Verification Continuum^{TM}

Hector
User Guide

Version P-2019.06-SP2, December 2019

SYNOPSYS®

Contents

1 Introduction

This chapter provides an introduction to Hector and its capabilities. The chapter is organized into the following sections:

- ❖ "What is Hector?"
- ❖ "Using this Manual"
- ❖ "Getting Help"
- ❖ "Using Hector Start-up Files and Command Line Options"

1.1   What is Hector?

Hector is a formal functional verification tool optimized for proving properties, including transaction equivalence, on data path designs. Hector can build formal representations of RTL and C++ design blocks and then analyze lemmas about the behavior of those blocks. Hectors supports the following features:

- ❖ Waveform and source code debugging of failed lemmas
- ❖ Powerful formal engines
- ❖ Automated checks for model correctness
- ❖ Support for multi-processing
- ❖ Assume-guarantee and case-splitting automation

The most powerful use model for Hector is the verification of transaction equivalence between two models. A transaction is a unit of computation with specific inputs and outputs. Transactions may be clocked or unclocked. Combinational circuits, and C++ functions, execute transactions with no time delay between inputs and outputs. Clocked RTL circuits may take one or more cycles to compute a transaction. In the case of pipelining, multiple transactions may be executing simultaneously in a single design block. The duration of a transaction (measured in clock cycles) may be fixed or data dependent, but there must be a known upper bound.

One of the key benefits of Hector is the modeling of transaction equivalence where the latency of a transaction is different in the two models compared. Hector supports constraint of inputs and comparison of outputs at transaction-relative clock phases independently in each design.

In addition to equivalence lemmas, Hector can be used to prove or invalidate lemmas about the behavior of a single design. These lemmas can be combined with equivalence lemmas in a two-design proof setup, or run independently on a single design block.

Many Hector users have previously used simulation to compare high level reference model behaviors against an RTL implementation. In this flow, inputs to the RTL block are monitored and transferred to the reference model. The outputs computed by the reference model are then compared against the outputs of

the RTL block and discrepancies are flagged. This method can find bugs, but with wide data path operand values it is often impossible to achieve complete coverage. An operation with two 16-bit inputs requires 4 billion stimulus patterns for exhaustive coverage. Hector's formal approach may achieve exhaustive coverage far beyond what is practical in simulation. If corner-case discrepancies are present, Hector will find them.

Hector can check lemmas on any design block compatible with the provided compilers. Synthesizable subsets of Verilog, VHDL and SystemVerilog are supported by the RTL compiler, including mixed language designs. The C++ compiler supports most of the language, with a few exceptions and coding style restrictions discussed later in this manual. Once the source designs are compiled into Data Flow Graph (DFG) format, Hector can combine and reason about them independent of source language. This allows Hector to check equivalence between two C++ models, two RTL models, or most commonly, a C++ reference model and an RTL implementation model.

Hector utilizes a TCL control script to compile the models, map them together, specify verification objectives, run the solvers, report results, and debug. Hector can be run interactively from the TCL shell, or in batch mode with a prepared script. A common usage is to encapsulate setup, run, and reporting commands in a script, and then transition to interactive commands for debug and analysis.

1.2   Using this Manual

It is recommended that you start reading this user manual with the "Quick Start Guide"chapter. This chapter provides the basic usage flow and references to later chapters with much more details. Once you know some basics, you can use keyword searches to find topics of interest in the later chapters.

1.3   Getting Help

Hector support can be reached through Solvnet and most standard Synopsys support channels. The support team uses Cases and STARS to log issues, defects and enhancement requests.

Many of the TCL command support terse help messages. Typing a command name at the TCL prompt with a `-help` provides more information.

1.4   Using Hector Start-up Files and Command Line Options

Hector is started with the command `hector`. When Hector starts it will attempt to read three `.hectorrc` files in the following order.

1. `$HECTOR_HOME/.hectorrc`
2. `~/.hectorrc`
3. `./.hectorrc`

The Hector TCL interpreter sources each file in turn, if it is present. If the `-script <filename>` option is specified, it will read `<filename>` after the `.hectorrc` files have been read. If the `-script <filename>` option is not specified, Hector will look in the current directory for a default startup script called `command_script.tcl` and source it if it is present.

A log file named `hector.log` is automatically created and all hector output is captured in this file. The `-logfile` option can be used to specify a different file name for the log.

Syntax:

```
hector [-script <filename>] [-no_init] [-exec <commands>]

[-logfile <filename>] [-verbose <n>] [-session <dir>]
```

```
[-version] [-help]
```

Options:

❖ `-script <filename>:` Specifies a script to be loaded instead of the default `command_script.tcl`.

❖ `-no_init`: Prevents the loading of all `.hectorrc` scripts at invocation. This does not prevent the loading of the file specified with the `-script` option.

❖ `-exec <script>`: Specifies a TCL script to execute after the setup is complete. Hector will not be interactive and will exit when the script exits. The argument is the actual script text, not a filename. For example:

```
hector -script hector.tcl -exec "compose; solve"
```

This command loads the command script `hector.tcl`, executes the compose and the solve command, and then exits.

❖ `-logfile <filename>:` Writes a copy of Hector's output to the indicated file. If this switch is not provided, the output is written to `hector.log` in the current working directory

❖ `-verbose <n>`: Sets the output verbosity level. The default is `n=0`. If `n > 0`, more verbose messaging is enabled.

❖ `-session <dir>`: Uses the specified directory as the temporary work area. The default is `SCV_WORK`.

❖ `-version`: Prints the software version string.

❖ `-help`: Prints a usage message.

2 Quick Start Guide

The goal of this chapter is to describe how to set up and run your first example using Hector in the following sections:

- ❖ "Setting up Hector [Chapter 3]"
- ❖ "Writing a Hector Wrapper for C/C++ Model [Chapter 4]"
- ❖ "Declaring the *Specification Design [Chapter 4]*"
- ❖ "Declaring the *Implementation Design [Chapter 4]*"
- ❖ "Mapping the Designs [Chapter 5]"
- ❖ "Invoking Hector"
- ❖ "Analyzing the Results [Chapter 6]"
- ❖ "Debugging Failed Lemmas [Chapter 7]"

A typical Hector example consists of a specification design, an implementation design, and a Hector command script file.

For simplicity and clarity a small C++ to RTL equivalence checking example is used. This example can be found in the hector installation at: `$HECTOR_HOME/tutorial/quickstart`.

2.1   Setting up Hector [Chapter 3]

The chapter "Installing Hector" of this manual describes how you can install Hector. In this section, it is assumed that you have already set following environment variables. Depending on the counter-example debug methodology, additional setup variables for VCS, Verdi or DDD may be required.

- ❖ `HECTOR_HOME =  <path-to-installation>`
- ❖ `PATH =  $HECTOR_HOME/bin:$PATH`
- ❖ `SNPSLMD_LICENSE_FILE  =  123456@servermachine`

2.2   Writing a Hector Wrapper for C/C++ Model [Chapter 4]

The C++ source code for this example is in the file `$HECTOR_HOME/tutorial/quickstart/foo.cc`. The following lines have been added to the C++ model for Hector.

```
#include <Hector.h>
void hector_wrapper()
{
    int a, b, mul, madd;
    Hector::registerInput("in_a", a);
```

```
        Hector::registerInput("in_b", b);

        Hector::registerOutput("out_mul", mul);

        Hector::registerOutput("out_madd", madd);


        Hector::beginCapture();

        compute(a, b, mul, madd);

        Hector::endCapture();

    }
```

This wrapper function is required because the C++ language does not have a notion of input and output. For example the function,

```
        Hector::registerInput("in_a", a);
```

indicates that the variable `a` should be treated as an input and it is to be referred to as `in_a` in all the subsequent steps of Hector.

The directives `Hector::beginCapture` and `Hector::endCapture` specify the start and end of the transaction respectively.

2.3   Declaring the *Specification* Design [Chapter 4]

The script file for this example can be found at
`$HECTOR_HOME/tutorial/quickstart/command_script.tcl`. The following TCL procedure is added to the script to provide information about the specification design.

```
    proc compile_spec {} {

        create_design -name spec -top hector_wrapper

        cppan foo.cc

        compile_design spec

    }
```

The `create_design` command creates a design with name *spec* and also specifies that the top-level function is `hector_wrapper` (see section "Writing a Hector Wrapper for C/C++ Model [Chapter 4]"). The `cppan` command analyzes the C++ program. It takes as argument a list of g++ compiler options and files. Finally, the `compile_design` command compiles the design named *spec* into a DFG.

2.4   Declaring the *Implementation* Design [Chapter 4]

The next step is to add the following TCL procedure in the command script file for the implementation design.

```
    proc compile_impl {} {

        create_design -name impl -top ima –clock clk –reset rst -negReset

        vcs foo.v

        compile_design impl

    }
```

Here again, the `create_design` command creates a design with name *impl* and the top-level module as *ima*. The –clock option specifies the name of the clock input port. The –reset option identifies the reset input to the design and the –negReset option indicates that the reset is active low. If the reset were active high, then

the option –negReset would be omitted (the default is active high). Next, the VCS command analyzes the Verilog program and the compile_design command compiles the *impl* design into a DFG.

2.5 Mapping the Designs [Chapter 5]

Hector maps the designs during the *compose* step using commands in a TCL procedure. *Assume* commands are used to specify that inputs of the two designs must be equal during specified phases of the transaction. *Assume* commands can also be used to restrict the legal values of inputs. *Lemma* commands indicate which output signals are to be checked for equivalence and at which phases. This data is passed to Hector by calling the following set TCL procedure:

```
set_user_assumes_lemmas_procedure "ual"
proc ual {} {
  map_by_name -inputs -specphase 1 -implphase 1    # map inputs
  map_by_name -outputs -specphase 1 -implphase 3   # map outputs
  assume op = impl.in_opcode(1) == 1'b0            # constrain an input
}
```

The map_by_name command is a short-hand way to create a series of assume and lemma commands, one for each input or output in the two designs with matching names. The -specphase and –implphase switches indicate the timing correspondence between transactions in the two models. The first map_by_name command in the TCL procedure is equivalent to the following two commands:

```
assume _scv_assume_0 = (spec.in_a(1)[3:0]) == (impl.in_a(1)[3:0])
assume _scv_assume_0 = (spec.in_b(1)[3:0]) == (impl.in_b(1)[3:0])
```

The second map_by_name command is equivalent to the following two commands:

```
lemma _scv_lemma_2 = (spec.out_mul(1) == impl.out_mul(3))
lemma _scv_lemma_3 = (spec.out_madd(1) == impl.out_madd(3))
```

For this small example, the map_by_name command does not save much typing. But for designs with many inputs and outputs, and matching naming schemes, it can make the scripts much more compact and readable.

2.6 Invoking Hector

Hector is invoked with the following command -- where script.tcl is the name of the command script file.

```
% hector –script script.tcl
```

The default script name is command_script.tcl. Therefore, in this case Hector can be run simply by issuing hector without the -script option.

This places Hector in an interactive TCL environment. Issuing the following four commands will compile and formally compare the two models.

❖ compile_spec: Compiles the C/C++ model

❖ compile_impl: Compiles the RTL model

❖ compose: Generates a design wrapper that composes the two designs

❖ solve: Proves the equivalence or in-equivalence of each corresponding output

2.7   Analyzing the Results [Chapter 6]

To find the status of the lemmas after the `solve` command completes execution, use the `listproof` command. This prints out the following message:

```
================================================================
outputs
  Assumptions                  3
  User Defined Lemmas          2
  Hector Generated AEP Lemmas 0
  Cutpoints                    0
  Proof Strategy               default
  Parent Proof                 none
----------------------------------------------------------------------
             Assumptions
Use Name                       Expression
----------------------------------------------------------------------
  *   amap                     spec.in_a(1) == impl.in_a(1)
  *   bmap                     spec.in_b(1) == impl.in_b(1)
  *   op                       impl.in_opcode(1) == 1'b0


----------------------------------------------------------------------
Lemmas
Use Status    Name                     Expression
----------------------------------------------------------------------
  *   success  mulCheck         spec.out_mul(1) == impl.out_mul(3)
  *   failed   addCheck         spec.out_madd(1) == impl.out_madd(3)
----------------------------------------------------------------------
Tasks: Out of 1 tasks created
           1 have finished
Results: Out of 2 enabled lemmas
           1 were successful
           0 were conditionally successful
           1 failed
           0 were conditionally failed
           0 were inconclusive (timeout)
           0 were not tried


Status for proof "outputs": FAILED
================================================================
```

There is one lemma for each pair of corresponding outputs. The listing above indicates that the logic that computes `out_mul` in both the C++ and the RTL model is functionally equivalent. However, the logic cones that drive `out_madd` in the two models are not equivalent.

2.8   Debugging Failed Lemmas [Chapter 7]

To debug the failed lemmas use the `simcex` command. The `simcex` command simulates a counter example found by the `solve` command and creates one of several compatible formats for counter example debugging. For example, VCD files for the failing lemma can be generated using the following command:

```
simcex _scv_lemma_3 -vcd mycex.vcd
```

We can also use Hector Debugger (HDB) to debug the C++ design using the following command:

```
simcex _scv_lemma_3 -hdb
```

3 **Installing Hector**

This chapter describes the various steps involved in downloading and installing Hector in the following sections:

- ❖ "Setting up Hector"
- ❖ "Licensing and Keys"
- ❖ "Platform Issues"
- ❖ "Debugger Support"

You will need the Synopsys installer package. Use the following link if you do not already have it installed: `http://www.synopsys.com/install`. Look for a link to `installer_INSTALL_README.txt`a for specific instructions.

Then download the `common` and platform specific `linux` and `suse` compressed tarballs. You should have specific instructions for this from an e-mail or your CAE contact.

From the directory where you downloaded the tarballs, run the installer, answering the prompts.

3.1   Setting up Hector

Assuming that you are running bash on a linux system, the following environment variables must be set:

- ❖ `export HECTOR_HOME=<path-to-installation>`
- ❖ `export PATH=$PATH:$HECTOR_HOME/bin`

And, the following Synopsys license file must be referenced:

- ❖ `export SNPSLMD_LICENSE_FILE=123456@servermachine`

3.2   Licensing and Keys

Hector implements Synopsys Common Licensing. Hector may use the following features:

- ❖ VC STATIC COMPILE
- ❖ HECTOR
- ❖ HECTOR-CORE
- ❖ VC-FORMAL-BASE
- ❖ VC-FORMAL-DPV
- ❖ VC-FORMAL-DPV-HDB
- ❖ VC-FORMAL-DPV-CPP11
- ❖ VCFV-Verdi-Plugin

You may also need VCS and/or Verdi keys for some debug activities, though these keys are not used by Hector directly.

3.3 Platform Issues

Hector for Linux is built on RHE 4.0 Linux and runs on any binary compatible platform. Hector supports the current version of QSC as outlined in the Synopsys release train documentation.

3.4 Debugger Support

To use Hector's HDB debugger, you should have the Gnu DDD debugger interface installed on your computer and visible in your path. Many Linux distributions include DDD binaries, but you can also download source and binaries from `http://www.gnu.org/software/ddd/`. The minimum required version of DDD is 3.3.8.

To utilize DVE or Verdi debugging tools you must have VCS and/or Verdi installed on your system and visible in your search paths. Refer to documentation for those tools for installation and licensing requirements.

4 Compiling a Design

The first step in Hector is to compile the designs into formal models that are represented in DFGs. This chapter describes the commands required for compiling a design using Hector in the following sections:

The designs can be written in various languages, such as C, C++, SystemC, Verilog, VHDL, and mixed language (MX). A generic template for compiling a design is described first and in later sections individual options for each language are described.

4.1   Compiling a Design

Use the following steps to compile a design:

1. Create a design using the `create_design` command.

2. Analyze the design (This is a language specific step).

3. Elaborate the design (optional step).

4. Generate the DFG by calling the `compile_design` command.

Note

Typically, these commands are enumerated in a TCL procedure and then call that procedure from the Hector shell. The subsequent sections discuss these steps in more detail.

4.1.1   Creating a Design

The syntax for creating a design using the `create_design` command is as follows:

```
create_design –name <spec|impl> -top <topname>

    [-options <string>]

    [-clock <name] [-reset <name>] [-negReset]

    [-lang <c|c++|scdt|systemc|verilog|vhdl|sverilog|mx>]
```

Options:

- ❖ `-name <designname>`: Specifies the name of the design. Possible values are *spec* or *impl*.
- ❖ `-top <topname>`: Specifies the top level module/function name for Hector analysis.
- ❖ `-options <string>`: Used to provide any additional compiler specific options.
- ❖ `-clock <name>`: Specifies the name of the clock port in a single clock design.
- ❖ `-reset <name>`: Specifies the name of the reset signal in a design with simple reset.
- ❖ `-negReset`: Specifies the polarity of the reset signal.
- ❖ `-lang <name>`: Specifies the language for the design. Possible values are c, c++, scdt, systemc, verilog, vhdl, mx, or sverilog.

Hector tries to deduce the language of the design from the `analyze` commands. However, if Hector is unable to deduce it, then you must specify the language type by using the `-lang` option.

4.1.2   Analyzing Source Files

This section describes how you can use the different `analyze` commands. This command is language dependent.

4.1.2.1   Analyzing C++ Files

The command for analyzing C++ files is as follows:

```
cppan <options> <list of filenames>
```

Options:

- ❖ `<options>`: The g++ compiler options for compiling the files.
- ❖ `<list of filenames>`: List of files to compile.

By default Hector adds the options `-DHECTOR -w` to all designs. Hector also adds the following list of default places to look for include files.

```
-I.
-I${HECTOR_HOME}/local
-I${HECTOR_HOME}/local/include
-I${HECTOR_HOME}/local/systemc_2_1.oct_12_2004.beta_hector/include
-I${HECTOR_HOME}/local/gccinclude/
-I${HECTOR_HOME}/local/gccinclude/c++/4.8.3
-I${HECTOR_HOME}/local/gccinclude/c++/4.8.3/x86_64-redhat-linux
-I${HECTOR_HOME}/local/gccinclude/c++/4.8.3/backward
-I${HECTOR_HOME}/local/gccinclude/c++/4.8.3/x86_64-redhat-linux/4.8.3/include
-I/usr/include
```

4.1.2.2   Analyzing SystemC Files

For details about compiling SystemC designs, see section "Support for SystemC Data Types".

4.1.2.3   Analyzing Verilog Files

For Verilog only designs, the command to analyze (and elaborate) the files is as follows:

```
vcs <options> <list of filenames>
```

In a mixed language design containing both Verilog and VHDL, the command for analyzing Verilog files is as follows:

```
vlogan <options> <list of filenames>
```

4.1.2.4   Analyzing VHDL Files

The command for analyzing VHDL files is as follows:

```
vhdlan <options> <list of filenames>
```

4.1.3   Elaborating the Design (Optional)

This step is used only for VHDL or mixed language designs where elaboration options are required. The command for elaborating the design is as follows:

```
vcs <options>
```

This command is modeled on the VCS elaboration command. The VCS elaboration step requires the name of top level entity or module, however, Hector does not require this information as it is already you would have already provided it when creating a design (with `-top` option). This command is not needed if no elaboration time options need to be given.

4.1.4   Generating the DFG

The command for generating the DFG is as follows:

```
compile_design name
```

The argument name should match the name used while using the `create_design` command. A data-flow graph named `name.dfg` is written in Hector's work directory.

4.2   Compiling a C/C++ Design

Use the following steps to compile a C++ design:

1.   Create a design using `create_design` command.

2.   Analyze C++ files using `cppan` command (at least one required).

3.   Generate the DFG by calling `compile_design` command.

Example

```
proc compile_spec { } {

    create_design -name spec -top main

    cppan –Iinclude –DCHECKFP foo.cc

    compile_design spec

}
```

4.3   Compiling a C++11 Design

Hector supports compiling C++11 design. The original C++99 front-end continues to be supported. To use the newer C++11 front-end version, you require an additional license. By default, the original C++99 front-end is used to compile C/C++ models. To enable the C++11 front-end, the following command must be placed in the Hector setup file.

```
set _hector_comp_use_new_flow true
```

When using the C++11 front-end, it is possible to list all the files compiled and included during the compilation. This listing contains all the included files: system files, hector specific files and user-defined include files. When the C++11 front-end is invoked, it creates a file in the current directory called `<design name>.includes`, where `<design name>` is the argument used for the `create_design` option `'-name'`.

For example, if the following compile procedure is used:

```
create_design -name spec -top main
cppan -I. -I../include  a.cc b.cc
compile_design spec
```

The following file is created by the C++11 front-end in the current directory.

```
spec.includes
```

Once the `compile_design` command completes, the `list_include_files` command can be used to find all included files.

For example, the `list_include_file spec.includes` command generates the following listing:

```
a.cc

    b.cc

    /tools/hector/local/Hector_builtin.cc

    /tools/hector/local/Hector_lib.cc

    <all user defined include files>

    <all hector defined include files>

    <all system defined include files>
```

All the C/C++ files are listed first, followed by all the include files. Note the included files are listed in lexicographic sort order.

The new C++11 frontend supports `-m32` and `-m64` options in the `cppan` command ("Compiling a C/C++ Design").

The `-m32` option is the default, and makes the compiler produce 32-bit code, whereas the `-m64` option makes the compiler produce 64-bit code.

The new C++11 frontend also supports the `__uint128` and `__int128` data types. Both require using the `-m64` in the `cppan` command. These are extensions to the standard and implement 128-bit wide integers/unsigned integers.

To filter out all the system files, use the `-excludeDirs <dir list>` option:

For example:

```
list_include_file spec.file -excludeDirs "/usr/include  /user/local/include"
```

You can also eliminate any included file that begins with any of the directory names specified using the `-excludeDirs` option. Similarly, you can remove all of the hector internal includes.

The option `-sortByFile` creates an include listing for each file specified on the `cppan` command line.

For example, the following command:

```
        list_include_file spec.file -excludeDirs "/usr/include  /user/local/include
/tools/hector" -sortByFile
```

creates the following listing:

```
            a.cc

            <all user defined include files>

            b.cc

            <all user defined include files>

            /tools/hector/local/Hector_builtin.cc

            /tools/hector/local/Hector_lib.cc
```

Observe that none of the hector *.cc files contain any user-defined includes.

4.4   Support for Math Library Functions

Hector is being shipped with its own math library that provides support for a subset of functions that can be found in the Linux system math-library libm.a (typically from glibc).

Note

All functions in Hector's math library, except fmaf, have been formally compared against an openly available libm implementation. Note that the bit-patterns for non-canonical floating-point results like NaN or invalid numbers in extended precision may not match exactly between different libm implementations.

4.4.1   Prerequisites

The libm.a math library is currently only available for the C++11/14 compile flow, which can be enabled using the following application variable:

```
set_hector_comp_use_new_flow true
```

4.4.2   Using the math library

The math library needs to be explicitly linked to the executable by using the following command:

```
cppan –link -lm
```

The -link option tells the cppan command that the following options should be interpreted as linker options. The -lm tells the linker to link against the math library libm.a.

4.4.3   Example

Assume you want to call the ceilf function from your C++ file test.cc:

```
#include <math.h>
int main()
    {
        float in;
        float out;

        Hector::beginCapture();
        out = ceilf(in);
        Hector::endCapture();
    }
```

The declarations of all math functions can be accessed by including

```
#include <math.h> or #include <cmath>.
```

Without linking the math library, an error about unresolved symbols after the compilation is reported. To avoid that, add the `cppan -link -lm` command:

```
set_hector_comp_use_new_flow true
create_design spec -top main
cppan test.cc
cppan -link -lm
compile_design
```

👉 **Note**

For more details on the supported math library functions, see section "Supported Math Library Functions".

4.5 Compiling an RTL Design

Hector supports synthesis compatible RTL coding styles. Behavioral models that violate synthesis requirements may produce errors or corrupted formal models.

There are two distinct flows for compiling RTL designs. For Verilog only designs the compilation steps are as follows:

1. Create a design using `create_design` command.
2. Analyze/elaborate Verilog files using VCS command.
3. Generate the DFG by calling `compile_design` command.

For more details about the step 2 (VCS command) see to the *VCS®/VCSi™ User Guide*.

For VHDL and MX designs the compilation steps are:

1. Create a design using the `create_design` command.
2. Analyze Verilog files using the `vlogan` command.
3. Analyze VHDL files using the `vhdlan` command.
4. Elaborate the design using the VCS command (optional command).
5. Generate the DFG by using the `compile_design` command.

👉 **Note**

Steps 2 to 4 follow the VCS unified use model for compiling MX designs. The fourth step is optional because Hector can perform this step on its own using the information provided by you in the `create_design` step.

The Hector commands vcs, vlogan, and vhdlan accept many of the same options supported by the similar commands in the VCS-MX simulator. For example, the following options are supported in Hector: (`-pvalue`, `-gfile`, `-gvalue`, `+define`, `+include`, `-f`, `-v` , `-y`). For more information about the use of these switches, see the *VCS® MX/VCS® MXi™ User Guide.* The `-work` option is not supported.

Hector can also read Synopsys Designware foundation components. The steps are described in chapter "Appendix: Using Designware Components" on page 155.

4.5.1 Formal RTL Compilation Options

Hector RTL compiler accepts additional options that impact the formal model (DFG) that is generated. These are passed using `-options <string>` option of the `create_design` command. The list of these options is as follows:

1. `-errorOnMultipleDrivers`. This option tells Hector to produce an error during compilation when multiple drivers are present.

2. `-dware:` Specifies the location where the Designware library analysis results are stored. For more information on using DesignWare libraries, see chapter "Appendix: Using Designware Components".

4.5.1.1 Examples

The following examples show how you can compile designs of different languages in Hector:

4.5.1.1.1 Verilog Only

```
proc compile_impl {} {

    create_design -name impl -top play \
            -clock clk -reset rst –negReset

    vcs play.v

    compile_design impl

}
```

4.5.1.1.2 SystemVerilog

```
proc compile_impl {} {

    create_design -name impl -top play \
            -clock clk -reset rst –negReset

    vcs –sverilog play.v

    compile_design impl

}
```

4.5.1.1.3 VHDL

```
proc compile_impl {} {

    create_design -name impl -top DW01_add_inst

    vhdlan wrapper.vhd DW01_add.vhd DW01_add_cla.vhd

    compile_design impl

}
```

4.5.1.1.4 VHDL

```
proc compile_impl {} {

    create_design -name impl -top test_multpipe \
        -reset RESET -negReset -clock CLK

    vhdlan mult_pipeline.vhd
```

```
        compile_design impl

    }
```

4.5.1.1.5   Mixed Verilog and VHDL (MX)

```
proc compile_impl {} {

    create_design -name impl -top DW01_add_inst

    vlogan  wrapper.v

    vhdlan  DW01_add.vhd DW01_add_cla.vhd

    compile_design impl

    }
```

4.5.2   RTL Language Limitations

4.5.2.1   VHDL

The following VHDL constructs are not supported:

- ❖ Real data types or user data types based on real or containing real.
- ❖ Dynamic allocation.
- ❖ User-written resolution functions.
- ❖ Static sized strings can be used, but are best avoided.
- ❖ Pointers or access types.
- ❖ Additionally, the VPD/FSDB flow cannot handle physical types or multi-dimensional arrays where the dimension is greater than two.

4.5.2.2   Verilog and System Verilog

The following Verilog and SystemVerilog constructs are not supported:

- ❖ Real data types or user types containing real
- ❖ Dynamic allocation (such as string)
- ❖ Classes
- ❖ Unions, packed or unpacked
- ❖ SystemVerilog events
- ❖ Escaped names in top-level ports

4.6   Writing Hector Compatible C/C++ Designs

Hector compiles C/C++ code into a formal model called a DFG. This section provides guidelines on how to write C/C++ code that is compatible with Hector.

4.6.1   Preparing the Source Files

Add a small wrapper function to the C/C++ source code. Its purpose is to:

1. Specify the C/C++ code that should be analyzed and compared to the implementation design.
2. Identify the inputs and outputs of the C/C++ design.

The file containing the wrapper function must include the file `Hector.h` to provide definitions for the following Hector directives.

❖ To specify a variable to be input use the following function:

```
Hector::registerInput(<varname>, <var>);
```

❖ To specify a variable to be output use the following function:

```
Hector::registerOutput(<varname>, <var>);
```

Once you specify these variables, the variables that are already declared will be treated as inputs/outputs. The `<varname>` argument is a string that will be used to identify this variable when mapping between the inputs/outputs of the two models is performed in the Hector command script. The `<var>` is the variable.

All input and output declarations must appear before the `Hector::beginCapture()` directive. Currently, if an input/output declaration appears after the `Hector::beginCapture()` directive, Hector will quietly ignore it.

The following directives must be included to denote which parts of the C/C++ code are to be analyzed by Hector and compared to the RTL model. The code bracketed by these two directives defines the functionality that will be compared to the RTL model.

```
Hector::beginCapture();

Hector::endCapture();
```

When the `beginCapture()` directive is encountered, all the live variables are collected into a set. This includes any global variables, any statically declared variables (either within or outside a function) and any variables currently on the program stack. Any variables in this set that are not declared as either inputs or outputs to Hector (using the directives above) will be treated as state holding variables. When the `endCapture()` directive is encountered, it signals that the computation of the SL model is complete.

The `beginCapture()` and `endCapture()` directives mark the begin and end of a transaction. It is very important that initialization of the model happens at the right place. Consider a class `Compute` that initializes its data members in the constructor. It also implements a function `run()` that performs the computation. Note that the following two scenarios lead to rather different formal models:

4.6.1.1   First Scenario

```
Compute c;

Hector::beginCapture();

c.run();

Hector::endCapture();
```

In this scenario, the `beginCapture()` call turns all variables in `Compute c` into registers. The registers are initialized fully symbolic, that is, after the `beginCapture()` they become unconstrained and lose their current value. The only exception are variables that are declared as `const` because those are guaranteed not to change during the execution and do not need to become registers. This scenario would be used if information can be carried over from one transaction to the next (through data members of `Compute`). The transaction itself calls `c.run()` and writes into the data members of `c`, which can then be read by subsequent transactions. Initialization of `Compute c` happens only once at the very beginning, that is, before the first transaction.

4.6.1.2   Second Scenario

```
Hector::beginCapture();

Compute c;
```

```
    c.run();
    Hector::endCapture();
```

In this scenario, the initialization of the `Compute c` class members happens within the transaction. This means that the transaction is fully self-contained and no information is carried over to the next transaction (at least not through any of the data-members of `c`). No registers will be generated at the `beginCapture()` call. Initialization will happen at the beginning of every transaction, not just once.

4.6.1.3   Simple Example

In the simplest cases, the C/C++ code to be analyzed by Hector is conveniently encapsulated in a single function and the user-created wrapper simply needs to call this function. This case is illustrated in the Quickstart example discussed in the "Quick Start Guide" chapter. All the C/C++ functionality that is to be compared to the implementation model is contained in the `compute` function.

4.6.1.4   Intermediate Example

In more complex cases, the code to be analyzed is encapsulated in a single function, but the number and/or names of the arguments to that function do not match the number/name of the arguments in the RTL model. Consider the following example:

Original C++ Function:

```
struct fpn {
    int mantissa;
    int exponent;
};
void floating_divide(fpn dividend, fpn divisor, fpn& quotient)
{
// floating point divide code here.
}
```

Original RTL Model:

```
module fp_divide(mantissa_dividend,
            exponent_dividend,
            mantissa_divisor,
            exponent_divisor,
            mantissa_quotient,
            exponent_quotient);
    input [24:0] mantissa_dividend;
    input [16:0] mantissa_divisor;
    input [24:0] exponent_dividend;
    input [16:0] exponent_divisor;
    output[24:0] mantissa_quotient;
    output[24:0] exponent_quotient;
```

```
            // RTL code for floating point divide

        endmodule
```

There are three arguments to the `floating_divide` function in the C++ code and six ports on the RTL model. The user-created Hector wrapper code (as shown subsequently) designates the individual *struct* members as inputs/outputs so as to match the RTL names exactly and then calls the `floating_divide` function.

C++ Hector Wrapper Function:

```
    void wrapper()

    {

        struct fpn dividend, divisor, quotient;

        // create and declare the inputs and outputs

        Hector::registerInput("mantissa_dividend", dividend.mantissa);

        Hector::registerInput("exponent_dividend", dividend.exponent);

        Hector::registerInput("mantissa_divisor", divisor.mantissa);

        Hector::registerInput("exponent_divisor", divisor.exponent);


        Hector::registerOutput("mantissa_quotient", quotient.mantissa);

        Hector::registerOutput("exponent_quotient", quotient.exponent);


        Hector::beginCapture();

        floating_divide(dividend, divisor, quotient);

        Hector::endCapture();

    }
```

The naming of inputs/outputs shown in the above example is used for convenience during the mapping process. It is not required to name the inputs/outputs in C/C++ model to be exactly same as the RTL model, however, this will make the setup easier.

The most complex cases involve situations where the core functionality (to be compared with the implementation model) is not completely separated from the system level testbench code. These cases will often require someone familiar with the system level model to create a wrapper that cleanly separates the test bench code from the model.

4.6.2   Using Hector::registerInput and Hector::registerOutput

You can tell Hector which variables should be marked as inputs or outputs using the functions `Hector::registerInput()` and `Hector::registerOutput()`. Since the declarations for both functions are the same, only the `Hector::registerInput()` function is described in this document.

The `Hector::registerInput()` is available in the following forms:

```
    template <class T> void registerInput(const char* name_p, T& obj);

    template <class T> void registerInput(const char* name_p, int i1, T& obj);

    template <class T> void registerInput(const char* name_p, int i1, int i2, T& obj);
```

Example:

```
int b;

Hector::registerInput("b", b);
```

The object is passed as a reference so that Hector can determine the bit width. Passing a pointer will cause a compilation error. The versions with additional integers allow registering inputs in loops. The name can contain placeholders %1, %2 that refer to the integers and will be replaced by the integer values.

Example:

```
int arr[4][3];

for (int i = 0; i < 4; i++) {

    for (int j= 0; j < 3; j++) {

        Hector::registerInput("arr_%1_%2", i, j, arr[i][j]);

    }

}
```

The placeholders can be used multiple times in the name (example arr_%1_%2_%1). It is an error to refer to an index that does not exist (example %0 or %4). It is also an error to have a % following anything else but a number. The name itself should be a legal C++ variable name. It can only contain the characters 0-9, a-z, A-Z and _ (it cannot start with a number). The % is only allowed for placeholders. The name cannot be empty. Any violation will lead to an error.

Errors will also be generated for the following violations:

1. The `Hector::registerInput()` always requires that the program contains calls to `Hector::beginCapture` and `Hector::endCapture`. Furthermore, all `Hector::registerInput` calls need to happen before the `Hector::beginCapture` call.

2. There cannot be multiple calls to `Hector::registerInput` with the same name. Inputs must be named uniquely.

3. If an input name in `Hector::registerInput` is the same as an output name in `Hector::registerOutput`, by default an error is issued. Note that it is legal in principle to have inputs and outputs with the same name. However, if assumptions and lemmas are used, it can become confusing to find out when the input and when the output is being used.

4. The same variable / object cannot be used in multiple `Hector::registerInput` calls. For example, the following is illegal:

```
struct foo { int a; int b; } myfoo;

Hector::registerInput("foo",  myfoo);

Hector::registerInput("foo_a", myfoo.a);
```

The integer `myfoo.a` would be part of two inputs, which is illegal.

> **Note**
> This restriction does not hold for outputs. Multiple calls of `Hector::registerOutput` can refer to the same object.

4.6.2.1   Registering Variables with Complex Types

The variables/objects that can be used in `Hector::registerInput` can have arbitrary types. Class objects, structs, arrays are perfectly valid. By default, `Hector::registerInput` will mark the entire object as an input. Sometimes, this behavior is not desirable. For example, when registering SystemC data-types, usually

the entire object should not become an input but only a single field of the class. In user provided classes, the same effect can be achieved by explicitly adding a `registerInput` method to the class. The `registerInput` method is responsible for registering just the necessary field.

Example:

```
class myclass {
    void registerInput(const char* name_p, bool directly = true) {
        Hector::registerInput(name_p, d_data);
    }
    int     d_type;
    int     d_data;
    char    d_name[10];
};
myclass myobject;
Hector::registerInput("myob", myobject);
```

The `Hector::registerInput` function automatically determines if the object or any of its base classes has a method named `registerInput`; if yes, it will be called. Otherwise, the default behavior of registering the entire object will take effect. The `directly` parameter can be used to distinguish calls to the `registerInput` method through `Hector::registerInput` from direct calls. If the method is called through `Hector::registerInput`, `directly` is set to false.

The `registerInput` method is already supplied by the SystemC data-types and the `HectorBV` class, which means that all classes that derive from SystemC data-types or the `HectorBV` class should be handled automatically.

4.6.2.2   Support for Unions

`Hector::registerInput` does not support unions. If the type of the registered object is a union, Hector will error out in the compilation. There is an easy workaround for this; instead of registering the entire union, just register the largest field of the union.

4.6.3   Understanding Mismatches between Hector and GCC

The C/C++ standard leaves a significant number of behaviors undefined or implementation-defined. Any of these behaviors can lead to differences between Hector and GCC. In practice though, only very few of them are encountered more frequently. Most of them are related to the difference in 32-bit versus 64-bit compilation.

Hector's front-end is 32-bit, that is, if a discrepancy occurs, the first course of action is to use GCC with the `-m32` option to ensure a 32-bit build.

4.6.3.1   Different Results when Using *long int* Data-type

The *long int* data-type has a different bit-width when compiled with 32-bit GCC versus 64-bit GCC. On a 32-bit machine it is 32-bits whereas on a 64-bit machine it is 64 bit. Hector always behaves like a 32-bit GCC, that is, a long will be 32-bits.

The *long* data-type is not portable and should be avoided at all cost. If guaranteed bit-widths are required, use the `uint8_t`, `uint16_t`, `uint32_t` and `uint64_t` data-types defined in `stdint.h/cstdint`.

4.6.3.2   Different Results when using *bool* Data-type

The *bool* data-type must NOT be used to describe 8-bit values. In general, it is not legal to assume that a *bool* will be represented in 8-bits. In fact, it is completely compiler dependent of how an integer values is converted to a *bool*. Some compilers convert non-zero values to *true* and zero to *false*. Others (including GCC in most cases) ignore all bits but the least significant one. The least significant bit alone determines whether the value is interpreted as *true* or *false*. Hector tries to mimic GCC by also only using the least significant bit of a *bool*.

4.6.4   Hector's C/C++ Compiler Assumptions

Hector's C/C++ front-end is configured for 32-bit architecture (or 32-bit compilation on a 64-bit architecture). In particular the sizes of various common data-types in Hector are as follows:

❖   `char` – 1 byte

❖   `short int` – 2 bytes

❖   `int` – 4 bytes

❖   `long` – 4 bytes

❖   `long long` – 8 bytes

❖   `pointers` – 4 bytes

Caution

A common mistake is to assume that the size of *long* is 8 bytes.

4.6.5   Compiling Loops

During DFG creation Hector must unroll all loops in the C++ program. For loops with static bounds, Hector can usually determine the number of *unrollings* with no user guidance. If the number of loop iterations exceeds a preset threshold, Hector will produce the following error:

```
*** Error SCC-002: Maximum number of loop iterations reached (500). Termination
check was successful.

Maximum number of iterations is determined by option
'_hector_sym_maxiter_allgood'.
```

This error message means that Hector tried to unroll the loop for 500 times, but the loop did not terminate. If the loop really has a static bound greater than 500, then you can set the Hector variable `_hector_sym_maxiter_allgood` to a larger fixed upper bound. If there is a significant amount of computation in the loop body, this can lead to DFG size explosion and poor performance.

