



OpenPiton Synthesis and Back-end Manual

Wentzlaff Parallel Research Group

Princeton University

openpiton@princeton.edu

History of versions

Version	Date	Author(s)	Changes
1.0	04/03/16	MM	Initial version

Contents

1	Introduction	1
2	Supported Third-Party Tools and Environments	2
2.1	Job Queue Managers	2
2.2	EDA Tools	2
2.2.1	Verilog Pre-Processor	2
2.2.2	Synthesis	2
2.2.3	Static Timing Analysis	3
2.2.4	RTL to Netlist and Netlist to Netlist Equivalence Checking	3
2.2.5	Place and Route	3
2.2.6	GDSII Merge	4
2.2.7	Design Rule and Layout Versus Schematic Checking	4
2.2.8	Reference Methodology	4
3	Directory Structure and File Organization	7
3.1	Directory Structure	7
3.2	Common File Extensions/Naming Conventions . .	8
4	Environment Setup	9
5	OpenPiton Synthesis and Back-end Flow Overview	10
6	Patching Synopsys Reference Methodology to OpenPiton Flow	13
7	Process Technology Setup	14
8	Running the OpenPiton Synthesis and Backend Flow	17

8.1	Checking the OpenPiton Synthesis and Backend Results	18
9	Running a New Verilog Module Through the Flow	19
	References	20

List of Figures

1	OpenPiton Synthesis and Back-end Directory Structure	7
2	OpenPiton Synthesis and Back-end Flow Diagram	11

List of Tables

2	Common OpenPiton synthesis and back-end file extensions/naming conventions	8
3	Process specific synthesis and backend scripts. Referenced from <code>\${PITON_ROOT}/piton/ tools/synopsys/script/</code>	16
4	OpenPiton Synthesis and Back-end Flow Run Commands	17
5	OpenPiton Synthesis and Back-end Flow Supported Modules	18
6	OpenPiton Synthesis and Back-end Flow Results Locations. Referenced from module specific <code>synopsys</code> directory.	19

1 Introduction

This document discusses the OpenPiton synthesis and back-end infrastructure, including static timing analysis (STA), RTL to netlist and netlist to netlist equivalence checking (RVS, for RTL vs. schematic), place and route (PAR), design rule checking (DRC), and layout versus schematic checking (LVS).

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.

The synthesis and back-end support in OpenPiton is designed to enable users to run their modified OpenPiton designs through a publicly available tool flow for ASIC prototyping or area, power, and timing characterizations. It is meant to serve as both a reference and a starting point and eliminates the need to write boiler plate scripts. This enables the rapid development of tape-out or publication ready designs.

This document covers the following topics:

- Supported third-party tools and environments
- Directory structure and file organization
- OpenPiton environment setup
- Synthesis and back-end tool flow overview
- Porting to a specific process technology
- Running Steps of the flow
- Running a new Verilog module through the flow

2 Supported Third-Party Tools and Environments

This section discusses the different third-party tools/environments that are supported and/or required by OpenPiton. For supported operating systems (OSs), Unix shells and script interpreters, please see the OpenPiton Simulation Manual. This section lists tools specific to the synthesis and back-end scripts. This includes job queue managers and EDA tools.

2.1 Job Queue Managers

SLURM (Simple Linux Utility for Resource Management) is optional and all OpenPiton synthesis and back-end scripts support using it to submit batch jobs. Currently SLURM has been tested with version 15.08.8.

2.2 EDA Tools

2.2.1 Verilog Pre-Processor

OpenPiton uses the PyHP Verilog pre-processor (v1.12) 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 simply printing to stdout. The PyHP pre-processor executes the Python code and generates a Verilog file with the Python snippets replaced by their output on stdout. Verilog files intended to be pre-processed by PyHP (files with embedded python) are given the file extension `.pyv.v` or `.pyv.h` for define/include files. PyHP is distributed with the OpenPiton download. Any PyHP Verilog must be run through the PyHP pre-processor before being synthesized, which the OpenPiton scripts take care of.

