Bluespec Compiler (BSC)
User Guide

Revision: 3 February 2022

Trademarks and copyrights

Contents

1 Getting Started

1.1 Introduction

This document explains the mechanics and logistics of compiling, simulating, and analyzing a Bluespec SystemVerilog (BSV) specification with the Bluespec Compiler (BSC) and Bluetcl, a collection of Tcl extensions, scripts, and packages providing BSC-specific features to Tcl. A Bluetcl reference is provided in Appendix B.

For information on how to design and write specifications in the Bluespec SystemVerilog environment, please refer to the *Bluespec SystemVerilog Reference Guide*, *BSV by Example* guide, and other tutorials provided by Bluespec Inc or the B-Lang organization.

BSC is free and open-source software (FOSS) administered by the B-Lang organization and available on GitHub at: https://github.com/B-Lang-org/bsc

1.2 Installing BSC

1.2.1 Download the software

TBD

1.2.2 System Requirements

TBD

1.2.3 Install the software

TBD

1.3 Components of BSC Release

BSC is released with the following components:

- BSC compiler: The compiler takes BSV syntax and generates a hardware description, for either Verilog or Bluesim.
- BSC library packages: BSC is shipped with a set of libraries which provide common and useful programming idioms and hardware structures.
- Verilog library modules: Several primitive BSV elements, such as FIFOs and registers, are expressed as Verilog primitives.
- Bluesim: a cycle simulator for BSV designs.
- Bluetcl: a collection of Tcl extensions, scripts, and packages providing BSC-specific features to Tcl.

Also included is documentation, such as this User Guide.

- User Guide: This manual which explains how to run BSC, what flags are available, and how to read the tool output.
- Libraries Reference Guide: A reference that fully describes the libraries that are released with BSC.

1.4 Utilities

BSV editing modes are provided for the editors `emacs`, `vim`, and `jedit`. The files are in subdirectories in the `$BLUESPEC_HOME/util` directory. Each directory contains a `README` file with installation instructions for the editor.

The `$BLUESPEC_HOME/util` directory also contains an GNU enscript `.st` file for printing Bluespec SystemVerilog language files. A `README` file in the directory contains instructions for installation and use.

1.5 Quick Start

From the command line, you can invoke the Bluespec compiler with:

```
bsc  arguments
```

2 Designing with Bluespec

2.1 Components of a BSV Design

A BSV program consists of one or more outermost constructs called packages. All BSV code is assumed to be inside a package. Furthermore, BSC and other tools assume that there is one package per file, and they use the package name to derive the file name. For example, a package called `Foo` is assumed to be located in the file `Foo.bsv`.

The design may also include Verilog modules, VHDL modules, and C functions. Additional files will be generated by BSC as a result of the compile, link, and simulation steps. Some files are only generated for a particular back end (Bluesim or Verilog), others are used by both back ends. The following table lists the different file types and their roles.

File Types in a BSV Design

File Type	Description	Bluesim	Verilog
`.bsv`	BSV source File	√	√
The `.bo` file is an intermediate file not viewed by the user			
`.bo`	Binary file containing code for the package in an intermediate form	√	√
`.ba`	Elaborated module file	√	√
`.v`	Generated Verilog file		√
`.h`	`C++` header files	√	
`.cxx`	Generated `C++` source file	√	
`.o`	Compiled object files	√	
`.so`	Compiled shared object files	√	

2.2 Overview of the BSC process

This section provides a brief overview of the stages of designing with BSC. Later sections contain more detailed explanations of the compilation and linking processes. Refer to Section 3 for a complete listing of the flags available for guiding the compiler.

Designing with BSC has three distinct stages. You can use the BSC command throughout each stage of the process. Figure 1 illustrates the following steps in building a BSV design:

1. A designer writes a BSV program, including Verilog, VHDL, and C components as desired.
2. The BSV program is compiled into a Verilog or Bluesim specification. This step is comprised of two distinct stages:
   (a) pre-elaboration - parsing and type checking
   (b) post-elaboration - code generation
3. The compilation output is either linked into a simulation environment or processed by a synthesis tool.

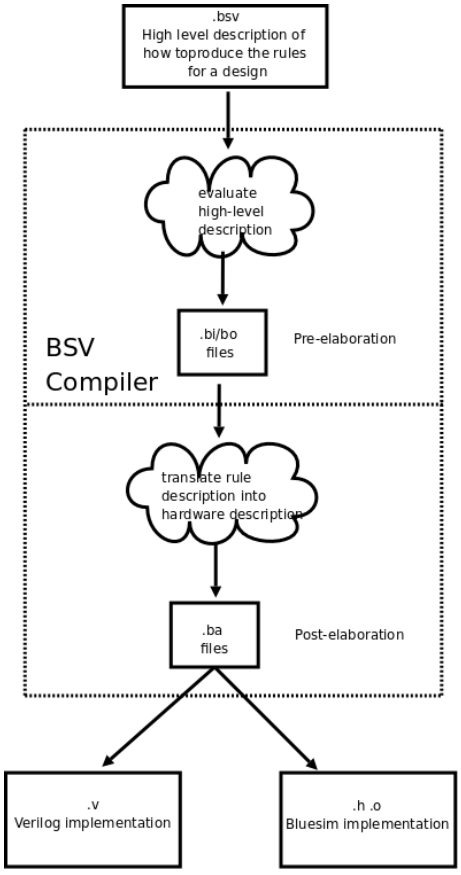

Figure 1: BSV compilation stages

We use the *compilation* stage to refer to the two steps, type checking and code generation, as shown inside the dotted box in Figure 1. As the figure shows, the code generation specification that is output by BSC is subject to a second run of the compiler to link it into a simulation or synthesis environment. We refer to this as the *linking* stage, even though the same compiler is used to perform the linking. BSC is required for linking Bluesim generated modules. For Verilog, the

generated modules can be handled as you would any other Verilog modules; they can be linked with the Bluespec compiler or you can choose to use the generated Verilog files manually instead.

You can perform the above actions — compile, link, and simulate — directly from a Unix command line.

Once you've generated the Verilog or Bluesim implementation, the design can be analyzed using Bluetcl commands. BSC can also dump information during compilation and linking, via flags.

2.3 Help

For a summary of how to run BSC and a listing of all flags, type:

```
bsc -help
```

from a Unix command prompt. Compiler flags are documented here in Section 3.

2.4 Compile

There are two stages in compilation, type checking and code generation, executed by a single compile command. The simplest compilation of a BSV design is to run only the first stage of the compiler, generating the `.bo` files. Once the type check is complete, you have enough module information to use Bluetcl to browse the contents of packages.

The second stage (code generation) generates an elaborated module file (`.ba`) and, when the target is Verilog, a (`.v`) file. These generated files have the same name as the module they implement, not the name of the package or file they come from.

To run the compiler through to code generation, a module must be marked for synthesis. The recommended method is to use the `synthesize` attribute in the BSV code. You can also specify a top module with the `-g` compiler flag. See Section 2.4.3 and the BSV Reference Guide for information on synthesizable modules.

A package often imports other packages. BSC can automatically recompile imported packages, if necessary, if the `-u` option is specified on the command line. Section 2.4.2 discusses importing packages. Section 2.4.4 contains a detailed explanation of techniques and considerations when compiling a collection of BSV modules.

The compiler automatically runs through to code generation if no errors are encountered in the type checking stage and a module is marked for synthesis. If errors are encountered in the type check stage, the compiler will halt before generating the `.bo` file. In this case, Bluetcl cannot be used to browse the packages, as it depends on these intermediate files. If errors are encountered in the second stage, the `.bo` file will be created but the `.ba` file will not. In this case, Bluetcl cannot be used to browse the modules of a design or the scheduling of rules and methods in a design. Library packages provided with BSC can always be browsed with Bluetcl, since the `.bo` files for these packages are always available.

To view scheduling graphs, the compiler flag `-sched-dot`, described in Section 3.13, must be specified for compilation. This flag generates the `.dot` files from which graphs can be generated.

2.4.1 Compiling a File

A BSV source file is compiled from the command line with a command like this:

```
bsc [flags] Foo.bsv
```

where one or more compiler flags may be specified after the `bsc` command. Section 3.1 describes compiling from the command line in more detail.

If the file imports packages defined in other sources, you will need to compile those other source files first. BSC can be directed to automatically compile any imported files, if necessary, before compiling the selected file, by providing the `-u` flag on the command line. The compiler compares the time stamp on the `.bo` file to determine if the imported file has changed since the last compilation. When you compile with `-u`, only changed files will be recompiled.

2.4.2 Importing other packages

To compile a package that imports another package, BSC needs the `.bo` file from the imported package. One way to provide this file is to run the compiler on each imported file. Or direct BSC, via the `-u` flag, to automatically determine which files are needed and recompile as necessary. If the `.bo` file already exists, the compiler will only recompile if the file has changed since the last compilation, as indicated by the imported file having a more recent date than the file being compiled.

For example, to compile a package `Foo` that imports another package `Baz`, BSC needs to examine the file `Baz.bo`. If `Baz` is in the file `Baz.bsv`, then this file needs to be run through the compiler to produce the necessary `.bo` file before the compiler can be invoked on `Foo.bsv`. If you use the `-u` flag on the command line, the compiler will check to see if `Baz.bo` exists, and if it exists, it will check the compilation date. The compiler will recompile the `Baz` file if necessary.

BSC is shipped with a set of library files providing common and useful hardware structures, such as `FIFO` and `UInt`. They are described in the BSC Library Reference Guide. The source code for these packages is already compiled and the `.bo` files are found in a library directory with the compiler installation (in the same way that C header and object files are stored in standard *include* and *library* directories). The compiler looks for these files in:

```
%/Libraries/
```

The BSC `Libraries` directory is automatically added to the search path by BSC, unless overridden via flags.

If you are importing packages from other directories, the directories must be added to the search path. The flags which modify the path are described in Section 3.6.

2.4.3 Specifying modules for code generation

A module can be selected for code generation either in the BSV code or at compile-time. The recommended method is to mark the module for code generation in the BSV code, using the `synthesize` attribute (see the BSV Reference Guide for more information on attributes). The alternative is, at compile-time, to use the `-g` flag (Section 3.1), which instructs the compiler to generate code for a particular module. The `-g` flag can be used multiple times within a compile command line to specify multiple modules for code generation.

Whether the generated code will be Bluesim or Verilog depends on which back end has been selected, by using the `-verilog` or `-sim` command line flag.

Not all modules written in BSV are synthesizable. To be synthesized the module must be of type `Module` and not of any other module type that can be defined with `ModuleCollect`. A module is synthesizable if its interface is a type whose methods and subinterfaces are all convertible to wires.

A method is convertible to wires if it meets the following conditions:

- its argument types are convertible to wires which means either

  - it is in the Bits class OR
  - it is a function whose argument and return type are convertible to wires

- its return type is `Action` OR
- its return type is a `value` or `ActionValue` where either
  - the value is convertible to bits (i.e. in the `Bits` class) OR
  - the field is an exported clock or reset.

A module to be synthesized is allowed to have non-interface inputs, such as clocks and resets. Parameters to the module are allowed if they are convertible to bits.

Clock and Reset subinterfaces are convertible to wires.

If none of the modules are marked for synthesis, the compiler will not generate a hardware description (a Verilog `.v` file or a Bluesim `.ba` file).

2.4.4 Understanding separate compilation

BSC has two main stages; first it converts BSV modules into a collection of states and rules, and then it converts the rule-representation into a hardware description.

When compiling a collection of BSV modules, it is up to the user to decide which of these modules should be compiled to hardware separately, and which should be subsumed into the parent module. By default, all hierarchy is flattened into one top-level module in the final hardware description, but the user can specify modules which should stay in the hierarchy and have separate hardware descriptions.

What happens when a module `m1` instantiates another module `m2`? If the submodule `m2` is provided as a BSV description, that description will need to be compiled into a set of rules and then those rules combined with the rules for `m1` to be converted, by the code generation stage, into a hardware description.

If `m2` is provided as a hardware description (that is, implemented in a Verilog file or in Bluesim header and object files), then the hardware description for `m1` will contain an instantiation of `m2`. The implementation of `m2` is kept in its own file. For the Verilog back end, this produces a `m1.v` file with a Verilog module `m1` which instantiates `m2` by name and connects to its ports but doesn't contain the implementation of `m2`. Both implementation files, `m1.v` and `m2.v`, must be provided to the simulation or synthesis tools.

Even if `m2` is provided as a BSV description, the user can decide to generate a separate hardware description for the module. This is done by putting the `synthesize` attribute in the BSV description or using the `-g` flag, indicating that the module should be synthesized as a separate module, apart from the instantiating module.

The implementation in a `.bo` reflects whether hardware was generated for a module. If a hardware description was generated for a module, then the implementation in the `.bo` will be merely a pointer to the location of that description (be it `.v` or `.o`). If hardware was not generated for the module, then an entirely BSV representation will be stored in the `.bo` file.

Thus, a single `.bsv` file can be compiled in different ways to produce very different `.bo` files. When the compiler is generating hardware for another BSV file that imports this package, it will need to read in the information in the `.bo` file. How it is compiled depends on the flags used. Therefore, compiling the new file will be affected by how the imported file was compiled earlier! It is important, therefore, to remove these automatically generated files before beginning a new compilation project, especially if a different module hierarchy is desired.

For example, if a user were to generate Verilog for a module `mkFoo` just for testing purposes, the `Foo.bo` would encapsulate into its own description the information that a Verilog module had been generated for module `mkFoo`. If the user then wanted to generate Verilog for a larger design, which included this module, but wanted the larger design to be compiled into one, hierarchy-free Verilog module, then the `.bo` file would have to be deleted so that a new version could be created that only contained the state-and-rules description of the module.

2.4.5 Interfacing to foreign modules and functions

Foreign modules and functions can be included as part of a BSV model. A designer can specify that the implementation of a particular BSV module is provided as either a Verilog module or a C function.

Importing Verilog modules

Using the `import "BVI"` syntax, a designer can specify that the implementation of a particular BSV module is an RTL (Verilog or VHDL) module, as described in the BSV Reference Guide. The module is treated exactly as if it were originally written in BSV and then converted to hardware by the compiler, but instead of the `.v` file being generated by the compiler, it was supplied independently of any BSV code. It may have been written by hand or supplied by a vendor as an IP, etc. The files for these modules need to be linked in to the simulation. This process is described in Section 2.5.1 for Bluesim simulations and 2.5.3 for Verilog simulations.

Several primitive BSV elements, such as FIFOs and register files, are expressed this way — as Verilog primitives. When simulating or synthesizing a design generated with the Verilog back end, you will need to include the appropriate hardware descriptions for these primitives. Verilog descriptions for BSC's primitive elements can be found in:

```
${BLUESPECDIR}/Verilog/
```

Note: We attempt to be sure that the Bluesim and Verilog models simulate identically. Simulations using 4-state (X and Z) logic, user supplied Verilog, or other unsupported or nonstandard parts are never guaranteed to match.

Importing C functions

Using the importBDPI syntax, the user can specify that the implementation of a BSV function is provided as a C function. The same implementation can be used when simulating with Bluesim or with Verilog. In Bluesim, the imported functions are called directly. In Verilog, the functions are accessed via the Verilog VPI or the SystemVerilog DPI . The compilation and linking procedures for these backends are described in Sections 2.5.1 for Bluesim simulations, and 2.5.3 for Verilog simulations.

2.5 Link

The compiled hardware description must be linked into a simulation environment before you can simulate the project. The result of the linking stage is a binary which, when executed, simulates a module. BSC is required for linking Bluesim generated modules and can be used to link Verilog modules as well. To link from the command line, use the `bsc` command along with the appropriate flags, as described in Section 3.1.

2.5.1 Linking with Bluesim

For the Bluesim back end, linking means incorporating a set of Bluesim object files that implement BSV modules into a Bluesim simulation environment. See Section 6 for a description of this environment. Bluesim is specified by using the `-sim` flag. In an installation of BSC, the files for this simulation environment are stored with the other Bluesim files at: `${BLUESPECDIR}/Bluesim/`.

Specifically, the linking stage generates a C++ object for each elaborated module. For each module, it generates *module*`.h` and *module*`.cxx` files which are compiled to a `.o` file. The C++ compiler to use is determined from the `CXX` environment variable (the default is `c++`) and any flags specified in `CXXFLAGS` or `BSC_CXXFLAGS` are added to the command line. Also generated are the files `model_`*topmodule*`.h` and `model_`*topmodule*`.cxx` which are the top level that combines the individual modules into a single model, implementing a global schedule computed by combining the schedules from all the individual modules in the design. Once compiled to `.o` files, these objects are linked with the Bluesim library files to produce an `.so` shared object file. This shared object file can be dynamically loaded into Bluetcl using the `sim load` command. For convenience, a wrapper script is generated along with the `.so` file which automates loading and execution of the simulation model.

If you want to see all the `CAN_FIRE` and `WILL_FIRE` signals, you must specify the `-keep-fires` flag (described in Section 3.12) when compiling and linking with Bluesim.

The typical command to link BSV files to generate Bluesim executables is:

```
bsc -sim -e -keep-fires mkFoo
```

Imported Verilog modules in Bluesim

Using the `import "BVI"` syntax, a designer can specify that the implementation of a particular BSV module is a Verilog module. The module is treated exactly as if it were originally written in BSV, but was converted to hardware by the compiler.

Bluesim does not currently support importing Verilog modules directly. If a Bluesim back end is used to generate code for this system, then a Bluesim model of the Verilog module needs to be supplied in place of the Verilog description. Such a model would need to be compiled from a BSV description and used conditionally, depending on the backend. The environment functions `genC` and `genVerilog` (as defined in the BSV Reference Guide) can be used to determine when to compile this code.

For example, you might have a design, `mkDUT`, which instantiates a submodule `mkSubMod`, which is a pre-existing Verilog file that you want to use when generating Verilog:

```
module mkDUT (...);
   ...
   SubIFC submod <- mkSubMod;
   ...
endmodule
```

You would write an `import "BVI"` statement:

```
  import "BVI" module mkSubMod (SubIFC); ... endmodule
```

But this won't work for a Bluesim simulation - Bluesim expects a `.ba` file for `mkSubMod`.