For loops with data dependent bounds, Hector still tries to find the maximum unrolling limit. If no limit can be found after 50 iterations, the following error will appear:

```
*** Error SCC-001: Cannot detect if loop terminates because check always came back
inconclusive (ran 50 iterations).

Maximum number of iterations is determined by option
'_hector_sym_maxiter_allfail'.
```

This means that Hector is not able to determine when the loop terminates. In this situation, insert a pragma in the C++ code to indicate the required loop unrolling limit as follows:

```
#pragma max_iterations <number>
```

This pragma directs Hector to unroll the loop for the specified number of times. Hector will create a lemma to formally prove that the iteration limit is large enough. Specifying an iteration limit that is too large will cause unnecessary growth in the DFG size and a reduction of Hector's performance.

In some cases the upper bound on a loop iteration count is determined by a variable that is constant after initialization. Declaring that variable with the *const* specifier may help Hector determine that the loop bound is fixed.

Example

The following example contains a *while* loop. The compilation step will fail without the `max_iterations` pragma. The upper bound on the number of iterations in the while loop is 63. With the `max_iterations` pragma the compilation step finishes successfully.

```
typedef unsigned long long uint64;

typedef int int32;

uint64 a, b, significand, result;

int32 exponent;


int compute () {

    significand = a*b;

    if (significand != 0) {

        #pragma max_iterations 63

        while ((significand >> 63) == 0) {

        exponent-=1;

        significand <<= 1;

        }

    }

    result = significand;

}
```

4.6.5.1   Debugging Long Compilations

Compilation can take a long time if the program has loops with many iterations. In order to find out the cause of long compilation times, use *Control-C* at any time. Hector prints a stack trace describing the location of the statement that is currently being compiled. Compilation itself is not interrupted and continues automatically. Only if you use Control-C two times in a short period (less than half a second), the compilation is interrupted and Hector returns to the command prompt.

4.6.6   Using Assertions and Checks for Malformed Programs

User inserted assertions in form of `assert` statements are automatically checked during compilation. If the compiler can determine that an assertion is violated, an error message is printed. If it can figure out that an assertion can never be violated, the assertion will be ignored. In most cases, however, the compiler itself will not be able to determine if an assertion can be violated or not. In those cases, the compiler will put the assertion into the DFG and defer the check to the back-end. The assertions will show up as separate lemmas in the final proof.

In addition to user-generated assertions, the Hector C/C++ compiler will conduct its own checks to make sure that the program is well-formed. For example, according to the C++ standard, out-of-bound accesses of arrays are illegal and the behavior of such a program is undefined. The Hector compiler performs out-of-bounds checking for each array access and will print an error message if such an illegal access happens. In cases where the compiler cannot conclusively prove that no out-of-bounds access is possible, the check will be deferred to the back-end. The bounds checks will show up as separate lemmas in the final proof.

Other examples of malformed program checks that are automatically performed by Hector are division by zero checks and checks for shifts where the value of the second operand exceeds the bit-width of the first operand (for example shifting left a 32-bit value by 40).

In general, any behavior that is undefined according to the C++ standard needs to be checked. There are, however, two categories of undefined behavior:

- ❖ Behavior that can cause programs to crash
- ❖ Behavior that in practice may return an undefined value but does not usually crash the program

Bounds checks and division by zero belong to the first category, whereas excess shift checks belong to the second category. The first category is always checked in Hector. The compiler's behavior regarding the second category depends on the Hector variable `_hector_sym_strict_malformed_checks`. If it is set to true (which is the default) shifts are always checked, too. If it is set to false, such shifts will only produce undefined (fully symbolic) values but no additional checks will be created for them. Sometimes, this is desirable because if the program can guarantee that the undefined value does not propagate to the outputs, it might still execute just fine.

4.6.7   Using Black-Boxing to Ignore C Functions

If you know that certain functions are NOT in the cone of influence of the outputs and global variables, then those functions can be ignored by Hector by using the black-boxing feature. This is particularly useful, when you do not have the implementation of certain library functions and you know that these functions do not impact the functionality of the design. For example, C++ I/O stream functions (such as `cout`, `printf`, `cerr`) can be easily ignored using black-boxing.

You can provide a *ignore-list* of *side-effect-free* functions. A function is side-effect free if it does not modify global variables.

☞ **Note**
    Hector DOES NOT check the ignore-list for the absence of side-effects.

Hector treats the return value and the pointer arguments of a black-boxed function as free inputs. This is a safe over-approximation that guarantees that if the function return value or its pointer arguments are in the cone of influence of outputs, then the solvers will consider all possible values for them.

Avoid the following two scenarios as it could lead to incorrect verification:

🛑 **Caution**
    The black-boxed function has side effects; in this case, Hector may produce false equivalence or false counter-example.

🛑 **Caution**
    The black-boxed function has NO side effects, but the return value or the pointer arguments of the function is in the cone-of-influence of the outputs, in this case, Hector may provide a false counter-example, but never a false equivalence proof.

You can provide a list of ignore functions using the following directive in the TCL script.

```
    ignore_functions    <list_of_function_names>
```

The function name is provided with its mangled name or by using a regular expression.

Example

Consider the following example with two functions, one top function *dut* and a debug function *print*. Let us assume that the source code for print is not available. However, *print* does not influence the output or the global variables of the program, as such it is safe to ignore it. Also, note that if the function is not ignored, then Hector will not be able to compile the example.

```
#include <iostream>
void print ();
int  dut (int ina, int inb, int idx) {
    res_t out;
    out = in_a + in_b;
    out  += idx;
    print();
    return out;
}
```

In TCL script file add the following:

```
proc compile_spec { } {
    create_design -name spec -top main
    ignore_functions "print"
    cppan –Iinclude –DCHECKFP foo.cc
    compile_design spec
}
```

4.6.7.1   Limitations

🛑 **Caution**

> This technique simplifies the compilation phase, however Hector can still fail if the front-end does not parse some constructs (such as Run Time Type Information (RTTI) constructs).

4.6.7.2   Troubleshooting

Question: Hector is still looking for (or compiling) the ignored function?

💡 **Hint**

> The function name is probably not matching. Try using a regular expression like `*function_name*`. Note that this regular expression may match more than one function. Ideally, you should use the full mangled name of the function to avoid any confusion.

4.6.8   Using Floating Point

Hector supports floating point data types (*float* and *double*) in the C/C++ code. The result of floating point operations depend on the rounding mode. Hector supports four IEEE rounding modes:

1. Round to Nearest Even

  2. Round to Negative Infinity

  3. Round to Positive Infinity

  4. Round To Zero

The default rounding mode in Hector is *Round to Nearest Even*. Hector generates a formal model for floating point operations by making use of the *SoftFloat* library.

4.6.8.1   Setting Rounding Mode

You can change the default rounding mode by using the following function in the C/C++ code,

```
Hector::setFloatRoundingMode(rmode_t);
```

where, `rmode_t` can take one of the following four values.

  1. `Hector::FloatRoundNearest` (default)

  2. `Hector::FloatRoundNegInf`

  3. `Hector::FloatRoundPosInf`

  4. `Hector::FloatRoundZero`

When using floating point computations in Hector, there is a chance that Hector's floating point computation may not always agree with the result of the program when run on an x86 machine. This underlying cause could be that floating point computations are not always deterministic when using GCC on an x86 machine.

Non-deterministic means that compiling the same floating point program with different GCC options (for example, optimized and debug or in 32-bit mode and 64-bit mode) can exhibit the same differences in the two executables.

In Hector, single precision floating point operations are ALWAYS carried out in single precision according to IEEE standard. Likewise, double precision floating point operations are ALWAYS carried out in double precision also according to the IEEE standard.

This is NOT the case in x86 architectures. What happens in a single precision operation is that the operands are copied into the internal floating point processor registers. The internal registers always use extended precision, so the single precision operands are converted to extended precision. Then, the operation is carried out in extended precision. When the result is written to memory as single precision result, the extended precision value is converted to single precision and the result is written. This by itself is perfectly legal according to the IEEE standard and there is no difference between this procedure and performing the operation natively in single precision.

Where the difference comes into play is if the result is NOT written to memory but instead is kept in extended precision in the internal register. The compiler optimizes subsequent floating point operations such that the results are kept in the internal registers as long as possible. If this happens, there is NO conversion to single precision happening until the final operation that writes the result back into memory. So, the result can vary depending on how many subsequent floating point operations the compiler can carry out in the internal registers before the result is written to memory.

One way to avoid this source of non-determinism is to use the `-ffloat-store` option in GCC. This option forces the compiler to write the result of a floating point computation to memory immediately. The program does run slower but the result is more predictable. This is especially important if the result used in a bit-accurate comparison against an RTL model.

4.6.8.2   Using Not a Number (NaN)

Caution

The bit representation of a NaN in Hector is not canonical. When NaNs are generated in different places in the C/C++ code their bit representation is not guaranteed to be identical.

4.6.9   Coding Guidelines

This section describes a few coding guidelines that are, in general, helpful for generating better formal model using Hector.

4.6.9.1   Using Loops with Static Bounds

It is recommended that you write loops with static bounds. Even a loop without static bounds can often be rewritten to a loop with static bounds if the maximum number of iterations is known. For example, consider the following loop:

```
for (int i = 0; i < k; i++) { ... }
```

This loop can be transformed into the following loop if it is known that k is always less than 10.

```
for (int i = 0; i < 10; i++) {

    if (i >= k) break;

    ......

}
```

The following example counts the number of leading zeros in a 32-bit significand.

```
// significand is of type unsigned int

int count= 0;

while (significand && (significand >> 31) == 0) {

    count++;

    significand <<= 1;

}
```

This while loop can be converted to a loop with static bounds as follows:

```
int count= 0;

for (int i = 0; i < 32 && significand; i++) {

    if ((significand >> 31) != 0) break;

    count++;

    significand <<= 1;

}
```

4.6.10   Writing Portable C/C++ Code

The following sections provide examples of portable C/C++ code that you can use:

4.6.10.1   Using Exact-width Integer Types

☞ **Note**

It is recommended that you always use types that have known size (#bits), regardless of 32-bit or 64-bit architecture (or 32-bit compilation on a 64-bit machine). For example, you can use the types defined in `stdint.h` such as `int8_t`, `int16_t`, `int32_t` etc. to get integer types with bit widths 8, 16, 32, respectively.

4.6.10.2   Avoiding *Long* with GCC Compiler on Linux

🖐 **Caution**

When writing code that is portable across both 32-bit and 64-bit Linux machines (using GCC), do not use *long* data types. This is because a long is 4 bytes on 32-bit machines (or 32-bit compilation on a 64-bit machine) and 8 bytes on 64-bit compilation.

4.6.10.3   Using Template Classes

Consider the following example:

```
//Arbitrary sized integer with maximum size 32

class newInt{

    private:

        int len;

        int val;

    public:

        newInt(int l) : len(l), val(0){};

        init(){

                val = 0;

                for (int i = 0; i < len; i++)

                val |= (1 << i);

        }

};
```

This class implements an arbitrary sized integer up to a maximum size of 32. It keeps the length of the integer in the variable `len`. However, this variable is only initialized once in the constructor. During construction of the formal model Hector may not be able to figure out that `len` is not modified and it is a constant. In this case `len` will be treated as a symbolic variable and the for-loop in `init` will become unbounded.

💡 **Hint**

This problem can be fixed in two ways. One is to make the variable constant using the *const* keyword. Another and more favorable one is the use of template. For example, the above class can be rewritten in this form:

```
//arbitrary sized integer with maximum size 32

template<int W>

class newInt{

    private:

        int val;
```

```
    public:

        newInt() : val(0){};
        init(){
                val = 0;
                for (int i = 0; i < W; i++)
                        val |= (1 << i);
        }
    };
```

4.6.10.4   Using Arbitrary Precision Datatypes

Languages such as C/C++ do not provide support for arbitrary precision datatypes. This means that you need to write your own code to implement various operations such as 128-bit multiplication or 128-bit addition in C/C++. In such cases it is recommended that you follow a uniform methodology throughout the code. For example, you can implement a template based class as shown in the previous section. Some customers have reported success using Mentor's Algorithmic C datatypes in Hector.

Alternatively, you can use SystemC datatypes. For more information on support of SystemC datatypes, see section "Support for SystemC Data Types".

4.6.10.5   Using C/C++ Library Function Calls

In general, all source code must be available to Hector (except as described in section "Using Assertions and Checks for Malformed Programs"). This prohibits the use of most C/C++ library function calls unless the source code is available and compiled with the design files. However, the source code is not required for following C++/C library functions:

❖ `void *     memset ( void * ptr, int value, size_t num );`

❖ `void *     memcpy (void * s1, const void * s2, size_t n);`

❖ `uint16_t   ntohs (uint16_t netshort);`

❖ `size_t     strlen ( const char * str );`

Also the source code is not needed for the following math functions:

❖ `double floor (double x)`

❖ `float floorf (float x)`

❖ `double fabs (double x)`

❖ `float fabsf(float x)`

❖ `double pow (double base, double exp)`

❖ `floatpowf(float base, float exp)`

❖ `double modf (double x, double *inpart)`

❖ `float modff(float x, float *inpart)`

❖ `double ldexp(double ad, int n)`

The following are the supported native floating point intrinsic functions:

❖ `floor, floorf, floorl`

❖ `ceil, ceilf, ceill`

❖ `trunc, truncf, truncl`

- ❖ abs, labs, llabs
- ❖ fabs, fabsf, fabsl
- ❖ modf, modff
- ❖ ldexp
- ❖ __isinff, __isinf, __isinfl
- ❖ __isnanf, __isnan, __isnanl

The pow function is supported, and can be used in the following three different implementations:

- ❖ pow, powf

   These are floating point pow operations, and they are close to the IEEE standard. They should only get used in cases where the RTL also is a floating-point pow operation.

   If it still should be used in compilation, add this to cppan:

   ```
   cppan design.cc $env(HECTOR_HOME)/local/pow/Hector_pow.cc
   ```

- ❖ pow

   A simpler floating point double version that computes a*a*a*…*a. This does not have the precision of 1, but is good enough for simple pow operations, like powers of 2, including negative powers like this: pow(2, -3);

   To use it, add this to cppan:

   ```
   cppan design.cc $env(HECTOR_HOME)/local/pow/simple_pow.cc
   ```

- ❖ ipow

   This is an integer version of pow. The second operand must be positive.

   To use it, add this to cppan:

   ```
   cppan design.cc $env(HECTOR_HOME)/local/pow/ipow.cc
   ```

In general, the pow function should be avoided as it results in very hard formal models. The pow function should never be used to compute a mask. It is really bad coding practice and it is extremely inefficient, even in simulation. Such cases should always be replaced by (1 << n).

Due to the complexity/precision issues of "pow", it is not supported by default. Instead, you should make a conscious decision if they really need it, and decide which implementation type it should be.

4.6.10.6   Using Floating Point Datatypes

Floating point (FP) datatypes and operations on FP datatypes should be used only when they are necessary. This is because the logic required to implement FP operations is more complex than corresponding integer operations and increases verification complexity. If you compare a model using FP arithmetic with one using integer arithmetic, Hector may not be able to prove equivalence.

For example consider the two functions `mult1` and `mult2` below:

```
float mult1(uint8 a, uint8 b) { return ((float) a) * ((float) b)); }

float mult2(uint8 a, uint8 b) { return (float) (a*b); }
```

Both `mult1` and `mult2` produce identical results, because the precision of the float type is sufficient to exactly represent the input uint8 type. If rounding or truncation were a possibility, the functions might not be equivalent. Function `mult1` contains three floating point operations due to casting of *a* to a float, casting of *b* to a float and multiplication of two floats. The Hector formal model dataflow graph (DFG) for `mult1` has 674 nodes. On the other hand `mult2` has only one floating point operation that casts the product of *a* and *b* to

a float. The DFG for `mult2` has only 248 nodes. The function `mult2` is more efficient than `mult1` for verification.

4.6.10.7   Debugging Code

You should disable function calls that are not useful in a formal analysis, for example, `debug()`, `print()`, or `statistics()` calls. Although this is not a requirement (if the implementation of these functions is available), it makes Hector's task easier. One way to do this is by using the black boxing feature as described in section "Using Assertions and Checks for Malformed Programs". Another way is to use `#ifndef HECTOR` directive to exclude code that is not relevant to formal analysis.

Note
Hector's compiler implicitly defines HECTOR.

4.6.11   C/C++ Features Not Supported

Caution
Avoid the following language features/constructs in their C/C++ code as they not supported by Hector:

❖ There is no support for virtual inheritance. The compilation step command will print an error message and Hector will abort if these constructs are encountered.

❖ There is limited support for dynamic memory allocation using new/delete. Matching new and delete calls are supported if they both occur between the `beginCapture()` and `endCapture()` directives. In this circumstance, the allocated memory is not treated as a state holding variable (and therefore does not need to be mapped to memories in the other model). Malloc and free are not supported.

❖ A limited form of (static) recursion is supported. If the depth of recursion exceeds M (default value of 40), the analysis will stop with an error message. You can set the Hector variable `_hector_sym_maxstackdepth` to a numeric value to alter the value of M.

Note
This Hector variable controls the depth of the entire function call stack, not just the recursive calls.

❖ There is no support for Standard Template Library (STL) and `std::string` classes.

❖ There is no support for dynamic casts and constructs depending on runtime type information (RTTI) such as `typeid()`.

❖ Hector has limited support for pointer arithmetic. Operations on unitialized pointers or operations that cause a pointer dereference outside of a statically declared object, will cause errors during design compilation.

4.7   Writing Hector Compatible RTL Designs

The following are the limitations on the RTL model:

❖ It must be a single clock design; or, if the design contains multiple clocks, they must be derived from a single clock in a synchronous manner (independent of any other signals in the design except the reset). Hector does not support using *assume* expressions to define clock behavior.

❖ The model must be synthesizable by *DesignCompiler*.

4.8   Hector Tips

Use the following tips to obtain better performance from Hector:

❖ If there are errors during the compilation step, you can just re-invoke the command after editing some source files, that is, you do not have to exit and then restart the Hector shell.

❖ Use TCL procedures to encapsulate common command sequences, such as those required to create, analyze, and compile designs. Invoking those procedures from the interactive command prompt can save extra typing.

4.9   Using Predefined Hector Variables

Predefined Hector variables can be used to control the execution of Hector on a given problem. These variables must be set to a new value by using the `set` TCL procedure. That is for a Hector variable `<var>` following TCL procedure must be called to set the value of `<var>` to a value `<val>` as follows:

```
set_<var> <val>
```

The value of a Hector variable can be obtained by using the corresponding `get` TCL procedure. The usage is as follows:

```
get_<var>
```

For example, in order to set Hector variable `user_assumes_lemmas_procedure` to `foo` the following TCL procedure must be called.

```
set_user_assumes_lemmas_procedure "foo"
```

In order to get the current value of `user_assumes_lemmas_procedure` the following TCL procedure must be called:

```
get_user_assumes_lemmas_procedure
```

The list of predefined Hector variables can be found in section "Hector not Converging on Problems".

5 **Setting up the Equivalence Problem**

This chapter describes how you can set up an equivalence problem in the following sections:

- ❖ "Definition of Transaction Equivalence"
- ❖ "Transaction-Relative Time"
- ❖ "Assumptions and Lemmas"
- ❖ "Controlling Model Initialization"
- ❖ "Composing the Equivalence Problem"

5.1   Definition of Transaction Equivalence

Hector is designed to test for transaction equivalence between two designs. In each design, you must be able to define a *transaction*, which is a unit of computation that produces a specific set of output values from a specific set of input values. In a combinational circuit, or C function, outputs are computed from input values in essentially zero time. In clocked sequential circuits, a transaction may span multiple clock cycles. In pipelined designs, multiple transactions may be in process at the same time. A primary value of Hector is the ability to model transaction equivalence between transactions of different latency.

5.1.1   Bit Accuracy

Hector equivalence requires bit accuracy. This means that the specific bit patterns in multi-bit variables or bit-vectors must exactly match for two transactions to be considered equivalent. In practice, this may mean accommodation of different data widths in the transaction comparison, or adding additional logic to one of the designs to obtain bit-level equivalence.

5.1.2   Maximum Transaction Latency

All transactions modeled in Hector must have a fixed maximum latency (in terms of clock phases). Transactions may have variable (data dependent) latencies as long as the upper bound is known. If the maximum latency is very large, Hector may have difficulty compiling or solving the problem. The internal design representation is unrolled (duplicated) for each clock phase, so a large latency combined with a large design size can cause a size explosion. Latencies of a few 10's of cycles are usually fine, but hundreds of cycles are usually too much.

5.1.3   Mapping of Transaction Inputs, Outputs, and Memories

To model the equivalence problem between two design representation, Hector must know the correspondence between transaction inputs and outputs. Mappings must be provided which indicate that inputs to the two designs must be assumed equivalent, and where outputs are produced that must be

checked for equivalence. In addition to input and output ports, any memory elements that are read or written during a transaction must also be described in the Hector setup.

By default, Hector makes no assumptions about transaction inputs. All representable bit patterns can be applied to input ports, and any arbitrary value may be present in registers and memory locations. For designs which can only accept subsets of the representable input values, or when transactions can be initiated only when registers or memories are properly initialized, the Hector setup will need to reflect those restrictions. Otherwise, false equivalence failures (discrepancies) will be reported.

> **Note**
> Not all memories require mapping between the two designs, and some memories may only be present in one design. Only those memories that carry information from one transaction to the next require mapping.

5.1.4 Sequences of Transactions

It is also important to understand that Hector only verifies a single transaction. If no assumptions are made about the starting state of the transaction, then that single transaction is representative of an unbounded series of transactions. However, if there are restrictions on the transaction starting state, or the results of a transaction is dependent on the results of previous transactions, the problem is more complex. In this case we must define an *inductive invariant* or set of states from which is it legal to initiate a new transaction. To prove that an unbounded sequence of transactions is valid, we must prove that the design always returns to a legal transaction initiation state after every transaction (assuming that the transaction also started in a legal state). You must provide this *inductive invariant* state, Hector does not find it automatically.

5.1.5 Initial States

The initial values of storage elements in a design are important in some situations and not important for others. For example, some transaction-oriented designs are dependent on state values established by prior transactions, or just on configuration registers that must hold specific values. Other state values, such as pipeline stage registers, should not impact the function of a transaction.

Hector can execute proofs in two different modes. The default mode, and also the most *safe* mode, is to prove properties without considering the initial states of registers or memories. Proofs obtained in this mode guarantee that there is no coupling between transactions through state values. This is the mode used by the `solve` and `solveNB` commands.

A second proof mode is available when explicit initial states must be considered in the proof. Section "Controlling Model Initialization" discusses how you can control the initial states of transactions. Even when those steps are taken, you must also use `solve_init` or `solveNB_init` to make sure the initial state is considered during the proof.

5.2 Transaction-Relative Time

Hector references transaction-relative time delays using the definition of clock phases shown in the Figure 5-1. This labeling of transaction phases makes it possible to check for equivalence between designs with different latencies.

5.2.1 Combinational (Non-sequential) Models

For combinational RTL (or C++ functions), calculation of outputs from inputs does not have any latency. For example, if C function inputs are established in phase 0, the function output may also be tested in phase 0. When different inputs are applied in phase 1, the new outputs also available in phase 1. Unless explicitly constrained by the user, all inputs can be driven with a fresh value on each clock phase.

Therefore, if the design is purely combinational, the following setup (phase 0 formulation):

```
assume a(0) == 1
assume b(0) == 2
lemma out(0) == 3
```

yields the same result as this one (phase 1 formulation):

```
assume a(1) == 1
assume b(1) == 2
lemma out(1) == 3
```

For a C model, it is safe to use either the phase 0 or the phase 1 formulation, if assumes/lemmas refer only to inputs/outputs. If lemmas/assumes refer to state variables (see section "C++ State Variables - Establishing an Initial State Value"), then the phase 1 formulation is safer. The C++ model has an implicit clock, and the global variables are updated at the posedge of this implicit clock, which happens during the transition from phase 1 to phase 2. Therefore, if a lemma refers to a state variable, use phase 1 for input/output references, and use phase 2 for state variable references. Assumes on phase 0 has no effect on the value of state variable.

5.2.2   Sequential Models

For sequential models, it is best to always use the phase 1 formulation.

The clock waveform is high (phase 0), low (phase 1), high (phase 2), low (phase3) and so on. So, if the design is posedge triggered, the assume must be in phase 1 because that's the value that is sampled into the register of the design at the posedge between phase 1 and phase 2. The value then appears in the register in phase 2. If the assume was only in phase 0, it does not have any effect because the register samples the input at phase 1, which is unconstrained. So, this does not generate false positives, but it is likely to generate false negatives because the assumption does not have any effect.

5.3   Assumptions and Lemmas

At the lowest level, the proofs in Hector can be carried out by writing assumptions and lemmas. In this section, the syntax of assume and lemma commands is described. An example is also provided showing their use.

You can specify assumptions on RTL signals and/or C-variables and lemmas which need to be proven. Assumptions are specified using the following command:

```
assume <name> = <expression>
```

Proof obligations or lemmas are specified using the following command:

```
lemma <name> = <expression>
```

The argument `<name>` is an arbitrary unique name assigned to the assumption or lemma (for better identification). The `<name>` is optional, but Hector will assign a name if you do not. The `<expression>` is a boolean expression in an extension of Verilog syntax (and semantics). When Hector runs the proof, it proves the validity of each lemma under all assumptions. If a lemma fails, Hector can generate a counter-example trace for it using `simcex` command.

Signals/variables which are used in the `<expression>` usually have a prefix that indicates whether the signal is in the specification or the implementation. If it is a variable in the specification, it has the prefix `spec.`. If it is a signal in the implementation, it has the prefix `impl.`. In C++ to RTL equivalence checking C++ is conventionally marked as the specification and the RTL as the implementation.

In addition to that, all proofs are run on a model of the design where both the specification and the implementation design are unrolled in time. This makes it possible to relate variables/signals at different

time steps with each other. The phase of a signal is given in parentheses after the signal name. The time-step is always a positive number and refers to a half clock cycle.

For example, to refer to variable `myvar` in the C-code at phase 0, use `spec.myvar(0)`. To refer to signal `mysig` in the RTL-code at phase 2, use `impl.mysig(2)`.

Assumptions may be replicated for the whole duration of the proof using the `-always` option. If `-always` is specified, then at least one signal in the expression is required to not have a phase specified. The original expression is replicated once for each phase in the proof. Those signals which did not have a phase in the original expression are given the current phase in the replication.

```
assume a1 = -always (spec.d(0) == impl.d)
```

Is the same as:

```
assume a1 = (spec.d(0) == impl.d(0)) &&

(spec.d(0) == impl.d(1)) &&

(spec.d(0) == impl.d(2)) &&

(spec.d(0) == impl.d(3)) &&

…
```

Some examples of assumptions and lemmas are as follows:

```
assume a1 = (spec.ain(1)   == impl.ain(1))

assume a2 = (spec.bin(1)   == impl.b(1) + impl.c(3))

assume a3 = -always (impl.en == 1'b1)

lemma  l1 = (spec.cout(1)  == impl.cout(4)[31:0])
```

✋ Caution

Hector uses a two-state logic system, hence cannot handle *X* in assume/lemma expressions. For example, following type of assumption is not supported.

```
assume (impl.mysignal_4a(0)   ==   1'bx)
```

If you want to specify that `mysignal_4a` at phase 0 is not initialized, it can be implicitly expressed by not specifying any constraint/assumption on that signal at phase 0.

5.3.1   Using Assumptions and Lemmas in a Proof

The assume and lemma commands must be placed inside a TCL procedure. You must specify the name of the TCL procedure by setting the `user_assumes_lemmas_procedure` variable.

For example, suppose the TCL procedure `my_assumes_lemmas` contains the assumptions and lemma commands as follows:

```
proc my_assumes_lemmas {} {

    assume a1 = ……

    lemma    l1   = ……

}

set_user_assumes_lemmas_procedure "my_assumes_lemmas"
```

In order to use them in the proof you need to call the `solve` or `solveNB` command (for more information, see section "Prove the Equivalence or In-equivalence").

It is also possible to pass the name of the `assumes_lemmas_procedure` directly on the `solve` command without setting the variable.

5.3.2   Bit-Width and Signed-ness Mismatch in Expressions

Note that SystemVerilog syntax and semantics are used in the assume/lemma expressions. If the LHS and RHS of an equality have different bitwidths, then the lower-bitwidth side is zero extended or sign extended based on the signed-ness of the operands. For example, if in (a == b), both a and b are signed signals of different bitwidths, the smaller of the two is automatically sign-extended to the bigger one, even without specifying `$signed`. In most cases, you will not see this behavior because SystemVerilog signed-ness propagation rules say that as soon as only one of the two operands is unsigned, the entire operation is treated as unsigned. In this case the smaller of the two operands will be zero-extended. In Verilog, wires and regs are by default unsigned. Even part-selects of signed bit-vectors are unsigned. If you want to interpret them as signed, they need the explicit $signed in the assume/lemma expression. For example:

```
assume a1 = (spec.ain(1)   == \$signed(impl.ain(1)))
```

In C, the signed-ness is taken from the type, that is, an `int` is signed, and `unsigned int` is unsigned. If you want to be sure that a comparison should be signed, put `$signed` on both sides.

The automatic zero-extension/sign-extension impacts the proof as follows. In an expression `(a==b)` where a is 2-bits wide and b is 4-bits wide, and one of the two operands is unsigned, `a` will be zero extended to match the bitwidth of `b`. This means that when expression `(a==b)` is used as an assumption, then it is implicitly assumed that the most significant two bits of `b` are zero. Note that this could lead to an over-constrained proof if the user did not intend the upper 2-bits of b to be zero. One solution to this problem is to explicitly make sure that the bitwidths on both side of equality are equal, for example, by assuming `(a == b[1:0])`. Similarly, if we use the expression `(a == b)` inside a lemma, then Hector will implicitly try to prove that most significant two bits of b are zero (if this is a not a case a counterexample will be produced).

5.3.3   Lemma Hierarchy in a Proof

Hector solves the proof on an unrolled dataflow graph (DFG). Each lemma in the proof is assigned a logic level. Logic level of a node is the maximum number of nodes on a path from an input to that node. In a Hector proof, lemmas are processed in the increasing order of logic level starting with the lemma that has the lowest logic level. Once a lemma is proven it is used as assumption when proving lemmas at the higher logic levels. This makes the solving of lemmas with higher logic level easier. This is a form of assume-guarantee reasoning that happens automatically in Hector. For hard convergence problems the user can assist the tool by proving lemmas between intermediate design points. If Hector can prove these intermediate lemmas, they will be used to simplify the proof of lemmas at higher logic levels.

If a lemma is disproved, then that lemma is not used as assumption when proving lemmas at higher logic level.   Hector also supports explicit assume-guarantee grouping and ordering of lemmas as described later in this manual.

5.3.4   Example

Figure 5-1 shows a simple example shows the use of assume and lemma commands:

Figure 5-1   Waveform of the addtop Example
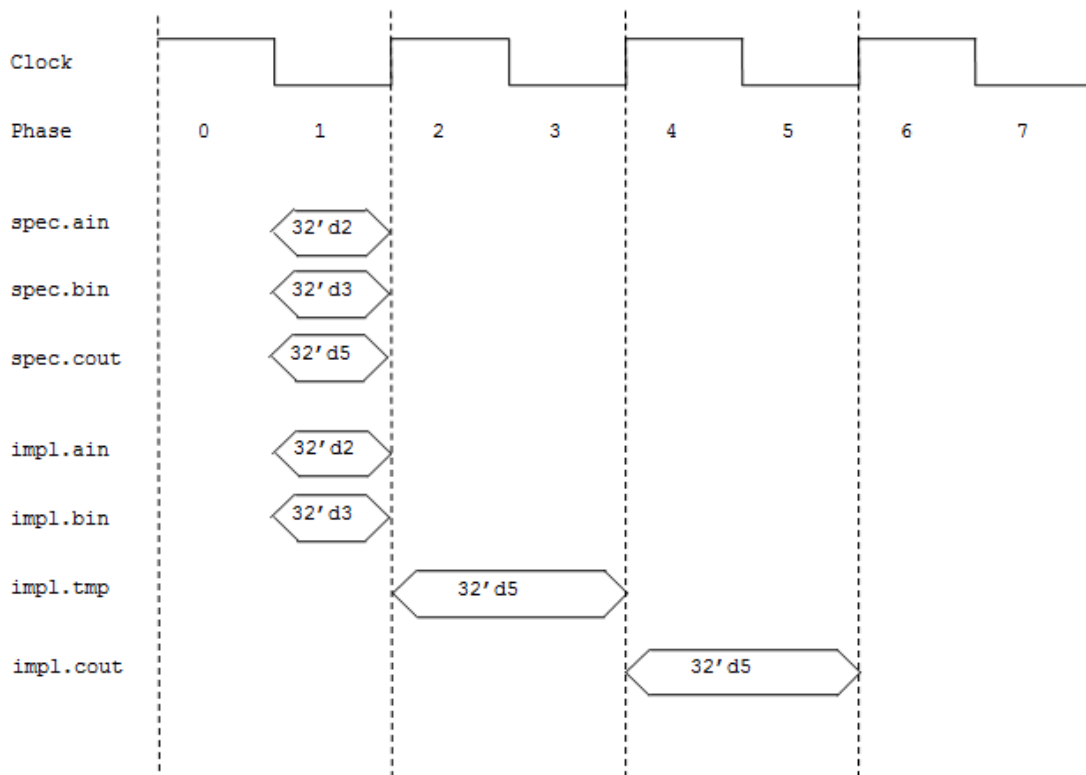
Assume that we have a design where the C-code adds two integer numbers:

```
int main()
{
        int ain, bin, cout;
        Hector::registerInput("ain",  ain);
        Hector::registerInput("bin",  bin);
        Hector::registerOutput("cout", cout);

        Hector::beginCapture();
        cout = ain + bin;
        Hector::endCapture();
}
```

Let us also assume that the corresponding RTL does the same computation but is pipelined, that is, the output is valid after two cycles:

```
module addtop (clk, ain, bin, cout);
    input          clk;
    input [31:0]   ain;
    input [31:0]   bin;
```

```
        output [31:0] cout;

        reg [31:0]  tmp;

        reg [31:0]  cout;


        always @(posedge clk) begin

            tmp  <= ain + bin;

            cout <= tmp;

        end

    endmodule
```

To prove that both outputs correspond, assume that C-variables `ain` and `bin` correspond to the RTL inputs `ain` and `bin` before the computation. The registers get their next-state on the positive clock edge between phase 1 and phase 2 (see Figure 5-1). This means that the inputs need to be applied right before that, that is, at phase 1 (for the first pipeline stage) so that the inputs can propagate through the combinational logic before the positive clock edge happens. Thus, we need to assume that the inputs to C and RTL match in phase 1. This is done using assumptions `a1` and `a2` below.

```
        assume a1 = (spec.ain(1) == impl.ain(1))

        assume a2 = (spec.bin(1) == impl.bin(1))
```

Then, we need to prove that the C-variable `cout` is equivalent to the RTL signal `cout` after two cycles. As C is untimed the result is available in the phase 1 itself. The RTL signal `cout` will get the computation result only after the second positive clock edge (that is, phase 4).

```
        lemma l1 = (spec.cout(1) == impl.cout(4))
```

Lemma l1 checks if the C variable `cout` at phase 1 is equivalent to RTL signal `cout` at phase 4. Running Hector will prove that lemma l1 holds. The Figure 5-1 shows an example waveform for the signals. The proof is performed using fully symbolic values for the signals.

5.3.5   Mapping Inputs and Outputs by Name

Hector provides commands to automatically map inputs and outputs between two designs based on names. Before we discuss these commands let us examine two possible options when mapping two signals with different bitwidths. Suppose we want to map `spec.x[8:0]` with `impl.x[5:0]`, then we have two options:

❖ `VERILOG_MAP`: Zero/sign extend smaller quantity `impl.x` so that bitwidths are the same and add `spec.x == extended(impl.x)` as the mapping. This is currently done by the `assume` command and implements Verilog semantics. By using these semantics, you are implicitly forcing higher bits of `spec.x` to either 0's for zero-extension or sign bit for sign extension.

❖ `LSB_MAP`: Only add the mapping `spec.x[5:0] == impl.x[5:0]` and leave the higher three bits of `spec.x` unconstrained.

For an assumption mapping two inputs of differing bitwidths it is *safest* to use `LSB_MAP` as it is less constraining than `VERILOG_MAP`.

For a lemma checking two outputs of different bitwidths it is safe to use `VERILOG_MAP` as it is checking for a more stricter property. You can try to check that MSBs of wider output are zero or equal to sign bits.

The `map_by_name` command described below maps inputs/outputs in two designs based on names. By default, it resolves the bitwidth mismatches by using `LSB_MAP` for mapping inputs and `VERILOG_MAP` for

mapping outputs in two designs. That is, this commands tries to make *safe* choices. The options to `map_by_name` command are:

```
map_by_name [-inputs] [-outputs]
    [-flatten "path-to-arr3 path-to-arr4"]
    [-flatten_all]
    [-split "path-to-arr5 path-to-arr6"]
    [-split_all]
    [-exclude "path-to-arr1 path-to-arr2"]
    [-exclude_all_but "path-to-arr1 path-to-arr2"]
    [-type  "verilog" | "lsbonly"]
    [-sign_extend][-ignore_case]
```

At least one of the two options `-inputs` or `-outputs` must be given. Both can be given as well.

You can provide several paths as arguments to the switches by using double quotes above the enclose lists of paths.

The `-specphase` option specifies the phase for the specification signal. The `-implphase` option specifies the phase for the implementation signal. When mapping inputs the `specphase/implphase` must be greater than equal to -1. When either phase is specified as -1 this command will generate a `-always` assumption. When mapping outputs the `specphase/implphase` must be greater than or equal to 0.

When the `-inputs` option is given, this command maps primary inputs using *assumes* in the specified phases between two designs that have matching names. `-type lsbonly` is used by default.

When the `-outputs` option is given, this command maps primary outputs using lemmas in the specified phases between two designs that have matching names. `-type verilog` is used by default.

By default the matching is case sensitive. The `-ignore_case` switch can be used to do case insensitive matching.

By default the mapping type is *safe* which means `-type lsbonly` for assumptions and `-type verilog` for lemmas. You can override this by passing type of mapping explicitly using `-type` option. In this case the user specified type is used for both input/output mapping.

The `-sign_extend` option can be specified so that smaller bit-width quantity is sign-extended to be equal in bitwidth to the larger quantity. This option is only used when:

❖ Comparing outputs by default

❖ When `-type verilog` is given in that case sign extension is employed for both inputs/outputs.

Note that `map_by_name` is not aware of the *signedness* of port declarations, and the `-sign_extend` option applies to all ports affected by the command, regardless of how they are declared.

All signal paths, both scalars and vectors, specified in the argument to the `-exclude` switch will be silently excluded from the automatic mapping. You need to map these signals manually using `assume` statements. The `-exclude_all_but` option will cause all signals except those specified by this option to be excluded from the mapping.

If an array exists in either design and is not included in a `-flatten`, `-split` or `-exclude` option, Hector will generate an error and stop. To map inputs, Hector creates `assume` statements, and to map outputs, it creates `lemma` statements. These `assumes` and `lemmas` will show up when the `listassumes` or `listlemmas` commands are given.

It is only necessary to specify one signal in a matching pair. For instance, if `in1` is an input in both the *spec* and *impl* designs, then to map them with the flattening option it is sufficient to specify `-flatten {spec.in1}`.

The input `impl.in1` does not need to be specified. This is true for the `-split` and `-exclude` options as well.