2.2.2 Synthesis

OpenPiton uses Synopsys Design Compiler (DC) in Topographical mode for synthesis. The OpenPiton synthesis scripts have been tested with the following DC versions:

- syn_I-2013.12-SP4

We expect the scripts to work with other version with minimal modifications and hope to support more versions in the future. If

you find that OpenPiton synthesis is stable using another Synopsys DC version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.3 Static Timing Analysis

OpenPiton uses Synopsys Primetime (PT) for static timing analysis (STA). The OpenPiton STA scripts have been tested with the following PT versions:

- pt_J-2014.06

We expect the scripts to work with other version with minimal modifications and hope to support more versions in the future. If you find that OpenPiton STA is stable using another Synopsys PT version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.4 RTL to Netlist and Netlist to Netlist Equivalence Checking

OpenPiton uses Synopsys Formality (FM) for RTL to netlist and netlist to netlist equivalence checking (RVS, for RTL vs. schematic). The OpenPiton RVS scripts have been tested with the following FM versions:

- fm_J-2014.09-SP3

We expect the scripts to work with other version with minimal modifications and hope to support more versions in the future. If you find that OpenPiton RVS is stable using another Synopsys FM version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.5 Place and Route

OpenPiton uses Synopsys IC Compiler (ICC) for place and route (PAR). The OpenPiton PAR scripts have been tested with the following ICC versions:

- icc_I-2013.12-SP4

We expect the scripts to work with other version with minimal modifications and hope to support more versions in the future. If you find that OpenPiton PAR is stable using another Synopsys ICC version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.6 GDSII Merge

OpenPiton uses Synopsys IC Workbench Edit/View Plus (ICWBEV) for merging GDSII designs. The OpenPiton merge GDSII scripts have been tested with the following ICWBEV versions:

- icwbev_plus_J-2014.06

We expect the scripts to work with other version with minimal modifications and hope to support more versions in the future. If you find that OpenPiton merge GDSII is stable using another Synopsys ICWBEV version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.7 Design Rule and Layout Versus Schematic Checking

OpenPiton uses Mentor Graphics Calibre for design rule checking (DRC) and layout versus schematic checking (LVS). The OpenPiton DRC and LVS scripts have been tested with the following Calibre versions:

- ixl_cal_2013.2.35.25

We expect the scripts to work with other versions with minimal modifications and hope to support more versions in the future. If you find that OpenPiton DRC and LVS is stable using another Synopsys ICWBEV version, please let us know at openpiton@princeton.edu so we can update the list on our website.

2.2.8 Reference Methodology

The OpenPiton synthesis and back-end flow is based on the Synopsys Reference Methodology (RM). Because of IP issues, the OpenPiton synthesis and back-end scripts have been released as a patch to the Synopsys RM. Thus, users will need access to Synopsys RM in order to make use of the OpenPiton synthesis and back-end flow. The OpenPiton synthesis and back-end flow supports patching from the following versions and settings of the Synopsys RM:

- Synthesis
 - DC-RM-I-2013.12-SP2
 - * Settings:
 - RTL Source Format: VERILOG

- QoR Strategy: DEFAULT
- Physical Guidance: TRUE
- Hierarchical Flow: TRUE
- MCMM Flow: FALSE
- Multi-Voltage UPF: FALSE
- Clock Gating: TRUE
- Leakage Power: TRUE
- DFT Synthesis: FALSE
- Lynx Compatible: FALSE
- Static Timing Analysis
 - PT-RM.I-2013.12
 - * Settings:
 - Flow: PT
 - Back Annotation Mode: SBPF
 - Distributed Multi-Scenario Mode: FALSE
 - Multivoltage Scaling Mode: FALSE
 - UPF Mode: FALSE
 - Clock Gating and Threshold Voltage Group Reporting: TRUE
 - Power Analysis: OFF
 - Fix DRC ECO: TRUE
 - Fix Timing ECO: TRUE
 - Fix Setup: COMBINATIONAL
 - Fix Hold: COMBINATIONAL
 - Fix Leakage ECO: FALSE
 - Link to TetraMax: FALSE
 - Advanced OCV Mode File: FALSE
 - CRPR Mode: FALSE
 - Derate Mode: FALSE

