Low Power Option of Ambit BuildGates Synthesis and Cadence PKS

Product Version 4.0.8
May 2001

Contents

Preface

This preface contains the following sections:

About This Manual

This manual describes how to run the Cadence® Low Power Synthesis (LPS) option for Ambit® BuildGates® Synthesis and Cadence® Physically Knowledgeable Synthesis (PKS). See *Chapter 5* of the Command Reference for Ambit® BuildGates® Synthesis and Cadence® PKS for details on the LPS text commands. To use this manual, you should be familiar with IC power consumption concepts and issues and with the BuildGates Synthesis software.

Other Information Sources

For more information about LPS and other related products, you can consult the sources listed here.

- *Command Reference for Ambit® BuildGates® Synthesis and Cadence® PKS*
- *Ambit® BuildGates® Synthesis User Guide*
- *Test Synthesis for Ambit® BuildGates® Synthesis and Cadence® PKS*
- *PKS User Guide*
- *Timing Analysis for Ambit® BuildGates® Synthesis and Cadence® PKS*

The LPS option and the BuildGates Synthesis software may be used with other tools during the design flow. The following documents provide information about these tools.

- Cadence® Silicon Ensemble™ Place-and-Route Reference

Syntax Conventions

This section provides the Text Command Syntax used in this document.

Text Command Syntax

The list below describes the syntax conventions used for the LPS documentation.

Important

Command names and arguments are case sensitive. User-defined information is case sensitive for Verilog designs and, depending on the value specified for the global variable `hdl_vhdl_case`, may be case sensitive as well.

`literal`	Nonitalic words indicate keywords that you must enter literally. These keywords represent command or option names.
`argument`	Words in italics indicate user-defined arguments or information for which you must substitute a name or a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument.
`[ ]`	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices from which you can choose one.
`{ }`	Braces are used to indicate that a choice is required from the list of arguments separated by OR-bars. You must choose one from the list.

```
{ argument1 | argument2 | argument3 }
```

One exception is when braces contain a instructions to provide more optional information for the argument. For example, this argument:

```
-instance {list_of_instances}
```

allows you to specify multiple instances. The list *must* be enclosed in braces, with instances separated by a space. Do not use braces if you only have one object to specify.

`...`	Three dots (...) indicate that you can repeat the previous argument. If the three dots are used with brackets (that is, `[argument]...`), you can specify zero or more arguments. If the three dots are used without brackets (`argument...`), you must specify at least one argument, but can specify more.
#	The pound sign precedes comments in command files and examples.

1 Introduction

This chapter contains background information about power consumption, an overview of the Low Power Synthesis (LPS) option of Ambit® BuildGates® Synthesis and Cadence® PKS, and information about launching the LPS option in both tools:

Power Issues in the Electronics Industry

With the remarkable success and growth of personal computing devices, consumer electronics, and wireless communication systems, there is an urgent demand is for high-speed computation and complex functionality. However, average power consumption has become a critical design concern. Because of shrinking chip size, the current density per unit area is also increasing. A related issue is the high cost of packaging and cooling strategies of high-performance processors, such as 500-1000 MHz clocks.

These issues shift focus from design performance and area concerns to a need for lower power and consideration for power dissipation. The challenge is to meet these needs without compromising overall chip performance. Power consumption in design modules must be identified and their power consumption must be reduced. These are the goals of LPS.

To summarize, these are a few of the reasons power has become a critical issue in chip design:

- Shrinking geometry, which in turn increases current density.
- Increasing clock frequency.
- Growing cost of packaging and cooling.
- Need for longer battery life for portable and handheld electronic devices.
- Necessity of integrating more functions to improve system performance.

Overview of the LPS Design Flow

Conventional low power design flow involves error-prone RTL modifications and time consuming iterations, as shown in (A) of Figure 1-1 on page 12. In the first step, designers manually create a power-conscious RTL code based on personal estimates. After synthesizing the RTL code, they try to minimize power in the design based on the given timing constraints. In the final step of estimating power, designers often discover that their power requirements are not met. They have to go back to the first, second, or third step to fix the problems.

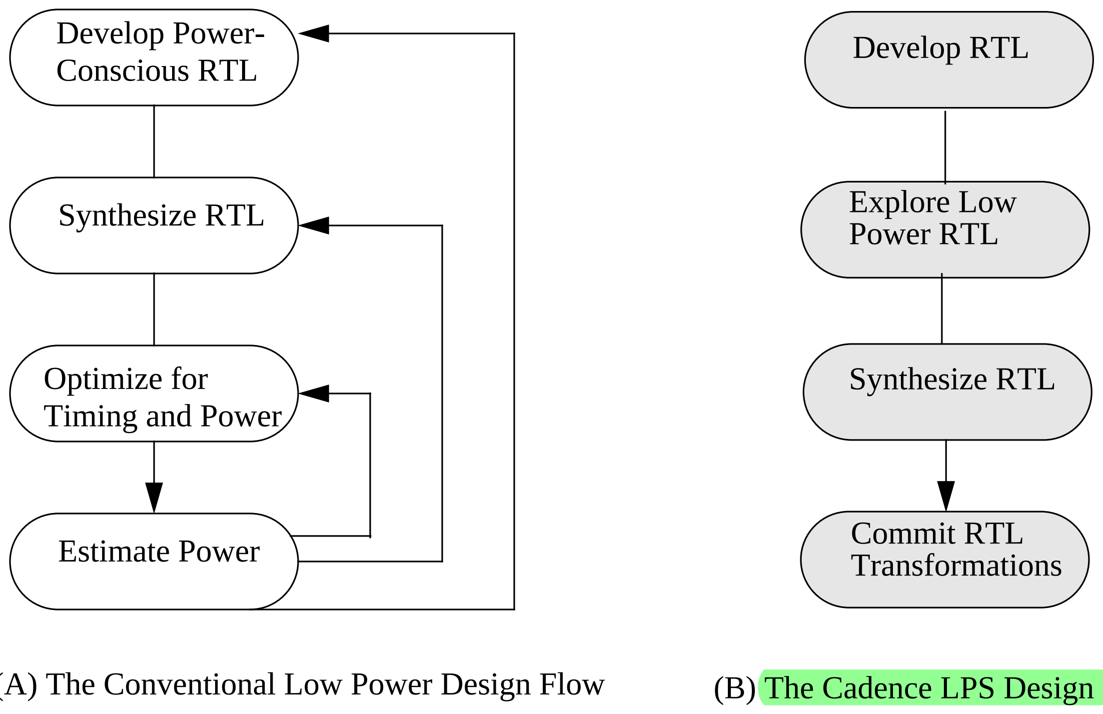

Develop Power-Conscious RTL
Synthesize RTL
Optimize for Timing and Power
Estimate Power
Develop RTL
Explore Low Power RTL
Synthesize RTL
Commit RTL Transformations
(A) The Conventional Low Power Design Flow
(B) The Cadence LPS Design Flow

Figure 1-1 The Conventional (A) Versus Cadence (B) LPS Design Flow

The Cadence LPS design flow provides both power estimation and optimization capabilities that also consider timing and area constraints. As shown in (B) of Figure 1-1 on page 12, designers begin with developing their RTL code. However, unlike the conventional low power design flow, designers do not have to worry about power in this step because the second step does the work for them. In the second step, LPS explores thousands of RTL transformations for reducing power in the design. Then, after synthesizing the RTL code, LPS minimizes the power by committing the RTL transformations and performing gate-level power optimizations while satisfying the timing constraints.

Design Stages of Power Reduction

There are distinct design stages where power can be reduced:

- System level

  The highest level in the design stage to address power reduction. One way to reduce power at this level is to:

  - ❑ Put the whole block of the design to sleep.

- Architectural level

  At this stage, you are looking at how to manage the system power. There are three ways you can do this:

  - ❑ Create a balance between extra flip-flops/muxes and through-put (pipelining).
  - ❑ Minimize fanout by weighing the use of a local bus versus a global bus.
  - ❑ Reduce the number of larger drivers across blocks by using local decoding.

- RTL and gate level

  There are two ways of addressing power at this stage:

  - ❑ Make a choice for state coding — gray coding versus binary coding.
  - ❑ Reduce the output glitch power by selecting either the Moore machine or Mealy machine.

- Physical analysis level

  You can still work on power reduction at this stage, but there is far less space for doing so.

At each of these design stages, you make trade-offs between accuracy and potential power savings. The following graphic illustrates these trade-offs at the various levels:

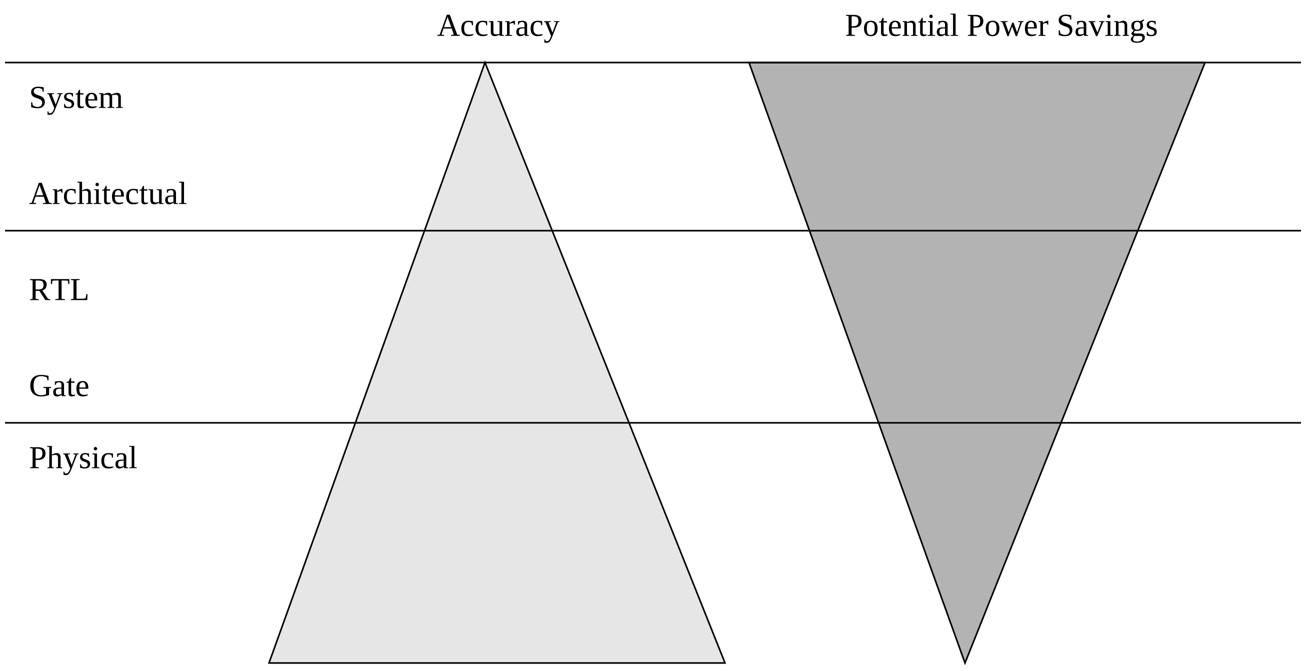


Figure 1-2 Power Reduction Trade-offs at Different Design Stages

The LPS option can be used during the RTL, gate, and physical levels.

Running the LPS Option

You can run the LPS option with the BuildGates Synthesis or Cadence PKS command line or their graphical user interfaces (GUIs).

LPS Option License

To run LPS option in the `ac_shell`, `pks_shell`, or GUI environments, you must first have an available `envisia_lowPower_option` license. Without an LPS option license, power commands and GUIs are not enabled.

Note: You must also have the BuildGates Synthesis or Cadence PKS license. The licenses are, respectively, `Ambit_BuildGates` and `Envisia_PKS`.

Using the `ac_shell` Command Line

In BuildGates Synthesis, enter the following command at the UNIX prompt to run the LPS option:

```
ac_shell -power
```

To run the LPS option in the Cadence PKS environment, enter the following command at the UNIX prompt:

```
ac_shell -pks -power
```

Note: The prompt changes to `pks_shell`.

For more information about using `ac_shell`, see the *Command Reference for Ambit BuildGates Synthesis and Cadence PKS*.

Using the Graphical User Interface

To launch the LPS option in BuildGates Synthesis from the GUI, enter the following command at the UNIX prompt:

```
ac_shell -gui -power
```

To run the LPS option in the Cadence PKS GUI, enter the following command at the UNIX prompt:

```
ac_shell -pks -gui -power
```

For more information about using the GUI, see the *Ambit BuildGates Synthesis User Guide*.

LPS-PKS Limitations

You can run all LPS commands and options in Cadence PKS, but, because LPS performs power optimizations based on the Wire Load Model (WLM), there are two post-placement command exceptions:

- You can not run the `do_optimize` command using the `-power` and `-pks` options at the same time. Each option works with this command on its own.
- The `do_xform_optimize_power` command does not have a `-pks` option to allow it to run within Cadence PKS.

Note: Commands that you *can* run after placement are the power analysis commands—`get_power` and `report_power`—and the power-conscious timing optimization command `do_xform_optimize_slack -power`.

2

Before Running the LPS Option

This chapter provides details the basics of power consumption and calculations , as well as library requirements for running the LPS option with BuildGates Synthesis and Cadence PKS.

Power Consumption

Power consumption for a circuit includes two general types of power: static and dynamic power.

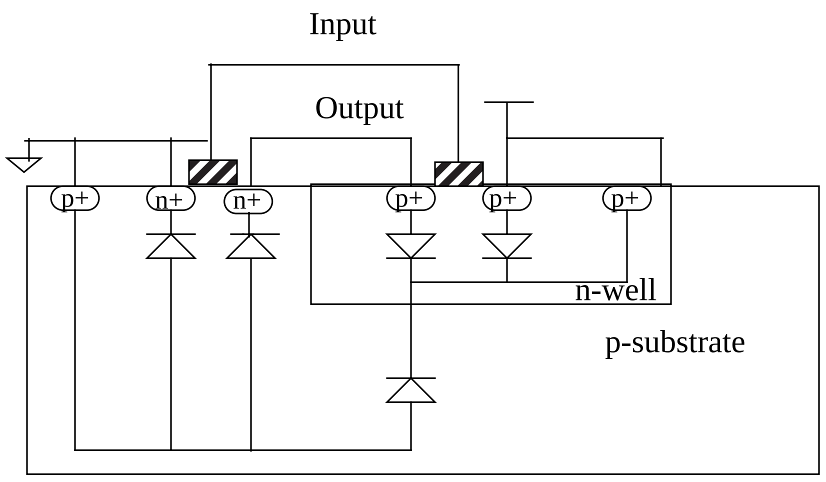
Static Power Dissipation

Static power is generally dependent on the state of the cell. Static power dissipation can be defined as power that is lost while a circuit's signals are not actively switching.

Static power dissipation includes

- Leakage power caused by sub-threshold leakage current in CMOS circuits

Figure 2-1 Leakage Power



  This is a characteristic of a technology cell, so leakage power values are obtained from the technology library.

  Also, LPS supports state-dependent leakage power, which is specified in the cell description section of your `.lib` file

- Standby power dissipation caused by the DC current being continuously drawn from power to ground

  This type of power dissipation is insignificant for CMOS circuits, so LPS ignores it.

Note: LPS currently works only with CMOS technology.

Dynamic Power Dissipation

The overall dynamic power of a cell is based on the effects of voltage and temperature, load, input slew rate, as well as the cell's state. Dynamic power dissipation is defined as power lost while a circuit is actively switching at a given frequency.

Dynamic power dissipation includes these two types of power:

- Short-circuit power

  Dissipated when a short exists between the voltage supply and ground rails created during output transitions that cannot be instantaneously turned on and off.

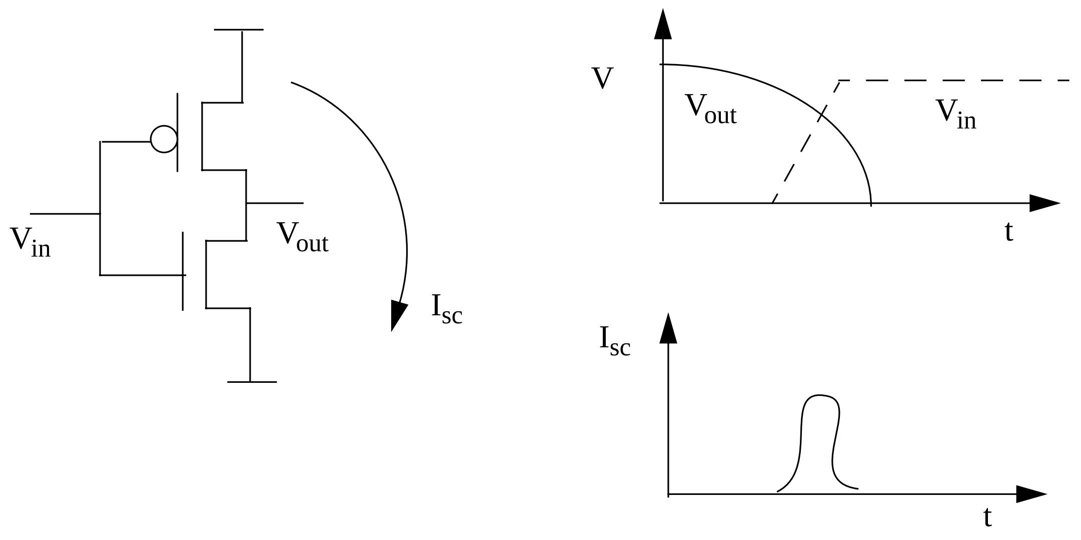


Figure 2-2 Short Circuit Power

  This is a subset of the broad category of internal power, which is defined as all power dissipated within a cell.

- Switching power (or capacitive load power)

  The power required to switch the entire load. The capacitive load includes internal capacitance, capacitance of the net, and the capacitance of the input pins. Power consumed by the internal capacitance is modeled as part of the cell power.

Note: The Cadence LPS design flow does not account for power consumption caused by a Partial Transition.

How LPS Models Power Dissipation

LPS uses the following power types to model power dissipation:

1. Leakage power
2. Internal power

   This consists of the short-circuit power and switching power of the internal cell.

3. Capacitive load power (net power) is calculated with the following equation:

   `1/2 CV`^{2}` * TR`

   Where:

   `1/2` is not dependent on the Duty Cycle.

   `C` is capacitance, with pico Farads as the default units.

   `V` is supply voltage, displaying in volts.

   `TR` is the Toggle Rate, which is stored as nanoseconds and outputs power results in milliwatts.

Note: LPS does proper scaling of the time unit. So, if you read in your first technology library with nano secondsas the default timing unit and then give a different unit for toggle counts in your TCF file, LPS scales the results.

Power Estimation

LPS estimates power at the gate level in your design, giving you the ability to characterize and evaluate various design alternatives. This means you can make design tradeoffs between performance and power.

When LPS calculates power estimates, it takes the following inputs into account:

- A TLF (`.tlf`) or ALF (`.alf`) file. See Ambit Library Format (ALF) File on page 108 for more information about ALF.

  See Supported TLF Power Statements on page 29 for more details about the supported power statements for the TLF file and Sample .tlf File Containing a Power Table on page 31 for an example of a power table section in a TLF file.

  See the *Timing Library Format Reference* for details on the power statements.

- Power models in the technology file (TLF or ALF)

  Currently, the table format model and constant values are supported.

- Slew from the timing analysis engine

- Net capacitance estimates from the BuildGates synthesis electrical system

- Switching activity on each net

  This could be a toggle count format (TCF) file (See Generating A Toggle Count Format (TCF) File on page 27 for more information) or using the probabilistic technique to calculate the activities (See The Probabilistic Technique For Computing Switching Activities on page 21 for more information).

