Bitcoin - Case Study User Guide
Low Power Flow and Methodology
Golden UPF Flow

Version 1.1, October 2017

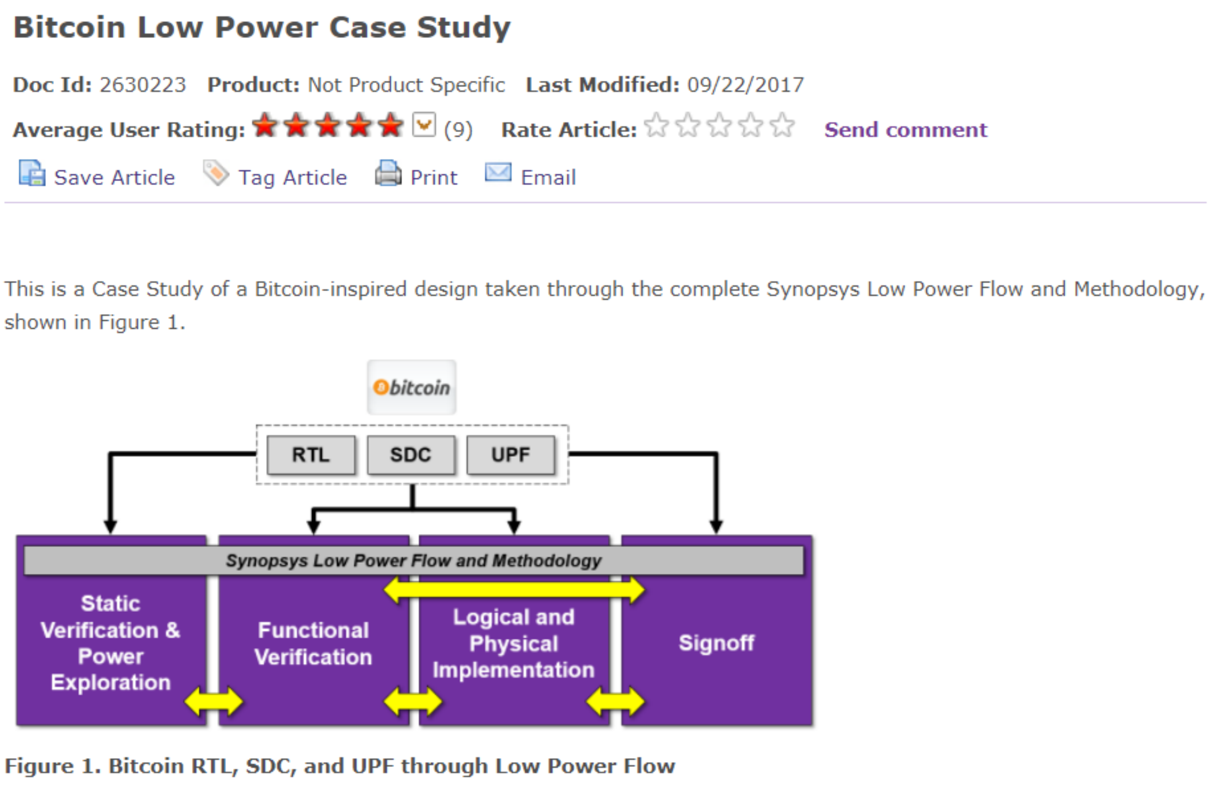
Copyright Notice and Proprietary Information

Table of Contents

Bitcoin – Case Study User Guide (Golden UPF)

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.



Bitcoin Low Power Case Study

Doc Id: 2630223 **Product:** Not Product Specific **Last Modified:** 09/22/2017

Average User Rating: ★★★★★ (9) **Rate Article:** ☆☆☆☆☆ **Send comment**

Save Article Tag Article Print Email

This is a Case Study of a Bitcoin-inspired design taken through the complete Synopsys Low Power Flow and Methodology, shown in Figure 1.

