Emulation App User Manual

Revision: 30 July 2014

Contents

1 Introduction

This document describes Bluespec's Emulation App, including Bluespec SystemVerilog (BSV) support for co-emulation in the SCE-MI standard (available from Accellera at the website http://www.eda.org/itc.

1.1 Purpose

As design size and complexity increase, design teams are turning to emulation to provide the speed necessary for modeling, architectural exploration, design, and verification. Emulation App allows modeling at multiple levels of abstraction and supports verification through both simulation and emulation. Designed to work with multiple FPGA platforms, Bluespec's Emulation App provides fast bringup and portability and facilitates rapid testbench re-partitioning.

An Emulation App system, shown in Figure 1, has two major components: an FPGA containing a Device-Under-Test (DUT) and a PC host containing the emulation console. The two sides are connected by a physical PCIe link. The FPGA may also contain a synthesizable testbench (TB). The PCIe link communicates with the FPGA components and the emulation console through implementation-independent transactors. The hardware abstraction of the physical connections provides deployment flexibility and insulates the designer from the underlying communication protocol. The co-emulation link provides controllability, observability, and performance between the host and the FPGA sides of the emulation system, without requiring the user to write any communication code.

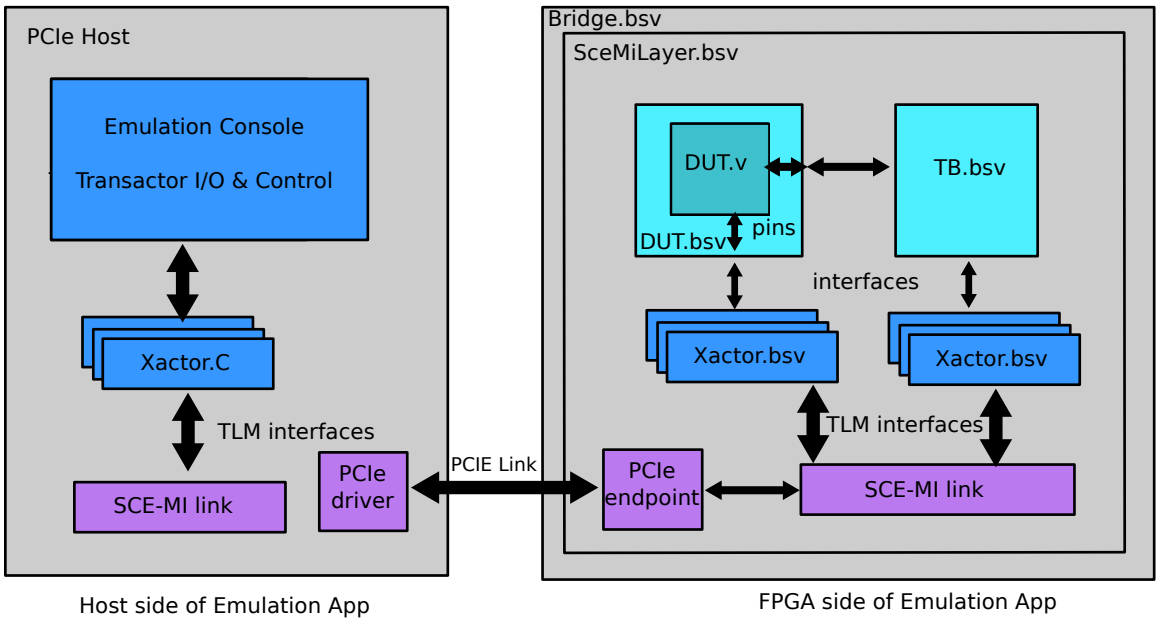

Figure 1: Overview of Emulation App

Emulation App provides:

- Access to an industry-standard co-emulation API
- Seamless migration of a design across supported co-emulation hosts
- Ability to use provided transactors or write your own transactors in BSV

- Ability to use a standard un-timed C++ testbench with a design in simulation or emulation
- Hardware abstraction of physical connections for deployment flexibility
- Flexible generation of co-emulation link infrastructure based on custom scenarios
- Easily generated user-friendly GUI control of software testbench, including probes on signals of interest

1.2 Requirements

Emulation App requires a working Bluespec tool installation along with additional components specified by your environment, such as a C++ compiler and, if desired, FPGA board. Bluespec supports simulation with Bluesim or a third-party Verilog simulator, in addition to emulation on a variety of emulation boards. Currently supported emulation boards are listed in Section 3.5. Adding a new FPGA board to this list typically requires some engineering work by Bluespec. Please contact Bluespec support for advice on whether the FPGA board you are using is supported.

This guide assumes familiarity with the Bluespec SystemVerilog language and C++. Please refer to the BSV Reference Guide, user guide, and tutorials for information on how to design and write specifications in the Bluespec SystemVerilog environment.

A detailed Emulation App tutorial with source code examples is provided in the release to guide you through developing an application. The tutorial starts with a simple example and demonstrates many aspects of implementation. Later sections add complexity, discussing specific features and considerations of designing with Emulation App, allowing you to choose topics and customize the tutorial based on your design and interests.

The complete SCE-MI standard is available from Accellera's website.

1.3 Installation and licensing

Refer to the *Bluespec SystemVerilog User Guide* for information on installing Bluespec and licensing. Bluespec uses the FLEXnet licensing package and Emulation App is an individually licensed product.

You may need to install a third party tool to use a particular feature of Emulation App.

- Simulation requires either Bluesim or a third-party Verilog simulator.
- The **build** utility requires python 2.4 or later.
- Emulation requires a Bluespec-supported emulation board.
- The host environment utilizing the GUI and probe components requires a C++ compiler and Tcl/Tk. A version of Tcl/Tk 8.5 is included in the Bluespec release.
- To view waves requires a third-party waveform viewer. The Bluespec Development Workstation (BDW) can interface directly to viewers provided by SpringSoft/Novas (Verdi, Debussy, nWave) and GtkWave.
- To view scheduling graphs through the BDW requires the `graphvix` package, including the Tcl extensions.
- To view scheduling graphs without the BDW, you can use any of a number of 3rd-party packages capable of displaying dot files, including the graphviz viewer application called `dot`, which converts a `.dot` file to a pdf or png format. See www.graphviz.org for more information.

1.4 Components of Emulation App

Emulation App includes the following components:

- Host-side (C++) and hardware-side (BSV) implementations of the SCE-MI standard API
- BSV Hardware transactors to connect your design into Emulation App
- C++ transactors for developing a testbench in SystemC or C++
- BSV transactors for developing a software testbench in BSV, allowing the development of synthesizable testbenches
- Components to generate a complete GUI-based testbench
- Automated `build` utility to control the build flow for systems including emulation, simulation, and fpga board requirements
- GUI-based HDL editor and probe insertion tool
- An infrastructure linkage tool connecting the software and hardware sides
- Bluespec Emulation App Documentation
  - This user guide
  - *Bluespec Tutorial: Emulation App*

1.5 Overview of the hardware side

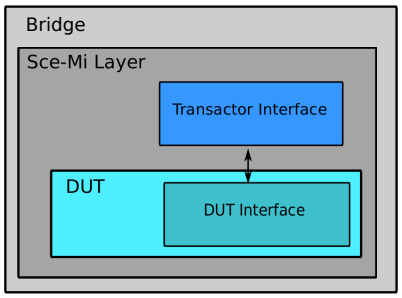

Figure 2: Hardware Overview

The most basic hardware side implementation, for emulation or simulation, contains the three user-specific layers shown in Figure 2. For simplicity, each layer can be thought of as a single file, though the actual implementation may be comprised of multiple files. The executable loaded on the FPGA and/or simulated is built from these files. A more complex system, such as one including a synthesizable testbench, will contain more components while still fitting this basic paradigm.

1. DUT layer: Device-Under-Test (DUT) written in either Verilog or BSV. If written in Verilog, the DUT definition will include a BSV wrapper file.
2. SceMi layer: DUT-specific, FPGA-independent layer connecting the DUT into the Emulation App using Bluespec-provided or user-written transactors. The user edits this layer to instantiate the DUT and to connect it to clocks and transactors.
3. Bridge layer: a FPGA-specific layer comprised of a template file shipped with Emulation App. It takes no user input and is specific to the selected FPGA board and configuration. There is a also a TCP-based bridge file to use with simulations.

1.5.1 DUT

The DUT is a Verilog or BSV implementation of the design being tested. If written in Verilog (or other RTL, such as VHDL), you create a BSV wrapper which imports the Verilog file and translates the Verilog input and output pins to BSV interfaces, enabling fast and verifiable connections to the transactors. The DUT does not change when you include it in an Emulation App environment or when the FPGA changes.

The DUT communicates with the SceMiLayer through its interface.

1.5.2 SceMiLayer

The SceMiLayer is the connection layer, linking the platform-independent DUT with the FPGA-dependent bridge, inserting the DUT into the Emulation App environment.

The SceMiLayer contains transactors which convert between the untimed messages from the software host and the timed events on the FPGA. This layer is DUT-dependent and FPGA-independent. The transactors used in the SceMiLayer depend on the interface provided by the DUT, as well as the clocking requirements of the DUT and the testbench. If you change the platform implementation, such as switch to a different emulator board, the SceMiLayer will usually not change.

1.5.3 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. All the coordination and communication logic is implemented in this layer.

Bridge files for supported emulation and simulation systems are provided as part of the Emulation App product.

1.6 Overview of the host side

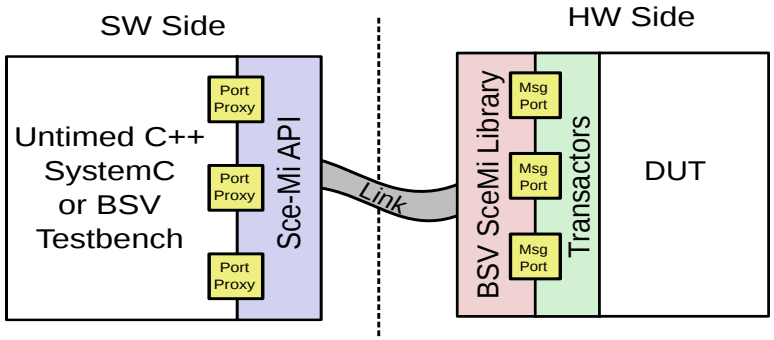

Figure 3: Software transactors use proxies on testbench to connect with the message ports and hardware transactors on the fpga

The host side contains the untimed software testbench along with software transactors to connect to the hardware transactors on the FPGA, as shown in Figure 3. Bluespec provides a large number of proxies, software transactors, and utilities to help the user write a C++ testbench for the host side.

Included in Emulation App is a utility to define probes on selected signals, which can be viewed through a third-party waveform viewer. A set of C++ and Tcl/Tk components are provided to add a front-end GUI to control the testbench and the probes.

1.7 Tools

Emulation App includes following tools for developing and building the project:

- `build` (Section 2 C): a simple and powerful tool to build a project. The utility codifies the rules for building the Emulation App system so you don't have to remember specific commands, directory structures, file names, and dependencies. A single `build` project configuration file can generate code for multiple FPGA boards and emulation targets.
- `HDL editor` (Section 7): a GUI-based tool which allows you to edit the generated HDL file to add probes and make other HDL-specific changes. You can use the tool to modify signal names and handle other tedious edits in a controlled manner. No changes are made to the source design and there is no additional design work required to add probes. The probe signals are viewed through a GUI on the PC host and as waveforms through a third-party waveform viewer.
- `scemilink` (Section B.2): the Emulation App infrastructure linkage tool. Scemilink takes the description of the hardware side and generates a text file (`.params`) describing the clocks, transactors, message ports and other link information. This file is read by the host side during initialization to bind the software side message port proxies to the hardware side message ports. Scemilink generates other files as required by the emulation hardware.

2 Building the Emulation App system

A powerful feature of Emulation App is the `build` utility, which facilitates configuring and building a project. This section provides an overview of the build process along with the `build` utility; Appendix B describes the build flows for different configurations in more detail, while Appendix C contains the complete `build` utility documentation.

The build flow for a given system has many variations depending on the emulation hardware and the project configuration. The `build` utility codifies the rules for building the system, controlling the build based on directives describing the project environment. The utility can be used for simple and complex projects including compiling individual components, building a simulation environment, and building a complete FPGA-based emulation system.

The `build` utility runs from a project configuration (`.bld`) file; the default configuration file is named `project.bld`. This file is similar to a makefile; it is composed of a set of build targets, where each target describes a component of the project (DUT, testbench, etc.) along with commands (directives) describing the options and parameters of the target including directories, files, module names, compiler options, emulation boards, and emulator options. When run, the utility determines the commands to execute and the execution order, for the specified target.

There can be multiple build targets for similar functions. For example, you can have two targets describing how to build the hardware DUT, where one generates Verilog code while the other generates code for Bluesim. When running the `build` utility you select the appropriate target for that iteration.

When the `build` command is entered on the command line, along with one or more build target names, the utility determines the correct commands and the proper sequence of commands required to execute each stage. The project is built consistently and reliably each time, and defaults are only specified once per project. The complete reference documentation for the `build` utility can be found in Appendix C.

The following table lists the options you can specify on the command line with the `build` command.

Option	Alternate Syntax	Description
`--version`		show program's version number and exit
`--help`	`-h`	show help message and exit
`--doc`	`-d`	print full documentation, then exit
`--init`		generate an initial project.bld file from a project.init file
`--initforce`		generate an initial project.bld file from a project.init file, the file will be written over even if it already exists
`--from=STAGE`	`-f STAGE`	begin building from STAGE
`--to=STAGE`	`-t STAGE`	finish building after STAGE
`--project=FILE`	`-p FILE`	read project description from FILE
`--list`	`-l`	list targets and stages, then exit
`--verbose`	`-v`	print verbose messages to stdout
`--dry-run`		display the commands executed in each stage without executing

As an example we'll examine a project file containing two targets to build the hardware-side: `bsim_dut` to generate for Bluesim, and `vlog_dut` to generate for Verilog. And on the software side, the software testbench target is `tb`. Section 2.1 defines the directives shown here in more detail.

The file also contains two other targets: `[Default]` and `[dut]`. `[DEFAULT]` is the only reserved target name in a build file. In this section you specify the targets to be built if no target is provided on the command line.

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

The next section `[dut]` contains the directives for the DUT regardless of the technology. This is not a reserved name, but a name chosen to be meaningful. In this example the `[dut]` target defines common directives that are shared by multiple DUT targets. The directives in the `[dut]` section of this file are included in both the `[bsim_dut]` and `[vlog_dut]` targets, through the `extends-target` directive.

```
[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
```

The final target contains the directives for the software testbench:

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

To compile the DUT for Bluesim, at a unix command prompt type:

```
build bsim_dut
```

To compile for Verilog simulation, type:

```
build vlog_dut
```

Once the build is complete, you can run the simulation or load the FPGA with the executables and synthesize the design.

2.1 Project file targets

In this section we're going to examine some of the directives and settings most commonly used in the `project.bld` file. Except for the `DEFAULT` target, all the target names are provided as examples and can be changed to any name that is meaningful for your project.

2.1.1 Default

The only reserved target name is `[DEFAULT]`. Sepcify the targets to be built by default with the `default-targets` directive. You can also specify options to be used in every build, such as the `bsc-compile-options` and `bsc-link-options`.

```
[DEFAULT]
default-targets:     bsim_dut tcl_tb
bsc-compile-options: -aggressive-conditions -keep-fires
bsc-link-options:    -keep-fires
```

2.1.2 DUT

[dut] In this example the target `[dut]` directives specify the top-file and describe the project layout for the hardware side, including directories for third-party tools. The name of the top-module is derived from the top file (`mk<topfile>`) unless it the `top-module` directive is used. The directive `hide-target` indicates the target cannot be specified directly, but instead is used by another target through the `extends-target` directive. In this case, `[dut]` doesn't provide enough information to be used on its own, so it cannot be specified directly but only through `[vlog_dut]` and `[bsim_dut]`.

```
[dut]
hide-target
top-file:              Bridge.bsv
verilog-directory:     vlog_dut
binary-directory:      bdir_dut
simulation-directory:  simdir_dut
info-directory:        info_dut
altera-directory:      quartus
xilinx-directory:      xilinx
```

Simulation In this example we have a target for simulating to Bluesim (`[bsim_dut]`) and for simulating to Verilog (`[vlog_dut]`), both utilizing TCP as the communication channel.

Note that both targets include the directives contained in the target `[dut]` through the `extends-target` directive.

```
[bsim_dut]
extends-target: dut
build-for:      bluesim
scemi-type:     TCP
scemi-tcp-port: 7735
exe-file:       dut

[vlog_dut]
extends-target: dut
build-for:      verilog
scemi-type:     TCP
scemi-tcp-port: 7735
exe-file:       dut
```

Synthesis The synthesis targets include the `[dut]` directives and specify the board the target is building for. This example has targets for the dini 7006 board and the Virtex ml605 board.

```
[7006_dut]
extends-target: dut
build-for:      dn7006
scemi-type:     PCIE_DINI

[ml605_dut]
extends-target: dut
build-for:      ml605
scemi-type:     PCIE_VIRTEX5
```

2.1.3 BSV testbench

The testbench can be written in BSV and simulated. The targets for a simulated BSV testbench mirror the DUT targets: the target `[bsv_tb]` is a hidden target whose directives are extended into a Bluesim target (`[bsim_tb]`) and a Verilog target (`[vlog_tb]`).

[bsv_tb] The `[bsv_tb]` target includes a top-file directive, as well as directives describing the project layout (directory structure). The directive `scemi-tb` designates a target as being a SCE-MI testbench.

```
[bsv_tb]
hide-target
scemi-tb
top-file:             Testbench.bsv
verilog-directory:    vlog_tb
binary-directory:     bdir_tb
simulation-directory: simdir_tb
info-directory:       info_tb
exe-file:             tb
```

[bsim_tb] and [vlog_tb] These targets use the directives in `[bsv_tb]` along with a `build-for` directive indicating the target platform.

```
[bsim_tb]
extends-target: bsv_tb
build-for:      bluesim

[vlog_tb]
extends-target: bsv_tb
build-for:      verilog
```

2.1.4 C++ testbench

The `[tcl_tb]` target contains the directives to build a C++ testbench. In this example the testbench is comprised of two main files: `TclTb.cpp` and `GetPutXactor.cpp`.

```
[tcl_tb]
extends-target: dut
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 GetPutXactor.cpp \
   ${BLUESPECDIR}/tcllib/include/bsdebug_common.cpp
```

2.2 Build messages

The `build` utility output displays the commands executed. An example, generating for a target `bsim_dut`:

```
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/lib/board_support/bridges -elab -D SCEMI_TCP -u -g
mkBridge Bridge.bsv
<<< 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 the build utility executes, it logs the output of commands into log files using the naming scheme `<target>_<stage>.log`. If the `--verbose` or `-v` option is given, the command output is also displayed on the screen.

2.3 Quick-start emulation prototyping script

One feature in the `build` utility is a quick-start prototyping capability allowing you to build a complete emulation environment, including a GUI testbench, starting with just a Verilog DUT. The script assumes that there is no `project.bld` or other Emulation App files in the directory.

The prototyping script performs the following steps:

- Generates the project definition files:
  - `project.init`
  - `project.bld`
- Generates the hardware-side files:
  - Bridge layer: `Bridge.bsv`
  - SceMiLayer: `SceMiLayer.bsv`
  - BSV-wrapped DUT: `Bsv`*verilogfilename*`.bsv`
- Generates the software-side testbench:
  - `Bsv`*verilogfilename*`.cpp`
  - `Bsv`*verilogfilename*`.h`
  - `TclTb.cpp`
- Generates the software-side GUI:
  - `debug-gui`
  - `gui_dut.tcl`
  - `gui_top.tcl`

```
[DEFAULT]
# The verilog file containing the top module
top-file:

# The name of the top module
top-module:

# The target clock frequency (mhz) of the design
design-clock-frequency:

# The intended simulation/emulation target
#  - possible choices are:
#    bluesim    - build and link for bluesim simulation
#    verilog    - build and link for Verilog simulation
#    dn7002     - build Verilog RTL and synthesize for a Dini 7002 board
#    dn7006     - build Verilog RTL and synthesize for a Dini 7006 board
#    dn7406     - build Verilog RTL and synthesize for a Dini 7406 board
#    dn10ghxtll - build Verilog RTL and synthesize for a Dini PCIe_10G_HXT_LL board
#    ml507      - build Verilog RTL and synthesize for a Xilinx ML507 board
#    ml605      - build Verilog RTL and synthesize for a Xilinx ML605 board
#    xupv5      - build Verilog RTL and synthesize for a Xilinx XUPV5 board
build-for:

# Specify the clock and reset signals of the design. Format: Signal:Port
# These are the signals that will be controlled by scemi, CLK and RST_N etc...
#
# For example, to connect SceMi CLK signal to input port named clock
#              and SceMi RST_N to input port named resetN of the module.
#
# scemi-control-signals: CLK:clock RST_N:resetN
#
scemi-control-signals:
```

Figure 4: Default project.init file

2.3.1 Running the script

The following steps build a complete emulation or simulation environment, including GUI testbench, from a Verilog file. The `project.init` file is used by the `build` utility to generate most of the Emulation App files, including the `project.bld` file. The script first generates a shell `project.init` file which you edit to specify the basic parameters of the project, including the top module and verilog file.

1. Create a working directory for your project containing the Verilog file (can be in a subdirectory). There should not be an existing `project.bld` file in the directory.
2. Type `build` to start the script
   - A `project.init` file is generated (Figure 4) and the following message is displayed:

     ```
     No project.bld file found.
     -> To generate an initial project.bld file,
        please edit project.init and run "build --init".
     ```

   - Edit the `project.init` filling in:
     - `top-file`: fully qualified name of the Verilog file from the current working directory
     - `top-module`: name of the top verilog module
     - `design-clock-frequency`: speed in Mhz for the clock speed x
     - `build-for`: simulation or emulation target (select from list displayed)
     - `scemi-control-signals`: if there are clock and reset signals add their names, if not leave blank

3. Generate the `project.bld` file by running `build` again with the `--init` flag

   ```
   build --init
   ```

4. Build the default targets

   ```
   build
   ```

5. Run the simulation (or emulation)

3 Designing the hardware-side

The hardware side is implemented through three user-specific layers: the DUT, the SceMiLayer, and the Bridge, as shown in Figure 5. While these files may actually be composed of multiple files, we can use this basic paradigm to understand how the components fit together.

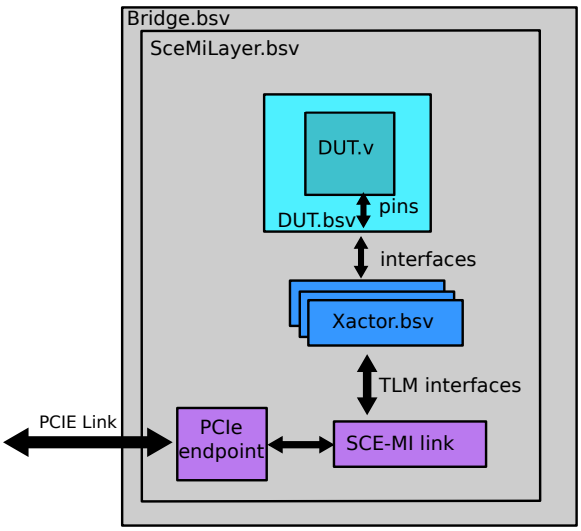

Figure 5: Hardware side with Verilog DUT

3.1 Preparing the DUT

The DUT is the design being tested. The DUT can be written in BSV, Verilog, or other hardware definition language. No special coding is required to include a DUT in the Emulation App environment and the DUT is independent of the emulation target. Switching emulation boards does not require any changes to the DUT.

One requirement is that the DUT must provide BSV interfaces to communicate with the Emulation App definitions in the SceMiLayer. For DUTs written in BSV this is automatic. If the DUT is 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. The next section describes how to wrap a Verilog DUT using `importBVI` so that it can be included in the Emulation App system. If you are using a DUT written completely in BSV, the next step is to prepare the SceMiLayer, Section 3.3.

3.2 Wrapping a Verilog DUT

3.2.1 Interfaces

Interfaces provide a means to group wires into bundles with specified uses, described by methods. BSV does not have input, output and inout pins like Verilog and VHDL. Instead, signals and buses are driven in and out of modules with methods. Most modules have several methods to drive all the buses in and out of a module. These methods are grouped together into interfaces.

In BSV we separate the declaration of an interface from the definition of a module. This allows us to reuse common interfaces across designs. All modules which share a common interface have the same methods, and therefore the same number and types of inputs and outputs. However, those methods may have different implementations in different modules.

When we wrap a Verilog module we are importing the Verilog file and translating the input and output pins of the DUT to BSV interfaces, enabling fast and verifiable connections to the transactors. Emulation App comes complete with a set of general-purpose transactors (Section 3.3.3), where each transactor provides a specific interface. To use these transactors, the Verilog file should be wrapped to provide the same interface as the transactor you would like to use. You can also write your own transactors to use with other interfaces.

3.2.2 TLM interfaces

A common paradigm between two blocks uses a get/put: one side *gets* or retrieves an item from an interface and the other side *puts* or gives an item to an interface. These types of interfaces are used in *Transaction Level Modeling* or TLM for short. This pattern is so common in system design that BSV provides parameterized library interfaces, modules, and functions for this purpose.

When choosing the interface for a Verilog DUT, you can use an existing BSV interface or define your own interface. The basic building blocks of TLM are the `Get` and the `Put` interfaces, defined in the package `GetPut`. These are always a good starting point when selecting interfaces for a design.

The `Get` interface defines a `get` method, similar to a dequeue, which retrieves an item from an interface and removes it at the same time. The type of the method is `ActionValue`, meaning it changes state (removes the item) and returns a value.

```
interface Get#(type element_type);
    method ActionValue#(element_type) get();
endinterface: Get
```

The `Put` interfaces defines a `put` method, similar to an enqueue, which gives an item to an interface. The type of the method is `Action`, meaning it just changes state (adds the item).

```
interface Put#(type element_type);
    method Action put(element_type x1);
endinterface: Put
```

As you can see, the interfaces are polymorphic, that is they can operate on more than one type of data. You define the type of the data within your design.

To use these interfaces, the `GetPut` package must be imported into your design:

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

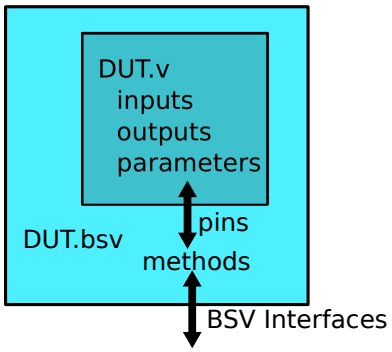

Figure 6: BSV Wrapper around Verilog DUT

A Verilog module encompasses a set of pins for the inputs, outputs, and parameters. In BSV, these are represented by methods and interfaces. The `import "BVI"` statement defines a BSV module providing an interface whose methods encapsulate the possible transactions clients can perform, connecting the Verilog input and output ports to BSV objects, as shown in Figure 6.

Constructing the wrapper involves assigning the input and output ports to methods within interfaces and defining the clocks and resets. Scheduling annotations are added to fully describe the behavior of the methods.

3.2.3 Writing a wrapper for a TLM-based DUT

In this example, we're wrapping a Verilog file, `mkFactorialServer.v`, implementing a factorial server as the DUT. 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:

```
module mkFactorialServer(CLK,
                         RST_N,

                         request,
                         EN_request,
                         RDY_request,

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

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

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

Defining the interface The first task is to examine the DUT and by using your knowledge of its functionality and requirements define how you want to use the input and output wires. Since in this example we are using a TLM-based paradigm, we can define the request, `inRequest`, as a `Put` interface [1] and the response, `outResponse`, as a `Get`.

