SystemVerilog Configurations and Tool Flow Using SCons (an Improved Make)

Don Mills

Microchip Technology Inc.
2355 W Chandler Blvd
Chandler, AZ 85224
don.mills@microchip.com

Dillan Mills

Microchip Technology Inc.
2355 W Chandler Blvd
Chandler, AZ 85224
dillan.mills@microchip.com

***Abstract* - This paper begins by reviewing the basics of SystemVerilog configurations and then explains how to use configurations to specify a unique file definition for a cell declared inside a design. With these definitions, the paper presents a set of model files and configuration settings that will work with the simulators available to the authors. The intent is to demonstrate what files and switches are needed to apply SystemVerilog configurations to a design across each simulator. Please note - this paper will not compare tools or features; it is simply a "how-to" paper.**

Makefiles were provided by the simulator vendors providing a baseline to start from for the paper. Even with the simple configurations presented, they grew complex and difficult to manage, with lots of repeated code. This provides a natural candidate for converting to an SCons script. The second part of this paper will show the basics of using SCons as an improved, modern replacement for Makefiles. For brevity, this paper will only provide inline example code for a single simulator. The examples will remain relatively generic, but the completed scripts will be capable of replicating the configuration Makefiles, and these replications will be presented next to the Makefiles in their respective appendix. A complete SCons example containing support for each simulator is also contained in the appendix.

TABLE OF CONTENTS

TABLE OF FIGURES AND EXAMPLES

I. Introduction

The motivation for this paper is twofold. First, a few years back one of the authors wanted to implement SystemVerilog configurations with one of the tool vendors but could not get all the pieces quite right. The author tried looking up documentation from the vendor and came up short. Next was a search of the internet only to have the search recommend several papers by none other than the author himself. These papers focused on the basics of configurations but gave no details about any tool-specific setup to use configurations. The authors felt publishing the hooks needed to set up configurations with the primary tool vendors would not be enough content to justify a paper for DVCon. However, while working with each tool vendor to get an initial configurations sample set up, each vendor provided a Makefile to run the tool commands. This leads to the second part of this paper which will introduce SCons as a modern and improved alternative to Make. One of the co-authors has been using SCons on current projects to manage tools and design flow, so it seemed logical to combine configurations (state what is to be implemented) with SCons (execute the implementation).

II. Configurations

Put simply, configurations define the source used for each instance in a design. For most designs, the configuration is self-implemented based on the design hierarchy and the files provided. However, there are some situations when the source used for a simulation might need to be switched. A common example is switching a behavioral model of an analog block with a spice model for use in an analog-digital mixed simulation, called an AMS or ADMS simulation. Another example is to swap out an RTL model of a component in a large system with its gate version. This is useful for very large designs where full system gate-level simulations can be painfully long[1].

A. *Configurations Past and Present*

Configurations have been part of Verilog from its inception, they were just called `` `ifdef `` macros [1]. The `` `ifdef `` macro has been widely used and is still a significant part of managing design compilation and configuration. They can be used at a high level to select which files to compile or at a granular level to select between multiple implementations of design models. `` `ifdef `` macros are one of the only ways to selectively configure ports for a module. This is needed to define power ports for UPF behavioral models that have multiple power ports, something that is not added automatically by UPF. UPF will auto-imply power ports if there is only one voltage defined for the model. But we digress, details for UPF behavioral modeling will have to be the subject of another paper. The `` `ifdef `` macro configuration model is applied during compilation, meaning every time a change is needed, the affected design must be recompiled. Also, note that the `` `ifdef `` macro configuration is coded as part of the model.

One of the features added in the Verilog 2001 specification is the `generate` statement [2]. `Generate` statements, in a broad sense, are like `` `ifdef `` macros where they can allow a design to selectively choose implementations of design models. `Generate` statements are configured using parameters that are set during the elaboration prior to simulation. `Generate` statements are used more to configure how the base RTL design will be implemented, such as port and bus sizes, rather than swapping versions of the design between simulations. `Generate` can be used to select between specific instantiations, but as noted above, this must be hard-coded and embedded in the design. There are lots of usage models that can be applied to `generate` statements that are beyond the scope of this paper.

The differences between `generate` statements and `` `ifdef `` macros are how and when selections are made. `` `ifdef `` macros are managed by either embedding `` `define foo `` in the compiled code or with a `+define+foo` added to the compilation arguments. `Generate` statements are updated during elaboration and are configured using parameters. Using `generate` allows for changes to be implemented without recompiling.

Both `` `ifdef `` and `generate` provide unique features not supported by their counterpart. Since Verilog/SystemVerilog already has these two ways of configuring a design, what does the language configuration feature provide that is not already present? First, for both `` `ifdef `` and `generate` configurations, the designer must embed these selections in the code. Second, and more important, the language configuration model allows for defining the source used for specific

[1] These gate models need a wrapper to account for required setup time between RTL to gate data paths. Feel free to contact the authors regarding this concept if needed.

instances of a component, and this language configuration model is done external to the design rather than embedded in the design.

There are some situations where `ifdef macros are the only solution. For example, as previously noted, `ifdef macros are used within components to enable additional ports for UPF behavioral models that have multiple power ports. At the system level, `ifdef macros are a reasonable solution to swap the full DUT from RTL to gates in the TB. However, this approach can have limited flexibility when only part of a DUT is desired to be switched to gates. This is something that is done with very large designs.

Formal Verilog configurations were added as part of Verilog 2001 and then updated/enhanced with the SystemVerilog generations of the language [2, 3]. The primary usage model of configurations is to externally select the source model for instances in the design.

B. *SystemVerilog Configurations*

Before discussing SystemVerilog configuration details, it is needful to mention what the focus of this part of the paper is, I.E. what is and what is not in this paper. The motivation of this paper is to show a working model with tool switches to use SystemVerilog configurations. This paper is not going to highlight all the features of configurations such as using configurations to set parameters. Nor will this paper diagram all the specific tool hooks and features. Please refer to section 33 of the IEEE 1800-2017 specification for all the features and details of SystemVerilog configurations [4]. This paper will use a set of model files and configuration settings with a few common variations that will work with all three simulators. This will allow the paper to show a setup that works with the tools discussed in this paper. The objective is to show what files are needed and which switches are needed to apply SystemVerilog configurations to a design using any of the simulators. This paper will not compare tools or features; it is simply a "how-to" paper.

The design used for this paper is diagrammed in Figure 1, below. The code for module `top` and module `dual_adder` is included after the diagram in Example 1 and Example 2.

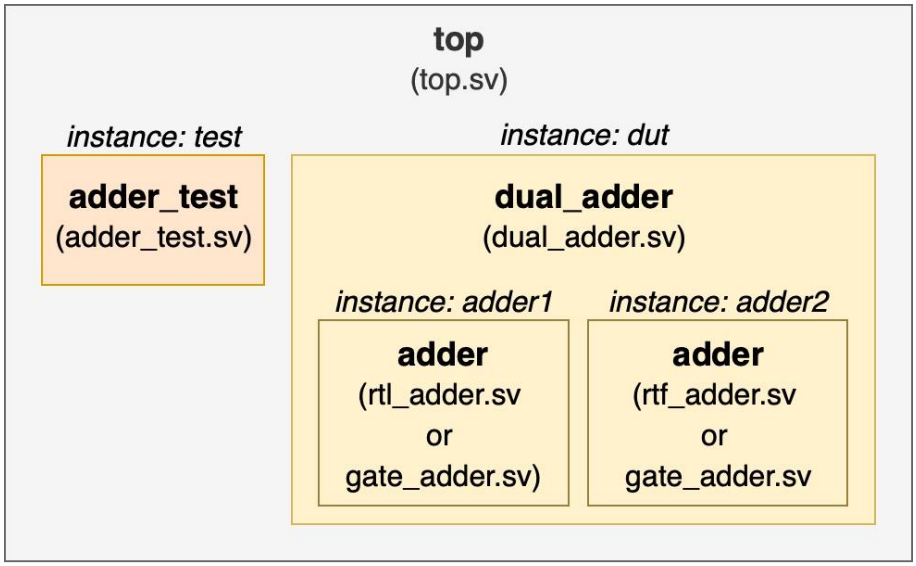

Figure 1. Example design used in this paper

In Figure 1, the text in bold is the RTL module name for each file represented by each block.

```
1.  module top;
2.    timeunit 1ns/1ns;
3.
4.    logic  a, b, ci;
5.    logic  sum1, co1;
```

```
6.    logic  sum2, co2;
7.
8.    adder_test test (.*);
9.    dual_adder dut  (.*);
10.
11. endmodule:top
```

Example 1. top.sv

```
1.  module dual_adder (
2.    input  var a, b, ci,
3.    output var sum1, co1,
4.    output var sum2, co2);
5.
6.    timeunit 1ns/1ns;
7.
8.    adder adder1 (.*,
9.                  .sum(sum1),
10.                 .co (co1 ));
11.
12.   adder adder2 (.*,
13.                 .sum(sum2),
14.                 .co (co2 ));
15.
16. endmodule:dual_adder
```

Example 2. dual-adder.sv

To use SystemVerilog configurations, there are two primary definitions that need to be declared, a libmap and a configuration declaration. These declarations are typically declared in separate files. For this paper, these definitions will be defined in files named `configs.sv` and `libmap.sv` (or `libmap_rtl.sv`). The libmap declaration specifies which files are associated with which library. The config file will reference files from the library definitions for use in the current simulation. As part of experimenting with library options for this paper, the authors used several different libmaps, but only two are detailed and shown in this paper.

```
1.  library rtlLib  top.sv,
2.                  adder_test.sv,
3.                  rtl_adder.sv,
4.                  dual_adder.sv;
5.
6.  library gateLib gate_adder.sv;
```

Example 3. libmap.sv

The libmap detailed in Example 3 is a common, simple libmap definition declaring all the primary code (top, test, and design) in one library, `rtlLib` in this case. A gate version of the adder is in a second library called `gateLib`. The intention of this libmap is to provide a gate adder source model for a replacement to the RTL adder if the configuration dictates.

An arguably better approach is to divide the design files into three libraries: a top/test library, an RTL library, and a gate library, as shown in Example 4. This alternative style is preferred by the authors as it clearly separates the code into library groups top/test, RTL, and gate which in turn allows for more flexibility and readability.

```
1.  library rtlLib  rtl_adder.sv,
2.                  dual_adder.sv;
3.
4.  library gateLib gate_adder.sv,
5.                  gate_adder_alt.sv;
6.
7.  library testLib top.sv,
8.                  adder_test.sv;
```

Example 4. libmap_rtl.sv

Along with the library mapping definitions, a configuration needs to be declared. The following three config examples show a base model configuration that will be used to extend and swap out instance definitions later in this paper. The configuration declaration begins and ends with the keywords `config` and `endconfig`, respectively. The `config` statement will declare the name of the configuration as shown in line 1 of Example 5.

```
1.  config rtl_config;
2.    design   rtlLib.top;
3.    default liblist rtlLib;
4.  endconfig
```

Example 5. rtl_config based on Example 3. libmap.sv

The `design` statement (line 2) declares the top module of the overall simulation environment and, as shown, may also specify the library the top module resides in. A config can only have one `design` statement but may have multiple top files listed. The `default liblist` specifies the library or libraries to be searched to find the components of the design.