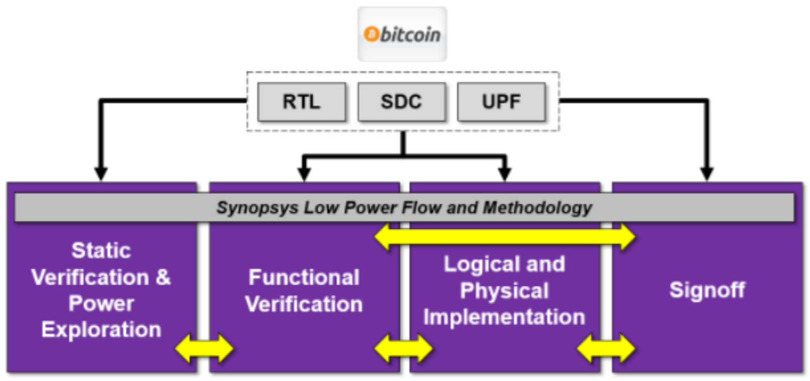

bitcoin
RTL
SDC
UPF
Synopsys Low Power Flow and Methodology
Static Verification & Power Exploration
Functional Verification
Logical and Physical Implementation
Signoff

Figure 1. Bitcoin RTL, SDC, and UPF through Low Power Flow

Version History

Version	Release Date	Comments
v1.0	March 2017	Initial Revision
v1.1	October 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.

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.

UPF Prime Flow Documentation

This "Golden UPF" document is meant as a supplementary document to the Bitcoin Case Study User Guide, which details work through the UPF Prime Flow. If you need detailed information about each of the point tools in the flow, including a technical overview and how to run the tools, the Bitcoin Case Study User Guide has a more complete set of information for you to review.

Note the Golden UPF Flow is not yet fully tested with "bitcoin", the larger design in this design kit. Only the smaller "bit_slice" design has been fully tested and run. We are currently working on the "bitcoin" design through Golden UPF Flow, and will update the documentation and design kit when ready.

In this document, we will focus on the Golden UPF Flow, and how to take the Bitcoin Reference Design through the tools with that in mind. As such, this document is organized as follows:

Chapters 1-3: Overview of the Reference Design and the Synopsys Low Power Flow and Methodology

Chapters 4: Step-by-Step example of "bit_slice", which is the basic building block of the bit_coin design. This illustrates the Golden UPF Low Power Flow and Methodology with a flat design.

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_gupf_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.

