Bitcoin - Case Study User Guide
Low Power Flow and Methodology

Version 1.1, September 2017

SYNOPSYS®

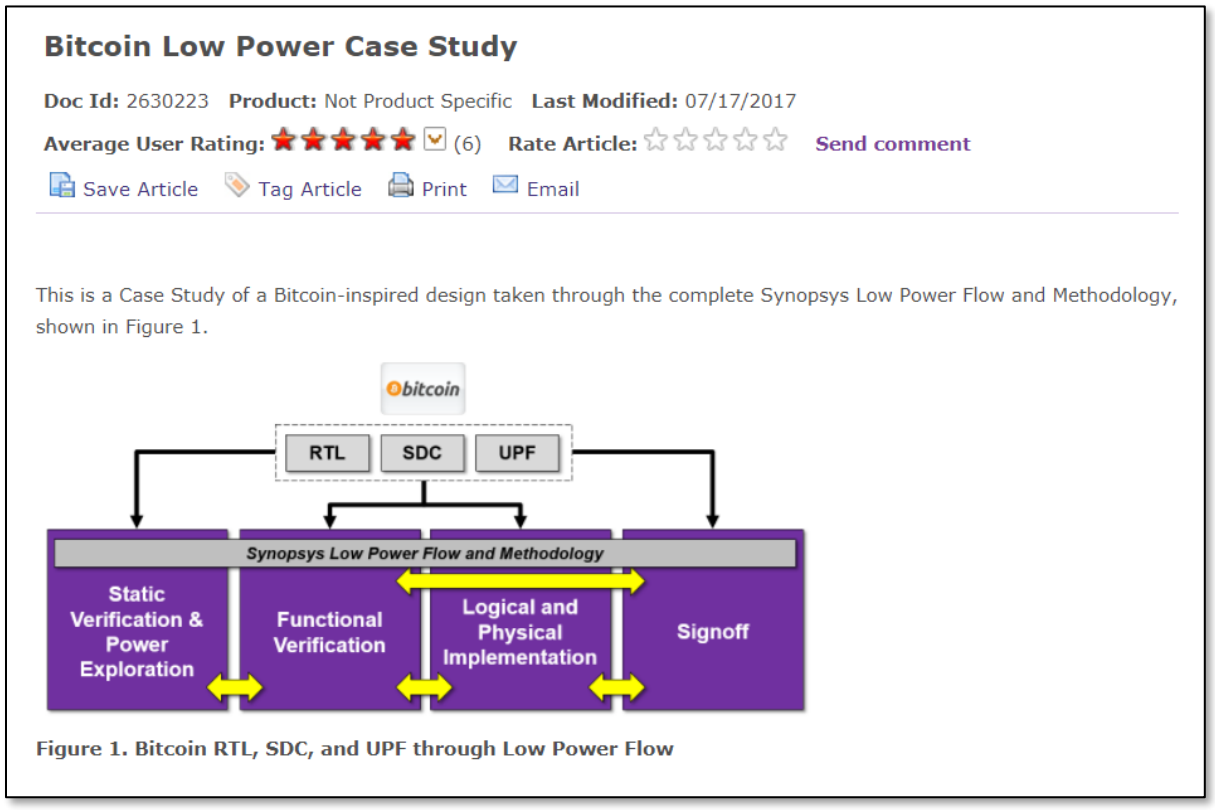
Copyright Notice and Proprietary Information

Table of Contents

Bitcoin – Case Study User Guide

This document contains the details for a Case Study centered around the Synopsys Low Power Flow and Methodology. The example design is inspired by Bitcoin, a crypto-currency standard used throughout the world. All applicable design data including RTL, SDC, UPF, Testbench, Technology Libraries, detailed Makefiles and tool scripts are included as part of the reference package. This package can be downloaded via SolvNet and should be used in conjunction with this User Guide.

On Solvnet, search for "bitcoin" or the SolvNet Doc Id shown below. Links are included to download the Reference Design and Technology Libraries.



Version History

Version	Release Date	Comments
v1.0	March 2017	Initial Revision
v1.1	September 2017	PowerReplay Added, Changes to VCS NLP GLS Methodology, Changes to Tech Libraries for PT-PX

Chapter 1, Introduction

There are many challenges in Low Power ASIC design, implementation, and verification that many teams face in today's state-of-the-art SOCs and IPs. Among these are:

- Coding UPF, choosing the right style which can consistently work through an entire flow
- Having a broad-flow understanding of a full Low Power Methodology
- Understanding how to run and test the various tools of a full Low Power Methodology

This Case Study User Guide and accompanying Reference Design will help you with each of these, and give you a more thorough and detailed understanding of how to better design, implement, and verify your Low Power SOC and IP.

The Case Study User Guide is organized as follows:

- Chapters 1-3: Overview of the Reference Design and the Synopsys Low Power Flow and Methodology
- Chapters 4-17: Step-by-Step view of each individual tool used in various stages of the flow
- Chapter 18: Discusses how and where to measure power at various stages of the flow as the Low Power Measurement Flow
- Chapter 19: Future Plans and Observations, discusses future work and limitations seen in the flow

Each of individual tool chapters (Chapters 4-17) are organized as follows:

- Overview: Brief discussion of the tool, it's main purpose, and how it fits in the flow
- Scripts & Mechanics: How to invoke the tool and the main elements of a basic tool script
- Bitcoin Experience: Details on results of the Bitcoin design taken through the specific tool

Note, the approach with each section assumes the user is not familiar with the given tool, so we take a "broad-flow" approach to presenting the material, not necessarily a "deep-flow" approach. Detailed training, of course, is available via other sources, but the idea for this Case Study is to focus on the high-level overall low power flow and methodology.

Download and Install Reference Design and Tech Libraries

The Reference Design includes all the input source (RTL, SDC, UPF), scripts and methodology for all the tools in the Low Power Flow, as well as the technology libraries. The download package includes separate tarballs for the Reference Design and Tech Libraries as listed below.

```
bitcoin_v1.1.tgz (Reference Design)
bitcoin_v1.1_lib.tgz (Tech Libraries)
```

Unpack the "Tech Libraries" in a central location, which can be accessed globally by users. Unpack the "Reference Design" as needed for the individuals wanting to run experiments, and then add pointers to the "lib" and "tech" directories as shown in the next section.

Files and Directories

Unpack the Reference Design and Tech Libraries, as shown below.

```
% cd <PATH_TO_GLOBAL_AREA>
% gtar zxvf bitcoin_v1.1_lib.tgz

% cd <INDIVIDUAL_WORKAREA>
% gtar zxvf bitcoin_v1.1.tgz
```

The files and directories for the Reference Design are shown below.

```
Files and Directories
---------------------

.
├── def           : DEF files for implementation
├── lib           : Tech libraries
├── README        : This file
├── rtl           : RTL Source Code
├── sdc           : SDC Source Constraints
├── tech          : Tech Libraries
├── tools         : Tools Directory
│   ├── dc        : Design Compiler, DFTMAX, Power Compiler (Logic Synthesis, DFT, Power Optimization)
│   ├── fm        : Formality (Logical Equivalence Checking)
│   ├── icc2      : ICC II (Place/Route)
│   ├── Makefile  : Top level Makefile
│   ├── pt        : PrimeTime (Static Timing Analysis), PrimeTime PX (Power Analysis), PrimeTime ECO (ECO)
│   ├── setup.csh : Top level setup file (please edit this per your environment)
│   ├── sg        : SpyGlass (Static Checking for Lint/SDC/CDC + RTL Power Estimation)
│   ├── vcs_nlp   : Native Low Power Simulation
│   ├── vclp      : Static UPF Checking
│   └── verdi_paa : Power Analysis Acceleration (Accelerated FSDB generation for PrimeTime PX)
├── upf           : UPF Source
├── verification  : Verification Models
└── verilog_pg    : PG Verification Models
```

Figure 1.1 – Files and Directories

Note the “lib” and “tech” directories include all the technology libraries for the design, including all views for logical and physical implementation. *These directories are quite large, so it’s best to copy them to a central area and then create links to them in each Reference Design installation.* Figure 1.2 shows this, where the libraries are stored just above the install of the Reference Design. Of course, you can store the libraries wherever is convenient for your team to access globally.

```
-rw-r--r--  1 tony synopsys 5274 Sep  5 09:31 README
drwxr-xr-x  2 tony synopsys 4096 Sep  5 09:31 def
drwxr-xr-x  2 tony synopsys 4096 Sep  5 09:31 rtl
drwxr-xr-x  2 tony synopsys 4096 Sep  5 09:31 sdc
drwxr-xr-x  2 tony synopsys 4096 Sep  5 09:31 upf
drwxr-xr-x  8 tony synopsys 4096 Sep  5 09:31 verification
drwxr-xr-x  2 tony synopsys 4096 Sep  5 09:31 verilog_pg
drwxr-xr-x 10 tony synopsys 4096 Sep  5 10:38 tools
lrwxrwxrwx  1 tony synopsys   23 Sep  7 13:51 lib -> ../bitcoin_v1.1_lib/lib
lrwxrwxrwx  1 tony synopsys   24 Sep  7 13:51 tech -> ../bitcoin_v1.1_lib/tech
```

Figure 1.2 – lib and tech directories stored separately, common links added per install

Tool Versions

For this Reference Design, we are using the following versions of the tools listed, shown in Figure 1.3. If you choose different versions of the tools, it's possible you will get different results from what is documented here. Note we include a "setup.csh" file in "bitcoin_v1.1/tools" directory. *Please modify this file to suit your environment.*

SpyGlass	2017.03-SP1
VC LP	2017.03-SP1
VCS NLP	2017.03-SP1
Verdi Power Aware	2017.03-SP1
PowerReplay	2017.03-SP1
DC, DFT, PwC	2016.12-SP2
TetraMAX	2016.12-SP2
PrimeTime, PT-PX, ECO	2016.12-SP2
Formality	2016.12-SP2
ICC II	2016.12-SP2
StarRC	2016.12-SP2

Figure 1.3 - Tool Versions Used

Technology Libraries

The technology libraries included with the Reference Design are the "saed32" Synopsys Libraries, which is based on 32nm technology and used for illustration purpose. Views for low power static verification, simulation, synthesis, P&R, and signoff are provided in the given technology libraries.

Running the Reference Design

You can choose to run the Reference Design at different "levels" and with "individual tools" or the "complete flow." The Reference Design methodology is completely flexible and modular to include all, or just a select set of, blocks and tools.

For example, you can run the flow just at the "bit_slice" level, to do quick checks on how to run the basics of a tool, or to simply experiment. Alternatively, you can run at the "bit_coin" level (or even at "bit_top" if you'd like), if you want more complex examples.

You can also choose to run the "full flow." For example, to run the entire flow for "bit_slice", you can issue the following commands.You can choose to run the "full flow". For example, to run the entire flow for "bit_slice", you can run the following commands.

```
% cd bitcoin_v1.1/tools
# modify setup.csh first to suit your env
% source setup.csh
% make bit_slice
```

It will take about 30 minutes to go through the entire flow with bit_slice. Note we do some shortcuts in ICC II for this flow to save runtime.

For bit_coin, you can run the "full flow" as well, with the following commands.

```
% cd bitcoin_v1.1/tools
% source setup.csh
% make bit_coin
```

Note for this case, it will take 12+ hours to get through the entire flow.

For each individual tool at any stage, you can also run without having to re-run DC and ICC II, since their outputs are already included with the package. In that case, you would change into the specific tool directory and perform the specific Make target for your desired run. For example:

```
# Run VCS NLP @RTL
% cd bitcoin_v1.1/tools/vcs_nlp
% gmake

# Run VC LP @Gate
% cd bitcoin_v1.1/tools/vclp
% make bit_coin_gate
```

```
# Run FM @PG
% cd bitcoin_v1.1/tools/fm
% make bit_coin_rtl2pg
```

Finally, there are some dependencies in the flow that do require that some tools are run beforehand, namely when a tool (SpyGlass PE, PowerReplay) require an FSDB from VCS NLP, DC which requires a SAIF from VCS NLP, or PrimeTime, which requires an FSDB from PowerReplay. Those dependencies are shown in the top level Makefile, in case you need to run them manually.

Chapter 2, Bitcoin Overview

Technology

Bitcoin is a popular digital currency that is in widespread use today. The compute servers that make up the Bitcoin Network comprise the most powerful network on the planet, and it's become so just in the past ten years.

A Bitcoin Mining server is essentially a massive set of parallel hashing functions, whose sole purpose is to solve mathematical challenges as quickly as possible, to gain a reward of Bitcoins. Thus, mining is the method in which new Bitcoins are made available. These Miners in the early days were based on CPUs and later GPUs; but later FPGAs and now specialized ASICs have come into play. A massive collection of Mining Systems, also known as Mining Farms, are now dominant in the Bitcoin Network.

Motivation

For the Mining Farms, the cost of storage and electricity become deciding factors in their profitability; hence an improvement in the overall efficiency of a system leads directly to its overall cost-effectiveness. For example, in 2014, the state-of-the-art Mining System had a throughput of 2 THash/sec (TeraHashes per second) and was powered by a 1400W power supply. Two years later in 2016, the upgraded system had a throughput of 14 THash/sec and was powered by a 1375W power supply. That is a 7X improvement in power efficiency in just 2 years!

Clearly, the ASICs in that system were significantly improved, and thus achieved the incredible performance improvement. In other words, the same system could use 7X less electricity with the same throughput.

For this and other reasons, Synopsys chose to use Bitcoin, a real-world design with real-life impact, to create a Low Power Case Study.

Design

The "Bitcoin-inspired" design is implemented using a hierarchical approach. Our reasons for doing so are as follows:

- For Physical Design, we utilize MIMs (multiply-instantiated modules), so any changes to the base design will only require a single change to an instance that gets propagated, rather than multiple changes to several instances.
- For UPF, a hierarchical design is much more challenging than a flat design
- For many customers, a hierarchical approach is the preferred design style

The microarchitecture for the Bitcoin design is shown in Figure 2.1.

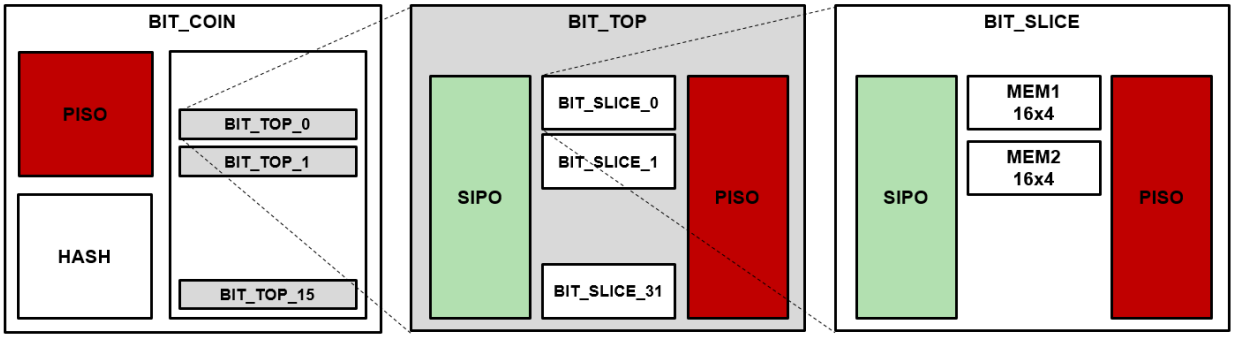


Figure 2.1 - Bitcoin Microarchitecture

Per Figure 2.1, the fundamental building block of the system is "BIT_SLICE", which is comprised of a SIPO (Serial In-Parallel Out) block, two memories, and a PISO (Parallel In-Serial Out) block. The BIT_SLICE block is instantiated 32-times in "BIT_TOP", which also has SIPO and PISO blocks. And finally, BIT_TOP is instantiated 16 times in "BIT_COIN" which has a PISO and HASH block.

Note the Reference Design is "Bitcoin-inspired" because the HASH used is a simplified version of the real hash, a SHA-256 (Secure Hash Algorithm, 256-bit). We made this and other design choices to ensure the runtime of the design through the tools was reasonable and relatively fast while still mimicking a real-world design.

UPF (Unified Power Format)

The UPF power intent for any hierarchical low power design can be quite challenging. And even though the microarchitecture of the design is quite simple from Figure 2.1, it's clear in Figure 2.2, the complexity of this low power design is substantial.

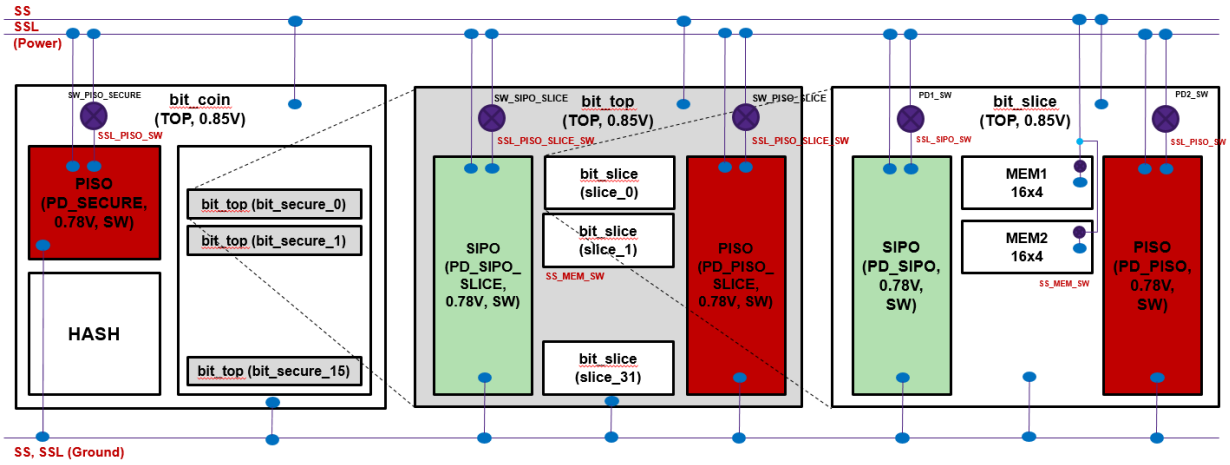


Figure 2.2 - Bitcoin Power Diagram

Per Figure 2.2, notice we have a multi-voltage design, with the SIPO and PISO blocks run at a lower voltage. The memories, HASH, and top level logic run at the higher voltage. The SIPO and PISO blocks at each level are switched and the memories in "bit_slice" have internally switched supplies. All the SIPO blocks are tied together, so all SIPOs at all levels are commonly switched. Same is true for the PISO blocks.

Based on all of this, we have the following statistics at the top level:

Total Power Scope defined	529
Total Power Domain defined	1586
Total power switch defined	1057
Total isolation policy defined	3633
Total retention policy defined	1057
Total PST defined	529
Total add_pst_state defined	2116
Total unique supply source	6350
Total supply set defined	12093

So, it's evident the relatively simple microarchitecture becomes a very difficult Low Power challenge. To simplify the many states and policies, we needed to make significant adjustments to the UPF. In short, for the SIPO, PISO, and internal memories, we tied them all together via UPF at the top level, and commonly referred to the shorted supplies in the top-level PST. This is shown in Figure 2.3. You can refer to the source UPF in the Reference Design for details on how exactly that was coded. If not for this, we would have an unreasonable and unmanageable 6300+ entries in the top-level PST.

4 states	2 always-on supplies		2 supply shutdown		Ground	3 supply shutdown		
	High V	Low V	SW High V	SW Low V	GND	SH1	SH2	MS
ALL_ON	ON	ON	ON	ON	ON	ON	ON	ON
PON_SOFF	ON	ON	ON	OFF	ON	ON	OFF	ON
POFF_SON	ON	ON	OFF	ON	ON	OFF	ON	ON
MEM_OFF	ON	ON	OFF	OFF	ON	OFF	OFF	OFF

Figure 2.3 - Top Level PST

We also include the following UPF constructs in this example:

- **NOR-Isolation**, which greatly reduces routing resource in that when shut-down, the NOR-ISO cell ties the output to GND vs. having to have a dedicated backup power route
- **PSW ack function and delay** (power switch acknowledge)
- **Retention** for both SIPO and PISO blocks
- **Isolation - Heterogenous Fanout** examples
- **UPF 2.1 Constructs for PST** (power state tables)
- **Explicit Supply Sets**
- **Full Refinement of PG** (power, ground)

These constructs were chosen to enable the UPF to be used across the entire flow. We tried many experiments with many different styles of UPF, including all the latest UPF features, but we ended up with this subset for this Reference Design.

Future Work

The current Reference Design was built mostly for illustration purposes for the Low Power Flow and Methodology. We have a long list of enhancement that are ready to implement, which includes: changes and updates to the UPF, additional features in the RTL and design, and more details for the verification plans and such. More detail is available in Chapter 19, the Future Plans and Observations section.

Chapter 3, Low Power Flow and Methodology

The complete Synopsys Low Power Flow and Methodology is shown in Figure 3.1.

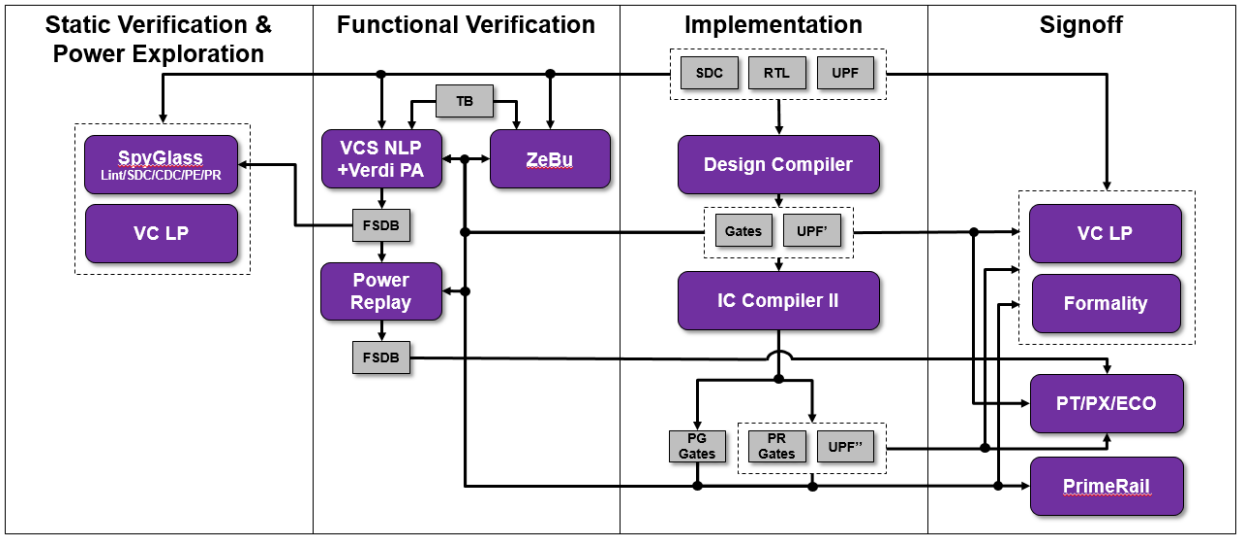


Figure 3.1 - Synopsys Low Power Flow and Methodology

Static Verification & Power Exploration

In the first stage, the RTL and SDC are taken through SpyGlass Lint, SDC, and CDC to perform static analysis to ensure the RTL and SDC are checked for syntax, correctness, completeness. CDC (clock-domain crossing) checks are also run to ensure there are no metastability issues with crossing asynchronous clocks.

VC LP is then run on the RTL + UPF to ensure the UPF is complete and correct. This step is vital in ensuring a clean UPF throughout the rest of flow, as it can be difficult to find and address these issues as the flow progresses.

SpyGlass PE can be used to explore power at the RTL-level, and gain relative insight to how RTL changes can affect things like clock-gating efficiency and thus effect the overall power. Note the switching activity from simulation (RTL FSDB) from the next step can be used to drive the power estimation values at this stage.

Functional Verification

In the second stage, the RTL, UPF and Testbench are taken through VCS NLP (Native Low Power) and Verdi Power Aware (PA) for functional low power simulations. Here the dynamic simulations are run to verify, among other things, the power sequence and the various ways in the which the supplies can be turned off and on during operation. ZeBu can be used for emulation purpose. PowerReplay can be used to map RTL FSDB to Gate FSDB, thus providing more accuracy in less time with more flexibility for PrimeTime PX Power Estimation during signoff.

Implementation

In the third stage Design Compiler® (DC), DFTMAX™, and Power Compiler™ are used to synthesize the RTL with SDC and UPF as inputs. The resulting Gates + UPF' are fed as input to IC Compiler™ II (ICC II), which performs placement and routing (P&R) and generates PG Gates + UPF".

Signoff

In the fourth stage VC LP is called again to perform signoff checks on the Gates + UPF' after synthesis, as well as PG Gates + UPF" after P&R. Formality is also used to ensure functional equivalence of the RTL + UPF vs. Gates + UPF' and RTL + UPF vs. PG Gates + UPF". PrimeTime® is used to measure timing for signoff, while PrimeTime PX (PT-PX) is used to perform power analysis for signoff. Again, we can use the FSDB generated from PowerReplay in this step to help accelerate these power measurements. PrimeTime ECO can be used to drive ECO changes from PrimeTime back to ICC II. Finally, PrimeRail can be used to measure IR-Drop and other measurable data related to the power grid and plan.

Chapter 4, SpyGlass Lint/SDC/CDC/PE

Overview

SpyGlass is an Early Design (RTL) Analysis Tools enable efficient verification and optimization of SoC Designs. The SpyGlass platform provides designers with insight about their design, early in the process at RTL. It functions like an interactive guidance system for design engineers and managers, finding the fastest and least expensive path to implementation for complex SoCs.

Refer to the Figure 4.1 below for the input to the SpyGlass. For this case study, we are not using UPF as UPF aware Power Exploration feature is new and will be available starting from 2017.03 release.

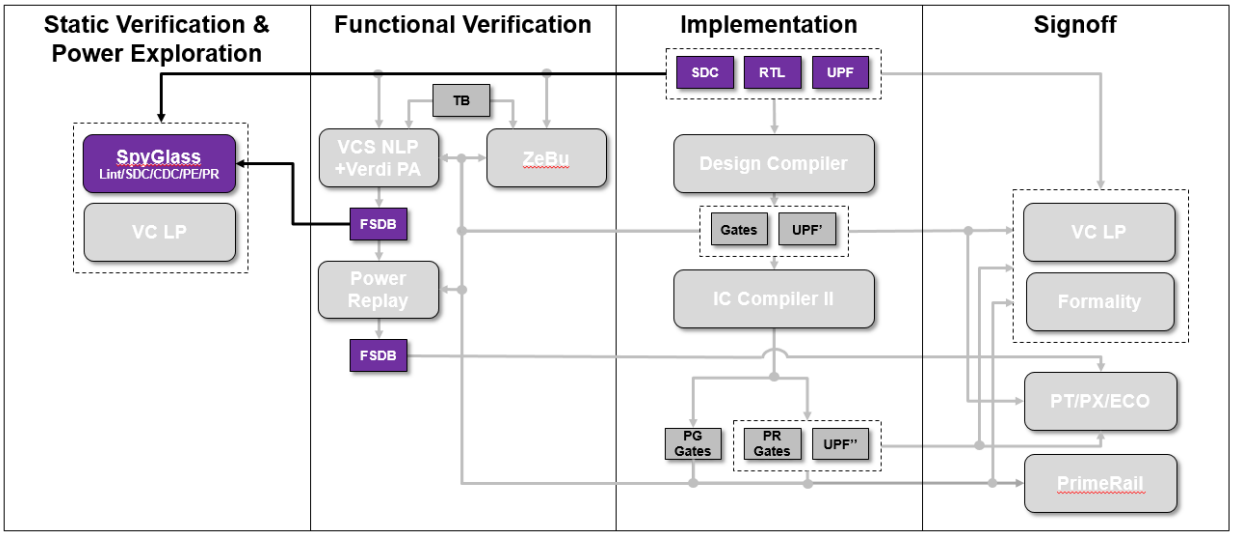


Figure 4.1 – SpyGlass in the Synopsys Low Power Flow and Methodology

SpyGlass is a platform of Static Verification tools. Incorporating SpyGlass as RTL sign-off tool for Lint, SDC checking, Clock Domain Crossing, Reset Domain Crossing, Power Exploration and for stuck-at and ATPG coverage analysis @ RTL, significantly reduces the design effort, makes the design flow convergent and reduces the risk of missing design schedule. Another big advantage of a single platform is that it significantly reduces learning curve, as same setup can be leveraged to run different analysis and single debug environment for resolving issues.

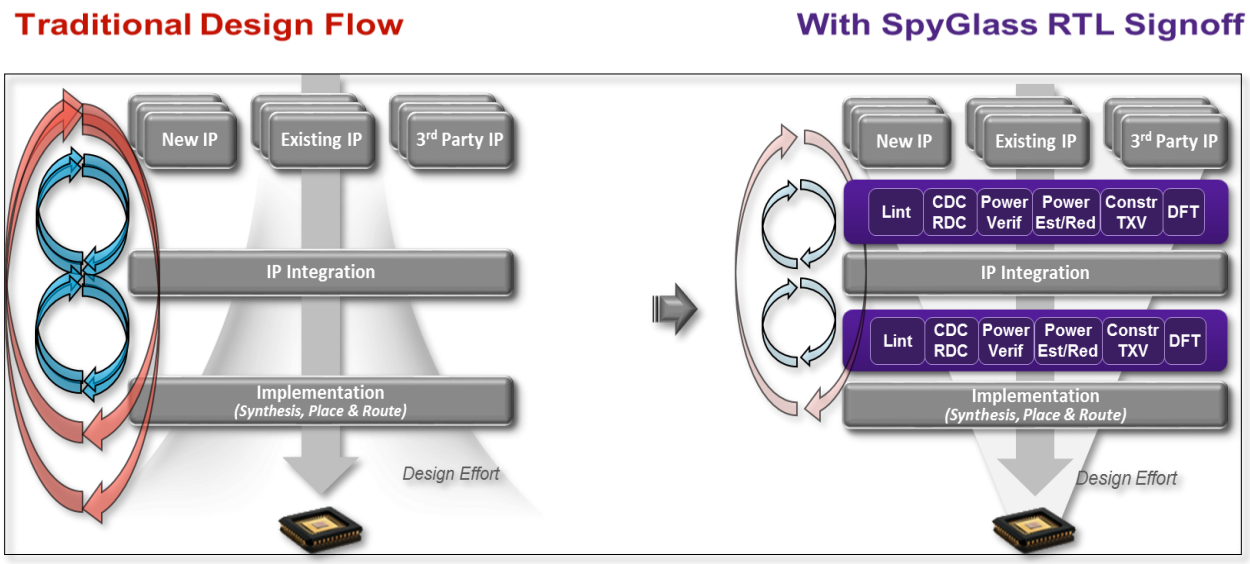


Figure 4.2 - Traditional Design Flow vs. RTL Signoff Flow with SpyGlass

For bit_coin, checks for Lint, SDC, CDC, and Power Exploration were run. In this guide, we will focus on CDC and RTL Power exploration in detail. You can review Lint and SDC results by checking the directories below:

```
% cd tools/sg/bit_coin

# lint results
# results/bit_coin/bit_coin/lint/lint_rtl

# constraints results
# results/bit_coin/bit_coin/constraints/sdc_audit
# results/bit_coin/bit_coin/constraints/sdc_check
```

SpyGlass CDC Overview

SpyGlass CDC solution is a comprehensive CDC verification tool, with mix of Structural, Formal and Hybrid (Simulation based) technologies. Structural identification is based on protocol independent verification, identifies good and bad data and control crossing structures. Formal verifies the structurally identified good crossing structures will work as intended. Users that are not formal savvy can leverage Hybrid Simulation based approach, which allows to write SVA for CDC assumptions and CDC protocols, that user can plug into their simulation environment and functionally verify CDC with simulator of their choice. Hierarchical CDC technology meant for large SoCs, significantly reduces the run-time, number of violations at top-level due to its divide and conquer approach and allows to iterate and close faster.

Even though RTL is signed-off as CDC clean, it goes through implementation flow which could potentially introduce CDC issues not being present at RT-Level. This challenge will be addressed by new technology "Smarter Netlist CDC" solution. This technology leverages RTL CDC analysis and reports only the additional CDC issues that are introduced during implementation flow.

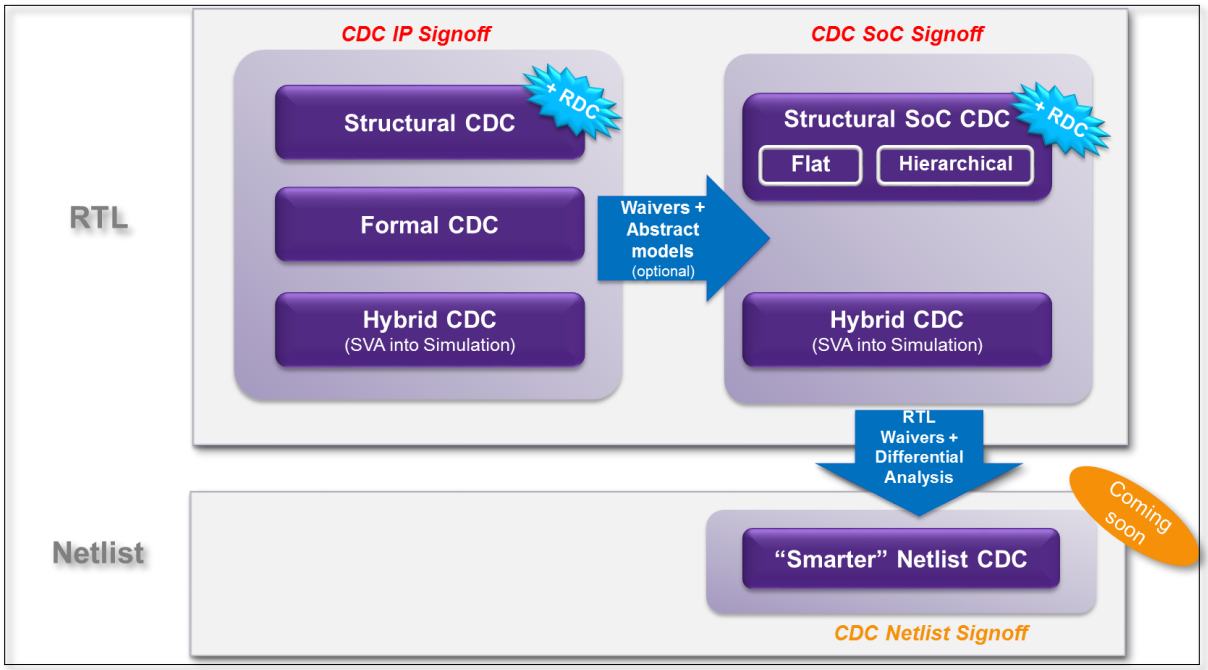


Figure 4.3 - SpyGlass CDC Solution

SpyGlass RTL Power Exploration Overview

Every milliwatt of power matters, regardless of the application. Designers can no longer wait for the final netlist to get accurate power numbers, as full visibility is needed as soon as RTL coding starts, when the most rewarding modifications can be made. At smaller technology nodes, dynamic power is becoming increasingly more dominant and the reduction of overall activity has become a necessity. As designs become vastly larger, designers need a tool like SpyGlass that pinpoints major power usage while suggesting modifications with the highest ROI. It starts with power exploration which applies to SoC for both critical and non-critical blocks.

There are two use models for Power Estimation, first is differential power exploration to measure the impact of RTL changes, and second use is for those who rely on accurate power estimation. At the RT-Level, the first use model is more appropriate. If the focus is early power correlation with PrimeTime PX, then PowerReplay with PrimeTime PX is more relevant, refer to Chapters 12 and Chapter 19 for more information.

The next step after power estimation is analysis of power metrics which is called power profiling. This includes metrics such as average clock gating efficiency, register output activity density (ROAD) etc. The last step is the power reduction, identifies power reduction opportunities. Taking RTL through Power exploration and incorporating the feedback into RTL you get power efficient RTL.

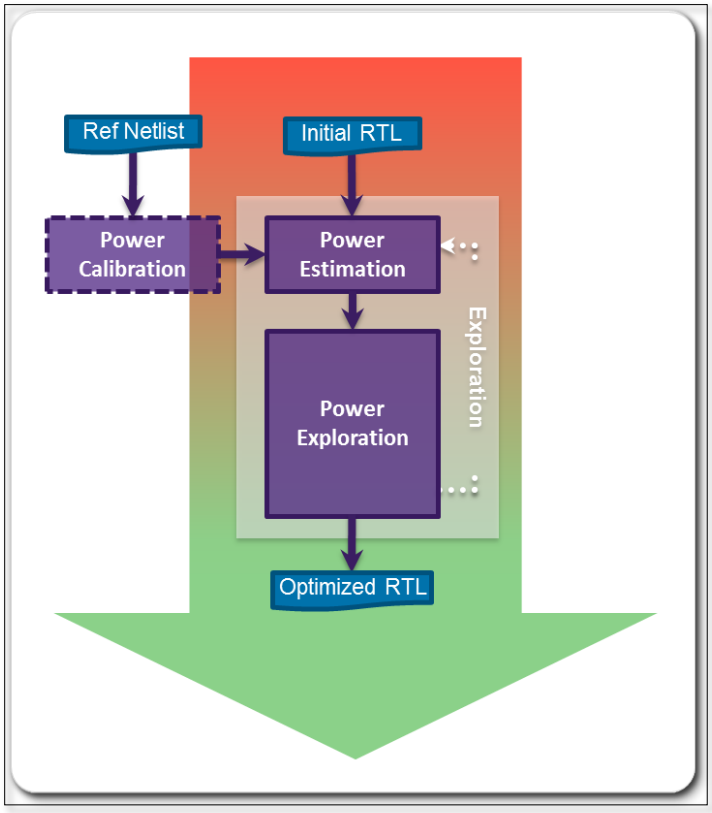


Figure 4.4 - SpyGlass RTL Power Exploration Flow

Scripts & Mechanics

To run SpyGlass, you need a Tcl based setup script called project file, which contains information like deign file, libraries, waivers, constraints, and any tool related options/parameters. Below is the example of sample project file. The "common option section" shows some of the global options like specifying the reports directory location, specifying the spyglass working directory, enabling system Verilog parsing etc.