- Save Session: FALSE
- Timing Models: GENERATION
- Hierarchical Model Type: BOTH
- Path-Based Analysis: FALSE
- Reports: ENHANCED
- Lynx Compatible: FALSE
- Place and Route
 - ICC-RM.I-2013.12-SP4
 - * Settings:
 - Focus on QoR or Feasibility: QOR
 - Two Pass place_opt Flow: DEFAULT
 - Multicorner-Multimode (MCMM) Optimization: FALSE
 - Multivoltage or Multisupply: NONE
 - Physical Guidance: TRUE
 - Flip Chip Design Style: FALSE
 - In-Design Rail Analysis: TRUE
 - Zroute: TRUE
 - Advanced Node: FALSE
 - Concurrent Clock and Data Optimization Flow: FALSE
 - Floorplan Style: DEFAULT
 - Design Style: DEFAULT
 - Flow Style: ODL
 - Lynx Compatible: FALSE

We plan to release patches for a larger variety of versions and settings to increase accessibility, so keep an eye out on www.openpiton.org for additional supported versions.

3 Directory Structure and File Organization

This section discusses the OpenPiton synthesis and back-end directory structure and common file extensions. It mainly discusses this in the context of the synthesis and back-end portion of the infrastructure. For a more general discussion, see the OpenPiton Simulation Manual.

3.1 Directory Structure

This section discusses the directory structure within the root directory of the OpenPiton download specific to the synthesis and back-end flow. For a discussion of other directories in the OpenPiton download, please see the OpenPiton Simulation Manual and/or OpenPiton FPGA Manual. Figure 1 shows the important directories in the OpenPiton synthesis and back-end flow directory structure.

```
piton/
├── design/
│   └── chip/
├── tools/
│   ├── calibre/
│   └── synopsys/
```

Figure 1: OpenPiton Synthesis and Back-end Directory Structure

The `piton/` directory is the only top-level directory relevant to the synthesis and back-end flow (no files are generated in the current working directory, so it is not necessary to use `build/`). All of the scripts for the synthesis and back-end flow are located in `piton/`. Within `piton/`, there are two relevant directories: `design/` and `tools/`.

The `design/` directory, as discussed in the OpenPiton Simulation Manual, houses all of the synthesizable Verilog RTL design files for OpenPiton. The `design/` directory is broken down into several sub-directories to organize RTL for different purposes (see OpenPiton Simulation Manual), but the only one pertinent to the synthesis and back-end scripts is `chip/`, as it contains the Verilog design files for an OpenPiton chip which will be synthesized and PAR'd. Within `chip/` the directory structure follows major points in the Verilog module design hierarchy.

The synthesis and back-end scripts specific to a module, such as floorplanning, power/ground network (PGN), and constraints scripts, are located in directory named `synopsys/` under the directory for that module. For example, the module specific synthesis and back-end scripts for the OpenSPARC T1 core are located in `piton/design/chip/tile/sparc/synopsys/` while the top-level RTL for the core (excluding sub-modules) is located in `piton/design/chip/tile/sparc/rtl/`. This logically fits with the `design/` directory's purpose, as the tasks these scripts perform are actually part of the design along with the Verilog RTL.

The `tools/` directory within `piton/`, as stated in the OpenPiton Simulation Manual, contains all of the scripts and tools used in OpenPiton, including the synthesis and back-end scripts that are agnostic to a module. These scripts are really what drive the flow and only call the synthesis and back-end scripts in the `design/` directory for module specific tasks or information. The module agnostic synthesis and back-end scripts are split into two subdirectories: `calibre/` for DRC and LVS scripts that use Mentor Graphics Calibre and `synopsys/` for all other tasks which use Synopsys tools.

