Bluespec Tutorial
Implementing a Software-Hardware Co-Execution with emVM

January 18, 2012

Contents

Table of Contents 2

1 Introduction 7

1.1 Purpose . . . 7
1.2 Tutorial overview . . . 7
1.3 Methodology . . . 8

2 Example 1: Simple factorial design 9

2.1 Factorial example overview . . . 9
2.2 Hardware side . . . 10
2.2.1 Overview . . . 10
2.2.2 DUT . . . 11
2.2.3 SceMiLayer . . . 13
2.2.4 Bridge layer . . . 15
2.3 Building the hardware side . . . 15
2.3.1 Build utility . . . 15
2.3.2 Build messages . . . 16
2.3.3 Compiler messages . . . 17
2.3.4 Infrastructure linkage . . . 17
2.4 Software side . . . 18
2.4.1 Overview . . . 18
2.4.2 Functional description . . . 19
2.4.3 Testbench . . . 19
2.5 Building the complete example . . . 20
2.5.1 Defining the `project.bld` file . . . 21
2.5.2 Running the build utility . . . 23
2.5.3 Simulating the example . . . 23

3 Example 2: Replacing hardware-side transactors 24

3.1 DUT . . . 24
3.2 SceMiLayer . . . 24
3.3 Bridge layer . . . 25
3.4 The software testbench . . . 25
3.5 Building and running the example . . . 26

4 Example 3: Generating data types for the C++ testbench 26

4.1 Modifying the Hardware Side . . . 26
4.2 Modifying the Software Side Testbench . . . 27
4.3 Build . . . 28

5 Example 4: Controlling the hardware-side clocks **28**
5.1 DUT . . . 29
5.2 SceMiLayer . . . 29
5.2.1 Defining a clock port . . . 29
5.2.2 SceMiClockConfiguration . . . 29
5.2.3 Instantiating the clock port . . . 30
5.2.4 Defining the clock controller . . . 31
5.2.5 Controlling the clock . . . 31
5.3 Build and simulate the example . . . 32
5.3.1 Compiler messages . . . 32
5.3.2 Viewing waveforms . . . 33

6 Example 5: Multiple clock ports **33**
6.1 Compiler messages . . . 34
6.2 Waveforms . . . 35

7 Example 6: Writing hardware transactors in BSV **35**
7.1 ClockedServerXactor . . . 35
7.1.1 Message Ports . . . 36
7.1.2 Clock Control . . . 37
7.1.3 Rules . . . 37
7.1.4 Interface definitions . . . 38
7.1.5 Implementing polymorphism . . . 38
7.2 SceMiLayer . . . 39
7.3 Results . . . 39

8 Building a GUI-controlled Software Testbench with Probes **40**
8.1 Example 7: Writing software-side transactors in C++ . . . 40
8.1.1 Header file . . . 41
8.1.2 Source code file . . . 41
8.1.3 Testbench . . . 43
8.1.4 Building the example . . . 44
8.2 Example 8: Writing a GUI-based Testbench . . . 45
8.2.1 Modify the Hardware Side - Instantiate Simulation Control . . . 46
8.2.2 Modify the Software Testbench . . . 46
8.2.3 Define GUI Environment . . . 48
8.2.4 Build . . . 49
8.2.5 Run Simulation from the GUI . . . 49

8.3 Example 9: Adding probes for debugging . . . 49
8.3.1 HDL Editor . . . 50
8.3.2 Invoking the HDL editor . . . 50
8.3.3 Defining Probes through the HDL editor . . . 50
8.3.4 Run the simulation . . . 51

9 Example 10: Wrapping a Verilog DUT 52
9.1 The Verilog file . . . 52
9.2 Writing the wrapper . . . 54
9.2.1 Define the interface . . . 54
9.2.2 Define the module . . . 54
9.2.3 Clocks and Resets . . . 55
9.2.4 Define methods: input and output ports . . . 55
9.2.5 Schedule . . . 56
9.2.6 BSV wrapper . . . 56
9.3 SceMiLayer . . . 57
9.4 Building the example . . . 57

10 Example 11: Using TLM transactors to integrate a Verilog AHB memory 57
10.1 Hardware side overview . . . 58
10.2 Writing the wrapper . . . 58
10.2.1 Define the interface . . . 58
10.2.2 Define the module . . . 58
10.2.3 Clocks and Resets . . . 59
10.2.4 Define methods: input and output ports . . . 59
10.2.5 Schedule . . . 60
10.3 SceMiLayer . . . 61
10.4 Building the hardware side . . . 63
10.5 The software testbench . . . 63
10.5.1 Data definitions . . . 63
10.5.2 SlaveXactor.h . . . 63
10.5.3 SlaveXactor.cpp . . . 64
10.5.4 TclTb.cpp . . . 65
10.5.5 GuiDut.tcl . . . 67
10.6 Build and run the example . . . 68

11 Example 12: Implementing a synthesizable testbench **69**

11.1 Hardware side overview . . . 69
11.2 Stimulus Generator . . . 70
11.3 Top Level mkTop . . . 71
11.4 SceMiLayer . . . 71
11.5 The software testbench . . . 71
11.6 Build and run the example . . . 72

A The source files **73**

A.1 Example 1: Factorial Example Files . . . 73
A.1.1 DUT . . . 73
A.1.2 SceMiLayer . . . 74
A.1.3 Bridge . . . 74
A.1.4 Parameters File (mkBridge.params) . . . 74
A.1.5 C++ Testbench . . . 75
A.1.6 Build File (project.bld) . . . 76
A.2 Example 2: Replacing hardware-side transactors . . . 77
A.2.1 SceMiLayer . . . 77
A.2.2 C++ Testbench . . . 77
A.3 Example 3: Generating data types . . . 78
A.3.1 DUT . . . 78
A.3.2 SceMiLayer . . . 79
A.3.3 C++ Testbench . . . 80
A.3.4 Build File (project.bld) . . . 81
A.4 Example 4: A Single Clock Port . . . 82
A.4.1 SceMiLayer - Single Clock Port . . . 82
A.5 Example 5: Multiple Clock Ports . . . 83
A.5.1 SceMiLayer - Two Clock Ports . . . 83
A.6 Example 6: Writing hardware-side Transactors . . . 84
A.6.1 ClockedServerXactor . . . 84
A.6.2 SceMiLayer - calling Transactor . . . 86
A.6.3 C++ Testbench . . . 86
A.7 Example 7: Writing Software Transactors in C++ . . . 87
A.7.1 FactXactor.h . . . 87
A.7.2 FactXactor.cpp . . . 88
A.7.3 Tb.cpp . . . 89
A.8 Example 8: Writing a GUI-based Testbench . . . 90

A.8.1 TclTb.cpp . . . 90
A.8.2 gui_dut.tcl . . . 97
A.8.3 project.bld . . . 99
A.9 Example 9: Adding Probes for Debugging . . . 100
A.9.1 project.bld . . . 100
A.10 Example 10: Wrapping a Verilog Dut . . . 101
A.10.1 FactorialServer.v . . . 101
A.10.2 mkFactorialServer.bsv . . . 105
A.10.3 SceMiLayer.bsv . . . 106
A.10.4 project.bld . . . 106
A.11 Example 11: Using TLM transactors to integrate a Verilog AHB Memory . . . 107
A.11.1 BSV wrapper . . . 107
A.11.2 SceMiLayer.bsv . . . 109
A.11.3 mkBridge.params . . . 110
A.11.4 SlaveXactor.h . . . 111
A.11.5 SlaveXactor.cpp . . . 112
A.12 Example 12: Implementing a synthesizable testbench . . . 113
A.12.1 Tb.bsv . . . 113
A.12.2 SceMiLayer.bsv . . . 115
A.12.3 TbXactor.cpp . . . 115
A.12.4 TbXactor.h . . . 116
A.12.5 project.bld . . . 117

1 Introduction

This tutorial aims to provide a complete example for implementing a emVM system with Bluespec tools and components. You'll see how easy it is to set up and get running with a simple factorial example. We'll then add features and complexity and explore the full functionality of the environment.

1.1 Purpose

This document is a tutorial on how to implement a emVM style emulation environment with Bluespec, Bluespec-provided components, and 3rd party tools. It is expected that you have some experience or training in Bluespec SystemVerilog. The following references may be helpful as you move through this tutorial:

- *Bluespec SystemVerilog Reference Guide* for a complete reference on the Bluespec SystemVerilog language and library packages
- *Bluespec SystemVerilog User Guide* for the mechanics of using the Bluespec tools, including the Bluespec Development Workstation
- *emVM User Guide* as a reference on emVM and Bluespec's implementation of SCE-MI
- The complete SCE-MI standard is available from Accellera's website `www.eda.org/itc`.

1.2 Tutorial overview

There are two connected environments in an emVM system: a hardware side containing a device under test (DUT) and a software side containing a testbench. The primary example throughout this tutorial is a simple DUT calculating factorials. The DUT has a single input and a single output; it receives a value, calculates its factorial, and returns the result. In these examples the hardware side is written in BSV while the software side is written in C++. The two sides communicate through a link with a SCE-MI based protocol for messages. Both sides contain components for generating and receiving messages.

While emVM systems often include an emulation board on the hardware side, the examples in this tutorial use simulation (either Bluesim or Verilog) and communicate over a TCP link. This environment should be available to all Bluespec users and is the simplest in which to demonstrate and learn the tools and techniques available. These examples can easily be ported to any supported emulation board. This tutorial does not currently discuss porting the examples onto an emulation board.

After implementing the simple factorial example, the remaining sections refine and add features to both the hardware and software sides. Many of the sections are independent tutorials; use them to build a custom tutorial based on your areas of interest.

- Example 1: Simple factorial design (Section 2)
- Example 2: Replacing hardware-side transactors (Section 3)
- Example 3: Generating data types for the C++ testbench (Section 4): using `generateSceMiHeaders`
- Controlling the hardware-side clocks (Section 5)
  - Example 4: A single clock port (Section 5.2.1)
  - Example 5: Multiple clock ports (Section 6)

- Example 6: Writing hardware-side transactors in BSV (Section 7)
- Running the emulation from a GUI-based testbench (Section 8)

  The following three sections describe the steps required to build a fully-functional GUI-controlled software testbench including probes. These features can be implemented independently, but together allow you to build a complete GUI environment.

  - Example 7: Writing software-side transactors (Section 8.1)
  - Example 8: Creating a GUI Testbench (Section 8.2)
  - Example 9: Adding probes for debugging (Section 8.3)
- Example 10: Wrapping a Verilog DUT (Section 9)
- Example 11: Using TLM transactors to integrate a Verilog AHB memory (Section 10)
- Example 12: Implementing a synthesizable testbench (Section 11)

1.3 Methodology

An emVM system supports software-hardware co-execution by connecting an un-timed software testbench (the "SW" side) to a cycle-accurate device (the "HW" side) through an abstraction bridge. The HW side is usually an emulation platform, but may also be a simulation of the HW. The abstraction bridge can be separated into a collection of individual transactors which convert un-timed messages (or transactions) into a series of cycle-accurate events, or conversely, composes a series of clocked events into a single un-timed message. Each transactor handles a separate part of the interaction. The transactors are implementation-independent, promoting deployment flexibility and portability.

The hardware side hosts a design-under-test (the "DUT") and a number of transactors which handle communication and control clocking. When sending and receiving messages, transactors have the ability to freeze controlled time long enough for message decomposition and composition operations to complete, before transferring data to or from the DUT.

A transactor is comprised of a small number of message port and clock control primitives as defined by the SCE-MI standard. Each message port in a transactor on the emulation host is paired with a port proxy in the testbench on the software side. The emVM infrastructure abstracts away the details of the communication channel linking the testbench to the emulation platform. Insertion of the communication channel logic into the design is performed by an infrastructure linkage tool (the "ILT"). Bluespec supports both vendor-supplied and Bluespec linkage tools, supporting custom scenarios.

When using an emVM system the end-user is responsible for writing the un-timed testbench and the DUT hardware description. For many applications, the end-user simply instantiates Bluespec-provided transactors to connect the DUT with the software testbench and to provide clocking and control of the DUT, as described in Section 2. For more complex systems, the user may need to write their own transactors or modify existing transactors, as described in Section 7.

Bluespec provides an implementation of the SCE-MI standard, as well as higher-level transactors to help the end-user implement SCE-MI in their environment. Bluespec's SCE-MI library supports modes of infrastructure linkage that use 3rd-party SCE-MI platforms as well as BSV implementations for platforms without native SCE-MI support.

The following components comprise an emVM implementation with Bluespec SystemVerilog, as shown in Figure 1:

- Hardware Side

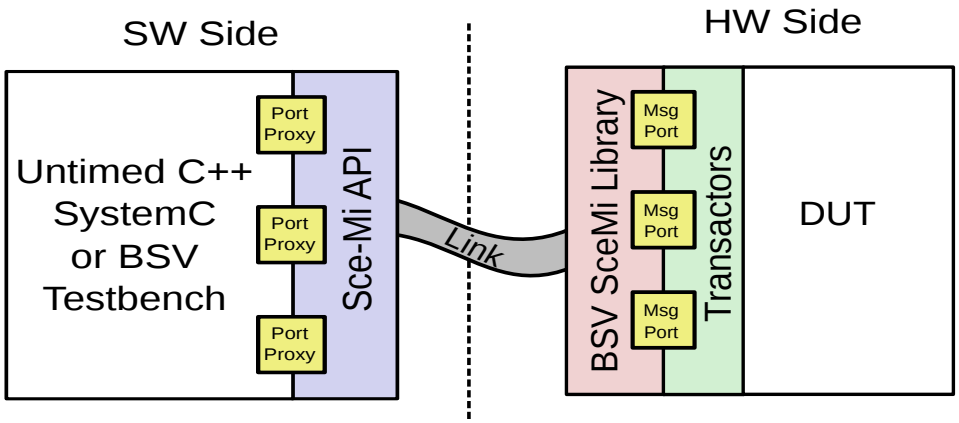

Figure 1: SCE-MI provides a standard and portable mechanism for connecting an un-timed testbench with a DUT running on an emulation host platform.

  - Design under test (DUT)
    The DUT is user written and can be in any hardware description language including BSV.
  - Transactors connecting DUT and hardware message ports
    The transactors can be written in any hardware description language. Bluespec provides a library of transactors that can be combined with user-written transactors.
  - BSV SCE-MI Library
    Bluespec provides a complete implementation of the SCE-MI library.

- Software Side
  - Un-Timed Testbench
    The testbench is user written and can be in C++ or any language, including BSV.
  - Software transactors connecting testbench and software message port proxies
    Bluespec provides software transactors in both C++ and BSV.

- Linkage Tool
  - The linkage tool is specific to the emulation platform. Bluespec provides an infrastructure linkage tool for multiple platforms. You can also use other vendor-specific infrastructure linkage tools for specific emulation environments.

This tutorial uses BSV for the DUT and hardware transactor modules, C++ for the software testbench and transactors, and the Bluespec infrastructure linkage tool.

2 Example 1: Simple factorial design

2.1 Factorial example overview

To show how easy it is to get an‘ emVM system up and running, we'll work through a very simple example of a DUT with a single input and a single output. The DUT is a factorial generator; it receives a value from the software testbench, calculates the factorial, and returns the calculated value to the testbench. The DUT provides a `Server` interface, defined in the library package `ClientServer`. This could be just about any DUT receiving a value and generating a result.

We'll connect the DUT to emVM-provided transactors and have a C++ program send data and then receive the result and display it. This first example uses the TCP link type; in simulation, we'll be communicating over a TCP/IP socket. Both the HW simulation and the SW testbench run as separate processes using VPI calls to communicate between processes. This example does not use an emulation board; we're simulating the design in Bluesim. You could also simulate with a Verilog simulator.

2.2 Hardware side

2.2.1 Overview

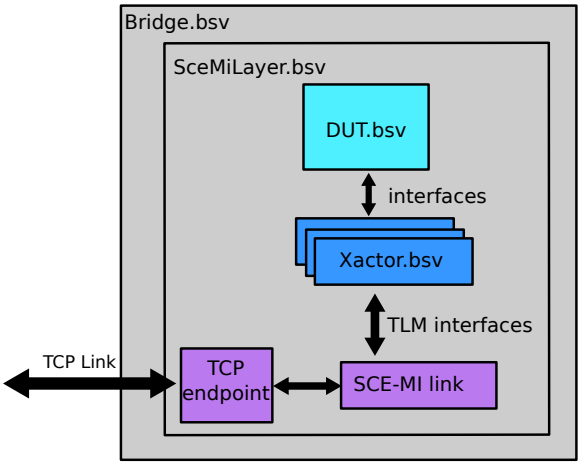

Figure 2: Hardware side components

The hardware side of any emVM system has three basic components, as shown in Figure 2. These components can reside in the same package, or could be made up of many packages; for simplicity the factorial example presented here is divided into the following three files:

- **DUT.bsv**: the design under test (DUT)
- **SceMiLayer.bsv**: the platform-independent layer providing the connection between the DUT and the bridge layer
- **Bridge.bsv**: The platform-dependent layer, which in its simplest form is design-independent

At the heart of the hardware side is the design you're testing (DUT), a typical BSV design. There are no particular restrictions or requirements on the DUT definition, and it can include Verilog and other RTL components. No special code must be added to the DUT to use it within an emVM environment. The only requirement is the explicit declaration of the module type `[Module]` in the module declaration statement.

The second component is the SceMiLayer, a platform independent layer composed of hardware transactors and the DUT instantiation. The transactors handle communication to and from the DUT and connect the DUT to the message ports. The transactors also control the clocking of the DUT, bridging the un-timed transactions on the SW side with the cycle-accurate execution of the DUT on the hardware side. Transactor modules are of type `SceMiModule`, allowing the accumulation of SCE-MI related items in addition to the state elements and rules accumulated in an ordinary module (type `Module`). The transactors are independent of the emulation host platform

and the hardware configuration. Bluespec provides a library of hardware transactors which can be used as is, modified, or supplemented by user-written transactors.

The third component is the bridge layer. The bridge layer appends the FPGA-specific elements to the design. This is the top module of the project and is where the board specifics are taken into consideration. The coordination and communication logic is implemented in this layer. Bridge files for all supported emulation and simulation systems are included in the emVM product.

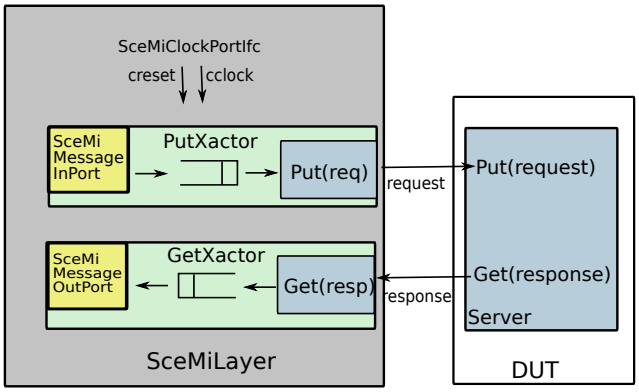

Figure 3: Factorial Example using Get and Put Xactors

2.2.2 DUT

The DUT we are using in this example is a factorial generator. The interface provided by the DUT module, `mkFactorialSource`, is `Server` interface, which has a `Put` subinterfaces named `request` and a `Get` subinterface named `response`. The `Put` interface, `request`, receives a value from the testbench. The module calculates the factorial, and sends the result back through the `Get` interface, `response`.

The `Server` interface is defined in the `ClientServer` package, which is imported into the DUT module.

```
interface Server#(type req_type, type resp_type);
    interface Put#(req_type)  request;
    interface Get#(resp_type) response;
endinterface: Server
```

The DUT does not change when changes are made to the emVM environment, for example to use a different communication channel or emulation platform. No special code is added to the DUT to use it within an emVM environment. The complete source code can be found in Appendix A.1.1.

```
package DUT;

import GetPut::*;
import FIFO::*;
import ClientServer::*;
```

When defining the DUT module for use in an emVM design, you must specify the module type, which typically is `[Module]`. The `mkFactorialServer` module provides the `Server` interface described above.

```
(* synthesize *)
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64) ));
```

The `mkFactorialServer` module definition uses FIFOs to store the values moving in and out of the module. There are also registers to store intermediate results and a flag.

```
   FIFO#(UInt#(64)) req_fifo <- mkFIFO();
   FIFO#(UInt#(64)) resp_fifo <- mkFIFO();

   Reg#(Bool) started <- mkReg(False);
   Reg#(UInt#(64)) next_count <- mkReg(0);
   Reg#(UInt#(64)) curr_result <- mkRegU();
```

There are three rules in the DUT: `start_computation`, `do_computation`, and `finish_computation`.

```
   rule start_computation (!started);
      // Get the value
      let val = req_fifo.first();
      req_fifo.deq();
      // Set up the computation state
      curr_result <= 1;
      next_count <= val;
      started <= True;
      $display("Starting request: %h", val);
   endrule

   rule do_computation (started && (next_count > 0));
      let next_result = curr_result * next_count;
      curr_result <= next_result;
      // handle overflow
      if (next_result == 0)  begin
         $display("Overflow!");
         next_count <= 0;
      end
      else begin
         next_count <= next_count - 1;
      end
   endrule

   rule finish_computation (started && (next_count == 0));
      started <= False;
      resp_fifo.enq(curr_result);
      $display("Sending result: %h", curr_result);
   endrule
```

The package ends with the subinterface definitions.

```
   interface Put request = toPut(req_fifo);
   interface Get response = toGet(resp_fifo);
```

2.2.3 SceMiLayer

The `mkSceMiLayer` module contains the implementation-independent components, the transactors and clocks. In this package the DUT is instantiated and connected to the hardware transactors.

SceMiModule

The default BSV module type is `Module`, but you can define other BSV module types to accumulate items in addition to state elements and rules. The `SceMi` package defines the module type `SceMiModule` to accommodate the infrastructure supporting clock ports, clock controllers, and message ports on the hardware side.[1]

The DUT does not have anything emVM-specific about it. It is a synthesizable BSV or RTL design of the default module type `Module`. The SceMiLayer on the other hand, contains the transactors and other emVM constructs. They are of the module type `SceMiModule` to support the additional infrastructure required by emVM.

To define a module to be a SCE-MI module, add `[SceMiModule]` between the keyword `module` and the name of the module in the module definition statement:

```
module [SceMiModule] mkModule (....);
```

mkSceMiLayer

A design may contain a hierarchy of modules, but ultimately they need to be instantiated by one top module. The `mkSceMiLayer` module is the top module of type `SceMiModule`.

The `mkSceMiLayer` module is in the `SceMiLayer` package. The packages `SceMi`, `DefaultValue`, and for this design, `GetPut` and `ClientServer`, must be imported in addition to the `DUT`.

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;

import DUT::*;
```

We define a type synonym for the Server interface:

```
typedef Server#(UInt#(64), UInt#(64)) ServerIfc;
```

The module definition includes the module type, `SceMiModule`.

```
module [SceMiModule] mkSceMiLayer ();
```

[1] An ordinary Bluespec module, when instantiated, adds its own state elements and rules to the growing accumulation of state elements and rules defined in the design. In a SCE-MI-based design, additional SCE-MI related items must be accumulated as well. It is desirable to keep these additional design details separate from the main design, keeping the natural structure of the DUT intact.

There are two types of clocks in an emVM system: the uncontrolled clock and one or more controlled clocks. The uncontrolled clock is a free-running clock used for interacting with the hardware ports and clock controllers. The DUT runs on a controlled clock which can be stopped by the user through any transactor during operations.

Every system must instantiate an instance of the `mkSceMiClockPort` module, defining a clock port for the DUT. The emVM system includes a default clock configuration that you can use to define the clock port parameters. You can also change individual parameters within the default configuration to match the controlled clock for your system.

In this example we're using the default clock configuration as defined. The DUT is in only one clock domain, so we'll instantiate a single clock port based on the default configuration:

```
    SceMiClockConfiguration conf = defaultValue;
    SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);
```

The `buildDut` module connects the DUT with the instantiated clockport, `clk_port`. An instantiation of this module is required within the `SceMiLayer` module. The instantiated module provides the interface type of the DUT.

```
    ServerIfc dut <- buildDut(mkFactorialServer, clk_port);
```

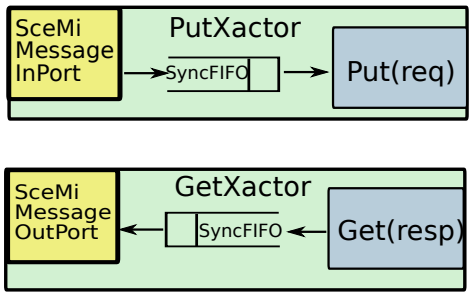

Figure 4: SceMiLayer Transactors

The `mkGetXactor` and `mkPutXactor` transactors, shown in figure 4, connect the interfaces from the DUT to the hardware message ports. The `Get` interface connects to the `GetXactor` and the `Put` interface connects to the `Put` transactor.

```
    Empty dutout <- mkGetXactor(dut.get_result, clk_port);
    Empty dutin  <- mkPutXactor(dut.put_request, clk_port);
```

In Section 3 we will replace the two transactors with the `mkServerXactor`.

The `mkShutdownXactor` is instantiated for shutting down the simulation gracefully, in tandem with the software side shutdown.

```
    Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

2.2.4 Bridge layer

The `mkBridge` module is the top module of the project and is where the platform-independent SceMiLayer is transformed into the platform-dependent implementation. The `mkBridge` module implements the hardware-side coordination logic and instantiates the infrastructure required for the emulation board or target environment. The complexity of the bridge layer depends on the link type, where TCP (used in this example) is the simplest implementation. With different link types, the `mkBridge` module can become much more complex but most of the complexities are handled automatically by the Bluespec library.

The `mkBridge` module is the top-level module which connects the hardware side to the physical link. It handles the communication between the hardware and software sides. As the I/O mechanism varies, this top-level module incorporates the changes required to support and best utilize the different platforms and emulator features. In our simple TCP example, the `mkBridge` module consists of a single line, instantiating the `buildSceMi` module, returning an empty interface. When implementing on an FPGA platform, it will be more complex as it handles the connections to the FPGA pins.

```
package Bridge;

import SceMi::*;
import SceMiLayer::*;

(* synthesize *)
module mkBridge ();
   Empty scemi <- buildSceMi(mkSceMiLayer, TCP);
endmodule
endpackage
```

Because the bridge layer is implementation-specific, the file will change with the target enviroment. Bluespec provides a set of `Bridge.bsv` files to implement the top-level module for TCP and supported emulation boards.

