Physical Synthesis Tutorial

Gord Allan

September 3, 2003

This tutorial is designed to take a simple digital design from RTL through to a routed layout.

Contents

ls	List the items in the current directory.
$cd[dir]$	Change to directory $< dir >$.
$cp < source >< dest >$	Copy source file to destination
$rm < file >$	Remove (or delete) $< file >$
$more < file >$	Displays the contents of a file, pausing on each page.
$lp < file >$	Prints a file to the standard printer.
$man < command >$	Gives help on any unix command. eg. *man ls*

Table 1: Common Unix Commands

1 Introduction

This tutorial accompanies a set of files which can be obtained from *www.doe.carleton.ca/ gallan/digflow.gz*. Together, they document how to take a sample design, a 16-bit x 8-bit signed multiplier through the CMC supported design flow from RTL description through to layout.

1.1 Introduction to UNIX

This tutorial assumes a basic knowledge of UNIX. The tutorial is run almost entirely from the unix command prompt. For those unfamiliar with unix, some basic commands are listed in Table 1. A good online reference can be found at *www.strath.ac.uk/CC/Courses/IntroToUnix*.

1.2 Tutorial Installation

This tutorial can be obtained from *www.doe.carleton.ca/~gallan/digflow*. In order to install and configure the tutorial, follow these steps:

1. Save the appropriate version of *digflow.gz* to your home directory on the unix system.
2. Unzip the gzipped file to a tar file — *gunzip digflow.gz*
3. Untar the tarball to create the directory structure — *tar -xvf digflow.tar*
4. Ensure you are using C-Shell[1].
5. Add the line *source ~/digflow/setup.digflow.csh* to your ~/.cshrc file.

[1] Issue *echo $SHELL* from a command prompt, the value should be either */bin/csh* or */bin/tcsh*. If it is not, add the line *tcsh* to your ~/.bashrc file

1.3 Related Documentation

The documentation can be divided into the following categories:

- *Cadence Tools*
  Online documentation is available via the *cdsdoc* command. This brings up a document browser which allows you to select or search for help on any of the Cadence tools. Selecting a document in the browser will, eventually, open a Netscape window pointing to the relevent document[2]. All of this documentation is provided in both .html and .pdf form and is physically located at */CMC/tools/cadence/{tool-stream}/doc/{tool}*. Within cdsdoc, there are many possible libraries. To get access to all relevent libraries, overwrite the file *~/.cdsdoc/cdsdoc.ini* with the one from *digflow/samples/cdsdoc.ini*.

- *Standard Cells*
  There are two standard cell libraries available to us in the .180 um technology — from Virtual Silicon Technologies (VST) and from Artisan. Shortcuts to the standard cell documentation (.pdf's) are located in *digflow/vstlib* and *digflow/artlib*. More information is available within the */CMC/kits/cmosp18/...* directory structure if neccessary.

- *Technology Parameters*
  As with the standard cells, a shortcut to the process parameter documentation is provided in *digflow/tech*. This file contains all of the electrical characteristyics regarding resistance and capacitance for different layers and operating conditions.

- *Synopsys Documentation*
  If using Synopsys' tools, the Synopsys On-Line Documentation (SOLD) can be accessed by typing the *sold* command. Within this documentation there is a very good description of RTL coding styles for proper synthesis — applying to both Synopsys and Cadence synthesis tools.

[2] If Netscape is too slow, when it opens it will not be pointing to the proper document. Re-selecting the document in the browser should fix the problem.

2 Design Flow

ASIC design flows vary widely according to the current state of EDA (Electronic Design Automation) tools and company preferences. The current flow is based primarily on tools provided by Cadence Design Systems, but where appropriate, competing tools are mentioned.

In this document we will focus on the steps from RTL Design through to Global Routing, but for completeness the entire ASIC flow is described.

- *Specification* — The system design must meet any intended standards. Referencing the standard, the designer would typically create custom C models for their portion of the design. System-level verification is performed by integrating these models with reference designs and ensuring performance requirements are met. Typical tools for system level design and specification include Matlab/Simulink, Cadence's SPW, and Synopsys' Co-Centric. SystemC and other variants are also emerging to perform system level design and verification.

