overview

The HDR-RF Test Waveform program at MITRE provides a working example of common interfacing in a deployed system. The details of this system can be found in reference [3]; refer to this paper for more details. Using the techniques outlined in this paper, the HDR-RF Test Waveform program realized the benefits of portability, reusability, and automation that common interfacing enabled.

One of the platforms chosen to host the HDR-RF Test Waveform consists of 15 user-programmable FPGAs on multiple cards in a commercial 6U VME chassis. This system isn't necessarily similar to the platform on which the final waveform will be deployed, so portability is very important. Furthermore, managing this many designs is very cumbersome in a traditional FPGA design flow. The following diagram shows the components needed in the test waveform.

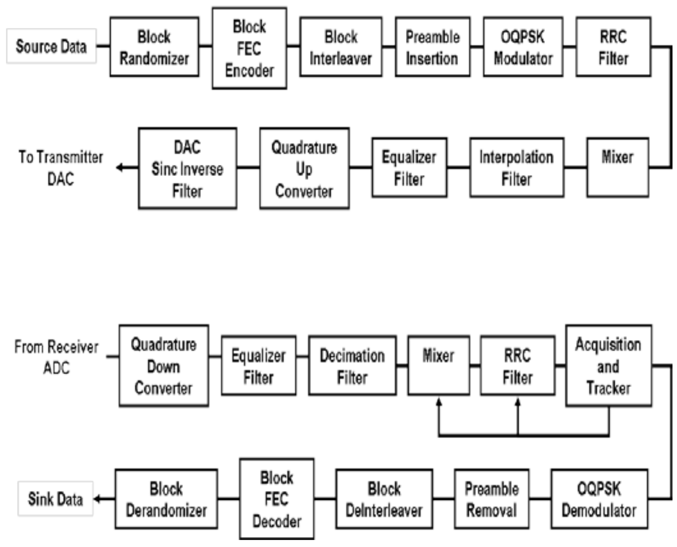

Figure 6 Components of the Test Waveform

The entire test asset that makes up the test waveform is shown below. This is a block diagram of the system that shows the FPGAs in context.

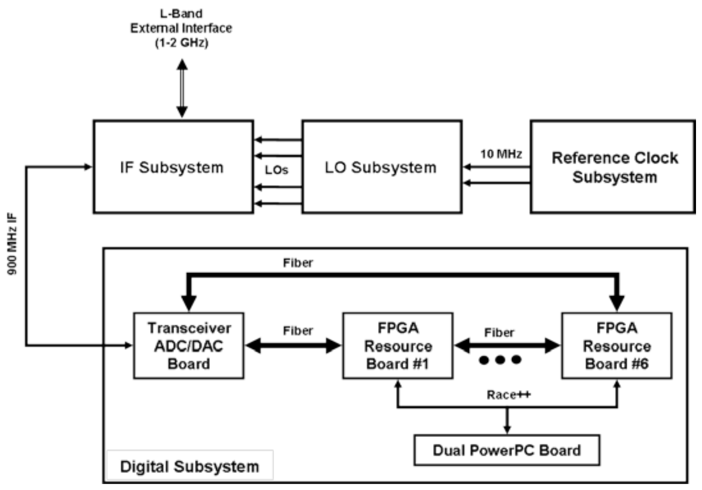

Figure 7 Subsystems of the Test Asset

The simulation infrastructure used to support development of this system is shown in the next figure.