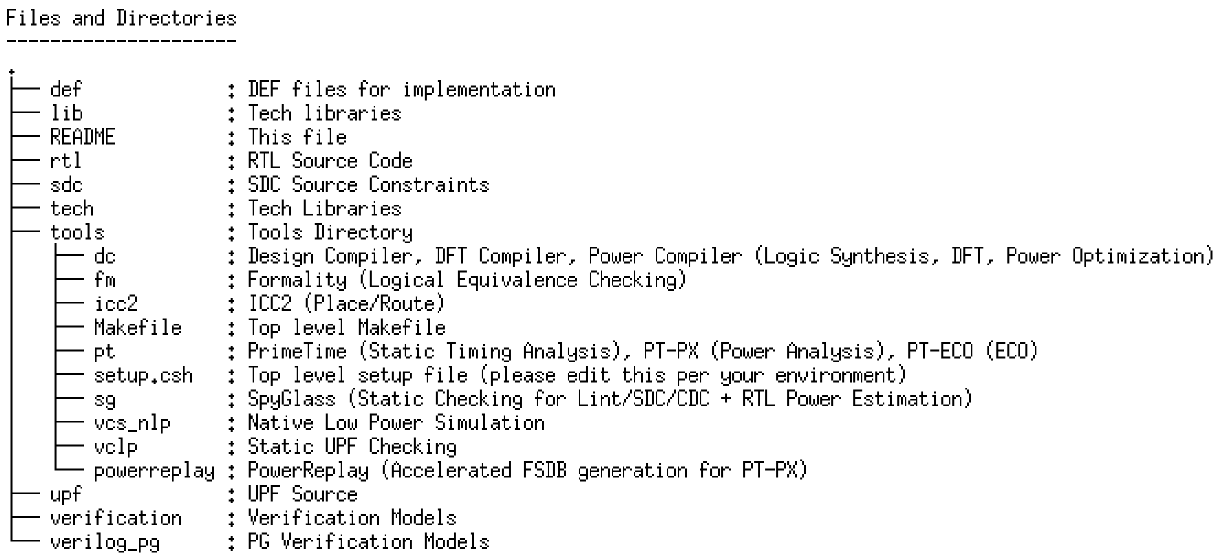
Files and Directories

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_gupf_v1.1.tgz
```

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

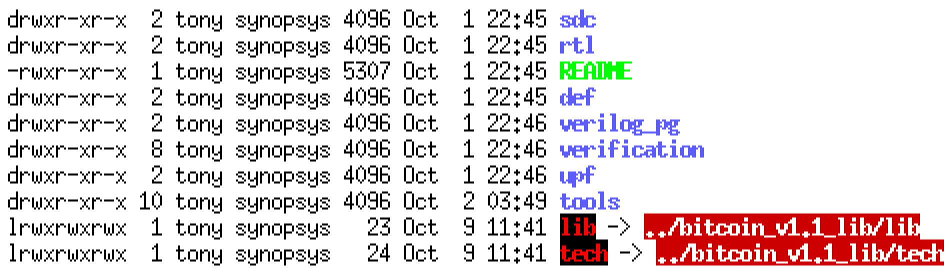


```
Files and Directories
---------------------

.
├── 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, DFT Compiler, Power Compiler (Logic Synthesis, DFT, Power Optimization)
│   ├── fm          : Formality (Logical Equivalence Checking)
│   ├── icc2        : ICC2 (Place/Route)
│   ├── Makefile    : Top level Makefile
│   ├── pt          : PrimeTime (Static Timing Analysis), PT-PX (Power Analysis), PT-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
│   └── powerreplay : PowerReplay (Accelerated FSDB generation for PT-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.



```
drwxr-xr-x  2 tony synopsys 4096 Oct  1 22:45 sdc
drwxr-xr-x  2 tony synopsys 4096 Oct  1 22:45 rtl
-rwxr-xr-x  1 tony synopsys 5307 Oct  1 22:45 README
drwxr-xr-x  2 tony synopsys 4096 Oct  1 22:45 def
drwxr-xr-x  2 tony synopsys 4096 Oct  1 22:46 verilog_pg
drwxr-xr-x  8 tony synopsys 4096 Oct  1 22:46 verification
drwxr-xr-x  2 tony synopsys 4096 Oct  1 22:46 upf
drwxr-xr-x 10 tony synopsys 4096 Oct  2 03:49 tools
lrwxrwxrwx  1 tony synopsys   23 Oct  9 11:41 lib -> ../bitcoin_v1.1_lib/lib
lrwxrwxrwx  1 tony synopsys   24 Oct  9 11:41 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_gupf_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-SP5
PrimeTime, PT-PX, ECO	2016.12-SP3
Formality	2016.12-SP5
ICC II	2016.12-SP5
StarRC	2016.12-SP3

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 (bit_slice) 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 tools.

For example, you can run the flow just for simulation, and do quick checks on how to run the basics of a tool, or to simply experiment.

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. For example, to run the entire flow for “bit_slice”, you can run the following commands.

```
% cd bitcoin_gupf_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.

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.

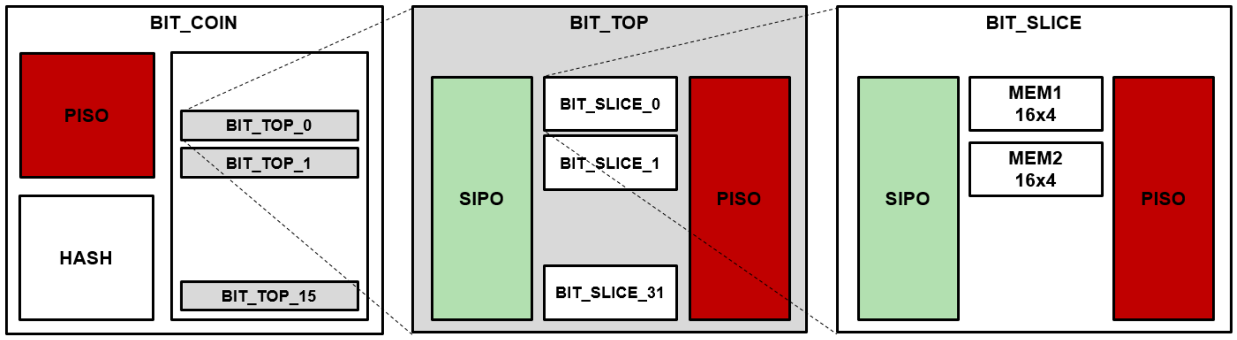


Figure 2.1 - Bitcoin Microarchitecture

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.