Arrays specified by the `-flatten` option will be turned into a scalar by concatenating the elements, with the first element on the rightmost side and the last element on the leftmost side. This would then match either a similarly flattened signal in the other design or a scalar of the same name in the other design. For instance, suppose you have the following declarations

```
input [7:0] mem [2:0];              // in the specification

input [6:0] mem [0:2];              // in the implementation
```

This results in the following assumption being generated when `-flatten` is used:

```
assume {spec.mem[2][4:0], spec.mem[1], spec.mem[0]} == \      {impl.mem[0],
impl.mem[1], impl.mem[2]}
```

Note that only a portion of the leftmost element of the specification signal is mapped. If you use the `-type verilog` option instead, the mapping will be as follows:

```
assume {spec.mem[2], spec.mem[1], spec.mem[0]} == \  {impl.mem[0], impl.mem[1],
impl.mem[2]}
```

Consider an array `sig1[2]` and a scalar `sig1`. If both designs contain the array, and if the arrays are included in the flatten list, then `sig1` in the two designs will be flattened and mapped. If one if the designs has a scalar `sig1` and the other design has an array `sig1` (included in the flatten list), then the flattened `sig1` in one design will be mapped to the scalar `sig1` in the other design. The `-flatten_all` switch may be used instead of `-flatten` – in that case, all arrays will be flattened.

Arrays included in the *split* list will be turned into N scalars, which will then match a similarly split array of the same name in the other design. The `-split_all` switch may be used instead of `-split`, in which case all arrays will be split. The example mentioned previously would generate the following three assumptions:

```
assume ( spec.mem[2][6:0] == impl.mem[0][6:0] )

assume ( spec.mem[1][6:0] == impl.mem[1][6:0] )

assume ( spec.mem[0][6:0] == impl.mem[2][6:0] )
```

If the option `-type verilog` is also specified, these three assumptions will change as follows:

```
assume ( spec.mem[2] == impl.mem[0] )

assume ( spec.mem[1] == impl.mem[1] )

assume ( spec.mem[0] == impl.mem[2] )
```

If two arrays need to be mapped with the `-split` option, they must contain the same number of elements. Note that the dimensions do not have to match, only the number of elements needs to match. Therefore, the following arrays can be mapped:

```
Input [7:0] mem [3:0][1:0];    // in the specification

Input [6:0] mem [7:0];         // in the implementation
```

However, the following two arrays cannot be mapped:

```
Input [7:0] mem [3:0][1:0];    // in the specification

Input [6:0] mem [3:0][2:0];    //in the implementation
```

Note that `-split_all` and `-flatten_all` must not be used simultaneously. If the `-exclude` option is used in conjunction with either of them, then they are applicable to all signals except those in the excluded list.

The `-exclude` and `-exclude_all_but` options cannot both be specified.

Examples:

❖ `map_by_name -inputs -outputs -specphase 1 –implphase 1`

Maps both inputs and outputs in two designs in phase 1. This would be appropriate if both spec and *impl* are combinational designs.

❖ `map_by_name -inputs -outputs -specphase 1 –implphase 1 -ignore_case`

Maps both inputs and outputs using case insensitive matching in phase 1. This may be useful when one of the designs is VHDL.

❖ `map_by_name -inputs -specphase -1 –implphase -1`

 Map inputs in all phases between the two designs.

❖ `map_by_name -outputs -sign_extend -specphase 1 –implphase 2`

 Map outputs in phase 2 and use sign extension

❖ `spec.array1` is an array with four elements, each 16-bits

`impl.array1` is an array with four elements, each 10-bits

If both are flattened using the command `map_by_name –flatten "spec.array1 impl.array1"`:

`spec.array1` becomes a single 64-bit signal

`impl.array1` becomes a single 40-bit signal

These signals can then be matched using the rules specified in the `map_by_name` command (signed or unsigned, `verilog` or `lsbonly` as specified by the `-type` switch)

If both signals are split using the command `map_by_name –split "spec.array1 impl.array1"`:

`spec.array1` becomes four signals, each 16-bits long

`impl.array1` becomes four signals, each 10-bits long

These signals can then matched one-by-one using the rules specified in the `map_by_name` command (signed or unsigned, `verilog` or `lsbonly` as specified)

❖ `spec.array2` is an array with four elements, each 16-bits

`impl.array2` is an array with eight elements, each 10-bits

If both the arrays are flattened:

`spec.array2` becomes a single 64-bit signal

`impl.array2` becomes a single 80-bit signal

These signals can then matched using the rules specified in the `map_by_name` command (signed or unsigned, `verilog` or `lsbonly` as applicable)

If both these signals are split:

`spec.array2` becomes four signals, each 16-bits long

`impl.array2` becomes eight signals, each 10-bits long

This will generate an error. If one is flattened and one is split, there is no match, so the `map_by_name` command does not map these to each other, thus generating an error.

Hint

The `map_by_name` commands must be placed in the user assumes and lemmas TCL procedure. The `assumptions/lemmas` added by this command can be viewed using `listassumes/listlemmas/listproof` commands.

If the two designs do not have ports with matching names or some ports do not have matching names, then the use following command can be used to *safely* map a specified pair of signals from two designs.

```
map_signal_pair -sig1 <name> -sig2 <name>
    -sig1phase <num> -sig2phase <num>
    [-assume]  [-type] [-sign_extend]
```

By default this commands adds a lemma to check two signals are equal. To make it an assumption use the `-assume` option. By default the `type` is `safe` which means `LSB_MAP` for assumptions and `VERILOG_MAP` for lemmas. You can override this by passing type of mapping explicitly. Possible options are: `safe, verilog, lsbonly`.

5.3.6   Using Assumptions and Lemmas on Memories

We can map an array (or memory) by explicitly mapping each memory location in the two arrays. For example:

```
assume spec.mem[0](0) == impl.mem[0](0)

assume spec.mem[1](0) == impl.mem[1](0)

…
```

Or, using assumptions generated in a TCL for loop:

```
for {set i 0} {$i < 100} {incr i} {
    assume spec.mem[$i](0) == impl.mem[$i](0)
}
```

These give you a fair amount of flexibility, but come at a cost as the memory increases in size. If the memory is large, and particularly if each transaction only addresses a small number of unique locations in the memory, it can be much more efficient to map the memory using `forall` construct.

The `forall` keyword introduces a temporary variable which is iterated over all memory locations to map the elements in two memories. The following is the simplest `forall` example:

```
assume forall(i) (spec.mem\[i\](0) == impl.mem\[i\](0))
```

It expresses the relationship that each memory location in spec.mem is equal to the corresponding memory location in impl.mem as shown in Figure 5-2. The temporary variable introduced by `forall` construct is `i`.

Figure 5-2   Simple Memory Mapping

spec.mem   impl.mem
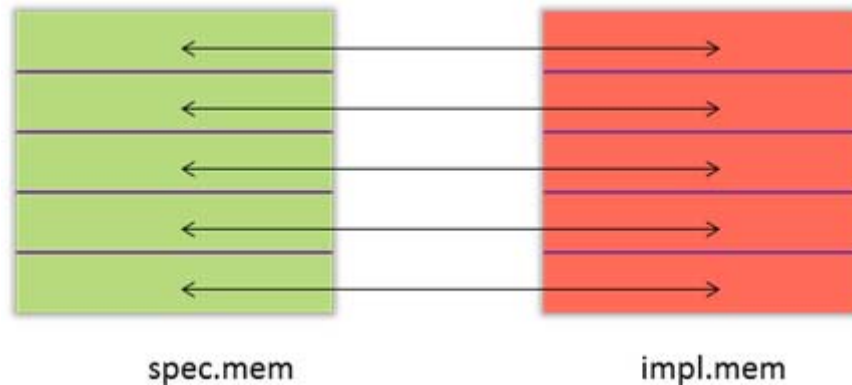
This single assumption has the same meaning as the multiple assumption examples we discussed above. The square brackets have to be escaped or else the TCL interpreter will try to run a proc named `i`. We can also escape the entire expression with curly braces like this:

```
assume { forall(i) (spec.mem[i](0) == impl.mem[i](0)) }
```

The temporary variable introduced by `forall` can be used in expressions. Here is a way to map an array of length 5 with a reverse mapping between the spec and the `impl` (see Figure 5-3):

```
assume { forall(i) (spec.mem[i](0) == impl.mem[4 - i](0)) }
```

Figure 5-3   Reverse Memory Mapping

spec.mem   impl.mem
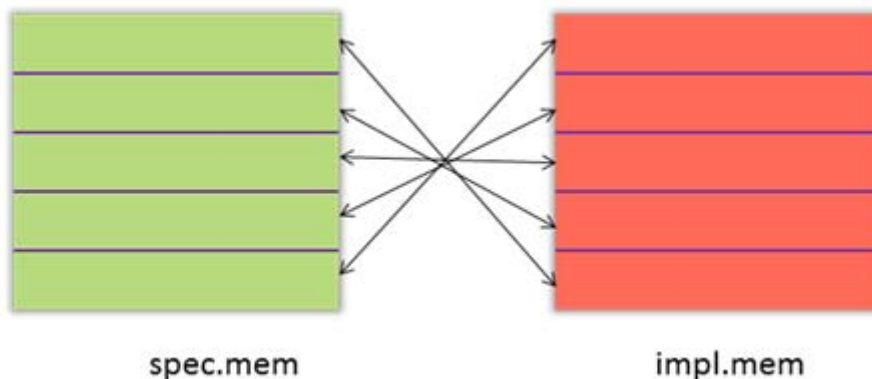
The `forall` keyword is part of the assume/lemma expression language, it is not part of the TCL language, so it is important to not confuse it with the TCL keyword `for` as it is used in the examples above. Likewise, `i` is not a TCL variable, so we do not use `$i` to get `i`'s value. Notice also that the temporary variable does not have a phase. The temporary variable cannot be used inside any phase specification. The `forall` construct is intended to expand over memory array indices (not time).

The temporary variable used with `forall` must be used as a read address for some array/memory. This enables Hector to compute a bound on the temporary variable. Following error message is issued if Hector is unable to compute the bound on the temporary variable.

```
*** Error IND-345: In assumption <NAME1>: No bound for quantifier variable
'<NAME2>' was computed.  Please make sure that quantifier variable is used as read
address for some array/memory.
```

The temporary variable can be used outside of an index expression as part of an implication. This example maps only locations 15 through 40:

```
assume { forall(i) ((i >= 15) && (i <= 40))

                ->(spec.mem[i](0) == impl.mem[i](0)) }
```

And this example shows how to map a one dimensional array to a 2D array declared `[N/2][2]`:

```
assume { forall(i) (spec.mem[i](0) == impl.mem[i/2][i%2](0))
```

Two dimensional arrays should be mapped using two `foralls`:

```
assume { forall(i) forall(j)

    (spec.mem[i][j](0) == impl.mem[i][j](0)) }
```

When the memories occurring in `forall` assumption/lemma are addressed by performing arithmetic on the `forall` temporary variable (for example, 15-i, i%2) the resulting proof may not be complete. This has to do with the additional complexity this introduces to the address logic in the proof.

In this event, Hector prints out a warning message *IND-021: Quantifier that may cause incompleteness detected*. It means that it is possible for the prover to generate false negatives (the proof is false, but it is not) due to incomplete assumption instantiation. The Hector team is working to reduce the frequency of this warning.

5.3.6.1   Using Templates

The *template* directive is used to specify how the bits of a single element in the system-level (SL) memory are to be mapped to the bits of a single element in the RTL memory.

Figure 5-4 is an example of two memories that have the same number of elements. However, the number of bits used to represent each element is different. An array of `struct elem` is declared in the SL model where the fields of the `struct` are defined using data types native to C/C++. Both fields (`a` and `b`) of `struct` elem are 8 bits. In the RTL model, each `struct` is represented by three bits. A template directive is used to define a correspondence between these two memory layouts. The following directives are used to capture the mapping of Figure 5-4.

```
template t1 {

    a= [2:1];

    b= [0:0];

}

forall (i) (spec.MA[i] == template t1 (impl.MB[i]))
```

Figure 5-4   Memory Layout Differences
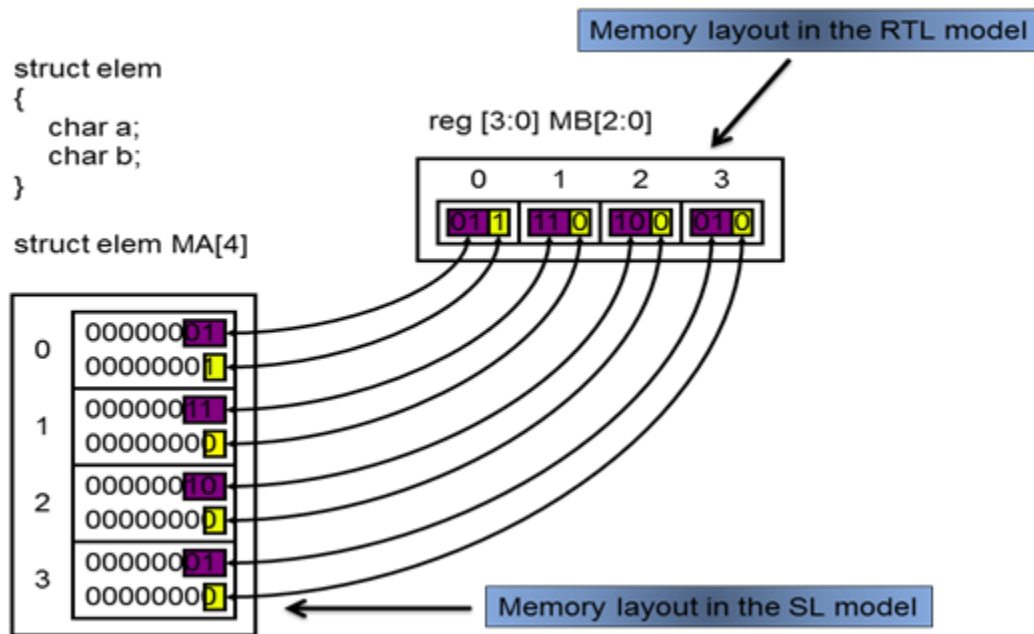
The left hand side of each assignment in the template is a field in the C/C++ struct. The right hand side of each assignment is the bit range used to represent that field in the RTL memory. Currently, there is no support for C/C++ structs that contain other structs or that contain arrays. If the bit width of the right hand side of an assignment is less than the bit width of the left hand side, an unsigned cast is used to make the bit widths equal. If the bit width of the left hand side is less than the bit width of the right hand side, this is an error and it will be reported.

Once a template has been created it is used inside a `forall` assumption or a lemma so that Hector generates the right mappings between the individual elements of two memories. The syntax of this assumption/lemma requires that one side be a read from SL memory and the other side be an application of template to a read from RTL memory.

In Figure 5-5 is an example of a situation that is similar to the one in Figure 5-4 except that the memory elements in MB appear in reverse order compared to the SL model. To express this memory mapping we will use:

```
template t1 {

    A= [2:1];

    B= [0:0];

}
forall (i) (spec.MA[i] == template t1 (impl.MB[3-i]))
```

Figure 5-5   Memory Layout Differences and Reverse Memory Mapping
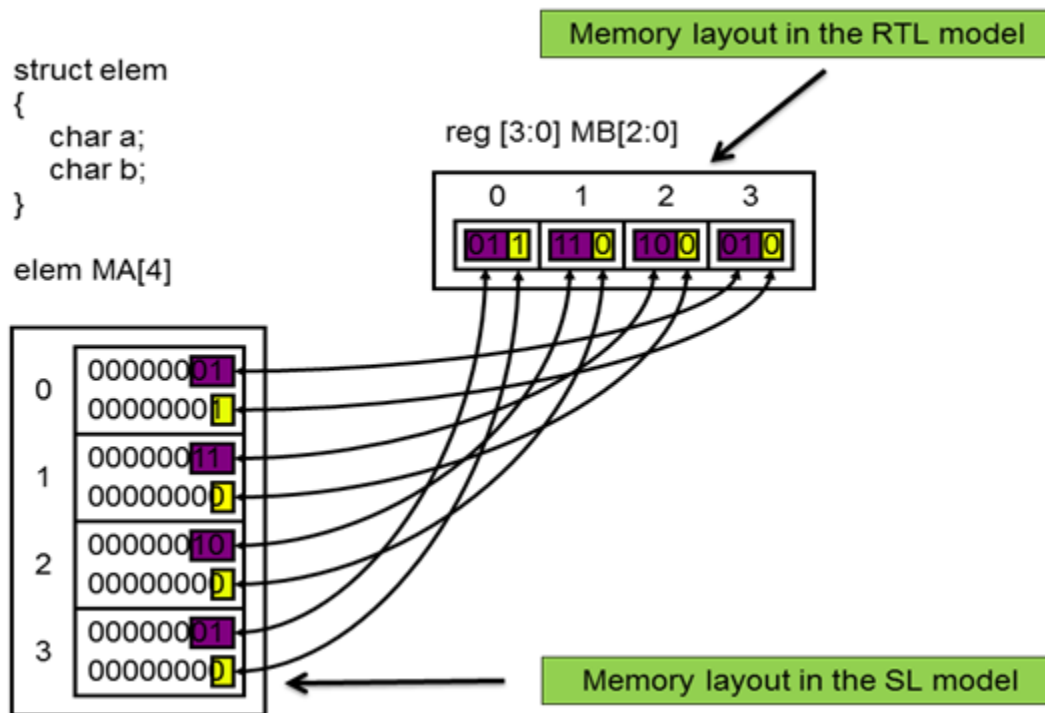
5.3.7   Creating Lemmas for Variable Length Transactions

If the length of a transaction is data dependent, we can write lemmas using a valid signal generated when the transaction result appears at the RTL design output. Here are two ways to setup this problem in Hector:

❖ Write lemmas for different phases where the output could be valid. The conditional operator activates the check only when the output actually appears.

For example, say outputs can be available between (`start_phase`, `end_phase`). Add the following lemma generation code to the `user_assumes_lemmas_procedure`.

```
set  atleast_one "impl.valid($start_phase)"

for { set p $start_phase } { $p < $end_phase } { set p [expr $p+2] } {

        lemma l${p} = impl.valid($p) -> (spec.out(1) == impl.out($p))

        set atleast_one "${atleast_one} || impl.valid($p)"

}

lemma atleast1 = $atleast_one
```

Pros: No need to change in code.

Cons: May create lots of lemmas

❖ Add code (registers) to the RTL so that it holds the value of the outputs until the longest latency. This will require only the following lemma:

```
lemma l0 = (spec.out(1) == impl.out($end_phase));
```

Pros: Creates only one lemma.

Cons: Requires changes to RTL.

5.3.8   Defining Control Signal Waveforms

Many sequential designs require reset, enable, or other control waveform for correct transaction operation. You can use `assume` commands to create these waveforms. Write a TCL loop over the phases specifying required signals values as needed.

For example, if you want reset to be high for two clock cycles (4 phases) and then low for the rest of transaction (which takes 10 clock cycles) you can write following TCL code in the `user_assumes_lemmas` TCL procedure.

```
for { set i 1 } { $i <= 24 } { incr i } {

   if {$i <= 3} {

      assume  reset($i) == 1'b1

   } else {

      assume  reset($i) == 1'b0

   }

}
```

5.4   Controlling Model Initialization

Hector does not automatically initialize any state-holding elements (registers or memories) in designs. The following sections describe how you can initialize a design using Hector.

It is important to understand that even if you initialize the state of your designs as described below, Hector will only consider the initial state if the proof is started with the `solve_init` or `solveNB_init` commands. The more common `solve` and `solveNB` commands ignore any initial states you specify and look for a general proof that is independent of those states.

5.4.1   Establishing an Initial State for RTL Models

The RTL compiler establishes the initial state for RTL models that have a clock and reset port defined. The reset port is asserted, the clock line is toggled once (in both directions), we wait until all signal transitions have been propagated and then the value of all (state holding) signals are recorded.

It is also possible to specify an arbitrary reset signal waveform using assume commands. This technique creates a proof only for a transaction that begins with reset. It may be necessary to separately prove transactions that do not begin with reset. Registers that are not given a value during the reset sequence can have their initial value specified via the assume command (see section "Assumptions and Lemmas").

🛑 Caution

> VHDL alert! If the RTL design is described in VHDL, all references to implementation names (`impl.foo`) in this section must use all upper case letters – regardless of how they are specified in the design. VHDL is a case-insensitive language and the VHDL parser converts all names to upper case.

5.4.2   C++ State Variables - Overview

When the `beginCapture()` directive is encountered in the C/C++ code, all of the variables that are created at that point are partitioned into one of three groups: inputs, outputs and state variables.

The variables that are treated as inputs and outputs are identified by means of the `Hector::registerInput()` and `Hector::registerOutput()` directives. All other variables are treated as state variables.

In the following example, when the `Hector::beginCapture()` is executed, the following variables are created:

```
A B C D E F J K L M N P Q R S T V     in  out
```

in and *out* are declared as an input and output respectively. And the remaining variables are treated as state variables.

```
A B C D E F J K L M N P Q R S T V
```

☞ Note

According to the C++ language standard, the variables J K L M are not created until 'func1' is called for the first time. However, Hector creates all statically declared variables when `Hector::beginCapture()` is executed. This interpretation is better suited for hardware modelling.

```cpp
#include "Hector.h"

      const int A = 10;
            int B = 11;
            int C;
static const int D = 12;
static      int E = 13;
static      int F;


void func1(int in, int& out1) {
      const int G = 14;// not a state var
            int H = 15;// not a state var.
            int I;// not a state var.
static const int J = 16;
static      int K = 17;
static      int L;
static      int M = 18;

        Hector::show("G", G);
        Hector::show("H", H);
        Hector::show("I", I);
        Hector::show("J", J);
        Hector::show("K", K);
        Hector::show("L", L);
        Hector::show("M", M);
   out1 = in + G + H + I + J + K + L ;
   M = M + 1;
}


int main() {
      const int N = 19;
            int P = 20;
            int Q;
static const int R = 21;
static      int S = 22;
static      int T;
            int V;
            int in, out;
```

```
        Hector::registerInput("in", in);
        Hector::registerOutput("out", out);
        Hector::beginCapture();
        Hector::showall("pre_");

    func1(in,V);
    out = V  + A + B + C + D + E + F + N + P + Q + R + S  + T +  p1.val2 + p2.val1;
    Hector::endCapture();
```

5.4.2.1   C++ State Variables - Establishing an Initial State Value

The initial values of the state holding variables in the C/C++ model can be specified in a number of ways:

❖ Some computation in the C/C++ code prior to the *beginCapture()* statement.

```
    C = 22;
    F = init_func();
```

❖ Adding an initializer to the variable declaration in the C/C++ code.

```
    const int A = 10;
          int B = 11;
```

❖ By using an `assume` command in the TCL script

```
    assume (spec.C(1) == 'd5)
```

According to the C++ language standard, any statically declared variables that are not initialized has their initial value set to zero. Likewise any global variables that are not initialized, has their initial value set to zero. This is the behavior that is observed if the C/C++ model is compiled with GCC and run. However, it is often desirable to ignore the initial state during a proof, and treat the state variables as free variables - as this leads to a less constrained proof.

When a state variable is treated as a free variable:

1. The proof procedure assumes that any value is possible for the initial state

2. Counter examples may contain concrete value assignments for these variables.

5.4.2.2   C++ State Variables - using the initial state

The first and second method of specifying an initial state above do not necessarily mean that the initial value is used in the proof. That is determined by:

1. The use of the `const` qualifier when a variable is declared.

2. The value of the `_hector_cfg_zero_init (true / false)` TCL variable.

3. The value of the `_hector_sym_init_read_only_state_vars  (true / false)` TCL variable.

4. Whether `solve` (does not use initial state) or `solve_init` (uses the initial state) is invoked to run the proof.

Note that the third style of specifying the initial state that explicitly uses an `assume` statement in the TCL, both defines the initial state and include its use in the proof.

There are two modes for using the initial state:

1. Backward compatibility mode (`hector_cfg_zero_init  == false` and `_hector_sym_init_read_only_state_vars == true`)

    This mode insures that the way state variables are initialized and used is the original method. This method does have some inconsistencies in the way state variables are initialized and used. This is why the consistency mode was created.

2. Consistency mode (`_hector_cfg_zero_init  == true`)

    This mode ensures that the initialization and use of state variables is consistent for all types of statically and globally declared variables. It also ensures consistency in the way the original and new front end compilers treat initialization and use.

5.4.2.2.1   Consistency Mode

The remaining of this section provides more details about the consistency mode. That is, the rest of this section assumes (`_hector_cfg_zero_init  == true`).

Suppose a state variable is declared using the `const` qualifier.

```
const int A = 10;
```

Then the initial state of A is always used in the proof regardless of whether `solve` or `solve_init` is invoked, and regardless of the value of the `_hector_sym_init_read_only_state_vars` TCL variable. This is consistent with the semantics of the `const` qualifier. Therefore, in the above example the initial value of the following variables is always used in the proofs performed by both `solve` and `solve_init`.

> A D J N R

The `_hector_sym_init_read_only_state_vars` TCL variable controls whether or not the initial value of a read-only state variable is used in a proof. A read-only state variable is one that:

1. Is not declared using the const qualifier
2. Is given an initial value
3. Is not altered (assigned to) during the transaction.

When this variable is false, the initial values of read-only state variables is not used when `solve` is called, and is used when `solve_init` is called. When this variable is true, the initial values of read-only state variables is used when either `solve` or `solve_init` is called. The default value of this variable is false. This TCL variable has no effect on state variables that have no initial state specified or on state variables that are altered during the transaction. In the above example, the following variables are read-only state variables.

> B E K P S

The state variable M is not a read-only state variable because it is assigned a new value during the transaction.

Table 5-1 summarizes this discussion using the above example. The columns indicate whether `solve` or `solve_init` is invoked and the value of the `_hector_sym_init_read_only_state_vars` TCL variable. An x indicates the state variable is treated as a free variable.

In summary, there are three use models available for using initial state information in a proof.

1. Set `_hector_sym_init_read_only_state_vars` to false, and invoke solve. Only the initial value of state variables qualified with the `const` qualifier is used in the proof.
2. Set `_hector_sym_init_read_only_state_vars` to true, and invoke solve. The initial value of state variables qualified with the `const` qualifier is used in the proof. Additionally, the initial value of any read-only state variable is used in the proof.

3. Invoke `solve_init`. The initial value of all state variables that have an initial value is used in the proof.

Table 5-1   Consistency Mode Example

State variable	Solve / false	Solve / true	Solve_init / false	Solve_init / true
A	10	10	10	10
B	X	11	11	11
C	X	0	X	X
D	12	12	12	12
E	X	13	13	13
F	X	0	0	0
J	16	16	16	16
K	X	17	17	17
L	X	0	0	0
M	X	X	18	18
N	19	19	19	19
P	X	20	20	20
Q	X	X	X	X
R	21	21	21	21
S	X	22	22	22
T	X	0	0	0
V	X	X	X	X

5.5   Composing the Equivalence Problem

To create the Hector internal test framework using the values specified by all the commands in the interface procedure use the following command:

```
compose
```

Note
Whenever the specification or the implementation design is compiled, the compose command must be run again afterwards.

5.5.1   Composing a Single Design

When running Hector on a single design, you can use the following special form of compose. If you only have a specification design, run `compose` as follows:

```
compose -noimpl
```

If the you only have a implementation design, run compose as follows:

```
compose -nospec
```

When referring to the variables during verification of a single design (in assume/lemma), you should not use a prefix *spec.*, *impl.*, or `ext..` An error will be reported if these prefixes are used. For this reason it is not recommended to use these compose forms if you also plan to check equivalence between two designs. Your lemmas and assumes will be more portable if you always compose two designs and use the *spec* and `impl` prefixes on signal names.

6 Solving the Equivalence Problem

This chapter describes the various commands and options associated with solving the equivalence problem that is setup using commands from the previous chapters, in the following sections:

- ❖ "Prove the Equivalence or In-equivalence"
- ❖ "Automatic Lemmas"
- ❖ "Reports and Logs"
- ❖ "Solve Scripts in Hector"
- ❖ "Hector in Multi-Processor Environment"

6.1 Prove the Equivalence or In-equivalence

A proof in Hector consists of a set of assumptions and a set of lemmas (proof obligations) and the goal is to prove the lemmas by making use of the assumptions.

6.1.1 Proofs, Sub-proofs, and Tasks

A proof P with assumptions A and lemmas L can be solved by breaking it into sub-problems. In Hector a sub-problem can be one of the two types

6.1.1.1 Task

A task has the same set of assumptions (A) as P but it can operate on a subset of lemmas in the original proof. A task has a specific solve script attached to it. A task can be thought of as a leaf-level proof that cannot be further broken down into sub-problems. Task is also the unit of multi-processing in Hector. That is, different tasks can be run on different machines/processors if enough compute resources are available (see section "Hector in Multi-Processor Environment").

6.1.1.2 Sub-Proof

A sub-proof is a proof that can have additional/fewer assumptions and/or lemmas in it.

A given proof can be solved by creating multiple tasks and/or multiple sub-proofs. One example would be the sub-proofs created by case-splitting on input values. Each sub-proof contains additional constraints to implement the case-split.

Figure 6-1 shows a proof P that is being solved using three tasks T1, T2, and T3. Tasks T1, T2, T3 can be run in parallel in a multi-processor environment.

Figure 6-1   Proof with Three Tasks
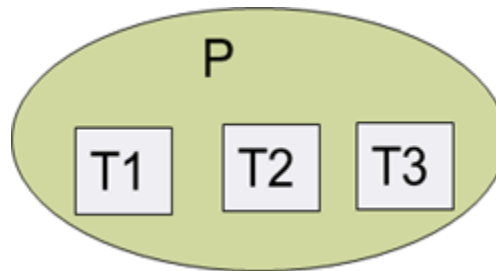
Figure 6-2 shows a proof P that is divided into two sub-proofs P1 and P2. P1 is being solved using tasks T1, T2, and T3, while P2 is being solved using task T4.

Figure 6-2   Proof with Sub-proofs and Several Tasks
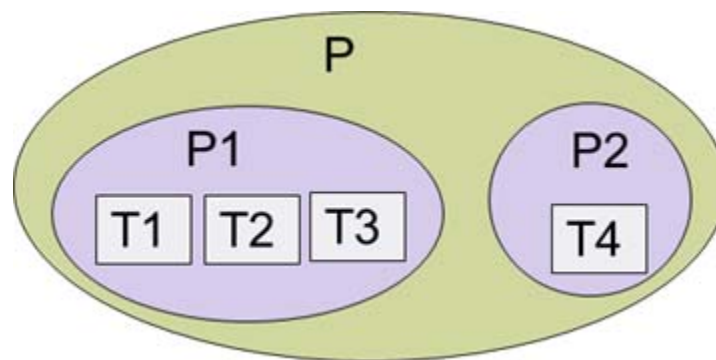
6.1.2   Blocking Commands

The simplest way to prove the equivalence or in-equivalence of each corresponding output and other related lemmas is to create a single proof with only one task in it as shown in Figure 9.

Figure 6-3   Proof with One Task
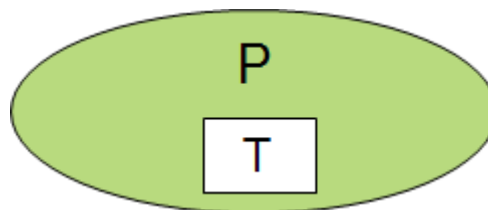
This can be done automatically in Hector by using the following command:

```
solve [-ual <UserAssumesLemmasTCLProc>] \
     [-script <solveScriptName>]
```

This command assumes that all the state holding variables or registers act as pseudo primary inputs. To assume the initial state of the state holding variables or registers in the proof use the following command:

```
solve_init [-ual <UserAssumesLemmasTCLProc>] \
          [-script <solveScriptName>] \
```

Note that the proof obtained using the `solve_init` command is bounded, that is, it only proves the first transaction starting from the initial state. Intuitively, the `solve_init` and the `solve` commands prove the

base case and the inductive case respectively, of an induction proof. Invoking `solve_init` and `solve` commands result in an exhaustive proof starting from the initial state.

These commands are blocking commands, that is, these commands return to the Hector shell only when they finish.

The `-ual` option can be used to provide the name of the TCL procedure that has assumptions/lemmas. If this is not provided Hector uses global TCL variable `user_assumes_lemmas_procedure`. The `-script` option can be used to provide the name of the solve script. If this is not provided Hector uses global TCL variable `custom_solve_script`.

💡 Hint
It is recommended that you use the options such as `-ual` or `-script` instead of using the corresponding global variables. If you use both, the switches passed to the `solve` commands will take precedence.

6.1.3   Non-blocking Commands

Another way to prove the equivalence or in-equivalence of each corresponding output and other related lemmas is to use a non-blocking command `solveNB` or `solveNB_init`. This command allows you to run the proof in the background and returns to the Hector shell immediately. In general, this command is very flexible and can be used to create several different sub-proofs and tasks.

A proof with name `<proofname>` is created when you use the following command:

```
solveNB <proofname> [-ual <UserAssumesLemmasTCLProc>] \
                    [-script <solveScriptName>]
```

The meaning of the command options is same as described for `solve` command. You can also use the `solveNB_init` command as follows:

```
solveNB_init <proofname> [-ual <UserAssumesLemmasTCLProc>] \
                         [-script <solveScriptName>]
```

By default, the `solveNB P` and `solveNB_init P` commands create a proof P and a single task T in P. P is solved using T. The task T has the same assumptions and lemmas as P.

In a regression setup, you cannot manually enter commands, that is, in a regression script you have to wait for the started proof to finish before sampling the result. This is done using the `proofwait` command.

6.1.3.1   The proofwait Command

Syntax

```
proofwait [-nokill] [-anyChange]  [prooflist]
```

The `proofwait` command suspends execution until some change occurs in the proofs being monitored. Once the `proofwait` command has been issued, no further input is accepted from the command line until `proofwait` completes. If it is necessary to get back the command line prompt so that other commands can be typed on the command line, enter **cntl-C**. This will return the command line prompt. If 'prooflist' is not specified, then all proofs will be monitored. The `Proofwait` command is able to detect changes in either the status of the lemmas or the status of the proofs and returns to the command line prompt when such a change occurs.

Waiting for Proofs

```
proofwait [-nokill] [prooflist]
```

The `proofwait` command suspends execution until each proof in `'prooflist'` has a conclusive status. A proof has a conclusive status if either there are no running / pending tasks in it or all lemmas in it have a status of either success, fail, or conflict. Note that a proof can be conclusive and still have running / pending tasks in it. All Active tasks in a proof will be t once the proof has a conclusive status. A typical example is when multiple solve scripts are run in parallel. As soon as one script is conclusive, tasks running the other scripts are terminated. If the `-nokill` option is specified, the active tasks are not terminated.

Return codes

❖ 0: Error return. Indicates a proof name specified that did not exist.

❖ 1: Either **cntl-C** was detected or all proofs have a conclusive status.

Waiting for lemmas

```
proofwait -anyChange [-nokill] [prooflist]
```

When the `'-anyChange'` switch is specified, a response to changes in the proof process is enabled. The `proofwait` command blocks until any lemma in any of the specified proofs transitions to a terminal status. Terminal status is currently defined as one of the following: {success, failed, nottried, inconcl, conflc, error, killed}. Note that cond-success and cond-failure are not terminal statuses.

This allows the user's TCL script to continue executing and analyze what action is needed based on the lemma status change. A TCL loop can be created to support iterative execution of user analysis code after each lemma (or set of lemmas) reaches terminal status. For example, counter-examples can be generated and the debug process started while the rest of the lemmas are still being solved.

The use of -anyChange might improve run time in some cases. For example, if a proof contains two tasks t1 and t2. If t1 and t2 contain the same lemmas, lemma1 and lemma2 and that lemma1 (but not lemma2) is proved in t1 and that lemma2 (but not lemma1) is proved in t2. Without the `-anyChange` switch, either t1 or t2 must complete before the proof is completed. However, the `-anyChange` switch would allow to conclude the proof is completed and both t1 and t2 would be terminated.

The return status of `'proofwait'` will be extended to indicate the reason that the command has exited:

❖ 0: An error occurred (and was reported to the screen)

❖ 1: `proofwait` returned, because all work on the selected proofs is complete

❖ 2: `proofwait` returned, because some lemma status has been updated

❖ 3: `proofwait` was interrupted by ^C

6.1.3.2   The getLemmaUpdates Command

Syntax:

```
getLemmaUpdates [list of proof names]
```

The `getLemmaUpdates` command returns a list of lists identifying the task-lemma pairs in each proof that have changed status since the last call to the `getLemmaUpdates` command. The returned values from this command are only valid when the `-anyChange` switch is used with the `proofwait` command. The `getLemmaUpdates` command may return a zero-length list if no task-lemma pairs have changed since it was last called. Each element of the returned list is a list consisting of the following elements in the listed order:

1. proof name

2. lemma name

3. composite lemma status

4. taskId

5. task lemma status

Where,

- ❖ *proof name* and *lemma name* are the names specified by the user in the Hector setup.

- ❖ The *taskId* is an integer that ranges from 0 to the number of tasks scheduled. A *taskId* of 0 is used as a taskId placeholder for some special case handling - it does not actually correspond to task assigned to a worker (this is explained in more detail below). When *taskId* is non-zero, each taskId-lemma pair appears exactly once in the lists returned by the `getLemmaUpdates` command. When *taskId* is zero, the same taskId-lemma pair may appear multiple times in the lists returned by the `getLemmaUpdates` command.

- ❖ The *task lemma status* is the result found by the worker assigned taskId. If the *taskId* is 0, the task lemma status is set to "-".

- ❖ The *composite lemma status* is a summary of the all the *task lemma status* results that have already been reported for this lemma (including the current one).

If other information about the proof or lemma is required (for example, the name of the solve script, the name of the machine upon which it was executed and so on), the task ID can be used to query the `taskStatus` command.

Example 1: Multiple Solve Scripts

Suppose lemma L1 appears in tasks 1, 2 and 3 (as would be the case when multiple solve scripts are specified), and further suppose that the first three calls to getLemmaUpdates return the following three lists:

```
{myproof  L1   inconcl  1 inconcl}
{myproof  L1   success  2 success}
{myproof  L1   success  3 inconcl}
```

Note that L1 in each of the three tasks refers to the same lemma, since the set of assumptions is the same for all the three cases. You can see that L1 in task 1 resulted in *inconcl* and so the composite lemma status is also *inconcl* since there were no prior results for L1. L1 in task 2 resulted in *success* (that is, task lemma status == success), so you can conclude that the composite lemma status is *success*. Finally, L1 in task 3 resulted in *inconcl*. However, since task 2 had already concluded *success* for L1, the composite lemma status is still 'success'.

Reporting both forms of the lemma status allows an analysis of how each individual task performed on a lemma as well as how all the tasks that have completed so far have performed on a lemma.

Example 2: Case Splitting and Assume / Guarantee

Suppose there is one lemma L1 and further suppose there is a case split on a single bit B, then there will be 3 proofs:

1. Root

2. case_B_0 for B == 0

3. case_B_1 for B == 1

Note that although each of the three proofs has a lemma named L1, they are unique lemmas since the set of assumptions in each of the proofs is unique. Proofs 2 and 3 each have a task associated with the proof. Proof 1 does not have an associated task. The result for L1 in the root proof is determined by the result(s) of one (or both) of the other two proofs (see the section "Case Splitting" for more on this). In fact, L1 in the root

proof is evaluated twice - once when proof 2 reaches a terminal status and once when proof 3 reaches a terminal status.