The wrapped module will return a single interface. Since we have two interfaces (the `Get` and the `Put`), we combine them into one. The request is a `UInt#(64)` and the response is a `UInt#(64)`, to match the `Bit#(64)` ports in the Verilog.

```
interface FactServer;
   interface Put#(UInt#(64)) inRequest;
   interface Get#(UInt#(64)) outResponse;
endinterface
```

Defining the module The `import "BVI"` statement is used to wrap the Verilog module. The header of the statement is:

```
import "BVI" mkFactorialServer =
module [Module] mkFactorialServer (FactServer);
```

where `mkFactorialServer` is the name of the Verilog file (`mkFactorialServer.v`), the name of the BSV module is `mkFactorialServer`, and `FactServer` is the name of the interface provided by the module.

The remaining parts of the statement are clocks and resets, interfaces and methods, and schedule statements.

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`.

Defining 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 enqueueing data into a fifo) or that a state change is to occur (such as "popping" data from a stack).
  - Clock or Reset inputs

[1] The first letter of data type identifiers, including interface types, are always upper case, while the first letter of variable names, are lower case. So in our case, the interface type (Put) is upper case, while the name of the interface (inRequest) is lower case.

- 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 Get/Put example. We've already grouped (and named) the input and output signals according to their functions:

```
// 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;
```

The methods must be grouped into interfaces. We already determined that this design has two interfaces: `inRequest` and `outResponse`. Each interface is going to have a single method, with a ready and an enable.

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

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

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

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

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

- 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 (inRequest_put) C (inRequest_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 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 (inRequest_put) CF (outResponse_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.

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 `inRequest`, and it has a `put` method. So the full name of the method is: `inRequest_put`.

3.3 Preparing the SceMiLayer

The SceMiLayer is the Emulation App connection layer, linking the FPGA-independent DUT with the hardware-dependent bridge, inserting the DUT into the emulation environment.

The SceMiLayer contains transactors which convert between the untimed messages from the software testbench and timed events on the FPGA. This layer is design-dependent, and for simple emulation environments, platform-independent. The transactors used depend on the interface provided by the DUT, as defined in the wrapper or BSV file, as well as the clocking requirements of the DUT and the testbench. If you switch emulation boards, the SceMiLayer will usually not change. More complex designs, utilizing many components of the emulation board, may impact the SceMiLayer.

The basic components of the SceMiLayer are:

- An instance of a controlled clockport, used to clock the DUT. This clock can be stopped by the user as required.
- An instantiation of the DUT, inserting the DUT into the Emulation App environment.
- Instantiation of transactors, translating messages from the software testbench into messages for the DUT.
- Instantiation of the shutdown transactor, enabling the software testbench to stop the emulation or simulation.

The following packages are usually imported into the SceMiLayer:

- `DefaultValue`: to use the provided default clock configuration
- `SceMi`: containing the underlying Emulation App communication definitions
- `DUT`: the package containing the DUT definition.

The example in this section also imports the `GetPut` package containing the `Get` and `Put` constructs used by the interface.

```
package SceMiLayer;

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

import DUT::*;
```

3.3.1 Clocks

There are two types of clocks in the SceMiLayer: the uncontrolled clock and one or more controlled clocks. The uncontrolled clock is a free-running clock used for interacting with the ports and clock controllers. The DUT runs on a controlled clock which can be stopped by the user through transactors during operations. The Emulation App environment also contains a third clock, the FPGA clock or reference clock from the FPGA. The controlled clocks are aligned with the rising edge of the uncontrolled clock. The fastest controlled clock can be at the same frequency as the uncontrolled clock. There is a phase shift between the reference clock of the FPGA and the controlled and uncontrolled clocks defined in the SceMiLayer. See Appendix A.6 for more details on clocking in the SceMiLayer.

You instantiate an instance of the `mkSceMiClockPort` module in the SceMiLayer, defining a clock port for the DUT. In Emulation App we use a default clock configuration to define the clock port parameters. You can change the clock definitions by modifying the individual parameters in the default configuration.

The following statements use the default clock configuration (named `confdut`) and instantiate a controlled clock port (named `clk_port`) using the default configuration.

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

3.3.2 Instantiating the DUT

The DUT must be instantiated in the SceMiLayer using either the `buildDut` or `buildDutWithReset` module. These modules connect the DUT with the instantiated clock port (and reset, if appropriate). The interface returned by the `buildDut` statement is the interface provided by the DUT.

For example, let's assume a DUT defined in the module `mkDUT` providing a `FactServer` interface. The interface contains a `Put` and a `Get` subinterface:

```
interface FactServer;
   interface Put#(UInt#(64)) inRequest;
   interface Get#(UInt#(64)) outResponse;
endinterface
```

The DUT is instantiated without a reset, using the following statement:

```
  DUTServerIfc dut <- buildDut(mkDUT, clk_port);
```

where `dut` is the name chosen for this instantiation of `mkDUT`.

The SCE-MI transactors used to connect the DUT into the Emulation App environment depend on the interface provided by the DUT.

3.3.3 Transactors

A transactor is a form of abstraction gasket which connects messages from the software side testbench with events on the hardware side (FPGA). The transactors decompose un-timed messages to a series of cycle-accurate clocked events, or conversely, compose a series of clocked events into a single message. The transactors can also manage the controlled clocks, stopping and starting the DUT clock as necessary.

A hardware-side transactor may include one or more of the following, as shown in Figure 7:

- input and output message port macros
- one or more controlled clocks
- hardware necessary to store requests and responses, synchronizing the clock domain crossings

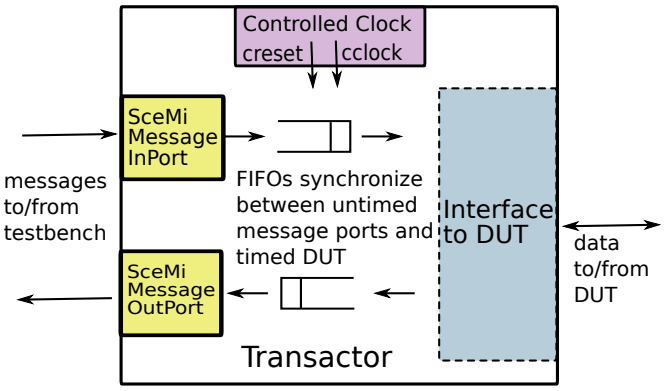

Figure 7: Generic Transactor

The hardware-side transactors are independent of the the emulation host platform and connection scheme. The build layer instantiates the proper platform-specific modules.

The BSV SCE-MI library includes a package of BSV hardware-side transactor modules (`SceMiXactors.bsv`). These transactors are available when you import the `SceMi` package:

```
import SceMi :: * ;
```

The following transactors are provided by Bluespec to instantiate the SceMiMessage ports (`SceMiMessageInPort` and `SceMiMessageOutPort`). They contain FIFOs to store data between the controlled and uncontrolled clock domains but do not stall the controlled clock.

There are two transactors instantiating the inport message port, `mkInPortXactor` and `mkPutXactor`, each providing a different interface type to the DUT. The `mkInportXactor` provides a `Get` interface, and requires a `mkConnection` module to connect it with the DUT. The `mkPutXactor` provides a `Put` interface and connects directly to the `Put` interface of the DUT.

The outport message ports have two correlating transactors: `mkOutPortXactor` and `mkGetXactor`. The `mkOutPortXactor` provides a `Put` interface and requires a `mkConnection` between the transactor and the DUT. The `mkGetXactor` provides a `Get` interface and connects directly to the DUT.

The following examples show two ways of connecting a DUT that provides a `FactServer` interface, defined by a `Put` and a `Get` interface:

```
interface FactServer;
   interface Put#(UInt#(64)) inRequest;
   interface Get#(UInt#(64)) outResponse;
endinterface
```

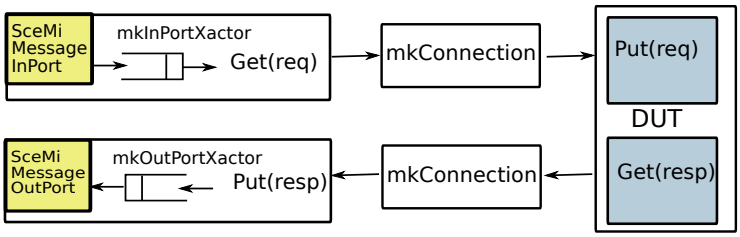

Figure 8: Using mkInPortXactor and mkOutPortXactor

The first example uses the `mkInPortXactor` and `mkOutPortXactor` transactors as shown in Figure 8:

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

Get#(Value) dutin <- mkInPortXactor(clk_port);
mkConnection (dutin, dut.inRequest);
Put#(Value) dutout <- mkOutPortXactor(clk_port);
mkConnection (dutout, dut.outResponse);
```

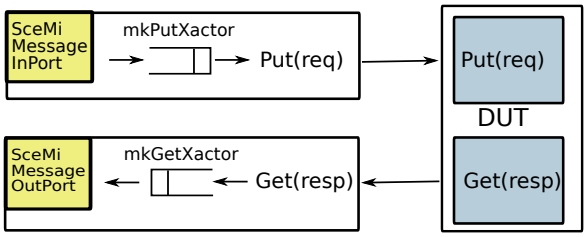

Figure 9: Using mkPutXactor and mkGetXactor

This example uses the `mkPutXactor` and `mkGetXactor` transactors as shown in Figure 9:

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

Empty dutin  <- mkPutXactor(dut.inRequest, clk_port);
Empty dutout <- mkGetXactor(dut.outResponse, clk_port);
```

Complete specifications for the following hardware-side transactors are provided in Appendix E.3. You can modify these transactors or build your own as necessary to implement the functionality required by your design.

- `mkInPortXactor`: Bluespec abstraction of a SCE-MI outport, provides a `Get` interface. Connects to the `Put` interface of a DUT through a `mkConnection` module.
- `mkPutXactor`: Bluespec abstraction of a SCE-MI inport, connects directly to the `Put` interface of a DUT.

- `mkOutPortXactor`: Bluespec abstraction of a SCE-MI outport, provides a `Put` interface. Connects to the `Get` interface of a DUT through a `mkConnection` module.
- `mkGetXactor`: Bluespec abstraction of a SCE-MI outport, connects directly to the `Get` interface of a DUT.
- `mkServerXactor`: Bluespec abstraction of a SCE-MI outport and a SCE-MI inport, connects directly to the `Server` interface of a DUT.
- `mkClientXactor`: Bluespec abstraction of a SCE-MI inport and a SCE-MI outport, connects directly to the `Client` interface of a DUT.
- `mkValueXactor`: Sends a value to the DUT.

The following transactors are not used to interface with the DUT but to allow the software side testbench to control the hardware simulation and/or emulation:

- `mkShutDownXactor`: Allows the software testbench to stop the simulation.
- `mkSimulationControl`: Allows the software testbench to control the simulation.

3.3.4 SceMiLayer Example

Let's continue with a DUT providing a `FactServer` interface.

Two of the standard Emulation App transactors provided are the `mkGetXactor` and the `mkPutXactor`. The `mkGetXactor` connects to the `Get` subinterface provided by the DUT to handle the messages (responses) coming from the DUT, back to the software testbench. The `mkPutXactor` connects to the `Put` subinterface and handles the requests coming from the software testbench. Note that the arguments of the transactors are the names of the subinterface of instantiated DUT (`dut`) and the instantiated clock port (`clk_port`).

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

The final transactor needed is the `mkShutdown` transactor, enabling the software testbench to shut down the emulation or simulation.

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

The complete SceMiLayer file is shown below:

```
package SceMiLayer;

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

import DUT::*;

module [SceMiModule] mkSceMiLayer ();

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

   // Instantiate the dut
   FactServer dut <- buildDut(mkDUT, clk_port);

   // Instantiate the transactors with the DUT
   Empty dutout <- mkGetXactor(dut.outResponse, clk_port);
   Empty dutin  <- mkPutXactor(dut.inRequest, clk_port);

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

3.4 Adding a synthesizable testbench

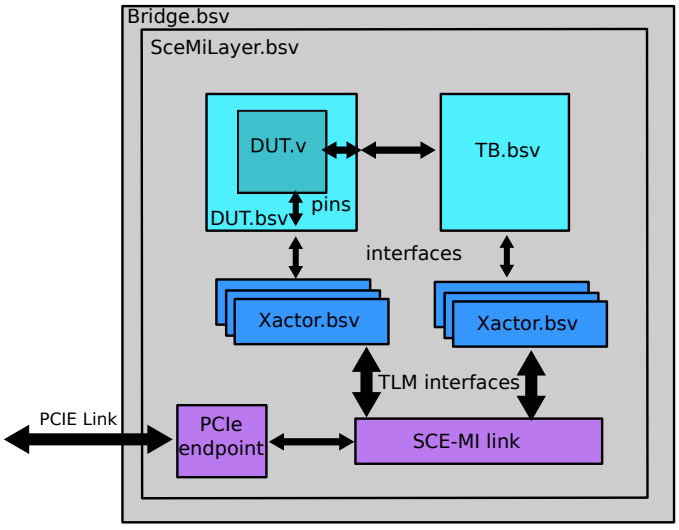

Figure 10: Hardware side of Emulation App

One bottleneck in an FPGA implementation can be the amount of data moving across the PCIe communications link. One way to reduce the traffic across the link is to move all or part of the testbench on to the emulation board.

As you can see in Figure 10, the SceMiLayer can instantiate both the design under test (DUT), as well as a testbench (TB). The testbench can be written in Verilog, and wrapped to provided BSV interfaces, or written in BSV, as shown in the figure. The DUT and the TB communicate directly, and also connect individually through the hardware transactors to communicate through the PCIe link to the host side.

The testbench is partitioned to minimize the data sent over the PCIe link. For example, the host may send a signal to send a packet, but the packet is assembled and sent to the DUT by the TB on the FPGA.

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

3.5 Preparing the bridge layer

The bridge layer, as provided by the file `Bridge_<board>.bsv`, appends the FPGA-specific elements to the design. Copy the provided file into a file named `Bridge.bsv`. This file contains the top module of the project and is where the board specifics are taken into consideration. All the coordination and communication logic is implemented in this layer.

Emulation App is shipped with the bridge file for the selected FPGA board. No user input is required. If a different emulation board is selected, the bridge file is replaced with a new file.

Emulation App currently supports the following emulation boards:

- Dini 7002
- Dini 7006
- Dini 7406
- Xilinx ML605

Adding a new board to this list typically requires some engineering work. Please contact Bluespec support for advice on whether or not the boards you are using are supported.

The bridge layer files for supported emulation boards and for TCP simulations are supplied in the directory `$BLUESPECDIR/board_support/bridges`, including support for designs utilizing DDR2 memory.

For simulations utilizing TCP as the communications link between the software testbench and the hardware DUT, use the bridge file `Bridge_TCP.bsv`.

Throughout this manual we use `Bridge.bsv` to refer to the board-specific bridge file appropriate for your design.

4 Designing the host testbench

The structure of the host-side testbench is extremely flexible; you can use a wide variety of tools and software to write a testbench which communicates through the PCIe link to the FPGA. To develop a testbench in a C++ with a Tcl/Tk-based GUI, Emulation App provides a library of components, including proxies, transactors, and utilities.

This section describes how to generate a complete testing environment on the software side including a GUI front-end to control tests and probes. This environment supports both a testbench contained entirely on the software host, as well as one in which some test components are implemented in the FPGA to minimize communication traffic across the link.

4.1 Software-side Transactors

The bottom layer of the testbench contains software-side transactors which implement the port proxies to communicate with the hardware transactors on the FPGA. Like the hardwre transactors, the software transactor specification is based on the design of the DUT. Each message port from the DUT on the FPGA communicates with the testbench through a port proxy on the software side. The software transactor constructor definition includes a data type and name, which must match the message port data type and name in the SCE-MI paramters file, `mkBridge.params`.

In this section we'll examine the components of a software transactor, using as an example a transactor named `GetPutXactor` which communicates with a DUT providing a Get/Put interface on the hardware side.

The transactor is defined in two files: the header file (`GetPutXactor.h`) and the source code file (`GetPutXactor.cpp`).

4.1.1 Header file

To use the Bluespec provided C++ classes, the header file must include the file `bsv_scemi.h`.

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

The header file contains the declaration of the transactor class, where the class name is the same as the transactor name. The software transactors, which contain the message port proxies, are defined within the file. There are three types of transactors commonly contained within the software transactor. The first two are data transactors, where the direction of the transactor is from the view of the FPGA. The input data transactors are for data being sent *in* to the FPGA (and out from the testbench) while the output data transactors are for data coming *out* from the FPGA (and into the testbench). The data types of the transactors must match the data types of the hardware-side message ports. A third transactor type, a shutdown transactor, allows the software testbench to stop the emulation.

The data types of the software transactors must match those defined by the hardware transactors, as found in the `.params` file generated by the infrastructure linking tool scemilink. The default file is called `mkBridge.params`. If the HDL editing tool is used (Section 7) the default file name is `mkBridge_edited.params`.

In this example there are three transactors: a data in (`m_datain`), a data out (`m_dataout`) and a shutdown transactor (`m_shutdown`). `m_datain` is defined as a member of the Bluespec-provided inport transactor class, `InportProxyT`, with a data type of `BitT<64>`. `m_dataout` is defined as a member of the the Bluespec-provided outport transactor class, `OutportQueueT`, with a data type of `BitT<64>`. `m_shutdown` is a member of the Bluespec-provided shutdown transactor class.

```
// Define a class for the top-level transactor
class GetPutXactor {
 protected:
  // Data Xactors
  InportProxyT<BitT<64> >         m_datain;
  OutportQueueT<BitT<64> >         m_dataout;
  // Shutdown Xactor
  ShutdownXactor                 m_shutdown;
```

The input is defined as the input into the hardware-side DUT: it *puts* a request into the DUT. The output *gets* a response from the DUT. The Bluespec-provided C++ class `InportProxyT` has methods for blocking and non-blocking versions of `sendMessage` method. The provided C++ class `OutportQueueT` has both blocking and non-blocking methods for `getMessage`. These methods are used to define the `putRequest` and `getResponse` methods in the transactor. The methods are declared in the `GetPutXactor.h` header file, while the full definition is in the source code file `GetPutXactor.cpp`.

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

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

 void shutdown();
```

4.1.2 Source code file

The constructor code for the transactor contains the method definitions for software-side port proxies which communicate with the hardware-side message ports. The names in the definition must match the message port names in the SCE-MI paramters file, `mkBridge.params`. These names are generated by the infrastructure linkage program and are based on the names of the hardware transactors.

```
GetPutXactor::GetPutXactor(SceMi *scemi)
  : m_datain  ("", "bsci_dutin_inport", scemi)
  , m_dataout ("", "bsci_dutout_outport", scemi)
  , m_shutdown ("", "bsci_shutdown", scemi)
```

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

The remainder of the file defines the methods.

```
// Non-blocking put request
// Send the request to the DUT
bool GetPutXactor::putRequestNB (long request)
{
  BitT<64> reqbit(request);
  bool sent = m_datain.sendMessageNonBlocking(reqbit);
  return sent;
}
// Blocking put request
bool GetPutXactor::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 GetPutXactor::getResponseNB (long &resp)
{
  BitT<64> respbit;
  bool gotone = m_dataout.getMessageNonBlocking (respbit) ;
  if (gotone) {
    resp = respbit.get64();
  }
  return gotone;
}

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

4.2 Testbench

The purpose of the testbench is to validate the DUT on the FPGA. Section 4.1 discussed how to write a software transactor to handle communication to the FPGA. In this section, we'll use the software transactor in a very simple testbench. In Section 5 we'll look at a more complex, GUI-based testbench.

The testbench file (`Tb.cpp` in this example), must include the Bluespec-provided C++ classes along with the transactor for the DUT, defined in the previous section.

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

The first part of the file initializes the SCE-MI communication channel:

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

The testbench starts and runs the SCE-MI service thread.

```
 SceMiServiceThread scemi_service_thread (sceMi);
```

When complete, the `shutdown` transactor is used to stop the simulation, and then the service thread is stopped.

```
  //Stop the hardware side
  getx.shutdown();

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

The tests are defined in the file.

```
void runGetPutTest ( class GetPutXactor &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;
   }
}
```

The next section will describe how to use the software transactor `GetPutXactor` within a testbench with a GUI for control and display.

5 Implementing a GUI testbench

You can easily add a front-end GUI to control your emulation or simulation using a tcl layer and components shown in Figure 11.

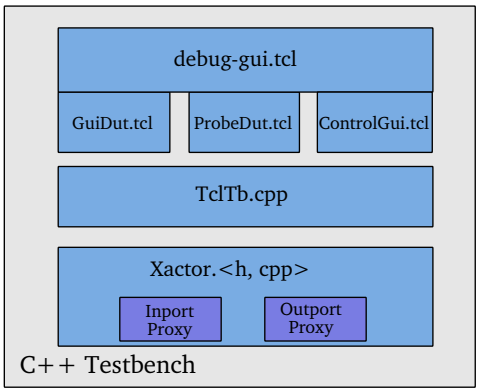

Figure 11: Host-side C++ Testbench

5.1 Components

The emulation GUI window has three sections, shown in Figure 12:

- Emulation Control - allows the user to start and stop the emulation clocks. This section is defined by the `ControlGui` tcl library package, provided in Emulation App. There is no user modification required to use this package.
- DUT control - a DUT-specific section where the user controls sending requests and receiving responses from the DUT. You can also send a software reset or perform other test task in this section. Messages received from the DUT are displayed in the **Messages** window. This functionality is provided in the `GuiDut` package (`gui_dut.tcl`) which must be modified based on the DUT and test design.
- Probe Control - in this section the user controls the wave viewer and all probe funtionality, as defined in the `ProbeGui` package. No user modification is required to use this package.

The files `ControlGui.tcl` and `ProbeGUI.tcl`, used to build the graphical interface, are provided in the directory `$BLUESPECDIR/tcllib/scemi`. These files are not dependent on the testbench design; they do not change based on DUT-specific inputs and outputs.

The Tcl/Tk file `debug_gui` contains the commands to setup the GUI environment. Edit this file to select a wave viewer, set the wave viewer options, load and setup the Emulation App GUI control window, and configure the software side of the environment. The file assumes the parameters file is the default `mkBridge.params`, but you can edit the file to specify a different name.

5.2 Customizing the DUT control section

Most of the DUT-specific modifications for the GUI are in the file `gui_dut.tcl`, which defines the DUT control section. In this file you define the inputs and outputs for the emulation, the variables for each action, the types of operation (puts or gets), labels and actions. The sample GUI shows a *put* button, a *get* button and a message box to display messages received by the host.

5.3 Testbench

Bluespec provides a C++ template for the testbench file utilizing the GUI interface. This file, `TclTb.cpp`, is the software testbench, initializing the software testbench transactor along with the GUI components. The provided templated must be modified for the characteristics of the DUT and the software-side transactors of your design.

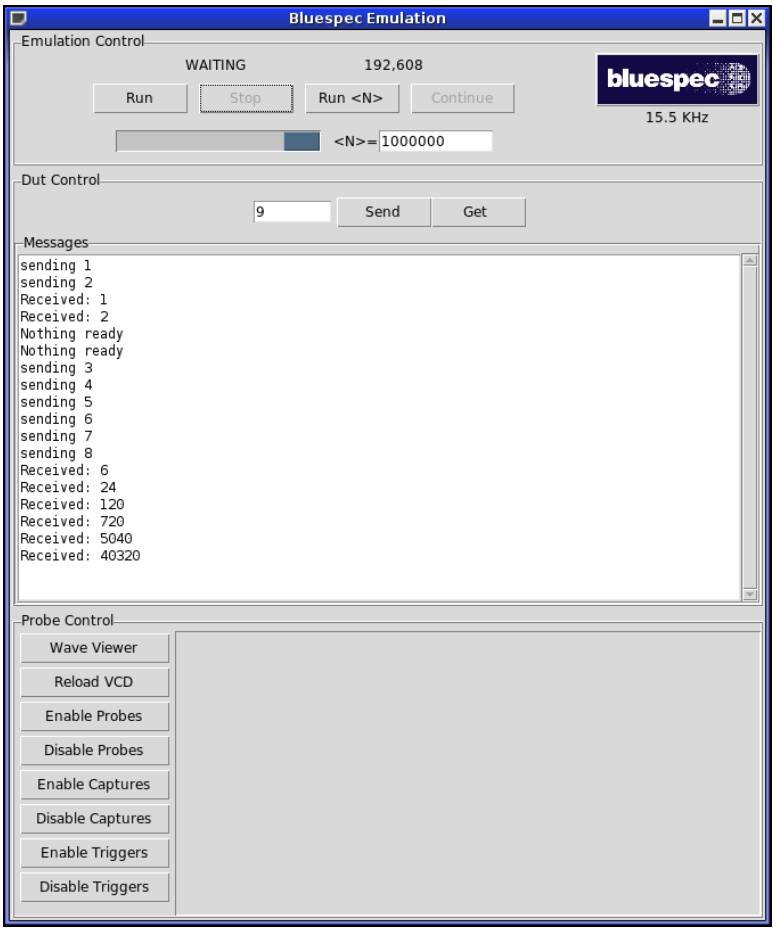

Figure 12: Simple Testbench GUI

5.4 Hardware-side requirements

The GUI front-end controls the simulation from the software side through the `SimulationControl` software transactor. This requires that the hardware side has an instance of `mkSimulationControl` instantiated in the `SceMiLayer` module. Edit the `SceMiLayer.bsv` file to add the following statement, where `confdut` is the name of the `SceMiClockConfiguration` in your design.

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

5.5 Building the GUI

The build utility faciliates building the components for a complete GUI-based emulation testbench. The following target is an example of the directives used to build the GUI testbench. All testbench files, including `TclTb.cpp` and the software transactor definitions (`GetPutXactor.cpp` in this example) must be listed in the directive `c++-files`.

```
[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 GetPutXactor.cpp $BLUESPECDIR/tcllib/include/bsdebug_common.cpp
```

5.6 Running the GUI

Once the project is built, execute the file `debug_gui` to start the GUI. The design must already be running in either emulation or simulation.

6 Running the environment

6.1 Emulation

Most of the configuration for the Emulation App system for a specific emulation board is administered through directives in the `project.bld` file. Within the file you can specify directory structures as well as options for Quartus and Xilinx tools. For a complete list of build directives, including FPGA options, see Appendix C.

Emulation App uses PCIe as the interface between the host testbench and the emulation board. Once the `build` process is complete, load the FPGA bit files on to the emulation board. The process for loading and initialization is board-specific. Some boards require a reboot of the host machine (the host machine must see the PCIe board at boot), while others may allow you to hot swap the board. Refer to your emulation board vendor documentation for instructions.

6.1.1 Dini tools

Emulation App includes the following utilities to load the synthesis files generated by the `build` process onto the dini board through a usb connection.

- `7002`: Program to load bit files on to dini 7002 board through usb
- `7006`: Program to load bit files on to dini 7006 board through usb
- `7406`: Program to load bit files on to dini 7406 board through usb

These utilites are found in the directory `$BLUESPECIR/board_support/dini/tool`. Run `make` to build the utility programs. The following package must be installed on your machine:

```
libncurses5-dev
```

6.2 Simulation

You can setup your `project.bld` file to include targets for simulation. In simulation, the connection type will be TCP. Since the bridge layer defines the platform-specific details of the design, you will use a different bridge file `Bridge.bsv` for simulation than for emulation. That should be the only change required to the design. Emulation App includes a bridge file for simulating over TCP, `Bridge_TCP.bsv`. As with the other bridge files, it is located in the directory:

```
$BLUESPECIR/board_support/bridges
```

Build the project, specifying the simulation target for the dut. The testbench will often be the same for both simulation and emulation. To run the simulation, the TCP environment requires two separate processes: the DUT running on a simulator (either Bluesim or a Verilog simulator) and the testbench running in C++. The DUT must be started first.