2.3 Building the hardware side

The hardware side of our first example is now complete; the next step is to compile the design and check for errors. The top file of the design is `Bridge.bsv` and the top module is named `mkBridge`.

While compilation at this stage is the same as compiling any other BSV design, Bluespec provides a utility called `build` to facilitate all aspects of building a project, including compilation, simulation and emulation. In this tutorial we're going to use the utility for all build activities.

2.3.1 Build utility

The build flow for a given design has many variations, depending on the link type, the emulation hardware, and the project configuration. Bluespec provides the `build` utility to facilitate setting up and building a project. The utility codifes the rules for building Bluespec projects in a single place and eliminates most of the boilerplate involved in setting up a project. With this utility, you don't have to remember all of the specific commands and dependencies to build a given project. The build utility is not emVM specific, but is especially useful when building emVM projects because of the added configuration options and variations.

The build utility runs from a project configuration (`.bld`) file. Within this file you describe the specifics of the project including directories, files, module names, bsc options, and emulation platforms. The file is composed of a set of build targets, where each target describes a component of the project (DUT, testbench, etc.) along with directives describing the options and parameters of the target. There can be multiple definitions or versions for a particular target. For example you could have two targets describing how to build the hardware DUT, `vlog_dut` to generate code for a Verilog simulator and `bsim_dut` to generate code for Bluesim. When running the `build` utility you would select the appropriate target for your selected simulator.

When the `build` command is entered along with one or more targets, the utility determines the stages necessary to build the targets. For each target, the utility determines the correct commands and the sequence of commands required to execute each stage. The goal is that sensible defaults are used and any deviations from the defaults are only specified once per project. Complete reference documentation for the build utility can be found in the appendix of the Bluespec emVM guide.

The default project description file is `project.bld`, but you can specify a different file to be used in its place.

At this point, we'll use the `build` utility to compile the hardware testbench. Later in the tutorial will discuss how to create and modify a `project.bld` file.

The target to compile the hardware side for Bluesim is `bsim_dut`. To compile the design, type at a unix command prompt:

```
build bsim_dut
```

If you'd prefer to compile for a Verilog simulator, type:

```
build vlog_dut
```

2.3.2 Build messages

The `build` utility output displays the commands executed. For this example you should see the following messages:

```
Building target bsim_dut...
>>> Entering stage delete_build_dirs.
*** exec: rm -rf build build
<<< Exiting stage delete_build_dirs.
>>> Entering stage make_build_dirs.
*** exec: mkdir -p build
*** exec: mkdir -p build
<<< Exiting stage make_build_dirs.
>>> Entering stage compile_for_bluesim.
*** exec: bsc -sim -simdir build -bdir build -p +:/raid/home/kczeck/Builds/Bluespec-2010.06.beta1/li
<<< Exiting stage compile_for_bluesim.
>>> Entering stage generate_scemi_parameters.
*** exec: scemilink --sim --simdir=build --path=build:+ --port=7381 mkBridge
<<< Exiting stage generate_scemi_parameters.
>>> Entering stage link_for_bluesim.
*** exec: bsc -sim -simdir build -bdir build -scemi -o bsim_dut -e mkBridge
<<< Exiting stage link_for_bluesim.
Done building target bsim_dut.
```

As you can see from the messages above, once the compile step was complete, the infrastructure linkage tool `scemilink`, described in Section 2.3.4 was run.

2.3.3 Compiler messages

Compilation at this stage is the same as compiling any other BSV design. The SCE-MI components are included by importing the package `Scemi.bsv`. Unique to SCE-MI based designs are the compiler messages describing the clocks in the design as determined by the `SceMiClockConfiguration` parameters. The compiler messages are found in the file `bsim_dut_compile_for_bluesim.log`.

The first set of messages describe all the clocks found in the design. In our factorial design there is a single controlled clock:

```
(SCE-MI) Default clock group
(SCE-MI) Clock #0
  Numerator: 1
  Denominator: 1
  DutyHi: 0
  DutyLo: 100
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 1 controllers for clock #0
```

The next set of messages describe the physical clock generation attributes:

```
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 208, column 16:
(SCE-MI) Common clock time scale = 1
(SCE-MI)    Group    Clock   Period     Rise     Fall     Freq
(SCE-MI)  default        0        1         uclock      50.00 MHz
(SCE-MI) Note: clock frequencies assume 100 MHz reference clock
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 459, column 47:
(SCE-MI) Max reset count = 16
```

We'll look at defining and controlling clocks in more detail in Section 5.

2.3.4 Infrastructure linkage

Once the hardware side is complete, the next step is *infrastructure linkage*. Infrastructure linkage takes the description of the hardware side and generates a text file (the parameters file) describing the clocks, transactors, message ports, and physical link information. This file is read by the software side during initialization to bind the software side message port proxies to the hardware side message ports. The infrastructure linkage tool may generate other files in addition to the parameters file, as required by the target environment.

Bluespec provides a tool *scemilink* to perform infrastructure linkage for TCP environments in addition to Bluespec-supported emulation boards. When we ran the `build` utility to compile the hardware side, it also ran `scemilink` and generated the parameters file.

The scemilink tool generates the parameters file used to connect the message port proxies on the software side to the message ports on the hardware side. The name of the parameters file is the top module name, followed by a `.params`. For our factorial example, the parameters file generated is named `mkBridge.params`. When writing the C++ testbench for the software side we'll refer to the `.params` file for the names of the transactors, portnames, and data types. You can view the parameters file to verify port names and data types, but you should never edit the parameters file.

This section of the `mkBridge.params` file describes the message port transactors:

```
MessageOutPort 0 TransactorName ""
MessageOutPort 0 PortName       "scemi_dutout_outport"
MessageOutPort 0 PortWidth      64
MessageOutPort 0 ChannelId      5
MessageOutPort 0 Type           "UInt#(64)"

MessageInPort 0 TransactorName ""
MessageInPort 0 PortName       "scemi_dutin_inport"
MessageInPort 0 PortWidth      64
MessageInPort 0 ChannelId      2
MessageInPort 0 Type           "UInt#(64)"
```

From the file you can see that the name for the message outport is `scemi_dutout_outport` and the name for the message inport is `scemi_dutin_inport`. You will use these names when writing the C++ testbench.

The complete parameters file generated for the factorial example can be found in Appendix A.1.4.

For Verilog simulation using a TCP link, the scemilink step generates a Verilog file that contains the VPI calls for the TCP link. This file is called `scemilink.vlog_fragment` and is placed in the directory specified by `vdir` with the other Verilog files. The infrastructure linkage tool will generate other files as necessary for the selected emulation target.

2.4 Software side

2.4.1 Overview

In this example, the software testbench is written in C++ using Bluespec-provided proxies, transactors and utilities to create the testbench. The Bluespec C++ library components implement the software macros, while providing a higher-level interface in which much of the low-level processing is handled for the testbench developer. The testbench could also be implemented in BSV or any other language.

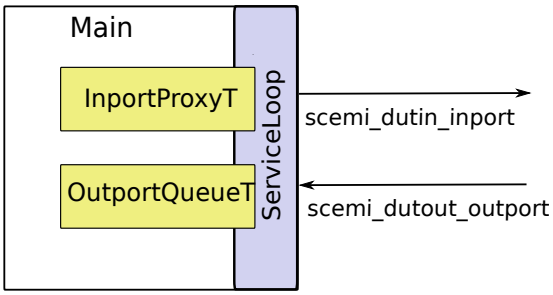

Figure 5: Overview of SW Side

The testbench in our example has two message port proxies, an inport and an outport. On the testbench side the names of the message ports are based on the direction of the data from the *hardware* side perspective, as shown in Figure 5. The inport sends data out of the testbench - into the hardware DUT; the outport receives data into the testbench - out of the hardware DUT.

2.4.2 Functional description

The testbench defines two message prort proxies based on the Bluespec-provided C++ classes: `InportProxyT` to send values and `OutportQueueT` to receive the calculated factorials. These are both templated classes; the methods provided can be used with different data types. In our example, the data type for both the output and input values are the C++ equivalents, defined in the `BSVType` class, of the `UInt#(64)` data types in the hardware design.

The testbench sends the values 0 to 9 to the hardware side and receives back the calculated factorials. Both the values for the requested factorial and calculated factorial received are sent to the standard output device.

2.4.3 Testbench

The software side consists of a single C++ file, which we're naming `Tb.cpp`.

The file `bsv_scemi.h` contains the Bluespec C++ implementation of the software side for SCE-MI. We need to include this file and use the `std` namespace.

```
using namespace std;
#include "bsv_scemi.h"
```

The following line is required as described in the SCE-MI specification:

```
    int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
```

The `mkBridge.params` file was generated when we ran the `build` utility by `scemilink`, as described in Section 2.3.4. The file contains the specifications of the message ports, clocks, and link parameters (in this case TCP address and TCP port). The software testbench reads this file to connect up the message ports.

```
    SceMiParameters *params = new SceMiParameters( "mkBridge.params" )
    SceMi *sceMi = SceMi::Init( sceMiVersion, params );
```

Each message port on the hardware side has a corresponding port proxy on the software side. Bluespec provides a large library of C++ proxies and transactors for developing testbenches in SystemC or C++ utilizing the standard un-timed SCE-MI testbench API.

The C++ library contains two C++ inport proxies to communicate with the `SceMiMessageInPort`: `InportProxyT` and `InportQueueT`, and two C++ outport proxies to communicate with the `SceMiMessageOutPort`: `OutportProxyT` and `OutportQueueT`. The `Queue` versions of the proxies add a `WaitQueueT` to the proxy, where the process receiving from the queue can block until data is available. The C++ library classes are templated classes allowing the methods provided to be used with different data types.

In our testbench, we initialize a `InportProxyT` to communicate with the hardware-side `PutXactor` and a `OutportQueueT` to communicate with the hardware-side `GetXactor`. We are using the `OutportQueueT` because it contains a `getMessage` method. This method blocks if there is no message to receive. Full documentation for all the transactors and port proxies can be found in the *Bluespec emVM* manual.

The names of the transactors used in initializing the proxies are found in the `mkBridge.params` file.

```
  // Initialize the SceMi inport
  InportProxyT<BitT<64> > in_proxy ("", "scemi_dutin_inport", sceMi);

  // Initialize the SceMi outport
  OutportQueueT<BitT<64> > out_proxy ("", "scemi_dutout_outport", sceMi);
```

The `SceMiServiceThread` starts and runs the SCE-MI service thread.

```
   // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);
```

The testbench sends 10 messages containing the 10 input values, displays the value sent, and then displays the 10 calculated values received back from the hardware side.

```
  for (SceMiU32 i=0; i<10; i=i+1) {
    in_proxy.sendMessage(i);
    cout << "Requested factorial: " << i << endl;
  }
  for (SceMiU32 i=0; i<10; i=i+1) {
    BitT<64> fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
  }
```

When completed, a finish is sent to the hardware side, the SCE-MI service thread is stopped, and SCE-MI is shutdown.

```
  // Shutdown the simulation
  shutdown.blocking_send_finish();

  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);
  delete params;
}
```

The source code for the complete example can be found in Appendix A.1.

2.5 Building the complete example

Once the software testbench and hardware models are written, use the `build` utility to prepare the example for simulation. Figure 6 illustrates the complete tool flow for an emVM example using BSV on the hardware side and a C++ testbench on the software side.

For this example, the `build` utility executes the following steps to prepare the models for simulation:

1. Compile BSV DUT and transactors, generating RTL description of the hardware side.
2. Run **scemilink**, the infrastructure linkage tool to analyze the hardware description and generate a parameters file which is loaded by the software testbench.

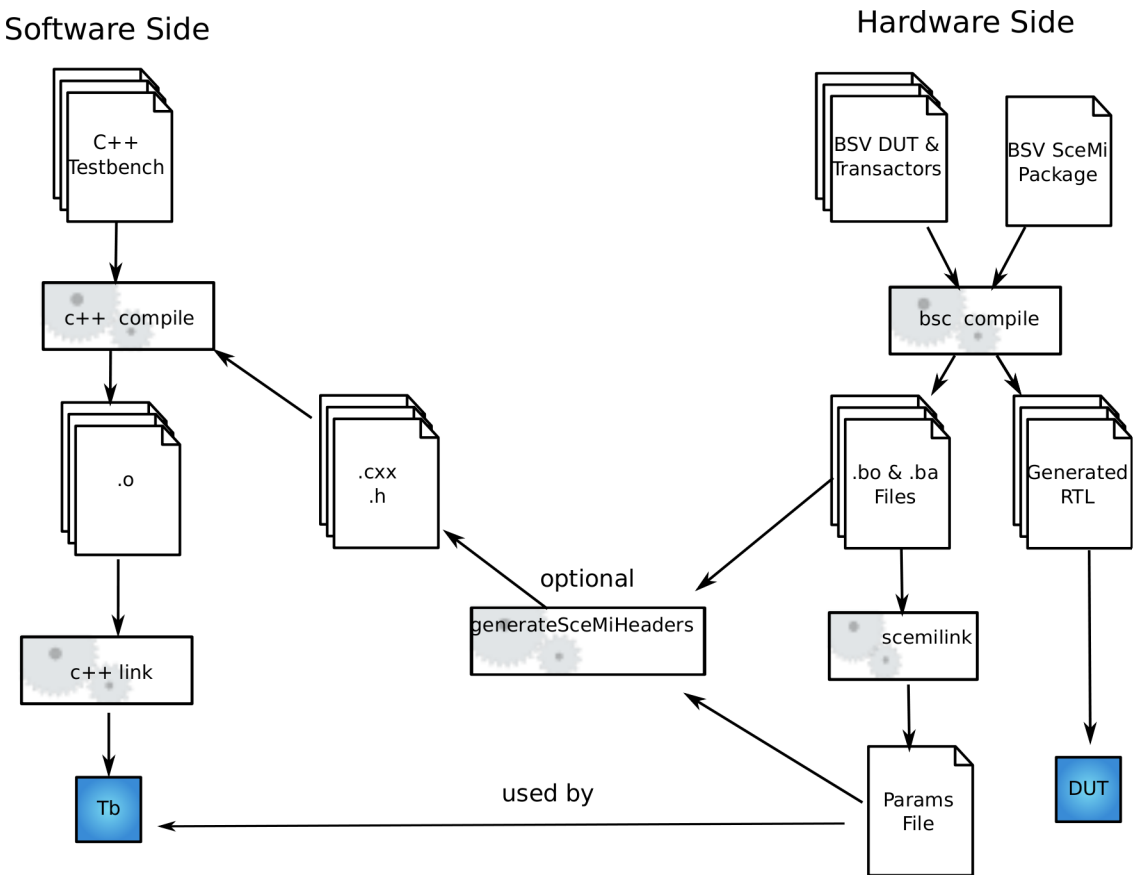

Figure 6: Tool Flow for TCP Example

3. Compile the C++ software testbench.
4. Link the the testbench with the software side of the emVM infrastructure and the parameters file to generate an executable program.

The optional step, `generateSceMiHeaders`, generates C++ classes that correspond to BSV data types used in the hardware model, including structs and enums. As there are no advanced types in this example, the tcl script is not used. Section 4 provides an example using this script.

2.5.1 Defining the `project.bld` file

We discussed the `build` utility and the `project.bld` file briefly in Section 2.3.1. In this section we'll discuss the composition of the `project.bld` file in more detail.

Let's examine the build file for our factorial example. The complete file can be found in Appendix A.1.6. The default file name is `project.bld`. As discussed earlier, the file is composed of a set of build targets, where each target describes a component of the project. The first line of each section is a section header describing the target, with the target name enclosed in brackets. Except for the reserved name `DEFAULT`, you can use any name for a section header. Meaningful, descriptive names are recommended.

[**DEFAULT**] The `[DEFAULT]` section contains the directives which apply to all targets. The `default-targets` directive describes which targets are built by default, if a target is not provided on the command line, In this case the default targes are for the DUT compiled for a Bluesim simulator, and the C++ testbench.

```
[DEFAULT]
default-targets:   bsim_dut tb
```

[**dut**] This section provides common directives used when building the DUT, whether for Bluesim or a Verilog simulator. You can define targets which are only referenced by other targets, but never used on their own through the `hide-target` directive. The referencing target will contain an `extends-target` directive which includes the hidden directives as part of its own. Defining a hidden target minimizes repetition of common information, allowing you to define the directives once, and include them in multiple targets. In this example, the `dut` target is referenced by both the Bluesim (`bsim_dut`) and Verilog (`vlog_dut`) build targets.

If not specified, the top module is derived from the name of the top file, adding `mk` in front of the top file name. In this case, the top module will default to `mkBridge`, which is the name of our top module.

```
[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build
```

[**bsim_dut**] This section defines the remaining parameters to build the DUT for Bluesim. It includes all the directives in the `[dut]` section, as indicated by the `extends-target` directive.

```
[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut
```

[**vlog_dut**] This section also includes the directives from the `[dut]` section along with the remaining directives to build for a Verilog simulator.

```
[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       vlog_dut
```

[tb] The directives in this section build the C++ testbench. The SCE-MI testbench is indicated by including the directive `scemi-tb` in the target building the testbench. There are no parameters required for the `scemi-tb` directive.

```
[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp
exe-file: tb
```

[clean] Every build file should have a `[clean]` section. The build utility will determine when the target should be run.

```
[clean]
run-shell: rm -rf build
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log
run-shell: rm -f vlog_dut directc_* vlog_dut*.log
run-shell: rm -f tb *.params tb*.log
```

2.5.2 Running the build utility

The command to build the project with the build utility is:

```
build [options] [target]
```

For our example, let's build the application and simulate with Bluesim. We need to build both the DUT and the testbench:

```
build bsim_dut tb
```

where `bsim_dut` is the name of the DUT in Bluesim target, and `tb` is the name of the testbench target. Since these are the default targets, we could also just type:

```
build
```

To see all commands and messages generated as the project is built, use the verbose option (`--verbose` or `-v`). To display the commands that would be executed in each stage, without executing them, use the `--dry-run` flag.

2.5.3 Simulating the example

After the build is complete, the final step in our example is to simulate the design. In this example we aren't using an emulation system; we'll be simulating the DUT with Bluesim, communicating with the C++ testbench.

The TCP environment requires 2 separate processes: the DUT running on Bluesim and the testbench running in C++. The DUT has to be started first.

To simulate the example:

1. Start the DUT. The executable generated from our build file is `bsim_dut`, as specified by the `exe-file` directive in the `bsim_dut` section.

```
./bsim_dut
```

2. In a new shell, from the same directory, start the testbench. The testbench executable name is specified by the `exe-file` directive in the `tb` section of the build file.

```
./tb
```

In the shell running the DUT, you will see the output from the display statements in the DUT. In the shell running the testbench, you will see the display statements generated by the software testbench.

3 Example 2: Replacing hardware-side transactors

The DUT in Example 1 provided a `Server` interface which communicated with a `PutXactor` and a `GetXactor` in the SceMiLayer. Another way to model this example would be to use a single transactor in the SceMiLayer to create the input and output message ports, instead of using a transactor for each message port. Bluespec provides a `ServerXactor` which connects to the `Server` interface provided by the DUT. This section modifies the previous example to use the `ServerXactor` in the SceMiLayer. The complete source code for this section can be found in Appendix A.2.

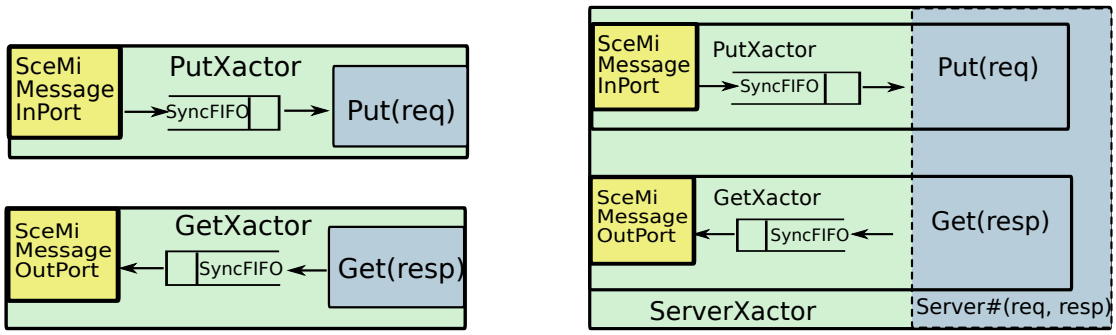

Figure 7: Put and Get Transactors vs. Server Transactor

Figure 7 shows the `PutXactor` and `GetXactor` used in the Example 1 on the left with the `ServerXactor` in this example on the right. The `ServerXactor` combines the `PutXactor` and the `GetXactor` into a single transactor.

3.1 DUT

This example uses the same DUT as used in Example 1.

3.2 SceMiLayer

In the SceMiLayer, a single transactor, the `ServerXactor` replaces the two transactors. The `ServerXactor` still generates two message ports: a `SceMiMessageInPort` and a `SceMiMessageOutport`.

The server transactor contains both the inport and outport message ports.

```
    ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

    Empty dutconnect <- mkServerXactor(dut, clk_port);
```

3.3 Bridge layer

There are no changes required in the `Bridge` package.

3.4 The software testbench

The software testbench has the same two port proxies as in Example 1. The inport proxy, `InportProxyT`, sends values and the outport proxy, `OutportQueueT`, receives the calculated values. This doesn't change, even though the hardware side has a single transactor; it still creates two message ports to connect to the port proxies. What does change, however, are the names of the message ports.

The names of the message ports are found in the file `mkBridge.params`, generated when the `build` utility is run to build the DUT. Build the DUT, by running:

```
build bsim_dut
```

Open the newly generated `mkBridge.params` file to find the new message port names. Use these names to update the software testbench. Never edit the `.params` file.

The hardware transactor in the SceMiLayer is named `dutconnect`:

```
   Empty dutconnect <- mkServerXactor(dut, clk_port);
```

The following excerpt from the `mkBridge.params` file shows the corresponding message port names:

```
MessageOutPort 1 PortName       "scemi_dutconnect_resp_outport"
MessageInPort 1 PortName       "scemi_dutconnect_req_inport"
```

Modify the file `Tb.cpp`, and change the message port names in the port proxy definition statements:

```
// Initialize the SceMi inport
InportProxyT<BitT<64> > in_proxy ("", "scemi_dutconnect_req_inport", sceMi);

// Initialize the SceMi outport
OutportQueueT<BitT<64> > out_proxy ("", "scemi_dutconnect_resp_outport", sceMi)
```

This is the only change you need to make in the C++ testbench.

3.5 Building and running the example

Follow the same steps as before to build and simulate the example:

1. Build the entire example:

```
build bsim_dut tb
```

2. Start the DUT

```
./bsim_dut
```

3. In a new shell, start the testbench

```
./tb
```

The design generates the same output as Example 1.

4 Example 3: Generating data types for the C++ testbench

BSV provides a strong, static type-checking environment, allowing the definition of meaningful data types. When writing a co-emulation system, you have to ensure that the data types on the software side are compatible with the data types on the hardware side. Bluespec provides a set of C++ classes, based on the base class `BSVType`, defining C++ classes for Bluespec data types. [2] When you define more complex data types, such as enums and structs, on the hardware side, a corresponding C++ class must also be defined. The Bluespec tcl script, `generateSceMiHeaders.tcl` generates C++ classes and files to define more complex data types to correspond with the data types defined in the BSV hardware design.

In this section, we continue to modify the factorial example to include complex data types to demonstrate the `generateSceMiHeaders` utility.

The DUT in this example is the factorial DUT from the first and second examples, providing a `Server` interface, but returning a different data type. This example performs the same tasks, but instead of returning the factorial as a `UInt#(64)`, both both the input value and the calculated value are returned. And the calculated value is returned as a `Maybe` data type. A `Maybe` includes a `valid` bit, indicating whether the returned type is valid or not. For example, if when calculating the return value there is an overflow, the returned value would not be valid.

4.1 Modifying the Hardware Side

First, we're going to define a new dataype named `Factresp`, which is a structure made up of the value in, and the factorial returned as a `Maybe`. The `Maybe` type is a standard Bluespec defined type which encapsulates any data type with a valid bit, and provides functions for testing the validity and extracting the data. The data part of the `Maybe`, containing the calculated factorial, is defined in this example as a `UInt#(64)`.

```
typedef struct {UInt#(64) value_in;
                Maybe#(UInt#(64)) factorial;
               } Factresp deriving (Bits);
```

[2] Documentation of the `BSVType` classes can be found in the *emVM user manual*.

In the `Server` interface the type of the `Get` is the structure `Factresp` instead of the `UInt#(64)`.

```
module [Module] mkFactorialServer (Server#(UInt#(64), Factresp));
```

There are other changes within the DUT, to take into account the new data types. The complete code can be found in Appendix A.3. No changes are required to either the `SceMiLayer` or `Bridge` packages.

4.2 Modifying the Software Side Testbench

The only change we've made to the interface of the factorial example is replacing the data type in the `Get` subinterface from `UInt#(64)` to the structure we've defined named `Factresp`. On the software testbench side we now have to replace the old data type with our new data type. This occurs in two places: the definition of the outport proxy `OutportQueueT`, and the `getMessage` statement.

```
// Initialize the SceMi outport
OutportQueueT<Factresp > out_proxy ("", "scemi_dutconnect_resp_outport", sceMi);
```

```
    Factresp fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
```

What's still missing is the C++ definition of the data type `Factresp`. The `generateSceMiHeaders` utility creates the C++ header and code files for the new data type. When run, the utility generates the file `SceMiHeaders.h` and all other files needed to define any new BSV data types in C++. The file `SceMiHeaders.h` must be included in the testbench file.

```
// Automaticlly generated by: ::SceMiMsg
// DO NOT EDIT
// C++ Class with SceMi Message passing for Bluespec type:  All SceMi types
// Generated on: Tue Mar 23 11:13:51 EDT 2010
// Bluespec version: 2010.02.beta4 2010-02-26 19729

#pragma once

#include "Maybe_UInt_64.h"
#include "Factresp.h"
```

Two new data types are created:

- `Factresp` corresponding to the structure of the same name
- `Maybe_UInt_64` corresponding to the BSV type `Maybe#(UInt#(64))`

4.3 Build

The `build` utility will create the `SceMiHeaders.h` file and other C++ files necessary as long as the build file includes the relevant directives. The directives related to the C++ testbench, including the `generateSceMiHeaders` utility, are in the target containing the `build-for:c++` directive; the `[tb]` target in our example.