Suppose proof 2 finishes first. The following two lists is returned by the first call to the `getLemmaUpdates` command.

```
{case_B_0    L1  success 1 success}
{root_proof   L1  inconcl  0  - }
```

When proof 1 finishes, the following two lists would be returned.

```
{case_B_1    L1  success 2 success}
{root_proof   L1  success  0  - }
```

Note that the task-lemma pair lemma == L1 and taskId == 0 appears in the returned lists twice - once for each sub-proof of the case split. The first time that pair appears, the composite lemma result is inconcl since the case_B_1 result is not yet known.

Note also that L1 in proof 1 may actually be labeled with multiple terminal states over time. That is, when L1 in proof 2 becomes *success*, then L1 in proof 1 is evaluated and its status transitions from *nottried* to *inconcl*. When L1 in proof 3 becomes *success*, then L1 in proof 1 is evaluated and its status transitions from *inconcl* to *success*. Therefore, *inconcl* may or may not be a terminal status for lemmas associated with task 0.

6.1.3.3   The getTaskUpdates Command

Syntax:

```
getTaskUpdates [list of proof names]
```

The 'getTaskUpdates' command returns a list of task IDs identifying all the tasks that have either had a lemma transition to a terminal status or been terminated since the last call to 'getTaskUpdates'.

This command works only when the `-anyChange` switch is used with the `proofwait` command.

6.1.3.4   Example use of -anyChange

6.1.3.4.1   Generating Counter-Examples

To support early access to counter-example waveforms for failed lemmas without interrupting the proof running in the background. This can be achieved by the following script fragment:

```
…
solveNB p
while { [proofwait -anyChange] == 2 } {
  foreach change [getLemmaUpdates] {
    if {[lindex $change 2] == "failed" } {
      set proofName [lindex $change 0]
      set lemmaName [lindex $change 1]
      set taskID    [lindex $change 3]
      set fsdbName "${proofName}_{$taskID}_${lemmaName}"
       cdproof $proofName
      puts "Saving CEX for lemma $lemmaName"
       simcex -fsdb $fsdbName $lemmaName
```

```
      }
    }
        …
```

6.1.3.4.2   Terminating the Remaining Tasks After the First lemma Failure

Another use is to terminate any tasks running in a particular proof as soon as any lemma in the proof fails.

```
      …
solveNB p
solveNB p1
solveNB p2
while { [proofwait -anyChange] == 2 } {
  foreach change [getLemmaUpdates] {
    if {[lindex $change 4] == "failed"} {
       # get all pending and running tasks in the proof.
     set tlist [taskList -r -p [lindex $change 0]]
      if { [llength $tlist] != 0 } {
        taskKill $tlist
      }
    }
  }
}
…
```

6.1.3.4.3   Optimizing Proof Run Time

This declares a proof finished as soon as all lemmas are proved regardless of which task proved them.   As described in "The proofwait Command" on page 69, you can declare a proof finished even though all the tasks of the proof are still running. Unless the -nokill switch is specified, if all lemmas in a task are proven (even by other tasks), that task will be terminated.

```
      …
solveNB p1
solveNB p2
 set proofs [list p1 p2]
   while { 1 } {
     set ret [proofwait -anyChange $proofs]
      if {$ret != 2} {
        break
      }
    }
```

6.1.3.5   Interactive Proof Creation and Control

The `solveNB` command creates a proof, and possibly sub-proofs, and also creates tasks to execute the proofs. You can gain more fine control over proof execution using alternative commands. To create a proof (and sub-proofs) without executing them right away use the following command:

```
gen_proof     # Generate a proof with a given name.
           [-ual ualName] (Name of the Assumption and Lemmas TCL procedure)
           [-script scriptName] (Name of solve script)
           [-init] (Assume initial state)
            proofName (Name of the proof)
```

The `-ual` option and `-script` option can be used to over-ride the settings of TCL variables `user_assumes_lemmas_procedure` and `custom_solve_script` respectively. The `-init` option is used when you want to prove the lemmas assuming a specific initial state.

Once proofs are created, you can use the `listproofs` command to display them. To start tasks to execute one or more proofs, use the following command:

```
run_solver # Restart solving tasks in a proof.
           [-script scriptName] (Name of solve script)
           [-all] (Start solvers in all leaf proofs)
           [proofName] (Name of the proof)
```

The `-script` option provides an opportunity to over-ride the previously specified solve script. The `-all` option is used to create and start tasks to run all proofs and sub-proofs that include unsolved lemmas. Proofs with all lemmas solved are not started. Use the `proofName` option to start tasks associated with a specific proof, otherwise tasks will be created for all proofs containing unsolved lemmas.

Sometimes you may want to stop running a particular task, but do not want to interrupt other running tasks or loose the results of completed tasks. For example, you might realize that a task is not going to succeed with the chosen `solve` script and want to with to restart with a different script. Or, you may suspect that a hardware problem in a grid system may have caused one of your tasks to hang. You can stop a task with the following command:

```
killTasks [-ids <idList> ] [ -all ] [ -proof <proofName> ]
```

This command stops the execution of the specified task(s) and puts them in the *killed* state. You can specify tasks that needs to be killed using the `taskID` numbers or the proof name. The `-all` option kills all the scheduled or running tasks.

If you want to restart the proof associated with the task, use the `run_solver` command. For more information about using Hector in a multi-processor environment, see section "Hector in Multi-Processor Environment".

6.1.3.6   The proofstatus Command

The following command returns 1 if all lemmas in all proofs in the prooflist were successful. Otherwise, this commands returns 0.

```
proofstatus [prooflist]
```

If `[prooflist]` is not provided, the proofstatus identifies all the top-level proofs and reports the status of each top-level proof:

❖ Returns 0 at the first non-passing proof, and terminates the run.

❖ Returns 1 if all the top-level proofs are passing.

6.1.3.7   Task Timeout

You can control the time allocated for a given task by setting the following Hector variable:

```
set__hector_task_timeout <seconds>
```

The use of this variable is discouraged in a multi-processor environment because of heterogeneous nature of machines. A task that completes on one machine may not finish on another in the same time. This can happen on a single machine as well due to load from other jobs.

This variable is useful when you want to run many tasks sequentially on a single machine.

6.2   Saving and Restoring Proofs

Hector proofs can be saved once they have been created by `solveNB` or `gen_proof`. A subsequent Hector session can restore the proofs, including the status of lemmas that are proven or falsified. This capability can be used to release compute resources and restart the proofs later when resources become available again. Another possible use is to check-point a proof with many sub-proofs that is partially complete and then try alternative solving strategies on the remaining sub-proofs without having to repeat the sub-proofs that have already solved. Sub-proofs that have inconclusive lemmas at time of the save may be restarted after the restore operation.

To save the status of the Hector proofs, use the following command:

```
save_proofs <project_name>
```

The `<project_name>` identifies the saved proof state and is stored under the `SCV_WORK` directory. You can execute `save_proofs` more than once during a hector session as long as you provide unique project names.

In a subsequent Hector shell invocation, you can restore the status of the saved proofs with the command:

```
restore_proofs <project_name>
```

The `listproofs` command can be used to check the list of restored proofs. The `restore_proofs` command does not resume execution of the uncompleted proofs. To restart the unfinished proofs use the command:

```
run_solver -all
```

If you only want to start a specific proof, you can provide a proof name, and an optional solve script to override the previously specified script:

```
run_solver <proof_name> [ -script <script_name> ]
```

After executing `run_solver` you can monitor progress with the `listtasks` command.

6.3   Automatic Lemmas

Hector generates lemmas for variety of checks; this section describes these lemmas. In many cases you can disable checking of these lemmas to improve runtime performance. This may cause Hector to report *conditional* proof results since disabling the automatic checks may hide potential problems.

6.3.1   Assertion Lemmas

For each assert statement used in the C++ model, where the compiler cannot figure out if the assertion always holds, a lemma is generated. The names of these lemmas start with the prefix `_scv_as_`.

🛑 **Caution**

> You can disable the checking of the max iteration lemmas by setting the following Hector variable to false. Disabling the lemmas is not recommended.
>
> ```
> set_hector_check_asserts false
> ```

6.3.2   Excess Shift Lemmas

Excess shift lemmas check if a shift with a shift value that was greater or equal the bitwidth of the first operand occurred in C++ model. This is not necessarily a problem if the result doesn't propagate to the outputs. So by default these lemmas are disabled.

You can enable the checking these lemmas by setting following flag. These lemmas start with the prefix `_scv_shft_`.

```
set_hector_check_excess_shift true
```

6.3.3   Mandatory Automatically Extracted Properties (AEPs)

Following automatic lemmas are mandatory because they are assumed to be true in the proofs of the user-defined lemmas. This means that if any of the automatic lemmas below do not prove, all the results for the user-defined lemmas may be invalid. Therefore, if the results of the Hector-generated lemmas are ignored (or if the generation of these lemmas are turned off altogether), false positive and/or false negative results are possible for the user-defined lemmas.

6.3.3.1   Array Bound Violation Lemmas

For each array access in the C++ model, where the compilation step cannot figure out if there is a bound violation, a lemma is generated. The names of these lemmas start with the prefix `_scv_bd_`.

🛑 Caution

You can disable the checking of the bound lemmas by setting the following Hector variable to false. Disabling these lemmas is not recommended.

```
set_hector_check_array_bounds false
```

6.3.3.2   Max Iteration Lemmas

For each max iteration pragma used in the C++ model, Hector generates a lemma to check if it is really an upper bound. The names of these lemmas start with the prefix `_scv_max_`.

🛑 Caution

You can disable the checking of the max iteration lemmas by setting the following Hector variable to false. Disabling these lemmas is not recommended.

```
set_hector_check_maxiter_pragmas false
```

6.3.3.3   Division by Zero Lemmas

Hector inserts lemmas to check whether division by zero can occur or not in C++ model. The names of these lemmas start with the prefix `_scv_div_`.

🛑 Caution

You can disable the checking of the division lemmas by setting the following Hector variable to false. Disabling these lemmas is not recommended.

```
set_hector_check_divbyzero false
```

6.3.3.4   Null Pointer Lemmas

Null pointer lemmas check if a C++ pointer dereference can ever be executed on a null pointer. These lemmas are enabled by default, but you disable the checking these lemmas by setting following flag. These lemmas start with the prefix `_scv_nullptr_`.

Caution

> You can disable the checking of the null pointer lemmas by setting the following Hector variable to false. However, disabling these lemmas is not recommended.
>
> ```
> set_hector_check_nullptrderef false
> ```

6.3.3.5   RTL Lemmas

Hector's RTL compiler will add AEPs for following situations where behavior is either explicitly unspecified by the standard or is considered an error. Errors that are data-independent are mostly caught and diagnosed by the front end. The compiler adds AEPs in following cases.

1. Divide by zero (DBZ)

2. Out of bounds (OOB) checks. These can be introduced due to:

    a. Data dependent array accesses. The AEP checks that access is within the bounds.

    b. In VHDL, an integer-valued expression can be assigned to an integer subrange type, but any values assigned must be in range. Writing an out of range value will cause runtime errors in simulation (if checks are enabled; for VCS the check option is needed to enable checks), and synthesis will treat out of range values as *dont-care* (potentially creating surprising results). The RTL compiler generates AEPs at each possible point where an out-of-range assignment can occur. Namely a signal assignment, a variable assignment, passing an argument to a user-written function, or passing an argument to a builtin function. All of these points are checked for out of bound access.

        An example of RTL AEPs is given in section "RTL Automatically Extracted Properties (AEPs) Example".

Caution

> You can disable the checking of the RTL division-by-zero and out-of-bounds checks lemmas by setting the following Hector variable to false. However, disabling these lemmas is not recommended.
>
> ```
> set_hector_check_comb_aeps false
>
> set_hector_check_reg_aeps false
> ```

6.3.4   Controlling AEP Lemmas

In addition to the individual AEP control commands already discussed, Hector provides a single command to enable or disable all the AEPs. This removes the need to use individual AEP control commands to enable/disable AEPs in certain situations.

```
set_aep_selection < disable_all | default | enable_all >
```

The `disable_all` option stops the checking of all automatic lemmas. The `enable_all` option enables checking for all AEPs. The `default` option restores the default condition of either enabled or disabled to each check.

You can generate a report of the AEP's in *enabled* state using the `list_aep_selection` command. For example:

```
hector> list_aep_selection
        maxiter_pragmas        true
        array_bounds           true
        divbyzero              true
        nullptrdefef           true
        asserts                true
```

```
comb_aeps              true
reg_aeps               true
excess_shift       false
falsepaths         false
```

You can also use the `get_aep_selection` command to return the same information in the form of a TCL list.

6.4   Reports and Logs

After the solve command completes, the result of all the lemmas are listed as shown below.

```
lemma _scvout_0(3): spec.out_a[31:0] = impl.out_a[3:0] : success

lemma _scvout_1(3): spec.out_b[31:0] = impl.out_b[3:0] : failed
```

The exact name that Hector assigns to each lemma is shown in the above list.

The listing above indicates that the logic that computes `out_a` in both the C/C++ and the RTL model is functionally equivalent. However, the logic cones that drive `out_b` in the two models is not the same. Each lemma will have one of the following status indications.

Table 6-1   Lemma Status Indications

success	The lemma was proven to be true
failed	The lemma was proven to be false
nottried	The proof procedure stopped before this lemma was processed
inconcl	The proof procedure timed out trying to prove this lemma
conflc	The constraints or assumptions were inconsistent (contradictory)
error	Indicates memory out occurred or an error happened when using MP (multiple-processor) Hector, use the `listtask` command to see more information.
killed	Indicates that the task was terminated by the user or by a compute resource manager.
cond-sucess	Indicates the solvers returned success for a user-defined lemma, but not all mandatory AEPs resulted in success. This could occur because: • The proofs for some mandatory AEPs have not yet finished. • Some of the mandatory AEPs do not have success status. • Some of the mandatory AEPs were disabled in the proof.
cond-failed	Indicates the solvers returned failed for a user-defined lemma, but not all mandatory AEPs (section "Mandatory Automatically Extracted Properties (AEPs)") resulted in success. This could occur for the same three reasons mentioned for the cond-sucess status.

Note also that whether or not a user-defined lemma is categorized as *success* or `cond-success` may change during the course of a Hector run. Initially, before any mandatory AEPs finish, all user-defined lemmas that result in `success` will be reported as `cond-success`. However, once all the hector-generated lemmas result in *success*, then those same user-defined lemmas will be reported as `success`.

In order to provide backward compatibility, a TCL variable will control whether the conditional success/failed status are used. The following line will cause the new conditional status to be not used.

```
set_use_new_proof_status "false"
```

The default for this variable is *true* which uses conditional success/failed status. We recommend uses to not disable reporting of conditional success/failed status as it can mask real problems with the design/setup.

6.4.1   Viewing Status of Proofs

You can view the status of proofs issuing following commands:

❖   List status of all existing proofs.

```
listproofs
```

❖   List status of all existing tasks. If the `-summary` switch is provided, print the number of tasks with each status and some runtime statistics.

```
listtasks
```

❖   Select a proof.

```
cdproof <proofname>
```

❖   Lists the assumptions and lemmas in the currently selected proof.

```
listproof
```

❖   Lists the assumptions in the currently selected proof.

```
listassumes
```

❖   Lists the lemmas in the currently selected proof.

```
listlemmas [-status <status>] [-type <t>]
```

Where `<status>` is one of the status indications listed above. For example, you can do following to get list of failed lemmas:

```
listlemmas -status failed
```

The type argument `<t>` can take two values `aep` or `user`. For example, to get list of AEP lemmas:

```
listlemmas -type aep
```

❖   Returns a TCL list of all lemmas.

```
set lemmaList [getlemmas -status <status>]
```

The `getlemmas` command can also be used to obtain lemmas with a specific status.

❖   Lists information (error messages, mostly) posted by the specified task.

```
listtask <taskId>
```

❖   List details of tasks such as which lemmas are present in which task.

```
listTaskDetails [-ids <idList> ] [ -proof <proofName> ]
```

❖   Returns a TCL list of information reported by `listtasks`

```
gettasks
```

6.5 Solve Scripts in Hector

A proof in Hector is orchestrated using a solve script. You do not need to be aware of the details of the solve script, however they sometimes have to change the solve script depending on the difficulty of the equivalence problem.

Hector uses many state-of-the art solvers and advanced techniques for performing equivalence checking proofs. A solve script tells Hector how to use various solvers. Having the right solve script is important in order to solve a given problem in reasonable time.

6.5.1 Overriding the Solve Script

Sometimes the default solve scripts are not strong enough for the current problem. This will be indicated by `inconcl` or inconclusive results on the lemmas. Furthermore, it may also be possible to improve the run-times by selecting an alternate solve script. Hector comes with various predefined solve scripts. You can make use of a predefined solve script by setting the `custom_solve_script` variable in the command script TCL file.

Usage example:

```
set_custom_solve_script  "orch_abo_sat"
```

The list of available solve scripts can be obtained by following command:

```
get_all_solve_scripts
```

Table 6-2 lists guidelines on what solve script to use depending on the type of designs. It is recommended that you try the underlined scripts when you are not sure about the script to use. Hector uses a default solve script if no solve script is selected.

Table 6-2   Solve Scripts

Design characteristics	Solve scripts to try
CLASS A: Designs that do NOT make use of non-linear arithmetic. That is, designs that do not contain multipliers/dividers with non-constant inputs. **Examples**: Integer and Floating point (FP) addition, subtraction, comparison. Conversions involving FP and integer operands. Resource allocation algorithms.	orch_satonly orch_abo_sat orch_sat_bb orch_abo_sat_bb orch_abo_equiv_sat_bb
CLASS B: Data path dominated designs. Can contain non-linear operators such as multiplication or division. **Examples:** Discrete cosine transform, Designs with data path optimizations, Designs with many multipliers, Double precision floating point addition	orch_rr_sat orch_rr_inteq_sat orch_expensive_solve1 orch_expensive_solve2 orch_configuration2_a orch_expensive_solve* scripts be very slow for large designs. In general, more powerful (but slower) than scripts in CLASS A.

Table 6-2   Solve Scripts

CLASS C: Designs implementing floating point multiplication **Design:** Single and double precision floating point multiplication, floating point multiply and add	orch_multipliers orch_custom_fma
CLASS D: Designs with truncated multiplier implementation	orch_custom_decompose Set the Hector variable '_hector_rew_decompose_splitter' to specify the number of truncated columns in multiplier.
CLASS E: Designs with few input bits (less than 32). For example, sin, cos, reciprocal of single precision floating point numbers.	orch_custom_bit_operations orch_custom_bit_operations1

Note

> The solve scripts in each class are listed in increasing order of their solve power. A more powerful script is more likely to solve a problem than a less powerful script. However, a more powerful script can consume more time and memory.

The solve script can be specified for each proof as well, by setting:

```
set custom_solve_script_by_proof_name(<name>) "solve script name"
```

This will override the solve script for the proof named `<name>`.

6.5.2   Running with Multiple Solve Scripts

When a hard equivalence checking problem is encountered, you can run multiple solve scripts on the same problem, so that the chances of getting a proof or a counterexample are maximized, and the time taken to find the proof is minimized.

In order to run multiple solve scripts (different orchestrations) on the same problem, set the following variable to true.

```
set_hector_multiple_solve_scripts true
```

Hector will use all solve scripts when `_hector_multiple_solve_scripts` is true. For each solve script a task will be created to solve the problem. These tasks can be run in parallel in a multi-processor environment. On a single processor these tasks will be run sequentially.

You can specify a specific set of solve scripts to use by setting the following variable:

```
set_hector_multiple_solve_scripts_list [list solve_script1 …]
```

The name of all solve scripts can be obtained by the following command:

```
show_all_solve_scripts
```

6.5.3   Controlling Resource Limits of a Solve Script

The following Hector variables can be set to control the resource usage of Hector's solvers. You can use the TCL `set` command to modify the default values of these variables. You can also use `built_in` commands such as `set_resource_limit` or `set_orch_init_sat_time_limit` to modify these variables. The benefit of using the `set_*` commands is to catch spelling errors in the variable names.

❖ `resource_limit <num>`

   This variable must be set to a positive integer. It specifies the resource limit for each call to the default proof engine (usually a SAT solver). This variable has an effect on all the solve scripts mentioned above. The default value is 3600. Each solve script invokes the default proof engine with this resource limit once for each pair of outputs being compared. Note that the above resource limit may not correspond to wall clock time. You can retrieve the current resource limit value with `get_resource_limit`.

❖ `orch_init_sat_time_limit <num>`
   This variable must be set to a positive integer. It specifies the resource limit for each call to the default proof engine to solve the assumption constraints. This variable has an effect on all the solve scripts mentioned above. The default value is 120. It is recommended to use the default value, but if your proof is completing with *nottried* status, and *listtask <n>* reports the following message, it might help to try a larger value:

   ```
   Warning: unable to find a satisfying assignment for the constraints before the
   resource limits were exhausted.
   ```

❖ `orch_solve_outputs_effort <num>`

   Case num = 0: Prove the outputs entire word at once.

   Case num = 1: Prove the outputs one bit at a time starting from LSB.

   Case num = 2: Prove the outputs one bit at a time starting from LSB, but use more effort and engines.

   Case num > 2: Prove the outputs one bit at a time starting from LSB, but use expensive orchestration.

   This variable has an effect on only the `orch_multiplier` solve script. The default value is 0.

❖ `_hector_rew_heavy_num_mults <num>`

   Use expensive rewrites only when number of multipliers is less than the specified number. This variable has an effect on only `orch_multiplier, orch_expensive_solve*, orch_custom_fma` solve scripts. The default value is 8. It is recommended to use the default value unless the output of `listtask` shows messages indicating that this value should be increased.

6.6   Hector in Multi-Processor Environment

Hector allows a user to start multiple proofs or tasks in parallel. This section describes the steps for running Hector in multi-processor (MP) mode.

6.6.1   Running Hector in MP Mode

Following steps are required to run Hector in MP mode:

❖ Create a *host* file that specifies the multi-processor resources available to run MP Hector. For LSF and SGE environments this file usually contains a single line. Use the following command to specify the name of the host file:

   ```
   set_host_file <host_file>
   ```

If the `host_file` is not specified Hector will run multiple proofs on the same machine sequentially. The syntax of the host file is described in the next section. The `set_host_file` command must be used before the first call to `solveNB`. You cannot change MP settings after the first tasks are created.

❖ Use the `solveNB <proofname>` command to create a proof. The `solveNB` command returns the control to you immediately so you can issue other `solveNB` command or commands such as `listtasks` that monitor proof progress. You must provide a unique proof name as an argument to `solveNB`. The proof name will be used in reports and in commands such as `cdproof`.

For example:

```
solveNB -ual my_assumes_1 proof1

solveNB -ual my_assumes_2 proof2

solveNB -ual my_assumes_3 proof3

proofwait proof1 proof2 proof3
```

6.6.2 Specifying the MP Resources

Hector supports five types of transports to dispatch workers on multiple remote nodes:

❖ Platform-LSF (Platform Load Sharing Facility)

❖ SGE (Sun Grid Engine)

❖ RSH (remote shell)

❖ SSH (secure shell)

❖ Netbatch

Depending on the transport chosen, you must supply different types of system-specific information. The *hostfile* is a consistent way to communicate information to Hector for any of the above job dispatch mechanisms.

Hector can handle only one job-dispatch mechanism per session. For example, if RSH is the preferred mechanism, the *hostfile* should contain a list of valid hosts. Alternatively, if SGE is the preferred mechanism, the *hostfile* contains the SGE command format and the number of slots (processors) required for the parallel job to execute.

An example of host file specifying LSF as the transport mechanism.

```
# HOSTS INFORMATION FILE (LSF) with 2 slots

# Each valid host definition occupies an entire line.

# Line with '#' as the first character is a comment line.

# enabled| hostname| #slots| tmpDir| mode| command

#

# The end-user environment should be properly configured and capable of starting
LSF jobs

1| | 2 | /tmp | LSF | bsub -q bnormal
```

Table 6-3   Syntax of Host File

Field	Value	Purpose

Table 6-3   Syntax of Host File

1 ('1')	Integer – can be '0' or '1'	A '0' signifies that the host cannot be used, a '1' denotes a usable WORKER machine
2 (' ')	String, valid hostname	This is the hostname of a valid WORKER machine. For SGE, LSF, or NB this field is empty
3 ('2')	Integer – any valid integer	This number is used to denote the number of worker slots on this host (rsh and ssh) or farm (lsf and sge). -1 indicates unlimited.
4 ('/tmp')	String – a valid, writable directory	This directory is used for local data storage. This is critical for performance if there are many parallel worker processes.
5 ('LSF')	String – command mode	The string signifies the command-mode used to connect to the WORKER machine. The valid entries RSH, SSH, LSF, NB, and SGE.
6 ('bsub –q bnormal')	String – the connection command	The given string denotes the remote command used to connect to the WORKER machine. **For LSF the connection command must start with 'bsub'.** If your environment has a custom LSF command you need to alias bsub to the custom LSF command for your environment. For example, alias bsub 'qsub –m rel4 –q o_cpu_8G' Similarly, for SGE the connection command must begin with 'qsub'.

Other examples:

```
# HOSTS INFORMATION FILE (SGE) with 3 slots

# Each valid host definition occupies an entire line.

# Line with '#' as the first character is a comment line.

# enabled| hostname| #slots| tmpDir| mode| command

#

# The end-user environment should be properly configured and capable of starting
SGE jobs

1| | 3 | /tmp | SGE | qsub -P bnormal –l arch=glinux
```

6.6.3   Specifying the MP Resources via TCL Commands

An alternative to specifying the MP resources in a host file is to provide the same information in four TCL variables. Setting the following TCL variables:

```
set_host_file_num_workers "10"

set_host_file_temp_dir "/tmp"

set_host_file_transport "LSF"

set_host_file_grid_cmd "bsub -q bnormal"
```

creates a host file containing the following single line:

```
1 | | 10 | /tmp | bsub -q bnormal
```

Note

All four variables must be set to something other than "" to generate a host file.

6.6.4   Additional Steps for SSH

When using SSH as the remote dispatch mechanism perform these additional steps to make SSH password-less. This needs to be done once by each user.

```
% cd  ~

% mkdir .ssh

% cd .ssh

% ssh-keygen  -t dsa

  Generating public/private rsa key pair.

  Enter file in which to save the key (~/.ssh/id_dsa): <return>   (no entry)

  Enter passphrase (empty for no passphrase):   <return>  (no entry)

  Enter same passphrase again: <return>  (no entry)

  Your identification has been saved in ~/.ssh/id_dsa

  Your public key has been saved in ~/.ssh/id_dsa.pub

% cp  id_dsa.pub   authorized_keys

% vi config      (Put the following lines in the file)

  UserKnownHostsFile=/dev/null

  StrictHostKeyChecking=no

  PasswordAuthentication=no

  ForwardX11=no

  LogLevel=QUIET

% ssh 0 date    (Testing, no password should be required)

% ssh <somehost> date    (Testing, no password should be required)
```

7 Debugging Failed Lemmas

Hector provides various options to debug the lemmas that failed during the proof. This chapter describes the various debugging options in the following sections:

- ❖ "Counter-examples"
- ❖ "The simcex Command"
- ❖ "Inserting Debug Code in the C++ Models"
- ❖ "Debugging with DVE"
- ❖ "Debugging using the Hector Debugger"
- ❖ "Debugging Mandatory AEPs"
- ❖ "Sharing Counter Examples"

7.1   Counter-examples

When Hector disproves a lemma, a counter-example is generated to illustrate one example scenario leading to the lemma failure. A counter-example generally consists of assignments to input variables and state variables. Execution of the spec and *impl* designs for one transaction will expose the lemma failure. Each counter-example applies to a single lemma failure. If Hector finds multiple lemma failures, it will generate multiple counter examples, but only one per lemma.

Any inputs or state variables that do not affect the success or failure of the target lemma are left in `don't care` states. When debugging in an RTL environment, or with Hector's HDB C++ debugger, `don't care` variables appear with X values. C debuggers (`gdb` for example) do not support the concept of unknown values so `don't care` variables are assigned random values.

Counter examples in C/C++ programs may include state variables on the call stack. This includes both the program counter and stack pointer so C counter-examples may appear to begin at a point of execution different from the start of the initial transaction. The Hector debugger (HDB) can handle this type of counter example as explained in the subsequent sections.

7.2   The simcex Command

The starting point for counter example debug in Hector is the `simcex` command whose syntax is as follows:

```
simcex <lemma_name> [-add n] [-vcd name] [-txt name] [-assumes name] [-vpd path]
[-gui] [-dve] [-fsdb path] [-verdi] [-separate_windows]
[-joint_compile] [-gdb] [-ddd] [-hdb] [-xterm] [-name <spec|impl>]
```

This command simulates a counter example found by the `solve` command and dumps or displays the result in a variety of formats. For the C to RTL flow, the most common usage of this command is:

```
simcex <lemma_name> -dve
```

Table 7-1 lists the options for the simcex command:

Table 7-1   Simcex Options

-add *n*	Adds n phases of simulation to the end.
-vcd *filename*	Generates the counter-example from the formal model. Writes the counter-example to filename in the VCD format.

-txt *filename*	Generates the counter-example from the formal model.
	
	Writes the counter-example in to filename in text format. The text file has two sections. The first section provides the input values for each phase. The format of each row is:
	
	    Signal_Name Value Offset Bitwidth
	
	The second section prints results of simulating the design on Hector's formal model (DFG) for each phase. Only the signals that change are printed. The format of each row is:
	
	    Signal_Name Value

-assumes *filename*	Generates the counter-example from the formal model. Writes the counter-example to filename in Hector assume format.
-vpd *pathname*	Generates the counter-example from the simulation model. Writes the counter-example in VPD format. Dumps the waveforms to three VPD files. The <pathname> is broken into dirname and basename parts as the standard unix utilities would. The *basename* part of <pathname> is part of the resulting dump filename. The files are saved in $work/cex_simulation and are named <basename>.spec.vpd, <basename>.impl.vpd, and <basename>.info.vpd. The *info* VPD file contains extra Hector-specific information, such as the hector_phase signal. These result files are copied to the *dirname* part of <pathname>.
-dve	Runs DVE on the result of the [-vpd] option. If [-vpd] is not specified, then Hector uses the lemma name as the basename and no files are copied.
-gui	Runs DVE at the start of simulation for RTL debugging. This is the same as simv –gui. Prevents generation of the VPD or FSDB file.
-fsdb *filename*	Generates the counter-example from the formal model. Writes the counter-example to filename in FSDB format.
-verdi	Infers the -fsdb option. The FSDB file will be <lemmaname>.fsdb. The environment variable VERDI_HOME must be set.
-hdb	Debug the counter example in the Hector HDB debugger. The debugger will use Gnu DDD (Data Display Debugger) if it is visible on your path, otherwise it will start in command prompt mode in an xterm.
-xterm	If option –hdb is specified, run HDB in an xterm, not in DDD. Without option –hdb, -xterm has no effect.
-gdb	Debug the counter example in the Hector GDB debugger. The debugger will use Gnu DDD (Data Display Debugger) if it is visible on your path, otherwise it will start in command prompt mode in an xterm.

Table 7-1   Simcex Options

-ddd	Runs DDD (Data Display Debugger) on the result of the [-gdb] option. DDD must be installed and visible on your path.
-name <spec|impl>	This option is used to specify which design to debug, 'spec' or 'impl' (the default is spec for -hdb, -gdb, and -ddd). For -dve and -verdi, both designs are debugged by default. This option is valid only for the -dve, -verdi, -hdb, -gdb, and -ddd options. In all other cases this option is ignored.
-joint_compile	(-vpd and -dve only) Attempts to compile both designs and the information module into a single design that has multiple Verilog top modules. A single dump file is generated. This is the nicest presentation when it works, but there are many, many design configurations and options which will cause this flow to fail, often without a useful error message. Caution is advised. If this option fails, but the same "simcex" command without the option works, then the designs probably cannot be compiled together. This *should* work for most C to RTL comparisons where the RTL is Verilog-top. It *will* fail for all VHDL-top designs.
-print	Turn on logging of counter-example simulation messages. By default these messages are suppressed.

7.2.1   The *Simulation Debug* Flow

The `-vpd`, `-dve`, and `-gui` options all start the simulation debug flow. Your designs are compiled with VCS using the same options that were specified in the `create_design ... compile_design` sequence in the Hector setup file. The stimulus from the counter-example on the formal model is applied to each design in the simulation environment with full debug enabled. This provides the most accurate playback of the counter-example in terms of the model that the designer is most familiar with - the simulation model.

By default, Hector compiles and simulates the spec and impl models separately, then loads the signal data into a single DVE window. It also generates a module named info that contains a transaction phase reference counter. This may be helpful in tracing the timing of inputs and outputs of pipelined designs. See section "Debugging with DVE" for a simple example of this flow.

The DVE source window only shows one design at a time and has a pull-down to choose between them. Its waveform window will accept signals from any of these. This is the most general solution for DVE. It will work with RTL2RTL even when both designs are VHDL-top. DVE should never need this option.

The `simcex` command also has a backward-compatibility option: `-joint_compile` which will attempt to compile both designs and the information module into a common design. If successful, it will dump a single waveform file which will appear to have multiple Verilog-top modules. In this case, there are no pull-downs to manage on either display tool. This flow works for many kinds of C2RTL proof and for some RTL2RTL proofs. There are many, many design configurations which cannot be compiled with VCS in this way, and they will fail, often without a useful error message. If you experiment with this option and it fails, please try again without it. All VHDL-top designs will fail with this option.

The `-gui` option starts DVE in debug mode, the same as invoking `simv -gui` on a VCS compile. DVE is a source-level debugger, you can set breakpoints, force values, and single step through your design. Useful for many things. For more information, see the *VCS®/VCSi™ User Guide* and *VCS®/VCSi™ Discovery Visualization Environment User Guide*.

7.2.1.1   Resolving Mismatched Timescales

The *info* module and any modules that represent C code are separately compiled Verilog modules. Their timescale can differ from your RTL if you have explicit timescale or simulator precision directives. As it rather difficult to discover this in advance, Hector provides a command that you can make these other Verilog modules match:

```
set_simcex_precision (1|10|100)(s|ms|us|ns|ps|fs)
```

Example: `set_simcex_precision 10ns`

This command can be placed at the top level in your `command_script.tcl` file, or you may run it manually immediately before a `simcex` command. The precision is the second number in a Verilog `` `timescale `` directive. So if your timescale is 1ns/1ps, then 1ps is what you want to pass along to `simcex`. VHDL does not have an equivalent to *timescale*, but if you have a `sim_res`, `time`, or `time_res` option on your VCS simulation, that is the value we want here.

This command only impacts the compile of the C and *info* modules, it does not modify your RTL build(s) in any way.

7.2.2   Using the Formal Model Flow

The `-fsdb`, `-verdi` and `-vcd` options all start the formal model debug flow. The `-fsdb` and `-verdi` options only work when the `create_design` and `compile_design` compile commands are used to compile the design. If the older style (and now deprecated) - 'compc' and 'comprtl' compile commands are used, the `-fsdb` and `-verdi` options cannot be used.

The `simcex` command with the `-fsdb <filename>` option generates the counter-example from the formal model and dumps a FSDB file into *<filename>* and creates a file verdi.play in the current directory. The `-verdi` option implies the `-fsdb` option and additionally, it brings up the Verdi debugger and populates it with the design data and waveform information. When the `-fsdb` option is used without the `-verdi` option, the following command populates the Verdi browser with the design data and the Verdi waveform window with the signals from the failing lemma.

```
verdi -play verdi.play
```

The `simcex` command with the `-vcd` option dumps a Verilog VCD file. This waveform file contains signals internal to Hector as well as those from the RTL or C designs. The signal names contains prefixes or suffixes to identify its role (input, output, ps (present-state of a register), or show). Furthermore, the formal compilers can remodel signals such that a signal with a given name in the VCD dump may not correspond exactly to the values that the signal would exhibit in the original RTL.

7.2.3   Differences Between the Simulation Debug and Formal Model Views

For the most part, the simulation and formal views should agree. There are some important differences worth noting:

7.2.3.1   Simulation View Shows Incorrect Value on Cutpoint After Release

Hector allows cutpoints to be asserted per phase. When the cutpoint is released, the simulator has no idea what value would have been there, so the current value stays the same until the next update of that signal. This incorrect value can propagate to registers and cause the two views to diverge. The workaround for this is to cut signals and assign values for the whole proof rather than for isolated phases.

7.2.3.2   Simulation and Formal Views Disagree on Cutpoint Value While Forced

This is an intentional design choice, but it causes confusion when comparing the two views. Suppose you cut signal `foo` in your proof with `cutpoint bar foo(4)`. In the formal model, `foo` refers to the original signal and can still be used in lemmas. The value of `foo(4)` will be the same as it always was. `bar` is an override for the value of `foo` at phase 4, and its value is forced onto the loads of `foo`.

In the formal waveform view, `foo` is unchanged. The override `bar` does not exist as a separate signal in this view, but the override can be examined by looking for signals with names like `$cut_foo` and `$cuten_foo`. `$cut_foo` is the override value, it is enabled when `$cuten_foo` is true.

In the simulation debug view, there is only `foo`. There is no mechanism in the simulator to support separate value-holding objects `foo` and `bar` in your original RTL, so Hector forces `foo` to the cutpoint value. The rest of the simulation is correct, the only thing that is missing is that you cannot see what the original value of `foo` would have been.

This means that when you are comparing VCD and VPD dumps, they will disagree on the value of `foo` on those phases where the cutpoint is active.

7.2.3.3   Simulation and Formal Views Disagree due to Explicit Non-zero Delay

The formal model of the RTL is compiled with the synthesis notion that all explicit delays are zero. That is, the actual time value of the delay is ignored. When the counterexample is simulated on the RTL, the simulator will substitute the original delay that was specified. If the difference in delays causes any difference in the order of edges, and there are processes that are sensitive to those edges, (clock buffers and clock gating) then the simulation replay can be considerably different than the formal view.

This represents a potential simulation/synthesis mismatch that Hector itself cannot detect. If VCS is your simulator, you might try diagnosing this using the `+delay_mode_zero` option which forces all explicit delays to zero.

7.3   Inserting Debug Code in the C++ Models

Hector supports two mechanisms to help debug C++ models. The first is `Hector::printf()`. This function is the Hector analog to the `printf()` command in the C library. It can be placed anywhere in the C model. For instance:

```
int position = x + y;

Hector::printf("The value of position is: %d\n", position);
```

During either of the `simcex` replay the value of the variable *position* will be printed out in the Hector command transcript every time the command is executed. Note that the `-print` switch must be added to the `simcex` command to enable `Hector::printf()` reporting.

Examples:

❖ To print an *unsigned long* use the type specifier `%lx` (x is for hexadecimal).

```
unsigned long A;

Hector::printf("A=%lx", A);
```

❖ To print a *unsigned long long* use the type specifier `%llx`

```
unsigned long long A;

Hector::printf("A=%llx", A);
```

7.3.1   Using Hector::show Commands

In the C++ model, temporary variables are not accessible in Hector proofs. Only global variables and variables that are declared as inputs/outputs (through `registerInput()`, `registerOutput()`) are visible. To make temporary variables visible (to the prover as well as in the counter-example waveform), Hector provides the `Hector::show()` and `Hector::showall()` commands.

The `Hector::show` command is currently implemented in the following forms:

```
template<class T> static void show(const char* txt_p, const T& v);

template<class T> static void show(const char* txt_p, int i1, const T& v);

template<class T> static void show(const char* txt_p, int i1, int i2, const T& v);

template<class T> static void show(const char* txt_p, int i1, int i2, int i3,
                                                  const T& v);

extern void showall(const char* txt_p);
```

Examples:

```
int a;

Hector::show("my_new_name_for_a", a);

Hector::show("a_incremented", a + 1);


int arr[10];

for (int i = 0; i < 10; i++) {

    Hector::show("arr_elem[%1]", i, arr[i]);

}
```

The first form of the show command makes the temporary value v visible at exactly the point in the program where the show command is called. The `txt_p` argument is a new name under which the value in the `v` argument can be referred to in assumptions and lemmas. The value will also show up in counter-example waveforms for debug visibility.