- *RTL Design* — With parameters from the system designer, the hardware engineer must efficiently implement the required algorithm. This is done at the Behavioural or Register-Transfer-Level (RTL) using constructs such as adders, multipliers, memories and finite state machines. The mapping from a system level algorithm to a hardware description is typically a manual process, though there are efforts to automate it. Verification of the RTL design is performed by comparing its I/O vectors with those applied to the system-level model. Simulation of RTL can be done using tools such as Cadence's NC-Verilog, Synopsys' VCS, or Mentor's Modelsim.

- *Generic Mapping* — This automated step takes the RTL description and attempts to map it to generic hardware components such as gates, flip-flops, and adders. If there are portions of the RTL which cannot be described by hardware (ie. unsynthesisable code) or other problems (eg. latch inferencing), they are often found at this stage. The mapping step is contained within the main synthesis tool where the available tools are Synopsys' Design Compiler(DC) and Cadence's Buildgates/PKS.

- *Constraints* — After mapping to generic hardware, the designer *could* immediately compile the design into digital library cells. Doing so, however, the tool will pick the smallest available architectures to do the job (eg. ripple-carry adders vs. carry look-ahead). This leads to slower designs. Most often, the design will be required to operate with a certain throughput, and thus, a certain clock frequency ($f_{critical}$). By constraining the design, the user guides the tool to optimize certain paths.

- *Floorplanning* — As technologies become smaller, delay due to interconnect resistance and capacitance becomes more significant than gate-delays. Therefore, if two cells are physically beside each other they will experience much less delay than if seperated by the length of the chip. Thus,

in order to fully determine whether a design will meet timing and area requirements, it must be physically layed-out. During this step, the basic floorplan of the chip is described so that the interconnect delays can be estimated during compilation.

- *Power Planning* — Each cell must be connected to power and ground along its edges. To protect the chip wiring, the current through any particular wire must be limited below some threshold. Based on your design's speed, layout, and toggling activity, power rails must be distributed across the design so that this limit is not violated.

- *Compiling* — From the generic HW mapping, the tool picks elements from the digital library and logically arranges them to perform the required tasks within the timing constraints.

- *Scan Insertion* — If all of a design's flip-flops can be configured to form a long shift-register, manufacturing faults can be detected. Tools can automatically place multiplexors at the input to all flip-flops and link them together into a 'scan-chain.' During normal operation the circuit is unaffected, but when a test signal is asserted the scan-chain can be used to isolate manufacturing defects. Synopsys' DC, Cadence's PKS, and Mentor's FastScan can automatically insert the additional circuitry to allow scan-testing.

- *Clock Tree Insertion* — Ideally the clock signal will arrive to all flip-flops at the same time. Due to variations in buffering, loading, and interconnect lengths, however, the clock's arrival is skewed. A clock-tree insertion tool evaluates the loading and positioning of all clock related signals and places clock buffers in the appropriate spots to minimize skew to acceptable levels. Some clock tree insertion tools, all from Cadence, include CTSGen, ctgen, and CTPKS.

- *Optimization* — After placing the cells, adding scan circuitry and inserting a clock-tree, the design may no longer meet timing requirements. This optimization step can restructure logic, re-size cells, and vary cell placement in order to meet constraints.

- *Routing* — Up until this point, all timing estimates assume that signals can be routed without being detoured, as can be caused by wiring congestion. After initial optimization, the routing is actually performed in two steps:

    1. *Global Routing* creates a coarse routing map of the design. It evaluates areas which are highly congested and plans how signals should go around those area. After global routing, the design can be re-timed using more accurate interconnect data.
    2. *Final Routing* uses the plan from the global route and lays out the metal tracks and vias to physically connect the cells. Two final-routers are available - WarpRoute and NanoRoute.

- *Parasitic Extraction* — Once the detailed routing tracks are inserted, an extraction tool is used to more accurately determine the resistance and capacitance of each net. Two such extraction tools are 'Fire and Ice' and 'HyperExtract.' These tools can also be used to determine the cross-coupling capacitance between two signals which are important when evaluating signal integrity.

- *Post-Routing In-Place-Optimization* — After importing the parasitic information (usually in the form of a .rspf file), timing is re-evaluated to ensure it meets the constraints. At this stage limited changes can be performed, such as cell re-sizing and net re-routing in attempts to 'close timing'.