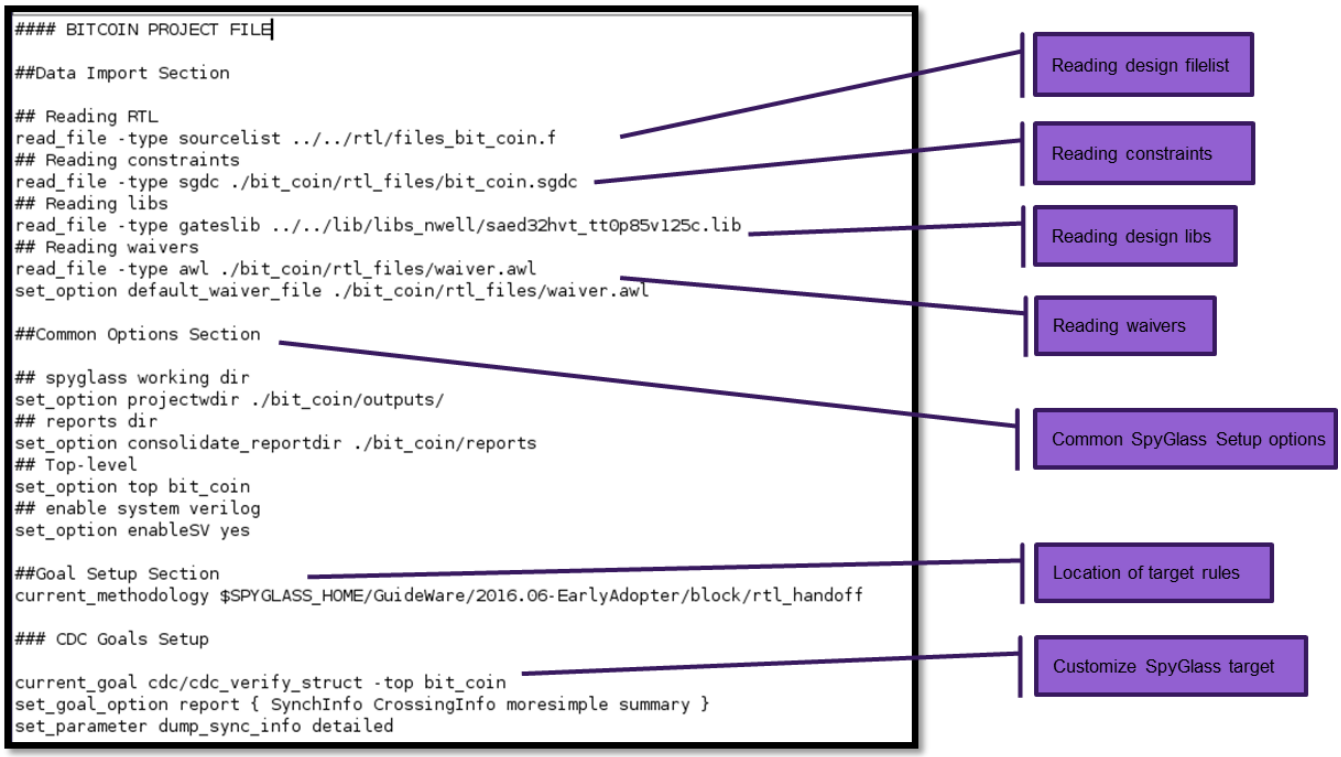


Figure 4.5 - SpyGlass Setup Project File

Once you have the project file, you can run spyglass using following command (assuming spyglass is in $path).

```
% spyglass -project bit_coin.prj -batch -goal lint/lint_rtl
```

The command above is to run lint in batch mode, you can use the similar command to run SDC checking, CDC, Power profiling by specifying the appropriate target (-goal).

If design instantiates DesignWare, then user need the following mechanism in SpyGlass to get them resolved and not having them as black-box. SpyGlass has some of the DesignWare modules netlist as part of its installation, those are resolved with "dw" option, the ones that are not part of the installation user need to set environment variable "SPYGLASS_DC_PATH" to the DC installation directory.

```
# In SpyGlass Project File
set_option dw yes
setenv SPYGLASS_DC_PATH <PATH_TO_DC_INSTALL>
```

In Figure 4.6, we see some of the key reports from SpyGlass.

Report Name	Report Overview
spyglass.log	Provides all the log information while the SpyGlass analysis is running
moresimple.rpt	Report includes all necessary information like rule name, file name, line number, severity label, and the violation message as dumped by the respective rules
elab_summary.rpt	Gives you the details about parameter/generic values in terms of actual vs. default such that you can validate and debug
summary.rpt	Displays a summary list of message counts by each particular rule type along with the severity class and rule short help
waiver.rpt	Waiver report is generated when messages are waived during SpyGlass run

Technology Specific Reports	Report Overview
CDC	**SynchInfo.rpt:** Reports all the synchronized crossings **CKSGDCInfo.rpt**: Reports the applied CDC constraints **CDC-report.rpt:** CDC setup and analysis summary
Power	**pe_audit.rpt:** The report contains detailed information about the inputs provided for the SpyGlass Power exploration **pe_summary.rpt**: Describes the various aspects of power consumption of the design **pe_debug_info.rpt**: Contains debug information of nets that are not set from the simulation file **pe_adv_reduction.rpt**: Shows clock gating statistics. In addition, contains information about the power saved by new clock gating candidates **pe_design_stats.rpt**: Shows the details of the technology cells used to synthesize the design

Figure 4.6 - SpyGlass Reports

Bitcoin Experience

To run bit_coin with SpyGlass, first make sure your SpyGlass software and licenses are configured and installed. Then, simply type the following commands:

```
# Assume directory is bitcoin_v1.1
% cd tools/sg
# To run lint
% make bit_coin_lint
# To run sdc
% make bit_coin_constraints
# To run CDC
% make bit_coin_cdc
# To run RTL Power Exploration, create RTL FSDB first
% cd ../vcs_nlp; gmake; cd ../sg
% make bit_coin_power
# To review results and debug in GUI
% make bit_coin_debug
```

This Makefile target will execute the following command, and you can peruse the script and logs to learn more.

The bit_coin design was started as a single clock design. After three RTL releases, CDC (two asynchronous clocks) was introduced. This impacted the entire design flow from verification to implementation to floor planning, and all the previous effort was wasted due to CDC fixes which changed RTL significantly. The point is CDC analysis should not be an afterthought, it should be part of RTL sign off flow.

After resolving majority of the lint CDC issues (some of them has been waived either not critical or left to be addressed later), the next step is to take it through RTL power exploration to identify architectural power deficiencies. This is designer's tool as the feedback provided need designs knowledge to understand the impact and make a call on which one to incorporate.

The very first step of the Power exploration is the "Power Guidance", this can be run as early as RTL coding starts, consider this as of Lint for power, doesn't require FSDB. This provides quick feedback to make gross architectural changes to save power. Below are two examples from bit_coin design:

1) Clock gate sharing same clock and enable can be replaced with global clock gate. Sharing gated clocks and having one gated clock as close to the clock source as reduces power consumption. This is shown in Figure 4.7.

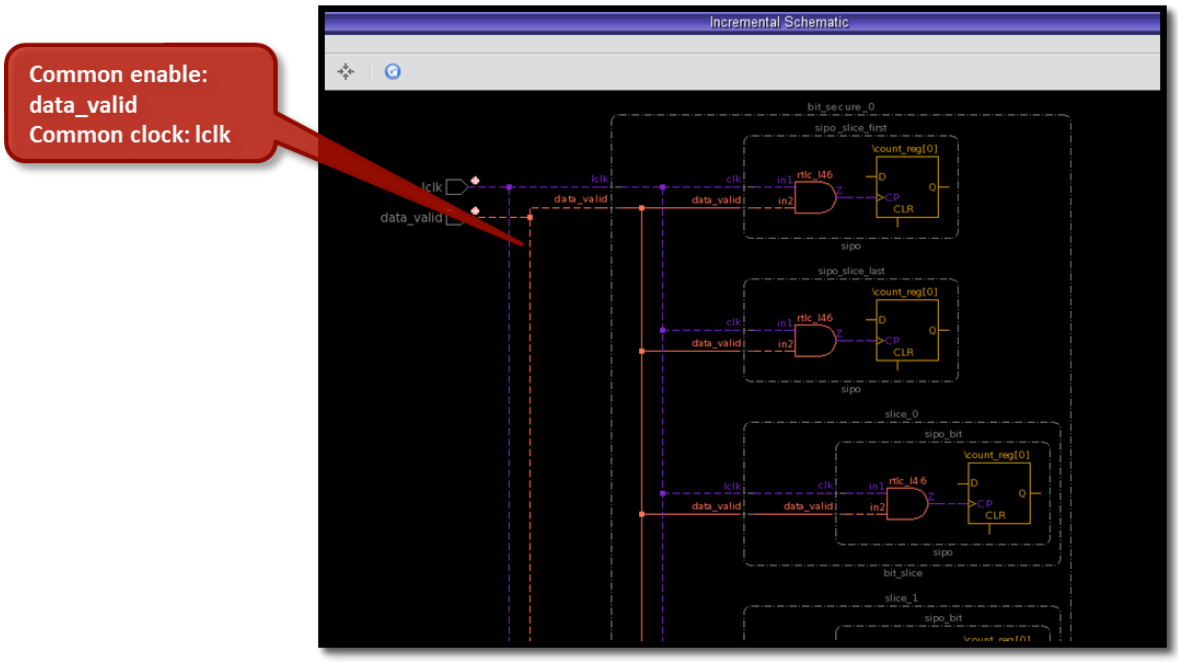


Figure 4.7 – Common Enable

2) Identifies the 4-bit counter present in the design and reports a signal that can used to gate specific bits of that counter. User can configure to not report counters that have fewer bits than the value set by this "set_parameter pe_min_counter_width 5" by default it is set to 2.

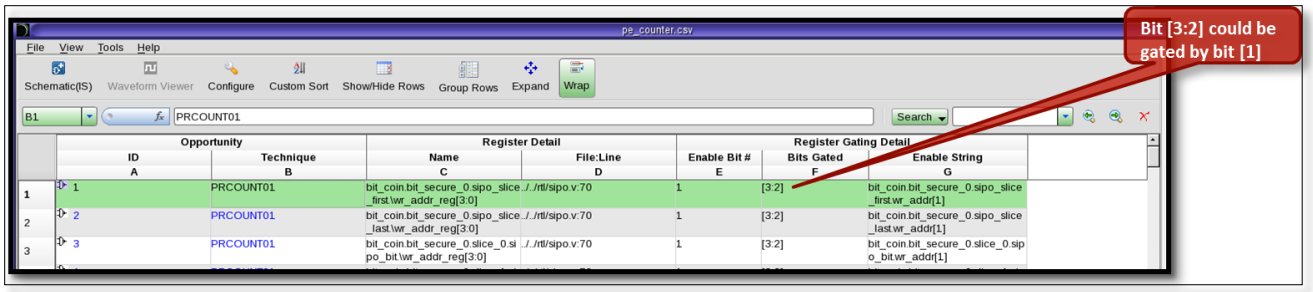


Figure 4.8 – Counter Reports

The next step is Power profiling which provides various power efficiency metrics that give insight into power characteristics of the design. **Clock gating ratio** is the ratio of the registers in a block that have been clock gated. **Clock gating efficiency** can be viewed as the ratio of the clock frequency at the clock pin of the register to the root clock. These metrics are based on the activity data read in, so user need to make sure it's the right/representative set of vector for power exploration

SpyGlass can report those metrics in different formats. In GUI, the main view is called "Power Explorer View", from their user can dive into "Register View", "Tree-Map view" etc. Power Explorer captures all the key power related metrics and user has ability to configure these metrics (add and remove). Then there are spread-sheets generated as well.

In the bit_coin design, you have a block that the **Clock gating ratio is 21%** and this might or might not be an acceptable as the target depending on the design. Then **Average Clock gating efficiency** is 5%, and SpyGlass reports New Power reduction opportunities. Refer to the Figure 4.9 below for power explore view and register view for bit_coin design.

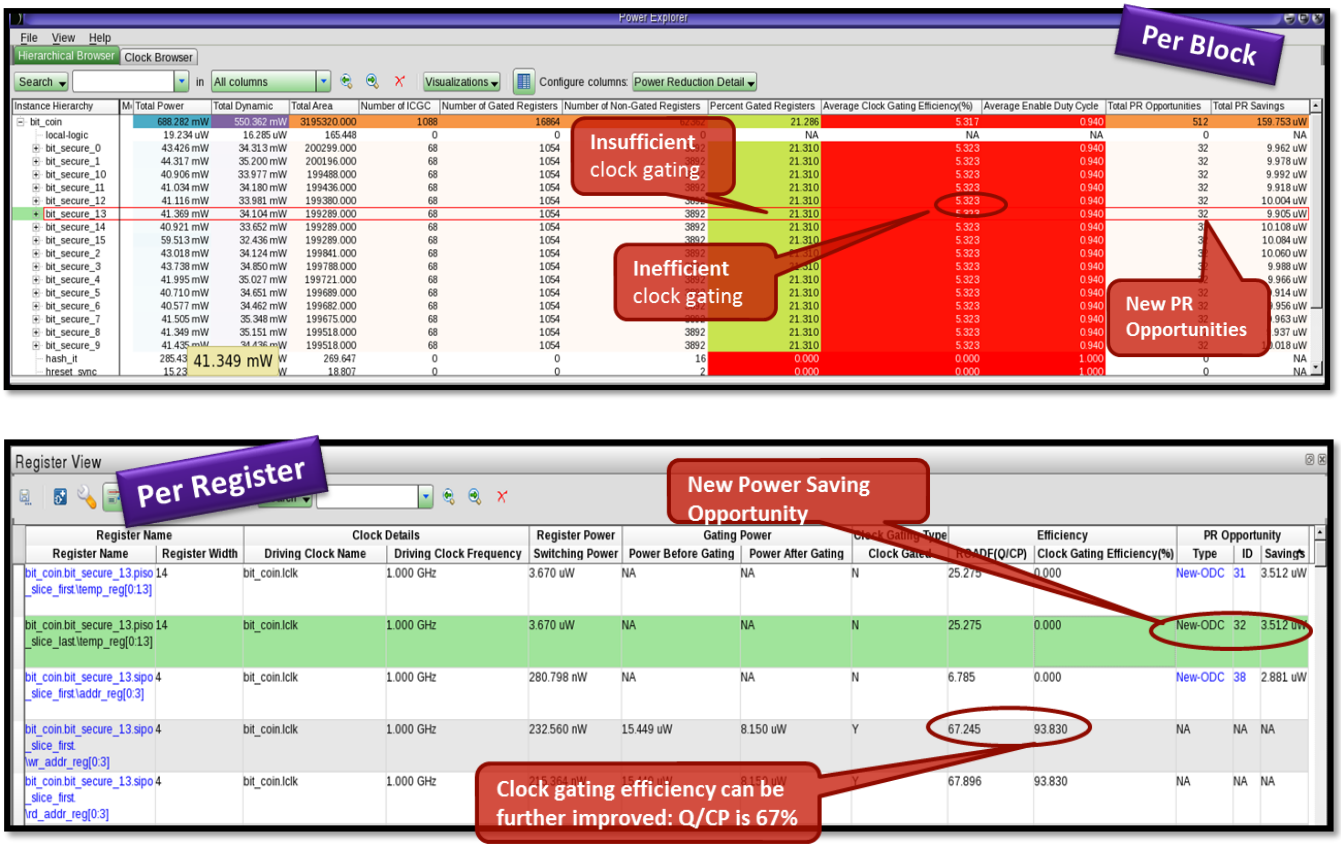


Figure 4.9 – Power Explore and Register Views

From all this data user, can easily make a call like if CGE is relatively low, but more importantly there is hardly any activity, CGE can be further improved. And SpyGlass Power reduction indeed use such data to identify further power reduction opportunities.

Chapter 5, VC LP @RTL

Overview

VC LP performs static checks of the UPF and RTL. The purpose of VC LP is to ensure the UPF is consistent with respect to the RTL and is correctly structured. Refer to Figure 5.1, you can see inputs to VC LP are RTL, UPF and optionally SDC. For the purposes of this case study, we will not use the SDC as input, but note it can be used to try more detailed functional and architectural checks on clocks, resets, etc.

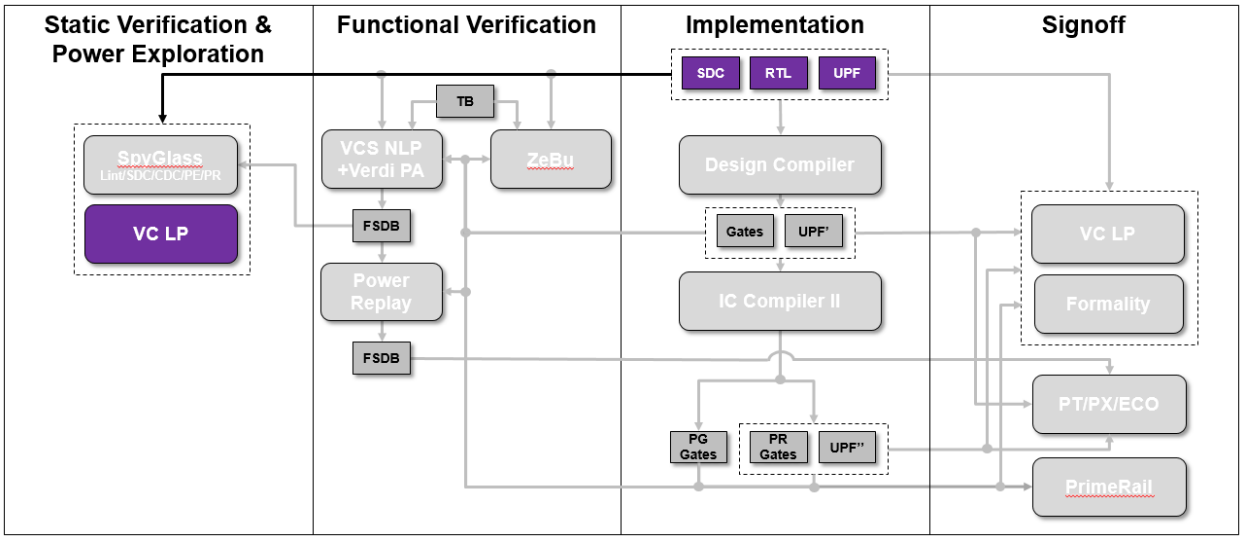


Figure 5.1 - VC LP@RTL in the Synopsys Low Power Flow and Methodology

VC LP is Synopsys' next generation Low Power Static Checking and Signoff Tool, and it has several features of importance:

- Built from ground-up for Advanced SOCs
- Consistency and correlation with implementation
- Comprehensive set of applications
- Efficient debug and root cause analysis
- 3-5X performance and capacity advantage
- Ease-of-use, Ease-of-Adoption
- Tcl Interface and commands, like Design Compiler and PrimeTime
- Comprehensive set of messages, clearly organized by family, object, and problem
- Report compression
- Verdi debug GUI interface

VC LP, at the RTL level performs UPF checks to ensure consistency and correctness. Note as the design progresses through logical and physical implementation, another set of checks (architectural, structural, functional, and PG checks) are also performed. This is shown in Figure 5.2.

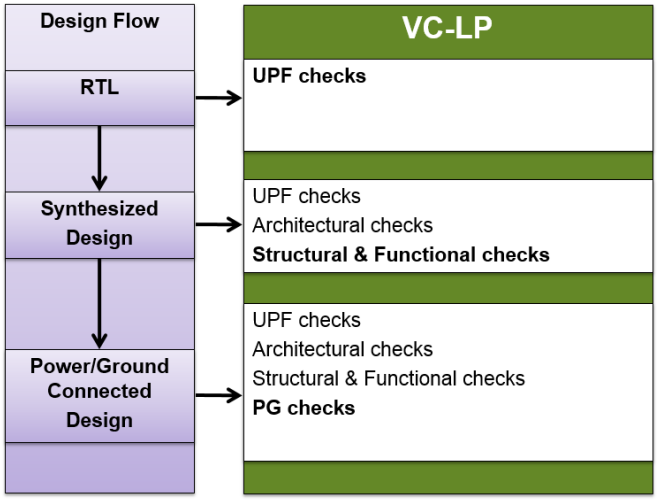


Figure 5.2 - VC LP in the Design Flow

As per the types of issues you can expect to find with UPF, consider the block diagram in Figure 5.3, where you have 3 domains, namely A, B, and C. Domain B is a switchable block, while Domains A and C are always-on. Referring to Domain B, it's possible to find "Missing or Redundant Isolation" strategy issues, "Incorrect Level Shifter" strategy issues, and a host of other UPF and structural issues. These can be identified very early in the design process, and corrected before running implementation or simulation.

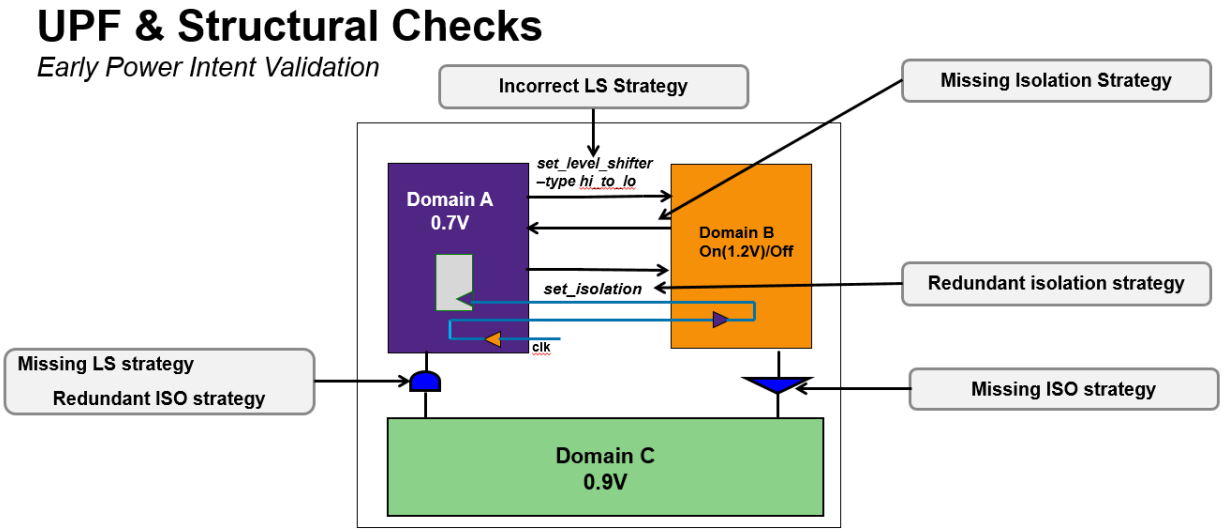


Figure 5.3 - VC LP UPF & Structural Checks

To run VC LP is straightforward. Referring to Figure 5.4, the tool scripts and GUI are easily run and then violations are stored in a persistent database. The database can be queried with built-in or custom commands, and the results can be processed, waived, or fixed and the process continues until clean UPF is achieved.

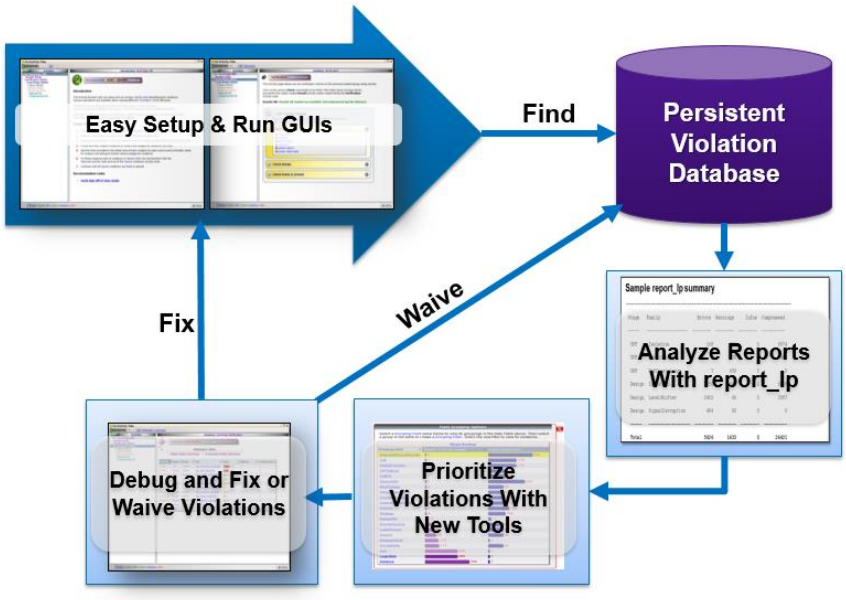


Figure 5.4 - VC LP Command Flow

In summary and as shown in Figure 5.5, VC LP performs checks at various stages of the design flow and ensures a future-proof methodology for validating your current and future designs.

Power Consistency Check	• Validate power-intent (UPF) defines necessary policies for all power-modes
Architectural Check	• Ensure power domain partitions do not lead to functional errors in design
Structural Check	• Check that necessary and sufficient low-power structures are present in implemented design
Functional Check	• Ensure functionality of low-power structures in design consistent with power-intent (UPF) and cell library
PG Check	• Verify power/ground pin connectivity in post-layout design is consistent with power-intent (UPF) and cell library

Figure 5.5 - VC LP Checks

Scripts & Mechanics

For RTL level checks with VC LP, the scripting is very simple. To start, you should have a setup file, similar to .synopsys_dc.setup (or pt or fm), which includes your search_path, link_libraries, and other variables. Note the example in Figure 5.6 which includes the variable "enable_verdi_debug" which uses the Verdi GUI for Debug.

```
# File : synopsys_vcst.setup
set search_path .
lappend search_path ../../rtl
lappend search_path ../../verification
lappend search_path ../../lib
…

set hvt_libs " \
saed32hvt_tt0p78v125c.db \
…
saed32hvt_pg_tt1p05v125c.db \
"
set rvt_libs " \
saed32rvt_tt0p78v125c.db \
…
saed32rvt_pg_tt1p05v125c.db \
"
set lvt_libs " \
saed32lvt_tt0p78v125c.db \
…
saed32lvt_pg_tt1p05v125c.db \
"

set mem_libs " sram2rw16x4_tt1p78v125c.db  sram2rw16x4_tt1p85v125c.db"

set link_library "$hvt_libs $rvt_libs $lvt_libs $mem_libs"

set enable_verdi_debug true
```

Figure 5.6 - VC LP setup file

Now, you can build a basic script to analyze/elaborate your design, load your UPF, and then perform checks and reports, and finally save a session, as shown in Figure 5.7.

```
# File : go_vclp_bit_coin.tcl
# VC-LP Script @RTL
source ./scripts/synopsys_vcst.setup

analyze -f Verilog -vcs "-f ../../rtl/files_bit_coin.f"
elaborate bit_coin

load_upf ../../upf/snps_bit_coin.upf

check_lp -stage upf

report_lp -list

save_session
```

Figure 5.7 - VC LP basic script

Next, you can launch VC LP to perform your run, and launch the GUI, if needed, to debug, as shown in Figure 5.8. To start the VC LP GUI, simply type "start_gui" at the prompt.

```
% vc_static_shell -f go_vclp_bit_coin.tcl | tee log.rtl.vclp

vc_static_shell> start_gui
```

Figure 5.8 - VC LP launch

If you'd like to restore a saved session, simply re-launch VC LP with -session and -restore arguments, as shown in Figure 5.9.

```
% vc_static_shell -restore -session vclp_sess/vclp_sess_bit_coin_rtdb
```

Figure 5.9 - VC LP Restore Session

From there, you can launch the GUI via the "start_gui" command and use the Verdi Interface to debug your design. For more detailed information on VC LP, please consult the VC LP User Guide, found on SolvNet.

Bitcoin Experience

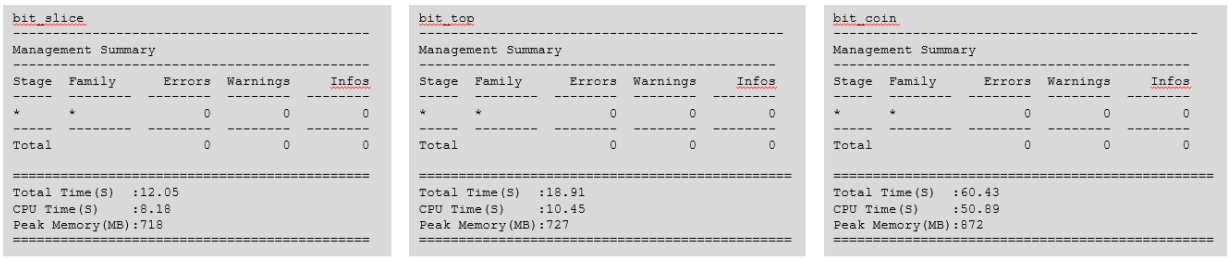
To run Bitcoin with VC LP, first make sure your VC LP software and licenses are configured and installed. Then, simply type the following commands:

```
% cd bitcoin_v1.1/tools/vclp
% make bit_coin
```

This Makefile target will execute the following command, and you can peruse the script and logs to learn more.

```
% vc_static_shell -f scripts/go_vclp_bit_coin.tcl -session
vclp_sess/vclp_sess_bit_coin | tee logs/log.vclp.bit_coin
```

Running Bitcoin through VC LP at the RTL stage was clean, fast, and efficient, noted in Figure 5.10.



Design	Errors	Warnings	Infos	Total Time (S)	CPU Time (S)	Peak Memory (MB)
bit_slice	0	0	0	12.05	8.18	718
bit_top	0	0	0	18.91	10.45	727
bit_coin	0	0	0	60.43	50.89	872

CLEAN FAST EFFICIENT

Figure 5.10 - VC LP Results with Bitcoin @RTL

You can see that even though Bitcoin is a complex UPF design with thousands of distinct power domains, VC LP could cleanly check it with 0 errors and warnings in just 1 minute.

Note when saving sessions, the default directory is “vcst_rtdb” and subsequent runs will overwrite that directory and save the last one to “vcst_rtdb.bak”. If you are running multiple blocks in the same directory, it makes sense to invoke VC LP with a “-session” argument and include a sub-directory, in this case vclp_sess/sess_bit_slice or vclp_sess/sess_bit_coin, to differentiate between the blocks. This is shown in Figure 5.11.

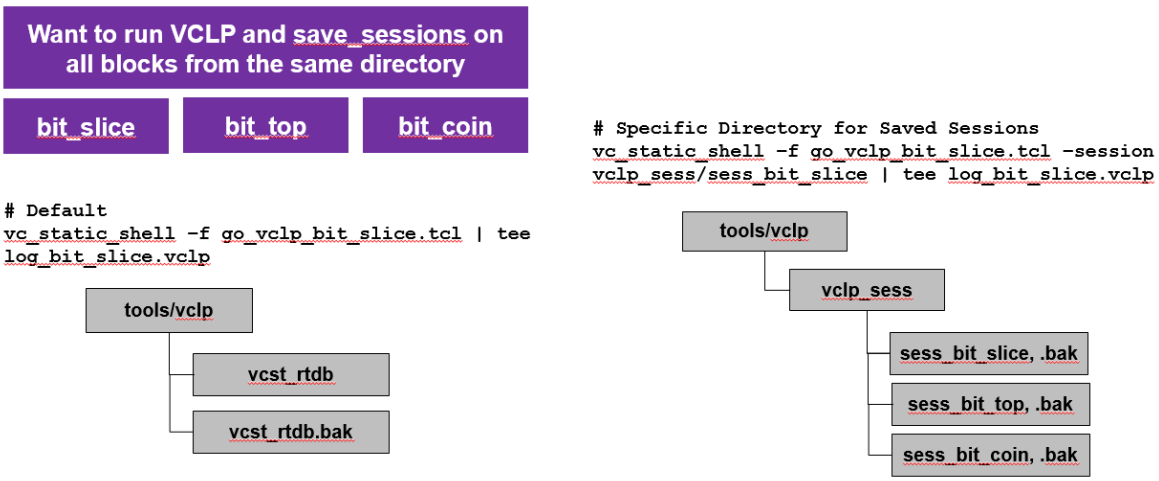


Figure 5.11 - VC LP Saved Sessions

When parsing UPF at the bit_coin level, you will see lots of UPF Parse Warnings found in the log file, examples of which are shown in Figure 5.12. These warning messages can be ignored in this case, and we are working to be able to store these into the persistent violation database, so they can be queried and processed, and well as create a mechanism to suppress or reduce the number of messages.

```
     50 Supplemental UPF Filename =
     51 [Info] UPF_FILE_PARSING: Upf file parsing
     52         Parsing top level Upf file '/global/gts_corpac2/bitcoin/users/tony/v1.1/upf/snps_bit_coin.upf'.
…
   8329 /global/gts_corpac2/bitcoin/users/tony/v1.1/upf/snps_bit_slice.upf:2: [Warning]
UPF_ATTR_INVALID_OBJECT: Invalid object for attribute
   8330         Instance 'bit_coin/bit_secure_15/slice_29' specified with '-elements' of
'set_design_attributes' cannot be specified for attribute 'enable_state_propagation_in_add_power_state'.
   8331 /global/gts_corpac2/bitcoin/users/tony/v1.1/upf/snps_bit_slice.upf:40: [Warning] TCL_OPT_NYI: Command
option not yet implemented
   8332         The command 'set_isolation' does not support the option '-name_suffix'.
…
  15564 /global/gts_corpac2/bitcoin/users/tony/v1.1/upf/snps_bit_slice.upf:92: [Warning] UPF_SPACWISOPSRC:
Source supply set of port attribute and isolation strategy on the same design object are conflicting.
  15565         Port attribute 'bit_coin/bit_secure_15/slice_31/data_valid' and isolation strategy
'bit_secure_15/slice_31/TOP/ISO_TOP2_MEM' on same design object 'bit_secure_15/slice_31/data_valid' have
conflicting source supply sets. Strategy source supply set is 'bit_secure_15/slice_31/TOP.SS_MEM_SW' and port
attribute source supply set 'bit_secure_15/slice_31/TOP.SSL'
```

Figure 5.12 - UPF Parse Warnings

Lastly, note for NOR Isolation which is used in the bit_coin design, VC LP requires an app_var to be used, this is shown in Figure 5.13.

NOR ISO

```
# File : go_vclp_bit_coin.tcl

set DESIGN_TOP bit_coin
set REPORTS_DIR ./reports

if ![[file exists $REPORTS_DIR]] {
  sh mkdir -p $REPORTS_DIR
}

source ./scripts/synopsys_vcst.setup

# NOR ISO
set_app_var enable_iso_nor true

analyze -f verilog -vcs "-f ../../rtl/files_${DESIGN_TOP}.f"
elaborate ${DESIGN_TOP}

load_upf ../../upf/snps_${DESIGN_TOP}.upf

redirect -file $REPORTS_DIR/check_lp.${DESIGN_TOP}.upf.rpt {check_lp -stage upf}

redirect -file $REPORTS_DIR/report_lp.${DESIGN_TOP}.list.rpt {report_lp -list}

save_session

exit
```

App var is needed

Figure 5.12 - NOR ISO app_var

Chapter 6, VCS NLP + Verdi Power Aware @RTL

Overview

VCS NLP (Native Low Power) is used at RTL stage to dynamically verify the design in the presence of UPF (power intent). Refer to figure 6.1 below, inputs to this flow are RTL, UPF and testbench (commonly written in UVM). For debug we use Verdi Power-Aware since the user needs the ability to visualize the power instrumented signals along with knowing the state of supply rails and power domains. The output from the tool to capture the switching activity for power analysis can be in the form of FSDB, SAIF or even VCD, in this flow we used FSDB.

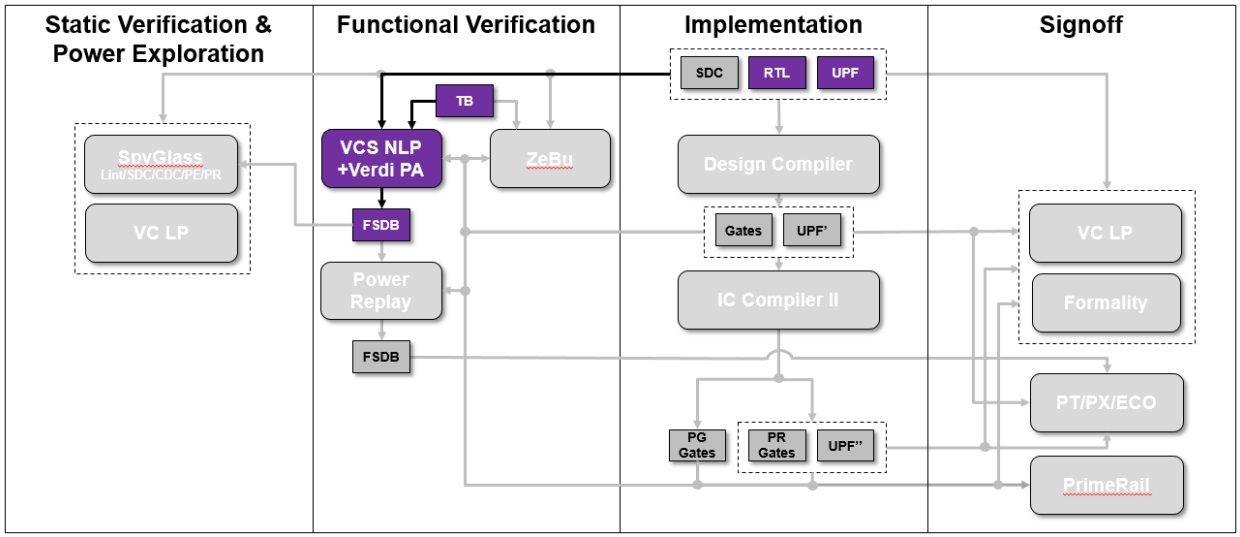


Figure 6.1 - VCS NLP@RTL in the Synopsys Low Power Flow and Methodology

Key VCS NLP features

- Support for all the commonly used low power techniques like
  - power gating, power gating with retention, low power standby, multi-voltage.
- Tape-out proven IEEE 1801 support
- Accurately model and simulate new low power states and behaviors
- Support for Verdi and X-propagation technologies
- Automatic generation of Low Power assertions to validate the UPF intent
- Automatic covergroup creation for Low power objects (PST, supply, etc)

Key Verdi Power-Aware features

- Support for debug of static and dynamic impact of power state changes on design through Verdi Power-Aware
- Plan and track the power aware verification coverage through the Verdi Coverage interface

In contrast to verifying the power intent statically (which VC LP does as example) the primary focus here should be to validate the same UPF dynamically - power and related control signal sequencing and effects of that. Figure 6.2 illustrates this point.

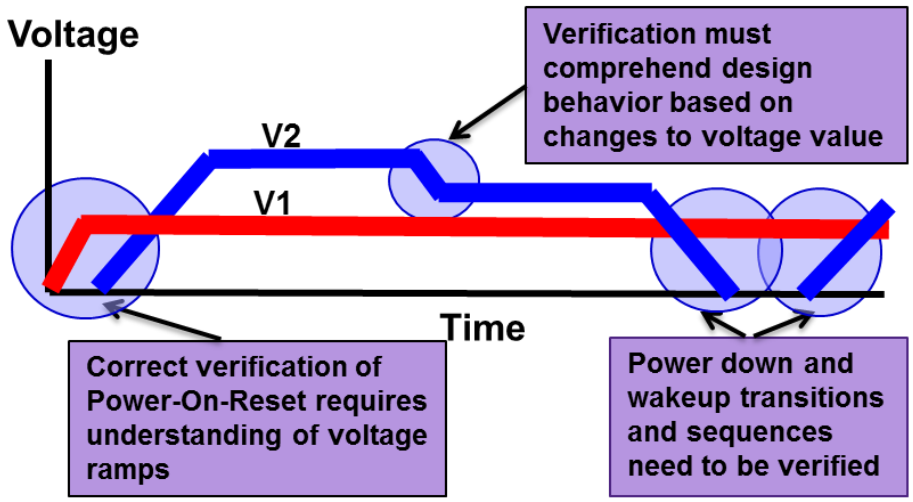
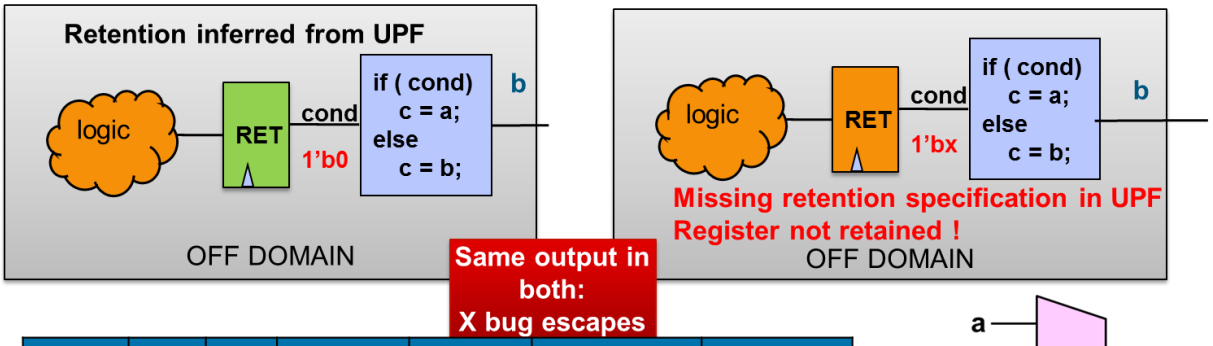


Figure 6.2 - VCS NLP@RTL Dynamic Low Power Verification

A recommended feature to also use especially in the context of VCS NLP is xprop which addresses the X-optimism issues that exist in RTL. In figure 6.3 below (top half) it shows the regular RTL simulation semantics where the output 'b' is same in both cases with and without retention. The table in figure 6.3 shows different possible outcomes based on simulation at RTL (without xprop), HW, GLS (netlist simulation) and finally RTL simulation with xprop.



cond	a	b	c(RTL)	c(HW)	c(GLS with circuit 1)	c(with XPROP tmerge)
x	0	0	0	0	0	0
x	0	1	1	0/1	x	x
x	1	0	0	0/1	x	x
x	1	1	1	0/1	1	1

Figure 6.3 – VCS NLP with xprop

Verdi Power-Aware has additional panes specifically for Low Power, figure 6.4 shows the screenshot of the tool. For example, in the UPF source pane you can see the UPF annotated with state values and these UPF objects (commonly UPF supply nets) can be dragged and dropped in nWave to see the values in waveform. Additionally, VCS-NLP instrumented signals like <sig>_UPF_ISO_PRE also can be seen in nWave.