3.2 Common File Extensions/Naming Conventions

Table 2 lists common file extensions and naming conventions used in the OpenPiton synthesis and back-end flow and explanations of the files they are used for:

Table 2: Common OpenPiton synthesis and back-end file extensions/naming conventions

File extension	Description
<code>.tcl</code>	TCL scripts. Call APIs provided by Synopsys tools' augmented TCL shells.
<code>.tpl</code>	Power and ground mesh template files for Synopsys IC Compiler. Called from TCL scripts to create PGNs.
<code>.excpt</code>	Exception file for synthesis and back-end flow checking scripts. Each line is a Python regular expression which creates an exception for an error or warning found in a log file.
<code>.sh</code>	Bash script.

block.list	List of blocks (Verilog modules) for synthesis and place and route. Maps a short name to a directory path and parameters for submitting SLURM jobs for different modules.
.v	Verilog design files.
.pyv.v	Verilog design files with embedded Python code. A .pyv.v file is run through the PyHP pre-processor prior to building simulation models, generating a .tmp.v file with the embedded Python code replaced by the output from executing it. The .tmp.v file is then used to build the simulation model.
.tmp.v	Temporary Verilog design files generated by the PyHP pre-processor from .pyv.v files. Python code embedded in a .pyv.v file is replaced by the output from executing it in the resulting .tmp.v.
.h/.vh	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.

4 Environment Setup

This section discusses the environment setup for the OpenPiton synthesis and backend flow. A script is provided, `piton/piton.settings.bash`, that does most of the work for you, however, some of the environment will need to be setup on your own. Below are a list of steps to setup the OpenPiton environment.

1. The `PITON_ROOT` environment variable should point to the

root of the OpenPiton package

2. Synopsys and Mentor Graphics tools (see Section 2) need to be in your `PATH` environment variable. Generally, this is accomplished through a script provided with the installation of the tools or by your system administrator. Note that you only need the tools on your path for the parts of the flow you plan to use. For example, if you only plan to use synthesis, you only need Synopsys Design Compiler in your `PATH` variable.
3. 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 OpenPiton Synthesis and Back-end Flow Overview

This section gives a high level overview of the OpenPiton synthesis and back-end flow. Figure 2 depicts the flow graphically. The flow is identical to that used to tapeout the Piton ASIC prototype, however references to the specific technology used have been removed, due to IP issues. This means that the flow will not run out of the box without modifications for a specific process, as discussed in Section 7. Despite this, the flow serves as a great reference and starting point for those who want to tapeout an ASIC with OpenPiton or want to do performance, area, and/or power characterization. In addition, all of the infrastructure for the flow is seamlessly integrated into OpenPiton (setup for running modules through the flow, integration into directory structure, etc.), making it easy to use and modify while using OpenPiton. We anticipate releasing a version of the tools that utilize the Synopsys 32/28nm technology library in the future so the flow references a process available to at least academic users.

The OpenPiton synthesis and back-end flow is based on the Synopsys Reference Methodology (RM). Because of IP issues, the

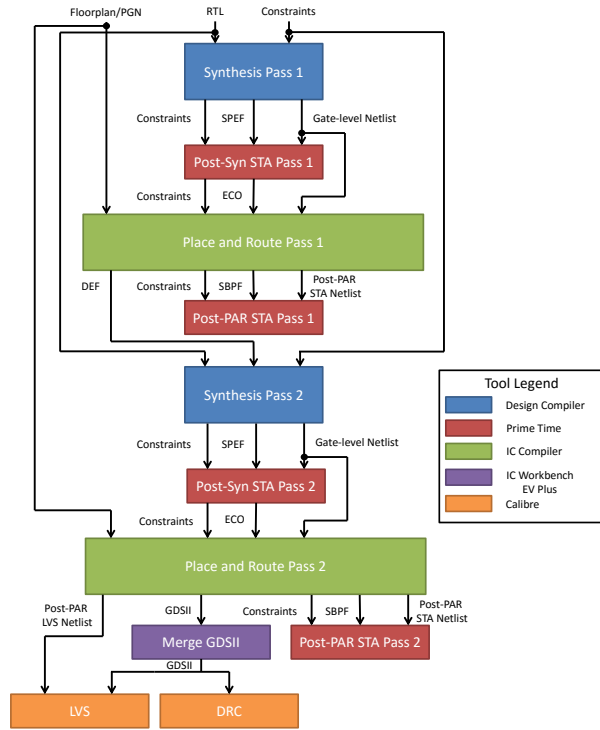


Figure 2: OpenPiton Synthesis and Back-end Flow Diagram

OpenPiton synthesis and back-end scripts have been released as a patch to the Synopsys RM flow. Thus, users will need access to the Synopsys RM, however this is most likely not a problem as the flow is mostly a Synopsys flow, so users will need a Synopsys license anyways. In addition, users will need to run an additional step, as described in Section 6, to apply a patch to the Synopsys RM scripts and end up with the OpenPiton flow. The Synopsys RM and tools are available for free to academic users through the Synopsys University Program.