- The `c++-header-targets` directive must have a value other than `none`. The valid values for this directive are `inputs, outputs, probes, all, none`. The default value is `all`, so if the directive is not included, the utility will be run. If `none` is selected, the `generatedSceMiHeaders` utility is not run. To generate the C++ data types, the `outputs` (or `all`) value must be specified.
- The `.h` files are written to the directory specified by the `c++-header-directory` directive. It defaults to the value in the `c++-source-directory` directive, which defaults to the current working directory.
- The `generateSceMiHeaders` utility uses the `top-file` and `top-module` parameters. These directives must be either specified directly in the testbench section or through the `extends-target` directive. In our example, we've defined these in the `[dut]` section, which we extend into the `[tb]` section, as shown below.

```
[dut]
hide-target
extends-target: paths
top-file:       Bridge.bsv
top-module:     mkBridge
scemi-tcp-port: 7500

[tb]
extends-target: dut
scemi-tb
build-for: c++
c++-files: Tb.cpp
exe-file: tb
c++-header-targets: outputs
```

The complete `project.bld` file can be found in Appendix A.3.4.

Once the build is complete, the design is simulated as in the earlier examples.

5 Example 4: Controlling the hardware-side clocks

The factorial designs we've been examining have all instantiated a single clock port which was never stopped. One of the implications of converting between un-timed message bit streams on the SW side and clocked events on the HW side is that the DUT clock may need to be stopped so that the testbench can catch up. The transactor stops the clock long enough for operations to occur, freezing time on the HW side when the SW side is called. This section examines the hardware clocks: defining clock ports, clock controllers, and multiple clock ports in an emVM design.

All HW transactors are clocked by a free running uncontrolled clock generated internally by the infrastructure. A controlled clock is a clock that can be disabled by any transactor during operations,

freezing time long enough to complete an operation before resuming clocking of the DUT. When a controlled clock is stopped, the DUT is *frozen in controlled time*. Most applications have at least one controlled clock and may have multiple controlled clocks.

The methodology assumes a free-running uncontrolled clock which is the fastest clock in the system, and allows the creation of additional controlled clocks which allow hardware transactors to precisely control the clocking of the DUT.

5.1 DUT

In this section we'll continue to use the factorial DUT we've used in the previous examples.

5.2 SceMiLayer

In this simple example we'll instantiate the transactor to the DUT and define the clocks and clock controller in the `SceMiLayer` package. We'll use the same `ServerXactor` we used in the Example 2, so the DUT connections are the same as the previous example.

```
// Instantiate the dut
ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

// Connect the dut to SceMi
Empty dutconnect <- mkServerXactor(dut, clk_port);
```

In this section we're going to focus on the statements used to define the clock port and clock controller.

5.2.1 Defining a clock port

A clock port is created by instantiating the `mkSceMiClockPort` module The parameters of the clock are defined by the `SceMiClockConfiguration` structure. Every SCE-MI design must have at least one clock port, clock number 0, which must be an undivided clock. There may be multiple clock ports in a design.

5.2.2 SceMiClockConfiguration

The `SceMiClockConfiguration` structure defines the waveform parameters for the clock including waveform frequency, duty cycle, and phase shift. It also specifies the number of cycles to hold reset for the clock domain.

Bluespec defines an instance of the `DefaultValue` class for the `SceMiClockConfiguration` structure. The default instance defines a single fast controlled clock with an active posedge, a don't care negedge, and no phase shift, suitable for many single-domain posedge-flop based designs. Note that since we are using an instance of `DefaultValue`, the `DefaultValue` package must be imported.

To define the clock configuration, define a variable of type `SceMiClockConfiguration` and set it equal to `defaultValue`. In the earlier examples we used the default clock configuation to define the clock port:

```
SceMiClockConfiguration conf = defaultValue;
SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);
```

To modify the default configuration, you simply redefine the members of the structure which differ from the defined default. Note: The clock with `clockNum = 0` must exist, and must be an undivided clock.

```
SceMiClockConfiguration conf = defaultValue;
conf.clockNum = 0;
conf.dutyHi = 50;
conf.dutyLo = 50;
```

The following tables lists the complete `SceMiClockConfiguration` structure. You can choose to modify any of these fields as necessary.

SceMiClockConfiguration Structure			
Field	Type	Description	Allowed or Recommended Values
`clockNum`	`Integer`	Unique clock identification number	a small integer > 0
`clockGroup`	`Integer`	Unique clock group assignment	`noClockGroup` or a small integer > 0
`ratioNumerator`	`Integer`	Clock period numerator	an integer > 0
`ratioDenominator`	`Integer`	Clock period denominator	an integer $\leq$ `ratioNumerator`
`dutyHi`	`Integer`	Duty cycle high-phase length	0 means don't-care
`dutyLo`	`Integer`	Duty cycle low-phase length	0 means don't-care
`phase`	`Integer`	Clock waveform phase shift amount	$0 \leq phase < dutyHi + dutyLo$
`resetCycles`	`Integer`	Minimum number of cycles of reset	an integer > 0

5.2.3 Instantiating the clock port

To instantiate a clock port:

```
SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);
```

where the argument `conf` is the `SceMiClockConfiguration` defined above.

The `mkSceMiClockPort` module provides the interface `SceMiClockPortIfc` consisting of two subinterfaces, a Clock and a Reset.

```
interface SceMiClockPortIfc;
   interface Clock cclock;
   interface Reset creset;
endinterface: SceMiClockPortIfc
```

The clock defined above, `clk_port`, contains a Clock named `clk_port.cclock` and a Reset named `clk_port.creset`.

5.2.4 Defining the clock controller

We just created a clockport named `clk_port`. To be able to control or stop the clock, we need to instantiate a clock controller. For each clock port, zero or more clock controller (`mkSceMiClockControl`), instances are permitted. The clock controller allows edge-by-edge control of the advancement of the clock.

To create a clock controller named `clkcntrl`, instantiate the `mkSceMiClockControl` module with the parameters `clockNum`, `allow_posedge`, and `allow_negedge`. The association of a clock controller and a clock port is achieved through the unique clock identification number, `clockNum`, from the clock port instantiation. The other two parameters are boolean values which control the advancement of the clock. In this case, we are using the same value, `allow_clock`, for both the parameters. You can control the edges independently by setting them to different variables. The `allow_posedge` and `allow_negedge` parameters must be expressions over registered values with no implicit conditions.

```
Bool allow_clock = True;
SceMiClockControlIfc clkcntrl <- mkSceMiClockControl(conf.clockNum,
                                                     allow_clock,
                                                     allow_clock);
```

The `mkSceMiClockControl` module provides the `SceMiClockControlIfc` interface which provides:

- the uncontrolled clock and reset of the transactor (`uclock, ureset`)
- methods which notify when a controlled clock edge is imminent (`pre_posedge`, `pre_negedge`)

```
interface SceMiClockControlIfc;
   interface Clock uclock;
   interface Reset ureset;
   (* always_ready *)
   method Bool pre_posedge();
   (* always_ready *)
   method Bool pre_negedge();
endinterface: SceMiClockControlIfc
```

The interface methods are in the uncontrolled clock domain (`uclock`).

5.2.5 Controlling the clock

Once the clock controller is defined for a particular clock port, the boolean values `allow_posedge` and `allow_negedge` can be used to control the advancement of the clock. In the instantiation above, we set the values of `allow_posedge` and `allow_negedge` to `allow_clock` which is defined as True. This creates a free-running controlled clock, that is a clock that isn't stopped. If the design is not going to stop the clock, it is not necessary to define a clock controller. If a clock controller is defined for the clock, the clock will only advance when the values of `allow_posedge` and `allow_negedge` are True.

A more useful application of a clock controller is one in which the clock is stalled as required by the design. For example, you might want the controlled clock to run only when there is data available from the testbench. We'll look at this example in Section 7. For now, let's look at stopping the clock based on the clock cycle count.

First, we instantiate a register to hold the count of the uncontrolled clock, `uclock_count`. We'll also instantiate a Boolean register named `clock_enable` that we set to `False` when we want to stop the clock. By setting `allow_clock` to the value of `clock_enable`, we can use it to stop the clock.

```
Reg#(UInt#(16)) uclock_count <- mkReg(0, clocked_by uclock, reset_by ureset);
Reg#(Bool) clock_enable <-  mkReg(True, clocked_by uclock, reset_by ureset);
Bool allow_clock = clock_enable ;

SceMiClockControlIfc clkcntrl_fast <- mkSceMiClockControl(conf.clockNum,
                                                          allow_clock,
                                                          allow_clock);
rule control_clock;
   uclock_count <= uclock_count + 1;
   if (uclock_count == 10 || uclock_count == 15 ||
       uclock_count == 80  || uclock_count == 92)
      clock_enable <= !clock_enable;
endrule
```

5.3 Build and simulate the example

- Build the example

  `build`

- Start simulating the DUT in Bluesim

  `./bsim_dut &`

- Start the testbench

  `./tb`

5.3.1 Compiler messages

While compiling the example, the BSV compiler (`bsc`) emits messages describing the clocks.

```
(SCE-MI) Begin clock generation
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 138, column 51:
(SCE-MI) Default clock group
(SCE-MI) Clock #0
  Numerator: 1
  Denominator: 1
  DutyHi: 50
  DutyLo: 50
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 2 controllers for clock #0
```

The next set of messages from the compiler describes the physical clock generation attributes.

```
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 208, column 16:
(SCE-MI) Common clock time scale = 2
(SCE-MI)    Group    Clock   Period     Rise     Fall     Freq
(SCE-MI)  default        0        2        0        1   25.00 MHz
(SCE-MI) Note: clock frequencies assume 100 MHz reference clock
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 459, column 47:
(SCE-MI) Max reset count = 32
```

5.3.2 Viewing waveforms

Figure 8 shows the clock signals. The controlled clock, `cclock`, stops, while the uncontrolled clock continues.

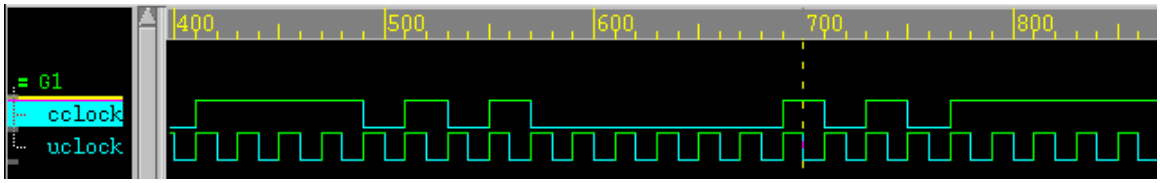

Figure 8: Waveforms with stopped controlled clock

6 Example 5: Multiple clock ports

In this next example we'll create a second controlled clock port.

In the `SceMiLayer` package we'll define two different clocks, `clkport_fast` and `clkport_slow`. For each clock we've defined the `SceMiClockConfiguration` using the `defaultValue` instance, and then modified it for the specific clock.

```
// Fast clock (same as uclock and don't care duty cycle)
SceMiClockConfiguration conf_fast = defaultValue();
conf_fast.clockNum = 0;
conf_fast.dutyHi = 50;
conf_fast.dutyLo = 50;
SceMiClockPortIfc clkport_fast <- mkSceMiClockPort(conf_fast);

//Slow clock (divide by 4 and 50% duty cycle)
SceMiClockConfiguration conf_slow = defaultValue();
conf_slow.clockNum = 1;
conf_slow.ratioNumerator = 4;
conf_slow.dutyHi = 50;
conf_slow.dutyLo = 50;
SceMiClockPortIfc clkport_slow <- mkSceMiClockPort(conf_slow);
```

The fast and slow clocks are shown in Figure 9:

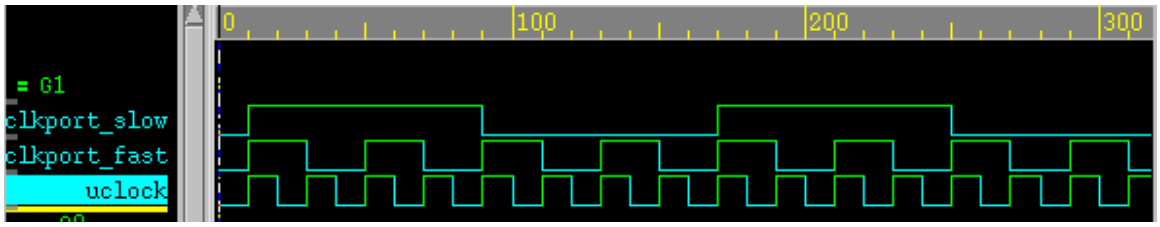

Figure 9: Waveforms of fast and slow clocks

Each clock has a clock controller defined for it. The `allow_clockfast` parameters are always True, while the `allow_clockslow` parameters are the value of `clock_enable`:

```
    Bool allow_clockfast = True ;
    Bool allow_clockslow = clock_enable ;

    // Instantiate a controller for the controlled clock
    // so that we can stall the clock while the port is busy
    SceMiClockControlIfc clkcntrl_fast <- mkSceMiClockControl(
                                                conf_fast.clockNum,
                                                allow_clockfast,
                                                allow_clockfast);

    SceMiClockControlIfc clkcntrl_slow <- mkSceMiClockControl(
                                                conf_slow.clockNum,
                                                allow_clockslow,
                                                allow_clockslow);
```

The slow clock is stopped at intervals defined by the rule `control_clock`. When one controlled clock is stopped, all controlled clocks in the system are stopped.

```
    rule control_clock;
       uclock_count <= uclock_count + 1;
       if (uclock_count == 2 || uclock_count == 5 ||
           uclock_count == 10 || uclock_count == 15 ||
           uclock_count == 20  || uclock_count == 25)
          clock_enable <= !clock_enable;
    endrule
```

6.1 Compiler messages

When you build the example, you'll see the following compiler messages describing the two clocks.

```
(SCE-MI) Default clock group
(SCE-MI) Clock #0
  Numerator: 1
  Denominator: 1
  DutyHi: 50
  DutyLo: 50
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 2 controllers for clock #0
(SCE-MI) Clock #1
  Numerator: 4
  Denominator: 1
  DutyHi: 50
  DutyLo: 50
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 1 controllers for clock #1
```

The next set of messages describes the physical clock generation attributes.

```
SCE-MI) Common clock time scale = 2
(SCE-MI)    Group    Clock   Period     Rise     Fall     Freq
(SCE-MI)  default        0        2        0        1   25.00 MHz
(SCE-MI)  default        1        8        0        4   6.250 MHz
(SCE-MI) Note: clock frequencies assume 100 MHz reference clock
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 459, column 47:
(SCE-MI) Max reset count = 128
```

6.2 Waveforms

An emVM system may include multiple clocks, at different frequencies, with multiple transactors controlling them. The SCE-MI standard specifies that when any controlled clock is stopped, all controlled clocks in the system must stop. Figure 10 shows that when the slow clock is stopped, the fast clock is also stopped.

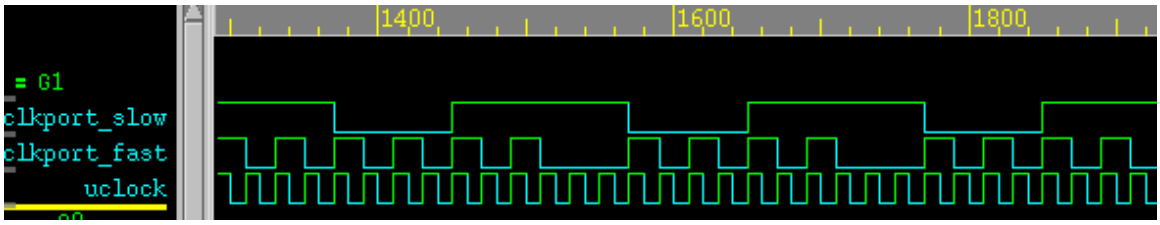

Figure 10: Stopped slow clock stops fast clock

7 Example 6: Writing hardware transactors in BSV

In our first examples we used several different BSV transactors on the hardware side: the `GetXactor` and `PutXactor` in Example 1, which were then replaced with the `ServerXactor` in Example 2. These transactors are all provided by Bluespec in the `BSVSource/SceMi` library.

You can also write your own transactors to implement different functionality and facilitate reuse. The transactors provided in the Bluespec library assume a free-running controlled clock; they do not allow the testbench to stop the controlled clock. For our next example, we're going to write a BSV transactor called `ClockedServerXactor`, a transactor which stalls the clock and provides a `Client` interface, to connect with a DUT providing a `Server` interface. The design using the `ClockedServerXactor` is shown in Figure 11.

7.1 ClockedServerXactor

The `ClockedServerXactor` transactor provides a `Client#(UInt#(64), UInt#(64))` interface, to connect a `Server#(UInt#(64), UInt#(64)` interface from the DUT. It is similar to the `ServerXactor` used in Example 2, but instantiates a clock controller (`mkSceMiClockControl`) to allow the testbench to stall the controlled clock. In this example, the clock is stopped whenever there are no requests to be processed.

The transactor module takes as arguments the clock configuration parameter and the clockport interface. The controlled clock is instantiated in the `SceMiLayer` package. The transactor uses the clock number of the controlled clock to instantiate the clock controller as well as the clock and reset signals of the controlled clock. The `SceMiClockConfiguration` provides the clock number and `SceMiClockPortIfc` provides the controlled clock and reset.

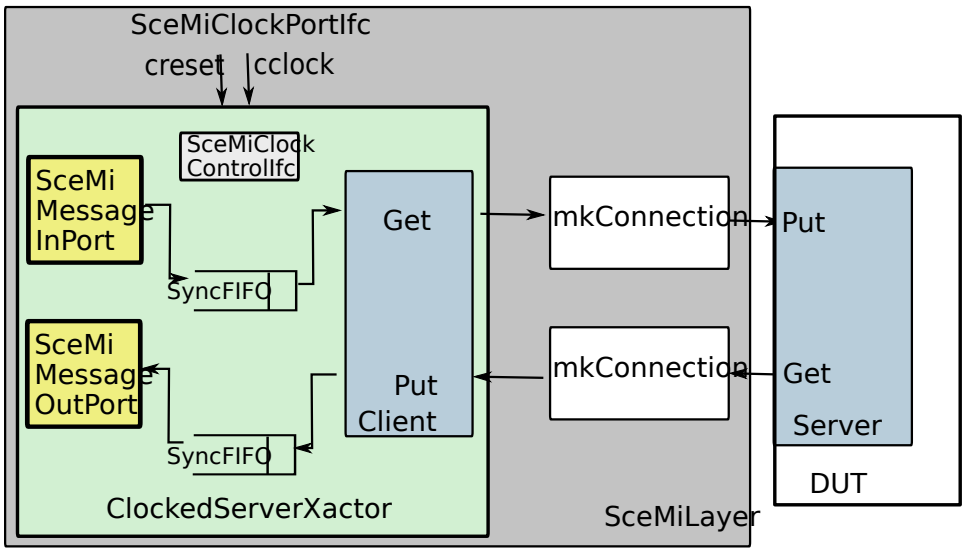

Figure 11: Example with ClockedServerXactor

```
module [SceMiModule] mkClockedServerXactor
                        #( parameter SceMiClockConfiguration cclk_conf,
                           SceMiClockPortIfc clkport)
                         ( Client#(UInt#(64), UInt#(64));
```

We need to access both the controlled clock, passed in from the `SceMiLayer`, and the uncontrolled clock, to instantiate the SyncFIFOs:

```
   // Access the controlled clock and reset
   Clock cclock = clkport.cclock;
   Reset creset = clkport.creset;

   // Access the uncontrolled clock and reset
   Clock uclock <- sceMiGetUClock;
   Reset ureset <- sceMiGetUReset;
```

7.1.1 Message Ports

The hardware transactors communicate with the hardware side of the message channnel through the message ports. When writing your own transactors you instantiate the message ports directly: `inport` for data coming in from the testbench, and `outport` for the data moving out to the testbench.

```
   // The input port
   SceMiMessageInPortIfc#(UInt#(64)) inport <- mkSceMiMessageInPort();

   // The output port
   SceMiMessageOutPortIfc#(UInt#(64)) outport <- mkSceMiMessageOutPort();
```

SyncFIFOs are used to cross data between the uncontrolled and controlled clock domains. The inbound data moves from the uncontrolled to the controlled domain, the outbound from the controlled to the uncontrolled domain. The DUT runs in the controlled clock domain.

```
// Use a SyncFIFO to cross the data into the controlled domain
SyncFIFOIfc#(UInt#(64)) res_fifo <- mkSyncFIFO(2, uclock, ureset, cclock);

// Use the SyncFIFO to cross the data into the uncontrolled domain, to
SyncFIFOIfc#(UInt#(64)) out_fifo <- mkSyncFIFO(2, cclock, creset, uclock);
```

7.1.2 Clock Control

We instantiate a clock controller, `clk_cntrl`, for the controlled clock. The transactor is defined such that the controlled clock only runs when there are outstanding requests. Since the transactor handles both requests coming in from the testbench and results sent back to the testbench, the controlled clock starts once the first request is received, and continues until the last response is sent. We determine this time period by instantiating two PulseWires: `started_req` for when a request is received and and `finished_req` for when the response is sent. The register `outstanding_reqs` is incremented each time a request is received and decremented each time a response is sent. Therefore, whenever `outstanding_reqs != 0`, the controlled clock runs.

```
PulseWire started_req <- mkPulseWire(clocked_by uclock, reset_by ureset);
PulseWire finished_req <- mkPulseWire(clocked_by uclock, reset_by ureset);
Reg#(UInt#(16)) outstanding_reqs <- mkReg(0, clocked_by uclock,
                                          reset_by ureset);

Bool allow_clock = (outstanding_reqs != 0);

SceMiClockControlIfc clk_cntrl <- mkSceMiClockControl(
                        cclk_conf.clockNum, allow_clock, allow_clock);
```

7.1.3 Rules

The rule `count_reqs` increments and decrements the value of `outstanding_reqs`:

```
 rule count_reqs;
   if (started_req && !finished_req)
      outstanding_reqs <= outstanding_reqs + 1;
   else if (finished_req && !started_req)
      outstanding_reqs <= outstanding_reqs - 1;
endrule: count_reqs
```

The rules `request`, `req_from_tb`, and `resp_to_tb` control moving the data between the SceMiMessagePorts and the SyncFIFOs.

```
rule request;
   inport.request();
endrule

rule req_from_tb;
   let x <- toGet(inport).get;
   res_fifo.enq(x);
   started_req.send();
   $display("count in: %d", outstanding_reqs);
endrule: req_from_tb

rule resp_to_tb;
   outport.send(out_fifo.first());
   out_fifo.deq();
   $display("count out: %d", outstanding_reqs);
   finished_req.send();
endrule: resp_to_tb
```

7.1.4 Interface definitions

The value from the inport, stored in the `res_fifo`, is provided as a `Get` interface. The value going to the outport, stored in the out_result fifo, is provided as a `Put` interface. The SceMiLayer connects the `Client` interface from the transactor with the `Server` interface of the DUT.

```
interface Get request = toGet(res_fifo);
interface Put response = toPut(out_fifo);
```

7.1.5 Implementing polymorphism

Ideally a transactor will be reused in multiple design. In this section we'll modify the transactor so that instead of always using `UInt#(64)` for the data types of the input and output message data, we'll use the polymorphic data types `req_ty` and `resp_ty`. The specific data types for the application will then be specified when the transactor is instantiated in the SceMiLayer.

The definition of the `Client` specified that the data type of both the incoming and outgoing messages was `UInt#(64)`:

```
Client#(UInt#(64), UInt#(64))
```

Instead, we're going to define a polymorphic interface, in which the request and response datatypes can be different:

```
Client#(req_ty, resp_ty)
```

Provisos must be added to the module declaration statement to ensure that the actual data types are in the `Bits` class:

```
module [SceMiModule] mkClockedServerXactor
                        #( parameter SceMiClockConfiguration cclk_conf,
                           SceMiClockPortIfc clkport)
                         ( Client#(req_ty, resp_ty))
   provisos (Bits#(req_ty, req_ty_sz),
             Bits#(resp_ty, resp_ty_sz)) ;
```

The only other change is that the message port instantiations and SyncFIFO instantiations have to be changed to use the appropriate data type (either `req_ty` for the inport or `resp_ty` for the outport) instead of `UInt#(64)`.

The complete code for the `ClockedGetPutXactor` package is found in Appendix A.6.1.

7.2 SceMiLayer

The `SceMiLayer` package is virtually unchanged from the original example.

We have to import the `Connectable` package and our new transactor, in addition to the packages imported in the earlier example:

```
import Connectable::*;

import ClockedServerXactor::*;
```

The transactor instantiation is modified to use the new transactor. At the beginning of the file, we define the data type for the interface along with the `ServerIfc`:

```
typedef Server#(UInt#(64), UInt#(64)) ServerIfc;
typedef Client#(UInt#(64), UInt#(64)) ClientIfc;
```

We instantiate our new transactor, and then connect the DUT with the transactor:

```
    // Instantiate the SCE-MI transactor
    ClientIfc xactor <- mkClockedServerXactor(conf, clk_port);

    // Connect the DUT to the transactor
    mkConnection(dut.request, xactor.request);
    mkConnection(dut.response, xactor.response);
```

The complete code for the `SceMiLayer` package can be found in Appendix A.6.2.

7.3 Results

The `Build.bsv`, `DUT.bsv` and `Tb.cpp` files are not changed at all from the previous examples.

The following waves are from the factorial example, with data in and data out. You can see the controlled clock stops when no data is available.

