



OpenPiton Simulation Manual

Wentzlaff Parallel Research Group

Princeton University

openpiton@princeton.edu

Revision History

Revision	Date	Author(s)	Description
1.0	06/10/15	MM	Initial version
2.0	04/01/16	MM	Second version
2.1	04/03/16	WPRG	Third version
2.2	10/21/16	MM	Updates for new release
2.3	09/21/17	JB	Updates for new simulators

Contents

1	Introduction	1
2	Supported Third-Party Tools and Environments	1
2.1	Operating Systems	2
2.2	Unix Shells	2
2.3	Script Interpreters	2
2.4	Job Queue Managers	3
2.5	EDA Tools	3
2.5.1	Verilog Pre-Processor	3
2.5.2	Verilog Simulator	3
3	Directory Structure and File Organization	4
3.1	Directory Structure	4
3.2	Common File Extensions/Naming Conventions . .	6
4	Environment Setup	8
5	Simulation	9
5.1	Simulation Models	10
5.1.1	Types of Simulation Models	11
5.1.2	Building a Simulation Model	13
5.1.3	Configuring the <code>manycore</code> Simulation Model	15
5.1.3.1	Configuring the Number of Tiles	15
5.1.3.2	Configuring Cache Parameters .	16
5.2	Running a Simulation	16
5.2.1	Assembly Tests	17
5.2.2	C Tests	19
5.2.3	Unit Tests	19
5.2.4	<code>sims</code> Simulation Run Flow/Steps	20

5.3	Running Advanced Simulations Using the <code>manycore</code> Simulation Model	22
5.3.1	Specifying Number of Threads and Thread Mapping for a Simulation	23
5.3.2	Specifying Monitor Arguments for a Simulation	24
5.3.3	Debugging Simulations with <code>sims</code>	24
5.4	Running a Regression Suite	24
5.5	Running a Continuous Integration Bundle	26
5.6	Determining Test Coverage	28
A	<code>sims</code> manpage	29
B	<code>contint</code> manpage	51
C	Ubuntu 14.04/16.04 Dependencies and Workarounds	51
C.1	Dependencies	51
C.2	VCS simulation workaround	52
	References	53

List of Figures

1	OpenPiton Directory Structure	4
2	Unit testing infrastructure source-sink model . . .	12
3	sims Simulation Model Build Flow/Steps	14
4	sims Simulation Model Run Flow/Steps	21

List of Tables

2	Common file extensions/naming conventions . . .	6
3	OpenPiton Simulation Models	11
4	OpenPiton Regression Suites	25
5	OpenPiton Continuous Integration Bundles	27

1 Introduction

This document introduces the OpenPiton simulation infrastructure and how it is used to configure and run simulations. It also discusses the OpenPiton test suite, how to add new tests, and how to determine test coverage. Some of the information in this document is based on the OpenSPARC T1 Processor Design and Verification User Guide [1].

The OpenPiton processor is a scalable, configurable, open-source implementation of the Piton processor, designed and taped-out at Princeton University by the Wentzlaff Parallel Research Group in March 2015. The RTL is scalable up to half a billion cores, it is written in Verilog HDL, and a large test suite (~8000 tests) is provided for simulation and verification. The infrastructure is also designed to be configurable, enabling configuration of the number of tiles, sizes of structures, type of interconnect, etc. Extensibility is another key goal, making it easy for users to extend the current infrastructure and explore research ideas. We hope for OpenPiton to be a useful tool to both researchers and industry engineers in exploring and designing future manycore processors.

This document covers the following topics:

- Supported third-party tools and environments
- Directory structure and file organization
- OpenPiton environment setup
- Building simulation models and running simulations
- Tools for running regressions and continuous integration bundles
- The OpenPiton test suite
- Creating new tests (assembly and C)
- Determining test coverage

2 Supported Third-Party Tools and Environments

This section discusses third-party tools/environments that are supported and/or required to use OpenPiton. Specifically, it discusses supported operating systems (OSs), Unix shells, script interpreters, job queue managers and EDA tools. For the most up-

to-date information, please check the OpenPiton website, www.openpiton.org.

2.1 Operating Systems

The current release only supports Linux distributions. It has been tested with the following distributions:

- Ubuntu 12.10
- Ubuntu 14.04*
- Ubuntu 16.04*
- Springdale Linux (Custom Red Hat distro) 6.6/6.8
- macOS High Sierra 10.13.6

We expect OpenPiton to work out of the box on most other Linux distributions, but it has not been tested and, thus, we provide no guarantees. There are currently no plans to expand OS support. If you find that OpenPiton is stable on another Linux distribution/version, please let us know at openpiton@princeton.edu so we can update the list on our website.

*: Please see Appendix C for more information.

2.2 Unix Shells

OpenPiton currently only supports the Bash Unix shell. While environment setup scripts are provided for CShell, OpenPiton has not been tested for use with CShell and we do not claim that it is supported.

2.3 Script Interpreters

Python is required in order to run PyHP preprocessor and other python scripts. Currently it has been tested with version 2.6.6.

Perl is required in order to run several Perl scripts. It is configured in `$PITON_ROOT/piton/piton_setting.bash` through the `PERL_CMD` environment variable and the default path is `/usr/bin/perl`. Please modify the path to the correct one if Perl is installed on a different path in your own environment. Currently Perl has been tested with version 5.10.1.

2.4 Job Queue Managers

SLURM (Simple Linux Utility for Resource Management) is optional and many OpenPiton scripts support using it to submit batch jobs. Currently SLURM has been tested with versions 15.08.8 and 16.05.5.

2.5 EDA Tools

2.5.1 Verilog Pre-Processor

OpenPiton uses the PyHP Verilog pre-processor (v1.12) [2] to improve code quality/readability and configurability. PyHP allows for Python code to be embedded into Verilog files between `<% %>` tags. The Python code can generate Verilog by printing to `stdout`. The PyHP pre-processor executes the Python code and generates a Verilog file with the embedded Python replaced by it's corresponding output on `stdout`. Verilog files with embedded Python intended to be pre-processed by PyHP are given the file extension `.pyv.v` or `.pyv.h` for define/include files. PyHP is distributed with the OpenPiton download. More details on how PyHP integrates into the simulation infrastructure is discussed in Section 5.1.2

2.5.2 Verilog Simulator

Currently, Verilog simulation is supported using Synopsys VCS, Cadence Incisive, Mentor ModelSim, and Icarus Verilog. OpenPiton has been tested with the following simulator versions:

- vcs_mx_I-2014.03
- vcs_mx_vL-2016.06
- Cadence Incisive Simulator 15.2
- Mentor ModelSim 10.6a
- Icarus Verilog 10.1.1

If using a different version of any of these simulators, you may need to run `mkplilib clean; mkplilib ncverilog` before building your simulation model.

Icarus Verilog 10.1.1 works well and we have also had success with the latest Git sources (as of release 6). However, Icarus does not support the full range of PLI used by OpenPiton and

as such some assembly tests and simulation monitors are not supported.

3 Directory Structure and File Organization

This section discusses the OpenPiton infrastructure directory structure and common file extensions used in OpenPiton.

3.1 Directory Structure

This section discusses the OpenPiton infrastructure directory structure. Figure 1 shows the organization of important directories. At the top level are the `build/`, `docs/`, and `piton/` directories.

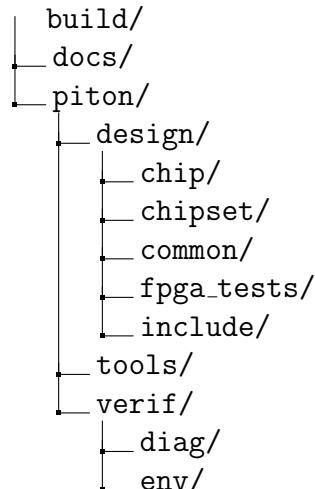


Figure 1: OpenPiton Directory Structure

The `build/` directory is a working directory for OpenPiton and is shipped empty. As far as simulation and testing goes, it acts as a scratch directory for files generated when building simulation models, compiling tests, running simulations, etc. For example, all of the simulation models are built into corresponding directories within the `build/` directory. We recommend that most OpenPiton tools are run from within this directory, as many tools generate files in the current working directory. For example, running a simulation generates a log file which is stored, by default, in the directory from which the simulation was run. It is convenient to keep all generated files within the `build/` directory so they are easy to locate and clean up. Feel free to create your

own directory hierarchy within `build/` to further organize your working space, it is yours to customize.

All of the OpenPiton documentation is kept in the `docs/` directory. It is conveniently distributed all in one place with the OpenPiton download. The most up to date documentation is also available on our website, www.openpiton.org.

The `piton/` directory contains all of the design files, verification files, and scripts. It is therefore logically broken down into `design/`, `verif/`, and `tools/` directories.

The `design/` directory contains all synthesizable Verilog for OpenPiton and is broken down into several subdirectories: `chip/`, `chipset/`, `common/`, `fpga_tests`, and `include/`. Within these four subdirectories, the directory hierarchy follows the major points in the design's module hierarchy. In addition to Verilog design files, these directories contain flist files, which list Verilog files for a given design and are referenced by simulation tools to determine which Verilog files are needed to build portions of the design.

The `chip/` directory contains the Verilog design files for a scalable, configurable OpenPiton chip, please see the OpenPiton Microarchitecture Manual for more details on the design. The `chipset/` directory contains the Verilog design files for the chipset FPGA portion of the OpenPiton system which communicates to an OpenPiton chip through the chip/fpga bridge and provides access to main memory, multiplexes memory-mapped I/O, and routes packets between OpenPiton chips (see OpenPiton Microarchitecture Manual for more details). The `common/` directory includes Verilog design files common to other top-level subdirectories within the `design/` directory. The `fpga_tests/` directory contains Verilog design files for a number of top-level designs which test different portions of the design on FPGA, such as I/O and memory controllers. The `include/` directory contains Verilog files which define global macros for OpenPiton. These macros are used to set parameters for different portions of the design.