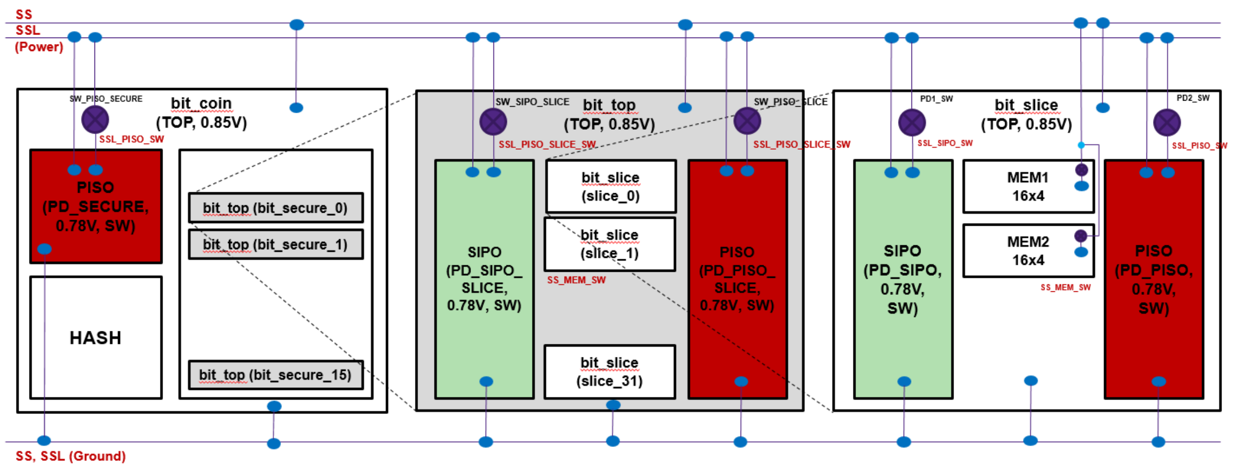


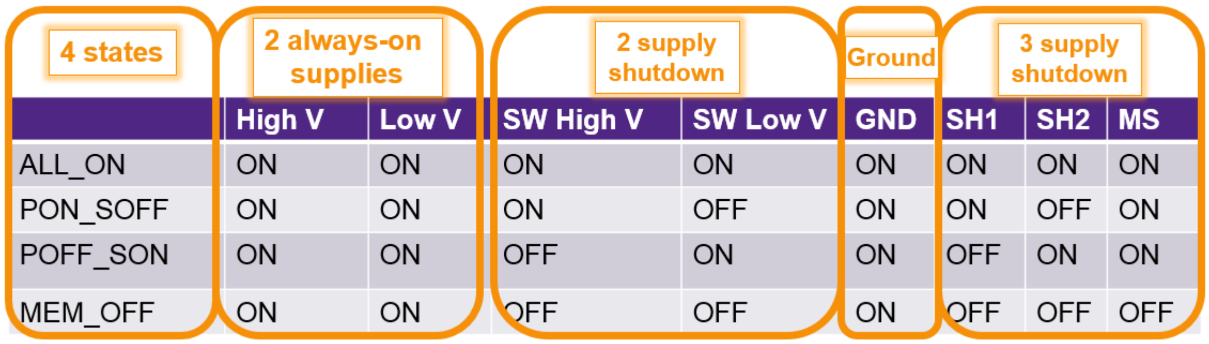
Figure 2.2 - Bitcoin Power Diagram

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. We will provide updates to the reference design and methodology periodically.