How Power Estimation Works

Power estimation consists of two steps:

1. Obtaining Switching Activities on Nets
2. Performing Power Calculations

Obtaining Switching Activities on Nets

In this step, LPS reads in the net switching activities and stores them as assertions on nets. Assertions are toggle count values and probabilities taken from The TCF File. The switching activities of the new nets created during optimization are incrementally computed whenever the power of that net is required. See The Incremental Switching Activity Calculation on page 21 or The Probabilistic Technique For Computing Switching Activities on page 21 for more details on how switching activities are calculated by LPS.

The Incremental Switching Activity Calculation

LPS monitors changes in the circuit. Whenever there is a query on a net, LPS checks for an assertion and returns that value if the assertion exists. If an assertion does not exist for that net, LPS looks for a previously calculated value. If there is no value, LPS recalculates the value using the probabilistic technique. If there is an existing value, but the fanin cone has changed, causing an invalidation of the value, LPS recalculates the value.

The Probabilistic Technique For Computing Switching Activities

To compute the switching activity of an internal node, LPS uses the probabilistic technique for propagating the switching activities from the nodes containing the asserted values. If none of the logic nodes in the fanin cone contain assertion values, LPS assumes default values on the primary inputs and outputs of sequential elements. The default value of signal probability is 0.5, and the default value of the toggle count is half the clock frequency driving the node. If a node is not driven by a clock, the toggle count is assumed to be zero. The probabilistic technique takes Spatial Correlation into consideration; but because LPS does not propagate across sequential elements, Temporal Correlation is not considered.

Performing Power Calculations

The following equations demonstrate how cell-based internal power calculations are performed by the LPS tool:

P_{total} = Leakage Power(P_{leak})+Internal Power(P_{int})+Load Power(P_l)

Where:

$P_{leak} = \Sigma P_{leak}$ (cell)

$P_{int} = \Sigma P_{int}$ (cell)

Instance_level internal power

$P_{int}(cell) = \Sigma TR(Y) * \Phi(C(Y)_{,} S)$

Y is an output node of the cell

TR(Y) is the toggle rate of the Y output of the cell

C(Y) is the load capacitance of output Y

$\Phi(C(Y)_{,} S)$ is obtained from the lookup table in the technology library.

S is a statistical measure of the slew of the inputs causing a toggle on Y.

Path-Based Internal Power

LPS estimates the number of time each arc is activated based on the switching activity information of the input and output pins of the cell. It then calculates the dynamic power based on that and the power models in the library.

For example:

Rise_Power = Σ #transition ($xi^{\wedge}$->$y^{\wedge}$) * Power_table (xi^->y^, C_L, $Slew_{xi^{\wedge}}$) +
Σ #transition (xiv->y^) * Power_table (x_{iv}->$y^{\wedge}$, C_L, Slew xiv)

Fall_Power = Σ #transition (xi^->yv) * Power_table (xi^->yv, C_L, Slewxi^) +
#transition (x_{iv}->yv) * Power_table (x_{iv}->yv, C_L, $Slew_{xiv}$) }

Pin+ = Rise_Power + Fall_Power

The TCF File

A Toggle Count Format (TCF) file contains net or pin switching activities, as well as signal probabilities. The TCF file can be flat or hierarchical.

TCF File Syntax

In the TCF file, blocks delimited by { and } have the following form:

```
$BLOCK_ID ( "$BLOCK_NAME" )
"$PROPERTY_NAME" : "$PROPERTY_VALUE" ;
}
```

`$BLOCK_ID` can have the following values:

- `tcffile`
    - There can only be one `tcffile` in a TCF file and it must be the topmost block. You do not have to set the `$BLOCK_NAME` for the `tcffile` block.
    - `tcfversion` indicates the version of the format.
    - `generator` indicates the creator of the file.
    - `genversion` indicates the version of the creator.
    - `date` indicates day and time that the file was generated.
    - `duration` indicates the total simulation time.
    - `unit` indicates the simulation time unit.
- `instance` must specify either a flat or hierarchical instance. It must correspond to an instance in the netlist. For a hierarchical instance in the netlist, the `instance` block contains all the blocks for the instances within the hierarchical instance. The `$BLOCK_NAME` for an `instance` block is the instance name. Currently, there is no property defined for the `instance` block.
- `net` must specify a block for a list of nets of a hierarchical instance or of the top level module. It must contain a list of properties which correspond to the nets of a hierarchical instance or the top level module. The name of the property is the name of the net, and the value is the total toggle count and the signal probability.
- `pin` must specify a block for a list of pins of a flat instance or ports of a hierarchical instance. It must be contained within an instance block. The block has the properties of the pins of the instance. The name of the property is the name of the pin. The value is the total toggle count and the signal probability. For a hierarchical instance, the `pin` block contains the switching information for the ports of the instance.

Note: The hierarchical names are with respect to the module name provided in toggle.count.report. If no module name is provided, LPS points to what is specified in toggle.count.

Flat TCF File Example

By default, the creator of TCF file generates the hierarchical format. However it can also generate a flat version TCF file. A flat TCF file reports the net names and portnames with the absolute path from the top module. Unlike the hierarchical TCF file, the flat TCF file has only one `instance` block, one `net` block and one `pin` block. For the `pin` and `net` block, the pin name and the net name must be the full hierarchical name.

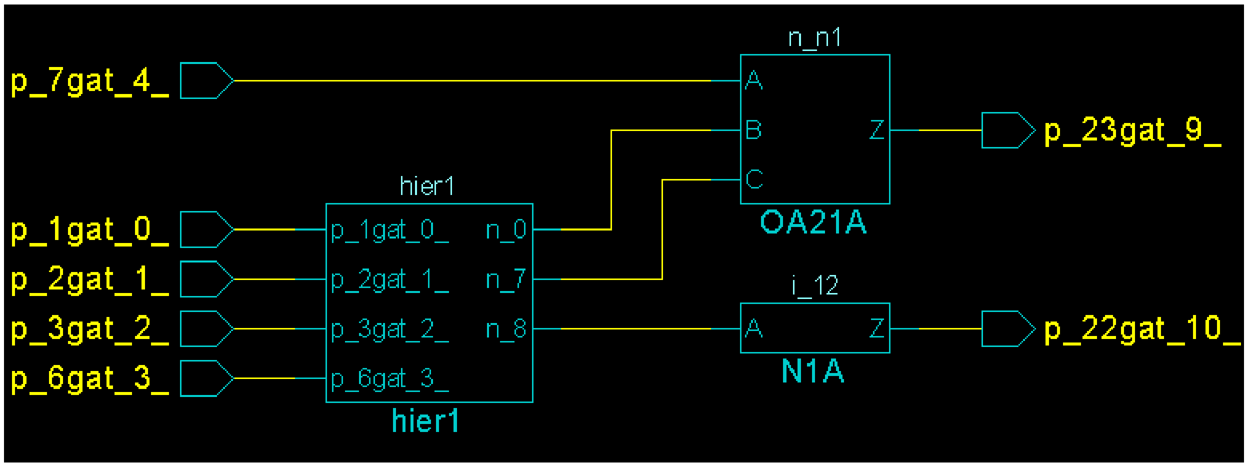
The following is the flat TCF file for the same design used in A Hierarchical TCF Example For a Gate-Level Netlist.

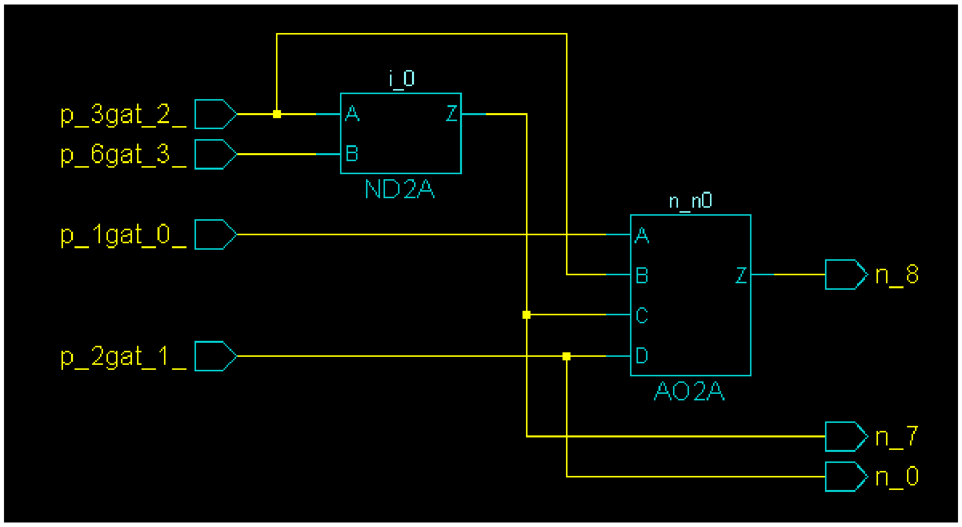
```
tcffile () {
   tcfversion    :       "1.0";
   generator     :       "BGPower Verilog PLI";
   genversion    :       "1.0";
   date          :       "Wed Aug  9 16:45:44 2000";
   duration      :       "1.501000e+05";
   unit          :       "ns";
   instance () {
     net () {
        "p_22gat_10_"   :        "0.587916  739";
     }
     pin () {
        "i_12/Z"        :        "0.566029  747";
        "n_n1/B"        :        "0.516522  475";
        "hier1/i_0/Z"   :        "0.773700  478";
        "hier1/n_n0/Z"  :        "0.433971  747";
        "hier1/n_n0/A"  :        "0.517588  521";
        "hier1/n_n0/D"  :        "0.516522  475";
        "hier1/i_0/A"   :        "0.487675  488";
        "hier1/i_0/B"   :        "0.492405  493";
        "n_n1/A"        :        "0.504664  506";
     }
   }
}
```

A Hierarchical TCF Example For a Gate-Level Netlist

A hierarchical TCF file reports the activity in the hierarchically. Each net or pin's activity is reported inside the block corresponding to the hierarchical instance it belongs to.

Following are graphics of a simple gate netlist followed by its corresponding hierarchical TCF file:





```
tcffile () {
   tcfversion    :       "1.0";
   generator     :       "BGPower Verilog PLI";
   genversion    :       "1.0";
   date          :       "Wed Aug  9 16:20:07 2000";
   duration      :       "1.501000e+05";
   unit          :       "ns";
   instance() {
     net() {
       "p_22gat_10_": "0.587916  739";
     }
     instance("n_n1") {
       pin() {
         "B"    :         "0.516522  475";
         "A"    :         "0.504664  506";
       }
     }
     instance("i_12") {
       pin() {
         "Z"    :         "0.566029  747";
       }
     }
     instance("hier1") {
       instance("i_0") {
         pin() {
           "Z"  :         "0.773700  478";
           "A"  :         "0.487675  488";
           "B"  :         "0.492405  493";
         }
       }
       instance("n_n0") {
         pin() {
           "Z"  :         "0.433971  747";
           "A"  :         "0.517588  521";
           "D"  :         "0.516522  475";
         }
       }
     }
   }
}
```

Generating A Toggle Count Format (TCF) File

You can generate a toggle count format (TCF) file by using a simulator with a VCD conversion script (as seen in (A) of Figure 2-3) or by using PLI routines linked to a Verilog simulator (as shown (B) of Figure 2-3).

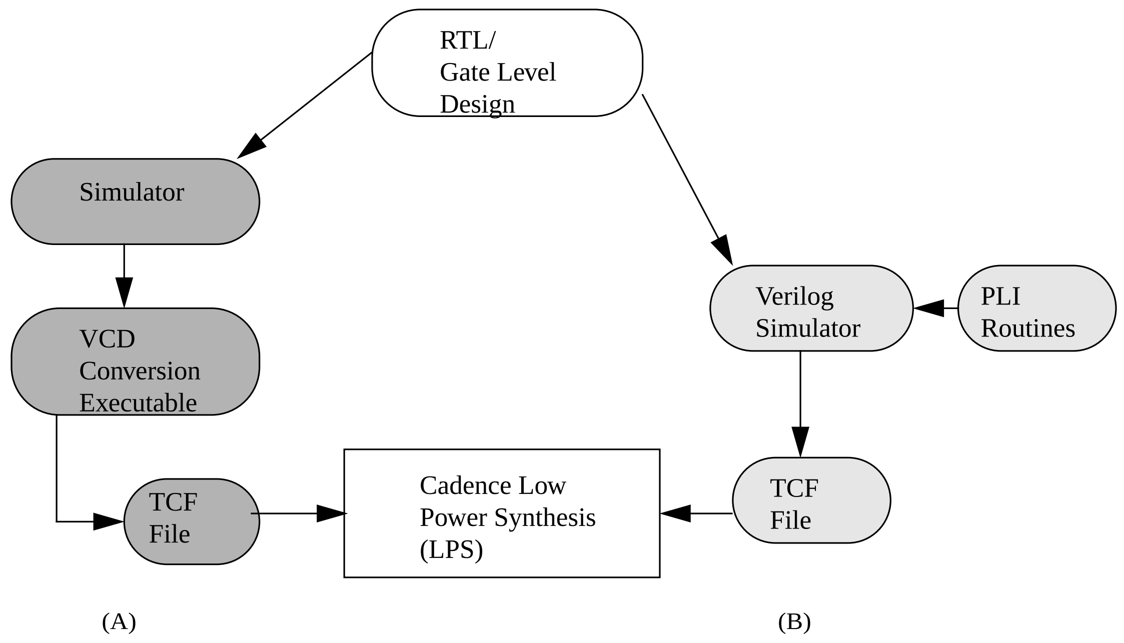
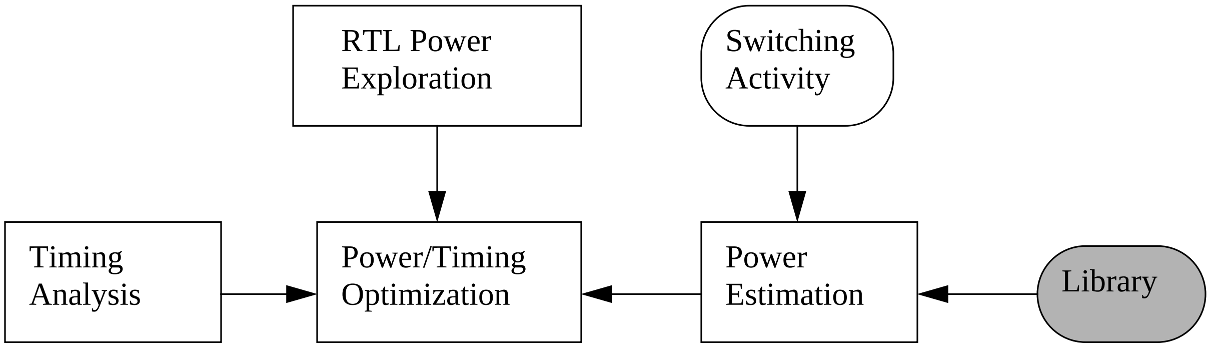


Figure 2-3 Methods of Generating a Toggle Count Format (TCF) File

See Step 4: Reading In a Toggle Count Format (TCF) File on page 83 for more details about each method for generating a TCF file.

Library Requirements



In order to perform power estimation for your design, you must provide the following:

- Synthesis libraries
  - ❑ A `.lib` or `.alf` file containing either a cell-based or arc-based power table

  A cell-based power table is described at the cell group level of the library. An arc-based power table describes power consumed by a transition on an arc. See Library File Examples on page 30 for further detail.

  Note: In addition to the traditional 2D lookup table support, LPS now has 3D (or 3rd dimension) library support which is the capacitance of an additional output pin. This output pin is described using the `equal_or_opposite_output` construct in your `.lib` file and the `OTHER_PINS` construct in your `.tlf` file.

  In the case of your `.lib` file, the three variables of the 3D power template are:

  ```
  input_transition_time
  total_ouput_net_capacitance
  equal_or_opposite_output_net_capacitance or total_output2_net_capacitance
  ```

  - ❑ A `.tlf` file containing an energy table.

  The energy model includes the following:

  ```
  INPUT_SLEW_AXIS
  LOAD_AXIS
  LOAD2_AXIS
  ```

  See Sample .tlf File Containing a Power Table on page 31 for details.

- A simulation library

  See Sample Simulation Library on page 34 for details.

Supported TLF Power Statements

Here is a list of power statements supported or not supported in TLF 4.3.

Power Statement	**Supported**
`CELL_SPOWER`	Yes
`PIN_POWER`	Yes
`VOLT_MULT_`*`power`*	Yes
`TEMP_MULT_`*`power`*	Yes
`PROC_MULT_`*`power`*	Yes
`SUPPLY_CURRENT`	No
`GROUND_CURRENT`	No
`WAVETABLE`	No
`SC_ENERGY`	Yes
`INTERNAL_ENERGY`	Yes
`TOTAL_ENERGY`	Yes
`VOLT_MULT_`*`parameter`*	Yes
`TEMP_MULT_`*`parameter`*	Yes
`PROC_MULT_`*`parameter`*	Yes

Table 2-1 Supported TLF Power Statements

Library File Examples

Sample .lib File Containing a Cell-Based Power Table

```
power_lut_template(pwr_lu_ramp_6_1.8200_fanout_10_3.8817) {
       variable_1 : input_transition_time;
       index_1 ("0.0050, 0.0150, 0.0450, 0.1350, 0.4050, 1.2150");
       variable_2 : total_output_net_capacitance;
       index_2 ("0.0038, 0.0076, 0.0152, 0.0304, 0.0607, 0.1214, \
                 0.2428, 0.4857, 0.9714, 1.9427");
}
cell ( AND2A ) {
     ...
     internal_power (pwr_lu_ramp_6_1.8200_fanout_10_3.8817) {
         related_outputs : "Z";
         related_inputs : "B A";
         values ("0.2697, 0.2688, 0.2703, 0.2833, 0.3031, 0.3488, \
                  0.4468, 0.6488, 1.0401, 1.8267", \

                 "0.5271, 0.5196, 0.5112, 0.5143, 0.5242, 0.5468, \
                  0.6283, 0.8204, 1.2051, 1.9851", \
                 "1.1475, 1.1400, 1.1217, 1.1050, 1.0918, 1.0979, \
                  1.1497, 1.3022, 1.6605, 2.4273")
     }
}
```

Sample .lib File Containing an Arc-Based Power Table

```
cell ( AND2A ) {
   pin ( Z ) {
   internal_power() {
     rise_power(POWER_DEFAULT_C_INT) {
         index_1 (" 0.2180,  0.8120,  1.2080,  1.6040,  2.0000 ");
         index_2 (" 0.0059,  0.0234,  0.0468,  0.0934,  0.1401,
                    0.1867,  0.2334 ");
         values ("0.04778, 0.05298, 0.05595, 0.05999, 0.06251,
                  0.06366, 0.06348  ",\
                 ...
                 "0.09101, 0.09485, 0.09742, 0.10114, 0.10362,
                  0.10495, 0.10516  ");
     }
     fall_power(POWER_DEFAULT_C_INT) {
       ...
     }
   related_input : "A" ;
   }
   ...
}
```

Sample .tlf File Containing a Power Table

```
CELL(NR2L
    ...
    ENERGY_Model(PowerRiseModel0
       (Spline
          (INPUT_SLEW_AXIS 0.218000 0.812000 1.208000 1.604000 2.000000)
        (LOAD_AXIS 0.005900 0.023400 0.046800 0.093400 0.140100 0.186700 0.233400)
           data
             (  (0.047780 0.052980 0.055950 0.059990 0.062510 0.063660 0.063480)
                ...
                (0.091010 0.094850 0.097420 0.101140 0.103620 0.104950 0.105160)
             )
          )
       )
       ENERGY_Model(PowerFallModel0
         ...
       )
       ...
       PATH(A => COUT 01 01 INTERNAL_ENERGY(PowerRiseModel0))
       PATH(A => COUT 10 10 INTERNAL_ENERGY(PowerRiseModel0))
       ....
       PATH(CIN => SUM 01 10 INTERNAL_ENERGY(PowerFallModel5))
       PATH(CIN => SUM 10 10 INTERNAL_ENERGY(PowerFallModel5))
    )
)
```

Sample .lib files Containing a 3D Power Table

Example 1

Note: In this example, `variable_3` in this sample could also be `total_output2_net_capacitance`.

```
power_lut_template(energy_template_7x3x3) {
     variable_1 : input_transition_time;
     variable_2 : total_output_net_capacitance;
     variable_3 : equal_or_opposite_output_net_capacitance;
     index_1 ("1000, 1001, 1002, 1003, 1004, 1005, 1006");
     index_2 ("1000, 1001, 1002");
     index_3 ("1000, 1001, 1002");
   }
internal_power() {
       ......
       rise_power(energy_template_7x3x3) {
         index_1 ("0.0500, 0.1000, 0.4000, 0.9000, 1.4000, 2.0000, 3.0000");
         index_2 ("0.00060, 0.15000, 0.51000");
         index_3 ("0.00060, 0.15000, 0.51000");
         values ( \
           "0.3635, 0.3765, 0.3545", "0.3823, 0.3705, 0.3475", "0.3820, 0.3785,
0.3475", \
           ...
           "0.7706, 0.7545, 0.7325", "0.7636, 0.7505, 0.7285", "0.7634, 0.7595,
0.7285");
       }
ENERGY_Model(PowerRiseModel2
       (Spline
          (INPUT_SLEW_AXIS 0.050000 0.100000 0.400000 0.900000 1.400000 2.000000
3.000000)
          (LOAD_AXIS 0.000600 0.150000 0.510000)
          (LOAD2_AXIS 1000.000000 1001.000000 1002.000000)
             data
                (
                  (.....)
                )
       )
    )
```

Example 2

In this cell example, the third axis is the capacitance of `QN`, which is specified using the `equal_or_opposite_output` format.