The way to write one BSV file for both Verilog and Bluesim is to change `mkSubMod` to be a wrapper, which conditionally uses a Verilog import or a BSV-written implementation, depending on the backend:

```
module mkSubMod (SubIFC);
   SubIFC _i <- if (genVerilog)
                   mkSubMod_verilog
                else
                   mkSubMod_bluesim;
   return _i;
endmodule

// note that the import has a different name
import "BVI" mkSubMod =
    module mkSubMod_verilog (SubIFC); ... endmodule

// an implementation of mkSubMod in BSV
module mkSubMod_bluesim (SubIfc);
   ...
endmodule
```

This code will import Verilog when compiled to Verilog and it will use the native BSV implementation otherwise (when compiling to Bluesim).

Imported C functions in Bluesim

Using the importBDPI syntax, the user can specify that the implementation of a BSV functions is provided as a C function. When compiling a BSV file containing an import-BDPI statement, an elaboration file (`.ba`) is generated for the import, containing information about the imported function. When linking, the user will specify the elaboration files for all imported functions in addition to the elaboration files for all modules in the design. This provides the BSC with information on how to link to the foreign function. In addition to this link information, the user will have to provide access to the foreign function itself, either as a C source file (`.c`), an object file (`.o`), or from a library (`.a`).

When user provided `.c` files are to be compiled and linked, the C compiler to be used is given by the `CC` environment variable and the flags by the `CFLAGS` and `BSC_CFLAGS` variables. The default compiler is `cc`. If the extension on the file is not `.c`, but `.cxx`, `.cpp` or `.cc`, the C++ compiler will be used instead. The default C++ compiler is `c++`, but the compiler invocation can be controlled with the `CXX`, `CXXFLAGS` and `BSC_CXXFLAGS` environment variables.

Arguments can also be passed through `bsc` directly to the C compiler, C++ compiler and linker using the **-Xc**, **-Xc++** and **-Xl** options, respectively.

As an example, let's say that the user has a module `mkDUT` and a testbench `mkTB` in the file `DUT.bsv`. The testbench uses the foreign C function `compute_vector` to compute an input/output pair for testing the design. Let's assume that the source code for this C function is in a file called `vectors.c`. The command-line and compiler output for compiling and linking this system would look as follows:

```
# bsc -u -sim DUT.bsv
checking package dependencies
compiling DUT.bsv
Foreign import file created: compute_vector.ba
code generation for mkDUT starts
Elaborated Bluesim module file created: mkDUT.ba
code generation for mkTB starts
Elaborated Bluesim module file created: mkTB.ba

# bsc -sim -e mkTB -o bsim mkTB.ba mkDUT.ba compute_vector.ba vectors.c
Bluesim object created: mkTB.{h,o}
Bluesim object created: mkDUT.{h,o}
Bluesim object created: model_mkTB.{h,o}
User object created: vectors.o
Simulation shared library created: bsim.so
Simulation executable created: bsim
```

An elaboration file is created for the foreign name of the function, not the BSV name that the function is imported as. In this example, `compute_vector` is the link name, so the elaboration file is called `compute_vector.ba`.

In this example, the user provided a C source file, which bsc has compiled into an object (here, `vectors.o`). If compilation of the C source file needs access to header files in non-default locations, the user may specify the path to the header files with the `-I` flag (see Section 3.6).

If the user has a pre-compiled object file or library, that file can be specified on the link command-line in place of the source file. In that situation, BSC does not need to compile an object file, as follows:

```
# bsc -sim -e mkTB -o bsim mkTB.ba mkDUT.ba compute_vector.ba vectors.o
Bluesim object created: mkTB.{h,o}
Bluesim object created: mkDUT.{h,o}
Bluesim object created: model_mkTB.{h,o}
Simulation shared library created: bsim.so
Simulation executable created: bsim
```

In both situations, the object file is finally linked with the Bluesim design to create a simulation binary. If the foreign function uses any system libraries, or is itself a system function, then the linking stage will need to include those libraries. From the command line, the user can specify libraries to include with the `-l` flag and can specify non-default paths to the libraries with the `-L` flag (see Section 3.6).

2.5.2 Creating a SystemC Model Instead of a Bluesim Executable

Instead of linking `.ba` files into a Bluesim executable, the linking stage can be instructed to generate a SystemC model by replacing the `-sim` flag with the `-systemc` flag. All other aspects of the linking stage, including the use of environment variables, the object files created, and linking in external libraries, are identical to the normal Bluesim tool flow.

When using the `-systemc` flag, the object files created to describe the design in C++ are not linked into a Bluesim executable. Instead, some additional files are created to provide a SystemC interface to the compiled model. These additional SystemC files use the name of the top-level module extended with a `_systemc` suffix.

```
# bsc -sim GCD.bsv
Elaborated Bluesim module file created: mkGCD.ba

# bsc -systemc -e mkGCD mkGCD.ba
Bluesim object created: mkGCD.{h,o}
Bluesim object created: model_mkGCD.{h,o}
SystemC object created: mkGCD_systemc.{h,o}
```

Remember to define the `SYSTEMC` environment variable to point at your SystemC installation (See Section A).

There are a few additional restrictions on models with which `-systemc` can be used. The top-level interface of the model must not contain any combinational paths through the interface. For the same reason, ActionValue methods and value methods with arguments are not allowed in the top-level interface.

Additionally, value methods in the top-level interface must be free of scheduling constraints that require them to execute after rules in the design. This means that directly registered interfaces are the most suitable boundaries for SystemC model generation.

The SystemC model produced is a clocked, signal-level model. Single-bit ports use the C++ type `bool`, and wider ports use the SystemC type `sc_bv<N>`. Subinterfaces (if any) are flattened into the top-level interface. The names of ports obey the same naming conventions (and the same port-naming attributes) as the Verilog backend (See Section 5.1).

The SystemC model interface is defined in the produced `.h` file, and the implementation of the model is split among the various `.o` files produced. The SystemC model can be instantiated within a larger SystemC design and linked with other SystemC objects to produce a final system executable, or it can be used to cosimulate inside of a suitable Verilog or VHDL simulator.

Division of Functionality Among Files	
File	Purpose
module.{cxx,h,o}	Implementation of modules
model_*topmodule*.{cxx,h,o}	Implementation of the full design and schedule
*topmodule*_systemc.{cxx,h,o}	Top-level SystemC interface

The *module*.{cxx,h,o} files contain the implementations of the modules, each as its own C++ class. The classes have methods corresponding to the rules and methods in the BSV source for the module and member variables for many logic values used in the implementation of the module.

The model_*topmodule*.{cxx,h,o} files combine the individual modules into a single model, defined as a C++ class. The class contains the scheduling logic which sequences rules and method calls and enforces the scheduling constraints during rule execution. The scheduling functions are called only through the simulation kernel, never directly from user code.

The *topmodule*_systemc.{cxx,h,o} files contain the top-level SystemC module for the system. This module is an SC_MODULE with ports for the module clocks and resets as well as for the signals associated with each method in the top-level interface. Its constructor instantiates the implementation modules and initializes the simulation kernel. Its destructor shuts down the simulation kernel and releases the implementation module instances. The SystemC module contains SC_METHODs which are sensitive to the module's clocks and transfer data between the SystemC environment and the implementation classes, translating between SystemC data types and BSV data types.

When linking the produced SystemC objects into a larger system, all of the `.o` files produced must be linked in, as well the standard SystemC libraries and Bluesim kernel and primitive libraries.

```
# c++ -I/usr/local/systemc-2.1/include -L/usr/local/systemc-2.1/lib-linux \
      -I$BLUESPECDIR/Bluesim -L$BLUESPECDIR/Bluesim \
      -o gcd.exe mkGCD.o mkGCD_systemc.o model_mkGCD.o top.cxx TbGCD.cxx \
      -lsystemc -lbskernel -lbsprim -lpthread
```

2.5.3 Linking with Verilog

For the Verilog back end, linking means invoking a Verilog compiler to create a simulator binary file or a script to execute and run the simulation. Section 5 describes the Verilog output in more detail. The Verilog simulator is specified by using the `-vsim` flag.

The `-vsim` flag (along with the equivalent `BSC_VERILOG_SIM` environment variable) governs which Verilog simulator is employed; at present, natively supported choices for `-vsim` are `cvc`, `cver`, `isim`. `iverilog`, `modelsim`, `ncverilog`, `questa`, `vcs`, `vcsi`, `verilator`, `veriwell`, and `xsim`. If the simulator is not specified `bsc` will attempt to detect one of the above simulators and use it.

When the argument to `-vsim` contains the slash character (`/`), then the argument is interpreted as the name of a script to run to create the simulator binary. Indeed, the predefined simulator names listed above refer to scripts in the BSC distribution; thus, `-vsim vcs` is equivalent to `-vsim $BLUESPECDIR/lib/exec/bsc_build_vsim_vcs`. The simulator scripts distributed with BSC are good starting points should the need to use an unsupported simulator arise.

In some cases, you may want to append additional flags to the Verilog simulator command that is used to generate the simulator executable. The `BSC_VSIM_FLAGS` environment variable is used for this purpose.

To add directories to the search path when linking Verilog designs, use the `-vsearch` flag, described in Section 3.6. For example, adding the flag `-vsearch +:verilog_libs` to the BSC command will add the directory `verilog_libs` to the simulator search path (for simulators such as `iverilog` and `vcs/vcsi`). This is equivalent to adding `-y <directory>` to the Verilog compilation command.

The generated Verilog can be put into a larger Verilog design, or run through any existing Verilog tools. BSC also provides a convenient way to link the generated Verilog into a simulation using a top-level module (`main.v`) to provide a clock for the design. The BSC-provided `main.v` module instantiates the top module and toggles the clock every five simulation time units. From the command line the following command generates a simulation binary `mkFoo.exe`:

```
bsc -verilog -e mkFoo -o mkFoo.exe
```

With this command the top level Verilog module `main` is taken from `main.v`. `main.v` provides a clock and a reset, and instantiates `mkFoo`, which should provide an `Empty` interface. An executable file, `mkFoo.exe` is created.

The default `main.v` allows two plusarg arguments to be used during simulation: `+bscvcd` and `+bsccycle`. The argument `+bscvcd` generates a value change dump file (VCD) or a FSDB file; `+bsccycle` prints a message each clock cycle. These are specified from the command line:

```
./mkFoo.exe +bscvcd +bsccycle
```

BSC pre-processor macros

When linking with BSC Verilog files (including `main.v`), the following pre-processor macros can be used to impact the behavior of the Verilog simulation. The `-D` flag, which defines macro values for the `` `defines `` statements must be specified, as described in Section 3.7. These macros are specified when you link the simulation executable, not during runtime.

BSC Verilog Macros		
Name	Description	Example
BSV_ASSIGNMENT_DELAY	Delays assignment at the start of simulation.	-D BSV_ASSIGNMENT_DELAY = #0
BSV_TIMESCALE	Sets the timescale of the simulation.	-D BSV_TIMESCALE = 1ns/1ps
BSV_NO_INITIAL_BLOCKS	Sets initial values to X at the start of simulation.	-D BSV_NO_INITIAL_BLOCKS
BSV_FSDB	Generates a FSDB file during simulation.	-D BSV_FSDB
BSV_POSITIVE_RESET	Changes the sense of reset to asserted hi from asserted low	-D BSV_POSITIVE_RESET
BSV_ASYNC_RESET	Use ASYNC reset for FIFO packages and packages using Counter.v	-D BSV_ASYNC_RESET
BSV_RESET_FIFO_HEAD	Allow reset on head element of FIFO	-D BSV_RESET_FIFO _HEAD
BSV_RESET_FIFO_ARRAY	Allow reset on array elements of FIFO	-D BSV_RESET_FIFO_ARRAY

Imported Verilog functions in Verilog

When Verilog code is generated for a system that uses a Verilog-defined module, the generated code contains an instantiation of this Verilog module with the assumption that the `.v` file containing its definition is available somewhere. This file is needed if the full system is to be simulated or synthesized (the linking stage). Note that VHDL modules can be used instead of Verilog modules if your simulator supports mixed language simulation.

When simulating or synthesizing a design generated with the Verilog back end, you need to include the Verilog descriptions for these primitives. The Verilog descriptions for BSC's primitive elements (FIFOs, registers, etc.) can be found in:

```
${BLUESPECDIR}/Verilog/
```

This directory also contains the file `Bluespec.xcf`, a Xilinx XCF constraint file to be used when synthesizing with Xilinx.

Imported C functions in Verilog

In a BSV design compiled to Verilog, foreign functions are simulated using either the Verilog Procedural Interface (VPI) or the SystemVerilog Direct Programming Interface (DPI). The default is to use VPI, unless the `-use-dpi` flag is provided (see Section 3.3).

For VPI, the generated Verilog calls a user-defined system task anywhere the imported function is needed. The system task is implemented as a C function which is a wrapper around the user's imported C function, to handle the VPI protocols.

For DPI, the generated Verilog calls a function anywhere the imported function is needed. That function is declared as an import-DPI function at the top of the file. In most cases, the declaration imports the user's C function directly, without a wrapper, because SystemVerilog's DPI types closely match BSV's BDPI types. As a result, using DPI is likely more efficient than using VPI, though some simulators may only support one. For polymorphic BDPI functions, the DPI and BDPI types differ and so a wrapper is needed. BSC does not yet support generating the DPI wrapper.

The usual Verilog flow is that BSV modules are generated to Verilog files, which are linked together into a simulation binary. The user has the option of doing the linking manually or by calling BSC. Imported functions can be linked in either case.

As with the Bluesim flow, when compiling a BSV file containing an import-BDPI statement, an elaboration file is generated for the import, containing information about the imported function. However, with Verilog generation, a wrapper function may also be generated. For example, using the scenario from the previous section but compiling to Verilog, and using VPI, the user would see the following:

```
# bsc -u -verilog DUT.bsv
compiling DUT.bsv
Foreign import file created: compute_vector.ba
VPI wrapper files created: vpi_wrapper_compute_vector.{c,h}
code generation for mkDUT starts
Verilog file created: mkDUT.v
code generation for mkTB starts
Verlog file created: mkTB.v
```

The compilation of the import-BDPI statement has not only generated an elaboration file for the input but has also generated the file `vpi_wrapper_compute_vector.c` (and associated header file). This file contains both the wrapper function `compute_vector_calltf()` as well as the registering function for the wrapper, `compute_vector_vpi_register()`. The registering function is what tells the Verilog simulator about the user-defined system task. Included in the comment at the top of the file is information needed for linking manually.

When linking manually, this C file typically needs to be compiled to an object file (`.o` or `.so`) and provided on the command line to the Verilog linker, along with the object files for the user's function (in this example, `vectors.c`). The Verilog linker also needs to be told about the registering function. For some Verilog simulators, the registering function is named on the command-line. For other simulators, a C object file must be created containing the array `vpi_startup_array` with pointers to all of the registering functions (to be executed on start-up of the simulation). An example of this start-up array is given in the comment at the top of the generated wrapper C files. Some simulators require a table for imported system functions (as opposed to system tasks). The table is provided in a file with `.tab` or `.sft` extension. The text to be put in these files is also given in the comment at the top of the wrapper file. The text also appears later in the file with the tag "`tab:`" or "`sft:`" prepended. A search for the appropriate tag (with a tool like `grep`) will extract the necessary lines to create the table file.

Linking via BSC does all of this automatically:

```
# bsc -verilog -e mkTB -o vsim mkTB.v mkDUT.v compute_vector.ba vectors.c
VPI registration array file created: vpi_startup_array.c
User object created: vectors.o
VPI object created: vpi_wrapper_compute_vector.o
VPI object created: vpi_startup_array.o
Verilog binary file created: vsim
```

To perform linking via BSC, the user provides on the command-line not only the Verilog files for the design but also the foreign import files (`.ba`) for each imported function and the C source or object files implementing the foreign functions. As shown in the above example, the linking process will create the file `vpi_startup_array.c`, containing the registration array, and will compile it to an object file. The linking process will then pass all of the VPI files along to the Verilog simulation build script (see Section 2.5.3) which will create any necessary table files and invoke the Verilog simulator with the proper command-line syntax for using VPI.

Using DPI is a simpler process. No separate registration files are needed because the imported function is declared by a SystemVerilog import-DPI statement in the generated Verilog itself. And

for most import-BDPI functions, no wrappers are generated either. This makes manual linking particularly easier, although linking via BSC is also possible using DPI. The Verilog simulation build script will consult the BSC flags to determine whether imported functions are to be provided to the simulator as DPI or VPI. It is therefore necessary to use the same flags for choosing DPI/VPI during linking as were used during compilation.

If the foreign function uses any system libraries, or is itself a system function, then the Verilog linking will need to include those libraries. As with the Bluesim flow, the user can specify to BSC the libraries to include with the `-l` flag and can specify non-default paths to the libraries with the `-L` flag (see Section 3.6).

2.6 Simulate

The result of linking is a simulation executable.

You can generate a VCD file from either Bluesim or any of the supported Verilog simulators. When simulating with Bluesim, use the `-V` flag. For Verilog simulators using the BSC-provided `main.v` file, specify the `+bscvcd` flag during simulation.

To dump a FSDB file directly from a supported Verilog simulator using the BSC-provided `main.v` file, you need to specify the `+bscvcd` flag during simulation and the `-D BSV_FSDB` flag during linking. Note that not all simulators can generate an FSDB file. Additional command line arguments are required, dependent on the simulator. Bluesim does not generate FSDB files at this time.

3 BSC flags

You can obtain an up-to-date listing of the available flags along with brief explanations by going to a Unix command line and entering:

```
bsc  -help
```

Most flags may be preceded by a `-no` to reverse their effect. Flags that appear later on the command line override earlier ones.

The following flags make the compiler print more or less progress-report messages as it does its work:

```
-quiet                be less talkative
-q                    same as -quiet
-verbose              be more talkative
-v                    same as -verbose
```

The compiler has four levels of verbosity: quiet, normal, verbose, and extra verbose. The default is normal verbosity and the above flags can be used to move up and down one step.

3.1 Common compile and linking flags

The following flags are the common flags used by the compiler.

```
-g module              generate code for 'module' (requires -sim or -verilog)
-u                     check and recompile packages that are not up to date
-sim                   compile BSV generating Bluesim object
-verilog               compile BSV generating Verilog file
-vsim simulator        specify which Verilog simulator to use
-e module              top-level module for simulation
-o name                name of generated executable
-elab                  generate a .ba file after elaboration and scheduling
```

A BSV source file is compiled with a command like this:

```
bsc [flags] Foo.bsv
```

where `Foo.bsv` is the top file in the design.

