Using OCP/SystemC Channels with an Embedded Bluespec Model

Revision: 15 May 2007

Copyright © 2000 – 2007 Bluespec, Inc.

A SystemC model can be generated from a Bluespec SystemVerilog design using the `-systemc` flag during the `bsc` link stage (see *User Guide*). The generated SystemC model has a clocked, signal-level interface which reflects the (flattened) top-level interface of the BSV design.

The OCP foundation supplies standard libraries for implementing and interacting with OCP interfaces in SystemC models. This application note describes how to use these libraries with a BSV design which contains OCP interfaces or sub-interfaces, so that they can interact at the TLM-level with the SystemC environment in which they are embedded.

Contents

1 BSV Support for OCP Interfaces

There are many ways to incorporate OCP functionality into a BSV model:

- using the AzureIP bus foundation IP library
- using a special-purpose OCP-IP bus implementation library
- writing an OCP interface adapter around some custom logic

These approaches differ in the implementation details, ease of implementation, flexibility and maintenance costs, but they all must provide a model with a suitable OCP interface.

The OCP standard allows a very flexible port structure with many optional ports. An OCP interface in BSV would typically consist of four (always ready and always enabled) method groups to:

- pass a request
- pass acknowledgement of the request
- pass a response
- pass acknowledgement of the response

Related bus signals originating at an interface may be represented as individual single-argument methods, whereas the same bus signals would be arguments to one method in the receiving interface. One way to define an OCP master interface is shown below:

```
interface OCP_Master_Ifc;
   (* prefix = "" *) interface OCP_MasterReq_Ifc  masterReq;
   (* prefix = "" *) interface OCP_MasterResp_Ifc masterResp;
endinterface: OCP_Master_Ifc

// The request part of the master interface
(* always_ready *)
interface OCP_MasterReq_Ifc;
   (* result = "MCmd" *)
   method MCmd                          mCmd;
   (* result = "MAddr" *)
   method UInt#(TMul#(addr,addr_wdth)) mAddr;
   (* result = "MData" *)
   method Bit#(TMul#(mdata,data_wdth)) mData;
   ...
   // Accept coming from the slave
   (* enable = "SCmdAccept" *)
   method Action sCmdAccept();
endinterface: OCP_MasterReq_Ifc

// The other half of the master interface, where the slave puts it response
(* always_ready *)
interface OCP_MasterResp_Ifc;
   (* always_enabled, prefix = ""  *)
   method Action putResponse ((* port = "SResp" *)
                              SResp sResp,
                              (* port = "SData" *)
                              Bit#(TMul#(sdata,data_wdth)) sData,
                              ... );
   (* result = "MRespAccept" *)
   method Bit#(respaccept) mRespAccept;
endinterface: OCP_MasterResp_Ifc
```

1.1 OCP Interfaces in the Generated SystemC

The SystemC model generated from an OCP interface description in BSV will have ports for each bus signal (defined by the method arguments and return values). These ports will have type bool for single-bit ports and type sc_bv<N> for N-bit wide ports. Ports for method arguments are sc_in<...> ports and for method return values are sc_out<...> ports. In addition, ther should be some clock port on the module (of type sc_in<bool>) which serves as the OCP bus clock.

An OCP interface generated from a BSV description would typically look like:

```
/* SystemC model definition */
SC_MODULE(...) {

 /* clock and reset inputs */
 public:
  sc_in<bool > CLK;
  sc_in<bool > RST_N;

 /* method ports */
 public:

  /* masterResp_mRespAccept method ports */
  sc_out<bool > MRespAccept;

  /* masterResp_putResponse method ports */
  sc_in<sc_bv<2 > > SResp;
  sc_in<sc_bv<32 > > SData;

  /* masterReq_sCmdAccept method ports */
  sc_in<bool > SCmdAccept;

  /* masterReq_mData method ports */
  sc_out<sc_bv<32 > > MData;

  /* masterReq_mCmd method ports */
  sc_out<sc_bv<3 > > MCmd;

  /* masterReq_mAddr method ports */
  sc_out<sc_bv<24 > > MAddr;

  ...
```

Such a model can be instantiated inside a larger SystemC environment, as long as the environment is able to interoperate using the OCP bus through the clocked, signal-level interface provided. When the environment cannot use the provided interface directly, channel adapters can be used to convert between the signal-level interface and the external interface. The use of standard OCP TLM channel adapters is described in Section 2.2.