The configuration in Example 5 declares that all the files used for the design are in the `rtlLib`. The next two configurations are based on using the "preferred" `libmap_rtl.sv` where the top module and test code is in the `testLib`. In Example 6 and Example 7, the `design` statement declares the top module "`top`" resides in the `testLib`.

```
1.  config rtl_config1;
2.    design   testLib.top;
3.    default liblist rtlLib;
4.  endconfig
```

Example 6. rtl_config1 based on Example 4. libmap_rtl.sv

An interesting observation of Example 6 is that `testLib` is not listed in the `default liblist`. The simulation vendors must infer the `testLib` from the design statement as a library to search for components not found in `rtlLib`. In this case, the `adder_test.sv` would be the file in the `testLib` library. Only two of the vendors do this. To model a configuration that works cleanly with all three vendors and to remove the implied library for searching, the authors recommend specifying the library in `default list` as shown in Example 7.

```
1.  config rtl_config2;
2.    design   testLib.top;
3.    default liblist rtlLib testLib;
4.  endconfig
```

Example 7. rtl_config2 based on Example 4

It must be noted here how annoying this `default liblist` format is. This list is NOT comma-delimited, but space-delimited. How is this Verilog/SystemVerilog????? Got to love design by committee!!!

As noted previously, the primary purpose/value of SystemVerilog configurations is to substitute instances without the need to modify code or embed selection code in the design files. For the design described in this paper, the following examples will show two ways to use a configuration to swap the module an instance of the adder uses in the DUT. Though there are other ways to swap out a module using configurations, the methods shown in this paper are recommended by the authors.

```
1.  config cell_config;
2.    design      testLib.top;
3.    default     liblist rtlLib testLib;
4.    cell adder liblist gateLib;
5.  endconfig
```

Example 8. cell_config with liblist

The configuration modeled in Example 8 declares the `top` module is found in library `testLib`. The instantiated components in the design except for the `adder` will be found in the libraries specified in the `default liblist rtlLib` and `testLib`. All instances of `cell adder` will use the module `adder` found in the `gateLib`.

```
1.  config cell_config1;
2.    design testLib.top;
3.    default liblist rtlLib testLib;
4.    cell adder use gateLib.adder;
5.  endconfig
```

Example 9. cell_config1 with use gateLib.adder

```
1.  config cell_config2;
2.    design testLib.top;
3.    default liblist rtlLib testLib;
4.    cell adder use gateLib.adder_alt;
5.  endconfig
```

Example 10. cell_config2 with use gateLib.adder_alt

In the configurations shown in Example 9 and Example 10, the source code for the `cell adder` is from the `gateLib`. By specifying the "`use`" keyword, the configuration can call a specific module from this library. This allows for multiple definitions of a model designated by unique module names to be selected for use. In these examples, there is a gate module `adder` and a gate module `adder_alt` in the `gateLib`. Utilizing the "`use`" keyword, a specific module is selected for the `cell`. The module names selected can but do not have to match the cell name designated in the DUT (in this case `adder`).

So what is the difference between line 4 in Example 8 and line 4 in Example 9 or Example 10?

```
     Example 8,  Line 4     cell adder liblist gateLib;
```

and

```
     Example 9,  Line 4     cell adder use gateLib.adder;
     Example 10, Line 4     cell adder use gateLib.adder_alt;
```

The first style using `liblist` indicates that `adder` comes from the `gateLib` and will match a model name `adder`. The second using the "`use`" designator indicates a specific module from the `gateLib` to be used. If there is only one `adder` module in the library for use with `cell adder`, then `liblist` is sufficient.

The next two examples are a variation of the previous examples. The difference is that instead of replacing all instances of `cell adder`, these examples will replace a specific instance of `cell adder`. The selection method to designate the source code for the specific instance is the same as with the previously referenced examples, Example 8 through Example 10, using `liblist` or `use` to select the source code from a specific library.

```
1.  config inst_config;
2.    design   testLib.top;
3.    default  liblist rtlLib testLib;
4.    instance top.dut.adder2 liblist gateLib;
5.  endconfig
```

Example 11. inst_config with liblist

```
1.  config inst_config1;
2.    design testLib.top;
3.    default liblist rtlLib testLib;
4.    instance top.dut.adder2 use gateLib.adder;
5.  endconfig
```

Example 12. inst_config1 with use gateLib.adder

```
1.  config inst_config2;
2.    design testLib.top;
3.    default liblist rtlLib testLib;
6.    instance top.dut.adder2 use gateLib.adder_alt;
4.  endconfig
```

Example 13. inst_config2 with use gateLib.adder_alt

Note the difference between these config examples and Example 8 through Example 10 is the keyword `instance` rather than `cell.` It is important to note that based on the code setup for this paper, one of the vendors requires the `configs.sv` be compiled last for the `use` clause to work. This is not a big deal and there may be a vendor switch to remove this requirement but at the time of writing this paper, a switch has not yet been found.

C. *Tool Specifics to Compile and Simulate Configurations*

The primary point of the paper is to document how to simulate configurations with the SystemVerilog simulators. The three simulators available to the authors are from Cadence, Mentor, and Synopsys (alphabetical order.) The intent of the paper is not to compare or contrast these vendors, but simply to show the setup for each tool to simulate configurations.

1) Cadence Configuration Setup

The authors used the Cadence Xcelium simulator to compile and simulate the configuration experiments for this paper. The example below shows the Xcelium `xrun` switches used.

```
1.  xrun -libmap libmap.sv     -compcnfg configs.sv -f source_code_cadence.f -top rtl_config   -exit
2.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top rtl_config1  -exit
3.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top rtl_config2  -exit
4.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config  -exit
5.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config1 -exit
6.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config2 -exit
7.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config  -exit
8.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config1 -exit
9.  xrun -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config2 -exit
```

Example 14. Cadence Xcelium compile and simulate

Notes regarding the `xrun` command line:

- The `-libmap` switch specifies the configuration library for `xrun` compilation/simulation.
- The `-compcnfg` switch specifies the configuration file used for `xrun` compilation/simulation.
- The `configs.sv` should not be included in the source code `.f` file. If it is, the simulation will still work but will issue lots of warnings.
- The `-top` switch specifies which config from the `config.sv` file will be used to specify the top unit for simulation.

For reference, the `source_code_cadence.f` file is listed in Example 15.

```
1.  adder_test.sv
2.  dual_adder.sv
3.  gate_adder.sv
4.  gate_adder_alt.sv
5.  rtl_adder.sv
6.  top.sv
```

Example 15. source_code_cadence.f file

The full Makefile used for the testing with Cadence Xcelium is listed in Appendix A: Makefile.cadence.

2) Mentor Configuration Setup

The Mentor Questa setup uses a two-step flow. The first step is to compile, and the second step is to simulate.

The compile step specifies the `libmap` using the `-libmap` switch with the `vlog` command.

```
1.  vlog -libmap libmap.sv     -f source_code.f
2.  vlog -libmap libmap_rtl.sv -f source_code.f
```

Example 16. Questa compile

The only difference between the source code file `source_code.f` used here and the one used for the Cadence command line is that this source code file includes the `configs.sv` file (compare Example 15 and Example 18). Also, in this source code file (Example 18), the `configs.sv` file is listed last as noted in the discussion above regarding the "`use`" option in a configuration.

The second step is to simulate the design. The Questa command line to simulate the compiled design is shown in Example 17.

```
1.  vsim  rtl_config      -c    -do "run -all; exit"
2.  vsim  rtl_config1     -c    -do "run -all; exit"
3.  vsim  rtl_config2     -c    -do "run -all; exit"
4.  vsim  cell_config     -c    -do "run -all; exit"
5.  vsim  cell_config1    -c    -do "run -all; exit"
6.  vsim  cell_config2    -c    -do "run -all; exit"
7.  vsim  inst_config     -c    -do "run -all; exit"
8.  vsim  inst_config1    -c    -do "run -all; exit"
9.  vsim  inst_config2    -c    -do "run -all; exit"
```

Example 17. Questa simulation

The argument to `vsim` is the top design, which for a configuration is defined in the configuration itself. Thus, for Questa `vsim`, the command is simply `vsim` followed by the selected configuration name.

Notes regarding this Example:

- The `-c` option indicates command line rather than interactive mode.
- The `-do` will run all and exit automatically with the `-c` option. These commands can be placed in a file as shown in Appendix D: Full SCons and Configurations Example, sub-section K: run_all.sim.

The source code file used for the compile step is shown in Example 18.

```
1.  adder_test.sv
2.  dual_adder.sv
3.  gate_adder.sv
4.  gate_adder_alt.sv
5.  rtl_adder.sv
6.  top.sv
7.  configs.sv
```

Example 18. source_code.f file

Note the source code file shown in Example 18 is used by both the Mentor Questa setup and the Synopsys VCS setup which is detailed in the next section.

The full Mentor Makefile used for this paper is in Appendix B: Makefile.mentor.

3) Synopsys Configuration Setup

The Synopsys VCS flow is also a two-step flow separating the compilation from the simulation.

```
1.  vlogan -full64 -diag libconfig -sverilog -libmap libmap.sv     -f source_code.f
2.  vlogan -full64 -diag libconfig -sverilog -libmap libmap_rtl.sv -f source_code.f
```

Example 19. vcs compile

The VCS `vlogan` compile command uses the following two switches to compile a configuration for simulation. Note the `configs.sv` is included in the `source_code.f` file.

- `-diag libconfig`
- `-libmap libmap_rtl.sv`

Example 20 shows the `vcs` simulation command for simulating with a selected configuration.

```
1.  vcs -full64 -diag libconfig -debug_access -R rtl_config   -ucli -i run_vcs.do
2.  vcs -full64 -diag libconfig -debug_access -R rtl_config1  -ucli -i run_vcs.do
3.  vcs -full64 -diag libconfig -debug_access -R rtl_config2  -ucli -i run_vcs.do
4.
5.  vcs -full64 -diag libconfig -debug_access -R cell_config  -ucli -i run_vcs.do
6.  vcs -full64 -diag libconfig -debug_access -R cell_config1 -ucli -i run_vcs.do
7.  vcs -full64 -diag libconfig -debug_access -R cell_config2 -ucli -i run_vcs.do
8.
9.  vcs -full64 -diag libconfig -debug_access -R inst_config  -ucli -i run_vcs.do
10. vcs -full64 -diag libconfig -debug_access -R inst_config1 -ucli -i run_vcs.do
11. vcs -full64 -diag libconfig -debug_access -R inst_config2 -ucli -i run_vcs.do
```

Example 20. vcs simulate

Notes regarding the vcs command:

- The selected configuration is noted after the "`-R.`"
- The `-diag libconfig` is optional but gives good information.
- The `-debug_access` is optional.

