Sce-Mi Co-Emulation with Bluespec SystemVerilog

Revision: 3 March 2009

Copyright © 2000 – 2009 Bluespec, Inc.
# Contents

## Table of Contents

1 Introduction 3
   1.1 Purpose 3
   1.2 Requirements 3
   1.3 Methodology 3

2 Co-Emulation Support Library 4
   2.1 Types 4
   2.2 Message Ports 5
      2.2.1 SceMiMessageInPort 5
      2.2.2 SceMiMessageOutPort 6
   2.3 Clocking 6
      2.3.1 SceMiClockPort 6
      2.3.2 SceMiClockControl 8
   2.4 buildSceMi 9
   2.5 EVE Clock Structure Support 10
      2.5.1 mkSceMiExternalClockPort 10
      2.5.2 buildSceMiWithExternalClocks 11
      2.5.3 sceMiLookupClock 11
   2.6 Utilities 12
      2.6.1 Interfaces 12
      2.6.2 Modules 13

3 BSV Sce-Mi Testbench Library 14
   3.1 Message Port Proxies 14
      3.1.1 SceMiMessageInPortProxy 14
      3.1.2 SceMiMessageOutPortProxy 15

4 Compiling the Design 17
   4.1 Compiling for 3rd-Party Platforms using the SCEMI Linkage Type 17
   4.2 Compiling for Simulation using the TCP Linkage Type 17
   4.3 Compiling for Zebu or ZTide using the EVE Linkage Type 17
   4.4 Setting the Link Type 18
   4.5 Infrastructure Linkage 19
   4.6 BSV Link Stage 19
   4.7 Compiling a C++/SystemC Testbench 19
   4.8 Using the TCP Linkage Type 19
   4.9 Compiling a BSV Testbench 20
1 Introduction

This document 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).

1.1 Purpose

BSV support for co-emulation conforming to the Sce-Mi standard provides:

- Access to an industry-standard co-emulation API from BSV
- Seamless migration of a design across supported co-emulation hosts
- Ability to write transactors in BSV
- Ability to use a standard Sce-Mi untimed C/C++ testbench with a BSV design in simulation

1.2 Requirements

Support for Sce-Mi-style co-emulation in BSV requires a working Bluespec tool installation, and this guide assumes familiarity both with Bluespec SystemVerilog language and the Sce-Mi co-emulation methodology.

Please refer to the BSV Reference Guide, user guide, style guide, and tutorials for information on how to design and write specifications in the Bluespec SystemVerilog environment.

The Sce-Mi standard is available from Accellera’s website.

1.3 Methodology

A Sce-Mi-style co-emulation system comprises an untimed software testbench (the “SW” side) linked to an emulation platform (the “HW” side) through some communication channel. The emulation platform hosts a design-under-test (the “DUT”) and a number of transactors which handle communication to and from the DUT and control the clocking of the DUT. The transactors bridge the untimed transactions on the SW side with the cycle-accurate execution of the DUT, using a small number of message port and clock control primitives defined by the Sce-Mi standard.

Each message port in a transactor on the emulation host is paired with a port proxy in the testbench on the software side. The co-emulation infrastructure abstracts away the details of the communication channel linking the testbench to the emulation platform. Insertion of the communication channel logic into the design is performed by an infrastructure linkage tool (the “ILT”).

When using a Sce-Mi co-emulation system, the end-user is responsible for writing the untimed testbench, which could use either the standard C++ Sce-Mi API or the BSV Sce-Mi testbench library (section 3) to create and use port proxies. The end-user must also write the DUT hardware description. The transactors which control the DUT can either be written by the end-user or be pre-built standard transactors. The transactors use the standard Sce-Mi macros for message ports and clock generation.

The vendor supporting Sce-Mi on the emulation platform is responsible for providing the C++ library which implements the Sce-Mi testbench API, the implementation of the Sce-Mi macros used by the transactors, and the infrastructure linkage tool which connects the SW and HW side through a communication channel.