1. Build the project, selecting a simulation target
2. Start the DUT, by executing the `.exe` file generated by the build in the DUT target
3. In a new shell, start the testbench, by executing the file generated by the build in the testbench target or by executing `debug_gui` to start the GUI

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

7 Probes

Probes provide a way to send signals from Emulation App to a waveform viewer. There are two classes of probes: value probes continuously acquire data each cycle while capture probes collect and store data until a specific event, a *trigger*, occurs. With both types of probes the data is written to a VCD file during simulation or emulation, which is viewed through a waveform viewer. The probes can be controlled through the GUI discussed in the previous section.

Probes are defined with the HDL Editor, an optional step run after the HDL file is generated and before the design is loaded onto the emulation board. Through the GUI-based editor you define probes and make other HDL-specific changes. You can use the tool to modify signal names and handle other tedious edits in a controlled manner. When directed, the utility generates a new HDL file and new parameter (`.params`) file. The definitions and changes can be saved in a script file (`<file name>.script`) and used in subsequent runs to modify the generated HDL files in a consistent manner.

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

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

These steps are implemented through the **build** utility. No changes are made to the source design and there is no additional design work or restrictions on the design to add probes. The HDL editor will generate a single message port for all probes. The hardware-side probe transactor is built automatically by the `buildDUT` module. When running the emulation or simulation, the testbench writes out a VCD file once all the probe data is collected. This technique is scalable and can be used with multiple FPGAs.

7.1 HDL editor objects

The following objects are currently supported in the HDL editor:

- Value Probe: Continuously captures and sends data across the communication link.
- BSV Probe: Captures signals defined using the BSV `mkProbe` statement (`$PROBE` signal in the Verilog).
- Capture Probe: Captures and stores data until a trigger event occurs, sending data immediately preceeding and after the trigger event across the communication link at one time.
- Probe Trigger: Defines an event on which a capture probe is based. The trigger definition is distinct from the signal definition and can reside in a different module.
- Cosim Probe: Generates scan data for a co-simulation, instead of signal data. You specify a module to be co-simulated and, optionally, a trigger for the co-simulation.

The software testbench captures the data sent by the probe into a VCD file, which can be viewed and reported on. Since a value probe is continuously sending data, there is higher data traffic which requires stopping the clock and may slow down the system. A capture probe stores the signal data in a BRAM in the emulation system until the trigger event occurs and for a set number of clock cycles after the trigger event. The data is sent only after the data collection is complete, generating less data traffic across the link. The system waits until all data has been sent before writing the probe data into a single VCD file. Even though the value probes are continuously sending data, the VCD file will not be written until all capture probes have completed sending their data.

7.2 Invoking the HDL editor

The HDL editor will usually be invoked from the build script, but can also be invoked from the command line, as described in Section D.2.

To invoke the tool during the build process, the `run-design-editor` directive must be set to `True` in the `project.bld` file. Below is a sample of common build file settings for the design editor using the GUI interface and loading a script:

```
run-design-editor: True
design-editor-output-directory: vlog_edited
design-editor-edit-params: True
design-editor-options:  --gui
design-editor-script:   <scriptname>.script
```

7.3 Overview of the editor

The HDL editor screen, shown in Figure 13, is divided into three section:

- Design hierarchy: Complete module hierarchy from the HDL file
- Signals: All signals for the module selected from the design hierarchy
- Design Edits: Probe definition and other design edits as defined in the loaded script or through the GUI

The **Design Hierarchy** frame displays the complete module hierarchy read from the HDL file. Click on a module to display all the signals for the module in the **Signals** frame.

You can filter by name the signals displayed for a given module, using the **Filter** field in the **Signals** frame. For example, if you want to display only signals containing the string `data`, you would enter `data*` in the **Filter** field, as shown in Figure 14.

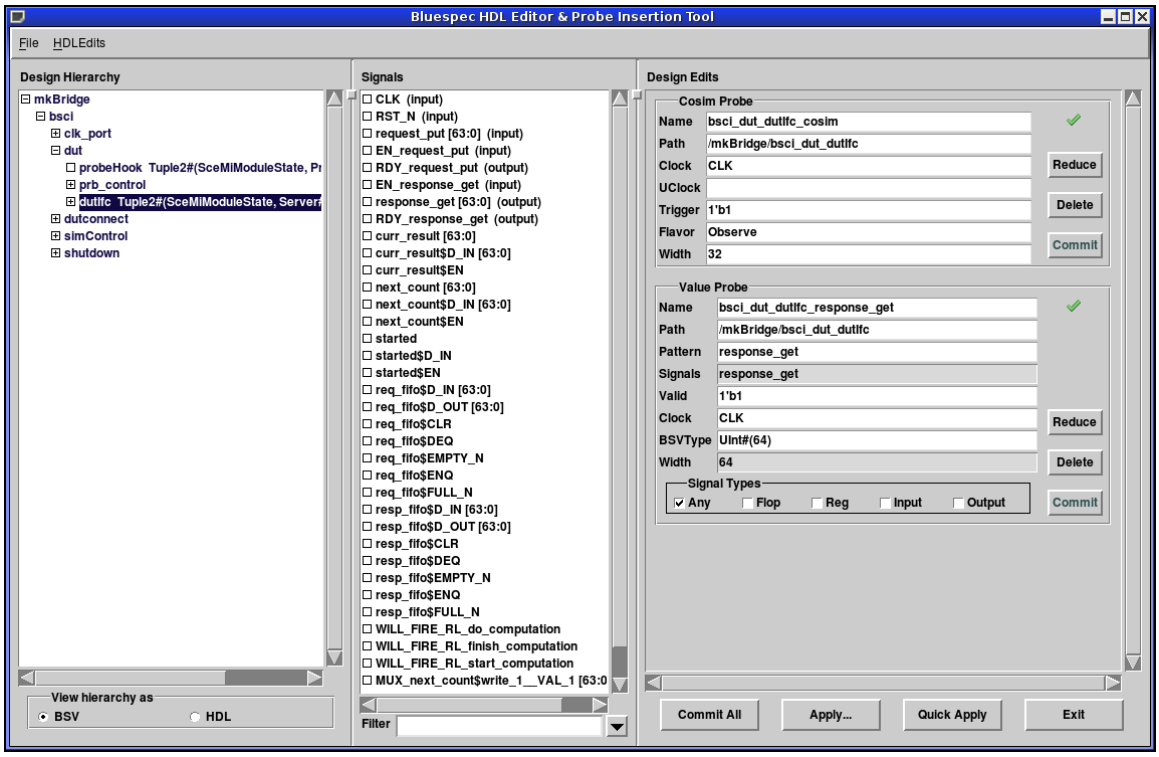

Figure 13: HDL Editor

7.4 Adding objects

7.4.1 From the menu bar

To add a probe from the menu bar, select **HDLEdits→*object type***. It will create a probe or trigger of the selected type. A single probe can include multiple signals.

You must specify a name for the probe, along with which signals (one or more) are included in the probe. Value, BSV, and Capture probes allow you to create a probe based on signal type, where the types are `Any, Flop, Reg, Input, Output`. You can also use pattern matching with regular expressions to select the signals. All signal names matching the pattern and the signal type are included in the probe.

When multiple signals are in a probe, the width is the sum of the width of all the signals.

7.4.2 For a selected signal

To add a probe on a specific signal:

1. Select a module from the Design Hierarchy
2. Select a signal from the Signals list
3. From the menu bar, select **HDLEdits→*object type***

The tool will fill in the selected signal name, pattern, and BSV type fields. The signal type will default to `Any`. The name will default to the concantenation of the path and the signal name. You can change this to a more meaningful name.

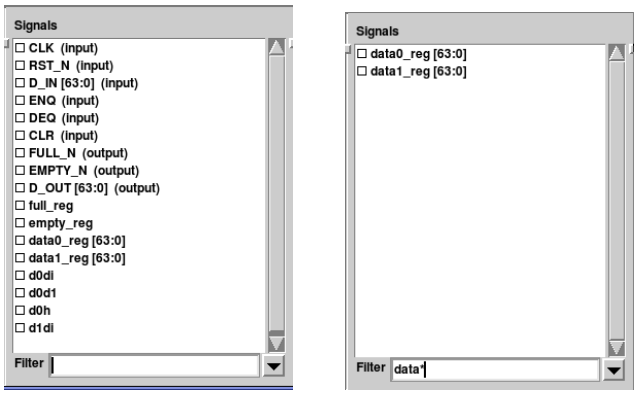

Figure 14: Signal Filter

7.4.3 Value Probe

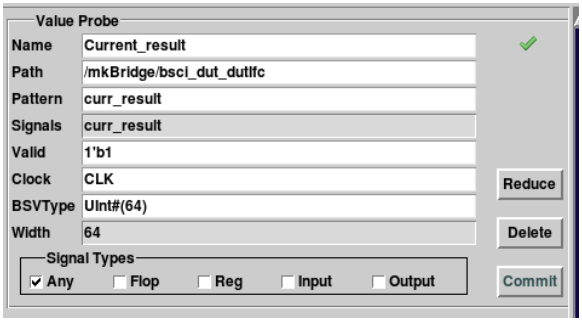

Figure 15: Value Probe Design Edits

A value probe continuously captures and sends the value of a signal across the link to the software testbench. For a probe on a single signal, the name of the probe is creating by combining the path and the signal name. It is recommended that you replace the default name with a more meaningful name. For a probe made up of multiple signals, you must define the name. Each probe must have a unique name. To avoid capturing noise, the probe has a `Valid` field, defining when the data values of the signal are valid. By default, this value is set to be always valid, but can be modified to be any single-bit expression in the module.

7.4.4 Capture probe

A capture probe captures and stores the values of one or more signals in a BRAM on the emulation board (or in memory during simulation). The data from each capture probe is stored in its own BRAM until the trigger event occurs and for a set number of clock cycles after the trigger event. The data is sent only after data collection is complete, generating less traffic across the link. The system waits until all data has been sent before writing the probe data into a single VCD file. Until the data is sent, the capture probe does not impact the controlled clock of the DUT.

In specifying a capture probe the designer makes trade-offs between the amount of data captured and the size of the BRAM. The more frequently data is captured, the larger the BRAM must be to contain the data.

RunWidth The amount of data captured depends on the RunWidth parameter, where RunWidth $= \log_2 cycles$. Capture probes use the technique of run length encoding to compress the data collected. The RunWidth is measured is bits and determines how often the data is sampled. The more bits in the RunWidth, the less frequently the data is sampled, and the more the data is compressed. If your data changes frequently, the RunWidth will be small. For example if you expect your data to change every cycle, the RunWidth $= \log_2 1 = 1$. If the data changes every 1000 cycles, then the RunWidth $= \log_2 1000 = 10$.

A capture probe is defined by all the fields in a value probe along with the following fields:

Capture Probe Field Definitions		
Parameter	Description	Default Value
Trigger	A one bit expression in a module defining the point in time when data should start being collected. The expression must be in the module.	`1'b0`
Depth	The size in number of words of the BRAM storing the probe data	32
RunWidth	Width of a run length of code, where $RunWidth = \log_2 cycles$. Run length encoding is used to compress the data. The run length is measured in bits and determines how much the data is compressed. The more bits in the run length the more the data is compressed. The more frequently the data changes, the less compression you can have. If you expect your data to change every cycle, the run length would = 1. If the data changes every 1000 cycles, then the runwidth $= \log_2 1000 = 10$.	4
DumpDelay	Number of clock cycles after the trigger to continue to collect data before dumping the data into a vcd file	50

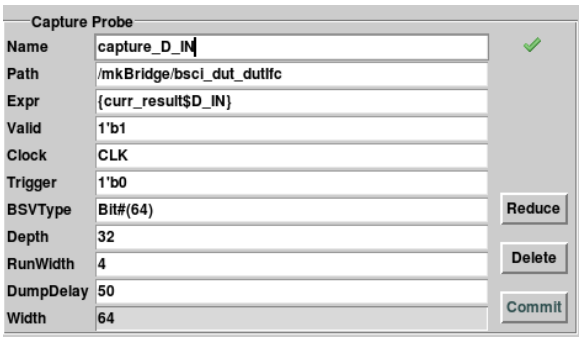

Figure 16: Capture Probe Design Edits

The capture probe data is collected when either

- The data changes OR
- The RunWidth counter rolls over

7.4.5 Probe trigger

When the value or expression triggering a capture probe is not in the same module as the signal being captured, a **Probe Trigger** must be defined. When the expression is True, the capture probe (or probes) are triggered. A probe trigger may trigger one or more different capture probes. A capture probe may be triggered by one or more probe triggers.

Probe Trigger Field Definitions		
Data Field	Description	Default Value
Name	The default name is generated by combining the path and signal names; you can replace the default name with a more meaningful name.	Path+Signal
Path	Full path of the module in the design hierarchy	
Expr	Name of the signal that when True, triggers the probe. Must be a single bit expression.	
Clock	The clock signal for the expression. Modify this field if the clock name is not CLK.	CLK
Captures	The names of the capture probes triggered by this probe trigger.	

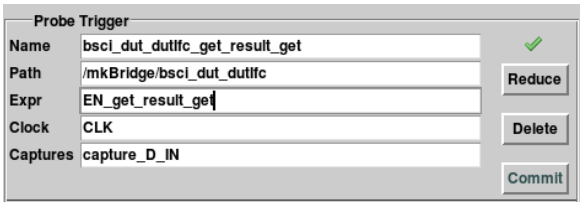

Figure 17: Probe Trigger Design Edits

7.5 State indicator

The icon in the upper right corner of the design edits section for each probe out indicates which of three edit states the probe is in:

- * yellow star: the probe has not been committed or changes have been made to the probe which are not yet committed
- ✓ green check: the probe and all changes are committed
- ! red exclamation point: an error was found in the edits and they have not been committed.

The **Commit All** button at the bottom of the window verifies and commits changes for all objects.

7.6 Saving the changes

When you are done editing the HDL file, there are three types of changes to save: the HDL files with the probes added, an updated SceMiParams file, and a replay script file containing the probe definitions.

- HDL files: the new Verilog files are written to the specified output directory
- SceMiParams: A new SceMiParams file contains the additional object definitions
- Replay script: saves the defined design edits in a script that can be applied to future compilations of the design.

There are four buttons at the bottom of the HDL editor screen, as shown in Figure 18:

Figure 18: Save and Exit Options

- **Commit All** verifies and commits changes for all objects; the changes are not yet saved to a file
- **Apply...** brings up the **Apply Changes Dialog** window, shown in Figure 19. From within this window, the **Save to Script** button saves the script, while the **Apply** button writes out the new HDL files, and if selected, a modified SceMiParams file as well.
- **Quick Apply** applies all the changes, writing out a new HDL file, saving the script, and updating the SceMiParams file, use the defaults specified in the `project.bld` file. This button does not bring up the dialog window, but will allow you to save all the changes at one time.
- **Exit** to exit the editor. If the HDL edits have not been applied to the source, a **Confirm Exit** dialog box will be displayed.

Apply changes dialog From within the **Apply Changes Dialog**, shown in Figure 19, you can change file locations and file names, and select which components to save.

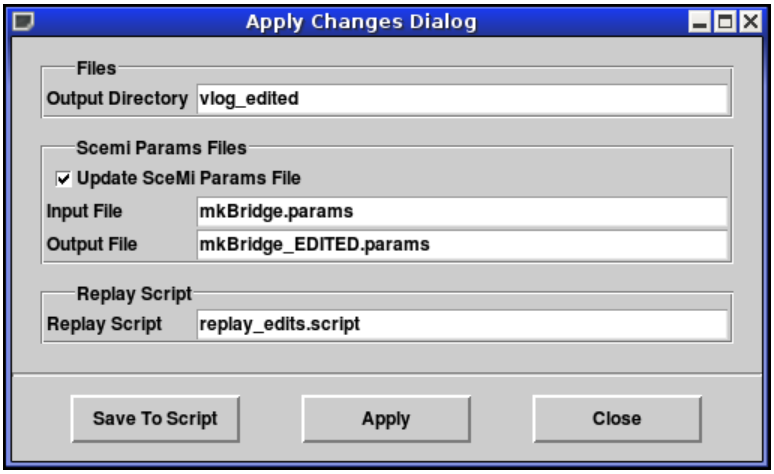

Figure 19: Applying Changes

The **Apply** button adds the defined probes to the HDL files. The new files are in directory specified by the **Output Directory** field; the default directory is `vlog_edited`.

If the **Update SceMiParams File** check box is selected, a new SceMiParams file is written, adding in the probe definitions. The default name for the edited SceMiParams file is `<topmodule>_Edited.params`, such as `mkBridge_Edited.params`. Use the **Output File** field to specify a different file name. If a name other than the default name is used for the SceMiParams file, name must be specified with the `design-editor-output-params` directive in the build file.

To save the probes you have defined in a session, use the **Save to Script** button. This saves all the definitions of probes and triggers, to be used in future sessions. The default name of the script file is `replay_edits.script`, or you can enter a different name in the **Replay Script** field. This file will be used when the HDL editor is run in batch mode, or loaded when the editor is run in GUI

mode. If a file name other than the default name is used, the `design-editor-script` directive in the build file must be specify the file name.

7.7 Viewing the probes in the GUI testbench

To define and view probes within the GUI testbench discussed in Section 5 you must modify the `[dut]` target of the `project.bld` file to:

- Invoke the HDL editor (`run-design-editor`)
- Use the edited `.params` file (`design-editor-output-params`)

No changes are made to the software testbench or any other component of the system.

The following directives direct the build process to run the HDL editor and specify the files to be used.

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

When running the HDL editor in batch mode, use `design-editor-script` to specify the name of the script.

8 Co-Simulating a design

Cosim combines the speed of FPGA prototyping with the detailed visibility provided by simulation. After running many cycles on the FPGA, you can add a cosim probe on a module to to pull it into simulation for more detailed analysis.

1. Enable cosim
2. Probes added on to each state eleement
3. Initial state of all state elements recorded
4. Every cycle, any changed in inputs recorded
5. Values recorded in `.scd` file (Default name: `scemi_test.scd`)
6. Process `.scd` file to bgenerate `scemi_test.vcd`
7. View `.vcd` file in a waveform viewer

With co-simulation (cosim) you can simulate part of a design while emulating the complete design in the FPGA. Using the HDL editor and the same GUI testbench developed in Section 5, you add a cosim probe on a module and control it through the GUI. The process and tools work the same as with other probes in Emulation App. Instead of probing a value, a cosim probe pulls a module out of emulation and runs it in simulation, allowing easier debug of the module.

8.1 Adding a cosim probe

You select the module to co-simulate by adding a cosim probe on the module through the HDL editor, described in Section 7.1. While you can invoke the HDL editor directly from the command line, the more common methodology is to use the `project.bld` file.

You only need to define the cosim probe once through the HDL editor. Subsequent builds use the saved replay script to modify the HDL file, adding the cosim probe on the new version of the build files.

The recommended `build` settings, using a saved script and running the editor in batch mode, are:

```
run-design-editor: True
design-editor-output-directory: vlog_edited
design-editor-edit-params: True
design-editor-options:  --batch
design-editor-script:  replay_edits.script
```

8.2 Running the simulation

The GUI testbench will automatically display and control cosim probes once they are defined through the HDL editor. No changes are required to the testbench or GUI definition. After running the `build` utility to build both the hardware and testbench components, start the emulation and the testbench as you would any Emulation App design. The GUI will include the cosim probe, as shown in Figure 20.

Through the GUI, you can enable the cosim probe or run the simulator. Cosim replaces the selected module on the FPGA with the version running in simulation.

8.3 Using the workstation with cosim

As part of building the hardware side, the `build` process will generate a Bluespec project file, used by the Bluespec Development Workstation (BDW). You can use the BDW as described in the *Bluespec SystemVerilog User Guide* to analyze the module, viewing waveforms, scheduling graphs and module information. You will only be able to view the waveforms of the module being cosimulated.

To open the BDW and load the cosim module, at the command line type:

```
bluespec cosim.bspec
```

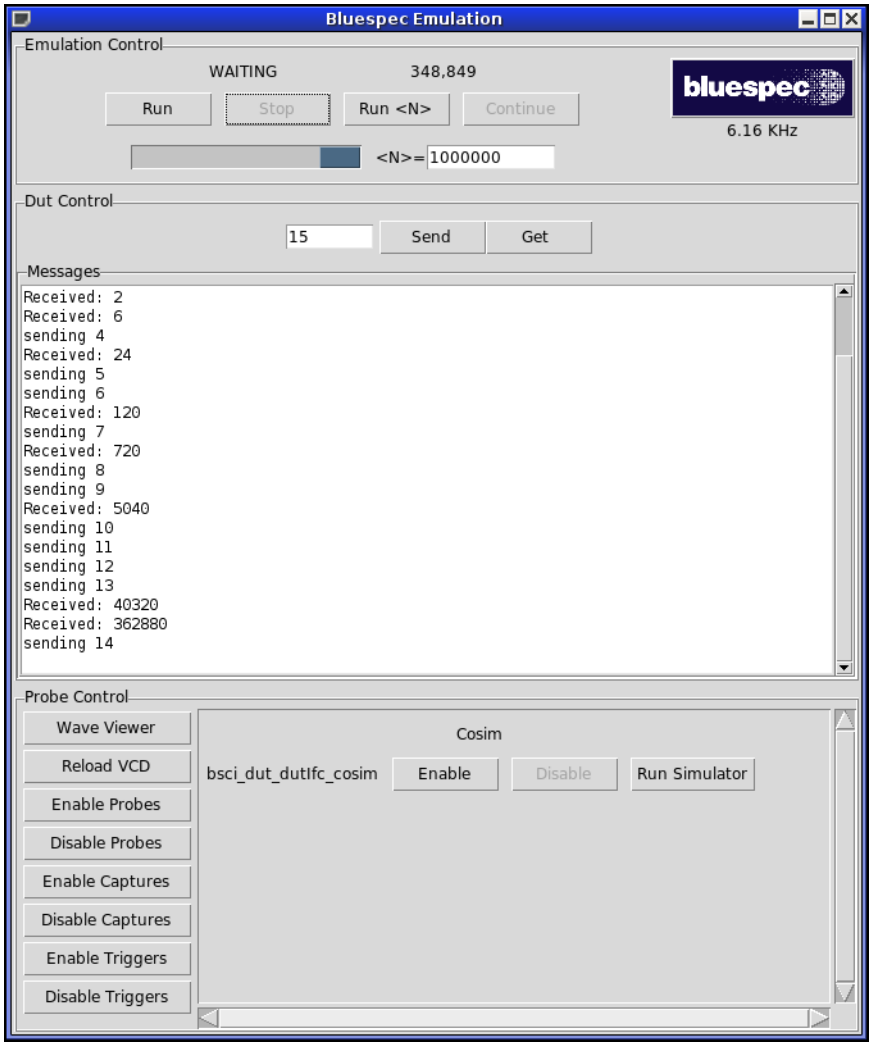

Figure 20: Testbench GUI with cosim probe

A Bluespec SCE-MI implementation overview

This section describes Bluespec SystemVerilog (**BSV**) support for co-emulation in the style of the SCE-MI standard (available from Accellera at the website http://www.eda.org/itc).

Bluespec provides a complete implementation of the SCE-MI standard, including transactors for the software testbench in both C++ and BSV, a complete implementation of the hardware transactors in BSV, and a linkage tool for multiple emulation platforms.

BSV SCE-MI support provides:

- Access to an industry-standard co-emulation API
- Seamless migration of a design across supported co-emulation hosts
- Ability to write transactors in BSV
- Ability to use a standard SCE-MI un-timed C/++ testbench with a BSV design in simulation

- Hardware abstraction of physical connections for deployment flexibility
- Flexible generation of co-emulation link infrastructure based on custom scenarios

A.1 Methodology

A SCE-MI-style co-emulation 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 (DUT) and a number of transactors which handle communication between the software side and the DUT and control clocking of the DUT. 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.

There are two types of clocks in a SCE-MI system: the uncontrolled clock and one or more controlled clocks. The DUT runs on a controlled clock, in which the edges of the clock are controlled SCE-MI infrastructure surrounding the DUT. The emulation hardware is operated on an uncontrolled clock. The uncontrolled clock is used only within the transactor modules and is not exposed to 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 a transactor the software side. The SCE-MI 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 (ILT). Bluespec supports both vendor-supplied and Bluespec linkage tools, supporting most custom scenarios.

A.2 Bluespec SCE-MI hardware environment

The hardware side has three layers, as shown in Figure 21:

- Design under test (DUT): *SCE-MI and platform independent*
- SCE-MI Layer: *emulation platform independent, DUT-specific*
- Bridge: *design independent*

At the heart of the hardware side is the design under testing (DUT). The DUT is a typical BSV design; it must be fully synthesizable and must explicitly specify the module type (`Module`). Otherwise there are no particular restrictions or requirements. It can include any combination of BSV, Verilog, and other RTL components. No special code is added to the DUT when using it within a SCE-MI environment.

The second component is the SCE-MI layer, which is composed of transactors and instantiates the DUT. The transactors handle communication to and from the DUT through the message ports, control start-up and shut-down, and control the clocking of the DUT. The SceMiLayer bridges the un-timed transactions on the SW side with the cycle based execution of the DUT. For many designs, the SCE-MI layer is independent of the emulation host platform and the hardware configuration.

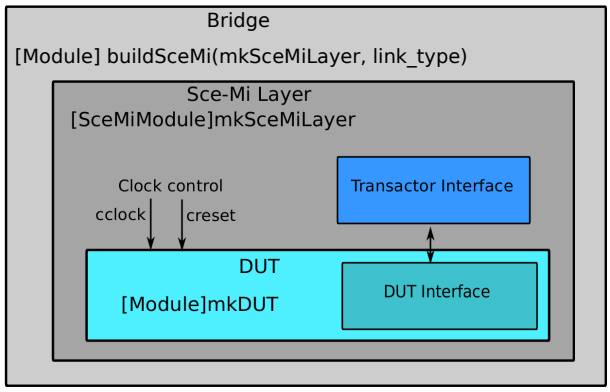

Figure 21: Hardware Overview

Bluespec provides a rich library of transactors, which can be used as written, or modified by the user. Or you can write your own transactors.

The platform specific bridge layer is the third component. The bridge includes the SCE-MI builder (Section A.5.2) which processes the platform-independent transactors, adding in emulation and platform-dependent components. The SCE-MI builder instantiates different modules depending on the hardware platform, to implement the SCE-MI message ports and clock functions. The designer is insulated from the implementation details of the emulation platform; these details are handled by the SCE-MI builder and linkage tool.

The Bluespec release contains BSV library packages to support co-emulation in a SCE-MI environment. To access the co-emulation support library, import the `SceMi` package:

```
import SceMi :: * ;
```

The `SceMi` package is found in the `$BLUESPECDIR/BSVSource/SceMi` directory, along with other BSV SCE-MI packages.

A.3 Object types in a SCE-MI design

A.3.1 Module types

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

The default BSV module type is `[Module]`, which, when instantiated, adds state elements and rules to the design. The module type `[SceMiModule]` is defined to handle collections of SCE-MI-related items in addition to state elements and rules. The hardware transactors and other elements of the SceMiLayer are module types of `[SceMiModule]` to manage the clock ports, message ports, and other SCE-MI components.

The following types of modules are commonly used in a Bluespec SCE-MI system, though designs may define additional types:

- `[Module]`: the default BSV module type. This is the type used for the DUT, except when the DUT has been annotated with probes.

- `[SceMiModule]`: module type of transactors, to accommodate the infrastructure supporting SCE-MI clock ports, clock controllers, and message ports on the emulation platform. This is the type used in the SceMiLayer.
- `[ProbeModule]`: module type of DUTs containing probes, when probes are defined within the DUT. Bluespec provides another method of defining probes using the HDL editor, described in Section 7, which does not modify the DUT.

Only modules of type `[Module]` are synthesizable. One function of the `SceMiBuilder` typeclass described in Section A.5.2, is to convert modules of type `[SceMiModule]` to type `[Module]` for synthesis.