Note: The `equal_or_opposite_output` in this example is a construct of the `internal_power` group of the .lib file. In Example 1 on page 32, `equal_or_opposite_output_net_capacitance` was used, which is a construct in the `power_lut_template` definition in the `.lib` file. For more details, see the Synopsys Library Compiler documentation.

```
pin(Q) {
    direction : output;
    capacitance : 0.0;
    function : "IQ";
    internal_power() {
      related_pin : "CK";
      equal_or_opposite_output : "QN";
      rise_power(energy_template_7x3x3) {
        index_1 ("0.1000, 0.3000, 0.5000, 1.2000, 2.0000, 3.5000, 5.1000");
        index_2 ("0.00060, 0.15000, 0.51000");
        index_3 ("0.00060, 0.15000, 0.51000");
        values ( \
          "0.1151, 0.1117, 0.0944", "0.1140, 0.1058, 0.0884", "0.1137, 0.1123,
0.0874", \
          "0.1146, 0.1112, 0.0932", "0.1136, 0.1053, 0.0872", "0.1132, 0.1118,
0.0872", \
          "0.1138, 0.1105, 0.0927", "0.1128, 0.1046, 0.0867", "0.1125, 0.1111,
0.0867", \
          "0.1143, 0.1109, 0.0931", "0.1132, 0.1050, 0.0871", "0.1129, 0.1115,
0.0871", \
          "0.1143, 0.1108, 0.0937", "0.1132, 0.1050, 0.0867", "0.1128, 0.1114,
0.0867", \
          "0.1190, 0.1110, 0.0935", "0.1134, 0.1050, 0.0875", "0.1130, 0.1115,
0.0865", \
          "0.1193, 0.1111, 0.0936", "0.1134, 0.1051, 0.0876", "0.1131, 0.1121,
0.0866");
      }
```

Sample Simulation Library

```
‘delay_mode_path
‘suppress_faults
‘enable_portfaults
‘timescale 1 ns / 10 ps
‘celldefine

module ND2A ( Z, A, B);
output Z;
input A, B;
     parameter CMOS_TO_TTL = 0;
     parameter CLOAD$Z = 0;
     nand #(0.0, 0.0) ( Z, A, B);
specify
     ( A => Z ) = ( 0.0153:0.0190:0.0259, 0.0201:0.0250:0.0340 );
     ( B => Z ) = ( 0.0193:0.0240:0.0327, 0.0266:0.0330:0.0449 );
endspecify
endmodule

‘endcelldefine
‘nosuppress_faults
‘disable_portfaults
```

3 Using Power Optimizations

This chapter provides detailed background about how the various LPS optimizations work to minimize the power in your design.

Power Optimization

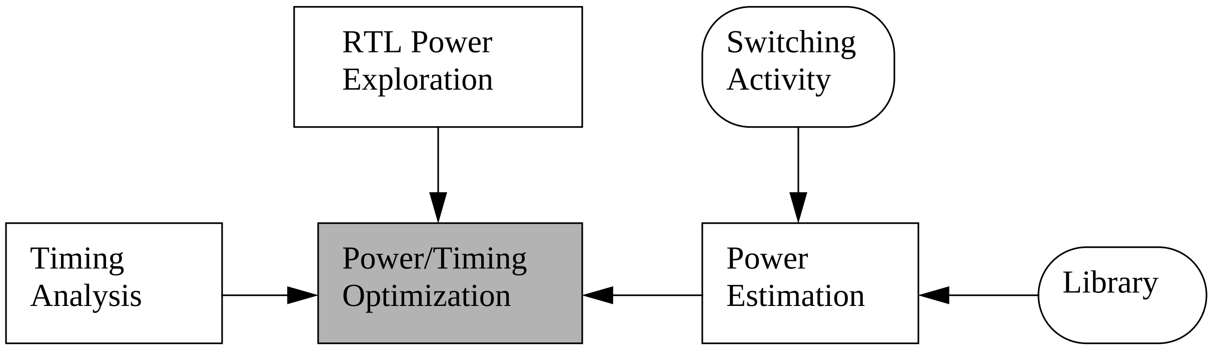


Figure 3-1 Power Optimization

The primary goal of the Low Power Synthesis (LPS) option is to minimize the power of the design without compromising the specified frequency and other design goals.

In addition to RTL power techniques, power optimization is performed on combinational logic blocks, with some techniques also targeted at reducing the power of sequential elements. See Gate-Level Power Optimization on page 51 for more information on these techniques.

Prerequisites

If switching activities are not provided for the primary inputs and register boundaries, default values are assumed. Though not required, switching activities of internal nodes (if present) also aid in optimization. The internal node switching activities can be calculated with the LPS option using a probabilistic technique.

Register-Transfer Level (RTL) Transformations

At the RTL, there is generally higher power savings than at the gate-level. RTL transformations can determine which part of a circuit is computing results that are used and which are not. These determinations are made in a given clock cycle. Any sequential or functional unit that is found not to be computing anything useful during a clock cycle can be shut off by either clock-gating or sleep-mode transformations.

Clock Gating

Data is loaded into registers very infrequently in many designs, but the clock signal switches at every clock cycle. This drives a large capacitive load, making clock signals a major source of dynamic power dissipation.

Gating a group of flip-flops that are enabled by the same control signal reduces unnecessary clock toggles.

Power consumption is lowered by using clock gating because

- Power is not consumed during the idle period because the register is shut off by the gating function.
- Power is saved in the clock circuitry.
- The logic on the enable circuitry in the original design is removed.

The clock gating technique consists of the following three steps:

1. Exploration

   Identifying periods of inactivity in the registers, as well as the set of registers for clock gating.

2. Insertion

   Putting in gating logic for clocks in the inactive periods identified during the exploration phase.

3. Commitment

   Committing gating logic based on power savings, performing timing checks for the setup and hold times of the gated clocks, and removing gating logic where timing constraints are violated or if there is no possible power savings.

Figure 3-2 illustrates how clock gating can help improve the circuit’s power.

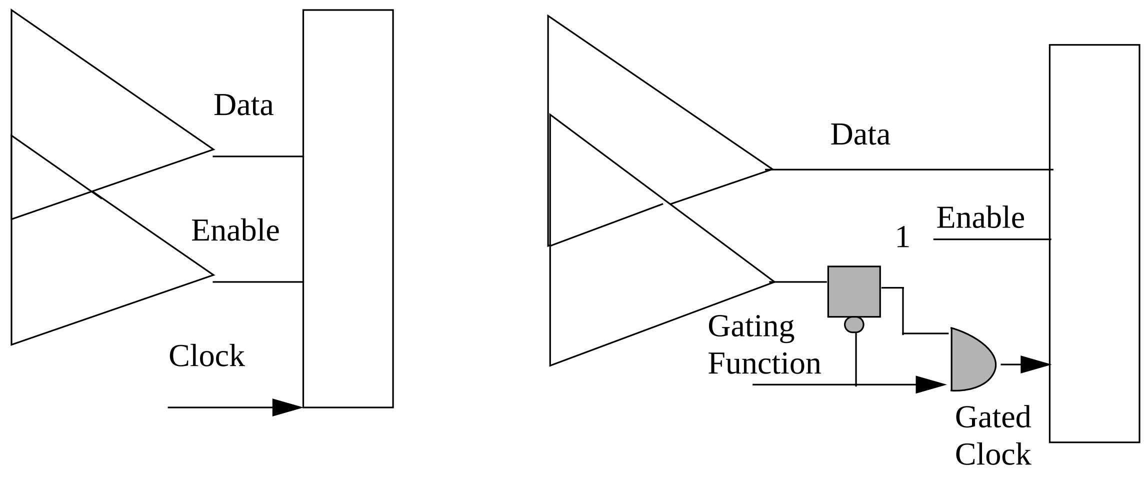


Figure 3-2 Clock Gating

During RTL exploration, LPS identifies all the clock-gating candidates and creates one gating logic per register bank. LPS also creates an empty module called `LPS_CG_EN_0` that has only one input port connected to the enable signal for the clock gating. This allows LPS to remember the clock-gating candidates and their corresponding enable functions after timing optimization. In this way, LPS is able find the gating logic and make decisions later about commitment based on timing and power information.

HDL Coding Style for Clock Gating

In LPS, registers that are conditionally loaded are considered for clock gating. This condition is used for gating the clock.

Examples of conditional statements and their impact on LPS clock gating follow.

Example 1

```
module ex1(in, out, en, clk);
input en, clk;
input [10:0] in;
output [10:0] out;
reg [10:0] out;
always @(posedge clk) begin
      if (en)
      out <= in;
end
endmodule
```

In Example 1, signal en is used for gating `clk`, but, if it is written differently, LPS will not insert clock-gating logic. See Example 1a.

Example 1a

```
module ex1a (in, out, en, clk);
      input in, clk;
      input [10:0] in;
      output [10:0] out;
      output [10:0] out;

      always @(posedge clk) begin
      if(en) begin
          out <= in;
      end else begin
         if (!en)
        out <= out;
         end
      end
      endmodule
```

In this example, the register `out` is always loading data, so LPS will not gate the clock.

Example 2a

Generally, LPS inserts gating on incomplete case statements. Cadence recommends not completing cases for unreachable states by using default statements.

```
module ex2(state, clk);
   input clk;
   output [2:0] state;
   reg [2:0] state;
   always @(posedge clk)
   case (state)
    3’b000 : state <= 3’b001;
    3’b001 : state <= 3’b010;
    3’b010 : state <= 3’b011;
    3’b011 : state <= 3’b100;
    3’b100 : state <= 3’b000;

   endcase
   endmodule
```

In this case, the registers are gated when the state belongs to `3’b101 - 3’b111`, but, if you make the case complete by adding `default construct`, LPS will not clock gate it.

Example 2b

```
module ex2(state, clk);
   input clk;
   output [2:0] state;
   reg [2:0] state;
   always @(posedge clk)
   case (state)
    3’b000 : state <= 3’b001;
    3’b001 : state <= 3’b010;
    3’b010 : state <= 3’b011;
    3’b011 : state <= 3’b100;
    3’b100 : state <= 3’b000;
    default : state <= state;
   endcase
   endmodule
```

In this case, LPS will not insert clock gating because it is a full case statement.

Gating Cell Criteria

The data input of the gating cell must meet the following criteria:

- It must be stable before the clock input makes a transition from the controlling value to the non-controlling value (setup time).
- It must be stable after the clock input returns to a controlling value (hold time)

Note: The setup and hold times can be changed using the `set_clock_gating_check` command. See Chapter 6 of the Command Reference for details.

Inserting Non-Latch Gating Logic

If you choose, LPS can insert non-latch gating logic when doing clock gating. Use the `set_clock_gating_options -no_latch` command. See *Chapter 5* of the Command Reference for details.

Clock Gating Multi-Clock Domains Under DFT Settings

Multi-Clock Domains

LPS supports design flows with multiple clock domains. Each clock domain refers to the logic in a given DFT clock domain. A DFT clock domain refers to all registers driven by the physical clock pin after being propagated through buffers, inverters, clock splitters (using the `set_dft_transparent` command), and any existing gated clocks (using the `test_mode signal` command set to bypass them).

Note: When there is a complex gate on the clock network, the clock is propagated through the gate if it reduces to a buffer or inverter under the DFT settings, such as `set_test_mode_setup` and `set_dft_transparent`.

Controllability

Adding gating logic to your design means the clock signal cannot be controlled for Test Synthesis, resulting in a potential reduced fault coverage for the design. The controllability feature of the LPS option improves the test coverage by adding controllability logic, as well as Observability logic.

Controlling Clock Domains

You can have different test control signals associated with each clock domain. There are two ways LPS does this:

- Automatically

  LPS creates test ports and connects these test ports to the corresponding controling portion of clock-gating logic.

- Manually

  You can use existing test control ports by assigning them to the clock domains, using the `set_test_mode_setup` command.

You can have an association by clock or test pin. If grouped by test pin, you could have more than one clock domain (`clk1`, `clk2`, ...) controlled by the same test pin. If grouped by clock, you have only one clock domain, such as `clk2`, controlled by one test pin.

Controllability Examples

Controllability and Observability Logic

The following flow:

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Set timing constraints
set_test_mode_setup Test_Mode 1
set_clock_gating_options -control Test_Mode
do_xform_optimize_generic -clock_gate
do_xform_map -hier
check_dft_rules
do_xform_connect_scan -hier
```

is represented in Figure 3-3 on page 43.

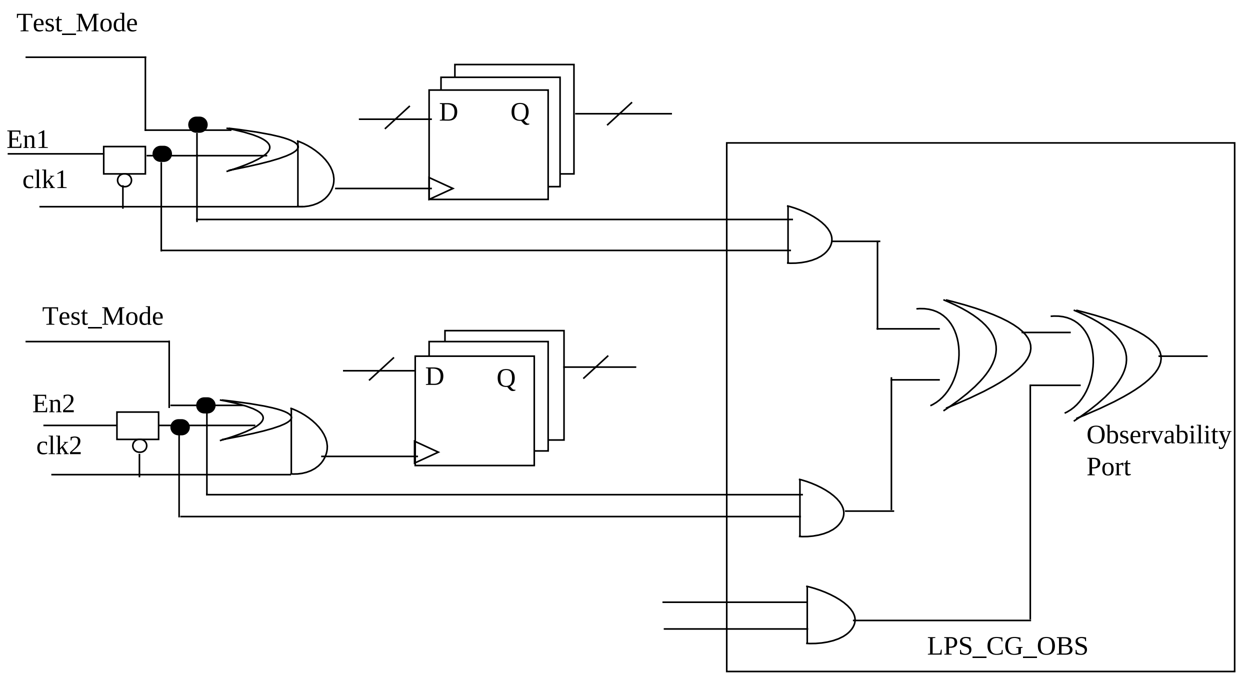


Figure 3-3 Controllability and Observability Logic

Different Test_Mode Ports for Different Clock Domains

```
set_test_mode_setup Test_Mode_1 1 -clock clk_1
set_test_mode_setup Test_Mode_2 1 -clock clk_2
set_test_mode_setup Test_Mode_2 1 -clock clk_3
```

See *Chapter 4* of the Command Reference for more information about using the `set_test_mode_setup` command.

Controllability and Observability on Different Domains

```
set_clock_gating_options -control -observe -domain [dft_doman | clock_net]
```
or
```
set_clock_gating_options -control Test_Mode_1 -observe -domain_all
```

In the second case, one test_mode port, `Test_Mode_1`, controls all domains.

See *Chapter 5* of the Command Reference for more information about using the `set_clock_gating_options` command.

Automatically Creating Test_Mode Ports

To set one test_mode port for all clock domains, use the `set_clock_gating_options` command this way:

```
set_clock_gating_options -auto_test_port -control
```

To create one test_mode port per clock domain, use the `set_clock_gating_options` command this way:

```
set_clock_gating_options -auto_test_port -control -domain clock_net
```

See *Chapter 5* of the Command Reference for more information about using the `set_clock_gating_options` command.

Glitch-Free Gating Logic

```
set_clock_gating_options -ctrl_before_latch -control Test_Mode_1
-domain [dft_domain | clock_net]
```

Note: You can also use the `-ctrl_before_latch` option with observability logic:

```
    set_clock_gating_options -ctrl_before_latch -control Test_Mode_1
    -domain [dft_domain | clock_net] -observe -obs_style register
```

Observability

Observability is possible through both ports and registers. The LPS option works with Test Synthesis to put registers into a scan chain and they are observed there, instead of adding more observability ports.

Creating Observability Registers

Use the following command and options to create observability registers:

```
set_clock_gating_options -control -observe -domain clock_net -obs_style register
-xor_depth
```

Note: Observability registers are only possible with `clock_net` set as the domain.

See Script Example of Creating Observability Registers on page 47 to see how observability fits into an LPS flow and see *Chapter 5* of the Command Reference for more details about using the `set_clock_gating_options` command.

Creating Module-Based Observability Registers

You can avoid the creation of an observability port in any hierarchical module by using the `set_clock_gating_options` command to create module-based observability registers, as seen here:

```
set_clock_gating_options -control -observe -domain clock_net
-obs_style reg_module
```

LPS completes the observability logic for an observability domain in the register of that module.

See Script Example of Creating Module-Based Observability Registers on page 47 to see how observability fits into an LPS flow and see *Chapter 5* of the Command Reference for more details about using the `set_clock_gating_options` command.

How LPS Makes Clock-Gating Latches Observable

There are many ways to make clock-gating latches observable. The two main methods LPS uses are:

- Adding observability ports

  New ports are created up to the top level module, which is connected to the output of the top of the xor tree. If the xor tree is the tree for everything, the observability logic is xored together.

  You can have just one observability port created for the whole design or have multiple ports created, where each port is associated with one clock domain or one control signal domain.

  You can do this with options included with the `set_clock_gating_options` command. See *Chapter 5* of the Command Reference for details.

- Adding observability flip-flops.

  Instead of creating new observability ports, flip-flops are connected to the output of the xor trees. So, when you add scan chains, these flip-flops become part of the scan chain. This means that the clock-gating logic is observable. The same clock driving the clock gating cells will also drive the newly-added flip-flops.

How LPS Traces Back the Clock Net

There are two ways in which LPS can trace back the clock net:

- Automatically

  The `set_test_mode_setup` command reduces the combinational logic to inverters and buffers under the dft test setting so that the clock net can be traced back.

- Manually, where you specify a way to trace back by using the `set_dft_transparent` command.

  You would typically use this method for tracing back the clock net if you have IP blocks or sequential modules in your design.

How LPS Inserts Registers

LPS inserts registers in these cases:

- If the xor tree has reached the depth as set by the `-xor_depth` option of the `set_clock_gating_options` command.
- When the xor tree has the depth less than the user-defined limit:
  - If the clock is not defined beyond that module.
  - At the top module if there are no more observability nets to be inserted.
- In the same hierarchy if `reg_module` is set.

If there are *n* observability points in an observability domain, then the depth is $\lg_2 n$. The depth or number of levels from the observability point to the output of the xor tree is bounded by the xor depth.

So, if your `xor_depth` value is set to 5 (the default), a register could be inserted at 4 if LPS cannot find a clock beyond the depth of 4.

Note: LPS tries to create a balanced binary XOR tree, so the depth is `lgn`.

Clock Polarity and Observability Registers

By default, the clock phase of observability registers created by LPS have the opposite clock polarity of the driving registers. If this is not the behavior you want, use the `-same_polarity` option of the `set_clock_gating_options` command:

```
set_clock_gating_options -control -observe -domain clock_net
-obs_style [register | reg_module] -same polarity
```

Observability Examples

Script Example of Creating Observability Registers

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Set timing constraints
set_test_mode_setup Test_Mode_1 1 -clock clk1
set_test_mode_setup Test_Mode_2 1 -clock clk2
set_test_mode_setup Test_Mode_2 1 -clock clk3
set_clock_gating_options -control -domain clock_net
set_clock_gating_options -observe -obs_style register
do_xform_optimize_generic -clock_gate
do_xform_map -hier
check_dft_rules
do_xform_connect_scan -hier
```

Script Example of Creating Module-Based Observability Registers

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Set timing constraints
set_test_mode_setup Test_Mode_1 1 -clock clk1
set_test_mode_setup Test_Mode_2 1 -clock clk2
set_test_mode_setup Test_Mode_2 1 -clock clk3
set_clock_gating_options -control -domain clock_net
set_clock_gating_options -observe -obs_style reg_module
check_dft_rules
do_xform_optimize_generic -clock_gate
do_xform_map -hier
do_xform_connect_scan -hier
```

Sleep Mode

Although clock gating is very effective in reducing the dynamic power dissipation of a digital circuit, it is restricted to saving power on sequential elements and clock circuitry only.

One such case of restricted power savings is shown in Figure 3-4 on page 48. Figure 3-4 (A) shows an RTL description of a simple digital system, where register C takes the result of the multiplier whenever enable is on. This means that the power dissipated or consumed by the multiplier is wasted because it keeps doing useless computations when it is the result of the register B that is needed by the output register. Also, because the multiplier is one of the most power consuming function units, the total amount of power wasted is quite significant.

Clock gating cannot be used in Figure 3-4 (A) because the output of input register B is always used by either the multiplier or the comparator. One solution to this problem is to shut down the function unit when its results are not used. To do this, LPS performs sleep mode analysis. As seen in (B) of Figure 3-4 on page 48, LPS inserted `AND` gates to stop the signal transitions at the inputs of the multiplier. The result is that no dynamic power dissipation occurs when result of the multiplier is not needed.

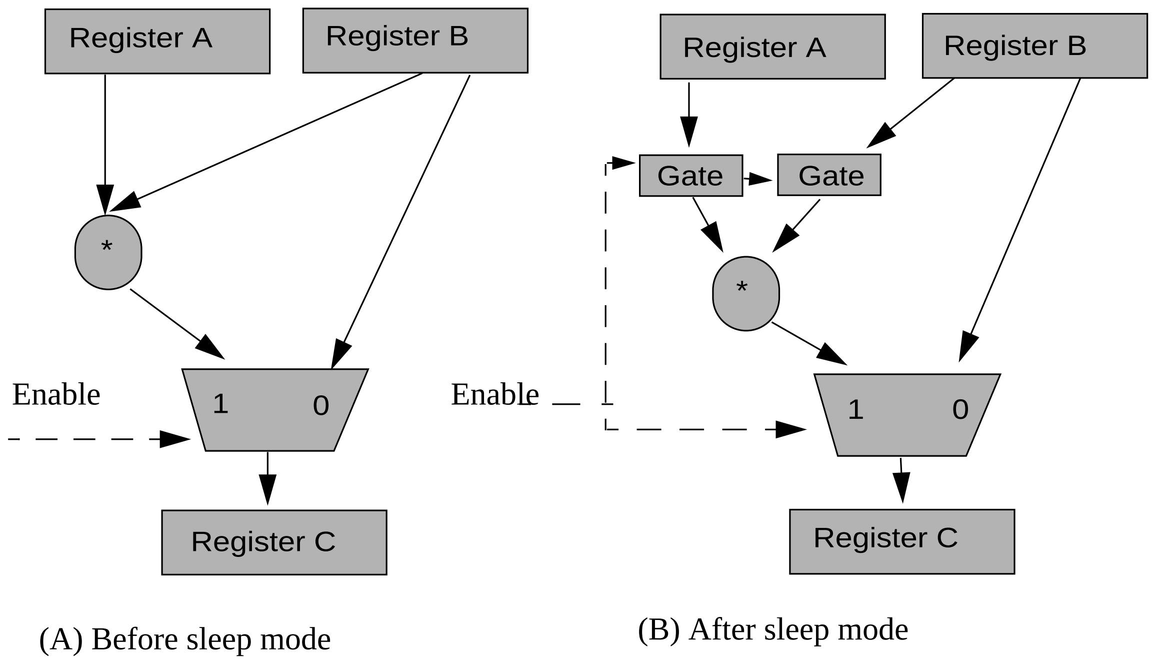


Figure 3-4 Sleep Mode Transformation

Sleep Mode Transformation Challenges and Benefits

The biggest challenge of RTL transformations is that they are best performed at the RTL, when accurate power and timing information is not available. Because of the inaccuracy of power estimation at the RTL, sleep mode transformations performed at the RTL may actually lead to a power increase of the whole chip. More importantly, performing sleep-mode transformations at RTL may result in a final gate-level implementation that violates timing constraints, even though the design meets timing without the transformations. Although the tool could undo the sleep mode logic, LPS would have been forced to work unnecessarily on the “wrong” critical paths and then make an extra effort to optimize the “real” critical paths after undoing the unneeded transformations.

The Sleep Mode Transformation Process

LPS uses a two-phase approach to solving the RTL transformation challenges:

- An exploration phase at the RTL level
- A commitment phase at the gate level

Sleep-Mode Exploration

During the exploration phase, a functional and structural analysis of the Control Data Flow Graph (CDFG) is performed to identify potential sleep-mode transformation candidates, including arithmetic units, functional blocks, and even entire logical hierarchical blocks. Control logic necessary for shutting down these candidates is also identified. Candidates and control logic are then “marked” on the gate-level netlist. No actual implementation is performed in this phase because inserting the logic at this phase might increase the timing or power. The commitment of these transformations is delayed until gate-level optimization has been done and accurate power and timing information is available.

Sleep-Mode Commitment

In the commitment phase, a decision of whether to commit each individual transformations is made based on whether the power is reduced *without* violating the timing constraints. If a sleep-mode candidate violates the timing constraints, LPS can modify the original control logic to partially commit the transformation to meet the timing constraints with some power savings, instead of blindly removing the gating logic without any power savings. This technique also saves runtime.

HDL Coding Style Examples for Sleep Mode

The following two writing style examples demonstrate how different HDL coding styles may produce different results in LPS.

Example 1

In this example, LPS determines that the condition for the results of the adder being useful is `(en[1] & en[0])` and the condition for the results of the subtractor being useful is also `(en[1] & en[0])`.

```
module a (en,in1, in2, in3, clk, out);
input clk;
input [1:0] en;
input [7:0] in1,in2, in3;
output [7:0] out;
reg [7:0] out,y;

always @(posedge clk)
  if (en[1]) begin
    y = in2 + in3;
    if (en[0])
       out = y-in1;
  end
endmodule
```

Example 2

For this example, due to the case statement, the condition for the results of the adder and the subtractor being useful is `en = 11` which is logically the same as in the Example 1.

```
module a (en,in1, in2, in3, clk, out);
input clk;
input [1:0] en;
input [7:0] in1,in2, in3;
output [7:0] out;
reg [7:0] out;
wire [7:0] y;

assign y = in2 + in3;
always @(posedge clk)
  case(en)
    2'b11 :  out = y-in1;
  endcase
endmodule
```

In Example 2, however, the condition `en[1]&en[0]` is taken directly from the corresponding logic in the netlist as it is expressed as a single entity in the case statement. In Example 1, the condition `en[1]&en[0]` is computed by LPS and the logic is stored in the extracted sleep mode module. As a result, LPS is able to do partial sleep mode commitment for Example 1, when it is required. Therefore, to benefit from the full power of LPS sleep mode optimization, Cadence recommends breaking down conditions in `if` statements if it is feasible to do so.

Gate-Level Power Optimization

During gate-level power optimization, transformations performed to minimize the power in your design are as follows:

- Gate Sizing on page 52
- Pin Swapping on page 53
- Buffer Removal on page 53
- Gate Merging on page 53
- Slew Optimization on page 54
- Logic Restructuring on page 55

These LPS optimization techniques work with the timing analysis engine to identify trade-off points in the power delay curve. When combined with the various effort levels, you can choose between faster runtime or a better quality result in terms of power.

Gate Sizing

Gate sizing is the process of changing the CMOS gate size, smaller or larger, so the total power is minimized without violating timing constraints. The goal is to find the gate size that consumes the least power. In general, reducing the size of a gate leads to a decrease in power and an increase in delay.

The overall internal cell power of a gate consists of:

- ❑ Leakage power
- ❑ Short-circuit power
- ❑ Power consumed by parasitic capacitance

Sizing the gate reduces the short-circuit power and power consumed by parasitic capacitance. Gate sizing could, however, increase the short-circuit power of the fanout logic because the slew of the output signal could increase when this transformation is performed. LPS makes tradeoffs between the gate sizing and the increase of power due to slew in order to optimally reduce the power of the design.

Figure 3-5 on page 52 demonstrates the importance of gate selection for gate sizing to maximize power savings.

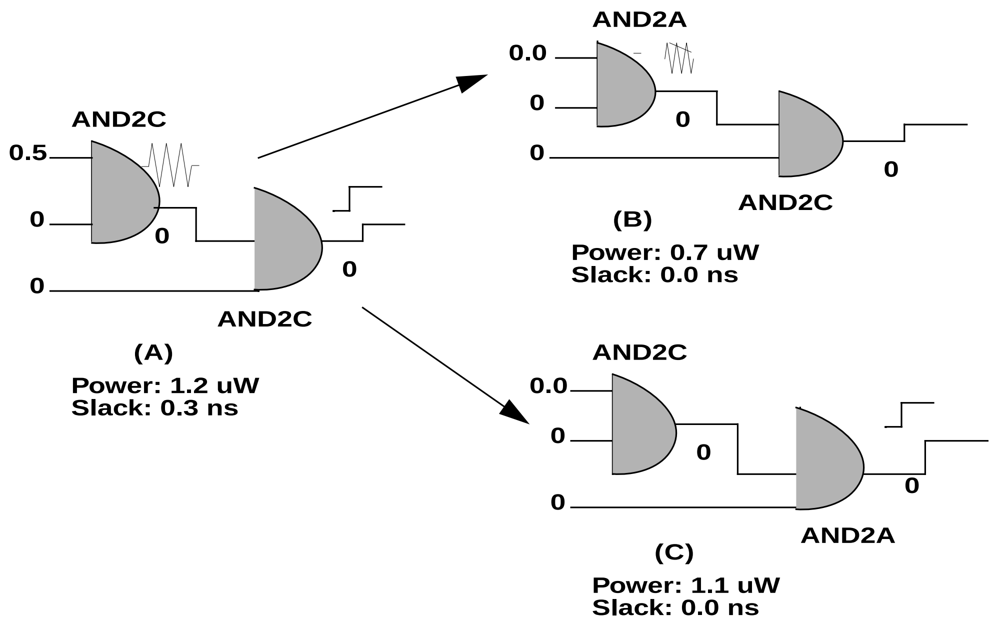


Figure 3-5 Gate Sizing

Pin Swapping

Pin swapping matches nets connected to the symmetric pins so power dissipation can be minimized. Pins with higher capacitance are matched with nets that have lower switching activity. Figure 3-6 on page 53 shows how pin swapping works.

Pin swapping can actually increase the delay of the design if not done carefully. LPS finds the optimal configuration that reduces power without affecting timing.

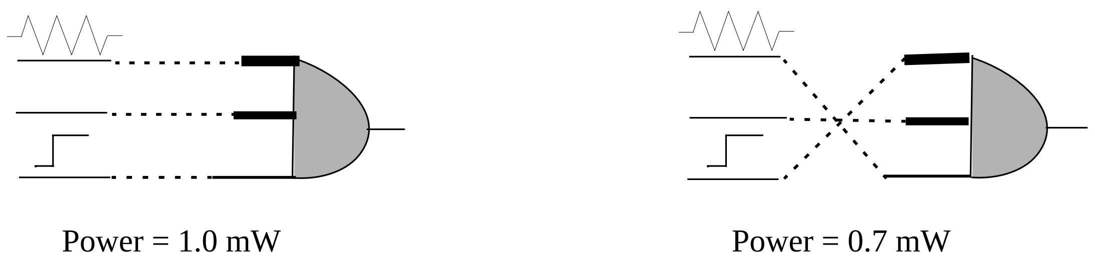


Figure 3-6 Pin Swapping

Buffer Removal

Sometimes timing optimization adds buffers to shield the critical path from a high capacitive load. During timing optimization, the critical path itself might shift. This results in unnecessary buffers on noncritical paths.

While these unnecessary buffers are removed as a postprocessing step, only a limited number can be removed without violating timing. So, in order to maximize savings for power, the power optimizer removes unnecessary buffers that drive highly active nets.

Note: The power optimizer selectively removes power-driven buffers that minimize the power without violating timing constraints.

Gate Merging

A major portion of dynamic power consists of driving the capacitive load of a net. A significant amount of that power can be saved by dissolving the net because the dynamic power of a net is proportional to the switching activity of the net. This savings is achieved by merging the gates on either side of the net. See Figure 3-7 on page 54 to see an example of gate merging.

Note: Although gate merging eliminates the power loss on the net, it could increase the internal power of the new gate. LPS makes the correct tradeoffs.

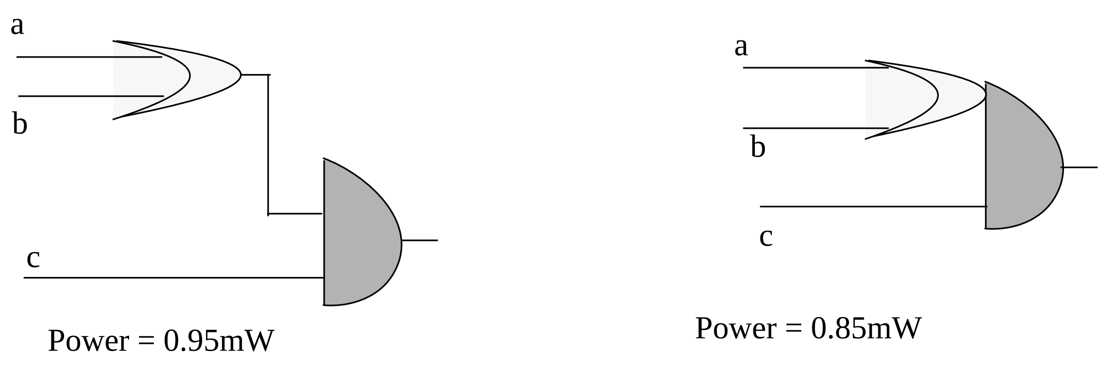


Figure 3-7 Gate Merging

Slew Optimization

The internal power of a cell depends largely on the slew of the input signals. During that time, power is drained from the design power rails to ground. If a signal has a large slew, driving the signal through a buffer improves the signal's slew and reduces the overall power of the cells driven by the signal.

See Figure 3-8 on page 55 for an example of how slew optimization works during timing-constrained power optimization. LPS has the capability of reducing slew. The challenge is in choosing the set of gates to be buffered. LPS performs a detailed analysis to identify the optimal partition that will reduce power as much as possible without violating timing.

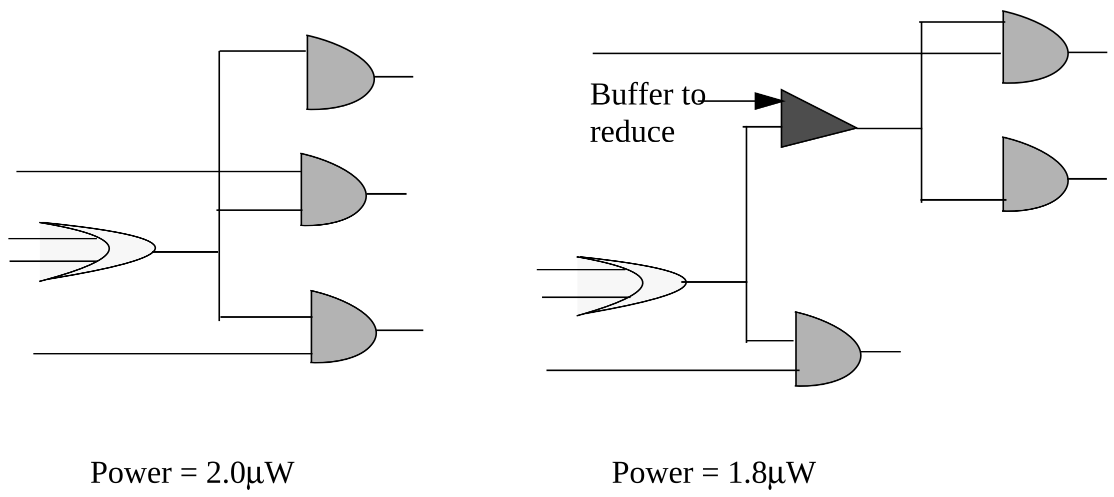


Figure 3-8 Slew Optimization

Logic Restructuring

Logic restructuring is different from other transformations performed during power optimization. Unlike gate merging, buffer removal, and other small local transformations that do not make significant changes to the circuit structure, logic restructuring actually changes the topological structure and internal functions of a circuit without compromising the circuit's functional integrity.

Figure 3-9 on page 56 shows a circuit before logic restructuring. Signal A has the lowest switching activity among all inputs. Although the final output of the circuit has a low switching activity, the two internal nodes are highly active, causing the bulk of switching power dissipation.

Figure 3-9 Circuit Before Logic Restructuring

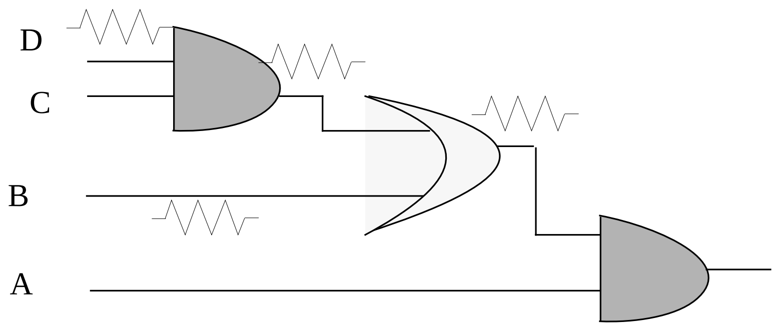


Figure 3-10 on page 56 shows how logic restructuring can find a more efficient circuit implementation that cannot be done by other power optimization transformations. The new circuit dissipates much less power than the original circuit shown in Figure 3-9 because the logic functions at the two internal nodes are now less active.

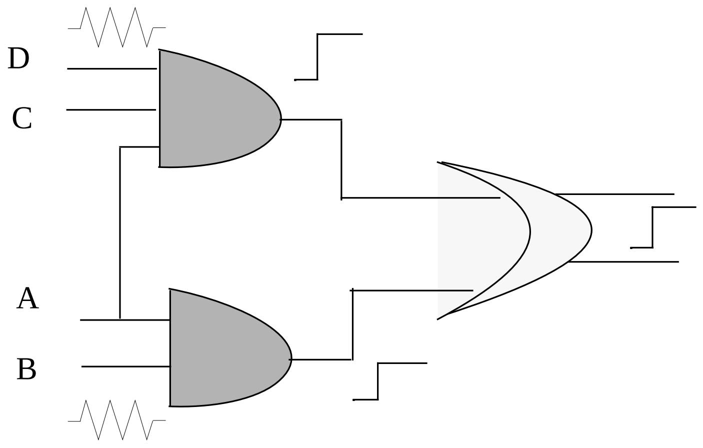


Figure 3-10 Circuit After Logic Restructuring

Power Analysis

LPS has both a text mode and GUI mode of power analysis.

Text-Mode Power Analysis

There are two LPS commands for reporting power analysis:

- `report_power`
- `get_power`

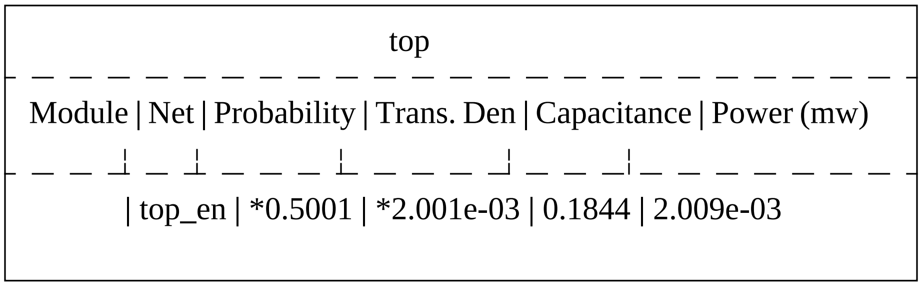
See *Chapter 5* of the Command Reference for syntax details.

Example of a Text-Mode Power Report

If you entered the following command syntax:

```
report_power -net [find -net top_en]
```

Your power report would look something like this:



```
                         top
------------------------------------------------------------
Module | Net | Probability | Trans. Den | Capacitance | Power (mw)
------------------------------------------------------------
       | top_en | *0.5001 | *2.001e-03 | 0.1844 | 2.009e-03
```

Figure 3-11 Sample Power Report

Where:

- `Module` is the module or instance name.
- `Net` is the name of the net.
- `Probability` is the probability of the named net being logic ‘1’ during the simulation capture period.
- `Trans.Den.` is the toggle count during the simulation capture period (transition density).

- `Capacitance` is the same capacitance unit as specified in your library.
- `Power` is the power consumption, represented in milliwatts (mW).
- The asterik (`*`) denotes that the value comes from your TCF file. All other values (without an asterik) are calculated by LPS.

GUI-Mode Power Analysis

The LPS GUI gives you:

- Colorized module and schematic browsing
- Schematic queries for instances and nets
- Pie chart displays for instances and nets
- Physical power map displays
- Textual power reports
- Power annotation on instances and nets

See *Chapter 4* of this manual for GUI command details.

Example of a GUI-Mode Power Report

To obtain a power report on a net using the GUI,

- Click the right mouse button on one net in the Schematic View of your design.
- Select *Power* from the pulldown menu.

```
Net power (top_en): prob (0.50015*), toggle rate (0.00200*), cap (0.18440), power
(2.010000uW) <NAVIGATES-106>
```

Figure 3-12 Sample GUI Power Analysis Report

Where:

- The *prob* is the probability of the net being logic ‘1’ during the simulation capture period.
- The *toggle rate* is the toggle count per simulation capture period. Also called transition density.
- The *cap* is the capacitance, specified in the same unit used in your library.
- The *power* is the power consumption calculated by LPS.
- The asterik (`*`) denotes that the value comes from your TCF file and is not calculated.

4

LPS Graphical User Interfaces

This chapter provides an overview of the graphical user interfaces (GUIs) for the Low Power Synthesis option of Ambit® BuildGates® Synthesis and Cadence® PKS:

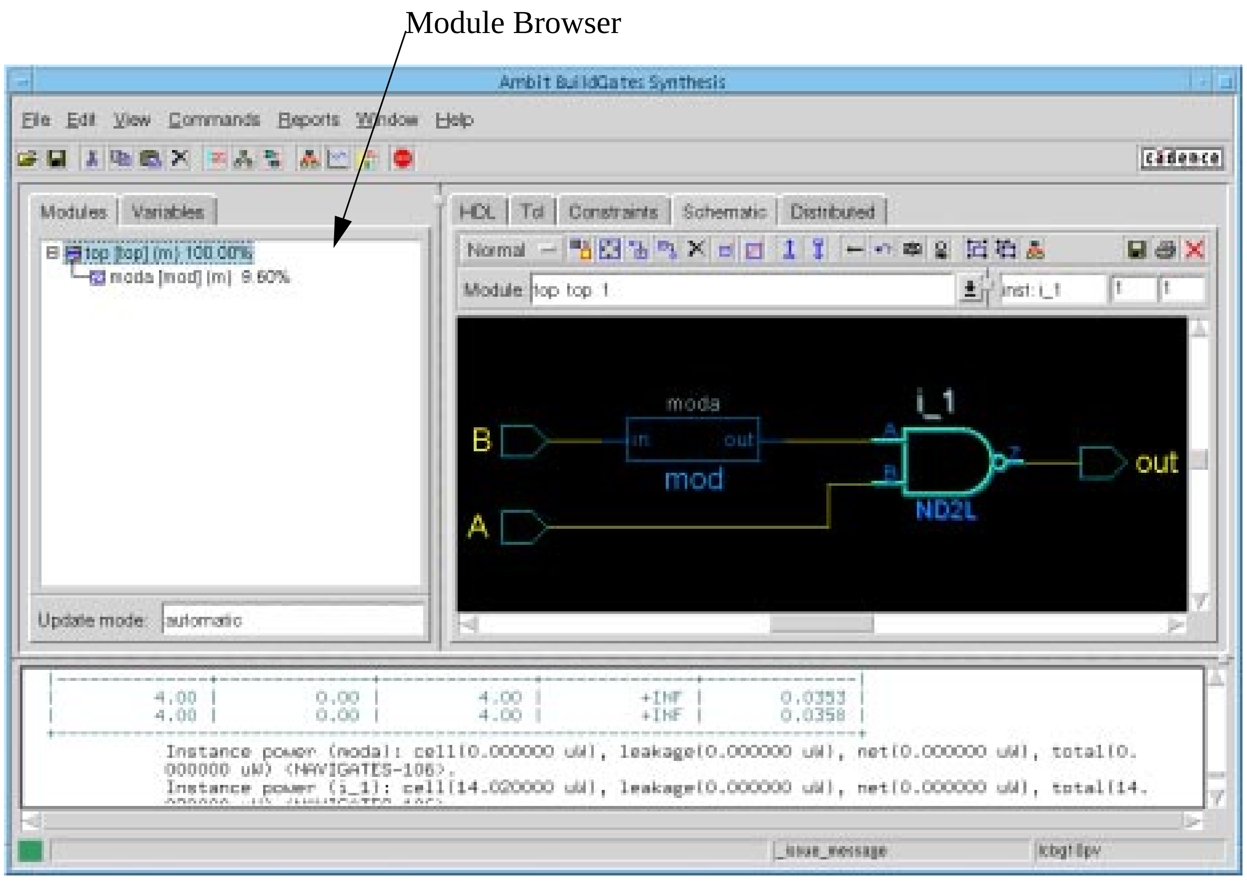
The Module and Schematic Views of Power

There are two ways to view power in the BuildGates Synthesis GUI:

- <u>The Module Browser</u>
- <u>The Schematic View of Power</u>

The Module Browser

The Module Browser displays the module hierarchy and instance paths in your design.



```
|---------------+---------------+---------------+---------------+---------------|
|          4.00 |          0.00 |          4.00 |          +INF |        0.0353 |
|          4.00 |          0.00 |          4.00 |          +INF |        0.0358 |
+---------------+---------------+---------------+---------------+---------------+
          Instance power (moda): cell(0.000000 uW), leakage(0.000000 uW), net(0.000000 uW), total(0.
          000000 uW) <NAVIGATES-106>.
          Instance power (i_1): cell(14.020000 uW), leakage(0.000000 uW), net(0.000000 uW), total(14.
```

Figure 4-1 The Module Viewer

The Module Browser provides these methods of viewing the power distribution in your design:

- Color-coded letters for the power structure names

  Colors used represent ten power levels ranging from level 0 (default: 00.00%) to level 9 (default: 20.00%). Colors and their corresponding power levels can be viewed using The Power Level Legend form. To change the power levels, see Setting Color Preferences on page 72.

- Power values for instances in the structure

  Values are displayed as either a percentage or an absolute value, depending on the setting selected on the *General Preferences – Power* form.

  See Setting Power Preferences on page 70 for more information.

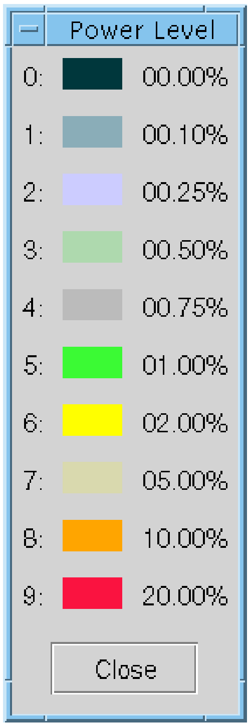
- The ability to display power structure viewed in the Schematic Viewer

  Double-clicking your mouse on a power structure name in the Module Browser displays the schematic equivalent view of it in the Schematic Viewer.

Note: These viewing methods are controlled by settings on the General Preferences form.

The Power Level Legend

The *Power Level Legend* form lets you view colors assigned to power levels being displayed in the Module Browser and Schematic Viewer.


Power Level

0:	⬛	00.00%
1:	🟦	00.10%
2:	🟪	00.25%
3:	🟩	00.50%
4:	⬜	00.75%
5:	🟩	01.00%
6:	🟨	02.00%
7:	🟨	05.00%
8:	🟧	10.00%
9:	🟥	20.00%

Figure 4-2 Power Level Legend Form

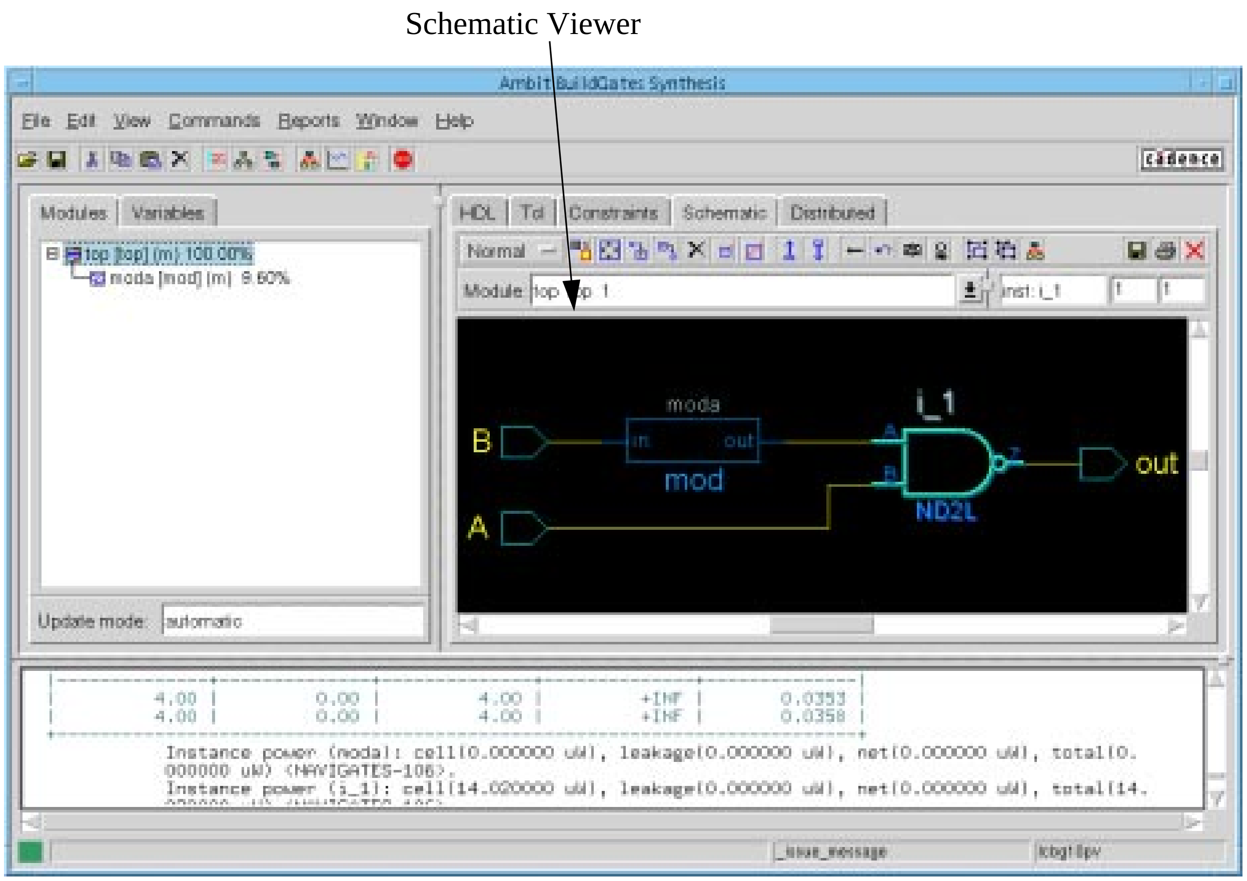
There are three columns on the *Power Level Legend* form:

Color	Displays the colors assigned to each layer. To change the power level colors assigned to each level, see Setting Color Preferences on page 72.
Level	Displays the ten power levels from 0 (the lowest) to 9 (the highest).
Value	Displays the values assigned to each power level. By default, level 0 is 00.00% and level 9 is 20.00%. To change these power level values, see Setting Color Preferences on page 72.

For more information about setting color preferences, see the *Ambit BuildGates Synthesis User Guide*.

The Schematic View of Power

You can use the Schematic Viewer to see the power distribution in your design.