- *Signal Integrity Fixes* — If the cross-coupling capacitance between two signal lines is high, quick transitions on one net can affect the other. Within the EDA tools, these nets are referred to as 'victims' and 'agressors'. Agressors are characterized by large drivers and quick transistion times, whereas victims posess the opposite characteristics. Signal integrity violations can be divided into two categories:

  1. *Crosstalk* is caused when a victim and agressor pair transition at the same time. The victim may be either sped up (if both signals transition in the same direction), or delayed. This variation is then taken into account for either best or worst case timing analysis.
  2. *Glitching* is caused when a transition on the agressor net can cause a logical change (from 1-to-0 or 0-to-1) on the victim net.

  In either case, the signal integrity tool (Cadence's CeltIC) identifies the victim and agressor nets for repair. To fix such a violation, buffers can be inserted, nets can be re-routed, or shielding can be inserted between the offending nets. After any signal integrity fixes, extraction is re-done and timing closure must be verified.

- *Physical Checks* — Once timing closure has been assured, various physical checks are carried out. If any changes are made, extraction should be re-done and timing re-evaluated:

  - *Antenna Check* — During manufacture, when a metal patch is being deposited charge builds upon it. If the charge builds faster than it can be dissipated than a large voltage can be developed. If a transistor's gate is exposed to this large voltage then it can be destroyed. This is referred to as an antenna violation. To prevent this, leakage diodes can be inserted to drain excess charge, or long metal traces on a single layer can be prevented.
  - *Layout vs. Schematic (LVS)* — The LVS tool extracts the connectivity information from the routed layout and compares it with the final logical netlist. An LVS match confirms that errors were not introduced during the physical layout of the design. Tools to perform

for LVS include Cadence's Assura (formerly Diva, formerly Dracula) and Mentor's Calibre.

– *Design Rule Checking (DRC)* — The design rule check validates that the spacing and geometry in the design meets the requirements of the foundry. The same tools used for LVS are used to perform DRC.

3 HDL Coding Guidelines

Many of these items are taken, with permission, from "HDL Coding Guidelines," by Damjan Lampret and Jamil Khatib, June 7, 2001, www.opencores.org

3.1 Description

The guidelines are of different importance, and fall into three classes

- *Good practice* - signifies a rule that is common good practice and should be used in most cases. This means that in some cases there are specific problems that violate this rule.
- *Recommendation* - signifies a rule that is recommended. It is uncommon that a problem can not be solved without violating this rule.
- *Strong recommendation* - signifies a hard rule, this should be used in all situations unless a very good reason exists to violate it.

3.2 Resets

Resets make the design deterministic. It prevents reaching prohibited states and avoides simultation/synthesis mismatches.

- Recommendation: All flip-flops should have a reset. *Prevents simulation/synthesis mismatches.*
- Recommendation: Resets should be active-low. *Cell libraries contain active-low reset flops. Coding them as such prevents the insertion of unwanted buffering on the reset logic.*
- Recommendation: Resets should be asynchronous. *Most flops have them. Maintains compatibility between ASIC/FPGA code. Easier debugging.*
- Good Practice: The active-low reset should be applied asynchronously, de-asserted synchronously.

```
// synchronize the external reset
always @(posedge clk)
    rst_sn <= rst_an_pushbutton;

// reset comes off once when pushbutton is 'high' AND posedge clk
assign rst_an = rst_sn & rst_an_pushbutton;
```

All flops reset as soon as the pushbutton is applied — eases debugging. The reset track has a full clock cycle to de-assert after a clock edge — eases timing.

- Strong Recommendation: Active-low, asynchronously reset flops are coded as follows:

```
always @(posedge clock or negedge rst_an)
    if(~rst_an)  q <= 0;
    else         q <= d;
```

- Strong Recommendation: On an FPGA or CPLD the reset should be globally connected. *FPGAs and CPLDs have fixed routing that are connected to all device resources.*

3.3 Clocks

- Recommendation: Signals that cross different clock domains should be sampled before and after the crossing domains (double sampling is preferred). *Prevent meta-stability state.*

- Good practice: Use as few clock domains as possible in any design.

- Recommendation Do not use clocks or reset as data or as enable. Do not use data as clock or as reset. Code such as this must be prevented:

```
always @(posedge signal) begin ... end
```

  Synthesis results may be different than HDL, causes timing verification problems.