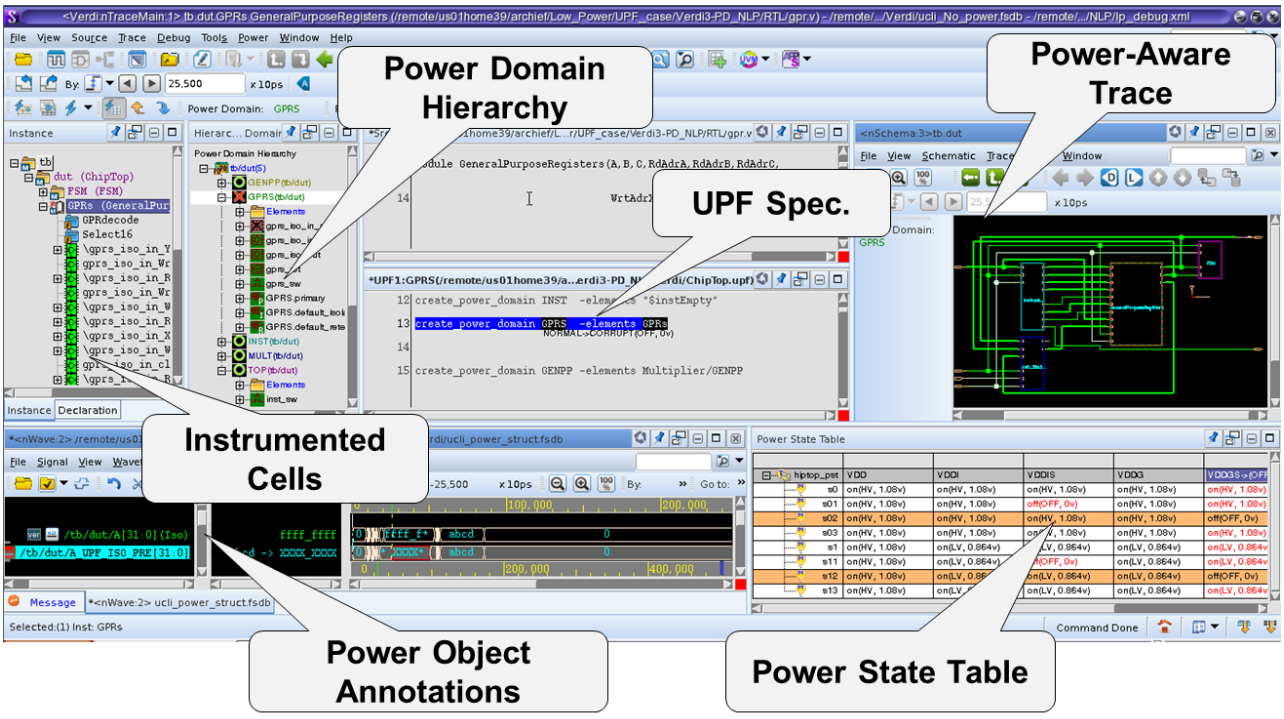


Figure 6.4 – Verdi Power-Aware@RTL

Another key aspect or metric useful for low power verification especially if you have a constrained random simulation environment is functional coverage metrics. With VCS NLP the tool can automatically create covergroups for UPF objects. For example, you can create PST state coverage using this switch -power=cov_pst_state

This UPF Low Power coverage metric can be merged with other metrics that you might have (code or functional) to get an overall score. There are two common ways to look at the reports you can view the HTML file or you can use Verdi to interactively visualize the coverage metrics.

Figure 6.5, shows the typical screenshot of how UPF LP coverage metric for PST might look like.

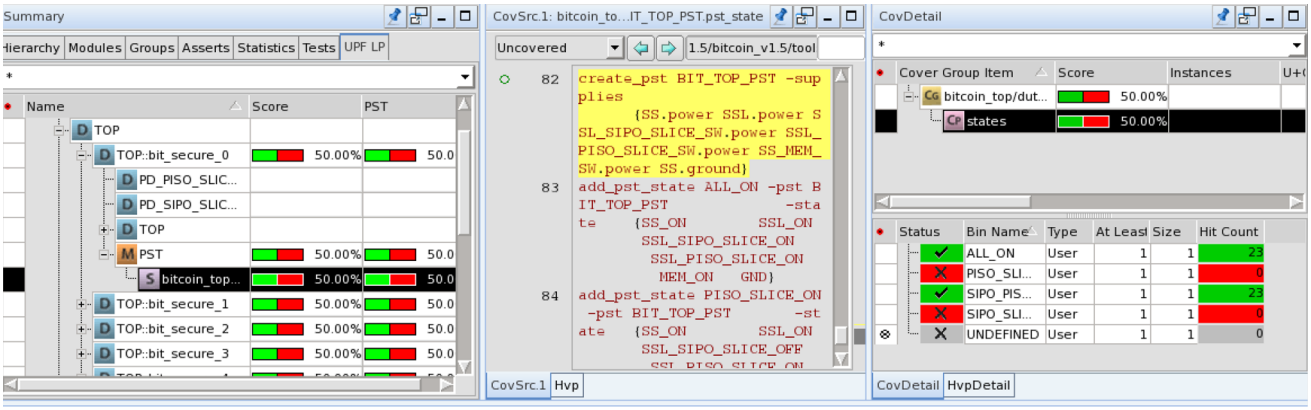


Figure 6.5 – Verdi Power-Aware@RTL UPF Low Power Coverage Metrics

Scripts & Mechanics

When compared with regular simulation flow in the simplest form for VCS NLP simulation you need just these two additional switches -upf <upf_file> -power_top <design/UPF top module>. In this flow we are passing few additional options since we have a macro (SRAM memory). To handle macros in NLP you can pass DB files using -power_config <config file>. For debug especially if the intention is to use this FSDB for power analysis the most common options will be -debug_access+pp -debug_region=lib+cell

Figure 6.6 shows the screenshot of a typical VCS compile command, note some additional options are being passed/shown below since they help in compile time turn-around for incremental compiles.

```
vcs  -partcomp -fastpartcomp=j4 -timescale=1ns/1ps -sverilog -l vcs.log +warn=noIWNF -ntb_opts uvm-1.2 +define+UV
M_POST_VERSION_1_1 +incdir+../../verification/include+../../verification/src+../../verification/env+../../verific
ation/tests+../../verification/hdl  -power=pst_state_msg+dump_hvp  -power_top bit_coin  +define+UPF +warn=noUPF_C
ONYI,noIWNF,noTFIPC -upf ../../upf/snps_bit_coin.upf -mvrpt=logs/RTL_DEBUG_UPF/mvsim_native_reports -xprop -power
=cov_pst_state -power_config ./scripts/config.tcl  -debug_region=lib+cell -debug_access+pp -lca -kdb +define+DUMP
 \
                ../../verification/tests/tb_top.sv ../../verification/hdl/bitcoin_top.sv \
               -F ../../verification/hdl/filelist.f -Mdir=output/RTL_DEBUG_UPF/csrc -o output/RTL_DEBUG_UPF/simv
 \
               -l logs/RTL_DEBUG_UPF/compile.log \
               -partcomp_dir=output/RTL_DEBUG_UPF/partitionlib
cp -R output/RTL_DEBUG_UPF/simv.daidir outputsfromvcs/simv_RTL_UPF.daidir
```

Figure 6.6 – VCS Compile Command

For running the simulation since we use UVM, you must pass +UVM_TESTNAME=<test>. Figure 6.7 shows the screenshot of a typical VCS runtime command again it shows some additional options to control the location of FSDB file, log file and control the number of transactions.

```
./output/RTL_DEBUG_UPF/simv +min_random_count=50 +max_random_count=50 -power ./scripts/lp.tcl  +fsdbfile+./logs/R
TL_DEBUG_UPF/dump.fsdb +SAIF_REPORT=./logs/RTL_DEBUG_UPF/bitcoin.saif -l logs/RTL_DEBUG_UPF/run.log \
        +ntb_random_seed=1 +timeout=200000 +UVM_TESTNAME=btcoin_lp_test +fsdb+all
```

Figure 6.7 - VCS Runtime Command

Bitcoin Experience

Assuming the suggested tools are installed and licenses available. To run Bitcoin with VCS NLP along with Verdi Power-Aware at RTL

```
% cd bitcoin_v1.1/tools/vcs_nlp
% make
```

The default 'make' target is to compile & run RTL simulation with UPF and dump FSDB. The below shows various arguments (most common) that you can pass to 'make' based on the intended flow.

Note: The make flow below shows compile+run together as part of the default target, you can split the flow into steps to compile and run separately (explained later).

1. To compile & run without dumping

   ```
   % make MODE=NODEBUG
   ```

2. To compile & run without UPF and without dumping

   ```
   % make MODE=NODEBUG LP=NOUPF
   ```

3. To compile & run with a different UPF but still with dumping

   ```
   % make UPF_FILE=<path to the upf file>
   ```

4. Default UVM transactions are 50, to compile & run it for longer or shorter period

   ```
   % make MAX_TRANS_CNT=<integer value>
   ```

The target for only compiling is

```
% make comp
```

This target can be used in combination with the above list arguments to 'make'. For example, if you wanted to only compile without DEBUG the command will be

```
% make comp MODE=NODEBUG
```

Likewise target for only running is

```
% make run
```

This target can be used in combination with the above listed arguments to 'make'. Just be aware though that some options are only applicable at compile not at runtime. For example, if you wanted to run the test longer (more transaction) then you could do this,

```
% make run MAX_TRANS_CNT=100
```

But you can't pick a different UPF file at runtime this can only be done at compile.

Also important to note, to keep the data unique, each flow (with or without UPF, with or without debug …) is written to its own unique directory. The directory structure that is followed is explained below,

Compile outputs go under 'output' directory

- Created based on Makefile arguments PHASE, MODE, LP
- Examples
  - make → simv/csrc written into output/RTL_DEBUG_UPF
  - make LP=NOUPF → simv/csrc written into output/RTL_DEBUG_NOUPF
  -

All log files including FSDB dump go under 'log' directory

- Same concept as compile, examples
  - make -> compile.log, run.log, dump.fsdb written into logs/RTL_DEBUG_UPF
  - make LP=NOUPF -> compile.log, run.log, dump.fsdb written into logs/RTL_DEBUG_NOUPF

For validate simulation results you must focus both on the functional checks as reported by UVM and NLP assertion summary captured at the end of simulation in the VCS run log. While you will not see any UVM failure message you will see that in Low Power message summary it has 23 warning messages as shown in figure 6.8

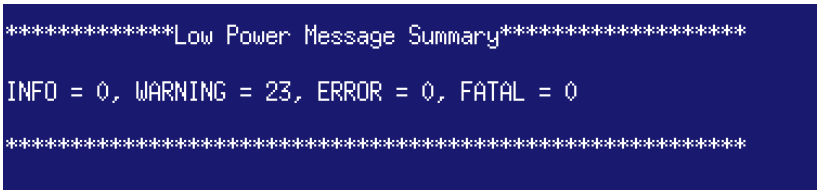


Figure 6.8 – VCS NLP Low Power Message Summary

The exact message coming out if you refer the run.log will be as shown in figure 6.9

```
[2736501 ps] [WARNING] [LP_ISOPN_OFF] State of isolation power net 'piso_sw_out' for isolati
on strategy 'ISO_PISO_SECURE' of power domain 'bitcoin_top/dut/PD_PISO_SECURE' changed to PA
RTIAL_ON.

UVM_INFO ../../verification/src/btcoin_drv.sv(144) @ 2738000: uvm_test_top.env.mast_drv [bit
coin_DRIVER] Completed transaction...
UVM_INFO ../../verification/src/btcoin_drv.sv(127) @ 2738000: uvm_test_top.env.mast_drv [bit
coin_DRIVER] Starting transaction...
UVM_INFO ../../verification/src/btcoin_drv.sv(129) @ 2738000: uvm_test_top.env.mast_drv [DRV
_RUN]
```

Figure 6.9 – Warning from VCS NLP Log File

This message reflects the power state of the isolation supply net which in more usual case could be potential electrical issue but in this case it can be waived. Since the isolation cell/policy in question is NOR isolation cell.

Another data point for VCS NLP is the coverage of PST states and PST transitions. Goal is to get high coverage for PST states including PST state transitions but currently the coverage numbers are low. The below figure 6.10 shows that our coverage is about ~50% which is on the lower end of what we would like to see. This is future work that will be addressed.

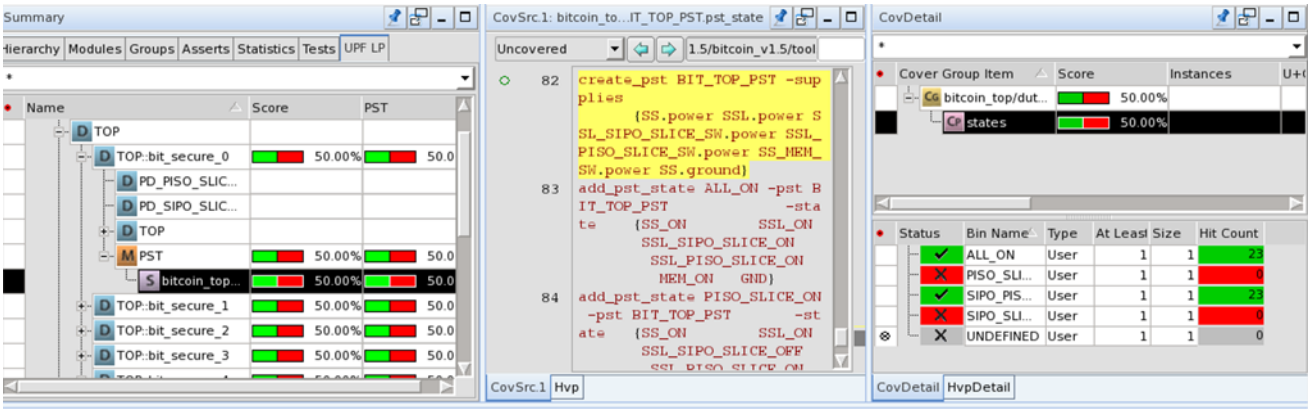


Figure 6.10 – VCS NLP Coverage Metrics

Lastly in general the runtime performance overhead of running with UPF is usually in the range of about 20% when compared with non-UPF simulation other things being the same. In this exercise the impact is more noticeable. This by itself isn't cause of concern since the total turn-around is still less than ~15 mins. Primary reason for the impact of UPF being more visible is because relative to UPF the Verilog code/structure is much simpler which is usually not a common scenario.

Chapter 7, Design Compiler and Power Compiler

Overview

Design Compiler (DC) is an important part of the low power solution and is the frontend portion of the implementation flow. DC synthesizes RTL into a gate level netlist, while satisfying all the design constraints for timing, area, test, and power. These constraints are specified by SDC (Synopsys Design Constraints) and UPF, and then generates a gate level netlist + UPF' to be used as input for ICC II as shown in Figure 7.1.

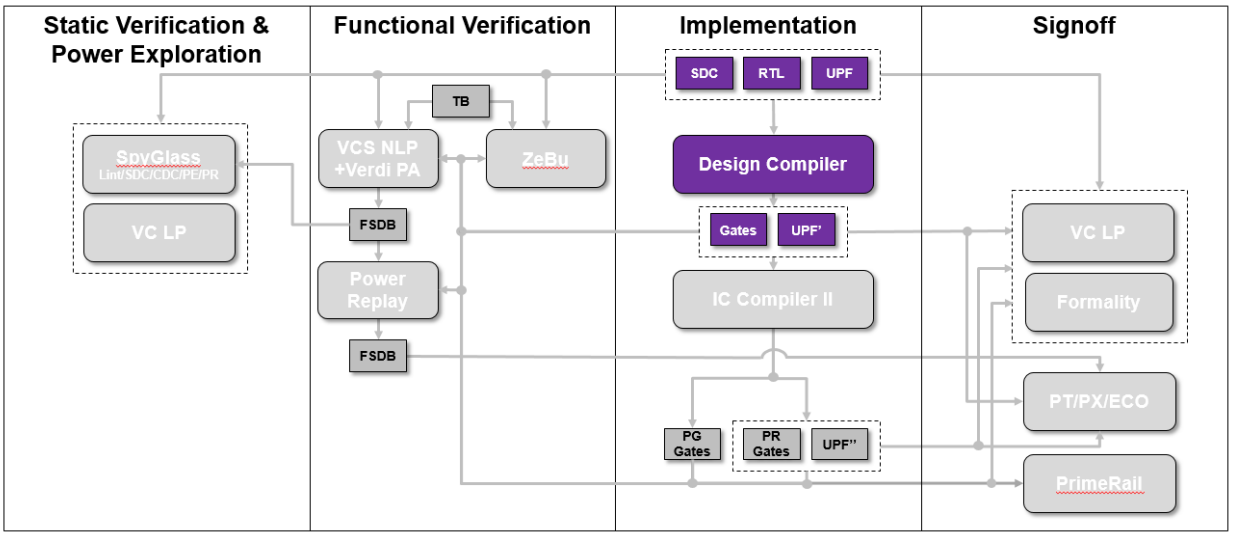


Figure 7.1 – DC/PwC in the Synopsys Low Power Flow and Methodology

At the core of the Design Compiler is Design Compiler Graphical, which performs 2 important tasks:

1) Provides concurrent optimization of timing, area, power, and test for high performance designs
2) Provides topographical physical synthesis, which allows designers to:
   - Accurately predict post-layout timing, area, and power during RTL synthesis, which ensures better correlation with the final physical design.
   - Generate a better starting point for place and route and eliminates costly iterations.

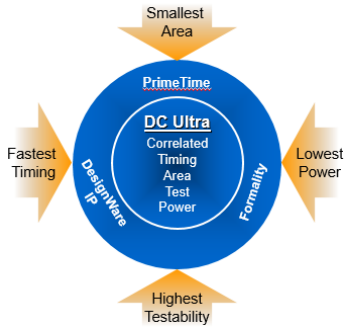


Figure 7.2 – Design Compiler Graphical

Design Compiler Graphical performs following tasks during synthesis:

- Optimization for multicorner-multimode (MCMM) designs
- Reduction of routing congestion during synthesis
- Improved area and timing correlation with ICC II
- Improved runtime and routability in ICC II
- Ability to create and modify floorplans using floorplan exploration with the Synopsys Physical Guidance (SPG) technology, providing enhanced placement and the capability to pass seed placement to ICC II to improve quality of results (QoR), correlation, and routability

Figure 7.3 - DC Graphical

In addition, DC Explorer is an important tool in the DC product family as it can improve your productivity during development of RTL and constraints by enabling fast synthesis in the early design stages. Using DC Explorer provides the following benefits:

- Developing new RTL and integrating it with third-party IP where many legacy RTL blocks can be a time-consuming process when designers lack a fast and efficient way to explore and improve the data
- Fix design issues
- Create a better starting point for RTL synthesis.

DC Explorer overcomes these problems by allowing early RTL exploration, leading to a better starting point for RTL synthesis and accelerating overall design implementation.

Design Compiler tool family also includes following related tools

- HDL Compiler Tools
  - Reads HDL files and performs translation and architectural optimization of the designs
- DesignWare Library
  - A DesignWare Library is a collection of reusable circuit-design building blocks (components) that are tightly integrated into the Synopsys synthesis environment
- DFTMAX™
  - The Synopsys test synthesis solution
  - Used for comprehensive power-aware test
  - Provides high compression for pin-limited test
- Power Compiler
  - Low power synthesis for many powerful low power implementation techniques, as shown in Figure 7.4.
  - Offers a complete methodology for power including analyzing and optimizing designs for static and dynamic power consumption
  - Provides Multi-voltage synthesis and power management insertion based on UPF

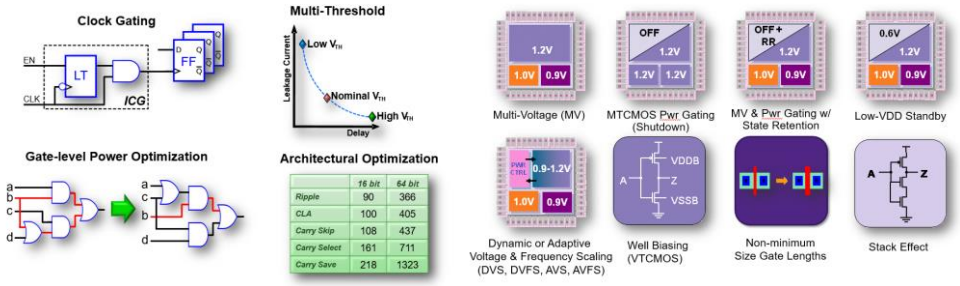


Figure 7.4 - Low Power Techniques

Scripts & Mechanics

In this section, we'll discuss the basics of starting and working with DC shell, and go over a basic DC script. Follow these steps to start dc_shell.

```
% dc_shell -topographical
```

Note that DC's Major release cycle is 9 months, and SP releases every 6 weeks, and Bug patches release as needed. Following is a typical DC shell prompt after invoking the tool

```
                Design Compiler Graphical
                Power Compiler (TM)
                Design Compiler(R)

        Version L-2016.03-SP5 for linux64 - Oct 14, 2016

            Copyright (c) 1988 - 2016 Synopsys, Inc.
   This software and the associated documentation are proprietary to Synopsys,
 Inc. This software may only be used in accordance with the terms and conditions
 of a written license agreement with Synopsys, Inc. All other use, reproduction,
            or distribution of this software is strictly prohibited.
Initializing...
Initializing gui preferences from file  /remote/us01home1/gargoyle/.synopsys_dv_prefs.tcl
dc_shell -topo>
```

Figure 7.5 - dc_shell prompt

The following is a basic script, which goes from reading the design, sourcing constraints, performing the optimization step, and finally writing out the finished design files for ICC II.

```
# typical hidden file for previous setup (it will be automatically sourced at start up)
source scripts/.synopsys_dc.setup
# directory for DC RTL elaboration files
define_design_lib -path ./work work

read_verilog [list ../../rtl/bit_slice.v ../../rtl/sipo.v ../../rtl/piso.v]
current_design bit_slice

Link

# operating condition and  timing constraints
source -e -v ../../sdc/bit_slice.sdc
load_upf ../../upf/snps_bit_slice.upf
source -e -v scripts/snps_set_voltage.tcl
check_mv_design
set DESIGN_NAME bit_slice
set RESULTS_DIR ./results_bit_slice

compile_ultra -no_autoungroup –scan

change_names -rule verilog –hier

write -f verilog -hier -o ${RESULTS_DIR}/${DESIGN_NAME}.vg
write -f ddc -hier -o ${RESULTS_DIR}/${DESIGN_NAME}.ddc
save_upf ${RESULTS_DIR}/${DESIGN_NAME}.upf
write_sdc ${RESULTS_DIR}/${DESIGN_NAME}.sdc
write_script -output ${RESULTS_DIR}/${DESIGN_NAME}.script
write_parasitics -output ${RESULTS_DIR}/${DESIGN_NAME}_compiled.spef
write_sdf ${RESULTS_DIR}/${DESIGN_NAME}_compiled.sdf
```

Figure 7.6 - Simple DC Script

There are two important DC variables, typically sourced in the hidden setup file during start up: the target_library and link_library variables.

The target_library variables points to the location of the liberty file or .db files for the standard cells (and/or/inv/registers) which will be used to synthesize the design.

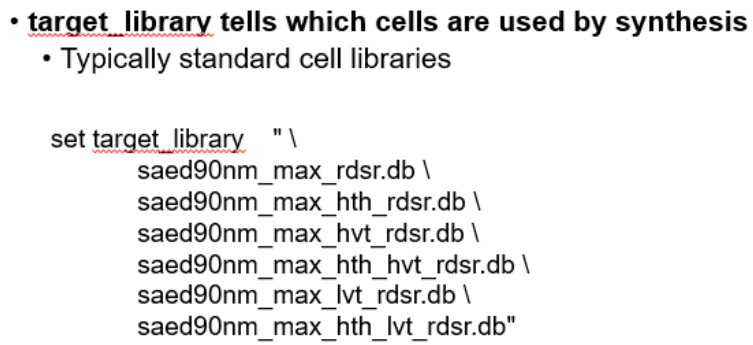


Figure 7.7. Target Library Variable

Secondly, the link_library will point to all the cells which maps to the design.

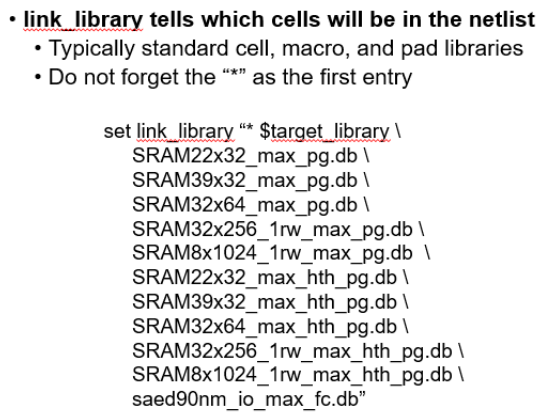


Figure 7.8 - Link Library Variable

Bitcoin Experience

To synthesize bit_coin with Design Compiler, first make sure your software and licenses are configured and installed. Then, simply type the following commands:

```
% cd bitcoin_v1.1/tools/dc
% make all
```

Model-Based Hierarchical Synthesis Flow

From the Makefile, there are 3 synthesis targets followed by two data generation steps. The scripts used for the targets are shown here, and can be used individually for testing and experimentation.

```
% dc_shell-t -topo -f scripts/bit_slice.tcl |tee logs/log.dc.bit_slice
% dc_shell-t -topo -f scripts/bit_top.tcl |tee logs/log.dc.bit_top
% dc_shell-t -topo -f scripts/bit_coin.tcl |tee logs/log.dc.bit_coin
```

Recall from the block diagram that bit_coin is quite modular, suggesting that a Multiply-Instantiated Module (MIM) flow may be appropriate. This is exactly what we implement in this design, and for Design Compiler this implies a bottom up flow. This results in the three targets shown above.

In **`bit_slice.tcl`** we compile the RTL, perform scan insertion, and create a Block Abstract Model (BAM) that can be used as we move up the hierarchy. This saves runtime as well as simplifying the scripts for the parent hierarchical levels. The BAM contains all the logical and physical information required at the bit_top level, as well as the power intent specified in the UPF used to compile bit_slice.

The **`bit_top.tcl`** script reads in the BAM for bit_slice instead of requiring the source files. Once bit_top is implemented, a BAM is created to be used at the bit_coin level.

The **`bit_coin.tcl`** script reads in both the bit_top and bit_slice BAMs (remember bit_top used the bit_slice BAM) and uses these to synthesize at the bit_coin level.

Note that this flow results in the need for a UPF file that has provisions for loading the lower level UPF files when a flat flow is required. This can be seen in the following files:

```
bitcoin_v1.1/upf/snps_bit_coin.upf
bitcoin_v1.1/upf/snps_bit_top.upf
```

You will set the following to select when the lower level UPF files are loaded in bit_top:

```
if { $synopsys_program_name != "dc_shell"} {
  for {set i 0} {$i < 32} {incr i} {
    load_upf -scope slice_${i} $UPF_DIR/snps_bit_slice.upf
  }
}
```

The UPF for bit_slice is loaded onto the 32 instances seen in bit_top.

New Low Power Cell – NOR Isolation

Isolation cells are special cells designed to protect logic which is powered from drivers which have had their power shut down. These floating inputs can cause anomalous behavior in powered logic. A typical circuit using isolation is shown in Figure 7.9.

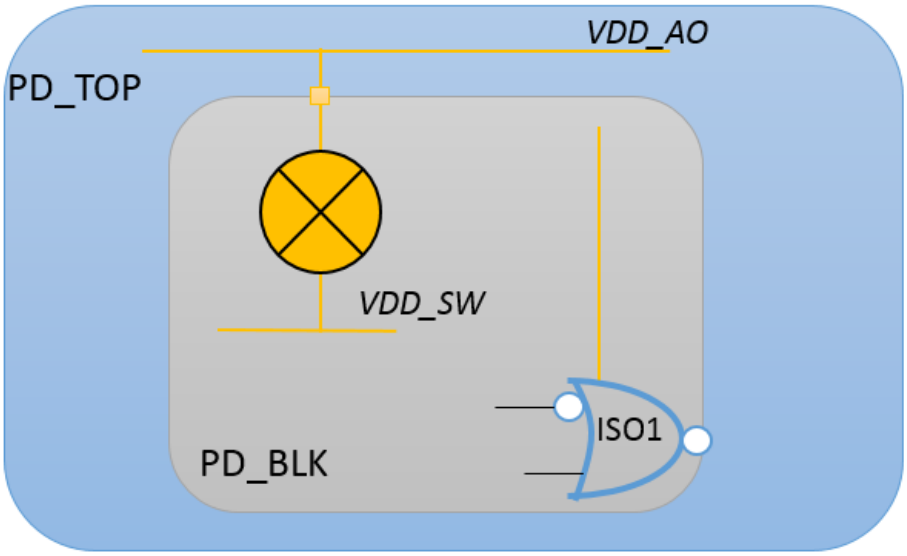


Figure 7.9 – NOR Isolation Cell

Typical UPF for such an application is as follows:

```
set_isolation ISO1 \
  -domain PD_BLK \
  -isolation_supply_set SS_VDD_AO \
  -clamp_value 0 \
  -applies_to outputs
set_isolation_control ISO1 -domain PD_BLK \
  -isolation_signal {control[0]} \
  -isolation_sense high \
  -location self
```

The special power management cells appropriate for this type of design have historically been:

- Single-rail cells with a floorplanned area in PD_BLK to support connection to VDD_AO. This adds complexity to the physical design of the block, but can result in a smaller design.
- Dual-rail Isolation cells with a backup power pin which can be connected to VDD_AO. These cells have a primary power that can be connected to VDD_SW so placement does not require extra floor planning, just extra routing resources for the backup power. These cells are typically larger than a standard cell but offer an ease of use advantage.

The new NOR-isolation cell has advantages over both cells. It is a single-rail cell that can be placed in a shutdown region but still provide clamping to powered logic. The requirements are that the output be clamped to “0” and that the isolation enable signal be active high. For these cells, the active high enable signal is used to clamp the output to ground, without an active supply on the cell.

You may wonder how this information is conveyed to the tools -- new Liberty attributes required are shown in Figure 7.10.

```
/* Library Model – Example */
cell (ISONOR) {
  is_isolation_cell : true;
  pg_pin (VDD) {
    pg_type : "primary_power";
    permit_power_down : true;
  }
  pin (D) {
    related_power_pin : "VDD";
    related_ground_pin : “VSS";
    isolation_cell_data_pin : true;
    alive_during_power_up : true;
    …
  }
pin (ISO) {
    related_power_pin : "VDD";
    related_ground_pin : “VSS";
    isolation_cell_enable_pin : true;
    alive_during_partial_power_down : true;
    …
  }
  pin (Q) {
    related_power_pin : "VDD";
    related_ground_pin : “VSS";
    alive_during_partial_power_down : true;
    power_down_function : “(!VDD*!EN) + VSS";
    function : “(ISO’)*D";
    …
```

Figure 7.10 – Liberty Attributes for NOR Isolation Cell

The attributes in red are described in the Library Compiler User Guide available on SolvNet, and allow the tools to understand the relationships between the PG and signal pins for the cell. These attributes are required for proper understanding of the cells, and are very important for use of these cells.

As low power designs progress, many interesting and clever cells are being designed. Liberty is actively being enhanced to keep up with our users, and low power designers need to keep up to speed on the cells available to them.

The UPF for NOR isolation strategies is different as well. Support has been enhanced to use the following to infer NOR isolation cells:

```
set_isolation ISO1 \
  -domain PD_BLK \
  -isolation_supply_set {} \
  -clamp_value 0 \
  -applies_to outputs
```

Note the isolation supply is specified as "{}" telling the tools to choose NOR isolation from the target library.

In addition, Design Compiler has the capability to use the original UPF coding and select a NOR isolation cell if one is available in the target_library for Design Compiler, preferring it to dual rail cells. This is the style we use in our bit_coin design currently.

Chapter 8, DFT (Design for Test)

Overview

DFTMAX performs the scan stitching and scan compression insertion on the gate level netlist. This is part of the frontend portion of the implementation flow in Synopsys Test methodology. DFTMAX respects the power intent of the design and implements scan stitching and scan compression insertion while satisfying all the design constraint for timing, area, and test. The scan and compression inserted netlist is handed off to ICC II for P&R. Finally, TetraMAX™ performs ATPG on the PG netlist from ICC II to generate high quality manufacturing test patterns.

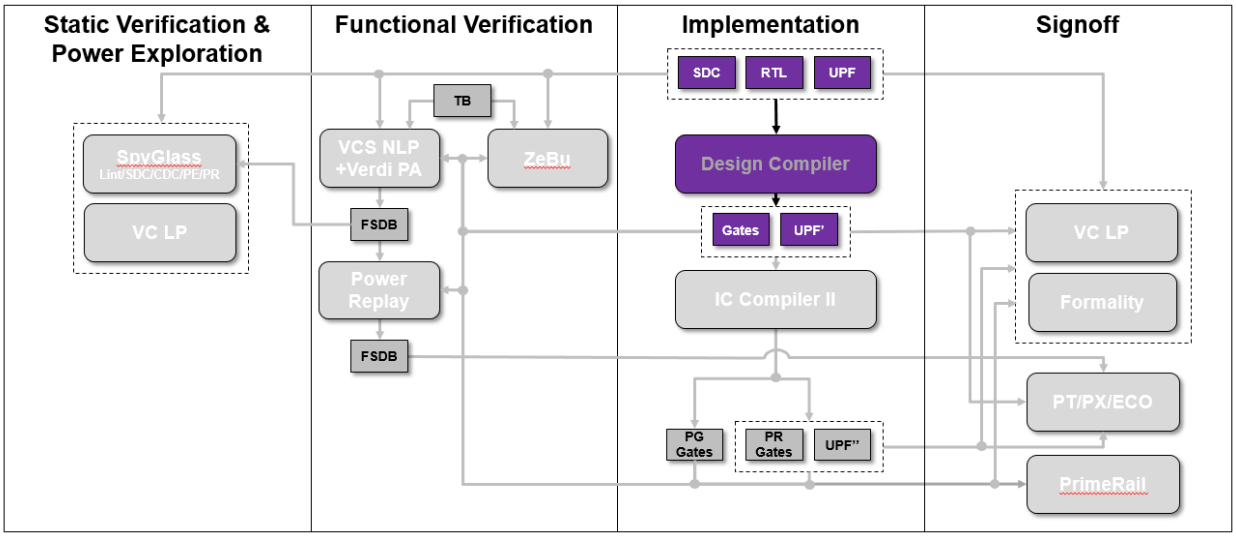


Figure 8.1 - DFT in the Synopsys Low Power Flow and Methodology

DFTMAX Overview

The Synopsys DFTMAX tool provides a one-pass test synthesis solution that enables you to implement test data volume scan compression without affecting the functional, timing, or power requirements of your design.

DFTMAX scan compression provides the following key benefits and features:

- Significant test time and test volume reduction
- Same high test coverage and ease of use as standard scan
- No impact on design timing or design physical implementation
- One-pass test compression synthesis flow

DFTMAX tool also provides additional capabilities beyond scan compression, such as boundary-scan insertion, core wrapping with maximized register reuse, and automatic test point insertion.

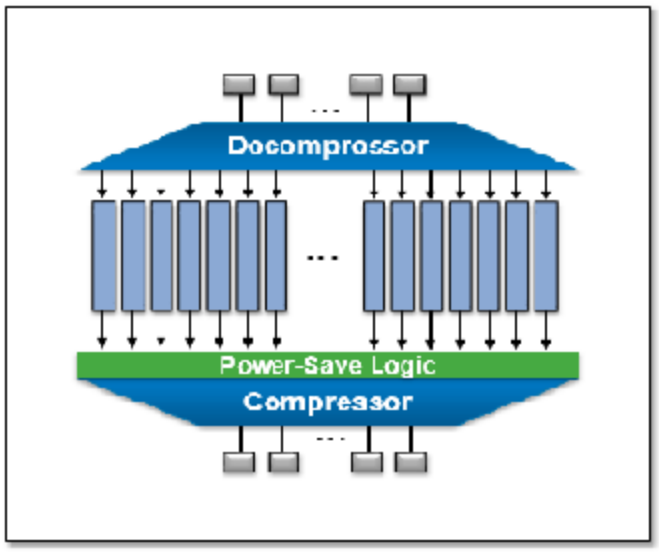


Figure 8.2 - DFTMAX Compression Architecture

TetraMAX Overview

TetraMAX ATPG automatically generates high quality manufacturing test patterns. TetraMAX ATPG is optimized for a wide range of test methodologies and integrated with DFTMAX. The unparalleled ease-of-use and high performance provided by TetraMAX ATPG allows RTL designers to quickly create efficient, compressed test patterns for even the most complex designs. TetraMAX provides the following key benefits and features:

- Improves product quality with comprehensive fault model support and power-aware test patterns
- Increases designer productivity by leveraging integration with Synopsys test compression tools
- Generates test patterns for even the largest and most complex SoCs
- Enables faster yield ramp by quickly isolating defect locations

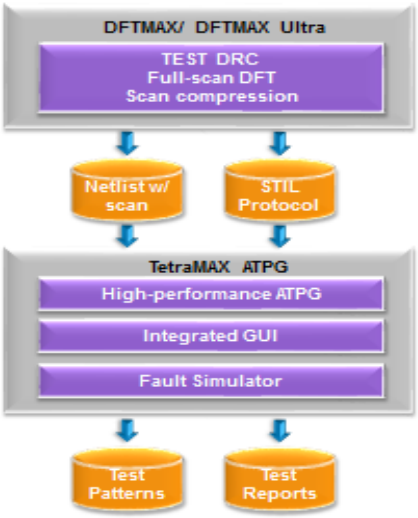


Figure 8.3 - Integrated Test Flow using TetraMAX ATPG

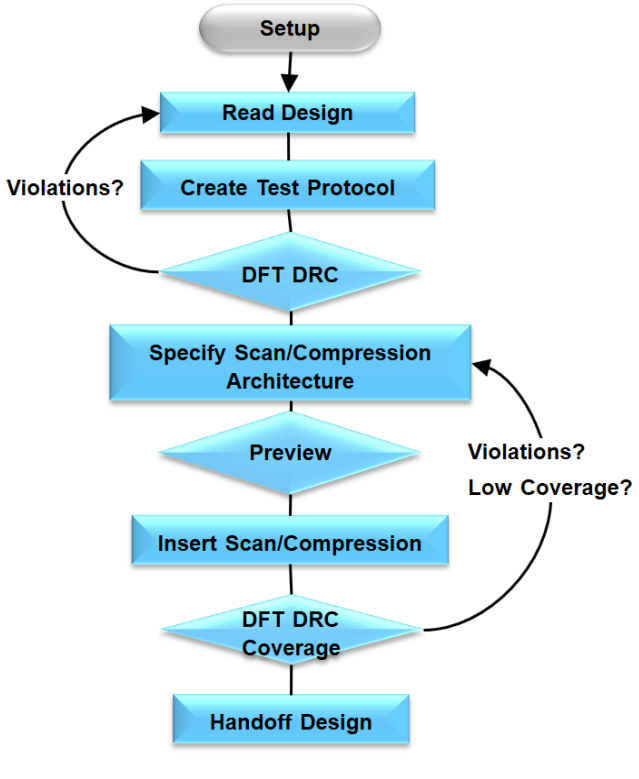
Scripts & Mechanics

DFTMAX can be invoked from within the following synthesis tools:

- Design Compiler
- Design Vision

Please refer to Chapter 7 for invoking the Synthesis tools and setting up the environment in Design Compiler/Design Vision.