Chapter 3, Low Power Flow and Methodology

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

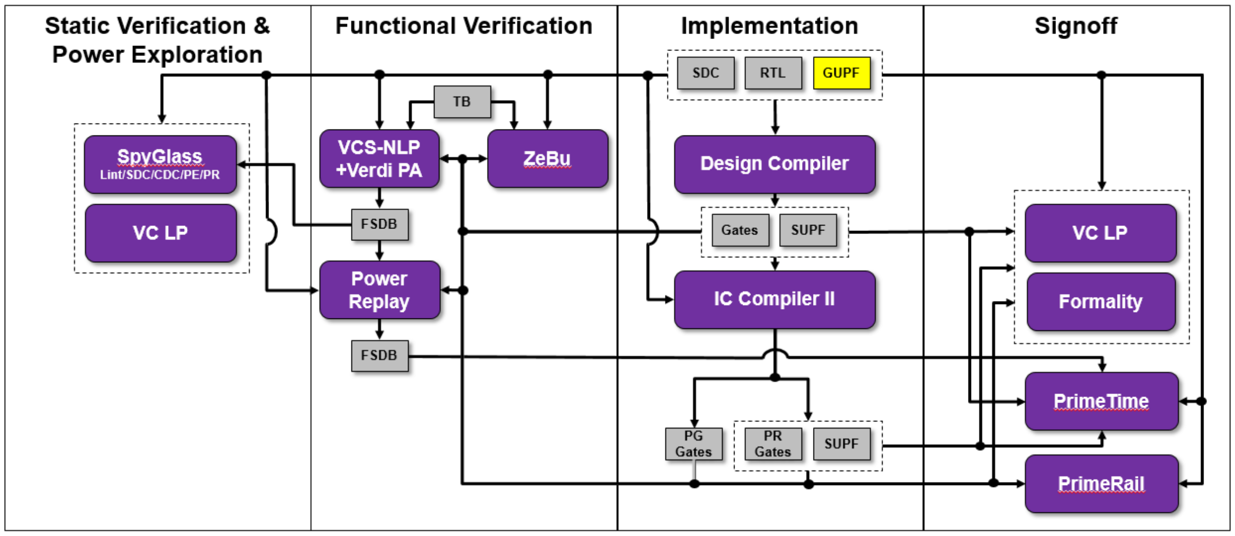


Figure 3.1 - Synopsys Low Power Flow and Methodology with Golden UPF

GUPF = Golden UPF, SUPF = Supplemental UPF

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 + GUPF 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, GUPF 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 GUPF as inputs. The resulting Gates + GUPF + SUPF are fed as input to IC Compiler™ II (ICC II), which performs placement and routing (P&R) and generates PG Gates + SUPF.

Signoff

In the fourth stage VC LP is called again to perform signoff checks on the Gates + GUPF + SUPF after synthesis, as well as PG Gates + GUPF + SUPF after P&R. Formality is also used to ensure functional equivalence of the RTL + GUPF vs. Gates + GUPF + SUPF and RTL + GUPF vs. PG Gates + GUPF + SUPF. 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, bit_slice

In this chapter, we will take the bit_slice block, a basic building block of the Bitcoin reference design, through the Full Low Power Flow and Methodology.

As mentioned in Chapter 1 “Running the Reference Design”, to run the bit_slice design, simply run the commands below, assuming you have installed the reference design and libraries.

```
% cd bitcoin_gupf_v1.1/tools
# modify setup.csh first to suit your env
% source setup.csh
% make bit_slice
```

This will the following tools in order:

```
SpyGlass Lint
SpyGlass SDC
SpyGlass CDC
VCS NLP @RTL (generated FSDB)
SpyGlass PE (uses FSDB from previous step)
VC LP @RTL
Design Compiler, Power Compiler, DFT Compiler
Formality @RTL2Gate
VC LP @Gate
VCS NLP @Gate
PowerReplay
IC Compiler II
VC LP @PG
Formality @RTL2PG
PrimeTime, PrimeTime-PX
```

In the following sections of this chapter, we will detail the work and experience with bit_slice through each of the tools described above, including the UPF and script changes necessary to implement the Golden UPF Flow. Note this is considered an “addendum” to the UPF Prime Flow, so it’s highly recommended to review the Bitcoin Case Study User Guide for the UPF Prime Flow first, before reviewing the following information.