Note also that some of the directory structure and script infrastructure for the OpenPiton synthesis and back-end flow is based on the synthesis flow released in the OpenSPARC T1 [1], although the flow itself is completely different (completely different synthesis scripts).

The OpenPiton synthesis and back-end flow is mainly a Synopsys tool flow. Figure 2 shows a two-pass flow, however the number of passes is configurable. Increasing the number of passes improves the quality of results, but with diminishing returns. The Verilog RTL along with design constraints are first passed to Synopsys

Design Compiler, which synthesizes a gate-level netlist from the behavioral RTL. The resulting netlist, along with the constraints, are passed to Synopsys PrimeTime to perform post-synthesis static timing analysis (STA). This analyzes the gate-level netlist against the constraints specified and reports the results. If the constraints are not met, Synopsys PrimeTime may output an engineering change order (ECO) file, which suggests modifications to the netlist to meet the constraints. Although not shown in the figure, in parallel to STA, Synopsys Formality is run, providing the initial RTL and synthesized RTL for equivalence checking (RVS, for RTL vs. schematic).

The gate-level netlist, constraints, ECO file, and physical floorplan (specified by the user) are passed to Synopsys IC Compiler to perform placement and routing (PAR). However, before PAR, the ECO modifications are applied to the gate-level netlist to give Synopsys IC Compiler a better chance of meeting the constraints. After PAR is complete, the post-PAR netlist is again passed to Synopsys PrimeTime, along with the constraints, to perform STA and check the design against the constraints. Although not shown in the figure, Synopsys PrimeTime may output an ECO file again, which can be fed back into Synopsys IC compiler, along with the post-PAR netlist and physical layout library to apply the ECO. We have found the ECOs from PrimeTime to be very useful in meeting timing. Also not shown in the figure, Synopsys Formality is run again to check the equivalence of the synthesized and physical netlists.

The output of this flow is a fully placed and routed design in GDSII format. If it meets the constraints, design rule checking (DRC) and layout versus schematic checking (LVS) are performed. However, it is likely the constraints may not be met after the first pass. In this case, results can be improved by performing the same flow up to this point a second time, while passing physical information from the output of the first IC Compiler pass and the ECO from the output of PrimeTime. Any number of passes through this flow is possible, but we saw diminishing returns after two passes. If constraints are still not met, it may be the case that the constraints and/or floorplan must be modified.

After a GDSII of the design that meets the constraints is obtained, it must be merged with the GDSII of any third-party IP to generate the final GDSII. This is done with the Synopsys IC Workbench EV Plus tool. After the final merged GDSII is gener-

ated, it is passed to Mentor Graphics Calibre for LVS and DRC checking. DRC requires the GDSII and the DRC deck from the process development kit. LVS only requires the GDSII and the post-PAR gate-level netlist.

If DRC and LVS are not met, there are a few things that can be done. For DRC, going back and changing the original design constraints or the floorplan can reduce DRC violations. This is particularly true if this is the first time a module is run through the flow. It is likely iteration is required to achieve a design that is not over constrained and can meet DRC. Alternatively, if the design must fit in the floorplan and constraints specified, DRC can be fixed manually in the Synopsys IC Compiler graphical user interface (GUI). Synopsys IC Compiler also has hooks to import your DRC results and highlight them in the GUI.

We found that LVS violations were generally not caused by the Synopsys tools performing a functionally incorrect mapping, but likely due to something the user has done or not thought about. Examples of this include name collisions between modules and LVS issues in third-party IP. It really requires diving into the LVS detailed report and finding where the root problem is. We have almost never found the design or Synopsys tools were the problem.