The `run_vcs.do` file referenced in the `vcs` simulation command is listed in Example 21.

```
1. run
2. exit
```

Example 21. run_vcs.do file

Finally, a library reference file is also needed for `vcs` to run. This file is called `synopsys_sim.setup` and contains the list of libraries declared in the `libmap` files. This file resides in the directory where the `vcs` simulation is run from and is referenced by the VCS tool.

```
1. WORK    > DEFAULT
2. DEFAULT : work
3. rtllib  : rtlLib
4. gatelib : gateLib
5. testlib : testLib
```

Example 22. synopsys_sim.setup file

The Synopsys Makefile used for the experiments in this paper is in Appendix C: Makefile.synopsys.

D. *Advanced Concepts for a Future Paper*

Using the library maps and the configuration declarations for the example design, the adder module can successfully be configured to use either the RTL version or the gate-level version of the module across each tested simulator. The examples also showed how to select an alternative gate-level module version that is named different from the cell instantiation. Other features not discussed in the paper include:

- Setting parameter in configuration
- Setting a hierarchical configuration for a subsection of a design
- Nested configurations
- Configurations to specify details of generic interconnects

III. SCons

The remainder of this paper discusses possible approaches for extending SCons to compile and simulate SystemVerilog code. The Makefiles used in the previous section work, but even with the simple configuration presented, they have already grown complex and difficult to manage, with lots of repeated code. This provides a natural candidate for converting to an SCons script. For brevity, inline example code will only be provided for a single vendor. A complete example using each major vendor is contained in Appendix D: Full SCons and Configurations Example. The examples will remain relatively generic, but the completed scripts will be capable of replicating the configuration Makefiles, and these replications will be presented next to the Makefiles in their respective appendix. This section is based on a similar tutorial on the SCons wiki but has been heavily adapted to be specific to SystemVerilog and its nuances [5].

A. *What is SCons?*

SCons is an open-source software construction tool [6]. It is an improved, cross-platform substitute for GNU Make. Some of the benefits include:

- Configuration files are Python scripts, providing the full power of the Python programming language to solve build problems.
- Automatic dependency tracking.
- Detects source changes by MD5 signature (optionally, can be configured to detect by traditional timestamp).
- Improved support for parallel builds (like `make -j`, but works regardless of directory hierarchy).
- Great support for hierarchical builds that can match the hierarchy of a chip development project.
- Designed to be cross-platform, so a single script can be used on Linux® and on Windows® if needed.
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt, and SWIG, and building TeX and LaTeX documents. Support for additional languages is straightforward.
- Uses a self-contained environment configuration separate from the user's Unix environment which makes debugging more consistent – anybody who runs the script will have the same set of environment variables and flags set.
- Unifies compiling across a team by extracting all common functionality into a base file that can be imported into each individual project.

The SCons executable is typically installed system-wide and build configurations are placed in a file named `SConstruct`. When the `scons` command is invoked in a directory, it will search for an `SConstruct` file by default.

B. *A Simple SystemVerilog Builder, and its Evolution to a Tool*

SCons uses Python objects named builders to compile software across different programming languages. Each built-in language (listed above) has a builder capable of compiling the code for that language. SCons can be extended with additional builders to add support for more languages. This section will outline the process of creating a builder for SystemVerilog.

1) Start with the Command Line

Starting with a single input file, `rtl_adder.sv`, the compile command "`vlog rtl_adder.sv`" creates a `work` library, which contains several collateral files. SCons needs a target file to latch onto, and the file `work/_lib.qdb` is a good choice for this[2]. This command makes a good starting candidate for implementing in SCons, starting with a `Command()` action.

[2] A Directory can be used as a target in SCons, but because there is no data to monitor, the target will be rebuilt every time SCons is invoked, which removes SCons' ability to automatically track dependencies. Any file generated by the command is a good candidate for a target.

2) Command Wrapper

```
1.  import os
2.  env = Environment(ENV={'PATH': os.environ['PATH']})
3.  env.Command('work/_lib.qdb', 'rtl_adder.sv', 'vlog $SOURCE')
```

Example 23. A simple command wrapper and SConstruct file

This Python code does two things. First, it sets up the construction environment and copies the system path into it, so the tools are available to use[3]. Second, it creates a `Command()` builder which will execute the vlog command when `scons` is invoked at the command line. The first major benefit of SCons can be seen with this configuration: file modification detection. Invoking `scons` will compile rtl_adder.sv, but only if it detects a change to the file since the last time `scons` was invoked. If nothing has changed, `scons` will report that the target is up to date and exit. If the project is a single file, this is a great solution, but it is not very reusable as both the work library name and the source file are hard-coded.

3) Simple Builder

One step above a `Command()` action is a `Builder()`. A Builder lets us pass in the command line argument as an action, then call it at a later point with the source files.

```
1.  import os
2.
3.  vlogbld = Builder(
4.      action='vlog $SOURCES',
5.      suffix='.qdb',
6.      src_suffix='.sv')
7.
8.  env = Environment(
9.      ENV = {'PATH' : os.environ['PATH']},
10.     BUILDERS = {'Vlog': vlogbld})
11.
12. env.Vlog('work/_lib', ['top', 'adder_test', 'rtl_adder'])
```

Example 24. A simple builder and SConstruct file

This builder specifies a `suffix` and a `src_suffix`, so they can be left off the call to the builder (this is not required). Within the call to the `Vlog` builder, test files were added by the authors to the source list as well, so now a change to any one of them will cause `scons` to recompile. The builder is more portable now, but it still needs to be manually pasted into each `SConstruct` that needs it.

4) First Version of a Tool

Tools are Python modules or packages that SCons uses to modify an environment. They can alter environment variables, add builders, or otherwise modify an environment to prepare for any required tasks. They are a convenient location to store SCons code that will be shared by a team. The only SCons requirement to be a proper tool is to define two functions, `exists()` and `generate()`. The first method is used by SCons to determine if all the conditions to use the tool are met. This method can be used to check that the needed executables are available in the current `PATH`. The `generate()` method is what modifies the Environment.

```
1.  from SCons.Script import *
2.
3.  ############################
4.  # Builders
5.  ############################
6.  _vlog_builder = Builder(
```

[3] SCons recommends hard-coding every necessary variable into the construction environment to guarantee portability instead of copying variables from the system environment. The authors recommend copying from the system environment when it would be difficult to hard code the value needed.

```
7.      action='vlog $SOURCES',
8.      suffix='.qdb',
9.      src_suffix='.sv')
10.
11.
12. def generate(env):
13.     """Add Builders and construction variables to the Environment."""
14.
15.     env['BUILDERS']['Vlog'] = _vlog_builder
16.
17.
18. def exists(env):
19.     return 1
```

Example 25. An initial framework for an SCons tool

This tool provides the same functionality as the builder previously implemented. Tools are saved in a different location. By default, SCons will look in the folder `site_scons/site_tools/` for extra tools. Create the folder hierarchy `site_scons/site_tools/questa` and place the above code in a file named `__init__.py` (this is how Python declares packages). To use the tool in an `SConstruct` file, modify it like this:

```
1.  import os
2.
3.  env = Environment(
4.      ENV = {'PATH' : os.environ['PATH']},
5.      TOOLS = ['questa'])
6.
7.  env.Vlog('work/_lib', ['top', 'adder_test', 'rtl_adder'])
```

Example 26. SConstruct file using the tool defined in Example 25

This directory could be shared in version control and made available to each project. This would allow each team member to use the same compile commands, which will reduce bugs.

C. *Prettying It Up*

There are still some issues with this tool: the work library name is still hard-coded, there is no way to specify additional command-line arguments, and it would be nicer to be able to use the `run.f` file list instead of manually specifying each source file. Additionally, the important follow-up step of simulating the design after compiling is still missing.

1) Scanners

By default, SCons will only track changes for the files specified in the source list. This means that if the project source files are contained in a `.f` file list, SCons will not monitor them for changes. However, SCons has a class named `Scanner` that can be used to process files for additional dependencies. A Scanner object can be configured to search `.f` files and `.sv` files (packages, libraries) for additional dependencies and allow SCons to track every SystemVerilog file in the project automatically. This is one feature that is an improvement from Make.

```
12. ##############################################################################
13. # SCANNERS:
14. #   Scanners to parse .f files and .sv files for dependencies
15. ##############################################################################
16. f_re = re.compile(r'^-f\s+(\S+\.f)$', re.M | re.I)
17. sv_re = re.compile(r'^(/?[^/+]\S+\.s?vh?)$', re.M | re.I)
18. include_re = re.compile(r'^\s*`include\s+"(\S+)"$', re.M | re.I)
19.
20. def ffile_scan(node, env, path, arg=None):
21.   contents = node.get_text_contents()
22.
23.   sv_files = sv_re.findall(contents)
24.   sv_files = [sv.strip() for sv in sv_files]
25.   f_files = f_re.findall(contents)
```

```
26.
27.   while f_files:
28.     for f in f_files:
29.       # The following line is used to expand any environment variables
30.       # in the filepath using the custom SCons environment. This will
31.       # catch any variables declared in the SConstruct.
32.       ef = subprocess.check_output('echo ' + f, shell=True,
    env=env['ENV']).strip()
33.       if os.path.isfile(ef):
34.         current_dir = os.path.dirname(ef) + '/'
35.         contents = env.File(ef).get_text_contents()
36.
37.         sv_files.extend([(current_dir + x.strip()) for x in
    sv_re.findall(contents)])
38.         f_files.extend([(current_dir + x.strip()) for x in
    f_re.findall(contents)])
39.         sv_files.append(str(ef))
40.       f_files.remove(f)
41.
42.   results = []
43.   for f in env.File(sv_files):
44.     results.extend(svfile_scan(f, env, path, arg))
45.
46.   return results
47.
48. def svfile_scan(node, env, path, arg=None):
49.   contents = node.get_text_contents()
50.   includes = include_re.findall(contents)
51.
52.   starting_dir = str(node.dir) + '/'
53.
54.   if includes == []:
55.     return [node]
56.
57.   results = [str(node)]
58.   for inc in includes:
59.     if os.path.exists(starting_dir + inc):
60.       results.append(starting_dir + inc)
61.
62.   return env.File(results)
63.
64. svscan = Scanner(
65.   name='svfile',
66.   function=svfile_scan,
67.   argument=None,
68.   skeys=['.v', '.vh', '.sv', '.svh'])
69.
70. fscan = Scanner(
71.   name='ffile',
72.   function=ffile_scan,
73.   argument=None,
74.   skeys=['.f'])
```

Example 27. .f and .sv file scanner objects and functions

In the `SCANNERS` section, there are two simple scanners. The first one, `ffile_scan`, searches a `.f` file for other `.f` files and for `.sv` files. It will continue looping through any `.f` file it finds until it is left with a list of only `.sv` files. It then passes the list of `.sv` files over to `svfile_scan`, which searches each file for additional files that been included

using `include. When the scan completes, scons will have a full dependency list of every file in the design[4]. If any file changes, calling scons will cause a re-compile. Otherwise, scons will report that the design is up to date.