In a typical scan insertion flow, after setting up the environment in Design Compiler, the design is read by specifying the appropriate file read commands. Use the current_design and link commands to link the top-level of the current design. You can then perform pre-DFT design rule checking (DRC) to activate test design rule checking. However, before running this process, you must first define the related DFT signals (clock, asynchronous set/reset, constant pins, etc) and create a test protocol that includes the information needed to perform DRC. After fixing all violations during the RTL test DRC process, you configure scan architecture and specify test ports, define test modes, and identify and mark any cells that you do not want to have scanned. You can set many of these configuration parameters by using commands such as set_scan_configuration, set_dft_signal, or set_scan_element.




```
source setup.csh

read_ddc ./bit_slice/results/bit_slice.ddc

set_dft_signal -view existing_dft -type ScanClock -port [list
hclk lclk] -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port reset -
active 0

create_test_protocol

dft_drc -verbose

set_scan_configuration -chain_count 4 -clock_mixing no_mix -
voltage_mixing false -power_domain_mixing false -reuse_mv_cells
false
report_scan_configuration


preview_dft -show all -test_points all


insert_dft


dft_drc

compile_ultra -incremental -scan


write -format verilog -hierarchy -output
./results_bit_slice/bit_slice_dft.vg
write_test_protocol -output bit_slice.spf
create_block_abstraction
write -f ddc -h -o $RESULTS_DIR/${DESIGN_NAME}_bam.ddc
```

Figure 8.4 - Typical Scan Insertion Flow with a sample script

Before performing scan insertion, you can preview your scan design by running the preview_dft command. This command generates a scan chain design that satisfies scan specifications on your current design and displays it. This allows you to preview your scan chain design without synthesizing it and change your specifications as necessary. After configuring and previewing your design, assemble

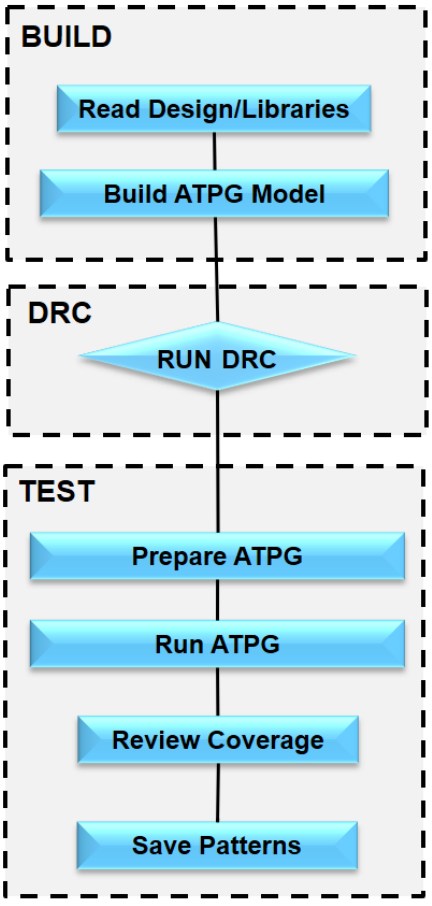
the scan chains by using the insert_dft command. After you perform DFT insertion, you can perform post-DFT design rule checking again to ensure that no new violations have been introduced into your design. These checks are more comprehensive than those in pre-DFT DRC, and they check for the correct operation of the scan chain. After you correct all the reported violations, you can proceed with the next step of handing off the design to the downstream tools.

TetraMAX

TetraMAX can be launched in either shell mode or using the TetraMAX GUI

```
tmax my_command_file -shell
tmax my_command_file
tmax -help
```

In a basic ATPG flow, you read your netlist and the verilog library models into TetraMAX using the read_netlist command. This creates an in-memory ATPG design model. You can then perform design rule checking (DRC) to activate test design rule checking.




```
read_netlist ../../lib/verilog/saed32nm_hvt.v -library
read_netlist ...
read_netlist ../pt/signoff_data/bit_coin.v.gz


run_build bit_coin

set_drc -allow_unstable_set_resets
add_pi_constraints 0 scan_enable

run_drc results_bit_coin/bit_coin.compression.spf

report_clocks -gating

set_atpg -power_budget 75
set_atpg -power_effort low
set_atpg -shift_power_budget 75
set_atpg -shift_power_effort low
set_atpg -quiet_chain_test
set_atpg -fill adjacent

add_faults -all
run_atpg -auto

report_power
analyze_faults -class AN

write_patterns ./results_bit_coin/patterns.stil -format
stil -unified_stil_flow -replace
write_testbench -input  ./results_bit_coin/patterns.stil -
output results_bit_coin/tb_tmax.v -replace
```

Figure 8.5 – TetraMAX ATPG Script Example

But before starting DRC, make sure you specified the appropriate STIL procedures and DRC settings. During DRC, TetraMAX ATPG performs a set of checks to ensure that the scan structure is correct and to determine how to use the scan structure for test generation and fault simulation. These checks include ensuring that the scan chains operate properly, identifying scan cells and ensuring that clocks obey the required rules. To prepare for ATPG, you can specify general ATPG settings, set up the fault list, select the fault model (stuck-at, IDDQ, path delay, hold time, transition, or bridging), select the pattern source (internal, external, or random patterns). ATPG creates a sequence of test patterns that enable an ATE to distinguish between the correct circuit behavior and the faulty circuit behavior caused by the defects. After running ATPG, you can review a set of output reports that provide coverage information on faults, patterns, coverage and other data relevant to ATPG. Based on these reports, you can make incremental adjustments to meet your ATPG goals, such as obtaining a good balance between pattern compaction and execution speed. Finally, you can write out the patterns in different format and also a testbench to simulate the patterns in VCS simulator.

Power-Aware DFT Features

The increasing presence of multiple voltage domains in designs has resulted in the need for DFT insertion to respect the power intent of the design. DFTMAX reads in your UPF and respects the power intent of the design and during DFT insertion attempts to minimize voltage and power-domain crossings to reduce the number of level shifters and isolation cells required. By default, the tool will not try to stitch the scan chains across different voltage and power domains, but the user can override this behavior using the -voltage_mixing and -power_domain_mixing of the set_scan_configuration command. The tool can also reuse existing level shifters and isolation cells if they are already on the scan path. Note that only combinational multi-voltage cells are reused and sequential multi-voltage cells are not.

Another feature in DFTMAX is functional output gating that can reduce power during the shift mode. During scan testing, while scan data shifts through scan chains, the functional logic driven by the scan flip-flops also toggles. This can cause increased power dissipation during testing, which could damage the design under test. When you enable functional output gating, the tool inserts gating logic to suppress toggling on the functional output of scan flip-flops that either you specify or the tool automatically selects. The tool uses AND-gating or OR-gating logic, depending on which constant value most reduces toggling from other ungated signals entering the fanout logic cone.

In a compressed scan architecture, an XOR compression tree combines the shift outputs from all compressed chains into a reduced set of scan out data signals. This XOR compression tree is needed only during scan shifting in that codec's compressed scan mode. At other times, the compression logic is not needed, but it will still toggle when the tail scan flip-flops of the compressed chains toggle. This is of particular concern during mission mode, when the flip-flops are clocked at their full operating frequency. To address this, there is a feature in DFTMAX that can insert gating at the inputs to the XOR compression tree to eliminate this toggling activity and reduce power consumption in other modes of operation.

During scan shift, there is significant toggle activity in the scan chains. At high scan shift frequencies, this can result in higher-than-desired shift power consumption. The shift power groups feature helps reduce power consumption during scan shift in DFTMAX compressed scan modes. This feature inserts

AND gates at the decompressor outputs before each compressed scan chain. The chains are gated in groups that are controlled by a shift power control (SPC) chain

Power-Aware ATPG Features

A typical ATPG run targets as many faults as possible within a particular pattern. However, this approach can cause unintended ATE failures for designs containing a large number of flip-flops that toggle at any given time.

The TetraMAX power-aware ATPG feature calculates the fanout of clock-gating structures and other clock sources during DRC. This approach enables you to specify capture and shift power budgets for generating power-aware ATPG vectors. You can specify a budget as a percentage of scannable flip-flops and thereby limit the number of flip-flops that can toggle.

TetraMAX ATPG lowers the overall peak and average flip-flop switching by selectively turning on and off the respective clock-gating cells which control the flip-flops. This selective switching affects capture for stuck-at testing and launch and capture for transition fault testing.

Power-aware ATPG is not intended to be used for power analysis. TetraMAX ATPG efficiently estimates the relative power of test patterns, which generally correlates well with actual power consumption. However, you should be advised that this approach is not a precise calculation of the actual power metrics. Performing a full power analysis during ATPG may cause an unacceptable increase in runtime and is therefore not used for power-aware ATPG.

Bitcoin Experience

DFT in Bitcoin is run in the same session as DC synthesis. First make sure your Design Compiler software and licenses are configured and installed. Then, simply type the following commands:

```
% cd bitcoin_v1.1/tools/dc
% make bit_slice
% make bit_top
% make bit_coin
```

From the Makefile, there are 3 synthesis targets which also execute their respective DFT targets. The scripts used for the targets are shown here, and can be used individually for testing and experimentation.

```
% dc_shell-t -topo -f scripts/bit_slice.tcl |tee logs/log.dc.bit_slice
% dc_shell-t -topo -f scripts/bit_top.tcl |tee logs/log.dc.bit_top
% dc_shell-t -topo -f scripts/bit_coin.tcl |tee logs/log.dc.bit_coin
```

In **`bit_slice.tcl`**, you will find a script dft_bs.tcl being sourced. This dft_bs.tcl script performs scan insertion. The DFT insertion flow in bit_slice is basic scan insertion with autofix enabled. The voltage_mixing and power_domain_mixing is disabled for the scan chain stitching. After implementing bit_slice a Block Abstraction Model (BAM) is created to be used at bit_top level.

In **`bit_top.tcl`**, you will find the BAM for bit_slice is read and the dft script dft_bt.tcl being sourced. Bit_top dft insertion flow is identical to the bit_slice flow. Once bit_top is implemented, a BAM is created to be used at the bit_coin level.

The **`bit_coin.tcl`** script reads in both the bit_top and bit_slice BAMs and the dft_script dft_bc.tcl is read. At the bit_coin level, scan compression is enabled to implement compression. Also, the power_domain_mixing was enabled to mix flops of different power domains to keep the scan chain within the limits. After scan insertion and incremental compile, the STIL Protocol File (SPF) is written out for TetraMAX to perform DRC and run ATPG.

ScanDEF is written out at each level for scan chain reordering and repartitioning in ICC II.

The preview_dft -show all command before the insert_dft reports the operating condition and the power domain of a scan cell whenever a scan path crosses a voltage or power domain. It also indicates whether a scan cell is driving a level shifter or an isolation cell. This provides an idea of the scan chain architecture with voltage and power_domain mixing information.

Below is the snippet of the preview_dft report of the bit_slice level.

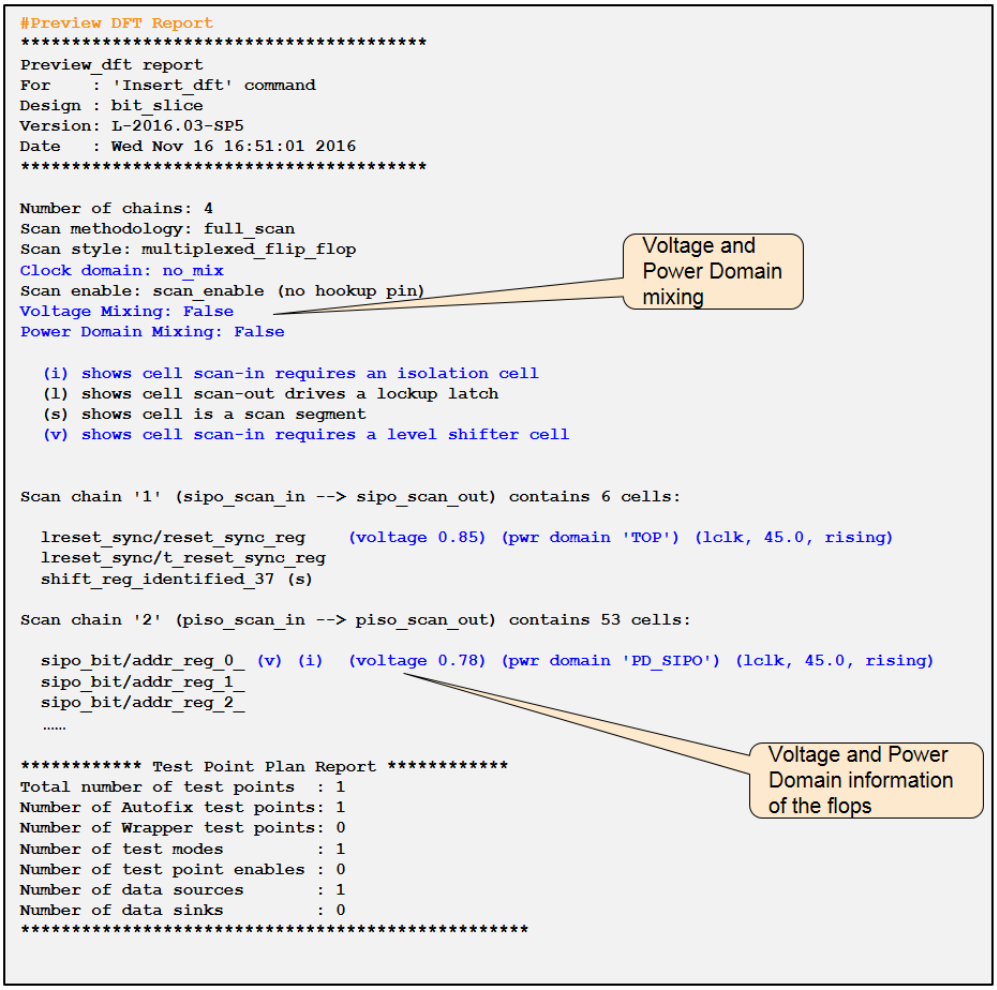


Figure 8.6 – bit_slice preview_dft Report

Below is the snippet of the preview_dft report of the bit_coin hierarchy:

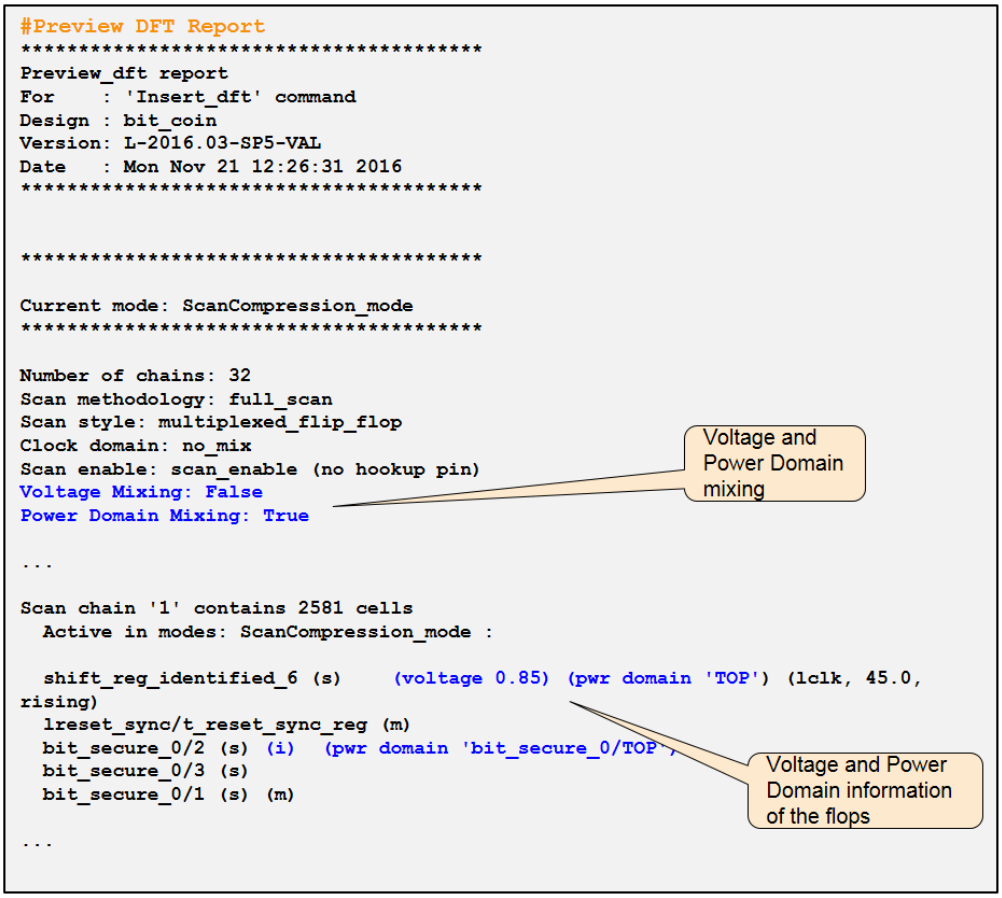


Figure 8.7 – bit_coin preview_dft Report

The scan architecture shown by preview_dft was as expected for bit_slice, bit_top and bit_coin. Another important output file that needs to be reviewed is the ScanDEF file that also has voltage and power domain information of the flops as part of the PARTITION names of the chains.

Below is the snippet of the scandef file at the bit_slice and bit_coin level respectively:

```
#SCANDEF
NAMESCASESENSITIVE ON ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN bit_slice ;

SCANCHAINS 4 ;

- 1
+ START PIN sipo_scan_in
+ FLOATING lreset_sync/reset_sync_reg ( IN SI ) ( OUT Q )
           lreset_sync/t_reset_sync_reg ( IN SI ) ( OUT Q )
+ ORDERED temp_0_reg ( IN SI ) ( OUT Q )
          temp_1_reg ( IN D ) ( OUT Q )
          temp_2_reg ( IN D ) ( OUT Q )
          load_data_from_memory_reg ( IN D ) ( OUT Q )
+ PARTITION lclk_45_45_0.850000_TOP
+ STOP PIN sipo_scan_out ;

- 2_UPF2
+ START sipo_bit/addr_reg_0_ Q
+ FLOATING sipo_bit/addr_reg_1_ ( IN SI ) ( OUT Q )
           sipo_bit/addr_reg_2_ ( IN SI ) ( OUT Q )
        ….
           sipo_bit/wr_reg ( IN SI ) ( OUT Q )
+ PARTITION lclk_45_45_0.780000_PD_SIPO
+ STOP sipo_bit/wr_tmp_reg SI ;
```

Figure 8.8 – bit_slice SCANDEF snippet

```
#SCANDEF
NAMESCASESENSITIVE ON ;
DIVIDERCHAR "/" ;
BUSBITCHARS "[]" ;
DESIGN bit_slice ;

SCANCHAINS 4 ;

- 1
+ START PIN sipo_scan_in
+ FLOATING lreset_sync/reset_sync_reg ( IN SI ) ( OUT Q )
           lreset_sync/t_reset_sync_reg ( IN SI ) ( OUT Q )
+ ORDERED temp_0_reg ( IN SI ) ( OUT Q )
          temp_1_reg ( IN D ) ( OUT Q )
          temp_2_reg ( IN D ) ( OUT Q )
          load_data_from_memory_reg ( IN D ) ( OUT Q )
+ PARTITION lclk_45_45_0.850000_TOP
+ STOP PIN sipo_scan_out ;

- 2_UPF2
+ START sipo_bit/addr_reg_0_ Q
+ FLOATING sipo_bit/addr_reg_1_ ( IN SI ) ( OUT Q )
           sipo_bit/addr_reg_2_ ( IN SI ) ( OUT Q )
        ….
           sipo_bit/wr_reg ( IN SI ) ( OUT Q )
+ PARTITION lclk_45_45_0.780000_PD_SIPO
+ STOP sipo_bit/wr_tmp_reg SI ;
```

Figure 8.8 – bit_coin SCANDEF snippet

The scandef file written out at bit_slice, bit_top and bit_coin level were as expected with the correct partition names and floating and ordered elements.

You can also perform ATPG on the bit_coin design. To run ATPG using TetraMAX make sure software and licenses are configured and installed. Then, type the following commands:

```
% cd bitcoin_v1.1/tools/dc
% make bit_slice
```

In the Makefile, there is a target which executes tmax. The scripts used for the target is shown here:

```
% tmax scripts/tmax.tcl -shell
```

In bit_coin ATPG, the power-aware ATPG features are enabled and the capture and shift power budgets are specified to generate the patterns. Below is the snippet of the ATPG coverage report and the power analysis summary report. Also you can see the power analysis summary report provides the average and peak shift and capture switching activity of the patterns that are generated.

```
#Coverage and Pattern count numbers
report_summaries
 Uncollapsed Stuck Fault Summary Report
 ----------------------------------------------
 fault class                     code   #faults
 ------------------------------  ----  ---------
 Detected                         DT    1650143
 Possibly detected                PT      18447
 Undetectable                     UD      55214
 ATPG untestable                  AU     177204
 Not detected                     ND        938
 ----------------------------------------------
 total faults                            1901946
 test coverage                             89.85%
 ----------------------------------------------
           Pattern Summary Report
 ----------------------------------------------
 #internal patterns                        110
     #basic_scan patterns                  110
 ----------------------------------------------
```

```
#Power report
report_power
 ---------------------------------------------------------------------------
-------
                                Power Analysis Summary
 ---------------------------------------------------------------------------
-------
      Number of Scan Cells                      79226
      Number of Patterns                        0-109
      Cycles Per Load                           3463
      Average Shift Switching                   10571.71
      Average Capture Switching                 24401.18
      Peak Shift Switching                      33524 (pattern: 20
cycle: 3460)
      Peak Capture Switching                    58891 (pattern: 82)
```

Figure 8.10 – bit_coin ATPG Report

The coverage is slightly on the lower side for this first version of the bit_coin design. However, there is scope for improvement as the number of AU faults is on the higher side.

Chapter 9, VC LP @Netlist

Overview

After Gate Level Implementation, VC LP is run to ensure the Gates + UPF' are complete and correct. This is shown in Figure 9.1.

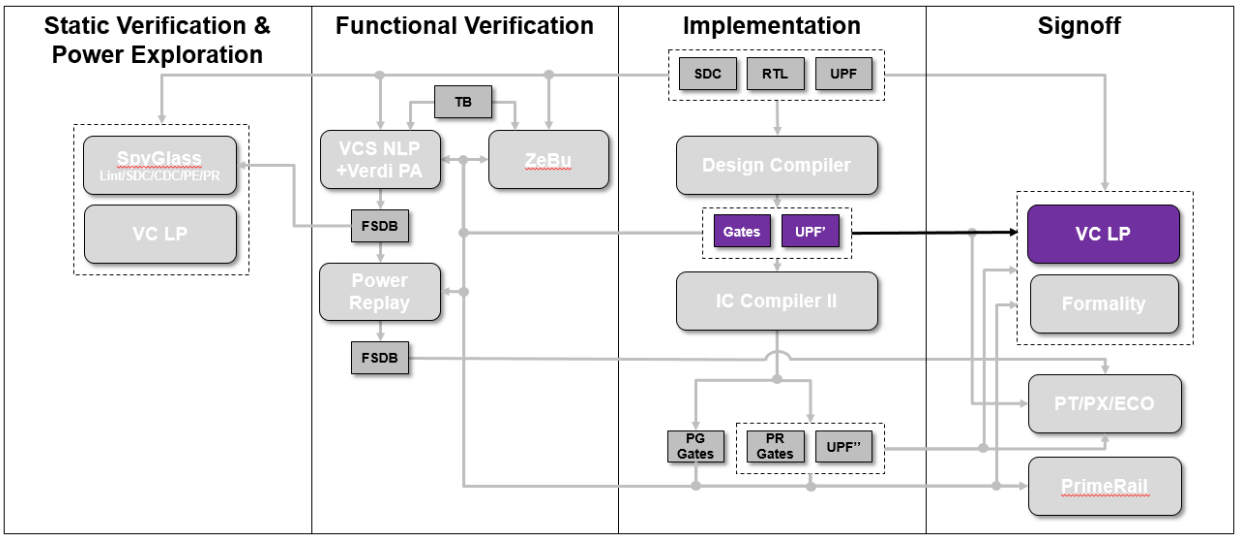


Figure 9.1 - VC LP@Netlist in the Synopsys Low Power Flow and Methodology

Recall from Chapter 5, the flow for VC LP, shown again in Figure 9.2.

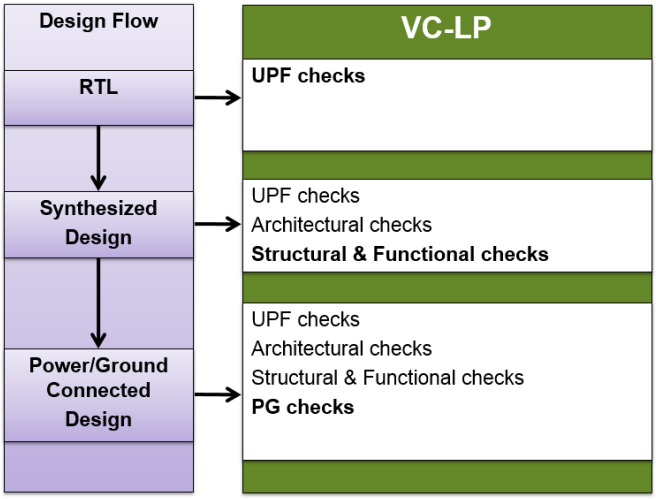


Figure 9.2 - VC LP in the Design Flow

In this section, we'll focus on the "Synthesized Design", where we will take the Gate Level Netlist and UPF' from Design Compiler and perform UPF and other checks.

Scripts & Mechanics

The scripts and mechanics are very much like the RTL Stage in Chapter 5, but here we simply change the design being read (netlist instead of RTL), the UPF (UPF' instead of RTL UPF), and add another check_lp command (-stage design). This is shown in Figure 9.3.

```
# File : go_vclp_bit_coin_gate.tcl
# VC-LP Script @Netlist
source ./scripts/synopsys_vcst.setup

read_verilog -netlist ../icc2/outputs2icc2/bit_slice.vg
read_verilog -netlist ../icc2/outputs2icc2/bit_top.vg
read_verilog -netlist ../icc2/outputs2icc2/bit_coin.vg
current_design $DESIGN_TOP
link

load_upf ../icc2/outputs2icc2/wrap_bit_coin.upf

check_lp -stage upf
check_lp -stage design

report_lp -list

save_session
```

Figure 9.3 - VC LP @Netlist

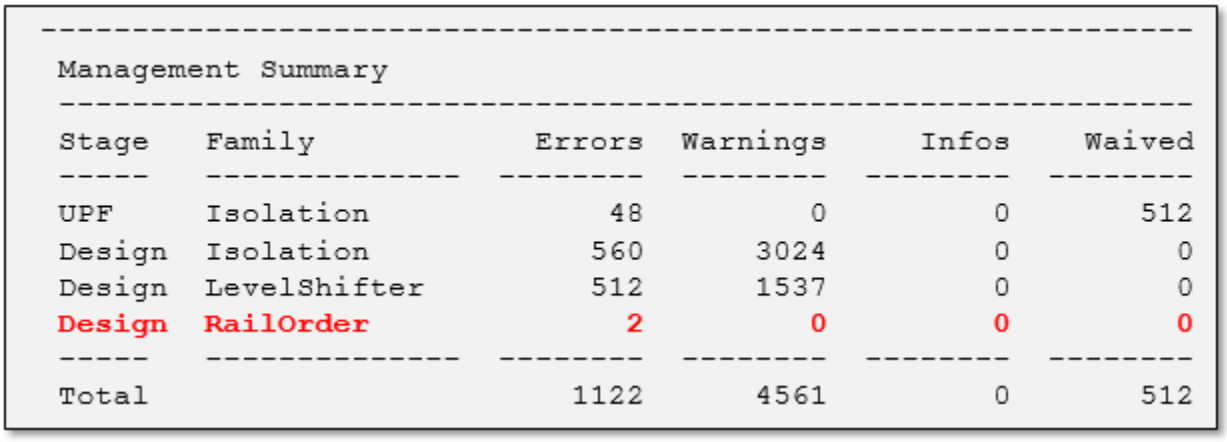
Bitcoin Experience

To run the VC LP @Netlist, you can use the following commands:

```
% cd bitcoin_v1.1/tools/vclp
% make bit_coin_gate
```

You can view the detailed scripts in the "scripts" directory and the results reports and logs in the "reports" and "logs" directories.

In this case, we see the following results, shown in Figure 9.4.



Management Summary

Stage	Family	Errors	Warnings	Infos	Waived
UPF	Isolation	48	0	0	512
Design	Isolation	560	3024	0	0
Design	LevelShifter	512	1537	0	0
Design	RailOrder	2	0	0	0
Total		1122	4561	0	512

Figure 9.4 - Results for VC LP @Netlist

We'll cover a few of these messages in this section. To review these issues, simply restore the session and launch the debug GUI. You can use the Makefile and "start_gui" command to accomplish this.

```
% make debug_bit_coin_gate
vc_static_shell> start_gui
```

For the "RailOrder" errors (2 of them), refer to Figure 9.5. You'll see that a regular buffer got inserted between the "isolation_signals" input port and the enable pin of an ISOLATION CELL, then resulting in the RAIL_BUFINV_STATE error message, which says that a buffer/inverter instance has a supply off, but the sink (ISO CELL) has supply on.

• RAIL_BUFINV_STATE (2 Design Stage Errors)

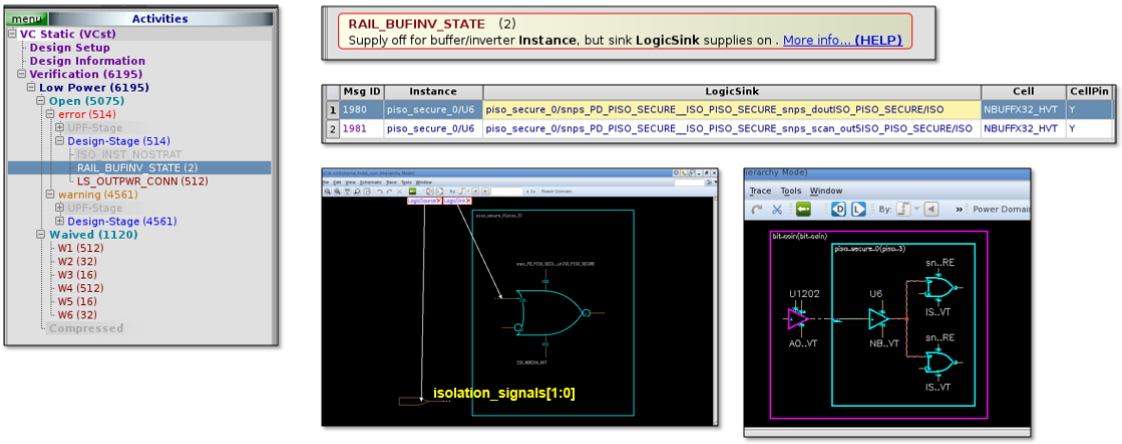


Figure 9.5 - RAIL_BUFINV_STATE Error

Notice, we also have 512 “LS_OUTPWR_CONN” Errors, shown in Figure 9.6.

• LS_OUTPWR_CONN (512 Design Stage Errors)

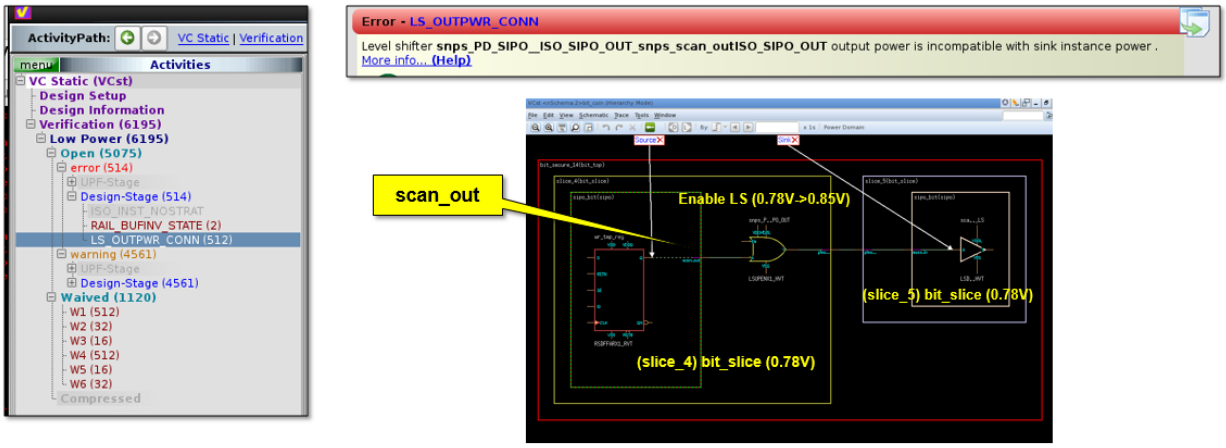


Figure 9.6 - LS_OUTPWR_CONN Error

Here we see that an Enabled-Level Shifter was added to an instance of bit_slice to its “scan_out” port, because it’s top level port is connected to the bit_top level, which is at the higher voltage. However, when the design is put together, the scan_out port is connected to another bit_slice instance, and therefore the Level Shifter is not needed, and hence the error. This is because in synthesis, the order in which DFT and mv_cell insertion was done should be reviewed.

There are other errors present in the design, and it’s a good idea for the user to review them. Many of the remaining issues are due to heterogeneous fanout and are essentially false negatives. But, we’ll leave it to the user to investigate those further.

Chapter 10, Formality @RTL2Netlist

Overview

Formality is Synopsys' logic equivalency checker. As shown in Figure 10.1, you can see the inputs for Formality are RTL+UPF and Gates + UPF'. Additionally, an SVF (Synopsys Verification Format) file from Design Compiler is also input to Formality.

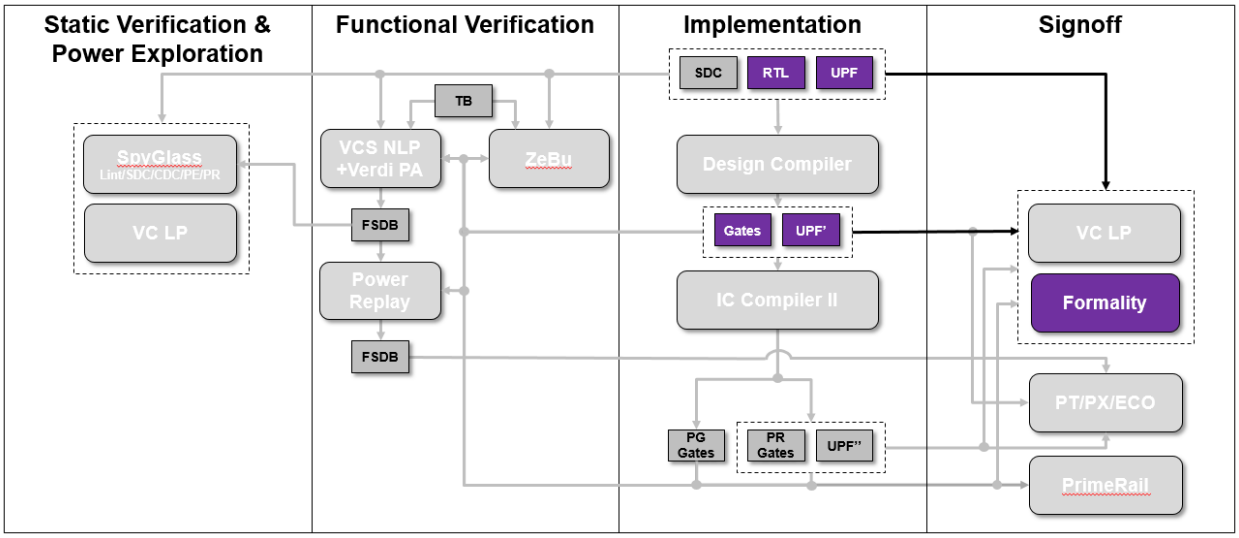


Figure 10.1 - Formality in the Synopsys Low Power Flow and Methodology

Formality has several features of importance:

- Engineered compatibility with Design Compiler
  - Verifies all default optimizations
  - Supports advanced datapath optimizations
  - Added logic like scan must be disabled with set_constant command
- Automatic SVF guidance
  - Reduces setup
  - Eliminates unnecessary verification iterations
  - Guidance is generated by Design Compiler and proven in Formality, or it will not be used
- comprehensive low power equivalence checks
  - Verifies the design in all power states as defined in the power state table
  - Supports advanced low-power design techniques such as clock gating, power gating state retention
  - Supports special low-power cells such as isolation cells and level shifters
- Manual ECO implementation with Formality Ultra

Formality performs RTL-to-gates and gates-to-gates equivalency checks over various stages of the design flow. This is shown in Figure 10.2.

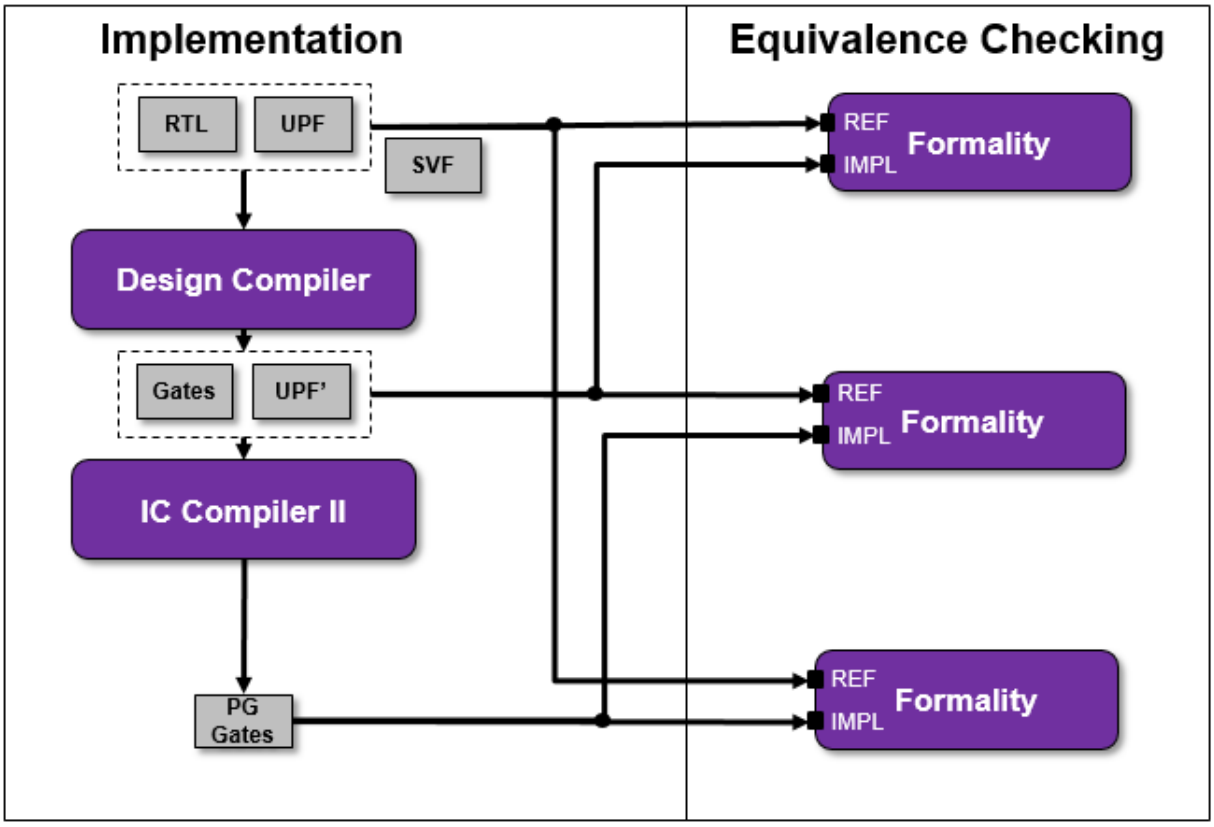


Figure 10.2 - Formality in the Design Flow

Formality has the following low power capabilities shown in Figure 10.2.