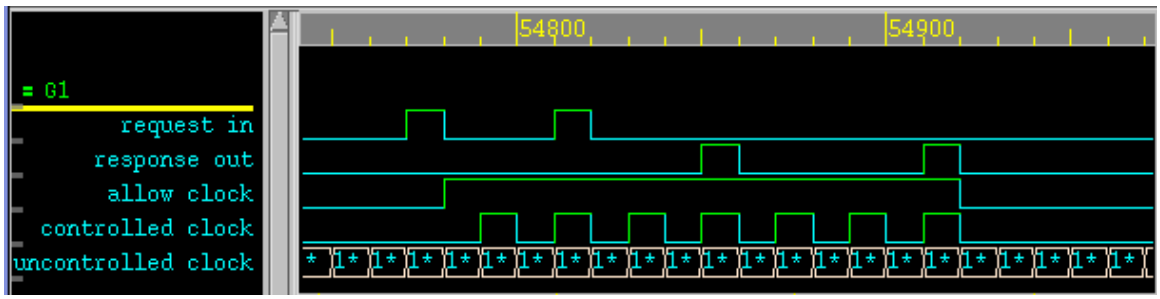

Figure 12: Waveforms with controlled and uncontrolled clock

8 Building a GUI-controlled Software Testbench with Probes

The next sections describe how to build a fully-functional GUI-based software testbench. These techniques can be used when simulating or emulating the design. We are going to continue using the factorial DUT used in the earlier examples. In the next three sections we will build the components comprising the testbench, shown in Figure 13:

- A software transactor encapsulating the inputs and outputs to the hardware side (Section 8.1)
- A GUI front end to the software testbench (Section 8.2)
- Probes into the design (Section 8.3)

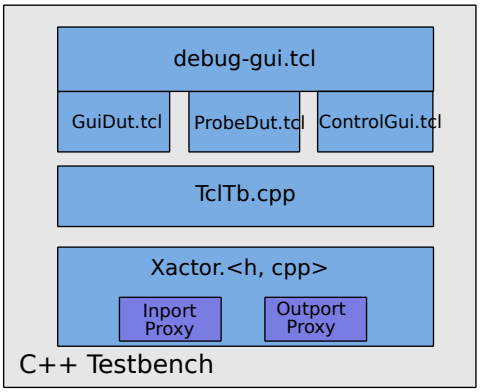

Figure 13: Host-side C++ Testbench

8.1 Example 7: Writing software-side transactors in C++

In Section 2.4, we wrote a software testbench in C++, consisting of a single C++ file, Tb.cpp. The file contained two transactors: `InportProxyT` to send the values from the testbench to the HW side, and `OutportQueueT` to receive the values back from the HW side. Both transactors are provided as part of emVM in the `$BLUESPECDIR/SceMi` directory.

In this section, we're going to write a custom transactor in C++ named `FactXactor` that encapsulates the design-specific ports in our testbench to facilitate adding a GUI front-end to our testbench. The transactor is defined in two files: the header file `FactXactor.h` and the source code file `FactXactor.cpp`.

8.1.1 Header file

Let's look at the header file `FactXactor.h`. The complete transactor definition can be found in Appendix A.7. First, we have to include Bluespec's SCE-MI C++ api:

```
#include "bsv_scemi.h"
```

The class name is the same as the transactor name, `FactXactor`. Within the header file three transactors are defined: the data in (`m_datain`), the data out (`m_dataout`), and a shutdown transactor (`m_shutdown`). `m_datain` is defined as the emVM-provided inport transactor, `InportProxyT`, with a data type of `BitT<64>`. `m_dataout` is defined as the emVM-provided outport transactor, `OutportQueueT`, also with a data type of `BitT<64>`. `m_shutdown` uses the emVM-provided shutdown transactor.

```
class FactXactor {
 protected:

  // Data Xactors
  InportProxyT<BitT<64> >       m_datain;
  OutportQueueT<BitT<64 > >     m_dataout;
  // Shutdown Xactor
  ShutdownXactor                m_shutdown;

 public:
  // Constructor
  FactXactor (SceMi *scemi) ;
  // Destructor
  ~ FactXactor();
```

The input is defined as an input into the hardware-side DUT: it *puts* a request into the DUT. The output *gets* a response from the DUT. Our transactor has methods for blocking and non-blocking versions of `putRequest` and `getResponse`. The methods are declared in the header file, with the full definition in the source code file, `FactXactor.cpp`.

```
  // Public interface
  bool putRequestNB(long);
  bool putRequestB (long);
  bool getResponseNB (long &);
  bool getResponseB (long &);
};
```

8.1.2 Source code file

The source code is found in `FactXactor.cpp`.

The constructor for the transactor contains the software side port proxies which communicate with the hardware side message ports. The names used in the constructor definition must match the message port names in the SceMi parameters file, `mkBridge.params`.

```
FactXactor::FactXactor(SceMi *scemi)
  : m_datain  ("", "scemi_dutin_inport", scemi)
  , m_dataout ("", "scemi_dutout_outport", scemi)
  , m_shutdown ("", "scemi_shutdown", scemi)
{
```

The `shutdown` method is defined:

```
void FactXactor::shutdown()
{
  m_shutdown.blocking_send_finish();
}
```

Since `m_datain` is an instance of the `InportProxyT`, it inherits the methods from the proxy. The method `putReqestNB` is an instance of the `sendMessageNonBlocking` method, returning True when the data is sent.

```
bool FactXactor::putRequestNB (long request)
{
  BitT<64> reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}
```

The blocking put request blocks until the message is sent and should not be called from a SCE-MI call back as deadlock will occur.

```
bool FactXactor::putRequestB (long request)
{
  BitT<64> reqbit(request);
  m_datain.sendMessage(reqbit);
  return true;
}
```

The get responses are instances of the `OutportQueueT` class. The non-blocking get response returns True and populates `resp` if a response is available.

```
bool FactXactor::getResponseNB (long &resp)
{
  BitT<64> respbit;
  bool gotone = m_dataout.getMessageNonBlocking (respbit) ;
  if (gotone) {
    resp = respbit.get64();
  }
  return gotone;
}
```

The blocking get response blocks if there is no message to receive. The `getMessage` method should not be called from a SCE-MI call back as deadlock will occur. The blocking method is recommended; it will only return a value if a value has been received. If you use the non-blocking method, the testbench may report values which have no meaning.

```
bool FactXactor::getResponseB (long &resp)
{
  BitT<64> respbit = m_dataout.getMessage () ;
  resp = respbit.get64();
  return true;
}
```

8.1.3 Testbench

The file `Tb.cpp` contains the testbench code, using the `FactXactor` transactor instead of explicitly using the `InportProxyT` and the `OutportQueueT` port proxies.

First, the testbench includes the file `FactXactor.h`, containing the factorial transactor definition. The initialization is the same as in our other examples:

```
#include "FactXactor.h"

// --------------------

void runFactorialTest (class FactXactor &xactor);

int main (int argc, char *argv[]) {

  // Initialize SceMi
  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );
```

The testbench creates the `FactXactor` transactor, runs the tests, and then stops the simulation and the service thread.

```
  // Create our transactor
  FactXactor factx(sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  // Run tests with this xactor
  runFactorialTest (factx) ;

  //Stop the simulation side
  factx.shutdown();

  // Stop and join with the service thread, then shut down scemi --
  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);

}
```

The test portion of the testbench is in the method `runFactorialTest`. Note that we're using the blocking methods for both the put request and the get response methods.

```
void runFactorialTest ( class FactXactor &xactor)
{
  long resp;
  for (int i = 0; i < 10; ++i) {
    xactor.putRequestB(i);
    cout << dec << "Sent: " << i << endl;
  }

  for (int i = 0; i < 10; ++i) {
    xactor.getResponseB(resp);
    cout << dec << "Received: (" << i << ") " << resp << endl;
   }
}
```

8.1.4 Building the example

The build file from the original factorial example can be used with one simple addition. In the target `[tb]` you need to add the new transactor file, `FactXactor.cpp` to the list of C++ files.

```
[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp FactXactor.cpp
exe-file: tb
```

This time, we're going to build for a Verilog simulator using the target `[vlog_dut]` and the C++ testbench:

```
build vlog_dut tb
```

Once the build is complete, you can run the verilog simulation and the testbench. Remember to start the simulator before starting the testbench. To run them in the same session, type:

```
./vlog_dut &
./tb
```

You should see the testbench results:

```
Sent: 0
Sent: 1
Sent: 2
Sent: 3
Sent: 4
Sent: 5
Sent: 6
Sent: 7
Sent: 8
Sent: 9
Received: (0) 1
Received: (1) 1
Received: (2) 2
Received: (3) 6
Received: (4) 24
Received: (5) 120
Received: (6) 720
Received: (7) 5040
Received: (8) 40320
Received: (9) 362880
SceMi Service thread finished!
```

8.2 Example 8: Writing a GUI-based Testbench

The emVM system includes components and templates to build a GUI front-end to control the testbench. The simple GUI, shown in Figure 14, has three control sections: Emulation, DUT, and Probe Control. The GUI can be used with or without probes defined; in this section we'll discuss using the GUI to control the simulation or emulation without probes. In Section 8.3 we'll add probes to the testbench and GUI.

The GUI testbench can be used with any hardware environment. In this example, we'll continue using a TCP simulation in Verilog. This same testbench could be the host side to a design running in emulation.

The following steps add a GUI to the factorial example and run the simulation and testbench from the GUI using the `FactXactor` transactor defined in the previous section.

1. Modify the hardware side: edit the `SceMiLayer.bsv` file to instantiate the Simulation Control transactor (`mkSimulationControl`), giving the software testbench control over the simulation.
2. Modify the software testbench: edit the software testbench template file `TclTb.cpp` to create a design-specific testbench for the example.
3. Specify the GUI environment
4. Run `build` for the hardware (`vlog_dut`) and software targets (`cpp_tb`)
5. Run the simulation
   (a) Start the hardware simulation (`dut.vexe`)
   (b) Start the testbench GUI (`debug_gui`)

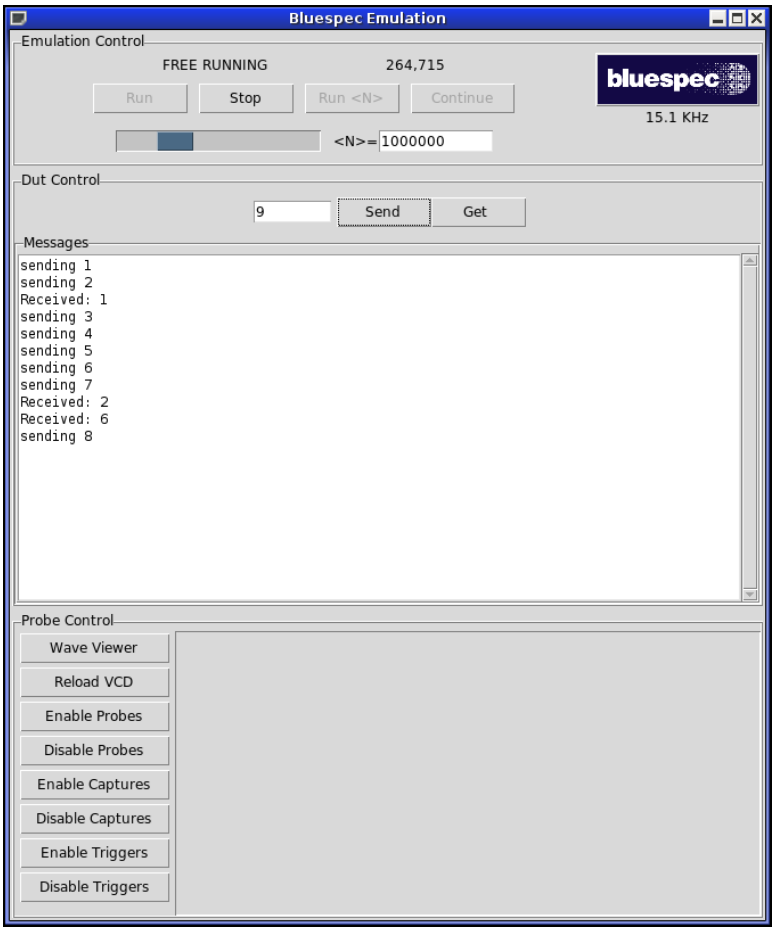

Figure 14: Simple Testbench GUI

8.2.1 Modify the Hardware Side - Instantiate Simulation Control

The only change on the hardware side is to instantiate the simulation control transactor, allowing the testbench to control starting and stopping the simulation in addition to sending and receiving values.

The GUI uses the C++ transactor `SimulationControl` to control the simulation from the software side. This requires the hardware-side `mkSimulationControl` transactor to be instantiated in the `SceMiLayer.bsv` file. Add this line to the `SceMiLayer.bsv` file:

```
Empty  simControl <- mkSimulationControl(conf);
```

where `conf` is the `SceMiClockConfiguration`.

The rest of the hardware side remains unchanged.

8.2.2 Modify the Software Testbench

Bluespec provides a C++ template (`TclTb.cpp`) which utilizes a GUI interface to control the testbench. The user must modify the template for the specific characteristics of the DUT and the

software transactor communicating with the DUT (`FactXactor` in our example). Below is a summary of some of the modifications made to the file to use the `FactXactor` transactor and create a GUI for the DUT. For the complete code listing, see A.8. The following excerpts highlight the areas of the file which require customization. The final `TclTb.cpp` file will be different for every DUT, based on inputs, outputs, and functionality.

- Add `#include` statements for the software transactor, `FactXactor` in our example:

```
#include "FactXactor.h"
```

- Define and initialize global data for the transactor:

```
// static extension global data
class SceMiGlobalData {
public:
  bool                  m_initialized ;
  SceMi                 * m_scemi;
  FactXactor            * m_factX;
  SceMiServiceThread    * m_serviceThread;
  SimulationControl     * m_simControl;
  ProbesXactor          * m_probeControl;

  // Simple initializer invoked when the extension is loaded
  SceMiGlobalData ()
    : m_initialized(false)
    , m_scemi(0)
    , m_factX(0)
    , m_serviceThread(0)
    , m_simControl(0)
    , m_probeControl(0)
  {}
```

- Initialize transactor:

```
    /* initiate the SCE-MI transactors and threads */
    // Create the transactor
    m_factX = new FactXactor (m_scemi);
```

- Edit `destroy` method to include DUT transactor:

```
  // Destruction -- called from bsdebug::scemi delete
  void destroy () {
    m_initialized = false ;
    // Stop the simulation side
    // if (m_inport) m_inport->shutdown();
    // if (m_outport) m_outport->shutdown();
    if (m_factX) m_factX -> shutdown();
  ...
    //Delete the Dut transactor
    delete m_factX; m_factX = 0;
```

- Edit the Dut command ensemble, to define the inputs and outputs.

```
// implementation of the Dut command ensemble
// dut request <int>
// dut response
static int Dut_Cmd(ClientData clientData,  //  &(GlobalXactor.m_pipelineX)
                   Tcl_Interp *interp,     // Current interpreter
                   int objc,               // Number of arguments
                   Tcl_Obj *const objv[]   // Argument strings
         )
{
  // Command table
  enum DutCmds { Dut_Request, Dut_Response };
  static const cmd_struct cmds_str[] = {
    {"request",         Dut_Request,   "int"}
    ,{"response",       Dut_Response,        ""}
    ,{0}                        // MUST BE LAST
  };
```

- Define case subcommand for request and response

```
  switch (command) {
    case Dut_Request:  // request x
      {
       //code for how to handle request
      }

    case Dut_Response:
      {
       //code for how to handle response
      }
```

8.2.3 Define GUI Environment

Next we'll examine the components of the tcl-based GUI to control and observe the test sequence. To execute the GUI you execute the file `debug_gui.tcl` which calls the tcl script `gui_top.tcl` to build the emulation window. There are three components to the emulation window:

1. Emulation Control - allows the user to control (start and stop) the hardware clocks. This component is found in emVM in the `ControlGui` library package. No modification is required by the user.

2. DUT control - sends various requests to the DUT and receives responses from the DUT. It may also send a software-reset or perform other test tasks. Messages received from the host are displayed in the message pane. This functionality is provided in the `GuiDut` package (`gui_dut.tcl`) and requires modification for the specifics of the DUT and the tests.

3. Probe Control - allows the user to enable probe logging and to view the probes on a waveform viewer. This is provided in emVM in the `ProbeGui` package. No modification is required by the user.

The Tcl/Tk file `debug_gui` sets up the GUI environment. Edit this file to specify a wave viewer along with the wave viewer options, load and setup the software side of the emVM environment. If the parameters file is named something other than `mkBridge.params`, edit this file to specify the name.

Most of the DUT-specific modifications for the GUI are in the file `gui_dut.tcl`, defining the DUT control component of the emulation control window. In our example, we have a *put* button, a *get* button, and a message box to display messages from the host. See A.8.2 for the source code for the GUI control.

8.2.4 Build

The target `tcl_tb` in the `project.bld` file contains the C++ directives required to build the GUI testbench.

```
[tcl_tb]
extends-target: dut
exe-file: cpp_tb
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: generated_c
c++-source-directory: .
c++-header-aliases
c++-options: -g -O0
shared-lib: libbsdebug.so
c++-files: TclTb.cpp FactXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp
```

Run build with the default targets: `vlog_dut` and `tcl_tb`.

8.2.5 Run Simulation from the GUI

As before, you start the hardware simulation, (`dut.vexe`) and then the software testbench (`debug_gui`). When you execute the software testbench the GUI window will be displayed, as shown in Figure 14.

To start the simulation (or emulation, if using an emulation board), click on the `Run` button in the `Emulation Control` section. This starts the clock. To send and receive data from the DUT, use the `Send` and `Get` buttons in the `Dut Control` section. The third section, `Probe Control`, is not used in this example. We'll add probes in the next section.

8.3 Example 9: Adding probes for debugging

Once you have the GUI configured and running it is very simple to add probes to return signal values. In this section we'll take the previous example and add a probe on the value of the register, `curr_result`.

emVM includes the HDL editor to add probe definitions directly to the design without modifying the DUT. Refer to the *emVM User Guide* manual for more information on the HDL editor.

8.3.1 HDL Editor

The HDL editor enables editing of the generated hardware description language (HDL) files to add probes and make other HDL-specific changes without modifying the source files. You can also use the tool to modify signal names and handle tedious edits in a controlled manner. The utility generates a new HDL file, and if selected, a new parameters (`.params`) file.

The HDL Editor is an optional step, run after the HDL file is generated and before the design is loaded onto the emulation board. You can save all defined edits including probe definitions in a script file (`<file name>.script`) to use in later runs, allowing consistent HDL modifications without duplicating effort.

The methodology for using the HDL editor to define probes in an emVM design has the following steps:

- Design without considering probes
- Compile BSV design to generate Verilog
- Run the HDL Editor, defining probes and any other edits, adding the probe definitions to the Verilog file
- Compile and link software testbench
- Continue with the simulation or emulation flow

The HDL editor generates a single message port between the software and hardware sides, which communicate via a single ProbeXactor. This technique is scalable and can be used with multiple FPGAs.

8.3.2 Invoking the HDL editor

The HDL editor can be invoked from either the command line or from the build script. We're going to use the build script. To invoke the tool during the build process, the `run-design-editor` directive in the build file must be set to `True`.

Open the `project.bld` file from the previous example and add the following directives to the `[dut]` target:

```
run-design-editor: True
design-editor-output-directory: vlog_edited
design-editor-edit-params: True
design-editor-options: --gui
```

8.3.3 Defining Probes through the HDL editor

Execute the `build` command to build the system. At the appropriate step in the build sequence, the HDL editor screen, shown in Figure 15 will display.

To add a probe to the design, select an object from the Design Hierarchy, then select a signal. For example, we're going to add a probe on the value of the register `curr_result`, as shown in Figure 16.

1. **Design Hierarchy** frame

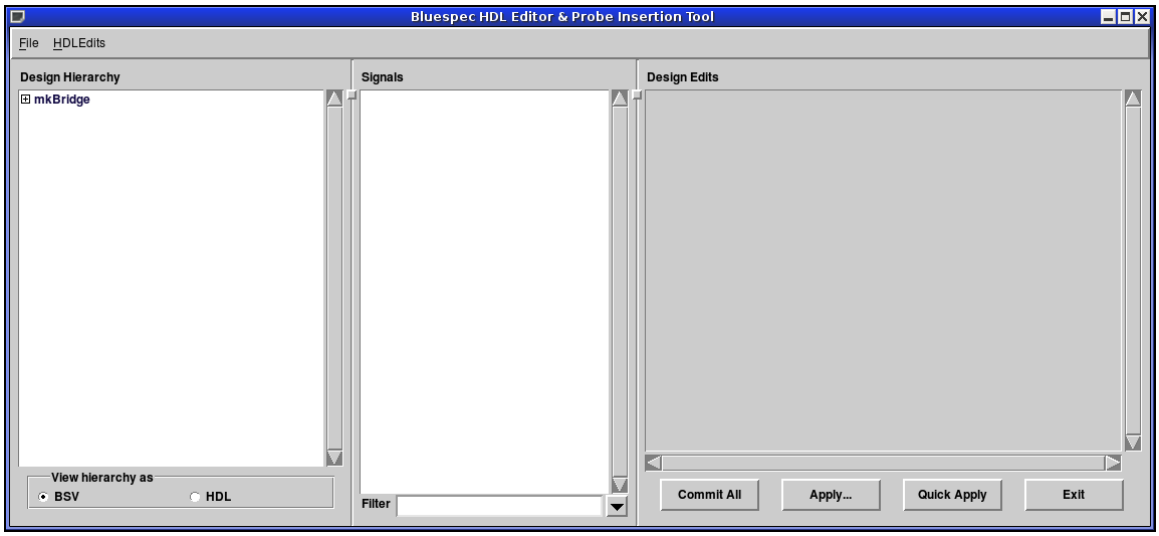

Figure 15: HDL Editor

- Expand `mkBridge/ scemi/dut/dutIfc/curr_result`

2. **Signals** frame
   - Select the first signal: `curr_result[63:0]`
3. **HDLEdits** menu
   - select **Add Probe**. A new object will be added in the **Design Edits** frame.
4. Edit any of the probe information. For example, the default name is generated by combining the path and signal; modify the name to a more meaningful name: `result`.
5. Select **Commit** or **Commit All** to verify and commit the edits.
6. Save the changes
   - Select **Apply...** The **Apply Changes Dialog** will be displayed. There are two types of changes to save. Apply both types of changes.
     - **Save to Script** saves the probe definition and other edits in a script to be reused.
     - **Apply** writes out a new `.v` file, and if selected, parameters file. These are the files used as you continue through the emulation process.
   - You could also select **Quick Apply** which will save both the script and a new Verilog file without the dialog window.
7. **Close** the Apply Changes window.
8. **Exit** the HDL editor to resume the build process.

Once you close the HDL editor, the build process will continue, using the edited Verilog file with the added probe.

8.3.4 Run the simulation

The simulation is run exactly like the previous example. Start the simulation `dut.vexe` then the software testbench `debug_gui`. When you execute the software testbench the GUI window will be displayed, as shown in Figure 17. Notice the probe definition on the bottom of the GUI.

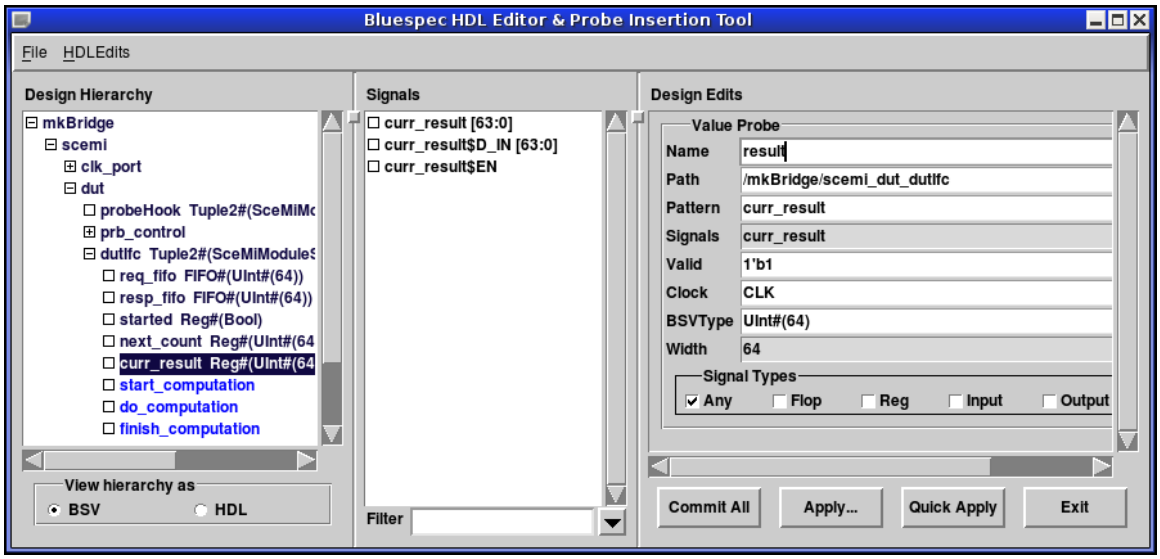

Figure 16: HDL Editor with Probe Defined

9 Example 10: Wrapping a Verilog DUT

In this section we're going to look at a simple example in which the factorial server DUT is provided as a Verilog module. The DUT in an emVM system can be written in BSV, Verilog, or other hardware definition language, the requirement being that the DUT must provide BSV interfaces to communicate with the emVM transactors in the SceMiLayer. For a DUT written in Verilog or VHDL you can easily generate a BSV wrapper for the DUT, using the `importBVI` statement in BSV, to provide BSV interfaces. This section describes how to write the BSV wrapper.

9.1 The Verilog file

The Verilog file `mkFactorialServer.v` implements the factorial server used in the earlier examples. The factorial generator has an input (`request`) and returns data (`response`). There are ready and enable signals on both the input and the output: `EN_request`, `RDY_request`, `EN_response` and `RDY_response`. There is a single clock and reset defined.