- Recommendation: Don't use gated clocks. *It negatively effects timing and can cause unwanted glitching. If necessary, they will be implemented at the top level of an IC.*

- Strong Recommendation: Clock signal must be connected to global dedicated reset or clock pin on an FPGA or CPLD. *This is because such pins provide low skew routing channels.*

3.4 Naming Conventions

- Good Practice: Try to write one module in one file. *The File name should be the same as the module's name.*

- Recommentation: Try to use named notation for instantiating instead of positional notation. *For easier debugging and understanding the code.*

- Good Practice: Keep the same signal name through different hierarchies. *So tracing after the signal will be easy. Enable easy netlist debugging.*

- Good Practice: Suffix signal names with *_a* for asynchronous and *_n* for active-low. eg. *rst_an* is an active-low asynchronous reset signal. *Helps keep logic clear.*

- Recommendation: Start buses at bit 0. *Some tools don't support buses that don't start at bit 0.*
- Recommendation: Use MSB to LSB for busses. *This is to avoid misinterpretation through the design hierarchy.*

3.5 Synchronous design and timing optimization

- Strong Recommendation: Use only synchronous design. *It avoids problems in synthesis, in timing verification and in simulations.*
- Recommendation: Avoid using latches. *They causes synthesis, testing, and timing verification problems.*
- Strong Recommendation: Do not use delay elements.
- Strong Recommendation: All blocks external IOs should be registered. *It prevents long timing paths.*
- Good Practice: Block internal IOs should be registered. *This is a design issue but is recommended in most cases.*
- Recommendation: Avoid using FlipFlop with negedge clock. *Causes synthesis problems and timing verification problems.*
- Strong recommendation: Include all signals that are read inside a combinational process in its sensitivity list. (i.e. Signals on Right Hand Side RHS of signal assignments or conditions. *This is to prevent simulation/synthesis mismatches.*
- Strong recommendation: Ensure variables are assigned in every branch of a combinational logic process. *Prevents inferring of unwanted latches.*

3.6 General rules

- Strong Recommendation: In RTL, never initialize registers in their declaration. Use proper reset logic. *Initialization statements can not be synthesised.*
- Recommended: Write fsms in two always blocks — one for sequential assignments (registers) and the other for combinational logic. *This provides more readability and prediction of combinational logic size.*
- Strong Recommendation: Use non blocking assignment (<=) in clocked blocks, and blocking assignment (=) in combinational blocks. *Synthesis tools expects for this format. Makes the simulation respond deterministically.*
- Recommendation: Try to use the 'include' command without a path. *HDL should be environment independent.*

- Good Practice: Compare buses with the same width. *The missing bits may have unexpected value in the comparison process.*

- Strong recommendation: Avoid using long if-then-else statements and use case statement instead. *This is to prevent inferring of large priority decoders and makes the code easier to be read.*

- Strong Recommendation: Avoid using internal tri-state signals. *They increase power consumption and make backend tuning more difficult.*

3.7 Simulation and Debugging

- Strong Recommendation: Test benches should be intelligent enough to determine sucessfull operation without user interaction. *Reduces development time and human oversights.*

- Strong Recommendation: The same test-bench should be used for RTL and gate-level simulations. *Ensures that synthesis and optimization is sucessfull.*

- Recommendation: Try to write the test bench in two parts, one for data generation and checking and one for interfacing to the device-under-test. The interface to the device should be written with normal hardware coding rules in place. *This is to isolate data (results checking) from the hardware interfacing. By writing the interface logic with conventional hardware description (ie. registers), it allows for interchangable RTL and gate level simulation.*

- Good Practice: Use *$display("%t - (%m) Message", $time, vars...)* liberally to provide information while debugging a design.