- Formality comprehensive low power equivalence checks
  - Verifies the design in all power states as defined in the power state table
  - Supports advanced low-power design techniques such as clock gating, power gating state retention
  - Supports special low-power cells such as isolation cells and level shifters

Formality performs the following low power functional equivalency checks and this is how it compares with VC LP. This is shown in Figure 10.3.

	Formality	VC-LP
Verifies functional implementation of LP intent specified in UPF matches Reference	✓	
Detects functional difference in all power states	✓	
Detects errors in UPF	✓	✓
Equivalent power-connectivity in a PG netlist and corresponding UPF file	✓	✓
Island ordering checks	✷	✓
Incorrect power switches	✷	✓
Missing/Incorrect isolation cells and level shifters	✷	✓
Missing/Incorrect Enabled Level Shifters (ELS)	✷	✓
Incorrect power connectivity of ELS	✷	✓
Redundant isolation cells		✓
Redundant level shifters		✓

Figure 10.3 - Formality vs VC LP comparison

Running Formality is straightforward and involves reading in both the reference and implementation designs, libraries, and UPF.

```
# Example Formality Script
read_db {low_power_library.db special_lp_cells.db}
read_verilog -r { top.v block1.v block2.v block3.v }
set_top r:/WORK/top
load_upf -r top.upf
read_verilog -i { post_dc_netlist.v }
set_top i:/WORK/top
load_upf -i top_post_dc.upf
```

In a low power flow, there are two different options:

1.Turn on all supplies and run ALL-On mode
2. Run in ALL-State mode

- all-supplies-on
  - **set verification_force_upf_supplies_on true**
    - 2010.03 default was false: “all-power-states”
    - 2011.09 default is now true: “all-supplies-on”
- all-power-states

It’s recommend to first run in ALL-ON mode. Then, if that passes, verify in ALL-State mode.

Below is a list of items which ALL-State can catch which ALL-ON cannot:

- *Incorrect supply connections*
- *Defective Design + UPF*
- *Power state transition failure*
- *Incorrect buffer in Shutdown PD*

Figure 10.4 below shows an incorrect buffer insertion which ALL-State would catch while All-ON would not.

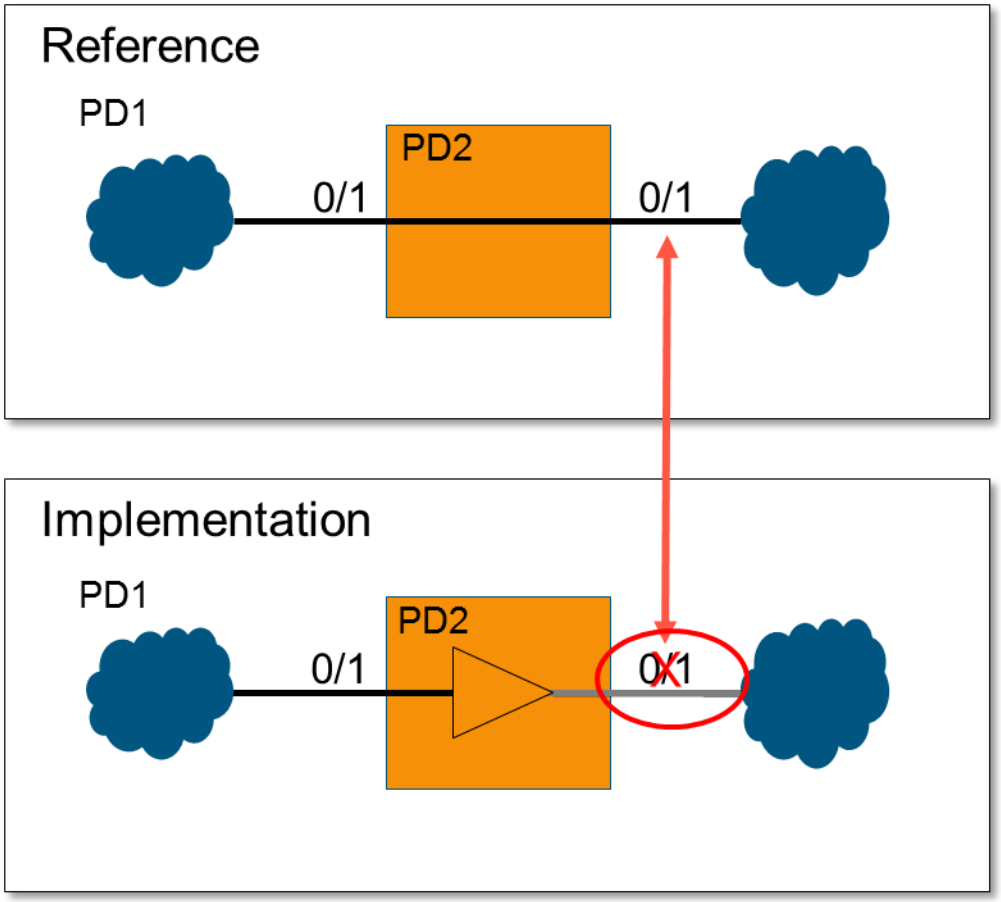


Figure 10.4 - Errors caught with ALL-State mode

In summary, Formality can catch functional differences between two designs in the low power UPF flow. It recommended to run a two-pass flow where the first pass is run in ALL-ON mode and the second pass is All-State mode. This will ensure all power state transitions functional differences are detected.

Scripts & Mechanics

The Formality scripting is very simple. To start, you should have a setup file, like .synopsys_dc.setup which includes your search_path, link_libraries, and other variables. For our Reference Design we've created a setup file called synopsys_fm.setup shown in fig 10.5.

```
set search_path .
lappend search_path ../../rtl
lappend search_path ../../verification
lappend search_path ../../lib
lappend search_path ../../lib/sram/lib_dbs
lappend search_path ../../lib/stdcell_hvt/db_nldm_gm
lappend search_path ../../lib/stdcell_lvt/db_nldm
lappend search_path ../../lib/stdcell_rvt/db_nldm

set hvt_libs " \
saed32hvt_tt0p78v125c.db \
saed32hvt_tt0p85v125c.db \
saed32hvt_tt1p05v125c.db \
saed32hvt_ulvl_tt0p78v125c_i0p78v.db \
saed32hvt_ulvl_tt0p85v125c_i0p85v.db \
saed32hvt_ulvl_tt1p05v125c_i0p78v.db \
saed32hvt_dlvl_tt0p78v125c_i0p78v.db \
saed32hvt_dlvl_tt0p85v125c_i0p85v.db \
saed32hvt_pg_tt0p78v125c.db \
saed32hvt_pg_tt0p85v125c.db \
saed32hvt_pg_tt1p05v125c.db \
"
```

Figure 10.5 - Formality Setup File

Now, you can build a basic script to analyze/elaborate your design, load your UPF, and then perform checks and reports, and finally save a session (as shown in Figure 10.6).

```
# SETUP FILE
source ./scripts/synopsys_fm.setup

# LIBRARIES
read_db -tech $link_library

# FM RETENTION REGISTER MODELS
lappend search_path ../../verification
remove_design -shared i:/*/R*DFF*

# READ REFERENCE (RTL)
read_verilog -r -f ../../rtl/files_${DESIGN_TOP}.f
set_top r:/WORK/${DESIGN_TOP}
load_upf ../../upf/snps_${DESIGN_TOP}.upf

# READ IMPLEMENTATION (DC Gates)
read_verilog -i -tech -libname TECH_RET_IMP [list ../../verification/FM/FM_TECH_RET_MODELS.v]
read_verilog -i ../dc/results_bit_slice/${DESIGN_TOP}.vg
set_top i:/WORK/${DESIGN_TOP}
load_upf ../dc/results_bit_slice/${DESIGN_TOP}.upf

# CONSTANTS
set_constant $impl/scan_enable 0

# MATCH/VERIFY/DEBUG
match

verify
save_session -replace ./fm_sess/fm_sess_${DESIGN_TOP}_rtl2gate.fss

redirect -file ${REPORTS_DIR}/${DESIGN_TOP}.fm.rtl2gate.rpt {report_status}
```

Figure 10.6 - Formality basic script

Next, you can launch Formality to perform your bit_slice verification as shown in figure 10.7.

```
fm_shell -f scripts/go_fm_bit_slice_rtl2gate.tcl | tee logs/log.fm.bit_slice_rtl2gate
```

Figure 10.7 - Formality bit_slice launch

Next, you can launch Formality to perform your bit_coin verification as shown in figure 10.8.

```
fm_shell -f scripts/go_fm_bit_coin_rtl2gate.tcl | tee logs/log.fm.bit_coin_rtl2gate
```

Figure 10.8 - Formality bit_coin launch

If you'd like to restore a saved session, simply re-launch Formality and use the restore_session command, as shown in Figure 10.9.

```
fm_shell (setup)> restore_session fm_sess/fm_sess_bit_coin_rtl2gate.fss
Loading session file '/path/gokcek/bitcoin_v1.5/tools/fm/fm_sess/fm_sess_bit_coin_rtl2gate.fss'
Info:  File created with Formality version 'M-2016.12-SP1-Dev-4356290 Jan 11 2017'.
Cleared container 'i'
Cleared container 'r'
Removed shared technology library 'GTECH'
Loading db file '/global/apps5/fm_2016.12-SP1/libraries/syn/gtech.db'
Warning: Unable to restore variable 'synopsys_auto_setup_filter' (FM-142)
Current container set to 'i'
Reference design set to 'r:/WORK/bit_coin'
Implementation design set to 'i:/WORK/bit_coin'
Session restored
Last verification status
   Reference design:   r:/WORK/bit_coin
   Implementation design:  i:/WORK/bit_coin
   Verification SUCCEEDED
   Diagnosis not run
1
fm_shell (verify)>
```

Figure 10.9 - Formality Restore Session

From there, you can launch the GUI via the "start_gui" command to debug your design. For more detailed information on Formality, please consult the Formality User Guide, found on SolvNet.

Bitcoin Experience

To run Bitcoin with Formality, first make sure your Formality software and licenses are configured and installed. Then, simply type the following commands:

```
% cd bitcoin_v1.1/tools/fm
% make bit_slice_rtl2gate
% make bit_coin_rtl2gate
```

This Makefile target will execute the following command, and you can peruse the script and logs to learn more.

```
% fm_shell -f scripts/go_fm_bit_slice_rtl2gate.tcl | tee
logs/log.fm.bit_slice_rtl2gate

% fm_shell -f scripts/go_fm_bit_coin_rtl2gate.tcl | tee
logs/log.fm.bit_coin_rtl2gate
```

Running Bitcoin through Formality at the RTL to gates Verification stage was clean, fast, and efficient, noted in Figure 10.10.

Design	Failures	Errors	Total Time (sec)	CPU Time (sec)	Peak Memory (MB)
bit_slice	0	0	22	12	472
bit_coin	0	0	4757 (1.32 Hrs)	4430	5373

Figure 10.10 - Formality bit_slice and bit_coin Results

Formality and DC treat the following cases differently. Formality issues an error, while DC issues a warning. Links to SolvNet articles are included for the messages below.

```
connect_supply_net: UPF supply net /VDDI is unresolved and already
driven; cannot connect supply pin /vddsw2/vout as an additional
driver. (FM_UPF-103)
```
https://solvnet.synopsys.com/retrieve/1843472.html

```
(FM_UPF-124) error message is issued when the hierarchical block
ports have a UPF driver supply attribute that is different from the
actual driver supply net seen in the top-level design context.
```
https://solvnet.synopsys.com/retrieve/2468948.html

To get past these issues, the following variables are used in the Formality runs:

```
set upf_warn_on_failed_parallel_resolved_check true
set upf_warn_on_failed_port_attribute_check true
```

Note for both bit_slice and bit_coin RTL2Gate runs, Formality successfully verified both cases, as shown in Figures 10.11 and 10.12.

```
************************************ Verification Results ************************************
Verification SUCCEEDED
   ATTENTION: Top-level/black_box ports were assumed to be related to the primary supplies.
              These results are dependent on this assumption being independently verified.
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
----------------------------------------------------------------------------------------------

 Reference design: r:/WORK/bit_slice
 Implementation design: i:/WORK/bit_slice
 239 Passing compare points
----------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut    Port   PDCut     DFF     LAT   TOTAL
----------------------------------------------------------------------------------------------
Passing (equivalent)          54       0       4       3       6      22     150       0     239
Failing (not equivalent)       0       0       0       0       0       0       0       0       0
Not Compared
  Don't verify                 0       0       0       0       5       0       0       0       5
**********************************************************************************************
```

Figure 10.11 - Formality bit_slice Results

```
************************************ Verification Results ************************************
Verification SUCCEEDED
   ATTENTION: Top-level/black_box ports were assumed to be related to the primary supplies.
              These results are dependent on this assumption being independently verified.
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
----------------------------------------------------------------------------------------------

 Reference design: r:/WORK/bit_coin
 Implementation design: i:/WORK/bit_coin
 120520 Passing compare points
----------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut    Port   PDCut     DFF     LAT   TOTAL
----------------------------------------------------------------------------------------------
Passing (equivalent)       27648       0    1026    1553     121   10946   79226       0  120520
Failing (not equivalent)       0       0       0       0       0       0       0       0       0
Not Compared
  Don't verify                 0       0       0       0       4       0       0       0       4
**********************************************************************************************
```

Figure 10.12 - Formality bit_coin Results

Chapter 11, VCS NLP + Verdi Power Aware @Netlist

Overview

VCS NLP (Native Low Power) is used at GLS stage to dynamically verify the design in the presence of UPF (power intent). Refer to figure 11.1 below, inputs to this flow are GLS (netlist out of DC), UPF' and testbench (commonly written in UVM). For debug we use Verdi Power-Aware since the user needs the ability to visualize the power instrumented signals along with knowing the state of supply rails and power domains. The output from the tool to capture the switching activity for power analysis can be in the form of FSDB, SAIF or even VCD, in this flow we used FSDB.

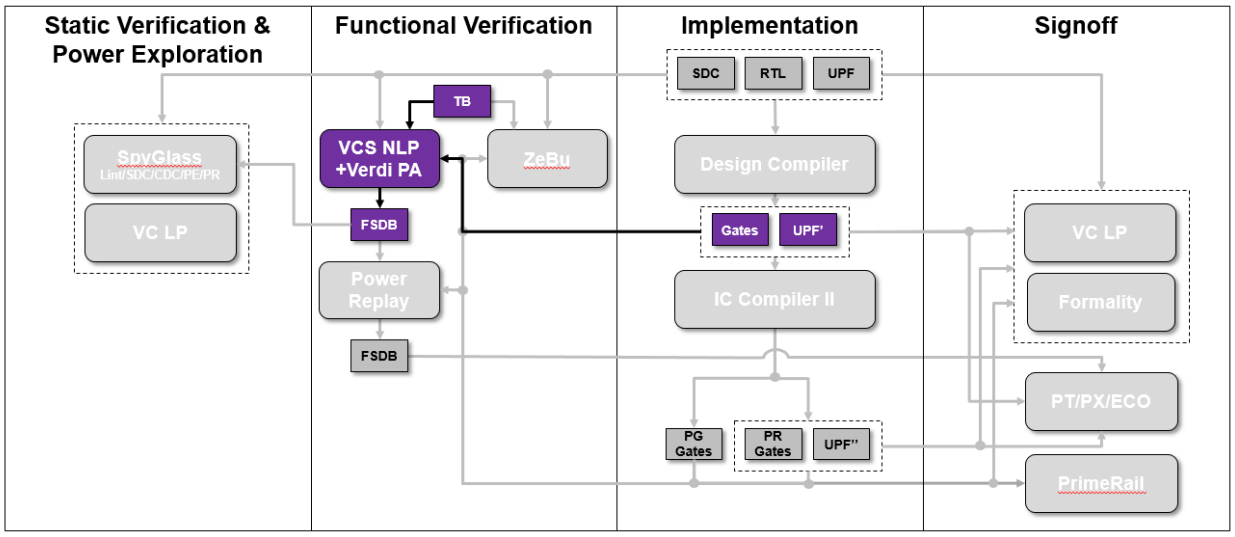


Figure 11.1 - VCS NLP@GLS in the Synopsys Low Power Flow and Methodology

As mentioned in chapter 6 overview for RTL flow, likewise GLS flow for VCS NLP in contrast to verifying the power intent statically the primary focus here should be to validate the same UPF dynamically - power and related control signal sequencing and effects of that. Figure 11.2 illustrates this point.

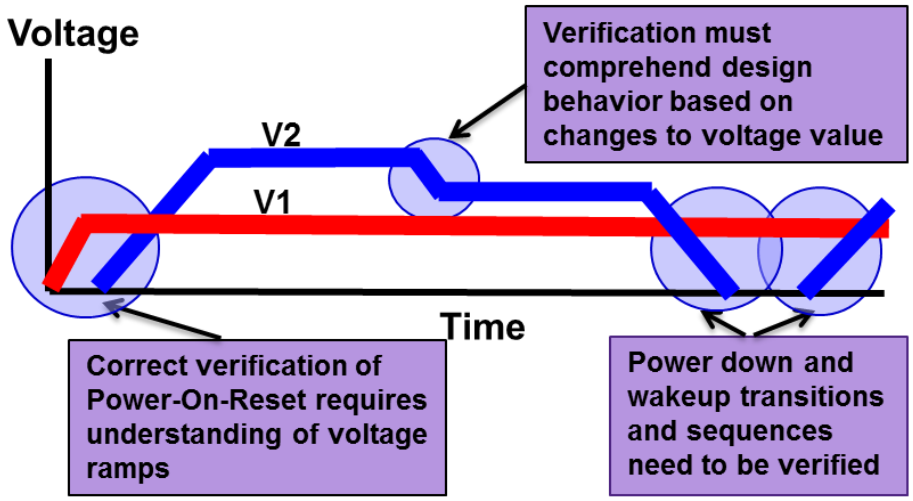


Figure 11.2 - VCS NLP@GLS Dynamic Low Power Verification

Scripts & Mechanics

At GLS stage like RTL the flow is very similar, you will need these switches -upf <upf_file> -power_top <design/UPF top module>. UPF file in this case comes from DC output (UPF'), similar to the design (netlist) which is also an output from DC.

Since it is netlist you also need to point to the standard cell Verilog libraries. For NLP simulations, you have two choices (assuming you have both options available) to use power-aware Verilog libraries or non-power aware libraries. The recommendation is to use power-aware Verilog libraries when available. The primary reason for this is that Verilog modelling (not limited to just ports) inherently allows you to model the power intent more accurately when compared with NLP port based corruption (which is based on DB's).

Some other key difference to note is that as opposed to running VCS NLP in protected mode (default mode) you will run VCS NLP in accurate mode -power=accurate. Secondly -power_config <config_file> will have a much larger list of DB's being loaded for all the standard cells & low power cells in the design including any the macro's.

Lastly xprop which solves the X-optimism issue at RTL is not applicable here so there is no need to pass this switch. For debug if the intention is to use this FSDB for power analysis also the most common options remain the same as before -debug_access+pp -debug_region=lib+cell

Figure 11.3 shows the screenshot of a typical VCS compile command, note some additional options are being passed/shown below since they help in compile time turn-around for incremental compiles.

```
vcs  -partcomp -fastpartcomp=j4 -timescale=1ns/1ps -sverilog -l vcs.log +warn=noIWNF -ntb_opts uvm-1.2 +define+UV
M_POST_VERSION_1_1 +incdir+../../verification/include+../../verification/src+../../verification/env+../../verific
ation/tests+../../verification/hdl  +vcs+initreg+random -power=pst_state_msg+dump_hvp  -power_top bit_coin  +defi
ne+UPF +warn=noUPF_CONYI,noIWNF,noTFIPC -upf ./scripts/wrap_bit_coin_gls.upf -mvrpt=logs/GLS_DEBUG_UPF/mvsim_nati
ve_reports -power=accurate -power_config ./scripts/config_gls.tcl  -debug_region=lib+cell -debug_access+pp -lca -
kdb +define+DUMP \
                ../../verification/tests/tb_top.sv ../../verification/hdl/bitcoin_top.sv \
               -F ../../verification/hdl/filelist_gls.f +define+NO_SAIF +nospecify -Mdir=output/GLS_DEBUG_UPF/cs
rc -o output/GLS_DEBUG_UPF/simv \
               -l logs/GLS_DEBUG_UPF/compile.log \
               -partcomp_dir=output/GLS_DEBUG_UPF/partitionlib
```

Figure 11.3 – VCS Compile Command

For running the simulation since we use UVM, you must pass +UVM_TESTNAME=<test>. Figure 11.4 shows the screenshot of a typical VCS runtime command again it shows some additional options to control the location of FSDB file, log file and control the number of transactions.

```
./output/GLS_DEBUG_UPF/simv +min_random_count=50 +max_random_count=50 +vcs+initreg+0 -power ./scripts/lp.tcl  +fs
dbfile+./logs/GLS_DEBUG_UPF/dump.fsdb +SAIF_REPORT=./logs/GLS_DEBUG_UPF/bitcoin.saif -l logs/GLS_DEBUG_UPF/run.lo
g \
       +ntb_random_seed=1 +timeout=200000 +UVM_TESTNAME=btcoin_lp_test +fsdb+all
```

Figure 11.4 - VCS Runtime Command

Bitcoin Experience

Assuming the suggested tools are installed and licenses available. To run Bitcoin with VCS NLP along with Verdi Power-Aware at GLS (compare this section with chapter 6, key difference is use of PHASE=GLS)

```
% cd bitcoin_v1.1/tools/vcs_nlp
% make PHASE=GLS
```

The default 'make' target is to compile & run GLS simulation with UPF and dump FSDB. The below shows various arguments (most common) that you can pass to 'make' based on the intended flow.

Note: The make flow below shows compile+run together as part of the default target, you can split the flow into steps to compile and run separately (explained later).

5. To compile & run without dumping

   ```
   % make MODE=NODEBUG PHASE=GLS
   ```

6. To compile & run without UPF and without dumping

   ```
   % make MODE=NODEBUG LP=NOUPF PHASE=GLS
   ```

7. To compile & run with a different UPF but still with dumping

   ```
   % make UPF_FILE=<path to the upf file> PHASE=GLS
   ```

8. Default UVM transactions are 50, to compile & run it for longer or shorter period

   ```
   % make MAX_TRANS_CNT=<integer value> PHASE=GLS
   ```

The target for only compiling is

```
% make comp PHASE=GLS
```

This target can be used in combination with the above list arguments to 'make'. For example, if you wanted to only compile without DEBUG the command will be

```
% make comp MODE=NODEBUG PHASE=GLS
```

Likewise target for only running is

```
% make run PHASE=GLS
```

This target can be used in combination with the above listed arguments to 'make'. Just be aware though that some options are only applicable at compile not at runtime. For example, if you wanted to run the test longer (more transaction) then you could do this,

```
% make run MAX_TRANS_CNT=100 PHASE=GLS
```

But you can't pick a different UPF file at runtime this can only be done at compile.

Also, important to note, to keep the data unique, each flow (with or without UPF, with or without debug …) is written to its own unique directory. The directory structure that is followed is explained below,

Compile outputs go under 'output' directory

- Created based on Makefile arguments PHASE, MODE, LP
- Examples
  - Make PHASE=GLS → simv/csrc written into output/GLS_DEBUG_UPF
  - make LP=NOUPF PHASE=GLS→ simv/csrc written into output/GLS_DEBUG_NOUPF

All log files including FSDB dump go under 'log' directory

- Same concept as compile, examples
  - Make PHASE=GLS -> compile.log, run.log, dump.fsdb written into logs/GLS_DEBUG_UPF
  - make LP=NOUPF PHASE=GLS -> compile.log, run.log, dump.fsdb written into logs/GLS_DEBUG_NOUPF

During VCS NLP compile you might see the warning message as show in figure 11.5. This warning UPF_PAC is based on multiple CSN (connect_supply_net) statements that might exist in UPF'.

NLP defaults to first CSN statement and any subsequent CSN's are flagged as warning messages. Implementation tools have a way to select the last CSN connection based on this variable setting 'mv_allow_pg_pin_reconnection'. VCS NLP support was also added recently so if you are on the latest rev you shouldn't see this message. You do need to pass the below switch to enable this functionality

-power= mv_allow_pg_pin_reconnection

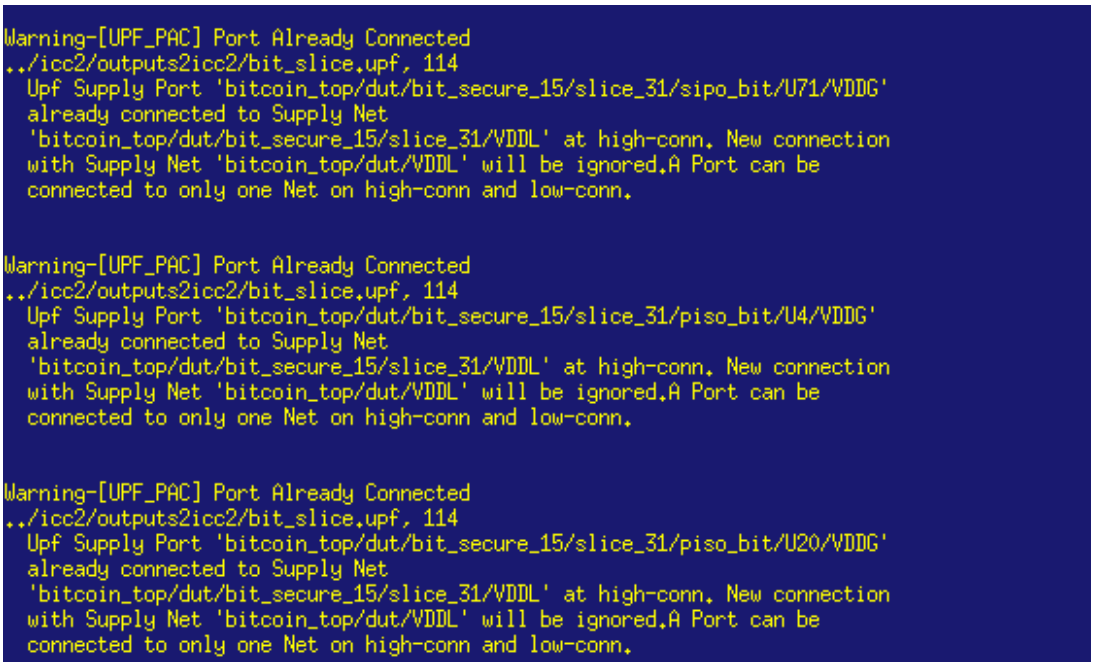


Figure 11.5 – VCS NLP Warning Messages during compile

Chapter 12, PowerReplay with PrimeTime PX

Overview

PowerReplay can be used to map RTL FSDB into Gate Level FSDB for use with PrimeTime PX for Power Estimation, as shown in Figure 12.1. By configuring PowerReplay, you can perform accelerated average power analysis by running parallel simulations or run a single simulation to extract a timing window for peak power analysis. PowerReplay helps improve the overall efficiency of FSDB generation, by improving runtime, while maintaining accuracy.

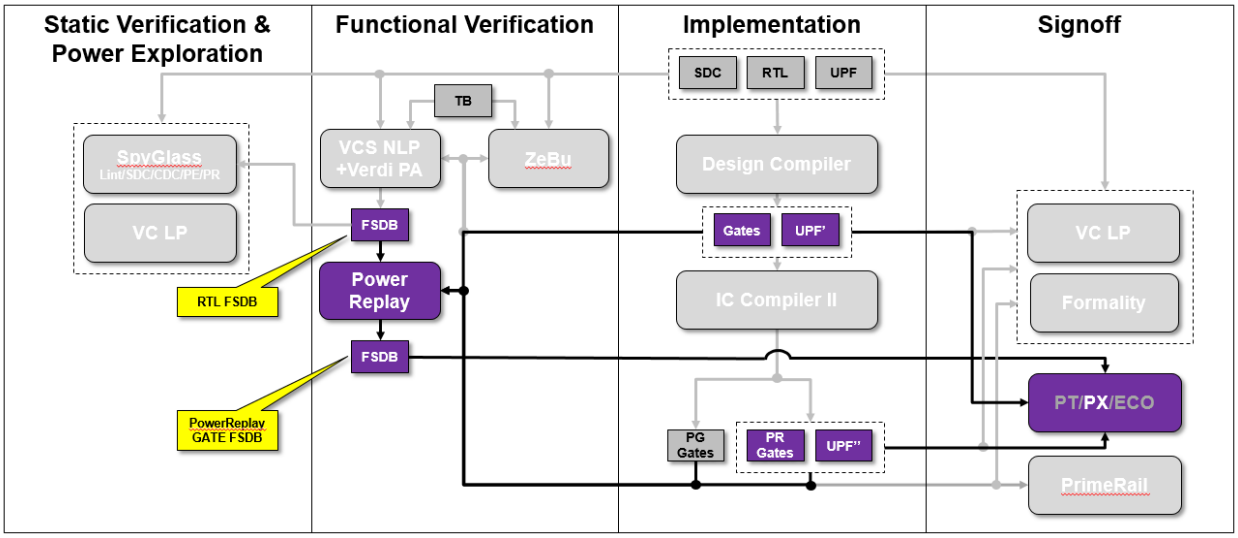


Figure 12.1 - PowerReplay in the Synopsys Low Power Flow and Methodology

In the current design methodology, where Gate Level Simulations are used to drive PrimeTime PX power calculations, these runs can take days to weeks to complete, as shown in Figure 12.2.

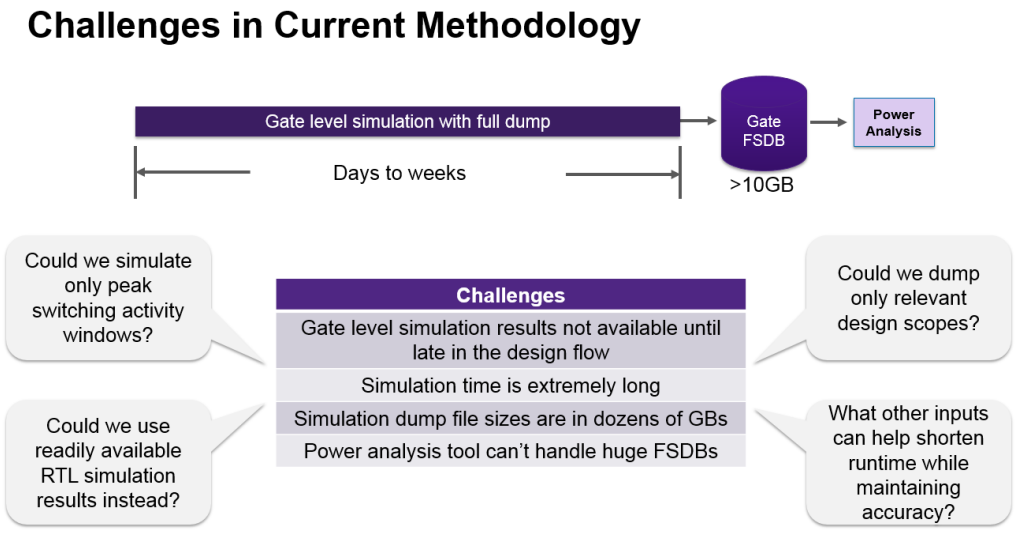


Figure 12.2 - Challenges with Gate Level Simulation

Gate Level Simulations don't provide much flexibility in simulating just peak switching activity windows or dumping only specific relevant scopes, as you must run the often-lengthy reset and configuration steps before getting to those points.

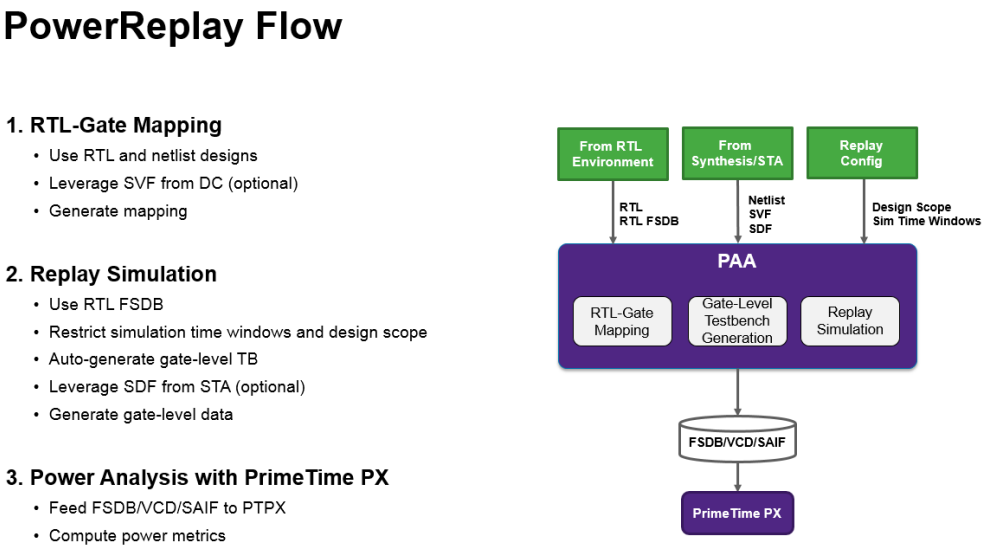


Figure 12.3 - PowerReplay Flow

PowerReplay, as shown in Figure 12.3, performs RTL-Gate mapping, and then uses the RTL FSDB, RTL, Netlist, SVF, SDF, and configurations to specify scopes and specific timing windows to replay the gate level simulations. The resulting FSDB can then be used with PrimeTime PX for Power Estimation purposes.

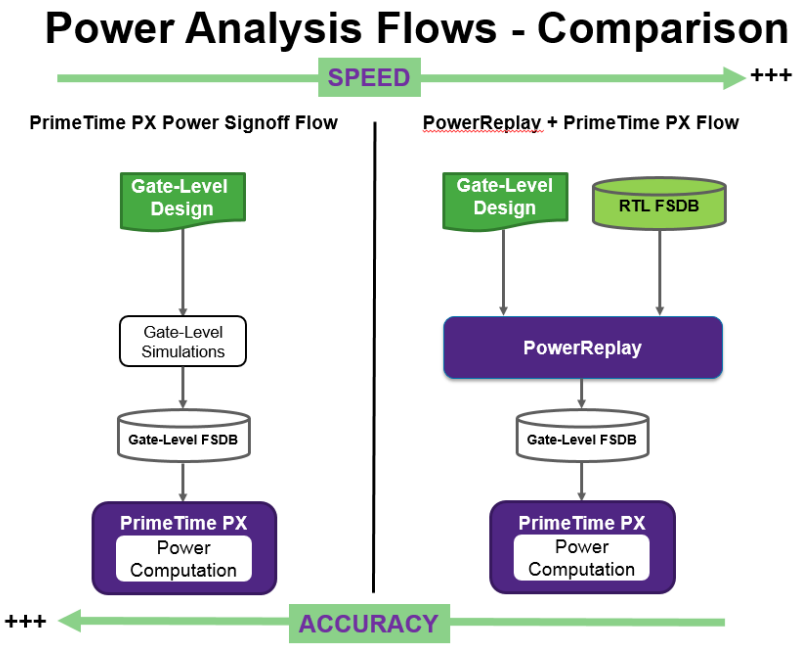


Figure 12-4. Power Analysis Flow Comparison

In Figure 12.4, we see a comparison of the various flows in which FSDBs can be read in PrimeTime PX. The most accurate, but slowest is using the Gate Level Netlist, running Gate Level Simulations, and then using Gate Level FSDB + Netlist into PrimeTime PX. While, the PowerReplay flow uses the RTL FSDB and the Gate Level Design to generate the PR Gate FSDB input to PrimeTime PX. This flow has a <5% accuracy versus the Gate Level FSDB, while having the ability to save 10X+ in runtime by using peak windows or running parallel simulations.

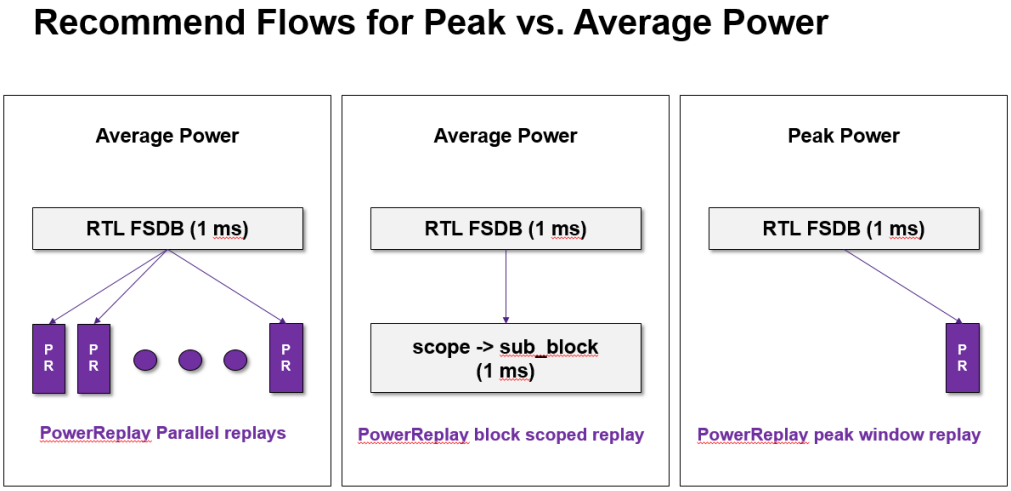


Figure 12-5. Peak vs. Average Power

In Figure 12.5, you'll see for average power, the best way to use PowerReplay for acceleration is to run parallel "what-if" simulations (replays) or to scope to a sub block. By doing this, you can realize significant time savings. For peak power, you'll leverage the RTL FSDB and configuring the timing window just to the peak timing window. And in this case, you'll save the long runtime associated with reset and configuration, and focus directly on the timing windows desired.

We can also combine the methods, as shown in Figure 12-6. Here for average power, we can use both parallel replays in addition to a sub scope. For peak power, we can run parallel replays, within a sub-scope on a specific peak timing window.

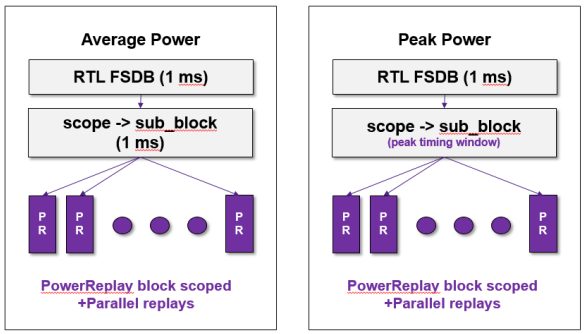


Figure 12-6. Combining parallel replays, sub scope, and timing window

Scripts & Mechanics

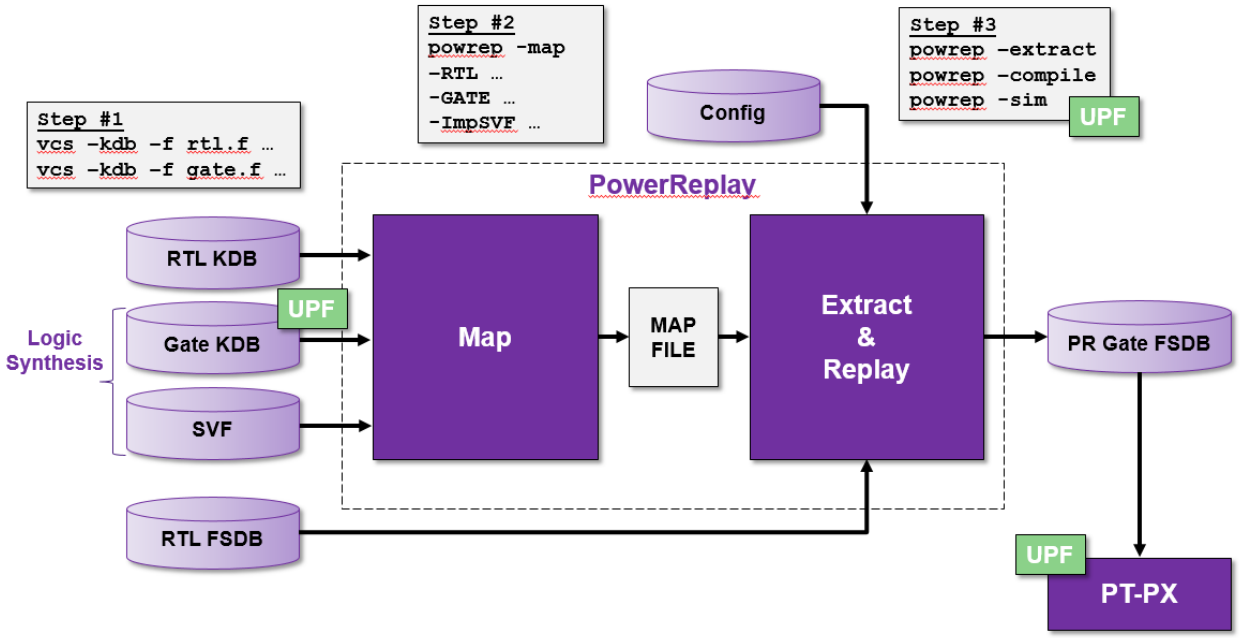


Figure 12.7 - PowerReplay Step-By-Step

Referring to Figure 12.7, there are 5 inputs to the PowerReplay and a single output.