All scripts and tools used in the OpenPiton infrastructure reside in the `tools/` directory. We will not document in detail the scripts and tools, other than how to use them, which is what the following sections of this document are about. There are a few locations worth pointing out within the `tools/` directory: the

location of the `sims` configuration files, `tools/src/sims/`, and the `contint` configuration files, `tools/src/contint/`. The use of the configuration files within will be explained in Section 5.

Last, the `verif/` directory houses all verification files. This includes assembly and C tests, or diags, unit tests, and simulation models. Within `verif/`, the `diag/` directory contains all assembly and C tests. In addition, it also contains diaglists, which define parameters for certain tests and define groups of tests, or regressions, and common assembly and C test infrastructure (boot code, etc.). The `env/` directory contains non-synthesizeable Verilog files (testbenches) needed to build simulation models. For unit testing simulation models (see Section 5.1.1), the tests are located within the `env/` directory as opposed to the `diag/` directory. In general, the `manycore` simulation model will run assembly and C tests in the `diag/` directory, and all other simulation models will run based on unit tests in `env/`. Infrastructure for unit testing is provided in the `env/test_infstrct/` directory along with a script to quickly and easily generate a new simulation model, `env/create_env.py`.

3.2 Common File Extensions/Naming Conventions

Table 2 lists common file extensions and naming conventions:

Table 2: Common file extensions/naming conventions

File extension	Description
<code>.v</code>	Verilog design files.
<code>.pyv.v</code>	Verilog design files with embedded Python code. A <code>.pyv.v</code> file is run through the PyHP pre-processor prior to building simulation models, generating a <code>.tmp.v</code> file with the embedded Python code replaced by the output from executing it. The <code>.tmp.v</code> file is then used to build the simulation model.
<code>.tmp.v</code>	Temporary Verilog design files generated by the PyHP pre-processor from <code>.pyv.v</code> files. Python code embedded in a <code>.pyv.v</code> file is replaced by the output from executing it in the resulting <code>.tmp.v</code> .
<code>.h/.vh</code>	Verilog macro definition files.

.pyv.h/.pyv.vh	Verilog macro definition files with embedded python code. A .pyv.h/.pyv.vh file is run through the PyHP pre-processor prior to building simulation models to generate a .tmp.h/.tmp.vh with the embedded Python code replaced by the output from executing it. The .tmp.h/.tmp.vh file is then included from other Verilog design files and used in building the simulation model.
.tmp.h/.tmp.vh	Temporary Verilog macro definition files generated by the PyHP pre-processor from .pyv.h/.pyv.vh files. Python code embedded in a .pyv.h/.pyv.vh file is replaced by the output from executing it in the resulting .tmp.h/.tmp.vh.
Flist./flist	Verilog file lists. These are referenced from simulation model configuration files to determine which design files are required to build that model.
.diaglist	List of diags, assembly or C tests, which specify test parameters and make up sims regressions.
.s	Assembly file.
.c/.h	C implementation and header files.
.pal	PAL is a perl framework for generating randomized assembly tests. The .pal files are the source files.
.vmh/vmb	Hex/binary Verilog memory files.
.config	Configuration files for simulation models. These specify file lists needed to build a simulation model, default parameters, build and run arguments, etc.
.xml	XML files, generally used by contint to specify continuous integration bundles.
.py	Python scripts.
.log	Log files.
.image/.img	Memory image files.
.html	HTML files.

4 Environment Setup

This section discusses the environment setup for running simulations with OpenPiton. A script is provided, `piton/piton_settings.bash`, that does most of the work for you, however, there are a few environment variables that must be set first. Below are a list of steps to setup the OpenPiton environment for simulation.

1. The `PITON_ROOT` environment variable should point to the root of the OpenPiton package
2. The Synopsys environment for simulation should be setup separately by the user. Besides adding correct paths to your `PATH` and `LD_LIBRARY_PATH` environment variables and including the Synopsys license file or your license server in the `LM_LICENSE_FILE` environment variable (usually accomplished by a script provided by Synopsys or your system administrator), the OpenPiton tools specifically reference the `VCS_HOME` environment variable which should point to the root of the Synopsys VCS installation.

- **Note:** Depending on your system setup, Synopsys tools may require the `-full64` flag. This can easily be accomplished by adding a bash function as shown in the following example for VCS (also required for URG):

```
function vcs() { command vcs -full64 "$@"; };  
export -f vcs
```

3. Similarly to the Synopsys environment, the Cadence and ModelSim environments must also be set up separately by the user. The `NCV_HOME` environment variable should point to the root of the Cadence Incisive Simulator installation. `MODELSIM_HOME` environment variable should point to the root of the Mentor ModelSim installation.
4. The environment variable `ICARUS_HOME` should point to the level above the `bin` and `lib` folders containing the installation files for Icarus Verilog. If the Icarus executable `iverilog` is accessible at `/usr/bin/iverilog`, then `ICARUS_HOME` should point to `/usr`.
5. (OPTIONAL) The Xilinx environment for FPGA prototyping and generating Xilinx IP must be setup separately by the user, similar to the Synopsys environment. In gen-

eral, Xilinx or your system administrator should provide a script for doing this. In particular, the `XILINX` environment variable must point to the root of the Xilinx ISE installation in order for OpenPiton to use any Xilinx tools and/or IP. This is mainly relevant to Section ?? of this document for simulation using Xilinx IP simulation models, but is more pertinent to topics discussed in the OpenPiton FPGA Manual. This setup is not necessary if no Xilinx tools or IP are used (if you don't plan to use FPGA implementations).

6. (OPTIONAL) In order to run C tests in OpenPiton, a GCC compiler targeting the SPARC V9 architecture must be used to compile the tests. Currently, this compiler is not released with OpenPiton. Thus, the `PITON_GCC` environment variable must point to a GCC binary that targets the SPARC V9 architecture. Please contact us at openpiton@princeton.edu or on the OpenPiton Google group if you need help or more information on setting this up.
7. Run `"source $PITON_ROOT/piton/piton.settings.bash"` to setup the OpenPiton environment
 - **Note:** A CShell version of this script is provided, but OpenPiton has not been tested for and currently does not support CShell.

There are two environment variables set by the environment setup script that may be useful while working with OpenPiton:

- `DV_ROOT` points to `$PITON_ROOT/piton`
- `MODEL_DIR` points to `$PITON_ROOT/build`

5 Simulation

Running a simulation with OpenPiton requires two components: a simulation model and a test. This section will discuss how to build simulation models, how to run tests on simulation models, and how to use high-level simulation infrastructure, i.e. regressions and continuous integration bundles.

The `sims` tool is used to build models, run tests, and run regressions. It uses information from configuration files to setup a simulation environment and make calls to the Verilog simulator (e.g. Synopsys VCS). It may also call other tools (e.g.

PyHP preprocessor, compiler, assembler, test generation scripts, etc.) in order to compile tests into the proper format or perform other tasks. `sims` outputs log files, temporary configuration files, and results files to the current working directory by default, therefore it is recommended that you call `sims` from within the `$PITON_ROOT/build` directory to keep all temporary/generated files in one place. The manpage for `sims` is provided in Appendix A of this document for convenience.

The `contint` tool is used to run continuous integration bundles. It operates similarly to `sims` and ultimately calls `sims` to compile simulation models and run tests. More details on `contint` will be discussed in Section 5.5. The manpage for `contint` is provided in Appendix B of this document for convenience.

5.1 Simulation Models

A simulation model is made up from a set of design under test (DUT) Verilog files, a set of top-level Verilog files which create a testbench environment, a list of Verilog file lists (Flists) which specify the DUT Verilog files as well as the top-level testbench Verilog files, and a list of Verilog simulator (e.g. Synopsys VCS) command line arguments. This is all used to eventually call the Verilog simulator (e.g. Synopsys VCS) to compile a simulation executable and supporting files to a simulation model directory, `$MODEL_DIR/<simulation_model_dir>` (`MODEL_DIR` defaults to `$PITON_ROOT/build`), where `<simulation_model_dir>` is the name of the simulation model. Within the simulation model directory, specific instances of that model are built into a directory with a build ID, by default this is set to `rel-0.1/`, but may be overridden using the `-build_id=NAME` argument to `sims` (for more details see the `sims` man page in Appendix A).

The DUT Verilog files are generally located within the `$PITON_ROOT/piton/design` directory, while the top-level testbench Verilog files are located within `$PITON_ROOT/piton/verif/env/<simulation_model_dir>`, where `<simulation_model_dir>` is the name of the simulation model. This clearly separates synthesizable Verilog from non-synthesizable Verilog. Flists, which are simply lists of Verilog files, are generally co-located in the same directory as the Verilog files they list. A simulation model specifies a list of Flists, which in aggregate specify all Verilog files needed for that model, in a `sims` configuration file located in `$PITON_ROOT/piton/tools/src/sims/`. The name of the configuration

Table 3: OpenPiton Simulation Models

Name	Type
manycore	C/Assembly
chip_fpga_bridge	Unit Test
dmbr	Unit Test
dmbr_test	Unit Test
fpga_chip_bridge	Unit Test
fpga_fpga_hpc_bridge	Unit Test
fpga_fpga_lpc_bridge	Unit Test
host_fpga_comm	Unit Test
ifu_esl	Unit Test
ifu_esl_counter	Unit Test
ifu_esl_fsm	Unit Test
ifu_esl_htsm	Unit Test
ifu_esl_lfsr	Unit Test
ifu_esl_rtsm	Unit Test
ifu_esl_shiftreg	Unit Test
ifu_esl_stsm	Unit Test
jtag_testbench	Unit Test
memctrl_test	Unit Test
sdctrl_test	Unit Test
uart_serializer	Unit Test

file for a given simulation model name is `<simulation_model_name>.config`. This configuration file also lists the Verilog simulator (e.g. Synopsys VCS) command line arguments and default `sims` command line arguments.