2 OCP Libraries for SystemC

The OCP-IP supplies OCP libraries for use in SystemC environments. These libraries provide standardized mechanisms for working with IP blocks with OCP interfaces both at the TLM and signal-level.

The OCP interconnect modeling methodology recognizes 4 levels of abstraction:

- TL0 - Clocked, signal-level interconnect
- TL1 - Clocked, transaction-level interconnect
- TL2 - Timed, transaction-level, protocol-independent interconnect
- TL3 - Untimed, transaction-level, abstract interconnect

The interface for an OCP bus in a SystemC model generated from `bsc` is at the TL0 level.

In addition to SystemC models which implement OCP channels at each level of abstraction, the OCP-IP also provides adapter modules capable of bridging between channels at different levels.

Both the TLM channels and the adapters are documented in their respective packages downloadable from ocpip.org.

2.1 TLM Channels

The OCP Channel package supplied by the OCP-IP is a SystemC library which implements OCP channels at different levels of abstraction.

TL0 Channel The TL0 channel requires no special interconnect module. Ports of the OCP interface can be individually bound using the standard SystemC signal channels.

TL1 - TL3 Channels OCP bus configurations are represented by templated classes which determine the types used for bus address and data lines, the set of signals included in the bus, etc. The OCP interconnects closely follow the standard SystemC idiom of interfaces, ports and channels.

Classes used with the OCP TL1 Channel	
Class	**Purpose**
OCP_TL1_DataCI<...>	Describes bus address and data width
OCP_TL1_MasterIF<...>	Interface for a TL1 OCP master
OCP_TL1_MasterPort<...>	A TL1 OCP master port
OCP_TL1_SlaveIF<...>	Interface for a TL1 OCP slave
OCP_TL1_SlavePort<...>	A TL1 OCP slave port
OCP_TL1_Channel<...>	A TL1 OCP interconnect channel

These channels provide a transaction-level interface which abstracts away from the signal-level details of the TL0-level. For example, given an OCP_TL1_MasterPort bound to an OCP channel, you can initiate an OCP request using SystemC code like:

```
Request req;

req.MCmd  = OCP_MCMD_WR;
req.MAddr = 42;
req.MData = 1;

while (!port->startOCPRequest(req))
   wait(port->RequestEndEvent() | port->SThreadBusyEvent() | port->ResetEndEvent());
```

Full documentation of the master and slave interfaces and the other channel types is provided in the OCP channel distribution.

2.2 Channel Adapters

When a SystemC model generated from BSV, which has a TL0-level interface, needs to be embedded in a SystemC environment which utilizes transaction-level OCP channels, channel adapter classes must be used to bridge between the transaction-level and signal-level interfaces. Standardized channel adapters are provided for this purpose by the OCP-IP.

OCP channel adapters are OCP channels which have a higher-level interface at one end and a lower-level interface at the other. For instance, an OCP2_TL0_TL1_Slave_Adapter binds to the slave port of an OCP_TL1_Channel on one side and to the individual ports of a TL0-level OCP master interface on the other.

3 TL1 Channel Example

This section provides an example of how a SystemC model generated from a BSV source model can be incorporated into a SystemC environment using TL1-level OCP masters and slaves. The example utilizes a very simple OCP bridge or relay which simply passes OCP requests from a master on one OCP bus along to a slave on another OCP bus and forwards responses back to the master. The OCP bridge is implemented in BSV which is converted to SystemC using the `-systemc` flag during the `bsc` link stage. Stimulus is provided by a SystemC model which initiates requests through an OCP master interface, and a SystemC model of the target replies to the requests through an OCP slave interface. Both the initiator and target OCP interfaces are TL1 transaction-level interfaces, but the OCP bridge model uses TL0 signal-level interfaces. The initiator and target are connected to the bridge through a pair of OCP TL1 channels representing the two OCP buses, each coupled with an OCP TL1/TL0 channel adapter.