- Good Practice: Ensure the 'timescale command is specified only once. *Different 'timescale causes simulation problems: races and too long paths.*

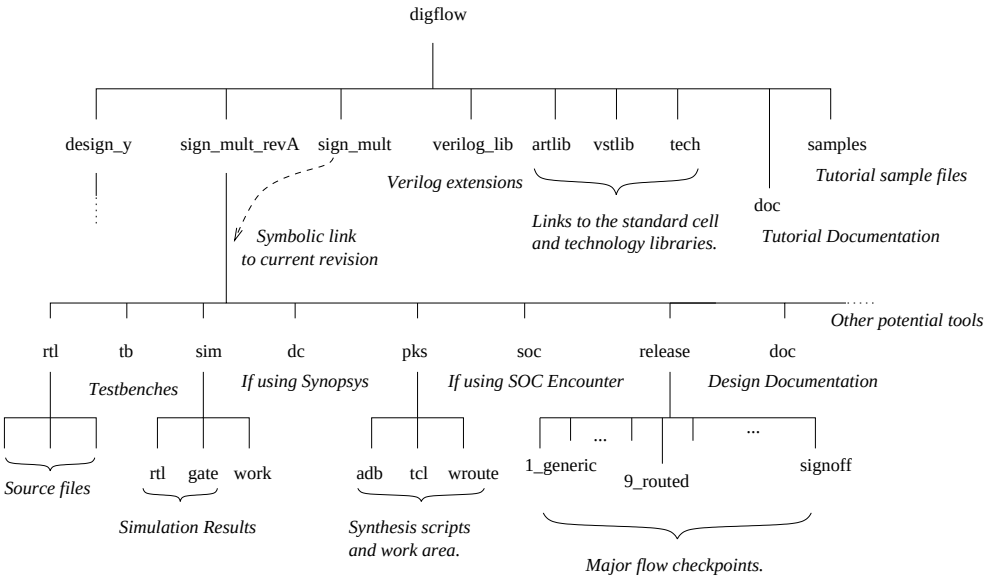


Figure 1: Tutorial Directory Structure

4 The 16x8 Signed Multiplier

4.1 Directory Structure

Before starting any project it is important to organize the directory hierarchy logically. The structure that comes with this flow is shown in Figure 1. At the top level, there are links to current designs and library locations. The links to the library information are there for convenience, allowing the tools to reference common locations across different system configurations. In addition to the library data, design directories exist for each major project, or project revision. Also for convenience, a symbolic link is created which points to the current project.

Within each project, directories exist for the RTL source code, testbenches, simulation runs, and for each major tool used in the design flow. There is also a *Release* directory which holds all the relevent files at a certain point in the design flow. This approach allows for easy handoff of design data between tools, and provides check-points in the design which can be restored in case of problems.

4.2 Multiplier Design

Figure 2 is the schematic representation of the signed multiplier used in this tutorial. Provided in the tutorial's RTL directory (digflow/signed_mult/rtl) are 4 variations of the design, all of which perform the same ultimate function.

- *Instantiating a 'Canned' Multiplier*: In this implementation, we specifi-

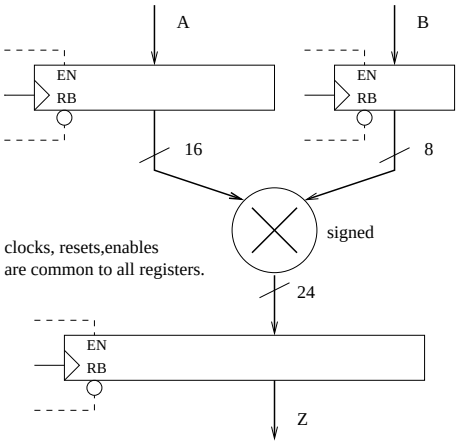


Figure 2: 16-bit * 8-bit Signed Multiplier Sample Design

cally instantiate a signed multiplier that is provided by Synopsys in it's DesignWare Component Library. [3] This approach tends to give the best synthesis results, but requires that these components be researched and available on the target system.

- *Behavioural Description*: The simplest way to describe a multiplier is to use verilog's $*$ operator. Without extra precautions however, this will not work for signed values. To perform signed multiplication, the inputs A and B must first be sign-extended to the width of the result — in this case 24 bits. Then, performing $Z = A_{extended} * B_{extended}$ will create a 24x24 unsigned multiplier, producing a 48-bit result. Of which, the least-significant 24-bits are actually our signed result. We then rely on the synthesis tool to remove the unnecessary logic for the upper half of the multiplier. Depending on the tool, this approach synthesises almost as well as instantiating an optimzed signed multiplier. [4]

- *Structural Description*: Many experienced designers still tend to write structural descriptions of their hardware, assuming that they can do a better job structuring the logic than the synthesis tool. This is likely a holdback to the time when the tools weren't nearly as competant as today. For datapath components (eg. adders, multipliers, etc...) this approach almost always results in less efficient designs than those generated automatically. In this case, an 'optimal' signed multiplier was coded without using any high-level constructs. The resultant circuit was twice as large and half as fast as the circuit synthesised from the behavioural description.