SpyGlass Lint/SDC/CDC/PE

As of 2017.03-SP1, Spyglass supports UPF as LCA (limited customer availability). But, since we've run SpyGlass without UPF, the results here are the same as the UPF Prime Flow. For more information, please reference the Bitcoin Case Study User Guide.

The reports for bit_slice can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/sg/bit_slice/reports
  bit_slice_constraints_sdc_audit
  bit_slice_constraints_sdc_check
  bit_slice_constraints_sdc_exception_struct
  bit_slice_cdc_cdc_verify_struct
  bit_slice_power_power_est_profiling
  bit_slice_power_power_mem_reduction
  bit_slice_lint_lint_rtl
```

You can peruse the reports to check on the Lint, SDC, CDC, and PE checks done by SpyGlass.

VCS NLP @RTL

At the RT-Level, VCS NLP uses the same methodology as the UPF Prime Flow. So, the results here are the same as the UPF Prime Flow. For more information, please reference the Bitcoin Case Study User Guide.

The reports and outputs for VCS NLP @RTL can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/vcs_nlp/
  logs/bit_slice
    RTL_DEBUG_UPF
    RTL_DEBUG_NOUPF
  outputsfromvcs/bit_slice_RTL*
```

You can peruse the compile and run logs, as well the FSDBs generated with and without UPF.

VC LP @RTL

At the RT-Level, VC LP uses a similar methodology as the UPF Prime Flow, except for one command option, the -strict_check option from load_upf. Note the following below, where we enable strict checking for object naming mismatches between UPF file and objects in design.

```
load_upf ../../upf/logical.upf -strict_check true
```

The logs and reports for VC LP @RTL can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/vclp/
  logs/log.vclp.bit_slice
  reports/ report_lp.bit_slice.list.rpt
```

Reviewing the logs and reports, you’ll see that VC LP is clean for bit_slice @RTL.

Design Compiler, Power Compiler, DFT Compiler

For Design Compiler and Power Compiler, there are a couple of important options that need to be considered.

The first, a variable enables the Golden UPF flow within Design Compiler and Power Compiler.

```
set enable_golden_upf true # [default as false]
```

This variable is modified in the following Tcl file:

```
bitcoin_v1.1_gupf/tools/dc/scripts/dc_variables.tcl
```

The second, the "–supplemental" option for the "save_upf" command is required.

```
save_upf ${RESULTS_DIR}/${DESIGN_NAME}.upf

save_upf –supplemental ${RESULTS_DIR}/${DESIGN_NAME}.sup.upf  -
include_supply_exceptions
```

The logs, reports, and outputs for the DC, PwC, and DFT runs are found in the following directories:

```
bitcoin_gupf_v1.1/tools/dc/
  logs/log.dc.bit_slice_quick
  reports_bit_slice_quick/*
  results_bit_slice_quick/*
```

Note the DC results are copied to:

```
bitcoin_gupf_v1.1/tools/icc2/outputs2icc2
```

Formality @RTL2Gate

For Formality at RTL2Gate, there is one option to consider, namely the supplemental UPF needed for equivalence checking.

```
load_upf ../../upf/snps_${DESIGN_TOP}.upf \
-supplemental ../icc2/outputs2icc2/bit_slice.sup.upf \
-strict_check false \
-target dc_netlist
```

The logs and reports from Formality can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/fm
  logs/log.fm.bit_slice_quick_rtl2gate
  reports/bit_slice.fm.rtl2gate.rpt
```

Per the report, you'll see that bit_slice passes EC cleanly @RTL2Gate.

VC LP @Gate

For VC LP at the Gate or PG Netlist Level, there is one option to consider, namely the supplemental UPF needed for static UPF checking and the “-strict_check false” to avoid checking for objects at netlist level.

```
load_upf ../../upf/snps_bit_slice.upf \
  -supplemental ../icc2/outputs2icc2/${DESIGN_TOP}.sup.upf \
  -strict_check false
```

The logs and reports for VC LP @Gate can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/vclp/
  logs/log.vclp.bit_slice_quick_gate
  reports/report_lp.bit_slice_quick.gate.list.rpt
```

Reviewing the logs and reports, you’ll see that VC LP has a few errors and warnings related to NOR-ISO and heterogenous fanout and UPF Supply Undriven.