```
|----------------+---------------+---------------+---------------+---------------|
|          4.00 |          0.00 |          4.00 |          +INF |         0.0353 |
|          4.00 |          0.00 |          4.00 |          +INF |         0.0358 |
+----------------+---------------+---------------+---------------+---------------+
            Instance power (moda): cell(0.000000 uW), leakage(0.000000 uW), net(0.000000 uW), total(0.
            000000 uW) <NAVIGATES-106>.
            Instance power (i_1): cell(14.020000 uW), leakage(0.000000 uW), net(0.000000 uW), total(14.
```

The Commands Menu

To open the *Commands Menu* that contains the power options

- Check that you have a power option license.
- Make sure that you do not have any schematic objects selected in the Schematic Viewer.
- Right-click your mouse. The *Commands* menu opens.

The power-related commands are

Colorize Power Levels If selected, power levels are colorized in the Schematic Viewer based on the power levels seen on the Power Level Legend Form.

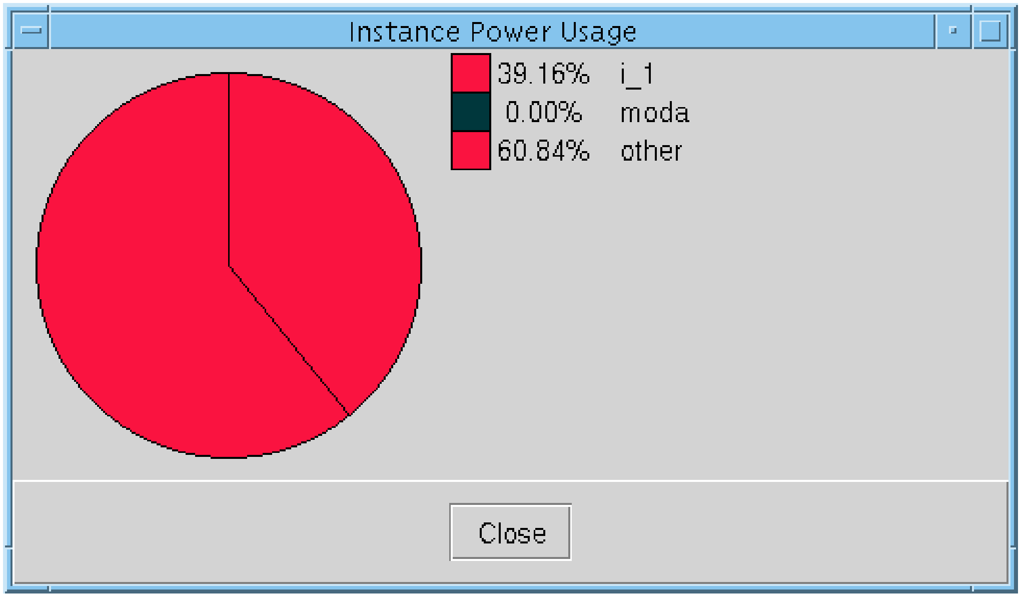
Annotate Instance Power If selected, displays the power value below each instance in the schematic view. If you have a flat netlist, instance power is the sum of internal power and leakage power. For a hierarchical netlist, instance power is the sum of internal and leakage power of primitive cells in the hierarchy and power consumed by the nets driven by the cells.

Annotate Net Power If selected, displays power values at the source of each net in the schematic view. You may choose to view signal probability, toggle count, capacitance, and power values for the net by using the *Schematic Preferences – Highlighting* dialog.

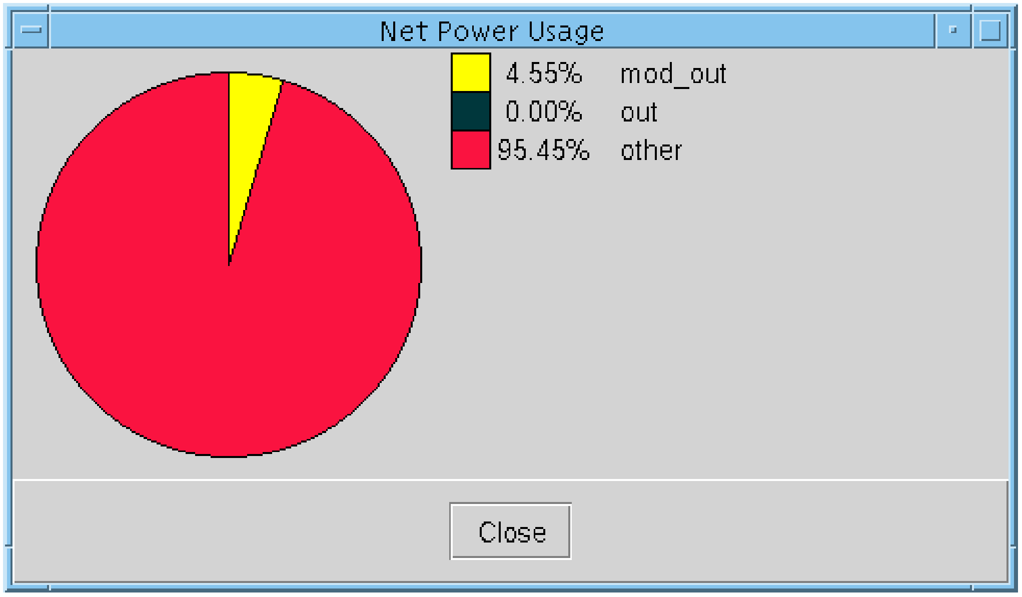
Worst Power Highlights (in the Schematic Viewer) the specified number of objects that contain the worst power values for the current module:

Power For:	Number to View:
`Nets`	3, 5, or 10
`Instances`	3, 5, or 10
`Both`	3, 5, or 10

Instance Power Piechart Opens a dialog box (*Instance Power Usage*) displaying a piechart of up to ten instances that consume the most power and the remainder.



Net Power Piechart	Opens a dialog box (*Net Power Usage*) displaying a piechart of up to ten nets that consume the most power and the remainder.



Note: All other schematic options visible on the *Commands* menu that are not power related operate the same way as they would when you are not in the power mode. See the *Ambit BuildGates Synthesis User Guide* for more information.

Other Schematic Power Options

Instances

If you have an instance selected in the Schematic Viewer and right-click the mouse, the *Instance Menu* opens.

Selecting *Power* from the menu displays the power value of the selected instance in the console window.

Nets

If you have a net selected in the Schematic Viewer and right-click the mouse, the *Net Menu* opens.

Selecting *Power* from the menu returns the power value of the selected net in the console window.

Pins

If you have a pin selected in the Schematic Viewer and right-click the mouse, the *Pin Menu* opens.

Selecting *Power* from the menu returns the power value of the selected pin in the console window.

Generating Power Reports

The Report Power Form

The *Report Power* form lets you generate a power report based on selected options.

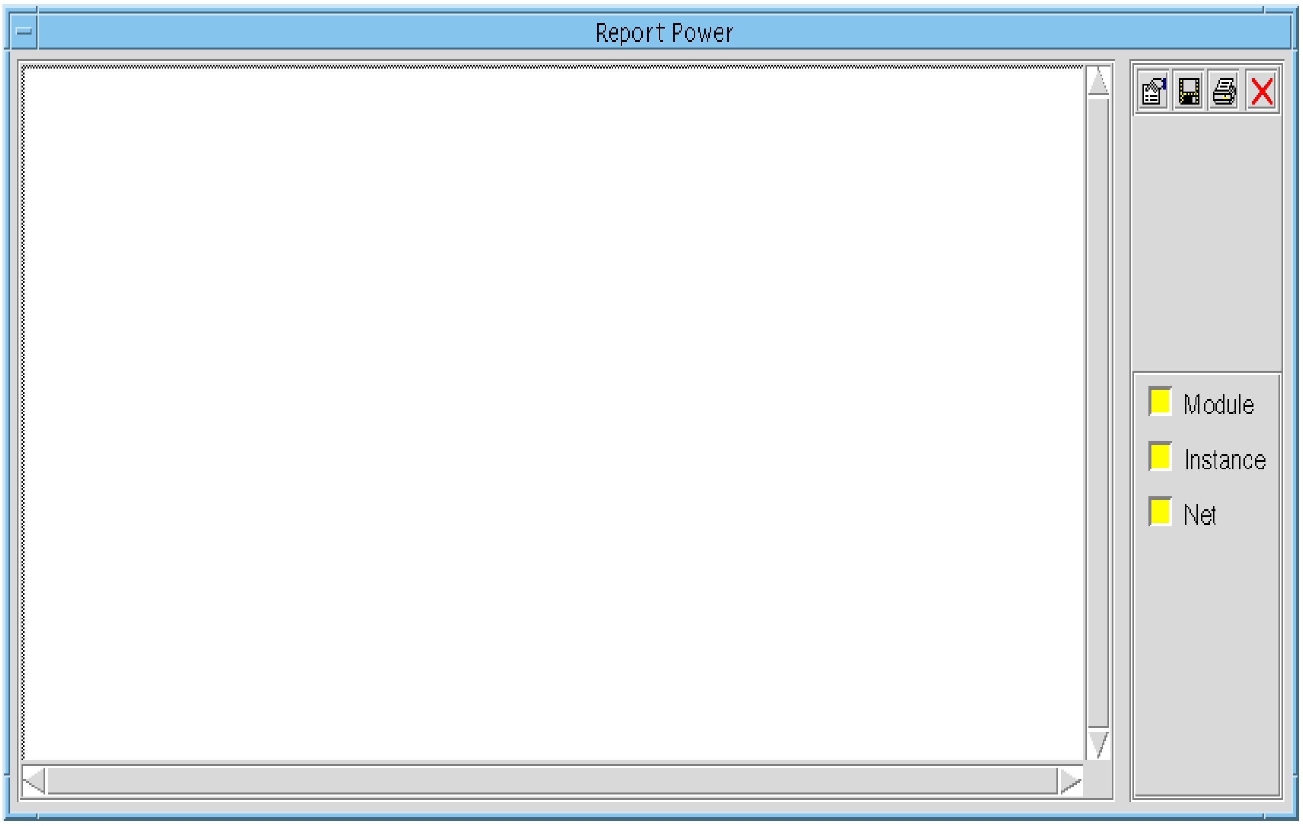
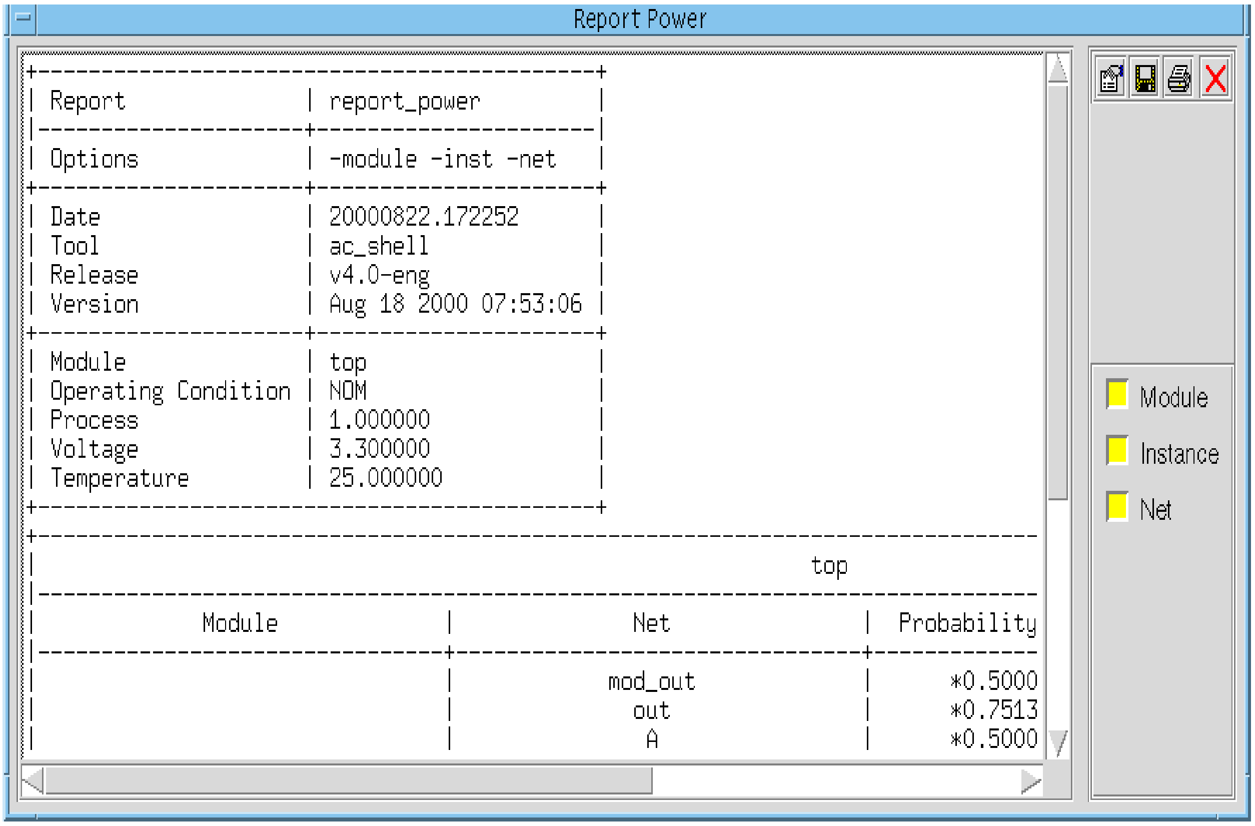


Figure 4-3 The Report Power Form

Module	Reports the power of all instances in the current module.
Instance	Reports the power of all instances.
Net	Reports the power of all nets.

Sample Power Report



```
+-------------------------------------------+
| Report               | report_power       |
|----------------------+--------------------|
| Options              | -module -inst -net |
+----------------------+--------------------+
| Date                 | 20000822.172252    |
| Tool                 | ac_shell           |
| Release              | v4.0-eng           |
| Version              | Aug 18 2000 07:53:06 |
+----------------------+--------------------+
| Module               | top                |
| Operating Condition  | NOM                |
| Process              | 1.000000           |
| Voltage              | 3.300000           |
| Temperature          | 25.000000          |
+-------------------------------------------+
+-------------------------------------------------------------------------------
|                                                            top
|-------------------------------------------------------------------------------
|             Module              |             Net              |  Probability
|---------------------------------+------------------------------+--------------
|                                 |           mod_out            |       *0.5000
|                                 |             out              |       *0.7513
|                                 |              A               |       *0.5000
```

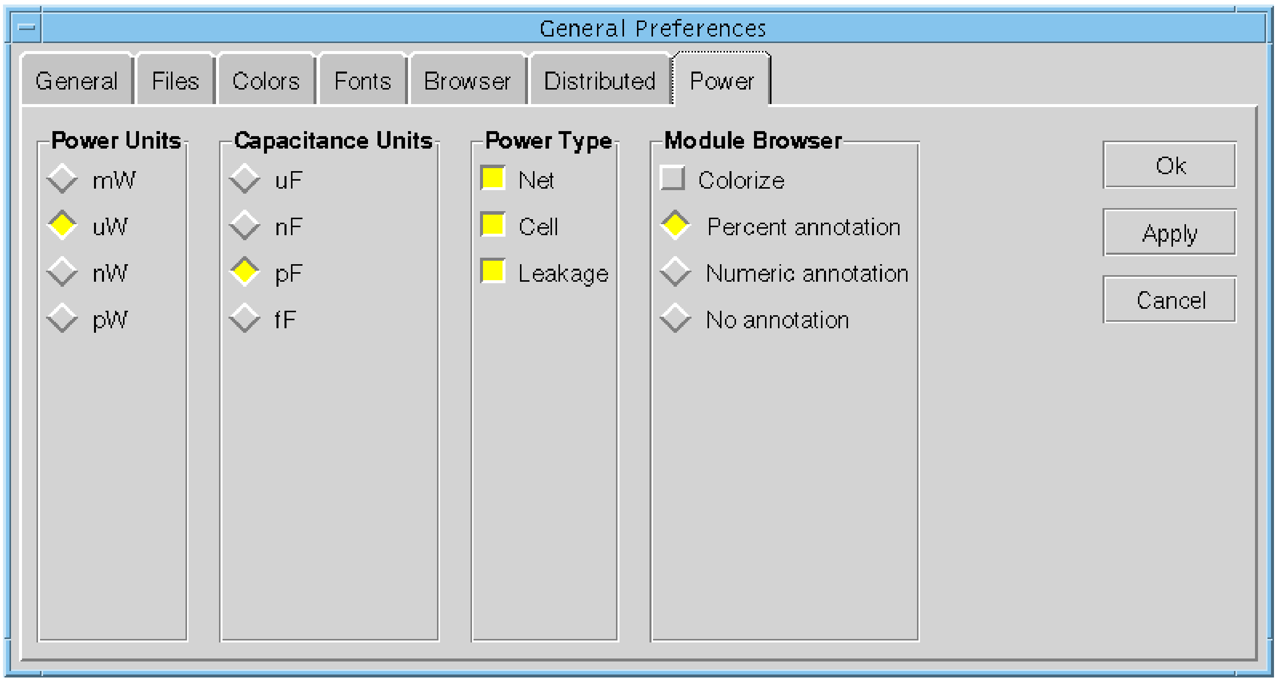
Figure 4-4 Power Report Example

Setting General Preferences

Setting Power Preferences

To set power preferences, such as what power units you would like to use and the style of power annotation displayed, you must open the *General Preferences – Power* form.

1. From the *View* menu of the BuildGates Synthesis GUI, select *General Preferences*. The *General Preferences* form opens.
2. Click the *Power* tab to display the *General Preferences – Power* form, as seen in Figure 4-1.



Power Units: mW [✓ uW] nW pW

Capacitance Units: uF nF [✓ pF] fF

Power Type: [✓ Net] [✓ Cell] [✓ Leakage]

Module Browser: Colorize [✓ Percent annotation] Numeric annotation No annotation

Figure 4-5 The General Preferences — Power Form

Power Units

mW	Sets the power units to milliwatts.
μ*W*	Sets the power units to microwatts.
nW	Sets the power units to nanowatts.
pW	Sets the power units to picowatts.

Capacitance Units

μ*F*	Sets the capacitance units to microfarads.
nF	Sets the capacitance units to nanofarads.
pF	Sets the capacitance units to picofarads.
fF	Sets the capacitance units to femtofarads.

Power Type

For selected instances in your design, *Power Type* lets you select the components of the total power calculated to be reported. By default, all types are on, meaning that the total power is calculated from the sum of net, cell, and leakage power.

Net	Calculates total power based on net power only.
Cell	Calculates total power based on cell power only.
Leakage	Calculates total power based on leakage power only.

General Options

Colorize module browser	Uses colors (as seen in The Power Level Legend form) to display power characteristics in the module and schematic viewers. To change values and colors assigned to power levels, see Setting Color Preferences on page 72.
Percent power annotation	Uses percentages to display the total power beside each instance name in the module view of power.

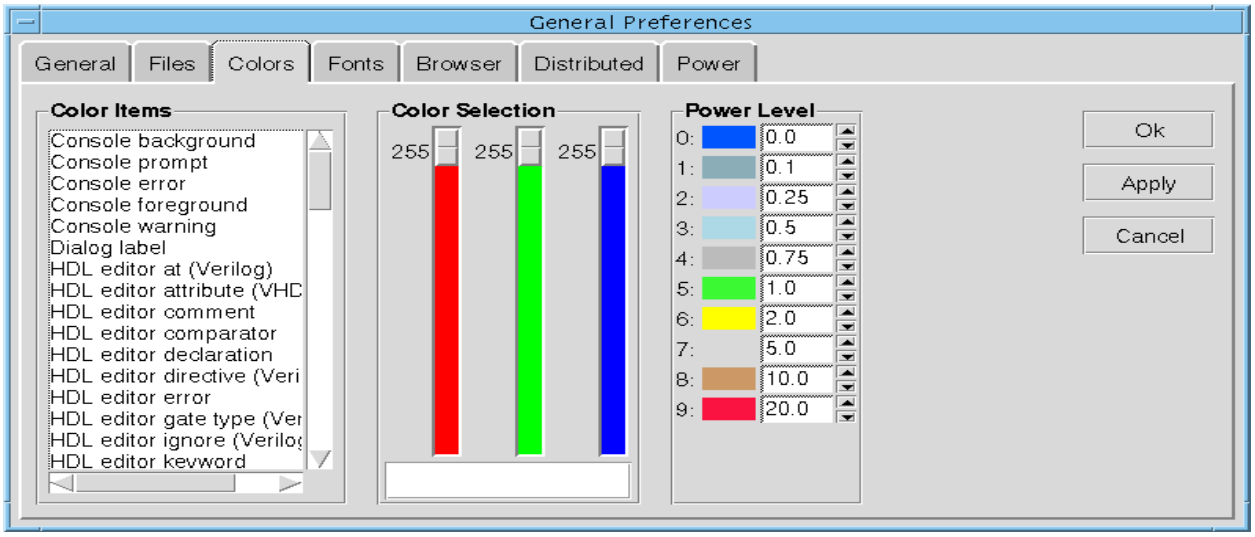
Note: Percent power levels are relative to the total power with respect to the top timing module. For a schematic, the percent power levels used for colors in the schematic are relative to the current module.

Numeric power annotation	Uses floating point numbers to display power values beside each instance name in the module view of power.
No annotation	Does not display any power annotation.

Setting Color Preferences

To set color preferences for power, open the *General Preferences – Color* form.

1. From the *View* menu of the BuildGates Synthesis GUI, select *General Preferences*. The *General Preferences* form opens.
2. Click the *Color* tab to display the *General Preferences – Power* form, as seen in Figure 4-2.



Color Items

Console background
Console prompt
Console error
Console foreground
Console warning
Dialog label
HDL editor at (Verilog)
HDL editor attribute (VHD
HDL editor comment
HDL editor comparator
HDL editor declaration
HDL editor directive (Veri
HDL editor error
HDL editor gate type (Ver
HDL editor ignore (Verilog
HDL editor keyword

Color Selection

255 255 255

Power Level

0:	🟦	0.0
1:	⬛	0.1
2:	🟪	0.25
3:	🟦	0.5
4:	⬜	0.75
5:	🟩	1.0
6:	🟨	2.0
7:		5.0
8:	🟫	10.0
9:	🟥	20.0

Figure 4-6 The General Preference — Color Form

Color Items	Displays a list of items in the database that have a color assigned to them. If you want to edit the color of an item, you must first select it by clicking on it in the *Color Item* column.

Color Selection	Adjusts the amount of red, green, and blue for the color of the selected object. Use your mouse to click and move the slider to adjust the colors accordingly.
Power Level	To change the values, you can either adjust the spinner controls for each power level or type the new value in the text field.

Viewing Probability and Toggle Count Information

You can now get the switching activity information on individual nets, as well as the overall distribution of the switching activity information for a particular instance.

The new toggle count histogram lets you view the probability distribution or toggle count information on a per net basis. You can use the histogram after reading in your TCF file and at other times during the remainder of your optimization and/or analysis flow because the histogram is updated throughout the flow.

Using the Command Line For Toggle Count and Probability Information

If you want to get toggle count information using the command line, use the `report_tc_stats` command.

Using the GUI For Toggle Count and Probability Information

In the main GUI, select *Reports-->Power*. Then, select either *Toggle Rate Histogram* or *Power Probability Histogram*.

You can also get these histograms in the Module Browser.

For the toggle rate:

1. Click the right mouse button.
2. Select *Power Histogram-->Toggle Rate*
3. The toggle rate histogram opens. See Figure 4-6 on page 74.

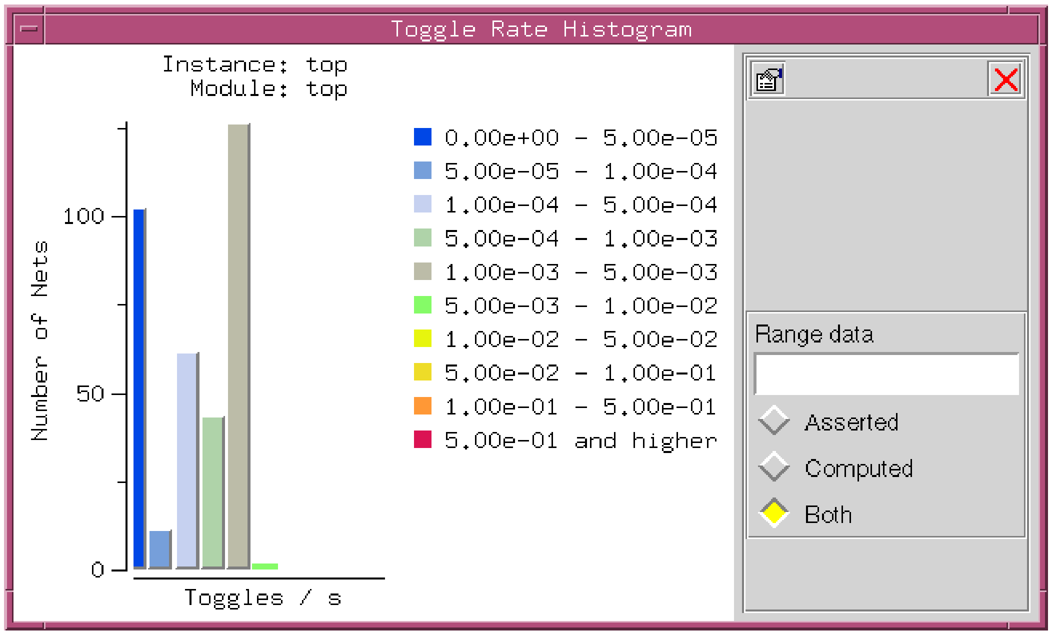


Figure 4-7 Toggle Rate Histogram

For the probability:

1. Click the right mouse button.
2. Select *Power Histogram-->Probability*
3. The probability histogram opens. See Figure 4-7 on page 75.

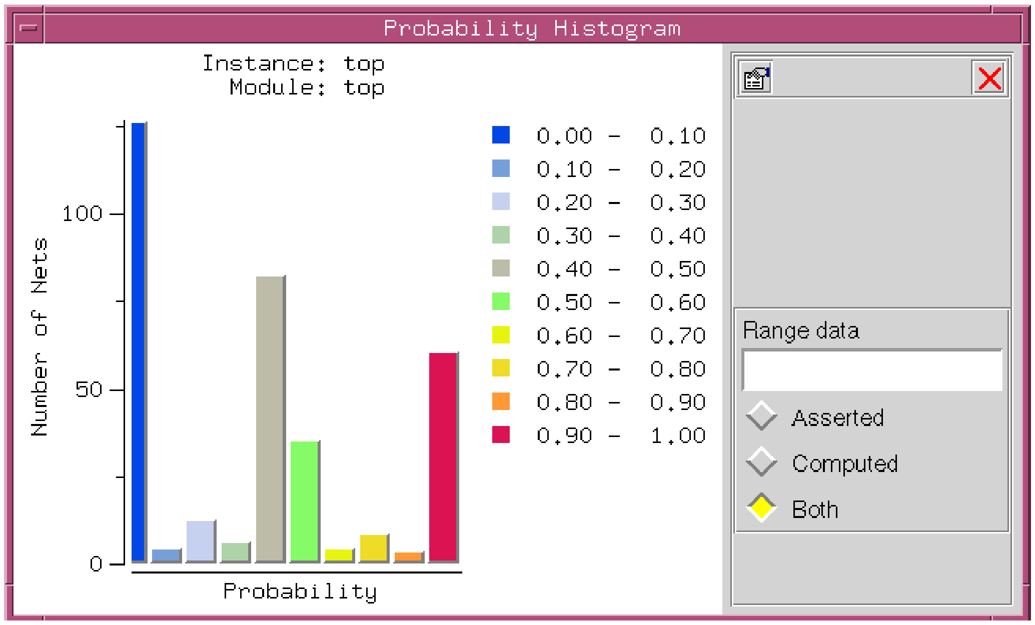


Figure 4-8 Probability Histogram

5

The Basic Power Optimization Flow

This chapter takes you through the basic power optimization flow using the LPS option of Ambit® BuildGates® Synthesis and Cadence® PKS, including concepts, terminology, and basic tasks:

Overview of the Low Power Synthesis (LPS) Design Flow

The new LPS design flow lets you lower the power consumption in your design from within the BuildGates Synthesis environment. The shaded boxes in Figure 5-1 highlight the LPS-specific steps in the Cadence® Synthesis Place-and-Route (SP&R) flow.

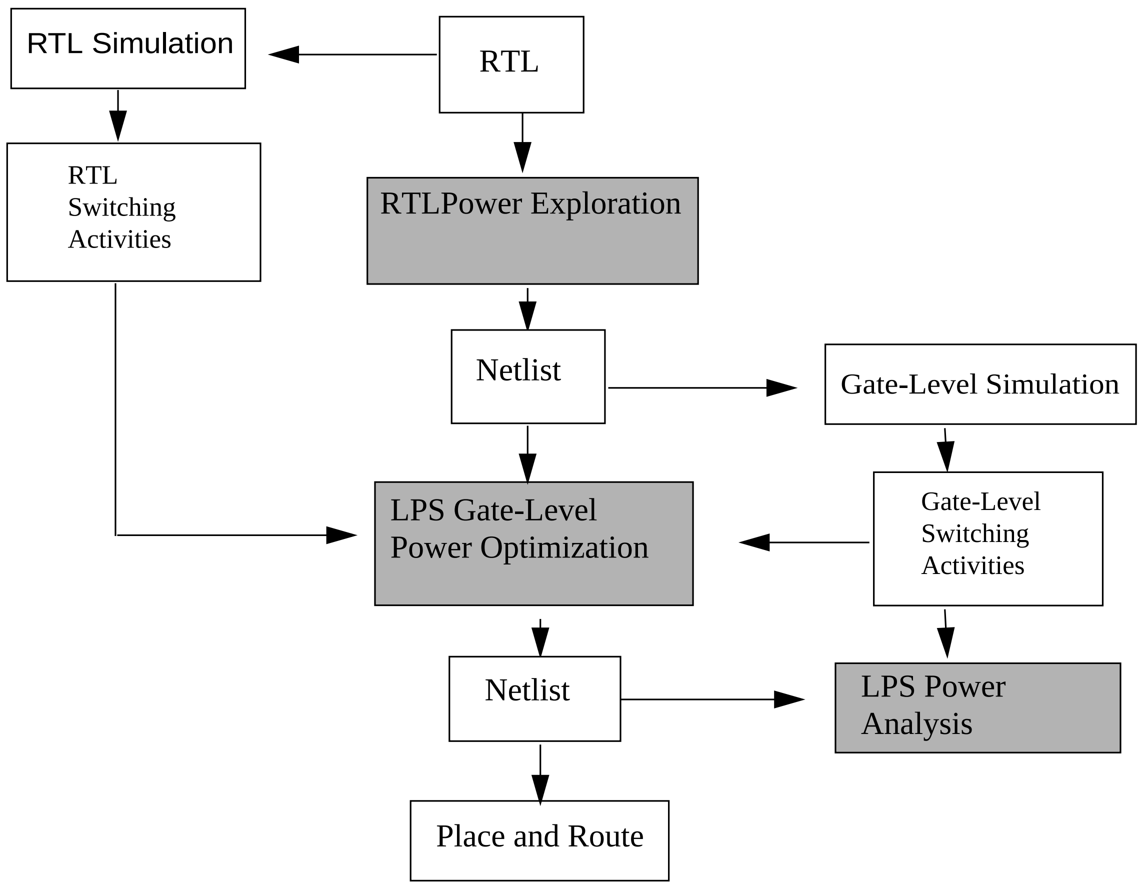


Figure 5-1 LPS in the Cadence SP&R Flow

As seen in Figure 5-1, power analysis and power optimization are performed at various points in the design flow. When the RTL is ready for synthesis, RTL power exploration is performed as part of the normal synthesis process. The design is analyzed to determine where power savings can be made. Clock-gating and sleep-mode logic are inserted accordingly. After the netlist is generated, a more detailed power analysis is done after simulating the gate-level netlist. Gate-level switching activity helps drive the gate-level power optimization where techniques, such as gate sizing, logic restructuring and buffer removal, are analyzed for potential power savings that preserve timing. The final step of the gate-level power optimization is committing all inserted RTL and gate-level transformation logic.

The remainder of this chapter walks through these basic steps of the LPS design flow.

Before You Start the LPS Flow

Here are some files that you will need to read into BuildGates Synthesis as you proceed through the LPS flow:

- A synthesis library containing power information
  - Use an `.alf` or `.tlf` file
  - Check that power information is available by using the `check_library -power` command.
- A simulation library
- A testbench
- A design (Verilog, VHDL, or gate level)
- A timing constraints file

Step 1: Reading in the Library

Terms and Concepts You Should Be Familiar With

- Ambit Library Format (ALF) File on page 108
- Supported TLF Power Statements on page 29
- Operating Conditions on page 109
- Library Characterization on page 109

Commands Used

If you have an `ALF` file, use the `read_alf` command to read in that library file.

If you have a `TLF` file, use the `read_tlf` command to read in that library file.

These commands are also available from the Graphical User Interface (GUI) of the tool. See the *Ambit BuildGates Synthesis User Guide* for more information.

After Reading in Your Library (Optional)

Setting Operating Conditions

After reading in your library file(s), you can use the `set_operating_conditions` command to specify the process, voltage, and temperature requirements for your design.

The software performs a check to verify that your library characterization and operating condition settings match. If they do not match, the software scales the values.

Checking Power Properties

To identify cells with power models in your library files, use the `check_library -power` command. See the *PKS chapter* of the Command Reference for details.

Step 2: Reading in the Design

Terms and Concepts You Should Be Familiar With

- RTL on page 109

Commands Used

Use either the `read_verilog` command or `read_vhdl` command to read in your design.

Note: Sleep-mode analysis and clock gating will not work later in the LPS flow if you read in a netlist format here. RTL transformations do not work with the netlist format; however, gate-level power optimizations can be performed with a netlist format design file.

Optional and Alternate Steps

Using a Netlist Format File

If you have already gone through the timing flow and are reading in a netlist format, you can read in that netlist format at this stage in the flow and then go to Step 6: Committing Logic and Performing Gate-Level Power Transformations on page 89 to skip the RTL stage and perform the gate-level power optimizations.

See the Gate-Level Power Optimization Only Flow on page 94 for more information.

Step 3: Exploring Power at the RTL

Terms and Concepts You Should Be Familiar With

- Clock Gating on page 108
- Timing Library Format (TLF) File on page 110
- Sleep Mode on page 109

Commands Used

It is recommended that you use both sleep-mode and clock-gating exploration. See Optional and Alternate Steps on page 82 to find out when you might want to use RTL power explorations separately.

Setting Sleep-Mode Options (Optional)

Set options for committing sleep-mode logic later in the flow during gate-level optimization by using the following command:

```
set_sleep_mode_options
```

Note: You can perform this task any time before you do any gate-level optimization.

See the *Chapter 5* of the Command Reference for detailed syntax for the `set_sleep_mode_options` command.

Performing Sleep-Mode Exploration

Run the `do_build_generic` command with the sleep mode option as shown:

```
do_build_generic -sleep_mode
```

This command explores possibilities for power savings by inserting sleep-mode logic. Logic is inserted but is not committed until you perform gate-level optimization. To removed inserted logic, see Removing Inserted Logic on page 82.

See the *Chapter 2* of the Command Reference for details about `do_build_generic` command usage.

Applying Timing Constraints

Source your timing constraints file.

```
source my_constraints.tcl
```

See *Setting Timing Constraints* in Timing Analysis for Ambit® BuildGates® Synthesis and Cadence® PKS for details.

Setting Clock-Gating Options (Optional)

Set options for committing clock-gating logic later in the flow during gate-level optimization using the following command:

```
set_clock_gating_options
```

The options related to exploration of power savings that *may* be set to change the default behavior of clock gating before running the `do_xform_optimize_generic` command are `-control`, `-observe`, `-drv`, `-ignore`, `-no_latch`, `-force`, and `-pref_map`.

The options that *may* be set to change the default behavior of clock gating before doing gate-level power optimization with the `do_optimize -power` command are `-minsize`, `-ignore`, `-force`, and `-no_timing`.

See the *Chapter 5* of the Command Reference for detailed syntax for the `set_clock_gating_options` command.

Performing the Clock-Gating Exploration.

Run the `do_xform_optimize_generic` command as shown:

```
do_xform_optimize_generic -clock_gate
```

The `do_xform_optimize_generic` command explores possibilities for power savings by inserting clock-gating logic. Logic is inserted but is not committed until you perform gate-level optimization. To remove inserted logic, see Removing Inserted Logic on page 82.

See the Command Reference for details about the *`do_xform_optimize_generic`* command usage.

Optional and Alternate Steps

Sleep Mode Versus Clock Gating

See Chapter 6, "Alternate Power Flows." for information about sleep-mode only and clock-gating only flows.

Not Performing Power Explorations

You do not have to perform sleep-mode or clock-gating analysis in order to run gate-level power optimization later in the design flow. You can run the `do_xform_optimize_power` command to perform gate-level power optimization only.

See Chapter 6, "Gate-Level Power Optimization Only Flow." for more information about the gate-level only power optimization flow.

Removing Inserted Logic

To remove the sleep-mode logic during gate-level optimization, run the following commands:

```
set_sleep_mode_options -remove
do_xform_optimize_power -no_gatelevel_opt
```

To remove inserted clock-gating logic, run the following commands:

```
set_clock_gating_options -remove
do_xform_optimize_power -no_gatelevel_opt
```

Checking for Design Rule Violations

You can run the `check_dft_rules` command before and after running the `do_xform_optimize_generic -clock_gate` command to be sure you do not have design rule violations before or after clock-gating RTL exploration.

Step 4: Reading In a Toggle Count Format (TCF) File

Terms and Concepts You Should Be Familiar With

- Operating Conditions on page 109
- Toggle Count Format (TCF) File on page 110
- The Incremental Switching Activity Calculation on page 21
- The Probabilistic Technique For Computing Switching Activities on page 21

Prerequisite

If you do not already have a toggle count format (TCF) file, you need to generate one. See Optional and Alternate Steps on page 83 for details about generating a TCF file.

Commands Used

Use the `read_tcf` command to read in your TCF file.

```
read_tcf TCF_filename.tcf
```

Optional and Alternate Steps

Generating a TCF File

There are four methods for creating a TCF file:

- Generating a TCF File From PLI Routines For a Verilog Design on page 83
- Generating a TCF File from a Value Change Dump (VCD) File on page 86
- Generating a TCF File for a VHDL Design on page 87
- Generating a TCF File for a Mixed (Verilog and VHDL) Design on page 88

Generating a TCF File From PLI Routines For a Verilog Design

When linked to a Verilog simulator, Programming Language Interface (PLI) routines intercept events on any net in a design and compute the toggle rate.

The PLI routines first initialize all nets in the module list specified as a parameter. Then, they traverse down the hierarchy, searching for the given module and calculating transition activities for each net. Finally, all switching activities are transferred into the TCF file which LPS uses to estimate power.

To generate switching activity for a Verilog design, you first need to start the process of counting the toggles in the testbench and then generate a report.

The following Verilog function calls need to be invoked to generate TCF files.

- $toggle_count *`list_of_modules`*
  - This function initializes the nets that needs to be monitored. The nets inside any of the modules in the *`list_of_modules`* is monitored. The monitoring of these nets starts right after the execution of this function.
- $toggle_count_report *`tcf_file`*, *`top_module`*
  - This function traverses all the nets that were monitored and reports the switching information in the *`tcf_file`*. If the *`top_module`* is provided, then it prints the net and pin names with this as the top. Otherwise, the net and pin names are printed with respect to the topmost module of the design.

The following steps take you through these tasks.

1. Change your testbench as follows:

```
initial begin
     if ($test$plusargs("toggle")) begin
     $toggle_count(tb_top.design_top.sub1, tb_top.design_top.sub2,...)
end
end

initial begin
     #(whole simulation period) ;
     if ($test$plusargs("toggle")) begin
     $toggle_count_report_hier or _flat("<tb.tcf, tb_top.design_top.sub2>");
end
$finish;
end
```

Note: The *`module_name`* is optional. If you do not provide the module name, LPS assumes the top module.

Here is a graphical representation of this testbench example:

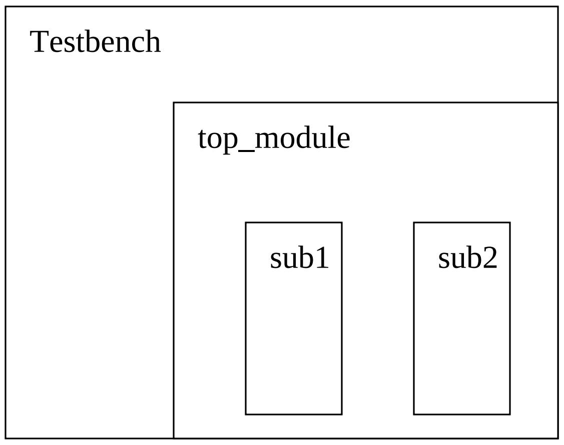


Figure 5-2 Testbench Hierarchy

2. Set the following (optional):

`/ambit_install_dir/BuildGates/`*`version`*`/lib/archives/`*`os/platform`*`:$LD_LIBRARY_PATH`

If you want to call the simulator from a C shell environment, you need to set this.

3. Add the following two options to your simulator:

- In Verilog-XL
  - `+loadpli1=lpspli:lps_bootstrap`
  - `+toggle`
- In NC-SIM,
  - Add following options to ncvlog

    ```
    -toggle
    ```
  - Add following options to ncelab

    ```
    -loadpli1=lpspli:lps_bootstrap
    -toggle
    ```

You should be able to call your simulator as usual, as shown here:

```
setenv LD_LIBRARY_PATH /ambit_install_dir/BuildGates/v4.0/lib/archives/sunos5/
sparc verilog -f <verilog_opt>
```

4. Check that a TCF file named `TCF_`*`name`*`.tcf` was created after simulation.

5. Read in your TCF file.