[3] The documentation for Designware components can be accessed via the 'sold' command to open the Synopsys On-Line Documentation.

[4] Using Cadence PKS the resultant design was 10% larger than using the DesignWare multiplier, wheras Synopsys' Design Compiler, produced a circuit twice as large.

- *Paramaterized Behavioural Description*: This description and architecture is equivalent to the sign-extension solution earlier, but, in this case the operand widths of A and B are specified as parameters. This allows the code to be re-used in any situation and is higly encouraged. On the other hand, paramaterized code is often more difficult to read and understand. A UNIX symbolic link is used to make this file the default for this tutorial.

4.3 Verification Platform

The cardinal rule of verification is that test-benches should be able to evaluate a circuit's performance without user interaction. In most cases this is performed by applying a set of inputs and automatically comparing the outputs against proper results.

Most often, the proper results (or expected vectors) can be generated within verilog itself. As a software language, similar to C, it can perform all basic floating point and integer operations. Also, included in *digflow/verilog_lib/lib* is a library which expands verilog to perform complex functions using system calls such as $sin(realval) or $powxy(3.1415, realval). Performing vector checks and error accounting within verilog, keeps the verification environment in one tool,reducing complexity. We use this method in the case of the signed multiplier, since expected vectors are easily generated 'in-house' using integer arithmetic.

When the trusted results can not be generated within verilog, or have been generated using system-level design tools, there are two choices.

- A co-simulation environment can allow the verilog to run along-side the system level model and the results can be actively compared.
- The system tool can print IO vectors to files, and read into a verilog testbench using the $readmemh system function.

In many cases it is convenient to ignore the effects of hardware induced latency when we compare results versus expected vectors. To achieve this, functions are provided in *digflow/verilog_lib/src/vector_search.v* that search for the partial occurance of one vector within another.

Overall, the testbench structure shown in Figure 3 is used in this tutorial. It is also recommended for use in your own designs. To ensure proper synthesis, the same set of tests should be used to verify your RTL and gate level designs. We accomplish this using two top-level wrapper files, rtlsim.v and gatesim.v, which call the same testbench.

The main testbench, *main_tb.v*, provides a framework for running many small independent tests. It is responsible for initializing variables, instantiating the device under test (DUT), providing IO facilities for individual tests, and for including any common functions which may be usefull. By housing many small tests in a common environment, large-scale verification can be performed while minimizing testbench complexity.