5.1.1 Types of Simulation Models

OpenPiton supports two different types of simulation models: assembly/C test simulation models, namely the `manycore` simulation model, and unit test simulation models. Table 3 lists the OpenPiton simulation models and their type. Rather evident from their names, the assembly/C test simulation model builds a model of at least one core with surrounding infrastructure such that assembly and C tests may be run on it, while the unit test simulation models directly test a small, specific portion of the design using input and output vectors. Both types of models are compiled the same way, however, require different types of tests when running a simulation using the model. The details

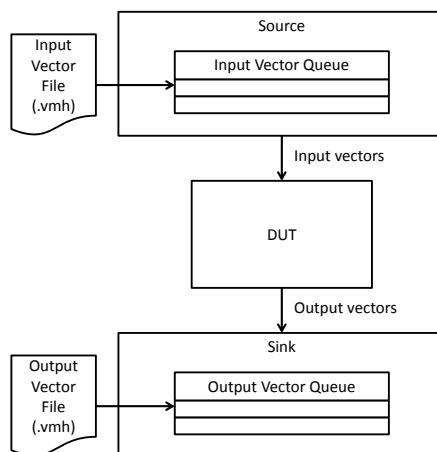


Figure 2: Unit testing infrastructure source-sink model

of the **manycore** simulation model will not be discussed, as it is quite complex and provides quite a bit of configurability and flexibility already. Therefore, we do not expect users to have to create a new assembly/C test simulation model or modify the **manycore** simulation model. If your use case of OpenPiton does require this, please post to the OpenPiton Google group or email openpiton@princeton.edu for questions and/or advice.

The **manycore** simulation model is currently the only simulation model to support assembly/C tests. It creates a 2D mesh of OpenPiton tiles which represents a single OpenPiton chip. For more details on the OpenPiton architecture, please refer to the OpenPiton Microarchitecture Manual. The number of tiles in each dimension is configurable, with a maximum of 256 in each dimension (limited by core addressability). The sizes of caches and other structures in the design is configurable in addition to other parameters within the core, i.e. thread count, presence of an FPU or SPU, etc. There are plans to support multi-chip simulation models in a future release to make 1/2 billion cores realizeable (using per core addressability and chip addressability space).

A unit testing framework is provided for the unit test simulation models in `$PITON_ROOT/piton/verif/env/test_infstrct`. In general, the existing OpenPiton unit test simulation models use this testing infrastructure. The OpenPiton unit testing infrastructure follows a source-sink model, as shown in Figure 2. Essentially, a source Verilog module provides new input vectors to the DUT on every cycle and a sink Verilog module checks

the output vectors from the DUT on every cycle against an expected value. The input vectors and expected output vectors are supplied through `.vmh/.vmb` files, which are read into the source/sink's input/output vector queue. The `.vmh/.vmb` files are lists of input/output vectors, where each line represents an entry into the source/sink's input/output vector queues. Thus, each line represents an input vector or expected output vector for a given unit testing simulation cycle. Consequently, tests for unit testing simulation models are specified by the names of these `.vmh/.vmb` files. The `.vmh/.vmb` files are commonly located in `$PITON_ROOT/verif/env/<simulation_model_name>/test_cases` and are loaded into the simulation model at run-time. This allows for many different source-sink `.vmh/.vmb` file pairs testing different parts of the design to be run on the same simulation model.

5.1.2 Building a Simulation Model

All simulation models are built the same way, using the `sims` tool. In general, the simulation model name is specified through the `-sys=NAME` argument. This along with the `-vcs_build` argument instructs `sims` to build the simulation model using the Synopsys VCS Verilog simulator. Build commands for VCS, Incisive, ModelSim, and Icarus Verilog are shown below:

- Synopsys VCS: `sims -sys=<simulation_model_name> -vcs_build`
- Cadence Incisive: `sims -sys=<simulation_model_name> -ncv_build`
- Icarus Verilog: `sims -sys=<simulation_model_name> -icv_build`
- Mentor ModelSim: `sims -sys=<simulation_model_name> -msm_build`

where `<simulation_model_name>` is the name of the simulation model you wish to build. Figure 3 shows the main steps invoked by this command. First, the results directory is setup (defaults to `$PWD`, but can be changed with the `-results_dir=PATH` option to `sims`). This is where the logs and simulation results are stored. Next, the `sims` configuration files are found through the `sims` master configuration file, `$PITON_ROOT/piton/tools/src/sims.config` by default (can be changed with the `-sims_config=FILE` option to `sims`), and are parsed to determine the valid simulation model to build and the configuration parameters for it. The

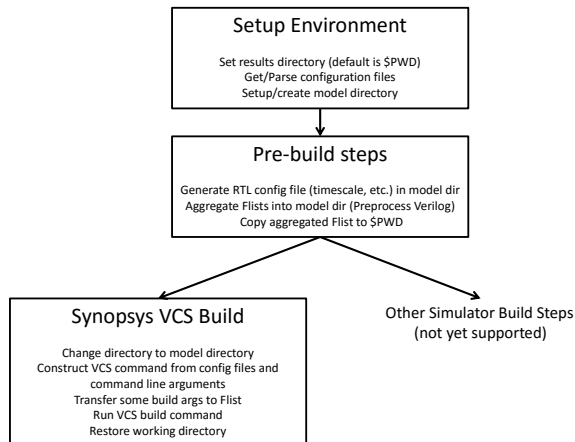


Figure 3: **sims** Simulation Model Build Flow/Steps

final environment setup step is to create and setup the model directory (defaults to `$MODEL_DIR`, but can be changed with the `-model_dir=PATH` option to **sims**). This is the directory where models will be built into and stored for multiple uses.

After the environment is setup, **sims** performs a few pre-build steps. This includes generating a RTL configuration file. This file is always included when building the model and define things like the Verilog `timescale` directive for the whole simulation model. **sims** also aggregates all of the Flists specified in the configuration file into one large flist in the model directory and then copies it into the current working directory. As **sims** aggregates the Flists together, it automatically detects files that need to be preprocessed by PyHP (via the file extension) and runs the pre-processor to generate the temporary design file to be used when building the simulation model. Any other Verilog preprocessing also occurs here. Lastly, some of the command line arguments are merged into the Flist to simplify the Verilog simulator invocation command.

The last step is to invoke the Verilog simulator to build the simulation model. This is the point where different simulators and their command line arguments can be used. First, **sims** changes the current working directory to the directory in which the model is to be built (determined by `-model_dir=PATH`, `-sys=NAME`, and `-build_id=NAME` arguments). Next, the simulator command is constructed based on the simulation model configuration file (all RTL design files, simulator command line arguments, etc.) and any **sims** command line arguments. Finally, the Verilog simula-

tor is called to build the simulation model. For Synopsys VCS, this produces a `simv` simulation executable in the simulation model directory. The executable can be called with a test and various command line arguments to run a simulation, discussed in Section 5.2. After the simulation model is built, the current working directory is restored to its previous location.

It is also worth noting that any call to `sims` generates a `sims.log` file in the current working directory containing a duplicate of everything printed to `stdout` during the execution of that command. In addition, `sims` maintains a history file, `history.sims` of all commands executed from that directory. These files can be useful in working with `sims` in general.

While there are many `sims` command line arguments to allow for configuring and controlling the simulation model build process, the RTL design, etc., for most simulation models, providing the `-sys=NAME` and `-vcs_build/-ncv_build/-icv_build` arguments is all that is needed. One other argument worth pointing out for VCS simulation is `-debug_all`, which allows for simulations to be run with the DVE GUI (using the `-gui` argument to the simulation run command) for debugging purposes. Please refer to the `sims` manpage in Appendix A for a detailed description of each command line argument. The `manycore` simulation model provides many configurability options and is thus discussed in the next section.

5.1.3 Configuring the `manycore` Simulation Model

The `manycore` simulation model is quite configurable, and, thus, has some `sims` command line arguments that can be used when `-sys=manycore` is specified. Specifically, the number of tiles can be modified and different types of IP block simulation models can be used.

5.1.3.1 Configuring the Number of Tiles The `-x_tiles` and `-y_tiles` options are used to specify the number of tiles in the x-dimension and y-dimension, respectively, of the 2D tile mesh created by the `manycore` simulation model. The default is `-x_tiles=1` and `-y_tiles=1`, which builds a single tile OpenPiton chip. The maximum value for both options is 256, due to the addressing space reserved in the NoC routers for cores. The number of cores in the system can be further expanded by

connecting multiple OpenPiton chips together, however, support for this will be available in a future release.

5.1.3.2 Configuring Cache Parameters All cache modules including L1 instruction/data, L1.5 and L2 caches can be configured on both cache size and associativity. This can be done by specifying the following switches either on the `sims` command line or in the config file (eg. `manycore.config`).

- `-config_l1i_size` and `-config_l1i_associativity`
- `-config_l1d_size` and `-config_l1d_associativity`
- `-config_l15_size` and `-config_l15_associativity`
- `-config_l2_size` and `-config_l2_associativity`

All these parameters need to be set as powers of 2, otherwise they may cause incorrect cache behavior. As an example, to override the L2 cache to be 32KB with 8-way associativity, add `-config_l2_size=32768` and `-config_l2_associativity=8` to the build command, eg.:

```
sims -vcs_build -sys=manycore -config_l2_size=32768 -  
config_l2_associativity=8
```

Only the default cache configs are exhaustively tested, though the other configs should work with most tests in the test suite. A few things to keep in mind when applying other configs:

- Lowering L1I/D cache sizes/associativities below certain sizes is incompatible with the TLBs within the core.
- Certain tests in the regression suite will fail due to assumptions of cache line placements referencing the default cache sizes.

Although these configurations are only given as experimental features, please let us know if there are any bug or difficulty in changing the caches in the forum.