Bluespec’s Sce-Mi library supports modes of infrastructure linkage that use 3rd-party Sce-Mi platforms as well as BSV implementations for platforms without native Sce-Mi support.
2 Co-Emulation Support Library

The Bluespec release contains a BSV library that supports co-emulation with a Sce-Mi-like interface. To access the co-emulation support library, import the SceMi package:

import SceMi :: * ;

2.1 Types

SceMiModule  Modules in the SceMi package do not use the standard Module monad. Instead, Sce-Mi modules use the SceMiModule monad. Therefore, transactors which use these modules should also be in the SceMiModule monad, and cannot use the (* synthesize *) module attribute. To denote that a module uses the SceMiModule monad, use the extended module definition syntax:

module [SceMiModule] mkTransactor(..);

SceMiLinkType  Each module in the SceMi package can adapt its implementation to any one of several emulation host platforms. The desired implementation is selected through a module parameter of type SceMiLinkType. SceMiLinkType is an enumeration of all supported emulation host platforms.

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Emulation Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVE</td>
<td>ZeBu and zTIDE emulation systems from EVE</td>
</tr>
<tr>
<td>SCEMI</td>
<td>Any Sce-Mi-compliant emulation platform</td>
</tr>
<tr>
<td>TCP</td>
<td>Bluesim or Verilog simulation, via TCP socket</td>
</tr>
</tbody>
</table>

\(^1\)It is not necessary to understand monads to use the SceMi package. Simply remember to add the [SceMiModule] syntactic element to Sce-Mi module definitions.
typedef enum { TCP, SCEMI, EVE } SceMiLinkType deriving (Eq, Bits);

See section 4 for more detail on the supported emulation platforms.

2.2 Message Ports

The SceMi package provides message input and output ports parameterized by the type of the message payload. Each port 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.

2.2.1 SceMiMessageInPort

All message input port implementations expose the SceMiMessageInPortIfc#(msg_type) interface with the following methods:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_data</td>
<td>Bool</td>
<td>Returns True when data is available on an input port.</td>
</tr>
<tr>
<td>read</td>
<td>msg_type</td>
<td>Returns the data on an input port, if any.</td>
</tr>
<tr>
<td>ack</td>
<td>Action</td>
<td>Acknowledge data read.</td>
</tr>
</tbody>
</table>

interface SceMiMessageInPortIfc#(type msg_type);

(* always_ready *)
method Bool has_data();
method msg_type read();
(* always_ready *)
method Action ack();
endinterface: SceMiMessageInPortIfc

The has_data method on an input port will return True when there is data available on the port. The data can be accessed via the read method. The data on the port will persist until the ack method is called, at which point the port will present the next available data or become unready if there is no additional data available.

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

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

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

<table>
<thead>
<tr>
<th>mkSceMiMessageInPort</th>
<th>Instantiates an input port of the correct type based on the link_type module parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>module [SceMiModule] mkSceMiMessageInPort#</td>
<td></td>
</tr>
<tr>
<td>(parameter SceMiLinkType link_type)</td>
<td></td>
</tr>
<tr>
<td>(SceMiMessageInPortIfc#(a) ifc)</td>
<td></td>
</tr>
<tr>
<td>provisos(Bits#(a, _));</td>
<td></td>
</tr>
</tbody>
</table>
2.2.2 SceMiMessageOutPort

All message output port implementations expose the SceMiMessageOutPortIfc#(msg_type) interface with the following methods:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>accepting_data</td>
<td>Bool</td>
<td>Returns True when the output port is accepting data.</td>
<td></td>
</tr>
<tr>
<td>send</td>
<td>Action</td>
<td>Sends data on an output port.</td>
<td>msg</td>
</tr>
</tbody>
</table>

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 an out-port macro which conforms to the SCE-MI standard.

<table>
<thead>
<tr>
<th>mkSceMiMessageOutPort</th>
<th>Instantiates an output port of the correct type based on the link_type module parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>module [SceMiModule] mkSceMiMessageOutPort#</td>
<td>(parameter SceMiLinkType link_type)</td>
</tr>
<tr>
<td></td>
<td>(SceMiMessageOutPortIfc#(a) ifc)</td>
</tr>
<tr>
<td></td>
<td>provisos(Bits#(a, _));</td>
</tr>
</tbody>
</table>

2.3 Clocking

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

The SceMi package provides modules which generate and manipulate controlled clocks.