- The inputs are: RTL KDB, Gate+UPF KDB, SVF, RTL FSDB, and Configuration
- The output is: PR Gate FSDB

There are 3 steps then to run PowerReplay.

1. Generate RTL KDB and Gate+UPF KDB
2. Run “Map” to generate mapping file
3. Run “Extract, Compile, and Sim” with UPF to generate the PowerReplay Gate FSDB

The PR Gate FSDB is then read into PrimeTime PX, along with the Gate Netlist + UPF’ (or ICC II Netlist + UPF”) for power calculation.

In Figures 12.8 and 12.9, you can review some basic script templates for PowerReplay. For more details, look at the actual scripts and Makefiles used for Bitcoin in “bitcoin_v1.1/tools/powerreplay”.

Basic PowerReplay Run – UPF

```
vcs -kdb=keep_unresolved_db -lca -timescale=1ns/1ns -sverilog \
  …
  -f ../../rtl/files_bit_coin.f \
  -o rtl_noupf_simv
```

```
vcs -kdb=keepunresolved_db -lca -timescale=1ns/1ns -sverilog \
  …
  -power_config scripts/vcs_config.tcl \
  -power_top bit_coin \
  -upf "../icc2/outputs2icc2/wrap_bit_coin.upf"
  -f scripts/files_bit_coin_icc2_gate.vg.f \
  -o icc2_gate_upf_simv
```

```
 powrep -map \
  -RTL "-lib rtl_noupf_simv.daidir/work -vtop bit_coin.vtop" \
  -GATE "-lib icc2_gate_upf_simv.daidir/work -vtop bit_coin.vtop" \
  -impSVF "../icc2/outputs2icc2/bit_slice.svf \
    ../icc2/outputs2icc2/bit_top.svf \
    ../icc2/outputs2icc2/bit_coin.svf"
```

```
powrep -extract \
  -vtop bit_coin.vtop \
  -lib icc2_gate_upf_simv.daidir/work \
  -config ./scripts/bit_coin.rtl2icc2gate_upf.wi_config \
  -replay_extract
powrep -compile -config ./scripts/bit_coin.rtl2icc2gate_upf.wi_config
powrep -sim -config ./scripts/bit_coin.rtl2icc2gate_upf.wi_config
```

```
# File : _upf.vcs_wi_compile.rc
vcs -sverilog -f wi_run.f \
  -debug_access+f+w+cbk+fwn \
  -debug_region=cell+lib \
  -power_top bit_coin \
  -power_config ../scripts/vcs_config_wisim.tcl \
  -upf "../wrap_bit_coin.upf"
```

Figure 12.8 - PowerReplay Script Templates

wisim configuration files

```
# File : _upf.wi_config
set FSDB                    = ../vcs_nlp/outputsfromvcs/RTL_UPF.fsdb
set Scope                   = bitcoin_top.dut
set Map                     = powrep_map.list.gz
set Time_Unit               = ns
set simulation_Compile_Script  = ./scripts/bit_coin.rtl2gate_upf.vcs_wi_compile.rc
set simulation_Run_Script      = ./scripts/bit_coin.rtl2gate_upf.vcs_wi_run.rc
```

```
# File : _upf.vcs_wi_compile.rc
vcs -sverilog -f wi_run.f -debug_access+f+w+cbk+fwn -debug_region=cell+lib \
  -power_top bit_coin -power_config ../scripts/vcs_config_wisim.tcl \
  -upf "../wrap_bit_coin.upf"
```

```
# File : _upf.vcs_wi_run.rc
simv +notimingcheck
```

RTL FSDB

Map File

Simulator Configuration Files

Necessary For replay

Helps speed up replay

Figure 12.9 - Replay Configuration File Example

Bitcoin Experience

To run the PowerReplay with bit_coin, you can use the following commands:

```
% cd bitcoin_v1.1/tools/powerreplay
% make bit_coin_upf
```

If you peruse the Makefile in this directory, you'll see the steps being run to generate the KDBs, run mapping, and finally run the replay to generate to the PR FSDB.

The final output is stored in the "powrep_replayLog/wi_result.fsdb" and this is passed to PrimeTime PX for power estimation.

In our tests, we saw very good correlation with the PR FSDB vs. the Gate Level FSDB, with 2.6% difference in power calculation with PrimeTime PX with UPF, as shown in Figure 12.10. Note however, this is with a relatively short simulation. With a longer simulation time, we expect the correlation to improve even more.

FSDB Used	PTPX Total Power (W)	%DIFF
GLS, NO UPF	3.6942	
PR GLS, NO UPF	3.8055	3.0%
FSDB Used	**PTPX Total Power (W)**	**%DIFF**
GLS, UPF	0.7246	
PR GLS, UPF	0.7054	2.6%

Figure 12.10 - PrimeTime PX Correlation Results

As for runtime, we looked at some longer runtimes and as you can see in Figure 12.11, the time savings results when you select a timing window and run parallel simulations. In this case, we could run many multiple parallel replays to get 7X runtime improvement (or much greater if we do more), or simply run one for peak power analysis.

FSDB Used	Simulation Time (ns)	Runtime (CPU Secs)	Runtime (CPU Hrs)	%DIFF
RTL, UPF	0 to 60000	1015.27	0.28	-
GLS, UPF	0 to 60000	12153.87	3.37	
PR GLS, UPF	30000 to 32500	1740.03	0.48	~7X

Figure 12.11 - PowerReplay vs. GLS Runtime Comparisons

Chapter 13, IC Compiler II

Overview

The Synopsys IC Compiler II (ICC II) tool provides a complete netlist-to-GDSII design solution, which combines proprietary design planning, physical synthesis, clock tree synthesis, and routing for logical and physical design implementations throughout the design flow.

IC Compiler II tool supports:

- Placement and Optimization
- Clock Tree Synthesis (CTS)
- Routing
- Extraction
- Timing Analysis
- Engineering Change Orders (ECO)
- Chip Finishing
- ASCII Output Interfaces and Reporting

Inputs for ICC II include a Verilog gate-level netlist, a detailed floorplan in Design Exchange Format (DEF), timing constraints, physical and timing libraries, and foundry-process data. It generates as output a Verilog gate-level netlist, a DEF file of placed netlist data, and timing constraints.

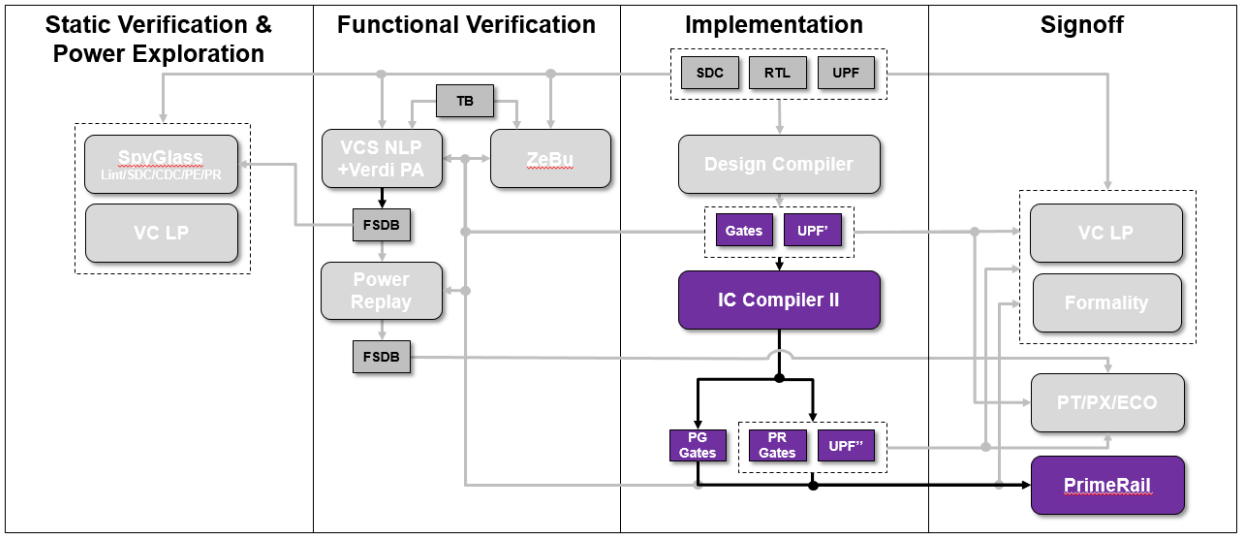


Figure 13.1 – ICC II in the Synopsys Low Power Flow and Methodology

IC Compiler II Design Planning

The IC Compiler II tool is designed for efficient design planning, placement, routing, and analysis of very large designs. You can use the design planning and chip-level analysis capabilities to perform initial floor planning and feasibility analysis.

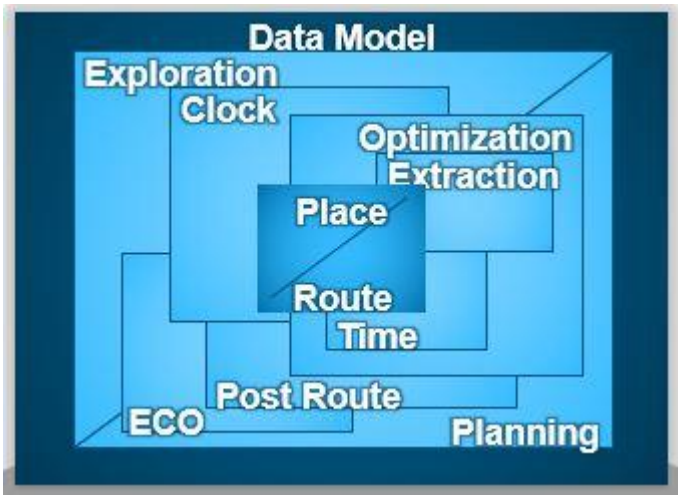


Figure 13.2 – ICC II Features

IC Compiler II Low Power

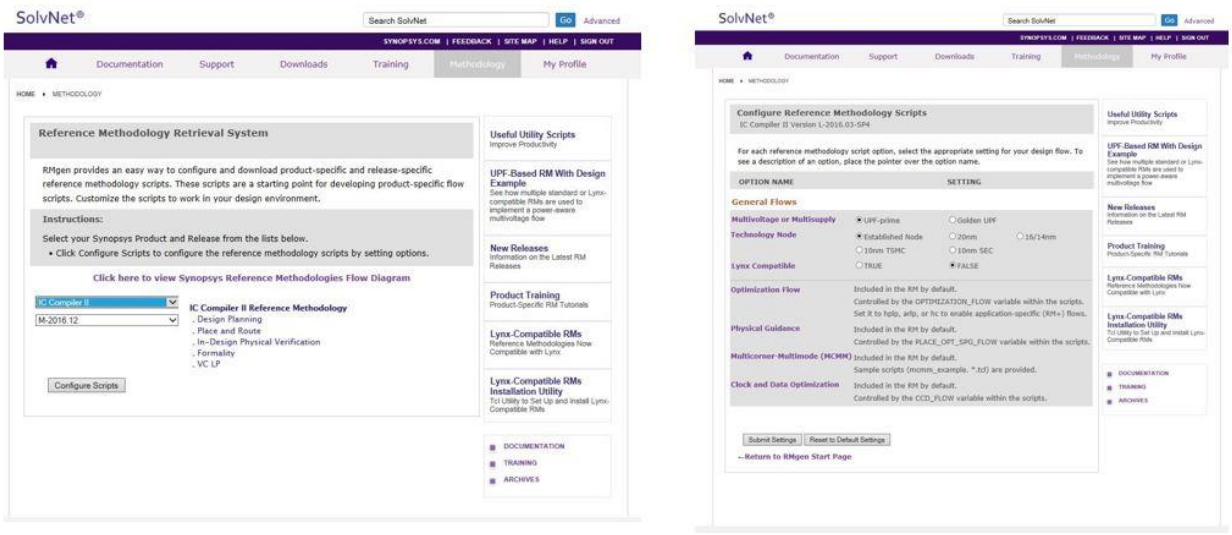
ICC II uses Unified Power Format (UPF) to specify the power intent for multi-voltage designs. Using UPF commands, you can specify the supply network, power states and power management for the design. The same set of low-power design specification commands is to be used throughout the design, analysis, verification, and implementation flow.

The UPF language provides a way to specify the power requirements of a design; however, it does not specify how those requirements are implemented. UPF specifies how to create a power supply network for each design element, the behavior of supply nets with respect to each other, and how the logic functionality is extended to support dynamic power switching to design elements. It does not contain any placement or routing information.

Scripts & Mechanics

You can download and configure the ICC II Reference flow Scripts from Solvnet using the link below:

https://solvnet.synopsys.com/rmgen



https://solvnet.synopsys.com/rmgen/

Figure 13.3 – ICC II Reference Methodology

Tool Version and Scripts used for Bitcoin implementation

```
# IC Compiler II Version Used
2016.12-SP1

# RM Script Version Used
IC Compiler II-RM_L-2016.03 with modifications

# Design Planning RM Target Scripts
bitcoin_v1.1/tools/icc2/rm_icc2_dp_scripts

# Block Implementation Scripts
bitcoin_v1.1/tools/icc2/bit_slice_scripts/
bitcoin_v1.1/tools/icc2/bit_top_scripts/
bitcoin_v1.1/tools/icc2/bit_coin_scripts/

# Other Scripts
bitcoin_v1.1/tools/icc2/scripts/
```

IC Compiler II Design Planning and Implementation Flow

Figure 13.4 shows an overall flow for IC Compiler II Design Planning and Implementation.

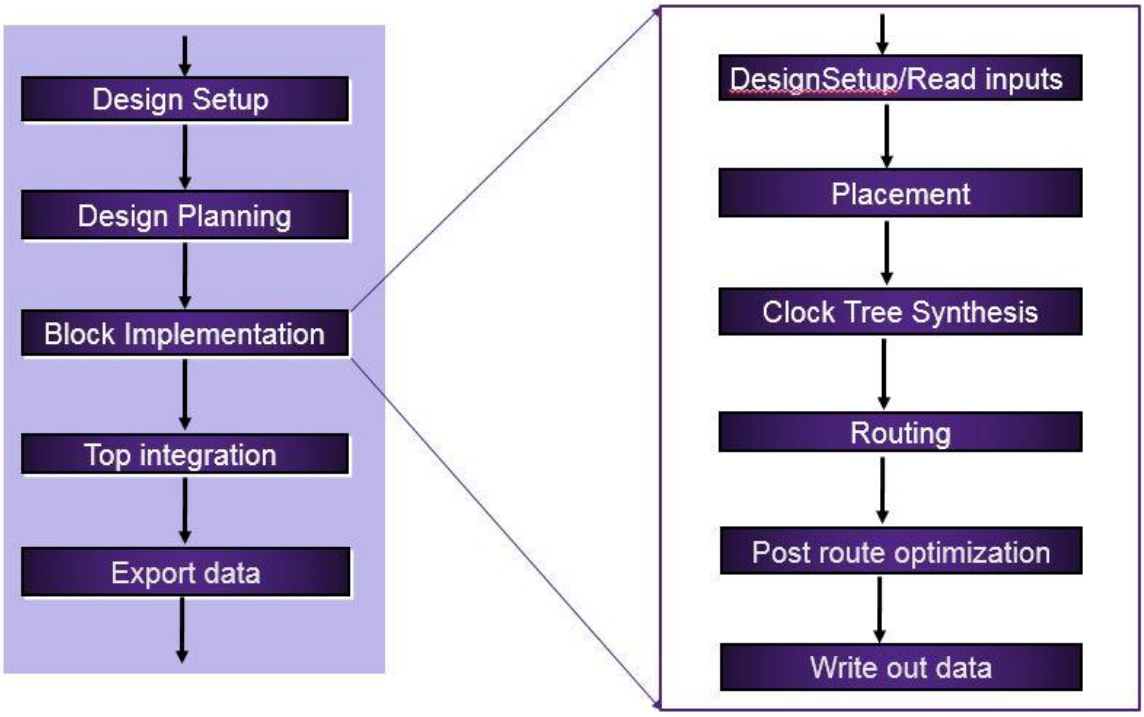


Figure 13.4 – IC Compiler II Design Planning and Implementation Flow

Running IC Compiler II for Bitcoin

To run the Bitcoin Design with ICC II you can use the following commands:

```
% cd bitcoin_v1.1/tools/icc2
% source runme
```

The “runme” file runs everything from Design Planning to block implementation, top integration and signoff. The “runme” file sources “run_dp” , “run_pr” and “run_signoff” files .

Bitcoin Experience

Bitcoin is a very small design by today's standards, but it does have its own challenges.

We decided to implement Bitcoin with multiple levels of hierarchy. We have two levels of hierarchy under the top. The top-level is bit_coin which contains bit_top. Under bit_top, we have bit_slice. bit_coin contains Multiply Instantiated Modules (MIMs), 16 instantiations of bit_top and each bit_top in turn contains 32 instantiations of bit_slice. We designed Bitcoin for low power which means we need to create voltage areas and add power switches.

ICC II takes care of the placement of isolation and level shifter cells and it will insert always-on buffers.

Below is the Bitcoin block diagram:

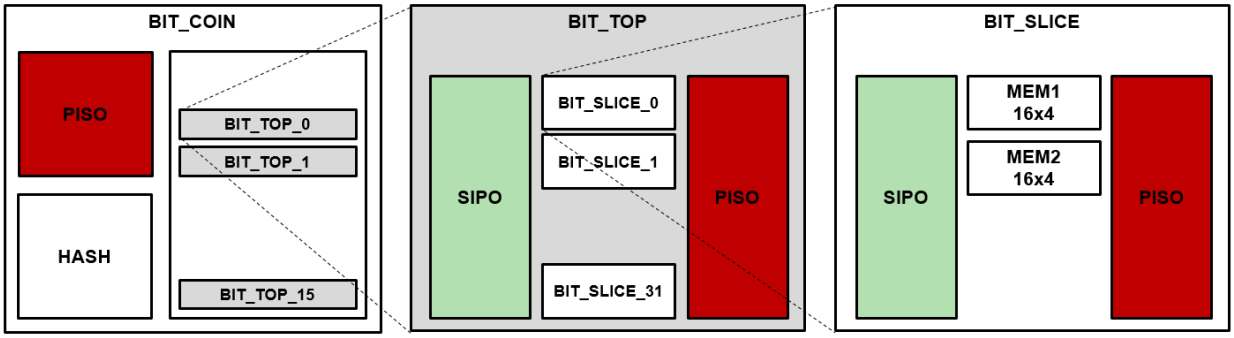


Figure 13.4 - Bitcoin Microarchitecture

Bitcoin is a multi-voltage design, with the SIPO and PISO blocks run at a lower voltage, but the memories, HASH, and top-level logic run at a higher voltage. The SIPO and PISO blocks at each level are switched. The memories in "bit_slice" have internal switched supplies. All the SIPO blocks are tied together, so all SIPOs at all levels are commonly switched. The same is true for the PISO blocks. Below is a power diagram for Bitcoin:

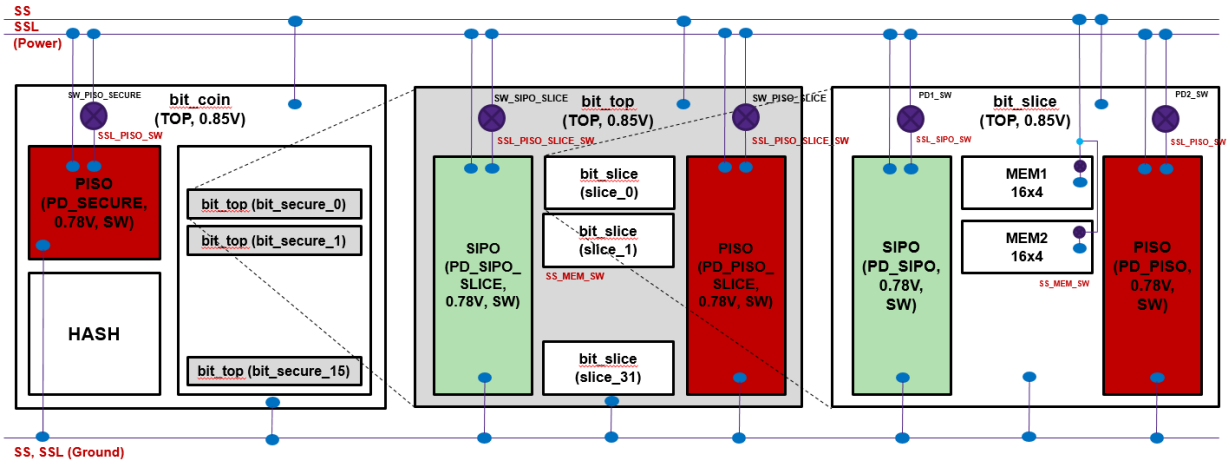


Figure 13.5 - Bitcoin Power Diagram

Bitcoin Floorplan

At the bit_coin level we placed the 16 bit_top instances in a 4x4 fashion. The 4 bit_top instances in each row are abutted together. The 2nd row is created by Flipping the 1st row along the x-axis and then abutting it with the 1st row.

Similarly, we created the 3rd and 4th rows. Space was left in the middle for top-level cells placement. Shown below is the Bitcoin Floorplan:

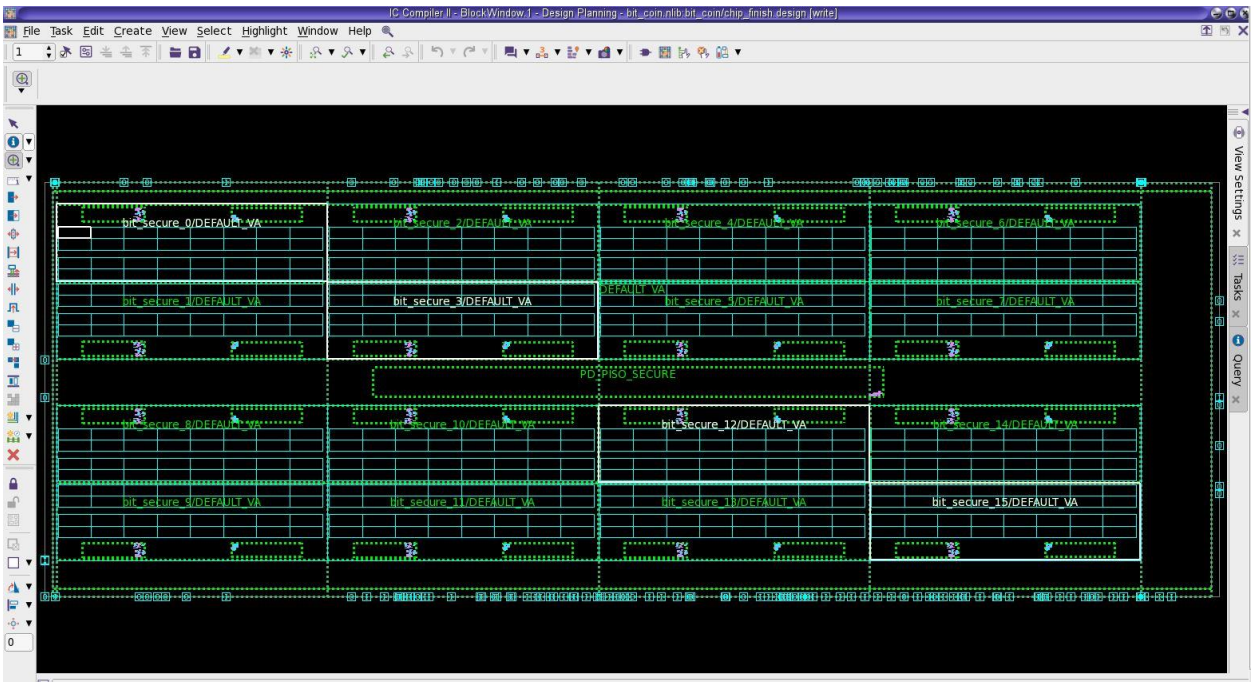


Figure 13.6 - Bitcoin Floorplan

Floorplanning for Low Power

There are three main considerations while Floorplanning for a Low Power Design:

- Voltage Area Creation
- PG Network Synthesis
- Power Switch Insertion

Voltage Area Creation

To implement a low power design with power domains, we need to create a corresponding voltage area (VA) for each power domain. A voltage area is a physical region where all the elements of the corresponding power domain will be placed. A 1:1 correspondence between the voltage area and power domain is recommended. This command shows an example of how to create a voltage area:

```
create_voltage_area -power_domain PD_PISO -region {{ 127.915
1.873} {148.057 56.643 }} -is_fixed
```

PG Network Synthesis

First off, we need to create the Power-Ground (PG) Network. The PG (power and ground) Network for bit_coin was created using the ICC II Parameter-based Power Network Synthesis (PPNS) Feature. We have three PG stripes for bit_coin. VDDH is the high voltage supply, VSS is the ground and VDDL is the switchable low voltage supply.

We divided the top-level into power regions and created the PG ring and PG mesh. PG was pushed down from the top-level to the bit_top and bit_coin levels. To make sure that all the 16x32=512 bit_slice instances are considered as MIMs for physical implementation, we need to make sure that the PG network is identical in all of the 512 bit_slice instances, so we must plan and align the PG mesh accordingly. Below is how bit_coin looks with the PG mesh inserted:

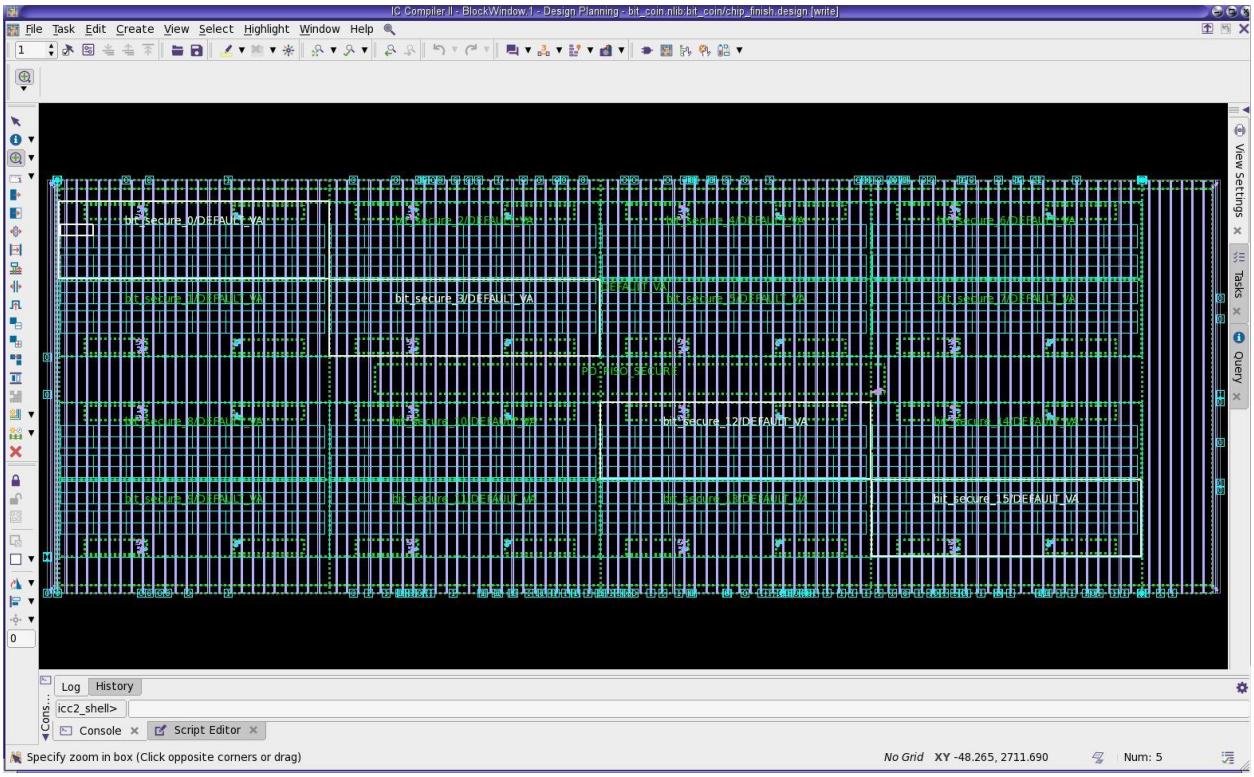


Figure 13.7 - Bitcoin Floorplan with PG Mesh

The following is an example script for the PG ring creation using PPNS:

```
create_pg_ring_pattern ring_pat -horizontal_layer @hlayer -
  horizontal_width {@hwidth} -horizontal_spacing {@hspace} -
  vertical_layer {@vlayer} -vertical_width {@vwidth} -
  vertical_spacing {@vspace} -corner_bridge @cbridge -parameters
  {hlayer hwidth hspace vlayer vwidth vspace cbridge }

 set_pg_strategy ring_strat -core -pattern {{name: ring_pat} {nets:
  {VDDH VSS VDDL}} {offset: {3 3}} {parameters: {M9 10 2 M8 10 2
  false}}} -extension {{stop: design_boundary}}

compile_pg -strategies ring_strat -ignore_drc
```

The following is an example of PG mesh creation using PPNS:

```
set pp_regions {pp_region1 pp_region2 pp_region3 pp_region4
pp_region0 pp_region5}

create_pg_mesh_pattern mesh_pat1 -parameters {width1 off1} \
-layers { {vertical_layer: M8} {width: @width1} {offset: @off1}
{spacing: 2} {pitch: 200} }

create_pg_mesh_pattern mesh_pat2 -parameters {width1 off2} \
-layers { {vertical_layer: M8} {width: @width1} {offset: @off2}
{spacing: 2} {pitch: 200} }

create_pg_mesh_pattern mesh_pat3 -parameters {width1 off2} \
-layers { {vertical_layer: M8} {width: @width1} {offset: @off2}
{spacing: 2} {pitch: 200} }

set off1 30

set off2 160

foreach pp_region $pp_regions {
set_pg_strategy mesh_strat1 -pg_regions $pp_region \
  -pattern {{pattern: mesh_pat1} {nets: {VDDH VSS VDDL}}
{parameters: 3 104} } \
  -extension {{stop: outermost_ring}}

set_pg_strategy mesh_strat2 -pg_regions $pp_region \
  -pattern {{pattern: mesh_pat2} {nets: {VDDH VSS VDDL}}
{parameters: 3 170} } \
  -extension {{stop: outermost_ring}}

set_pg_strategy mesh_strat3  -pg_regions $pp_region \
  -pattern {{pattern: mesh_pat3} {nets: {VDDH VSS VDDL}}
{parameters: 3 40} } \
  -extension {{stop: outermost_ring}}
  compile_pg -strategies {mesh_strat1 mesh_strat2 mesh_strat3} -
ignore_drc
}
```

Power Switch Insertion

Since the low voltage supply is switched we need to insert power switches. The example code below inserts power switches at the bit_slice level.

```
create_power_switch_array -power_switch PD2_SW -voltage_area
PD_PISO -x_pitch 180 -y_pitch 1.6  -pg_straps [get_shapes -
of_objects [get_nets VDDL]
```

There are multiple ways of connecting power switches. We can connect them in Fishbone, High Fanout, or Daisy Chain as shown in Figure 13.8.

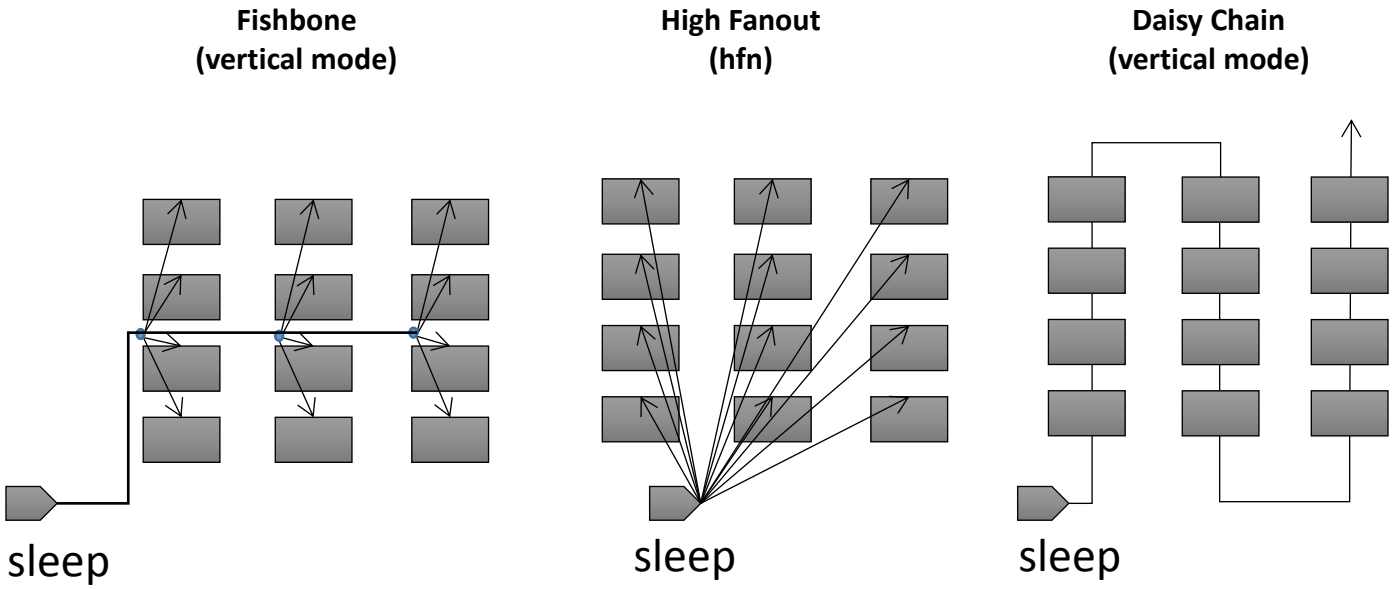


Figure 13.8 – Power Switch Insertion Choices

The mode of connection and the number of switches is decided by the designer after analyzing factors such as area, maximum voltage drop and in-rush current. For bit_coin, we connected the switches as a Daisy Chain. Below is an example showing how to connect the power switches in a daisy chain:

```
connect_power_switch –source sipo_bit/shut_down_signals \
  -ack_out  PG_ack_signals[0] \
  -ring_direction clockwise \
  -mode daisy \
  -port_name PG_ack_signals[0] \
  -voltage_area PD_SIPO \
  -ack_port_namePG_ack_signals[0] \
  -direction horizontal
```

Placement for Low Power

The ICC II placer is VA-aware and places the special cells in the voltage areas corresponding to the power domain where the cell belongs. ICC II inserts always-on buffers and special cells where needed. The picture below shows the placement of special cells at the bit_slice level. Note Yellow = Retention, Green = AON, Red = Enable Level Shifter, and White = Isolation.

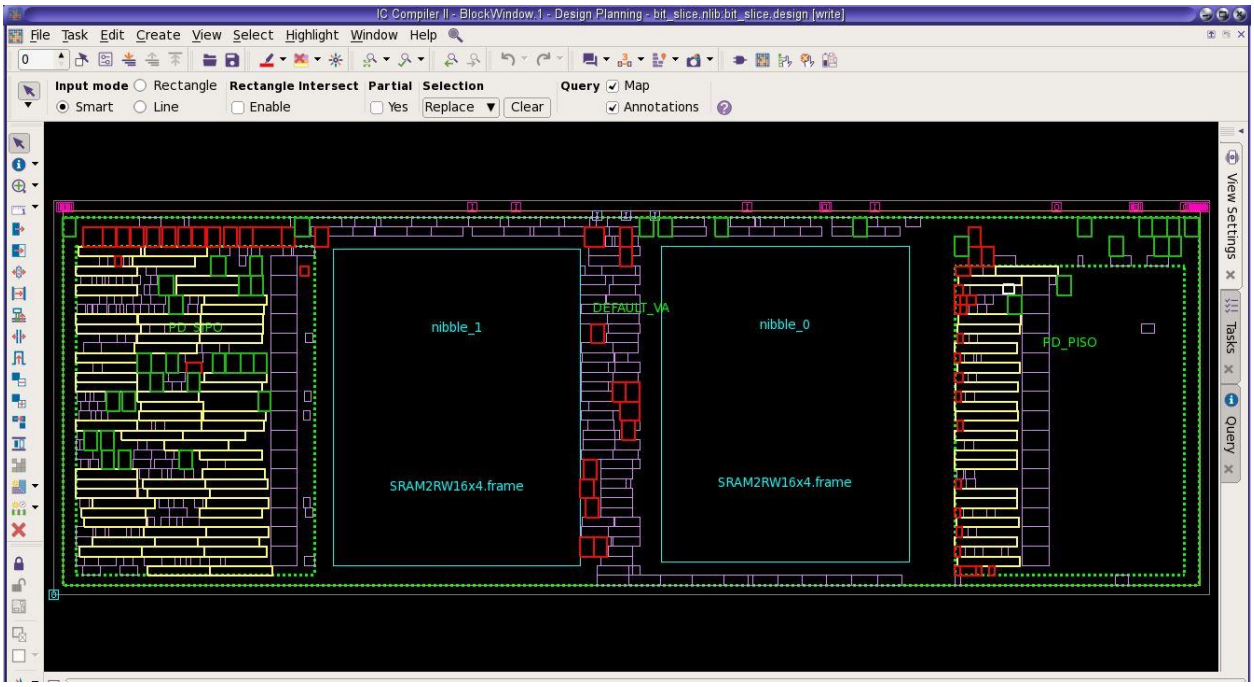


Figure 13.9 – Low Power Cell Insertion

Clock Tree Synthesis (CTS) for Low Power:

The ICC II CTS engine is also VA-aware**.** Register clusters are created respecting voltage areas and clock routing is confined to the voltage areas.

Figure 13-10 shows CTS routing at the bit_slice level. Please note that the clock routes are not zigzagging in and out of the voltage area.