5.2 Running a Simulation

This section discusses how to run a simulation using a simulation model. Specifying simulation tests/stimuli is different for the different types of simulation models/tests, however other parameters to `sims` needed to run a simulation are the same.

The `-sys=NAME` is required to specify which simulation model is to be used, this is the same value used when building the simulation model. If you have multiple instances of the same simulation model built (possibly with different design parameters), the `-build_id=NAME` argument allows you to select between them. This argument defaults to `rel-0.1`. Note that the **manycore** simulation model requires the `-x_tiles` and `-y_tiles` arguments to be specified if they were specified when building the simulation model, however this is a special case. Other simulation models generally do not have required, model-specific, simulation run arguments. Along with the `-sys=NAME` argument and any other arguments required by the simulation model, the `-vcs_run/-ncv_run/-icv_run` arguments instruct **sims** to run a simulation using VCS/Incisive/ModelSim/Icarus respectively.

One other argument worth mentioning is `-gui`. This optional argument requires the `-debug_all` argument to be specified when building the simulation model, and instructs **sims** to run the simulation in the DVE GUI, which enables waveform viewing, breakpointing, signal tracing, etc. This has only been tested with VCS.

In the basic case, the last argument that must be supplied is the simulation stimuli, or the test.

5.2.1 Assembly Tests

The simulation stimuli, or test, for an assembly test is specified as simply the name of the assembly file corresponding to that test. The assembly file argument is specified as the first argument without an option identifier:

- VCS: `sims -sys=manycore -x_tiles=X -y_tiles=Y -vcs_run <assembly_test_file>`
- Incisive: `sims -sys=manycore -x_tiles=X -y_tiles=Y -ncv_run <assembly_test_file>`
- ModelSim: `sims -sys=manycore -x_tiles=X -y_tiles=Y -msm_run <assembly_test_file>`
- Icarus: `sims -sys=manycore -x_tiles=X -y_tiles=Y -icv_run <assembly_test_file>`

or using the `-asm_diag_name=NAME` argument:

- VCS: `sims -sys=manycore -x_tiles=X -y_tiles=Y -vcs_run -asm_diag_name=<assembly_test_file>`
- Incisive: `sims -sys=manycore -x_tiles=X -y_tiles=Y -ncv_run -asm_diag_name=<assembly_test_file>`
- ModelSim: `sims -sys=manycore -x_tiles=X -y_tiles=Y -msm_run -asm_diag_name=<assembly_test_file>`
- Icarus: `sims -sys=manycore -x_tiles=X -y_tiles=Y -icv_run -asm_diag_name=<assembly_test_file>`

For example, to run the assembly test `princeton-test-test.s` you would run the following command:

- VCS: `sims -sys=manycore -x_tiles=X -y_tiles=Y -vcs_run princeton-test-test.s`
- Incisive: `sims -sys=manycore -x_tiles=X -y_tiles=Y -ncv_run princeton-test-test.s`
- ModelSim: `sims -sys=manycore -x_tiles=X -y_tiles=Y -msm_run princeton-test-test.s`
- Icarus: `sims -sys=manycore -x_tiles=X -y_tiles=Y -icv_run princeton-test-test.s`

or

- VCS: `sims -sys=manycore -x_tiles=X -y_tiles=Y -vcs_run -asm_diag_name=princeton-test-test.s`
- Incisive: `sims -sys=manycore -x_tiles=X -y_tiles=Y -ncv_run -asm_diag_name=princeton-test-test.s`
- ModelSim: `sims -sys=manycore -x_tiles=X -y_tiles=Y -msm_run -asm_diag_name=princeton-test-test.s`
- Icarus: `sims -sys=manycore -x_tiles=X -y_tiles=Y -icv_run -asm_diag_name=princeton-test-test.s`

All of the provided assembly tests are located in `$PITON_ROOT/piton/verif/diag/assembly`. You can trivially locate one you would like to run, specify it to `sims` as above, and run a simulation of that test. There are many other arguments available when running assembly tests which control different parts of the simulation, i.e. number of threads, maximum simulation cycles, enabling/disabling of verification monitors, assembler arguments,

etc. More complex simulation run commands involving these types of arguments are discussed in Section 5.3.

5.2.2 C Tests

All C tests are located in `$PITON_ROOT/piton/verif/diag/c`. In addition, there is an assembly file associated with each C test in `$PITON_ROOT/piton/verif/diag/assembly/c`. The assembly file contains directives to the assembler which instruct it to invoke the compiler. Note that the `PITON_GCC` environment variable must be set to a GCC binary that targets the SPARC V9 architecture in order to be able to compile C tests in OpenPiton. The assembly directives point to the corresponding C files associated with this test. Specify the associated assembly file the same way assembly tests are specified in order to run the C test. For example, in order to run `factorial.c`, which has a corresponding `factorial.s`, you would run the following command:

- VCS: `sims -sys=manycore -x_tiles=X -y_tiles=Y -vcs_run factorial.s`
- Incisive: `sims -sys=manycore -x_tiles=X -y_tiles=Y -ncv_run factorial.s`
- ModelSim: `sims -sys=manycore -x_tiles=X -y_tiles=Y -msm_run factorial.s`
- Icarus: `sims -sys=manycore -x_tiles=X -y_tiles=Y -icv_run factorial.s`

Similar to assembly tests, there are many other arguments available when running C tests allowing for more complex simulations. These arguments apply to both assembly and C tests and are therefore discussed in Section 5.3.

5.2.3 Unit Tests

Unit tests that use the OpenPiton testing infrastructure are located within the simulation model directory for which the unit test applies, `$PITON_ROOT/piton/verif/env/<simulation_model_name>/test_cases`. As described in Section 5.1.1, unit tests are specified by the `.vmh/.vmb` Verilog memory files. There are generally two files in the test cases directory associated with each unit test, one for the source, `<unit_test_name>_src.vmh`, and one for the sink, `<unit_test_name>_sink.vmh`. The `<unit_test_name>`

is supplied as a plusarg to the Verilog simulator (e.g. Synopsys VCS), `+test_case=<unit_test_name>`. The testing infrastructure adds the corresponding suffix (`._src.vmh` or `._sink.vmh`) to load the source and sink memory files and run the test. In order to do this using `sims`, the `-sim_run_args=OPTION` is used. This option causes `sims` to pass the supplied `OPTION` directly to the Verilog simulator (e.g. Synopsys VCS). Thus, to run the `test_step` unit test for the `ifu_esl_counter` simulation model, which has the `test_step_src.vmh` and `test_step_sink.vmh` files located within `$PITON_ROOT/piton/verif/env/ifu_esl_counter/test_cases`, you would run the following command:

- VCS: `sims -sys=ifu_esl_counter -vcs_run -sim_run_args=+test_case=test_step`
- Incisive: `sims -sys=ifu_esl_counter -ncv_run -sim_run_args=+test_case=test_step`
- ModelSim: `sims -sys=ifu_esl_counter -msm_run -sim_run_args=+test_case=test_step`
- Icarus: `sims -sys=ifu_esl_counter -icv_run -sim_run_args=+test_case=test_step`

5.2.4 `sims` Simulation Run Flow/Steps

The steps invoked when running a simulation with `sims` are depicted in Figure 4. The initial environment setup steps for running a simulation are mostly identical to that of building a model. However, it is not necessary to create and setup a model directory, as it is assumed a model is already built.

After the initial setup, the assembler and/or compiler must be called in order to assemble/compile assembly and C tests. Unit tests skip this step. `sims` first locates the assembly file specified at the command line to run the assembly or C test. The `-asm_diag_root=PATH` argument specifies where `sims` should look for the assembly file and defaults to `$PITON_ROOT/piton/verif/diag` for the `manycore` model. Once found, `sims` copies the assembly file to the current working directory as `diag.s` and extracts any `sims` command line arguments designated by `!SIMS+ARGS:` in the assembly file. The assembler command is constructed based on `sims` command line options and configuration files and is executed, providing `diag.s` as input. Note, as mentioned in Section 5.2.2, C tests are run by referencing their associated as-

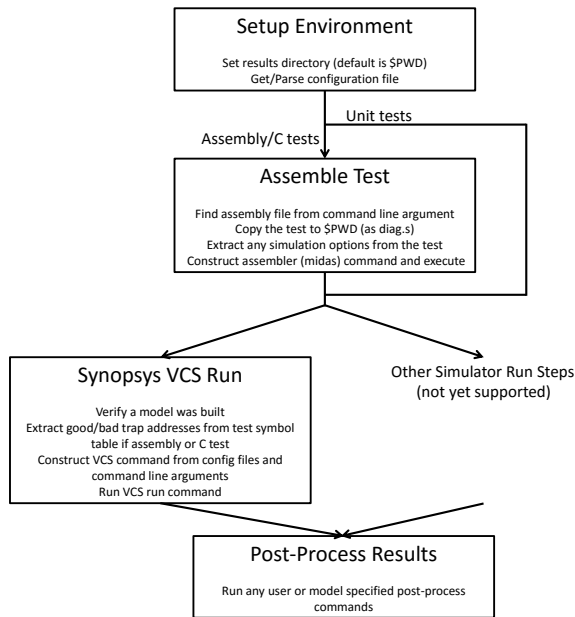


Figure 4: `sims` Simulation Model Run Flow/Steps

sembly file which contains directives to the assembler to call the C compiler. Thus, the above process is the same for both assembly and C tests.

Running the assembler generates a number of files in the current working directory. A few are worth pointing out:

- `diag.exe` - an ELF formatted binary of the test
- `mem.image` - a memory image with the test code and associated infrastructure (boot code, interrupt handlers, etc.) embedded at correct addresses
- `midas.log` - a log of the assembler run
- `symbol.tbl` - the symbol table for the test.