```
-----------------------------------------------------------------------------
ISO_STRATEGY_SOURCE  (1 error/0 waived)
-----------------------------------------------------------------------------
1.  Nor-style isolation strategy PD_PISO/ISO_PISO_OUT_SSL present, but strategy supply
    on when source supply off

-----------------------------------------------------------------------------
UPF_SUPPLY_UNDRIVEN  (1 error/0 waived)
-----------------------------------------------------------------------------
1.  UPF supply net VDDM does not have any driver
```

For the NOR-ISO error, the issue is actually because of VCS NLP, which does not yet take the recommended empty list “” as an option for set_isolation. Thus, the UPF is passed to DC, then to VC LP. To get around this, the UPF should have its isolation supply set set to “”, not SSL as shown below.

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

When VCS NLP supports this, which will be soon, this error in VC LP will go away. For the heterogeneous fanout, unfortunately, that’s a more difficult problem, and we’ll have to either waive those issues or leave them as false negatives at this point.

You can safely ignore the UPF SUPPY DRIVEN error, this is an artifact of the internal power switches of the memories.

VCS NLP @Gate

For VCS NLP at the Gate or Netlist Level, there are several options to consider, namely changes to the configuration options for VCS.

- -power_top → delete power_top add set_design_top bit_slice in GUPF (../../upf/logical.upf)
- -power_config → modify from power_config to -gupf_config
- --partcomp → deleted #partition compile is not supported with Golden UPF Flow
- -upf → delete –upf and add read_upf in gupf_config
- add, -scope, -target dc_netlist, -strict_check false options

In summary, the "read_upf" command is modified below in "tools/vcs_nlp/scripts/config_gls.tcl":

```
read_upf ../../upf/snps_bit_slice.upf \
  -supplemental ../icc2/outputs2/icc2/bit_slice_sup.upf \
  -scope bit_slice \
  -target dc_netlist \
  -strict_check false
```

The reports and outputs for VCS NLP @RTL can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/vcs_nlp/
  logs/bit_slice
    GLS_DEBUG_UPF
    GLS_DEBUG_NOUPF
  outputsfromvcs/bit_slice_GLS*
```

You can peruse the compile and run logs, as well the FSDBs generated with and without UPF.

PowerReplay

For PowerReplay, there are several options to consider in the setup and running of the tool, namely regarding the VCS setup.

First off, in the KDB generation step, you'll need to change the -power_config option to -gupf_config, just like VCS NLP.

```
vcs -kdb=keep_unresolved_db \
  -lca \
  -timescale=1ns/1ns \
  -sverilog \
  +define+UPF \
  -f scripts/files_bit_slice.vg.f \
  -gupf_config scripts/vcs_config.tcl \
  -l logs/2_bit_slice_gate_upf_kdb.log \
  -o gate_bit_slice_upf_simv
```

Likewise, in the configuration files for running VCS during the wisim replay, you'll modify similarly.

```
# tools/powerreplay/scripts/bit_slice.rtl2gate_upf.vcs_wi_compile.rc

vcs -sverilog \
  -f wi_run.f \
  -debug_access+f+w+cbk+fwn \
  -debug_region=cell+lib \
  +nospecify \
  -gupf_config ../scripts/vcs_config_wisim.tcl
```

```
# tools/powerreplay/scripts/vcs_config_wisim.tcl

set_design_attributes -attribute iso_nor TRUE
read_upf ../../../upf/snps_bit_slice.upf \
  -supplemental ../../icc2/outputs2icc2/bit_slice.sup.upf \
  -scope bit_slice \
  -target dc_netlist \
  -strict_check false
```

The logs, reports, and outputs from PowerReplay can be found in the following directory:

```
bitcoin_gupf_v1.1/tools/powerreplay/powrep_replayLog
```

And the correlation report can be found here:

```
bitcoin_gupf_v1.1/tools/powerreplay/powrep_map_summary.log
```