2.3.1 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.
interface SceMiClockPortIfc;
    interface Clock cclock;
    interface Reset creset;
endinterface: SceMiClockPortIfc

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

```
module [SceMiModule] mkSceMiClockPort#(
    parameter Integer clockNum,
    parameter Integer ratioNumerator,
    parameter Integer ratioDenominator,
    parameter Integer dutyHi,
    parameter Integer dutyLo,
    parameter Integer phase,
    parameter Integer resetCycles,
    parameter SceMiLinkType link_type)
(SceMiClockPortIfc ifc);
```

The `link_type` parameter specializes the clock port implementation for the target emulation platform, and it must match the `link_type` parameter supplied to other instantiations of the SceMi package modules.

The meaning of the other `mkSceMiClockPort` module parameters 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 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 $100 \times \frac{dutyLo}{dutyLo + dutyHi}$ percent of the time and high for $100 \times \frac{dutyHi}{dutyLo + dutyHi}$ 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 corresponding clock edge within the cycle is not important.

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.

2.3.2 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, methods which allow the transactor to stall the controlled clock, 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.

<table>
<thead>
<tr>
<th>SceMiClockControlIfc</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>uclock</td>
<td>interface</td>
<td>The uncontrolled Clock.</td>
</tr>
<tr>
<td>ureset</td>
<td>interface</td>
<td>The uncontrolled Reset.</td>
</tr>
<tr>
<td>allow_posedge</td>
<td>Action</td>
<td>Allow the next positive edge of the controlled clock.</td>
</tr>
<tr>
<td>pre_posedge</td>
<td>Bool</td>
<td>True if the controlled clock will rise on the next edge of the uncontrolled clock.</td>
</tr>
<tr>
<td>allow_negedge</td>
<td>Action</td>
<td>Allow the next negative edge of the controlled clock.</td>
</tr>
<tr>
<td>pre_negedge</td>
<td>Bool</td>
<td>True if the controlled clock will fall on the next edge of the uncontrolled clock.</td>
</tr>
</tbody>
</table>

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

All of the interface methods are in the uncontrolled clock domain (uclock). Calls to allow_posedge or allow_negedge have no effect during cycles in which the edge is not imminent.

The association of a clock controller and a clock port is achieved through the clockNum parameter of the mkSceMiClockControl module.

<table>
<thead>
<tr>
<th>mkSceMiClockControl</th>
<th>Instantiates a clock controller module.</th>
</tr>
</thead>
</table>
| module [SceMiModule] mkSceMiClockControl#(
    parameter Integer clockNum,
    parameter SceMiLinkType link_type)
(SceMiClockControlIfc ifc); |
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockNum</td>
<td>Integer</td>
<td>Unique clock identification number (Must match a mkClockPort instantiation)</td>
</tr>
<tr>
<td>link_type</td>
<td>SceMiLinkType</td>
<td>Selects emulation host platform type</td>
</tr>
</tbody>
</table>

The `link_type` parameter specializes the clock controller implementation for the target emulation platform, and it must match the `link_type` parameter supplied to other instantiations of the SceMi package modules.

### 2.4 buildSceMi

The message port and clock generation modules in the SceMi package all use the SceMiModule module monad type, which supports the additional infrastructure needed to coordinate the various clock ports, clock controllers and message ports. To return to the standard Module module monad, the design must contain a single instance of the buildSceMi module.

The `buildSceMi` module is a higher-order module with a polymorphic interface type. It takes a module in the SceMiModule monad as an argument, implements the necessary Sce-Mi coordination logic and returns the argument module’s interface in the regular Module monad.

```plaintext
buildSceMi

Takes a module in the SceMiModule monad, builds the SceMi infrastructure and returns the interface in the Module monad.

module [Module] buildSceMi#(SceMiModule#(i) mod,
                            SceMiLinkType link_type)
              (i ifc);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>SceMiModule#(i)</td>
<td>A module in the SceMiModule monad, with interface i, which is a parent of all other SceMi package instances in the design.</td>
</tr>
<tr>
<td>link_type</td>
<td>SceMiLinkType</td>
<td>Selects emulation host platform type</td>
</tr>
</tbody>
</table>
In a typical Sce-Mi system, `buildSceMi` is used at the top-level of the bridge code:

```
SceMiLinkType linktype = TCP;
...