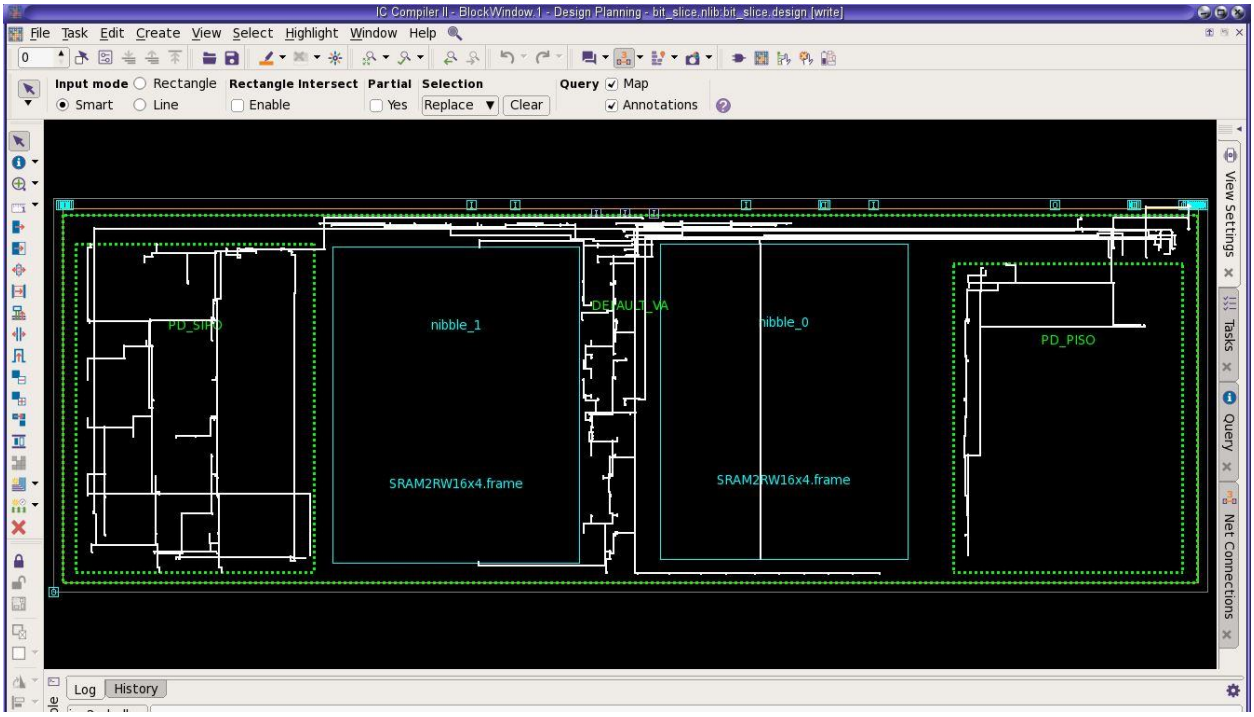


Figure 13.10 – Low Power Clock Tree Synthesis

Routing for Low Power Designs:

ICC II's router follows certain rules while routing a design with voltage areas.

The rules are explained in the diagram below:

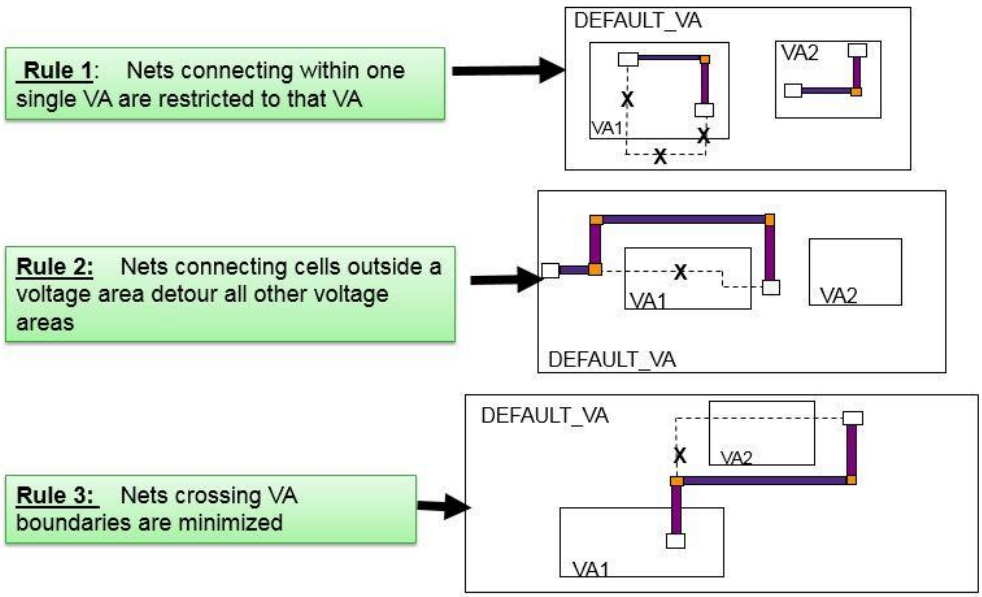


Figure 13.11 – Low Power Routing Rules

For bit_coin, we observed that above rules were being followed while routing.

Figure 13.12 shows a net detouring to avoid passing over a voltage area at the bit_top level.

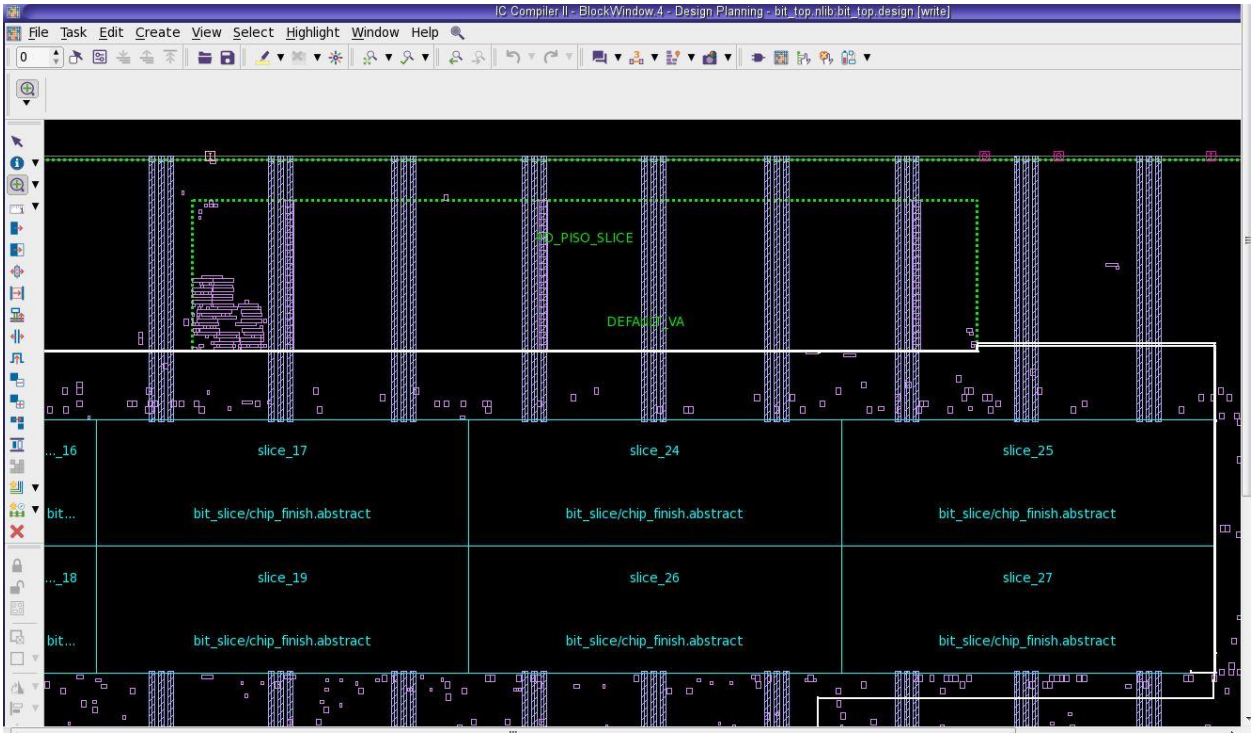


Figure 13.12 – Low Power Routing Example

For secondary PG pin routing, always-on buffers/inverters, retention registers, isolation cells, power switch cells and level Shifters have secondary PG pins that need to be connected to the correct always-on supply.

ICC II's detailed router performs this secondary PG pin connection. These connections can be done in either the pre-route or post-route stage.

Figure 13.13 shows the secondary PG connections at the bit_slice level.

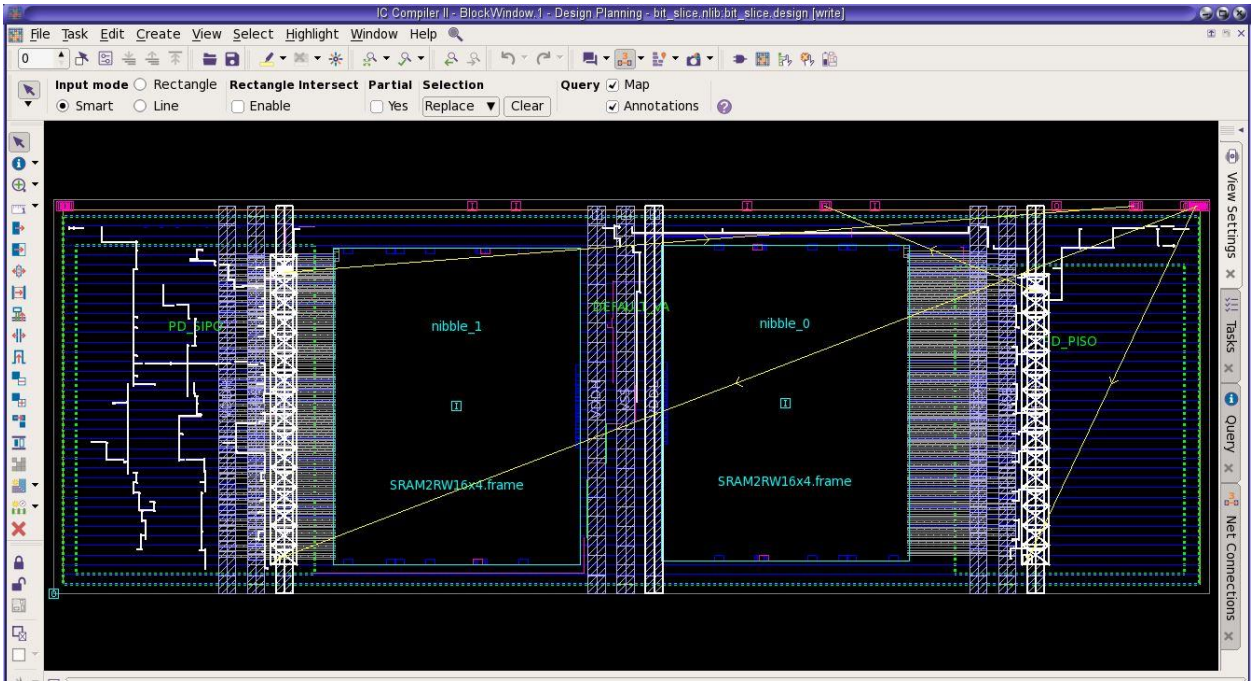


Figure 13.13 – Secondary PG Routing Example

Voltage Drop Analysis Using Prime Rail

PrimeRail is a power network analysis tool to analyze voltage (IR) drop, electromigration (EM), and identify power network integrity problems of the design. PrimeRail enables you to locate potential voltage drop and electromigration violations for system-on-chip (SoC) designs.

It provides accurate power network analysis with signoff quality for all design levels and types such as full-chip, block-level, top-level designs, multi-voltage designs, etc.

PrimeRail offers gate-level static and dynamic voltage drop, electromigration, and power network integrity analyses with full-chip signoff with package parasitics. PrimeRail also performs analysis on switched power domains during full-chip rail analysis.

When rail analysis is complete, PrimeRail provides maps with color highlights to show the problem areas in an easy-to-use graphical user interface. The what-if analysis feature enables you to experiment with user-defined taps, virtual decoupling capacitors, power values, or current values to see the effects of IR drop and electromigration. Early rail analysis can also be done before the design is complete—for example, when only the power routing is done or right after the placement stage in a power network analysis.

PrimeRail can be run from within ICC II which is called In-Design PrimeRail. In-Design Rail Analysis utilizes embedded PrimeRail analysis and fixing guidance technology to enable designers to easily perform power network verification throughout physical implementation within ICC II. By identifying and fixing voltage drop and electromigration issues earlier in the flow, designers can eliminate costly iterations late in the design process. For bit_coin, we performed IR drop analysis on bit_slice using Prime Rail and were able to generate the IR drop and other maps, as shown in Figure 13.14.

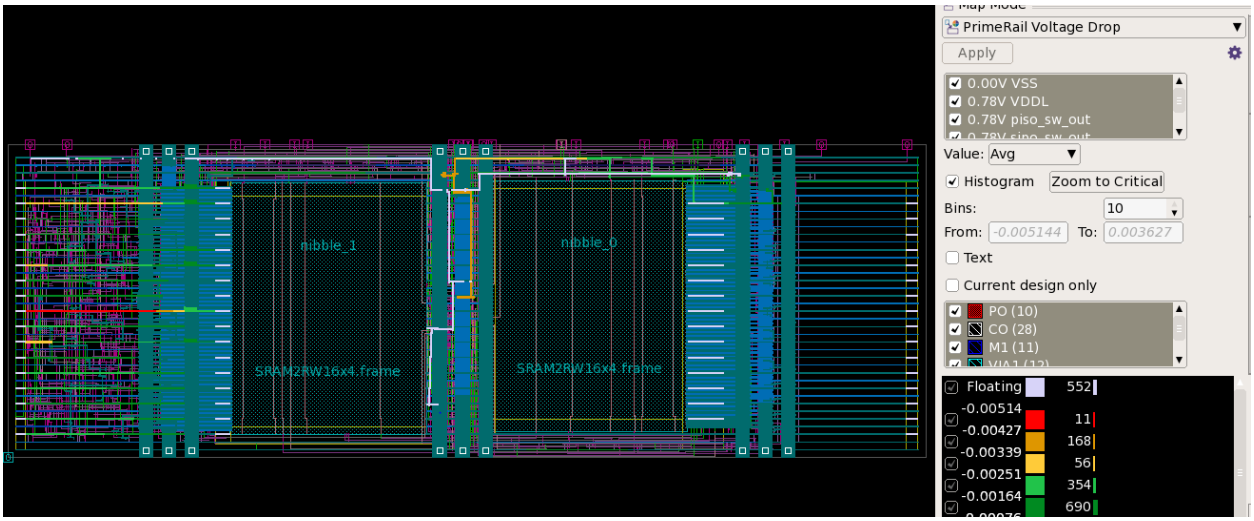


Figure 13.14 – PrimeRail Example on bit_slice

Chapter 14, VC LP @PG Netlist

Overview

After Place/Route, VC LP is run to ensure the PG Gates + UPF" are complete and correct. This is shown in Figure 14.1.

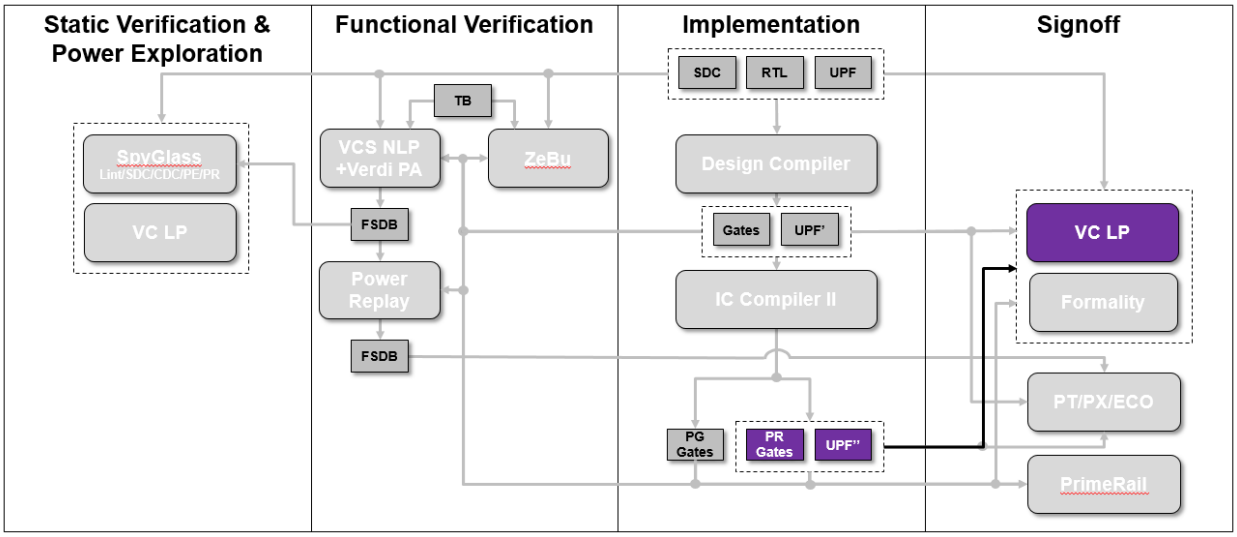


Figure 14.1 - VC LP@PG in the Synopsys Low Power Flow and Methodology

Recall from Chapters 5 and 9, the flow for VC LP, shown again in Figure 14.2.

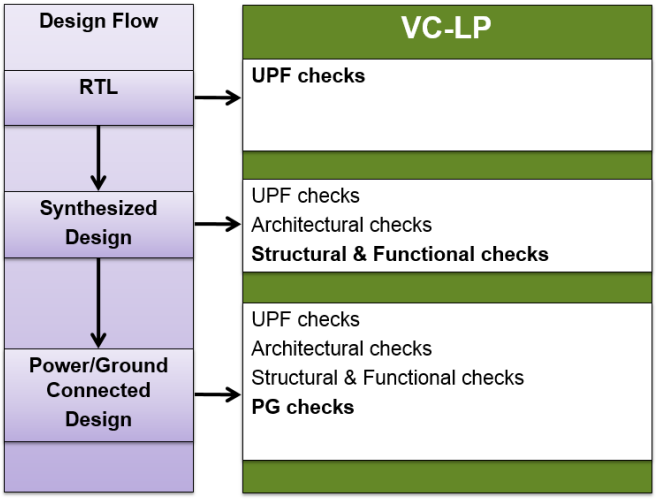


Figure 14.2 - VC LP in the Design Flow

In this section, we'll focus on the "Power/Ground Connected Design", where we will take the Place/Route Netlist and UPF" from ICC II and perform UPF and other checks.

Scripts & Mechanics

The scripts and mechanics are very much like the RTL Stage in Chapter 5 and Netlist Stage in Chapter 9, but here we simply change the design being read (PG netlist), the UPF (UPF"), and add another check_lp command (-stage pg). This is shown in Figure 14.3.

```
# File : go_vclp_pg_pg.tcl
# VC-LP Script @PG
source synopsys_vcst.setup

set ICC_RESULTS_DIR ../pt/signoff_data

read_verilog -netlist ${ICC_RESULTS_DIR}/bit_slice.vclp.v.gz
read_verilog -netlist ${ICC_RESULTS_DIR}/bit_top.vclp.v.gz
read_verilog -netlist ${ICC_RESULTS_DIR}/bit_coin.vclp.v.gz
current_design $DESIGN_TOP
link

load_upf ${ICC_RESULTS_DIR}/wrap_bit_coin.vclp.upf

check_lp -stage upf
check_lp -stage design
check_lp -stage pg

report_lp -list

save_session
```

Figure 14.3 - VC LP @PG

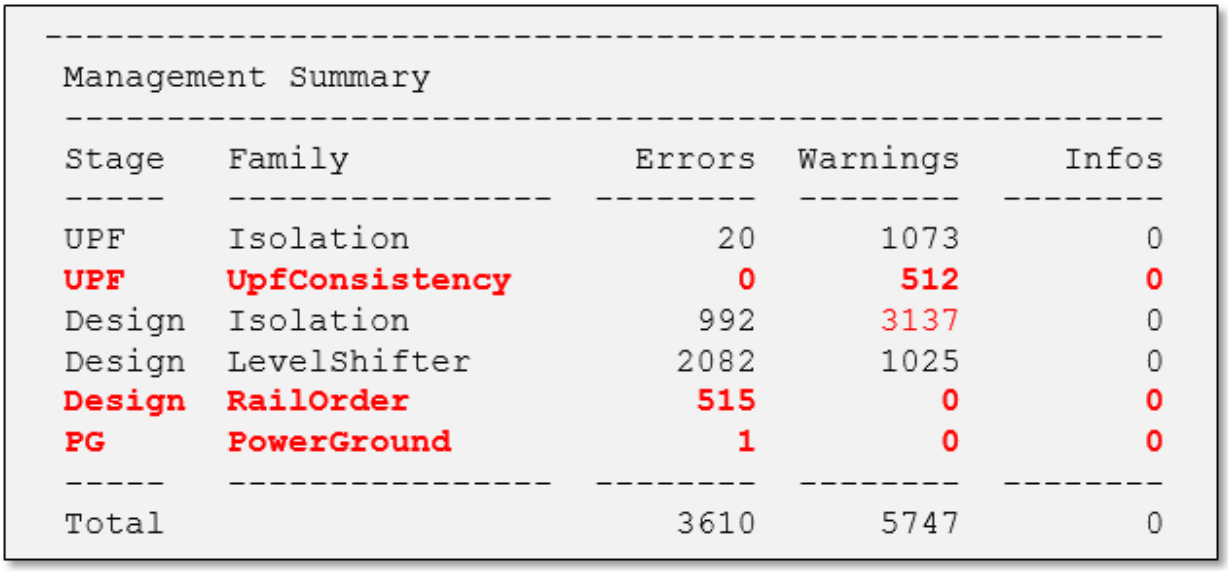
Bitcoin Experience

To run the VC LP @PG, you can use the following commands:

```
% cd bitcoin_v1.1/tools/vclp
% make bit_coin_pg
```

You can view the detailed scripts in the "scripts" directory and the results reports and logs in the "reports" and "logs" directories.

In this case, we see the following results, shown in Figure 14.4.



Management Summary

Stage	Family	Errors	Warnings	Infos
UPF	Isolation	20	1073	0
UPF	**UpfConsistency**	**0**	**512**	**0**
Design	Isolation	992	3137	0
Design	LevelShifter	2082	1025	0
Design	**RailOrder**	**515**	**0**	**0**
PG	**PowerGround**	**1**	**0**	**0**
Total		3610	5747	0

Figure 14.4 - Results for VC LP @PG

We'll cover a few of these messages in this section. To review these issues, simply restore the session and launch the debug GUI. You can use the Makefile and "start_gui" command to accomplish this.

```
% make debug_bit_coin_pg
vc_static_shell> start_gui
```

For RAIL_BUFINV_LEAKAGE, we have 512 Design Stage Errors, as shown in Figure 14.5. Note we have a chain on AON buffers that been inserted between two PISO modules, one in bit_slice, another in bit_top. Since the PISOs are both shut off together, the AON buffers are on in between, while the source supply is OFF, hence the error.

• RAIL_BUFINV_LEAKAGE (512 Design Stage Errors)

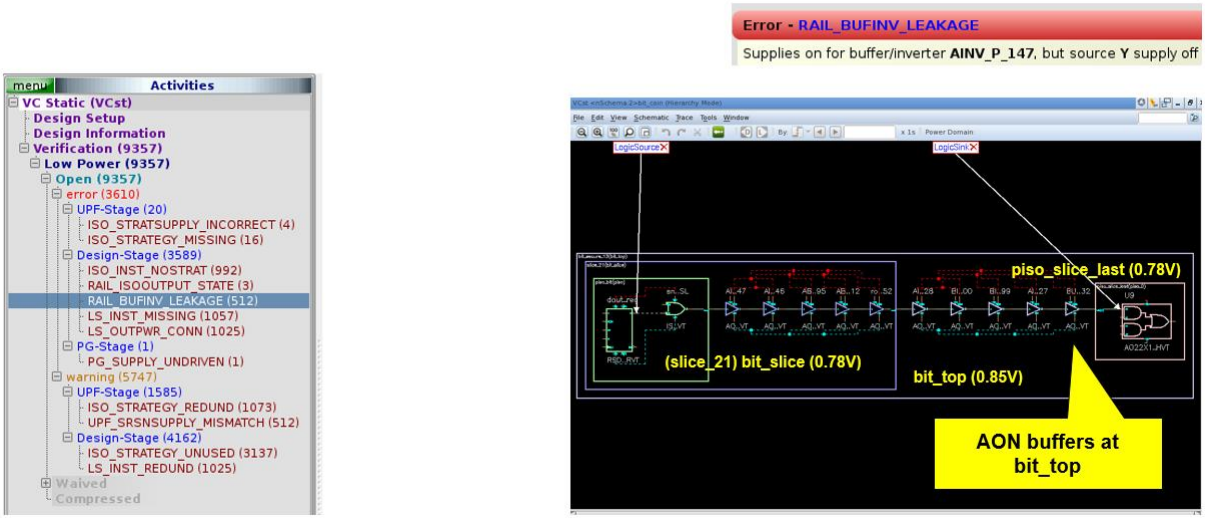


Figure 14.5 - RAIL_BUFINV_LEAKAGE

For PG_SUPPLY_UNDRIVEN, we have 1 PG Stage Error. In this case, the VDDM supply port that we used to short all the internal memories together is essentially undriven at the top, hence the error message given. This is shown in Figure 14.6.

• PG_SUPPLY_UNDRIVEN (1 PG Stage Error)

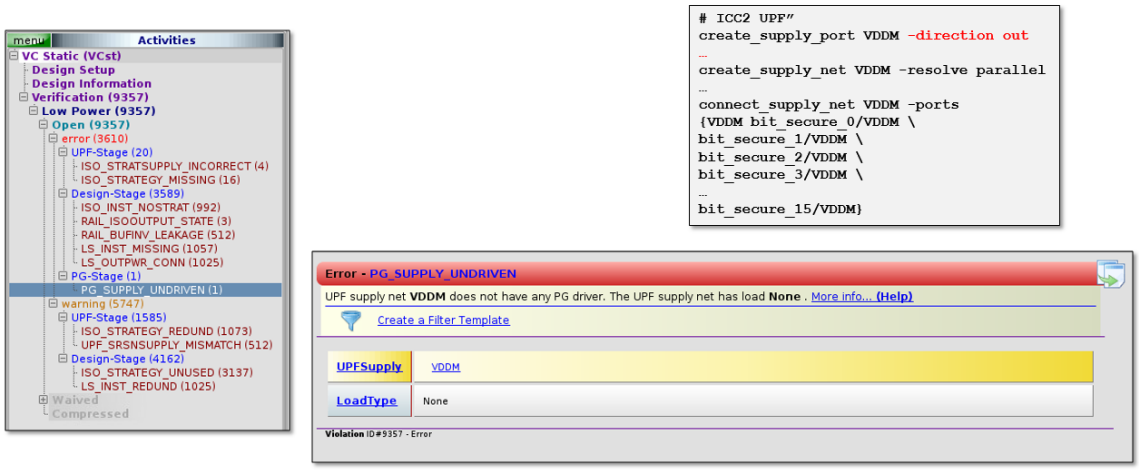


Figure 14.6 - PG_SUPPLY_UNDRIVEN

There are other errors present in the design, and it's a good idea for the user to review them. Many of the remaining issues are due to heterogeneous fanout and are essentially false negatives. But, we'll leave it to the user to investigate those further.

Chapter 15, Formality @RTL2PG Netlist

Overview

Formality performs RTL2PG and Gate2PG equivalency checks in the Low Power Flow. This is shown in Figures 15.1 and 15.2.

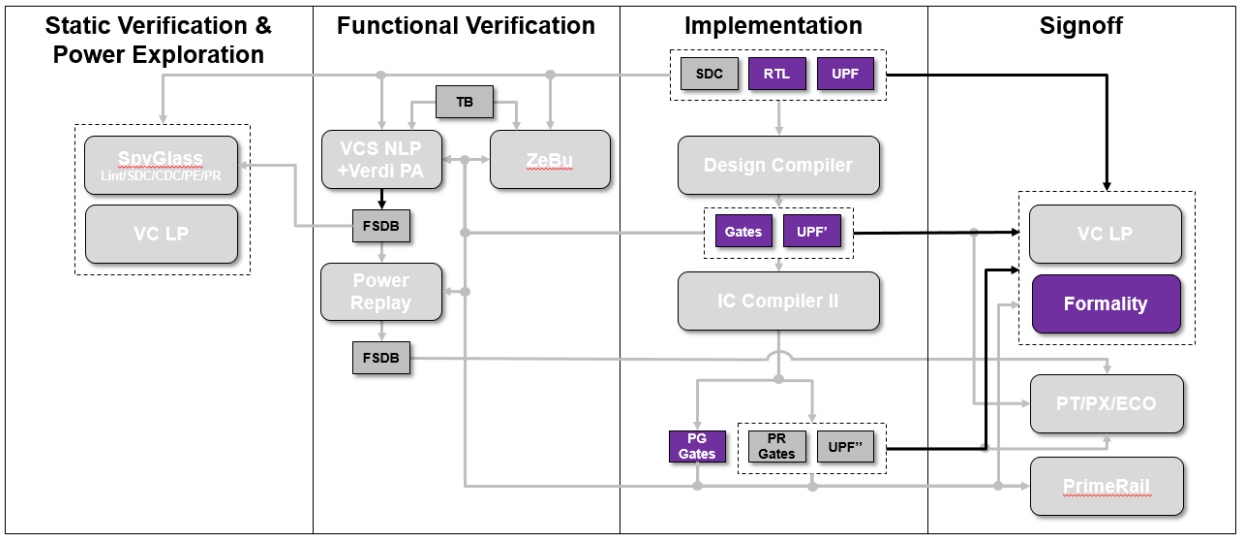


Figure 15.1. Formality in the Synopsys Low Power Flow and Methodology

The following are required for verifying RTL2PG verification.

- UPF cannot be loaded onto a PG netlist
- Don’t use set_constant on the supply ports/nets on the PG netlist
- Can only verify all-on state because PG design has no PST associated with it

Scripts & Mechanics

An example of a Formality RTL2PG script is shown (RTL+UPF vs. ICC II PG Netlist) here:

```
read_db {low_power_library.db special_lp_cells.db}
read_verilog -r -f rtl_files.f
set_top r:/WORK/top
load_upf -r top.mapped.upf
read_verilog -i { post_icc_netlist.v }
set_top i:/WORK/top
verify
```

Bitcoin Experience

To run Bitcoin with Formality, first make sure your Formality software and licenses are configured and installed. Then, simply type the following commands:

```
% cd bitcoin_v1.1/tools/fm
% make bit_slice_rtl2pg
% make bit_coin_rtl2pg
```

This Makefile target will execute the following command, and you can peruse the script and logs to learn more.

```
% fm_shell -f scripts/go_fm_bit_slice_rtl2pg.tcl | tee
logs/log.fm.bit_slice_rtl2pg

% fm_shell -f scripts/go_fm_bit_coin_rtl2pg.tcl | tee
logs/log.fm.bit_coin_rtl2pg
```

Running Bitcoin through Formality at the RTL2PG verification stage was clean, fast, and efficient, noted in Figure 15.2.

Design	Failures	Errors	Total Time (sec)	CPU Time (sec)	Peak Memory (MB)
bit_slice	2	0	17	12	472
bit_coin	0	0	4520 (1.25 Hrs)	4504	5836

Figure 15.2 - Formality bit_slice and bit_coin Results

Figure 15.3 shows how Formality caught failing points in bit_slice on pwr_ack signals. The failures were due to power switches not being implemented so there was a real difference between the RTL and PG.

```
************************************ Verification Results ************************************
Verification FAILED
   ATTENTION: Verification was run with all UPF supplies constrained to their ON state.
              This is only a partial verification result as it does not cover all
              operational states.
   ATTENTION: Top-level/black_box ports were assumed to be related to the primary supplies.
              These results are dependent on this assumption being independently verified.
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
--------------------------------------------------------------------------------------------

 Reference design: r:/WORK/bit_slice
 Implementation design: i:/WORK/bit_slice
 230 Passing compare points
 2 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
--------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut     Port   PDCut     DFF      LAT   TOTAL
--------------------------------------------------------------------------------------------
Passing (equivalent)          54       0       0       0       4      22     150        0     230
Failing (not equivalent)       0       0       0       0       2       0       0        0       2
Not Compared
  Don't verify                 0       0       0       0       4       0       0        0       4
********************************************************************************************
```

Figure 15.3 - Formality bit_slice Results

In Figure 15.4, Formality successfully verified the top-level design (bit_coin).

```
************************************ Verification Results ************************************
Verification SUCCEEDED
   ATTENTION: Top-level/black_box ports were assumed to be related to the primary supplies.
              These results are dependent on this assumption being independently verified.
   ATTENTION: synopsys_auto_setup mode was enabled.
              See Synopsys Auto Setup Summary for details.
--------------------------------------------------------------------------------------------

 Reference design: r:/WORK/bit_coin
 Implementation design: i:/WORK/bit_coin
 119478 Passing compare points
--------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut     Port   PDCut     DFF      LAT   TOTAL
--------------------------------------------------------------------------------------------
Passing (equivalent)       27648       0    1026       0     120   11458   79226        0  119478
Failing (not equivalent)       0       0       0       0       0       0       0        0       0
Not Compared
  Don't verify                 0       0       0       0       4       0       0        0       4
********************************************************************************************
```

Figure 15.4 - Formality bit_coin Results

Chapter 16, VCS @PG Netlist

Overview

As a final sign-off VCS is used at PG stage to dynamically verify the design intent with all the low power structures implemented inside the netlist. Refer to figure 16.1 below, inputs to this flow are PG (netlist out of ICC II) and testbench (commonly written in UVM). For debug Verdi is still used though given that there is no UPF it is very much like standard HDL debug. I should mention that while user has the option to still pass UPF'' (out of ICC II) there isn't any additional value running with UPF'' at PG (power-ground) netlist assuming all the models are power-aware and deemed to be functionally accurate.

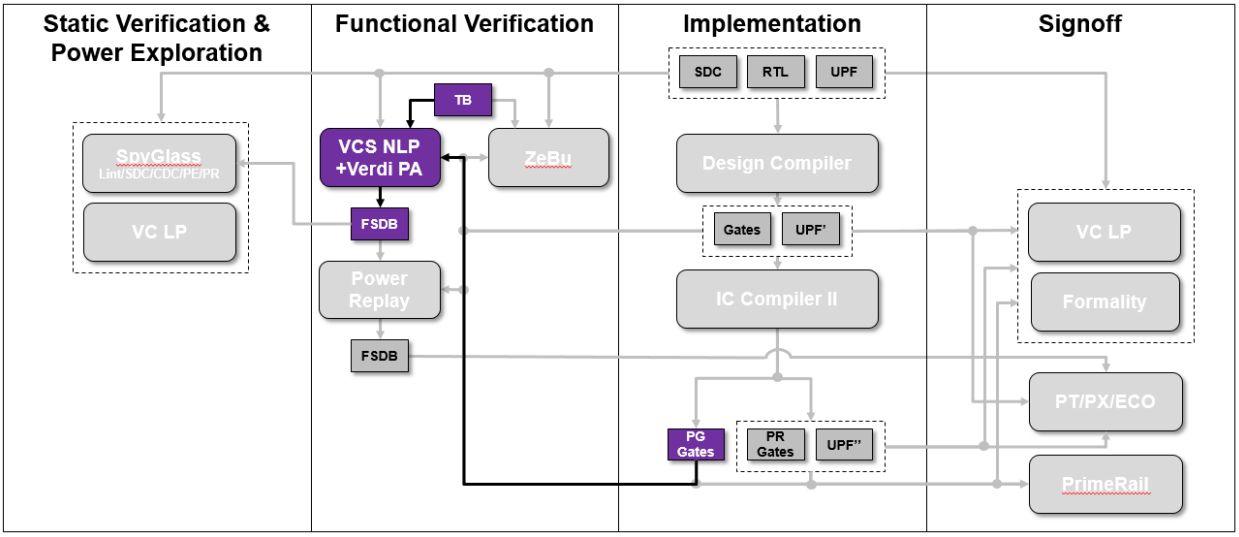


Figure 16.1 - VCS@PG in the Synopsys Low Power Flow and Methodology

As mentioned in chapter 6 and chapter 11 overview's VCS in PG netlist flow in contrast to verifying the power intent statically (in VC LP as example) the primary focus here should be to validate the same design dynamically - power and related control signal sequencing and effects of that. Figure 16.2 illustrates this point.

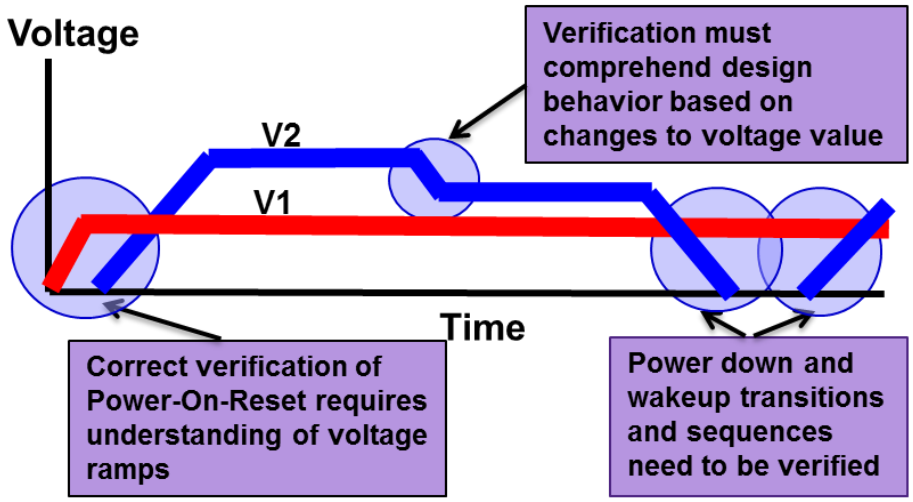


Figure 16.2 - VCS@PG Dynamic Low Power Verification

Scripts & Mechanics

In PG netlist simulation since it is a PG connected netlist with power aware libraries there isn't any additional value of running with NLP (as in passing UPF) hence the suggestion to go with standard Verilog flow. For debug the most common options remain the same as before -debug_access+pp -debug_region=lib+cell

Figure 16.3 shows the screenshot of a typical VCS compile command, note some additional options are being passed/shown below since they help in compile time turn-around for incremental compiles.

```
vcs  -partcomp -fastpartcomp=j4 -timescale=1ns/1ps -sverilog -l vcs.log +warn=noIWNF -ntb_opts uvm-1.2 +define+UV
M_POST_VERSION_1_1 +incdir+../../verification/include+../../verification/src+../../verification/env+../../verific
ation/tests+../../verification/hdl  +vcs+initreg+random  -debug_region=lib+cell -debug_access+pp -lca -kdb +defin
e+DUMP \
                ../../verification/tests/tb_top.sv ../../verification/hdl/bitcoin_top.sv \
               -F ../../verification/hdl/filelist_pggates.f +define+NO_SAIF +nospecify -power=rtlpg -Mdir=output
/PG_DEBUG_NOUPF/csrc -o output/PG_DEBUG_NOUPF/simv \
               -l logs/PG_DEBUG_NOUPF/compile.log \
               -partcomp_dir=output/PG_DEBUG_NOUPF/partitionlib
```

Figure 16.3 – VCS Compile Command

For running the simulation since we use UVM, you must pass +UVM_TESTNAME=<test>. Figure 16.4 shows the screenshot of a typical VCS runtime command again it shows some additional options to control the location of FSDB file, log file and control the number of transactions.

```
./output/PG_DEBUG_NOUPF/simv +min_random_count=50 +max_random_count=50 +vcs+initreg+0 +fsdbfile+./logs/PG_DEBUG_N
OUPF/dump.fsdb +SAIF_REPORT=./logs/PG_DEBUG_NOUPF/bitcoin.saif -l logs/PG_DEBUG_NOUPF/run.log \
       +ntb_random_seed=1 +timeout=200000 +UVM_TESTNAME=btcoin_test +fsdb+all
cp logs/PG_DEBUG_NOUPF/dump.fsdb outputsfromvcs/PG_NOUPF.fsdb
```

Figure 16.4 – VCS Runtime Command

Bitcoin Experience

Assuming the suggested tools are installed and licenses available. To run Bitcoin with VCS along with Verdi at PG

```
% cd bitcoin_v1.1/tools/vcs_nlp
% make PHASE=PG
```

The default 'make' target is to compile & run PG simulation and dump FSDB and as noted above there is no UPF being passed. The below shows various arguments (most common) that you can pass to 'make' based on the intended flow.