The next step is to invoke the Verilog simulator to run the simulation. In the case of Synopsys VCS, this consists of calling the `simv` executable compiled when building the model with a number of command line arguments. First, `sims` verifies that the specified simulation model has been built (`simv` exists for Synopsys VCS). The next step is to extract the addresses of the good and bad trap handler routines from the test symbol table. This is another step that is skipped for unit tests. Good and bad traps are special types of traps that are used in tests to indicate

a pass or fail. The addresses are passed to the simulation model and are used to determine when the test is finished and whether it passed or failed.

Finally, the Verilog simulation command line is constructed based on the configuration files and any `sims` command line options. For assembly and C tests, the good and bad trap handler addresses and the assembly test are passed to the Verilog simulator, although the test is actually read into the simulator from the `mem.image` file generated by the assembler. For unit tests, the source and sink memory files to be used are passed directly from the `sims` command line to the simulator command line through the `-sim_run_args=OPTION` argument. Lastly, the command is executed to kick off the simulation, logging the output to `sim.log`.

After the simulation completes, any user or model specified post-processing commands are run. For instance, the `manycore` simulation model, i.e. for assembly and C tests, has two post-processing scripts by default. The `manycore` simulation model specifies in its configuration script to run `perf` and `regreport` through the `-post_process_cmd=COMMAND` argument. The former extracts the performance from the simulation log to `perf.log` and the latter extracts the pass/fail status of the test to `status.log`. These steps are not necessary but provide nice summaries of what happened in a test. Additional post-processing steps can be added by the user either via the command line or configuration file.

As always, the `sims` command for running a simulation generates a `sims.log` file which logs all output during the execution of the command and a `history.sims` file which logs the history of `sims` commands executed from a directory.

5.3 Running Advanced Simulations Using the `manycore` Simulation Model

While the previous sub-section discussed running rather trivial simulations, this section goes into running more advanced simulations with the `manycore` simulation model. The unit testing simulation models are generally rather trivial and do not have many simulation run options other than which unit test is to be run. Therefore, unit tests simulation models will not be discussed in this sub-section.

As mentioned previously, there are many potential arguments to

`sims` simulation run commands for assembly and C tests, most of them dictated by the `manycore` simulation model top-level testbench code. In this section we will discuss some of these arguments and how to construct more complex `sims` simulation commands. The `sims` manpage is provided in Appendix A of this document for convenience and details all `sims` arguments.

5.3.1 Specifying Number of Threads and Thread Mapping for a Simulation

For most of the multi-threaded tests in our test suite, you can specify the total number of software threads and the mapping from software threads to physical hardware thread units (and thus cores) for a simulation. By default each core contains two hardware thread units in OpenPiton, therefore each core can be mapped with up to two threads. The total number of software threads can be configured by adding `-midas_args=-DTHREAD_COUNT=thread_count` into simulation run commands. By default, thread mapping starts with the first core in an incremental order. E.g. if the number of threads is set to 4 and there are two hardware thread units per core, the default mapping will map those 4 threads to the first two cores. Thread mapping can be changed to a regular strided pattern by adding `-midas_args=-DTHREAD_STRIDE=stride_number` to the simulation run command. The stride number defines the number of hardware thread units that are skipped between two neighboring threads. It is set to 1 by default meaning no skipping. E.g. if the number of threads is 2 and the stride number is 2, those 2 threads will be mapped to the first hardware thread unit of the first core and the first hardware thread unit of the second core. Arbitrary thread mappings can be managed by adding the `-midas_args=-DTHREAD_MASK=thread_mask` option to the simulation run command. After you specify those numbers, you also need to set the argument `-finish_mask=mask_vector` to notify which hardware thread units should read good trap for the test to be considered passing. The finish mask is a bit vector in hex. Previously there were four hardware threads in an OpenSPARC T1 core, each corresponding to one bit in the finish mask. By default in OpenPiton, we reduce the number of hardware threads to two per core, but we still leave the bit position for the other two removed threads for better configurability (Notice: there are no unused bit positions left for `-DTHREAD_MASK`). Therefore, in the above example of 4 threads mapped into the first two cores, the finish mask should be set to 33. In another example of 2 threads with a stride number of 2, the finish mask

should be set to 11. More complex examples are shown below:

Running 32 threads on 16 cores:

```
-midas_arg=-DTHREAD_COUNT=32 -finish_mask=3333333333333333
```

Running 16 threads on 16 cores, allocating one thread to each core (one to each of the first hardware thread units):

```
-midas_arg=-DTHREAD_COUNT=16 -midas_args=-DTHREAD_STRIDE=2  
-finish_mask=1111111111111111
```

Running 2 threads on 16 cores, allocating one on the first core and the other on the last core:

```
-midas_arg=-DTHREAD_COUNT=16 -midas_args=-DTHREAD_MASK=40000001  
-finish_mask=1000000000000001
```

5.3.2 Specifying Monitor Arguments for a Simulation

There are several monitors for different components of the OpenPiton chip. Each monitor sets a number of rules and any violation will trigger a test failure. Some monitors can be turned off for a simulation. E.g. The execution unit monitor can be turned off by adding `-sim_run_args=+turn_off_exu_monitor=1` in the run command. The monitor for L2 cache can be disabled by adding `-sim_run_args=+disable_l2_mon` into simulation run commands. This is useful when testing special ASI functions of L2 or you want to stop displaying L2 monitor information during simulations.

5.3.3 Debugging Simulations with `sims`

The default `manycore` simulation outputs a lot of useful information into the simulation log, which includes register updates, L1.5 and L2 cache pipelines, NoC messages and memory accesses. This information can be helpful to debug simple errors, especially memory system related errors. More comprehensive debugging process can be done by checking detailed waveforms via Synopsys DVE tool. This can be enabled by adding `-debug_all` flag for simulation model build and `-gui` flag for simulation run. As yet, this has only been tested with Synopsys VCS.

5.4 Running a Regression Suite

A regression is a set of simulations/tests which run on the same simulation model. The regression suite can run all tests with

Table 4: OpenPiton Regression Suites

Name	Description
all_tile1_passing	All single tile tests
tile1_mini	a mini set of single tile tests
all_tile2_passing	All 2-tile tests
tile2_mini	a mini set of 2-tile tests
tile4	All 4-tile tests
tile8	All 8-tile tests
tile16	All 16-tile tests
tile36	All 36-tile tests
tile64	All 64-tile tests

a single run command and generate a summarized report for all tests. Table 4 lists major regression suites that can be run in OpenPiton. For a complete list of all regression suite, please check the file `$PITON_ROOT/piton/verif/diag/master_diaglist_princeton`.

New regression groups can be created by modifying `$PITON_ROOT/piton/verif/diag/master_diaglist_princeton`, which defines the regressions. A regression is defined in this file by a XML tag, where tests listed between the opening and closing tags are part of that regression. The name of the regression is defined by the text in the XML tag followed by the simulation model to be used for the regression and any default `sims` arguments for all tests in the regression in the opening tag. Tests are listed between the open and closing tags in the format `<test_id> <sims_args>`, where `<test_id>` is a unique ID for that test in the regression and `<sims_args>` define the arguments to `sims` to run that test. A simple regression definition is shown below. It defines a regression called `princeton_test` with regression-wide `sims` arguments `-sys=manycore -x_tiles=1 -y_tiles=1` and includes a single test with the ID `princeton-test-test` and `sims` arguments `princeton-test-test.s`. While this test definition is simple, as the only `sims` argument is the test assembly file, multiple `sims` arguments can be included in the `sims_args` section of the test definition.

```
<princeton-test -sys=manycore -x_tiles=1 -y_tiles=1>
    princeton-test-test      princeton-test-test.s
</princeton-test>
```

In order to run a regression, specify the `-group=<regression_name>` and `-sim_type=<simulator>` argument to `sims`, where `<regression_name>` is the name of the regression defined by the XML tags in `$PITON_ROOT/piton/verif/diag/master_diaglist_princeton`. `sim_type=vcs` specifies that Synopsys VCS should be used for the simulation, which takes the place of `-vcs_build` and `-vcs_run` arguments when building and running a normal test. Similarly, `-sim_type=ncv` specifies Cadence Incisive, `-sim_type=msm` specifies Mentor ModelSim and `-sim_type=icv` specifies Icarus Verilog. The `-slurm -sim_q_command=sbatch` arguments can also be provided to the regression run command to launch each test in a SLURM batch job, allowing for tests to be run in parallel. Below is an example of how to run the `tile1_mini` regression group.

- VCS: `sims -sim_type=vcs -group=tile1_mini`
- Incisive: `sims -sim_type=ncv -group=tile1_mini`
- ModelSim: `sims -sim_type=msm -group=tile1_mini`
- Icarus: `sims -sim_type=icv -group=tile1_mini`

The simulation model will be built and all simulations will be run sequentially (unless the SLURM arguments are provided). In addition to the simulation model directory, a regression directory will be created in the `$MODEL_DIR` directory in the form `<date>_<id>` which contains the simulation results. The `<id>` field is just a counter that starts at zero and is incremented for each regression run on that `<date>`. In order to process the results from each test of the regression and obtain a summary of the regression results, the `regreport` command should be used, providing the regression directory (`$MODEL_DIR/<date>_<id>`) as an argument as below.

```
cd $MODEL_DIR/<date>_<id>
regreport $PWD > report.log
```

5.5 Running a Continuous Integration Bundle

Continuous integration bundles are sets of simulations, regression groups, and/or unit tests. The simulations within a bundle are not required to have the same simulation model, which is the main advantage of continuous integration bundles over regression groups. The continuous integration tool requires a job queue manager (e.g. SLURM) to be present on the system in order to parallelize simulations. As yet, the continuous integration tool

Table 5: OpenPiton Continuous Integration Bundles

Name	Description
git_push	a compact set of tests designed to run for every git commit
git_push_lite	a light version of git_push with fewer tests
nightly	a complete set of tests designed to run every night
pal_tests	a set of PAL tests
all_tile1_passing	All single tile tests
tile1_mini	a mini set of single tile tests
all_tile2_passing	All 2-tile tests
tile2_mini	a mini set of 2-tile tests
tile4	All 4-tile tests
tile8	All 8-tile tests
tile16	All 16-tile tests
tile36	All 36-tile tests
tile64	All 64-tile tests

only supports Synopsys VCS. Table 5 lists major bundles that can be run in OpenPiton.