The value `v` can be a temporary variable or an arbitrary expression of variables. Note that is declared as a reference and not as a pointer, so the user needs to specify the object itself and not a pointer to it. The show command does not prevent the user from specifying a pointer as value. Since pointers in Hector do not carry any printable information, it will always be considered a user error if the value is a pointer. Hector will print an error message in this case and stop the compilation.

The other three forms allow the user to generate more meaningful names for arrays. The i1, i2 and i3 arguments are indices that can be referred to in the name by the position specifiers %1, %2, %3.

Example:

```
int arr[3][4][5];

for (int I = 0; I < 3; i++) {

    for (int j = 0; j < 4; j++) {

        for (int k = 0; k < 5; k++) {

            Hector::show("arr[%1][%2][%3]", i, j, k, arr[i][j][k]);

        }
```

```
        }

    }
```

The position specifiers can be used multiple time in the name (`arr_%1_%2_%1`). It is an error to refer to an index that does not exist (`%0 or %4`). It is also an error to have `a %` followed by anything else but a number. The only exception is a double `%%` which will show up as a single % in the name.

Internally, type information is stored along with the newly generated signal. This means that you can further traverse the type hierarchy in assumptions and lemmas. For example:

```
    struct foo {

        int x;

    }

    struct bar {

        foo a[10];

    }

    bar mybar[5];

    int main()

    {

        Hector::show("mb", mybar);

        Hector::show("mf", mybar[0].a[3]);

    }
```

In assumptions, you can not just refer to `mb` and `mf` but also to their elements and sub-fields:

```
    assume ax = (mb(0)[3].a[0].x == 3);

    lemma lx = (mf(0).x == 2);
```

It should be noted, however, that you should choose the name used in `Hector::show()` wisely. Anything that is not a regular Verilog name needs to be escaped in an assumption or lemma. For example:

```
    Hector::show("myb[3]",  mybar[3]);
```

The name `myb[3]` is not a regular Verilog name (because of the brackets) and so if it should be used in an assumption or lemma, it needs to be escaped:

```
    assume ay = (\\myb[3] (2).a[4].x == 7);
```

👉 Note

The space after the name that marks the end of the escaped name. The phase specification comes after the escaped name.

Note that the same variable may appear in multiple `Hector::show()` function calls at different points in the C++ code execution flow. The same `Hector::show()` call may also be executed multiple times if it is in a loop or sub-function. Each call to `Hector::show()` will store the value of the argument variable at the point of the call. It is recommend that you use a different string argument in each `Hector::show()` invocation, especially when the same variable may be logged more than once during a transaction. If `Hector::show()` is called within a loop, HDB generates unique name strings for each loop iteration by appending digits to the string specified in the function call.

The `Hector::showall()` function provides an easy way to enable debug visibility for all live variables at a point in the C++ code. The string argument passed in the `showall()` method is prefixed to all the live

variables to make names that will appear in counter-example waveforms. During C++ compilation in Hector, messages will be generated when `Hector::show` or `Hector::showall` are executed. Corresponding variable names will appear in counter-example waveform displays.

7.4 Debugging with DVE

For counter example debug of RTL models using waveforms, the most direct method is to invoke the following command at the Hector prompt:

```
hector>  simcex -dve <lemma_name>
```

This will execute a simulation of the counter example in VCS and invoke the DVE waveform debugger to display the resulting waveforms. Figure 7-1 illustrates the DVE hierarchy window and shows the drop-down selector to choose between the specification and implementation designs and a special info module containing a transaction phase counter. Signals can be dragged from each hierarchy into a common waveform window to create a composite view as shown in Figure 7-2.

Note that for RTL designs, all internal signal values are available for display. In C models, only inputs, outputs, and variables referenced in Hector::show() functions are available in the waveform view. C variables declared in `Hector::registerInput()` and `Hector::registerOutput()` functions are listed using the names passed to those functions, with `_in` or `_outs` suffixes appended.

Figure 7-1   DVE Hierarchy Window Showing VPD Selectors
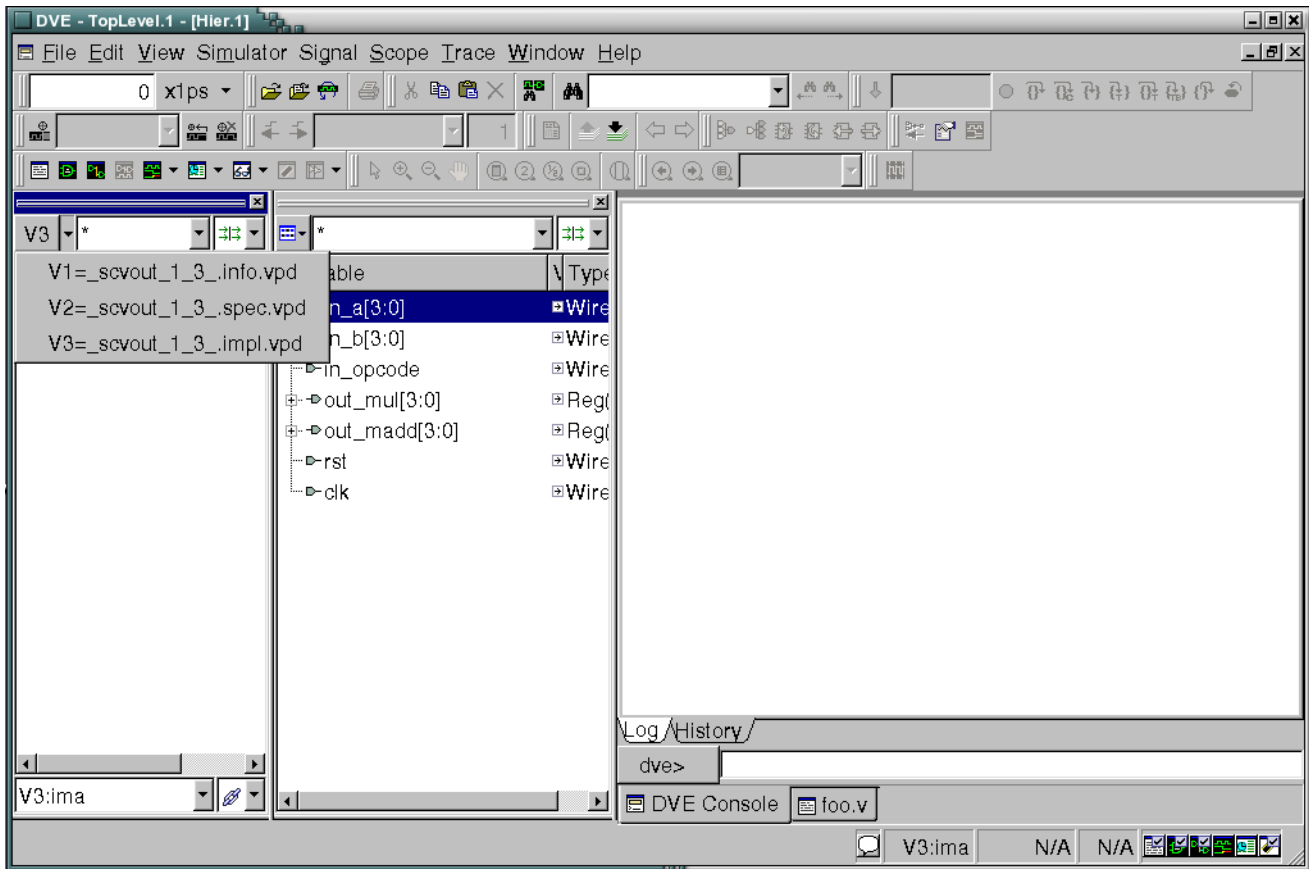
Figure 7-2   DVE Waveform Showing RTL, Phase, and Specification Model Ports
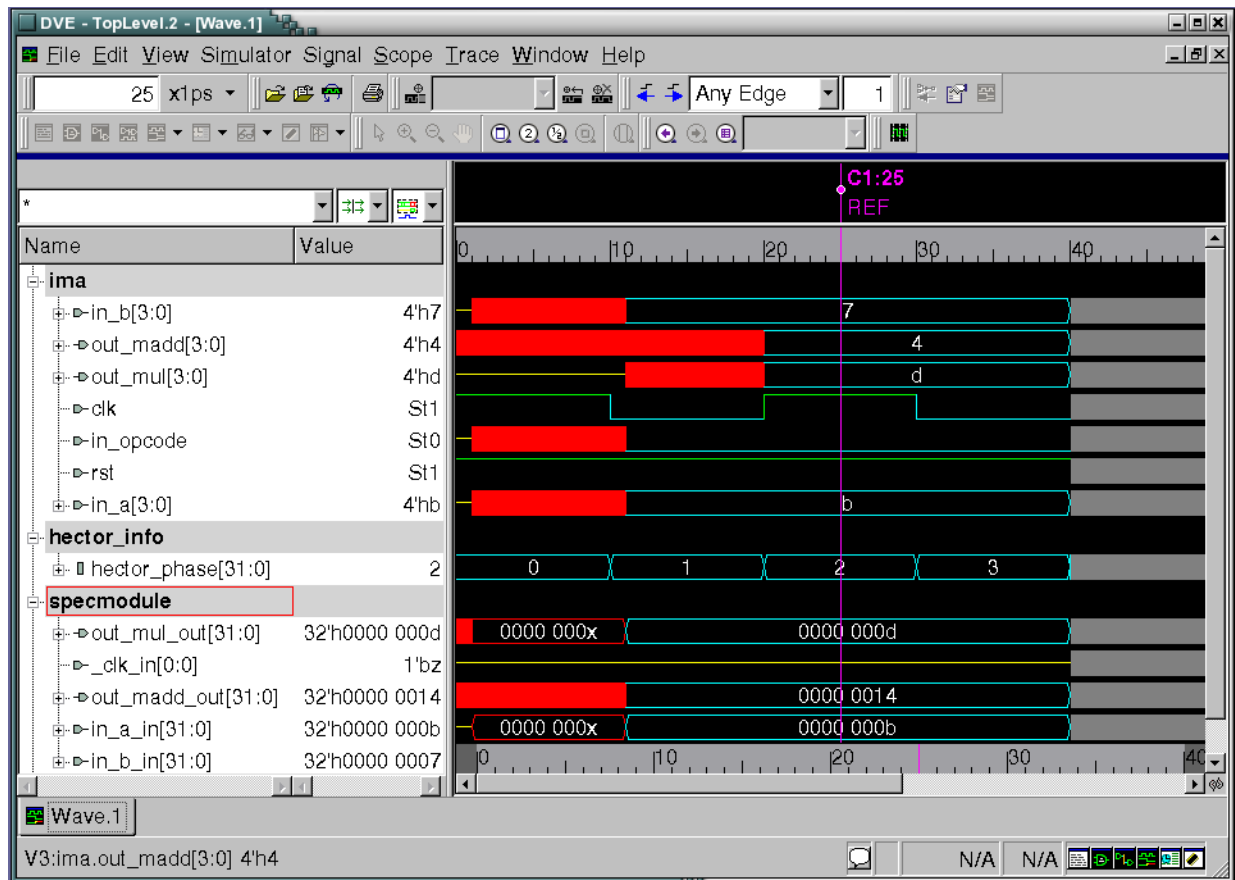
For additional information on using DVE for RTL debug, see the *VCS®/VCSi™ User Guide*.

7.5   Debugging using the Hector Debugger

Hector counter-examples begin at the start of a transaction, marked in the C++ code with the
`Hector::beingCapture()` function call. Note that this may not be the first transaction executed by the
system so state variables may have values only attainable after multiple transactions.

HDB enables debugging of C/C++ programs using Hector's internal data structures but still behaves much
like GDB from the user's perspective. One difference from GDB debugging is that HDB will flag array
bounds violations and track unknown ('X') variable values.

Note
HDB does not yet support debug of SystemC models.

7.5.1   Starting the Hector Debugger HDB

You can start the Hector debugger using the `simcex` command in the Hector shell. The debugger is invoked
in a separate window under the `ddd` debugger interface or in an xterm.

Example: Invoke HDB on the *spec* design with the counter-example for mylemma(1):

```
hector> simcex mylemma(1) -hdb
```

By default, HDB debugs the *spec* design. To debug a C/C++ *impl* design, use the `-name` switch. For example:

```
hector> simcex mylemma(1) –hdb –name impl
```

Figure 7-3   Debugging in HDB with DDD
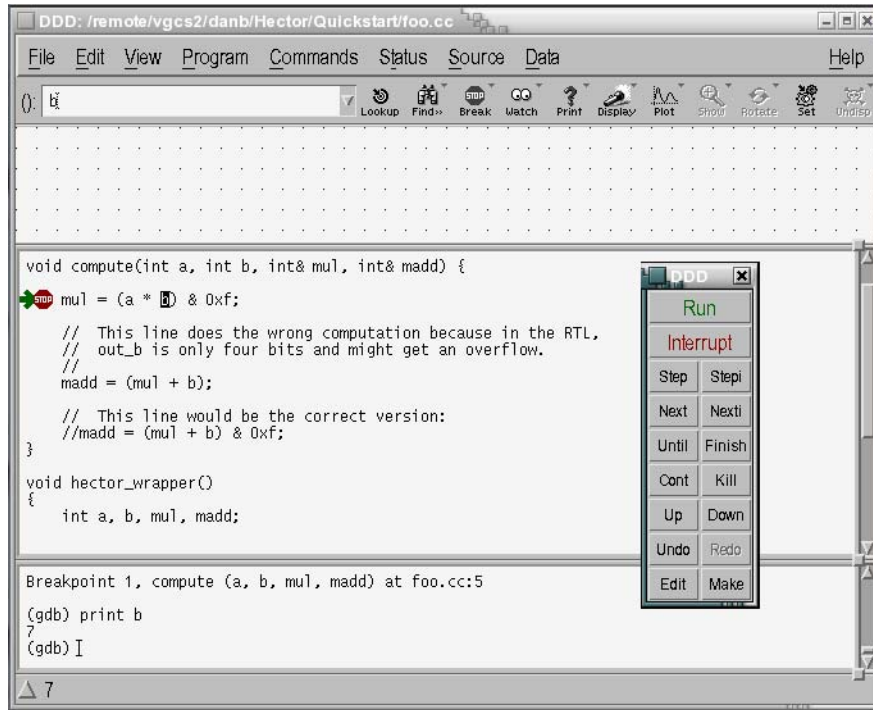
For C/C++ programs, the `simcex` command also produces a file named `pack.hdb` in the `SCV_WORK` directory that contains all the information that HDB needs for debugging the counter-example. It can directly be run in HDB or DDD:

```
sh> hdb SCV_WORK/pack.hdb
```

or

```
sh> ddd --debugger "hdb –gdbmode 2 SCV_WORK/pack.hdb"
```

The main advantage of the `pack.hdb` file is that it can easily be handed over to other designers without having to copy the entire hector directory structure. Note that the `pack.hdb` only packages the control-flow-graph and the counter-example but not the source files. Thus, the other designer will have to run HDB in a directory that has the exact same source files and source directory structure in place.

7.5.2   Using HDB Commands

The Hector Debugger HDB supports a subset of GDB commands. The following section lists supported commands and highlights any differences with GDB behavior.

7.5.2.1   Run commands

The subsequent sections list out the Run commands and their usage.

7.5.2.1.1   The restart Command

Syntax

```
restart
```

Resets the program counter to the beginning of the counter-example. This command does not start the program execution.

7.5.2.1.2   The run Command

Syntax

```
run
```

Starts the counter-example execution. Since execution does not stop before the end of the counter-example, breakpoints need to be set at locations that are of interest.

7.5.2.1.3   The list Command

Syntax

```
list [<location>]
```

Lists the specified function or line. Without an argument, lists ten more lines after or around the previous listing. If an argument is given, this argument must refer to a location in the program. The listing then prints ten lines around that line. Locations can be specified in the following ways:

❖   LINENUM, to list around that line in the currently selected stack frame.

❖   FILE:LINENUM, to list around that line in that file.

❖   FUNCTION, to list around the beginning of that function.

7.5.2.1.4   The step, next Commands

Syntax

```
step(alias: s)

next(alias: n)
```

Single step the program until it reaches a different source line. Currently, no arguments are supported. The step command also steps into function calls. The `next` command does not step into function calls. Instead, the function call is treated like a single instruction. Aliases `s` and `n` exist for step and next.

7.5.2.1.5   The until Command

Syntax

```
until [<location>]
```

If no argument is specified, executes until the program reaches a source line greater than the current location in the selected frame. If a location is specified, executes until the program reaches a source line greater than the specified location. The location may be a line number, a function name, or a colon-separated pair of filename and line number. If a line number is specified, breaks at the start of code for that line. If a function is specified, breaks at the start of code for that function.

Examples:

```
until 342

until foofunction

until myfile.cc:453
```

7.5.2.1.6   The finish Command
Syntax

```
finish
```

Executes until selected stack frame returns. This command is usually used to execute the rest of a function.

7.5.2.1.7   The break Command
Syntax

```
break [<location>]
```

Sets breakpoint at the specified line or function. The location can be a line number, a function name, or a colon separated pair of filename and line number. The location is the same format as described in the list command or the until command. If no location is specified, the current execution point of the selected stack frame is used.

7.5.2.1.8   The cont Command
Syntax

```
cont
```

Continues the program being debugged after a breakpoint was reached.

7.5.2.2   Stack Commands
The subsequent sections list out the Stack commands and their usage.

7.5.2.2.1   The where Command
Syntax

```
where (alias: bt or backtrace)
```

Prints back trace of all stack frames. Currently, no arguments are supported. Stack frames are numbered #0, #1 to #n, where #0 is the current frame.

7.5.2.2.2   The up Command
Syntax

```
up
```

Selects and prints stack frame that called the current one. Currently, no arguments are supported.

7.5.2.2.3   The down Command
Syntax

```
down
```

Selects and prints stack frame that is called by the current one. Currently, no arguments are supported.

7.5.2.2.4   The frame Command
Syntax

```
frame [<number>]
```

Without an argument, prints the selected stack frame. If a frame number is given, the corresponding frame is selected. Frame number 0 refers to the frame of the currently executed statement.

7.5.2.3 Data Commands

The subsequent sections list out the Data commands and their usage.

7.5.2.3.1 The print Command

Syntax

```
print [/FMT] <expr> (alias: p)
```

Prints the value of the given expression. Variables accessible are those of the currently selected stack frame plus all those whose scope is global. Currently, only the following operators are implemented:

```
address_of ('&'), dereference('*'), dot('.'), arrow('->') and array accesses
('[<n>]')
```

The printed expressions are numbered as $1, … $n but it is currently not possible to refer to them in an expression. None of GDBs printing options are currently supported.

The expression may be preceded with /FMT, where FMT is a format letter. The format can be /x (hex), /d (decimal), /u (unsigned decimal), /t (binary), /a (address), /c (char) or /s (string).

7.5.2.3.2 The display Command

Syntax

```
display [/FMT] <expr>
```

Prints value of the given expression each time the program stops. Without an argument, prints all currently defined display expressions. Command `undisplay` or `display delete` can be used to cancel display requests. The `display` command supports the same expressions as the print command. As with the `print` command, a format can be specified. For valid format letters, see the `print` command ("The print Command").

7.5.2.3.3 The info Command

❖ `info line`

Prints information about the currently executed line in the source code and the address of the corresponding CFG node.

❖ `info cex`

Prints the currently loaded counter-example. All variables in the counter-example with their hex-values are printed. Note that this is a HDB specific function and does not exist in GDB.

❖ `info display`

Prints the status of the user-defined display expressions and whether a display is enabled or not.

❖ `info breakpoints`

Prints the status of user-defined breakpoints. The Type column is currently always *breakpoint*. Watchpoints are not implemented. The *Disp* column describes what happens after the breakpoint gets hit. Currently, only *keep* is implemented. The *En* column indicates whether the breakpoint is currently enabled. The *Address* and *What* columns indicate the CFG node and the file / line number of the breakpoint location.

❖ `info sources`

Prints all the source and include files that were used in the compilation.

7.5.2.4   Other Commands

The subsequent sections list out a few other frequently used commands and their usage.

7.5.2.4.1   The enable Command

Syntax

```
enable display [<number1>]…[<number n>]
```

Enables expressions to be displayed whenever the program stops. Arguments are the space separated numbers of some display expressions printed by *info display*. If no argument is given, all display expressions are enabled.

Syntax

```
enable [breakpoints] [<number1>]…[<number n>]
```

Enables some breakpoints. Arguments are the space separated numbers of breakpoints printed by *info breakpoints*. If no argument is given, all breakpoints are enabled. The `breakpoints` keyword is optional. A simple *enable* always refers to breakpoints.

7.5.2.4.2   The disable Command

Syntax

```
disable display [<number1>]…[<number n>]
```

Disables expressions to be displayed whenever the program stops. Arguments are the space separated numbers of some display expressions printed by *info display*. If no argument is given, all display expressions are disabled.

Syntax

```
disable [breakpoints] [<number1>]…[<number n>]
```

Disables some breakpoints. Arguments are the space separated numbers of some breakpoints printed by the `info breakpoints` command. If no argument is given, all breakpoints are disabled. The `breakpoints` keyword is optional. A simple `disable` always refers to breakpoints.

7.5.2.4.3   The delete Command

Syntax

```
delete display [<number1>]…[<number n>](alias: undisplay)
```

Deletes some expressions to be displayed when the program stops. Arguments are the space separated numbers of some display expressions printed by `info display`. If no argument is given, all display expressions are deleted. The deleted expressions will not be printed anymore and will not show up in `info display`.

Syntax

```
delete breakpoints [<number1>]…[<number n>]
```

Deletes some breakpoints. Arguments are breakpoint numbers separated by spaces. If no argument is given, all breakpoints are deleted. The `breakpoints` keyword is optional.

7.5.3   Accurate Bounds and Undefined-checking

In a regular debugger (like GDB), an out-of-bounds access may or may not cause a segmentation fault. This depends on the actual memory layout and how *far* away from used memory the access happens. This means that running the program in the debugger can miss out-of-bounds accesses. If accurate bounds-checking

and undefined-checking is disabled, HDB will behave in the same way. By default, however, accurate checking is enabled. This means that HDB puts more effort into detecting out-of-bounds writes and will generate segmentation faults for them more accurately. Having bounds-checking enabled also allows HDB to check for access to undefined variables. For example, if an uninitialized local variable is read, the result will be undefined. Also, if the result of an operation is not defined by the standard, for example, a shift with a negative shift value, and the result is used subsequently, the result will be undefined. This is a useful tool for debugging X's that are propagated to an output during Hector's counter-example simulation. These X's usually result from reading something undefined. All undefined variables in HDB will be printed as having a value of `<undef>` instead of a random value that a regular debugger would print. To disable / enable bounds-checking and undefined values, use the following command:

```
set bounds-check <on/off or true/false>
```

In addition to printing undefined values, warnings can be printed whenever an undefined value is read by the program. By default, these warnings are enabled. To disable / enable them, use:

```
set warn-undefined <on/off or true/false>
```

You can make HDB stop when undefined values are produced (instead of just printing a warning) by setting the following flag:

```
set fail-undefined true
```

Note that when HDB is used with a graphical front-end, warnings for accesses to undefined variables should be turned off because they will confuse the command parser of the front-end.

7.6   Debugging Mandatory AEPs

Hector automatically adds certain automatically extracted properties (AEPs) to ensure correctness of the model (section "Mandatory Automatically Extracted Properties (AEPs)"). When a lemma is added for an AEP it can be identified by the filename and line number that is shown with listlemmas. An example of an RTL AEP is as follows:

```
lemma spec._SCV_COMB_AEP_0(1): ver.v:8__PROP_DBZ_8 : failed
```

The filename is `ver.v` and the line number is 8 and it is a divide-by-zero (DBZ) check.

```
lemma spec._SCV_COMB_AEP_1(1): ver.v:9__PROP_OOB_9 : success
```

The filename is `ver.v` and the line number is 9 and it is an out-of-bounds (OOB) check.

7.7   Sharing Counter Examples

A typical debug scenario occurs when a verification engineer finds a discrepancy using Hector and then wants to share the counter example with an RTL designer.

Historically, the entire hector working directory (for example, SCV_WORK) had to be put in a tar file and sent to the RTL designer. This approach has the following two drawbacks:

- ❖ The amount of disk space used by SCV_WORK could be quite large.
- ❖ Cross probing in Verdi between the hierarchy browser, the RTL source window, and the waveform viewer generally does not work well.

The capability described in this section is not a complete solution for maintaining cross probing when counter examples are relocated. However, it provides guidelines for ensuring cross probing is available in many more cases.

7.7.1   Use model

When the `simcex` command is run, and either the `-verdi` or the `-fsdb` options is specified, a tar file that contains all the Verdi database files created during the compile step, is created.

If simcex is invoked as…

```
simcex out_check -fsdb out_check.fsdb
```

A file *out_check.tar* is created in the current directory, that contains only the necessary subset of SCV_WORK required by Verdi. When this tar file is unpacked, the following directory structure should be seen:

```
hector_cex.out_check
    |
    +--  out_check.fsdb
    |
    +--  verdi.play
    |
    +--SCV_WORK
        |
        +-- spec_compile
        |
        +-- impl_compile
        |
        +-- verdi_compose_compile
```

To run the counter example in Verdi, use the following commands:

```
cd hector_cex.out_check
verdi -play verdi.play  [-arch32]
```

7.7.1.1   Use Model 1

Use model 1 is enabled if all the RTL files were referenced using absolute file names during the compile step. This, in turn, allows the cross probing to RTL source files when the out_check.tar file is unpacked in a new location.

Use model 1 will also be enabled when relative path names are used if the location in which the out_check.tar file is unpacked. This allows all the RTL files to be accessed with the same relative path names.

7.7.1.2   Use Model 2

Use model 2 can be used when the requirements for use model 1 is not met. In use model 2, when the *out_check.tar* file is unpacked in a new location, any signal value can be displayed in the waveform window of Verdi, but no cross probing to the RTL source code is possible.

Cross probing capability can be restored in use model 2, if the hector setup file is copied to hector_cex.out_check, and hector is used to recompile the design in the new location.

Resolving the proofs is not required. However, this may require some changes to the hector setup script to ensure all the source files are referenced with absolute path names.

8 Advanced Proof Techniques

A proof in Hector consists of a set of assumptions and a set of lemmas (proof obligations) and the goal is to prove the lemmas by making use of the assumptions. This chapter, describes various techniques that are in general very useful in solving hard problems in Hector in the following sections:

- ❖ "Partitioning Lemmas in a Given Proof"
- ❖ "Assume-guarantee Based on Lemma Partitions"
- ❖ "Case Splitting"
- ❖ "Black-boxing"
- ❖ "Using Cutpoints"
- ❖ "Using the Hector Data Path Solver Engine"
- ❖ "Division and Square Root Verification in Hector"

Most hard problems lead to convergence issues in the solvers, and can be addressed by using some of the following techniques. This chapter uses concepts of proofs and tasks as introduced in section "Proofs, Sub-proofs, and Tasks".

Some of these techniques create multiple problems that need to be solved. These sub-problems can be solved sequentially or in parallel. Multi-processor (MP) Hector provides a convenient way of distributing these problems across multiple processors.

8.1   Partitioning Lemmas in a Given Proof

You can break a proof consisting of many lemmas into a number of tasks each containing a smaller number of lemmas (but with the same assumptions). In order to enable partitioning of lemmas the user needs to set the following variable to true:

```
set_hector_lemma_partition true
```

Hector puts each lemma in a new partition; that is, each task will contain one lemma to be proven.

You can also control the number of lemmas in the tasks by setting the following variable:

```
set_hector_lemma_partition_size n
```

When the partition size is specified Hector will create a task for lemmas 1 to n, a task for lemmas n+1 to 2n, and so on. The lemmas are considered in the order in which they are specified by you in `user_assumes_lemmas_procedure`. You can also create partitions of different sizes by setting the above variable to a list of partition sizes.

```
set_hector_lemma_partition_size [list n1 n2 n3 … n{l}]
```

where `n{i}` denotes the number of lemmas in partition `{i}`. If the sum of partition sizes `(n1+ …. + n{l})` is less than the total number of lemmas, then the remaining lemmas are put into partitions of size `n{l}` each.

The user can also finely control the lemmas in each partition and the solve script used for the partition by providing a list of partitions (list of lemma names or patterns) and optionally the name of solve scripts to use for solving lemmas in a given partition. The syntax is:

```
set_hector_lemma_partition_list [list partition1 partition2 …]
```

where, `partition<i>` is a list which can contain lemma names or patterns (all lemmas matching with the pattern are included in the partition). In addition, `partition<i>` can contain at most one entry of the form `"-script <solve_script_name>` to specify a solve script to use when solving the lemmas in a partition. Some examples are as follows:

```
set_hector_lemma_partition_list [list [list le29 le41] _scv_bd*]

set_hector_lemma_partition_list \

    [list  [list le29 le41 "-script solve_hard"] \

           [list _scv_bd*  "-script solve_easy"]]
```

☞ Note

The pattern `_scv_bd*` matches all lemmas starting with the prefix `_scv_bd`.

The partitions that you provide are not required to be disjoint or complete. Hector will generate an additional partition to include any lemmas that were not specified by the user.

You can set the following variable so that Hector does not complete the lemmas that were not specified in the `hector_lemma_partitions` list.

```
set_hector_lemma_partition_complete false
```

Figure 8-1 shows that multiple tasks are created when the above option is given to solve the original proof P. Each task can operate on a subset of lemmas in P and can have a different solve script attached to it. The generated tasks can be run in parallel in a multi-processor environment.

Figure 8-1   Single Proof Partitioned in to Multiple Tasks
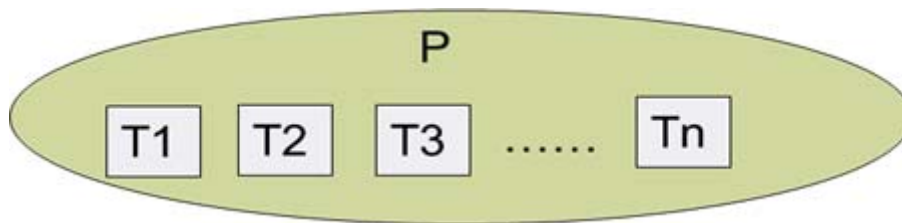
8.2   Assume-guarantee Based on Lemma Partitions

Assume-guarantee reasoning is technique for breaking a given proof into a number of sub-proofs. In each sub-proof intermediate lemmas about the given problem are proven. The intermediate lemmas are then used in later sub-proofs as assumptions to simplify proving other lemmas.

There are multiple ways of doing assume-guarantee reasoning. Hector provides an interface to do assume-guarantee reasoning based on lemma partitions. Suppose the given problem P has a set of assumptions A and a set of lemmas L that needs to be proven. Let `L1,…., L{n}` denote partitions of L such that the each lemma in L is present in at least one partition. The partitions are not required to be disjoint. When assume-guarantee reasoning is enabled Hector creates following sub-proofs to solve P:

```
Proof P1 to prove lemmas in L1 using assumptions in A

…..

Proof P{i} to prove lemmas in L{i} using assumptions in A and L{i-1}

…..

Proof P{n} to prove lemmas in L{n} using assumptions in A and L{n-1}
```

Note that the assumptions and lemmas in each sub-proof depends on the order of lemma partitions L1,…, L{n}.

In order to enable lemma based assume-guarantee reasoning you need to set the following variable to true.

```
set_hector_assume_guarantee true
```

Hector puts each lemma in a new partition. That is, each sub-proof will contain one lemma to be proven, and one additional assumption that the immediate preceding lemma is true. The lemmas are considered in the order in which they are specified by you in `user_assumes_lemmas_procedure`. If lemma partitions are defined as described in the previous section, the assume-guarantee mode will utilize those partitions.

You can set the following variable so that all earlier lemma partitions `L1,  ..,  L{i-1}` are used as assumptions when proving `L{i}`.

```
set_hector_ag_assume_all_prev true
```

The top level proof result is the same when this variable is set to true, but the performance may be improved (or degraded) depending on the specific lemmas and the designs.

You can also finely control the lemmas in each partition and the solve script used for the partition by providing a list of partitions (list of lemma names or patterns) and optionally the name of solve scripts to use for solving lemmas in a given partition. The syntax is:

```
set_hector_ag_partition_list [list partition1 partition2 …]
```

where `partition<i>` is a list which can contain lemma names or patterns (all lemmas matching with the pattern are included in the partition). In addition, `partition<i>` can contain at most one entry of the form `"-script <solve_script_name>"` to specify a solve script to use when solving the lemmas in a partition.

For example:

```
set_hector_ag_partition_list [list \

    [list lemma_1 lemma_2 "-script orch_sat_bb"] \

    [list lemma_3 lemma_4 "-script orch_expensive_solve2"] \

]
```

The partitions supplied by the user are not required to be disjoint or complete. Hector will generate an additional partition to include any lemmas that were not specified by you. Bounds check or other automatic lemmas will usually end up in an additional partition.

8.2.1   Naming Sub-proofs

Suppose you are trying to solve proof <P> with assume-guarantee and have created n lemma partitions. Then the following sub-proofs will be created:

```
<P>_ag1,…,  <P>_ag<n>
```

8.2.2   Proofs and Tasks Created

Figure 12 shows the sub-proofs and tasks that are created when assume -guarantee is enabled during `solveNB P`. The proof P is broken into sub-proofs `P1, …, Pn`. By default each sub-proof has one task in it. The tasks `T1, …, Tn` can be run in parallel in a multi-processor environment. If other options such as multiple solve scripts (section "Partitioning Lemmas in a Given Proof") or lemma partitioning (Section "Assume-guarantee Based on Lemma Partitions") are enabled each sub-proof can potentially have multiple tasks in it.

Figure 8-2   Proofs and Tasks Created
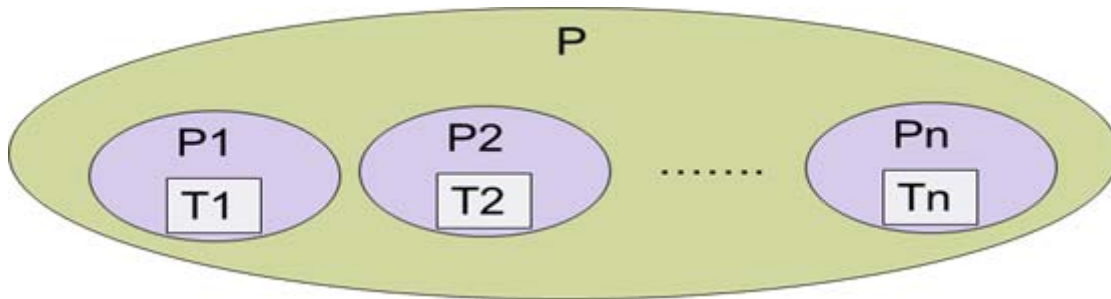
8.2.3   Understanding Proof Results and Debugging

Suppose that you started proof P using the `solveNB P` command and enabled assume-guarantee for solving P. You can use the `listproofs` command to see the status of the lemmas in P. The status of lemmas in P is updated based on the status of the lemmas in sub-proofs created for assume-guarantee reasoning. The status is updated in the following order.

1. If the original problem P has conflicting assumptions/constraints, then the status of all lemmas in P will be *conflc* to indicate contradictory assumptions.

    In this case you need to debug the assumptions/constraints in P.

2. Suppose a lemma L is present in original problem P. If L fails in any of the sub-proofs `P1, …, Pn`, then the status of L is set to *failed* in P.

    In this case you can generate a waveform for L to debug the failure.

3. We examine each proof `P1, …, Pn` in order. Suppose we are examining the proof `P{i}`. We consider two cases.

    a. All lemmas in `P1` to `P{i-1}` have passed. Suppose lemma l is successful in `P{i}`. If the status of l is not already set in P, then the status of l in P is set to *success*. Otherwise, the status of l is left unchanged.

    b. A lemma has failed in one of the proofs from `P1` to `P{i-1}`. In this case no action is performed when examining `P{i}`. This is because one of the earlier failing lemmas is either directly or indirectly used as an assumption in `P{i}`. So the success of a lemma in `P{i}` does not imply the success of that lemma in the original proof P.

8.3   Case Splitting

Case splitting is a technique for breaking a hard formal verification proof into a number of sub-problems that are easier to solve. Each sub-problem (case-split) restricts the search space by adding more assumptions to the original problem. You can do this manually by setting up multiple `user_assumes_lemmas`

procedures, each representing a case split. These separate proofs are then run sequentially or in parallel allowing the user to infer that the original problem is proven.

Alternatively, Hector provides a set of commands that automate case splitting. In addition to easier setup and execution, the automated case splitting methodology adds case completeness checking lemmas and aggregates the case level results to infer the top-level proof objectives.

The case splitting commands in Hector allow user to set up case splits hierarchically. That is, you can split the original problem P into sub-problems `P1, …, Pn`. These sub-problems can be split further into sub-problems. In general, case splitting commands allow you to create a tree of case splits as shown in the Figure 8-3. Each case consists of assumptions and lemmas.

Figure 8-3   Case Split Tree
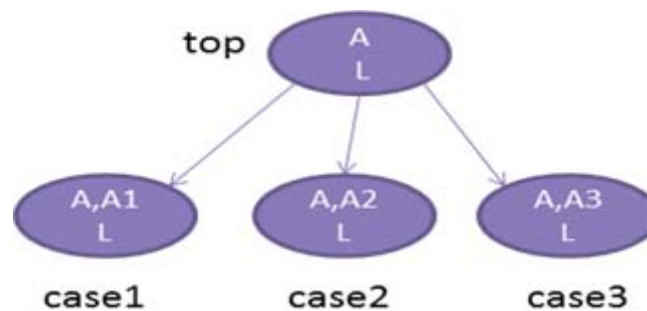
In this figure the A, A1, A2, A3 stand for assumptions and L stands for a lemma to be proven. The original problem *top* consists of proving L using A. We have split *top* into three cases by adding more assumptions to the original problem. For example, case2 consists of proving L using assumptions A and A2. The idea is to prove case1, case2, case3 instead of top. For a general tree of case splits the idea is to prove the leaf level sub-problems and infer the status of interior and root nodes of the case split tree from the leaf level results.

One important consideration in case splitting is to make sure that the case splits are complete. This ensures that if the lemmas pass in each of the sub-proofs, then they pass in the original proof as well. Hector will check the completeness of case splits created you. If a lemma fails in one of the sub-proofs, then it will fail in the original proof as well.

The commands described in the subsequent sections are used to automate case splitting.

8.3.1   Using Case Splits in Proofs

The case splitting commands need to be placed in a TCL procedure. The following Hector variable should be set to specify the name of the TCL procedure containing the case splitting commands.

```
set_hector_case_splitting_procedure <tcl proc>
```

The last case splitting strategy in **<tcl proc>** will be used during the proof. Alternatively, the user can put a `cdCaseSplitStrategy <name>` as the last command in `<tcl proc>` to select one of the case split strategies.

8.3.2   The caseSplitStrategy Command

The `caseSplitStrategy` command is used to provide a name to a collection of case splits. You can create multiple case split strategies. You can specify which case split strategy to use during the proof.

Syntax

```
caseSplitStrategy name [-script sname]
```

Options

- ❖ `-name`: Specifies a name used to refer a collection of case splits.