Note: The make flow below shows compile+run together as part of the default target, you can split the flow into steps to compile and run separately (explained later).

To compile & run without dumping:

```
% make MODE=NODEBUG PHASE=PG
```

Default UVM transactions are 50, to compile & run it for longer or shorter period.

```
% make MAX_TRANS_CNT=<integer value> PHASE=PG
```

The target for only compiling is:

```
% make comp PHASE=PG
```

This target can be used in combination with the above list arguments to 'make'. For example, if you wanted to only compile without DEBUG the command will be:

```
% make comp MODE=NODEBUG PHASE=PG
```

Likewise target for only running is:

```
% make run PHASE=PG
```

This target can be used in combination with the above listed arguments to 'make'. Just be aware though that some options are only applicable at compile not at runtime. For example, if you wanted to run the test longer (more transaction) then you could do this:

```
% make run MAX_TRANS_CNT=100 PHASE=PG
```

Also, important to note, to keep the data unique, each flow (with or without UPF, with or without debug) is written to its own unique directory. The directory structure that is followed is explained below,

Compile outputs go under 'output' directory

- Created based on Makefile arguments PHASE, MODE, LP
- Examples
  - Make PHASE=PG → simv/csrc written into output/PG_DEBUG_NOUPF
  - make PHASE=PG MODE=NODEBUG → simv/csrc written into output/PG_NODEBUG_NOUPF

All log files including FSDB dump go under ‘log’ directory

- Same concept as compile, examples
  - Make PHASE=PG -> compile.log, run.log, dump.fsdb written into logs/PG_DEBUG_NOUPF
  - make PHASE=PG MODE=NODEBUG -> compile.log, run.log, dump.fsdb written into logs/PG_NODEBUG_NOUPF

During VCS compile at PG gates one difference that was observed that if the netlist is bit-blasted coming from ICC II the netlist might contain module instantiation of the form as shown in figure 16.5. This shows module instantiation with port names that are escaped like below ‘.\I1[3] ’

From a Verilog standpoint tools look for an actual port – ‘.\I1[3] ‘ it can’t be mapped to a bus – I1 or a bit of a bus I1[3]. So, this will result in compile errors. Since ICC II can write out multiple forms of netlist so we used the non-bit blasted version like Formality.

```
    .ccd_setup_0 ( ccd_setup_0 ) , .p_abuf0 ( APSDRC_net_138 ) ,
    .ccd_setup_1 ( ccd_setup_2 ) ) ;
SRAM2RW16x4 nibble_0 ( .CE1 ( cts0 ) , .ACK ( \power_ack[0] ) ,
    .SLEEPIN ( memory_sleep ) , .CSB1 ( optlc_net_197 ) , .WEB1 ( n_6_net_ ) ,
    .OEB1 ( optlc_net_194 ) , .CE2 ( cts0 ) , .CSB2 ( optlc_net_196 ) ,
    .WEB2 ( n_6_net_ ) , .OEB2 ( optlc_net_194 ) , .\I1[3] ( pout2mem_3_ ) ,
    .\I1[2] ( pout2mem_2_ ) , .\I1[1] ( lv_scan_out ) ,
    .\I1[0] ( pout2mem_0_ ) , .\O1[3] ( \t_din2piso[3] ) ,
    .\O1[2] ( \t_din2piso[2] ) , .\O1[1] ( \t_din2piso[1] ) ,
    .\O1[0] ( \t_din2piso[0] ) , .\A1[3] ( \pout_addr2mem[3] ) ,
    .\A1[2] ( \pout_addr2mem[2] ) , .\A1[1] ( \pout_addr2mem[1] ) ,
    .\A1[0] ( \pout_addr2mem[0] ) , .\I2[3] ( pout2mem_7_ ) ,
    .\I2[2] ( pout2mem_6_ ) , .\I2[1] ( pout2mem_5_ ) ,
    .\I2[0] ( pout2mem_4_ ) , .\O2[3] ( \t_din2piso[7] ) ,
```

Figure 16.5 – Module Instantiation of Memory from ICC II

Chapter 17, PrimeTime, PrimeTime PX, PT-ECO

Overview

The PrimeTime suite provides a single, golden, and trusted signoff solution for timing, signal integrity, and power analysis. PrimeTime PX supports analysis in both averaged and time-based modes of low power designs which employs UPF power management specifications.

Key benefits include:

- Faster turnaround time with parallel processing and support for multiple scenario runs
- HSPICE-like accuracy to minimize design margins and advanced node support
- Improved productivity through faster timing reports, ECO capabilities, etc.

The PrimeTime family of products are shown in Figure 17.1 as part of the complete Synopsys Low Power Flow and Methodology.

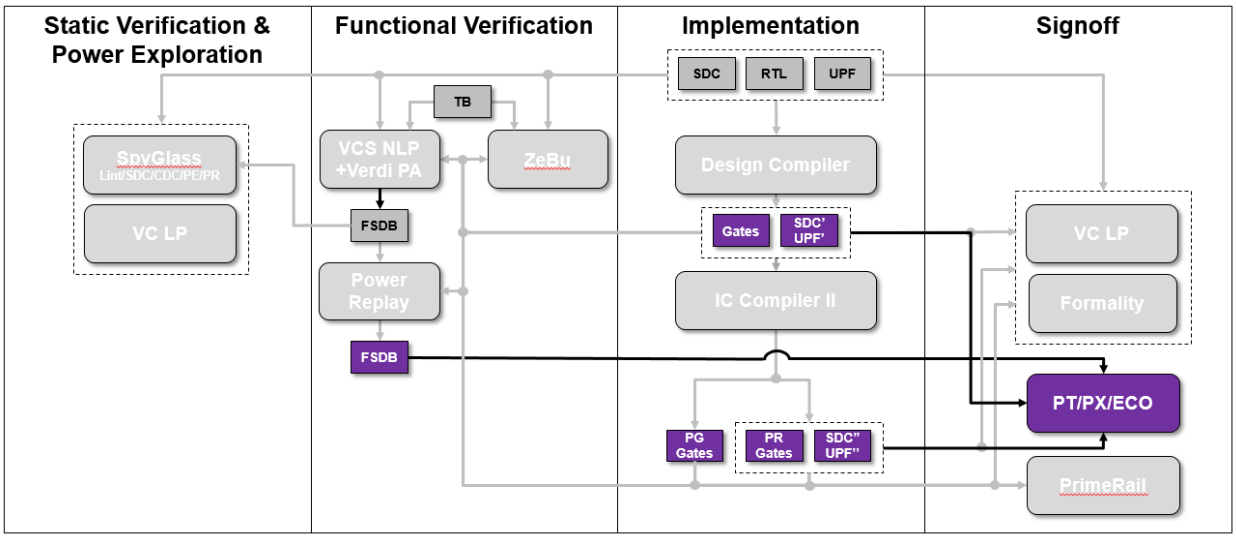


Figure 17.1 – PrimeTime Family in the Synopsys Low Power Flow and Methodology

In Figure 17.2, the PrimeTime signoff flow with ICC II is shown. Here, the output from ICC II and StarRC™ are read into PrimeTime, and then timing analysis and reports are generated. Also, if there are timing violations, physically-aware ECO analysis can be run where ECO scripts generated for ICC II can be used to close timing.

In the PrimeTime PX Flow, similar input from ICC II are read into PrimeTime PX where power analysis is performed and power reports are generated.

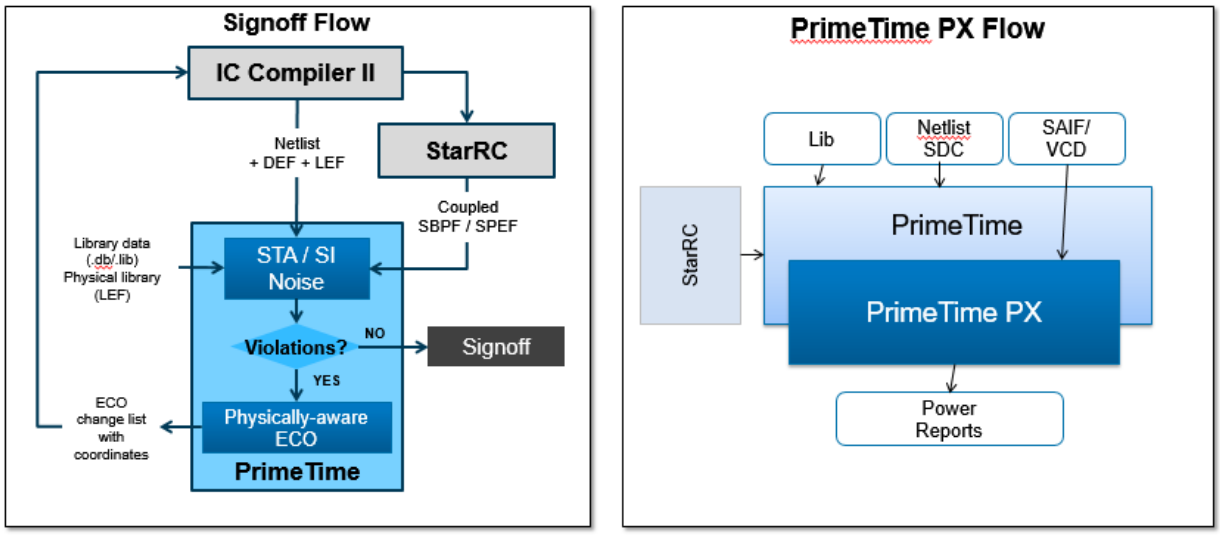


Figure 17.2 - PrimeTime Signoff Flow

PrimeTime PX supports two types of power analysis modes—averaged and time-based. In each of the modes, the tool supports various options to suit different types of designs and applications.

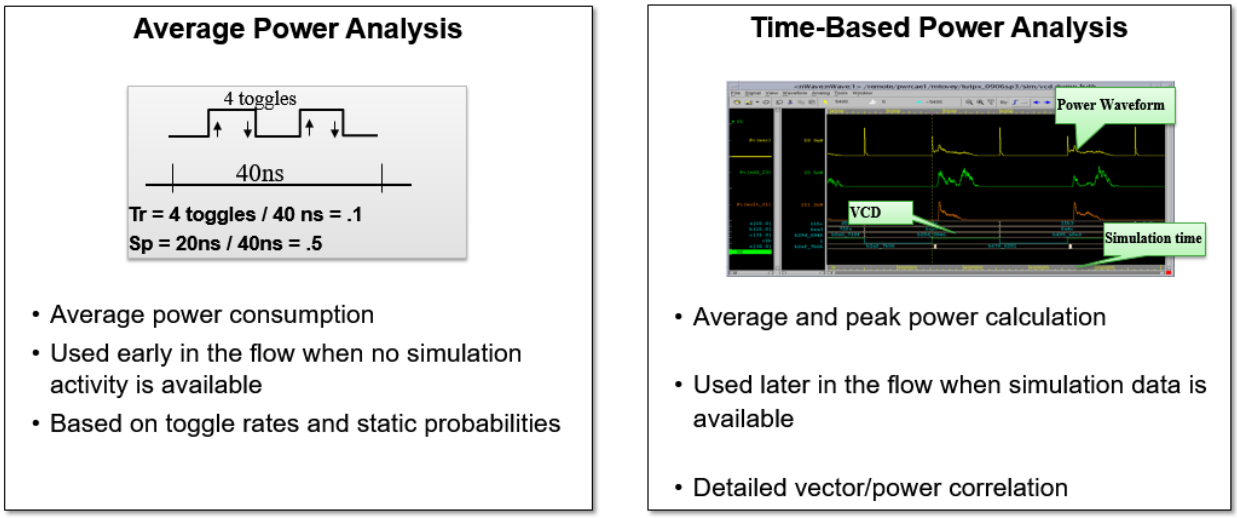


Figure 17.3 - Power Analysis Modes in PrimeTime PX

In the averaged power analysis mode, the tool calculates averaged power based on toggle rates. The sources of annotated activity can be the default toggle rates, user-defined switching activity, SAIF files, or VCD files generated from simulation.

In time-based analysis mode, the tool calculates the power per event to generate power waveforms over time, as well as the averaged power. Based on the type of event-based switching activity file (SAIF) provided, the tool performs instantaneous peak power analysis or cycle-accurate peak power analysis.

For average power calculation, Figure 17.4 shows the elements associated, including leakage, internal, and switching power.

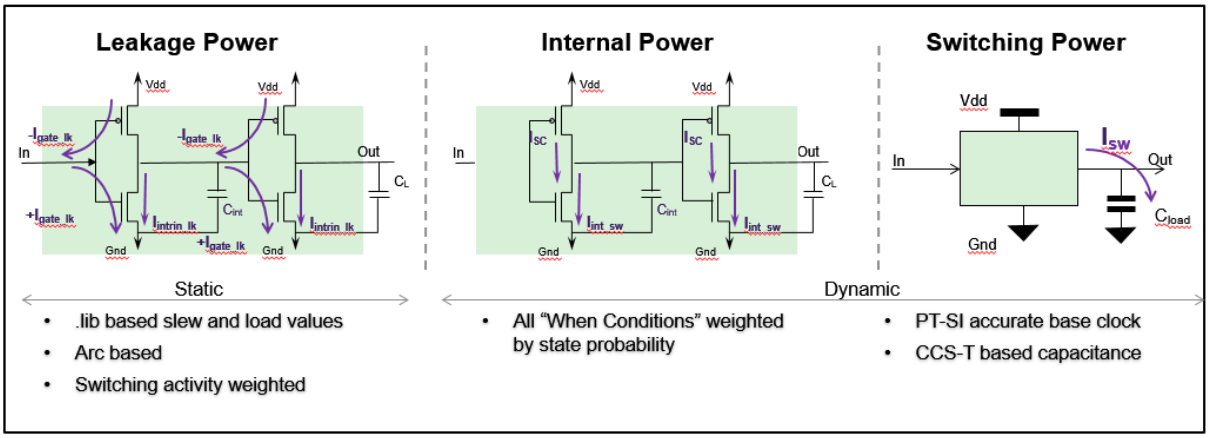


Figure 17.4 – Average Power Calculation

PrimeTime PX also includes sophisticated design analysis and debug tools as well, as shown in Figure 17.5

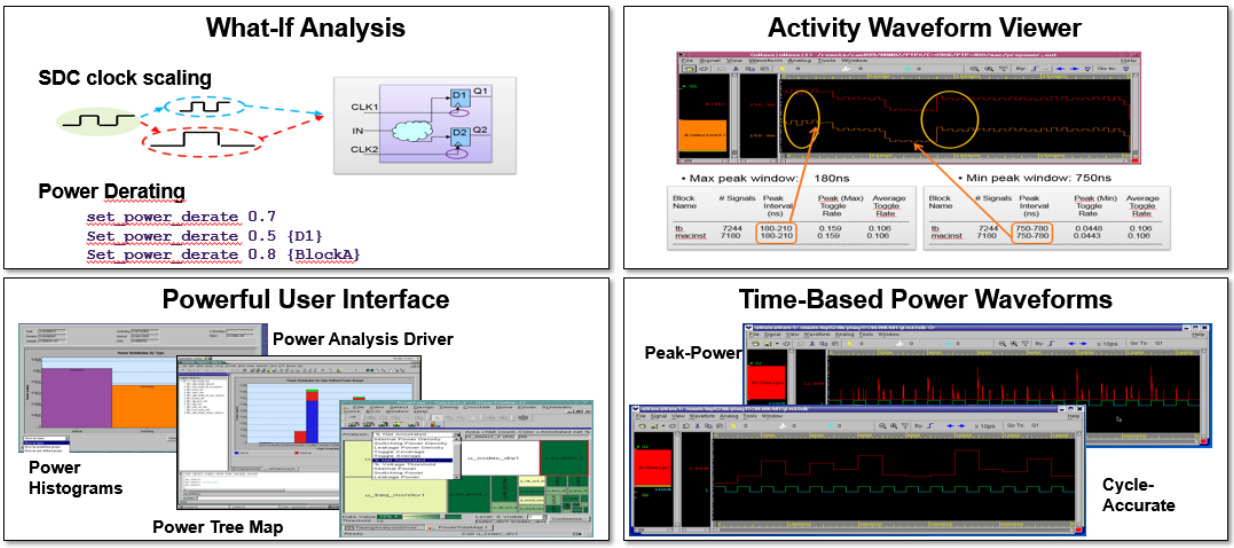


Figure 17.5 – Design Analysis and Debug in PrimeTime PX

Scripts & Mechanics

To start with PrimeTime PX, you should have a setup file, which includes information about following

- **Technology library:** Cell library containing timing and power characterization information for each cell. The quality of the library determines accuracy of the power results.
- **Gate-level netlist:** Flat or hierarchical gate-level netlist in Verilog, VHDL or Synopsys database format containing leaf-level instantiation of library cells.
- **Design constraints:** SDC file containing the design constraints to calculate the transition time on the primary inputs and to define the clocks.
- **Activity:** Design switching activity information either for average power analysis or for accurate peak power analysis.
- **Net parasitics:** Parasitics file (SPEF) containing net capacitance for all nets.

Let's go through basic sample code to understand how PT-PX is setup to perform power analysis.

PrimeTme Setup for Reading Technology Libraries

The search_path string tells PrimeTime to look for all input data files such compiled libraries, Verilog netlist, constraints, etc. The link_library variable defines all the standard cell and memory model dbs for the cells used in the gate level netlist. Below is sample code for defining search path and link library. This is shown in Figure 17.6.

```
# File : PrimeTime Setup

set search_path [list \
  lib \
  lib/sram/ndms/lib_dbs \
  lib/stdcell_hvt/db_nldm_gm \
  lib/stdcell_lvt/db_nldm \
  lib/stdcell_rvt/db_nldm]

set link_library [list \
  saed32hvt_tt0p78v125c.db \
  saed32hvt_tt0p85v125c.db \
  saed32hvt_tt1p05v125c.db \
  saed32hvt_ulvl_tt0p78v125c_i0p78v.db \
  saed32hvt_ulvl_tt0p85v125c_i0p85v.db \
  saed32hvt_ulvl_tt1p05v125c_i0p78v.db \
  saed32hvt_dlvl_tt0p78v125c_i0p78v.db \
  saed32hvt_dlvl_tt0p85v125c_i0p85v.db \
  saed32hvt_pg_tt0p78v125c.db \
  saed32hvt_pg_tt0p85v125c.db \
  saed32hvt_pg_tt1p05v125c.db \
  ...
  sram2rw16x4_tt1p78v125c.db \
  sram2rw16x4_tt1p85v125c.db]
```

Figure 17.6 – Example PrimeTime Setup File

PrimeTime Setup for Reading Design Data

The design data such as gate level netlist, UPF file, constraints, parasitics, and switching activity file is read for timing and power analysis, as shown in Figure 17.6. Additionally, the library scaling groups are added because we have a multi-voltage design.

```
read_verilog test.vg
current_design test
link
define_scaling_lib_group {lib_tt0p78v125c.db lib_tt0p85v125c.db lib_tt1p05v125c.db}
...

load_upf test.upf
read_sdc test.sdc
read_parasitics test.spef
read_fsdb test.fsdb
```

Figure 17.6 – Basic PrimeTime Script – Read Design Data

PrimeTime Setup for Timing/Power Analysis

Timing and power analysis are done after reading all the necessary design data and loading the design in the memory. update_timing and update_power commands are used for respective analysis.

Different timing and power reports are generated to analyze power reports, timing violations, and DRC violations (max_transition, max_cap, max_fanout, etc)

This is shown in Figure 17.7.

```
set power_enable_analysis TRUE
set si_enable_analysis TRUE
set power_analysis_mode time_based
set_power_analysis_options -waveform_output fsdb
update_timing -full
update_power
report_power
report_global_timing
report_constraint -all_viol
```

Figure 17.7 – Basic PrimeTime Script – Timing/Power Analysis

Bitcoin Experience

To run Bitcoin with PrimeTime, PrimeTime PX, and PrimeTime ECO, simply type the following commands:

```
% cd bitcoin_v1.1/tools/pt
% make bit_coin
```

This Makefile target will execute the following command, and you can peruse the script and logs to learn more.

```
% pt_shell -f scripts/go_pt_bit_coin.tcl | tee
logs/log.pt.bit_coin
```

Use the save_session and restore_session commands to save the current state of a PrimeTime PX tool session and restore the same session later to examine the results of timing and power analysis. To run Interactive PrimeTime session type following command:

```
% make bit_coin_debug
```

BitCoin PrimeTime QoR Report

Command report_qor -summary provides an overview of the quality of a design in Figure 17.8.

```
Summary of timing violations
--------------------------------------------------------------------------------
Setup WNS:   -1.56  TNS:    -98.79  Number of Violating Paths:    326
Hold  WNS:   -1.81  TNS:   -208.34  Number of Violating Paths:    433
--------------------------------------------------------------------------------
```

Figure 17.8 – QOR Report

A summary of Design Rule Violations is shown in Figure 17.9.

```
Design Rule Violations
------------------------------------------
Total No. of Pins in Design:          1015203
sequential_clock_pulse_width Count:     35803
max_capacitance Count:                  18695
min_capacitance Count:                   1228
max_transition Count:                    1996
sequential_clock_pulse_width Cost:     -6648.07
max_capacitance Cost:                -226817.62
min_capacitance Cost:                   -203.28
max_transition Cost:                    -139.61
Total DRC Cost:                      -233808.58
------------------------------------------
```

Figure 17.9 – Design Rule Violations Summary

BitCoin PTPX Power Analysis Report

For the PrimeTime PX tool to perform time-based power analysis, set the power_analysis_mode variable to time_based. To specify additional options for the analysis, use the set_power_analysis_options command.

```
set power_analysis_mode time_based
set_power_analysis_options -waveform_output fsdb
```

Figure 17.10 – PrimeTime PX power analysis options

By default, the report_power command outputs the top-level Group -based power consumption. You can generate other types of report by specifying options as -cell_power, -net_power or -hierarchy.

```
****************************************
Report : Time Based Power
Design : bit_coin
Version: M-2016.12-SP2
Date   : Tue Mar 21 07:47:52 2017
****************************************


  Attributes
  ----------
      i  -  Including register clock pin internal power
      u  -  User defined power group

                       Internal  Switching  Leakage    Total
Power Group            Power     Power      Power      Power   (     %)  Attrs
--------------------------------------------------------------------------------
clock_network             0.1041    0.0322    0.0519    0.1882 (26.27%)  i
register                -1.970e-03
                                  9.292e-03    0.1390    0.1463 (20.43%)
combinational           7.168e-03    0.0125    0.1099    0.1295 (18.08%)
sequential                0.0000    0.0000    0.0000    0.0000 ( 0.00%)
memory                    0.2519 3.508e-04    0.0000    0.2522 (35.22%)
io_pad                    0.0000    0.0000    0.0000    0.0000 ( 0.00%)
black_box                 0.0000    0.0000    0.0000    0.0000 ( 0.00%)

  Net Switching Power  =    0.0543   ( 7.58%)
  Cell Internal Power  =    0.3612   (50.43%)
  Cell Leakage Power   =    0.3008   (41.99%)
                         ---------
Total Power            =    0.7162  (100.00%)

X Transition Power     = 6.713e-03
Glitching Power        = 3.734e-06

Peak Power             =    4.5041
Peak Time              =      1035
```

Figure 17.11 – Power Report from PrimeTime PX

Chapter 18, Low Power Measurement Flow

Now that we've gone through the entire Synopsys Low Power Flow and Methodology, shown again as a reminder in Figure 19.1, we can spend some time discussing the Low Power Measurement Flow.

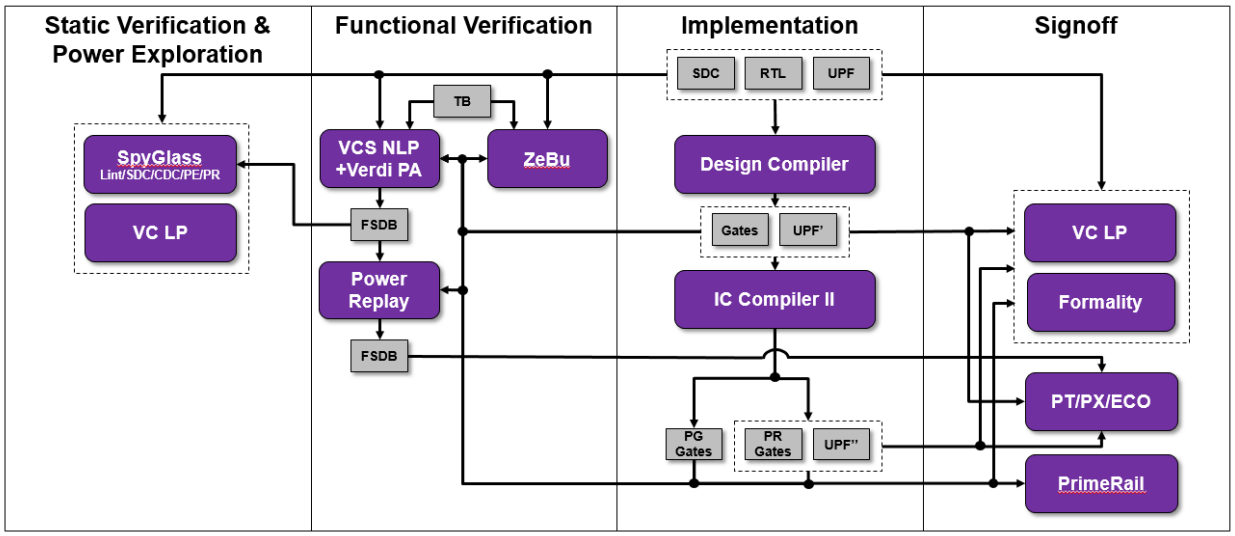


Figure 19.1 - Synopsys Low Power Flow and Methodology

Power Exploration vs. Power Correlation

First off, we need to distinguish between Power "Exploration" and Power "Correlation". At the RT-level, where the design is not yet mapped to gates with all the precision of a production synthesis tool; and all the power switch elements, isolation and retention cells, and other low power elements are not yet included, it's best to think of Power "Exploration" and "Relative Power" at this stage. Hence, the SpyGlass PE Flow, shown in Figure 19.2, shows the relative measure of power between RTL versions as something of interest.

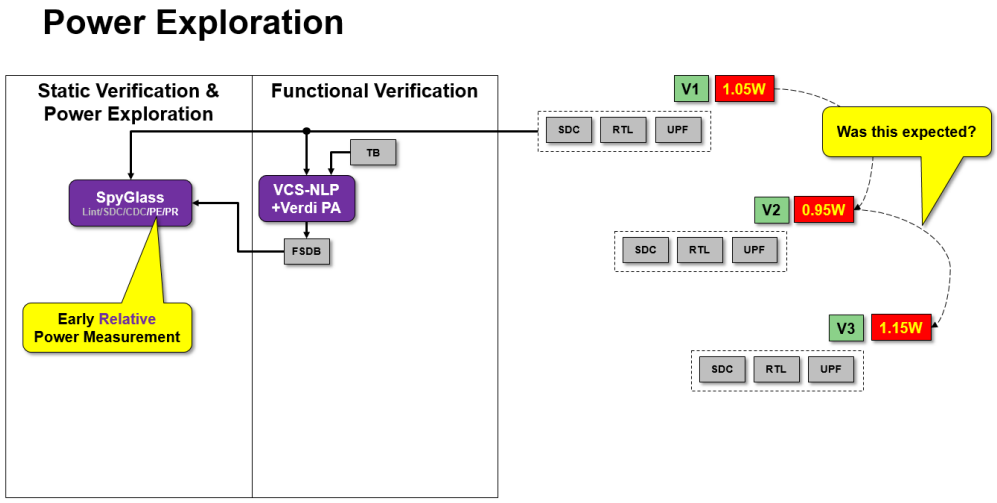


Figure 19.2 - Synopsys Power Exploration Flow

Once the RTL has been synthesized with a tool like DC (or DC Explorer), you can start thinking about Power "Correlation" since the gates coming from DC are much more in-tune with what will be produced downstream. It makes to do a "quick-and-dirty" synthesis run and measure power with PrimeTime PX, then correlate that with subsequent DC and then ICC II runs as well, as shown in Figure 19.3.

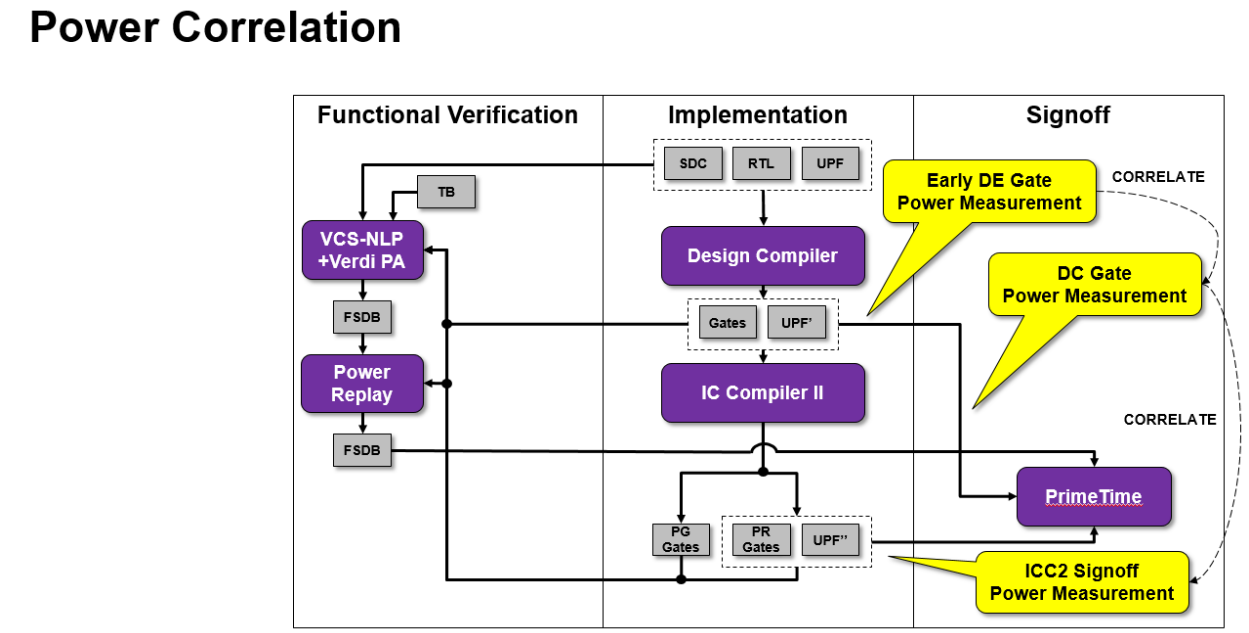


Figure 19.3 - Synopsys Power Correlation Flow

Power Measurement

For Bitcoin, we measured RTL power with SpyGlass, and used the following FSDBs as input for switching activity. The results are shown in Figure 19.4. Once we have different versions of the Bitcoin RTL, we could repeat these measurements and see if the relative power between version changes.

Bitcoin Power Measurements
SpyGlass PE

Tool	Input Data	Activity Data	Internal Power (W)	Switching Power (W)	Leakage Power (W)	Total Power (W)
SpyGlass	RTL	Vectorless	1.163	0.001	0.029	1.193
SpyGlass	RTL	RTL_NOUPF.fsdb	0.866	0.074	0.123	0.940
SpyGlass	RTL	RTL_UPF.fsdb	0.201	0.012	0.124	0.346

Use SpyGlass to measure and track EARLY RELATIVE POWER between RTL versions

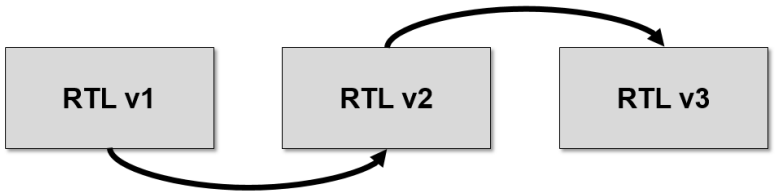


Figure 19.4 - Synopsys Power Exploration Flow Measurements

Lastly, we measure our "signoff" power with PrimeTime PX on the ICC II netlist. Here, we see the effect of using and not using UPF, and well as the variance when various FSDBs are used. We also see the varying results of using RTL FSDB vs. Gate FSDB vs. PR Gate FSDB. Our conclusions are that PR Gate FSDB are accurate enough for signoff, and hence we use it by default in our flow. Of course, in a "real" design, you should signoff with the most accurate data available, time permitting.

To correlate data with DC and other netlists from the same design version, we would repeat the measurements for a given FSDB over the many tools outputs of the netlist and correlate power in that way. That way, we could better understand the different implementation steps, and their overall effect on power.

Bitcoin Power Measurements
PT-PX

Tool	Input Data	Activity Data	UPF	Internal Power (W)	Switching Power (W)	Leakage Power (W)	Total Power (W)
PT-PX	ICC2 Netlist	Vectorless	No	0.2106	2.0876	0.6493	2.9475
PT-PX	ICC2 Netlist	RTL_NOUPF.fsdb	No	0.9960	1.4523	0.6954	2.2473
PT-PX	ICC2 Netlist	GLS_NOUPF.fsdb	No	0.1799	2.8466	0.6677	3.6942
PT-PX	ICC2 Netlist	PR_GLS_NOUPF.fsdb	No	0.1842	2.9537	0.6675	3.8055
PT-PX	ICC2 Netlist	Vectorless	Yes	0.2040	0.0435	0.5008	0.7484
PT-PX	ICC2 Netlist	RTL_UPF.fsdb	Yes	0.0266	0.3284	0.5249	0.8799
PT-PX	ICC2 Netlist	GLS_UPF.fsdb	Yes	0.0479	0.1589	0.5187	0.7246
PT-PX	ICC2 Netlist	PR_GLS_UPF.fsdb	Yes	0.0488	0.1409	0.5156	0.7054

Use PT-PX to measure and track CORRELATED POWER to Signoff

Figure 19.5 - Synopsys Power Correlation Flow Measurements

Chapter 19, Future Plans and Observations

Future Plans

Of course, with a Reference Design like this, one where so many tools are involved, it's impossible to get everything you want in one shot. We started out with grand ideas to implement all the latest UPF features and a very complex design, but because of time and resource constraints, we ended up with a subset of what we wanted.

The good news is that this Reference Design is a living kit, where we have plans to add new UPF and design features to it. Among those in consideration for future versions are the following:

- nwell
- Golden UPF
- Pads/Bumps
- Power Control Module
- Zero Pin Retention Flops
- UPF 3.0 PST (Power State Groups)
- DVFS/AVFS
- AMS
- Verification Plan
- IP considerations

We will work to get these incorporated in future revisions of this Reference Design Study. If you have comments or feedback, please let us know, we would appreciate your input.

Observations

Of course, in running a full flow with UPF, you're bound to experience "lessons learned" and things of that nature. To that end, we wanted to note a few things here.

Changing the source UPF, even with a minor switch, leads to a complete re-run of the entire flow

We painfully discovered this as we developed this Reference Design. If a tool downstream required a change to the source UPF, we pretty much had to start over. So, it makes a lot of sense to "pipe clean" your flow and agree upon a set of UPF constructs that work well throughout the flow, and flush them out accordingly, so that you don't have costly re-spins and re-runs late in the flow.

It makes sense to invest time up front with Static Checks

Running tools like SpyGlass and VC LP up front to take care of your source RTL, SDC, and UPF are invaluable in a Low Power Flow. We would even suggest that they are mandatory checks that should be run, since they allow the design teams to uncover issues very early in the design cycle.

Consider researching Power Measurement and the various ways it can be achieved

In a Low Power flow, the ultimate measure of your success can literally be measured in Watts. So, it makes sense for design teams to research and understand the power measurement methodologies. When to measure, with what switching activity to measure, with what representation to measure … these are all questions that should be asked up front, and agreed upon by the design team to ensure consistent power measurement over time.

NOR Isolation

For NOR-Isolation, the recommendation going forward is the following.

- isolation_supply_set is an empty list, namely {} or “”
- clamp_value is 0
- isolation_sense is high

For example:

```
set_isolation -domain PD_PISO \
  -isolation_supply_set {} \
  -clamp_value 0 \
  -elements {piso_bit/dout} \
  -isolation_sense high \
  -isolation_signal isolation_signals[1] \
  -location self \
  -name_suffix ISO_PISO_OUT_SSL \
  ISO_PISO_OUT_SSL
```

Note in the current Reference Design, we choose a different style of UPF because of limitations in the flow. This will be addressed in a future release of the flow and Reference Design.

Tool Dependencies

VCS NLP generates both FSDB and SAIF that are used for several tools in the flow. If you get errors where an FSDB or SAIF file is not found, then it’s likely you’ll need to run VCS NLP first. To run VCS NLP, use the following commands:

```
cd bitcoin_v1.1/tools/vcs_nlp
gmake
```

For PrimeTime PX, an FSDB from PowerReplay is required. And of course, PowerReplay requires an FSDB from VCS NLP in turn. To run VCS NLP and PowerReplay, use the following commands:

```
cd bitcoin_v1.1/tools/vcs_nlp
gmake
cd ../powerreplay
make bit_coin_upf
```

The following table shows the file dependencies between tools.

Source Tool	File Type	Target Tool
VCS NLP	RTL FSDB	SpyGlass PE (Power Exploration)
VCS NLP	RTL SAIF	Design Compiler
VCS NLP	RTL FSDB	PowerReplay
PowerReplay	PR Gate FSDB	PrimeTime PX

UPF Parsing Messages

In several of the tools that read in UPF as an input, you'll often see many parsing INFO and WARNING messages. We are working on ways to suppress and reduce the messages across the various tools in the flow.

Source and Tool Versions and Data

Many of the sections of this User Guide were done in parallel, so there may be some inconsistencies with the data presented in a section with the actual results from re-running the tools on the Reference Design or even between sections of this document. Note the source (RTL and UPF) and the tool versions used were in flux during the production of the Reference Design and testing through the Low Power Flow and Methodology, and this explains these differences. We will update the data and make it a point to have consistency throughout the document, as well as the Reference Design in the future. Note, the tool versions used may influence the actual data you see if you re-run the tools on the Reference Design, so pay attention to the recommended tool versions discussed earlier.

Intermittent Makefile Error

We did experience an intermittent Makefile error during our development. For both bit_slice and bit_coin, we sometimes see a FATAL during SpyGlass power_mem_reduction goal.

```
Goal power/power_mem_reduction     : run incomplete
=====================================================
    3 FATAL Severity Messages
    9 ERROR Severity Messages
    2 WARNING Severity Messages
    3 INFO Severity Messages

...

------------------------------------------------------------------
Report 'dashboard' created.
  File: <path>/tools/sg/bit_slice/outputs/bit_slice/html_reports/dashboard.html
Report 'datasheet' created.
  File: <path>/tools/sg/bit_slice/outputs/bit_slice/html_reports/datasheet.html
Report 'html' created.
  File: <path>/tools/sg/bit_slice/outputs/bit_slice/html_reports/goals_summary.html
make[1]: *** [bit_slice_power] Error 7
make[1]: Leaving directory `<path>/tools/sg'
make: *** [bs_static2] Error 2
```

There is no issue with SpyGlass here, it's purely some strange issue with the Makefile. If you experience this issue, simply use the commands below, which re-runs SpyGlass PE and all the subsequent tools included as a single target in the Makefile. For example:

```
# For bit_slice
% make bit_slice_cont

# For bit_coin
% make bit_coin_cont
```

PrimeTime read_vcd vs. read_fsdb

In the PrimeTime scripts, we use the command “read_vcd” to read in the PowerReplay FSDB, instead of the “read_fsdb” command. In our experiments, we noted that “read_vcd” was significantly faster and more memory efficient, hence we chose to use it for this Reference Design. Of course, we have filed the appropriate reports and will have this investigated and fixed in a future release.

PrimeTime runs on DC Netlists + UPF’

For PrimeTime, we can run on either the output of DC (netlist + UPF’) or the output of ICC II (P&R netlist + UPF”). In this Reference Design, we did experience some difficulty with the DC runs with PrimeTime, hence they are omitted in this version. We have filed the appropriate reports and will have this investigated and fixed in a future release.