New bundles can be created by adding XML files in the path `$PITON_ROOT/piton/tools/src/contint/`. A bundle usually consists of multiple regression groups. The format is XML based and quite straightforward, please refer to the `contint` README at `$PITON_ROOT/piton/tools/src/contint/README` and/or our existing XML files such as `git_push.xml` for the detailed format.

In order to run a continuous integration bundle, specify the `--bundle=<bundle_name>` argument to `contint`, where `<bundle_name>` refers to the name of the continuous integration bundle you would like to run, specified in the XML files. Below is an example of how to run the `git_push` bundle.

```
contint --bundle=git_push
```

All jobs will be submitted to the SLURM job scheduler by default. After all simulation jobs complete the results will be aggregated and printed to the screen. The individual simulation results will be stored in a new directory in the form `contint-<bundle name>_<date>_<id>` and can be reprocessed later to view the aggregated results again. This can be done with the `--check_results`

and `--contint_dir=contint_<bundle name>_<date>_<id>` arguments to `contint`. The `--bundle=<bundle_name>` argument must also be provided when re-checking results.

The exit code of `contint` indicates whether all tests passed (zero exit code) or at least one test failed (non-zero exit code). This can be useful for using `contint` in a continuous integration framework like Jenkins.

5.6 Determining Test Coverage

Coming Soon. This documentation will be included in a future release. Please email openpiton@princeton.edu or post to the OpenPiton discussion groups if you have questions on this topic.

A sims manpage

sims - Verilog rtl simulation environment and regression script

SYNOPSIS

sims [args ...]
where args are:

NOTE: Use "=" instead of "space" to separate args and their options.

SIMULATION ENV

-sys=NAME

sys is a pointer to a specific testbench configuration to be built and run. a config file is used to associate the sys with a set of default options to build the testbench and run diagnostics on it. the arguments in the config file are the same as the arguments passed on the command line.

-group=NAME

group name identifies a set of diags to run in a regression. The presence of this argument indicates that this is a regression run. the group must be found in the diaglist. multiple groups may be specified to be run within the same regression.

-group=NAME -alias=ALIAS

this combination of options gets the diag run time options from the diaglist based on the given group and alias. the group must be found in the diaglist. the alias is made up of diag_alias:name_tag. only one group should be specified when using this command format.

OPENPITON ARGUMENTS

-sys=manycore -x_tiles=X -y_tiles=Y

this combination of options for the "manycore" simulation model specifies a 2D

mesh topology of tiles, with X tiles in the x dimension and Y tiles in the y dimension. If -x_tiles and -y_tiles is not specified, the default is X=1 and Y=1. The maximum value for both X and Y is 1024.

-ed_enable

enable Execution Drafting in each core.

-ed_sync_method=SYNC_METHOD

sets the Execution Drafting thread synchronization method (TSM) to SYNC_METHOD. Possible values for SYNC_METHOD are "rtsm", "stsm", or "htsm". The default is "stsm". Please refer to the Execution Drafting paper or OpenPiton documentation for more information on TSMs.

-ibm

use simulation models from the IBM SRAM compiler. These are not provided with the OpenPiton download, but if the user has access to download them, there is infrastructure for them to be dropped in and used. Please refer to the OpenPiton documentation for more information on this option.

-xilinx

use simulation models from Xilinx IP, e.g. BRAMS, clock gen, etc., to simulate the FPGA version of OpenPiton. The Xilinx IP is not provided with the OpenPiton download, but if the user has access to download them, there is infrastructure for them to be dropped in and used. If you are planning to synthesize OpenPiton to an FPGA, it recommended to use this option for simulation. Please refer to the OpenPiton documentation for more information on this option.

-ml605

use block memories generated by ISE tools, required for ML605 evaluation board. Can be used only in conjunction with -xilinx option.

`-artix7`

use block memories generated by Vivado tool chain, required for Artix7 evaluation board. Can be used only in conjunction with `-xilinx` option.

`-vc707`

use block memories generated by Vivado tool chain, required for Xilinx VC707 evaluation board. Can be used only in conjunction with `-xilinx` option.

`-debug_all`

a shortcut for `-vcs_build_args=-debug_all`. In Synopsys VCS, this causes the simulation model to be built with the `-debug_all` flag. This allows for the simulation to be run in the DVE environment, convenient for waveform viewing and debugging.

`-gui`

a shortcut for `-sim_run_args=-gui`. In Synopsys VCS, this causes the simulation to be run within the DVE environment, convenient for waveform viewing and debugging. When building the simulation model specified by the `-sys` option, the `-debug_all` argument must have been passed to `sims`.

`-slurm -sim_q_command=sbatch`

specifies simulations should be submitted with the Simple Linux Utility for Resource Management (SLURM) and run in parallel. The `-sim_q_command=sbatch` must also be specified. The `-jobcommand_name` argument may also be used to specify the job name.

VERILOG COMPILATION RELATED

`-sim_type=vcs/ncv/icv`

defines which simulator to use `vcs`, `ncverilog`, or `icarus`, defaults to `vcs`.

`-sim_q_command="command"`

defines which job queue manager command to use to launch jobs. Defaults to /bin/sh and runs simulation jobs on the local machine.

`-ncv_build/-noncv_build`

builds a ncverilog model and the vera testbench. defaults to off.

`-ncv_build_args=OPTION`

ncverilog compile options. multiple options can be specified using multiple such arguments.

`-icv_build/-noicv_build`

builds an icarus model and the vera testbench. defaults to off.

`-icv_build_args=OPTION`

icarus compile options. multiple options can be specified using multiple such arguments.

`-vcs_build/-novcs_build`

builds a vcs model and the vera testbench. defaults to off.

`-vcs_build_args=OPTION`

vcs compile options. multiple options can be specified using multiple such arguments.

`-clean/-no_clean`

wipes out the model directory and rebuilds it from scratch. defaults to off.

`-vcs_use_2state/-novcs_use_2state`

builds a 2state model instead of the default 4state model. this defaults to off.

`-vcs_use_initreg/-novcs_use_initreg`

initialize all registers to a valid state (1/0). this feature works with `-tg_seed` to set the seed of the random initialization. this defaults to off.

`-vcs_use_fsdb/-novcs_use_fsdb`

use the debussy fsdb pli and include the dump calls in the testbench. this defaults to on.

`-vcs_use_vcsd/-novcs_use_vcsd`
 use the vcs direct kernel interface to dump out debussy files. this defaults to on.

`-vcs_use_vera/-novcs_use_vera`
 compile in the vera libraries. if `-vcs_use_ntb` and `-vcs_use_vera` are used, `-vcs_use_ntb` wins. this defaults to off.

`-vcs_use_ntb/-novcs_use_ntb`
 enable the use of NTB when building model (simv) and running simv. if `-vcs_use_ntb` and `-vcs_use_vera` are used, `-vcs_use_ntb` wins. this defaults to off.

`-vcs_use_rad/-novcs_use_rad`
 use the `+rad` option when building a vcs model (simv). defaults to off.

`-vcs_use_sdf/-novcs_use_sdf`
 build vcs model (simv) with an sdf file. defaults to off.

`-vcs_use_radincr/-novcs_use_radincr`
 use incremental `+rad` when building a vcs model (simv). defaults to off. this is now permanently disabled as synopsys advises against using it.

`-vcs_use_cli/-novcs_use_cli`
 use the `+cli` `-line` options when building a vcs model (simv). defaults to off.

`-flist=FLIST`
 full path to flist to be appended together to generate the final verilog flist. multiple such arguments may be used and each flist will be concatenated into the final verilog flist used to build the model.

`-graft_flist=GRAFTFILE`

GRAFTFILE is the full path to a file that lists each verilog file that will be grafted into the design. the full path to the verilog files must also be given in the GRAFTFILE.

`-vfile=FILE`

verilog file to be included into the flist

`-config_rtl=DEFINE`

each such parameter is place as a 'define in config.v to configure the model being built properly. this allows each testbench to select only the rtl code that it needs from the top level rtl file.

`-model=NAME`

the name of a model to be built. the full path to a model is

/tank/mmckeown/research/projects/piton/openpiton/build/model

`-build_id=NAME`

specify the build id of the model to be built. the full path to a model is

/tank/mmckeown/research/projects/piton/openpiton/build/mode

VERA COMPILATION RELATED

VERA and NTB share all of the vera options except a few. See NTB RELATED.

`-vera_build/-novera_build`

builds the vera/ntb testbench. default on.

`-vera_clean/-novera_clean`

performs a make clean on the vera/ntb testbench before building the model. defaults to off.

`-vera_build_args=OPTION`

vera testbench compile time options. multiple options can be specified using multiple such commands. these are passed as arguments to the gmake call when building the vera testbench.

`-vera_diag_args=OPTION`

vera/ntb diag compile time option multiple options can be specified using multiple such arguments.

`-vera_dummy_diag=NAME`

this option provides a dummy vera diag name that will be overridden if a vera diag is specified, else used for vera diag compilation

`-vera_pal_diag_args=OPTION`

vera/ntb pal diag expansion options (i.e. "pal OPTIONS -o diag.vr diag.vrpal") multiple options can be specified using multiple such arguments.

`-vera_proj_args=OPTION`

vera proj file generation options. multiple options can be specified using multiple such arguments.

`-vera_vcon_file=ARG`

name of the vera vcon file that is used when running the simulation.