A final consideration for synthesis and back-end flow is what to use for the input RTL. If one were to give the full OpenPiton RTL as input to synthesis, the flow would take an unreasonable amount of time and may not even complete successfully. For this reason, the design is generally broken down into hierarchical blocks. The leaf blocks are run through the flow first, and imported as black-boxes in the flow for modules higher in the hierarchy. This is done until the top-level chip is reached. Since the hierarchy may depend on process technology, design modifications, etc. the OpenPiton synthesis and back-end scripts make it easy to modify and define new module hierarchies.

6 Patching Synopsys Reference Methodology to OpenPiton Flow

In order to patch the Synopsys RM flow to the OpenPiton flow, follow the steps below:

1. Ensure you have downloaded and extracted the correct version(s) of the Synopsys RM as discussed in Section 2.

2. Run the `synrm_patch` script with the `--dc_rm_path=<Path to DC RM>`, `--pt_rm_path=<Path to PT RM>`, and/or `--icc_rm_path=<Path to ICC RM>` options. You can specify anywhere from one to all of these, but `synrm_patch` will only copy and patch the scripts for the RMs specified. For example, if you only plan to run synthesis and STA, not PAR, you can pass only the `--dc_rm_path=<Path to DC RM>` and `--pt_rm_path=<Path to PT RM>` options to `synrm_patch`.
3. After `synrm_patch` completes successfully, your OpenPiton synthesis and back-end flow should be correctly setup. Note `synrm_patch` checks the integrity of both the input RMs and the resulting OpenPiton flow to ensure the final result is correct.

7 Process Technology Setup

The OpenPiton synthesis and backend scripts make it easy to port to a specific process technology. Below are a list of files that need to be changed for a new process technology:

- `${PITON_ROOT}/piton/tools/synopsys/script/common/env_setup.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/process_setup.tcl`
- `${PITON_ROOT}/piton/tools/calibre/script/common/calibre_env`

The `${PITON_ROOT}/piton/tools/synopsys/script/common/env_setup.tcl` script should be modified to get environment variables pointing to locations needed by the back-end flow. There are a few examples provided in the file, but this usually includes environment variables that point to directories for standard cell libraries, process design kits (PDK), etc.

`${PITON_ROOT}/piton/tools/synopsys/script/common/process_setup.tcl` is the main location where changes will need to be made to port the Synopsys portion of the flow to a new process. This file should use environment variables from `${PITON_ROOT}/piton/tools/synopsys/script/common/env_setup.tcl` to point to necessary files for the synthesis and back-end flow, including standard cell libraries, technology files, Milkyway libraries and other library and design information. The file has directives to

help you fill in the required variables, please see the script itself for more details.

For DRC and LVS which use Mentor Graphics Calibre, `${PITON_ROOT}/piton/tools/calibre/script/common/calibre_env` provides all of the process specific setup. Specifically, it provides paths to LVS and DRC decks for the process as well as any process specific environment variables needed for DRC or LVS. Changes to this file are only required if you plan to run DRC and/or LVS.

Setting up the above three files will allow you to run the flow through for modules without SRAMs, however it is likely other changes are required for the design to be appropriate for that process. For example, the allowed routing metal layers for the design, the clock frequency, the number of tiles, double via definitions, power and ground network (PGN) routing, metal layers used for different purposes, etc. These types of modifications, which are not required to get the flow running but are required to correctly implement a design for that process, are made in the following files:

- `${PITON_ROOT}/piton/tools/synopsys/script/common/design_setup.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/floorplan/common_pgn.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/floorplan/core_pgn_mesh.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/floorplan/common_post_floorplan.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/dbl_via_setup.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/pt_eco_drc_buf.tcl`
- `${PITON_ROOT}/piton/tools/synopsys/script/common/vt_group_setup.tcl`

Lastly, modifications may also need to be made to module specific synthesis and back-end scripts, listed below. The process specific portions of these scripts include allowed metal layers

Table 3: Process specific synthesis and backend scripts. Referenced from `${PITON_ROOT}/piton/ tools/synopsys/script/`