- ❖ `-script sname:` Optional. It specifies the name of a solve script to use for all cases splits under this case split strategy. Individual case splits can override the solve script to use.

In addition following commands will be used to navigate between multiple case split strategies:

- ❖ `listCaseSplitStrategies`

- ❖ `cdCaseSplitStrategy name`

- ❖ `listCaseSplitStrategy`

The following commands apply to the currently selected case split strategy.

8.3.3   The caseBegin Command

The `caseBegin` command is used to start a case split.

Syntax

```
caseBegin name [-parent pname] [-script sname]
```

Options

- ❖ `name`: Specifies a name for this case split.

- ❖ `-parent name`: Optional. It specifies the name of the parent case split.   If this option is not given the case split is assumed to be a *top* level case split within the current case split strategy

- ❖ `-script sname:` Optional. It specifies the name of a solve script to use for this case split.

8.3.4   The caseAssume Command

The `caseAssume` command is similar to assume command in Hector. It is used to specify an assumption that forms the part of the currently selected case split. It is recommended that you use `caseAssume` only for assumptions that distinguish the different case splits from one another. Use caution when including complex assumptions that are expensive to compute in `caseAssume` statements. For example, if you have a complex multiplier lemma that you want to use as an assumption for selected case splits, placing it in a `caseAssume` statement will cause a completeness lemma to be generated that tries to solve that lemma. For this situation is it much better to create a conditional expression that includes the complex lemma and use a standard `assume` command to add it to the proof.

Syntax

```
caseAssume expr
```

Options

- ❖ `expr`: Is an expression involving specification and/or implementation inputs in different phases.

You can provide a name to the assumption by using the following command:

```
caseAssume name = expr
```

For more discussion on assume command and *phases* please see Section "Assumptions and Lemmas".

8.3.5   The caseEnumerate Command

The `caseEnumerate` command creates a collection of case splits by performing a specified type of enumeration on a given expression. It can be thought of as a macro that is expanded internally into multiple cases. Given an expression E of bitwidth n two common types of enumeration are:

- ❖ FULL: Creates 2n cases corresponding cases corresponding to all possible values of E
- ❖ LEADING_ONE: Creates n cases corresponding to the position of leading one in E

Note that both types of enumeration cover all possible values of the expression, but the `leading_one` enumeration creates far fewer splits. The key advantage of caseEnumerate command is that you do not need to write all case splits manually.

Syntax

```
caseEnumerate name -expr <expr> [-parent pname] [-type tname] \
                              [-script sname]
```

Options

- ❖ `name`: Specifies the name to refer to all the cases in the enumeration `-parent pname` is optional. It specifies the name of the parent case split.   If this option is not given the case split(s) are assumed to be *top* level case splits.

- ❖ `-type tname`: Optional. It specifies the type of enumeration (*full* or *leading1*). By default the type is assumed to be *full*.

- ❖ `-expr expr`: An expression involving specification and/or implementation inputs in different phases.

- ❖ `-script sname:` Optional. It specifies a solve script to use for all cases in the enumeration.

8.3.6   Examples

8.3.6.1   The caseEnumerate Commands

Figure 8-4   Case Split Strategy for Example foo
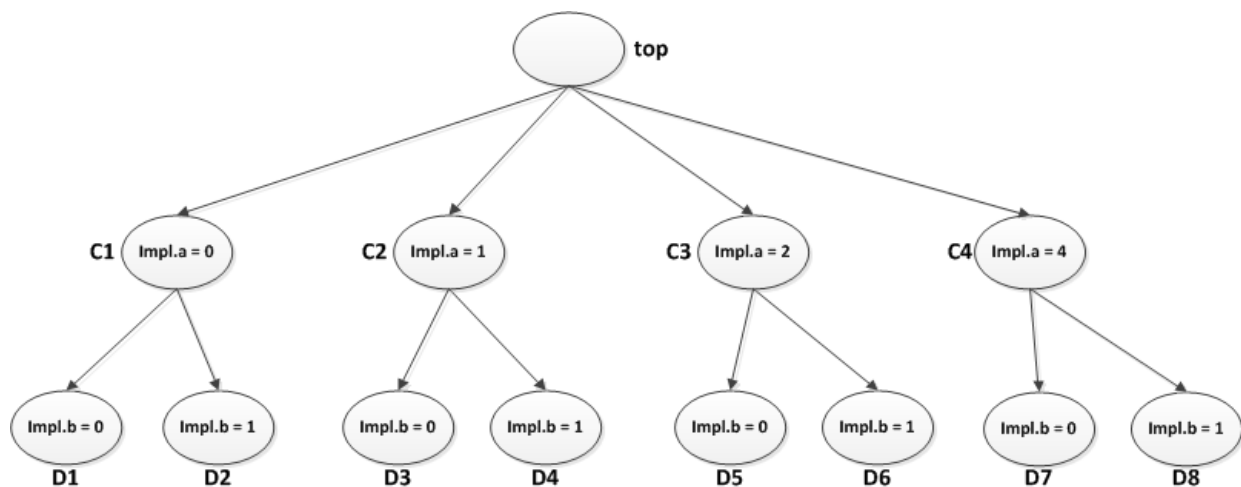
The case split strategy `foo` in Figure 8-4 enumerates all possible values of design inputs `impl.a` and `impl.b`.

```
caseSplitStrategy foo  -script "orch_multipliers"
```

```
caseEnumerate C -expr impl.a(1)

caseEnumerate D -expr impl.b(1) -parent C
```

Suppose the bitwidth of `impl.a` is 2 and `impl.b` is 1. Hector will then generate the case split tree as shown in Figure 8-4 for `foo`.

8.3.6.2   The caseBegin and caseEnumerate Commands

The case split strategy *bar* as shown in Figure 8-5 is a variation on the case splitting shown in previous example. All possible values of `ext.a` are considered. However, further case splitting is done only in the case when `ext.a` is equal to 1.

```
caseSplitStrategy bar

caseBegin C1

  caseAssume (impl.a(1) == 0)

caseBegin C2

  caseAssume (impl.a(1) == 1)

caseBegin C3

  caseAssume (impl.a(1) == 2)

caseBegin C4

  caseAssume (impl.a(1) == 3)

  caseEnumerate D -type FULL -expr "impl.b(1)" -parent C2
```

Assuming the bitwidth of `impl.b impl.b(1)` is 1 Hector will generate the following case split tree for *bar*.

Figure 8-5   Case Splitting Strategy for Example bar
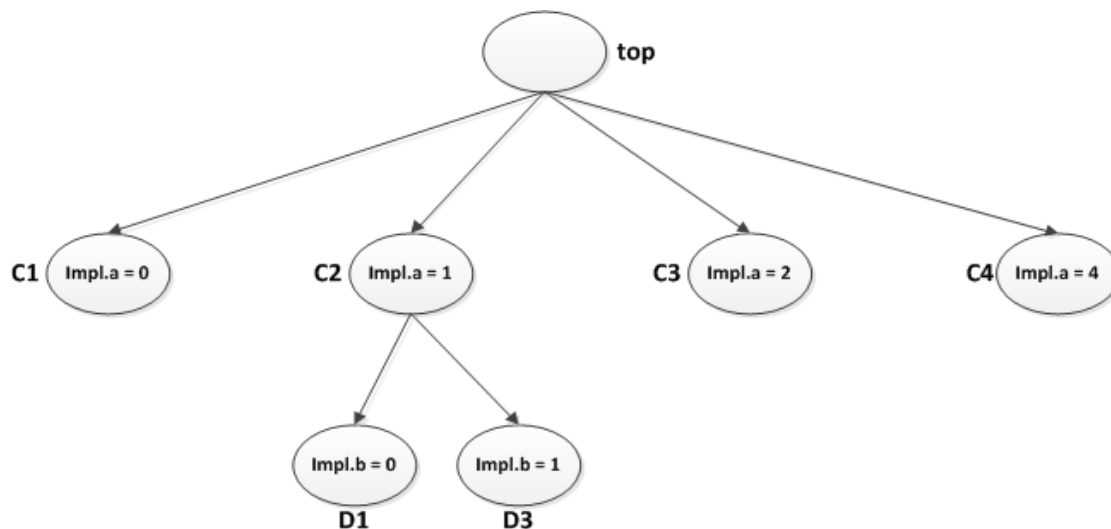
8.3.7   Sub-proofs Created During Case Splitting

When case splitting is used Hector creates a sub-proof for each node in the case splitting tree. For each internal node (not a leaf) in the case splitting tree a sub-proof is created to check the completeness of case split at that node. For each leaf node in the case splitting tree a sub-proof is created where the goal is to prove lemmas in original proof using original assumptions and the assumptions on path from the root of the tree to the leaf node.

8.3.7.1 Naming of Sub-proofs

Suppose you are trying to solve proof <P> with case splitting. For each internal node <IN> in the case split tree a proof named `<P>_<IN>_{completeness}` is created. For each leaf node <LN> in the case split tree a proof named `<P>_<LN>` is created.

Example

Consider the case splitting tree shown in Figure 8-5 (case split strategy *bar*). Suppose the top level problem is to prove lemma L using assumption A (and the proof name is P). Then the following sub-proofs will be generated for each node in the case splitting tree:

Completeness proofs at internal nodes

❖ Proof named `P_top_{completeness}` at node `top`

Using assumption A prove that `impl.a=0 or impl.a=1 or impl.a=2 or impl.a=3`.

❖ Proof named `P_C2_{completeness}` at node C2

Using assumption `A and impl.a=1` prove that `impl.b=0 or impl.b=1`.

Proofs at leaf nodes

❖ Proof named `P_C1` at node C1

Using assumption A and `impl.a=0` prove L.

❖ Proof named `P_D1` at node D1

Using assumption A and `impl.a=1` and `impl.b=0` prove L.

❖ Proof named `P_D2` at node D2

Using assumption A and `impl.a=1` and `impl.b=1` prove L.

❖ Proof named `P_C3` at node C3

Using assumption A and `impl.a=2` prove L.

❖ Proof named `P_C4` at node C4

Using assumption A and `impl.a=3` prove L.

8.3.7.2 Proofs and Tasks Created

Figure 8-6 shows the sub-proofs and tasks that are created when case splitting is enabled during `solveNB P`. The proof P is broken into sub-proofs `P1, …, Pn`. By default each sub-proof has one task in it. The tasks `T1, …, Tn` can be run in parallel in a multi-processor environment. If other options such as multiple solve scripts are enabled each sub-proof can potentially have multiple tasks in it.

Figure 8-6   Proofs and Tasks Created During Case Splitting
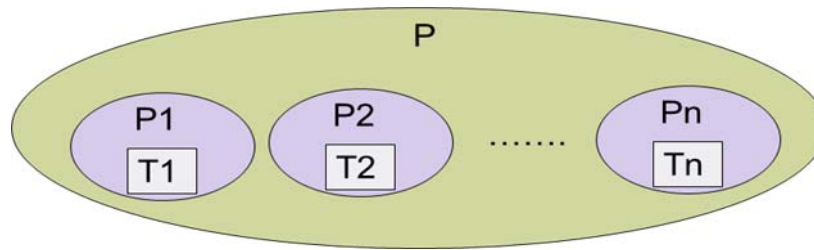
8.3.8   Understanding Proof Results and Debugging

Suppose you started proof P using the `solveNB P` command and enabled case splitting for solving P. You can use the `listproofs` command to see the status of the lemmas in P. The status of lemmas in P is updated based on the status of the lemmas in sub-proofs created for case splitting. The status is updated in the following order:

1. If the original problem P has conflicting assumptions/constraints, then the status of all lemmas in P will be `conflc` to indicate contradictory assumptions.

    In this case you need to debug the assumptions/constraints in P.

2. Suppose (1) does not apply. Suppose a lemma L is present in original problem P. If L fails in any of the leaf case splits, then the status of L is set to *failed* in P.

    In this case you can generate a waveform for L to debug the failure.

3. Suppose (1) does not apply and all completeness checks pass. Suppose a lemma L is present in original problem P. If L passes in each leaf case split proof, then the status of L is set to *success* in P.

4. Suppose (1) does not apply and at least one completeness check fails. This indicates that case splitting is not complete. Suppose a lemma L is present in original problem P. If L does not have a *failed* status in P, then the status of L is set to *inconcl* (inconclusive).

    When case splitting is not complete you can identify which completeness check failed by using `listproofs` command. You can select the failing completeness proof by using the `cdproof <proofname>` command. You can generate the waveform the failing completeness lemma using the `simcex` command.

5. If a completeness lemma does not converge, check to be sure that you did not include an expensive assumption in a `caseAssume` command. Assumptions arising from *assume-guarantee* methods are best included with standard `assume` commands.

☞ Note

Completeness lemmas are solved with the `orch_satonly` script. If another script is required to solve the completeness lemmas, use the `set_completeness_proof_solve_script` command to select it.

The `listproof` and `listlemmas` commands are very useful when examining proof results. The `listproof` command gives a detailed summary of the results of the proof. This includes all of the following:

❖ Number of assumptions defined by the user

❖ Number of lemmas generated by Hector and specified by the user

❖ Number of cutpoints

❖ Listing of the expression for each user assumption

❖ Listing of the expression for each user lemma, along with the proof status of the lemma

- ❖ Summary of lemma results (how many passed, failed, or were inconclusive)
- ❖ Final overall indication of proof status (SUCCESS, FAIL, etc)

You can use the `listlemmas` command to print just the status of each lemma (both Hector generated and user specified).

If you have used case-splitting or assume-guarantee techniques, it results in a hierarchy of proofs. In this situation, there is not just a single proof, but rather a hierarchy of proofs, with the main proof at the top of the hierarchy and the sub-proofs underneath it. To see the detailed results for subproofs, you must first provide `cdproof <[sub]proof-name>`. This establishes the context for subsequent `listproof` and `listlemmas` commands.

If the argument to the `cdproof` command is the top-level proof, then subsequent `listproof` and `listlemmas` commands will apply to the top-level proof. But if the argument to the `cdproof` command is a leaf-level subproof, then the subsequent `listproof` and `listlemmas` commands will give the status for the currently active subproof context as indicated by the most recent `cdproof` command.

In this case, the lemma status printed by these commands is marked with a hash (#) to indicate that the proof or lemma status being printed pertains to a subproof.

8.3.9   Conflicts in Case Splits

The use of case splitting leads to creation of many sub-proofs. You may notice that some of the sub-proofs have conflicting constraints and all lemmas in that sub-proof have `conflc` status. This situation is normal and does not indicate a user/tool error (assuming you have not over-constrained the top level proof).

For example, suppose you have a two bit signal x and our top level proof P has an assumption that `x != 3`. In order to simplify the proof you may decide to case split on all possible values of x. This can be done by using `caseEnumerate` command. This command leads to creation of four cases (sub-proofs) corresponding to `x = 0, x = 1, x = 2, x = 3`, respectively. The sub-proof corresponding to `x = 3` will contain conflicting constraints because the top level proof already has a constraint `x != 3`. You can see this by typing `listassumes`. They will see that both `x = 3` and `x != 3` are present as assumptions. In this case you do not need to do anything if the top level constraints are correct. Hector case splitting layer will automatically ignore the cases that are not allowed by top level constraints.

You should check the status of top level proof to determine which lemmas passed/failed or are inconclusive. Note that if the top level constraints are contradictory, then the status of all lemmas in all sub-proofs will be conflicting.

8.4   Black-boxing

Black-boxing can be used to simplify a proof by abstracting away implementation details that make the proof difficult or are not required for the proof.

Currently Hector only allows black-boxing for the RTL. When a module instance is black-boxed the outputs of that module instance become free inputs. When a module is black-boxed all instances of that module are black-boxed. You can access the inputs and outputs of the black-boxed module by using their hierarchical names. You can also put assumptions and lemmas on the input and output ports of a black-boxed module.

Black-boxing is a safe over-approximation of the design. That is, if you prove that two designs are equivalent under black-boxing, then they are equivalent without black-boxing as well. However, if a mismatch occurs, it may be a spurious mismatch introduced due to the over-approximation.

8.4.1   Creating Black Boxes in the RTL

The modules to be black-boxed in the RTL must be declared using the `set_blackbox` command. This is shown in the following example:

```
proc compile_impl {} {

    create_design -name impl -top demo -clock clk

    set_blackbox A B C

    vcs demo.v

    compile_design "impl"

}
```

This black-boxes all instances of modules A, B, and C. If you want to black-box a specific module instance, use the full hierarchical name instead, for example, `demo.mult.uA`.

8.4.2   Common Uses

The following examples describe two situations where black-boxing is useful.

8.4.2.1   Example: Ignoring Module

Suppose an RTL design instantiates a module `foo`. If it is expected that the outputs of module `foo` do not impact the proof result, the user can black box module `foo`. During the proof the outputs of module `foo` will be treated as primary inputs (taking all possible values).

Black-boxing `foo` can be useful when `foo` contains complex logic that is not relevant to the proof or if `foo` contains non-synthesizable constructs (for example, while loops) that can create problem during RTL compilation.

8.4.2.2   Example: Abstraction

Suppose an RTL design instantiates a module MULT, where MULT is a gate level implementation of a 32-bit multiplier. You want to check the equivalence of the given RTL design against a C model that uses a "*" operator. In order to simplify the equivalence check you can break the problem in two steps.

1. First abstract the instances of MULT by a "*" operator. This can be done by using black-boxing.

2. Do a separate proof that MULT module implements a "*" operator.

Suppose `impl.A.mul1` is an instance of MULT with inputs `impl.A.mul1.x` and `impl.A.mul1.y` and an output `impl.A.mul1.prod`. In order to black box `impl.A.mul1` we need to give the option `-blackbox=impl.A.mul1` to the `create_design` command. Then we can put an assumption that relates the inputs and the output of the black-boxed module by using a "*" operator. For example,

```
assume absM = (impl.A.mul1.prod(1) == (impl.A.mul1.x(1)*impl.A.mul1.y(1)))
```

The (1) in the assumption denotes the phase. The exact phases will depend on the problem.

8.5   Using Cutpoints

Cutpoints can be used to decompose a hard equivalence proof into smaller sub-problems that are easier to solve. In Hector cutpoints can be used to do assume-guarantee reasoning on the data flow graph (DFG). The basic use model is as follows. First, prove some relationships between the internal signals in the design. Then *cut* these signals and replace them by free inputs. In addition you can safely *assume* the relationships proven earlier on the corresponding free inputs. Next, use the modified design to prove the original proof obligations.

Cutpoints are a safe over-approximation of the design. That is, you can prove two designs are equivalent when employing cutpoints, then they are equivalent without cutpoints as well.

The following steps are involved when using cutpoints in Hector (entire source code for the example can be found in $HECTOR_HOME/tutorial/cutpoint).

8.5.1 Declaring a Cutpoint in the C/C++

The *Hector::cutpoint* directive declares a new cutpoint in the design. You must declare new cutpoints in the C++ code itself.

Example

```
int d = a * (b + a);
Hector::cutpoint("d_cut", d);
```

This declares a new cutpoint named *d_cut* for signal *d* with the given size (number of bytes). The name *d_cut* can be referred to in the later steps of Hector.

8.5.2 Declaring a Cutpoint in the RTL

Cutpoints in the RTL are declared using the `set_cutpoint` command before the design is compiled. This is shown in the following example:

```
proc compile_impl {} {

    create_design -name "impl" -top "demo" -clock clk

    set_cutpoint demo.partial

    vcs demo.v

    compile_design "impl"

}
```

This declares the signal with name `demo.partial` may be cut. The cutpoints are not enabled unless the `cutpoint` command is used in the `user_assumes_lemmas_procedure`.

8.5.3 Generated DFG for each Cutpoint

As mentioned earlier, when you compile your C or RTL design using Hector, a formal model - DFG is produced. In the DFG, the cutpoints show up as multiplexers:

```
<sig> = ite($_cuten_<name>, $_cut_<name>, <name>)
```

where, `<name>` is the cut-signal itself, `$_cut_<name>` is a fresh cut-variable and `$_cuten<name>` is an enable-input to the multiplexer. In regular operation mode (cutpoints disabled), the enable-input is low and `<name>` is connected to `<sig>`. To enable the `cutpoint`, `$_cuten_<name>` is set to high and the cut-variable is connected to `<sig>`. The cut-signal (`<name>`) itself is still in the circuit and can be used for assume-guarantee reasoning.

The `$_cut_<name>` and `$_cuten_<name>` inputs are not external inputs and they do not show up on the input-port list of the design, so they are not considered when it comes to building a testbench around the two designs.

Figure 8-7 below shows the change in the Hector DFG when the cutpoint command is used.

Figure 8-7   Cutpoint Circuit
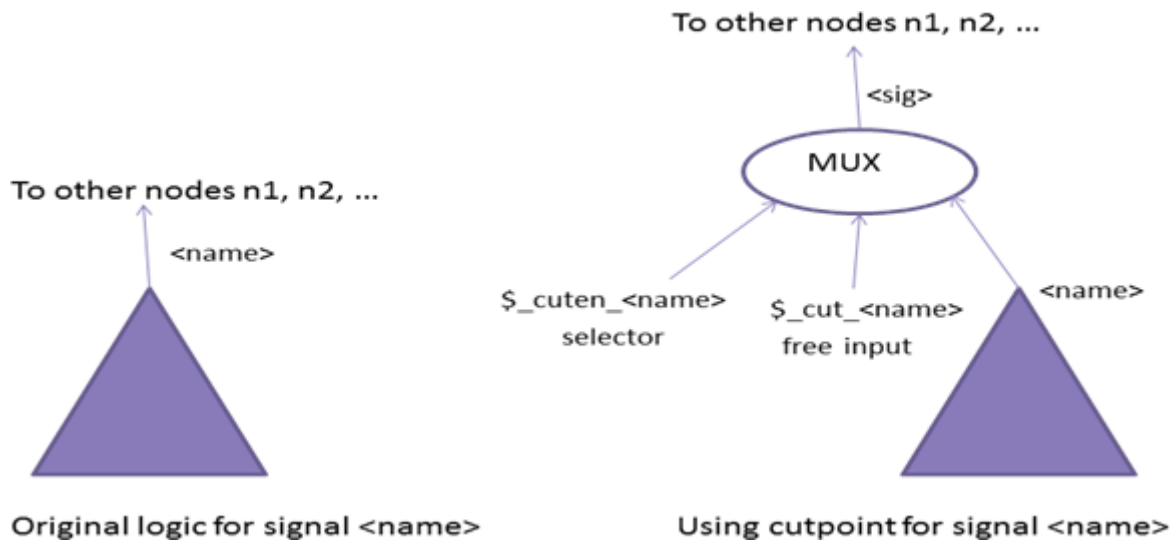
8.5.4   Using Cutpoints in Proof

By default the cutpoints are disabled, that is, if you run a proof it will not take into account any cut-signals. To activate a cutpoint, use the `cutpoint` command in the `user_assumes_lemmas_procedure`. For example:

```
cutpoint spec_cut = spec.d_cut(1);

cutpoint impl_cut = impl.partial(2);
```

The first command enables the cutpoint at the signal d in phase 1 and names it `spec_cut`. The second command enables a cutpoint for `impl.partial` in phase 2. This cutpoints is named `impl_cut`. The cutpoints are created in the unrolled model. From this point on, if you refer to `spec_cut` and `impl_cut` in the assumptions or lemmas, you refer to the fresh cut-variables in the named phases.

If you refer to `spec.d` or `impl.partial`, you refer to original signals before the cut. For example, you can prove some relationship between `spec.d(1)` and `impl.partial(2)` and then use that relationship as an assumption between the cut-variables `spec_cut` and `impl_cut`. To prove that the signals are the same, one can use a lemma, for example,

```
lemma internal = (spec.d(1) == impl.partial(2))
```

To use that as an assumption for later proofs, do

```
assume relation1 = (spec_cut == impl_cut)
```

The cutpoint commands should be placed in the same TCL procedure that is used to supply other assumptions and lemmas, that is, the one specified with the `user_assumes_lemmas_procedure` variable.

8.5.5   Making Cutpoints Conditional

There are some situations where it may be helpful to make cutpoints conditional. For example, proofs utilizing case splits may require cutpoints only for selected cases. One method of achieving that is to create the cutpoint using the methods described in this chapter and adding an assumption that conditionally makes the cutpoint equal to the original cut signal. Using the example mentioned in section 8.5.4, add the following assumption to make it a conditional cutpoint:

```
assume no_impl_cut = (no_cut_condition) -> (impl_cut == impl.partial(2))
```

8.5.6 Troubleshooting Cutpoints

This section describes some of the restrictions and limitations of using cutpoints in Hector.

❖ **Cutpoint on Simple Register Not Working**

Hector performs optimizations such as sub-expression elimination within a block. This may in turn remove some dependencies between registers that exist in the source code.

For example, consider the following code snippet:

```
input   [16:0] a;
input   [16:0] b;
reg     [16:0] multin_a;
reg     [16:0] multin_b;
reg     [31:0] ab;
always @( a or b ) begin
    multin_a = a;
    multin_b = b;
    ab = multin_a * multin_b;
end
```

Hector will optimize the expression for register `ab` to `a*b`, thereby removing the dependencies between `ab` and `multin_a` and `multin_b`. Now cutting the register `multin_a` or `multin_b` will have no effect on the register `ab`.

Workaround

The workaround is to split the above block into two processes by changing the source code.

```
always @( a or b ) begin
    multin_a = a;
    multin_b = b;
end
always @( multin_a or multin_b ) begin
    ab = multin_a * multin_b;
end
```

❖ **Cutpoint on Output is Not Allowed**

You cannot create a cutpoint on an output of a RTL design.

❖ **Cutpoint on Self Dependent Register is Not Allowed**

You cannot use cutpoint on registers whose expression depends on itself. For example, consider the following code snippet:

```
input  clk;
reg    [16:0] count;
always @( posedge clk ) begin
    count <= count + 1;
```

```
      end
```

In the above example the expression for *count* is dependent on itself. You cannot use cutpoints on such registers.

8.6   Using the Hector Data Path Solver Engine

Hector Data Path Solver (HDPS) is a specialized engine for proving sum-of-products lemmas. This engine tries to structurally identify (recognize) a gate-level implementation of these operations in the DFG. If HDPS is able to recognize and solve the sum-of-products lemma, that lemma may be used as an assumption to prove higher-level lemmas. For example, you can use HDPS to prove a lemma for an integer multiplier, and then use that lemma as an assumption to prove a floating-point multiplier.

8.6.1   Writing Lemmas for HDPS

HDPS can only prove lemmas in the following specific formats. Any lemmas that do not match the required syntax or otherwise fail the design structure matching process will be skipped by HDPS and will be solved with the script `orch_custom_bit_operations`. HDPS will report a lemma syntax issue for non-matching lemmas.

```
lemma <name> = <lhs-verilog-expr> == <rhs-verilog-expr> * <rhs-verilog-expr>

lemma <name> = <lhs-verilog-expr> == <rhs-verilog-expr> * <rhs-verilog-expr>
+ <rhs-verilog-expr>

lemma <name> = <lhs-verilog-expr> ==  <rhs-verilog-expr> + <rhs-verilog-expr> + …
```

The `lhs-verilog-expr` can be any Verilog expression. The `rhs-verilog-expr` must be a combination of constants, primary inputs, and cutpoints. HDPS will report a lemma syntax problem if any internal signals are included in the `rhs-verilog-expr`.

Note that you cannot use extract operators on individual product terms, but you can use Verilog width resolution rules to control sub-expression bit widths. For example, the following lemma will trigger a syntax error because of the extract operators to the right of the equality:

```
lemma hdps1 = out(5)[43:0] == (a(1) * b(1))[43:0] + c(1)[43:0]
```

If you rewrite the lemma as follows it will be accepted by HDPS:

```
lemma hdps1 = out(5)[43:0] == (a(1) * b(1)) + c(1)[43:0]
```

8.6.2   Controlling HDPS

Use the following commands to enable HDPS before using the `solve`, `solveNB`, or `gen_proof` commands.

```
set_custom_solve_script  "orch_custom_bit_operations"
set_hector_rew_use_dps_engine  true
```

If you are starting several proofs in sequence, ensure that you disable HDPS before starting any proofs that should not use it using `set_hector_rew_use_dps_engine false`.

Note that the previous recommendation to limit `orch_custom_bit_operations` to lemmas involving 32-bits or less does not apply when this script is used with HDPS.

There are a number of options that may be needed to make HDPS proofs successful. These options are passed to HDPS through the following command:

```
set_hector_rew_dps_options  "<options>"
```

Values that may be included in the `<options>` string are as follows:

❖ **-mode <value>:** A value from [0, 7] with a default of 0. If you do not specify a mode value, HDPS will use a generic solving strategy, but the mode selection may be required to solve the lemma. See instructions for the `run_all_hdps_options` command for an approach to select the best mode.

❖ **-encoding <type>:** The type of multiplier encoding. Options are `radix4booth`, `radix4mbooth`, `radix4boothB`, `gradeschool`, `radix8booth`, `radix8mboothA` and `unknown`. Selection of the appropriate encoding may be required to solve the lemma. See instructions for the `run_all_hdps_options` command for an approach to select the best encoding.

❖ **-no_abstraction:** Disable abstractions during adder recognition

❖ **-def_abstraction**: Enable the default aggressive abstraction technique that HDPS uses.

❖ **-iter_abstraction:** Enable iterative abstraction based on output bit width

👉 Note

Unfortunately, it is not obvious which abstraction works with a particular implementation of the multipliers, therefore. it is recommended to try all the options using the `run_all_hdps_options` command.

There are also multiple internal solve scripts that may be used by the HDPS engine. You can select them using the following command:

```
set_hector_rew_dps_solve_script <hdps_script>
```

The following HDPS internal solve scripts may be selected:

```
__hector_orch_custom_dps (default)
__hector_orch_custom_dps2
orch_abo_sat_bb
orch_expensive_solve1
```

There are two resource control commands for HDPS. To ensure that HDPS does not run too long in the case of a match failure, the recommendation is to keep the general resource limit small (100 – 300).

```
set_resource_limit 100
```

For cases in which resource limits are reached, the HDPS resources may need to be increased to solve the lemma.

```
set_hector_rew_dps_resource_limit 600
```

8.6.3   Recommended Initial HDPS Settings

If you are running HDPS for the first time on a new problem, the following settings are recommended. If no error messages are generated, but the result is inconclusive, you can try other settings. It is recommended that you run HDPS proofs using the `solveNB` command so that you can monitor the progress interactively.

```
set_custom_solve_script  "orch_custom_bit_operations"
set_hector_rew_use_dps_engine  true
     set_hector_rew_dps_solve_script __hector_orch_custom_dps2
set_resource_limit  100
set_hector_rew_dps_resource_limit 600
solveNB hdps_proof
```

8.6.4   Understanding HDPS Phases and Reports

HDPS solves lemmas in several distinct phases. If the result is inconclusive, or if an error message is generated, it is helpful to understand which phase of solving was active. To see the current solver status and sub-status for an HDPS proof, use the `listsolvers` command. This command accepts an optional `taskId` argument and supports several switches for filtering or controlling the verbosity of the output. Use

`listsolvers -help` to see the available options. Figure 8-8 is an example of the `listsolvers` command output.

Consider an example where HDPS is running and has reached the `pp_verified` sub-status. The meaning of the sub-status fields and the related *Reason* messages are described in tables Table 8-1 and Table 8-2.

Figure 8-8   Example of the listsolver Command Output

```
 File  Edit  View  Terminal  Tabs  Help
hector>
hector> listsolvers 1
Task [Id: 1 | Status: running | Result: - | Solved: 0% | Script: orch_custom_bit_operations]
    Proof [Name: ts_lemma_2_hdps]
    Lemma [Name: ts_lemma_2 | Status: running]
    HDPS:
        Options:     "-mode 5 -encoding radix4booth -iter_abstraction"
        Script:      __hector_orch_custom_dps2
        RR:          false
        Resource:    3600
        Start Time:  Mon Feb  8 15:07:29 2016
        Update Time: 5 secs
        Status:      running
        SubStatus:   pp_verified
        Reason:
            encoding - radix4booth
            termnum - 4

Total 1 entry listed.
1
hector>
```

The phase of the solve process is indicated by one of the following solver sub-status values:

Table 8-1   Solver Sub-status Values

Substatus	Interpretation
Started	The HDPS solver has started
IO_verified	The inputs and outputs of the design have been correlated with the lemma arguments
Pp_term_verified	The partial product of one of the terms has been verified. A term is sub-expression of the target lemma. Terms may be separated by add operators in the sum-of-products form. This sub-status may be repeated as multiple terms are analyzed.The reason field will indicate the term number and matched encoding.
Pp_verified	All partial products have been verified.
Cpa_partial_verified	Lemma verification is partially completed. The reason field will indicate the output bit width that has been verified. For more details, see section "Using Assume-Guarantee with Iterative Abstraction".
Cpa_verified	HDPS has successfully verified the lemma.

If a problem is encountered during HDPS solving, additional sub-status values may be reported from the following list:

Table 8-2   Additional Sub-status Values
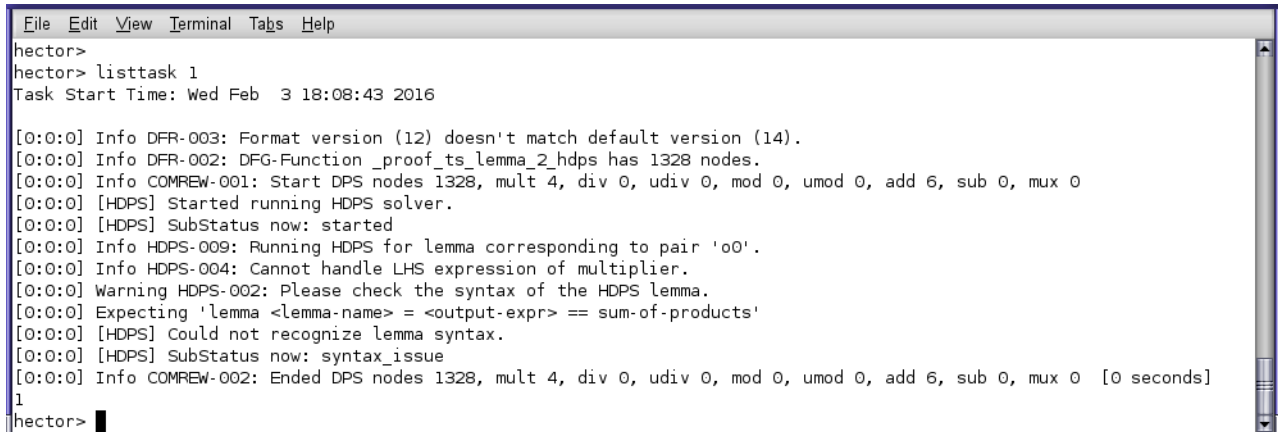
Failure Substatus	Interpretation
Syntax_issue	The lemma syntax violates one of the restrictions.
Io_mismatch	The support of the output signals does not match the support of the sum-of-products lemma. This usually indicates an error in the lemma.
Size_mismatch	The number of output bits to be checked is less than the maximum number of input bits in the sum-of-products expression.
Support_mismatch	The support of the output signals does not match the support of the sum-of-products lemma. The Reason field will list the mismatching inputs. For more information, see section "Handling Support Mismatches"
Dfgnode_mismatch	Cannot find the DFG node for a port. See the *Reason* field for the port name.
Unary_minus_not_supported	There is a unary minus operator in the sum-of-products expression. This can often be corrected by re-arranging the lemma expression.
Pp_not_found	Unable to match the partial products in the lemma to the design structures. The *Reason* field will indicate the specific bits involved. Rewriting the lemma or trying other encodings may help.
Cpa_partial_fail	The original abstraction attempt failed and an iterative abstraction technique is being tried. The *Reason* field will provide the proven output width.
Cpa_fail	HDPS could not solve the lemma. The *Reason* field will indicate the cause. If abstraction failed, try other modes and options. If the resource limit was reached, try increasing the resource limit or changing to the `orch_expensive_solve1` script.

In these cases, the main HDPS status will be reported as *inconclusive*. Note that even when HDPS reports inconclusive results, it still attempts to check the lemma with the `orch_custom_bit_operations` script.

This is helpful in detecting counter-examples for basic errors in the lemma or design. You can control the amount of effort that will be devoted to this solver by using the `set_resource_limit` command.

You can also monitor a history of messages generated by the HDPS solver using the `listtask` command. The following example shows typical messages in the case where the lemma was not written in a form acceptable to HDPS:

Figure 8-9   Lemma not Acceptable to HDPS

```
File  Edit  View  Terminal  Tabs  Help
hector>
hector> listtask 1
Task Start Time: Wed Feb  3 18:08:43 2016

[0:0:0] Info DFR-003: Format version (12) doesn't match default version (14).
[0:0:0] Info DFR-002: DFG-Function _proof_ts_lemma_2_hdps has 1328 nodes.
[0:0:0] Info COMREW-001: Start DPS nodes 1328, mult 4, div 0, udiv 0, mod 0, umod 0, add 6, sub 0, mux 0
[0:0:0] [HDPS] Started running HDPS solver.
[0:0:0] [HDPS] SubStatus now: started
[0:0:0] Info HDPS-009: Running HDPS for lemma corresponding to pair 'o0'.
[0:0:0] Info HDPS-004: Cannot handle LHS expression of multiplier.
[0:0:0] Warning HDPS-002: Please check the syntax of the HDPS lemma.
[0:0:0] Expecting 'lemma <lemma-name> = <output-expr> == sum-of-products'
[0:0:0] [HDPS] Could not recognize lemma syntax.
[0:0:0] [HDPS] SubStatus now: syntax_issue
[0:0:0] Info COMREW-002: Ended DPS nodes 1328, mult 4, div 0, udiv 0, mod 0, umod 0, add 6, sub 0, mux 0  [0 seconds]
1
hector> 
```

8.6.5   Finding the Best HDPS Modes and Options

It may be difficult to predict which combination of HDPS modes and options will solve a lemma. For problems that do not solve with the recommended defaults, it is best to try a selection of different modes in parallel using the `run_all_hdps_options` command on a multi-processor grid. This command will create a number of tasks all running the same HDPS proof with different options. If the `proofwait` command is invoked after the tasks are started, the first task to complete will kill any remaining tasks that are running or waiting to run.

The number of tasks created will be the product of the mode and switch choices specified by the command options. Using the default options will create forty eight parallel tasks. This number may be increased, or decreased using the following command options. Before using the `run_all_hdps_options` command, be sure to consider the machine and license resources that may be used by the generated tasks.

The switches can be used to control the range of options that will be generated are as follows:

```
hector> run_all_hdps_options -help
Usage: run_all_hdps_options # Run all hdps options.
        [-modes list-of-modes] (List of modes for HDPS (Values: i >= 0 && i < 8).)
        [-scripts list-of-scriptname] (Name of the HDPS script.)
        [-resource integer] (Resource limit for the HDPS script.)
        [-abstypes list-of-abs-types](List of abstraction types (Values: ef_abstraction,
                                      iter_abstraction, no_abstraction))
        [-rrtypes list-of-rr-types](List of redundancy removal types (Values: false,
                                                                       true))
        -encodings list-of-encodings (List of encodings.
                                     (Values: auto radix4 radix8 radeschool radix4booth
                                      radix4mbooth radix4boothB radix8booth
                                      radix8boothA))
         <name> (Name of the proof.)
```

The required arguments are the proof name and the list of encodings. If you specify the encodings `radix4` or `radix8`, tasks are generated to cover all the encodings related to these choices. For example, selecting

radix4 will create tasks to cover the options radix4booth, radix4mbooth, and radix4boothB. You can see the complete list of supported encoding is provided in the section "Controlling HDPS".

❖ The –modes switch allows selection of specific mode values between 0 and 7. For example, to select three modes use -modes {1 4 5}. If this switch is not provided, all eight modes will be tried.