Hardware Layer	Allowed Module Type	Synthesizable?
DUT	`[Module]` `[ProbeModule]`	✓
SceMiLayer	`[SceMiModule]`	
Bridge	`[Module]`	✓

The module type is specified in the module definition statement. When defining a DUT or transactor, add the module type, (`[Module]`, `[SceMiModule]`, or `[ProbeModule]`) between the keyword `module` and the name of the module. Example:

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

The top-level module of a DUT in a SCE-MI system must explicitly state the module type, which for most modules will be `[Module]`. When DUT modules have probe annotations, the module type is `[ProbeModule]`.

SCE-MI transactors must be of the module type `[SceMiModule]` to support the additional infrastructure required by SCE-MI.

A.3.2 Link types

One advantage of a SCE-MI-style system is portability of the design to multiple hardware environments. The variable `SceMiLinkType` is an enumeration of all Bluespec-supported emulation host platforms. At the top level of the design, the link type is declared, allowing the SCE-MI modules to generate the correct code for the selected implementation. No other part of the design is impacted by the choice of link type.

The following table shows the link types currently supported by Bluespec; the list will expand in future releases.

Link Type	Emulation Platform
`SCEMI`	Any SCE-MI-compliant emulation platform
`TCP`	Bluesim or Verilog simulation, via TCP socket
`PCIE_DINI`	Dini Group's emulation boards, via the PCIE bus
`PCIE_VIRTEX5`	Xilinx Virtex-5 boards, via the PCIE bus
`PCIE_VIRTEX6`	Xilinx Virtex-6 boards, via the PCIE bus

```
typedef enum { TCP
             , SCEMI
             , PCIE_VIRTEX5
```

```
, PCIE_VIRTEX6
, PCIE_KINTEX7
, PCIE_VIRTEX7
, PCIE_DINI
, UNDEFINED
} SceMiLinkType deriving (Eq,Bits);
```

A.4 Message port macros

The hardware side of the message channel is defined as a message port. The hardware transactors use these ports to access messages being sent and received from the testbench.

The `SceMi` package provides an input message port and an output message port definition, each parameterized by the type of the message payload. Each type of message port definition includes an interface definition and the definition for the module providing the interface.

The BSV message port modules are of type `SceMiModule`. When using message ports in a design, they are independent of the communication channel and emulation hardware. The message ports are converted to the specific emulation environment by the SceMiBuilder modules.

Transactors, described in Section 3.3.3, use the message port macros. You can use message ports through the Bluespec-provided transactors or when writing your own transactors.

A.4.1 SceMiMessageInPort

The `mkSceMiMessageInPort` module exposes the `SceMiMessageInPortIfc#(msg_type)` interface, which conforms to the SCE-MI standard for an in-port macro.

SceMiMessageInPortIfc		
Method		
Name	Type	Description
`request`	`Action`	A request sent to the software side indicating the hardware is ready to accept new input.
`request_pending`	`Bool`	Returns `True` when a request has been made; another request cannot be made.
`read`	`msg_type`	Returns the data read.
`ack`	`Action`	Acknowledges the data has been read.

```
interface SceMiMessageInPortIfc#(type msg_type);
   method Action request();
   method Bool request_pending();
   method msg_type read();
   method Action ack();
endinterface: SceMiMessageInPortIfc
```

A request is sent by the `request` method, indicating that the hardware side is ready to receive new input. The `request_pending` method will then return `True`. The message data is accessed via the `read` method; the `ack` method acknowledges that the data has been read. The data on the port will persist until the `request` method is called again, at which point the port will present the next available data or become unready if there is no additional data.

The `toGet` function allows an input port to be treated as a `Get` interface. You can call `toGet()` to do both a `read` and an `ack` in a single line of code. In this instance the `toGet` function does not call the `request` method on the port; you must explicitly call the `request` method in your rule. Otherwise, the host will not send any data and the `get` method will never become ready.

```
instance ToGet#(SceMiMessageInPortIfc#(a),a);
```

The module `mkSceMiMessageInPort` instantiates a SCE-MI in-port macro which conforms to the SCE-MI standard.

`mkSceMiMessageInPort`	Instantiates an input port of the correct type.

	`module [SceMiModule] mkSceMiMessageInPort`
	`(SceMiMessageInPortIfc#(a) ifc)`
	`provisos(Bits#(a,_));`

A.4.2 SceMiMessageOutPort

The `mkSceMiMessageOutPort` module exposes the `SceMiMessageOutPortIfc#(msg_type)` interface with the following methods:

SceMiMessageOutPortIfc				
Method			Argument	
Name	Type	Description	Name	Description
`accepting_data`	`Bool`	Returns `True` when the output port is accepting data.		
`send`	`Action`	Sends data on an output port.	`msg`	data of type *msg_type*

```
interface SceMiMessageOutPortIfc#(type msg_type);
   (* always_ready *)
   method Bool accepting_data();
   method Action send(msg_type msg);
endinterface: SceMiMessageOutPortIfc
```

The `accepting_data` method on an output port will return `True` when it is allowed to send data out the port. The `send` method is used to transmit the data.

The `toPut` function can be applied to a `SceMiMessageOutPortIfc` interface to convert it to a `Put` interface:

```
instance ToPut#(SceMiMessageOutPortIfc#(a),a);
```

The module `mkSceMiMessageOutPort` instantiates a SCE-MI out-port macro which conforms to the SCE-MI standard.

`mkSceMiMessageOutPort`	Instantiates an output port of the correct type based on the `link_type` module parameter.

	`module [SceMiModule] mkSceMiMessageOutPort#`
	`(SceMiMessageOutPortIfc#(a) ifc)`
	`provisos(Bits#(a,_));`

A.5 Build modules

There are two types of modules provided by Bluespec to connect or *build* the components of a SCE-MI design: `buildDut` and `buildSceMi`. The `buildDut` modules connect the DUT with a controlled clock in the SceMiLayer. The `buildSceMi` modules connect the design into the emulation environment.

A.5.1 buildDut

In the SceMiLayer the `buildDut` and `buildDutWithReset` modules instantiate the DUT and connect it with a specific controlled clock. Each SceMiLayer must have a `buildDut` or `buildDutWithReset` statement. These modules are of the type `SceMiModule`.

`buildDut`	Instantiates the DUT with a clock port
	`module [SceMiModule] buildDut#(Module#(DutIfc) mkDut, Integer clockPort) (DutIfc);`

`buildDutWithReset`	Instantiates the DUT with a clock port and reset
	`module [SceMiModule] buildDutWithReset#( Module#(DutIfc) mkDut,Integer clockPort, Reset dutReset) (DutIfc);`

A.5.2 SceMiBuilder

Bluespec provides pre-written bridge modules for many link types. For most designs you will not need to write your own bridge module and will not need to write any `buildSceMi` statements.

The `buildSceMi` modules connect the hardware side of the SCE-MI design, composed of the DUT and SceMiLayer modules, into the specified emulation or simulation environment. The top-level module of each design must contain one and only one instance of the `buildSceMi` module. This transformation adds the infrastructure related to the specific link type or emulation platform.

The `buildSceMi` module takes the collection of SCE-Mi-related items accumulated by the `[SceMiModule]` module type, and implements the SCE-MI coordination logic and infrastructure required by the hardware link type and emulation environment. As a part of this process the modules are converted from the `[SceMiModule]` type to the `[Module]` type, allowing them to by synthesized.

The emulation environment will often require many link-dependent statements to be added to the top module (or bridge module), along with the instantiation of the `buildSceMi` module. In the simplest environment, TCP, the bridge module contains only the `buildSceMi` statement.

There are different implementations of `buildSceMi`, each taking the link type as an argument. The `SceMiBuilder` typeclass (Section E.1) defines the instances of the `buildSceMi` module.

Typeclass

```
typeclass SceMiBuilder#(type ifc_type, type arg_type, type extended_ifc_type);
   module [Module] buildSceMi#( SceMiModule#(ifc_type) mod
                              , arg_type arg )
                              (extended_ifc_type ifc);
```

Module The basic `buildSceMi` instance is provided here. There are other instances defined to support passing in external clocks and resets, as well as instances to support PCIE links (Section E.1).

`buildSceMi`	Takes a module of type `SceMiModule`, builds the SceMi infrastructure and returns the interface as a `Module`.
	`module [Module] buildSceMi#(SceMiModule#(i) mod, SceMiLinkType link_type) (i ifc);`

`buildSceMi` Module Arguments		
Name	Type	Description
`mod`	`SceMiModule#(`*i*`)`	A module of the type `SceMiModule`, with interface *i*, which is a parent of all other `SceMi` package instances in the design.
`link_type`	`SceMiLinktype`	Selects emulation host platform type

A.6 Clocking

There are two types of clocks in a SCE-MI system: the uncontrolled clock and one or more controlled clocks. The uncontrolled clock is a free-running clock used for interacting with SCE-MI ports and clock controllers. The DUT runs on a controlled clock and can be stopped by the user through any transactor during operations. In an emulation environment there is a third clock, the base FPGA clock or reference clock from the FPGA. The controlled clocks are aligned with the rising edges of the uncontrolled clock and the fastest controlled clock can be at the same frequency as the uncontrolled clock. There is a phase shift between the reference clock of the FPGA and the controlled and uncontrolled clocks of the SCE-MI system.

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, the controlled clock, may need to be stopped so that the testbench can catch up. The SCE-MI transactor stops the clock long enough for operations to occur, freezing time on the HW side when the SW side is called. When the controlled clock is stopped, the DUT is *frozen in controlled time*. Most SCE-MI applications will have at least one controlled clock and may have multiple controlled clocks.

A.6.1 SceMiClockConfiguration

Clock identities and waveform parameters are specified by populating a `SceMiClockConfiguration` structure. Each structure identifies a clock and describes its waveform frequency, duty cycle and phase shift as well as specifying the number of cycles to hold reset for the domain.

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

The meaning of the `SceMiClockConfiguration` fields is defined in the SCE-MI standard document. Briefly, each clock is assigned a unique identification number and its waveform is defined using a period ratio, a duty cycle and a phase shift. The `clockGroup` field is a non-standard extension described in section A.6.4.

The period ratio (`ratioNumerator`/`ratioDenominator`) is relative to the fastest controlled clock in the system, and cannot be less than 1. A ratio of N/M means that for every N cycles of the fastest controlled clock there will be M cycles of this clock.

The duty cycle is defined using a sum of the `dutyLo` and `dutyHi` parameters. The waveform will hold the clock low for $100 \times$ `dutyLo`/(`dutyLo` + `dutyHi`) percent of the time and high for $100 \times$ `dutyHi`/(`dutyLo` + `dutyHi`) percent of the time. One or the other of `dutyLo` or `dutyHi` can be 0, which indicates a "don't care" duty cycle in which the exact placement of the following clock edge within the cycle is not important. Specifically, `dutyHi` = 0 gives a waveform in which the exact placement of the falling edge is unspecified. Likewise, `dutyLo` = 0 gives a waveform in which the exact placement of the rising edge is unspecified.

The phase shift is determined by the ratio of the `phase` parameter to the sum of `dutyLo` and `dutyHi`. It is an error if `phase` is less than 0 or greater than or equal to `dutyLo` + `dutyHi`.

The `resetCycles` field determines the minimum number of cycles of the controlled clock during which its associated controlled reset will be asserted.

```
typedef struct { Integer clockNum;
                 Integer clockGroup;
                 Integer ratioNumerator;
                 Integer ratioDenominator;
                 Integer dutyHi;
                 Integer dutyLo;
                 Integer phase;
                 Integer resetCycles;
               } SceMiClockConfiguration deriving (Eq);
```

An instance of the `DefaultValue` class is defined for the `SceMiClockConfiguration` structure. The default value 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.

```
instance DefaultValue#(SceMiClockConfiguration);
   function SceMiClockConfiguration defaultValue();
      SceMiClockConfiguration conf;
```

```
      conf.clockNum         = 0;
      conf.clockGroup       = noClockGroup;
      conf.ratioNumerator   = 1;
      conf.ratioDenominator = 1;
      conf.dutyHi           = 0;
      conf.dutyLo           = 100;
      conf.phase            = 0;
      conf.resetCycles      = 8;
      return conf;
   endfunction
endinstance
```

The easiest way to populate a `SceMiClockConfiguration` structure is to define an instance of the structure equal to the `defaultValue` instance and change only the members necessary to create the desired clock parameters. Example:

```
SceMiClockConfiguration conf = defaultValue;
conf.clockNum = 1;
conf.dutyHi = 50;
conf.dutyLo = 50;
```

To use the default instance, the package `DefaultValue` must be imported.

The first clock, `clockNum = 0` must be defined and must be an undivided clock.

A.6.2 SceMiClockPort

A controlled clock and its associated controlled reset are bundled into a `SceMiClockPortIfc`. This is the BSV adaptation of the SceMiClockPort macro interface described in the SCE-MI 2.0 specification.

SceMiClockPortIfc		
Method		
Name	Type	Description
`cclock`	interface	Controlled Clock output.
`creset`	interface	Controlled Reset output.

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

The waveform of the controlled clock is determined by `SceMiClockConfiguration` parameter supplied to the `mkSceMiClockPort` module.

`mkSceMiClockPort`	Creates a clock port as defined by the configuration parameter.

	`module [SceMiModule] mkSceMiClockPort#(`
	`parameter SceMiClockConfiguration conf )`
	`(SceMiClockPortIfc ifc);`

A.6.3 SceMiClockControl

For each `mkSceMiClockPort` instance, zero or more `mkSceMiClockControl` instances are permitted. Each clock controller allows a transactor to exert edge-by-edge control of the advancement of the controlled clocks. The `mkSceMiClockControl` module provides a `SceMiClockControlIfc` interface which provides the uncontrolled clock and reset to the transactor and methods which notify when a controlled clock edge is imminent. This is the BSV adaptation of the SceMiClockControl macro interface described in the SCE-MI 2.0 specification.

SceMiClockControlIfc		
Method		
Name	Type	Description
`uclock`	interface	The uncontrolled Clock.
`ureset`	interface	The uncontrolled Reset.
`pre_posedge`	`Bool`	True if the controlled clock will rise on the next edge of the uncontrolled clock.
`pre_negedge`	`Bool`	True if the controlled clock will fall on the next edge of the uncontrolled clock.

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

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

The association of a clock controller and a clock port is achieved through the unique clock identification number, the `clockNum` parameter of the `mkSceMiClockControl` module. The clock is only enabled when the boolean arguments `allow_pos_edge` and `allow_neg_edge` are `True`. When a clock controller is defined for a clock, the clock must be explicitly enabled (through the boolean arguments) for the clock to run. If the `allow_pos_edge` and `allow_neg_edge` are `False`, clock will not run.

`mkSceMiClockControl`	Instantiates a clock controller module.

	`module [SceMiModule] mkSceMiClockControl#`
	`(parameter Integer clockNum,`
	`Bool allow_pos_edge,`
	`Bool allow_neg_edge)`
	`(SceMiClockControlIfc);`

A.6.4 Clock grouping extension

A 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. The clock group extension relaxes this requirement by allowing the user to partition the controlled clocks into groups such that stopping a controlled clock in one group affects

only the other clocks in the same group – controlled clocks in other clock groups may continue to run while all of the controlled clocks in the affected group are stalled.

The assignment of controlled clocks to clock groups is determined by the `clockGroup` field of the `SceMiClockConfiguration` structure passed to `mkSceMiClockPort`. All clocks defined with the same `clockGroup` value will run or stall together. Clocks defined with different `clockGroup` values will be able to stop with affecting one another.

The standard SCE-MI behavior is achieved by placing all controlled clocks in the same clock group. The `noClockGroup` constant can be used for this purpose, and is the default value for the `clockGroup` field.

The extension is not supported for the `SCEMI` link type.

A.6.5 Clock compiler messages

During compilation of SCE-MI designs, the Bluespec compiler (bsc) will emit messages describing all of the clocks found. Example:

```
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 99, column 13:
(SCE-MI) Begin clock generation
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 136, column 51:
(SCE-MI) Default clock group
(SCE-MI) Clock #1
  Numerator: 100
  Denominator: 100
  DutyHi: 0
  DutyLo: 100
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 0 controllers for clock #1
(SCE-MI) Clock #2
  Numerator: 100
  Denominator: 33
  DutyHi: 0
  DutyLo: 100
  Phase: 0
  Reset Cycles: 8
(SCE-MI) Found 0 controllers for clock #2
```

In this example, clock #1 is modeling a 100MHz clock and clock #2 is modeling a 33 MHz clock. These are model clock speeds, not physical clock speeds, as we will see below.

The next message from the compiler describes the physical clock generation attributes:

```
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 206, column 16:
(SCE-MI) Common clock time scale = 33
(SCE-MI)    Group    Clock   Period     Rise     Fall     Freq
(SCE-MI)  default        1       33        0       16   1.515 MHz
(SCE-MI)  default        2      100        0       50   0.500 MHz
(SCE-MI) Note: clock frequencies assume 100 MHz reference clock
Compilation message: "../BSVSource/SceMi/SceMiClocks.bsv", line 457, column 47:
(SCE-MI) Max reset count = 1600
```

The first part tells you what common clock time scale was determined for these clocks. The common clock time scale is a measure of reference clock cycles that go into producing a single uncontrolled clock (uclock) edge so that all of the controlled edges can align with rising uclock edges and still maintain the correct relative spacing. The lower the number, the better.

Next the compiler prints a table listing the period (in reference clock cycles) of each clock, along with the cycle number at which it will rise and fall. It provides an estimate of the physical clock frequency assuming a 100MHz reference clock frequency. In this case, because the user specified a 100:33 ratio, the common clock time scale is quite high and the physical frequencies are much lower than the reference clock frequency. If you are willing to be less exact with the requested clock ratio (say 1:3 instead of 100:33) the common clock time scale will be much lower and you will realize faster physical clock speeds.

The final piece of information is the maximum reset count. This is the number of reference clock cycles that the reset will be maintained to ensure that all clocks have been in reset for their requested number of reset cycles. Once this number of reference clock cycles has elapsed in reset, the resets will be deasserted at the next point that all clock periods align.

In contrast to the relatively poor clock specification illustrated above, you can get the common clock time scale down to 1, such that the fastest controlled clock and the uncontrolled clock operate at the same frequency. These special cases will be listed explicitly in the table instead of giving rise and fall counts. Example:

```
(SCE-MI) Common clock time scale = 1
(SCE-MI)    Group    Clock   Period     Rise     Fall     Freq
(SCE-MI)  default        1        1          uclock      50.00 MHz
(SCE-MI)  default        2        1   inverted uclock   50.00 MHz
(SCE-MI)  default        3        2        0        1   25.00 MHz
(SCE-MI) Note: clock frequencies assume 100 MHz reference clock
```

A.7 Connecting hardware and software components

In a SCE-MI-style emulation system, each message port from the hardware side is paired with a port proxy in the testbench on the software side. The following table displays the hardware message port modules along with their corresponding testbench port proxies. Each message port will have an instantiation of the message port module on the hardware side, and a message port proxy on the software side, as shown in Figure 22.

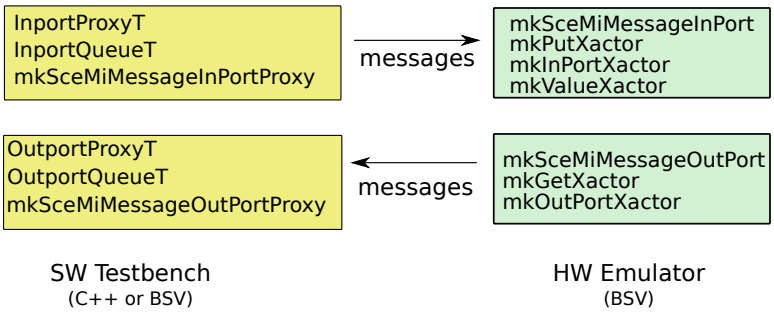

Figure 22: Corresponding Software Proxies and Basic Transactors

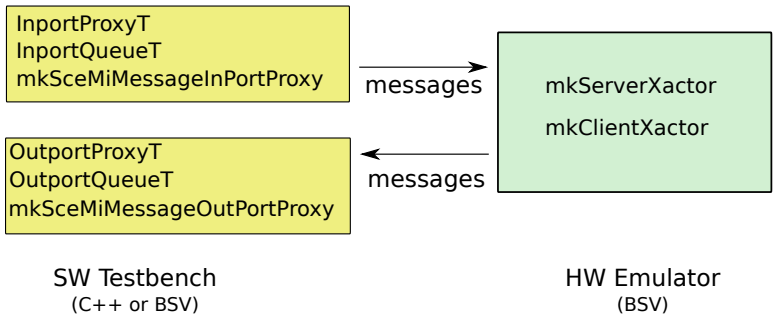

Figure 23: Corresponding Software Proxies and Transactors

Corresponding Software and Hardware Ports and Transactors			
Software Side		Hardware Side	
C++	BSV	BSV Message Ports	BSV Transactors
InportProxyT InportQueueT	mkSceMiMessageInportProxyT	mkSceMiMessageInPort	mkPutXactor mkInPortXactor mkValueXactor
OutportProxyT OutportQueueT	mkSceMiMessageOutportProxy	mkSceMiMessageOutPort	mkGetXactor mkOutPortXactor

The following packages can be used to control the simulation from the testbench. The software side control transactors communicate in both directions with the hardware side control transactors, as shown in Figure 24.

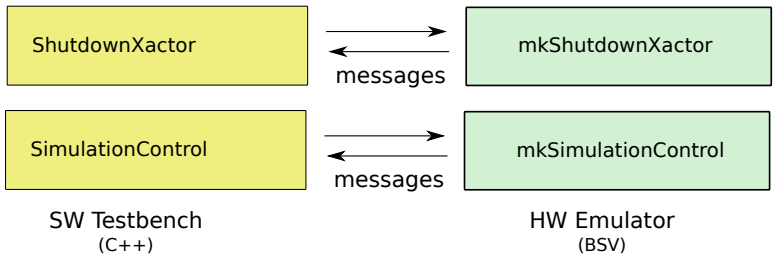

Figure 24: Corresponding Software and Hardware Control Transactors

Corresponding Software and Hardware Control Transactors		
Transactors		Description
C++ Testbench	BSV Hardware Side	
SimulationControl	mkSimulationControl	Provides simulation-like clock control.
ShutdownXactor	mkShutdownXactor	Stops the simulation.

A.8 Bluespec C++ testbench library

The software side of the message channel is defined as a port proxy. Software transactors use these port proxies to access messages being sent and received from the hardware side. As with the hardware side transactors, a single software transactor may contain one or more port proxies, as well as other testbench components.

Bluespec provides a large library of port-proxies, transactors, and utilities for creating a C++ SCE-MI testbench. The files are distributed in the Bluespec release in the `$BLUESPECDIR/SceMi` directory. Complete C++ specifications for these components are in Appendix F.

Bluespec's implementation of the SCE-MI standard is provided in the file `$BLUESPECDIR/SceMi/scemi.h`.

When using any of the Bluespec-provided library of C++ components, you must include the file `bsv_scemi.h` at the top of your C++ `.h` files.

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

A.8.1 Port proxies

The software-side port proxies correspond to the message ports on the hardware side. The inport and outport proxies are named based on the direction of the data from the perspective of the *hardware* side. Therefore, the inport proxies are for data going out of the testbench, but in to the DUT on the hardware side. The outport proxies are for data going in to the testbench, out from the DUT.

Bluespec provides the following C++ port proxies. See Appendix F.2 for complete C++ specifications.

- `InportProxyT`: C++ side of the `mkSceMiMessageInport` transactor, used to send messages to the hardware side.
- `InportQueueT`: Combines the `InportProxyT` with an output queue.
- `OutportProxyT`: C++ side of the `mkSceMiMessageOutport` transactor, used to receive messages from the hardware side.
- `OutportQueueT`: Combines the `OutportProxyT` with an input queue.

A.8.2 Data type utilities

Bluespec provides the following C++ data type classes to correspond to BSV data types and convert the BSV types to and from SceMiMessages. Refer to Appendix F.3 for the complete C++ specification.

- `BSVType`: Pure virtual base class for BSV types: `BSV_Primitive`, `BSV_Struct`, `BSV_TaggedUnion`, `BSV_Enum`, `BSV_Vector`
- `BitT`: Corresponds to Bluespec's `Bit#(n)` type.
- `BSVVectorT`: Corresponds to Bluespec's `Vector#(n,t)` type.
- `BSVVoid`: Corresponds to Bluespec's `void` data type.
- `StampedT`: Wrapper class for BSVType, adding the timestamp to the `SceMiMessageData`.

A.8.3 Simulation control classes

Bluespec provides the following classes to control simulation from the C++ testbench:

- `SimControlReq`: Populates `SceMiMessageData` with requests and format the `SimCommand` and `SimControlReq` values
- `SimControlResp`: Handles responses
- `SimulationControl`:Controls sending and receiving messages to start and stop the simulation and to handle messages about the status of the simulation.

Refer to Appendix F.4 for the complete specification of these classes.

A.8.4 ShutDownXactor

The C++ class `ShutdownXactor` communicates with the the BSV transactor `mkShutdownXactor` to stop the simulation. Refer to Appendix F.5 for the class specification.

A.8.5 Probes

Bluespec provides the `ProbesXactor`, which handles hardware probes defined through the Probe Insertion Tool.

Refer to Appendix F.6 for the specification of the `ProbesXactor` class.

A.8.6 Utilities

The following utility classes are provided for the C++ testbench:

- `WaitQueue`: Provides a queue for multi-threaded programs, where the process receiving from the queue can block until data is available.
- Thread Utilities
  - `Thread`: a pure virtual class that is the base class for the `SceMiServiceThread` class.
  - `SceMiServiceThread`: Specialization of the `Thread` class, used to start and stop the service thread without using SystemC
- Target Utilities
  - `Target`:Abstracts the target location so the same functions can be used to generate strings to a file or a buffer
  - `FileTarget`: Used to generate strings to a file
  - `BufferTarget`: Used to generate strings to a buffer

Refer to Appendix F.7 for the C++ specification of other utilities provided by Bluespec.

A.9 BSV testbench library

Bluespec provides a BSV library package that can be used to instantiate the proxies required to build a SceMi compatible testbench module. Since these proxies are written with Bluespec SystemVerilog (BSV), a testbench using this facility will be synthesizable. A synthesizable testbench can be put on the emulation platform, along with the rest of the system. See Appendix E.5 for the complete BSV specifications.

To access the BSV SceMi Testbench Library, import the SceMiProxies package:

```
import SceMiProxies :: * ;
```

The `SceMiProxies` package provides message input and output port proxies parameterized by the type of the message payload. Each port proxy definition includes an interface and a module providing the interface. Also provided are `ToGet` and `ToPut` instances to define interfaces as part of the `toGet` and `toPut` typeclasses. The SW port proxies correspond directly to the HW message ports. The BSV software port proxies are defined in Appendix E.5.

SceMiMessageInPortProxy The module `mkSceMiMessageInPortProxy` instantiates an in-port proxy macro which conforms to the SCE-MI standard. This is the SW side of the BSV module `mkSceMiMessageInPort` (Section A.4.1). The module exposes the `SceMiMessageInPortProxyIfc`.

SceMiMessageOutPortProxy The module `mkSceMiMessageOutPortProxy` instantiates an out-port proxy macro which conforms to the SCE-MI standard. This is the SW side of the BSV module `mkSceMiMessageOutPort` (Section A.4.2) and exposes the `SceMiMessageOutPortProxyIfc`.