The inputs, outputs and clocks are defined in the Verilog file:

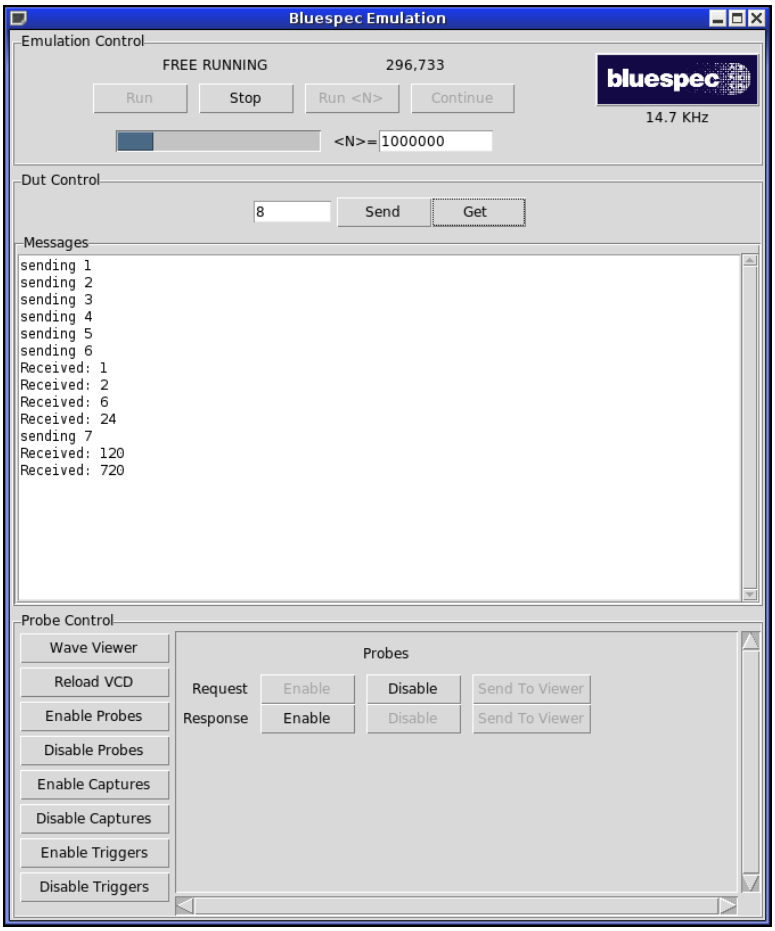

Figure 17: Testbench GUI with Probe

```
module mkFactorialServer(CLK,
                         RST_N,

                         IN_request,
                         EN_request,
                         RDY_request,

                         EN_response,
                         OUT_response,
                         RDY_response);
  input  CLK;
  input  RST_N;

  input  [63 : 0] IN_request;
  input  EN_request;
  output RDY_request;

  input  EN_response;
  output [63 : 0] OUT_response;
  output RDY_response;
```

9.2 Writing the wrapper

Writing the BSV wrapper for the Verilog file follows these steps:

1. Define BSV interface or import existing interfaces
2. Define module
3. Write clock and reset statements
4. Declare methods
5. Write schedule statements

9.2.1 Define the interface

The first task is to examine the DUT and by understanding its functionality and requirements define how you want to use the input and output wires. In Example 1 (Section 2) we used a `Server` interface which received a request and returned a response. We can use that same interface with our Verilog file. Remember, a `Server` interface contains the `Get` and `Put` subinterfaces:

```
interface Server#(type req_type, type resp_type);
    interface Put#(req_type)  request;
    interface Get#(resp_type) response;
endinterface: Server
```

To use the `Server` interface we must import the `ClientServer` and `GetPut` packages:

```
import ClientServer::*;
import GetPut::*;
```

In this example we are importing an interface already defined in BSV. You could instead define a new interface.

9.2.2 Define the module

The `import "BVI"` statement includes a module header that is the same as the typical module header. The module header for this example is:

```
import "BVI" FactorialServer =
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64) ));
```

where `FactorialServer` is the name of the Verilog file (`FactorialServer.v`). The name of the BSV module being defined is `mkFactorialServer`. Compare this statement with the module definition statement from Section 2:

```
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64) ));
```

They are identical with the addition of the `import "BVI"` statement before the module definition.

9.2.3 Clocks and Resets

For most single clock designs, the following clock and reset statements will suffice:

```
default_clock clk (CLK, (*inhigh*) clk_gate);
default_reset rst (RST_N);
```

where the name of the input clock port in the Verilog file is `CLK` and the name of the reset port is `RST_N`. For more complex clocking schemes, refer to the section on embedding RTL in a BSV design in the *Bluespec SystemVerilog Reference Manual*.

9.2.4 Define methods: input and output ports

The `method` statement is used to connect the Verilog input and output wires to methods in the BSV interface. The syntax imitates a function prototype in that it declares, not defines, the ports.

- Verilog input ports translate to:
  - Inputs into the module - arguments for any type method
  - EN signals indicating that an input should be driven into a design (such as enqueuing data into a fifo) or that a state change is to occur (such as "popping" data from a stack).
  - Clock or Reset inputs
- Verilog output ports translate to
  - Outputs from the module - return values from `ActionValue` or `Value` methods
  - RDY signals which signal that the module is ready to receive an input or that the output is valid
  - Clock or Reset outputs

Let's look at our simple example. We've already grouped (and named) the input and output signals according to their functions:

```
// action method request
input  [63 : 0] IN_request;
input  EN_request;
output RDY_request;

// actionvalue method response
input  EN_response;
output [63 : 0] OUT_response;
output RDY_response;
```

The methods must be grouped into interfaces. We already determined that this design has a single interface with two subinterfaces: `request` and `response`, corresponding the the `request` and `response` subinterfaces of the `Server` interface. Each subinterface is going to have a single method, with a ready and an enable.

The `Put` interface is always named `request` in a `Server` interface definition. The interface has a single method, named `put`, as specified in the `GetPut` package. The Verilog port providing the input value is `IN_request`. The Verilog port names are in the parentheses.

```
interface Put request;
   method put (IN_request) ready(RDY_request) enable(EN_request);
endinterface
```

The `get` method returns a value (`OUT_response`) represented by the Verilog output port. The name of the Verilog port immediately follows the `method` keyword. The format of the ready and enable ports is the same for all interface methods.

```
interface Get response;
   method OUT_response get ready(RDY_response) enable(EN_response);
endinterface
```

9.2.5 Schedule

We specify the scheduling constraints between the methods in the wrapper with `schedule` statements. Since we have only a single input and a single output statement, there are not many scheduling constraints to add.

The hierarchy separator is an underscore. Therefore, the full name of the methods are *interface name_method name*. For example, the `Put` interface is named `request`, and it has a `put` method. So the full name of the method is: `request_put`.

- An input port is usually defined to conflict with itself (C). This implies that the port can't be driven from different locations on the same clock cycle.

  ```
  schedule (request_put) C (request_put);
  ```

- If there are multiple inputs, each input may be conflict-free (CF) with other inputs. This is the most general case, but designs may have conflicts between inputs. Careful consideration is needed, based on your knowledge of the design, to specify any inputs which cannot be driven on the same clock cycle, as conflicts.

- Ports which can be used simultaneously are denoted as conflict-free (CF). Output ports are typically CF since the signal can be fanned out to different locations in the instantiating module.

  ```
  schedule (request_put) CF (response_get);
  ```

- In a registered design (a typical "read-modify-write" flow), the outputs will be scheduled before (SB) the inputs (i.e. we read from a register before updating it). If there is a combinational path from input to output, then the input should be sequenced before (SB) the output.

9.2.6 BSV wrapper

The completed BSV file `mkFactorialServer.bsv` used with the Verilog file `FactorialServer.v` is shown below.

```
import ClientServer::*;
import GetPut::*;

import "BVI" mkFactorialServer =
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64)));
   default_clock clk (CLK, (*inhigh*) clk_gate);
   default_reset rst (RST_N);

   interface Put request;
      method put (IN_request) ready(RDY_request) enable (EN_request);
   endinterface

   interface Get response;
      method OUT_response get ready(RDY_response) enable(EN_response);
   endinterface

    schedule (request_put) C (request_put);
    schedule (response_get) C (response_get);
    schedule (request_put) CF (response_get);
endmodule
```

9.3 SceMiLayer

The rest of the design remains the same, except that in this example the name of the package containing the DUT is `mkFactorialServer.bsv` instead of `DUT.bsv`. The complete SceMiLayer file is found in Appendix A.10.

9.4 Building the example

The only change required in the `project.bld` file is to add the name of the imported Verilog file. The `imported-verilog-files` directive in the build file lists the Verilog files to be included in linking the Verilog simulation or synthesis executable.

```
[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build
imported-verilog-files: mkFactorialServer.v
```

10 Example 11: Using TLM transactors to integrate a Verilog AHB memory

This example expands on the previous example of wrapping a Verilog DUT (Section 9) by using a more complex TLM transactor to communicate with the DUT instead of the basic `Get` and `Put` interfaces. In this example the DUT is an AHB slave memory provided as a Verilog file. The AzureIP premium offerings provided by Bluespec support development of bus-based designs implementing AXI and AHB protocols. We are going to use a subset of this library along with the TLM transactors

provided in the AzureIP fabric library to demonstrate how these library components can be used to easily implement complex designs in emVM. For the purposes of this tutorial, the `Ahb` package, which is based on the `AHB` package, is provided.

10.1 Hardware side overview

10.2 Writing the wrapper

As presented in Section 9, the steps for writing the BSV wrapper are:

1. Define BSV interfaces or import existing BSV interfaces
2. Define module
3. Write clock and reset statements
4. Declare methods
5. Write schedule statements

10.2.1 Define the interface

The BSV wrapper describes how to use the Verilog input and output wires, translating the Verilog pins to BSV interfaces. In this example, instead of generic `Get` and `Put` interfaces we are using the `AhbSlaveSTD` interface, which is a specific version of a pre-defined general `AHBXtorSlave` interface from the AzureIP premium `AHB` package.

To use the `AHBXtorSlave` interface we import the `Ahb` package. The file `TLM.defines` contains macros defining parameters for the data definitions. To simplify the code, we'll define names for the parameterized interfaces.

```
import Ahb      ::*;

`include "TLM.defines"

typedef AhbXtorSlave#(`TLM_PRM_STD) AhbSlaveSTD;
typedef AhbXtorMaster#(`TLM_PRM_STD) AhbMasterSTD;
```

```
interface AhbXtorSlave#(`TLM_PRM_DCL);
   interface AhbSlave#(`TLM_PRM)         bus;
   interface AhbSlaveSelector#(`TLM_PRM) selector;
endinterface
```

We'll examine the methods defined in the interface a bit later in the example.

10.2.2 Define the module

The `import "BVI"` statement includes the name of the Verilog module being imported (`mkAhbRam.v`), the name of the BSV module being defined (`mkAhbRam`), and the interface provided by the module.

```
import "BVI" mkAhbRam =
module [Module] mkAhbRam (AhbSlaveSTD);
```

10.2.3 Clocks and Resets

As this is a single clock, single reset design, the basic clock and reset statements are used.

```
default_clock clk (CLK, (*inhigh*) clk_gate);
default_reset rst (RST_N);
```

10.2.4 Define methods: input and output ports

The `method` statements are use to connect the Verilog input and output wires to the methods in the interface. Because we are using the `Ahb` package the inputs and outputs correspond exactly to the interface methods.

The inputs and outputs as defined in the Verilog file:

```
input  [31 : 0] HADDR;
input  [31 : 0] HWDATA;
input  HWRITE;
input  [1 : 0] HTRANS;
input  [2 : 0] HBURST;
input  [2 : 0] HSIZE;
input  [3 : 0] HPROT;
input  HREADY;
output [31 : 0] HRDATA;
output HREADYOUT;
output [1 : 0] HRESP;
output [15 : 0] HSPLIT;
input  HSEL;
```

In this example we express each input or output as a method and eliminate the associated ready and enable signals. For example, the input `HADDR` in the Verilog file:

```
input  [31 : 0] HADDR;
```

is represented by the method `haddr` in the BSV wrapper:

```
method haddr (HADDR)   enable((*inhigh*) en1);
```

Similarly, the outputs are expressed as:

```
method Verilog-output-port bsv-name();
```

An example is the output `HRDATA` in the Verilog file:

```
output [31 : 0] HRDATA;
```

is represented by the Value method `htdata` in the BSV wrapper:

```
   method AhbData#(‘TLM_PRM) hrdata;
```

Where `AhbData#(‘TLM_PRM)` is the data type.

```
interface AhbSlave#(‘TLM_PRM_DCL);
    // Inputs
   (* prefix = "", result = "unused0" *)
   method Action    haddr((* port = "HADDR" *)     AhbAddr#(‘TLM_PRM) addr);
   (* prefix = "", result = "unused1" *)
   method Action    hwdata((* port = "HWDATA" *)   AhbData#(‘TLM_PRM) data);
   (* prefix = "", result = "unused2" *)
   method Action    hwrite((* port = "HWRITE" *)   AhbWrite    value);
   (* prefix = "", result = "unused3" *)
   method Action    htrans((* port = "HTRANS" *)   AhbTransfer value);
   (* prefix = "", result = "unused4" *)
   method Action    hburst((* port = "HBURST" *)   AhbBurst    value);
   (* prefix = "", result = "unused5" *)
   method Action    hsize((* port = "HSIZE" *)     AhbSize     value);
   (* prefix = "", result = "unused6" *)
   method Action    hprot((* port = "HPROT" *)     AhbProt     value);
   (* prefix = "", result = "unused7" *)
   method Action    hreadyin((* port = "HREADY" *) Bool      value);

   // Outputs
   (* result = "HRDATA" *)
   method AhbData#(‘TLM_PRM) hrdata;
   (* result = "HREADYOUT" *)
   method Bool               hready;
   (* result = "HRESP" *)
   method AhbResp            hresp;
   (* result = "HSPLIT" *)
   method AhbSplit           hsplit;
endinterface
```

10.2.5 Schedule

The scheduling guidelines are:

- All outputs are scheduled to be conflict-free (CF)
- All outputs are scheduled before (SB) the inputs (read-modify-write)
- Each input conflicts (C) with itself, but is conflict-free (CF) with all other inputs.

Following the guidelines, a subset `schedule` statements for this example are:

```
// we schedule all outputs to be CF (conflict-free)
schedule (bus_hrdata, bus_hready, bus_hresp, bus_hsplit) CF
  (bus_hrdata, bus_hready, bus_hresp, bus_hsplit);

// scheduling all outputs to be SB inputs (read-modify-write)
schedule  (bus_hrdata, bus_hready, bus_hresp, bus_hsplit) SB
   (bus_haddr, bus_hwdata, bus_hwrite, bus_htrans, bus_hburst,
    bus_hsize, bus_hprot, bus_hreadyin, selector_select);

// each input C with itself, but it is CF with all other inputs
schedule (bus_haddr)  C  (bus_haddr);
schedule (bus_haddr)  CF (bus_hwdata, bus_hwrite, bus_htrans,
  bus_hburst, bus_hsize, bus_hprot, bus_hreadyin, selector_select);
```

The complete BSV wrapper is found in Appendix A.11.

10.3 SceMiLayer

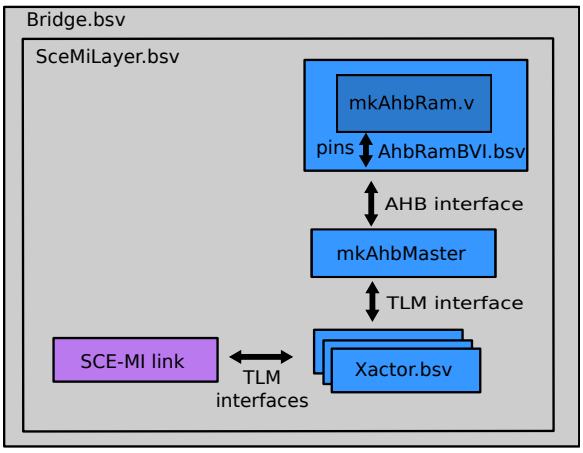

Figure 18: SceMiLayer converting AHB interface to TLM interface

The SceMiLayer, shown in Figure 18, contains the DUT-specific, FPGA-independent components instantiating the hardware-side emVM transactors and defining the clocks and resets. In this step we instantiate the DUT and connect it to the emVM transactors. As we saw in our previous example, the SceMiLayer transactors are based on `Get` and `Put` interfaces. Our AhbRam DUT provides an AHB interface. To connect the AHB interface with the emVM transactors in the SceMiLayer we use the AHB-Master transactor. The AHB-Master transactor has a simple get-put interface towards the host that translates TLM to complete AHB transactions to the DUT. This transactor is defined in the Bluespec-provided package `TLM2`.

The SceMiLayer package defines the `mkSceMiLayer` module which instantiates the emVM control components (clocks, resets, simulation control) and connects the DUT into the emVM fabric. Only a few of the steps are based on the specific requirements of the DUT. The rest can be used with other DUTs, unless you have different clock or reset requirements.

General Steps: These statements can be used as written in this example with other DUTs.

- Define an instance of a controlled clockport to clock the DUT. This clock can be stopped by the user as required.

- Define a new reset, to reset the DUT from the host software.
- Instantiated the transactors that enable host-to-DUT communication.
- Instantiate the reset transactor.

DUT-specific steps: These statements must be modified according to the definition and requirements of the DUT.

- Instantiate the DUT with the special `buildDutWithReset` constructor, which is required by the emVM environment. Or use the `buildDUT` constructor if there is no reset defined.
- Instantiate the appropriate TLM transactor to connect with the DUT-provided interface.
- Connect the DUT to the TLM transactor.
- Instantiate the inport and outport emVM transactors to communicate with the software host side.

The `buildDutWithReset` constructor instantiates the DUT, connecting it with the specified clock-port and reset, providing the interface type of the DUT. It is similar to the `buildDut` constructor we used in the previous example, but with a reset defined.

```
AhbSlaveSTD dut  <- buildDutWithReset( mkAhbRam, clk_port, dut_reset.new_rst );
```

The `mkAhbRam` module is our DUT in this example. The line below instantiates the AHB-Master transactor, which translates TLM requests from the host to AHB requests to the DUT, and in return translates AHB responses from the DUT to TLM responses to the host. This transactor was selected because it connects with the `AhbSlaveStd` interface of the DUT. The transactor used depends on the interface type and functionality of the DUT.

```
AhbMasterXActor#(`TLM_XTR_STD) xactor  <- mkAhbMaster(clocked_by c_clock,
                                          reset_by dut_reset.new_rst);
```

The DUT instance (`dut`) and the transactor instance (`xactor`) are connected via a `mkConnection` statement. `xactor.fabric` refers to the AHB subinterface of the AHB-Master transactor that connects to the DUT's interface.

```
mkConnection(xactor.fabric, dut);
```

The following statements instantiate the inport and outport TLM transactors. The `mkPutXactor` passes TLM requests from the host to the DUT, while the `mkGetXactor` gets the TLM responses from the DUT to send to the host. For each input into the DUT, a `mkPutXactor` is instantiated. For each output from the DUT, a `mkGetXactor` is instantiated. This example has a single input and a single output. The third transactor connects a signal coming from the host testbench to the reset. The testbench *puts* the reset value to the hardware side.

```
Empty inReq       <- mkPutXactor(xactor.tlm.rx, clk_port);
Empty outResp     <- mkGetXactor(xactor.tlm.tx, clk_port);
Empty rst_control <- mkPutXactor (soft_reset, clk_port);
```

The DUT is now fully instantiated and connected into the hardware side of the emVM system.

10.4 Building the hardware side

It is necessary at this point in the example to build the hardware side to generate the `mkBridge.params` file. This file is read by the software side during initialization to bind the software side message port proxies to the hardware side message ports. We generate the file now to find the names of the transactors, portnames, and data types, which we will use in writing the software-side transactors.

The command to build the hardware side for Verilog is:

```
build vlog_dut
```

In the next sections we'll refer to definitions from the `mkBridge.params` file. The complete file can be found in Appendix A.11.

10.5 The software testbench

The software testbench for this example requires a few updates, since the definition of the inputs and outputs is different than in our factorial examples. Most of these changes will be implemented by writing a new software-side transactor, `SlaveXactor`, and using that transactor in our testbench, `TclTb.cpp`.

10.5.1 Data definitions

The inputs and outputs of the previous factorial examples used `UInt#(64)` data types on the hardware side which corresponded to the the C++ data classes `BitT<64>`. In this example, the data types are more complex. As discussed in Section 4, emVM includes the `generateSceMiHeaders` utility to define more complex data types. Since our build file `project.bld` does not include the `c++-header-targets` directive, the default is that the utility will be run. It will automatically generate the `.h` files for the data types. Looking at the `mkBridge.params` file we see that we have two new data types: `TLMResponse#(4,32,32,10,Bit#(0))` and `TLMRequest#(4,32,32,10,Bit#(0))`. When we build the software side, the header files to define the data types will be generated.

10.5.2 SlaveXactor.h

In Section 8.1 we wrote the software-side transactor for the factorial example, `FactXactor`. In this section we'll look at how we need to modify that transactor to create the software transactor `SlaveXactor`.

The definitions of the message ports, including the data types, are found in the `mkBridge.params` file:

```
MessageOutPort 2 TransactorName ""
MessageOutPort 2 PortName       "scemi_outResp_outport"
MessageOutPort 2 PortWidth      58
MessageOutPort 2 ChannelId      8
MessageOutPort 2 Type           "TLM2Defines::TLMResponse#(4,32,32,10,Bit#(0))"

MessageInPort 2 TransactorName ""
MessageInPort 2 PortName       "scemi_inReq_inport"
MessageInPort 2 PortWidth      110
MessageInPort 2 ChannelId      3
MessageInPort 2 Type           "TLM2Defines::TLMRequest#(4,32,32,10,Bit#(0))"
```

As in the FactXactor.h file, the SlaveXactor.h has a InportProxyT (m_datain) and OutportQueueT (m_dataout) transactors. The difference is the data type. We have to include the header files for the new data types:

```
#include "TLMRequest_4_32_32_10_Bit_0.h"
#include "TLMResponse_4_32_32_10_Bit_0.h"
```

And we have to change the data transactor definitions to use the new data types:

```
  InportProxyT<TLMRequest_4_32_32_10_Bit_0 >     m_datain;
  OutportQueueT<TLMResponse_4_32_32_10_Bit_0 >   m_dataout;
```

Since there is a software-side controlled reset in this example, we also need to define a transactor for the reset. We also keep the shutdown transactor from our earlier definition.

```
  // Shutdown Xactor
  ShutdownXactor         m_shutdown;
  // soft-reset Xactor
  InportQueueT<Bool >    m_soft_rst;
```

The constructors are defined with the new data types:

```
  bool putRequestNB(const TLMRequest_4_32_32_10_Bit_0 &);
  bool putRequestB (const TLMRequest_4_32_32_10_Bit_0 &);
  bool getResponseNB (TLMResponse_4_32_32_10_Bit_0 &);
  bool getResponseB (TLMResponse_4_32_32_10_Bit_0 &);

  void shutdown();

  // send reset from the software side
  bool sendReset();
```

10.5.3 SlaveXactor.cpp

The names used in the constructor definition in the SlaveXactor.cpp file are found in the mkBridge.params file:

```
SlaveXactor::SlaveXactor(SceMi *scemi)
  : m_soft_rst("", "scemi_rst_control_inport", scemi)
  , m_datain ("", "scemi_inReq_inport", scemi)
  , m_dataout ("", "scemi_outResp_outport", scemi)
  , m_shutdown ("", "scemi_control", scemi)
```

The rest of the file is changed to use the appropriate data types.

```
bool SlaveXactor::putRequestNB (const TLMRequest_4_32_32_10_Bit_0 &request)
{
  TLMRequest_4_32_32_10_Bit_0 reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}
bool SlaveXactor::putRequestB (const TLMRequest_4_32_32_10_Bit_0 &request)
{
  TLMRequest_4_32_32_10_Bit_0 reqbit(request);
  m_datain.sendMessage(reqbit);
  return true;
}

// return true and populates resp if a response is available
// otherwise returns false
bool SlaveXactor::getResponseNB (TLMResponse_4_32_32_10_Bit_0 &resp)
{
  TLMResponse_4_32_32_10_Bit_0 respbit;
  bool gotone = m_dataout.getMessageNonBlocking (respbit) ;
  if (gotone) {
    resp = respbit;
  }
  return gotone;
}

bool SlaveXactor::getResponseB (TLMResponse_4_32_32_10_Bit_0 &resp)
{
  TLMResponse_4_32_32_10_Bit_0 respbit = m_dataout.getMessage () ;
  resp = respbit;
  return true;
}

bool SlaveXactor::sendReset() {
  m_soft_rst.sendMessage( true ) ;
  return true;
}
```

10.5.4 TclTb.cpp

The `TclTb.cpp` file is the middle layer between the DUT-transactor (`SlaveXactor.cpp`) and a software control layer (tcl in this specific example). It *translates* transactions from the tcl layer towards the DUT-transactor and defines the functions necessary for the tcl API.