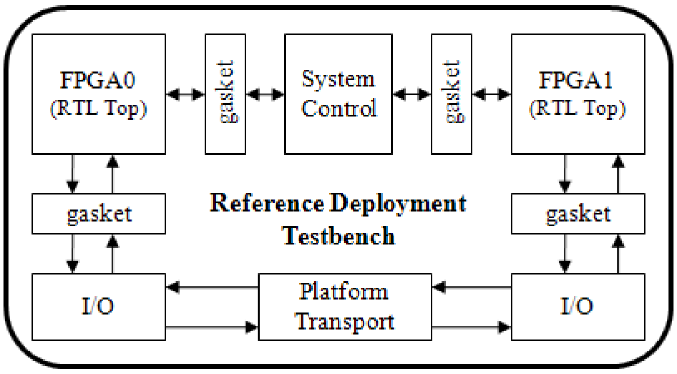

Figure 8 Simulation Infrastructure

Figure 8 shows only two FPGAs, however, the deployed system supports simulating all 12 of the main FPGAs in a similar fashion. Note the remaining 3 FPGAs are on the transceiver board. The FPGAs in this application use serial meshes for chip-to-chip communication. The on-board connections consist of multiple copper links while the board-to-board interconnect uses multiple fibre-optic connections. These interconnects are not shown in the component diagram (Figure 6) because they are platform-specific and are dependent on the specific deployment. They are, however, an important part of the system. Once the waveform has been partitioned into multiple FPGAs, these transport components are added. The next section takes a look at this platform-specific transport as an example of using OCP to seamlessly insert chip-to-chip communications without altering the waveform's function.

6.2. Implementing a Serial Transport with OCP

As mentioned earlier, the HDR-RF waveform uses a Dataflow OCP Profile to move data between components. If we need to break a connection between two components to partition the design into multiple FPGAs, then it makes sense to use the Dataflow OCP Profile so the transport can be simply dropped in.

In this case, we chose to use the Xilinx Aurora IP for this chip-to-chip interconnect because it handles multiple channel bonding, very low overhead on top of its native 8b/10b encoding, has provisions for flow control, and its available for the devices on the platform. The Xilinx Aurora IP does not, however, have a native OCP interface[7].

The Xilinx Aurora interfaces uses something they call Local-Link which is a fairly straightforward parallel connection to the Aurora IP and it maps very easily into the Dataflow OCP Profile outlined above. Like most IP, Aurora has some control and status registers that the user might want access to. For this, we implemented a Memory interface and a System interface for control.

The Xilinx CORE Generator code generator for Aurora creates one specific instance of the hundreds of configuration combinations available. Their code generator does not create a parameterized block. If you're trying to support a handful of different Aurora configurations you have to run the generator multiple times and manage all of the generated code. To solve this problem a code generator was written (in Python) which takes a subset of Aurora configurations needed to support a given deployment platform and runs Xilinx CORE Generator for you to create the necessary files, wraps these files in a VHDL module and dumps the code into a directory where the implementation tools can compile and link the design. This Python program also implements some workarounds for problems in Xilinx's generated code which doesn't compile properly in some cases. What the Python program does and how it does it is a topic for another paper. The script has been used to generate working Aurora cores for both Virtex-2 Pro and Virtex-5 devices. And the nice thing about this approach

[7] It is the author's hope that IP vendors will see the benefits to using OCP and widely adopt it in the industry.

is that we can move the IP from one platform to another and the waveform code doesn't change because the interfaces are the same for each platform.

6.3. Dataflow Using a FIFO Model

The easiest way to treat the Aurora core is using a FIFO model. On the transmit side data a master pushes into a FIFO and the Aurora core sends over the link. When the FIFO can no longer accept data, the slave asserts flow control and the upstream components have to handle the backpressure. On the receive side, the Aurora core takes its received data from the link and pushes it into a FIFO where the receive master attempts to send it to its connected slave. If the downstream components assert flow control, then the receive FIFO will back up until it has to assert flow control over the link to back-pressure its corresponding transmitter. All of this flow control is handled by the Aurora core when it is configured for full-duplex operation. Things get a bit trickier when the Aurora doesn't have a full-duplex link. This is yet another detail not presented in this paper.