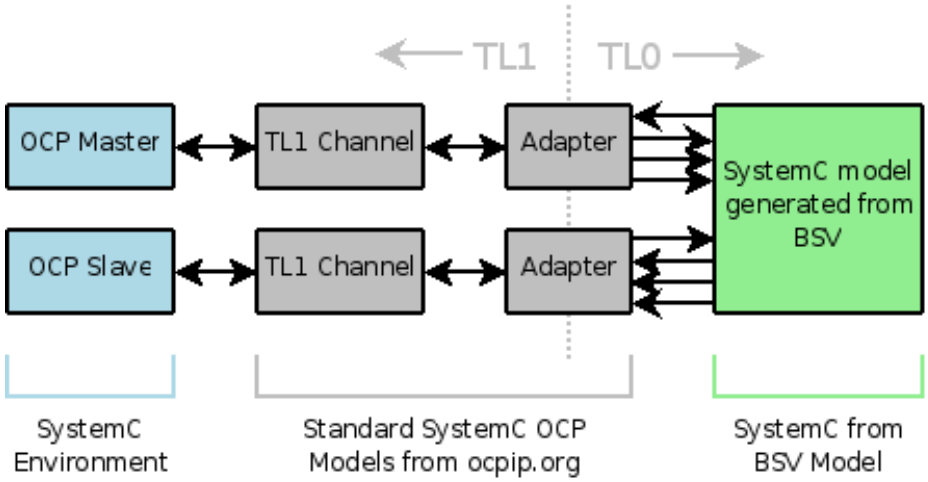

Figure 1: Example system with OCP TL1 master and slave connected to a SystemC model with TL0 interfaces.

3.1 Instantiating the Model, Channels and Adapters

Using the SystemC OCP channels and adapters requires installing these packages (they can be obtained from the OCP-IP website at ocpip.org) in addition to the regular SystemC libraries. Once the OCP channel and adapter libraries are installed, they can be included in the SystemC environment:

```
#include "ocp_tl_param_cl.h"
#include "tl1/ocp_tl1_channel.h"
#include "ocp2_tl0_tl1_master_adapter.h"
#include "ocp2_tl0_tl1_slave_adapter.h"
```

The SystemC model generated from the BSV bridge design is a standard SystemC module which can be instantiated normally, alongside the master and slave modules:

```
// Make the OCP master
mkMaster master("master");

// Make the OCP slave
mkSlave slave("slave");

// Make the OCP bridge
mkBridge bridge("bridge");
```

The OCP TL1 channels from the OCP-IP library are conceptually similar to standard SystemC channels such as signals, except that they implement a much more complex protocol across the channel. The TL1 channels are clocked, so they take a clock for the bus as an argument to the constructor. Each channel has a master and a slave port which can be bound to OCP TL1 ports on the master and slave modules.

```
// Connect the master and slave to two independent bus channels
OCP_TL1_Channel<BusDataCl> ocp_bus1("bus1", &clk);
master.port(ocp_bus1);

OCP_TL1_Channel<BusDataCl> ocp_bus2("bus2", &clk);
slave.port(ocp_bus2);
```

The two remaining ports on the OCP TL1 bus channels cannot connect directly to the bridge model because it uses OCP TL0 interfaces which consist of individual ports for the various signals groups of the OCP bus. To connect the TL1 channels to the TL0 interfaces on the bridge model, we use two OCP TL1/TL0 channel adapters. (We show the connections for only one of the adapters — the other adapter is connected in the same way).

```
// Adapter for bus1
OCP2_TL0_TL1_Slave_Adapter<BusDataCl>
  tl0_adapter1("adapter1", OCP2_TL0_MasterPorts::Factory<master_tl0_ports>());

// set the clock for sampling the TL0 signals
tl0_adapter1.Clk(clk);
tl0_adapter1.setClockPeriod(sc_time(10,SC_NS));

// connect the adapter to the TL1 bus channel
tl0_adapter1.SlaveP(ocp_bus1);
```