Process Specific File Path	Description
<code>common/env_setup.tcl</code>	Environment variable setup. Gets environment variables that are used throughout the scripts.
<code>common/process_setup.tcl</code>	Main process specific setup file. Includes standard cell libraries, technology files, layer mapping files, Milkyway libraries, etc.
<code>common/calibre_env</code>	Calibre process specific setup file. Sets the DRC and LVS decks and any environment variables used by these decks.
<code>common/design_setup.tcl</code>	Setup variables specific to a design. Includes things like clock frequency, allowed routing metal layers, etc.
<code>common/floorplan/common_pgn.tcl</code>	Preroutes the power and ground network (PGN) for modules without SRAMs or other IP macros. For modules with SRAMs or other IP macros, a module specific PGN script will be required.
<code>common/floorplan/core_pgn_mesh.tcl</code>	Templates for PGN meshes. Called from <code>common/floorplan/common_pgn.tcl</code> and module specific PGN scripts.
<code>common/floorplan/common_post_floorplan.tcl</code>	Post floorplan steps that are common to all modules. This is usually process specific but is also sometimes optional. We used this to place certain cells at a specific density throughout a module after floorplanning is complete, per a requirement from the foundry.
<code>common/dbl_via_setup.tcl</code>	Double via definition script. This is usually process specific but is also sometimes optional. Can define which vias are used for double via insertion.
<code>common/pt_eco_drc_buf.tcl</code>	This is also optional, but specifies a list of standard cell buffers that Synopsys PT can use for DRC fixing.
<code>common/vt_group_setup.tcl</code>	Optional, can group target libraries into threshold voltage groups for reporting. Can generate a report of what percentage of different groups were used, useful for knowing things like what portion of your critical path is made up of the lowest threshold voltage standard cells.

and SRAM IP macros for the design, including the necessary libraries.

- `${PITON_ROOT}/piton/design/chip/tile/sparc/ffu/synopsys/script/module_setup.tcl`
- `${PITON_ROOT}/piton/design/chip/tile/sparc/synopsys/script/module_setup.tcl`
- `${PITON_ROOT}/piton/design/chip/tile/dynamic_node/synopsys/script/module_setup.tcl`
- `${PITON_ROOT}/piton/design/chip/tile/synopsys/script/module_setup.tcl`

Table 3 lists all of the process specific synthesis and backend scripts we have discussed in this section along with a short description of the purpose of that file.

Table 4: OpenPiton Synthesis and Back-end Flow Run Commands

Command	Flow Step	Tool	Checking Script
<code>rsyn</code>	Synthesis	Synopsys Design Compiler	<code>csyn</code>
<code>rsta</code>	Static Timing Analysis	Synopsys Primetime	<code>csta</code>
<code>rrvs</code>	RTL vs. Schematic Equivalence Checking	Synopsys Formality	<code>crvs</code>
<code>rpar</code>	Place and Route	Synopsys IC Compiler	<code>cpar</code>
<code>reco</code>	Run ECO	Synopsys IC Compiler	<code>cpar</code>
<code>merge_gds</code>	Merge GDSII Designs	Synopsys IC Workbench Edit/View Plus	<code>cmerge_gds</code>
<code>rdrc</code>	Design Rule Checking	Mentor Graphics Calibre	<code>cdrc</code>
<code>rlvs</code>	Layout vs. Schematic Checking	Mentor Graphics Calibre	<code>clvs</code>
<code>rftf</code>	Full tool flow	All of the above	N/A

8 Running the OpenPiton Synthesis and Backend Flow

Now that you have the OpenPiton flow setup (correctly patched) and you have modified the scripts for porting to a specific process technology, you are ready to run the flow. Table 4 shows the difference commands used to run different steps of the OpenPiton synthesis and back-end flow, along with the tool that step uses and a corresponding script that can check the log files for that step and notify whether the run passed or failed. Note there are commands to run each individual step of the flow, as well as a single command to kick off the whole flow (run sequentially).