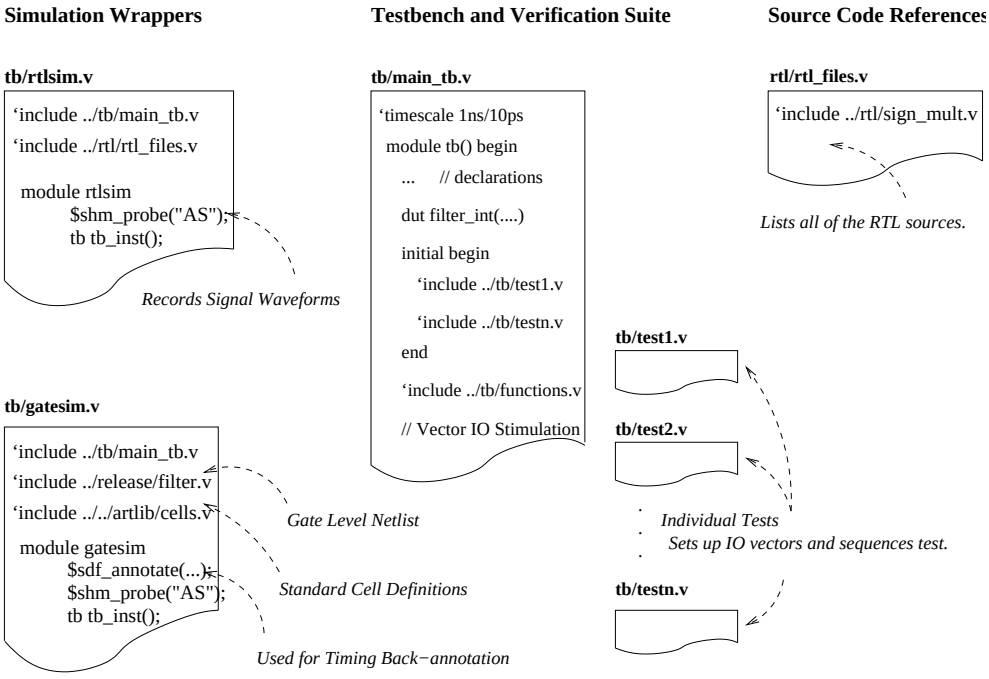


Figure 3: Testbench and Simulation File Structure

5 Verilog Simulation

Within the UNIX environment we will use Cadence's NC-Verilog for our simulations.

5.1 Setting up NC-Verilog

NC-Verilog is the new version of Cadence's Verilog-XL. It is much faster than most simulators since it compiles the code before executing it.

In theory, to simulate with NC-Verilog requires three seperate steps — compling, linking and execution — each of which normally uses a seperate command. However, for the purposes of this tutorial we are going to use NC-Verilog in Verilog-XL compatibility mode. This allows us to perform all three steps at once.

Unlike Verilog-XL, when NC-Verilog is run, it must have a directory available for storing temporary files. This is specified in *multiply/sim/cds.lib*. This file, the referenced work directory, and an empty file *hdl.var* must exist in the directory where ncverilog is run.

To simulate a set of files, one then issues the command [5]:

```
ncverilog [+options] testbench.v rtlfile1.v rtlfile2.v
```

5.2 Simulating a Design

1. Referring to Section 4.2, examine the file *multiply/rtl/signed_mult.v* to obtain some understanding of the sample design.

2. From the *multiply/sim* directory, run the command:

   ```
   ncverilog ../rtl/signed_mult.v
   ```

   Though this will not run a simulation, it will compile the design and inform you of any syntax errors. Note that the output from any ncverilog run is captured in the file *ncverilog.log*.

3. Familiarize yourself with the main testbench *../tb/main_tb.v*:

   - Line 1: The 'timescale directive should only be included once at the beginning of a simulation.
   - Line 7: The VERBOSE constant is used to determine the extent of debugging information displayed. 0 for None, and higher values to dump more information.

[5] For speedy operation, by default, NC-Verilog does not record waveform traces, even when told to. Using the "+access+r" options over-rides this behaviour. Running the setup script in this tutorial "aliases" ncverilog to ncverilog +access+r so that signal recording is on by default.

- Lines 27-28: The check_vectors routine in *verilog_lib/src/vector_search.v* searches for the occurance of *expected_buffer* in *output_buffer*. Since arrays cannot be passed in standard verilog, these must be global variables.
- Line 34: The instantiation of the multiplier, or the device-under-test (DUT).
- Line 47: If the vector search routines are used they must be included within the module definition.
- Line 53: The result from the DUT is converted to an integer using sign-extension.
- Lines 56-63: The interface to the DUT should behave like hardware, capturing the result on the positive edge of the clock like a register. The integer results are stored sequentially in *output_buffer* for later comparison.
- Lines 66-67: It is convenient to specify the inputs A and B as integers. This truncates them for application to the DUT.
- Line 73: Displays the IO vectors if the VERBOSE constant is above 0.
- Line 87: Start of main test sequencing.
- Lines 104-110: Reset the system at the start of each test. A good rule of thumb is not to change inputs at the active clock edge. As such we use the negative edge of the clock to trigger all changes to DUT inputs.
- Lines 115-121: Prepare random inputs for the DUT within the proper range of values.
- Line 123: Calculate the expected result using verilog's integer multiplication abilities.
- Line 127: Call the check_vector function to search for 90 consecutive matching positions between *output_buffer* and *expected_buffer*. The routine displays whether a match was found or not.
- Line 133: Start the next test using the same format as lines 104 through 130.