❖ The –scripts switch provides a way to select alternate HDPS solve scripts. If no selection is made, the script __hector_orch_custom_dps2 will be selected. See section "Controlling HDPS" for a complete list of recommended scripts.

❖ The –resource switch provides a way to control the resource limit for all of the generated tasks. If you do not provide this switch, the value previously specified by the set_resource_limit command will be used for all generated tasks.

❖ The –abstypes switch allows selection from the abstraction options def_abstration, iter_abstraction, and no_abstraction. If this switch is not provided, tasks will be generated to try all three options.

❖ The –rrtypes switch allows selection of options true or false for redundancy removal. If no selection is made, tasks to try both options will be generated.

☞ Note
Often, HDPS may not be able to find matching points due to redundancy in the model. On the other hand, applying redundancy can drastically change the structure of the multiplier and lead to convergence issues. Therefore, it is recommended to try both the options using the run_all_hdps_options command.

Once the best set of options are determined by parallel search, the usual method is to create a TCL proc that explicitly sets those options and use it for subsequent Hector runs of the same proof. This eliminates unnecessary machine and license use.

8.6.6   Handling Support Mismatches

When the output of the listsolvers or listtask command reports a sub-status of support_mismatch, you can try the following alternatives to resolve the issue.

1. Try enabling redundancy removal:

```
set_hector_rew_dps_rr true
```

2. Try removing control nodes in the DFG:

```
set_hector_ind_ignore_control_nodes true
```

3. Review the extra output support signals reported. If the number of bits is small, you can case-split on those inputs and prove each distinct combination independently. Look for messages of the following form:

```
[HDPS] Error: Extra output support:
<input_A>[<X>:<Y>](<phase>)_<Z>
<input_A>[<X>:<Y>](<phase>)_<Z1>
. . .
```

Now create a case splitting procedure in the Hector command script:

```
proc my_casesplit {} {
   caseSplitStrategy extraSupport
   caseEnumerate s1 -expr "input_A(<phase>)[<Y>+<Z>]"
   caseEnumerate s2 -expr "input_A(<phase>)[<Y>+<Z1>]"
. . .
}
```

8.6.7　Using Assume-Guarantee with Iterative Abstraction

When HDPS is run with the option `set_hector_rew_dps_options "-iter_abstraction"`, it may find a proof for only some bits of the target lemma. In this case you may be able to create an assumption from the proven bits and use that assumption in an assume-guarantee proof. To find out how many bits are proven, check the output of the `listtask <n>` command where `n` is the `taskID` of the HDPS proof. The report may contain the following lines:

```
[HDPS] Partially CPA check successful (width: <W1>).
[HDPS] Partially CPA check successful (width: <W2>).
[HDPS] Partially CPA check successful (width: <W3>).
[HDPS] Partially CPA check successful (width: <W4>).
```

The max of the above widths is the maximum bit width proven. At this point, you can rewrite the HDPS lemma up to the maximum width and prove it. This can in-turn help prove the actual lemma using the partial-width lemma as an assumption.

For example, given the original 44 bit wide lemma:

```
lemma hdps1 = out(5)[43:0] == (a(1)[23:0] * b(1)[23:0])
```

If HDPS was only able to prove 40 bits, then you can prove the following lemma:

```
lemma hdps2 = out(5)[39:0] == \
(a(1)[23:0] * b(1)[23:0])+ 40'b0
```

You can then use an assume-guarantee technique to prove the original lemma.

8.6.8　Frequently Asked Questions about HDPS

1. What if the inputs to the multiplier are not primary inputs?

   In this case, you must manually add cutpoints for the inputs of the multiplier.

2. What if HDPS failed to recognize the sum-of-product implementation?

   In this case, Hector does not learn anything from the HDPS engine and you have to use other technologies in Hector (like case splitting) to solve it.

3. What is the cost of running HDPS engine?

   The cost of running the HDPS engine can be expensive. It is recommended not to keep it on by default for all lemmas. You can disable the HDPS engine by setting the following option to false.

   ```
   set_hector_rew_use_dps_engine  false
   ```

4. Why is it taking a long time before I get an *inconclusive* result?

   If you are trying to select the best HDPS modes and options, start with a small resource limit to quickly eliminate inappropriate choices:

   ```
   set_resource_limit  100
   ```

   Once you find the best matching options, you may need to increase the HDPS resource limit to find a proof. For example:

   ```
   set_hector_rew_dps_resource_limit  3600
   ```

8.7　Division and Square Root Verification in Hector

This section presents a methodology for verifying that a given design implements division or square root. In section "Definition of Division" we define division. In section "Standard Restoring Division Algorithm" we provide a high level description of a standard restoring division algorithm (SRDA). Conditions that imply that a candidate SRDA implements division are presented. The methodology for verifying standard

restoring division algorithms in Hector is described in section "Verifying a Candidate SRDA in Hector". Examples illustrating this methodology are presented in section "Division Verification Examples". Most practical hardware implementations of division make use of non-restoring division algorithms. Verification of non-restoring division algorithms is discussed in section "Non-restoring Division Algorithm Verification". In section "Checking Equivalence of Designs with Dividers" we provide guidelines on how to check equivalence of two designs containing divider implementations.

In section "Definition of Integer Square Root" we define square root. We present the conditions that imply a given design does square root in section "Standard Restoring Square Root Algorithm". Hector commands for proving a design performs square root is given in section "Verifying a Candidate SRSA in Hector". In section "Square Root Verification Examples" an example is presented. In section "Adding Assumptions after Proving a Design does Square Root" we show how to assume that a given design does square root (after proving that it does square root).

We assume radix-2 in this section.

8.7.1   Definition of Division

Consider an algorithm T that takes two unsigned bitvector inputs dividend X and divisor D and produces two unsigned bitvector outputs Q and R. Suppose the bitwidth of D,Q,R is k and the bitwidth of X is 2k bits.

$$X = x_{2k-1} \ldots x_{k-1} \ldots x_0$$

$$D = d_{k-1} \ldots d_0$$

$$Q = q_{k-1} \ldots q_0$$

$$R = r_{k-1} \ldots r_0$$

The algorithm T implements unsigned bitvector division if the following conditions hold.

```
X = Q × D + R           (1)

R < D                   (2)
```

If the above conditions hold then Q is the quotient and R is the remainder.

8.7.2   Standard Restoring Division Algorithm

A standard restoring division algorithm (SRDA) has a for loop with k iterations. In each iteration one bit of quotient is computed starting from most significant bit (MSB) of the quotient. That is, iteration number i produces quotient bit q_{k-i} where $1 \le i \le k$ and $q_{k-i} \in \{0,1\}$. Denote the intermediate quotient and remainder in iteration i as Q_i and R_i. The relationships between intermediate quotient and individual quotient bits is as follows:

```
Q₁  =   q_{k-1}  0   0   …   0
Q₂  =   q_{k-1}  q_{k-2}  0  …  0
……
Q_k  =   q_{k-1}  q_{k-2}  q_{k-3}  …  q_0
```

The following conditions are sufficient to show that a candidate SRDA algorithm T implements division (define $Q_0=0$ and $R_0=X$):

```
X = Qᵢ* D+Rᵢ  (0 <= i <= k)        (3)

Rᵢ < (D * 2^{k-i}) (0 <= i <= k)    (4)
```

Note that when $i = k$ the above conditions imply that Q_k is the quotient and R_k is the remainder as per equations (1) and (2). Also note that when $i = 0$ equation (5) corresponds to the following condition (as $R_0 = X$):

$$X < (D * 2^k)$$

This condition is also referred to as the overflow prevention condition. It ensures that the final quotient will fit in k bits.

Instead of checking equation (3) for $0 <= i <= k$ you can check the following equation for $0 <= i < k$.

$$R_i = (q_{k-i-1} * 2^{k-i-1}) * D + R_{i+1} \quad (0 <= i < k) \qquad (5)$$

8.7.3   Verifying a Candidate SRDA in Hector

Suppose we are given a candidate SRDA algorithm T. In order to check if T implements division requires you to provide following information:

- ❖ Dividend X. The user provides a TCL procedure that returns X. Suppose this TCL procedure is called `<getDividend>`.

- ❖ Divisor D. The user provides a TCL procedure that returns D. Suppose this TCL procedure is called `<getDivisor>`.

- ❖ Partial quotients $Q_0, ..., Q_k$. The user provides a TCL procedure that takes one argument i and returns Q_i for each $0 <= i <= k$. Suppose this TCL procedure is called `<getQuotient>`.

- ❖ Partial remainders $R_0, ..., R_k$. The user provides a TCL procedure that takes one argument i and returns R_i. Suppose this TCL procedure is called `<getRemainder>`.

- ❖ Assumptions under which the algorithm implements division. The user provides a TCL procedure that contains input assumptions specified using `assume` commands. Suppose this TCL procedure is called `<inputAssumptions>`.

- ❖ Number of iterations k in the division algorithm. This is the same number as the number of quotient bits.

The above information about dividend, divisor, partial quotients, and partial remainders should be derived using internal signals of the design. The internal signals in a C/C++ design can be made visible by using `Hector::show` commands. The internal signals in a RTL design can be accessed directly by using hierarchical names.

Using the above information you can create a proof to check if T implements division. This is done by issuing the following command:

```
solveNB_division k <getDividend> <getDivisor> <getQuotient> <getRemainder>
                    <inputAssumptions> <identifier>
```

where `<identifier>` is the name that will be given to the generated proof. This command and other special commands required for division verification are not included in the default Hector shell, but can be added by including this in the `command_script` TCL,

```
source $env(HECTOR_HOME)/local/divchecks.tcl.
```

The `solveNB_division` command starts a proof containing following lemmas:

- ❖ One lemma for each $0 <= i < k$ corresponding to equation   (4)

  This gives rise to k lemmas. One additional lemma checks that $R_0 = X$.

- ❖ One lemma for each $0 <= i <= k$ corresponding to equation (5). This gives rise to `k+1` lemmas.

❖   One lemma for each $0 < i <= k$ that checks the definition of Q_i in terms of Q_{i-1} and quotient bit q_{k-i}. This gives rise to k lemmas. One additional lemma checks that $Q_0=0$.

If all lemmas in the generated proof are successful, then T implements division under the given assumptions. More precisely, under the input assumptions provided in `<inputAssumptions>`, dividend `<getDividend>`, divisor `<getDivisor>`:

> [getQuotient k] (Q_k) is the quotient.

> [getRemainder k] (R_k) is the remainder.

8.7.3.1   Multiple Proofs

The proof generated by `solveNB_division` command can be broken down into sub-proofs for quicker convergence on hard problems. The easiest way to do this is to use MP layer feature for assume-guarantee reasoning. For example, you can set following options:

> set_hector_assume_guarantee true

> set_hector_ag_partition_size 3

With the above options the `solveNB_division` command generates `k+1` sub-proofs where each sub-proof contains three lemmas to be proven. The lemma in proof j checks the correctness of iteration j of the division algorithm. In addition, we assume the correctness of the iteration j-1 when proving iteration j for `j >= 1`.

If all lemmas in all generated sub-proofs are successful, then T implements division under the given assumptions. The advantage of using the above command is that one can run generated proofs in parallel in a multi-processor (MP) environment and obtain significant speedups and/or obtain convergence in cases where a single proof containing lemmas for all iterations does not converge.

8.7.3.2   Proving a Subset of Iterations

The `solveNB_division` command can optionally take two arguments `<lowIter>` and `<highIter>` (as shown in the following example). When these options are used Hector will only check the correctness a proof for only iterations `i` between `<lowIter>` and `<highIter>`. This is useful when the user only wants to run a subset for all iterations (for example, during initial phase of debugging the setup for division proof).

> solveNB_division k <getDividend> <getDivisor> <getQuotient> <getRemainder>
>   <inputAssumptions> <identifier> [<lowIter>] [<highIter>]

The use of options `<lowIter>` and `<highIter>` is discouraged because it is your responsibility to make sure all iterations are proven in order to conclude that the given design implements division.

8.7.3.3   Variable Number of Iterations

The previous sections assumed that the given restoring algorithm has exactly k iterations. This is not always the case. It is possible that a given restoring algorithm computes the result in variable number of iterations. In such cases the you still need to provide k partition quotients/remainders in order to use the above procedure. This can be done by adding additional code in the Hector TCL file or by adding additional logic in the design itself for verification purpose.

8.7.4   Division Verification Examples

8.7.4.1   32-bit Unsigned Integer Division

The div1 function shown below takes two 32-bit unsigned integer inputs a and b and returns (by reference) two unsigned integer outputs *quo* and *rem*. Thes goal is to show that *quo* is the quotient and *rem* is the remainder when a is divided by b.

```
void div1(uint32 a, uint32 b, uint32 &quo, uint32 &rem)
{
  uint64 prem = a;
  uint32 pquo = 0;

  Hector::show("a",  a);
  Hector::show("b",  b);

  Hector::show("pquo",  pquo);  // creates 'pquo' handle in DFG
  Hector::show("pquo",  pquo);  // creates 'pquo0' handle in DFG
  Hector::show("prem",  prem);  // creates 'prem' handle in DFG
  Hector::show("prem",  prem);  // creates 'prem0' handle in DFG

  for (int i = 0; i < 32; i++)
    {
      int shift = 31 - i ;
      uint64 sdiv = ((uint64)b) << shift;

      pquo *= 2;
      if (prem >= sdiv)
        {
          pquo += 1;
          prem -= sdiv;
        }
      Hector::show("pquo",  pquo);  // creates 'quo1',…, 'quo32'handles
      Hector::show("prem",  prem);  // create 'rem1',…, 'rem32' handles
    }

  rem = prem;
  quo = pquo;
}
```

In order to verify div1 we have inserted `Hector::show` commands in the source code to make internal design signals visible. These signals can be used to write the TCL procedures that return dividend, divisor, partial quotients, and partial remainders. The TCL procedures are given below.

```
# k is 32
proc div1_dividend {} { return "{32'b0,spec.a(1)}" }
```

```
proc div1_divisor {} { return "spec.b(1)" }


# Bitwidth of quotient is k.

proc div1_quotient {i} {

    set Nmi [expr 32-$i]

    return "(spec.pquo${i}(1) << $Nmi)"

}


proc div1_remainder {i} { return "(spec.prem${i}(1))" }


# The algorithm implements division only when divisor is non-zero.

# This needs to be given as an assumption.

proc div1_assumptions {} {

    assume no_divzero = spec.b(1)  != 0

}
```

To start the proof issue the following command:

```
solveNB_division 32  div1_dividend div1_divisor div1_quotient div1_remainder
                     div1_assumptions div1_proof
```

The success of the above proof only establishes that `spec.pquo64(1)` is the quotient and `spec.prem64(1)` is the remainder when `spec.a(1)` is divided by `spec.b(1)`. In order to complete the proof one also needs to prove following lemmas separately:

```
lemma qcheck = spec.pquo64(1)  == spec.quo(1)

lemma rcheck = spec.prem64(1)  == spec.rem(1)
```

These lemmas show that the outputs returned by `div1` are actually quotient and remainder.

8.7.4.2   128-bit Unsigned Integer Division

The div2 function shown below takes two 64-bit unsigned integer inputs a and b and returns two unsigned integer outputs *quo* and *rem*. The goal is to show that *quo* is the quotient and *rem* is the remainder when $a*2^{63}$ is divided by b under certain assumptions mentioned below.

```
void div2(uint64 a, uint64 b, uint64 &quo, uint64 &rem)

{

    Hector::show("a",  a);

    Hector::show("b",  b);


    uint64 a_significand = a;

    uint64 b_significand = b;

    uint64 significand  = 0;
```

```
Hector::show("significand",    significand);
Hector::show("significand",    significand);
Hector::show("a_significand",  a_significand);
Hector::show("a_significand",  a_significand);


for (int i = 0; i < 64; i++)
{
   significand *= 2;
   if (a_significand >= b_significand)
     {
           significand += 1;
           a_significand -= b_significand;
      }
   a_significand *= 2;

   Hector::show("a_significand",  a_significand);
   Hector::show("significand",    significand);
}
rem = a_significand;
quo = significand;

}
```

In order to verify div2 we have inserted `Hector::show` commands in the source code to make internal design signals visible. The TCL procedures needed for verification and input assumptions are given below.

```
proc div2_dividend {} { return "{1'b0,spec.a(1),63'h0}" }
proc div2_divisor {} { return "spec.b(1)" }
proc div2_quotient {i} {
    set Nmi [expr 64-$i]
    return "(spec.significand${i}(1) << $Nmi)"
}
proc div2_remainder {i} {
    global N
    set Nmi [expr 63-$i]
    if {$Nmi >= 0} {
        return "({64'h0,spec.a_significand${i}(1)} << $Nmi)"
    } else {
        set Nmi [expr -$Nmi]
        return "({64'h0,spec.a_significand${i}(1)} >> $Nmi)"
```

```
        }
    }
    proc div2_assumptions {} {
        assume a_msbs  = spec.a(1)[63:62] == 2'b01
    assume b_msbs  = spec.b(1)[63:62] == 2'b01
    }
```

To start the proof we issue the following command:

```
    solveNB_division 64 \
        div2_dividend \
        div2_divisor \
        div2_quotient \
        div2_remainder \
        div2_assumptions \
        div2_proof
```

8.7.5   Non-restoring Division Algorithm Verification

Most hardware implementations of dividers make use of non-restoring division algorithms. See the standard texts on computer arithmetic to understand how non-restoring division algorithms work. Some key differences between restoring and non-restoring division algorithms given as follows:

❖   In restoring unsigned division algorithms the partial remainder is always non-negative. In a non-restoring algorithm the partial remainders can become negative.

❖   The quotient digit set for a non-restoring division algorithm can be redundant. For example, a radix-2 non-restoring division algorithm may produce a quotient digit from the set `{-1,0,1}` in every iteration.

An iterative non-restoring division algorithm also maintains a partial quotient NQ_i and a partial remainder NR_i in every iteration. At the end of last iteration a non-restoring division algorithm produces the same result as a restoring division algorithm. This is typically done by using a conversion routine that produces Q_k from NQ_k and R_k from RQ_k.

In order to verify a non-restoring division algorithm in Hector the user needs to perform following steps.

1.   Use NQ_i and NR_i signals in the design obtain Q_i and R_i for all `0 <= i <= k`. Note that this can be done at TCL level or by adding additional logic in the design itself. The core functionality of the non-restoring division implementation is left unchanged.

2.   Apply the techniques previously described to prove that the given algorithm implements division. That is, Q_k and R_k are the quotient and remainder, respectively, for the given dividend, divisor, and input assumptions.

3.   Prove that the final outputs (quotient/remainder) from the non-restoring division algorithm match Q_k and R_k, respectively.

8.7.6   Proving Signed Integer Division

Given dividend X and divisor D the result of signed integer division is quotient Q and remainder R that satisfies following equations:

```
X=Q×D+R               (6)
|R|<|D|               (7)
sign(R)= sign(D)      (8)
```

In many cases it is easier to prove that unsigned integer division between numbers. The above equations can be written as follows so that $|Q|$ is the quotient and $|R|$ is the remainder when $|X|$ is divided by $|D|$.

```
|X|=|Q|×|D|+|R|                      (9)
|R|<|D|                              (10)
sign(R)= sign(D)                     (11)
sign(Q)= sign(D)  xor sign(X)        (12)
```

8.7.6.1   Verifying Signed Integer Division in Hector

In order to verify signed integer division we require the user to provide following information:

❖ Dividend X. The user provides a TCL procedure that returns X. Suppose this TCL procedure is called `<getDividend>`.

❖ Divisor D. The user provides a TCL procedure that returns D. Suppose this TCL procedure is called `<getDivisor>`.

❖ Quotient Q. The user provides a TCL procedure that returns Q. Suppose this TCL procedure is called `<getQuotient>`.

❖ Remainder R. The user provides a TCL procedure that returns R. Suppose this TCL procedure is called `<getRemainder>`.

❖ Assumptions under which the algorithm implements division. The user provides a TCL procedure that contains input assumptions specified using assume commands. Suppose this TCL procedure is called `<inputAssumptions>`.

Using the above information you can create a proof to check if T implements division. This is done by issuing the following command:

```
solveNB_signed_division <getDividend> <getDivisor> <getQuotient> <getRemainder>
                        <inputAssumptions> <identifier>
```

where `<identifier>` is the name that will be given to the generated proof.

The `solveNB_signed_division` command starts a proof containing following lemmas:

❖ Lemma to check sign of remainder. The condition when remainder is zero is treated as an output don't care. This is because in two's complement representation there is no difference between positive zero and negative zero.

❖ Lemma to check sign of the quotient. The condition when quotient is zero is treated as an output don't care.

❖ Lemma to check that $|Q|$ is the quotient when $|X|$ is divided by $|D|$.

❖ Lemma to check that $|R|$ is the remainder when $|X|$ is divided by $|D|$.

If all lemmas in the generated proof are successful, then T implements signed division under the given assumptions.

8.7.7   Checking Equivalence of Designs with Dividers

Suppose we are given two designs *spec* and *impl* for equivalence checking and that both designs contain two divider implementations. Suppose the divider module in spec is *spec.D1* and the divider module in *impl* is *impl.D2*. The equivalence proof for *spec* and *impl* may take a long time or may not even converge if the algorithms used in *spec.D1* and *impl.D2* are different. For example, *spec.D1* might use a restoring division algorithm while *impl.D2* might use a non-restoring division algorithm. In order to obtain convergence we recommend the following strategy.

❖   Prove that *spec.D1* implements division by using techniques as previously described.

❖   Prove that *impl.D2* implements division by using as previously described.

❖   Abstract *spec.D1* by a word-level division '/' operator. This can be done by using an assumption relating the outputs of *spec.D1* with the inputs of *spec.D1*. One can also black box *spec.D1*.

❖   Abstract *impl.D2* by a word-level division '/' operator. This can be done by using an assumption relating the outputs of *impl.D2* with the inputs of *impl.D2*. One can also black box *impl.D2*.

❖   Check the equivalence of *spec* and *impl* using the abstracted versions of *spec.D1* and *impl.D2*.

Note that you needs to be careful obtain the input assumptions under which *spec.D1* and *impl.D2* implement division. These assumptions need to be satisfied when performing the abstraction (replacing an implementation by word-level '/' operator).

In order to help you with steps (3) and (4) Hector provides a TCL procedure that allows user to add assumptions expressing that a given design does division. This is described in the next section.

8.7.7.1   Adding Assumptions after Proving a Design does Division

Suppose you have used `solveNB_division` to show that a given design T does division. That is, all lemmas in the following call to `solveNB_division` are successful.

```
solveNB_division k <getDividend> <getDivisor> <getQuotient> <getRemainder>
                   <inputAssumptions> <identifier>
```

You can call the TCL procedure `solveNB_division_assumptions` to add the result of the above verification as assumptions in another proof.

```
solveNB_division_assumptions k <getDividend> <getDivisor> <getQuotient>
                                 <getRemainder> <inputAssumptions> <identifier>
```

The arguments to `solveNB_division_assumptions` are exactly the same as `solveNB_division`. This TCL procedure puts two assumptions in a proof.

❖   `(getQuotient k] (Q_k )` is the quotient.

❖   `(getRemainder k] (R_k)` is the remainder.

under the input assumptions provided in `<inputAssumptions>`, dividend `<getDividend>`, divisor `<getDivisor>`.

👉 **Note**
It is your responsibility to make sure that the proof generated by `solveNB_division` is successful (all lemmas pass) before calling `solveNB_division_assumptions` in another proof.

8.7.8   Definition of Integer Square Root

Consider an algorithm T that takes a bitvector input X and produces a bitvector output Q. Suppose the bitwidth of Q is k and the bitwidth of X is 2k bits.

```
X = x_{2k-1}....x_{k-1}...x_0
```

```
Q = q_{k-1}...q_0
```

We say that the algorithm T implements integer square root if the following conditions hold:

$$Q^2 <= X < (Q+1)^2 \qquad\qquad (13)$$

Instead of checking equation (13) one can check the following equivalent conditions where R is a k bit bitvector:

$$X = Q^2 + R \qquad\qquad (14)$$

$$0 <= R < (2Q+1) \qquad\qquad (15)$$

8.7.9   Standard Restoring Square Root Algorithm

A standard restoring square root algorithm (SRSA) has a for loop with k iterations. In each iteration one bit of result is computed starting from most significant bit (MSB) of the result. Given the similarity SRSA to restoring division we will refer to result in each iteration as partial quotient. That is, iteration number i produces quotient bit $q_{(k-i)}$ where $1 \le i \le k$ and $q_{(k-i)} \in \{0,1\}$. Let us denote the intermediate quotient and remainder in iteration i as Q_i and R_i. The relationships between intermediate quotient and individual quotient bits is as follows:

$$Q_1 = q_{(k-1)} \; 0 \;\; 0 \; ... \;\; 0$$

$$Q_2 = q_{(k-1)} \; q_{(k-2)} \; 0 \; ... \; 0$$

......

$$Q_k = q_{(k-1)} \; q_{(k-2)} \; q_{(k-3)} \; ..q_0$$

The following conditions are sufficient to show that a candidate SRSA algorithm T implements integer square root (we define $Q_0=0$ and $R_0=X$):

$$X = Q_i{}^2 + R_i \;\; (0 <= i <= k) \qquad\qquad (16)$$

$$R_i < ((2Q_i+1) \times 2^{(2k-2i)}) \;\; (0 <= i <= k) \quad (17)$$

$$R_i >= 0 \;\; (0 <= i <= k) \qquad\qquad (18)$$

Note that when `i = k` the above conditions imply that Q_k is the integer square root as per equations (14) and (15).

Instead of checking equation (16) for `0 ? i ? k` we can check the following equation for `0 <= i < k`.

$$R_i = (q_{k-i-1} \times (2Q_i + q_{k-i-1} \, 2^{k-i-1}) \times 2^{k-i-1}) + R_{i+1} \;\; (0 ? i < k) \qquad (19)$$

8.7.10   Verifying a Candidate SRSA in Hector

Suppose we are given a candidate SRSA algorithm T. In order to check if T implements division we require the user to provide following information:

❖ Input X. The user provides a TCL procedure that returns X. Suppose this TCL procedure is called `<getSqrtInput>`.

❖ Partial quotients $Q_0,...,Q_k$. The user provides a TCL procedure that takes one argument i and returns Q_i for each $0 \le i \le k$. Suppose this TCL procedure is called `<getQuotient>`.

❖ Partial remainders $R_0,...,R_k$. The user provides a TCL procedure that takes one argument i and returns R_i. Suppose this TCL procedure is called `<getRemainder>`.

❖ Assumptions under which the algorithm implements square root. Provide a TCL procedure that contains input assumptions specified using `assume` commands. Suppose this TCL procedure is called `<inputAssumptions>`.

❖ Number of iterations k in the square root algorithm. This is the same number as the number of quotient bits.

The above information about the input operand, partial quotients, and partial remainders should be derived using internal signals of the design. The internal signals in a C/C++ design can be made visible by using `Hector::show` commands. The internal signals in a RTL design can be accessed directly by using hierarchical names.

Using the above information you can create a proof to check if T implements square root. This is done by issuing the following command:

```
solveNB_sqrt k <getSqrtInput> <getQuotient> <getRemainder> <inputAssumptions>
                <identifier>
```

where `<identifier>` is the name that will be given to the generated proof. This command and others specifically related to square root verification are not included in the default Hector shell and must be enabled by adding this command to the `command_script` TCL file:

```
source $env(HECTOR_HOME)/local/sqrtchecks.tcl
```

The `solveNB_sqrt` command starts a proof containing following lemmas:

❖ One lemma for each $0 \leq i < k$ corresponding to equation (19). This gives rise to k lemmas. One additional lemma checks that $R_0 = X$.

❖ One lemma for each $0 \leq i \leq k$ corresponding to equation (17). This gives rise to k+1 lemmas.

❖ One lemma for each $0 \leq i \leq k$ corresponding to equation (18). This gives rise to k+1 lemmas.

❖ One lemma for each $0 < i \leq k$ that checks the definition of Q_i in terms of Q_{i-1} and quotient bit q_{k-i}. This gives rise to k lemmas. One additional lemma checks that $Q_0 = 0$.

If all lemmas in the generated proof are successful, then T implements square root under the given assumptions. More precisely, under the input assumptions provided in `<inputAssumptions>`, square root input `<getSqrtInput>`:

`[getQuotient k]($Q_k$)` is the integer square root of the given input.

8.7.10.1   Multiple Proofs

Multiple proofs can be started in the same way as division. See section "Multiple Proofs". Once can run lemmas for each iteration in parallel by setting the following variables:

```
set_hector_assume_guarantee true
```

```
set_hector_ag_partition_size 4
```

8.7.10.2   Proving a Subset of Iterations

The `solveNB_sqrt` command can optionally take two arguments `<lowIter>` and `<highIter>`. When these options are used Hector will only check the correctness a proof for only iterations i between `<lowIter>` and `<highIter>`. This is useful when you only want to run a subset for all iterations (for example, during initial phase of debugging the setup for square root proof).

```
solveNB_sqrt k <getSqrtInput> <getQuotient> <getRemainder> <inputAssumptions>
                <identifier> [<lowIter>] [<highIter>]
```

The use of options `<lowIter>` and `<highIter>` is discouraged because it is your responsibility to make sure all iterations are proven in order to conclude that the given design implements division.

8.7.10.3   Variable Number of Iterations

See section for more information.

8.7.11   Square Root Verification Examples

8.7.11.1   Integer square root of a 64-bit input

The `isqrt1` function takes a 64-bit unsigned integer input *num* and returns a 32-bit result *res*. The goal is to show that *res* is the integer square root of *num*.

```
unsigned long long isqrt1 (unsigned long long num)

{

    unsigned long long res = 0ULL;

    nsigned long long bit = 1ULL << 62;


    Hector::show("x", num);


    Hector::show("quo", res);

    Hector::show("rem", num);

    Hector::show("quo", res);

    Hector::show("rem", num);


    for (int i=0; i < 32; i++)

        {

             if (num >= res + bit)

             {

                 num -= (res + bit);

                  res = (res >> 1) + bitres >>= 1;

             }

        else

        {

             res >>= 1;

        }

        bit >>= 2;


        Hector::show("quo", res);

        Hector::show("rem", num);

        }

    return res;
```

```
    }
```

In order to verify isqrt1 we have inserted `Hector::show` commands in the source code to make internal design signals visible. The `Hector::show` calls before the *for* loop might appear to be redundant but the are actually critical to lemma generation process. Each *Hector::show()* call that references the same variable will be assigned a unique sequence number which will be used in the TCL procedures that return square root input, partial quotients, and partial remainders. These TCL procedures are as follows:

```
proc get_sqrt_input {} {
    return "spec.x(1)"
}
proc get_remainder {i} {
    return "spec.rem${i}(1)"
}
proc get_quotient {i} {
    set diff [expr 32 - $i]
    return "(spec.quo${i}(1) >> $diff)"
}
proc input_assumptions {} {}
```

To start the proof we issue the following command:

```
solveNB_sqrt 32 \
    get_sqrt_input \
    get_quotient \
    get_remainder \
    input_assumptions \
    isqrt1_proof
```

The success of the above proof establishes that `spec.quo32(1)` is the integer square root of the given input `spec.x(1)`.

8.7.12   Adding Assumptions after Proving a Design does Square Root

Suppose we have used `solveNB_sqrt` to show that a given design T does square root. That is, all lemmas in the following call to `solveNB_sqrt` are successful.

```
solveNB_sqrt k <getSqrtInput> <getQuotient> <getRemainder> <inputAssumptions>
                <identifier>
```

Then, you can call the TCL procedure `solveNB_sqrt_assumptions` to add the result of the above verification as assumptions in another proof.

```
solveNB_sqrt_assumptions k <getSqrtInput> <getQuotient> <getRemainder>
                            <inputAssumptions> <identifier>
```

The arguments to `solveNB_sqrt_assumptions` are exactly the same as `solveNB_sqrt`. This TCL procedure puts two assumptions in a proof:

[getQuotient k] (Q_k) is the quotient (result of square root)

[getRemainder k] (R_k) is the remainder

under the input assumptions provided in <inputAssumptions>, square root input <getSqrtInput>.

Note that it is your responsibility to make sure that the proof generated by solveNB_sqrt is successful (all lemmas pass) before calling solveNB_sqrt_assumptions in another proof.

9 Support for SystemC Data Types

This chapter describes the details of Hector's support for C++ designs utilizing SystemC datatypes in the following sections:

- ❖ *"Datatypes Supported by Hector"*
- ❖ *"Supported Methods"*
- ❖ *"Unsupported Methods"*
- ❖ *"Print and Dump Methods"*
- ❖ *"Integer Types"*
- ❖ *"Four-Valued Logic Types"*
- ❖ *"Maximum Bit Widths"*
- ❖ *"Writing the Design for Hector"*
- ❖ *"Compiling the Design"*
- ❖ *"Support for Overflow Flag in Fixed Datatypes"*
- ❖ *"Unsupported Datatypes/Classes"*
- ❖ *"Troubleshooting SC Datatypes Compile Errors"*

Hector does not support SystemC constructs and modeling styles except as described in this chapter.

SystemC provides a number of datatypes that are useful for hardware design. These datatypes are implemented as classes in C++. The supported and unsupported SystemC datatypes are described as well. If the design or program does not follow the restrictions mentioned in this section then Hector will produce an appropriate error or warning messages.

9.1 Datatypes Supported by Hector

- ❖ Integer Types
  - ✦ `sc_int`: finite precision signed integer
  - ✦ `sc_uint`: finite precision unsigned integer
  - ✦ `sc_bigint`: arbitrary precision signed integer
  - ✦ `sc_biguint`: arbitrary precision unsigned integer
- ❖ Fixed-point Types
  - ✦ `sc_fixed`: arbitrary precision signed fixed point number
  - ✦ `sc_ufixed`: arbitrary precision unsigned fixed point number

- ❖ Bit Vector Types
  - ✦ `sc_bit`: single bit
  - ✦ `sc_bv`: arbitrary size bit vector
- ❖ 4-valued Logic Types
  - ✦ `sc_logic`: single 4-valued logic type
  - ✦ `sc_lv`: arbitrary size logic vector

9.2   Supported Methods

Hector supports all arithmetic, bitwise, relational, shift, bit select, part select, assignment, and conversion to C integer methods provided by SystemC. All datatypes supported have their vector length/precision specified using template parameters such that it is known during compilation.

9.3   Unsupported Methods

9.3.1   Unsupported Methods for All Data Types

- ❖ Operators that use *string* as arguments or as return values.
- ❖ All methods for concatenation support (except concatenation of SystemC integers)

9.3.2   Unsupported Methods for Integer Types

- ❖ Methods `set_packed_rep`, `get_packed_rep`

9.3.3   Unsupported Methods for Fixed-Point Types

- ❖ Methods intended for internal use of the SystemC library, such as `quantization_flag`, `type_params`, `get_slice`, `set_slice`, `get_rep`, `set_rep`, `lock_observer`, `unlock_observer`, `observer_read`, `value` and `is_normal` are not supported.
- ❖ Overflow mode `SC_WRAP` and `SC_WRAP_SM` with `n_bits` > 0.

9.4   Print and Dump Methods

All print and dump methods are ignored assuming there are no side-effects in the methods.

9.5   Integer Types

SystemC has support for two types of integer datatypes: finite and arbitrary precision. The finite precision versions are available for more efficient simulation but are limited to 64 bits. From Hector's point of view `sc_int`/`sc_uint` are semantically equivalent to the `sc_bigint`/`sc_biguint` counterparts and have all the operations supported by the later. However, the use of `sc_int`/`sc_uint` should be limited to 64 bit precision, so that it adheres to the simulation semantics.

9.6   Four-Valued Logic Types

SystemC has support for 4-valued logic types. However, the underlying representation of this type in Hector is not as efficient as the other types. It is recommended to use the other types whenever possible. For example, if a design is using only Boolean logic, although it can be represented using `sc_logic` and `sc_lv`, it is recommended to use `sc_bit` or `sc_bv` for efficiency reasons. We support all four datatypes.

9.7   Maximum Bit Widths

The arbitrary precision datatypes have an internal implementation limit. The default values of these limits are listed in Table 9-1. These limits can be changed using compile time options. Hector assumes that the precision is set high enough so that the internal implementation limits do not change the behavior of the design. Note that Hector will find discrepancies if the limits are not set high enough.

Table 9-1   Maximum Bit Widths

Datatypes	Default Maximum Precision	Can be changed using compile-time option
Integer Types		
sc_int<W>	64	No
sc_uint<W>	64	No
sc_bigint<W>	512	SC_MAX_NBITS
sc_biguint<W>	512	SC_MAX_NBITS
Fixed-Point Types		
sc_fixed<wl, iwl, qmode, omode, n_bits>	1024	SC_FXMAX_WL
sc_fixed<wl, iwl, qmode, omode, n_bits>	1024	SC_FXMAX_WL

9.8   Writing the Design for Hector

The C++ code should be written as specified in this manual. This code can contain Hector directives such as `registerInput`, `registerOutput`, `beginCapture`, `endCapture`, and `show`. The SystemC datatypes should be used as any other C++ class.

9.8.1   Registering SystemC Variables with Hector

To register a SystemC datatypes as Input or Output with Hector, use the following functions:

```
Hector::registerInput(const char *name, T& obj);

Hector::registerOutput(const char *name, T& obj);
```

[Where, T is one of the allowed SystemC datatypes.]

🛑 **Caution**

Note that the above `registerInput/Output` functions internally makes only the value part of the datatypes as input or output, thereby, relieving you from knowing the implementation details of the datatype classes.

Example

```
#include <systemc.h>

#include <iostream>

typedef sc_biguint<48>data1;

typedef sc_int<16>data2;
```

```
        typedef sc_bigint<49>res_t;


        res_t dut (data1& ina, data2& inb, int idx) {
            res_t out;
            out= in_a + in_b;
            out  += idx;
            return out;
        }
        void main(){
            data1 inpa;
            data2 inpb;
            int i;
            res_t outp;
            Hector::registerInput("ina", inpa);
            Hector::registerInput("inb", inpb);
            Hector::registerInput("idx", i);
            Hector::registerOutput("out", outp);


            Hector::beginCapture();
                outp = dut(inpa, inpb, i);
            Hector::endCapture();
        }
```

9.9   Compiling the Design

You can compile the C++ code containing SystemC datatypes by creating a TCL procedure in the command script file and by using the `create_design`, `scdtan`, and `compile_design` commands. The `create_design` and `compile_design` commands are described in section "Compiling a C/C++ Design".

9.9.1   Analyzing the Files

The command for analyzing C++ code containing SystemC datatypes is as follows:

```
        scdtan <options> <list of filenames>
```

Options:

  ❖   `<options>`: g++ compiler options for compiling the files

  ❖   `<list of filenames>`: List of files to compile

Apart from all the default options used for the compilation of a C++ files (see Section "Analyzing C++ Files"), it also adds the include path for Hector's version of SystemC datatypes.

Example

```
        proc compc_spec { } {
```

```
create_design –name spec –top main

scdtan play.cc

compile_design spec

}
```

9.10   Support for Overflow Flag in Fixed Datatypes

Hector supports the `overflow_flag` for the `sc_fixed/sc_ufixed` datatypes. However, to separate implementation details from functionality, you should always use the accessor functions and never access the internal data variables directly. This in turn means that if we have external `sc_fixed` port variables then we have to declare it with two `registerInputs/Outputs` using the public access functions.