`-vera_cov_obj=OBJ`

this argument is passed to the vera Makefile as a OBJ=1 and to vera as -DOBJ to enable a given vera coverage object. multiple such arguments can be specified for multiple coverage objects.

NTB RELATED

NTB and VERA share all of the vera options except these:

`-vcs_use_ntb/-novcs_use_ntb`

enable the use of NTB when building model (simv). if `-vcs_use_ntb` and `-vcs_use_vera` are used, `-vcs_use_ntb` wins. defaults to off.

`-ntb_lib/-nontb_lib`

enables the NTB 2 part compile where the Vera/NTB files get compiled first into a libtb.so file which is dynamically loaded by

vcs at runtime. The libtb.so file is built by the Vera Makefile, not sims. Use the Makefile to affect the build. If not using -ntb_lib, sims will build VCS and NTB together in one pass (use Makefile to affect that build as well). default is off.

VERILOG RUNTIME RELATED

-vera_run/-novera_run

runs the vcs simulation and loads in the vera proj file or the ntb libtb.so file. defaults to on.

-vcd/-novcd

signals the bench to dump in VCD format

-vcdfile=filename

the name of the vcd dump file. if the file name starts with a "/", that is the file dumped to, otherwise, the actual file is created under 'tmp_dir/vcdfile' and copied back to the current directory when the simulation ends. use "-vcdfile='pwd'/filename" to force the file to be written in the current directory directly (not efficient since dumping is done over network instead of to a local disk).

-vcs_run/-novcs_run

runs the vcs simulation (simv). defaults to off.

-sim_run_args=OPTION

sim runtime options. multiple options can be specified using multiple such arguments.

-vcs_finish=TIMESTAMP

forces vcs to finish and exit at the specified timestamp.

-fast_boot/-nofast_boot

speeds up booting when using the ciop model. this passes the +fast_boot switch to the simv

run and the `-sas_run_args=-DFAST_BOOT` and `-midas_args=-DFAST_BOOT` to sas and midas. Also sends `-DFAST_BOOT` to the diaglist and config file preprocessors.

`-debussy/-nodebussy`

enable debussy dump. this must be implemented in the testbench to work properly. defaults to off.

`-start_dump=START`

start dumping out a waveform after START number of units

`-stop_dump=STOP`

stop dumping out a waveform after STOP number of units

`-fsdb2vcd`

runs fsdb2vcd after the simulation has completed to generate a vcd file.

`-fsdbfile=filename`

the name of the debussy dump file. If the file name starts with a "/", that is the file dumped to, otherwise, the actual file is created under 'tmp_dir/fsdbfile and copied back to the current directory when the simulation ends. Use `"-fsdbfile='pwd'/filename"` to force the file to be written in the current directory directly (not efficient since dumping is done over network instead of to a local disk).

`-fsdbDumplimit=SIZE_IN_MB`

max size of Debussy dump file. minimum value is 32MB. Latest values of signal values making up that size is saved.

`-fsdb_glitch`

turn on glitch and sequence dumping in fsdb file. this will collect glitches and sequence of events within time in the fsdb waveform. beware that this will cause the fsdb file size to grow significantly this is turned off by

default. this option effectively does this:
this is turned off by default. this option
effectively does this:

```
setenv FSDB_ENV_DUMP_SEQ_NUM 1
setenv FSDB_ENV_MAX_GLITCH_NUM 0
```

-rerun

rerun the simulation from an existing
regression run directory.

-post_process_cmd=COMMAND

post processing command to be run after vcs
(simv) run completes

-pre_process_cmd=COMMAND

pre processing command to be run before vcs
(simv) run starts

-use_denalirc=FILE

use FILE as the .denalirc in the run area.
Default copies 'env_base/.denalirc'

VLINT OPTIONS

-vlint_run/-novlint_run

runs the vlint program. defaults to off.

-vlint_args

vlint options. The <sysName>.config file
can contain the desired vlint arguments,
or they can also be given on the command
line. Typically the -vlint_compile is
given on the command line.

vlint also requires identification of a
rules deck.

-illust_run

run illust after x2e

-illust_args

illust options

-vlint_top

top level module on which to run vlint

VERIX OPTIONS

- verix_run/-noverix_run
runs the verix program. defaults to off.
- verix_libs
specify the library files to add to the vlist
- verix_args
verix template options. The <sysName>.config file can contain these desired verix arguments

verix also requires <top>.verix.tmplt in the config dir.
- verix_top
top level module on which to run verix

ZEROIN RELATED

- zeroIn_checklist
run 0in checklist
- zeroIn_build
build 0In pli for simulation into vcs model
- zeroInSearch_build
build 0in search pli for simulation into vcs model
- zeroIn_build_args
additional arguments to be passed to the 0in command
- zeroIn_dbg_args
additional debug arguments to be passed to the 0in shell

SAS/SIMICS RELATED

- sas/-nosas
run architecture-simulator. If vcs_run option is OFF, simulation is sas-only. If vcs_run

option is ON, sas runs in lock-step with rtl.
default to off.

-sas_run_args=DARGS
Define arguments for sas.

TCL/TAP RELATED

-tcl_tap/-notcl_tap
run tcl/expect TAP program. If vcs_run option
is OFF, simulation is tcl-only. If vcs_run
option is ON, tcl runs in lock-step with rtl.
default to off.
NOTE: You must compile with -tcl_tap as
well, to enable to enable functions that are
needed for running with tcl

-tcl_tap_diag=diagname
Define top level tcl/expect diag name.

MIDAS

midas is the diag assembler

-midas_args=DARGS
arguments for midas. midas creates memory
image and user-event files from the assembly
diag.

-midas_only
Compile the diag using midas and exit without
running it.

-midas_use_tgseed
Add -DTG_SEED=tg_seed to midas command line.
Use -tg_seed to set the value passed to midas
or use a random value from /dev/random.

PCI

pci is the pci bus functional model

-pci_args
arguments to be passed in to pci_cmdgen.pl for
generation of a pci random diagnostic.

`-pci/-nopci`
generates a random pci diagnostic using the
`-tg_seed` if provided. default is off.

`-tg_seed`
random generator seed for pci random test
generators also the value passed to `+initreg+`
to randomly initialize registers when
`-vcs_use_initreg` is used.

SJM

`sjm` is the Jbus bus functional model

`-sjm_args`
arguments to be passed in to `sjm_tstgen.pl` for
generation of an sjm random diagnostic.

`-sjm/-nosjm`
generates a random sjm diagnostic using the
`-tg_seed` if provided. default is off.

`-tg_seed`
random generator seed for sjm random test
generators also the value passed to `+initreg+`
to randomly initialize registers when
`-vcs_use_initreg` is used.

EFCGEN

`efcgen.pl` is a script to generate `efuse.img` files
(default random), which is used by the efuse controller
after reset. It is invoked by `-efc`.

`-efc/-noefc`
generates an efuse image file using the
`-tg_seed` if provided. default is off. Random
if no `-efc_args` specified.

`-efc_args`
arguments to be passed in to `efcgen.pl` for
generation of an efuse image file. Default is
random efuse replacement for each block.

`-tg_seed`
random generator seed for efcgen.pl script
also the value passed to +initreg+ to randomly
initialize registers when -vcs_use_initreg is
used.

VCS COVERMETER

`-vcs_use_cm/-novcs_use_cmd`
passes in the -cm switch to vcs at build time
and simv at runtime default to off.

`-vcs_cm_args=ARGS`
argument to be given to the -cm switch

`-vcs_cm_cond=ARGS`
argument to be given to the -cm_cond switch.

`-vcs_cm_config=ARGS`
argument to be given to the -cm_hier switch

`-vcs_cm_fsmcfg=ARGS`
argument to be given to the -cm_fsmcfg switch
specifies an FSM coverage configuration file

`-vcs_cm_name=ARGS`
argument to be given to the -cm_name switch.
defaults to cm_data.

DFT

`-dftvert`
modifies the sims flow to accomodate dftvert.
this skips compiling the vera testbench and
modifies the simv command line at runtime.

MISC

`-nobuild`
this is a master switch to disable all building
options. there is no such thing as -build to
enable all build options.

`-copyall/-nocopyall`

copy back all files to launch directory after passing regression run. Normally, only failing runs cause a copy back of files. Default is off.

`-copydump/-nocopydump`

copy back dump file to launch directory after passing regression run. Normally, only failing runs cause a copy back of non-log files. The file copied back is `sim.fsdb`, or `sim.vcd` if `-fsdb2vcd` option is set. Default is off.

`-tarcopy/-notarcopy`

copy back files using 'tar'. This only works in `copyall` or in the case the simulations 'fails' (per sims' determination). Default is to use 'cp'.

`-diag_pl_args=ARGS`

If the assembly diag has a Perl portion at the end, it is put into `diag.pl` and is run as a Perl script. This allows you to give arguments to that Perl script. The arguments accumulate, if the option is used multiple times.

`-pal_use_tgseed`

Send '`-seed=<tg_seed_value>`' to pal diags. Adds `-pal_diag_args=-seed=tg_seed` to midas command line, and `-seed=tg_seed` to pal options (`vrpal` diags). Use `-tg_seed` to set the value passed to midas or use a random value from `/dev/random`.

`-parallel`

when specifying multiple groups for regressions this switch will submit each group to Job Q manager to be executed as a separate regression. This has the effect of speeding up regression submissions. NOTE: This switch must not be used with `-injobq`

`-reg_count=COUNT`

runs the specified group multiple times in regression mode. this is useful when we want to run the same diag multiple times using a different random generator seed each time or

some such.

`-regress_id=ID`

specify the name of the regression

`-report`

This flag is used to produce a report of a an old or running regression. With `-group` options, `sims` produces the report after the regression run. Report for the previous regression run can be produced using `-regress_id=ID` option along with this option,

`-finish_mask=MASK`

masks for vcs simulation termination. Simulation terminates when it hits 'good_trap' or 'bad_trap'. For multithread simulation, simulation terminates when any of the thread hits bad_trap, or all the threads specified by the finish_mask hits the good_trap. example: `-finish_mask=0xe` Simulation will be terminated by good_trap, if thread 1, 2 and 3 hits the good_trap.