(* synthesize *)
module mkBridge();

    // SceMi transactors that generate clocks and an interface
    // to connect to the DUT.
    TestbenchIfc scemi <- buildSceMi(scemiXactors, linktype);

    // Design-under-test
    DutIfc dut <- mkDut(clocked_by scemi.cclock, reset_by scemi.creset);

    // Connect SceMi transactors to DUT using Connectable instance
    mkConnection(scemi.to_dut, dut);
endmodule: mkBridge

module [SceMiModule] scemiXactors(TestbenchIfc);

    // instantiate transactors, clock generators, etc.
    ...
endmodule: scemiXactors
```

### 2.5 EVE Clock Structure Support

EVE’s ZeBu platform imposes restrictions on the locations of clock ports and clock controllers within the transactor hierarchy. Specifically, all of the clock ports must be instantiated at the top level, above the transactor hierarchy while the clock controllers must be instantiated within the transactor hierarchy of the bridge code.

To conform to these restrictions the BSV Sce-Mi library supports definition of clocks outside of the `SceMiModule` monad, for the EVE linkage type only.

#### 2.5.1 `mkSceMiExternalClockPort`

Clocks can be defined in the standard `Module` monad using the `mkSceMiExternalClockPort` module. This module has exactly the same `SceMiClockPortIfc` interface as the `mkSceMiClockPort` module but operates in the `Module` monad instead of the `SceMiModule` monad, allowing it to be instantiated in the top level above the transactor hierarchy as required by the ZeBu tool chain.
<table>
<thead>
<tr>
<th>mkSceMiExternalClockPort</th>
<th>Creates a clock port as defined by the parameters. This is for use outside of the SceMiModule monad.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>module [Module] mkSceMiExternalClockPort#(</td>
</tr>
<tr>
<td></td>
<td>parameter Integer clockNum</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer ratioNumerator</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer ratioDenominator</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer dutyHi</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer dutyLo</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer phase</td>
</tr>
<tr>
<td></td>
<td>, parameter Integer resetCycles</td>
</tr>
<tr>
<td></td>
<td>, parameter SceMiLinkType link_type)</td>
</tr>
<tr>
<td></td>
<td>(SceMiClockPortIfc ifc);</td>
</tr>
</tbody>
</table>

### 2.5.2 buildSceMiWithExternalClocks

When controlled clocks and resets are defined outside of the SceMiModule monad, the Scm-Mi hierarchy should be constructed using the `buildSceMiWithExternalClocks` module in place of the usual `buildSceMi` module. `buildSceMiWithExternalClocks` takes a Vector of Clocks and a Vector of Resets as additional parameters. Within the SceMiModule monad, the first Clock and Reset of the vector become controlled clock and reset number 0, the next become controlled clock and reset number 1, and so on.

<table>
<thead>
<tr>
<th>buildSceMiWithExternalClocks</th>
<th>Takes a module in the SceMiModule monad, a Vector of Clocks and a Vector of Resets, builds the SceMi infrastructure and returns the interface in the Module monad.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>module [Module] buildSceMiWithExternalClocks#(</td>
</tr>
<tr>
<td></td>
<td>SceMiModule#(i) mod</td>
</tr>
<tr>
<td></td>
<td>, Vector#(n,Clock) clks_in</td>
</tr>
<tr>
<td></td>
<td>, Vector#(n,Reset) rsts_in</td>
</tr>
<tr>
<td></td>
<td>, SceMiLinkType link_type)</td>
</tr>
<tr>
<td></td>
<td>(i ifc);</td>
</tr>
</tbody>
</table>

### 2.5.3 sceMiLookupClock

Within the SceMiModule monad, the `sceMiLookupClock` module can be used to find the clock and reset associated with a clock number, if any are defined. This is useful when writing transactors which need to support clocks which may optionally be defined externally.