2) Environment Variables and Pseudo-Builders

```
76. ##############################################################################
77. # BUILDERS:
78. ##############################################################################
79. ### vlog
80. def generate_vlog(source, target, env, for_signature):
81.   action = [env['VLOG']]
82.   for s in source:
83.     if os.path.splitext(str(s))[1] == '.f':
84.       action.append('-F')
85.     action.append(str(s))
86.
87.   action.extend(['-work ${SIM_DIR}${WORK}'])
88.   action.extend(env['VLOG_ARGS'])
89.   action.extend(['-l $TARGET'])
90.
91.   return ' '.join(action)
92.
93. def Vlog(env, target, source, *args, **kw):
94.   """A pseudo-Builder wrapper for the vlog executable."""
95.
96.   _vlog_builder = Builder(generator=generate_vlog, suffix='.log')
97.
98.   result = _vlog_builder.__call__(env, target, source, **kw)
99.   env.Clean(result, ['.done', '${SIM_DIR}${WORK}'])
100.
101.  return result
102.
103.### vsim
104.def generate_vsim(source, target, env, for_signature):
105.  action = [env['VSIM']]
106.
107.  action.extend(['-lib ${SIM_DIR}${WORK}'])
108.  action.extend(env['VSIM_ARGS'])
109.  action.extend(['-appendlog -l $TARGET'])
110.
111.  return ' '.join(action)
112.
113.def Vsim(env, target, source, *args, **kw):
114.  """A pseudo-Builder wrapper for the vsim executable."""
115.
116.  _vsim_builder = Builder(generator=generate_vsim, suffix='.log')
117.
118.  result = _vsim_builder.__call__(env, target, source, **kw)
119.  env.Clean(result, ['.done'])
120.
121.  return result
```

Example 28. Tool builders and pseudo-builders for compiling and simulating

In the BUILDERS section, there is a second builder to handle simulating. The builders now use a generator function to create an action on the fly instead of using a hard-coded action. This gives a lot of flexibility in what command the builder uses. In this example, the builder loops through all the source files passed into the call to the builder and add

[4] These scanners are provided as an example - the regex used by them may not be comprehensive. For typical configurations, they will work, but if the user has anything complicated, the configurations might need to be adjusted to match the use-case.

a `-F` flag if it is a `.f` file. This way, `.sv` files and `.f` files can be mixed in the source list. The hard-coded work library name was also replaced with an environment variable and added a few other environment variables to help make the command generic and configurable by the user:

- The `env['VLOG']` variable points to the `vlog` executable. This could be used to configure for specific versions of `vlog` or to swap for an alternate tool (`valog`, for example).
- In the `generate_vlog()` and `generate_vsim()` methods, the work library is added using the `${SIM_DIR}${WORK}` variable combination. The target now points to the log file that will be generated by each command.
- The `env['VLOG_ARGS']` and `env['VSIM_ARGS']` variables are available so the user can add any additional arguments needed to the commands.
- The `${SIM_DIR}` variable allows the option to place all generated files in a separate directory. In this example, it will move the work library and the log files to a separate directory.

There are three different ways to access environment variables. If needed outside of a string, looking up the variable in the `env` dictionary is appropriate (`env['VLOG']`). Inside a string, if the variable has white-space both before and after it, the variable can be accessed using just a `$` sign - `'$VLOG'` would return the same thing. If the variable is used within another part of the string, wrapping the variable with curly braces allows it to be replaced with its value properly (`${SIM_DIR}vlog.log`). Additionally, this allows the option of special modifiers to be used to access different parts of the variable such as `dir` for the directory of the file and `file` for just the file name. The `scons` man page details several other modifiers [7].

The actual builders for `Vlog` and `Vsim` are wrapped in pseudo-builders. A pseudo-builder allows for extra functionality to be added to a builder such as modifying the source or target list, adding additional dependencies, or handling side-effects. The full power of Python is available within a pseudo-builder. In this example, each pseudo-builder adds a call to `Clean()`. This allows for specifying additional files that should be removed when `scons -c` is called at the command line. By default, `scons` will remove the target file only. Now it will remove the `work` directory and the `.done` file in addition to the log files.

3) Generate Method

```
124.##############################################################################
125.# TOOL FUNCTIONS:
126.#   generate() and exists() are required by SCons
127.##############################################################################
128.def generate(env):
129.  """Add Builders and construction variables to the Environment."""
130.
131.  env['VLOG'] = env.WhereIs('vlog')
132.  env['VSIM'] = env.WhereIs('vsim')
133.
134.  env.Append(SCANNERS=[svscan, fscan])
135.
136.  env.SetDefault(
137.    SIM_DIR='./',
138.    WORK='work',
139.    VLOG_ARGS=[],
140.    VSIM_ARGS=[],
141.  )
142.
143.  env.AddMethod(Vlog, "Vlog")
144.  env.AddMethod(Vsim, "Vsim")
145.
146.def exists(env):
147.  return True
```

Example 29. Tool required functions: generate() and exists()

In the `generate()` method, the environment is set up:

- This is a good location to set up the executable variables.

- By appending the two scanners created in Example 27 to the environment, they will automatically be called when `scons` detects a source file matching the filetypes they specified in their `skeys` list.
- The `SetDefault()` method allows for specifying default values for any needed environment variables. A user can then overwrite them as needed and use the defaults the rest of the time. A variable can also be directly set (as was done for the executable variables), which will not allow the user to change it.
- To add a pseudo-builder to the environment, use the `AddMethod()` function instead of assigning the method to the `env['BUILDERS']` dictionary.

4) Aliases, Dependencies, and AlwaysBuild

The `SConstruct` file now looks like this:

```
1.  import os
2.
3.  env = Environment(
4.      ENV = {'PATH' : os.environ['PATH']},
5.      TOOLS = ['questa'],
6.      SIM_DIR = 'sim/',
7.      WORK = 'work_lib',
8.  )
9.
10. vlog = env.Vlog('${SIM_DIR}vlog.log', 'run_mentor.f')
11. env.Alias('compile', [vlog])
12.
13. vsim = env.Vsim('${SIM_DIR}vsim.log', 'compile')
14. env.Alias('sim', [vsim])
15. AlwaysBuild('sim')
```

Example 30. SConstruct file using the tool defined in Example 27 through Example 29

In the `Environment()` initializer, alternate variables are set for `SIM_DIR` and `WORK`. The call to `Vlog` can now use these variables in its target. The source file list has been replaced with the `.f` file. Now that there are multiple targets, it is convenient to define an `Alias()`. Aliases allow for one or more targets to be called using a single target alias name. That alias can then be used as a dependency in future targets. In this example, the defined alias `'compile'` points to the `Vlog` target. The `Vsim` target uses `'compile'` as its dependency, which will ensure the design is always compiled before simulating. A `'sim'` alias is declared, then `AlwaysBuild()` is called on it to ensure a simulation every time `scons` is invoked, instead of stopping after the first time. Alternatively, a phony target name could be used that doesn't correlate to any generated files (such as `'simulate'`). Because nothing would be created matching the target, `scons` would attempt to build it every time it is called. The issue with this approach is if a file or folder does exist with the same name, `scons` will exit with an error, or delete the file. For example, if the target had been set to a phony `'sim'`, there would have been a conflict with the `SIM_DIR` variable where the work library is placed. It is better to be explicit about the target, then use `Alias()` to make it more user-friendly.

At this point, a call to `scons` will compile the design then start a simulation. Repeated calls to `scons` without modifying any of the source code will just simulate. A target can also be called explicitly: `scons compile` will not simulate with either compile or report up to date.

5) Hierarchical Builds

The tool is in a good state for individual module-level projects, but it has room for improvement at the chip level. SCons supports hierarchical builds, which can significantly reduce compile time at the chip level by compiling multiple modules simultaneously and by only recompiling what has been updated. To do this, SCons using a file (typically named SConscript) placed in a project subdirectory that acts as an extension of the construction environment. Within the SConstruct file, the compile target can be replaced with an SConscript function call like so:

```
1.  import os
2.
3.  env = Environment(
4.      ENV = {'PATH' : os.environ['PATH']},
5.      TOOLS = ['questa'],
6.      SIM_DIR = 'sim/',
7.      WORK = 'work_lib',
8.  )
9.
10. SConscript(['src/SConscript'], exports='env')
11.
12. vsim = env.Vsim('${SIM_DIR}vsim.log', 'compile')
13. env.Alias('sim', [vsim])
14. AlwaysBuild('sim')
```

Example 31. SConstruct file using an SConscript file for the compile step

This registers the file `src/SConscript` and will include its content when building. The contents of that file are the compile targets removed from the `SConstruct` file:

```
1.  Import('env')
2.
3.  vlog = env.Vlog('${SIM_DIR}vlog.log', 'run_mentor.f')
4.  env.Alias('compile', [vlog])
```

Example 32. The SConscript file referenced in Example 31

This concept can be significantly expanded such that a chip project could have a separate SConscript file for each submodule that is responsible for compiling the module, and all of them are linked together at the top-level `SConstruct`. All the submodule compile calls can be built in parallel, and only the submodules that have modified code will be recompiled on successive calls to `compile`.

There is one issue with the scanners and using `SConscript` files that the authors have not been able to reconcile yet. Even though `SConscript` will look for files relative to the directory its file is in (by default), the scanners appear to scan only from the location of `SConstruct`. One method around this issue is to have a default location where all `.f` files are placed that the scanner can start from, and each file then links to a `.f` file in the associated SConscript directory.

D. *Advanced Concepts for a Future Paper*

This introduction to SCons shows the basic steps needed to build a capable build tool and use it in an SConstruct file. There are many advanced concepts that can improve performance further. Some examples include:

- Adding command-line arguments to modify program flow dynamically.
- Using the `Help()` method to dynamically generate help text for the user.
- Creating Python-based targets that can dynamically generate TCL files used in a simulation.
- Using aliases to create conditional regression targets that modify a regression test list based on provided conditions.
- Using the built-in builders to compile C code needed for a chip simulation.
- Demonstrating a complex hierarchy of SConscript files and build tools.

IV. CONCLUSION

This paper introduced the basics of SystemVerilog configurations and then explained how use configurations to specify a unique file definition for a cell declared inside a design. With these definitions, the paper presented a set of model files and configuration settings that will work with the simulators available to the authors, and successfully demonstrated SystemVerilog configurations functioning across the simulators.

The Makefiles provided by the simulator vendors provided a baseline to introduce the concept of SCons as a newer, better way to manage design flow. The second part of this paper presented the basics for using SCons with SystemVerilog, and how it is an improvement over using Makefiles. A guide was presented that demonstrated the steps to creating a functional SCons tool extension to allow SCons to compile and simulate SystemVerilog code. While this tool is complete, it is still generic. It has support for modifying command-line arguments for the supported executables, but it doesn't handle any advanced management of command-line arguments. It is likely that within a company, each design team will have their own application of SCons and SystemVerilog tool extensions. This will allow each team to customize the builders to their specific environments, projects, and tools. Using the guide presented in this paper, the reader will be able to write these SCons builders and streamline project tool flows.