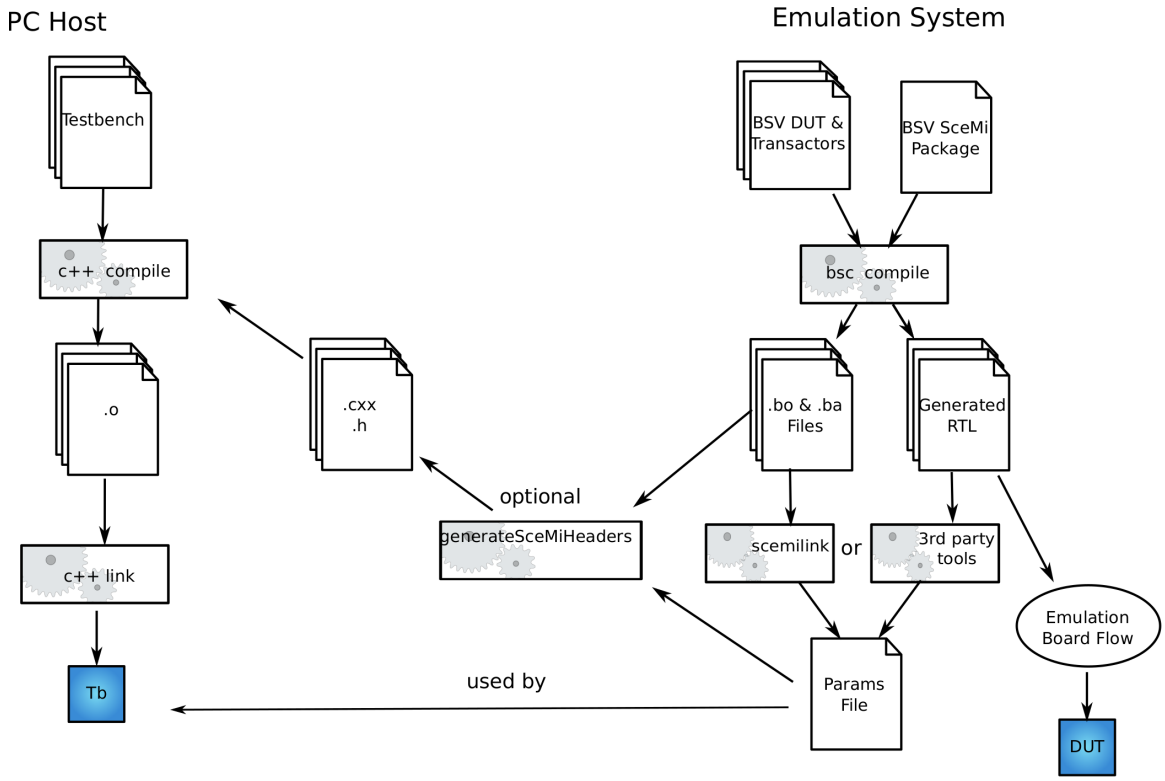

Figure 25: Common Build Flow

B Build Flows

B.1 Build flow overview

A generalized build flow for BSV systems including SCE-MI is shown in Figure 25. Each emulation platform will have its own specific build procedure, some adding steps, others skipping steps, but the general concepts shown here apply across all platforms. Because of the complexity involved with differences between emulation platforms, Bluespec provides the build utility (Section 2) to codify the build procedure, including setting options and determining the necessary steps for all platform variations.

The `build` utility implements all the detailed steps described in this section. When using the utility, you only need to describe your environment in a `project.bld` file then type:

```
build
```

The utility will implement all the required commands and steps, in order, to generate the final testbench and emulation files.

The hardware-side DUT and transactors are written in BSV, using the techniques described in Section A.2, and then compiled with the bsc compiler. When compiling for Verilog, as you will for most systems, you must specify the `-elab` flag to generate elaborated module files (`.ba`) files during compilation (these files are always generated for Bluesim). The bsc compiler will generate RTL to load on to the emulation board. Each board will have its own methods of loading and using the emulation board.

Here we show the software side as a C++ testbench, written using the techniques described in Section A.8, and compiled with a C++ compiler, generating a C++ output file. It could also be written in BSV, as described in Section A.9.

After compiling the hardware and software sides independently, you will have an RTL file and generated bsv (`.ba`) files on the hardware side, and C++ output files on the software side. What is missing is the connection between the two sides. This connection is handled by the infrastructure linkage tool, described in Section B.2. The infrastructure linkage tool generates the parameters file and any other files needed to connect the hardware and software components. For the `SCEMI` linkage type, the infrastructure linkage tool is supplied by the platform vendor. For other linkage types, Bluespec provides an infrastructure linkage tool called **scemilink**, described in Section B.2.

After linkage, the tcl script **generateSceMiHeaders.tcl** can be used to generate C++ classes that correspond to the BSV data types, including structs and enums. The tcl script reads the parameters file and the bsc elaborated module file (`.ba`) to generate the C++ files. While these datatypes can be written by the user directly, this optional step facilitates developing C++ software testbenches which are compatible with the hardware design in BSV.

Also run after linkage is the **HDL editor**, an optional utility used to modify the Verilog netlist, adding probes or performing other design edits. The `HDL editor` utility can be run from a GUI or as a batch process. It modifies the parameters file and writes out a new HDL (Verilog) file.

The C++ output files are linked, generating a testbench executable. During initialization, the software testbench reads the parameters file to bind the software side message port proxies to the hardware side message ports.

B.2 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 link data. The 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 will generate other files as required by the target environment.

For the `SCEMI` linkage type, use the infrastructure linkage tool supplied by the emulation platform vendor. These tools use the HDL generated by the hardware side. For other linkage types, Bluespec provides an infrastructure linkage tool called scemilink, which reads the `.ba` files generated by Bluespec.

The scemilink tool takes as input the `.ba` files produced from the initial bsc elaboration step (either `bsc -sim ...` or `bsc -verilog -elab ...`), analyzes the design, and produces a SCE-MI parameters file, shown in Figure 26, which can be loaded by the testbench. For the `TCP` linkage type, scemilink will also generate files needed to run the simulation, depending on whether the simulation target is Bluesim or a Verilog simulator. For Bluesim, scemilink will generate the files `scemilink.c` and `scemilink.h` to be integrated into the Bluesim executable. For Verilog, scemilink will generate a `scemilink.vlog_fragment` file to be integrated into the Verilog simulation model.

The `scemilink` command requires the name of the top module in the design (eg. `mkBridge`) as a command-line argument. Additional options include flags to control the parameter file name, search and file output paths. Use the `scemilink -h` command to display all available options.

The options required for scemilink and the entire build procedure depend on the design and emulation environment, and can be very complex depending on the target. The build utility, described in Section 2 simplifies this process and ensures that consistent and correct options are provided for scemilink and throughout the entire build process.

```
// Sce-Mi parameters file generated on: Fri Mar  5 10:36:24 EST 2010
// By: Bluespec scemilink utility, version 2010.02.beta4 (build 19729, 2010-02-26)

// ObjectKind Index AttributeName Value

Clock 0 ClockName          "bsci_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 1 TransactorName ""
MessageOutPort 1 PortName       "bsci_shutdown_ctrl_out"
MessageOutPort 1 PortWidth      1
MessageOutPort 1 ChannelId      3
MessageOutPort 1 Type           "Bool"

MessageOutPort 0 TransactorName ""
MessageOutPort 0 PortName       "bsci_dutConnection_outport"
MessageOutPort 0 PortWidth      64
MessageOutPort 0 ChannelId      2
MessageOutPort 0 Type           "UInt#(64)"

MessageInPort 0 TransactorName ""
MessageInPort 0 PortName       "bsci_shutdown_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    7381
```

Figure 26: The parameters file for a simple TCP example

B.3 Compiling the hardware side

When using the `SceMi` package, the linkage type determines which compilation steps are necessary as well as which emulation/simulation options are supported. This section describes specifics for compiling for different link types, as determined by the `SceMiLinkType` argument.

The table below shows which link types generate for emulation hardware (`HW`), whether the link type can generate for Verilog or Bluesim simulators, and the linkage tool used. The remainder of the section describes the specific build steps for each link type. Most of the details contained in this section can be handled for you automatically by the build utility, described in Section 2.

Hardware Environment by Link Type					
Link Type	HW	Simulator		Linkage Tool	Notes
		Verilog	Bluesim		
TCP		✓	✓	scemilink	Simulation only
SCEMI	✓	✓		vendor	Simulation requires 3rd party tool support
PCIE_VIRTEX5	✓			scemilink	
PCIE_VIRTEX6	✓			scemilink	
PCIE_DINI	✓			scemilink	

B.3.1 TCP linkage type and compiling for simulation

The TCP linkage type does not involve emulation hardware. When using the TCP linkage type, bsc can be used either to compile for simulation with Bluesim or a 3rd-party Verilog simulator. When using the -verilog option you must use the -elab flag to generate the .ba files during compilation.

In between the first and second bsc compilation stages, the scemilink infrastructure linkage tool must be run, as described in Section B.2. The second bsc compilation stage is then run as normal, with the addition of the -scemi command-line argument. This instructs the bsc link stage to integrate the output of scemilink into the generated simulation model.

During simulation, the Bluesim or Verilog simulation executable will listen on a TCP port. The port selection can be changed from the default using the --port option to scemilink. The simulator must be started before the testbench, so that the simulator is ready to accept the connection when the testbench starts.

B.3.2 SCEMI linkage type and 3rd-party platforms

When using the SCEMI linkage type, the bsc compiler is used only to generate Verilog output containing the SCE-MI instantiations. The initial compilation step is no different than for any other design. Once the Verilog is generated from bsc, the rest of the compilation can use any 3rd-party infrastructure linkage and integration tools which support the SCE-MI standards. Documentation for those tools is provided by the respective tool vendor.

B.3.3 PCIE_VIRTEX5, PCIE_VIRTEX6, and PCIE_DINI linkage types

The linkage types PCIE_VIRTEX5, PCIE_VIRTEX6, and PCIE_DINI use Bluespec's scemilink as the infrastructure linkage tool to generate SCE-MI specifications to be run on emulation boards using PCIE. The -elab flag must be specified during the bsc compilation step to generate the elaboration files required by the scemilink and generateSceMiHeaders programs.

B.4 Compiling the software testbench

B.4.1 Compiling a C++/SystemC testbench

In addition to the BSV SCE-MI library, the Bluespec software release includes a C++ library implementation of the SCE-MI API, as documented in the SCE-MI 2.0 Reference Manual.

The scemi.h header file is located in the ${BLUESPECDIR}/SceMi/ directory. The API library itself (libscemi.a) is distributed in different versions for different C++ compiler families. For example, if the ${BLUESPECDIR}/bin/bsenv c++_family script returns the value g++4_64, the correct library can be found at ${BLUESPECDIR}/SceMi/g++4_64/libscemi.a.

When compiling the testbench code, these options should be used to incorporate the SCE-MI API library:

Options Required to Compile C++ Testbench	
C++ Option	Description
`-I${BLUESPECDIR}/SceMi`	Path to include scemi.h file
`-L${BLUESPECDIR}/SceMi/<C++_family>`	Path to find libscemi.a for <C++_family>
`-lscemi`	Link libscemi into executable
`-lpthread`	standard thread library required for linking
`-ldl`	standard dynamic linking library

B.4.2 Compiling a BSV testbench

The common build flow in Figure 25 showed a C++ testbench. A SCE-MI software testbench can also be written using the BSV testbench library as described in Section A.9.

Compiling a testbench module written with the BSV testbench library is similar to compiling a regular BSV module. During the link stage of the BSV testbench, the `-scemiTB` flag must be added to the bsc command to link in the extra routines to communicate with the SceMi infrastructure.

Figure 27 illustrates the testbench compilation flows for both bluesim and verilog methodology.

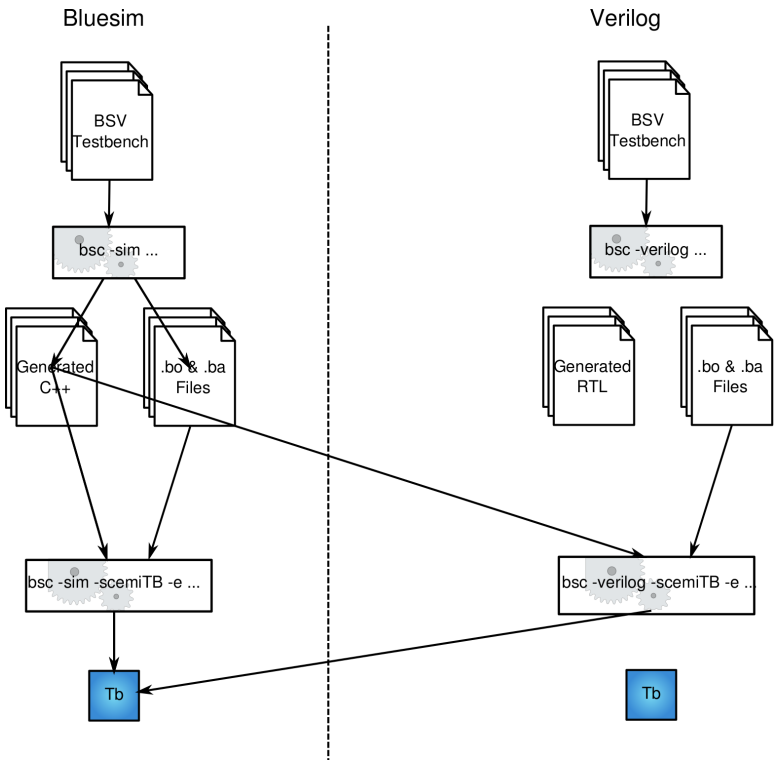

Figure 27: BSV Testbench Compilation flows.

C Build Utility

The build utility requires `python 2.4` or greater.

The build utility facilitates building a Bluespec project and is especially useful in complex environments such as those utilizing FPGAs. The utility takes a project configuration file and determines the stages necessary to build the entire project, or a portion (target) of the project. The utility then determines and executes the commands necessary to complete each stage.

The build utility codifies the rules for building a Bluespec project in a single place, eliminating most of the boilerplate involved in setting up a project. To use the build utility, create a project configuration file (`.bld`), specifying the details of your design, including directories, tools, module and file names, and a set of build targets. Examples of build targets include building the testbench for bluesim or building the DUT. Each target contains a set of build directives, or instructions. When executed, the build utility determines and executes the stages and commands necessary to build the selected target.

Section C.4 contains a simple build file (`.bld`), along with the list output generated from the file.

C.1 Usage

To build a project:

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

By default, the build utility uses the file `project.bld`. Use the `-p` option to specify a different project file name.

Option	Alternate Syntax	Description
`--version`		show program's version number and exit
`--help`	`-h`	show help message and exit
`--doc`	`-d`	print full documentation, then exit
`--init`		generate an initial project.bld file from a project.init file
`--initforce`		generate an initial project.bld file from a project.init file, the file will be written over even if it already exists
`--from=STAGE`	`-f STAGE`	begin building from STAGE
`--to=STAGE`	`-t STAGE`	finish building after STAGE
`--project=FILE`	`-p FILE`	read project description from FILE
`--list`	`-l`	list targets and stages, then exit
`--verbose`	`-v`	print verbose messages to stdout
`--dry-run`		display the commands executed in each stage without executing

The build utility reads a project description from a project configuration file, specified with the `--project` option. If none is given, it reads from the default file, `project.bld`.

The project description defines a number of build targets and directives that describe what the target is and how it should be built. All arguments to the build utility are interpreted as targets which should be built. If no target is supplied, the build utility will try to determine a sensible default action based on the project description file.

Each build target comprises a number of stages that are executed in sequence to build that target.

To see which targets are available in a project, and which stages are involved in building each target, use the `--list` option.

By default, all of the stages in a selected target are executed, so that the target is fully rebuilt from scratch. The user can select a subset of stages to execute using the `--to` and `--from` options.

The `--dry-run` option allows the commands executed in each stage to be displayed without actually executing any of them.

As the build utility executes, it logs the output of commands into log files using the naming scheme `<target>_<stage>.log`. If the `--verbose` option is given, the command output is also displayed on the screen.

C.2 Project Build File

The project build file (`.bld`) uses a simple configuration file syntax similar to a Windows .ini file syntax. The file is divided into sections, each section having a section header describing the target, with the target name enclosed in square brackets. Except for the reserved name `DEFAULT`, you can use any name, though descriptive names are recommended. Example:

```
[DUT]
...

[bsim.tb]
...
```

Syntax	Description
`[target]`	names of targets are enclosed in square brackets
`var: VALUE` or `var=VALUE`	Values are strings
`#`	indicates a comment until the end of the line
`[DEFAULT]`	special section whose directives apply to all targets

The section named `[DEFAULT])` is a special section, whose directives apply to all targets. The `DEFAULT` section is treated as a hidden section which is extended by all other sections.

Each section contains directives describing the target and its build procedure. The build utility constructs the correct command sequence based on the directives supplied for each target. A directive includes a variable and the value of the variable; a variable may have different values for different targets. Section C.3 lists the directives recognized by the build utility.

The directives are of the form:

```
var: VALUE
```

or:

```
var=VALUE
```

Values are typically strings, strings separated by spaces, or boolean values encoded as 'y'/'n', 'yes'/'no', '1'/'0', 'true'/'false' or 'on'/'off'. Boolean values can also use the alternate single-token syntax 'var' instead of 'var: true' and '!var' instead of 'var: false'.

Values can contain environment variable references of the form `${VAR}`. The content of the named environment variable is substituted into the value string before it is processed. It is an error if the named variable is not defined in the environment. The following special environment variables may be used by the build file:

Special Environment Variables	
Variable	Description
`${PROJECT_ROOT}`	holds the name of the directory containing the project file.
`${BUILD_TARGET}`	holds the name of the top-level target used to run the given directive.
`${BUILD_SECTION}`	holds the name of the section containing the current directive.

Comments begin with `#` and continue until the end of the line.

The backslash character is used in four escape sequences.

Backslash Escape Sequences	
`\#`	is interpreted as a single `#` character that does not start a comment.
`\$`	is interpreted as a single `$` character that does not introduce an environment variable name.
`\\`	is interpreted as a single `\\` character that does not escape the following character.
`\`	at the end of a line is a continuation marker for values that span multiple lines.

C.3 Directives

C.3.1 Targets

This section describes the directives that allow you to reuse targets within other targets. The `extends-target` directive takes the name of another section and includes all the directives from the other section. For example, you can define a target named `[dut]` which specifies details related to the design under test. You may also have two separate simulation targets: one which simulates the DUT in Bluesim (`[bsim_dut]`) and another which simulates the DUT with a Verilog simulator (`[vlog_dut]`). Both targets would include the `dut` target through the `extends-target` directory.

```
[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
```

Target Directives	
Directive	Description
`include-files: <files>`	permits the inclusion of other project description files
`hide-target`	indicates the target is used by the extends-target directive, and cannot be used as a target directly
`extends-target: SECTION`	incorporates the directives in SECTION in the current section
`default-targets: <targets>`	list in the `DEFAULT` section defining which targets are rebuilt by default
`sub-targets`	incorporates the sequence of stages for the listed targets, in order
`pre-targets`	Define targets which will run before the locally defined stages
`post-targets`	Define targets which will run after the locally defined stages
`skip-target-when:<command>`	if the exit status of the command is 0, skip the specified target stages
`skip-target-unless:<command>`	if the exit status of the command is not 0, skip the specified target stages

A project file can include other project files by using the `include-files` directive.

A hidden section cannot be used as a target directly, but can be used in the definition of other sections. The `hide-target` directive indicates the target is a hidden section.

The `extends-target` directive incorporates the directives of the named target (which may be hidden or not) into the current section. When a directive in the current section also appears in the extended section, the values are reconciled in a directive-dependent manner.

The `default-targets:` directive takes a list of target names. These are the targets which are rebuilt if no target is specified on the command line. This directive can only be used in the `[DEFAULT]` section.

The `sub-targets: <list of targets>` directive lists targets which should be run in sequence as the action of the current section. A section which uses the `sub-targets` directive will not be considered to build any direct products of its own. This directive can be used with an otherwise empty section to create a meta-target which names a sequence of build targets.

The `pre-targets` and `post-targets` directives define targets which will run before and after the stages defined locally in a section.

Targets can be made conditional using the `skip-target-when` directive. This directive provides a command to be executed in a shell prior to building the target. If the command returns an exit status of 0, the stages for the target will be skipped. There is also a `skip-target-unless` directive which will skip the target stages when the exit status is not 0.

Some targets attempt to clean directories before they run. If multiple targets are run and a later target wishes to clean a directory or file that an earlier target has already cleaned, the second cleaning will be suppressed.

C.3.2 Controlling bsc

This section describes the directives related to bsc commands, including the build target, compile options, and link options.

The build utility determines the steps and commands required to build for the platform as specified by the `build-for:<platform>` directive.

The default value for the `build-for` directive is `bluesim`. Valid values for the `platform` are:

```
bluesim - build and link for Bluesim
verilog - build and link for Verilog simulation
rtl     - build Verilog RTL but do not link for simulation
dn7002  - build Verilog RTL and synthesize for a Dini 7002 board
dn7006  - build Verilog RTL and synthesize for a Dini 7006 board
dn7406  - build Verilog RTL and synthesize for a Dini 7406 board
ml507   - build Verilog RTL and synthesize for a Xilinx ML507 board
ml605   - build Verilog RTL and synthesize for a Xilinx ML605 board
xupv5   - build Verilog RTL and synthesize for a Xilinx XUPV5 board
c++     - build software from C++ source code
systemc - build software from SystemC source code
```

bsc Directives	
Directive	Description
`build-for: <platform>`	target build platform, default value is `bluesim`
`bsc: <executable name>`	Command used to invoke the Bluespec compiler. Default value is `bsc` or uses $BSC if defined.
`bsc-compile-options: <options>`	bsc compilation stage options
`bsc-link-options: <options>`	bsc link stage options
`bsc-rts-options: <options>`	RTS options for the bsc compilation stage
`bsv-define: <definitions>`	macro definitions (of the form `<VAR>` or`<VAR>=<VAL>`) passed to the bsv compile stage with the `-D` option
`imported-c-files: <files>`	C source or object files to be included in linking the Verilog or Bluesim simulation executable
`imported-verilog-files: <files>`	verilog files to be included in linking the Verilog simulation executable or synthesis
`verilog-define: <definitions>`	definitions (of the form `<VAR>` or `<VAR>=<VAL>`) passed to the Verilog simulator with the `-D` option to the bsc link stage and to the synthesis tools
`verilog-simulator: <simulator>`	simulator for Verilog simulation, default value is `modelsim` or uses $VERILOG if defined.

C.3.3 Describing the project layout

The directives in this section specify the project files and directories. The first table describes directives for input directories and files, the second output directories and files.

Input Files and Directories			
Directive	Description	Becomes/ Used by	Default
top-file:<file name>	Top-level BSV file		Top.bsv
top-module:<module name>	Name of the top-level BSV module, default is derived from the top-level BSV file name top-file		mk<topfile>
board-top-module:<module name>	Alternate name of the top-level Verilog which will override the top-module directive when synthesizing or linking a targeted FPGA Verilog that may contain the design.		
bsv-source-directories: <directories>	Allows multiple source directories to be added to the -p path during bsc compilations.		
bsv-source-directory: <directory>	Directory containing the BSV source code	-p	.
verilog-lib-directories: <list of directories>	A list of paths to search through for verilog files during bsc compilation, verilog simulation, and FPGA synthesis.		

Output Files and Directories			
Directive	Description	Becomes/ Used by	Default
binary-directory:<directory>	Output directory for Bluespec binary data files	-bdir	.
exe-file:<executable name>	Executable file to generate for Bluesim, Verilog simulation or from a C++ compilation	-o	a.out
info-directory:<directory>	Output directory for Bluespec compiler information files	-info-dir	.
log-directory:<directory>	Output directory for build logs		.
simulation-directory: <directory>	Output directory for Bluesim code files	-simdir	.
verilog-directory:<directory>	Name of output Verilog directory	-vdir	.

C.3.4 Explicit command targets

run-shell-<STAGE>-<N>: <command>

Each run-shell-<STAGE>-<N> directive specifies a command to be executed using the shell. Commands are grouped by the <STAGE> part of the name, and within each stage commands are executed in ascending order of their numeric -<N> suffix.

stage-order: <list of stages>

When run-shell-<STAGE>-<N> directives are given for multiple stages, this directive specifies the order in which stages are to be executed. If not given, the order is undefined.

The directives in this section allow you to specify commands to be run at various points in the build. The first table lists commands to be executed after the specified stage, the second table lists commands to be run before the specified stage.

Explicit Command Targets <command> executed *after* stage	
Directive	Stage
post-stage-build_c++_tb: <command>	build C++ testbench
post-stage-build_systemc_tb: <command>	build systemC testbench
post-stage-compile_for_verilog: <command>	compile for Verilog
post-stage-edithdl_modify_verilog: <command>	edit HDL
post-stage-generate_scemi_parameters: <command>	generate SCE-MI parameters
post-stage-link_for_bluesim: <command>	link for Bluesim
post-stage-link_for_cosim: <command>	link for cosim
post-stage-link_for_verilog: <command>	link for Verilog

Explicit Command Targets <command> executed *before* stage	
Directive	Stage
pre-stage-build_c++_tb: <command>	build C++ testbench
pre-stage-build_systemc_tb: <command>	build systemC testbench
pre-stage-compile_for_bluesim: <command>	compile for Bluesim
pre-stage-compile_for_verilog: <command>	compile for Verilog
pre-stage-edithdl_modify_verilog: <command>	edit HDL
post-stage-generate_scemi_parameters: <command>	generate SCE-MI parameters
post-stage-link_for_bluesim: <command>	link for Bluesim
post-stage-link_for_cosim: <command>	link for cosim
post-stage-link_for_verilog: <command>	link for Verilog

C.3.5 FPGA Synthesis

The directives in this section are used with FPGAs, including specification of options to be passed to different tools.

FPGA Directory Directives		
Directive	Description	Default
board-support-directory: <directory>	The directory from which to retrieve board support files.	$BLUESPECDIR/board_support
altera-directory: <directory>	Directory in which to perform Quartus synthesis for Altera designs.	.
xilinx-directory: <directory>	The directory in which to perform XST synthesis for Xilinx designs.	.
board-top-fpga-file: <file name>	The top level verilog file that will be used for Quartus FPGA synthesis.	fpga_a.v.
board-top-fpga-module: <module name>	The top level verilog module that will be used for Quartus FPGA synthesis.	fpga_a.
board-top-module: <module name>	The top module that models the fpga and contains the FPGA fabric, components, and the top level design module to simulate in the FPGA.	fpga_a.

These directives can be used to specify the top level verilog file that will be used for Quartus FPGA synthesis for each specific FPGA on a board with multiple chips. The -x can have different values that match the designations of the chip to be synthesized (ie. 'a', 'b', etc...). For example, board-top-fpga-file-a directive specifies the top level verilog file for running Quartus synthesis on the fpga_a. When this directive is set, it overrides the board-top-fpga-file directive for specifying the top verilog file.

FPGA top Verilog file directives for synthesis	
Directive	Chip
`board-top-fpga-file-a: <file name>`	fpga_a
`board-top-fpga-file-b: <file name>`	fpga_b
`board-top-fpga-file-c: <file name>`	fpga_c
`board-top-fpga-file-d: <file name>`	fpga_d
`board-top-fpga-file-e: <file name>`	fpga_e

These directives can be used to specify the top level module that will be used for Quartus FPGA synthesis for each specific FPGA on a board with multiple chips. The `-x` can have different values that match the designations of the chip to be synthesized (ie. 'a', 'b', etc...). For example, `board-top-fpga-module-a` directive specifies the top level Verilog module for running Quartus synthesis on the `fpga_a`. When this directive is set, it overrides the board-top-fpga-module directive for specifying the top Verilog module.

FPGA top module directives for synthesis	
Directive	Chip
`board-top-fpga-module-a: <module name>`	fpga_a
`board-top-fpga-module-b: <module name>`	fpga_b
`board-top-fpga-module-c: <module name>`	fpga_c
`board-top-fpga-module-d: <module name>`	fpga_d
`board-top-fpga-module-e: <module name>`	fpga_e

These directives specify the top level Verilog module that will be used for Quartus FPGA simulation for a board with multiple chips. It specifies the name of the top module that modules the fabric, components, and the top level design module that the user wants to simulate in the fpga. The `-x` can have different values that match the designations of the chip to be simulated (ie. 'a', 'b', etc...). For example, `board-top-module-a` directive specifies the top level Verilog module for running simulation for the `fpga_a`. When this directive is set, it overrides the board-top-module directive for specifying the top Verilog module.