The testbench file `TclTb.cpp` should now use the `SlaveXactor` instead of the `FactXactor`.

```
#include "SlaveXactor.h"
#include "SceMiHeaders.h"

class SceMiGlobalData {
public:
  bool                        m_initialized ;
  SceMi                 * m_scemi;
  SlaveXactor                 * m_slaveX;
  SceMiServiceThread    * m_serviceThread;
  SimulationControl         * m_simControl;
  ProbesXactor          * m_probeControl;
```

The handling of the request has a different implementation since the data requirements of the AHB request are different that the factorial. For example, the `Dut_Request` has to be broken down into its components.

```
case Dut_Request:  // request x y
      {
        if (objc != 5) goto wrongArgs;
        TLMRequest_4_32_32_10_Bit_0 a2;
        RequestDescriptor_4_32_32_10_Bit_0 a1;
        int rw;
        int addr;
        int data;
        if (Tcl_GetIntFromObj(interp, objv[2], &rw) != TCL_OK) {
          return TCL_ERROR;
        }
        if (Tcl_GetIntFromObj(interp, objv[3], &addr) != TCL_OK) {
          return TCL_ERROR;
        }
        if (Tcl_GetIntFromObj(interp, objv[4], &data) != TCL_OK) {
          return TCL_ERROR;
        }
```

And when structuring the request, the fields must be set to appropriate values for the DUT:

```
        a1.m_command = rw ? TLMCommand::e_WRITE : TLMCommand::e_READ;
        a1.m_addr = addr;
        a1.m_data = data;
        a1.m_export_id = 0;
        a1.m_transaction_id = 0;
        a1.m_thread_id =0;
        a1.m_lock =0;
        a1.m_prty =0;
        a1.m_burst_size =3;
        a1.m_burst_mode = TLMBurstMode::e_INCR;
        a1.m_byte_enable = 15;
        a1.m_burst_length =1;
        a1.m_mode = TLMMode::e_REGULAR;

        a2.the_tag = 0;
        a2.m_Descriptor = a1;

        bool sent = slavex->putRequestNB(a2);
        Tcl_Obj *r = Tcl_NewBooleanObj( sent );
        Tcl_SetObjResult(interp, r );
```

Since this example uses a soft reset, a section for handling the reset must be added:

```
        if ( rw == 10 ) {
          slavex -> sendReset();
          Tcl_Obj *r = Tcl_NewBooleanObj( true );
          Tcl_SetObjResult(interp, r );
          break;
        }
```

10.5.5 GuiDut.tcl

The `GuiDut.tcl` file defines the functionality of the DUT control panel in the test GUI. In addition to the *Send* and *Get* buttons defined in the previous example, we must also define a button for the soft reset. Also, the request sent tot eh DUT is associated with 3 user inputs: 0/1 for read/write, an address, and 32 bit data (set as `entry1`, `entry2`, and `entry3` respectively.

```
   ## Button Frame
   set button_frame [ttk::frame $top.button_frame]
   set rstbut [ttk::button $button_frame.rst -text "Reset" -command
"GuiDut::do_reset"]
   set entry1 [ttk::entry $button_frame.e1 -textvariable GuiDut::e1
-width 8 -validate key -validatecommand "GuiDut::chk_num %P"]
   set entry2 [ttk::entry $button_frame.e2 -textvariable GuiDut::e2
-width 8 -validate key -validatecommand "GuiDut::chk_num %P"]
   set entry3 [ttk::entry $button_frame.e3 -textvariable GuiDut::e3
-width 8 -validate key -validatecommand "GuiDut::chk_num %P"]
   set putbut [ttk::button $button_frame.put -text "Send" -command
"GuiDut::do_send"]
   set getbut [ttk::button $button_frame.get -text "Get " -command
"GuiDut::do_get"]
```

Next, there are definitions for the procedures `do_reset`, `do_send`, and `do_get`, which are called when the reset, send and get buttons are pushed:

```
proc do_send {} {
     variable e1
     variable e2
     variable e3
     variable textbox

     set msg "Blocked"
     set sent [dut request $e1 $e2 $e3]
     if {$sent} {
         set msg "sending $e1 $e2 $e3"
         # change the address and data values whenever a message is sent.
         incr e2 4
         incr e3 1
      }
      $textbox insert end "$msg\n"
      $textbox yview end
}
```

10.6 Build and run the example

A complete `project.bld` file is in the example. The main changes are that the file includes the name of the imported Verilog file in the `[vlog_dut]` section and the `SlaveXactor` in the testbench section.

```
[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
scemi-tcp-port: 3375
exe-file:       dut.vexe
verilog-simulator: modelsim
imported-verilog-files: AhbRam/mkAhbRam.v  AhbRam/myTlmRam.v AhbRam/myAhbSlave.v


[tcl_tb]
extends-target: dut
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: c_tb
c++-source-directory: tcl
c++-header-aliases
c++-options: -g -O8 -I $BLUESPECDIR/SceMi/bsvxactors
shared-lib: tcl/libbsdebug.so
c++-files: TclTb.cpp SlaveXactor.cpp ${BLUESPECDIR}/tcllib/include/bsdebug_common.cpp
```

Commands to build and run the example:

- `build`
- `./dut.vexe $`
- `./debug_gui`

11 Example 12: Implementing a synthesizable testbench

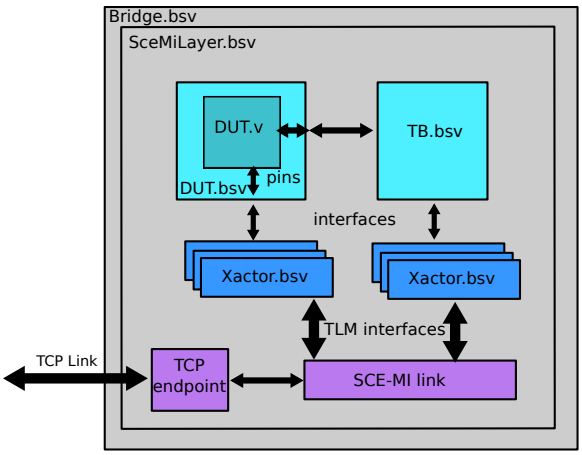

Figure 19: Hardware side with Synthesizable Testbench

In an emVM system, the quantity of data moving from the software testbench to the hardware side across the communications link can cause a bottleneck. One way to reduce the traffic across the link is to move all or part of the testbench on to the FPGA.

As you can see in Figure 19, the SceMiLayer can instantiate both the design under test (DUT) as well as a BSV or Verilog-based testbench (TB). The DUT and the TB communicate directly with each other. They can also connect independently through hardware transactors to the SceMiLayer, to communicate to the host side.

The testbench is structured to minimize the data sent across the communciation link. For example, the host sends a signal to send a packet, but the packet is assembled and sent to the DUT by the TB on the hardware side.

The remaining components of emVM remain the same; you can add probes and use a GUI to control the software side of the testbench.

In this example we're going to implement a very simple hardware testbench, to demonstrate the structure of a synthesizable testbench on the hardware side. A typical synthesizable testbench would most likely be much more complicated, with more data passing between DUT and the testbench.

11.1 Hardware side overview

We'll continue using the `mkFactorialServer` DUT, which takes a request value and returns the factorial of that value and we'll add a testbench, `Tb.bsv`, (TB) to the hardware side. TB includes two modules: a stimulus generator, `mkStimulusGen`, and a top level, `mkTop`, as shown in Figure 20. The stimulus generator and the factorial server are instantiated and connected in `mkTop` which receives a seed value from the software testbench, then sends the value and starts the stimulus generator. The stimulus generator sends 10 values to the factorial server, then stops until the next

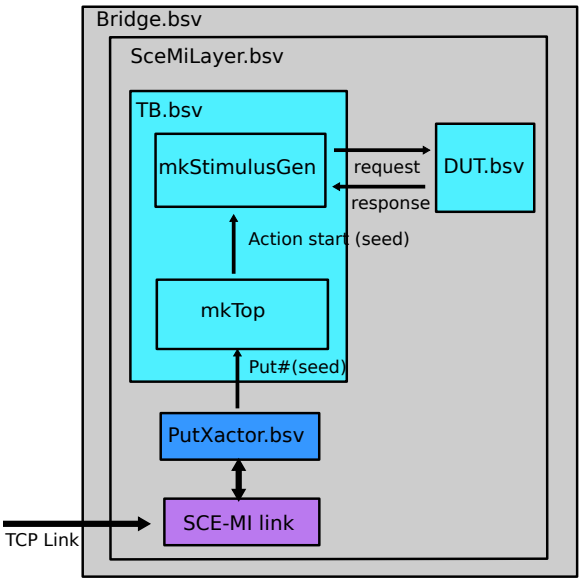

Figure 20: Synthesizable Testbench

seed value is sent from the software testbench. A single value is sent across the communication link from the software to the hardware side for each 10 factorial values requested, reducing the traffic on the communications link.

11.2 Stimulus Generator

The stimulus generator receives a seed value from the software testbench (through the `mkTop` module), starts the generator, and sends the requested value to the factorial server. The generator also receives back the calculated factorial. The `StimIfc` interface provided by the generator has three subinterfaces:

```
interface StimIfc;
   method Action start (UInt#(64) seed);
   interface Get#(UInt#(64)) stim_request;
   interface Put#(UInt#(64)) stim_response;
endinterface
```

The `Get` and `Put` interfaces communicate with the factorial server. The `start` methods receives the the seed value from `mkTop`.

```
   interface Put stim_response = toPut(f_in);
   interface Get stim_request = toGet(f_out);

   method Action start(UInt#(64) seed) ;
      value_sent <= seed;
   endmethod
```

11.3 Top Level mkTop

The testbench package `Tb.bsv` contains the stimulus generator and a top level module to control the testbench. Since the software testbench *puts* a value into the hardware side, the `mkTop` module provides a `Put#(UInt#(64))` interface.

```
module [Module] mkTop (Put#(UInt#(64)));
```

The value received through the `Put` interface is the seed value for the stimulus generator.

The `mkTop` module instantiates and connects the stimulus generator and the factorial server (DUT).

```
StimIfc     stim_gen <- mkStimulusGen;
ServerIfc   dut      <- mkFactorialServer;

mkConnection(stim_gen.stim_request, dut.request);
mkConnection(stim_gen.stim_response, dut.response);
```

11.4 SceMiLayer

The SceMiLayer remains the same in structure, with some minor changes. Instead of instantiating the DUT, the `mkTop` module is instantiated. The `mkTop` module provides a `Put#(UInt#(64))` interface, so the `mkPutXactor` transactor is used to connect the testbench interface into the emVM environment. There are no other changes to the SceMiLayer.

```
// Instantiate the testbench
Put#(UInt#(64)) testbench <- buildDut(mkTop, clk_port);

// Connect the tb interface to SceMi
Empty testconnect <- mkPutXactor(testbench, clk_port);
```

11.5 The software testbench

The software transactors and the software testbench change because the definition of the data moving between the software and hardware sides has changed. In previous examples, a value was sent from the software side and a value was received by the software side. In this example, the software testbench sends a value, but does not receive a value. The example includes a new transactor, `TbXactor`, to use in place of the `FactXactor` transactor. The `TbXactor` contains two transactors: an `InportProxyT` to send the seed value from the testbench to the HW side, and a `ShutdownXactor` to control the simulation from the software side. the `OutportQueueT` transactor, which received the calculated factorial, has been removed. These changes require modification of the following files:

- Transactor files (`TbXactor.cpp, TbXactor.h`): Create a new transactor `TbXactor` to replace `FactXactor`, removing the `OutportQueueT` transaactor.
- Testbench control file (`TclTb.cpp`): Replace the transactor name. Remove all references to data being returned from the hardware side.
- The GUI control file (`gui_dut.tcl`): Remove the variables and buttons for the `Get` buttons.
- `project.bld`: Change the transactor name from `FactXactor.cpp` to `TbXactor.cpp` in the `tcl_tb` section with the `c++-files` directive.

11.6 Build and run the example

The project is built and run as before:

- `build`
- `./dut.vexe $`
- `./debug_gui`

A The source files

The complete source code files to run each example are found in the subdirectory for the example. Source code listings below are only provided for a subset of files which have changes from previous versions or have other items of interest.

A.1 Example 1: Factorial Example Files

The source files for this example are found in the directory `example1`.

A.1.1 DUT

```
package DUT;

import GetPut::*;
import FIFO::*;
import ClientServer::*;

(* synthesize *)
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64) ));
   FIFO#(UInt#(64)) req_fifo <- mkFIFO();
   FIFO#(UInt#(64)) resp_fifo <- mkFIFO();

   Reg#(Bool) started <- mkReg(False);
   Reg#(UInt#(64)) next_count <- mkReg(0);
   Reg#(UInt#(64)) curr_result <- mkRegU();

   rule start_computation (!started);
      // Get the value
      let val = req_fifo.first();
      req_fifo.deq();
      // Set up the computation state
      curr_result <= 1;
      next_count <= val;
      started <= True;
      $display("Starting request: %h", val);
   endrule

   rule do_computation (started && (next_count > 0));
      let next_result = curr_result * next_count;
      curr_result <= next_result;
      // handle overflow
      if (next_result == 0)  begin
 $display("Overflow!");
         next_count <= 0;
      end
      else begin
         next_count <= next_count - 1;
      end
   endrule

   rule finish_computation (started && (next_count == 0));
      started <= False;
      resp_fifo.enq(curr_result);
      $display("Sending result: %h", curr_result);
   endrule
```

```
   interface Put request = toPut(req_fifo);
   interface Get response = toGet(resp_fifo);
endmodule

endpackage
```

A.1.2 SceMiLayer

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;

import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

   // Connect the dut interface to SceMi
   Empty dutout <- mkGetXactor(dut.response, clk_port);
   Empty dutin  <- mkPutXactor(dut.request, clk_port);

   Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

A.1.3 Bridge

```
package Bridge;

import SceMi::*;
import SceMiLayer::*;

(* synthesize *)
module mkBridge ();
   Empty scemi <- buildSceMi(mkSceMiLayer, TCP);
endmodule
endpackage
```

A.1.4 Parameters File (mkBridge.params)

```
// Sce-Mi parameters file generated on: Mon Jan 16 09:43:59 EST 2012
// By: Bluespec scemilink utility, version 2011.12.beta1 (build 26438, 2011-12-13)

// ObjectKind Index AttributeName Value
```

```
ClockBinding 0 TransactorName ""
ClockBinding 0 ClockName      "scemi_clk_port"

Clock 0 ClockName          "scemi_clk_port"
Clock 0 RatioNumerator     1
Clock 0 RatioDenominator   1
Clock 0 DutyHi             0
Clock 0 DutyLo             100
Clock 0 Phase              0
Clock 0 ResetCycles        8

MessageOutPort 2 TransactorName ""
MessageOutPort 2 PortName       "scemi_shutdown_ctrl_out"
MessageOutPort 2 PortWidth      1
MessageOutPort 2 ChannelId      6
MessageOutPort 2 Type           "Bool"

MessageOutPort 1 TransactorName ""
MessageOutPort 1 PortName       "scemi_dutout_outport"
MessageOutPort 1 PortWidth      64
MessageOutPort 1 ChannelId      5
MessageOutPort 1 Type           "UInt#(64)"

MessageOutPort 0 TransactorName ""
MessageOutPort 0 PortName       "scemi_dut_prb_control_data_out"
MessageOutPort 0 PortWidth      32
MessageOutPort 0 ChannelId      4
MessageOutPort 0 Type           "Bit#(32)"

MessageInPort 2 TransactorName ""
MessageInPort 2 PortName       "scemi_shutdown_ctrl_in"
MessageInPort 2 PortWidth      1
MessageInPort 2 ChannelId      3
MessageInPort 2 Type           "Bool"

MessageInPort 1 TransactorName ""
MessageInPort 1 PortName       "scemi_dutin_inport"
MessageInPort 1 PortWidth      64
MessageInPort 1 ChannelId      2
MessageInPort 1 Type           "UInt#(64)"

MessageInPort 0 TransactorName ""
MessageInPort 0 PortName       "scemi_dut_prb_control_control_in"
MessageInPort 0 PortWidth      17
MessageInPort 0 ChannelId      1
MessageInPort 0 Type           "SceMiSerialProbe::ProbeControl"

Link 0 LinkType   "TCP"
Link 0 TCPAddress "127.0.0.1"
Link 0 TCPPort    7381
```

A.1.5 C++ Testbench

```
using namespace std;
#include "bsv_scemi.h"

// --------------------
```

```
int main (int argc, char *argv[]) {

  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );

  // Control
  ShutdownXactor shutdown ("", "scemi_shutdown", sceMi);

  // Initialize the SceMi inport
  InportProxyT<BitT<64> > in_proxy ("", "scemi_dutin_inport", sceMi);

  // Initialize the SceMi outport
  OutportQueueT<BitT<64> > out_proxy ("", "scemi_dutout_outport", sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  for (SceMiU32 i=0; i<10; i=i+1) {
    in_proxy.sendMessage(i);
    cout << "Requested factorial: " << i << endl;
  }
  for (SceMiU32 i=0; i<10; i=i+1) {
    BitT<64> fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
  }

  // Shutdown the simulation
  shutdown.blocking_send_finish();

  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);
  delete params;
}
// --------------------
```

A.1.6 Build File (project.bld)

```
[DEFAULT]
default-targets:   bsim_dut tb

[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build

[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut

[vlog_dut]
extends-target: dut
```

```
build-for:      verilog
scemi-type:     TCP
exe-file:       vlog_dut

[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp
exe-file: tb

[clean]
run-shell: rm -rf build
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log
run-shell: rm -f vlog_dut directc_* vlog_dut*.log
run-shell: rm -f tb *.params tb*.log
```

A.2 Example 2: Replacing hardware-side transactors

The source files for the server factorial example are found in the directory `example2`.

A.2.1 SceMiLayer

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;

import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

   // Connect the dut interface to SceMi
   Empty dutconnect <- mkServerXactor(dut, clk_port);

   Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

A.2.2 C++ Testbench

```
using namespace std;
#include "bsv_scemi.h"
```

```
// --------------------
int main (int argc, char *argv[]) {

  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );

  // Control
  ShutdownXactor shutdown ("", "scemi_shutdown", sceMi);

  // Initialize the SceMi inport
  InportProxyT<BitT<64> > in_proxy ("", "scemi_dutconnect_req_inport", sceMi);

  // Initialize the SceMi outport
  OutportQueueT<BitT<64> > out_proxy ("", "scemi_dutconnect_resp_outport", sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  for (SceMiU32 i=0; i<10; i=i+1) {
    in_proxy.sendMessage(i);
    cout << "Requested factorial: " << i << endl;
  }
  for (SceMiU32 i=0; i<10; i=i+1) {
    BitT<64> fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
  }

  // Shutdown the simulation
  shutdown.blocking_send_finish();

  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);
  delete params;
}
// --------------------
```

A.3 Example 3: Generating data types

The source files for the factorial example with structures and advanced data types are found in the directory `example3`.

A.3.1 DUT

```
package DUT;

import GetPut::*;
import FIFO::*;
import ClientServer::*;

typedef struct {UInt#(64) value_in;
                Maybe#(UInt#(64)) factorial;
               } Factresp deriving (Bits);

(* synthesize *)
```

```
module [Module] mkFactorialServer (Server#(UInt#(64), Factresp));
   FIFO#(UInt#(64)) req_fifo <- mkFIFO();
   FIFO#(Factresp) resp_fifo <- mkFIFO();

   Reg#(Bool) started <- mkReg(False);
   Reg#(UInt#(64)) next_count <- mkReg(0);
   Reg#(Maybe#(UInt#(64))) curr_result <- mkRegU();
   Reg#(UInt#(64)) next_result <- mkRegU();
   Reg#(Factresp) fact_response <- mkRegU();

   rule start_computation (!started);
      // Get the value
      let val = req_fifo.first();
      req_fifo.deq();
      // Set up the computation state
      curr_result <= tagged Valid 1;
      fact_response.value_in <= val;
      next_count <= val;
      started <= True;
      $display("Starting request: %h", val);
   endrule

   rule do_computation (started && (next_count > 0));
      if (curr_result matches tagged Valid .curr)
          next_result <= curr * next_count;
      // handle overflow
      if (next_result == 0)  begin
 $display("Overflow!");
         curr_result <= tagged Invalid;
         next_count <= 0;
      end
      else begin
         curr_result <= tagged Valid next_result;
         next_count <= next_count - 1;
      end
   endrule

   rule finish_computation (started && (next_count == 0));
      started <= False;
      fact_response.factorial <= curr_result;
      resp_fifo.enq(fact_response);
      $display("Sending result: %h", curr_result);
   endrule

   interface Put request = toPut(req_fifo);
   interface Get response = toGet(resp_fifo);
endmodule

endpackage
```

A.3.2 SceMiLayer

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;
```

```
import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

   // Connect the dut interface to SceMi
   Empty dutconnect <- mkServerXactor(dut, clk_port);

   Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

A.3.3 C++ Testbench

```
using namespace std;

#include "bsv_scemi.h"

// --------------------

// Locally generated C++ equivalents for BSV types
#include "SceMiHeaders.h"


// --------------------

int main (int argc, char *argv[]) {

  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );

  // Control
  ShutdownXactor shutdown ("", "scemi_shutdown", sceMi);

  // Initialize the SceMi inport
  InportProxyT<BitT<64> > in_proxy ("", "scemi_dutconnect_req_inport", sceMi);

  // Initialize the SceMi outport
  OutportQueueT<Factresp > out_proxy ("", "scemi_dutconnect_resp_outport", sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  for (SceMiU32 i=0; i<10; i=i+1) {
    in_proxy.sendMessage(i);
    cout << "Requested factorial: " << i << endl;
  }
  for (SceMiU32 i=0; i<10; i=i+1) {
```

```
    Factresp fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
  }

  // Shutdown the simulation
  shutdown.blocking_send_finish();

  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);
  delete params;

}

// --------------------
```

A.3.4 Build File (project.bld)

```
[DEFAULT]
default-targets:   bsim_dut tb

[paths]
hide-target
verilog-directory:    build
binary-directory:     build
simulation-directory: build

[dut]
hide-target
extends-target:       paths
top-file:             Bridge.bsv
top-module:           mkBridge
scemi-tcp-port:       7500


[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       vlog_dut

[tb]
extends-target:     dut
scemi-tb
build-for: c++
c++-files: Tb.cpp
exe-file: tb
c++-header-targets: outputs

[clean]
run-shell: rm -rf build
```

```
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log
run-shell: rm -f vlog_dut directc_* vlog_dut*.log
run-shell: rm -f tb xsb.params tb*.log
run-shell: rm -f sw/MemRequest.h sw/SceMiHeaders.h
```

A.4 Example 4: A Single Clock Port

This DUT is used for examples 4, 5, and 6.

The source files for the single clock example are found in the directory `example4`.

A.4.1 SceMiLayer - Single Clock Port

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;
import Connectable::*;

import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors

   SceMiClockConfiguration conf = defaultValue;
   conf.clockNum = 0;
   conf.dutyHi = 50;
   conf.dutyLo = 50;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Access the uncontrolled clock and reset
   Clock uclock <- sceMiGetUClock;
   Reset ureset <- sceMiGetUReset;

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

   // Connect the dut to SceMi
   Empty dutconnect <- mkServerXactor(dut, clk_port);

   Reg#(UInt#(16)) uclock_count <- mkReg(0, clocked_by uclock, reset_by ureset);

   Reg#(Bool) clock_enable <-  mkReg(True, clocked_by uclock, reset_by ureset);

   Bool allow_clock = clock_enable ;

   // Instantiate a controller for the controlled clock
   // so that we can stall the clock while the port is busy
   SceMiClockControlIfc clkcntrl_fast <- mkSceMiClockControl(conf.clockNum,
                                                             allow_clock,
                                                             allow_clock);
   rule control_clock;
```

```
      uclock_count <= uclock_count + 1;
      if (uclock_count == 2 || uclock_count == 5 ||
          uclock_count == 10 || uclock_count == 15 ||
  uclock_count == 20  || uclock_count == 25)
 clock_enable <= !clock_enable;
   endrule

   Empty shutdown <- mkShutdownXactor();

endmodule
endpackage
```

A.5 Example 5: Multiple Clock Ports

The source files for the two clock port example are found in the directory `example5`.

A.5.1 SceMiLayer - Two Clock Ports

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;
import Connectable::*;

import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   // Fast clock (same as uclock and don't care duty cycle)
   SceMiClockConfiguration conf_fast = defaultValue();
   conf_fast.clockNum = 0;
   conf_fast.dutyHi = 50;
   conf_fast.dutyLo = 50;
   SceMiClockPortIfc clkport_fast <- mkSceMiClockPort(conf_fast);

   //Slow clock (divide by 4 and 50% duty cycle)
   SceMiClockConfiguration conf_slow = defaultValue();
   conf_slow.clockNum = 1;
   conf_slow.ratioNumerator = 4;
   conf_slow.dutyHi = 50;
   conf_slow.dutyLo = 50;
   SceMiClockPortIfc clkport_slow <- mkSceMiClockPort(conf_slow);

   // Access the uncontrolled clock and reset
   Clock uclock <- sceMiGetUClock;
   Reset ureset <- sceMiGetUReset;

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clkport_fast);

   // Connect the dut to SceMi
   Empty dutconnect <- mkServerXactor(dut, clkport_fast);
```

```
   Reg#(UInt#(16)) uclock_count <- mkReg(0, clocked_by uclock, reset_by ureset);

   Reg#(Bool) clock_enable <-  mkReg(True, clocked_by uclock, reset_by ureset);

   Bool allow_clockfast = True ;
   Bool allow_clockslow = clock_enable ;

   // Instantiate a controller for the controlled clock
   // so that we can stall the clock while the port is busy
   SceMiClockControlIfc clkcntrl_fast <- mkSceMiClockControl(conf_fast.clockNum, allow_clockfast, allow_clockfa

   SceMiClockControlIfc clkcntrl_slow <- mkSceMiClockControl(conf_slow.clockNum, allow_clockslow, allow_clocksl

   rule control_clock;
      uclock_count <= uclock_count + 1;
      if (uclock_count == 2 || uclock_count == 5 ||
          uclock_count == 10 || uclock_count == 15 ||
  uclock_count == 20  || uclock_count == 25)
 clock_enable <= !clock_enable;
   endrule

   Empty shutdown <- mkShutdownXactor();

endmodule
endpackage
```

A.6 Example 6: Writing hardware-side Transactors

The source files for the writing hardware-side transactors example are found in the directory `example6`

A.6.1 ClockedServerXactor

```
package ClockedServerXactor;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import AlignedFIFOs::*;
import ClientServer::*;
import Connectable::*;
import Clocks::*;

module [SceMiModule] mkClockedServerXactor
                        #( parameter SceMiClockConfiguration cclk_conf,
                           SceMiClockPortIfc clkport)
                         ( Client#(req_ty, resp_ty))
   provisos (Bits#(req_ty, req_ty_sz),
             Bits#(resp_ty, resp_ty_sz)) ;

   // Access the controlled clock and reset
   Clock cclock = clkport.cclock;
   Reset creset = clkport.creset;

   // Access the uncontrolled clock and reset
   Clock uclock <- sceMiGetUClock;
   Reset ureset <- sceMiGetUReset;
```

```
    // The input port
    SceMiMessageInPortIfc#(req_ty) inport <- mkSceMiMessageInPort();

    // Use a SyncFIFO to cross the data into
    // the controlled domain, to provide the signal for stalling the
    // controlled clock, and to store the data until it can be taken
    SyncFIFOIfc#(req_ty) res_fifo <- mkSyncFIFO(2, uclock, ureset, cclock);

    // The output port
    SceMiMessageOutPortIfc#(resp_ty) outport <- mkSceMiMessageOutPort();

    // Use the SyncFIFO to cross the data into the uncontrolled domain, to
    // provide the signal for stalling the controlled clock, and to store
    // the data until it can be served.
    SyncFIFOIfc#(resp_ty) out_fifo <- mkSyncFIFO(2, cclock, creset, uclock);

    PulseWire started_req <- mkPulseWire(clocked_by uclock, reset_by ureset);
    PulseWire finished_req <- mkPulseWire(clocked_by uclock, reset_by ureset);
    Reg#(UInt#(16)) outstanding_reqs <- mkReg(0, clocked_by uclock, reset_by ureset);
    Reg#(UInt#(16)) uclock_count <- mkReg(0, clocked_by uclock, reset_by ureset);

    Bool allow_clock = (outstanding_reqs != 0);

    // Instantiate a controller for the controlled clock
    // so that we can stall the clock while the port is busy
    SceMiClockControlIfc clk_cntrl <- mkSceMiClockControl(cclk_conf.clockNum, allow_clock, allow_clock);

     rule count_reqs;
       if (started_req && !finished_req)
          outstanding_reqs <= outstanding_reqs + 1;
       else if (finished_req && !started_req)
          outstanding_reqs <= outstanding_reqs - 1;
    endrule: count_reqs

    rule in_request;
       inport.request();
    endrule

    rule req_from_tb;
       let x <- toGet(inport).get;
       res_fifo.enq(x);
       started_req.send();
       $display("count in: %d", outstanding_reqs);
    endrule: req_from_tb

    rule resp_to_tb;
       outport.send(out_fifo.first());
       out_fifo.deq();
       $display("count out: %d", outstanding_reqs);
       finished_req.send();
    endrule: resp_to_tb

    interface Get request = toGet(res_fifo);
    interface Put response = toPut(out_fifo);
endmodule
endpackage
```

A.6.2 SceMiLayer - calling Transactor

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;
import Connectable::*;

import ClockedServerXactor::*;
import DUT::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;
typedef Client#(UInt#(64), UInt#(64)) ClientIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);

   // Instantiate the SCE-MI transactor
   ClientIfc xactor <- mkClockedServerXactor(conf, clk_port);

   // Connect the DUT to the transactor
   mkConnection(dut.request, xactor.request);
   mkConnection(dut.response, xactor.response);

   Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

A.6.3 C++ Testbench

Note the new names for message ports

```
using namespace std;

#include "bsv_scemi.h"

// --------------------

int main (int argc, char *argv[]) {

  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );

  // Control
   ShutdownXactor shutdown ("", "scemi_shutdown", sceMi);

  // Initialize the SceMi inport
  InportProxyT<BitT<64> > in_proxy ("", "scemi_xactor_inport", sceMi);
```

```
  // Initialize the SceMi outport
  OutportQueueT<BitT<64> > out_proxy ("", "scemi_xactor_outport", sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  for (SceMiU32 i=0; i<10; i=i+1) {
    in_proxy.sendMessage(i);
    cout << "Requested factorial: " << i << endl;
  }

  for (SceMiU32 i=0; i<10; i=i+1) {

    BitT<64> fact = out_proxy.getMessage();
    cout << "Calculated factorial received:" << fact <<endl;
  }


  // Shutdown the simulation
  shutdown.blocking_send_finish();

  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);
  delete params;

}
// --------------------
```

A.7 Example 7: Writing Software Transactors in C++

The source files for this section are found in the directory `example7`.