V. APPENDIX A: MAKEFILE.CADENCE

A. *Makefile.cadence*

```
all        : rtl        \
             rtl1       \
             rtl2       \
             cell       \
             cell1      \
             cell2      \
             inst       \
             inst1      \
             inst2

rtl   : clean
     xrun   -libmap libmap.sv     -compcnfg configs.sv -f source_code_cadence.f -top rtl_config   -exit
rtl1  : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top rtl_config1  -exit
rtl2  : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top rtl_config2  -exit

cell  : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config  -exit
cell1 : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config1 -exit
cell2 : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top cell_config2 -exit

inst  : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config  -exit
inst1 : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config1 -exit
inst2 : clean
     xrun   -libmap libmap_rtl.sv -compcnfg configs.sv -f source_code_cadence.f -top inst_config2 -exit

clean:
     rm -rf .simvision INCA_libs irun.* simvision*.diag xcelium.d xrun.*
```

B. *SConstruct.cadence*

```
EnsureSConsVersion(3, 0) # for Help() append

import os

Help('''
================================================================================
SConstruct for Cadence configurations:

Targets:
    compile
    compile_rtl
    sim_rtl / rtl
    sim_cell / cell
    sim_inst / inst

The simulation targets will automatically call the compile
targets as needed due to their dependencies.

Examples:
    scons -f SConstruct.cadence     (runs every option by default)
    scons -f SConstruct.cadence compile
    scons -f SConstruct.cadence sim_cell --svconfig=1

To clean:
    scons -f SConstruct.cadence -c

Notable SCons concepts:
  - Since the Makefile was using an all-in-one command, we added a
    builder to the xrun tool to handle running the all-in-one command
  - We set XRUN_ALL_ARGS in our xrunEnv object, overriding the tool's default
```

```
31.    - We tweaked the tool so that if it finds 'libmap' in the .sv filename,
32.      it will append it with the '-libmap' flag
33.    - We append a custom, per-target additional argument to XRUN_ALL_ARGS in
34.      each of the xrunEnv.AllIn1() calls. Anything passed in after the target and
35.      source lists overwrites the defaults in the tool and the xrunEnv object
36.    - We dynamically select a dependency for the 'sim_rtl' target based on
37.      the value passed in with the --svconfig option
38.  ================================================================================
39.  ''', append=True) # Only works on newer install of SCons
40.  # reference site_tools/site_init.py for help on --<options> if using
41.  # older version of SCons
42.
43.  ##############
44.  ### Env Setup
45.  xrunEnv = Environment(
46.    ENV={
47.      'PATH': os.environ['PATH'],
48.      'HOME': os.environ['HOME'],
49.    },
50.    TOOLS=['xrun'],
51.    SCANNERS=[svscan, fscan] + scanners,
52.    WORK='work',
53.    SVCONFIG = GetOption('svconfig'),
54.    XRUN_ALL_ARGS=['-compcnfg', '-exit'],
55.  )
56.
57.
58.  #################
59.  ### Simulate RTL
60.  sim_rtl = xrunEnv.AllIn1(
61.    'simulate_rtl_cadence.log',
62.    ['libmap_rtl.sv' if xrunEnv['SVCONFIG'] != '' else 'libmap.sv', 'configs.sv', 'source_code_cadence.f'],
63.    XRUN_ALL_ARGS=xrunEnv['XRUN_ALL_ARGS'] + ['-top rtl_config${SVCONFIG}']
64.  )
65.  Alias('sim_rtl', sim_rtl)
66.  Alias('rtl', sim_rtl)
67.
68.
69.  ##################
70.  ### Simulate Cell
71.  sim_cell = xrunEnv.AllIn1(
72.    'simulate_cell_cadence.log',
73.    ['libmap_rtl.sv', 'configs.sv', 'source_code_cadence.f'],
74.    XRUN_ALL_ARGS=xrunEnv['XRUN_ALL_ARGS'] + ['-top cell_config${SVCONFIG}']
75.  )
76.  Alias('sim_cell', sim_cell)
77.  Alias('cell', sim_cell)
78.
79.
80.  ##################
81.  ### Simulate Inst
82.  sim_inst = xrunEnv.AllIn1(
83.    'simulate_inst_cadence.log',
84.    ['libmap_rtl.sv', 'configs.sv', 'source_code_cadence.f'],
85.    XRUN_ALL_ARGS=xrunEnv['XRUN_ALL_ARGS'] + ['-top inst_config${SVCONFIG}']
86.  )
87.  Alias('sim_inst', sim_inst)
88.  Alias('inst', sim_inst)
89.
90.
91.  ########################
92.  ### Additional cleanup
93.  Clean([sim_rtl, sim_cell, sim_inst], [Glob('*~')])
```

VI. Appendix B: Makefile.mentor

A. *Makefile.mentor*

```
all         : rtl        \
              rtl1       \
              rtl2       \
              cell       \
              cell1      \
              cell2      \
              inst       \
              inst1      \
              inst2


compile:
    vlog -libmap libmap.sv     -f source_code.f
compile_rtl:
    vlog -libmap libmap_rtl.sv -f source_code.f

sim_rtl:
    vsim -c rtl_config          -do run_all.sim
sim_rtl1:
    vsim -c rtl_config1         -do run_all.sim
sim_rtl2:
    vsim -c rtl_config2         -do run_all.sim

sim_cell:
    vsim -c cell_config         -do run_all.sim
sim_cell1:
    vsim -c cell_config1        -do run_all.sim
sim_cell2:
    vsim -c cell_config2        -do run_all.sim

sim_inst:
    vsim -c inst_config         -do run_all.sim
sim_inst1:
    vsim -c inst_config1        -do run_all.sim
sim_inst2:
    vsim -c inst_config2        -do run_all.sim

rtl         : clean compile       sim_rtl
rtl1        : clean compile_rtl   sim_rtl1
rtl2        : clean compile_rtl   sim_rtl2

cell        : clean compile_rtl   sim_cell
cell1       : clean compile_rtl   sim_cell1
cell2       : clean compile_rtl   sim_cell2

inst        : clean compile_rtl   sim_inst
inst1       : clean compile_rtl   sim_inst1
inst2       : clean compile_rtl   sim_inst2

clean:
    rm -rf work transcript *Lib *~ *log .done
```

B. *SConstruct.mentor*

```
EnsureSConsVersion(3, 0) # for Help() append

import os

Help('''
================================================================================
SConstruct for Mentor configurations:

Targets:
    compile
    compile_rtl
    sim_rtl / rtl
```

```
    sim_cell / cell
    sim_inst / inst

The simulation targets will automatically call the compile
targets as needed due to their dependencies.

Examples:
    scons -f SConstruct.mentor     (runs every option by default)
    scons -f SConstruct.mentor compile
    scons -f SConstruct.mentor sim_cell --svconfig=1

To clean:
    scons -f SConstruct.mentor -c

Notable SCons concepts:
  - We set VSIM_ARGS in our vsimEnv object, overriding the tool's default
  - We tweaked the tool so that if it finds 'libmap' in the .sv filename,
    it will append it with the '-libmap' flag
  - We added to 'Clean' so SCons can remove the extra libraries generated
  - We append a custom, per-target additional argument to VSIM_ARGS in
    each of the vsimEnv.Sim() calls. Anything passed in after the target and
    source lists overwrites the defaults in the tool and the vsimEnv object
  - We dynamically select a dependency for the 'sim_rtl' target based on
    the value passed in with the --svconfig option
==============================================================================
''', append=True) # Only works on newer install of SCons
# reference site_tools/site_init.py for help on --<options> if using
# older version of SCons

###############
### Env Setup
vsimEnv = Environment(
  ENV={
    'PATH': os.environ['PATH']
  },
  TOOLS=['vsim'],
  SCANNERS=[svscan, fscan] + scanners,
  WORK='work',
  SVCONFIG=GetOption('svconfig'),
  VSIM_ARGS=['-c', '-do run_all.sim'],
)

#############
### Compile
com = vsimEnv.Com('compile_mentor.log', ['libmap.sv', 'source_code.f'])
Alias('compile', com)
Clean(com, ['rtlLib', 'gateLib'])


######################
### Compile RTL Only
com_rtl = vsimEnv.Com('compile_rtl_mentor.log', ['libmap_rtl.sv', 'source_code.f'])
Alias('compile_rtl', com_rtl)
Clean(com_rtl, [Glob('*Lib')])


##################
### Simulate RTL
sim_rtl = vsimEnv.Sim(
  'simulate_rtl_mentor.log',
  [com_rtl if vsimEnv['SVCONFIG'] != '' else com],
  VSIM_ARGS=vsimEnv['VSIM_ARGS'] + ['rtl_config${SVCONFIG}']
)
Alias('sim_rtl', sim_rtl)
Alias('rtl', sim_rtl)


##################
### Simulate Cell
sim_cell = vsimEnv.Sim(
  'simulate_cell_mentor.log',
```

```
84.    [com_rtl],
85.    VSIM_ARGS=vsimEnv['VSIM_ARGS'] + ['cell_config${SVCONFIG}']
86.  )
87.  Alias('sim_cell', sim_cell)
88.  Alias('cell', sim_cell)
89.
90.
91.  ##################
92.  ### Simulate Inst
93.  sim_inst = vsimEnv.Sim(
94.    'simulate_inst_mentor.log',
95.    [com_rtl],
96.    VSIM_ARGS=vsimEnv['VSIM_ARGS'] + ['cell_config${SVCONFIG}']
97.  )
98.  Alias('sim_inst', sim_inst)
99.  Alias('inst', sim_inst)
100.
101.
102. ######################
103. ### Additional cleanup
104. Clean([sim_rtl, sim_cell, sim_inst], ['transcript'])
105. Clean([com, com_rtl, sim_rtl, sim_cell, sim_inst], [Glob('*~')])
```

VII. Appendix C: Makefile.synopsys

A. *Makefile.synopsys*

```
all        : rtl        \
             rtl1       \
             rtl2       \
             cell       \
             cell1      \
             cell2      \
             inst       \
             inst1      \
             inst2

allsvcomp:
    vlogan -full64 -diag libconfig -sverilog -libmap libmap.sv     -f source_code.f

allsvcomp_rtl:
    vlogan -full64 -diag libconfig -sverilog -libmap libmap_rtl.sv -f source_code.f


rtl   : clean allsvcomp
    vcs -full64 -diag libconfig -debug_access -R rtl_config   -ucli -i run_vcs.do
rtl1  : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R rtl_config1  -ucli -i run_vcs.do
rtl2  : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R rtl_config2  -ucli -i run_vcs.do

cell  : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R cell_config  -ucli -i run_vcs.do
cell1 : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R cell_config1 -ucli -i run_vcs.do
cell2 : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R cell_config2 -ucli -i run_vcs.do

inst  : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R inst_config  -ucli -i run_vcs.do
inst1 : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R inst_config1 -ucli -i run_vcs.do
inst2 : clean allsvcomp_rtl
    vcs -full64 -diag libconfig -debug_access -R inst_config2 -ucli -i run_vcs.do


clean:
    rm -rf simv* csrc *Lib *~ *log .done .vlogan* ucli.key DVEfiles inter.vpd testlib
```