FPGA top module directives for simulation	
Directive	Chip
`board-top-module-a: <module name>`	fpga_a
`board-top-module-b: <module name>`	fpga_b
`board-top-module-c: <module name>`	fpga_c
`board-top-module-d: <module name>`	fpga_d
`board-top-module-e: <module name>`	fpga_e

General FPGA Directives		
Directive	Description	Default
`memory-clock-period: <float>`	If set, sets the reference clock period that is used to drive the memory interface, if one exists. This value propagates into the synthesis constraints.	8.0
`on-fpga: <list of FPGAs>`	Designator of the FPGA on which the design will be loaded.	`A`
`post-program-command: <command>`	Command to be run after programming the FPGA. This is useful, for example, to initialize or re-enable a PCIE device.	
`pre-program-command: <command>`	Command to be run before programming the FPGA. This is useful, for example, to shutdown a PCIE slot safely.	
`program-fpga: <bool>`	If set, after FPGA synthesis an attempt will be made to program the board (which requires sudo privileges). If this directive is not used, then a file will be generated which the user must manually load onto the FPGA.	`False`
`sodimm-style: <SODIMM style>`	Controls the synthesis directives associated with a SODIMM connector on this FPGA. Valid values are `NONE`, `DDR2`, `DDR3`, and `SRAM`	`NONE`
`scemi-clock-period: <float>`	If set, sets the reference clock period that SCEMI uses to create the uncontrolled and controlled clock domains. This value propagates into the synthesis constraints.	10.0

Options to be passed to Quartus Tools	
Directive	Quartus Tool
`quartus-asm-options: <list of options>`	Quartus asm tool
`quartus-drc-options: <list of options>`	Quartus drc tool
`quartus-fit-options: <list of options>`	Quartus fit tool
`quartus-map-options: <list of options>`	Quartus map tool
`quartus-sta-options: <list of options>`	Quartus sta tool
`quartus-tan-options: <list of options>`	Quartus tan tool
`use-quartus-sta: <bool>`	Use the new Quartus timing analyzer (quartus_sta) instead of the classic timing analyzer (quartus_tan). Default value is false.

The following directives specify the path to an alternate or supplemental file used by Quartus. The special identifiers `__BLUESPECDIR__`, `__BOARD__`, `__VDIR__`, `__TOP_MODULE__` and `__SODIMM_STYLE__` will be replaced everywhere they occur in the given file with appropriate substitutions.

Path to alternate file for Quartus Tools	
Directive	File
`quartus-qsf-file: <path>`	alternate QSF file
`quartus-qsf-supplement-file: <path>`	supplemental QSF file
`quartus-sdc-file: <path>`	alternate SDC template file
`quartus-sdc-supplement-file: <path>`	supplemental SDC file

Options to be passed to Xilinx Tools		
Directive	Xilinx Tool	Default
`xilinx-bitgen-options: <options>`	bitgen	`-w`
`xilinx-impact-options: <options>`	impact	
`xilinx-map-options: <options>`	map	see below
`xilinx-ngdbuild-options: <options>`	ngdbuild	`-intstyle ise -dd ./_ngo -nt timestamp`
`xilinx-par-options: <options>`	par	`-w -ol high`
`xilinx-trce-options: <options>`	trce	`-e 3`
`xilinx-xst-options: <options>`	xst	`-intstyle ise`
`xilinx-use-planahead: <bool>`	PlanAhead	`false`.

Default options to be passed to the Xilinx map tool (`xilinx-map-options`):

```
-w -pr b -ol high -timing -global_opt speed -logic_opt on -register_duplication
-retiming on -equivalent_register_removal on -xe n -lc auto
```

The following directives specify the path to an alternate or supplemental file used by the Xilinx map tool. The special identifiers `__BLUESPECDIR__`, `__BOARD__`, `__VDIRs__`, `__TOP_MODULE__` and `__SODIMM_STYLE__` will be replaced everywhere they occur in the given file with appropriate substitutions for the SCR, UCF, and XCF template files.

Path to alternate file for Xilinx Tools	
Directive	File
`xilinx-scr-file: <path>`	alternate SCR file
`xilinx-ucf-file: <path>`	alternate UCF template file
`xilinx-ucf-supplement-file: <path>`	supplemental UCF template file
`xilinx-xcf-file: <path>`	alternate XCF template file
`xilinx-xcf-supplement-file: <path>`	supplemental XCF template file

C.3.6 RTL editing

The `edithdl` utility, described in Section 7, allows the designer to add probes or other design edits to the RTL file generated during the build. This section describes the directives related to the `edithdl` utility.

By default, the utility will be run from a graphical interface. Set the `design-editor-options` to `-batch` to run the tool in batch mode. When running in batch mode you must specify a script to run from.

The utility modifies both the Verilog (RTL) file and the `.params` file.

RTL Editing Directives		
Directive	Description	Default
`run-design-editor:<bool>`	Enables using `edithdl` to modify the Verilog netlist	`False`
`design-editor-edit-params: <bool>`	Enables editing of params file when using `edithdl`	`False`
`design-editor-options: <options>`	Run in `-batch` or `-gui` mode	`-gui`
`design-editor-output-directory: <directory>`	Output directory for modified verilog files	
`design-editor-output-params: <file name>`	Name of the modified SCE-MI parameters file	`<top-module>._EDITED.params`
`design-editor-script: <file name>`	Name of script to use for batch mode	`replay_edits.script`
`design-editor-partition-for: <platform>`	Specifies the FPGA system to build for	`none`

The `design-editor-partition-for` directive specifies the FPGA system to build partitions for. The only current valid values are: `none` and `dn7002`

C.3.7 SCE-MI C++ Testbench

This section describes the directives related to the c++ testbench for a SCE-MI style co-emulation system. The source files for the testbench are specified by the `c++-files` directive, if no directive is provided, all files in the c++-source-directory with the extensions: `.c, .cc, .cpp, or .cxx` are used.

c++ Compiler Options		
Directive	Description	Default
`c++-compiler: <executable>`	c++ compiler	`g++` or `$CXX`
`c++-define: <macros>`	Macros (`<VAR>` or `<VAR>=<VAL>`) to pass to the c++ compiler with the `-D` option	
`c++-files: <source files>`	c++ source files for the testbench	
`c++-options: <options>`	Options passed to the c++ compiler	
`c++-source-directory: <directory>`	Directory containing c++ source files	.
`c++-header-aliases: <bool>`	Generate header files for relevant aliases for types being generated	`False`
`c++-header-directory: <directory>`	Directory in which to write generated SceMi Header `.h` files.	c++-source-directory
`c++-header-enum-prefix: <prefix string>`	Prefix for enums in SCE-MI Message Header files.	`e_`
`c++-header-member-prefix: <prefix string>`	Prefix for class members in SCE-MI Message Header files.	`m_`
`c++-header-probe-code: <bool>`	Used to generate an include file for all probes.	`False`
`c++-header-targets: <targets>`	Defines the groups of types of header files that will be generated. Valid values are: `inputs, outputs, probes, all, none`	`all`
`c++-header-types-package: <package name>`	Top level package defining the types needed for probes. The BSV source used for the probe types is found by consulting the bsv-source-directory and bsv-source-directories directives.	package of top-file

C.3.8 SCE-MI

Valid values for `scemi-type` are:

```
TCP
PCIE_DINI
PCIE_VIRTEX5
PCIE_VIRTEX6
SCEMI
EVE
```

Bsc compilation will be passed a `-D SCEMI_<scemi-type>` definition based on the `scemi-type` directive. If not used, a non-SCE-MI DUT build target will be assumed, or an appropriate SCE-MI target will be selected based on the build-for directive.

SCE-MI		
Directive	Description	Default
`probe-vcd-file: <file name>`	Specifies the VCD file containing probe waveforms.	`scemi_test.vcd`
`scemi-parameters-file: <file name>`	Name for SCE-MI parameters file, used as the `--params` option to scemilink.	value of top-module with a .params suffix
`scemi-tb: <bool>`	Used to designate a target as being a SCE-MI testbench.	`False`
`scemi-tcp-port: <number>`	TCP port number for the TCP link type, becomes the `--port` option to scemilink. Only used when `scemi-type=TCP`	7381
`scemi-type: <link type>`	Defines the type of SCE-MI link to use.	
`scemilink-options: <options>`	Supplies options to the scemilink utility.	
`shared-lib: <shared library name>`	Produce a shared library instead of an executable file.	
`systemc-home: <directory>`	The root of the SystemC installation, used for `-I` and `-L` options when a target built for SystemC uses the `scemi-tb` directive.	`$SYSTEMC`
`tcl-home: <directory>`	Root of the TCL area, used when the `uses-tcl` directive is supplied.	`$BLUESPECDIR/tcllib` or `$TCL_HOME`
`uses-tcl: <bool>`	Indicates that a c++ testbench target uses the TCL library.	`False`

C.3.9 Workstation

The build utility can be used with the Bluespec Development Workstation. The directives in this section indicate whether to create a workstation project file (`.bspec`), and if so, what the file name should be. If no file name is provided, the file will be named `project.bspec`).

Workstation Directives		
Directive	Description	Default
`create-workstation-project: <bool>`	Indicate that the Bluespec Workstation will be used to assist in debug. A default project file will be generated for use in the Workstation.	`False`
`workstation-project-file: <file name>`	Specifies Bluespec Workstation project file	`project.bspec`

C.4 Build File Example

C.4.1 Usage

To completely rebuild the default targets using the `project.bld` file:

```
build
```

To completely rebuild the `bsim_dut` target using the `demo.bld` project file:

```
build -p demo.bld bsim_dut
```

To see what targets and stages are supported by the `demo.bld` project:

```
build -p demo.bld --list
```

To stop after generating RTL for the `vlog_dut` target of the `demo.bld` project:

```
build -p demo.bld vlog_dut --to compile_for_verilog
```

To continue the Dini 7002 build process starting with the synthesis step in the 7002 target of the `demo.bld` project:

```
build -p demo.bld 7002 --from quartus_map
```

To see what commands would be executed to rebuild the `sw_tb` target of the `demo.bld` project:

```
build --dry-run -p demo.bld sw_tb
```

The simplest possible project file contains just a single target name:

```
[dut]
```

It defines a target named `dut` that builds a Bluesim executable for the `mkTop` module from a file called `Top.bsv`.

C.4.2 Typical project build file

A typical project file will define additional targets and provide more directives to specialize the build process. For example, this project file defines a SCE-MI dut and two versions of a testbench for it, one in BSV and one in SystemC.

```
[DEFAULT]
default-targets:      bsim_dut bsim_tb
bsc-compile-options:  -aggressive-conditions -keep-fires
bsc-link-options:     -keep-fires

# description of DUT
[dut]
hide-target
top-file:             gcd/Bridge.bsv
```

```
bsv-source-directory: gcd
scemi-type:           TCP
scemi-tcp-port:       3375
verilog-directory:    vlog_dut
binary-directory:     bdir_dut
simulation-directory: simdir_dut
info-directory:       info_dut
exe-file:             gcd_dut

[bsim_dut]
extends-target: dut
build-for:      bluesim

[vlog_dut]
extends-target:    dut
build-for:         verilog
verilog-simulator: cvc
run-design-editor: False

[edit_hdl]
design-editor-partition-for:    dn7002
design-editor-options:          --batch
design-editor-output-directory: vlog_edited
design-editor-script:           probe1.script
design-editor-edit-params:      True

# description of TB in BSV
[bsv_tb]
hide-target
scemi-tb
top-file:             gcd/Tb.bsv
top-module:           mkTestBench
verilog-directory:    vlog_tb
binary-directory:     bdir_tb
simulation-directory: simdir_tb
info-directory:       info_tb
exe-file:             gcd_tb

[bsim_tb]
extends-target: bsv_tb
build-for:      bluesim

[vlog_tb]
extends-target:    bsv_tb
build-for:         verilog
verilog-simulator: cvc

# description of TB in SystemC
[sysc_tb]
scemi-tb
build-for: systemc
c++-files: gcd/Tb.cpp
exe-file:  gcd_tb
```

D Tcl Scripts

The `build` utility codifies all the steps needed to completely build an application. When run, the utility may call other utilities, including tcl scripts, in addition to executing compiler commands. This section details Bluetcl utilities that are called as required by the `build` utility but can also be called directly from the command line. The usage examples include the `bluetcl -exec` command required when running the scripts from the command line. Refer to the **Bluetcl Reference** appendix in the *Bluespec SystemVerilog User Guide* for more information on using Bluetcl including a Bluetcl command reference and instructions for customization.

D.1 Generate SceMiHeaders

Each data type used on the hardware side must have a corresponding C++ data type on the host side. The following two scripts generate the C++ class files for the objects on the host.

D.1.1 generateSceMiHeaders.tcl

This script reads the `.params` file and generates the files for the necessary C++ classes for the data types and probes found in the `.params` file. At a minimum, the script requires the name of the `.params` file generated by the infrastructure linkage tool and the types to be generated. The following table lists the options that can be used with the script to pass parameters and specify prefixes for generated classes.

Usage:

```
bluetcl -exec generateSceMiHeaders <options> params_file
```

generateSceMiHeaders Options		
Option	Description	Default
`-package` *string*	Top package in the BSV design	
`-p` *path*	Specify the search path for source and intermediate files	
`-bdir` *path*	Specify the `-bdir` where Bluespec-generated `.bo` and `.ba` files are found	
`-outdir` *dir*	Specifies the directory to place generated files	
`-memberPrefix` *string*	Prefix for class member names	`m_`
`-enumPrefix` *string*	Prefix for enum type	`e_`
`-inputs`	Generate header files for all input types	
`-outputs`	Generate header files for all output types	
`-probes`	Generate header files for all probe types	
`-probe-code`	Generate an include file which adds all probes	
`-all`	Generate all header/include files	
`-aliases`	Generate header files for any relevant aliases	
`-vcd <path>`	VCD file name for probes	

The script generates a header file `SceMiHeaders.h` containing the `include` statements for generated types in addition the data class files.

Example:

```
bluetcl -exec generateSceMiHeaders -bdir build -p build:+ -outputs mkBridge.params
```

Output:

```
Loading all packages ...
Loading all packages complete.
Creating header files ...
Header file Maybe_UInt_64.h has been created.
Header file Factresp.h has been created.
Header file SceMiHeaders.h has been created.
Creating header files complete.
```

Contents of SceMiHeaders.h

```
// Automaticlly generated by: ::SceMiMsg
// DO NOT EDIT
// C++ Class with SceMi Message passing for Bluespec type:  All SceMi types
// Generated on: Thu Jul 12 14:18:12 EDT 2012
// Bluespec version: 2012.02.beta1 2012-02-28 27426

#pragma once

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

The script generates the complete C++ `.h` files for the Bluespec types, which in the above example are in the files `Factresp.h` and `Maybe_UInt_64.h`.

D.1.2 generateSceMiMsgData.tcl

The `generateSceMiMsgData` script allows you to specify the BSV package (as opposed to a `.params` file) and also the types to generate. This script generates the `.h` files to define the C++ classes for the data types specified in the command line.

Usage:

```
bluetcl -exec generateSceMiMsgData <options> -package package types [type2]*
```

You must supply one and only one package name and one or more types to be generated. You may need to enclose the type names in quotes.

generateSceMiMsgData Options		
Option	Description	Default
`-p <path>`	Bluespec search path	
`-bdir <dir>`	Bluespec bdir path	
`-outdir <dir>`	Directory for generated header files	
`-memberPrefix str`	Prefix for class member names	`m_`
`-enumPrefix str`	Prefix for enum type	`e_`
`-tagPrefix str`	Prefix for tagged union types	`tag_`

Example:

The DUT in this example has the following structure defined:

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

After compiling the example, use the following command to generate the `.h` files for the type `Factresp`:

```
bluetcl -exec generateSceMiMsgData -p build:+ -package Bridge "Factresp"
```

which returns:

```
Header file Maybe_UInt_64.h has been created.
Header file Factresp.h has been created.
```

D.2 edithdl

The HDL editor, described in Section 7, can be invoked directly from the command line or from the `build` utility.

Usage:

```
bluetcl -exec edithdl <options> <verilog command line>
```

By default, the tool will be run with the graphical user interface. Once you've created a script containing the probe definitions and other edits, you can invoke the tool in a batch mode, automatically running the script and generating a new Verilog file and parameters file.

HDL Editor Options	
Option	Description
`--outputdir <dir>`	Output directory for edited hdl file
`--script <scriptfile>`	Name of script file to run
`--params <paramsfile`	Params file to be edited
`--params-out <edited_paramefile>`	Modified params file
`--blackbox`	Control black boxing of array variables for cosim
`--batch`	Run in batch mode (requires –script)
`--gui`	Run using graphical interface (Default)
`--verbose`	Run in verbose mode
`-p <path>`	Search path for Bluespec .ba files
`-bsvmodule <module>`	BSV module to load

Default: –gui

Example:

```
bluetcl -exec edithdl <topmodule>.v
```

D.3 makedepend

The `makedepend` script reads each `.bsv` sourcefile in sequence and determines all files on which it is dependent. These dependencies are written to a makefile in such a way that `make` will know which object files mut be recompiled when a dependency has changed. The script is based on the unix `makedepend` function for C code.

Usage:

```
bluetcl -exec makedepend  <options> top_package_name
```

A `.bsv` file, or pattern matching file names, must be specified for the top package name, for example, `Bridge.bsv` or `SceMi*.bsv`. The `<options>` arguments are any valid bsc flags.

Example:

```
bluetcl -exec makedepend -p build:+ -bdir build -o dependency.file Bridge.bsv
```

Output:

```
## Automatically generated by bluetcl -exec makedepend -- Do NOT EDIT
## Date: Tue Jul 10 15:15:52 EDT 2012
## Command: bluetcl -exec makedepend -p build:+ -bdir build Bridge.bsv

build/Bridge.bo: Bridge.bsv build/SceMiLayer.bo
build/DUT.bo: DUT.bsv
build/SceMiLayer.bo: SceMiLayer.bsv build/DUT.bo
```

Example:

```
bluetcl -exec makedepend Sce*.bsv
```

Output:

```
## Automatically generated by bluetcl -exec makedepend -- Do NOT EDIT
## Date: Thu Jul 12 10:57:11 EDT 2012
## Command: bluetcl -exec makedepend SceMiLayer.bsv

DUT.bo: DUT.bsv
SceMiLayer.bo: SceMiLayer.bsv DUT.bo
```

D.4 listFiles

The `listFiles` script parses the design specified by the top module name and lists all the Verilog modules of the type specified by the options.

Usage:

```
bluetcl -exec listFiles  <options> top_module_name
```

listFiles Options	
Option	Description
-q	Do not print section headers
-p <path>	Bluespec search path
-bdir <dir>	Bluespec bdir directory
-vdir <dir>	Bluespec vdir directory
-generated	Print synthesized BSV modules
-primitives	Print Bluespec primitive modules
-imported	Print imported modules
-no-inline-fns	Print modules for no-inline functions
-all	Alias for -generated -primitive -imported -no-inline-fns

Example:

```
bluetcl -exec listFiles -p build:+ -bdir build -primitives mkBridge
```

Output:

```
# Bluespec library primitives:
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncReset0.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/CrossingRegUN.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/RWire.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/RegUN.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/MakeClock.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/ClockInverter.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncHandshake.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/CrossingRegN.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/RegN.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/ResetEither.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/FIFO2.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncPulse.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/BypassWire.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/CrossingBypassWire.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncBit05.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncFIFO.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/ClockGen.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncFIFOLevel.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/RWire0.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/MakeReset0.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/MakeResetA.v
/raid/home/kczeck/Builds/work2/bsc/inst/lib/Verilog/SyncResetA.v
```

E BSV library specifications

This section describes the Bluespec SystemVerilog type classes, data types, interfaces, modules and functions which are provided by the SceMi package (`SceMi.bsv`) to support co-emulation in a SCE-MI environment. To access the co-emulation support library, import the SceMi package:

```
import SceMi :: * ;
```

This will import all BSV objects defined in the packages `SceMiCore`, `SceMiXactors`, `SceMiProxies`, `SceMiserialProbe`, `SceMiSharedMemory`, `SceMiScan` and `ModuleContext`. These packages are found in the `$BLUESPECDIR/BSVSource/SceMi` directory.

E.1 Type classes

The `SceMiBuilder` typeclass defines instances of the `buildSceMi` module. Each instance takes a module of type `SceMiModule`, adds in all the necessary SCE-MI infrastructure for the link type, and returns a module of type `Module`. The `SceMiBuilder` typeclass is defined in the `SceMiCore` package.

```
typeclass SceMiBuilder#(type ifc_type, type arg_type, type extended_ifc_type);
   module [Module] buildSceMi#( SceMiModule#(ifc_type) mod
                              , arg_type arg )
                              (extended_ifc_type ifc);
```

TCP, SCEMI This `buildSceMi` instance supports passing in external clocks and resets.

```
instance SceMiBuilder#(i,SceMiClockArgs#(n),i);
   module [Module] buildSceMi#(SceMiModule#(i) mod, SceMiClockArgs#(n) args)(i);
```

V5 PCIE

```
instance SceMiBuilder#(i,SceMiV5PCIEArgs,SceMiV5PCIEIfc#(i,lanes))
   provisos(Add#(1,_,lanes));
 module [Module] buildSceMi#(SceMiModule#(i) mod, SceMiV5PCIEArgs args)
      (SceMiV5PCIEIfc#(i,lanes))
      provisos(Add#(1,_,lanes));
```

Dini PCIE

```
instance SceMiBuilder#(i,SceMiDiniPCIEArgs,SceMiDiniPCIEIfc#(i));
   module [Module] buildSceMi#(SceMiModule#(i) mod, SceMiDiniPCIEArgs args)
      (SceMiDiniPCIEIfc#(i));
```

E.2 Data types

The type `SceMiTime` is used to represent controlled time.

```
typedef UInt#(64) SceMiTime;
```

The type `SceMiCycleStamp` is used to represent cycle stamps.

```
typedef UInt#(64) SceMiCycleStamp;
```

The `SceMiLinkType` data type represents the different linkage types supported for Bluespec SCE-MI.

```
typedef enum { TCP
             , SCEMI
             , PCIE_VIRTEX5
             , PCIE_VIRTEX6
             , PCIE_DINI
             , UNDEFINED
             } SceMiLinkType deriving (Eq,Bits);
```

The `SceMiClockGenType` is an enum structure representing the different clock generation types.

```
typedef enum { FAST
             , FAST_INVERTING
             , REGULAR
             } SceMiClockGenType deriving (Eq,Bits);
```

E.3 SceMiXactors

Bluespec provides hardware-side BSV transactors instantiating the SceMi message ports (`SceMiMessageInPort` and `SceMiMessageOutPort`), along with FIFOs to store the data between the controlled and uncontrolled clock domains. These transactors are defined in the file `$BLUESPECDIR/BSVSource/SceMi/SceMiXactors.bsv` and can be copied and modified by the user. Each transactor provides an interface to the DUT. When the interface is a `Get` or a `Put` interface, a `mkConnection` module is necessary to connect the DUT with the transactor. When the provided interface is `Empty`, the DUT can be directly connected to the transactor.

The transactor message ports correspond message proxies on the software side, as described in Section A.7.

E.3.1 mkInPortXactor

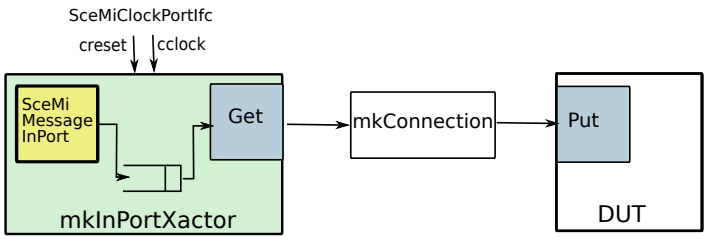

Figure 28: mkInPortXactor

`mkInPortXactor`	Allows a connection to a `Put` interface of a DUT by providing a `Get` interface. When used in a design a `mkConnection` module is required to connect the DUT `Put` to the OutPortXactor `Get`.
	`module [SceMiModule] mkInPortXactor #(SceMiClockPortIfc clk_port ) (Get#(ty)) provisos (Bits#(ty, ty_sz));`

E.3.2 mkPutXactor

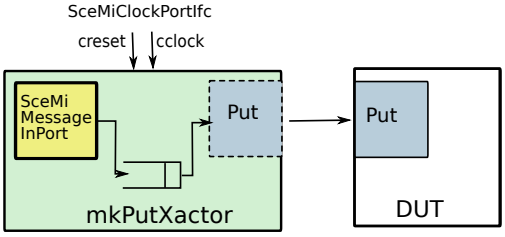

Figure 29: mkPutXactor

`mkPutXactor`	Bluespec module of SCE-MI inport. Directly connects the `Put` interface of a DUT, moving data into the hardware DUT. Takes the DUT's `Put` interface as a parameter.
	`module [SceMiModule] mkPutXactor #(Put#(ty) putifc, SceMiClockPortIfc clk_port ) (Empty) provisos (Bits#(ty, ty_sz));`

E.3.3 mkOutPortXactor

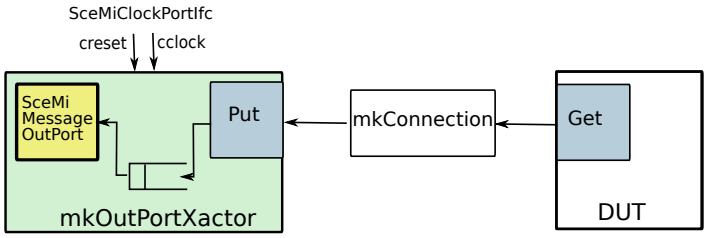

Figure 30: mkOutPortXactor

`mkOutPortXactor`	Allows a connection to a `Get` interface of a DUT by providing a `Put` interface. When used in a design a `mkConnection` module is required to connect the DUT `Get` to the OutPortXactor `Put`.
	`module [SceMiModule] mkOutPortXactor#( SceMiClockPortIfc clk_port ) ( Put#(ty) ) provisos (Bits#(ty, ty_sz));`

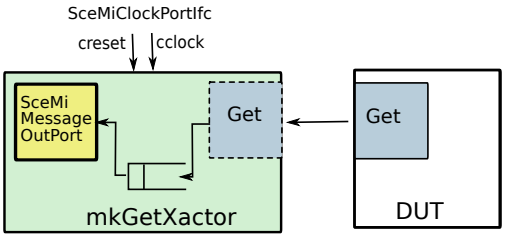

Figure 31: mkGetXactor

E.3.4 mkGetXactor

`mkGetXactor`	Bluespec abstraction of SceMi outport. Directly connects the `Get` interface of a DUT. Takes data from the hardware DUT's `Get` interface as a parameter.
	`module [SceMiModule] mkGetXactor#(Get#(ty) getifc, SceMiClockPortIfc clk_port ) (Empty) provisos (Bits#(ty, ty_sz));`

E.3.5 mkServerXactor

The `mkServerXactor` module connects a Server interface from the DUT to a SCE-MI transactor, as shown in Figure 32

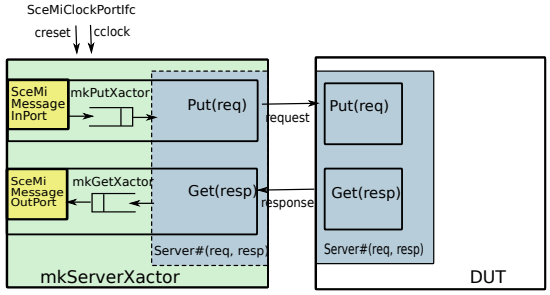

Figure 32: mkServerXactor

`mkServerXactor`	Directly connects to the Server interface of the DUT. Takes the DUT Server interface as a parameter.
	`module [SceMiModule] mkServerXactor#(Server#(req_ty, resp_ty) server, SceMiClockPortIfc clk_port ) (Empty) provisos (Bits#(req_ty, req_ty_sz), Bits#(resp_ty, resp_ty_sz));`

E.3.6 mkClientXactor

The `mkClientXactor` module connects to a DUT with a Client interface as shown in Figure 33.

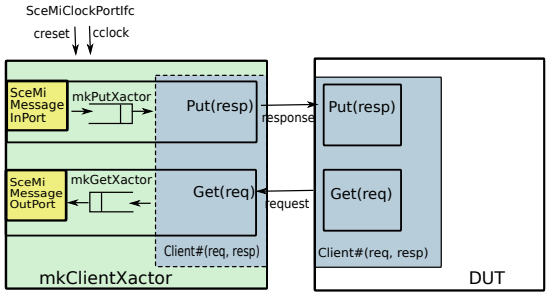

Figure 33: mkClientXactor

`mkClientXactor`	Directly connects to the Client interface of a DUT by taking the DUT interface as a parameter.
	`module [SceMiModule] mkClientXactor#(Client#(req_ty, resp_ty) client, SceMiClockPortIfc clk_port ) (Empty) provisos (Bits#(req_ty, req_ty_sz), Bits#(resp_ty, resp_ty_sz));`

E.3.7 mkValueXactor

`mkValueXactor`	Sends a value to the DUT. Provides a `ReadOnly` interface, using the initial value `initVal`.
	`module [SceMiModule] mkValueXactor#( SceMiClockPortIfc clk_port, ty initVal ) ( ReadOnly#(ty) ) provisos (Bits#(ty, ty_sz));`

E.3.8 mkShutdownXactor

The `mkShutdownXactor` module provides a transactor which corresponds to the testbench `ShutDownXactor`. With this transactor, the simulation is ended from the software testbench side; there is no shutdown control on the hardware side.

`mkShutdownXactor`	Stops the simulation from the testbench side.
	`module [SceMiModule] mkShutdownXactor# ();`

E.3.9 mkSimulationControl

The simulation control transactor, `mkSimulationControl`, allows simulation-like clock control from the software testbench. This transactor connects to the C++ `SimulationControl` transactor on the software side.

Data Types

The data types `SimCommand`, `SimControlReq`, and `SimStatusResp` are defined and used by the `mkSimulationControl` module. These match the C++ classes of the same name that are defined in the file `SimulationControl.h` (Section F.4).