You can see the correlation percentage is 100% for all the “filtered” bits, which means the correlation is excellent for this run. The resulting PowerReplay FSDB (later used in PrimeTime PX) is here:

```
bitcoin_gupf_v1.1/tools/powerreplay/powrep_replayLog/wi_result.fsdb
```

IC Compiler II

For IC Compiler II, the main changes have to do with how the UPFs are read in and written out. On the input side, the Golden and Supplemental UPFs are added to the Reference Methdology (RM).

```
icc2_common_setup.tcl: set UPF_FILE
"../../../upf/comb_for_pp.upf"

icc2_common_setup.tcl: set VERILOG_NETLIST_FILES
"../outputs2icc2/bit_slice.vg"
```

Load the supplemental UPF accordingly:

```
set DC_SUPPLEMENTAL_UPF_FILE    "../outputs2icc2/bit_slice.sup.upf"
;# A UPF file from Design Compiler

load_upf -supplemental $DC_SUPPLEMENTAL_UPF_FILE $UPF_FILE
```

Add the following app_options to scripts/icc2_upf_vars.tcl

```
set_app_options -name mv.upf.enable_golden_upf -value true
```

With the above option, you can use the following command to save the UPF:

```
save_upf ${OUTPUTS_DIR}/${WRITE_DATA_BLOCK_NAME}.sup.upf
```

The logs, reports, and outputs can be found in the following directory:

```
bitcoin_gupf_v1.1/tools/icc2/bit_slice/
  logs_icc2
  rpts_icc2
  outputs_icc2
```

Note, in the outputs_icc2 directory, there are different “flavors” of netlist, UPF, etc. depending on the target signoff tool.

Also, note the ICC2 run for bit_slice is considered a “quick and dirty” run, as we take several shortcuts in the interest of runtime. This is not to be considered a fully clean place/route run.

VC LP @PG

For VC LP at the PG netlist level, there's a small change to read in the supplemental UPF from IC Compiler II.

```
load_upf ../../upf/comb.upf -supplemental \
  ${ICC_RESULT_DIR}/chip_finish.sup.upf \
  -strict_check false
```

The logs and reports for VC LP @PG can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/vclp
  logs/log.vclp.bit_slice_quick_pg
  reports/report_lp.bit_slice_quick.pg.list.rpt
```

Perusing the VC LP reports at the PG level, you'll see several errors from the Gate (NOR ISO + Heterogeneous Fanout), as well as some other errors. It's an exercise for the user to debug and review these error and warning messages.

Formality @RTL2PG

For RTL2PG verification, there’s a few small change needed to change the SVF directory pointer and then read in the supplemental UPF from IC Compiler II.

```
set_svf ../dc/results_bit_slice_quick
…
load_upf ../../upf/comb.upf
```

The logs and reports from Formality can be found in the following directories:

```
bitcoin_gupf_v1.1/tools/fm
  logs/ log.fm.bit_slice_quick_rtl2pg
  reports/bit_slice.fm.rtl2pg.rpt
```

Per the report, you’ll see that bit_slice passes EC cleanly @RTL2PG.

PrimeTime, PrimeTime-PX

For PrimeTime, there's a small change to set a variable to enable the Golden UPF Flow, as well as a change to read in the supplemental UPF.

```
set enable_golden_upf true

load_upf snps_bit_slice.upf_for_pt \
-supplemental ${ICC_RESULTS_DIR}/chip_finish.sup.upf
```

The logs and reports from PrimeTime and PrimeTime PX can be found in the following directories:

```
/bitcoin_gupf_v1.1/tools/pt
  logs
  reports_bit_slice
```

Perusing the logs and reports, you'll notice a few things. One, we use the "read_vcd" command to load in the PowerReplay FSDB.

```
# LOAD FSDB
read_vcd -strip_path bitslice_top/dut
../powerreplay/powrep_replayLog/wi_result.fsdb
```

Two, the annotation percentage is given below:

```
======================================================================
Summary:
Total number of nets = 580
Number of annotated nets = 544 (93.79%)
Total number of leaf cells = 390
Number of fully annotated leaf cells = 339 (86.92%)
======================================================================
```

Lastly, the total power consumed based on the given FSDB is 3.3 mW.

```
Total Power            = 3.340e-03  (100.00%)
```