If no flags are provided, the compile stops after the type checking phase. To compile through code generation, you must provide a flag indicating whether the target is Bluesim (`-sim`) or Verilog (`-verilog`).

For example, to compile to code generation for Bluesim:

```
bsc -sim Foo.bsv
```

or for Verilog:

```
bsc -verilog Foo.bsv
```

As discussed in Section 2.4.3, when compiling to code generation a module must be specified, using either the `synthesize` attribute in the BSV code, or the `-g` flag at compile time. From the command line, multiple modules can be specified for code generation at the same time.

For example:

```
bsc -sim -g mkFoo -g mkBaz Foo.bsv
```

Linking requires a second call to the compiler, as described in Section 2.5. When linking you must specify the top-level module with the `-e` flag. The name following the flag must be the name of a BSV module and only one module can be specified. You must also specify the back end with either the `-sim` or `-verilog` flag.

For example, to link for Bluesim:

```
bsc -sim -e mkFoo
```

or for Verilog:

```
bsc -verilog -e mkFoo
```

The `-vsim` flag (along with the equivalent `BSC_VERILOG_SIM` environment variable) governs which Verilog simulator is employed. The natively supported choices for `-vsim` are `cvc`, `cver`, `isim`. `iverilog`, `modelsim`, `ncverilog`, `questa`, `vcs`, `vcsi`, `verilator`, `veriwell`, and `xsim`. If a simulator is not specified `bsc` will attempt to detect one of the above simulators and use it.

When using Bluetcl (or applications built on Bluetcl) to analyze a compiled design, a `.ba` file is required (for each module in the design). To generate this file when compiling add the `-elab` flag to the compile command.

3.2 Controlling default flag values

The environment variable BSC_OPTIONS enables the user to set default flag values to be used each time the compiler is called. If set, the value of BSC_OPTIONS is automatically prepended to the compiler option values typed on the bsc command line. This avoids the need to set specified flag values each time the compiler is called.

For instance, in order to control the default value of the -p (path) option, the BSC_OPTIONS environment variable could be set as follows:

```
# BSC Environment for csh/tcsh
setenv BSC_OPTIONS "-p .:./MyLib:+"

# BSC Environment for bash/ksh
export BSC_OPTIONS="-p .:./MyLib:+"
```

Once set, BSC would now search for packages in the ./MyLib directory before looking in the default library areas. Note that since the compiler recognizes multiple uses of the same flag on the command line, the user can use the -p flag along with the BSC_OPTIONS environment variable to control the search path. For example, if in addition to the BSC_OPTIONS set above the user enters the following bsc command, :

```
bsc -verilog -p ./MyLib2:+ Foo.bsv
```

the compiler would now use the path

```
./MyLib2:.:./MyLib:+
```

which is a prepending of the -p command line value to the value set by the BSC_OPTIONS environment variable.

3.3 Verilog back-end

The following additional flags are available when using the Verilog back end.

```
-remove-unused-modules  remove unconnected modules from the Verilog
-v95                    generate strict Verilog 95 code
-unspecified-to val     remaining unspecified values are set to:
                         'X', '0', '1', 'Z', or 'A'
-remove-dollar          remove dollar signs from Verilog identifiers
-Xv arg                 pass argument to the Verilog link process
-verilog-filter cmd     invoke a command to post-process the generated Verilog
-use-dpi                use DPI instead of VPI in generated Verilog
```

The -remove-unused-modules flag will remove from the generated Verilog any modules which are not connected to an output. This has has the effect of removing redundant or unused modules, which would also be done by synthesis tools. This option should be used on modules undergoing synthesis, and not be used for testbench modules.

The -v95 flag restricts the Verilog output to pure Verilog-95. By default, the Verilog output uses features which are not in the Verilog-95 standard. These features include passing module parameters by name and use of the $signed system task for formatting $display output. When the -v95 flag

is turned on, uses of these features are removed, but comments are left in the Verilog indicating the parameter names or system tasks which were removed.

The `-unspecified-to val` flag defines the value which any remaining unspecified values should be tied to. The valid set of values are: `X`, `0`, `1`, `Z`, or `A`, where the first four correspond to the Verilog value, and `A` corresponds to a vector of alternating ones and zeros. The default value is `A`. The choice of value is used by both the Verilog and Bluesim back ends. However, since Bluesim is a two-value simulator, it does not support the values `X` and `Z`. For final synthesis runs, the use of `X` (or `0`) is strongly suggested to give the best synthesis results.

The `-remove-dollar` flag causes identifiers in Verilog output to substitute underscores instead of dollar signs to separate instance names from port names. If this substitution causes a name collision, the underscore is suffixed with a number until a non-colliding name is found.

The `-Xv` flag passes the specified string argument to the Verilog link process. Only one argument can be passed with each `-Xv` flag. If you want to pass multiple arguments, then the flag must be specified multiple times, once for each argument.

The `-verilog-filter` flag invokes a command to process the Verilog file generated by the compiler. The command can be a Unix command or script; it must take a single argument, the name of the Verilog file. The flag can be used multiple times; the filters are applied in the order they are given on the command line.

The `-use-dpi` flag controls whether imported C functions in source designs are implemented in Verilog using the Verilog Procedural Interface (VPI) or the SystemVerilog Direct Programming Interface (DPI) . The flag must be consistently used during both the compilation and linking steps. During compilation, the flag controls how imported C calls appear in the generated Verilog; during linking, the flag controls how the C functions are provided to the simulator. By default, this flag is *off* and VPI is used.

3.4 Bluesim back-end

The following flags are available when using the Bluesim back end.

```
-parallel-sim-link jobs     specify the # of simultaneous jobs when linking Bluesim
-systemc                    generate a SystemC model
```

During the Bluesim linking process, the compiler generates C++ source files (`.cxx` and `.h` files) and then runs the C++ compiler on each one. By default, these compilations are performed serially. In some cases, the C++ compiler may take time to compile a file. In that case, it would be helpful to compile other files in parallel to reduce the total compilation time. The `-parallel-sim-link` flag directs the process to proceed in parallel, up to *jobs* number of simultaneous compilations.

The default value of *jobs* is 1, in which case the compilation occurs in serial. Only values greater than 1 will allow parallel compilations.

When compilation occurs in parallel, it is controlled by a Makefile, described in Appendix A.4. When the verbose (`-v`) flag is used to compile, you will now see the command for the `make` excecution. The commands for compiling each object will still be shown as well. Since the compilations are happening in parallel, the messages will be interspersed with each other and will not show up in serial order (Section 4.2.1).

The `-systemc` flag instructs the linking stage to generate a SystemC model instead of a Bluesim executable (Section 2.5.2). When using this flag, the object files created to describe the design in C++ are not linked into a Bluesim exectuable. Instead, some additional files are created to provide a SystemC interface to the compiled model. These additional SystemC files use the name of the top-level module extended with a `_systemc` suffix.

3.5 Resource scheduling (all back ends)

The following flags are available to direct resource scheduling:

```
-resource-off          fail on insufficient resources
-resource-simple       reschedule on insufficient resources
```

Resource scheduling for a particular interface method involves finding all rules that call that method. A single method name can refer to multiple ports in the hardware — for example, a double-ported RAM can have two *read* ports, but a design in BSV can use the name `read` and it will rely on the compiler to determine which port is being used. If the number of rules that use `read` is two or less, then there is no problem; each rule is connected to its own port and there is never any contention. If the number of rules vying for a method is more than the number of copies of that method, then a problem exists.

If `-resource-off` is specified, the compiler will give up and tell the user that resource scheduling is not possible. This is the default behavior. The straightforward way to proceed is by adding logic that explicitly arbitrates between the competing rules (choosing the more important one to fire depending on the situation).

The alternative way to resolve a resource conflict is to block competing rules until the number of rules vying for a method is less than the number of available ports for that method. This behavior can be turned *on* with the `-resource-simple` flag. The compiler selects rules to block from the competing rules arbitrarily (and may change its selection when different compilation flags or compiler versions are used), so this flag is not recommended for a completed design, but automatic resource arbitration can be useful when experimenting.

3.6 Setting the path

```
-i dir                 override $BLUESPECDIR
-p path                directory path (':' sep.) for source and intermediate
                       files
-bdir dir              output directory for .bo and .ba files
-simdir dir            output directory for Bluesim intermediate files
-vdir dir              output directory for .v files
-vsearch path          search path (':' sep.) for Verilog files
-info-dir dir          output directory for informational files
-I path                include path for compiling foreign C/C++ source
-L path                library path for linking foreign C/C++ objects
-l library             library to use when linking foreign C/C++ objects
-fdir dir              working directory for relative file paths during elaboration
```

There are default locations where the compiler looks for source and and intermediate files. The flags `-i` and `-p` are available to override the default locations or to specify additional directories to search in. See Section 2.4.2 for more information. The `-i` flag overrides the environment variable `BLUESPECDIR`, which is used in the default value for the directory path of the `-p` flag. The `-p` flag takes a path argument, which is a colon-delimited list of directories. This path is used to find Bluespec source and intermediate files imported by the package being compiled (including the standard prelude, and files included by the BSV preprocessor). The path can contain the character `%`, representing the `BLUESPECDIR` directory, as well as `+`, representing the current path. The default path is:

```
.:%/Libraries
```

The `-bdir`, `-simdir`, `-vdir`, and `-info-dir` flags specify where output files should be placed. The default is the directory in which the input file(s) reside.

The `-vsearch` flag specifies the search path used when linking Verilog files. The `-vsearch` flag takes a path argument in the same style of path specification as the `-p` flag, including the use of `+` and `%`. Since this flag specifies where to look for the `.v` files, the path specified by the `-vdir` flag is automatically added to the front of the `-vsearch` path.

The flags `-I`, `-L`, and `-l` are used during the linking stage when foreign C functions are imported. The `-I` and `-L` flags add to the path of where to find C header files and libraries, respectively. The libraries to be used during linking are specified by the `-l` flag.

The flag `-fdir` specifies where relative file paths will be based during elaboration, including calls to `openFile`.

3.7 Miscellaneous flags

Here are some other flags recognized by the compiler:

```
-D macro                define a macro for the BSV or Verilog preprocessor
-E                      run just the preprocessor, dumping result to stdout
-print-flags            print flag values after command-line parsing
-steps n                terminate elaboration after this many function
                        unfolding steps
-steps-max-intervals n  terminate elaboration after this number of unfolding
                        messages
-steps-warn-interval n  issue a warning each time this many unfolding steps are
                        executed
-reset-prefix name      reset name or prefix for generated modules
-show-timestamps        include timestamps in generated files
-show-version           include compiler version in generated files
```

Preprocessor macros may be defined on the command line using the `-D` option. Two versions are supported, a simple macro definition and an assignment of a string to a macro:

```
-D foo
-D size=148
```

Note that a space is required after the `-D`, and that no spaces are allowed in the macro names, values or around the equals.

The `-D` option can also be used during the linking run of bsc to define macro values for `‘define` statements in the Verilog.

The settings that are being used by the compiler can be dumped with `-print-flags`.

Function definitions in BSV are purely compile-time entities. The compiler replaces all function calls by their bodies and continually simplifies expressions. Function definitions may be recursive as long as this substitution and simplification process terminates, but of course the compiler cannot predict whether it will terminate. The `-steps`, `-steps-warn-interval` and `-steps-max-intervals` flags provide feedback and safety mechanisms for potentially infinite function unfoldings. The `-steps-warn-interval` tells the compiler to issue a compilation warning every time that many function unfolding steps are executed. This provides feedback to a designer that a particular design requires an unusual amount of effort to elaborate. A designer may choose to terminate elaboration and investigate whether there is a bug, infinite loop or an inefficient construct in a design

or they may choose to let elaboration proceed to see if additional time will result in elaboration completing. The `-steps-max-intervals` flag is the safety mechanism. It prevents an unattended compilation from consuming resources indefinitely by terminating elaboration after a certain number of function unfolding warnings. This means, for example, with the default values of 100000 for `-steps-warn-interval` and 10 for `-steps-max-intervals` an infinite compilation will execute for 1000000 steps, issuing 9 unfolding warnings before terminating with an unfolding error message. The `-steps` flag is a simpler version of this mechanism. It is equivalent to setting `-steps-warn-interval` to the argument of `-steps` and `-steps-max-intervals` to 1.

The default name for a reset in generated modules is `RST_N`. This can be changed with the `-reset-prefix <name>` flag. For example, to set all reset names to `RST_P` use: `-reset-prefix RST_P`.

BSC attempts to be helpful by including identifying information in the files it generates. Specifically, it records the compiler version and a timestamp of when the file was generated. This can interfere with build systems where rebuilds are determined by the hash of the tool inputs. Recording the timestamp causes a change on every build; avoiding that can significantly improve the effectiveness of the build cache. The `-no-show-timestamps` flag directs BSC to not include the timestamp. Removing the compiler version can help for files which have not otherwise changed between versions. The `-no-show-version` flag directs BSC to not include the compiler version.

3.8 Run-time system

These flags are passed along to the Haskell compiler run-time system that is used to execute BSC. Among the RTS flags available are:

```
-Hsize                  set the maximum heap size
-Ksize                  set the maximum stack size
```

As the compiler executes, it allocates its internal intermediate data structures in a heap memory managed by its run-time system (RTS). When compiling a large BSV design, the compiler may run out of heap space. If you encounter this, please rerun the compiler with a larger heap space, using the flags:

```
bsc ... +RTS -H<size> -RTS ...
```

For example, to use a 1 gigabyte heap, you would enter:

```
bsc ... +RTS -H1G -RTS ...
```

Similarly, if you run out of stack space, you can increase the stack with the `-K` RTS flag. If a design runs out of stack space, it is probably caught in an infinite loop. For large designs that involve many recursive functions, it may be necessary to increase the stack size. If you run out of stack space, first try increasing the stack to a reasonable size, such as 10 or 15 megabytes. If you still exhaust the stack memory, try examining your design for infinite loops.

Any flags encapsulated between `+RTS` and `-RTS` are passed to the run-time system and are not given to the compiler itself. In addition to `-H` and `-K`, various flags are available to control garbage collection, memory usage, function unfolding, etc. However, the user should never need to use these other flags.

3.9 Automatic recompilation

```
-u                      check and recompile packages that are not up to date
-show-compiles          show recompilations
```

The `-u` flag implements a `make`-like functionality. If a needed `.bo` file is found to be older or non-existent compared to the `.bsv` file, the latter is recompiled. Similarly, if a `.bsv` file has a modification time that is more recent than that of any of its generated Verilog or Bluesim modules, the `.bsv` file is recompiled.

The `-show-compiles` flag turns *on* the compiler output during recompilation of auxiliary files. It can also be used as `-no-show-compiles` to suppress the compiler output.

For the purposes of comparing modification times, the intermediate files (`.bo` and `.ba`) are assumed to be in the same directory as the `.bsv` source file. If no file is found there, the compiler then searches in the directory specified by the `-bdir` flag (if used). The generated Verilog files and Bluesim files are assumed to be in the same directory as the source unless the `-simdir` or `-vdir` flag is used, respectively.

3.10 Compiler transformations

```
-aggressive-conditions  construct implicit conditions aggressively
-split-if               split "if" in actions
-lift                   lift method calls in "if" actions
```

When a rule contains an `if`-statement, the compiler has the option either of splitting the rule into two mutually exclusive rules, or leaving it as one rule for scheduling but using MUXes in the production of the action. Rule splitting can sometimes be desirable because the two split rules are scheduled independently, so non-conflicting branches of otherwise conflicting rules can be scheduled concurrently. The `-split-if` flag tells the compiler to split rules. Splitting is turned *off* by default for two reasons:

- When a rule contains many `if`-statements, it can lead to an exponential explosion in the number of rules. A rule with 15 `if`-statements might split into 2^{15} rules, depending on how independent the statements (and their branch conditions) are. An explosion in the number of rules can dramatically slow down (and cause other problems) for later compiler phases, particularly scheduling.
- Splitting propagates the branch condition of each `if` to the predicates of the split rules. Resources required to compute rule predicates are reserved on every cycle. If a branch condition requires a scarce resource, this can starve other parts of the design that want to use that resource.

If you need the effect of splitting for certain rules, but do not want to split all the rules in an entire design using `-split-if`, use the `(*split*)` and `(*nosplit*)` attributes, as described in the BSV Reference Guide.

When rules are not split along `if`-statements, it is important to lift actions through the `if`-statement. If both branches of an `if`-statement call the same method but with different arguments, it's better to make one call to the method and MUX the argument. The `-lift` flag turns *on* this optimization. Lifting is recommended when rule splitting is turned *off*. When rule splitting is *on*, lifting is not required and can make rules more resource hungry. Currently, lifting with splitting *off* can result in poor resource allocation, so we recommend using `-no-lift` with `-split-if`.

When the action in a branch of an `if`-statement has an implicit condition, that condition needs to be propagated to the rule predicate. This can be done conservatively, by simply placing implicit conditions for all branches in the predicate. Or it can be done more aggressively (i.e. attempting to fire the concerned rule more often), by linking each implicit condition with its associated branch condition. The flag `-aggressive-conditions` turns *on* this feature. This flag is off by default because, as discussed above, propagating branch conditions to rule predicates can have undesirable effects. However, if `-split-if` is on, branch conditions will be propagated to rule predicates regardless, so we recommend using `-aggressive-conditions` with `-split-if`, since it may improve the generated schedule.

3.11 Compiler optimizations

`-opt-undetermined-vals`	`aggressive optimization of undetermined values`
`-sat-stp`	`use STP SMT for disjoint testing and SAT`
`-sat-yices`	`use Yices SMT for disjoint testing and SAT`

In late stages of the compiler, don't-care values are converted into specific constants. In order that the Verilog and Bluesim simulation paths produce exactly the same value dumps, the compiler assigns a value to the don't-care signals at the point where the Verilog and Bluesim back ends diverge. However, the Verilog back end can generate more efficient hardware if it is allowed to assign the don't-care signals better values based on context. The `-opt-undetermined-vals` flag permits the Verilog back end of the compiler to make better decisions about don't-care values. This flag is *off* by default. Turning this flag on may produce better hardware in Verilog, but can result in the Bluesim and Verilog simulations producing different intermediate values.

It is possible to change the underlying proof engine used by the compiler. You should not use these flags or switch proof engines unless you experience performance issues during the scheduling or Verilog optimization phases of the bsc compile. The default proof engine is the Yices solver.