B. *SConstruct.synopsys*

```
EnsureSConsVersion(3, 0) # for Help() append

import os

Help('''
================================================================================
SConstruct for Synopsys configurations:

Targets:
    compile
    compile_rtl
    sim_rtl / rtl
    sim_cell / cell
    sim_inst / inst

The simulation targets will automatically call the compile
targets as needed due to their dependencies.

Examples:
    scons -f SConstruct.synopsys     (runs every option by default)
    scons -f SConstruct.synopsys compile
    scons -f SConstruct.synopsys sim_cell --svconfig=1
```

```
23.
24.  To clean:
25.      scons -f SConstruct.synopsys -c
26.
27.  Notable SCons concepts:
28.    - We set VLOGAN_ARGS and VCS_ARGS in our vcsEnv object, overriding the
29.      tool's default
30.    - We tweaked the tool so that if it finds 'libmap' in the .sv filename,
31.      it will append it with the '-libmap' flag
32.    - We removed the -work parts of the commands from the tool builders
33.    - We added to 'Clean' so SCons can remove the extra libraries generated
34.    - We append a custom, per-target additional argument to VCS_ARGS in
35.      each of the vcsEnv.Sim() calls. Anything passed in after the target and
36.      source lists overwrites the defaults in the tool and the vcsEnv object
37.    - We dynamically select a dependency for the 'sim_rtl' target based on
38.      the value passed in with the --svconfig option
39.  ============================================================================
40.  ''', append=True) # Only works on newer install of SCons
41.  # reference site_tools/site_init.py for help on --<options> if using
42.  # older version of SCons
43.
44.  ###############
45.  ### Env Setup
46.  vcsEnv = Environment(
47.    ENV={
48.      'PATH': os.environ['PATH'],
49.      'HOME': os.environ['HOME'],
50.    },
51.    TOOLS=['vcs'],
52.    SCANNERS=[svscan, fscan] + scanners,
53.    WORK='work',
54.    SVCONFIG=GetOption('svconfig'),
55.    VLOGAN_ARGS=['-full64', '-diag libconfig', '-sverilog'],
56.    VCS_ARGS=['-full64', '-diag libconfig', '-debug_access', '-ucli -i run_vcs.do'],
57.  )
58.
59.  #############
60.  ### Compile
61.  com = vcsEnv.Com('compile_synopsys.log', ['libmap.sv', 'source_code.f'])
62.  Alias('compile', com)
63.  Clean(com, ['rtlLib', 'gateLib'])
64.
65.
66.  ######################
67.  ### Compile RTL Only
68.  com_rtl = vcsEnv.Com('compile_rtl_synopsys.log', ['libmap_rtl.sv', 'source_code.f'])
69.  Alias('compile_rtl', com_rtl)
70.  Clean(com_rtl, [Glob('*Lib')])
71.
72.
73.  #################
74.  ### Simulate RTL
75.  sim_rtl = vcsEnv.Sim(
76.    'simulate_rtl_synopsys.log',
77.    [com_rtl if vcsEnv['SVCONFIG'] != '' else com],
78.    VCS_ARGS=vcsEnv['VCS_ARGS'] + ['rtl_config${SVCONFIG}']
79.  )
80.  Alias('sim_rtl', sim_rtl)
81.  Alias('rtl', sim_rtl)
82.
83.
84.  ##################
85.  ### Simulate Cell
86.  sim_cell = vcsEnv.Sim(
87.    'simulate_cell_synopsys.log',
88.    [com_rtl],
89.    VCS_ARGS=vcsEnv['VCS_ARGS'] + ['cell_config${SVCONFIG}']
90.  )
91.  Alias('sim_cell', sim_cell)
92.  Alias('cell', sim_cell)
93.
```

```
94.
95.  ##################
96.  ### Simulate Inst
97.  sim_inst = vcsEnv.Sim(
98.    'simulate_inst_synopsys.log',
99.    [com_rtl],
100.   VCS_ARGS=vcsEnv['VCS_ARGS'] + ['cell_config${SVCONFIG}']
101. )
102. Alias('sim_inst', sim_inst)
103. Alias('inst', sim_inst)
104.
105.
106. ########################
107. ### Additional cleanup
108. Clean([sim_rtl, sim_cell, sim_inst], ['ucli.key'])
109. Clean([com, com_rtl, sim_rtl, sim_cell, sim_inst], [Glob('*~')])
```

VIII. Appendix D: Full SCons and Configurations Example

This example will consist of the following files and directory structure:

```
.
├── adder_test.sv
├── configs.sv
├── dual_adder.sv
├── gate_adder_alt.sv
├── gate_adder.sv
├── libmap_gates.sv
├── libmap_rtl.sv
├── libmap.sv
├── Makefile
├── Makefile.cadence
├── Makefile.mentor
├── Makefile.synopsys
├── rtl_adder.sv
├── run_all.sim
├── run_cadence_gates.f
├── run_cadence_rtl.f
├── run_cadence.f
├── run_mentor_gates.f
├── run_mentor_rtl.f
├── run_mentor.f
├── run_synopsys_gates.f
├── run_synopsys_rtl.f
├── run_synopsys.f
├── run_vcs.do
├── SConstruct
├── SConstruct.cadence
├── SConstruct.mentor
├── SConstruct.synopsys
├── site_scons
│   ├── site_init.py
│   └── site_tools
│       ├── vcs
│       │   └── __init__.py
│       ├── vsim
│       │   └── __init__.py
│       └── xrun
│           └── __init__.py
├── source_code_cadence.f
├── source_code.f
├── synopsys_sim.setup
└── top.sv
```

Note that the Makefile.* and SConstruct.* files are contained in the previous three appendices, so they will not be included here again. The remainder of the files contain the code used throughout the paper to demonstrate configurations and SCons.

The files are also available upon request from the authors.

A. *adder_test.sv*

```
1.  module adder_test
2.  (output var a, b, ci,
3.   input  var sum1, co1,
4.   input  var sum2, co2);
5.    timeunit 1ns/1ns;
6.
7.    initial                 // input stimulus
8.      begin
9.        a = 0;
10.       b = 0;              //              sum   co
11.       ci = 0;             // should get:   0     0
12.       #10 a = 1;          // should get:   1     0
13.       #10 b = 1;          // should get:   0     1
14.       #10 ci = 1;         // should get:   1     1
15.       #10 $stop;
16.       #1000 $finish;
17.     end
18.
19.   initial                       //response checking
20.     begin
21.       //display time in ns, 2 decimal places, 10 char column width
22.       $timeformat(-9,2," ns",10);
23.       //print message on change
24.       $monitor("At %t: \t a=%b  b=%b  ci1=%b  sum1=%b  co1=%b  sum2=%b  co2=%b",
25.                $realtime, a, b, ci, sum1, co1, sum2, co2);
26.     end
27. endmodule
```

B. *configs.sv*

```
1.  config rtl_config;
2.    design rtlLib.top;
3.    default liblist rtlLib;
4.  endconfig
5.
6.  config rtl_config1;
7.    design testLib.top;
8.    default liblist rtlLib;
9.  endconfig
10.
11. config rtl_config2;
12.   design testLib.top;
13.   default liblist rtlLib testLib;
14. endconfig
15.
16.
17. config cell_config;
18.   design testLib.top;
19.   default liblist rtlLib testLib;
20.   cell adder liblist gateLib;
21. endconfig
22.
23. config cell_config1;
24.   design testLib.top;
25.   default liblist rtlLib testLib;
26.   cell adder use gateLib.adder;
27. endconfig
28.
29. config cell_config2;
30.   design testLib.top;
31.   default liblist rtlLib testLib;
32.   cell adder use gateLib.adder_alt;
33. endconfig
34.
35.
36. config inst_config;
37.   design testLib.top;
38.   default liblist rtlLib testLib;
```

```
39.    instance top.dut.adder2 liblist gateLib;
40.  endconfig
41.
42.  config inst_config1;
43.    design testLib.top;
44.    default liblist rtlLib testLib;
45.    instance top.dut.adder2 use gateLib.adder;
46.  endconfig
47.
48.  config inst_config2;
49.    design testLib.top;
50.    default liblist rtlLib testLib;
51.    instance top.dut.adder2 use gateLib.adder_alt;
52.  endconfig
```

C. *dual_adder.sv*

```
1.   module dual_adder (
2.     input  var a,b,ci,
3.     output var sum1, co1,
4.     output var sum2, co2
5.   );
6.     timeunit 1ns/1ns;
7.
8.     adder adder1 (.*,
9.                   .sum(sum1),
10.                  .co (co1 ));
11.
12.    adder adder2 (.*,
13.                  .sum(sum2),
14.                  .co (co2 ));
15.
16.  endmodule:dual_adder
```

D. *gate_adder_alt.sv*

```
1.   module adder_alt
2.   (input  var a, b, ci,
3.    output var sum, co);
4.     timeunit 1ns/1ns;
5.
6.     // this file is the same as gate_adder.sv but
7.     // with a different module name
8.
9.     logic  n1, n2, n3;
10.
11.    xor    g1 (n1,  a,  b );
12.    xor #2 g2 (sum, n1, ci);
13.    and    g3 (n2,  a,  b );
14.    and    g4 (n3,  n1, ci);
15.    or  #2 g5 (co,  n2, n3);
16.
17.    initial $info("gate adder_alt is being used");
18.
19.  endmodule:adder_alt
```

E. *gate_adder.sv*

```
1.   module adder
2.   (input  var a, b, ci,
3.    output var sum, co);
4.     timeunit 1ns/1ns;
5.
6.     logic  n1, n2, n3;
7.
8.     xor    g1 (n1,  a,  b );
9.     xor #2 g2 (sum, n1, ci);
10.    and    g3 (n2,  a,  b );
11.    and    g4 (n3,  n1, ci);
```

```
12.    or  #2 g5 (co,  n2, n3);
13.
14.    initial $info("gate adder is being used");
15.
16.  endmodule:adder
```

F. *libmap_gates.sv*

```
1.   library rtlLib  rtl_adder.sv;
2.
3.   library gateLib gate_adder.sv,
4.                   gate_adder_alt.sv,
5.                   dual_adder.sv;
6.
7.   library testLib top.sv,
8.                   adder_test.sv;
```

G. *libmap_rtl.sv*

```
1.   library rtlLib  rtl_adder.sv,
2.                   dual_adder.sv;
3.
4.   library gateLib gate_adder.sv,
5.                   gate_adder_alt.sv;
6.
7.   library testLib top.sv,
8.                   adder_test.sv;
```