A.7.1 FactXactor.h

```
// Copyright Bluespec Inc. 2009‘-2010


#pragma once

// Include Bluespec’s SceMi C++ api
#include "bsv_scemi.h"

// Define a class for the top-level transactor
class FactXactor {
 protected:
  // Data members include transactors contained in the model
  // Data Xactors
  InportProxyT<BitT<64> >  m_datain;
  OutportQueueT<BitT<64 > >  m_dataout;
  // Shutdown Xactor
  ShutdownXactor  m_shutdown;

 public:
  // Constructor
  FactXactor (SceMi *scemi) ;
```

```
  // Destructor
  ~ FactXactor();

  // Public interface .....
  bool putRequestNB(long);
  bool putRequestB (long);
  bool getResponseNB (long &);
  bool getResponseB (long &);

  void shutdown();

 private:
  // Private helper members
};
```

A.7.2 FactXactor.cpp

```
// Copyright Bluespec Inc. 2009-2010


#include <iostream>
#include "FactXactor.h"

using namespace std;

// Constructor definition
FactXactor::FactXactor(SceMi *scemi)
  : m_datain  ("", "scemi_dutconnect_req_inport", scemi)
  , m_dataout ("", "scemi_dutconnect_resp_outport", scemi)
  , m_shutdown ("", "scemi_shutdown", scemi)
{

}

// Destructor definition
FactXactor::~FactXactor()
{
}


// Method definitions
// shutdown
void FactXactor::shutdown()
{
  m_shutdown.blocking_send_finish();
}

// Non-blocking put request
// Send the request to the DUT
bool FactXactor::putRequestNB (long request)
{
  BitT<64> reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}

// Blocking put request
bool FactXactor::putRequestB (long request)
```

```
{
  BitT<64> reqbit(request);
  m_datain.sendMessage(reqbit);
  return true;
}
// Non-blocking get response
// return true and populates resp if a response is available
// otherwise returns false
bool FactXactor::getResponseNB (long &resp)
{
  BitT<64> respbit;
  bool gotone = m_dataout.getMessageNonBlocking (respbit) ;
  if (gotone) {
    resp = respbit.get64();
  }
  return gotone;
}

// Blocking get response
bool FactXactor::getResponseB (long &resp)
{
  BitT<64> respbit = m_dataout.getMessage () ;
  resp = respbit.get64();
  return true;
}
```

A.7.3 Tb.cpp

```
using namespace std;

#include "bsv_scemi.h"

// c++ files needed for design
#include "FactXactor.h"

// --------------------

void runFactorialTest (class FactXactor &xactor);

int main (int argc, char *argv[]) {

  // Initialize SceMi
  int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
  SceMiParameters *params = new SceMiParameters( "mkBridge.params" );
  SceMi *sceMi = SceMi::Init( sceMiVersion, params );

  // Create our transactor
  FactXactor factx(sceMi);

  // Service SceMi requests
  SceMiServiceThread scemi_service_thread (sceMi);

  // Run tests with this xactor
  runFactorialTest (factx) ;


  //Stop the simulation side
  factx.shutdown();
```

```
  // Stop and join with the service thread, then shut down scemi --
  scemi_service_thread.stop();
  scemi_service_thread.join();
  SceMi::Shutdown(sceMi);

}


void runFactorialTest ( class FactXactor &xactor)
{
  long resp;
  //  while ( ! xactor.putRequestNB(1)) {} ;
  //  cout << "Sent: 1" << endl;

//   // wait for 2 data from dut -- polling and thread blocking
//   while (! xactor.getResponseNB(resp) );
//   cout << dec << "Received: " << resp << endl;

//  xactor.getResponseB(resp);
//  cout << dec << "Received: " << resp << endl;

  for (int i = 0; i < 10; ++i) {
    xactor.putRequestB(i);
    cout << dec << "Sent: " << i << endl;
  }

  for (int i = 0; i < 10; ++i) {
    xactor.getResponseB(resp);
    cout << dec << "Received: (" << i << ") " << resp << endl;


   }
}
// --------------------
```

A.8 Example 8: Writing a GUI-based Testbench

The source files for this section are found in the directory `example8`.

A.8.1 TclTb.cpp

```
// Copyright Bluespec Inc. 2009-2010

#include <iostream>
#include <stdexcept>
#include <string>
#include <cstdlib>
#include <cstring>

#include <pthread.h>

// Bluespec's version -- $BLUESPECDIR/tcllib/include
#include "tcl.h"

// Put c++ files needed for design here
```

```
#include "FactXactor.h"
#include "SceMiHeaders.h"

// Bluespec common code
#include "bsdebug_common.h"


using namespace std;

// the package name and namespace for this extension

#define PKG_NAME    "BSDebug"
#define NS          "bsdebug"
#define PKG_VERSION "1.0"

// static extension global data
class SceMiGlobalData {
public:
  bool m_initialized ;
  SceMi                 * m_scemi;
  FactXactor            * m_factX;
  SceMiServiceThread    * m_serviceThread;
  SimulationControl  * m_simControl;
  ProbesXactor          * m_probeControl;

  // Simple initializer invoked when the extension is loaded
  SceMiGlobalData ()
    : m_initialized(false)
    , m_scemi(0)
    , m_factX(0)
    , m_serviceThread(0)
    , m_simControl(0)
    , m_probeControl(0)
  {}

  ~SceMiGlobalData ()
  {
    if (m_initialized) {
      destroy();
    }
  }

  // Initialization -- call from bsdebug::scemi init <param>
  void init (const char *paramfile) {

    if (m_initialized) throw std::runtime_error ("scemi is already initialized");

     // Initialize SceMi
    int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
    SceMiParameters params( paramfile );
    m_scemi = SceMi::Init( sceMiVersion, & params );
    if (! m_scemi) throw std::runtime_error ("Could not initialize SceMi");

    // initialize ProbesXactor          * probeControl;
    m_probeControl = ProbesXactor::init("", "scemi_dut_prb_control", "scemi_test.vcd", m_scemi);

    /* initiate the SCE-MI transactors and threads */
    // Create the transactor
    m_factX = new FactXactor (m_scemi);
```

```
    m_simControl  = new SimulationControl ("", "scemi_simControl" ,m_scemi);

    // Start a SceMiService thread;
    m_serviceThread = new SceMiServiceThread  (m_scemi);

    m_initialized = true ;
  }

  // Destruction -- called from bsdebug::scemi delete
  void destroy () {
    m_initialized = false ;
    // Stop the simulation side
    // if (m_inport) m_inport->shutdown();
    // if (m_outport) m_outport->shutdown();
    if (m_factX) m_factX -> shutdown();


    // Stop and join with the service thread, then shut down scemi --
    if (m_serviceThread) {
      m_serviceThread->stop();
      m_serviceThread->join();
      delete m_serviceThread;  m_serviceThread = 0;
    }

    // Delete the simulation control
    delete m_simControl; m_simControl = 0;

    //Delete the Dut transactor
    delete m_factX; m_factX = 0;

    // Shutdown the probes transactor
    ProbesXactor::shutdown();

    // Shutdown SceMi
    if (m_scemi) {
      SceMi::Shutdown(m_scemi);
      m_scemi = 0;
    }
  }

} SceMiGlobal;




// forward declarations of C functions which are called by tcl
extern "C" {

  // Package intialization  and cleanup
  extern int Bsdebug_Init (Tcl_Interp * interp);
  extern int Bsdebug_Unload (Tcl_Interp * interp,  int flags);
  extern void Bsdebug_ExitHandler (ClientData clientData);

  static int SceMi_Cmd(ClientData clientData,
                       Tcl_Interp *interp,
                       int objc,
                       Tcl_Obj *const objv[]);
  static void SceMi_Cmd_Delete(ClientData clientData);
```

```
  static int Dut_Cmd(ClientData clientData,
                     Tcl_Interp *interp,
                     int objc,
                     Tcl_Obj *const objv[]);
  static void Dut_Cmd_Delete(ClientData clientData);

} // extern "C"



// Function called if/when the dynamic library is unloaded
// Function name must match package library name
int Bsdebug_Unload (Tcl_Interp * interp,  int flags)
{
  if (flags & TCL_UNLOAD_DETACH_FROM_PROCESS) {
    SceMiGlobalData *pglobal = & SceMiGlobal;
    pglobal->destroy();
    Tcl_DeleteExitHandler ( Bsdebug_ExitHandler, &SceMiGlobal);
  }
  return TCL_OK;
}

// Exit handler called during exit.
void Bsdebug_ExitHandler (ClientData clientData)
{
  SceMiGlobalData *pglobal = (SceMiGlobalData *) clientData;
  pglobal->destroy();
}

// Package initialization function -- called during package load/require
// function name must match package library name
int Bsdebug_Init(Tcl_Interp *interp)
{
  Tcl_Namespace* nsptr = NULL;

  try {
    // register the exit handler
    Tcl_CreateExitHandler( Bsdebug_ExitHandler, &SceMiGlobal);

    // Dynmaic binding of this extension to tcl
    if (Tcl_InitStubs(interp, TCL_VERSION, 0) == NULL) {
      return TCL_ERROR;
    }

    // Create a namespace NS
    nsptr = (Tcl_Namespace*) Tcl_CreateNamespace(interp, NS, NULL, NULL);
    if (nsptr == NULL) {
      return TCL_ERROR;
    }

    // Provide the tcl package
    if (Tcl_PkgProvide(interp, PKG_NAME, PKG_VERSION) != TCL_OK) {
      return TCL_ERROR;
    }

    // Register commands to this tcl extension
    // A top-level tcl bsdebug::scemi command -- application specific boilerplate
```

```
    Tcl_CreateObjCommand(interp,
 NS "::scemi",
 (Tcl_ObjCmdProc *) SceMi_Cmd,
 (ClientData) &(SceMiGlobal),
 (Tcl_CmdDeleteProc *) SceMi_Cmd_Delete);
    Tcl_Export(interp, nsptr, "scemi", 0);

    // A top-level tcl dut command -- application specific
    Tcl_CreateObjCommand(interp,
 NS "::dut",
 (Tcl_ObjCmdProc *) Dut_Cmd,
 (ClientData) &(SceMiGlobal.m_factX),
 (Tcl_CmdDeleteProc *) Dut_Cmd_Delete);
    Tcl_Export(interp, nsptr, "dut", 0);

    // Bluespec emulation control command
    Tcl_CreateObjCommand(interp,
                         NS "::emu",
                         (Tcl_ObjCmdProc *) Emu_Cmd,
                         (ClientData) &(SceMiGlobal.m_simControl),
                         (Tcl_CmdDeleteProc *) Emu_Cmd_Delete);
    Tcl_Export(interp, nsptr, "emu", 0);

    // Bluespec probe capture command
    Tcl_CreateObjCommand(interp,
                         NS "::probe",
                         (Tcl_ObjCmdProc *) Capture_Cmd,
                         (ClientData) &(SceMiGlobal.m_probeControl),
                         (Tcl_CmdDeleteProc *) Capture_Cmd_Delete);
    Tcl_Export(interp, nsptr, "probe", 0);

    // Other command can go here

  } catch (const exception & error) {
    Tcl_AppendResult(interp, error.what()
                     ,"\nCould not initialize bsdebug tcl package"
                     ,(char *) NULL);
    return TCL_ERROR;
  }

  return TCL_OK;
}




// implementation of the scemi command ensemble
// at the tcl level, the command will be
// bsdebug::scemi init <params file>
// bsdebug::scemi delete
static int SceMi_Cmd(ClientData clientData,     //  &(GlobalXactor),
                     Tcl_Interp *interp,       // Current interpreter
                     int objc,                // Number of arguments
                     Tcl_Obj *const objv[]    // Argument strings
        )
{
  // Command table
  enum ScemiCmds { scemi_init, scemi_delete };
  static const cmd_struct cmds_str[] = {
```

```
    {"init",scemi_init,"<params file>"}
    ,{"delete",scemi_delete,""}
    ,{0}                        // MUST BE LAST
  };

  // Cast client data to proper type
  SceMiGlobalData *pglobal = (SceMiGlobalData *) clientData;

  // Extract sub command
  ScemiCmds command;
  int index;
  if (objc == 1) goto wrongArgs;
  if (TCL_OK != Tcl_GetIndexFromObjStruct (interp, objv[1], cmds_str, sizeof(cmd_struct),
                                           "command", 0, &index ) ) {
    return TCL_ERROR;
  }


  command = (enum ScemiCmds) cmds_str[index].enumcode;
  switch (command) {
    case scemi_init:
    {
if (objc != 3) goto wrongArgs;
char *paramfile = Tcl_GetString(objv[2]);
try {
          pglobal->init(paramfile);
} catch (const exception & error) {
          Tcl_AppendResult(interp, error.what()
                           ,"\nCould not initialize emulation"
                           ,(char *) NULL);
  return TCL_ERROR;
}
      break;
    }
    case scemi_delete:
        pglobal->destroy();
        break;
  }
  return TCL_OK;

wrongArgs:
  dumpArguments (interp, cmds_str, Tcl_GetString(objv[0]));
  return TCL_ERROR;
}

static void SceMi_Cmd_Delete(ClientData clientData)
{
}



// implementation of the Dut command ensemble
// dut request <int>
// dut response
static int Dut_Cmd(ClientData clientData,     //  &(GlobalXactor.m_pipelineX)
                   Tcl_Interp *interp,      // Current interpreter
                   int objc,                // Number of arguments
                   Tcl_Obj *const objv[]    // Argument strings
         )
```

```
{
  // Command table
  enum DutCmds { Dut_Request, Dut_Response };
  static const cmd_struct cmds_str[] = {
    {"request",Dut_Request,"int"}
    ,{"response",Dut_Response,""}
    ,{0}                        // MUST BE LAST
  };

  // Check that client data has been set
  FactXactor *factx = *(FactXactor **) clientData;
  if (factx == 0) {
    Tcl_SetResult (interp, (char *) "Cannot use dut command before emulation initialization", TCL_STATIC );
    return TCL_ERROR;
  }

  // Extract sub command
  DutCmds command;
  int index;
  if (objc == 1) goto wrongArgs;
  if (TCL_OK != Tcl_GetIndexFromObjStruct (interp, objv[1], cmds_str, sizeof(cmd_struct),
                                           "command", 0, &index ) ) {
    return TCL_ERROR;
  }
  command = (enum DutCmds) cmds_str[index].enumcode;

  switch (command) {
    case Dut_Request:  // request x
      {
if (objc != 3) goto wrongArgs;
        long a1;
        if (Tcl_GetLongFromObj(interp, objv[2], &a1) != TCL_OK) {
          return TCL_ERROR;
        }

        bool sent = factx->putRequestNB(a1);
        Tcl_Obj *r = Tcl_NewBooleanObj( sent );
        Tcl_SetObjResult(interp, r );
break;
      }
    case Dut_Response:
      {
if (objc != 2) goto wrongArgs;
        long resp;
        bool gotit = factx->getResponseNB(resp);
        if (gotit) {
          // Convert result to a object and return
  ostringstream os;
  os << resp;
          Tcl_Obj *r = Tcl_NewStringObj( os.str().c_str(), os.str().size());
          Tcl_SetObjResult(interp, r );
        } else {
          Tcl_Obj *r = Tcl_NewStringObj( "", 0);
          Tcl_SetObjResult(interp, r );
        }
break;
      }
  }
  return TCL_OK;
```

```
wrongArgs:
  dumpArguments (interp, cmds_str, Tcl_GetString(objv[0]));
  return TCL_ERROR;
}

static void Dut_Cmd_Delete(ClientData clientData)
{
}
```

A.8.2 gui_dut.tcl

```
package require Tk
namespace import bsdebug::dut

namespace eval GuiDut {
    variable dut
    variable e1 1
    variable textbox
    variable updatetime 500

    proc mkDutControl { frame } {
        variable top
        variable putbut
        variable getbut
        variable textbox

        set top [ttk::labelframe $frame.dut_pane -text "Dut Control" -relief ridge]

        ## Button Frame
        set button_frame [ttk::frame $top.button_frame]

        set e1 [ttk::entry $button_frame.e1 -textvariable GuiDut::e1 -width 8 -validate key -validatecommand "G
        set putbut [ttk::button $button_frame.put -text "Send" -command "GuiDut::do_send"]

        set getbut [ttk::button $button_frame.get -text "Get " -command "GuiDut::do_get"]

        pack $e1      -padx 5 -pady 3 -side left -anchor w
        pack $putbut  -pady 3 -pady 3 -side left -anchor w
        pack $getbut  -pady 3 -pady 3 -side left -anchor w

        pack $button_frame -pady 3 -pady 3 -side left -anchor w

        ## Message box and scroll bar
        set tb [ttk::labelframe $top.status_frame -text "Messages" ]
        set textbox [text $tb.text -width 90 -height 20 -yscrollcommand "$tb.scroll set"]
        set scroll [ttk::scrollbar $tb.scroll -orient vertical -command "$textbox yview"]

        grid $textbox -row 0 -column 0 -sticky nsew
        grid $scroll -row 0 -column 1 -sticky ns
        grid columnconfigure $tb 0 -weight 1
        grid columnconfigure $tb 1 -weight 0
        grid rowconfigure    $tb 0 -weight 1

        pack $button_frame -anchor n -side top
        pack $tb -expand 1 -fill both -anchor n
```

```
        return $top
    }

    proc do_send {} {
        variable textbox
        variable e1
        set msg "Blocked"

        set sent [dut request $e1]
        if {$sent} { set msg "sending $e1" }

        incr e1 1

        $textbox insert end "$msg\n"
        $textbox yview end
    }

    proc do_get {} {
        variable textbox

        set res [dut response]
        if { $res == "" } {
            set msg "Nothing ready"
        } else {
            set msg "Received: $res"
        }
        $textbox insert end "$msg\n"
        $textbox yview end
    }

# ======
    proc getLoop {} {
        if { [catch getLoopInternal err] } {
            puts stderr "Status loop failed,  interface will not respond"
            puts stderr $err
        }
    }
    # Loop watching status
    proc getLoopInternal {} {
        variable textbox
        variable updatetime

        set res [dut response]
        while { $res != "" } {
    $textbox insert end "$res\n"
    $textbox yview end
            set res [dut response]
}
after $updatetime GuiDut::getLoop
        update
    }

# ======

    proc chk_num {s} {
        string is integer $s
    }

}
```

A.8.3 project.bld

```
[DEFAULT]
default-targets:   vlog_dut tcl_tb

[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build

[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       dut.vexe

run-design-editor: True
design-editor-output-directory: vlog_edited
design-editor-edit-params: True
design-editor-options:  --batch

[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp FactXactor.cpp
exe-file: tb

[tcl_tb]
extends-target: dut
exe-file: cpp_tb
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: generated_c
c++-source-directory: .
c++-header-aliases
c++-options: -g -O0
shared-lib: libbsdebug.so
c++-files: TclTb.cpp FactXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp

##########################################################################
# since xupv5 is so odd to remember ;)
[fpga_dut]
extends-target: dut
build-for:      xupv5
scemi-type:     PCIE_VIRTEX5

xilinx-directory:         xilinx
bsc-compile-options:      -remove-dollar -verilog-filter ${BLUESPECDIR}/bin/basicinout
bsc-compile-options:      -unspecified-to 0 -opt-undetermined-vals
```

```
bsc-compile-options:      -p +:${BLUESPECDIR}/board_support/bridges
board-support-directory:  ${BLUESPECDIR}/board_support

xilinx-xst-options:    -intstyle ise
xilinx-map-options:    -w -pr b -ol high -timing -logic_opt on -register_duplication -xe n -lc auto

# probably should add a few probes :)
run-design-editor: True
design-editor-edit-params: True
design-editor-options: --batch
design-editor-output-params: mkBridge_EDITED.params
design-editor-output-directory: vlog_edited

########################################################################
[ml605_dut]
extends-target: dut
build-for:      ml605
scemi-type:     PCIE_VIRTEX6

xilinx-directory:         xilinx
bsc-compile-options:      -remove-dollar -verilog-filter ${BLUESPECDIR}/bin/basicinout
bsc-compile-options:      -unspecified-to 0 -opt-undetermined-vals
bsc-compile-options:      -p +:${BLUESPECDIR}/board_support/bridges
board-support-directory:  ${BLUESPECDIR}/board_support

# probably should add a few probes :)
run-design-editor: True
design-editor-edit-params: True
design-editor-options: --batch
design-editor-output-params: mkBridge_EDITED.params
design-editor-output-directory: vlog_edited

########################################################################
[clean]
run-shell: rm -rf build
run-shell: rm -fR bsim_dut bsim_dut.so bsim_dut*.log vlog_edited transcript xilinx
run-shell: rm -fR vlog_dut directc_* vlog_dut*.log *.log work_* dut.vexe generated_c
run-shell: rm -f tb *.params tb*.log scemi_test.vcd*
```

A.9 Example 9: Adding Probes for Debugging

The source files for this section are found in the directory `example9`. Except for the `project.bld` file, all the files are the same as those used in Example 8. When you run the example, you will use the HDL editor to add probes.

A.9.1 project.bld

```
[DEFAULT]
default-targets:   vlog_dut tcl_tb

[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
```

```
simulation-directory: build

[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       dut.vexe

run-design-editor: True
design-editor-output-directory: vlog_edited
design-editor-edit-params: True
design-editor-options:  --gui

[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp FactXactor.cpp
exe-file: tb

[tcl_tb]
extends-target: dut
exe-file: cpp_tb
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: generated_c
c++-source-directory: .
c++-header-aliases
c++-options: -g -O0
shared-lib: libbsdebug.so
c++-files: TclTb.cpp FactXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp


[clean]
run-shell: rm -rf build
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log
run-shell: rm -f vlog_dut directc_* vlog_dut*.log
run-shell: rm -f tb *.params tb*.log
```

A.10 Example 10: Wrapping a Verilog Dut

The source files for this section are found in the directory `example10`.