The transmit FIFO has a very simple interface which is synchronous to its input side:

```
tx_fifo_data: Parallel data to be transmitted by the Aurora core
              (connected directly to the TX FIFO input)
tx_fifo_rfd:  Transmit FIFO is ready for more data
tx_fifo_nd:   Tells the FIFO that new data is present on tx_fifo_data
```

This is the combinatorial logic needed to connect the FIFO to the Transmit Dataflow OCP Slave Interface:

```
in_proc : process(tx_fifo_rfd, dfTxMasterIn)
  begin
    if (tx_fifo_rfd = '1'
        and to_integer(unsigned(dfTxMasterIn.mCmd)) = OCP_CMD_WRITE) then
      tx_fifo_nd                       <= '1';
      dfTxSlaveOut.sCmdAccept          <= '1';
      dfTxSlaveOut.sThreadBusy         <= '0';
    else
      tx_fifo_nd                       <= '0';
      dfTxSlaveOut.sCmdAccept          <= '0';
      dfTxSlaveOut.sThreadBusy         <= '1';
    end if;
  end process in_proc;
```

The receive FIFO has a similarly simple interface which is synchronous to its output side:

```
rx_fifo_data: Parallel data received by the Aurora core
              (connected directly to the RX FIFO output)
rx_fifo_rdy:  Receive FIFO has data
rx_fifo_ack:  Tells the RX FIFO to pop the head of the FIFO
```

This is the combinatorial logic needed to connect the FIFO to the Receive Dataflow OCP Master Interface:

```
out_proc : process(dfRxSlaveIn, rx_fifo_rdy)
  begin
    if (dfRxSlaveIn.sCmdAccept = '1')
       and (rx_fifo_rdy = '1')
       and (dfRxSlaveIn.sThreadBusy = '0') then
      rx_fifo_ack <= '1';
    else
      rx_fifo_ack <= '0';
    end if;


    if (rx_fifo_rdy = '1') then
      mCmd        <= std_logic_vector(to_unsigned(OCP_CMD_WRITE, mCmd'length));
    else
      mCmd        <= std_logic_vector(to_unsigned(OCP_CMD_IDLE, mCmd'length));
    end if;
  end process out_proc;
```

Notice that these processes are combinatorial. This was a conscious decision made to keep the complexity of the interface down and reduce latency. The data is registered at the FIFO boundary so all we are doing here is adding about one LUT worth of logic to the control signals. This is one of those design choices you make when creating your OCP profiles. In practice this has worked well and does not create problems with timing closure. It is fairly trivial to add a pipelined repeater component into the dataflow stream to increase throughput on large designs where components cannot be placed adjacent to one another.

7. Conclusions

Common interfacing is a technique essential for rapid prototyping but it applies equally well as a disciplined approach to system design. The OCP-IP Specification provides the necessary set of common definitions needed to unambiguously define any interface and gives the designer flexibility to accommodate a software and hardware protocol on top of that interface. This paper has shown a working example of these concepts on a multiple-FPGA implementation of a complex HDR-RF Test Waveform. Furthermore, a method for using XML as a tool for automating the synthesis and simulation environments needed to successfully deploy the system demonstrates the versatility of this approach.

8. References

[1] J. Bradley, K. Wagner, "Automating FPGA-Based System Implementation with Common Interfacing," SDR Forum Technical Conference, 2008.
[2] K. Skey, J. Bradley, and K. Wagner, "A Reuse Approach for FPGA-Based SDR Waveforms," MILCOM 2006, October 2006.
[3] J. Torres, "The HDR-RF Test Waveform – An Innovative Risk Reduction Product for FPGA-Based SATCOM Modems", MILCOM 2008, November 2008.
[4] Open Core Protocol Specification, Release 2.2, OCP International Partnership, http://www.ocpip.org, January 2007.
[5] IP-XACT v1.4: A Specification for XML Meta-data and Tool Interfaces, The SPIRIT Consortium, http://www.spiritconsortium.org, March 2008.
[6] LogiCORE IP Aurora v2.9, UG061 (v2.9), Xilinx, http://www.xilinx.com, March 2008.
[7] Virtex-5 LogiCORE Aurora v2.9 User Guide, UG353 (v2.9), Xilinx, http://www.xilinx.com, March 2008.
[8] Software Communications Architecture, http://sca.jpeojtrs.mil

9. Index