`-stub_mask=MASK`

mask for vcs simulation termination. Simulation ends when the stub driving the relevant bit in the mask is asserted. This is a hexadecimal value similar to `-finish_mask`

`-wait_cycle_to_kill=VAL`

passes a `+wait_cycle_to_kill` to the `simv` run. a testbench may chose to implement this `plusarg` to delay killing a simulation by a number of clock cycles to allow collection of some more data before exiting (e.g. waveform).

`-rtl_timeout`

passes a `+TIMEOUT` to the `simv` run. sets the number of clock cycles after all threads have become inactive for the diag to exit with an error. if all threads hit good trap on their own the diag exits right away. if any of the threads is inactive without hitting good trap/bad trap the `rtl_timeout` will be reached

and the diag fails. default is 5000. this is only implemented in the cmp based testbenches.

-max_cycle

passes a +max_cycle to the simv run. sets the maximum number of clock cycle that the diag will take to complete. the default is 30000. if max_cycle is hit the diag exits with a failure. not all testbenches implement this feature.

-norun_diag_pl

Does not run diag.pl (if it exists) after simv (vcs) run. Use this option if, for some reason, you want to run an existing assembly diag without the Perl part that is in the original diag.

-nosaslog

turns off redirection of sas stdout to the sas.log file. use this option when doing interactive runs with sas.

-nosimslog

turns off redirection of stdout and stderr to the sims.log file. use this option to get to the cli prompt when using vcs or to see a truncated sim.log file that exited with an error. this must be used if you want control-c to work while vcs is running.

-nogzip

turns off compression of log files before they are copied over during regressions.

-version

print version number.

-help

prints this

IT SYSTEM RELATED

-dv_root=PATH

absolute path to design root directory. this overrides DV_ROOT.

-model_dir=PATH
absolute path to model root directory. this overrides MODEL_DIR.

-tmp_dir=PATH
path where temporary files such as debussy dumps will be created

-sims_config=FILE
full path to sims config file

-env_base=PATH
this specifies the root directory for the bench environment. it is typically defined in the bench config file. It has no default.

-config_cpp_args=OPTION
this allows the user to provide CPP arguments (defines/undefines) that will be used when the testbench configuration file is processed through cpp. Multiple options are concatenated together.

-result_dir=PATH
this allows the regression run to be launched from a different directory than the one sims was launched from. defaults to
/tank/mmckeown/research/projects/piton/piton_master/docs/si

-diaglist=FILE
full path to diaglist file

-diaglist_cpp_args=OPTION
this allows the user to provide CPP arguments (defines/undefines) that will be used when the diaglist file is processed through cpp. Multiple options are concatenated together.

-asm_diag_name=NAME
-tpt_diag_name=NAME
-tap_diag_name=NAME
-vera_diag_name=NAME

```

-vera_config_name=NAME
-efuse_image_name=NAME
-image_diag_name=NAME
-sjm_diag_name=NAME
-pci_diag_name=NAME
    name of the diagnostic to be run.

-asm_diag_root=PATH
-tpt_diag_root=PATH
-tap_diag_root=PATH
-vera_diag_root=PATH
-vera_config_root=PATH
-efuse_image_root=PATH
-image_diag_root=PATH
-sjm_diag_root=PATH
-pci_diag_root=PATH
    absolute path to diag root directory. sims
    will perform a find from here to find the
    specified type of diag. if more than one
    instance of the diag name is found under root
    sims exits with an error. this option can be
    specified multiple times to allow multiple
    roots to be searched for the diag.

-asm_diag_path=PATH
-tpt_diag_path=PATH
-tap_diag_path=PATH
-vera_diag_path=PATH
-vera_config_path=PATH
-efuse_image_path=PATH
-image_diag_path=PATH
-sjm_diag_path=PATH
-pci_diag_path=PATH
    absolute path to diag directory. sims expects
    the specified diag to be in this directory.
    the last value of this option is the one used
    as the path.

```

ClearCase

```

-clearcase
    assume we are in ClearCase environment for
    setting DV_ROOT and launching Job Q manager
    commands. default is off.

```

```

-noclearcase
    force clearcase option off

-cc_dv_root=PATH
    ClearCase path to design root directory. this
    overrides .

```

ENV VARIABLES

sims sets the following ENV variables that may be used with pre/post processing scripts, and other internal tools:

```

ASM_DIAG_NAME : Contains the assembly diag name.
SIMS_LAUNCH_DIR : Path to launch directory where
sims is running the job.
VERA_LIBDIR    : Dir where Vera files are compiled.

DV_ROOT       : -dv_root if specifed
MODEL_DIR     : -model_dir if specified
TRE_SEARCH    : Based on -use_iver, -use_sims_iver
DENALI        : User defined
VCS_HOME      : User defined
VERA_HOME     : User defined

```

PLUSARGS

+args are not implemented in sims. they are passed directly to vcs at compile time and simv at runtime. the plusargs listed here are for reference purposes only.

```

+STACK_DIMM 32 bits physical address space - default
is 31 bits

+STACK_DIMM +RANK_DIMM 33 bits physical address
space - default is 31 bits

+max_cycle see -max_cycle

+TIMEOUT see -rtl_timeout

+vcs+finish see -vcs_finish

+wait_cycle_to_kill see -wait_cycle_to_kill

```

DESCRIPTION

sims is the frontend for vcs to run single simulations and regressions

HOWTO

Build models

Build a vcs model using DV_ROOT as design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build
```

Build a ncverilog model using DV_ROOT as design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -ncv_build
```

Build an icarus model using DV_ROOT as design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -icv_build
```

Build the vera testbench only using DV_ROOT as design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vera_build
```

Build a model from any design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build  
-dv_root=/home/regress/2002_06_03
```

Build a graft model from any design root

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build  
-dv_root=/model/2002_06_03  
-graft_flist=/regress/graftfile
```

Build a model and re-build the vera

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build  
-vera_clean
```

Build a model and turn off incremental compile

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build  
-clean
```

Build a model with a given name

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_build  
-build_id=mymodel
```

Run models

Run a diag with default model

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -vcs_run  
diag.s
```

Run a diag with a specified model

```
sims -sys=manycore -x_tiles=1 -y_tiles=1  
-build_id=mymodel -vcs_run diag.s
```

Run a diag with debussy dump with default model

```
sims -sys=manycore -x_tiles=1 -y_tiles=1 -debussy  
+dump=cmp_top:0 -vcs_run diag.s
```

Run regressions

Run a regression using DV_ROOT as design root

```
sims -group=tile1_mini
```

Run a regression using DV_ROOT as design root and
specify the diaglist

```
sims -group=tile1_mini -diaglist=/home/user/my_dialist
```

Run a regression using any design root

```
sims -group=tile1_mini  
-dv_root=/import/design/regress/model/2002_06_03
```

Run a regression using any design root and a graft model

```
sims -group=tile1_mini  
-dv_root=/regress/model/2002_06_03
```



```
-graft_flist=/home/regress/graftfile
```

B contint manpage

```
Usage: contint --bundle=<continuous integration bundle>
[options]
```

```
Options:
```

-h, --help	Print this usage message
--dryrun	Don't actually run, print commands
--check_results	Do not run simulations, just check results
--contint_dir=<dir>	Specify a name for the continuous integration run directory
--cleanup	Remove run directories and model directories when finished if all tests pass
--inverse	Inverts the exit code to return 0 if all tests failed and 1 otherwise, whereas the default is to return 0 if all tests pass and 1 otherwise.

C Ubuntu 14.04/16.04 Dependencies and Workarounds

C.1 Dependencies

Here is a inclusive, but possibly not be minimal, list of software packages for a clean Ubuntu 14.04/16.04 installation:

```
sudo apt-get install git csh libc6-i386 lib32stdc++6
libstdc++5 libstdc++5:i386 lib32gcc1 lib32ncurses5
lib32z1 libjpeg62 libtiff5 build-essential
libbit-vector-perl libgmp3-dev
```

There are a few more dependencies that got deprecated in new Ubuntu distros, of which we will have to install outside of apt-get. Download these files:

```
http://packages.ubuntu.com/precise/i386/libgmp3c2/download
http://packages.ubuntu.com/precise/amd64/libmng1/download
```

Then install with the command:

```
dpkg -i ${downloaded\_libgmp3c2}.deb
dpkg -i ${downloaded\_libmng1}.deb
```

Last, Ubuntu’s “dash” might be incompatible with some Synopsys installations, so revert back to the classic “sh” binary:

```
sudo dpkg-reconfigure dash
```

And select “No.”

C.2 VCS simulation workaround

Both Ubuntu 14.04/16.04 come with GCC versions that do not do circular dependency search when linking static libraries, which will cause compilation errors like belows:

```
libvcsnew.so: undefined reference to
'snpsGroupGetActive'
libvcsnew.so: undefined reference to
'miHeapProf_init'
libvcsucli.so: undefined reference to
'printIclBeginMarker'
```

If you have the above errors, add the following line to `$PITON_ROOT/piton/tool`

```
-vcs_build_args=${VCS_DIR}/linux64/lib/libvcsnew.so
${VCS_DIR}/linux64/lib/libvcsucli.so
${VCS_DIR}/linux64/lib/libsnpsmalloc.so
${VCS_DIR}/linux64/lib/libvcsnew.so
```

Note: tested with VCS MX L-2016.06.

References

- [1] Sun Microsystems, Santa Clara, CA, *OpenSPARC T1 Processor Design and Verification User's Guide*, 2006.
- [2] PyHP, “PyHP Official Home Page.” <http://pyhp.sourceforge.net>.