3.12 BSV debugging flags

The following flags might be useful in debugging a BSV design:

```
-check-assert            test assertions with the Assert library
-keep-fires              preserve CAN_FIRE and WILL_FIRE signals
-keep-inlined-boundaries preserve inlined register and wire boundaries
-remove-false-rules      remove rules whose condition is provably false
-remove-starved-rules    remove rules that are never fired by the generated
                         schedule
-remove-empty-rules      remove rules whose bodies have no actions
-show-module-use         output instantiated Verilog modules names
-show-range-conflict     show predicates when reporting a parallel-
                         composability error
-show-method-conf        show method conflict information in the generated code
-show-method-bvi         show BVI format method schedule information in the
                         generated code
-show-stats              show package statistics
-continue-after-errors   aggressively continue compilation after an error has
                         been detected
-warn-method-urgency     warn when a method's urgency is arbitrarily chosen
-warn-action-shadowing   warn when a rule's action is overwritten by a later rule
-suppress-warnings list  ignore a list of warnings (':' sep list of tags)
-promote-warnings list   treat a list of warnings as errors (':' sep list of tags)
-demote-errors list      treat a list of errors as warnings (':' sep list of tags)
-show-elab-progress      display trace as modules, rules, methods are elaborated
```

The `-check-assert` flag instructs the compiler to abort compilation if a boolean assertion ever fails. These are assertions which are explicitly embedded in a BSV design using the `Assert` package (see the Bluespec Reference Guide). If this flag is *off*, assertions of this type in a BSV design are ignored.

To view rule firings in the Verilog output, use the `-keep-fires` flag. This flag will direct the compiler to leave the `CAN_FIRE` and `WILL_FIRE` signals in the output Verilog (some of which might otherwise be optimized away). These signals are generated for each rule and indicate whether a rule's predicate would allow the rule to fire in the current cycle and whether the scheduler chose the rule to fire in the current cycle, respectively. Leaving these signals in the Verilog allows the designer to dump the signals to VCD and view the firings in a waveform viewer.

When elaborating a design, if the compiler determines that a rule's explicit condition is always false, it issues a warning about the situation and removes the rule from the design (so the presumably irrelevant rule does not interfere with scheduling). Sometimes, for debugging purposes it can be helpful to preserve this (never enabled) rule in the output code. That can be done by disabling the `-remove-false-rules` flag (i.e. passing `-no-remove-false-rules`). As you might expect, the compiler will find more false rules when aggressive optimization (i.e. `-O`) is turned on, but it can be helpful to turn *off* `-O` when you want to examine the condition that the compiler can prove false.

Similarly, the compiler might determine, after scheduling, that a rule will never fire because conflicting rule(s) block it whenever it is enabled. The compiler warns about such rules, but does *not* remove them by default because they probably indicate an important problem in scheduling the design. If you wish to removed these rules, you can use the `-remove-starved-rules` flag.

The compiler may also determine that the body of a rule has no actions, either because there are no actions in the body or because the compiler can prove at elaboration time that none of the actions in the body can happen. The `-remove-empty-rules` flag causes these rules to be removed when it is on, which it is by default. The compiler will generate a warning for such rules, since they are likely to indicate a problem in the design.

Conflict relationships between methods of the generated module's interface can be dumped (in the generated code) with the `-show-method-conf` flag. This is useful for documenting the interface protocol a generated module expects (particularly when the generated module is going to be called

by non-Bluespec modules). The `-show-method-bvi` flag is helpful when writing an `importBVI` statement. It displays the method conflicts in a format that can be cut and pasted into an `importBVI` statement. These flags are not enabled by default.

The `-show-stats` flag dumps various statistics at the end of each compiler stage (such as the number of rules and number of definitions). To find out what Verilog modules are instantiated by the generated module, use the `-show-module-use` flag. This flag causes the compiler to create a file `mkFoo.use` which contains a list of each Verilog module instantiated by module `mkFoo`, separated by newlines.

The `-show-range-conflict` flag is used to display more information when the compiler reports error message `G0004`. By default, the compiler omits the conditions of the method calls, because they can be very large expressions in some cases, which distract from debugging rather than help. When more detail is required, the `-show-range-conflict` flag can be turned on and the full condition is displayed.

By default, the compiler stops once it finds errors in a module. The `-continue-after-errors` flag allows the compiler to continue on to other modules and other phases after an error is encountered. This may be helpful in finding multiple errors in a single compile, though some of later errors may be misleading and vanish once the cause of initial error is fixed. Note that the compiler may not be able to successfully complete because of the cumulative effects of errors encountered.

The `-warn-method-urgency` flag displays a warning when a method and a rule have an arbitrary urgency order. By default the flag is on.

The `-warn-action-shadowing` flag displays a warning when there are two rules executing in the same cycle and calling the same Action method. In this case, the state update of the first method is ignored because it is overwritten by the state update of the second method. This can only occur for methods which are annotated as non-conflicting, for example, register writes. Otherwise the Action methods will conflict. This flag is on by default.

The `-suppress-warnings, -promote-warnings,` and `-demote-errors` flags provide control over which warnings and errors are displayed. The flags take a list of specific warnings or errors (separated by :) or the option `ALL` or `NONE`.

The `-suppress-warnings` flag ignores all warnings of the types provided in the list. For example, when compiling with the flag `bsc -suppress-warnings G0117:G0020`, the warnings of types `G0117` and `G0020` are not displayed.

When any warnings are suppressed a warning of type `S0080` is displayed along with the number of warnings suppressed. Example:

```
Warning: Unknown position: (S0080)
  2 warnings were suppressed.
```

This message can also be suppressed by including it in the list:

```
bsc -suppress-warnings G0117:G0020:S0080 ...
```

The `-promote-warnings` flag transforms a warning into an error, while the `-demote-errors` flag tranforms an error into a warning. A warning is displayed if you attempt to demote an error which cannot be demoted. Example:

```
Warning: Command line: (S0094)
  Cannot demote the following errors:
    G0047, G0048
```

A message tag can be in multiple lists, in which case the following process is followed. For a warning, the suppression list is checked first. If the warning is in that list, it is suppressed. Otherwise, the compiler checks whether it is in the promotion list. If it is in both the promotion and demotion lists, it is reported as a warning. If it is only in the promotion list, it is reported as an error. This allows you to specify `-promote-warnings ALL` and then specifically exempt certain warnings from being promoted by using `-demote-errors list`.

For an error, the promotion list is irrelevant. If an error is not in the demotion list, it is reported as an error. If the error is in the demotion list, the suppression list is checked. If it appears in the suppression list, then the warning is not reported at all.

The `-show-elab-progress` flag directs the compiler to print trace statements as it expands a module. If the compiler appears to hang, this flag enables the user to see what the last trace line was, which should indicate where in the design the hang occurs. Section 4.1.2 provides more detail on the messages generated by the flag.

3.13 Understanding the schedule

These flags generate output to help you understand the schedule for a generated module.

`-show-rule-rel r1 r2`	`display scheduling information about rules r1 and r2`
`-show-schedule`	`show generated schedule`
`-sched-dot`	`generate .dot files with schedule information`

If the rules in a design are not firing the way you thought they would, the `-show-schedule` and the `-show-rule-rel` flags may help you inspect the situation. See section 4.2.2 for a description on the output generated by these flags.

The `-sched-dot` flag generates `.dot` (DOT) files which contain text representation of a graph. The files are placed in the directory specified by the `-info-dir` flag. There are many tools in the `graphviz` family, for example `dotty`, which read, manipulate, and render DOT files to visible format. See `www.graphviz.org` for more information.

When specified, the following graph files are generated for each synthesized module (*mod* is the module name):

1. **conflicts** (*mod*_conflict.dot)
2. **execution order** (*mod*_exec.dot)
3. **urgency** (*mod*_urgency.dot)
4. **combined** (*mod*_combined.dot)
5. **combined full** (*mod*_combined_full.dot)

In each of these graphs, the nodes are rules and methods and the edges represent some relationship between pairs of rules/methods. In all graphs, methods are represented by a box and rules are represented by an ellipse, so that they are visually distinguishable.

conflicts (*mod*_conflict.dot) A graph of rules/methods which conflict either completely (cannot execute in the same cycle) or conflict in one direction (if they execute in the same cycle, it has the be in the opposite order). Complete conflicts are represented by bold non-directional edges. Ordering conflicts are represented by dashed directional edges, pointing from the node which must execute first to the node which must execute second.

When a group of nodes form an execution cycle (such as A before B before C before A), the compiler breaks the cycle by turning one of the edges into a complete conflict and emits a warning. This DOT file is generated before that happens, so it includes any cycles and can be used to debug any such warnings.

execution order (*mod*_exec.dot) This is similar to the conflicts graph, except that it only includes the execution order edges; the full-conflict edges have been dropped. As a result, there is no need to distinguish between the types of edges (bold versus dashed), so all edges appear as normal directional edges.

This DOT file is generated after cycles have been broken and therefore describes the final execution order for all rules/methods in the module.

urgency (*mod*_urgency.dot) The edges in this graph represent urgency dependencies. They are directional edges which point from a more urgent node to a less urgent node (meaning that if the rules/methods conflict, then the more urgent one will execute and block the less urgent one). Two rules/methods have an edge either because the user specified a `descending_urgency` attribute or because there is a data path (though method calls) from the execution of the first rule/method to the predicate of the second rule/method.

If there is a cycle in the urgency graph, bsc reports an error. This DOT file is generated before such errors, so it will contain any cycles and is available to help debug the situation.

combined (*mod*_combined.dot) In this and the following graph, there are two nodes for each rule/method. One node represents the scheduling of the rule/method (computing the `CAN_FIRE` and the `WILL_FIRE` signals) and one node represents the execution of the rule/method's body. The nodes are labelled `Sched` and `Exec` along with the rule/method name. To further help visually distinguish the nodes, the `Sched` nodes are shaded.

The edges in this graph are a combination of the execution order and urgency graphs. This is the graph in which the microsteps of a cycle are performed: compute whether a rule will fire, execute a rule, and so on.

In the rare event that the graph has a cycle, bsc will report an error. This DOT file is generated prior to that error, so it will contain the cycle and be available to help in debugging the situation.

combined full (*mod*_combined_full.dot) Sometimes the execution or urgency order between two rules/methods is under specified and either order is a legal schedule. In those cases, bsc picks an order and warns the user that it did so.

This DOT graph is the same as the combined graph above, except that it includes the arbitrary edges which the compiler inserted. The new edges are bold and colored blue, to help highlight them visually.

This is the final graph which determines the static schedule of a module (the microsteps of computing predicates and executing bodies).

As with the above graph, there are separate `Sched` and `Exec` nodes for each rule/method, where the `Sched` nodes are shaded.

3.14 C/C++ flags

These flags run the C preprocessor and pass arguments to C tools.

```
-cpp                    preprocess the source with the C preprocessor
-Xc arg                 pass argument to the C compiler
-Xc++ arg               pass argument to the C++ compiler
-Xcpp arg               pass argument to the C preprocessor
-Xl arg                 pass argument to the C/C++ linker
```

The `-cpp` flags runs the C preprocessor on the source file before the BSV preprocessor is run. The `CC` environment variable specifies which C compiler will be used. If the environment variable is not specified, the compiler will run the default (`cc`) which must be found in the path.

The flags `-Xcpp`, `-Xc`, `-Xc++`, and `-Xl` pass the specified argument to the C preprocessor, C compiler, C++ compiler and C/C++ linker respectively. Only one argument can be passed with each `-X` flag. If you want to pass multiple arguments, then the flag must be specified multiple times, once for each argument. Example:

```
-Xcpp -Dfoo=bar -Xcpp /l/usr/local/special/include
```

4 Compiler messages

4.1 Warnings and Errors

The following is an example of a warning from BSC:

```
Warning: "Test.bsv", line 5, column 9: (G0021)
  According to the generated schedule, rule "r1" can never fire.
```

All warnings and errors have this form, as illustrated below. They begin with the position of the problem, a tag which is unique for each message, and the type (either "Error" or "Warning").

```
<type>: <position>: (<tag>)
  <message>
```

The unique tag consists of a letter, indicating the class of message, and a four digit number. There are four classes of messages. Tags beginning with `P` are for warnings and errors in the *parsing* stage of the compiler. Tags beginning with `T` are *type-checking* and elaboration messages. Tags beginning with `G` are for problems in the back-end, or code-*generation*, including rule scheduling. Tags beginning with `S` are for file handling problems, command-line errors, and other *system* messages.

4.1.1 Type-checking Errors

If there is a type mismatch in your design, you will encounter a message like this:

```
Error: "Test.bsv", line 3, column 10: (T0020)
  Type error at:
  x

  Expected type:
  Prelude::Bool

  Inferred type:
  Prelude::Bit#(8)
```

This message points to an expression (here, `x`) whose type does not match the type expected by the surrounding code.

You can think of this like trying to put a square block into a round hole. The square type and the round type don't match, so there is a problem. The type of the expression (the block) doesn't match the type that the surrounding code is expecting (the hole). In the error message, the "expected type" is hole and the "inferred type" is the block.

4.1.2 Elaboration Messages

All errors and warnings during the elaboration phase include some additional lines at the end of the message to indicate where in the hierarchy the compiler was elaborating when the error occurred. The elaboration can be inside a rule, inside a method, or inside a module. And that location can be instantiated from inside a module, which was instantiated inside a module, and so on, up to the top level. There is one line of output for each level, with the last line showing the top module being elaborated.

```
Error: "Example.bsv", line 6, column 30: (T0051)
 Literal 17 is not a valid Bit#(4).
 During elaboration of the body of rule ‘shift’ at "Example.bsv", line 20,
 column 9
 During elaboration of ‘mkStage’ at "Example.bsv", line 43, column 7.
 During elaboration of ‘Loop’ at "Example.bsv", line 41, column 4.
 During elaboration of ‘mkTop’ at "Example.bsv", line 58, column 4.
```

The position for each level is provided for convenience.

Here is an example of the message when an error occurs in the method of the top module's interface:

```
During elaboration of the interface method ‘put’ at "Example.bsv",
line 86, column 8.
During elaboration of ‘mkTop’ at "Example.bsv", line 58, column 4.
```

When elaborating inside a rule, the error message would look like this:

```
During elaboration of rule "doDisp" at "Test.bsv", line 3, column 9.
...
```

Or if more information is known, the message can indicate which part of the rule was being elaborated, as shown by the following three examples:

```
During elaboration of the explicit condition of rule "doDisp" at "Test.bsv",
  line 3, column 9.
...
```

```
During elaboration of the body of rule "doDisp" at "Test.bsv", line 3,
  column 9.
...
```

```
During elaboration of the implicit condition of rule "doDisp" at "Test.bsv",
  line 3, column 9.
  ...
```

Similarly, if the error is during elaboration of the explicit or implicit condition of the method, that is reported as well. In addition to methods, there are also messages for output clocks, resets, and inouts in the top module's interface:

```
During elaboration of the interface output clock ‘clk_out’ at
"Example.bsv",
  line 6, column 8.
  ...
```

```
During elaboration of the interface inout ‘io_out’ at "Example.bsv",
  line 6, column 8.
  ...
```

The messages displayed at the end of elaboration are helpful when the compiler emits a message. Sometimes there's no message or the compiler appears to hang. In these cases, you can use the `-show-elab-progress` flag to see what parts of the design are taking up time. When the flag is set, the compiler prints status messages as it enters and exits parts of the design.

Messages are generated for submodules, rules (divided into explicit condition, implicit condition, and body), and the top-level interface (divided for methods and their implicit conditions). For submodules, the hierarchical instance name is provided in parentheses. Example:

```
[timestamp] elab progress: Elaborating module ‘mkGCD’
[timestamp] elab progress: (the_x) Elaborating module
[timestamp] elab progress: (the_x) Finished module
[timestamp] elab progress: (the_y) Elaborating module
[timestamp] elab progress: (the_y) Finished module
[timestamp] elab progress: Elaborating rule (flip)
[timestamp] elab progress: Elaborating rule explicit condition (flip)
[timestamp] elab progress: Elaborating rule body (flip)
[timestamp] elab progress: Elaborating rule implicit condition (flip)
[timestamp] elab progress: Finished rule (flip)
[timestamp] elab progress: Elaborating rule (sub)
[timestamp] elab progress: Elaborating rule explicit condition (sub)
[timestamp] elab progress: Elaborating rule body (sub)
[timestamp] elab progress: Elaborating rule implicit condition (sub)
[timestamp] elab progress: Finished rule (sub)
[timestamp] elab progress: Elaborating interface
[timestamp] elab progress: Elaborating method ‘start’
[timestamp] elab progress: Elaborating method implicit condition
[timestamp] elab progress: Elaborating method ‘result’
[timestamp] elab progress: Elaborating method implicit condition
[timestamp] elab progress: Finished elaborating module ‘mkGCD’
```

4.1.3 Scheduling Messages

Static execution order When multiple rules execute in the same cycle, they must execute in a sequence, with each rule completing its state update before the next rule begins. In order to simplify the muxing logic, BSC chooses one execution order which is used in every clock cycle. If rule `A` and rule `B` can be executed in the same cycle, then they will always execute in the same order. The hardware does not dynamically choose to execute them in the order "`A` before `B`" in one cycle and "`B` before `A`" in a later cycle.

There may be times when three or more rules cannot execute in the same cycle, even though any two of the rules can. Consider three rules `A`, `B`, and `C`, where `A` can be sequenced before `B` but not after, `B` can only be sequenced before `C`, and `C` can only be sequenced after `A`. For any two rules, there is an order in which they may be executed in the same cycle. But there is no order for all three. If the conditions of all three rules are satisfied, the scheduler cannot execute all of them. It must make a decision to execute only two – for example, only `A` and `B`. But notice that this is not all. The scheduler must pick an order for all three rules, say "`A` before `B` before `C`." That means that not only will `C` not fire when both `A` and `B` are chosen to execute, but also that `C` can never fire when `A` is executed. This is because the compiler has chosen a static order, with `A` before `C`, which prevents rule `C` from ever executing before rule `A`. Effectively, the compiler has created a conflict between rules `A` and `C`.