The checking scripts are provided for verifications that the step was run correctly. It does not notify of the result of that step, but only that the tool ran correctly without errors and warnings. For example, `cdrc` will not tell you if DRC passed, it will tell you if Mentor Graphics Calibre ran DRC without any errors and warnings. You will need to separately check the `reports/` directory inside the module specific `synopsys` directory for the module you are running through the flow (see Section 8.1). The checking scripts work by searching the tool log files for "error" and "warning". Along with the checking scripts are `.except` files, which list exceptions to these errors and warnings as a Python regular expression on each line. Both module specific and module agnostic `.except` files may be used to create exceptions (located in the corresponding `synopsys` directory). These checking scripts can be useful, as many of the tools will print many warnings that you may not care about. You may only care about differences in warnings between a valid run and your current run, which these tools help greatly with.

Both the tool flow run scripts and checking scripts accept a module nickname as input on the command line. The module nickname is mapped to the location of the module specific `synopsys` directory, along with some other parameters in `/${PITON_ROOT}/`

Table 5: OpenPiton Synthesis and Back-end Flow Supported Modules

Module Name	Description	Purpose
ffu	OpenSPARC T1 core floating-point front-end unit	Small module with one SRAM macro
sparc	OpenSPARC T1 core	Large module with many SRAM macros
dynamic_node	OpenPiton on-chip network router	Small module with no IP macros
tile	OpenPiton tile	Large, hierarchical module with many SRAM macros

`piton/tools/synopsys/block.list`. The number of passes the flow takes for a given module is also specified in this file. The blocks with module specific flow scripts provided are listed in Table 5. Note that the `ffu` and `dynamic_node` modules are provided as small standalone examples, but these modules were not run through the flow standalone when taping out the Piton chip. The hierarchical flow that was used to create the tile in Piton ran `sparc` through the flow, and then imported it hierarchically as a black box into the tile. The tile was then run through the flow flat, with the `sparc` core black-boxed inside. This hierarchical flow is provided along with the sample standalone `ffu` and `dynamic_node` modules (not black-boxed inside any other hierarchical designs). The top-level chip module for the Piton chip will be included in a future release.

Another command line argument that may be passed to the flow run scripts is `-slurm`. This will schedule the run step in the SLURM job queue manager. `rftf` will actually submit all individual steps of the flow individually to SLURM, setting dependencies correctly. This exposes more parallelism in the flow. If `-slurm` is not provided, the flow step(s) are run sequentially in the current Unix shell. We hope to support different job queue managers in the future, or potentially GNU Parallel, for greater accessibility.

8.1 Checking the OpenPiton Synthesis and Backend Results

Table 6 shows the locations where the results should be checked for each step of the flow. For example, where would you look for timing results after running STA? Or where would you find violations after running DRC? This information is all provided in the table and understanding the results should be rather straightforward. One note on `rlvs` is that the result file will actually contain two results. A result at the top of the file, and another result modway down the file (result will be obvious, it says "incorrect" or "correct". Always check the second result in the file. The first results will always be incorrect (in our experience).

Table 6: OpenPiton Synthesis and Back-end Flow Results Locations. Referenced from module specific `synopsys` directory.

Command	Results Location
<code>rsyn</code>	<code>reports/dc_shell, reports/dc_shell_pass*</code>
<code>rsta</code>	<code>reports/pt_shell, reports/pt_shell_dc_pass*, reports/pt_shell_icc_pass*</code>
<code>rrvs</code>	<code>reports/fm_shell, reports/fm_shell_dc_pass*, reports/fm_shell_icc_pass*</code>
<code>rpar</code>	<code>reports/icc_shell, reports/icc_shell_pass*</code>
<code>reco</code>	<code>reports/eco_shell, reports/echo_shell_pass*</code>
<code>merge_gds</code>	<code>results/</code>
<code>rdrc</code>	<code>reports/<design_name>.drc.summary</code> (bottom of file)
<code>rlvs</code>	<code>reports/<design_name>.lvs.report</code> (second result)

9 Running a New Verilog Module Through the Flow

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.

References

- [1] Sun Microsystems, Santa Clara, CA, *OpenSPARC T1 Processor Design and Verification User's Guide*, 2006.