H. *libmap.sv*

```
1.   library rtlLib  top.sv,
2.                   adder_test.sv,
3.                   rtl_adder.sv,
4.                   dual_adder.sv;
5.
6.   library gateLib gate_adder.sv,
7.                   gate_adder_alt.sv;
```

I. *Makefile*

```
1.   all  :
2.       make -j -f Makefile.mentor
3.       make -j -f Makefile.cadence
4.       make    -f Makefile.synopsys
5.       make    -f Makefile.mentor   clean
6.       make    -f Makefile.cadence  clean
7.       make    -f Makefile.synopsys clean
8.
9.   clean:
10.      make -f Makefile.mentor   clean
11.      make -f Makefile.cadence  clean
12.      make -f Makefile.synopsys clean
```

J. *rtl_adder.sv*

```
1.   module adder
2.   (input  var  a, b, ci,
3.    output var  sum, co);
4.     timeunit 1ns/1ns;
5.
6.     always_comb
7.       {co, sum} = a + b + ci;
8.
9.     initial $info("rtl  adder is being used");
10.
11.  endmodule:adder
```

K. *run_all.sim*

```
run -all
exit
```

L. *run_cadence_gates.f*

```
###########
# Cleanup old runs and enable elaboration debugging
-clean
-cleanlib
-libverbose

##########
# Compile and design configuration management files
# design configuration compiled into "worklib"
-libmap libmap_gates.sv
-compcnfg configs.sv

##########
# Source Code
-f source_code.f
```

M. *run_cadence_rtl.f*

```
###########
# Cleanup old runs and enable elaboration debugging
-clean
-cleanlib
-libverbose

##########
# Compile and design configuration management files
# design configuration compiled into "worklib"
-libmap libmap_rtl.sv
-compcnfg configs.sv

##########
# Source Code
-f source_code.f
```

N. *run_cadence.f*

```
###########
# Cleanup old runs and enable elaboration debugging
-clean
-cleanlib
-libverbose

##########
# Compile and design configuration management files
# design configuration compiled into "worklib"
-libmap libmap.sv
#-compcnfg configs.sv

##########
# Source Code
-f source_code.f
```

O. *run_mentor_gates.f*

```
#######
## Command file to compile the RTL version of the 1-bit adder and testbench
-libmap libmap_gates.sv

#######
#Source Code
```

```
7.  -f source_code.f
```

P. *run_mentor_rtl.f*

```
1.  #######
2.  ## Command file to compile the RTL version of the 1-bit adder and testbench
3.  -libmap libmap_rtl.sv
4.
5.  #######
6.  #Source Code
7.  -f source_code.f
```

Q. *run_mentor.f*

```
1.  #######
2.  ## Command file to compile the RTL version of the 1-bit adder and testbench
3.  -libmap libmap.sv
4.
5.  #######
6.  #Source Code
7.  -f source_code.f
```

R. *run_synopsys_gates.f*

```
1.  ##########
2.  # Compile and design configuration management files
3.  -libmap libmap_gates.sv
4.
5.  #######
6.  #Source Code
7.  -f source_code.f
```

S. *run_synopsys_rtl.f*

```
1.  ##########
2.  # Compile and design configuration management files
3.  -libmap libmap_rtl.sv
4.
5.  #######
6.  #Source Code
7.  -f source_code.f
```

T. *run_synopsys.f*

```
1.  ##########
2.  # Compile and design configuration management files
3.  -libmap libmap.sv
4.
5.  #######
6.  #Source Code
7.  -f source_code.f
```

U. *run_vcs.do*

```
1.  run
2.  exit
```

V. *SConstruct*

```
1.  EnsureSConsVersion(3, 0) # for Help() append
2.
3.  ######################
4.  # File: SConstruct   #
5.  ######################
6.
7.  Help('''
8.  Main SConstruct for configurations:
```

```
9.
10.  Runs all three SConstruct.<vendor> files in parallel using the num_jobs option.
11.  Some of these will fail due to overwriting the work libraries - additional
12.  steps would need to be taken to ensure each compile step uses a distinct work
13.  library.
14.
15.  Run "scons -n" to see all the commands this file would run printed out.
16.
17.  Run "scons -c" to clean everything.
18.  ''', append=True) # Only works on newer install of SCons
19.  # See the help text in each sub-file if using an older version of SCons
20.
21.  env = Environment()
22.
23.  # Default to -j 8
24.  SetOption('num_jobs', 8)
25.
26.  # See individual scons files for help:
27.  # scons -f SConstruct.cadence -h
28.  SConscript('SConstruct.cadence')
29.  SConscript('SConstruct.mentor')
30.  SConscript('SConstruct.synopsys')
```

W. *site_scons/site_init.py*

```
1.   """SCons.site_init
2.
3.   SCons site_init file
4.
5.   There normally shouldn't be any need to import this module directly,
6.   it will usually be imported by SCons automatically.
7.   """
8.   import os, re, subprocess
9.
10.
11.  ##############################################################################
12.  # OPTIONS:
13.  #   Add all command line options here (-o, --option)
14.  ##############################################################################
15.
16.  AddOption('--tool',
17.            dest='tool',
18.            type='choice',
19.            choices=['vcs', 'vsim', 'xrun'],
20.            nargs=1,
21.            action='store',
22.            default='vsim',
23.            help='The vendor tool to use [vcs|vsim|xrun]. Default: vsim')
24.
25.
26.  AddOption('--svconfig',
27.            dest='svconfig',
28.            type='choice',
29.            choices=['', '1', '2'],
30.            nargs=1,
31.            action='store',
32.            default='',
33.            help='The alternate configuration to use [1|2]. Default: None')
34.
35.
36.  ##############################################################################
37.  # SCANNERS:
38.  #   Scanners to parse .f files and .sv files for dependencies
39.  ##############################################################################
40.  f_re = re.compile(r'^-f\s+(\S+\.f)$', re.M | re.I)
41.  sv_re = re.compile(r'^(/?[^/+]\S+\.s?vh?)$', re.M | re.I)
42.  include_re = re.compile(r'^\s*`include\s+"(\S+)"$', re.M | re.I)
43.
44.  def ffile_scan(node, env, path, arg=None):
45.    contents = node.get_text_contents()
```

```
46.
47.    sv_files = sv_re.findall(contents)
48.    sv_files = [sv.strip() for sv in sv_files]
49.    f_files = f_re.findall(contents)
50.
51.    while f_files:
52.      for f in f_files:
53.        # We use the following line to expand any environment variables in the filepath
     using
54.        # our custom SCons environment. This will catch any variables declared in the
     SConstruct.
55.        ef = subprocess.check_output('echo ' + f, shell=True, env=env['ENV']).strip()
56.        if os.path.isfile(ef):
57.          current_dir = os.path.dirname(ef) + '/'
58.          contents = env.File(ef).get_text_contents()
59.
60.          sv_files.extend([(current_dir + x.strip()) for x in sv_re.findall(contents)])
61.          f_files.extend([(current_dir + x.strip()) for x in f_re.findall(contents)])
62.          sv_files.append(str(ef))
63.        f_files.remove(f)
64.
65.    results = []
66.    for f in env.File(sv_files):
67.      results.extend(svfile_scan(f, env, path, arg))
68.
69.    return results
70.
71.  def svfile_scan(node, env, path, arg=None):
72.    contents = node.get_text_contents()
73.    includes = include_re.findall(contents)
74.
75.    starting_dir = str(node.dir) + '/'
76.
77.    if includes == []:
78.      return [node]
79.
80.    results = [str(node)]
81.    for inc in includes:
82.      if os.path.exists(starting_dir + inc):
83.        results.append(starting_dir + inc)
84.
85.    return env.File(results)
86.
87.  svscan = Scanner(
88.    name='svfile',
89.    function=svfile_scan,
90.    argument=None,
91.    skeys=['.v', '.vh', '.sv', '.svh'])
92.
93.  fscan = Scanner(
94.    name='ffile',
95.    function=ffile_scan,
96.    argument=None,
97.    skeys=['.f'])
98.
99.  scanners = Environment().Dictionary('SCANNERS')
```

X. *site_scons/site_tools/vcs/__init__.py*

```
1.   """SCons.Tool.vcs
2.
3.   Tool-specific initialization for the VCS compiler.
4.
5.   There normally shouldn't be any need to import this module directly.
6.   It will usually be imported through the generic SCons.Tool.Tool()
7.   selection method.
8.   """
9.   import os
10.  from SCons.Script import *
11.
```

```
12. ########################################################################
13. # BUILDERS:
14. ########################################################################
15. ### vlogan
16. def generate_vlogan(source, target, env, for_signature):
17.   action = [env['VLOGAN']]
18.   for s in source:
19.     if os.path.splitext(str(s))[1] == '.f':
20.       action.append('-F')
21.     elif 'libmap' in str(s):
22.       action.append('-libmap')
23.     action.append(str(s))
24.
25.   action.extend(env['VLOGAN_ARGS'])
26.   action.extend(['-l $TARGET'])
27.
28.   return ' '.join(action)
29.
30. def Vlogan(env, target, source, *args, **kw):
31.   """A pseudo-Builder wrapper for the vlogan executable."""
32.
33.   _vlogan_builder = Builder(generator=generate_vlogan, suffix='.log')
34.
35.   result = _vlogan_builder.__call__(env, target, source, **kw)
36.
37.   # Ensures multiple vlogan calls don't attempt to write to the work directory
    simultaneously.
38.   env.SideEffect(str(result[0].dir)+'/${WORK}/AN.DB/.vcs_lib_lock', result[0])
39.
40.   # Removes the .done file and the work directory, in addition to the logfile
41.   env.Clean(result, [
42.     str(result[0].dir) + '/${WORK}',
43.     '.vlogansetup.args',
44.   ])
45.
46.   return result
47.
48. ### vcs
49. def generate_vcs(source, target, env, for_signature):
50.   action = [env['VCS'], '-R']
51.
52.   action.extend([
53.     '-Mdir=${TARGET.dir}/csrc',
54.     '-o ${TARGET.dir}/simv',
55.   ])
56.   action.extend(env['VCS_ARGS'])
57.   action.extend(['-l $TARGET'])
58.
59.   return ' '.join(action)
60.
61. def Vcs(env, target, source, *args, **kw):
62.   """A pseudo-Builder wrapper for the vcs executable."""
63.
64.   _vcs_builder = Builder(generator=generate_vcs, suffix='.log')
65.
66.   result = _vcs_builder.__call__(env, target, source, **kw)
67.
68.   # Removes the .done file and the logfile
69.   env.Clean(result, [
70.     str(result[0].dir)+'/csrc',
71.     str(result[0].dir)+'/simv',
72.     str(result[0].dir)+'/simv.daidir',
73.   ])
74.
75.   return result
76.
77.
78. ########################################################################
79. # TOOL FUNCTIONS:
80. #   generate() and exists() are required by SCons
81. ########################################################################
```

```
82.  def generate(env):
83.    """Add Builders and construction variables to the Environment."""
84.
85.    env['VLOGAN'] = env.WhereIs('vlogan')
86.    env['VCS'] = env.WhereIs('vcs')
87.
88.    env.SetDefault(
89.      SIM_DIR='./',
90.      WORK='work',
91.      TOP_TB='top',
92.      VLOGAN_ARGS=['-sverilog'],
93.      VCS_ARGS=[],
94.    )
95.
96.    env.AddMethod(Vlogan, "Com")
97.    env.AddMethod(Vcs, "Sim")
98.
99.  def exists(env):
100.   return True
```