If the compiler must introduce such a conflict, in order to create a static execution order, it will output a warning:

```
Warning: "Test.bsv", line 30, column 0: (G0009)
  The scheduling phase created a conflict between the following rules:
      ‘RL_One’ and ‘RL_Two’
  to break the following cycle:
      ‘RL_One’ -> ‘RL_Two’ -> ‘RL_Three’ -> ‘RL_One’
```

Rule urgency The execution order of rules specifies the order in which chosen rules will appear to execute within a clock cycle. It does not say anything about the order in which rules are chosen. The scheduling phase of the compiler chooses a set of rules to execute, and then that set is executed in the order specified by the static execution order. The order in which the scheduling phase chooses rules to put into that set can be different from the execution order. The scheduling phase may first consider whether to include rule `B` before considering whether to include rule `A`, even if rule `A` will execute first. This order of consideration by the scheduling phase is called the *urgency order*.

If rule `A` and `B` conflict and cannot be executed in the same cycle, but can be ready in the same cycle, then the first one chosen by the scheduler will be the one to execute. If rule `B` is chosen before rule `A` then we say that `B` is *more urgent than* `A`.

If two rules conflict and the user has not specified which rule should be more urgent, the compiler will make its own (arbitrary) choice and will warn the user that it has done so, with the following warning:

```
Warning: "Test.bsv", line 24, column 0: (G0010)
  Rule "one" was treated as more urgent than "two". Conflicts:
    "one" cannot fire before "two": calls to x.write vs. x.read
    "two" cannot fire before "one": calls to y.write vs. y.read
```

As you can see, this warning also includes details about how the compiler determined that the rules conflict. This is because an unexpected urgency warning could be due to a conflict that the user didn't expect.

If the conflict is legitimate, the user can avoid this warning by specifying the urgency order between the rules (and thus not leave it up to the vagaries of the compiler). The user can specify the urgency with the `descending_urgency` attribute. See the BSV Reference Guide for more information on scheduling attributes.

Note that methods of generated modules are treated as more urgent than internal rules, unless a wire requires that the rule be more urgent than the method.

Urgency between two rules can also be implied by a data dependency between the more urgent rule's action and the less urgent rule's condition. This is because the first rule must execute before the

scheduler can know whether the second rule is ready. See Section 4.1.4 for more information on how such paths are created.

If a contradiction is created, between the user-supplied attributes, the path-implied urgency relationships, and/or the assumed relationship between methods and rules, then an error is reported, as follows:

```
Error: "Test.bsv", line 8, column 8: (G0030)
  A cycle was detected in the urgency requirements for this module:
    ‘bar’ -> ‘RL_foo’
  The relationships were introduced for the following reasons:
    (bar, RL_foo) introduced because of method/rule requirement
    (RL_foo, bar) introduced because of the following data dependency:
      [WillFire signal of rule/method ‘RL_foo’,
       Enable signal of method ‘wset’ of submodule ‘the_rw’,
       Return value of method ‘whas’ of submodule ‘the_rw’,
       Output of top-level method ‘RDY_bar’,
       Enable signal of top-level method ‘bar’,
       CanFire signal of rule/method ‘bar’]
```

4.1.4 Path Messages

Some state elements, such as `RWire`, allow reading of values which were written in the same cycle. These elements can be used to avoid the latency of communicating through registers. However, they should only be used to communicate from a rule earlier in the execution sequence to a rule later in the sequence. Other uses are invalid and are detected by the compiler.

For example, if a value read in a rule's condition depends on the writing of that value in the rule's action, then you have an invalid situation where the choosing of a rule to fire depends on whether that rule has fired! In such cases, the compiler will produce the following error:

```
Error: "Test.bsv", line 20, column 10: (G0033)
  The condition of rule ‘RL_flip’ depends on the firing of that rule. This is
  due to the following path from the rule’s WILL_FIRE to its CAN_FIRE:
    [WillFire signal of rule/method ‘RL_flip’,
     Control mux for arguments of method ‘wset’ of submodule ‘the_x’,
     Argument 1 of method ‘wset’ of submodule ‘the_x’,
     Return value of method ‘wget’ of submodule ‘the_x’,
     CanFire signal of rule/method ‘RL_flip’]
```

Similarly, if the ready signal of a method has been defined as dependent on the enable signal of that same method, then an invalid situation has been created, and the compiler will produce an error (`G0035`). The ready signal of a method must also be computed prior to knowing the inputs to the method. If the ready signal of a method depends on the values of the arguments to that method, an error message will be reported (`G0034`).

A combinational cycle can result if a bypass primitive is used entirely within a single rule's action. In such cases, the compiler will produce an error explaining the source objects involved in the combinational path, in data-dependency order:

```
Error: "Test.bsv", line 4, column 8: (G0032)
  A cycle was detected in the design prior to scheduling. It is likely that
  an action in this module uses circular logic. The cycle is through the
  following:
    [Argument 1 of method ‘wset’ of submodule ‘the_rw’,
     Return value of method ‘wget’ of submodule ‘the_rw’]
```

4.2 Other messages

BSC can also emit status messages during the course of compilation.

4.2.1 Compilation progress

When the compiler finishes generating Verilog code for a module, it will output the location of the file which it generated:

```
Verilog file created: mkGCD.v
```

The following message is output when elaborating a design for Bluesim:

```
Elaborated Bluesim module file created: mkGCD.ba
```

When an elaboration file is generated to a Bluesim object, the following message is given:

```
Bluesim object created: mkGCD.{h,o}
```

If previously generated Bluesim object files still exist, are newer than the **.ba** file from which they were generated, and the module does not instantiate any modified submodules, then the existing object files will be reused. In this case the following message is seen instead of the message above:

```
Bluesim object reused: mkGCD.{h,o}
```

When the Bluesim object is linked to create a simulation binary, the following message is given:

```
Simulation shared library created: mkGCD.so
Simulation executable created: mkGCD
```

When using serial linking for Bluesim (`-parallel-sim-link 1`) (Section 3.4), the `-v` flag displays the `exec` command for each module and the resulting messages in serial order:

```
exec: c++ ... mkTbGCD.cxx
Bluesim object created: mkTbGCD.{h,o}
exec: c++ ... model_mkTbGCD.cxx
Bluesim object created: model_mkTbGCD.{h,o}
```

When parallel linking is turned on (`-parallel-sim-link > 1`), a makefile is used to process the C/C++ compilations. If the `-v` flag is set, you will see the command for the `make` execution called to execute the parallel process:

```
exec: make -f compile_mkTbGCD.mk
```

Since the compilations are occuring in parallel, the messages will be interspersed with each other and will not show up in serial order:

```
exec: c++ ... mkTbGCD.cxx
exec: c++ ... model_mkTbGCD.cxx
Bluesim object created: model_mkTbGCD.{h,o}
Bluesim object created: mkTbGCD.{h,o}
```

Automatic recompilation As described in Section 3.9, the `-u` flag can be used to check dependencies and recompile any needed packages which have been updated since their last compilation. The `-show-compiles` flag, which is *on* by default, will have the compiler output messages about the dependent files which need recompiling, as follows:

```
checking package dependencies
compiling ./FindFIFO2.bsv
compiling ./FiveStageCPUStall.bsv
compiling CPUTest.bsv
code generation for mkCPUTest starts
packages up to date
```

Verbose output The `-v` flag causes the compiler to output much progress information. First, the version of BSC is displayed. Then, as each phase of compilation is entered, a `starting` message is displayed. When the phase is completed, a `done` message is displayed along with the time spent in that phase. During the `import` phase, the compiler lists all of the header files which were read, including the full path to the files. During the `binary` phase, the compiler lists all of the binary files which were read. Prior to code generation, all of the modules to be compiled in the current package are listed:

```
modules: [mkFiveStageCPUStall_]
```

Then, code generation is performed for each module, in the order listed. Each is prefaced by a divider and the name of the module being generated:

```
*****
code generation for mkFiveStageCPUStall starts
```

After all modules have been generated, the binary (`.bo`) files are output for the package with the following message:

```
Generate interface files
```

Finally, the total elapsed time of compilation is displayed.

Whenever the C or C++ compiler is invoked from bsc (such as during Bluesim compilation or when compiling or linking foreign C functions), the executed command is displayed:

```
exec: c++ -Wall -Wno-unused -O3 -fno-rtti -g -D_FILE_OFFSET_BITS=64
-I/tools/bsc/lib/Bluesim -c -o mkGCD.o mkGCD.cxx
```

4.2.2 Scheduling information

There are two flags which can be used to dump the schedule generated by the compiler and the information which led to that schedule: `-show-schedule` and `-show-rule-rel`.

The `-show-schedule` flag outputs three groups of information: method scheduling information (if the module has methods), rule scheduling information, and the linear execution order of rules and methods (see the paragraph on static execution order in Section 4.1.3). The output is in a file *modulename*`.sched` in the directory specified by the `info-dir` (Section 3.6) flag.

For each method, the following information is given: the method's name, the expression for the method's ready signal (`1` if it is always ready), and a list of conflict relationships with other methods. Any methods which can execute in the same clock cycle as the current method, in any execution order, are listed as "conflict-free." Any methods which can execute in the same clock cycle but only in a specific order are labelled either "sequenced before" (if the current method must execute first) or "sequenced after" (if the current method must execute second). Any methods which cannot be called in the same clock cycle as this method are listed as "conflicts." The following is an example entry:

```
Method: imem_get
Ready signal: True
Conflict-free: dmem_get, dmem_put, start, done
Sequenced before: imem_put
Conflicts: imem_get
```

For each rule, the following information is given: the rule's name, the expression for the rule's ready signal, and a list of more urgent rules which can block the execution of this rule. The more urgent rules conflict with the current rule and, if chosen to execute, they will prevent the current rule from executing in the same clock cycle (see the paragraph on rule urgency in Section 4.1.3). The following is an example entry:

```
Rule: fetch
Predicate: the_bf.i_notFull_ && the_started.get
Blocking rules: imem_put, start
```

The `-show-schedule` flag will inform you that a rule is blocked by a conflicting rule, but won't show you why the rules conflict. It will show you that one rule was sequenced before another rule, but it won't tell you whether the other order was not possible due to a conflict. For conflict information, you need to use the `-show-rule-rel` flag.

The `-show-rule-rel` flag can be used, during code generation, to query the compiler about the conflict relationship between two rules. Since this requires re-running the compiler, it is most useful to give the wildcard arguments `\* \*` and dump all rule relationships in one compile.

```
-show-rule-rel \* \*
```

If you only want to see the conflict relationships for a single rule, you can use:

```
-show-rule-rel \* rulename2
```

which will output all the rule relationships for `rulename2`. No other uses of the wildcard argument `\*` are valid with this flag.

The following is an example entry in the `-show-rule-rel` output:

```
Scheduling info for rules "RL_execute_jz_taken" and "RL_fetch":
predicates are not disjoint
    <>
    conflict:
    calls to
      the_pc.set vs. the_pc.get
      the_bf.clear_ vs. the_bf.i_notFull_
      the_pc.set vs. the_pc.set
      the_bf.clear_ vs. the_bf.enq_
    <
    conflict:
    calls to
      the_pc.set vs. the_pc.get
      the_bf.clear_ vs. the_bf.i_notFull_
      the_bf.clear_ vs. the_bf.enq_
    no resource conflict
    no cycle conflict
    no attribute conflict
```

For the two rules given, several pieces of information are provided. If the compiler can determine that the predicates of the two rules are mutually exclusive, then the two rules can never be ready in the same cycle and therefore we need never worry about whether the actions can be executed in the same clock cycle. In the above example, the predicates could not be determined to be disjoint, so conflict information was computed.

Two rules have a `<>`-type conflict if they use a pair of methods which are not conflict free. The rules either cannot be executed in the same clock cycle or they can but one must be sequenced first. The compiler lists the methods used in each rule which are the source of the conflict.

Two rules have a `<`-type conflict if the first rule mentioned cannot be executed in sequence before the second rule, because they use methods which cannot sequence in that order. There is no entry for `>`-type conflicts; for that information, look for an entry for the two rules in the opposite order and consult the `<`-type conflict. Again, the compiler lists the methods used in each rule which are the source of the conflict.

If a conflict was introduced between two rules because of resource arbitration (see Section 3.5), that information will be displayed third. The fourth line indicates whether a conflict was introduced to break an execution order cycle (see Section 4.1.3). The fifth, and last, line indicates whether a conflict was introduced by a scheduling attribute or operator in the design, such as the `preempts` attribute (see the BSV Reference Guide for more information on pre-emption).

5 Verilog back end

The Verilog code produced by BSC can either be executed using standard Verilog execution/interpretation tools or it can be compiled into netlists using standard synthesis tools. The generated code uses BSC-provided modules such as registers and FIFOs; these can be found in `$BLUESPECDIR/Verilog`. These modules must be used for simulation or synthesis, though creating a simulator with `bsc -e` automatically includes them. For example, to run the `vcs` simulator, use the following command:

```
bsc -vsim vcs -e mkToplevel mkToplevel.v otherfiles.v
```

See Section 2.5.3 for details on choosing the Verilog simulator.

The `$BLUESPECDIR/Verilog` directory also contains the file `Bluespec.xcf`, a Xilinx XCF constraint file, to be used when synthesizing with Xilinx.

5.1 Bluespec to Verilog mapping

To aid in the understanding and debugging of the generated Verilog code, this section describes the general structure and name transformations that occur in mapping the original BSV source code into Verilog RTL. The section is based on a single example, which implements a greatest common denominator (GCD) algorithm. The source BSV code for the example is shown in Figure 2. The generated Verilog RTL is shown in Figures 3 and 4.

5.1.1 Interfaces and Ports

The interface section of a BSV design is used to specify the ports (input and outputs) of the generated Verilog code. The BSV interface specification for the GCD example is repeated below.

```
interface ArithIO_IFC #(parameter type aTyp); // aTyp is a parameterized type
    method Action start(aTyp num1, aTyp num2);
    method aTyp result();
endinterface: ArithIO_IFC
```

This interface specification leads to the following Verilog port specification. In the BSV specification shown in Figure 2, the type parameter `aTyp` has been bound to be a 51-bit integer.

```
module mkGCD(CLK,                  // input   1 bit  (implicit)
             RST_N,                // input   1 bit  (implicit)
             start_num1,           // input  51 bits (explicit)
             start_num2,           // input  51 bits (explicit)
             EN_start,             // input   1 bit  (implicit)
             RDY_start,            // output  1 bit  (implicit)
             result,               // output 51 bits (explicit)
             RDY_result            // output  1 bit  (implicit)
             );
```

Note that the generated Verilog includes a number of ports in addition to the `num1`, `num2`, and `result` signals that are specified explicitly in the interface definition. More specifically, each BSV interface has implicit clock and reset signals, whereas the generated Verilog includes these signals as `CLK` and `RST_N`. In addition, both the `start` and `result` methods have associated implicit signals.

The Verilog implementation of the `start` method (an input method) includes input signals for the arguments `num1` and `num2` which were explicitly declared in the BSV interface. The Verilog port names corresponding to these inputs have the method name prepended, however, to avoid duplicate port names. They have the generated names `start_num1` and `start_num2`. In addition to these explicit signals, there are also the implicit signals `EN_start`, a 1-bit input signal, and `RDY_start`, a 1-bit output signal.

Similarly, the Verilog implementation of the `result` method (an output method) includes the `result` output signal specified by the BSV interface, as well as the implicit signal `RDY_start`, a 1-bit output signal.

By default, the sense of the generated reset signal is asserted low. To generate a positive reset (asserted high), use the Verilog macro `BSV_POSITIVE_RESET`, described in Section 2.5.3. This is a global switch; it is not possible to generate mixed resets within a single design.

The default name of the reset for generated modules is `RST_N`, regardless of the sense of the reset. The name may be changed with the bsc flag `-reset-prefix <name>`, which assigns the new name to all synthesized modules. When specified in the link stage the `-reset-prefix` flag causes the Verilog

linker to define the macro `BSV_RESET_NAME` which used by the file `main.v` to properly connect to the top module. When generating a Verilog file, the `-reset-prefix` flag should be used in both the compile and the link stages.

To generate positive resets, the following flags are recommended:

```
-reset-prefix RST_P -D BSV_POSITIVE_RESET
```

where `RST_P` can be replaced by any name you choose for the positive reset. The `-D` flag is only required for the link stage, but you can use the same set of flags for both compile and link.

Since the implicit signal names are generated automatically by BSC, the BSV syntax provides a way in which the user can control the naming of these signals using attributes specified in the BSV source code. To rename the generated clock and reset signals, the following syntax is used:

```
(* osc="clk" *)
(* reset="rst" *)
```

There are also attributes to define a prefix string to be added to all clock oscillators, clock gates, and resets in a module:

More information on clock and reset naming attributes is available in the Bluespec Reference Guide.

The user may remove *Ready* signals by adding the attribute `always_ready` to the method definition. Similarly, the user may remove *enable* signals by adding the attribute `always_enabled` to the method definition. The syntax for this is shown below.

```
(* always_ready, always_enabled *)
```

More information on interface attributes is available in the BSV Reference Guide.

In addition to the *provided* interface, a BSV module declaration may include parameters and arguments (such as clocks, resets, and *used* interfaces). When such a module is synthesized, these inputs become input ports in the generated Verilog. If a Verilog parameter is preferred, the designer can specify this by using the optional `parameter` keyword. For example, consider the following module:

```
module mkMod #(parameter Bit#(8) chipId, Bit#(8) busId) (IfcType);
```

This module has two instantiation parameters, but only one is marked to be generated as a parameter in Verilog. This BSV module would synthesize to a Verilog module with parameter `chipId` and input port `busId` in addition to the ports for the interface `IfcType`:

```
module mkMod(busId,
             ...);
  parameter chipId = 0;
  input  [7 : 0] busId;
  ...
```

Parameters generated in this way have a default value of 0.

5.1.2 State elements

State elements, synthesized from `mkFIFO` and the like, are instantiated as appropriate elements in the generated Verilog. For example, consider the following BSV code fragment:

```
FIFO #(NumTyp) queue1 <- mkFIFO;   // queue1 is the FIFO instance
```

The above fragment produces the following Verilog instantiation:

```
FIFO2 #(.width(51)) queue1(.CLK(CLK),
                           .RST(RST_N),
                           .D_IN(queue1$D_IN),
                           .ENQ(queue1$ENQ),
                           .DEQ(queue1$DEQ),
                           .D_OUT(queue1$D_OUT),
                           .CLR(queue1$CLR),
                           .FULL_N(queue1$FULL_N),
                           .EMPTY_N(queue1$EMPTY_N));
```

Note that the Verilog instance name matches the instance name used in the BSV source code. Similarly, the associated signal names are constructed as a concatenation of the Verilog instance name and the Verilog port names.