```
typedef enum { Edges,Query,Stop,Resume }
        SimCommand deriving (Eq,Bits);

typedef Bit#(32) SimControlReq;

typedef Bit#(32) SimStatusResp;
```

SimCommand `SimCommand` is an enumerated data type indicating the type of request being sent. The following table describes the meaning of each of the enum value.

Enum	Description
`Edges`	Sends the number of edges for the clock to execute in the simulation.
`Query`	Asks for the number of clocks remaining in the simulation.
`Stop`	Stop the simulation.
`Resume`	Resume the simulation from where it was stopped.

SimControlReq The request to the hardware is of type `SimControlReq`. The `Bit#(32)` data type has two parts: the first 2 bits represent the request command, the remainder represents a count. The count is only used when the command is `Edges`, in which case it is the number of edges of the clock to execute in simulation.

SimControlReq		
	Data Type	Description
`command`	SimCommand	Enum value of the type of request
`count`	`Bit#(30)`	The number of edges for the clock to execute in simulation.

SimStatusResp The response received back from the hardware is of type `SimStatusResp` which is defined as `Bit#(32)`.

SimStatusResp		
	Data Type	Description
`isRunning`	boolean	`True` if the simulation is still running, otherwise `False`.
`isFreeRunning`	boolean	`True` if the simulation is free running, otherwise `False`.
`cyclesRemaining`	`Bit#(30)`	The number of cycles remaining in the simulation. Only used when the response is to a `Query` request.

Module The `mkSimulationControl` module provides control of the SCE-MI clocks from the software-side testbench in both simulation and emulation.

`mkSimulationControl`	Provides simulation-like clock control from the testbench.
	`module [SceMiModule] mkSimulationControl#( parameter SceMiClockConfiguration conf ) ( Empty ifc );`

Functions The functions `command`, `edges`, and `mkStatus` are provided for the `mkSimulationControl` module.

`command`	Returns a type `SimCommand` containing the value of the command part of the request.
	`function SimCommand command(SimControlReq req);`

`edges`	Returns a `UInt#(30)` with the value of the number of edges in the request.
	`function UInt#(30) edges(SimControlReq req);`

`mkStatus`	Returns the response, of type `SimStatusResp`. The response is comprised of the Bit values of `running`, followed by `freerunning` followed by `edge_count`.

	`function SimStatusResp mkStatus(Bool running,`
	`  Bool freerunning,`
	`  UInt#(30) edge_count);`

E.4 SceMiSharedMemory

Package

```
import SceMiSharedMemory :: *;
```

Description

The package `SceMiSharedMemory` defines two types of hardware side memory transactors. The first set are defined within a shared memory context, while the second set are defined within the `SceMiModule` context.

The `mkSharedMem` transactor enables you to map multiple memories in your design to a single physical memory, such as the memory on an FPGA. This transactor uses the ModuleContext mechanism to define a shared memory context, allowing the design to collect memory interfaces, and maintain the compile-time state of these additional items, without changing the structure of the design.

This package also provides the following transactors within the `SceMiModule` context: `mkBRAMXactor`, `mkHostMemXactor`, and `mkHwMemXactor`.

Data types

RAM_Request The RAM request layout is defined by the `Ram_Request` structure, shown in Figure 34. The element `read` (`r` in the figure) is the active-low write enable. The structure also contains the data and the address where the data is located.

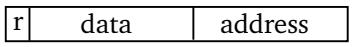

Figure 34: Physical layout of RAM_Request type

```
typedef struct {
   Bool read;

   data_t data;
   addr_t address;
} RAM_Request #(type addr_t, type data_t) deriving(Eq, Bits);
```

RAM_Response The RAM response type, shown in Figure 35, contains just data:

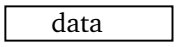

Figure 35: Physical layout of RAM_Response type

```
typedef data_t RAM_Response#(type data_t);
```

E.4.1 Interfaces

The following interfaces are exported from the `SceMiSharedMemory` package.

Interfaces exported from SceMiSharedMemory	
Interface Name	Description
`RAM_Server`	A type of server interface which puts a request and gets a response. This is the type of interface provided by a RAM.
`RAM_Client`	A client interface which gets a request and puts a response. This is the type of interface provided by a RAM client, such as a controller, uP, etc.
`RAM_ServerCmt`	A *Commit* version of the `RAM_Server` interface using `SendCmt` and `RecvCommit` interfaces defined in the `CommitIfc` package.
`RAM_ClientCmt`	A *Commit* version of the `RAM_Client` interface using `SendCmt` and `RecvCommit` interfaces defined in the `CommitIfc` package.
`MemClientIfc`	Combines a `RAM_ClientCmt` interface with a boolean method, `allow`, for the clock signal.
`DutWithSceMiMemIfc`	Combines a DUT interface with a `memClientIfc`.
`MemContextIfc`	Another name for the `MemClientIfc`.
`CompleteMemContextsIfc`	A `Tuple3` interface combining the `MemContextIfc`, `ProbeContextIfc`, and an Empty interface.
`MemContext`	A list of memory client interfaces (`MemContentIfc`).
`Cached_RAM_Server`	

The RAM interfaces defined in the `SceMiSharedMemory` package use the `Client`, `Server` and `ClientServer`, interfaces, defined in the package `ClientServer`.

The package also defines *Commit* versions of the RAM interfaces: `RAM_ServerCmt` and `RAM_ClientCmt`. These versions replace the `Server` and `Client` interfaces with `ServerCommit` and `ClientCommit` interfaces. The Commit interfaces, defined in the package `CommitIfc`, use `SendCmt` and `RecvCommit`

subinterfaces in place of `Get` and `Put` subinterfaces. When modules are connected through the Commit interfaces, an AND gate is *not* added in the connection between the modules, as it is with `Get` and `Put` interfaces.

RAM_Server and RAM_ServerCmt The `RAM_Server` and `RAM_ServerCmt` interfaces define interfaces presented by a RAM. The connected client *puts* the request and *gets* the response.

```
typedef Server#(RAM_Request#(addr_t, data_t), RAM_Response#(data_t))
               RAM_Server #(type addr_t, type data_t);

typedef ServerCommit#(RAM_Request#(addr_t, data_t), RAM_Response#(data_t))
                                 RAM_ServerCmt #(type addr_t, type data_t);
```

RAM_Client and RAM_ClientCmt The `RAM_Client` and `RAM_ClientCmt` interfaces define interfaces presented by a RAM client (controller, uP, etc.). The connected RAM *gets* the request and *puts* the response.

```
typedef Client#(RAM_Request#(addr_t, data_t), RAM_Response#(data_t))
               RAM_Client #(type addr_t, type data_t);

typedef ClientCommit#(RAM_Request#(addr_t, data_t), RAM_Response#(data_t))
                     RAM_ClientCmt #(type addr_t, type data_t);
```

MemClientIfc and MemContextIfc The `MemClientIfc` interface combines a `RAM_ClientCmt` with a boolean `allow` method. The method allows the clock signal.

```
interface MemClientIfc;
   interface RAM_ClientCmt#(Mem_addr_t, Mem_data_t) client;
   (*always_ready*)
   method Bool allow;
endinterface
```

`MemContextIfc` is another name for the `MemClientIfc`.

```
typedef MemClientIfc MemContextIfc;
```

MemContext The structure `MemContext` is a type of `UAList`, defined in the `UnitAppendList` package. It defines the list of client interfaces used in defining the context of a shared memory.

```
typedef struct {
   UAList#(MemContextIfc) ms;
} MemContext;
```

DutWithSceMiMemIfc The `DUTWithSceMiMemIfc` interface combines the interface from a DUT with a `MemClientIfc`.

```
interface DutWithSceMiMemIfc#(type i);
   interface i dutIfc;
   interface MemClientIfc memClient;
endinterface
```

CompleteMemContextsIfc The interface `CompleteMemContextIfc` is a `Tuple3` containing the interfaces provided by the memory context, the probes, and an `Empty` interface.

```
typedef Tuple3#(MemContextIfc, ProbeContextIfc, Empty) CompleteMemContextsIfc;
```

Cached_RAM_Server

```
interface Cached_RAM_Server#(type a, type d, numeric type n);
   interface RAM_Server#(a,d) server;
   method Action flush();
   method Bool flushing();
endinterface
```

E.4.2 mkRAM_Server

`mkRAM_Server`	Instantiates a RAM server module. Provides a `RAM_Server` interface. The address of the RAM server is given as a word address.

	`module mkRAM_Server#(parameter Integer memSize)`
	`(RAM_Server #(addr_t, data_t))`
	`provisos (Bits#(addr_t, ats) ,Bits#(data_t, dts) );`

Example:

```
typedef Bit#(31)  Addr_T;
typedef Bit#(64)  Data_T;

Integer bramSZ = 10*1024;

   RAM_Server#(Addr_T, Data_T) bramMem <- mkRAM_Server(bramSZ) ;
```

E.4.3 Shared Context Transactors

mkInitialCompleteContextWithClocks

`mkInitialCompleteContextWithClocks`	
	Defines an instance of the context used by the memory. Provides the CompleteMemContexts, which is used to define the module context.

	`module mkInitialCompleteContextWithClocks#(`
	`Vector#(CLOCKCONTEXTSIZE, Clock) cks,`
	`Vector#(CLOCKCONTEXTSIZE, Reset) rs)`
	`(CompleteMemContexts);`

runWithSharedMemory

runWithSharedMemory	
	Converts the module out of the memory context and provides explicit interfaces for the context.

	`module [ModuleContext#(c1)]`
	`runWithSharedMemory#(ModuleContext#(HCons#(MemContext,c1), i) mkM)`
	`(DutWithSceMiMemIfc#(i))`
	`provisos (Context#(ModuleContext#(c1),ClockContext));`

mkSharedMem This module instantiates a local cached memory model, `Cached_RAM_Server` interface.

mkSharedMem	Instantiates a shared RAM server. You can directly replace an instantiation of a `mkRegMem`.

	`module [mc] mkSharedMem#(Mem_addr_t base, Bool write_through)`
	`(Cached_RAM_Server#(addr_t, data_t, idxSize))`
	`provisos (IsModule#(mc, _a),`
	`Context#(mc, MemContext), Context#(mc, ClockContext),`
	`Bits#(addr_t, asz), Bits#(data_t, dsz),`
	
	`Mul#(dsz, a__, Mem_data_size),`
	`Add#(dsz, b__, Mem_data_size),`
	`Log#(a__, c__),`
	`Mul#(d__, 8, dsz),`
	`Add#(e__, c__, asz),`
	`Add#(e__, f__, Mem_addr_size),`
	`Add#(idxSize, g__, e__) );`

Example:

```
‘SynthBoundaryWithClocks(mkDut, Dut)
module [DutModule] mkDut0 (Dut);

   // Instances of 2 memories and their testers
   //RAM_Server#(Addr_T, Data_T) regMem <- mkRegMem(regfileSZ);
   Cached_RAM_Server#(Addr_T, Data_T, 5) cram1 <- mkSharedMem(0,True);
   let regMem = cram1.server;

   //RAM_Server#(Addr_T, Data_T) bramMem <- mkRAM_Server(bramSZ) ;
   Cached_RAM_Server#(Addr_T, Data_T, 5) cram2 <- mkSharedMem(1024,False);
   let bramMem = cram2.server;
```

E.4.4 SceMiModule Transactors

The modules in this section define transactors in the `SceMiModule` context.

mkBRAMXactor BRAM backed memory store with a SceMi Client Xactor.

The `mkBRAMXactor` implements the central memory as a BRAM. The `memSize` is the size of the provided memory, in 64-bit word units. The transactor creates an Inport and an Outport to form a `RAM_Server` for the central memory accessible from the host, allowing downloading of initial images and/or interrogation thereafter.

`mkBRAMXactor`	Implemements the central memory as a BRAM. Note: this instantiates a module in the `SceMiModule` context.

	`module [SceMiModule] mkBRAMXactor#(parameter Integer memSize`
	`, parameter SceMiClockConfiguration conf`
	`, MemClientIfc memuser)`
	`(Empty);`

mkHostMemXactor The `mkHostMemXactor` connects to a central memory provided on the software testbench. This transactor communicates with the corresponding C++ transactor `SparseMemXactor` on the software side. This transactor does not provide inport and outport communications across the link for loading and interrogating the central memory from the host. Instead, the `SparseMemXactor` provides `readMem` and `writeMem` methods for this purpose.

`mkHostMemXactor`	Connects to a central memory provided on the host. Note: this instantiates a module in the `SceMiModule` context.

	`module [SceMiModule] mkHostMemXactor#(`
	`parameter SceMiClockConfiguration conf`
	`, MemClientIfc memuser)`
	`(Empty);`

InternalMemXactor The `InternalMemXActor` is a typeclass which defines the hardware memory transactor modules. The module has two arguments: a `SceMiClockConfiguration`, and a `MemClientIfc` interface. There are two instances defined for the typeclass, one where the `MemClientIfc` is a `RAM_Client` interface, and one for which it is a `RAM_ClientCmt` interface.

```
typeclass InternalMemXActor #(type ifc);
   module [SceMiModule] mkHwMemXactor#(
                           parameter SceMiClockConfiguration conf
                         , MemClientIfc memuser)
                           (ifc);
```

Each instance adds a SCE-MI Client transactor in to the `MemClient`. The first provides a `RAM_client`, the second a `RAM_ClientCmt`.

mkHwMemXactor	Adds a SceMi Client Xactor in to the MemClient. Each instance provides a different type of RAM Client. The first instance provides a RAM_Client interface, the second a RAM_ClientCmt interface.
	instance InternalMemXActor#(RAM_Client#(Mem_addr_t, Mem_data_t)); module [SceMiModule] mkHwMemXactor#(parameter SceMiClockConfiguration conf , MemClientIfc memuser) (RAM_Client#(Mem_addr_t, Mem_data_t));
	instance InternalMemXActor#(RAM_ClientCmt#(Mem_addr_t, Mem_data_t)); module [SceMiModule] mkHwMemXactor#(parameter SceMiClockConfiguration conf , MemClientIfc memuser) (RAM_ClientCmt#(Mem_addr_t, Mem_data_t));

E.4.5 Using module context with shared memory

The implementation of shared memory utilizes the ModuleContext package, described in the *Bluespec SystemVerilog Reference Manual*. This package provides the capability of accumulating items (other than rules and state, which are always accumulated in a Bluespec module) and maintaining the compile-time state of these additional items, without changing the structure of the original design.

The ModuleContext mechanism allows the desginer to *hide* the details of the additional interfaces. Before the module can be synthesized, it must be converted (or *exposed*) into a module containing only rules and state elements, as the compiler does not know how to handle the other items. The ModuleContext package provides the mechanism to allow additional items to be collected, processed, and exposed. In the SceMiSharedMemory package, ModuleContext is used to collect memory interfaces, along with other Emulation App elements, such as probe and clock interfaces.

When using the ModuleContext mechanisms, all modules within the hierarchy must be in the same *context*, that is, they must all have the same module type. For BSV modules, the default module type is Module, but you can define other BSV module types as well. The ModuleContext mechanism allows you to define a module type based on context.

The ModuleContext package defines the typeclass Context. A Context typeclass has two type parameters: a module type (mc1) and a context (c2).

```
typeclass Context#(type mc1, type c2);
   module [mc1] getContext(c2) provisos (IsModule#(mc1, a));
   module [mc1] putContext#(c2 s)(Empty) provisos (IsModule#(mc1, a));
endtypeclass
```

A regular module type (Module) is an instance of Context where the context (c2) is void, that is, it isn't collecting anything other than states and rules.

In the SceMiSharedMemory package the RAM client interfaces are collected along with other SCE-MI objects, specifically probe contexts and clock contexts. The first context defined is the MemContext, a UAList [2] of RAM client (MemclientIfc) interfaces.

[2] A UAList is a structure defined in the package UnitAppendList which is a specific type of list used with the ModuleCollect mechanism.

The `MemContext` is collected along with the `ProbeContext` and `ClockContext` to define `CompleteMemContexts`. The shared memory module type is an instance of `Context` where the context is `CompleteMemContexts`: `ModuleContext#(CompleteMemContexts)`.

```
typedef HList3#(MemContext, ProbeContext, ClockContext) CompleteMemContexts
```

We can rename the contexts and the module type for ease of use in building examples:

```
typedef CompleteMemContexts           DutContext;
typedef ModuleContext#(DutContext)    DutModule;
```

Within each layer of the module hierarchy, up to the SceMiLayer, a synthesis boundary can be added, exposing the context as an interface. Through the exposed interface you can view the signals in and out of the module. The interface is then buried back into the context to synthesize the modules higher up the hierarchy.

Example, adding a synthesis boundary at the `mkTb` module, where the module type is `DutModule`:

```
‘SynthBoundaryWithClocks(mkTb, Tb)
module [DutModule] mkTb0 (Tb);
```

To use the shared memory within Emulation App, the memory context must be exposed in the SceMiLayer, so that the module can be used in the `SceMiModule` context. The `runWithSharedMemory` module pulls the module out of the `DutContext` so that it can be used in the `SceMiModule` context. Example:

```
DutWithSceMiMemIfc#(Tb) tb <- buildDut(runWithSharedMemory(mkTb), clk_port);
```

E.4.6 Example using SharedMemory

This section has excerpts from packages defining the context, the DUT, and the SceMiLayer, demonstrating some of the unique features used with shared memory.

Defining the Context The module context can be defined and created within a separate file, in this example `DUTContext.bsv`.

```
import HList::*;
import SceMi::*;
import Contexts::*;
import ModuleContext::*;
import SceMiSharedMemory::*;

//Rename for clarity and ease of use
typedef CompleteMemContexts              DutContext;
typedef ModuleContext#(DutContext)       DutModule;

module mkInitialDutContextWithClocks#(Vector#(CLOCKCONTEXTSIZE, Clock) cks,
                                      Vector#(CLOCKCONTEXTSIZE, Reset) rs)
                                     (DutContext);
...
endmodule
```

Defining the shared memory DUT The example DUT has the following features:

- The package `DutContext`, which defines the context, is imported.

```
import SceMiSharedMemory::*;
import Connectable::*;
import ClientServer::*;
import Defines::*;
import DutContext::*;
import ModuleContext::*;
```

- The DUT module, `mkDUT`, has a `SynthBoundaryWithClocks` statement to create a synthesis boundary exposing the module interface. This allows you to view the signals into and out of the module.

```
‘SynthBoundaryWithClocks(mkDut, Dut)
module [DutModule] mkDut0 (Dut);
```

- The module replaces regular memories, defined by `mkRegMem` and `mkRAM_Server` statements, with shared memories, defined by `mkSharedMem` statements.

```
    // Instances of 2 memories and their testers
    //RAM_Server#(Addr_T, Data_T) regMem <- mkRegMem(regfileSZ);
    Cached_RAM_Server#(Addr_T, Data_T, 5) cram1 <- mkSharedMem(0,True);
    let regMem = cram1.server;

    //RAM_Server#(Addr_T, Data_T) bramMem <- mkRAM_Server(bramSZ) ;
    Cached_RAM_Server#(Addr_T, Data_T, 5) cram2 <- mkSharedMem(1024,False);
    let bramMem = cram2.server;
```

Modifying the SceMiLayer The `buildDut` statement takes `runWithSharedMemory` as an argument. The `runWithSharedMemory` statement converts the module out of the `DutModule` context and creates explicit interfaces for those contexts.

```
DutWithSceMiMemIfc#(Dut) dut <- buildDut(runWithSharedMemory(mkDut), clk_port);
```

E.5 Synthesizable testbench library

Bluespec provides a BSV library package that can be used to create the proxies required to build a SceMi compatible testbench module. Since these proxies are written with Bluespec SystemVerilog (BSV), a testbench using this facility will be synthesizable. A synthesizable testbench can be put on the eumulation platform, along with the rest of the system.

To access the BSV SceMi Testbench Library, import the SceMiProxies package:

```
import SceMiProxies :: * ;
```

The SceMiProxies package provides message input and output port proxies parameterized by the type of the message payload. Each port proxy definition includes an interface and a module providing the interface. Also provided are ToGet and ToPut instances to define interfaces as part of the toGet and toPut typeclasses. The SW port proxies correspond directly to the HW message ports.

E.5.1 SceMiMessageInPortProxy

The message inpput port proxy `mkSceMiMessageInportProxy` exposes the `SceMiMessageInPortProxyIfc#(msg_type)` interface with the following methods:

SceMiMessageInPortProxyIfc				
Method			Argument	
Name	Type	Description	Name	Description
`accepting_data`	`Bool`	Returns `True` when the corresponding SceMiMessageInPort is ready to accept data.		
`send`	`Action`	Sends data from an input port proxy to the corresponding SceMiMessageInPort.	`msg`	data of type *msg_type*

```
interface SceMiMessageInPortProxyIfc#(type msg_type);
   (* always_ready *)
   method Bool accepting_data();
   method Action send(msg_type msg);
endinterface: SceMiMessageInPortProxyIfc
```

The `accepting_data` method on an input port proxy will return `True` when it is allowed to send data from the port proxy. The `send` method is used to transmit the data.

The `toPut` function can be applied to a `SceMiMessageInPortProxyIfc` interface to convert it to a `Put` interface:

```
instance ToPut#(SceMiMessageInPortProxyIfc#(a),a);
```

The module `mkSceMiMessageInPortProxy` instantiates an inport proxy which conforms to the SCE-MI standard. This is the SW side of the BSV module `mkSceMiMessageInPort` (Section A.4.1).

`mkSceMiMessageInPortProxy`	Instantiates an input port proxy of the correct type based on the `SceMiLinkType` module parameter.

	`module [SceMiModule] mkSceMiMessageInPortProxy#(`
	`  String paramFile`
	`, String transactorName`
	`, String portName)`
	`(SceMiMessageInPortProxyIfc#(a) ifc)`
	`  provisos(Bits#(a,sz));`

The `paramFile` input parameter specifies the name of the parameter file generated by scemilink infrastrucure linkage tool (section B.2).

The `transactorName` and `portName` are input parameters that will be used to map the SceMiMessageInPortProxy to the corresponding SceMiMessageInPort with the same transactor and port names on the HW side.

E.5.2 SceMiMessageOutPortProxy

The message output port proxy, `mkSceMiMessageOutPortProxy` exposes the `SceMiMessageOutPortProxyIfc#(msg_type)` interface with the following methods:

SceMiMessageOutPortProxyIfc		
Method		
Name	Type	Description
`has_data`	`Bool`	Returns `True` when data from the corresponding SceMiMessageOutPort is available on an output port proxy.
`read`	*msg_type*	Returns the data from the corresponding SceMiMessageOutPort on an output port proxy, if any.
`cycle_stamp`	`SceMiCycleStamp`	Returns the cycle stamp integer id of the most recent call to read method.
`shutdown`	`Action`	Shutdown the SceMi infrastructure.