<table>
<thead>
<tr>
<th>sceMiLookupClock</th>
<th>Returns the clock domain information associated with a particular clock domain number, if it is defined.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>module [SceMiModule] sceMiLookupClock#(Integer clockNum)</td>
</tr>
<tr>
<td></td>
<td>(Maybe#(DomainInfo));</td>
</tr>
</tbody>
</table>

The module takes a single `clockNum` argument and returns a value of type `Maybe#(DomainInfo)`. If no domain with the requested number is currently defined, the returned value will be tagged `Invalid`. If the requested domain is defined, a `Valid` value will be returned, containing the Clock and Reset of the domain as described by the structure:
typedef struct {
    Clock clk;
    Reset rst;
    Bool external;
} DomainInfo;

A typical use of this module is testing if a clock is defined externally, and instantiating the clock internally if it is not:

// If the clock was not defined externally, instantiate it here
Clock dut_clk;
Reset dut_rst_n;
Maybe#(DomainInfo) mdom <- sceMiLookupClock(dut_clock_num);
case (mdom) matches
    tagged Valid .dom:
        begin
            dut_clk = dom.clk;
            dut_rst_n = dom.rst;
        end
    tagged Invalid:
        begin
            let clk_port <- mkSceMiClockPort(...);
            dut_clk = clk_port.cclock;
            dut_rst_n = clk_port.creset;
        end
endcase

2.6 Utilities

The BSV Sce-Mi library includes a package of utility modules, which can be accessed by importing the SceMiUtils package.

import SceMiUtils :: * ;

2.6.1 Interfaces

ServerXactorIfc

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclock</td>
<td>Interface</td>
<td>Clock interface....</td>
<td>creset</td>
<td>Reset interface....</td>
</tr>
<tr>
<td>client</td>
<td>Interface</td>
<td>Client#(req,resp)</td>
<td></td>
<td>interface</td>
</tr>
</tbody>
</table>

interface ServerXactorIfc#(type req, type resp);
    interface Clock cclock;
    interface Reset creset;
    interface Client#(req,resp) client;
endinterface: ServerXactorIfc
2.6.2 Modules

<table>
<thead>
<tr>
<th>mkServerXactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>module [SceMiModule] mkServerXactor#(parameter Integer clockNum ,parameter SceMiLinkType lt) (ServerXactorIfc#(req,resp) ifc) provisos(Bits#(req,req_sz), Bits#(resp,resp_sz));</td>
</tr>
</tbody>
</table>
3 BSV Sce-Mi Testbench Library

To connect to the Sce-Mi emulation platform, end-user must supply a testbench module containing the Sce-Mi port proxies that communicate with the corresponding Sce-Mi message ports on the HW side. In addition to the previously mentioned standard C/C++ untimed SceMi testbench API, Bluespec also 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 providing the option for targeting an execution model that can reside together with the rest of the system on the same emulation platform.

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

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

3.1 Message Port Proxies

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.

3.1.1 SceMiMessageInPortProxy

All message input port proxy implementations expose the `SceMiMessageInPortProxyIfc#(msg_type)` interface with the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Argument</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accepting_data</td>
<td>Bool</td>
<td>accepting_data</td>
<td>Returns True when the corresponding SceMiMessageInPort is ready to accept data.</td>
</tr>
<tr>
<td>send</td>
<td>Action</td>
<td>send</td>
<td>Sends data on an from input port proxy to the corresponding SceMiMessageInPort.</td>
</tr>
</tbody>
</table>