Connecting the adapter to the bridge model's OCP interface requires declaring signals for the elements of the TL0 interface and binding them to the bridge model ports and the adapter's ports.

```
// TL0 interface signals for bus1
sc_signal< sc_bv<3> >  bus1_MCmd;
sc_signal< sc_bv<24> > bus1_MAddr;
sc_signal< sc_bv<32> > bus1_MData;
sc_signal< sc_bv<2> >  bus1_MThreadID;
sc_signal< bool >      bus1_MRespAccept;
sc_signal< sc_bv<2> >  bus1_SResp;
sc_signal< sc_bv<32> > bus1_SData;
sc_signal< sc_bv<2> >  bus1_SThreadID;
sc_signal< bool >      bus1_SCmdAccept;

// connect the adapter to the TL0 interface signals for bus1
master_tl0_ports& tl0_ports1 = dynamic_cast<master_tl0_ports&>(tl0_adapter1.m_tl0Ports);
tl0_ports1.MCmd(bus1_MCmd);
tl0_ports1.MAddr(bus1_MAddr);
tl0_ports1.MData(bus1_MData);
tl0_ports1.MThreadID(bus1_MThreadID);
tl0_ports1.MRespAccept(bus1_MRespAccept);
tl0_ports1.SCmdAccept(bus1_SCmdAccept);
tl0_ports1.SResp(bus1_SResp);
tl0_ports1.SData(bus1_SData);
tl0_ports1.SThreadID(bus1_SThreadID);

// connect the bridge to the TL0 interface signals for bus1
bridge.slave_MCmd(bus1_MCmd);
bridge.slave_MAddr(bus1_MAddr);
bridge.slave_MData(bus1_MData);
bridge.slave_MThreadID(bus1_MThreadID);
bridge.slave_MRespAccept(bus1_MRespAccept);
bridge.slave_SResp(bus1_SResp);
bridge.slave_SData(bus1_SData);
bridge.slave_SThreadID(bus1_SThreadID);
bridge.slave_SCmdAccept(bus1_SCmdAccept);
```

Connecting the other adapter in a similar fashion to ocp_bus2 completes the connection of the OCP bridge to the master and slave modules.

3.2 Building the System

There are five pieces that must be brought together to build a SystemC executable for the integrated system.

- the bridge SystemC model generated from the BSV source
- the SystemC models for the master and slave
- the SystemC models for the OCP channels and adapters
- the SystemC library
- the Bluesim kernel and primitive libraries

The first step is to compile the BSV source for the bridge design into a simulation file:

```
> bsc -sim -u bridge.bsv
checking package dependencies
compiling ./OCP_IFC.bsv
compiling bridge.bsv
code generation for mkBridge starts
Elaborated Bluesim module file created: mkBridge.ba
All packages are up to date.
```

Next, the `mkBridge.ba` file created in the previous step is used to build the object files and headers for the SystemC model and the Bluespec rule schedule:

```
> bsc -systemc -e mkBridge mkBridge.ba
Bluesim object created: mkBridge.{h,o}
Bluesim object created: schedule.{h,o}
SystemC object created: mkBridge_systemc.{h,o}
```

Finally, the existing SystemC models for the master, slave and overall system can be compiled and linked with the generated object files, the SystemC libraries, and the Bluesim libraries:

```
> c++ -I$SYSTEMC/include -L$SYSTEMC/lib-linux \
      -I$BLUESPECDIR/Bluesim -L$BLUESPECDIR/Bluesim/$CXX_FAMILY \
      -I$OCP_CHANNEL/ocp -I$OCP_CHANNEL/ocp/tl1 -I$OCP_CHANNEL/generic \
      -I$OCP_ADAPTERS/include -I$OCP_ADAPTERS/include/tl0_tl1 -I$OCP_ADAPTERS/src/tl0_tl1 \
       -o bridge_test \
       bridge.cc master_inst.cc slave_inst.cc \
       mkBridge_systemc.o mkBridge.o schedule.o \
       -lsystemc -lbskernel -lbsprim
```

The environment variables used above provide paths to the various headers and libraries and will vary from system to system.

Explanation of environment variables

Variable	Description	Example
BLUESPECDIR	location of Bluespec tools	`/tools/bluespec/latest/bsc/inst/lib`
SYSTEMC	location of SystemC library installation	`/tools/systemc/systemc-2.1`
CXX_FAMILY	C++ compiler family (can be found using $BLUESPECDIR/bin/c++family)	`g++4`
OCP_CHANNEL	location of OCP channel package	`/tools/systemc/ocpchannel2.1.3`
OCP_ADAPTERS	location of OCP channel adapter package	`/tools/systemc/ocpadapters`