Registers instantiated with `mkReg`, `mkRegU`, and `mkRegA` are treated specially. Rather than declare a bulky module instantiation, they are declared as Verilog `reg` signals and the contents of the module (for setting and initializing the register) are inlined. For example, consider the following BSV code fragment:

```
Reg #(NumTyp) x(); // x is the interface to the register
mkReg reg_1(x);   // reg_1 is the register instance

Reg #(NumTyp) y();
mkRegU reg_2(y);

Reg #(NumTyp) z();
mkRegA reg_3(z);
```

Which generates the following Verilog instantiation:

```
reg [50 : 0] reg_1, reg_2, reg_3;
wire [50 : 0] reg_1$D_IN, reg_2$D_IN, reg_3$D_IN;
wire reg_1$EN, reg_2$EN, reg_3$EN;

always@(posedge CLK)
begin
  if (!RST_N)
    reg_1 <= `BSV_ASSIGNMENT_DELAY 51'd0;
  else
    if (reg_1$EN) reg_1 <= reg_1$D_IN;
  if (reg_2$EN) reg_2 <= reg_2$D_IN;
end

always@(posedge CLK , negedge RST_N)
if (!RST_N)
  reg_3 <= `BSV_ASSIGNMENT_DELAY 51'd0;
else
  if (reg_3$EN) reg_3 <= `BSV_ASSIGNMENT_DELAY reg_3$D_IN;

`ifdef BSV_NO_INITIAL_BLOCKS
`else // no BSV_NO_INITIAL_BLOCKS
// synopsys translate_off
initial
begin
  reg_1 = 51'h2AAAAAAAAAAAA;
  reg_2 = 51'h2AAAAAAAAAAAA;
  reg_3 = 51'h2AAAAAAAAAAAA;
end
// synopsys translate_on
`endif // BSV_NO_INITIAL_BLOCKS
```

Register assignments are guarded by the macro `BSV_ASSIGNMENT_DELAY`, defined to be empty by default. In simulation, delaying assignment to registers and other state elements with respect to the relevant clock may be effected by defining `BSV_ASSIGNMENT_DELAY` (generally to "`#0`" or "`#1`") in the Verilog simulator.[1]

All registers are initialized with the distinguishable hex value `A` in order to guarantee consistent simulation in both Verilog and Bluesim, in the presence of multiple clocks and resets. This initialization is guarded by the macro `BSV_NO_INITIAL_BLOCKS`, which, if defined in the Verilog simulator or synthesis tool, disables the `initial` blocks.

The bsc command line option `-remove-unused-modules` can be used to remove primitives and modules which do not impact any output port. This option should only be used on synthesized modules, and not on testbenches.

5.1.3 Rules and related signals

For each instantiated rule, two combinational signals are created:

- `CAN_FIRE_rulelabel`: This signal indicates that the preconditions for the associated rule have been satisfied and the rule can fire at the next clock edge. The rule may not fire (execute) because the scheduler has assigned a higher priority to another rule and simultaneous rule firing causes resource conflicts.

[1] While the creative possibilities this feature opens—such as defining `BSV_ASSIGNMENT_DELAY` to "`~`"—may seem tempting at times, we discourage uses for purposes other than delaying assignment with respect to the clock edge.

```
typedef UInt#(51) NumTyp;

interface ArithIO_IFC #(parameter type aTyp); // aTyp is a paramerized type
    method Action start(aTyp num1, aTyp num2);
    method aTyp result();
endinterface: ArithIO_IFC

// The following is an attribute that tells the compiler to generate
// separate code for mkGCD
(* synthesize *)
module mkGCD(ArithIO_IFC#(NumTyp)); // here aTyp is defined to be type Int

    Reg#(NumTyp) x(); // x is the interface to the register
    mkRegU reg_1(x);  // reg_1 is the register instance

    Reg #(NumTyp) y(); // y is the interface to the register
    mkRegU reg_2(y);   // reg_2 is the register instance

    rule flip (x > y && y != 0);
        x <= y;
        y <= x;
    endrule

    rule sub (x <= y && y != 0);
        y <= y - x;
    endrule

    method Action start(NumTyp num1, NumTyp num2) if (y == 0);
        action
            x <= num1;
            y <= num2;
        endaction
    endmethod: start

    method NumTyp result() if (y == 0);
        result = x;
    endmethod: result

endmodule: mkGCD
```

Figure 2: BSV Source Code For The GCD Example

- `WILL_FIRE_rulelabel`: This signal indicates that the rule will fire at the next clock edge. That is, its preconditions have been met, and the scheduler has determined that no resource conflicts will occur. Multiple rules can fire during one cycle provided that there are no resource conflicts between the rules.

The `rulelabel` substring includes an unmangled version of the source rule name as well as a `RL_` prefix and optionally a `_<n>` suffix. This suffix appears when it is needed to create a unique name from the instances from different submodules.

5.1.4 Other signals

Signals beginning with an underscore (_) character are internal combinational signals generated during elaboration and synthesis. These should not be used during debug.

```
`ifdef BSV_ASSIGNMENT_DELAY
`else
`define BSV_ASSIGNMENT_DELAY
`endif

module mkGCD(CLK,
     RST_N,

     start_num1,
     start_num2,
     EN_start,
     RDY_start,

     result,
     RDY_result);
  input  CLK;
  input  RST_N;

  // action method start
  input  [50 : 0] start_num1;
  input  [50 : 0] start_num2;
  input  EN_start;
  output RDY_start;

  // value method result
  output [50 : 0] result;
  output RDY_result;

  // signals for module outputs
  wire [50 : 0] result;
  wire RDY_result, RDY_start;

  // register reg_1
  reg [50 : 0] reg_1;
  wire [50 : 0] reg_1$D_IN;
  wire reg_1$EN;

  // register reg_2
  reg [50 : 0] reg_2;
  reg [50 : 0] reg_2$D_IN;
  wire reg_2$EN;

  // rule scheduling signals
  wire WILL_FIRE_RL_flip, WILL_FIRE_RL_sub;

  // inputs to muxes for submodule ports
  wire [50 : 0] MUX_reg_2$write_1__VAL_3;

  // remaining internal signals
  wire reg_1_ULE_reg_2___d3;
```

Figure 3: Generated Verilog GCD Example (part 1)

```
 // action method start
  assign RDY_start = reg_2 == 51'd0 ;

  // value method result
  assign result = reg_1 ;
  assign RDY_result = reg_2 == 51'd0 ;

  // rule RL_flip
  assign WILL_FIRE_RL_flip = !reg_1_ULE_reg_2___d3 && reg_2 != 51'd0 ;

  // rule RL_sub
  assign WILL_FIRE_RL_sub = reg_1_ULE_reg_2___d3 && reg_2 != 51'd0 ;

  // inputs to muxes for submodule ports
  assign MUX_reg_2$write_1__VAL_3 = reg_2 - reg_1 ;

  // register reg_1
  assign reg_1$D_IN = EN_start ? start_num1 : reg_2 ;
  assign reg_1$EN = EN_start || WILL_FIRE_RL_flip ;

  // register reg_2
  always@(EN_start or
  start_num2 or
  WILL_FIRE_RL_flip or
  reg_1 or WILL_FIRE_RL_sub or MUX_reg_2$write_1__VAL_3)
  begin
    case (1'b1) // synopsys parallel_case
      EN_start: reg_2$D_IN = start_num2;
      WILL_FIRE_RL_flip: reg_2$D_IN = reg_1;
      WILL_FIRE_RL_sub: reg_2$D_IN = MUX_reg_2$write_1__VAL_3;
      default: reg_2$D_IN = 51'h2AAAAAAAAAAA /* unspecified value */ ;
    endcase
  end
  assign reg_2$EN = EN_start || WILL_FIRE_RL_flip || WILL_FIRE_RL_sub ;

  // remaining internal signals
  assign reg_1_ULE_reg_2___d3 = reg_1 <= reg_2 ;

  // handling of inlined registers

  always@(posedge CLK)
  begin
    if (reg_1$EN) reg_1 <= `BSV_ASSIGNMENT_DELAY reg_1$D_IN;
    if (reg_2$EN) reg_2 <= `BSV_ASSIGNMENT_DELAY reg_2$D_IN;
  end

  // synopsys translate_off
  `ifdef BSV_NO_INITIAL_BLOCKS
  `else // not BSV_NO_INITIAL_BLOCKS
  initial
  begin
    reg_1 = 51'h2AAAAAAAAAAA;
    reg_2 = 51'h2AAAAAAAAAAA;
  end
  `endif // BSV_NO_INITIAL_BLOCKS
  // synopsys translate_on
endmodule  // mkGCD
```

Figure 4: Generated Verilog GCD Example (part 2)

```
// Generated by Bluespec Compiler, version 2013.12.beta1 (build 32830, 2013-12-02)
//
// On Mon Dec 16 12:42:09 EST 2013
//
//
// Ports:
// Name                         I/O  size props
// RDY_start                      O     1
// result                         O    51 reg
// RDY_result                     O     1
// CLK                            I     1 clock
// RST_N                          I     1 reset
// start_num1                     I    51
// start_num2                     I    51
// EN_start                       I     1
//
// No combinational paths from inputs to outputs
//
```

Figure 5: Generated Verilog Header For The GCD Example

5.2 Verilog header comment

When BSC generates the Verilog file for a module, it includes a comment with information about the compile and the module's interface. The header for the GCD example is shown in Figure 5. This comment would appear at the top of the file `mkGCD.v`.

The header begins with information about the version of the BSC software which was used to generate the file and the date of compilation. The subsequent information relates to the module's interface and its Verilog properties.

The method conflict information, shown in Figure 6, documents the scheduling constraints on the methods. These are Bluespec semantics which must be respected when using the module. They are the same details which the user must provide when importing his own Verilog module (see `import "BVI"` in the BSV Reference Guide). This information is included or omitted based on the `-show-method-conf` flag and the `-show-method-bvi` flag , discussed in Section 3.12. These flags are *off* by default. The format of the information is similar to the method output of the `-show-schedule` flag (see Section 4.2.2).

The port information provides RTL-level information about the Verilog design. There is an entry for each port which specifies whether the port is an input or an output, the port's size in bits, and any properties of the port. The possible properties are `reg`, `const`, `unused`, `clock`, `clock gate`, and `reset`. The `reg` property indicates that there is no logic between the port and a register – if the port is an input then the value is immediately registered, and if the port is an output than the value comes directly from a register. The `const` property indicates that the value of the port never changes, it is constant. Ports with the `unused` property are not connected to any state element or other port, and so their values are unused. The `clock`, `clock gate`, and `reset` properties indicate that the port is a clock oscillator, clock gate, and reset port, respectively.

The final information in the comment is a list of any combinational paths from inputs to outputs. If there is an unregistered path from an input port `write_val` (corresponding to the `val` argument of method `write`) to an output port `read`, it will appear as follows:

```
// Combinational paths from inputs to outputs:
//   write_val -> read
```

```
// Generated by Bluespec Compiler, version 2013.12.beta1 (build 32830, 2013-12-02)
//
// On Mon Dec 16 12:40:14 EST 2013
//
// Method conflict info:
// Method: start
// Sequenced after: result
// Conflicts: start
//
// Method: result
// Conflict-free: result
// Sequenced before: start
//
// BVI format method schedule info:
// schedule start  C ( start );
//
// schedule result  CF ( result );
// schedule result  SB ( start );
...
```

Figure 6: Generated Verilog Header with method conflict information flags

Multiple inputs which have a combinational path to one output are grouped together, for brevity, as follows:

```
// Combinational paths from inputs to outputs:
//   (add_x,  add_y,  add_z) -> add
```

This situation arises often for read methods with arguments. Multiple outputs are not grouped; there is only ever one output listed on the right-hand side.

6 Bluesim back end

Bluesim is a cycle simulator for generated BSV designs. It is cycle-accurate with the Verilog generated for the same designs. Bluesim can output VCD files for a simulation and offers other debugging capabilities as described below.

6.1 Bluesim tool flow

When a BSV design is compiled and linked using the Bluesim back end, the compiler links the design with a driver, to produce a stand-alone executable. When the executable is invoked, the default driver "clocks" the circuit and executes it.

The Bluesim back-end compiles modules in a Bluespec design to C++ objects which contain the module data and temporaries and which define routines for executing the rules and methods of each module. In addition to the modules, scheduling routines are generated which coordinate the execution of rules throughout the entire design. Primitive modules, functions, and system tasks are implemented as elements of a library supplied with the compiler.

6.2 Cycle-accuracy between Bluesim and Verilog simulation

For all code compiled by bsc from BSV sources, one will observe identical cycle behavior whether executing in Bluesim or Verilog simulation (or even the SystemC plug-in). This is because all these back-ends go through identical scheduling in bsc. You should see exactly the same waveforms in VCD files generated from any of these kinds of simulations.

There is one situation you should be aware of where the waveforms seen in Bluesim and Verilog simulation may apparently be different, even though they are technically *equivalent*, and that is in situations where you use `"?"`. Note that this really means "don't care", i.e., any value is acceptable and all values are equivalent. Such a signal may indeed appear as different specific values in Bluesim and Verilog simulation, but the (perhaps initially surprising) difference is explained by remembering that for `"?"`, all values are equivalent.

The Standard Prelude section of the *Libraries Reference Guide* describes two Bool environment variables `genC` and `genVerilog` by which a BSV program can test whether it is being compiled for Bluesim or for Verilog, and can perform different static elaborations accordingly. Of course, the two different static elaborations may not have identical cycle behavior.

The following design practices or design errors can cause differences in the VCDs generated by Verilog and Bluesim simulations:

- If the `opt-undetermined-vals` option, as described in Section 3.11, is used the bsc compiler is free to choose different values for don't cares in the Verilog and the Bluesim simulations. For example, this could show up as a different value for the don't care bits of a `tagged Invalid` value of a `Maybe#(a)` type in a VCD waveform. This is not a bug, as the user has explicitly requested that the compiler optimize the don't care values for each backend by specifying the `opt-undetermined-vals` flag.
- Bluesim primitive implementations may differ internally from Verilog primitive implementations. Bluesim primitives are separate implementations from the Verilog implementations of the same primitives in the BSV libraries. In normal operation the Bluesim primitives and the Verilog primitives will behave the same. In error conditions, however, they may differ. For example, consider a `deq()` from an empty unguarded FIFO. The value returned is not guaranteed to be the same between Bluesim and Verilog when this error condition is encountered. Also signal names internal to the primitive which exist in Verilog may have no corresponding value in Bluesim. In most cases, the Bluesim waveforms attempt to reproduce the Verilog waveforms at the primitive ports or internal signals, although there are exceptions. In some cases the Bluesim VCD waveforms expose additional primitive state that is not visible in the Verilog waveforms.
- The `$random` system function may exhibit different behavior in Bluesim and Verilog simulation. In Bluesim, the implementation calls a `random()` function from the C library, whereas in Verilog simulation it uses the Verilog simulator's `$random` function.
- Bluesim simulation can differ from Verilog simulation in some situations in which design constraints are violated. For example, if a method is annotated as `always_enabled` but then is *not* always enabled, the behavior in Bluesim can differ from the behavior in Verilog when the method is not enabled. The compiler will provide a warning during compilation that it could not guarantee the method would always be enabled.
- There are some schedules which are acceptable for the Verilog back end but not for Bluesim, because they do not strictly follow the static scheduling discipline. This is not a simulation-time difference, but a difference that is caught at compile time. For Verilog the compiler will generate a warning but accept the design. For Bluesim the compiler will stop with an error message. Dynamic module arguments and `importBVI` have similar behaviors.

- There are some unsafe primitives that can violate atomic semantics and lead to simulation differences between Bluesim and Verilog. One example is `mkNullCrossingWire`, which is deprecated. The compiler will issue a warning when it is used in a design. Another example is `mkUnsafeRWire`. The `Unsafe` in the name indicates that you should be understand what you are doing when using it.

Bluesim behavior adheres closely to the atomic sematics of the BSV source langauage, while at times the Verilog simulator's behavior varies. The following differences can be observed in how the simulators behave:

- System tasks and imported C functions are called on the falling edge of the clock in Verilog simulation but on the rising edge in Bluesim. This can lead to differences in the order of `$display` output (particularly with muli-clock domain designs) and in the value of `$time()` between Bluesim and Verilog. If imported C functions are sensitive to ordering issues they may also show a difference. This may lead to differences in task calls or display output at the edges of simulation (startup, reset, and the end of simulation), where a task may be called in one simulator but not in the other.

  By TRS semantics the correct behavior is to execute all of the tasks and imported functions on the positive edge of the clock. However, the Verilog backend is forced to move these executions to the preceding negative edge of the clock to avoid complications with the Verilog simulation event ordering.

- Bluesim is a 2-state simulator, If using a 4-state Verilog simulator and X or Z values are introduced during simulation, these can show up in the VCD waveforms and affect the simulation behavior. Bluesim does not support X or Z values, so its behavior will differ.

- Even though Bluesim is a 2-state simulator, in some cases it will attempt to show X values in the VCD waveforms it produces. For instance, a wire which is not written during a cycle will take on an X value for that cycle in Bluesim VCD waveforms. It will not show up that way in the Verilog waveforms. This is an extra feature of Bluesim VCDs that make the waveforms easier to interpret.

- The Bluesim behavior adheres closely to the atomic semantics of the BSV source language, but the generated Verilog code's behavior is determined by the event-driven semantics of the Verilog simulator. This means that the Verilog simulator will propagate value changes independently, regardless of their relationship in the BSV semantics. For example, a change to a method argument value of a method that is not enabled will have no effect in Bluesim, but Verilog will propagate the value change through the circuit even though the associated enable signal is 0. The language design allows this to happen without affecting the operation of the design, although it is visible in the VCD waveforms. In some cases the use of `?` in the BSV code can extend the propagation path and make the changes in the VCD waveforms more numerous, even without the use of the `opt-undetermined-vals` flag.

- System task ordering between separately-synthesized modules can vary between Bluesim and Verilog. This is an artifact of undefined ordering in the Verilog execution model. The compiler will generate a warning whenever possible.

6.3 Bluesim simulation flags

The following flags can be given on the command line to the Bluesim simulation executable:

```
-c <commands> = execute commands given as an argument
-f <file>     = execute script from file
-h            = print help and exit
-m <N>        = execute for N cycles
-v            = print version information and exit
-V [<file>]   = dump waveforms to VCD file (default: dump.vcd)
+<arg>        = Verilog-style plus-arg
```

The `-c` flag provides one or more commands to be executed by the simulator (see Section 6.4 for command syntax).