A.10.1 FactorialServer.v

```
//
// Generated by Bluespec Compiler, version 2011.01.beta1 (build 23181, 2011-01-27)
//
// On Wed Feb  9 12:12:17 EST 2011
```

```
//
// Method conflict info:
// Method: request_put
// Conflict-free: response_get
// Conflicts: request_put
//
// Method: response_get
// Conflict-free: request_put
// Conflicts: response_get
//
//
// Ports:
// Name                         I/O  size props
// RDY_request_put               O     1 reg
// response_get                  O    64 reg
// RDY_response_get              O     1 reg
// CLK                           I     1 clock
// RST_N                         I     1 reset
// request_put                   I    64 reg
// EN_request_put                I     1
// EN_response_get               I     1
//
// No combinational paths from inputs to outputs
//
//

`ifdef BSV_ASSIGNMENT_DELAY
`else
`define BSV_ASSIGNMENT_DELAY
`endif

module mkFactorialServer(CLK,
 RST_N,

 request,
 EN_request,
 RDY_request,

 EN_response,
 response,
 RDY_response);
  input  CLK;
  input  RST_N;

  // action method request
  input  [63 : 0] request;
  input  EN_request;
  output RDY_request;

  // actionvalue method response
  input  EN_response;
  output [63 : 0] response;
  output RDY_response;

  // signals for module outputs
  wire [63 : 0] response;
  wire RDY_request, RDY_response;

  // register curr_result
```

```
reg [63 : 0] curr_result;
wire [63 : 0] curr_result$D_IN;
wire curr_result$EN;

// register next_count
reg [63 : 0] next_count;
wire [63 : 0] next_count$D_IN;
wire next_count$EN;

// register started
reg started;
wire started$D_IN, started$EN;

// ports of submodule req_fifo
wire [63 : 0] req_fifo$D_IN, req_fifo$D_OUT;
wire req_fifo$CLR,
     req_fifo$DEQ,
     req_fifo$EMPTY_N,
     req_fifo$ENQ,
     req_fifo$FULL_N;

// ports of submodule resp_fifo
wire [63 : 0] resp_fifo$D_IN, resp_fifo$D_OUT;
wire resp_fifo$CLR,
     resp_fifo$DEQ,
     resp_fifo$EMPTY_N,
     resp_fifo$ENQ,
     resp_fifo$FULL_N;

// rule scheduling signals
wire WILL_FIRE_RL_do_computation,
     WILL_FIRE_RL_finish_computation,
     WILL_FIRE_RL_start_computation;

// inputs to muxes for submodule ports
wire [63 : 0] MUX_next_count$write_1__VAL_1;

// remaining internal signals
wire [127 : 0] curr_result_1_MUL_next_count___d12;

// action method request
assign RDY_request = req_fifo$FULL_N ;

// actionvalue method response
assign response = resp_fifo$D_OUT ;
assign RDY_response = resp_fifo$EMPTY_N ;

// submodule req_fifo
FIFO2 #(.width(32'd64), .guarded(32'd1)) req_fifo(.RST_N(RST_N),
  .CLK(CLK),
  .D_IN(req_fifo$D_IN),
  .ENQ(req_fifo$ENQ),
  .DEQ(req_fifo$DEQ),
  .CLR(req_fifo$CLR),
  .D_OUT(req_fifo$D_OUT),
  .FULL_N(req_fifo$FULL_N),
  .EMPTY_N(req_fifo$EMPTY_N));

// submodule resp_fifo
```

```
FIFO2 #(.width(32'd64), .guarded(32'd1)) resp_fifo(.RST_N(RST_N),
   .CLK(CLK),
   .D_IN(resp_fifo$D_IN),
   .ENQ(resp_fifo$ENQ),
   .DEQ(resp_fifo$DEQ),
   .CLR(resp_fifo$CLR),
   .D_OUT(resp_fifo$D_OUT),
   .FULL_N(resp_fifo$FULL_N),
   .EMPTY_N(resp_fifo$EMPTY_N));

// rule RL_start_computation
assign WILL_FIRE_RL_start_computation = req_fifo$EMPTY_N && !started ;

// rule RL_do_computation
assign WILL_FIRE_RL_do_computation = started && next_count != 64'd0 ;

// rule RL_finish_computation
assign WILL_FIRE_RL_finish_computation =
   resp_fifo$FULL_N && started && next_count == 64'd0 ;

// inputs to muxes for submodule ports
assign MUX_next_count$write_1__VAL_1 =
   (curr_result_1_MUL_next_count___d12[63:0] == 64'd0) ?
     curr_result_1_MUL_next_count___d12[63:0] :
     next_count - 64'd1 ;

// register curr_result
assign curr_result$D_IN =
   WILL_FIRE_RL_do_computation ?
     curr_result_1_MUL_next_count___d12[63:0] :
     64'd1 ;
assign curr_result$EN =
   WILL_FIRE_RL_do_computation || WILL_FIRE_RL_start_computation ;

// register next_count
assign next_count$D_IN =
   WILL_FIRE_RL_do_computation ?
     MUX_next_count$write_1__VAL_1 :
     req_fifo$D_OUT ;
assign next_count$EN =
   WILL_FIRE_RL_do_computation || WILL_FIRE_RL_start_computation ;

// register started
assign started$D_IN = !WILL_FIRE_RL_finish_computation ;
assign started$EN =
   WILL_FIRE_RL_finish_computation ||
   WILL_FIRE_RL_start_computation ;

// submodule req_fifo
assign req_fifo$D_IN = request ;
assign req_fifo$ENQ = EN_request ;
assign req_fifo$DEQ = WILL_FIRE_RL_start_computation ;
assign req_fifo$CLR = 1'b0 ;

// submodule resp_fifo
assign resp_fifo$D_IN = curr_result ;
assign resp_fifo$ENQ = WILL_FIRE_RL_finish_computation ;
assign resp_fifo$DEQ = EN_response ;
assign resp_fifo$CLR = 1'b0 ;
```

```
  // remaining internal signals
  assign curr_result_1_MUL_next_count___d12 = curr_result * next_count ;

  // handling of inlined registers

  always@(posedge CLK)
  begin
    if (!RST_N)
      begin
        next_count <= `BSV_ASSIGNMENT_DELAY 64'd0;
started <= `BSV_ASSIGNMENT_DELAY 1'd0;
      end
    else
      begin
        if (next_count$EN)
  next_count <= `BSV_ASSIGNMENT_DELAY next_count$D_IN;
if (started$EN) started <= `BSV_ASSIGNMENT_DELAY started$D_IN;
      end
    if (curr_result$EN) curr_result <= `BSV_ASSIGNMENT_DELAY curr_result$D_IN;
  end

  // synopsys translate_off
  `ifdef BSV_NO_INITIAL_BLOCKS
  `else // not BSV_NO_INITIAL_BLOCKS
  initial
  begin
    curr_result = 64'hAAAAAAAAAAAAAAAA;
    next_count = 64'hAAAAAAAAAAAAAAAA;
    started = 1'h0;
  end
  `endif // BSV_NO_INITIAL_BLOCKS
  // synopsys translate_on

  // handling of system tasks

  // synopsys translate_off
  always@(negedge CLK)
  begin
    #0;
    if (RST_N)
      if (WILL_FIRE_RL_start_computation)
$display("Starting request: %h", $unsigned(req_fifo$D_OUT));
    if (RST_N)
      if (WILL_FIRE_RL_do_computation &&
  curr_result_1_MUL_next_count___d12[63:0] == 64'd0)
$display("Overflow!");
    if (RST_N)
      if (WILL_FIRE_RL_finish_computation)
$display("Sending result: %h", $unsigned(curr_result));
  end
  // synopsys translate_on
endmodule  // mkFactorialServer
```

A.10.2 mkFactorialServer.bsv

```
import ClientServer::*;
```

```
import GetPut::*;

import "BVI" mkFactorialServer =
module [Module] mkFactorialServer (Server#(UInt#(64), UInt#(64)));
   default_clock clk (CLK, (*inhigh*) clk_gate);
   default_reset rst (RST_N);

   interface Put request;
      method put (IN_request) ready(RDY_request) enable (EN_request);
   endinterface

   interface Get response;
      method OUT_response get ready(RDY_response) enable(EN_response);
   endinterface

    schedule (request_put) C (request_put);
    schedule (response_get) C (response_get);
    schedule (request_put) CF (response_get);
endmodule
```

A.10.3 SceMiLayer.bsv

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;

import mkFactorialServer::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the dut
   ServerIfc dut <- buildDut(mkFactorialServer, clk_port);
   //BusServer dut <- buildDut(mkDUT, clk_port);

   // Instantiate the transactors with the DUT
   Empty dutout <- mkGetXactor(dut.response, clk_port);
   Empty dutin  <- mkPutXactor(dut.request, clk_port);

   // Instantiate Simulation Control transactor
   Empty simControl <- mkSimulationControl(conf);

   Empty shutdown <- mkShutdownXactor();
endmodule
endpackage
```

A.10.4 project.bld

```
[DEFAULT]
```

```
default-targets:   vlog_dut tcl_tb

[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build
imported-verilog-files: mkFactorialServer.v

[bsim_dut]
run-shell: echo "INFO: Can't build bsim if you are importing verilog - target not built"

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       dut.vexe

[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp FactXactor.cpp
exe-file: tb

[tcl_tb]
extends-target: dut
exe-file: cpp_tb
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: generated_c
c++-source-directory: .
c++-header-aliases
c++-options: -g -O0
shared-lib: libbsdebug.so
c++-files: TclTb.cpp FactXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp


[clean]
run-shell: rm -rf build
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log
run-shell: rm -f vlog_dut directc_* vlog_dut*.log
run-shell: rm -f tb *.params tb*.log
```

A.11 Example 11: Using TLM transactors to integrate a Verilog AHB Memory

A.11.1 BSV wrapper

```
import Ahb      ::*;

`include "TLM.defines"

typedef AhbXtorSlave#(`TLM_PRM_STD) AhbSlaveSTD;
```

```
typedef AhbXtorMaster#(`TLM_PRM_STD) AhbMasterSTD;

import "BVI" mkAhbRam =
module [Module] mkAhbRam (AhbSlaveSTD);
   default_clock clk (CLK, (*inhigh*) clk_gate);
   default_reset rst (RST_N);
   interface AhbSlave bus;
      // Inputs
      method haddr (HADDR)   enable((*inhigh*) en1);
      method hwdata(HWDATA)  enable((*inhigh*) en2);
      method hwrite(HWRITE)  enable((*inhigh*) en3);
      method htrans(HTRANS)  enable((*inhigh*) en4);
      method hburst(HBURST)  enable((*inhigh*) en5);
      method hsize (HSIZE)   enable((*inhigh*) en6);
      method hprot (HPROT)   enable((*inhigh*) en7);
      method hreadyin (HREADY) enable((*inhigh*) en8);
      // Outputs
      method HRDATA hrdata();
      method HREADYOUT hready();
      method HRESP  hresp();
      method HSPLIT hsplit();
   endinterface
   interface AhbSlaveSelector selector;
      method select(HSEL) enable((* inhigh*) en10);
   endinterface

   // we schedule all outputs to be conflict-free
   schedule (bus_hrdata, bus_hready, bus_hresp, bus_hsplit) CF (bus_hrdata, bus_hready,
bus_hresp, bus_hsplit);

   // scheduling all outputs to be Sequenced Before inputs (read-modify-write)
   schedule  (bus_hrdata, bus_hready, bus_hresp, bus_hsplit) SB
      (bus_haddr, bus_hwdata, bus_hwrite, bus_htrans, bus_hburst, bus_hsize, bus_hprot,
bus_hreadyin, selector_select);

   // each input conflicts with itself, but it is conflict-free with all other inputs
   schedule (bus_haddr)  C  (bus_haddr);
   schedule (bus_haddr)  CF (bus_hwdata, bus_hwrite, bus_htrans, bus_hburst, bus_hsize,
bus_hprot, bus_hreadyin, selector_select);
   schedule (bus_hwdata) C  (bus_hwdata);
   schedule (bus_hwdata) CF (bus_hwrite, bus_htrans, bus_hburst, bus_hsize, bus_hprot,
bus_hreadyin, selector_select);
   schedule (bus_hwrite) C  (bus_hwrite);
   schedule (bus_hwrite) CF (bus_htrans, bus_hburst, bus_hsize, bus_hprot, bus_hreadyin, selector_select);
   schedule (bus_htrans) C  (bus_htrans);
   schedule (bus_htrans) CF (bus_hburst, bus_hsize, bus_hprot, bus_hreadyin, selector_select);
   schedule (bus_hburst) C  (bus_hburst);
   schedule (bus_hburst) CF (bus_hsize, bus_hprot, bus_hreadyin, selector_select);
   schedule (bus_hsize)  C  (bus_hsize);
   schedule (bus_hsize)  CF (bus_hprot, bus_hreadyin, selector_select);
   schedule (bus_hprot)  C  (bus_hprot);
   schedule (bus_hprot)  CF (bus_hreadyin, selector_select);
   schedule (bus_hreadyin) C (bus_hreadyin);
   schedule (bus_hreadyin) CF (selector_select);
   schedule (selector_select) C (selector_select);

endmodule
```

A.11.2 SceMiLayer.bsv

```
package SceMiLayer;

import DefaultValue::*;
import SceMi::*;
import AhbRamBVI::*;
// import AhbRam::*;
import GetPut::*;
import ClientServer::*;
import Clocks::*;
import Ahb::*;
import TLM2::*;
import Connectable::*;

`include "TLM.defines"

module [SceMiModule] mkSceMiLayer(Empty);

   ////////////////////////////////////////////////////////////////////
   SceMiClockConfiguration confdut = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(confdut);

   ////////////////////////////////////////////////////////////////////
   // Create a special reset for the dut, which can be controlled from the
   // gui; note that it is clocked by the controlled clock and based on the
   // controlled reset:
   let c_clock = clk_port.cclock;
   let c_reset = clk_port.creset;
   let            dut_reset <- mkReset(0, True, c_clock, clocked_by c_clock, reset_by c_reset);
   Reg#(UInt#(5)) reset_ctr <- mkReg(0, clocked_by c_clock, reset_by c_reset);

   // This rule (in the controlled domain) decrements the counter if non-zero
   // and asserts the new reset for those cycles:
   rule dec_reset_ctr (reset_ctr != 0);
      reset_ctr <= reset_ctr - 1;
      dut_reset.assertReset;
   endrule

   ////////////////////////////////////////////////////////////////////

   // instantiating the dut:
   AhbSlaveSTD dut        <- buildDutWithReset( mkAhbRam, clk_port, dut_reset.new_rst );

   // instantiating a AHB <-> TLM transactor
   AhbMasterXActor#(`TLM_XTR_STD) xactor  <- mkAhbMaster(clocked_by c_clock, reset_by dut_reset.new_rst);

   // connecting the dut to the transactor
   mkConnection(xactor.fabric, dut);

   Empty    control <- mkShutdownXactor();
   Empty simControl <- mkSimulationControl(confdut);

   ////////////////////////////////////////////////////////////////////
   Put#(Bool) soft_reset =
   interface Put;
      method Action put(Bool n);
         reset_ctr <= 10;
      endmethod
```

```
   endinterface;

   // instantiating Sce-Mi transactors
   Empty rst_control <- mkPutXactor (soft_reset, clk_port);
   Empty inReq       <- mkPutXactor(xactor.tlm.rx, clk_port);
   Empty outResp     <- mkGetXactor(xactor.tlm.tx, clk_port);
endmodule

endpackage
```

A.11.3 mkBridge.params

```
// Sce-Mi parameters file generated on: Mon Jan 16 09:48:30 EST 2012
// By: Bluespec scemilink utility, version 2011.12.beta1 (build 26438, 2011-12-13)

// ObjectKind Index AttributeName Value

ClockBinding 0 TransactorName ""
ClockBinding 0 ClockName      "scemi_clk_port"

Clock 0 ClockName          "scemi_clk_port"
Clock 0 RatioNumerator     1
Clock 0 RatioDenominator   1
Clock 0 DutyHi             0
Clock 0 DutyLo             100
Clock 0 Phase              0
Clock 0 ResetCycles        8

MessageOutPort 3 TransactorName ""
MessageOutPort 3 PortName       "scemi_simControl_resp_out"
MessageOutPort 3 PortWidth      32
MessageOutPort 3 ChannelId      9
MessageOutPort 3 Type           "Bit#(32)"

MessageOutPort 2 TransactorName ""
MessageOutPort 2 PortName       "scemi_outResp_outport"
MessageOutPort 2 PortWidth      58
MessageOutPort 2 ChannelId      8
MessageOutPort 2 Type           "TLM2Defines::TLMResponse#(4,32,32,10,Bit#(0))"

MessageOutPort 1 TransactorName ""
MessageOutPort 1 PortName       "scemi_dut_prb_control_data_out"
MessageOutPort 1 PortWidth      32
MessageOutPort 1 ChannelId      7
MessageOutPort 1 Type           "Bit#(32)"

MessageOutPort 0 TransactorName ""
MessageOutPort 0 PortName       "scemi_control_ctrl_out"
MessageOutPort 0 PortWidth      1
MessageOutPort 0 ChannelId      6
MessageOutPort 0 Type           "Bool"

MessageInPort 4 TransactorName ""
MessageInPort 4 PortName       "scemi_simControl_req_in"
MessageInPort 4 PortWidth      32
MessageInPort 4 ChannelId      5
MessageInPort 4 Type           "Bit#(32)"
```

```
MessageInPort 3 TransactorName ""
MessageInPort 3 PortName       "scemi_rst_control_inport"
MessageInPort 3 PortWidth      1
MessageInPort 3 ChannelId      4
MessageInPort 3 Type           "Bool"

MessageInPort 2 TransactorName ""
MessageInPort 2 PortName       "scemi_inReq_inport"
MessageInPort 2 PortWidth      110
MessageInPort 2 ChannelId      3
MessageInPort 2 Type           "TLM2Defines::TLMRequest#(4,32,32,10,Bit#(0))"

MessageInPort 1 TransactorName ""
MessageInPort 1 PortName       "scemi_dut_prb_control_control_in"
MessageInPort 1 PortWidth      17
MessageInPort 1 ChannelId      2
MessageInPort 1 Type           "SceMiSerialProbe::ProbeControl"

MessageInPort 0 TransactorName ""
MessageInPort 0 PortName       "scemi_control_ctrl_in"
MessageInPort 0 PortWidth      1
MessageInPort 0 ChannelId      1
MessageInPort 0 Type           "Bool"

Link 0 LinkType   "TCP"
Link 0 TCPAddress "127.0.0.1"
Link 0 TCPPort    3375
```

A.11.4 SlaveXactor.h

```
// Copyright Bluespec Inc. 2009-2010


#pragma once

// Include Bluespec's SceMi C++ api
#include "bsv_scemi.h"
#include "TLMRequest_4_32_32_10_Bit_0.h"
#include "TLMResponse_4_32_32_10_Bit_0.h"

// Define a class for the top-level transactor
class SlaveXactor {
 protected:
  // Data members include transactors contained in the model
  // Data Xactors
  InportProxyT<TLMRequest_4_32_32_10_Bit_0 >  m_datain;
  OutportQueueT<TLMResponse_4_32_32_10_Bit_0 >  m_dataout;
  // Shutdown Xactor
  ShutdownXactor  m_shutdown;
  // soft-reset Xactor
  InportQueueT<Bool >   m_soft_rst;

 public:
  // Constructor
  SlaveXactor (SceMi *scemi) ;
  // Destructor
  ~ SlaveXactor();
```

```
  // Public interface .....
  bool putRequestNB(const TLMRequest_4_32_32_10_Bit_0 &);
  bool putRequestB (const TLMRequest_4_32_32_10_Bit_0 &);
  bool getResponseNB (TLMResponse_4_32_32_10_Bit_0 &);
  bool getResponseB (TLMResponse_4_32_32_10_Bit_0 &);

  void shutdown();

  // send reset from the software side
  bool sendReset();

 private:
  // Private helper members
};
```

A.11.5 SlaveXactor.cpp

```
// Copyright Bluespec Inc. 2009-2010


#include <iostream>
#include "SlaveXactor.h"
#include "TLMRequest_4_32_32_10_Bit_0.h"
#include "TLMResponse_4_32_32_10_Bit_0.h"

using namespace std;

SlaveXactor::SlaveXactor(SceMi *scemi)
  : m_soft_rst("", "scemi_rst_control_inport", scemi)
  , m_datain ("", "scemi_inReq_inport", scemi)
  , m_dataout ("", "scemi_outResp_outport", scemi)
  , m_shutdown ("", "scemi_control", scemi)
{

}

SlaveXactor::~SlaveXactor()
{
}

void SlaveXactor::shutdown()
{
  m_shutdown.blocking_send_finish();
}

// Send the request to the DUT
bool SlaveXactor::putRequestNB (const TLMRequest_4_32_32_10_Bit_0 &request)
{
  TLMRequest_4_32_32_10_Bit_0 reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}
bool SlaveXactor::putRequestB (const TLMRequest_4_32_32_10_Bit_0 &request)
{
  TLMRequest_4_32_32_10_Bit_0 reqbit(request);
  m_datain.sendMessage(reqbit);
  return true;
}
```

```
// return true and populates resp if a response is available
// otherwise returns false
bool SlaveXactor::getResponseNB (TLMResponse_4_32_32_10_Bit_0 &resp)
{
  TLMResponse_4_32_32_10_Bit_0 respbit;
  bool gotone = m_dataout.getMessageNonBlocking (respbit) ;
  if (gotone) {
    resp = respbit;
  }
  return gotone;
}

bool SlaveXactor::getResponseB (TLMResponse_4_32_32_10_Bit_0 &resp)
{
  TLMResponse_4_32_32_10_Bit_0 respbit = m_dataout.getMessage () ;
  resp = respbit;
  return true;
}

bool SlaveXactor::sendReset() {
  m_soft_rst.sendMessage( true ) ;
  return true;
}
```

A.12 Example 12: Implementing a synthesizable testbench

The source files for this section are found in the directory `example12`.

A.12.1 Tb.bsv

```
package Tb;

import FIFO::*;
import ClientServer::*;
import GetPut::*;
import DUT::*;
import Connectable::*;

typedef Server#(UInt#(64), UInt#(64)) ServerIfc;

interface StimIfc;
   method Action start (UInt#(64) seed);
   interface Get#(UInt#(64)) stim_request;
   interface Put#(UInt#(64)) stim_response;
endinterface

(* synthesize *)
   module [Module] mkTop (Put#(UInt#(64)));

   Reg#(UInt#(64)) inValue <- mkReg(0);
   Reg#(Bool) startit <- mkReg(True);

   StimIfc     stim_gen <- mkStimulusGen;
   ServerIfc   dut      <- mkFactorialServer;
```

```
   mkConnection(stim_gen.stim_request, dut.request);
   mkConnection(stim_gen.stim_response, dut.response);

   rule startTb (startit && inValue!=0);
      // Get the value
      let val = inValue;
      stim_gen.start(val);
      $display("startTB");
      $display("Seed value = %0d", val);
      startit <= False;
   endrule

   rule resetTb (!startit);
      startit <= True;
      inValue <= 0;
   endrule

   return (toPut(asReg(inValue)));
endmodule: mkTop

(* synthesize *)
module [Module] mkStimulusGen (StimIfc);

   Reg#(Bool) started   <- mkReg(False);
   FIFO#(UInt#(64)) f_out      <- mkFIFO;      // To buffer outgoing requests
   FIFO#(UInt#(64)) f_in       <- mkFIFO;      // To buffer incoming responses

   Reg#(int)       count_sent  <- mkReg(10);
   Reg#(int)       count_rcvd  <- mkReg(10);
   Reg#(UInt#(64)) value_sent  <- mkReg(0);

   rule start_gen (!started && value_sent!=0);
      let val = value_sent;
      f_out.enq(val);
      count_sent <= count_sent - 1;
      started <= True;
      $display ("seed count = %0d", count_sent, $time);
      $display ("seed value sent = %0d", val);
   endrule

   rule gen_stimulus (started && (count_sent > 0));
      let x = value_sent;
      f_out.enq (x);
      value_sent <= value_sent + 1;
      count_sent <= count_sent-1;
      $display ("count sent = %0d", count_sent);
      $display ("value sent = %0d", x);
   endrule

   rule check_results (started && (count_rcvd > 0));
      let y     = f_in.first; f_in.deq;
      count_rcvd <= count_rcvd - 1;
      $display ("count received = %0d", count_rcvd);
      $display ("factorial received = %0d", y);
   endrule

   rule startover (started && count_rcvd == 0);
      $display("all done");
      started <= False;
```

```
      count_sent <= 10;
      count_rcvd <= 10;
      value_sent <= 0;
   endrule

    interface Put stim_response = toPut(f_in);
    interface Get stim_request = toGet(f_out);

   method Action start(UInt#(64) seed) ;
      value_sent <= seed;
   endmethod
endmodule: mkStimulusGen
endpackage: Tb
```

A.12.2 SceMiLayer.bsv

```
package SceMiLayer;

import SceMi::*;
import GetPut::*;
import DefaultValue::*;
import ClientServer::*;
import Connectable ::*;
import Tb::*;

import DUT::*;

module [SceMiModule] mkSceMiLayer ();

   //Instantiate the required control transactors
   SceMiClockConfiguration conf = defaultValue;
   SceMiClockPortIfc clk_port <- mkSceMiClockPort(conf);

   // Instantiate the testbench
   Put#(UInt#(64)) testbench <- buildDut(mkTop, clk_port);

   // Connect the tb interface to SceMi
   Empty testconnect <- mkPutXactor(testbench, clk_port);

   // Instantiate Simulation Control transactor
   Empty simControl <- mkSimulationControl(conf);

   Empty shutdown <- mkShutdownXactor();

endmodule
endpackage
```

A.12.3 TbXactor.cpp

```
// Copyright Bluespec Inc. 2009-2010


#include <iostream>
#include "TbXactor.h"
```

```
using namespace std;

// Constructor definition
TbXactor::TbXactor(SceMi *scemi)
  : m_datain  ("", "scemi_testconnect_inport", scemi)
  , m_shutdown ("", "scemi_shutdown", scemi)
{

}

// Destructor definition
TbXactor::~TbXactor()
{
}


// Method definitions
// shutdown
void TbXactor::shutdown()
{
  m_shutdown.blocking_send_finish();
}

// Non-blocking put request
// Send the request to the DUT
bool TbXactor::putRequestNB (long request)
{
  BitT<64> reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}

// Blocking put request
bool TbXactor::putRequestB (long request)
{
  BitT<64> reqbit(request);
  m_datain.sendMessage(reqbit);
  return true;
}
```

A.12.4 TbXactor.h

```
// Copyright Bluespec Inc. 2009‘-2010


#pragma once

// Include Bluespec’s SceMi C++ api
#include "bsv_scemi.h"

// Define a class for the top-level transactor
class TbXactor {
 protected:
  // Data members include transactors contained in the model
  // Data Xactors
  InportProxyT<BitT<64> >  m_datain;
  // Shutdown Xactor
  ShutdownXactor  m_shutdown;
```

```
 public:
  // Constructor
  TbXactor (SceMi *scemi) ;
  // Destructor
  ~ TbXactor();

  // Public interface .....
  bool putRequestNB(long);
  bool putRequestB (long);

  void shutdown();

 private:
  // Private helper members
};
```

A.12.5 project.bld

```
[DEFAULT]
default-targets:   vlog_dut tcl_tb

[dut]
hide-target
top-file:             Bridge.bsv
verilog-directory:    build
binary-directory:     build
simulation-directory: build

[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
exe-file:       bsim_dut

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
exe-file:       dut.vexe

[tb]
scemi-tb
build-for: c++
c++-header-targets: none
c++-files: Tb.cpp TbXactor.cpp
exe-file: tb

[tcl_tb]
extends-target: dut
exe-file: cpp_tb
build-for: c++
scemi-tb
uses-tcl
c++-header-directory: generated_c
c++-source-directory: .
c++-header-aliases
c++-options: -g -O0
```

```
shared-lib: libbsdebug.so
c++-files: TclTb.cpp TbXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp

[clean]
run-shell: rm -rf build xilinx vlog_edited work_* dut.*exe generated_c transcript
run-shell: rm -f bsim_dut bsim_dut.so bsim_dut*.log  ml605* *.so
run-shell: rm -f vlog_dut directc_* vlog_dut*.log  *.log
run-shell: rm -f tb *.params tb*.log
```