```
read_tcf TCF_name.tcf
```

Generating a TCF File from a Value Change Dump (VCD) File

Shipped with LPS is an executable that allows you to convert a VCD file to TCF. The executable is located in the software installation hierarchy at:

```
/ambit_install_dir/BuildGates/version/bin/os/platform_type/lpsvcd2tcf.exe
```

The VCD to TCF executable syntax is as follows:

```
lpsvcd2tcf top_module_name vcd_file_name -toggle {tcf_hier | tcf_flat}
-output_file {file_name} -divider -buschars
```

Where:

`-buschars {`*`bus_character`*`}`

Specifies how a bus is denoted. For example, you can specify a bus as either `bus[1]`or `bus{1}`. For the program to correctly parse the VCD file, you must correctly specify this argument to match the way you specify a bus. The default is `[ ]`.

`-divider {`*`divider_character`*`}`

Specifies how the hierarchical name is displayed. For example, if you specify the divider as ‘`.`’, the output would be something like this:
`ctrl.net1.net2`
If you specify the divider as ‘`/`’, the output would be something like this:
`ctrl/net1/net2`

`-output_file {`*`file_name`*`}`

Specifies the name of the output TCF file. Depending on the toggle value, the file must be either
`output_file_name``.tcf_flat` or
`output_file_name``.tcf_hier`.

`-toggle {tcf_flat | tcf_hier}`

The TCF file output type that you would like. The default is `tcf_hier`. Choose one of the following:

`tcf_flat`	Creates a flat TCF file. See the Flat TCF File Example on page 24.
`tcf_hier`	Creates a hierarchical TCF File. See A Hierarchical TCF Example For a Gate-Level Netlist on page 25.

`{`*`top_module_name`*`}`

Specifies the module for which you want to calculate power. Specifying this means that the activity for only that portion of the VCD file will be converted into the TCF file. For example, if your design as this hierarchy:
`TESTGEN_top/testgen_top/ctrl`
`TESTGEN_top/testgen_top/regs`
and you specify `ctrl` as the *`top_module_name`*, the `lpsvcd2tcf` executable reports only the activities for nets inside `ctrl`. If you specify `testgen_top` as the *`top_module_name`*, report activity for nets under both `ctrl` and `regs` is generated.

`vcd_file_name`

The VCD file which contains the event information that you want converted to a TCF file.

Here is an example of the VCD to TCF executable in use:

```
/ambit_install_dir/BuildGates/version/bin/sunos5/sparc32/lpsvcd2tcf.exe
design_a.vcd design_a.tcf
```

Note: A VCD file is generated directly from a VHDL simulator or it can be generated from a Verilog simulator using the `+dump` option.

Generating a TCF File for a VHDL Design

To generate a TCF File for a VHDL design, simply create a VCD file and then follow the process documented in Generating a TCF File from a Value Change Dump (VCD) File on page 86.

Generating a TCF File for a Mixed (Verilog and VHDL) Design

In the case of a mixed design, you can either follow the process of creating a VCD file and generating the TCF file from that, or you can add the PLI routines to the Verilog portion of the code. In the latter case, the PLI routine will generate the toggle counts for the part of the design written in Verilog.

Step 5: Getting Power Numbers (Optional)

Because you have already read in the library netlist, timing constraints, and a TCF file, you can now obtain the total power or power consumption for various objects in your design. You can perform this step anytime during the remainder of the flow.

Terms and Concepts You Should Be Familiar With

- Module
- Instance
- Power Consumption

Commands Used

Use the `get_power` command to get the total power of the current or specified instance.

User the `report_power` command to get power consumption information for specified objects in your design.

For detailed syntax for these commands, see *Chapter 5* of the Command Reference.

Step 6: Committing Logic and Performing Gate-Level Power Transformations

Terms and Concepts You Should Be Familiar With

- Gate Sizing on page 52
- Pin Swapping on page 53
- Buffer Removal on page 53
- Gate Merging on page 53
- Slew Optimization on page 54
- Logic Restructuring on page 55

Commands Used

In this final step of the basic LPS flow, you perform two tasks with one command: committing clock-gating and sleep-mode logic and performing gate-level power transformations.

Use the `do_optimize -power` command to commit the inserted logic and perform the gate-level transformations for meeting timing and to minimize power.

Optional and Alternate Steps

Committing Logic and Performing Gate-Level Transformations for a Timing-Optimized Netlist

If you have a timing-optimized netlist, you can use a different command in this step of the flow:

```
do_xform_optimize_power
```

The difference between the `do_optimize -power` command and the `do_xform_optimize_power` command is that the `do_optimize -power` command tries to improve timing whereas `do_xform_optimize_power` tries to maintain the original slack.

See Gate-Level Power Optimization Only Flow on page 94 if you have a timing-optimized netlist and you do not want to use clock-gating or sleep-mode analysis in your power flow.

Script Example

Here is a sample script that takes you through the basic LPS flow:

```
ac_shell -power
read_alf
read_verilog
do_build_generic -sleep_mode
source timing_constraints_file.tcl
do_xform_optimize_generic -clock_gate
do_xform_map -hier
write_verilog -hier ckt.v
# Perform simulation for toggle switching
read_tcf filename
do_optimize -power
report_power
```

6 Alternate Power Flows

This chapter takes you through some alternatives to the basic power optimization flow using the Low Power Synthesis (LPS) option of Ambit® BuildGates® Synthesis and Cadence® PKS, including sample scripts:

Sleep-Mode Only Flow

This flow omits clock-gating analysis and gate-level optimizations, focusing only on reducing power consumption for datapath components and arithmetic units. For this flow, only sleep-mode logic insertion and commitment are used to reduce power consumption.

See Sleep Mode on page 48 for more information about sleep mode.

Sample Sleep-Mode Only Flow Script

```
read_alf
read_verilog or read_vhdl
do_build_generic -sleep_mode
# Set timing constraints
do_xform_optimize_generic
do_xform_map -hier
write_verilog or write_vhdl -hier
# Simulate
read_tcf
do_optimize -power no_gate_level
```

Note: If you choose to use the Sleep-Mode Only Flow, keep in mind that LPS creates dangling ports when inserting sleep-mode logic. To avoid having dangling ports, run the `delete_unconnected_port` command before sleep-mode commitment during gate-level optimization as shown here:

```
read_alf
read_verilog
do_build_generic -sleep_mode
do_xform_optimize_generic -clock_gate
do_xform_map -hier
do_optimize -power low -effort high
delete_unconnected_port
```

Clock-Gating Only Flow

This flow omits sleep-mode analysis and gate-level transformations, focusing only on reducing power consumption for sequential elements that conditionally load data. For this flow, only clock-gating logic insertion and commitment are used to reduce power consumption.

See Clock Gating on page 37 for more information about clock gating.

Note: If you are really concerned about clock-tree synthesis for your design, Cadence recommends that you do not use the clock-gating technique.

Sample Clock-Gating Only Flow Script

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Set timing constraints
do_xform_optimize_generic -clock_gate
do_xform_map -hier
writer_verilog or write_vhdl -hier
# Simulate
read_tcf
do_optimize -power no_gate_level
```

Gate-Level Power Optimization Only Flow

This flow omits clock-gating and sleep-mode techniques when trying to reduce the power consumption of your design. Gate-level power transformations, such as buffer removal and gate sizing, are the only methods LPS used in this flow to address power problems.

See Gate-Level Power Optimization on page 51 for more details about the transformations used in this flow.

Sample Gate-Level Power Optimization Only Flow Script

```
read_alf
read_verilog or read_vhdl
do_build_generic
# Set timing constraints
do_xform_optimize_generic
do_xform_map -hier
writer_verilog or write_vhdl -hier
# Simulate
read_tcf
# Set timing constraints
do_optimize -power
```

Sample Script Starting from RTL

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
#Set timing constraints
do_xform_optimize_generic
do_xform_map -hier
write_verilog or write_vhdl -hier
# Simulate
read_tcf
do_optimize -power
```

Sample Script Starting from a Gate-Level Netlist

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Simulate
read_tcf
# Set timing constraints
do_optimize -power
```

Sample Script Starting from a Timing-Optimized Netlist

```
read_alf or read_tlf
read_verilog or read_vhdl
do_build_generic
# Simulate
read_tcf
# Set timing constraints
do_xform_optimize_power
```

Flow for Applying Different Transformations Per Module

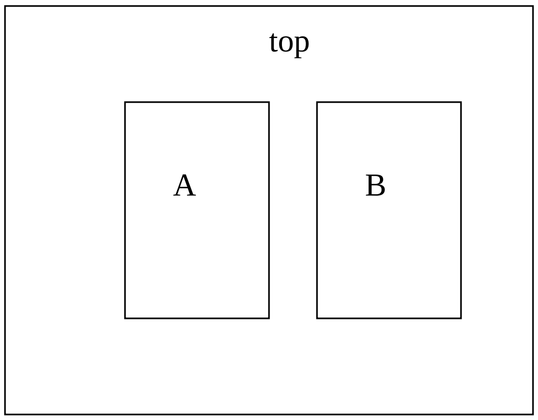
This flow allows you to use different methods of lowering power consumption for each module in your design. So, for example, if one module contains a lot of datapath components, you could use clock-gating analysis only. If a different module in the same hierarchy has a lot of sequential elements that conditionally load data, you could run sleep-mode analysis only. This flow makes that possible.

Sample Script for the Different Transformations Per Module Flow

The following example script explores and inserts sleep-mode logic in module A and explores and inserts clock-gating logic in module B. Gate-level power optimization is performed on the entire netlist. During that phase of the flow, sleep-mode logic in module A and clock-gating logic in module B is committed.

```
read_alf lib.alf
read_verilog design.v
do_build_generic -sleep_mode -module A
do_build_generic
set_current_module B
do_xform_optimize_generic -clock_gate
set_current_module top
do_xform_optimize_generic
do_xform_map -hier
write_verilog design.map.v
# Simulate
read_tcf design.tcf
do_optimize -power
```

The following graphics illustrates the design hierarchy for this script example:


top
A
B

DFT-LPS Flow

This flow lets you make a design with multiple clock domains observable.

See Clock Gating Multi-Clock Domains Under DFT Settings on page 41 for more information about observability of multiple clock domains.

Sample DFT-LPS Observability Flow Script

```
set_global dft_scan_path_connect tieblock
set_clock_gating_options
check_dft_rules
do_optimize_generic -clock_gate
do_optimize -power
set_global dft_scan_path_connect chain
do_xform_connect_scan_chain
```

Tips for the DFT-LPS Flow

For specific information about DFT commands, see *Chapter 6* of the Command Reference.

For more information about using DFT, see *Test Synthesis for Ambit BuildGates Synthesis and Cadence PKS*.

Preserving the Chain During a Top-Level Insertion

If you want to preserve the chain during top level scan chain insertion and you have a structural lower module with the scan chain intact:

1. Use the `set_dont_modify` or the `set_dont_touch_scan` command on that structural module.
2. Then run the top-level scan chain insertion with the `do_xform_scan_chain -hier` command.

Preserving Existing Scan Enable Ports

If you have RTL sub-modules that already have the `se` port and you want to preserve them when running DFT, you can do one of two things:

- Use the `set_dont_modify -network` command on the scan_enable network

  Because the `se` port of each module connects to anything, the connections between jtag/scan_enable and `se` will be done during `do_xform_optimize_generic`, if the `set_dont_modify -network` command is not set.

- Build the chain inside each module separately and then connect the scan chains on top. Here is the series of commands to do this:

  ```
  do_push_module submodule_id
  set_scan_mode se 1
  do_xform_connect_scan
  do_pop_module
  ```

Flat and Hierarchical Scan

To generate a hierarchical scan, use `write_scan_order_file -hier`.

To generate a flat scan, use `write_scan_order_file -flat`.

If you have more chains in the hierarchical scan as compared to the flat scan, run the `get_scan_chain_info -count` command to obtain the scan chain count.

For example, you might have two chains in your hierarchical scan and one chain in the flat scan. If `get_scan_chain_info -count` gives you a scan count of one, you might have a single chain that goes out of the module and enters back into the module. To view it as a single chain, view it from the top level and run `display_scan` in the GUI mode.

Automatic Test Port Creation

When you specify `set_clock_gating_options -auto_test_port`, the test port is automatically created on the top level.

Before you run LPS optimization, you might want to run the `check_dft_rules` command to see if you need to add `set_test_mode_setup` to the clock-gating scenario. If you specify `-auto_test_port`, only clock-gating components that are inserted by LPS are controlled by that test port.

LPS-PKS Flow

This flow includes power optimization as part of the more general PKS design flow.

Sample LPS-PKS Flow Script

```
read_alf
read_verilog or read_vhdl
do_build_generic -sleep_mode
# set timing constraints
do_xform_optimize_generic -clock_gate
do_xform_map -hier
write_verilog or write_vhdl -hier
# simulate
read_tcf
do_optimize -power-pks -stop
do_place
do_xform_optimize_slack -power
do_route
```

When you run the `do_optimize -power -pks -stop` command, WLM power optimization takes place. Sleep-mode and clock-gating candidates are committed or decommitted at this stage. Gate-level power optimizations are also performed here.

The `do_xform_optimize_slack -power` command in this flow does power-conscious timing optimization.

For more details about these commands, see *Chapter 5* of the Command Reference.

CT-PKS Power Flow

This flow includes power optimization as part of the CT-PKS flow.

Sample CT-Gen Power Flow Script

```
do_optimize -power -pks -stop
do_place
set_clock_tree_constraints
do_build_clock_tree -pin clk -power
```

7 Troubleshooting

This chapter helps you work through problems you might encounter when using the Low Power Synthesis (LPS) option of Ambit® BuildGates® Synthesis and Cadence® PKS.

General LPS Problems and Limitations

The following section describes problems you might encounter using the basic LPS flow as described in Chapter 5 of this manual.

When Conditions

LPS currently does not support when conditions that contain `pinQ` or `QN` pins or any sequential output pins. There is currently no solution for this limitation.

Clock-Gating and Sleep-Mode Options Still Set After `do_remove_design`

After running the `do_remove_design -all` command, options you set for either the `set_clock_gating_options` or `set_sleep_mode_options` command still exist.

The `do_remove_design -all` command does not affect either of these LPS commands. To change to the default settings, use the `-default` option for each command after running the `do_remove_design -all` command:

```
do_remove_design -all
set_clock_gating_options -default
set_sleep_mode_options -default
```

LPS-DFT Flow Problems

The following section describes problems you might encounter using the LPS-DFT flow.

Using the `ctrl_before_latch` Option Before `check_dft_rules`

In the LPS-DFT flow, you might see an error if you are using the `-ctrl_before_latch` option with the `set_clock_gating_options` command:

To resolve, use the workaround as mentioned in the LPS Option section of the *Known Problems and Solutions for Ambit BuildGates Synthesis* immediately before running the `check_dft_rules` command.

TDRC Violations When Using `check_dft_rules`

After running `do_xform_optimize_generic -clock_gate` and issuing the `check_dft_rules` command, you see many TDRC violations like this:

```
ERROR: Internally driven clock net ‘u2/n_53’ in module ‘block2’
```

There are several reasons you may see this error. Here are some suggestions for resolving the problem:

- You may not have set the controllability. Run the following command:

  ```
  set_clock_gating_options -control
  ```

  Then rerun `do_xform_optimize_generic -clock_gate` and the `check_dft_rules` commands

- Check to see if there are any warnings after the `do_xform_optimize_generic -clock_gate` command. You may see warnings that indicate that the clock-gating logic was not successfully added.

  If there are warnings, use the `set_test_mode_setup` command to set the test mode pin.

- Try to map the design and then check the DFT rules. If you have a generic netlist, the clock net might have some inverters or buffers.

Flip-Flops Do Not Have Output Nets

After mapping using the `do_xform_map -hier` command, you see warning messages saying that some flip-flops do not have output nets:

```
Info: Mapping module ‘LPS_CG_OBS_3’ ... <TCLNL-501>.
--> WARNING: Instance ‘LPS_CG_OBS_FF_34’ has no output net. <TM-203>.
Info: Mapping module ‘LPS_CG_OBS_2’ ... <TCLNL-501>.
--> WARNING: Instance ‘LPS_CG_OBS_FF_30’ has no output net <TM-203>.
Info: Mapping module ‘block2’ ... <TCLNL-501>.
```

LPS adds the flip-flops and these messages are the expected behavior because you have set the `set_clock_gating_options -obs_style` command to `register`. In this case, the observability registers have been created (`LPS_CG_OBS_FF_34` and `LPS_CG_OBS_FF_30`). Later in the LPS-DFT flow, these registers can be put into scan chains.

Top Level Flip-Flop Cannot Connect To Any Scan Chain

If your top level flip-flop (`OBS_FF`) can not be connected to any scan chain, this may mean that a portion of the design has been scan-mapped before you read it into the tool. Your flow might look like this:

```
do_push_module [find -module <module_name>]
set_test_mode_setup TestMode 1 -clock clk2
do_xform_connect_scan
do_pop_module
check_dft_rules
do_xform_optimize_generic -clock_gate
do_xform_map -hier
do_xform_connect_scan
```

If you simply remove `do_xform_connect_scan` from this flow, you should not have a problem connecting the top level flip-flop to a scan chain. So, the correct flow would look like this:

```
do_push_module [find -module <module_name>]
set_test_mode_setup TestMode 1 -clock clk2
do_pop_module
check_dft_rules
do_xform_optimize_generic -clock_gate
do_xform_map -hier
do_xform_connect_scan
```

The Observability Flip-Flop Is In a Different Scan Chain

If an observability flip-flop is in a different scan chain than the scan chain containing the clock-gating registers, it may be set to a different polarity. Try using the `-same_polarity` option with the `set_clock_gating_options` command.

An Observability Tree Is Outside the Sub-Module

If your observability tree is outside of the sub-module, it could be because the current xor_depth is less than the depth you defined. LPS inserts an observability block at the outermost hierarchy available.

For example, if you set the depth to 5, LPS could put the register in at 3 because it could not find a clock in a higher hierarchy and was forced to insert a register at that lower level.

See Chapter 3 for more details.

Warning When Setting Clock-Gating Options

You might see the following warning message when using the `-domain dft_domain` option for the `set_clock_gating_options` command:

```
WARNING: Observability style register requires observability domain to be
clock_net. Setting observability domain to clock_net <POPT-267>.
```

This means that you used the `set_clock_gating_options` command to set the observability style (`-obs_style`) to `register` or `reg_module`. In either case, you must set the `-domain` option to `clock_net`.

If you do not want this behavior, set the `-obs_style` option to port:

```
set_clock_gating_options -domain dft_domain -control -obs_style port
```

You will then be able to use `dft_domain` as the `-domain` setting for that attribute set.

Problem Setting Test Mode Port

You will have a problem if you try to use the `-auto_test_port` option with the `set_clock_gating_options` command to create a test mode port for a DFT domain. This option only creates ports for clock domains.

Error Associating Test Ports With Multiple Clock Domains

When using the following commands to associate test ports with their clock domains:

```
set_test_mode test1 1 -clock clk1
set_test_mode test2 1 -clock clk2
set_test_mode test1 1 -clock clk3
set_clock_gating_options -control
do_xform_optimize_generic -clock_gate
```

You get the following error message:

```
ERROR: No test port defined. If -domain is set to all, then test port must be defined
using -control. Clock gating will not insert testability logic <POPT-266>.
```

To resolve this error, you need to also specify the `-domain` option with the `set_clock_gating_options -control` command. Set it to either `clock_net` or `dft_domain`.

8

Glossary

Ambit Library Format (ALF) File

The library format file created by and used in the BuildGates synthesis software environment. See the *Ambit BuildGates Synthesis User Guide* for more information.

Assertions

Switching activities provided by an user through a Toggle Count Format (TCF) File.

Clock Gating

Selectively shutting off registers during inactive periods in order to lower power in the clock network and sequential elements.

Duty Cycle

The ratio of signal time to the total clock period for that signal.

Event

A logic state change, such as from 0 or 1.

INTERNAL_ENERGY

Dynamic energy (capacitive and short-circuit power) consumed by the cell when there is a transition in the specified path. This does not include energy released when charging the load capacitance connected to the output pins. Also, the constant (average) value per output load and input slew are supported. See the *Timing Library Format Reference* for more information.

Library Characterization

Editing your library to include information about operating conditions (process, voltage, and temperature). Complete this procedure before you read in the library.

Operating Conditions

The process, voltage, and temperature values you want to be taken into consideration.

Partial Transition

A transition which does not rise or fall all the way to the supply voltage or ground potential.

Power Exploration

LPS explores possibilities for lowering power in the design. None of the sleep-mode or clock-gating logic is actually committed; logic is only *inserted*. After running sleep-mode exploration and/or clock-gating exploration, you get a report of logic inserted in your design. Results are displayed in the console window. Commitment of inserted logic happens when you perform gate-level optimization.

RTL

Register transfer level.

SC_ENERGY

Short-circuit power consumed by the cell when there is a transition in the specified path. See the *Timing Library Format Reference* for more information.

Sleep Mode

Selectively shutting off combinational logic blocks, including datapath elements, functional blocks, and hierarchical modules, during inactive periods in order to reduce power in the combinational blocks and sequential elements.

Spatial Correlation

Two signals are considered to be spatially correlated if the occurrence of an Event for one of the signals affects the probability an event occurring on the other signal.

Temporal Correlation

Where the probability of an Event for a signal is dependent on an event occurring during a previous time step.

Timing Library Format (TLF) File

The library format file containing your timing information. See the *Timing Library Format Reference* for more information.

Toggle Count

The total number of toggles (signal transitions between 0 and 1 or 1 and 0) in the given simulation period at a node. Note that LPS displays toggle count in nano seconds(ns), but, when calculating power, LPS arrives as transition density using the following equation:

toggleCount/durationInNanoSeconds

Toggle Count Format (TCF) File

A Toggle Count Format (TCF) file contains net or pin switching activities, as well as signal probabilities. See The TCF File on page 23 for details.

Toggle Rate

The average number of toggle counts per second at a node.

TOTAL_ENERGY

This TLF construct captures the energy of the cell as well as the load it drives. LPS derives the internal cell power by subtracting the external load power for a given path from the total energy table.

XOR Tree Depth

The depth of the observability logic when decomposed into a balanced tree of two input xor gates.