Example

```
sc_fixed<14, 8> a;

Hector::registerInput("a_value", a);

Hector::registerInput("a_o_flag", a.overflow_flag());
```

And, in the Hector TCL file use:

```
assume a0 = spec.a_value(1) == impl.a_val(1);

assume a1 = spec.a_o_flag(1) == impl.a_oflag(1);
```

Caution

Note that the `registerInput` for a only registers the value part of the `sc_fixed` variable and not the entire object. Also, note that the `registerInput` for the overflow flag is same as registering a C++ variable. The above steps allows you to use the `sc_fixed` datatypes without any specific knowledge about the implementation of the datatype classes.

9.11   Unsupported Datatypes/Classes

The following underlying classes should not be used directly in the user designs. If an unsupported class or datatype is used, Hector produces an appropriate error message.

9.11.1   Unsupported Integer Types

❖ `sc_signed`

❖ `sc_unsigned`

❖ `sc_int_base`

❖ `sc_uint_base`

9.11.2   Unsupported Fixed-Point Types

❖ Limited Precision fixed-point types used for faster simulation

✦ `sc_fixed_fast`

✦ `sc_ufixed_fast`

❖ Unconstrained types and related context classes

✦ `sc_fix`

✦ `sc_ufix`

- ✦ `sc_fix_fast`
- ✦ `sc_ufix_fast`
- ✦ `sc_fxcast_context`
- ✦ `sc_fxcast_switch`
- ❖ Arbitrary precision value
  - ✦ `sc_fxval`
  - ✦ `sc_fxval_fast`
- ❖ Observer types
  - ✦ `sc_fxnum_observer`
  - ✦ `sc_fxnum_fast_observer`
  - ✦ `sc_fxval_observer`
  - ✦ `sc_fxval_fast_observer`

9.11.3   Unsupported Bit Vector Types

- ❖ `sc_bv_base`
- ❖ `sc_lv_base`

9.12   Troubleshooting SC Datatypes Compile Errors

9.12.1   Cannot Detect if Loop Terminates

If you are using the SystemC range operator then you may get the following error message:

```
Error:

*** Error SCC-001: Cannot detect if loop terminates because check always came back
inconclusive (ran 50 iterations).

Maximum number of iterations is determined by option
'_hector_sym_maxiter_allfail'.

File

'/<path>/Hector.h', around line 1065 Stack trace:

  in getRange

      called at …
```

For range operators if the lower and/or the upper bounds are not constant than Hector internally uses a loop to go through the range. So, if the bitwidth of the datatype is greater than the value of the above Hector flag `_hector_sym_maxiter_allfail` (by default it is set to 50), you will get this error message. Ideally, the flag should be set to the maximum of the bit widths in the design.

9.12.2   No Operator Matches these Operands

If you are using SystemC concatenation or range operators then you may get the following error message:

```
Error:
```

```
"foo.cpp", line xx: error: no operator "op" matches these operands operand types
are: sc_dt::hector_int_subref<120, true> op int sc_bigint<120> a = b(hi, lo) op c;
```

In the error message above the operator *op* can be any allowed binary operator and at least one of the operands is a range (`subref`) or a concatenation (`concref`) operand.

In Hector all the SC datatypes are implemented using static templates, instead of dynamic arrays (which is the case for OSCI SystemC). The benefit of using static templates is that we get efficient DFGs. However, a side-effect of this is that we lost the capability of implicit casting when the bitwidth is greater than 64, which is the reason for the above compile-time error.

Unfortunately, currently a general solution is not possible. Various of these operators are implemented, however there may be cases when g++ is able to compile the design but Hector cannot. The workaround for this scenario is to rewrite the code in two step, where the first step is the explicit casting and the second step is the operation.

```
sc_bigint<120> a = b(hi, lo);

a = a + c;
```

10 Appendix: Command Script File

The command script file is a TCL file and it is the mechanism by which the user specifies various pieces of information that control Hector's execution. It consists of Hector specific Hector variables and Hector specific TCL commands and standard TCL commands common to Synopsys script interpreters. The Hector specific TCL commands are divided into two categories: setup commands and run-time commands.

The Hector installation contains a template that can be used as a template for creating a command script file. This standard template provides a straightforward way to run most examples or serves as a starting point for customization of more complicated examples. It can be found in the Hector installation in the following location:

```
$HECTOR_HOME/tutorial/quickstart/command_script.tcl
```

10.1   Predefined Hector Variables

In order to set a predefined Hector variable the user should use `get_<var>/set_<var>` TCL procedures. For example, in order to set `user_assumes_lemmas_procedure` to `foo` the following TCL procedure must be called.

```
set_user_assumes_lemmas_procedure   "foo"
```

In order to get the current value of `user_assumes_lemmas_procedure` the following TCL procedure must be called.

```
get_user_assumes_lemmas_procedure
```

The Hector variables that are commonly used are listed below. Table 10-1 provides the name of the predefined variable, a simple description, and a reference in this document for more details on that variable.

Table 10-1   Hector Variables

completeness_proof_solve_script	Specifies a solve script to be used for completeness lemmas generated by case splitting.	Section "Case Splitting"
custom_solve_script	Specifies the custom solver script file.	Section "Overriding the Solve Script"
custom_solve_script_by_proof_name (<proofname>)	Specifies the custom solver script for proof 'proofname'.	Section "Overriding the Solve Script"
hector_check_array_bounds	Enables or disable array bounds lemmas.	Section "Mandatory Automatically Extracted Properties (AEPs)"

Table 10-1   Hector Variables

hector_check_asserts	Enables or disable assertion lemmas.	Section "Assertion Lemmas"
hector_check_maxiter_pragmas	Enables or disable max iterations pragma lemmas.	Section "Max Iteration Lemmas"
host_file	Provide host file name for Hector MP.	Section "Specifying the MP Resources"
_hector_ag_assume_all_prev	Assume all earlier lemma partitions when proving current lemma partition.	Section "Case Splitting"
_hector_ag_partition_list	Partition the set of lemmas based on the list given.	Section "Case Splitting"
_hector_ag_partition_size	Partition the set of lemmas such that there are n number of lemmas in each tasks. Also possible to specify a list of numbers for finer partitioning.	Section "Case Splitting"
_hector_assume_guarantee	Enable assume-guarantee reasoning	Section "Case Splitting"
_hector_case_splitting_procedure	Name of the TCL procedure where case splitting commands are specified.	Section "Case Splitting"
_hector_lemma_partition	Partition the set of lemmas into different tasks. Also possible to specify a list of numbers for finer partitioning.	Section "Assume-guarantee Based on Lemma Partitions"
_hector_lemma_partition_complete	Ask Hector to complete the lemmas that were not specified in the hector_lemma_partitions list.	Section "Assume-guarantee Based on Lemma Partitions"
_hector_lemma_partition_list	Partition the set of lemmas based on the list given.	Section "Assume-guarantee Based on Lemma Partitions"
_hector_lemma_partition_size	Partition the set of lemmas such that there are n number of lemmas in each tasks.	Section "Assume-guarantee Based on Lemma Partitions"
_hector_multiple_solve_scripts	Run multiple solve scripts (different orchestrations) on the same problem.	Section "Partitioning Lemmas in a Given Proof"
_hector_multiple_solve_scripts_list	Run multiple solve scripts on the same problem from the given list.	Section "Partitioning Lemmas in a Given Proof"

Table 10-1   Hector Variables

_hector_task_timeout	Control the time allocated for a given task.	Section "Task Timeout"
_hector_rew_heavy_num_mults	Expensive rewrites for multipliers are activated if the number of multipliers is <= the value of this variable.	Section "Overriding the Solve Script"
orch_distrib	Controls application of distributive rewrites for multipliers. Only useful with the orch_multipliers solve script.	Section "Overriding the Solve Script"
orch_init_sat_time_limit	Effort level for solvers to prove the assumption constraints.	Section "Overriding the Solve Script"
orch_solve_outputs_effort	Effort level for the Hector solvers.	Section "Overriding the Solve Script"
set_resource_limit	Effort level for the SAT solvers.	Section "Overriding the Solve Script"
user_assumes_lemmas_procedure	Identifies the TCL procedure that contains the user provided assumptions and lemmas.	Section "Using Assumptions and Lemmas in a Proof"

Less commonly used Hector variables are listed in Table 10-2:

Table 10-2   Other Hector Variables

_hector_sim_dont_use_x	Don't use X during random simulation.	Section "Establishing an Initial State for RTL Models"
_hector_sym_maxiter_allfail	Maximum iterations for loop unrolling when all tests are inconclusive.	Section "Hector's C/C++ Compiler Assumptions"
_hector_sym_maxiter_allgood	Maximum iterations for loop unrolling when all tests for loop condition are conclusive.	Section "Hector's C/C++ Compiler Assumptions"
_hector_sym_maxiter_partial	Maximum iterations for loop unrolling when some tests are conclusive and some inconclusive.	Section "Hector's C/C++ Compiler Assumptions"
_hector_sym_maxstackdepth	Maximum stack depth of recursive functions	Section "Coding Guidelines"
_hector_sym_strict_malformed_checks	Enable strict checking for malformed programs.	Section "Compiling Loops"

10.2 Hector Specific TCL Set-up Commands

The Hector specific TCL set-up commands perform additional configuration of the Hector environment. They can be executed in an interactive session or placed in the command script file.

Table 10-3   Hector Specific TCL Set-up Commands

assume	Provides assumptions on inputs or variables in C++ and on signals and registers in RTL.	Section "Assumptions and Lemmas"
caseAssume	Specify an assumption that forms the part of the currently selected case split.	Section "Generated DFG for each Cutpoint"
caseBegin	Start a case split.	Section "Declaring a Cutpoint in the RTL"
caseConstraint	Specify a constraint that forms the part of the currently selected case split.	Section "Using Cutpoints in Proof"
caseEnumerate	Creates a collection of case splits by performing a specified type of enumeration on a given expression.	Section "Troubleshooting Cutpoints"
caseSplitStrategy	Provide a name to a collection of case splits.	Section "Declaring a Cutpoint in the C/C++"
cutpoint	Enable a cutpoint signal.	Section "Division Verification Examples"
lemma	Generates a lemma that will be proven by Hector.	Section "Assumptions and Lemmas"
set_aep_selection	Controls the checking of automated lemmas.	Section "Controlling AEP Lemmas"

10.3 Hector Specific TCL Runtime Commands

The Hector specific TCL run-time commands cause some action to be performed by Hector. They are listed in Table 10-4 and can be executed in an interactive session or placed in the command script file. However, if multiple pair of design needs to be proven with Hector, it is recommended that a new invocation of Hector is created for each pair of design. At this time running multiple designs in a single Hector session does not work.

Table 10-4   Hector Specific TCL Runtime Commands

cdproof	Changes to a specific proof.	Section "Viewing Status of Proofs"
compile_design	Compiles the specified design in to a DFG.	Section "Generating the DFG"
compose	Creates the internal test framework.	Section "Composing the Equivalence Problem"
cppan	Analyzes a C/C++ program.	Section "Analyzing C++ Files"
create_design	Creates a specification or implementation design.	Section "Creating a Design"
getTaskDetails	Returns a TCL list with details of lemmas running in each task.	Section "Viewing Status of Proofs"

Table 10-4   Hector Specific TCL Runtime Commands

ignore_functions	Ignores a list of functions for C/C++/SystemC programs.	Section "Using Assertions and Checks for Malformed Programs"
killTasks	Kills one or more scheduled or running tasks.	Section "Interactive Proof Creation and Control"
listassumes	Lists all the assumptions of the currently selected proof.	Section "Viewing Status of Proofs"
listlemmas	Lists all the lemmas of the currently selected proof.	Section "Viewing Status of Proofs"
listproof	Lists the status of all the assumptions and lemmas of the currently selected proof.	Section "Viewing Status of Proofs"
listproofs	Lists the status of all existing proofs.	Section "Viewing Status of Proofs"
listtask	Lists information (error messages) posted by the task.	Section "Viewing Status of Proofs"
listtasks	Lists the status of all existing tasks.	Section "Viewing Status of Proofs"
listTaskDetails	Lists information about each lemma in each task.	Section "Viewing Status of Proofs"
list_aep_ selection	Lists the enable status of AEP lemmas.	Section "Controlling AEP Lemmas"
proofstatus	Returns 1 if all lemmas in all proofs in the list were successful.	Section "The proofstatus Command"
proofwait	Waits for each proof in the list provided to have a "conclusive" status.	Section "The proofwait Command"
reload_script	Sources the TCL command script again without exiting the Hector shell.	Section "Invoking Hector"
scdtan	Analyzes a SystemC datatypes program.	Section "Creating a Design"
simcex	Simulates a counter example, saves results in one or more forms.	Section "The simcex Command"
solve	Proves the equivalence or in-equivalence of each corresponding output. (blocking command)	Section "Blocking Commands"
solve_init	Proves the equivalence or in-equivalence of each corresponding output starting from the initial state. (blocking command)	Section "Blocking Commands"
solveNB	Proves the equivalence or in-equivalence of each corresponding output. (non-blocking command)	Section "Non-blocking Commands"
syscan	Analyzes a SystemC program.	Section "Creating a Design"

Table 10-4   Hector Specific TCL Runtime Commands

vlogan	Analyzes a Verilog program.	Section "Analyzing Verilog Files"
vhdlan	Analyzes a VHDL program.	Section "Analyzing VHDL Files"
vcs	Analyzes and elaborate the Verilog only designs. Elaborates an RTL design for VHDL and MX designs.	Section "Elaborating the Design (Optional)"

10.4   Supported Math Library Functions

The following math library functions are supported:

float   ceilf(float arg)

double   ceil(double arg)

long double   ceill(long double arg)

float   floorf(float arg)

double   floor(double arg)

long double   floorl(long double arg)

float   truncf(float arg)

double   trunc(double arg)

long   doubletruncl(long double arg)

float   roundf(float arg)

double   round(double arg)

long double   roundl(long double arg)

float   rintf(float arg)

double   rint(double arg)

long double   rintl(long double arg)

float   nearintf(float arg)

double   nearint(double arg)

long double   nearintl(long double arg)

float   fabsf(float arg)

double   fabs(double arg)

long double   fabsl(long double arg)

float	*fmaxf(float x, float y)*
double	*fmax(double x, double y)*
long double	*fmaxl(long double x, long double y)*
float	*fminf(float x, float y)*
double	*fmin(double x, double y)*
long double	*fminl(long double x, long double y)*
float	*fmaf(float x, float y, float z)*
float	*modff(float x, float* iptr)*
double	*modf(double x, double* iptr)*
long double	*modfl(long double x, long double* iptr)*
float	*std::copysign(float x, float y)*
double	*std::copysign(double x, double y)*
long double	*std::copysign(long double x, long double y)*
int	*sinff(float x)*
int	*sinf(double x)*
int	*isinfl(long double x)*
int	*std::isfinite(float x)*
int	*std::isfinite(double x)*
int	*std::isfinite(long double x)*
int	*std::isnormal(float x)*
int	*std::isnormal(double x)*
int	*std::isnormal(long double x)*
int	*isnanf(float x)*
int	*isnan(double x)*
int	*isnanl(long double x)*
int	*std::fpclassify(float x)*

int   *std::fpclassify(double x)*

int   *std::fpclassify(long double x)*

11 Appendix: Using Designware Components

You may want to perform equivalence checking on designs that have Designware (DW) parts in them. Two types of DW parts can be found at a customer site: encrypted synthesis parts and simulation parts that are available in plain text (with copyright notice).

There are number of problems with the use of plain text simulation models. The main technical problem is that the RTL compiler used by Hector is not always able to compile the simulation models. This is because the simulation models are designed to be non-synthesizable (for example, they make use of while loops). In order to address this problem a user needs to modify the simulation model such that is becomes *synthesizable* for Hector's RTL compiler Simon. The process of modifying a simulation model to make it synthesizable is tricky and raises the question of formal equivalence between the original simulation model and the modified simulation model. Another technical problem is that simulation models may not be formal verification friendly if they are not implemented efficiently. Furthermore, modifying a simulation model violates the copyright notice in the text file.

Given the problems involved with the use of simulation parts, we have enhanced Hector such that it can read encrypted DW synthesis parts. Some steps need to be performed in order to use Hector with encrypted Designware parts.

11.1   Analyze the Designware Source Tree for the Formal Models

Hector is shipped with a script called `analyze_identp_dware.sh` in `$HECTOR_HOME/local/dware/identp`. You can use this script to run appropriate analysis commands (`vlogan`/`vhdlan`) on the Designware source tree. The usage is as follows:

```
$HECTOR_HOME/local/dware/identp/analyze_identp_dware.sh \

    <$DW_HOME> \

    <$OUTPUT_DIR>
```

The first argument to the script is the directory location where the Designware directory is present (typically top level DC installation directory). This directory has subdirectories `dw, dw/scripts, dw/dw01, …, dw/dw06.`

The second argument is the name of the directory where the results of the analysis will be stored. Both arguments must be absolute paths.

For example:

```
$HECTOR_HOME/local/dware/identp/analyze_identp_dware.sh \

    /home/tools/syn_2010.12-SP1 \
```

```
    `pwd`/dware_analyze
```

REQUIREMENT: DC version (syn_2010.03-SP4) or above

> **Note**
> This step needs to be done only once for a given Hector version and version of Designware.

11.2   Analyze the Designware Source Tree for Counter-example Simulation

Hector is shipped with a script call `analyze_dware_sim.sh` in `$HECTOR_HOME/local/dware/identp`. This script should be used to run appropriate analysis commands (`vlogan/vhdlan`) on the Designware source tree. This produces a pre-analyzed library that the `simcex` command uses. The usage is as follows:

```
$HECTOR_HOME/local/dware/identp/analyze_dware_sim.sh \

    <$DW_HOME> \

    <$OUTPUT_DIR>
```

The first argument to the script is the directory location where the Designware directory is present (typically top level DC installation directory). This directory has subdirectories `dw`, `dw/scripts`, `dw/dw01`, ..., `dw/dw06`.

The second argument is the name of the directory where the results of the analysis will be stored. Both arguments must be absolute paths.

Example

```
$HECTOR_HOME/local/dware/identp/analyze_dware_sim.sh \

    /home/tools/syn_2010.12-SP1 \

    `pwd`/dware_analyze_simcex
```

REQUIREMENT: DC version (syn_2010.03-SP4) or above

> **Note**
> This step needs to be done once for each pair of Designware and VCS versions.

11.3   Using Designware Components in Formal Models

After the Designware source tree has been analyzed, specify the location of the directory where formal model analysis results were stored. This is done by passing `-dware=DIR` as an option to `create_design`.

When using synthesis Designware components the design effectively becomes a mixed HDL (MX) design. This is because some synthesis components are in VHDL. So the language must be specified as `mx` with the `create_design` command. Note one cannot use pure Verilog compile flow which uses single call to `vcs` command in the compilation commands. Instead one needs to first analyze files with `vlogan/vhdlan` (this is how one compiles MX designs and VCS-MX user guide will describe this in detail).

Some examples are shown below.

11.3.1   Example

The wrapper below instantiates the Designware `DW_fp_mult` component. The contents of the wrapper `DW_fp32_mult_wrapper.v` are below.

```
module DW_fp32_mult_wrapper (a, b, rnd, z, status);

    input  [31:0] a;
```

```
      input   [31:0] b;

      input    [2:0] rnd;

      output [31:0] z;

      output  [7:0] status;

      DW_fp_mult #(23, 8, 1) fp32_mult (

              .a(a),

              .b(b),

              .rnd(rnd),

              .z(z),

              .status(status));

   endmodule
```

You can compile this design as follows:

```
proc comprtl {} {

    create_design -name "impl" -top DW_fp32_mult_wrapper \

      -options "-dware=/user/foo/dware_analyze" \

      -lang "mx"

    vlogan  DW_fp32_mult_wrapper.v

    compile_design "impl"

}
```

11.3.2   Example

The following VHDL wrapper instantiates DW01_add:

```
library IEEE;

library DW01;

use IEEE.std_logic_1164.all;

use DW01.all;

entity DW01_add_inst is

  generic (inst_width : NATURAL := 8);

  port(inst_A, inst_B : in std_logic_vector(inst_width-1 downto 0);

      inst_CI : in std_logic;

      SUM_inst : out std_logic_vector(inst_width-1 downto 0);

      CO_inst : out std_logic);

end DW01_add_inst;

architecture inst of DW01_add_inst is

  component DW01_add

  generic (width : NATURAL := 8);

  port(A,B : in std_logic_vector(width-1 downto 0);
```

```
        CI : in std_logic;

        SUM : out std_logic_vector(width-1 downto 0);

        CO : out std_logic);

    end component;

begin

  U1: DW01_add

    port map (

      A   => inst_A,

      B   => inst_B,

      CI  => inst_CI,

      SUM => SUM_inst,

      CO  => CO_inst);

    end inst;
```

In order to compile the above wrapper we can use following `comprtl`:

```
proc comprtl {} {

    create_design -name "impl" -top DW01_add_inst \

      -options "-dware=/user/foo/dware_analyze" \

      -lang "mx"

    vhdlan wrapper.vhd

    compile_design "impl"

}
```

11.4   Caution when using Designware Synthesis Components

There is no one single implementation of certain Designware components like `DW02_multp`. This is because during synthesis Design Compiler can generate different implementations based on synthesis constraints. So even though the user may verify against the synthesis model of `DW02_multp` there is no guarantee that the same model will be used during synthesis. So the safest solution when using `DW02_multp` is to black box `DW02_multp`. After black-boxing the outputs of `DW02_multp` become primary inputs. One can safely assume that the sum of two outputs must equal the product of two inputs for `DW02_multp` (any implementation generated during synthesis must satisfy this constraint).

11.5   Using Designware Components in Counterexample Simulation

Once the Designware source tree has been analyzed you need to specify the location of the directory where analysis results were stored for `simcex`. This is done in Hector by setting the global TCL variable `_hector_dware_sim` to the absolute path to the directory into which they were analyzed.

11.5.1   Example

If you analyzed the library to `` `pwd` ``/dware_analyze_simcex as suggested above, a way to do this is:

```
set _hector_dware_sim $env(PWD)/dware_analyze_simcex
```

In all other cases, simply set `_hector_dware_sim` to the absolute path to the analysis directory:

```
set _hector_dware_sim /path/to/dware_analyze_simcex
```

12 Appendix: Frequently Asked Questions

12.1   Hector not Converging on Problems

Some ideas that can help get convergence on a problem are as follows:

1. Read "Advanced Proof Techniques" on advanced proof techniques. It describes many Hector features that can help get convergence.

2. If there are control signals in the design, you may want to case split on those. Case splitting is a powerful technique to break a problem into small sub-problems. It is described in section "Case Splitting".

3. When trying to compare an N bit word, you may want to check each of the bits independently. For example, you can write a TCL loop to generate a lemma for each bit. For example

   ```
   for {set I 0} {$i < $N} {incr } {

       lemma check_bit_$i = spec.out [$i]  == impl.out [$i]

   }
   ```

4. Arithmetic operators such as multiplication/division typically make the proofs much harder. If both *spec* and *impl* contain a *matching* hard operation such as multiplication, you can explicitly provide intermediate lemmas that relate the inputs to the multipliers and the expected relationship between the outputs of the multipliers. If you are able to prove these relationships, then you can create another proof where you can "assume" these relationships to guide Hector solvers to prove the design.

   After compiling the design you can use the `statsdfg` command to see what type of nodes are present in data flow graph. Note that not all of these nodes might be relevant for the proof (some of them might disappear due to constraints/constant propagation).

5. If you expect the two designs to have internal matching points, then you can provide those as lemmas in Hector. This can also help with convergence. Furthermore, it can help you isolate which parts of the code are contributing to the hardness of the proof.

12.1.1   HDPS not Triggering due to Difference in Support Size

For a lemma of the form LHS == RHS it is first checked whether the support (primary inputs in cone) of LHS and RHS of expression match exactly or not. If they do not match exactly, then HDPS does not trigger.

You can run `listtask <tasknum>` to see if HDPS triggered or not. If HDPS does not trigger due to difference in support size, the input bits that cause support to not match are printed.

For example, consider the following lemma:

```
Lemma l = spec.out(1) == spec.a(1) * spec.b(1)
```

It is possible that `spec.out(1)` depends on some extra primary input bits (other than `spec.a` and `spec.b`). These dependencies are structural in nature and not functional. Hector tries to eliminate such dependencies by doing rewrites/redundancy removal, however, this does not always guarantee that all dependencies will get eliminated. Such non-functional dependencies of either LHS or RHS can cause HDPS to not trigger.

In order to work around this one can simply case split on extra bits. By case splitting we force such bits to constants so in each case so the dependency disappears and HDPS triggers.

For example suppose in the above lemma `spec.out(1)` depended on another primary input bit `spec.cntrl(1)`. Then you can case split on this bit as follows:

```
proc my_casesplit {}  {

   caseSplitStrategy anyStrategyName

   caseEnumerate anyName -expr "spec.cntrl(1)"

}

set_hector_case_splitting_procedure "my_casesplit"
```

12.2   Using –reset Option with the create_design Command

When you specify `-reset` option Hector computes the reset state after asserting reset for one clock cycle. Once this is done, the reset signal is changed to opposite polarity for any further proofs. All this happens during compilation itself.

❖   When you call *solve* Hector does not assume initial/reset state. This is because Hector is trying to do a proof for a general transaction and not just the one starting from reset state. So that is why you do not see *reset* asserted for one clock cycle.

❖   When you do `solve_init` Hector will assume that transaction starts from reset state. That is, registers will be constrained to match with reset state. You will still not see reset asserted for one cycle. You will just see that registers have the right reset value.

If you do no not specify `-reset` during compilation, then the reset signal is just like any other free input and you can control it using assumes. This is more general because you can do reset for multiple cycles.

12.3   Convergence on Floating-point Multiply and Adds

This section describes some techniques that are useful for verifying FMAs.

The simplest technique is to run all solve scripts in parallel on a grid. If the two designs to be compared have a lot of similarity this is likely going to solve the problem. Next we describe what to do when this does not work.

Suppose you are comparing an RTL that implements FMA with a C++ model for FMA. Two situations occur commonly.

❖   RTL uses a * operation to multiply mantissas. Convergence technique for this is discussed below in section "Case Splitting for FMAs".

❖   RTL uses a gate level multiplier such as a Booth multiplier to multiply mantissas. This situation is more complex and the proof is typically done by breaking the proof into two sub-proofs using assume-guarantee technique.

   ✦   [GUARANTEE PART] Identify the internal signals that represent the input and the output of multiplier of mantissas. Suppose the input signals are *mant1, mant2* and multiplication result signal is `mant_prod`. Then we want to create one proof that checks a lemma of the form.

```
lemma multcheck = (mant1 * mant2) == mant_prod
```

Hector has a specialized solver called Hector Datapath Solver (HDPS) to solve lemmas of the above type. For more information, see section "Division and Square Root Verification in Hector".

✦ [ASSUME PART] Create another sub-proof where we use above lemma as an assumption.

```
assume multcheck = (mant1 * mant2) == mant_prod
```

The goal of this proof is to check the outputs of FMA operation under the above assumption. This proof is checking that the logic outside the multiplier such as rounding is working or not. Convergence technique for this is discussed below in section"Case Splitting for FMAs".

12.3.1   Case Splitting for FMAs

In order to get convergence for scenario (1) or (2B) you need to employ case splitting. Hector can divide the input space into various cases and try to prove each case separately based on user input about which cases to create. Case splitting is a powerful technique discussed in more detail in section "Using Cutpoints". Hector automatically checks whether case splitting is complete or not. Note that proof for each case split can be run on the grid in parallel using Hector Multi-processing (MP) capability. The use of MP is recommended to get reasonable convergence times.

An FMA operation has three floating-point operands `spec.a, spec.b, spec.c`. The goal is to compute

```
(spec.a * spec.b) + spec.c
```

The assumption that you are familiar with IEEE floating-point format and which bits correspond to exponents/mantissas.

A sample case split for a double precision FMA (you must use the right phase for signals based on the design).

One sample case split strategy. Depending on the design different case splitting may be needed.

```
proc casesplits {} {
    # We use "normal" to refer to an FP number that is
    # not denormal, NAN, infinity or zero.
    caseSplitStrategy basic
    # a, b are denormals and c is normal
    #
    caseBegin denormal_denormal_normal
    caseAssume (spec.a(1)[62:52] == 11'h0)
    caseAssume (spec.b(1)[62:52] == 11'h0)
    caseAssume (spec.c(1)[62:52] != 11'h7ff)
    caseAssume (spec.c(1)[62:52] != 11'h0)
    # one of the three operands is NAN or infinity
    #
    caseBegin inf_or_nan
    caseAssume (spec.a(1)[62:52] == 11'h7ff) || (spec.b(1)[62:52] == 11'h7ff) ||
    (spec.c(1)[62:52] == 11'h7ff)
```

```
# all of the three operands are denormal
#
caseBegin denormal_denormal_denormal
caseAssume (spec.a(1)[62:52] == 11'h0)
caseAssume (spec.b(1)[62:52] == 11'h0)
caseAssume (spec.c(1)[62:52] == 11'h0)
#  all of the three operands are normal
#
caseBegin normal_normal_normal
caseAssume (spec.a(1)[62:52] != 11'h7ff)
caseAssume (spec.b(1)[62:52] != 11'h7ff)
caseAssume (spec.c(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] != 11'h0)
caseAssume (spec.b(1)[62:52] != 11'h0)
caseAssume (spec.c(1)[62:52] != 11'h0)
#  a is denormal, b and c are normal
caseBegin denormal_normal_normal
caseAssume (spec.b(1)[62:52] != 11'h7ff)
caseAssume (spec.c(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] == 11'h0)
caseAssume (spec.b(1)[62:52] != 11'h0)
caseAssume (spec.c(1)[62:52] != 11'h0)
   # further case splitting in above case. The case split
   # is based on position of leading one in spec.a[51:0].
   #
   caseEnumerate in1_leading1 -type leading1 \
   -parent denormal_normal_normal -expr "spec.a(1)[51:0]"
#  b is denormal, a and c are normal
caseBegin normal_denormal_normal
caseAssume (spec.a(1)[62:52] != 11'h7ff)
caseAssume (spec.c(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] != 11'h0)
caseAssume (spec.b(1)[62:52] == 11'h0)
caseAssume (spec.c(1)[62:52] != 11'h0)
   # further case splitting in above case. The case split
   # is based on position of leading one inin spec.b[51:0].
   #
```

```
caseEnumerate in2_leading1 -type leading1 \
-parent normal_denormal_normal -expr "spec.b(1)[51:0]"
#   a, b are normal, c is denormal
#
caseBegin normal_normal_denormal
caseAssume (spec.a(1)[62:52] != 11'h7ff)
caseAssume (spec.b(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] != 0)
caseAssume (spec.b(1)[62:52] != 0)
caseAssume (spec.c(1)[62:52] == 0)
#   a, c are denormal, b is normal
#
caseBegin denormal_normal_denormal
caseAssume (spec.b(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] == 11'h0)
caseAssume (spec.b(1)[62:52] != 11'h0)
caseAssume (spec.c(1)[62:52] == 11'h0)
    # further case splitting in above case. The case split
    # is based on position of leading one in spec.a[51:0].
    #
    caseEnumerate dnd_in1_leading1 -type leading1 \
    -parent denormal_normal_denormal -expr "spec.a(1)[51:0]"
    # b, c are denormal, a is normal
    #
caseBegin normal_denormal_denormal
caseAssume (spec.a(1)[62:52] != 11'h7ff)
caseAssume (spec.a(1)[62:52] != 11'h0)
caseAssume (spec.b(1)[62:52] == 11'h0)
caseAssume (spec.c(1)[62:52] == 11'h0)
    # further case splitting in above case. The case split
    # is based on position of leading one in spec.b[51:0].
    #
    caseEnumerate ndd_in2_leading1 -type leading1 \
    -parent normal_denormal_denormal -expr "spec.b(1)[51:0]"
}
```

12.3.2   Over-constrained Proofs

Before attempting any difficult proof, try to solve some constrained problems. This will give you an idea about the difficulty of the problem.

- ❖ Check whether addition is working properly by setting either `spec.a` or `spec.b` to 1.0.
- ❖ Check whether multiplication is working properly by setting `spec.c` to 0.0.

If these proofs are not converging it is a good idea to understand why these proofs are hard. Without getting convergence in simple cases it will not be possible to get convergence on the general case.

12.4   Running Hector Without Re-compiling

The following writeup assumes that you are trying to compare the two designs. If you are starting from scratch the order is as follows:

1. Compile *spec* design. This generates `spec.dfg` and puts it in work directory.
2. Compile *impl* design. This generates `impl.dfg` and puts it in work directory.
3. Compose two designs. This step reads `spec.dfg`, `impl.dfg`, and writes `all.dfg` in work directory.
4. Proof step: This step reads `all.dfg` present in work directory, assumptions/lemmas and generates proofs and also dispatches them for solving.
5. Debug step. If there are counterexamples in (4), you need to do this.

Steps (1) and (2) can happen in any order. There is a sequential dependency between all other steps. That is:

(1)/(2) -> (3) -> (4) -> (5)

Note that all these steps need not be done in a single Hector invocation. You can exit and then do remaining steps in another run. Some examples

- ❖ **Scenario 1**: You can do (1) and (2) and exit Hector shell. Then start Hector shell and do (3), then quit again. And do (4) and (5).
- ❖ **Scenario 2**: You can do (2) and exit Hector shell. Then start Hector shell and do (1), (3), (4), (5).
- ❖ **Scenario 3**: You can do (1), (2), (3) and exit Hector shell. Then start Hector shell and do (4), (5). Note that step (4) will use `all.dfg` that was written in work directory in previous Hector invocation.

If you are changing one of the designs say *spec* design. Then following steps need to done again in following order:

(1) -> (3) -> (4) ->(5)

You can exit Hector shell after some steps and do other steps in another run but the dependency shown above be preserved. Also you do not necessarily have to exit Hector shell. You can do this in same shell as well.

Similarly, if you are changing *impl* design following steps need to be done:

(2) -> (3) -> (4) -> (5)

If you are changing assumes/lemmas procedure following steps need to be done:

(3) -> (5)

☞ **Note**
Whenever Hector TCL script is changed you need to reload it. Currently, this can be done by exiting Hector shell. In releases I-2014.03-SP4 or greater you can call `reload_script` command and do not need to exit Hector shell.

12.5   Having Multiple SCV_WORK Directories

The writeup below assumes that the user is not using `-session` option to specify which work directory Hector should use.

When you start Hector shell it creates a work directory called `SCV_WORK` to write intermediate results.

In addition a lock file called *lock* is created in the `SCV_WORK` directory. This lock file is only removed if when the user quits the shell.

If you start another Hector invocation in the same directory (while the first Hector shell is still open), then Hector notices that `SCV_WORK` is locked so it creates another work directory `SCV_WORK_0`.

The problem that you will face in this second Hector run using `SCV_WORK_0` is that you need to do all steps again such as compiling two designs, compose step, and solve step. This is because an `SCV_WORK_0` created from scratch will not have `spec.dfg`, `impl.dfg`, `all.dfg` etc.

Another problem when multiple hector processes are running in the same directory they will be writing to same `hector.log` and `hector_command.log` by default. If you gave `-logfile <name>` option, then you can make sure two different logs are written.

If you start more Hector runs in the same directory Hector creates more work directories `SCV_WORK_1`, `SCV_WORK_2`, … and so on.

Note
Sometimes if a Hector shell is terminated due to signals or a tool crash it may not remove the *lock* file in `SCV_WORK` directory. Therefore, when you start Hector, it will again create a new `SCV_WORK_0` directory. You can address this by manually removing lock file from `SCV_WORK` directory and starting Hector again.

Given the confusion created by multiple `SCV_WORK` directories, it is recommended that you start only one Hector run in a given directory. If multiple `SCV_WORK`s do get created, you can remove them and run Hector again.

When Hector starts it does print a message to indicate which work directory is being used. Always make sure `SCV_WORK` is in use.

```
                    Hector

         High-Level Equivalence C++ to RTL

      Version ….

       Copyright (c) 2007-2013 by Synopsys, Inc.

                 ALL RIGHTS RESERVED


     This software and the associated documentation are

     confidential and proprietary to Synopsys, Inc. Your

     use or disclosure of this software is subject to the

     terms and conditions of a written license agreement

     between you, or your company, and Synopsys, Inc.


     Info SHL-001: Storing temporary files in directory 'SCV_WORK'.

     hector>
```

12.6   RTL Automatically Extracted Properties (AEPs) Example

The following example explains some of the RTL AEP checks. Consider following VHDL code.

```
 1  entity test_convert is
 2    port (
 3      i1 : in  integer;
 4      o1 : out natural;
 5      o2 : out integer range 7 downto 0;
 6      o3 : out integer range 0 to 7;
 7      );
 8  end test_convert;
 9
10  architecture rtl of test_convert is
11    function f (
12      arg : natural)
13      return natural is
14    begin  -- f
15      return arg;
16    end f;
17
18  begin  -- rtl
19    o1 <= i1;
20    p1: process (i1)
21      variable tmp : integer range 7 downto 0;
22    begin  -- process p1
23      tmp := i1;
24      o2 <= tmp;
25    end process p1;
26    o3 <= f(i1);
27  end rtl;
```

Following AEP checks are introduced:

Line 19: An integer is assigned to natural. The AEP checks that integer is not negative.

Line 23: An integer is assigned to a range variable. The AEP checks that integer falls within the range.

Line 26: First i1 is passed to a function that accepts natural. Secondly, the output of the function is assigned to a range variable. These two AEPs checks are combined into a single AEP check.

13 Appendix: Supported Math Library Functions

Hector is being shipped with its own math library that provides support for a subset of functions that can be found in the Linux system math-library libm.a (typically from glibc).

For more details on how use math library functions, see section "Support for Math Library Functions".

☞ **Note**
All functions in Hector's math library, except fmaf, have been formally compared against an openly available libm implementation. Note that the bit-patterns for non-canonical floating-point results like NaN or invalid numbers in extended precision may not match exactly between different libm implementations.

13.1   Supported Math Library Functions

The following math library functions are supported:

float	*ceilf(float arg)*
double	*ceil(double arg)*
long double	*ceill(long double arg)*
float	*floorf(float arg)*
double	*floor(double arg)*
long double	*floorl(long double arg)*
float	*truncf(float arg)*
double	*trunc(double arg)*
long	*doubletruncl(long double arg)*
float	*roundf(float arg)*
double	*round(double arg)*
long double	*roundl(long double arg)*
float	*rintf(float arg)*

double   *rint(double arg)*

long double   *rintl(long double arg)*

float   *nearintf(float arg)*

double   *nearint(double arg)*

long double   *nearintl(long double arg)*

float   *fabsf(float arg)*

double   *fabs(double arg)*

long double   *fabsl(long double arg)*

float   *fmaxf(float x, float y)*

double   *fmax(double x, double y)*

long double   *fmaxl(long double x, long double y)*

float   *fminf(float x, float y)*

double   *fmin(double x, double y)*

long double   *fminl(long double x, long double y)*

float   *fmaf(float x, float y, float z)*

float   *modff(float x, float* iptr)*

double   *modf(double x, double* iptr)*

long double   *modfl(long double x, long double* iptr)*

float   *std::copysign(float x, float y)*

double   *std::copysign(double x, double y)*

long double   *std::copysign(long double x, long double y)*

int   *sinff(float x)*

int   *sinf(double x)*

int   *isinfl(long double x)*

int   *std::isfinite(float x)*

int   *std::isfinite(double x)*

int   *std::isfinite(long double x)*

int	*std::isnormal(float x)*
int	*std::isnormal(double x)*
int	*std::isnormal(long double x)*
int	*isnanf(float x)*
int	*isnan(double x)*
int	*isnanl(long double x)*
int	*std::fpclassify(float x)*
int	*std::fpclassify(double x)*
int	*std::fpclassify(long double x)*