The `-f` flag directs the simulator to execute commands from the given script file (see Section 6.4 for command syntax).

The `-h` flags directs the simulator to print a help message which describes the available flags.

The `-m` flag forces the simulation to stop after a certain number of cycles; the default behavior is to execute forever or until the `$finish` system task is executed.

The `-v` flag directs the simulator to print some version information related to the simulation model, including the compiler version used to create it and the time and date at which it was created.

The `-V` flag causes the simulator to dump waveforms to a VCD file. If a file name is provided the waveforms will be written the named file, otherwise the default file name "dump.vcd" will be used.

Arguments can be passed to the simulation model with `+<arg>`. These values can be tested by the BSV model via the `$test$plusargs` system task.

6.4 Interactive simulation

The simulator can be executed in an interactive or scripted mode. Commands and scripts can be given using the `-c` and `-f` flags. Alternatively, the simulation object can be loaded directly in Bluetcl using the `sim load` command.

Bluetcl extends a TCL shell with a `sim` command whose subcommands control all aspects of loading, executing and interacting with a Bluesim simulation object. For a list of Bluetcl `sim` subcommands, see the Bluetcl appendix B.4.2.

There are two ways to access these simulation commands, through scripting or interactively. When a model is compiled through the Bluesim backend, it generates a `.so` file containing the simulation object. It also generates an executable program that provides a convenient way to run and use the simulation object. Passing the `-c` or `-f` flags to the executable enables scripting the simulation.

To run the simulation interactively, the standard Bluetcl tool, described in Appendix B, should be used.

In addition to these actions accessible through the `sim` command, all of the normal functions of a TCL interpreter as well as additional BSC-specific extensions are available in Bluetcl and they can be freely intermixed.

Note that the TCL interpreter behaves differently when executing a script than when running interactively. In an interactive session, the TCL interpreter will print the value returned by each command (if any), but when executing a script output is only generated in response to an explicit output command (eg. `puts`).

Loading Bluesim

Before executing Bluesim commands interactively, the Bluesim package must be loaded into Bluetcl and the namespace must be imported. This is done automatically when using the `-c` and `-f` flags, but must be done manually when running interactively.

From Bluetcl, the following commands must be entered at the start of the interactive session, to load the Bluesim package and import the Bluesim namespace:

```
package require Bluesim
namespace import Bluesim::*
```

load and unload

Before working with a Bluesim simulation object, it must be loaded – this is done automatically when using the `-c` and `-f` flags but must be done manually when using Bluetcl directly. The full command to load a simulation object in Bluetcl is `sim load` followed by the name of the `.so` file and the name of the top module.

A simulation object can be unloaded using the `sim unload` command. Any active object is automatically unloaded when the simulator exits or before loading a new simulation object, so it is not normally necessary to manually perform a `sim unload`.

arg

The `sim arg` command allows a Verilog-style plusarg to be set interactively. The command `sim arg <string>` adds the supplied string to the end of the list of plusargs searched by the `$test$plusargs` system task. The "+" character should not be included in the string argument.

run, step, stop and sync

The `sim run` command runs the current simulation to completion.

The `sim runto` command runs the current simulation to the time given as its argument.

The `sim step` command advances the current simulation for a given number of cycles of the currently active clock domain.

The `sim nextedge` command advances the current simulation until the next edge in any clock domain. The currently active domain does not change, so a subsequent `sim step` command will still execute according to the active domain regardless of the clock edge to which a `sim nextedge` command advances.

By default, these commands will not return until the requested simulation activity is complete. However, `sim step`, `sim runto` and `sim run` can be instructed to return immediately by adding the keyword `async` to the end of the command sequence (eg. `sim step 100 async`). This will cause the command to return immediately so that additional commands can be processed while the simulation continues to run asynchronously.

There are two commands that synchronize with an asynchronously spawned simulation: `stop` and `sync`. The `stop` command will pause the simulation at the end of the currently executing simulation cycle. The `sync` command will wait for the simulation to complete normally before returning.

As examples of the behavior of the `run` and `step` simulation commands, assume that we have a simulation executable named "bsim". Then

```
bsim -c 'sim run'
```

is equivalent to just

```
bsim
```

and

```
bsim -c 'sim step 100'
```

is equivalent to using the `-m` flag

```
bsim -m 100
```

Note that when a model is loaded, simulation time is at 0 and no clock edges have occurred. Stepping 1 cycle from that point will advance past the first clock edge, and if the first rising edge of the active clock occurs at time 0 then the step command will move from before the edge at time 0 to after the edge at time 0.

time

The `sim time` command returns the current simulation time.

It could be used interactively within Bluetcl

```
% sim load bsim.so mkTop
% sim step 10
% sim time
90
```

or within a script

```
bsim -c 'sim step 10; puts [sim time]'
90
```

clock

The `sim clock` command provides information on the currently defined clocks and allows the user to change the active clock domain used by the `sim step` command.

With no argument, the `sim clock` command returns a list containing a clock description for each currently defined clock. Each clock description is itself a list of 10 different pieces of information about the clock domain:

- a unique number assigned to the clock domain
- a flag indicating if the clock is the currently active domain (1 indicates active, 0 indicates not active)
- the textual name of the clock domain
- the initial value of the clock (0 or 1)
- the delay before the first edge of the clock
- the duration of the low clock phase
- the duration of the high clock phase
- the number of elapsed cycles of this clock

- the current value of the clock signal (0 or 1)
- the time of the last edge of the clock

Here is sample output from a `sim clock` command for a design with 2 clock domains:

```
% sim clock
{0 1 CLK 0 0 5 5 12 1 110} {1 0 {mc$CLK_OUT} 0 0 0 0 3 0 100}
```

This output indicates that there are 2 domains. The first is domain number 0 and is the currently active clock. It is called "CLK" and is initially low, rises at time 0 and then alternates every five time units. At the current simulation time, 12 cycles have elapsed in the "CLK" clock domain and its current clock value is 1, after a rising edge at time 110. The second domain is number 1 and is not the currently active clock used for stepping. It is called "mc$CLK_OUT" and we have no timing information because it is not a periodic waveform (it is internally generated in the model). At the current simulation time, 3 cycles have elapsed in the "mc$CLK_OUT" domain and its current value is 0, after a falling edge at time 100.

To change the currently active clock, simply use the `sim clock <name>` form of the command, where the name argument specifies which clock to be made active. After executing this command, future `sim step` commands will step through cycles in the newly activated clock domain.

```
% sim clock {mc$CLK_OUT}
% sim clock
{0 0 CLK 0 0 5 5 12 1 110} {1 1 {mc$CLK_OUT} 0 0 0 0 3 0 100}
```

Note that the clock name argument was quoted in curly braces so that the TCL interpreter would treat the dollar-sign in the clock domain name as a literal dollar-sign.

ls, cd, up and pwd

Bluesim allows the user to navigate through the hierarchy of module instantiations using the `sim cd` and `sim up` commands.

To move down one or more levels of hierarchy, provide a path to the `sim cd` command. The path must consist of a sequence of instance names separated by '`.`'. A path which begins with `.` is considered to be an absolute path from the top of the hierarchy, but a path which does not begin with `.` is interpreted as a path relative to the current location.

The `sim up` command is used to move up the hierarchy into parents of the current directory. It can be given a numeric argument to control how many levels to ascend, or it can be used without an argument to move up one level.

As a special case, the `sim cd` command will return the user to the uppermost point in the hierarchy if used without a path argument.

To find your current location in the module hierarchy, use the `sim pwd` command.

At any point in the hierarchy, the `sim ls` command can be used to list the sub-instances, rules and values at that level of hierarchy. The command can be given any number of patterns to control which names are listed. If no argument is given, it is equivalent to `sim ls *`.

The patterns follow the standard syntax for filename globbing:

- ?: Matches any single character

- *: Matches any number of characters (possibly none)
- [...]: Matches any character inside of the brackets
- [a-z]: Matches any character in the specified range
- [!...]: Matches any character which does not match specification inside the brackets

The instance separator character '`.`' is never matched in a pattern. The special characters `?`,`*` and `[` can be escaped in a pattern using a backslash (\).

```
% sim pwd
.
% sim ls
{b__h380 signal} {CAN_FIRE_RL_done signal} {CAN_FIRE_RL_incr signal}
{count module} {level1 module} {mid1 module} {mid2 module} {RL_done rule}
{RL_incr rule} {WILL_FIRE_RL_done signal} {WILL_FIRE_RL_incr signal}
% sim ls level1.*
{level1.level2 module}
% sim cd level1.level2
% sim pwd
.level1.level2
% sim ls RL_*
{RL_incr rule} {RL_sub1_flip rule} {RL_wrap rule}
% sim up
% sim pwd
.level1
```

lookup, get and getrange

In addition to navigating through the instance hierarchy, Bluesim allows the user to examine the simulation values at run-time, using the `sim lookup`, `sim get` and `sim getrange` commands.

To get the value for a signal, you must first obtain a "handle" for the value using the `sim lookup` command. The command takes as an argument a pattern describing the absolute or relative path to the desired signal. A relative path is interpreted in the current directory unless an optional second argument is given containing the handle to a different starting directory. `sim lookup` will return a handle for every simulation object which matches the pattern argument.

Once the handle of a signal is known, its value can be obtained using the `sim get` command. This command takes one or more handles as arguments and returns the raw values associated with the handles, as sized hexadecimal numbers.

```
% set WF_incr [sim lookup .level1.level2.WILL_FIRE_RL_incr]
150533432
% sim get $WF_incr
1'h1
% sim ls .mid?.count
{mid1.count module} {mid2.count module}
% eval sim get [sim lookup .mid?.count]
4'h9 4'h1
```

The `sim getrange` command is a specialized command to get values for handles which represent multiple values, such the storage inside of a FIFO or register file. The command takes the handle for the value range object along with either a single address or a start and end address pair.

```
% sim getrange [sim lookup rf] 0 3
16'h0 16'h1 16'h2 16'h3
% sim getrange [sim lookup rf] 2
16'h2
% sim getrange [sim lookup fifo] 0
16'h8
```

Details about the simulation object referenced by a handle can be obtained using the `sim describe` command.

vcd

The `sim vcd` command controls dumping of waveforms to a VCD file. Use `sim vcd on` to enable dumping and `sim vcd off` to disable it. The form `sim vcd <file>` enables dumping to a file of the given name, rather than the default file named "`dump.vcd`".

version

The `sim version` command prints details about the tool version used to build the current simulation object. It returns a list of 5 TCL objects: the year of the release, the month of the release, the (optional) release tag, the revision number of the release, and the time at which the object was created (as a number of seconds since the start of 1970).

An example of using the `sim version` command to print the date and time at which a simulation object was created:

```
% puts [clock format [lindex [sim version] 4]]
Fri Dec 14 01:24:39 PM EST 2007
```

6.4.1 Command scripts for Bluesim

The Bluesim simulator can be run with a command script by using the `-c` or `-f` arguments.

```
./bluesim -f script.tcl
```

The contents of the script file can be standard TCL commands or any Bluetcl command extensions, including the `sim` commands. When used in this way, some aspects of Bluesim's behavior change to be more appropriate for executing scripts:

- No prompt is displayed.
- The result of each command is not printed. An explicit `puts` should be used to print command output in a script.
- Error messages include line numbers and stack traces.
- Errors and Ctrl-C end the simulation.

No `sim load` command is required when using a script, because the model will automatically be loaded before the script is executed and unloaded on exit.

No `exit` command is required when using a script, because the simulator will automatically exit when it reaches the end of the script file.

Comments can be included in the script file by starting a line with the TCL comment character `#`.

```
# Run 300 cycles
sim step 300

# Enable dumping VCD waveforms for the next 10 cycles
sim vcd on
sim step 10
```

6.5 Value change dump (VCD) output

The Bluesim simulator supports generation of a value change dump (VCD) to record the changes in user-selected state components. VCD files are an industry-standard way to record simulator state changes for use by external post-processing tools. For example, Novas Debussy, Undertow, and gtkWave are graphical waveform display programs that can be used to browse simulator state recorded in VCD files. FSDB files are currently not generated by Bluesim.

The Verilog system task `$dumpvars` may be used with no arguments to request VCD for all variables. Selective dumping with this task is not supported at this time. The Verilog system tasks `$dumpon` and `$dumpoff` can be used to turn VCD dumping on and off at specific times during simulation. Specifying `-V <file>` argument or using the `sim vcd <file>` command in a script will cause the simulator to output a VCD file of that name, in which the state of all registers and the internal signals of all BSV modules are dumped at the end of each cycle.

VCD files dumped by Bluesim attempt to match VCD files generated by Verilog simulation as closely as possible. Known differences between Bluesim- and Verilog-generated VCD files are documented in the *Known Problems and Solutions* (*KPNS*) document accompanying each release.

6.6 Bluesim multiple clock domain support

The Bluesim backend supports a subset of the multiple-clock-domain (MCD) features supported by the Verilog backend, including bit, pulse and word synchronizers as well as synchronized FIFOs. However, some MCD features supported in Verilog are not supported in Bluesim:

- mkNullCrossing

A Environment variables

You can customize your environment through the large number of environment variables set in your Unix shell.

All of these variables are option.

A.1 Installation

BLUESPECDIR	lib directory of Bluespec installation
BLUESPEC_HOME	home directory where Bluespec is installed
HOME	Unix home directory
SYSTEMC	directory in which SystemC is installed

The BLUESPECDIR variable points to the lib directory of the Bluespec installation. The variable BLUESPEC_HOME is an optional variable which points to the top level of the Bluespec installation.

The compiler needs to know where SystemC is installed so that it can compile C++ files that refer to SystemC files. The compiler uses the SYSTEMC variable to find standard SytemC files such as systemc.h and libsystemc.a. When the SYSTEMC variable is set, the compiler adds -I {$SYSTEMC}/include to the C++ compiler command line.

A.2 Options

BSC_OPTIONS	default options to BSC, Bluetcl, Bluewish, BDW
BLUETCL_OPTIONS	BSC options for Bluetcl, Bluewish, BDW
GHCRTS	sets run time flags -RTS and +RTS
BSV_VERILOG_SIM	specifies the Verilog simulator

The variable BSC_OPTIONS can be used to set any bsc flags documented in Section 3, except the RTS flags. To set the RTS flags described in Section 3.8, use the GHCRTS variable.

The variable BSV_VERILOG_SIM is another way to specify the Verilog simulator, providing the same function as the -vsim command-line flag. If the flag is given, its value is used. If the flag is not given, the compiler consults the environment variable. If the variable is not set, the compiler picks an available simulator.

A.3 C/C++ variables

CXX	specifies C++ compiler to use
CC	specifies C compiler to use
BSC_CXXFLAGS	Bluespec C++ compiler flags
BSC_CFLAGS	Bluespec C compiler flags
CFLAGS	C compiler flags
CXXFLAGS	C++ compiler flags

If either CXX or CC are not specified, the compiler will run the default C++ or C compiler, which must be found in the path.

The variables BSC_CXXFLAGS and BSC_CFLAGS are alternatives to using the -Xc++ and -Xc compiler flags (Section 3.14). BSC also uses the general C environment variables CFLAGS and CXXFLAGS.

A.4 Make variables

```
MAKE                    specifies name of make tool to use
BSC_MAKEFLAGS           Bluespec make flags
```

When the `-parallel-sim-link` flag (Section 3.4) is set to a value greater than 1, a makefile is generated, used, and then deleted for processing the parallel compiles for Bluesim. The compiler will look for `MAKE` in the environment to find the command to use. If `MAKE` is not defined, the command `make` is used. If the environment variable `BSC_MAKEFLAGS` exists, it is included on the command line.

B Bluetcl Reference

Bluetcl is a Tcl extension with a collection of scripts and packages providing an interface into the BSC view of a design. You can execute Bluetcl commands and scripts from a unix command line.

Bluetcl contains several layers (scripts, commands, packages) which should be familiar to the Tcl programmer. You can use Bluetcl extensions within Tcl scripts. More information on Tcl is available at `www.tcl.tk` or from the many books and references written about Tcl/Tk.

B.1 Invoking Bluetcl

Bluetcl commands can be run either interactively or through Tcl scripts. These commands load, execute, and interact with BSC-generated files. As with BSC, pre-elaboration information is obtained from the `.bo` files, and post-elaboration information is obtained from the `.ba` files.

Bluetcl commands can be invoked in the following ways:

- You can invoke Bluetcl from a unix prompt by typing `bluetcl`. This command provides a Tcl shell with the Bluetcl extensions.
- You can write and use Tcl scripts which use Bluetcl. For an example of a Tcl script provided with BSC, see Section B.5.1.

B.2 Packages and namespaces

Bluetcl is organized into a collection of packages, which are described in this appendix. The major packages are:

- The `Bluetcl` package which contains the low-level commands to interact with Bluespec files and designs.
- The `Bluesim` package containing Bluesim command extensions.[2]

All commands in the `Bluetcl` package are in the `Bluetcl` namespace. All commands in the `Bluesim` package are in the `Bluesim` namespace.

When referencing a command you must specify the namespace. Example:

```
Bluetcl::version
```

Alternately, you can import commands from a namespace. The following example imports all the commands in a namespace:

```
namespace import ::Bluetcl::*
```

Or you can import a single command:

```
namespace import ::Bluetcl::schedule
```

Refer to the `Tcl` documentation for additional information on packages and namespaces.

[2] The `sim` command for interacting with Bluesim simulation objects (`.so` files) is contained in both the `Bluetcl` and `Bluesim` packages.

B.3 Customizing Bluetcl

You can use Bluetcl, along with all standard Tcl constructs, to write scripts and issue commands. Bluetcl will source the setup file `$HOME/.bluetclrc` during initialization. You can customize Bluetcl by adding to the `.bluetclrc` file.

The `namespace import` command can be put in the `.bluetclrc` file, providing the command into the current namespace when the file is sourced. This will allow you to use just the command name in scripts or from the command line.

B.4 Package command reference

B.4.1 Conventions

The following conventions are used within the command reference:

name	identifier
keyword	as is
[...]	optional

When an argument (or token) contains embedded spaces, the argument may need to be enclosed in brackets ({ }).