Y. *site_scons/site_tools/vsim/__init__.py*

```
1.   """SCons.Tool.vsim
2.
3.   Tool-specific initialization for the Questa compiler.
4.
5.   There normally shouldn't be any need to import this module directly.
6.   It will usually be imported through the generic SCons.Tool.Tool()
7.   selection method.
8.   """
9.   import os
10.  from SCons.Script import *
11.
12.  ###############################################################################
13.  # BUILDERS:
14.  ###############################################################################
15.  ### vlog
16.  def generate_vlog(source, target, env, for_signature):
17.    action = [env['VLOG']]
18.    for s in source:
19.      if os.path.splitext(str(s))[1] == '.f':
20.        action.append('-F')
21.      elif 'libmap' in str(s):
22.        action.append('-libmap')
23.      action.append(str(s))
24.
25.    action.extend(['-work ${TARGET.dir}/${WORK}'])
26.    action.extend(env['VLOG_ARGS'])
27.    action.extend(['-l $TARGET'])
28.
29.    return ' '.join(action)
30.
31.  def Vlog(env, target, source, *args, **kw):
32.    """A pseudo-Builder wrapper for the vlog executable."""
33.
34.    _vlog_builder = Builder(generator=generate_vlog, suffix='.log')
35.
36.    result = _vlog_builder.__call__(env, target, source, **kw)
37.
38.    # Ensures multiple vlog calls don't attempt to write to the work directory
     simultaneously.
39.    env.SideEffect(str(result[0].dir)+'/${WORK}/_lib.qdb', result[0])
40.
41.    # Removes the .done file and the work directory, in addition to the logfile
42.    env.Clean(result, ['.done', str(result[0].dir)+'/${WORK}'])
43.
44.    return result
45.
46.  ### vsim
47.  def generate_vsim(source, target, env, for_signature):
```

```
48.    action = [env['VSIM']]
49.
50.    action.extend(['-lib ${TARGET.dir}/${WORK}'])
51.    action.extend(env['VSIM_ARGS'])
52.    action.extend(['-l $TARGET'])
53.
54.    return ' '.join(action)
55.
56.  def Vsim(env, target, source, *args, **kw):
57.    """A pseudo-Builder wrapper for the vsim executable."""
58.
59.    _vsim_builder = Builder(generator=generate_vsim, suffix='.log')
60.
61.    result = _vsim_builder.__call__(env, target, source, **kw)
62.
63.    # Removes the .done file and the logfile
64.    env.Clean(result, ['.done'])
65.
66.    return result
67.
68.
69.  ############################################################################
70.  # TOOL FUNCTIONS:
71.  #   generate() and exists() are required by SCons
72.  ############################################################################
73.  def generate(env):
74.    """Add Builders and construction variables to the Environment."""
75.
76.    env['VLOG'] = env.WhereIs('vlog')
77.    env['VSIM'] = env.WhereIs('vsim')
78.
79.    env.SetDefault(
80.      SIM_DIR='./',
81.      WORK='work',
82.      VLOG_ARGS=[],
83.      VSIM_ARGS=[],
84.    )
85.
86.    env.AddMethod(Vlog, "Com")
87.    env.AddMethod(Vsim, "Sim")
88.
89.  def exists(env):
90.    return True
```

Z. *site_scons/site_tools/xrun/__init__.py*

```
1.   """SCons.Tool.xrun
2.
3.   Tool-specific initialization for the Xcelium compiler.
4.
5.   There normally shouldn't be any need to import this module directly.
6.   It will usually be imported through the generic SCons.Tool.Tool()
7.   selection method.
8.   """
9.   import os
10.  from SCons.Script import *
11.
12.  ############################################################################
13.  # BUILDERS:
14.  ############################################################################
15.  ### xrun
16.  def generate_xrun(source, target, env, for_signature):
17.    action = [env['XRUN'], '-elaborate']
18.    for s in source:
19.      if os.path.splitext(str(s))[1] == '.f':
20.        action.append('-F')
21.      elif 'libmap' in str(s):
22.        action.append('-libmap')
23.      action.append(str(s))
24.
```

```
25.    action.extend([
26.      '-xmlibdirname ${TARGET.dir}/${WORK}',
27.      '-history_file ${TARGET.dir}/xrun.history',
28.    ])
29.    action.extend(env['XRUN_ARGS'])
30.    action.extend(['-l $TARGET'])
31.
32.    return ' '.join(action)
33.
34.  def Xrun(env, target, source, *args, **kw):
35.    """A pseudo-Builder wrapper for the xrun executable."""
36.
37.    _xrun_builder = Builder(generator=generate_xrun, suffix='.log')
38.
39.    result = _xrun_builder.__call__(env, target, source, **kw)
40.
41.    # Ensures multiple xrun calls don't attempt to write to the work directory
     simultaneously.
42.    env.SideEffect(str(result[0].dir)+'/${WORK}/run.d/cds.lib', result[0])
43.
44.    # Removes the work directory and the .history files, in addition to the logfile
45.    env.Clean(result, [
46.      str(result[0].dir)+'/${WORK}',
47.      env.Glob(str(result[0].dir)+'/*.history')
48.    ])
49.
50.    return result
51.
52.
53.  ### xrun_sim
54.  def generate_xrun_sim(source, target, env, for_signature):
55.    action = [env['XRUN_SIM'], '-R']
56.
57.    action.extend([
58.      '-xmlibdirname ${TARGET.dir}/${WORK}',
59.      '-history_file ${TARGET.dir}/xrun.history',
60.    ])
61.    action.extend(env['XRUN_SIM_ARGS'])
62.    action.extend(['-l $TARGET'])
63.
64.    return ' '.join(action)
65.
66.  def Xrun_sim(env, target, source, *args, **kw):
67.    """A pseudo-Builder wrapper for the xrun_sim executable."""
68.
69.    _xrun_sim_builder = Builder(generator=generate_xrun_sim, suffix='.log')
70.
71.    result = _xrun_sim_builder.__call__(env, target, source, **kw)
72.
73.    return result
74.
75.
76.  ### xrun_all
77.  def generate_xrun_all(source, target, env, for_signature):
78.    action = [env['XRUN_ALL']]
79.
80.    for s in source:
81.      if os.path.splitext(str(s))[1] == '.f':
82.        action.append('-F')
83.      elif 'libmap' in str(s):
84.        action.append('-libmap')
85.      action.append(str(s))
86.
87.    action.extend([
88.      '-xmlibdirname ${TARGET.dir}/${WORK}',
89.      '-history_file ${TARGET.dir}/xrun.history',
90.    ])
91.    action.extend(env['XRUN_ALL_ARGS'])
92.    action.extend(['-l $TARGET'])
93.
94.    return ' '.join(action)
```

```
95.
96.  def Xrun_all(env, target, source, *args, **kw):
97.    """A pseudo-Builder wrapper for the xrun_all executable."""
98.
99.    _xrun_all_builder = Builder(generator=generate_xrun_all, suffix='.log')
100.
101.   result = _xrun_all_builder.__call__(env, target, source, **kw)
102.
103.   # Removes the work directory and the .history files, in addition to the logfile
104.   env.Clean(result, [
105.     str(result[0].dir)+'/${WORK}',
106.     env.Glob(str(result[0].dir) + '/*.history'),
107.   ])
108.   return result
109.
110.
111. ##############################################################################
112. # TOOL FUNCTIONS:
113. #   generate() and exists() are required by SCons
114. ##############################################################################
115. def generate(env):
116.   """Add Builders and construction variables to the Environment."""
117.
118.   env['XRUN'] = env.WhereIs('xrun')
119.   env['XRUN_SIM'] = env.WhereIs('xrun')
120.   env['XRUN_ALL'] = env.WhereIs('xrun')
121.
122.   env.SetDefault(
123.     SIM_DIR='./',
124.     WORK='work',
125.     XRUN_ARGS=[],
126.     XRUN_SIM_ARGS=[],
127.     XRUN_ALL_ARGS=[],
128.   )
129.
130.   env.AddMethod(Xrun, "Com")
131.   env.AddMethod(Xrun_sim, "Sim")
132.   env.AddMethod(Xrun_all, "AllIn1")
133.
134. def exists(env):
135.   return True
```

AA. *source_code_cadence.f*

```
1.   adder_test.sv
2.   dual_adder.sv
3.   gate_adder.sv
4.   gate_adder_alt.sv
5.   rtl_adder.sv
6.   top.sv
```

BB. *source_code.f*

```
1.   adder_test.sv
2.   dual_adder.sv
3.   gate_adder.sv
4.   gate_adder_alt.sv
5.   rtl_adder.sv
6.   top.sv
7.   configs.sv
```

CC. *synopsys_sim.setup*

```
1.   WORK    > DEFAULT
2.   DEFAULT : work
3.   rtllib  : rtlLib
4.   gatelib : gateLib
5.   testlib : testLib
6.
```

```
7.   -- NOTES
8.   -- We can just elaborate the configs and the tool looks to the synopsys_sim.setup
9.   -- for the libraries.  You can override that with -liblist but using the setup
10.  -- file is more straightforward.
11.
12.  -- from the VCS documentation:
13.  -- Look for the libraries specified in -liblist at the command line, or
14.  --   look for the libraries specified in the synopsys_sim.setup file if
15.  --   -liblist is not passed.
```

DD. *top.sv*

```
1.   module top;
2.     timeunit 1ns/1ns;
3.
4.     logic  a, b, ci;
5.     logic  sum1, co1;
6.     logic  sum2, co2;
7.
8.     adder_test test (.*);
9.     dual_adder dut  (.*);
10.
11.  endmodule:top
```

IX. References

[1] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language," IEEE, 1996.

[2] "IEEE Standard Verilog Hardware Description Language," IEEE, 2001.

[3] "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language," IEEE, 2005.

[4] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," IEEE, 2018.

[5] W. Deegan, "ToolsForFools," SCons Foundation, 8 February 2018. [Online]. Available: https://github.com/SCons/scons/wiki/ToolsForFools. [Accessed 12 October 2019].

[6] "SCons: A software construction tool," SCons Foundation, 2019. [Online]. Available: https://scons.org. [Accessed 12 October 2019].

[7] Steven Knight and the SCons Development Team, "SCons 3.1.1 MAN page," The SCons Foundation, 2019. [Online]. Available: https://scons.org/doc/production/HTML/scons-man.html. [Accessed 11 November 2019].