```
interface SceMiMessageOutPortProxyIfc#(type msg_type);
   (* always_ready *)
   method Bool has_data();
   method msg_type read();
   (* always_ready *)
   method SceMiCycleStamp cycle_stamp();
   method Action shutdown();
endinterface: SceMiMessageOutPortProxyIfc
```

The `has_data` method on an output port proxy will return `True` when there is data available on the port. The data can be accessed via the `read` method.

The `cycle_stamp` method on the output port proxy returns an integer value of the number of cycles transpired during the simulation.

The `shutdown` method on the output port proxy stops and shutdowns the entire SceMi infrastructure. This method should only be called once on one of the instantiated SceMiMessageOutPortProxy at the end of a simulation run.

The `toGet` function can be applied to a `SceMiMessageOutPortProxyIfc` interface to convert it to a `Get` interface:

```
instance ToGet#(SceMiMessageOutPortProxyIfc#(a),a);
```

The module `mkSceMiMessageOutPortProxy` instantiates an outputport proxy which conforms to the SCE-MI standard. This is the SW side of the BSV module `mkSceMiMessageOutPort` (Section A.4.2).

`mkSceMiMessageOutPortProxy`	Instantiates an output port of the correct type based on the `link_type` module parameter.

	`module [SceMiModule] mkSceMiMessageOutPortProxy#(`
	`  String paramFile`
	`, String transactorName`
	`, String portName)`
	`(SceMiMessageOutPortProxyIfc#(a) ifc)`
	`  provisos(Bits#(a,sz));`

The `paramFile` input parameter specifies the name of the parameter file generated by scemilink infrastructure linkage tool (section B.2).

The `transactorName` and `portName` are input parameters that will be used to map the SceMiMessageOutPortProxy to the corresponding SceMiMessageOutPort with the same transactor and port names on the HW side.

F C++ library specifications

To connect a software testbench to the SCE-MI emulation platform, the SW testbench module must contain SCE-MI port proxies that correspond to the SCE-MI message ports on the HW side, as described in Section A.7. Bluespec provides two distinct libraries for creating a SCE-MI compatible testbench module:

- A complete library of C++ proxies and transactors for developing a testbench in SystemC or C++ utilizing the standard un-timed SCE-MI testbench API
- A library of components for developing the software testbench in BSV (Section E.5)

Bluespec provides a large library of port-proxies, transactors, and utilities for creating a C++ SCE-MI testbench. The files are distributed in the Bluespec release in the `$BLUESPECDIR/Scemi/bsvxactors` directory.

When using any of the Bluespec-provided library of C++ components you must include the file `bsv_scemi.h` at the top of your C++ `.h` files.

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

F.1 Templated classes

Most of the Bluespec-provided C++ testbench library classes are templated classes; the methods provided can be used with different data types. Whenever you see `T` in a method statement, it represents a templated type. For example, let's look at the `getMessage` method in the class `OutportQueueT` (F.2.5):

```
T getMessage ()
```

The return value is a datatype of `T`, representing the datatype defined for `SceMiMessageData`. Different implementations have different definitions for `SceMiMessageData`, but the `getMessage` method can be used with any of the implementations.

The methods in the data type classes (F.3), including the `BSVType` class, are all templated. For example, the meaning of the `idx` argument in the `getMember` method depends on the implemented data type:

```
virtual BSVType * getMember (unsigned int idx)
```

If the `BSVType` is a `BSVVectorT`, `idx` is the element index. If the `BSVType` is a `BitT` value, `idx` is the word index. The implementation depends on the instantiated data type. The `BSVVectorT` and `BitT` classes are also templated; both are parameterized on their size and, for the vector class, the type of elements.

F.2 Port proxies

F.2.1 Common methods

The following accessors are defined in the proxies `InportProxyT`, `InportQueueT`, `OutportProxyT`, and `OutportQueueT`.

PortWidth	The width of the port, as defined by SCE-MI.
	`unsigned int PortWidth () const`

PortName	The name of the port, as defined by SCE-MI.
	`const char* PortName () const`

TransactorName	The name of the transactor, as defined by SCE-MI.
	`const char* TransactorName() const`

setDebug	Sets the debug flag to the boolean value.
	`void setDebug(bool val)`

F.2.2 InportProxyT

The `InportProxyT` wraps the SceMi C++ API functions that are used to send messages to the hardware side. It combines methods used to initializes the ports and controls blocking. There are three different versions of the send method: `sendMessage`, `sendMessageNonBlocking`, and `sendMessageTimed`

InportProxyT	Constructor for the `InportProxyT` class.
	`InportProxyT (const std::string & hier, const std::string & instname, SceMi *scemi)`

sendMessage	This method blocks until the message is sent and should not be called from a SCE-MI call back as deadlock will occur.
	`void sendMessage (const T & t)`

sendMessageNonBlocking	This method does not block, returning true if the data is sent.
	`bool sendMessageNonBlocking (const T & t)`

`sendMessageTimed`	This method blocks, but with a timeout, returning true if the data is sent. The is an overloaded function; the compiler determines which version to use based on the arguments provided. The first takes a delta time, while the second version takes a `timespec` struct, which is an absolute time.

	`bool sendMessageTimed (const T & t, const time_t & delta_seconds, const long & delta_microseconds=0)`
	
	`bool sendMessageTimed (const T & t, struct timespec *expiration )`

F.2.3 InportQueueT

Like `InportProxyT`, the `InportQueueT` class is the C++ side of the BSV module `mkSceMiMessageInport`. This combines `InportProxyT` and `WaitQueueT` to allow a non-blocking send.

`InportQueueT`	Constructor for the `InportQueueT` class.
	`InportQueueT (const std::string & hier, const std::string & instname, SceMi *scemi)`

`sendMessage`	This method will block for control of the state, but will not otherwise wait for some action from the SceMi service thread, so it is safe to use inside the SceMi service thread.
	`void sendMessage (const T & t)`

F.2.4 OutportProxyT

The `OutportProxyT` class is the C++ side of the BSV module `mkSceMiMessageOutport`. It initializes the SceMiPort, including bindings, and controls receiving messages from the HW side.

`OutportProxyT`	Constructor for the `OutportProxyT` class.
	`OutportProxyT (const std::string & hier, const std::string & instname, SceMi *scemi)`

`setCallBack`	This callback is executed when the SceMi transactor has data. The first argument is the pointer given when the callback is registered and the second argument is the message received.
	`void setCallBack( void func (void *, const T &), void * ptr)`

F.2.5 OutportQueueT

Like `OutportProxyT`, the `OutportQueueT` class is the C++ side of the BSV module `mkSceMiMessageOutport`. This combines `OutportProxyT` and `WaitQueueT`.

`OutportQueueT`	Constructor for the `OutportQueueT` class.
	`OutportQueueT (const std::string & hier, const std::string & instname, SceMi *scemi)`

`getMessage`	This method 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. Returns the message.
	`T getMessage ()`

`getMessageNonBlocking`	This method does not block if there is no message to receive. It returns `True` if a message is received.
	`bool getMessageNonBlocking (T &t)`

`getMessageTimed`	This method blocks if there is no message to receive, but there is a timeout. Returns `True` if a message is received. The method is overloaded; the compiler will determine the correct version of the method to use based on the arguments provided. The first version takes a delta time, while the second takes an absolute time.

	`bool getMessageTimed (T &t, const time_t & delta_seconds, const long & delta_microseconds=0)`
	
	`bool getMessageTimed (T &t, struct timespec *expiration)`

F.3 Data type utilities

As discussed in Section F.1, all the provided data type utilities are templated classes; their implementations may depend on the instantiated data type.

F.3.1 BSVType

The `BSVType` class is a pure virtual base class for BSV types. All BSV types inherit the methods defined in this class.

The class defines two types: `BSVKind` and `PutTo`. `BSVKind` is an enum defining the different supported BSVTypes. `PutTo` defines the `operator<< ()` data type.

```
enum BSVKind {BSV_Primitive,
              BSV_Struct,
              BSV_TaggedUnion,
              BSV_Enum,
              BSV_Vector} ;

typedef std::ostream & PutTo ( std::ostream &os, const BSVType & x);
```

`setPutToOveride`	Allows user to override the `PutTo` (`operator<<()`) method for an BSVType class.
	`virtual void setPutToOverRide (PutTo *func) const;`

`getBSVType`	Adds the BSVType as a string to the return stream. Note that the return stream and the method argument are the same.
	`virtual std::ostream & getBSVType (std::ostream & os) const`

`getBitSize`	Returns the number of bits that are in the data type.
	`virtual unsigned int getBitSize () const`

`getBitString`	Outputs the value in the data type as a binary pattern stream.
	`virtual std::ostream & getBitString (std::ostream &os) const`

`getClassName`	Returns a char pointer of the class name.
	`virtual const char * getClassName() const`

`getKind`	Returns the data type as a BSVKind enum value.
	`virtual BSVKind getKind() const`

`getTaggedUnionTag`	Returns the tag value of a tagged union or 0 if the data value is not a tagged union structure.
	`virtual SceMiU32 getTaggedUnionTag () const`

getTaggedUnionWidth	Returns the width in bits the tag for a tagged union can be.
	`virtual unsigned int getTaggedUnionTagWidth () const`

getMemberCount	Returns the number of members in a given type.
	`virtual unsigned int getMemberCount () const`

getMember	Returns the member of a given type at the index value provided.
	`virtual BSVType * getMember (unsigned int idx)`

getMemberName	Returns the name of the member at the index value provided.
	`virtual const char * getMemberName (unsigned int idx) const`

F.3.2 Common methods

The `setMessageData` and `operator<<` methods are defined for the data type classes: `BitT`, `BSVVectorT`, `BSVVoid`, and `StampedT`.

setMessageData	Populates SceMiMessageDataInterface.
	`unsigned int setMessageData ( SceMiMessageDataInterface &msg, const unsigned int off=0 ) const`

operator<<	Formats the data element value into an output stream, where T is the data type.
	`friend std::ostream & operator<< (std::ostream &os, const T &d)`

F.3.3 BitT

`BitT` is the C++ class for Bluespec's `Bit#(n)` type where a `BitT` is an array of 32 bit words. The C++ class has `N` bits and uses `SceMiU32` as the word type for all `N>0`. The number of words in a `BitT` is `(N+31)/32`. The `BitT` class inherits all methods defined in the `BSVTypes` class.

The following table shows the values returned for the methods inherited from the class `BSVType`.

Method Name	Value Returned
getClassName	BitT
getBSVType	Bit#(N)
getBitSize	N
getKind	BSV_Primitive
getMemberCount	0
getMemberName	0
getMember	0

BitT	Constructor for the BitT class. Only the first word is constructed.
	`BitT (SceMiU32 d=0)`

BitT	Constructor for the BitT class from SCE-MI message.
	`BitT (const SceMiMessageDataInterface *msg, unsigned int &off)`

get	Returns the LSB word, returned as a SceMiU32.
	`SceMiU32 get () const`

get64	Returns an unsigned long long (64 bits) if the value of N allows it.
	`unsigned long long get64 () const`

getWord	Returns the word at the index idx as a ScemiU32.
	`SceMiU32 getWord ( unsigned int idx ) const`

setWord	Place the data in the SceMiU32 word d into BitT at index idx.
	`void setWord ( unsigned int idx, SceMiU32 d)`

setBit	Place the data in the SceMiU32 word d into BitT at index idx.
	`void setBit( unsigned int bitidx, SceMiU32 d )`

getBit	Return the data as a SceMiU32 word at index bitidx.
	SceMiU32 getBit(unsigned int bitidx) const

F.3.4 BSVVectorT

The file BSVVectorT.h provides the C++ equivalent type of Vector#(n,t) from BSV. It has the constructors and methods to convert BSV Vector types to and from SceMiMessages. This class inherits all methods defined in the BSVType class.

The following table shows the values returned for the methods inherited from the class BSVType.

Method Name	Value Returned
getClassName	VectorT
tegetBSVType	Vector#(N,T)
getBitSize	N*Bit size of data type T
getKind	BSV_Vector
getMemberCount	N
getMemberName	Vidx, where idx is the value of idx
getMember	Reference to the element at index idx

BSVVectorT	Constructor for the BSVVectorT class.
	BSVVectorT (const SceMiMessageDataInterface *msg, unsigned int &off)

operator[ ]	Returns a reference to the element at position idx in the vector container.
	T & operator[ ] (size_t idx) const T & operator[ ] (size_t idx) const

F.3.5 BSVVoid

The file BSVVoid.h provides the C++ equivalent type of the Bluespec void type. It has the constructors and methods to convert BSV void type to and from SceMiMessages. The BSVVoid class inherits all methods defined in the BSVTypes class.

BSVVoid	Constructor for the BSVVoid class.
	BSVVoid (const SceMiMessageDataInterface *msg, unsigned int &off)

F.3.6 StampedT

The file StampedT.h provides a wrapper class for a BSVType, adding the SceMiMessage timestamp with the data. A timestamp cannot be added to an outgoing message.

Data Types The following data types are defined in the `StampedT` class.

Name	Definition	Description
`m_time_stamp`	`SceMiU64`	Data type of the timestamp
`m_data`	`T`	Polymorphic data type of the data

`StampedT`	Constructor for the `StampedT` class.
	`StampedT (const SceMiMessageData *msg, unsigned int &off)`

`getTimeStamp`	Returns the timestamp
	`SceMiU64 getTimeStamp() const`

`getData`	Returns the data.
	`const T & getData() const`

F.4 Simulation control

The header file `SimulationControl.h` provides the C++ transactor side that communicates with the Bluespec simulation control module `mkSimulationControl`.

The file `SimulationControl.h` contains three classes: `SimControlReq` for control requests, `SimStatusResp` for status responses, and `SimulationControl` which controls sending and receiving messages.

The types of the data elements are specified in the BSV file, `SceMiXactors.bsv`, provided in the Bluespec distribution. The classes provided here match the size and types defined in `SceMiXactors.bsv`.

F.4.1 Data type `SimCommand`

`SimCommand` is an enumerated data type indicating the type of message being sent. The following table describes the meaning of each of the enum value. These match the BSV type `SimCommand` defined in `SceMiXactors.bsv`.

Enum	Description
`Edges`	Sends the number of edges for the clock to execute in the simulation.
`Query`	Asks for the number of clocks remaining in the simulation.
`Stop`	Stop the simulation.
`Resume`	Resume the simulation from where it was stopped.

```
enum   SimCommand { Edges, Query, Stop, Resume };
```

F.4.2 SimControlReq

A `SimControlReq` has two parts: the command, which is of type `SimCommand`, and a count. The count is only used when the command is `Edges`. The data definitions of `SimControlReq` matches the definition of the BSV type `SimControlReq` in the file `SceMiXactors.bsv`.

SimControlReq	Constructor for the SimControlReq class.
	SimControlReq (SimCommand cmd, unsigned int count)

setMessageData	Populates SceMiMessageData.
	unsigned int setMessageData (SceMiMessageData &msg, const unsigned int off=0) const

toString	Converts the SimCommand values to string values for output.
	static const char * toString (const SimCommand & c)

operator<<	Formats SimControlReq into an output stream.
	friend ostream & operator<< (ostream &os, const SimControlReq &req)

F.4.3 SimStatusResp

The response is received back from the hardware by the class SimStatusResp. The response has two parts, message data and offset; the offset is only meaningful when the response is to a Query request. The data definitions of SimStatusResp matches the definition of the BSV type SimStatusResp in the file SceMiXactors.bsv.

SimStatusResp	Constructor for the SimStatusResp class.
	SimStatusResp (const SceMiMessageData *msg, unsigned int & off)

isRunning	Returns a boolean indicating whether the simulation is still running.
	bool isRunning()

isFreeRunning	Returns a boolean indicating whether the simulation is free running.
	bool isFreeRunning()

cyclesRemaining	Returns the number of cycles remaining in the simulation.
	unsigned int cyclesRemaining()

operator<<	Formats SimStatusResp into an output stream.
	`friend ostream & operator<< (ostream &os, const SimStatusResp &resp)`

F.4.4 SimulationControl

The SimualationControl class sends messages to start and stop the simulation, and receives messages about the status of the simulation.

SimulationControl	Constructor for the SimulationControl class.
	`SimulationControl (const string & hier,` `  const string & inst, SceMi *scemi)`

sendCommand	Sends messages to the hardware side. There are two sendCommand methods.

	`void sendMessage (const SimControlReq &control)`
	`void sendMessage (const SimCommand cmd,`
	`  const unsigned int edges =0 )`

getStatus	Receives the message from the hardware side. This function blocks if there is no message to receive so should not be called from a SCE-MI call back as deadlock will occur.
	`SimStatusResp getStatus ()`

getStatusNonBlocking	Returns true if data is received. This version doesn't block if there is no message.
	`bool getStatusNonBlocking (SimStatusResp &t)`

getStatusTimed	Returns true if data is received. This version blocks, but with a timeout. This method is overloaded, the compiler will pick the correct version based on the arguments. The first version uses a delta time while the second uses an absolute time.

	`bool getStatusTimed (SimStatusResp &t, const time_t & delta_seconds,`
	`  const long & delta_microseconds=0)`
	
	`bool getStatusTimed (SimStatusResp &t,`
	`  struct timespec *expiration)`

F.5 ShutdownXactor

The header file `ShutdownXactor.h` provides the `ShutdownXactor` class, the C++ transactor side for the Bluespec shutdown transactor module. It communicates with the Bluespec module `mkShutdownXactor` to stop the simulation.

`ShutdownXactor`	Constructor for the `ShutdownXactor` class.
	`ShutdownXactor (const string hier, const string instname, SceMi *scemi)`

`blocking_send_finish`	Sends a finish message to the testbench.
	`void blocking_send_finish()`

`is_finished`	Receives a Bool indicating if the method is finished.
	`bool is_finished()`

F.6 SerialProbeXactor

The `ProbesXactor` is defined in the header file `SerialProbeXactor.h`. The `ProbesXactor` handles the probes instantiated on the hardware side and through the HDL editor.

`ProbesXactor`	Initiates a Probe transctor.
	`static ProbesXactor *init(const std::string & hier, const std::string & instname, const char *file_base, SceMi *scemi);`

`shutdown`	Deletes the static ProbesXactor.
	`static void shutdown();`

`enable`	Enables the probe or capture specified by probeNum.
	`void enable(unsigned int probeNum);`

`disable`	Disables the probe or capture specified by probeNum.
	`void disable(unsigned int probeNum);`

enabled	Returns the state of the probe specified by probeNum
	`unsigned int enabled(unsigned int probeNum);`

setDebug	Sets the debug flag to the boolean value.
	`void setDebug(bool val)`

F.7 Utiliites

F.7.1 WaitQueue

Class defines a queue for multithreaded programs where the process receiving from the queue can block until data is available.

WaitQueueT	Constructor for the `WaitQueueT` class.
	`WaitQueueT ()`

put	Push data onto the back of the queue.
	`void put(const T& t)`

get	Retrieve the data from the hardware side. This method blocks if there is no message to receive.
	`T get ()`

getNonBlocking	Retrieve the data from the hardware side. This version does not block, returns true if data is received.
	`bool getNonBlocking (T &t)`

`getTimed`	Retrieve the data from the hardware side. This version blocks, but with a timeout. This method is overloaded; the compiler will select the correct version based on the arguments. The first version uses a delta time while the second uses an absolute time.

	`bool getTimed (T &t, const time_t & delta_seconds, const long & delta_microseconds=0)`
	
	`bool getTimed (T &t, struct timespec *expiration)`

F.7.2 Thread

The `Thread` class is a pure virtual class. It is the base class for the `SceMiServiceThread`.

`start`	Starts a thread, returns a runtime error if the thread is already running.
	`void start()`

`stop`	Sets a stop flag and called process is responsible to terminate the action.
	`virtual void stop()`

`signal`	Send signal to process.
	`int signal(int sig)`

`join`	Set stop flag and waits for process to stop.
	`void join()`

F.7.3 SceMiServiceThread

The `SceMiServiceThread` is a specialization of `Thread` and is used to start and run the SCE-MI service thread, without using SystemC.

`SceMiServiceThread`	Constructor for the `SceMiServiceThread` class.
	`SceMiServiceThread( SceMi *pSceMi)`

F.7.4 Target

The `Target` class abstracts the target location for output. It is used to allow the same functions to generate strings to a file or to a buffer.

`Target`	Constructor for the `Target` class.
	`Target() : errors(std::list<std::string>()) {};`

`add_error`	Adds an error to the output string.
	`void add_error(const char* error)`

`handle_errors`	Outputs the error output string to the target.
	`void handle_errors()`

`write_char`	Formats different types of characters for output.
	`virtual void write_char(char c) = 0;` `virtual void write_char(char c, unsigned int count) = 0;`

`write_string`	Formats strings for output.
	`virtual void write_string(const char* fmt ...) = 0;`

`write_data`	Formats data for output.

	`virtual void write_data(const void* data,`
	`unsigned int size,`
	`unsigned int num) = 0;`

F.7.5 FileTarget

The `FileTarget` class sends output to a file. It has the same functions as the `Target` class: `write_char`, `write_string`, `write_data`.

F.7.6 BufferTarget

The `BufferTarget` class captures output in a string. It has the same functions as the `Target` class: `write_char`, `write_string`, and `write_data`, in addition to the functions `str` and `length`.

`str`	Captures a string value.
	`const char* str();`

`length`	Captures a length constant.
	`unsigned int length() const;`

G SCE-MI 2.0 Pipes

Instead of ports, the SCE-MI 2.0 specification implements transaction pipes to connect the hardware side and the DUT. Transation pipes are unidirectional, meaning that in any given pipe, the transactions flow in only one direction. The data sent by the producer is guaranteed to be received by the consumer in the same order when the consumer asks for the data.

Transaction pipes that pass one-way transactions from the software side to the hardware side are called *input pipes*. Pipes that pass transactions from the hardware side to the software side are called *output pipes*. Both ends of the transaction pipe call into the pipe, with one end calling the *send* function and the other calling the *receive* function.

G.1 Hardware Side Pipes

Bluespec provides hardware-side BSV transactors instantiating the SCE-MI pipes. Each transactor provides an interface to the DUT. When the interface provided is a `Get` or `Put`, a `mkConnection` module is necessary to connect the DUT with the transactor. When the provided interface is `Empty`, the DUT can be directly connected to the transactor.

The transactor pipes correspond to the pipes defined on the software side, as described in Section G.2.

G.1.1 Visibility

The data type `Visibility` is a BSV adaption of the `scemi_input_pipe` HDL-side interface, described in the SCE-MI 2.0 specification.

```
typedef enum { Deferred
             , Immediate
             , Fifo
             } Visibility deriving (Eq);
```

G.1.2 mkInPipeXactor

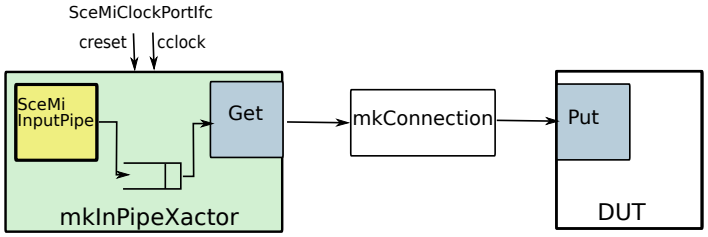

Figure 36: mkInPipeXactor

mkInPipeXactor	Hardware-side of the pipe for transactions passing data into the DUT. Allows a connection to a `Put` interface of a DUT by providing a `Get` interface. When used in a design a `mkConnection` module is required to connect the DUT `Put` to the InPipeXactor `Get`.
	`module [SceMiModule] mkInPipeXactor#(Integer depth, Visibility style, SceMiClockPortIfc clk_port) (Get#(ty)) provisos( Bits#(ty, ty_sz) ) ;`

G.1.3 mkPutPipeXactor

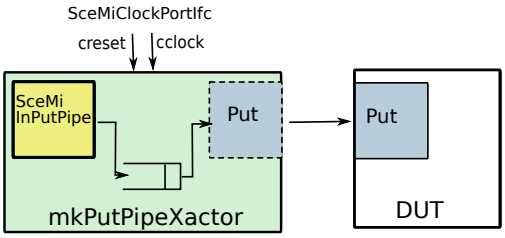

Figure 37: mkPutPipeXactor

mkPutPipeXactor	Hardware-side of a pipe that directly connects the pipe with the `Put` interface of a DUT, moving data into the hardware DUT. Takes the DUT's `Put` interface as a parameter.
	`module [SceMiModule] mkPutPipeXactor #(Put#(ty) putifc, Integer depth, Visibility style, SceMiClockPortIfc clk_port) (Empty) provisos ( Bits#(ty, ty_sz) ) ;`

G.1.4 mkOutPipeXactor

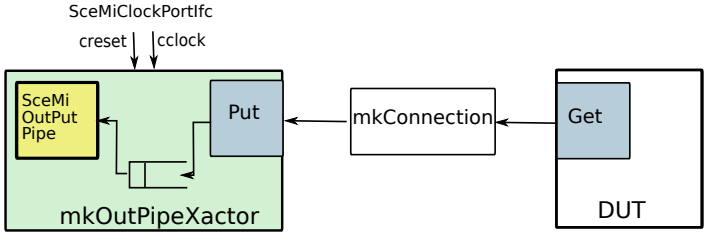

Figure 38: mkOutPipeXactor

mkOutPipeXactor	Hardware-side of the pipe for transactions passing data out from the DUT. Allows a connection to a `Get` interface of a DUT by providing a `Put` interface. When used in a design a `mkConnection` module is required to connect the DUT `Get` to the OutPipeXactor `Put`.
	`module [SceMiModule] mkOutPipeXactor#(Integer depth, Visibility style, SceMiClockPortIfc clk_port) (Put#(ty)) provisos ( Bits#(ty, ty_sz) );`

G.1.5 mkGetPipeXactor

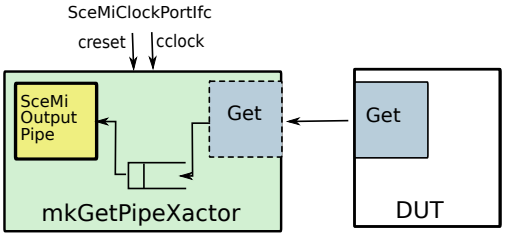

Figure 39: mkGetPipeXactor

mkGetPipeXactor	Hardware-side of a pipe that directly connects the pipe with the `Get` interface of a DUT, moving data from the hardware DUT to the output pipe. Takes the DUT's `Get` interface as a parameter.
	`module [SceMiModule] mkGetPipeXactor #(Get#(ty) getifc, Integer depth, Visibility style, SceMiClockPortIfc clk_port) (Empty) provisos ( Bits#(ty, ty_sz));`

G.1.6 mkServerPipeXactor

mkServerPipeXactor	
	The `mkServerPipeXactor` module connects a Server interface from the DUT to the SCE-MI pipe transactors.
	`module [SceMiModule] mkServerPipeXactor #(Server#(req_ty, resp_ty) server, Integer depth, Visibility style, SceMiClockPortIfc clk_port) (Empty) provisos ( Bits#(req_ty, req_ty_sz) , Bits#(resp_ty, resp_ty_sz) ) ;`

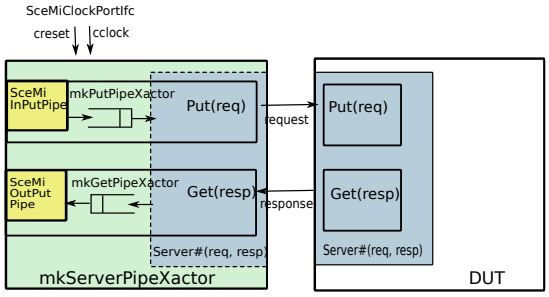

Figure 40: mkServerPipeXactor

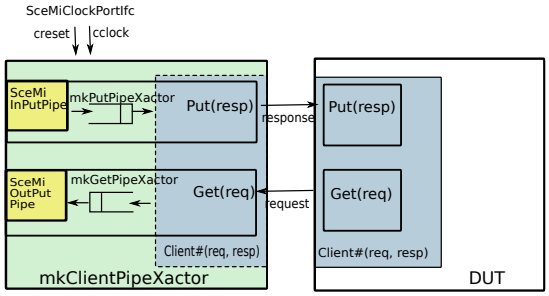

Figure 41: mkClientPipeXactor

G.1.7 mkClientPipeXactor

`mkClientXactor`	The `mkClientPipeXactor` module connects to a DUT with a Client interface. Takes DUT interface as a parameter.
	`module [SceMiModule] mkClientXactor#(Client#(req_ty, resp_ty) client, SceMiClockPortIfc clk_port ) (Empty) provisos (Bits#(req_ty, req_ty_sz), Bits#(resp_ty, resp_ty_sz));`

G.2 C++ Testbench Pipes

The provided transactors `InpipeXactorT` and `OutpipeXactorT` wrap the SCEMI C++ API functions that are used to send and receive data through the pipes. They combine methods used to initialize the pipes and control blocking. Bluespec also provides the full C++ API as defined in the SCE-MI 2.0 specification.

G.2.1 InpipeXactorT

The `InpipeXactorT` class is the C++ side of the BSV module `mkInPipeXactor`. It initializes the SceMiPort, including binidngs, and controls the software side of the pipe, sending messages from the hardware side.

InpipeXactorT	Constructor for the InpipeXactorT class.
	`InpipeXactorT(const std::string &name,` `  const std::string &path)`

send	This method blocks until the transactions is sent. The send method should not be called from a SCE-MI call back as deadlock will occur.
	`void send(const T &t)`

sendNB	This method does not block. It returns True if data is sent.
	`bool sendNB(const T &t)`

sendT	This method blocks, but with a timeout, returning True if data is sent. The method is overloaded; the compiler will determine the correct version of the method to use based on the arguments provided. The first version takes an absolute time, while the second takes a delta time.

	`bool sendT(const T &t, struct timespec *expiration)`
	
	`bool sendT(const T &t, const time_t &delta_seconds,`
	`  const long &delta_microseconds=0)`

G.2.2 OutpipeXactorT

The OutpipeXactorT class is the C++ side of the BSV module mkOutPipeXactor. It initializes the SceMiPort, including binidngs, and controls the software side of the pipe, receiving messages from the hardware side.

OutpipeXactorT	Constructor for the OutpipeXactorT class.
	`OutpipeXactorT (const std::string &name,` `  const std::string &path)`

receive	This method blocks if there is no transactin to receive. The receive method should not be called from a SCE-MI call back as deadlock will occur. Returns the message.
	`void receive(T &t)`

`receiveNB`	This method does not block if there is no transaction to receive. It returns `True` if a transaction is received.
	`bool receiveNB(T &t)`

`receiveT`	This method blocks if there is no transaction to receive, but there is a timeout. Returns `True` if a transaction is received. The method is overloaded; the compiler will determine the correct version of the method to use based on the arguments provided. The first version takes an absolute time, while the second takes a delta time.

	`bool receiveT(T &t, struct timespec *expiration)`
	
	`bool receiveT(T &t, const time_t &delta_seconds,`
	`  const long &delta_microseconds=0)`

Index