B.4.2 Bluetcl

This sections describes the commands in the `Bluetcl` package. These commands provide a low-level interface to access BSC-specific files (`.bo`/`.ba`) for use by Tcl programmers; they are not intended for interactive use.

Before using a command from the `Bluetcl package`, the following Tcl command must be executed, either in a script, from the command line, or in the `.bluetclrc` file:

```
package require Bluetcl
```

All commands in the `Bluetcl` package are in the `Bluetcl::` namespace. The namespace must be referenced, as described in Section B.3, either by using the `namespace import` command or by prepending the command name with `Bluetcl::`. Example:

```
Bluetcl::bpackage list
```

Bluetcl::bpackage

Controls loading and unloading of packages and returns package information. When a package is loaded, all dependent (imported) packages are loaded as well.

bpackage load *packname* [*packname ...*]	Reads in the `.bo` package and all imported packages. Packages are searched in the standard bsc way, via the `-p` flag. Returns a list of all packages which are loaded. One or more package names can be provided.
bpackage list	Returns the list of packages which are loaded.
bpackage clear	Clear all currently loaded packages.
bpackage depend	Returns package dependencies of all currently loaded packages.
bpackage search *regex*	Searches packages for names matching a regular expression.
bpackage types *packname*	Returns a list of type names found in the package.

Bluetcl::defs

Returns a list of the components defined in a package. Components returned include types, synthesized modules, and functions.

defs all *packname*	Returns a list of all components which are defined in the package.
defs type *packname*	Returns a list of all types which are defined in the package.
defs module *packname*	Returns a list of all module names defined in the package which are marked synthesize.
defs func *packname*	Returns a tagged structure list of all functions which are defined in the package.

Bluetcl::flags

Returns or sets the status of flags used by BSC.

flags show *flagname* [*flagname ...*]	Show the value of the specified flags. One ore more flag names can be provided.
flags set *flagname value* [*flagname value ...*]	Set the flags to the value provided. Multiple flags may be set in a single command.

Note: Values enclosed in brackets define a single token. For example, setting the value of the flag `-verilog-filter` with embedded spaces:

```
Bluetcl::flags set -verilog-filter {sed -e -e 's/XX/SS/'}
```

Bluetcl::help

Help with no arguments will list all available help topics. Optionally, an argument can be provided to get help on a specific topic. Also, 'help list' will return a string listing the names of all commands.

help	Returns a list of all help topics.
help list	Returns a string listing the name of all commands
help *command*	Returns help for the specified command.

Bluetcl::module

Returns information on synthesized (post elaboration) modules.

module load *modname*	Loads the module and all instantiated submodules into Bluetcl's workspace. Returns a list of the modules loaded.
module clear	Clear all loaded modules
module submods *modname*	Returns a 3-tuple. The first element of the tuple is a tag (primitive or user), the second is a list of pairs contain the synthesized submodule name and its interface type. The third element is a list of function which have not been in-lined.
module rules *modname*	Returns a list of rule names in the module.
module ifc *modname*	Returns a list of interface types in the module.
module methods *modname*	Returns a list of the flattened methods in the module.
module ports *modname*	Returns a list of the ports in the module.
module porttypes *modname*	Returns a list of the types of the ports in the module.
module list	Returns a list of all loaded modules.

Bluetcl::rule

Returns information about rules in a post elaboration module.

rule rel *modname rule1 rule2*	Shows the relationship between two rules in the module.
rule full *modname rule*	Returns a tagged structure detailing the rules position, predicates expression, attributes and method calls.

Bluetcl::schedule

Returns scheduling information for a synthesized module. The `schedule` command requires a subcommand and the module name.

schedule execution *modname*	Returns a list of rule/method names in execution order. For example, if `r1` fires after `r2`, then the output would be: `RL_r1 RL_r2`.
schedule methodinfo *modname*	Returns scheduling relationships between all pairs of methods.
schedule pathinfo *modname*	Returns a list of combinational paths through the module. Each element is a list of two elements: a list of inputs and an output that they connect to.
schedule urgency *modname*	Returns a list of lists, one for each rule/method, in urgency order. Each lists contains two elements: the rule name and a list of rules which would block that rule from firing.
schedule warnings *modname*	Returns a list of scheduling warnings. The result is a list of three elements: the position of the warning, the tag for the warning, and the complete warning message.

Bluetcl::sim

Controls all aspects of loading, executing and interacting with a Bluesim simulation object. These commands are used when running Bluesim interactively, as described in section 6.4.

This command is also provided in the `Bluesim` package. See section B.4.3 for the complete definition.

Bluetcl::submodule

Returns information about each submodule and which rules use the methods of the submodule.

submodule full *modname*	Returns information about each submodule in the specified module and the rules which use the methods of the submodule.

Bluetcl::type

Finds and returns type information.

type constr *typename*	Shows the type constructor for the provided type name. The type constructor is the type arguments needed for the type. Returns an error if the typename is not found in any of the loaded packages.
type full *typeconstructor*	Returns a tagged structure based on the type constructor argument. The type constructor provided must be fully qualified.

Bluetcl::version

Returns the current compiler version

version	Returns a list of 2 items: the compiler version and the build version. The compiler version is provided in year-month-(annotation) format.

B.4.3 Bluesim

The Bluesim package contains the `sim` command which controls Bluesim interactive mode. This command is also found in the `Bluetcl` package.

Before using a command from the Bluesim package, the following Tcl command must be executed, either in a script, from the command line, or in the `.bluetclrc` file:

```
package require Bluesim
```

All commands in the `Bluesim` package are in the `Bluesim::` namespace. The namespace must be referenced, as described in Section B.3, either by using the `namespace import` command or by prepending the command name with `Bluesim::`. Example:

```
Bluesim::sim clock
```

sim

Controls all aspects of loading, executing and interacting with a Bluesim simulation object. These commands are used when running Bluesim interactively, as described in section 6.4. This command is also provided in the `Bluetcl` package (B.4.2).

sim arg *string*	Set a simulation plus-arg. Adds the supplied *string* to the end of the list of plusargs searched by the `$test$plusargs` system task.
sim cd [*path*]	Change location in hierarchy. The path must consist of a sequence of instance names separated by a period (`.`). A path which begins with a `.` is an absolute path from the top of the hierarchy, but one which does not begin with a `.` is relative to the current location. No provided *path* will return the user to the uppermost point in the hierarchy.
sim clock	Returns a list containing a clock description for each currently defined clock.
sim clock [*name*]	Select the named clock, make it the active clock.
sim describe *handle*	Describe the object to which a symbol handle refers.
sim get *handle*	Returns the simulation value for the object with the provided *handle*. The value is returned as a sized hexadecimal number.
sim getrange *handle addr*	Get simulation values from a range.
sim load *model*	Load a bluesim model object.
sim lookup *pattern* [*root*]	Lookup symbol handles. Returns a handle for every simulation object which matches the *pattern*.

sim ls *pattern* *	List the sub-instances, rules and values at that level of the hierarchy. If a *pattern* is provided, it controls which names are listed. No pattern is equivalent to `sim ls *`.
sim nextedge	Advance simulation to the next clock edge in any domain.
sim pwd	Print current location in hierarchy.
sim run [**async**]	Run simulation to completion. The keyword `async` cause the command to return immediately so that additional commands can be processed while the simulation continues to run asynchronously.
sim runto *time* [**async**]	Run simulation to a given time. The keyword `async` cause the command to return immediately so that additional commands can be processed while the simulation continues to run asynchronously.
sim step [*cycles*] [**async**]	Advance simulation a given number of cycles. The keyword `async` cause the command to return immediately so that additional commands can be processed while the simulation continues to run asynchronously.
sim stop	:stop the simulation at the end of the currently executing simulation cycle.
sim sync	Wait for simulation to complete normally before returning
sim time	Display current simulation time.
sim unload	Unload the current bluesim model.
sim up [*N*]	Move up the module hierarchy into the parents of the current directory. It will move up `N` levels if N is provided.
sim vcd [**on** | **off** | *file*]	Control dumping waveforms to a VCD file named *file*. If no file name is provided, it will use the default file `dump.vcd`.
sim version	Show Bluesim model version information.

B.4.4 InstSynth

The `InstSynth` package contains scripts to generate instance specific synthesis in Bluespec SystemVerilog. These scripts use Bluespec's typeclass and overloading to match a module's instantiation with a specific instance which may instantiate a synthesized module.

When using any of the commands in the `InstSynth` package, the package must be loaded first.

```
package require InstSynth
```

The `InstSynth` package contains the commands `genTypeClass`, `genSpecificInst`, and `genSynthMod`.

InstSynth Commands	
`genTypeClass`	**genTypeClass** *packname* {*modname*}
	Creates an include file for a package containing a typeclass for overloading of a module and a default instance for the module. Multiple modules within the same package can be specified in a single command.
`genSpecificInst`	**genSpecificInst** *packname modname type*
	Modifies the *packname*`.include.bsv` file with an instance for each missing type.
`genSynthMod`	**genSynthMod** *packname modname type*
	Generates a synthesize module wrapper for a given module and type within a package. The generated module is returned as a string from this function.

Example using InstSynth.tcl to generate instance specific synthesis modules

Overview: This example demonstrates how to use the `InstSynth.tcl` package to use Bluespec's typeclass and overloading to match a module's instantiation with a specific instance to instantiate a synthesized module.

The example is composed of three `.bsv` files: `m1.bsv` which contains the module definition for `mkM1`, `m2.bsv` which contains the module definition for `mkM2` and instantiates multiple instances of `mkM1`, and `Top.bsv` containing the testbench `mkTb`. The two modules `mkM1` and `mkM2` are polymorphic. The testbench `mkTb` instantiates `mkM2` and is not polymorphic.

The polymorphic modules are instantiated in the hierarchy as shown in figure 7.

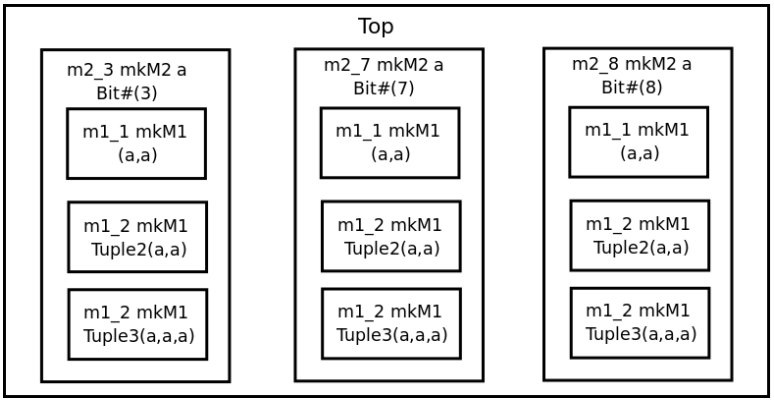

Figure 7: Example Module Hierarchy

Steps for using InstSynth

1. Compile your design with bsc as normal, creating the `.bo` files.
2. Generate typeclass and default instances for modules with the `genTypeClass` command, specifying the packages and modules for instance specific synthesis. The `genTypeClass` command will generate an include file (`<package>.include.bsv`) for each package.

The include file will contain a type class (named `MakeInst_<module>`) for each module and a general catch-all instance of that type class.

The type class contains one method, which is a module constructor. The method is named `<module>_Synth` and has the same arguments as the general polymorphic module. The instance of this type class is a thin wrapper which instantiates the polymorphic module and prints a message about the type which is synthesized.

Example of the include file for `m1.bsv` (`m1.include.bsv`) after this step:

```
typeclass MakeInst_mkM1 #(type ifc_t);
    module mkM1_Synth ( ifc_t ifc) ;
endtypeclass

instance MakeInst_mkM1 #(  ClientServer::Server#(a, a) )
    provisos (Bits#(a, sa)) ;

    module mkM1_Synth ( ClientServer::Server#(a, a) ifc) ;
        let _i <- mkM1  ;
        messageM ("No concrete definition of mkM1 for type " +
                    (printType (typeOf (_i))));
        messageM ("Execute: InstSynth::genSpecificInst m1 mkM1 {" +

                    " {" + (printType (typeOf(asIfc (_i)))) + "}"
                    + " }"  );
        return _i ;
    endmodule
endinstance
```

3. Manually edit the `.bsv` file to include the generated file. For example, add the line `‘include "<package>.include.bsv"` at the bottom of the file for `<package>`. In this example, you would add the line `‘include "m1.include.bsv"` to the file `m1.bsv`.

4. Manually edit the `.bsv` file to change the module constructor from `<module>` to `<module_Synth>` at each point you would like an instance synthesized. In this example, change `mkM1` to `mkM1_Synth` in the `m2.bsv` file, and change `mkM2` to `mkM2_Synth` in the `Top.bsv` file where you want to synthesize an instance.

5. Compile the design again with bsc. The compile will generate messages listing missing instances along with the `genSpecificInst` command to create each missing instance. Execute the commands one at a time to generate an instance for each missing type.

   The `genSpecificInst` command will modify the `include.bsv` files, adding the instances. For example, in this step, the following lines are added to the `m1.include.bsv` file to resolve the `Server#(a,a)` type.

```
module mkM1__ClientServer_Server_Bit_3_Bit_3_(
                            ClientServer::Server#(Bit#(3), Bit#(3)) ifc ) ;
    let _i <- mkM1  ;
    return _i ;
endmodule

instance MakeInst_mkM1 #( ClientServer::Server#(Bit#(3), Bit#(3)) ) ;
    module mkM1_Synth ( ClientServer::Server#(Bit#(3), Bit#(3)) ifc );
        let _i <- mkM1__ClientServer_Server_Bit_3_Bit_3_  ;
        messageM("Using mkM1__ClientServer_Server_Bit_3_Bit_3_ for mkM1 of
```

```
                type: " +
                  (printType (typeOf (_i))));
          return _i ;
      endmodule
endinstance
```

Note that the code added in this step does not add to or change the behavior of the design. Only the additional hierarchy is added.

6. Add provisos to the polymorphic modules to avoid early binding of the module. Otherwise compiling at this point will not show the specific instances because the instance of the _Synth is bound before the specific type of the module is known. In this example, mkM1_Synth would be bound before the specific type of mkM2 is known. To fix this, provisos are added to the polymorphic module mkM2. The module mkM2 has three instantiations of mkM1, therefore a proviso for each instantiation is added.

```
module mkM2 (Server#(a,a)) provisos (Bits#(a,sa)
                     ,MakeInst_mkM1#(Server#(a,a))
                     ,MakeInst_mkM1#(Server#(Tuple2#(a,a),Tuple2#(a,a)))
                     ,MakeInst_mkM1#(Server#(Tuple3#(a,a,a),Tuple3#(a,a,a)))
                     );
```

7. Compile with bsc again.

8. Continue until there are no missing instance messages. A Verilog file will be created for each synthesized module instance.

SynthInst Example Files

m1.bsv: Module mkM1 is defined in the package (file) m1.bsv:

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

module mkM1 (Server#(a,a))    provisos (Bits#(a,sa));
   FIFOF#(a) fifo <- mkFIFOF;

   interface request = toPut (fifo);
   interface response = toGet (fifo);
endmodule
```

m2.bsv: Module mkM2 is defined in the package m2.bsv. Note that there are three instantiations of mkM1. You can choose to synthesize any or all of the instances.

```
import FIFO::*;
import GetPut::*;
import ClientServer::*;
import m1 :: *;

module mkM2 (Server#(a,a))    provisos (Bits#(a,sa)
   Server#(a,a) m1_1 <- mkM1;
   Server#(Tuple2#(a,a),Tuple2#(a,a)) m1_2 <- mkM1;
```

```
   Server#(Tuple3#(a,a,a),Tuple3#(a,a,a)) m1_3 <- mkM1;

   rule r0;
      let { x1,x2 } <- m1_2.response.get();
      m1_3.request.put (tuple3 (x1,x2,x2));
   endrule

   interface Put request;
      method Action put (a x);
         m1_2.request.put (tuple2(x,x));
      endmethod
   endinterface

   interface Get response;
      method ActionValue#(a) get ();
         let { y1,y2,y3 } <- m1_3.response.get();
         return (y1);
      endmethod
   endinterface
endmodule
```

Top.bsv: The testbench is contained in the file `Top.bsv`

```
import ClientServer :: *;
import GetPut :: *;
import m2 :: *;

(* synthesize *)
module mkTb (Empty);
   Reg#(int) cycle <- mkReg (0);

   Server#(Bit#(3), Bit#(3)) m2_3 <- mkM2;
   Server#(Bit#(7), Bit#(7)) m2_6 <- mkM2;
   Server#(Bit#(8), Bit#(8)) m2_8 <- mkM2;

   rule r1;
      $display ("%0d: r1: put (%0d)", cycle, cycle);
      m2_3.request.put (truncate (pack (cycle)));
      m2_6.request.put (truncate (pack (cycle)));
      cycle <= cycle + 1;
      if (cycle > 8) $finish(0);
   endrule

   rule r2;
      let x_3 <- m2_3.response.get ();
      let x_6 <- m2_6.response.get ();
      $display ("%0d: r2: %0d,%0d <= get", cycle, x_3, x_6);
   endrule
endmodule
```

B.5 Bluetcl Scripts

Scripts are self-contained commands you run from a shell. A Tcl script may be include any combination of Bluetcl and Tcl commands.

The scripts described in this section are provided with BSC in the `$BLUESPECDIR/tcllib/bluespec` directory. To execute a script, type the fully qualified script name. For example, to execute the `expandPorts` script from a command prompt you would type:

```
$BLUESPECDIR/tcllib/bluespec/expandPorts.tcl
```

If you are already in a Tcl shell, type `exec` before the script name:

```
exec $BLUESPECDIR/tcllib/bluespec/expandPorts.tcl
```

B.5.1 expandPorts

Script to create a Verilog wrapper file which expands structures into separate Verilog ports.

Usage:

expandPorts.tcl {options} *packname modname module.v*

options	Optional command line switches:
-p *path*	path, if supplied to the bsc command
-verilog	compile to verilog (default)
-sim	compile to bluesim
-include *outfile*	output file for `include.vh`
-wrapper *outfile*	output file for `wrapper.v`
-rename file.tcl	Tcl script creating rename pin structure
-makerename	Create empty `.rename.tcl` file to edit for `-rename`
-interface *name*	Interface to expand - defaults to package name (*packname*)
packname	Name of the input `.bo` file.
modname	Name of the top level module.
module.v	bsc generated Verilog (`.v`) file for the module being wrapped.

Index

Commands by Namespace