```plaintext
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:

```plaintext
instance ToPut#(SceMiMessageInPortProxyIfc#(a), a);
```
The module `mkSceMiMessageInPortProxy` instantiates an in-port proxy macro which conforms to the SCE-MI standard.

```
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 infrastructure linkage tool (section 4.5).

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.

### 3.1.2 SceMiMessageOutPortProxy

All message output port proxy implementations expose the `SceMiMessageOutPortProxyIfc#(msg_type)` interface with the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_data</td>
<td>Bool</td>
<td>Returns True when data from the corresponding SceMiMessageOutPort is available on an output port proxy.</td>
</tr>
<tr>
<td>read</td>
<td>msg_type</td>
<td>Returns the data from the corresponding SceMiMessageOutPort on an output port proxy, if any.</td>
</tr>
<tr>
<td>cycle_stamp</td>
<td>SceMiCycleStamp</td>
<td>Returns the cycle stamp integer id of the most recent call to read method.</td>
</tr>
<tr>
<td>shutdown</td>
<td>Action</td>
<td>Shutdown the SceMi infrastructure.</td>
</tr>
</tbody>
</table>

```
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 shuts down 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:

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

The module `mkSceMiMessageOutPortProxy` instantiates an out-port proxy macro which conforms to the SCE-MI standard.

```verilog
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 4.5).

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.
4 Compiling the Design

When using the SceMi package, the choice of linkage type determines which compilation steps are necessary as well as which emulation/simulation options are supported.

<table>
<thead>
<tr>
<th>Link Type</th>
<th>HW</th>
<th>Verilog Sim</th>
<th>Bluesim</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCEMI</td>
<td>✔</td>
<td>?</td>
<td></td>
<td>Simulation requires 3rd party tool support</td>
</tr>
<tr>
<td>EVE</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>Verilog sim through EVE’s zTide product</td>
</tr>
<tr>
<td>TCP</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

4.1 Compiling for 3rd-Party Platforms using the SCEMI Linkage Type

When using the SCEMI linkage type, the bsc compiler is used only to generate Verilog output which contains the required Sce-Mi macro 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 uses the 3rd-party infrastructure linkage and integration tools. Documentation for those tools will be provided by the respective vendors.

The tool flow used with the SCEMI linkage type is represented schematically in Figure 2.

4.2 Compiling for Simulation using the TCP Linkage Type

When using the TCP linkage type, bsc can be used either to compile for simulation with Bluesim, or to generate Verilog for use with a 3rd-party Verilog simulator. When using the -verilog option you must use the -elab flag to ensure that .ba files are generated during the compilation.

In between the first and second bsc compilation stages, Blue-spec’s infrastructure linkage tool must be run, as described in section 4.5.

The final bsc compilation stage is run as normal, with the addition of the -scemi command-line argument. This instructs the bsc link stage to integrate the output of the infrastructure linkage tool into the generated simulation model.

Figure 3 illustrates the Bluesim and Verilog simulator compilation flows.

4.3 Compiling for Zebu or ZTide using the EVE Linkage Type

When using the EVE linkage type, the bsc compiler is used to generate Verilog output which contains the required EVE macro instantiations. The initial compilation step is no different than for any other design, but you must use the -elab flag to ensure that .ba files are generated during the compilation.

Figure 2: SCEMI compilation flow.
Once the Verilog and .ba files are generated from bsc, the infrastructure linkage tool is used to generate a parameters file to be loaded by the Sce-Mi testbench. The Verilog generated from bsc is simulated and/or synthesized using the standard EVE flow, which is described in the documentation accompanying EVE’s tools.

Figure 4 illustrates the Zebu/ZTide compilation flow.

### 4.4 Setting the Link Type

All instances of modules from the SceMi package have a SceMiLinkType argument or parameter, and this must be the same value for all instances in the design.

One simple way to achieve this is to use a definition like:

```plaintext
SceMiLinkType linktype = EVE;
```

in a package which can be imported wherever it is needed, and then to use the identifier “linktype” in each Sce-Mi module instantiation.

An alternative mechanism is to use a preprocessor definition such as “`LINKTYPE`” and then pass the value to the initial compilation stage using the `-D` compiler flag:

```plaintext
bsc ... -D LINKTYPE=EVE ...
```

Figure 3: Compilation flows using TCP linkage.

Figure 4: EVE compilation flow.
4.5 Infrastructure Linkage

For the SCEMI linkage type, the infrastructure linkage tool supplied by the emulation platform vendor should be used. For other linkage types, Bluespec provides an infrastructure linkage tool called “scemilink”.

The scemilink tool takes as input the .ba files produced from the initial bsc elaboration step (either bsc -sim ... or bsc -verilog -elab ... ). The scemilink tool will analyze the design from the .ba files and produce a Sce-Mi parameters file which can be loaded by the testbench. For the TCP linkage type, when given a --sim option, scemilink will produce scemilink.c and scemilink.h files to be integrated into the Bluesim executable. For the TCP linkage type, the --verilog option causes scemilink to generate a scemilink.vlog_fragment file to be integrated into the Verilog simulation model.

Scemilink requires the name of the top module in the design (eg. mkBridge) as its command-line argument. Additional options control the parameter file name, search and file output paths, etc. The scemilink -h command will describe all available options.

4.6 BSV Link Stage

When using the TCP linkage type, the scemilink tool will produce additional files which need to be integrated into the simulation model during the final bsc execution. This is accomplished by using the -scemi option on the bsc command-line.

4.7 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/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:

<table>
<thead>
<tr>
<th>C++ Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-I${BLUESPECDIR}/SceMi</td>
<td>Path to include scemi.h file</td>
</tr>
<tr>
<td>-L${BLUESPECDIR}/SceMi/&lt;C++family&gt;</td>
<td>Path to find libscemi.a for &lt;C++_family&gt;</td>
</tr>
<tr>
<td>-lscemi</td>
<td>Link libscemi into executable</td>
</tr>
</tbody>
</table>

4.8 Using the TCP Linkage Type

When using the TCP linkage type, 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 should be started before the testbench, so that the simulator is ready to accept the connection when the testbench starts.
4.9 Compiling a BSV Testbench

Compiling a testbench module written using the BSV testbench library is similar to compiling a regular BSV module. Except during the link stage of the BSV testbench, the extra routines needed to communicate with the SceMi infrastructure must be provided for linkage to produce the final model. To accomplish this, the "-scemiTB" flag must be added to the command statement of the bsc final link step.

Figure 5 illustrates the testbench compilation flows for both bluesim and verilog methodology.

5 Clock Waveforms

The Sce-Mi standard specifies how controlled clock waveforms are generated using the parameters to the clock ports and inputs to the clock controller instances, but emulation platforms may differ in how these controlled clock waveforms relate to the uncontrolled clock.

When using the SCEMI or EVE linkage types, the waveform generation is controlled by the respective emulation platforms.

When using the TCP linkage type, the clock waveforms are generated within the BSV library using a model of flexible controlled time. This clock generation strategy assigns as much controlled time as possible to each uncontrolled clock cycle, so that there are no wasted uncontrolled clock cycles in which no controlled clock edges occur.

Consider two controlled clocks, one with the ratio 1 : 1 and one with the ratio 3 : 2. If neither uses a “don’t care” edge, then the rising and falling edges of both controlled clocks must align with a rising edge of the uncontrolled clock. The resulting clock waveforms using uniform controlled time might look like Figure 6.
Using uniform controlled time there are several rising edges of the uncontrolled clock on which no controlled clock edge is generated. These edges are marked “wasted” in Figure 6, since they represent cycles in which the DUT is not performing useful work.

In a co-emulation environment the controlled clocks are not synchronized to any external reference, so it is possible to assign different amounts of controlled time to different uncontrolled clock cycles. The clock generation logic generated in BSV when the TCP linkage type is used does exactly this, resulting in a circuit which never wastes cycles. It will generate clock waveforms like those in Figure 7.
Index

BSV Testbench, 14
testbench, 19, 20
buildSceMi, 9
uncontrolled clock, 8
buildSceMiWithExternalClocks, 11
uncontrolled reset, 8
clock, 6, 20
waveforms, 20
clock control, 8
clock port, 6
clockNum, 7–9, 11
compilation, 17
controlled clock, 6
controlled reset, 6
dutyHi, 7

dutyLo, 7
execution, 19
infrastructure linkage tool, 19

link_type, 5–7, 9, 14, 16
message port proxies, 14
message ports, 5
mkSceMiClockControl, 8
mkSceMiClockPort, 6
mkSceMiExternalClockPort, 10
mkSceMiMessageInPort, 5
mkSceMiMessageInPortProxy, 14
mkSceMiMessageOutPort, 6
mkSceMiMessageOutPortProxy, 15
mkServerXactor, 13

phase, 7
ratioDenominator, 7
ratioNumerator, 7
resetCycles, 7

SceMiClockControlIfc, 8
SceMiClockPortIfc, 6
scemilink, 19
SceMiLinkType, 4, 18
EVE, 17
SCEMI, 17
TCP, 17
sceMiLookupClock, 11
SceMiMessageInPortIfc, 5
SceMiMessageInPortProxyIfc, 14
SceMiMessageOutPortIfc, 6
SceMiMessageOutPortProxyIfc, 15
SceMiModule, 4
serverXactorIfc, 12