4. Having looked at the RTL and the testbench, run the simulation from the *multiply/sim* directory, with the command:

```
ncverilog ../tb/main_tb.v ../rtl/signed_mult.v
```

Examine the output and note how the search function reports that the expected vectors were found in the recorded output stream. To get more detailed information, change Line 7 of main_tb.v to *'define VERBOSE*

2 and re-run the simulation. Now each result is displayed as it occurs, and the output and expected buffers are displayed by the search routine. Change the VERBOSE level to 1 and re-run the simulation to observe the difference.

5. Now we'll intentionally introduce a bug and view the simulation result. In *rtl/signed_mult.v*, change Line 78 to use the unextended inputs Areg and Breg instead of Aext and Bext. Re-run the simulation and examine the output to see how the errors are reported. Ensure you fix *rtl/signed_mult.v* before moving on.

6. Rather than using "NC-Verilog", we'll try using the slightly older (and slower) "Verilog-XL" for the next simulation (just so you can say you've used Verilog-XL). Replace "ncverilog" with "verilog" on the command line.

```
verilog ../tb/main_tb.v ../rtl/signed_mult.v
```

7. To see the advantage of the vector-search routines, run the testbench against a different implementation of the multiplier.

```
verilog ../tb/main_tb.v ../rtl/signed_mult_bisec.v
```

   In this design, the output is not registered within the module and so the results appear a cycle earlier. Note how the search-routine reports that the expected string was found at position 2 in the output buffer, not 3 as before. Without a flexible routine to match up the output and expected vectors, the test would have improperly failed.

5.3 Waveforms in UNIX simulations

5.3.1 Recording

Though log files should inform the user whether a test was successfull, they are not as usefull as waveforms for tracking down bugs. Unlike Silos on the PC's, Verilog-XL and NC-Verilog do not automatically record waveforms for viewing and debugging. To record such a waveform, we use the *$shm_open* and *$shm_probe* system functions. Since these are unavailable on non Cadence simulators, we should avoid putting them in the main testbench. Instead, we create a wrapper. Look at the file *tb/rtlsim.v*. Here, we issue the *$shm_open("rtlsim")* function to open a waveform database called rtlsim, and *$shm_probe("AS")* to record all-signals (AS). Instead, we could list specific signals within the *$shm_probe* statement. We then instantiate the main testbench to run underneath.

From the *multiply/sim* directory, run the simulation and record waveforms with:

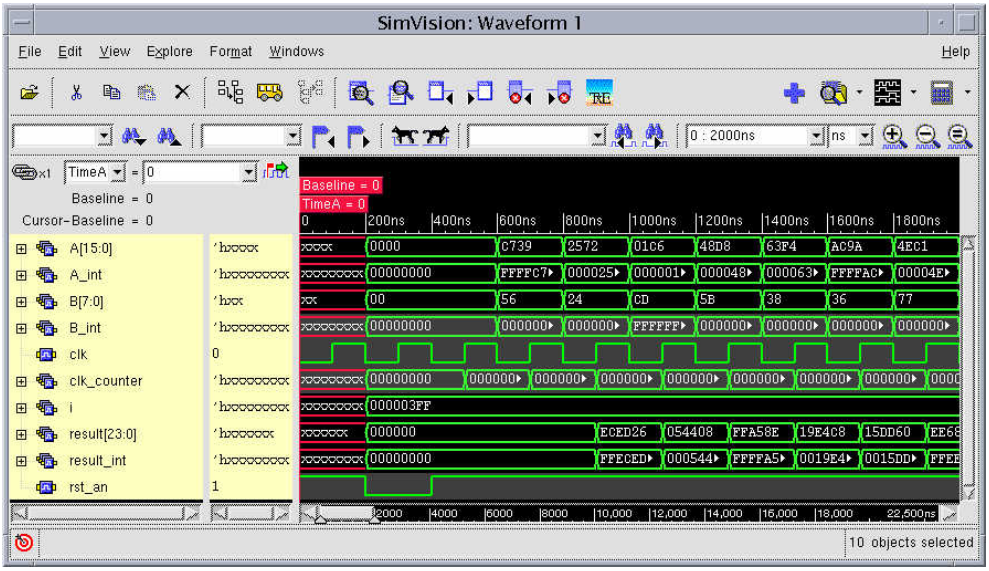


Figure 4: Simvision Waveforms for Signed Multiplier

```
ncverilog ../tb/rtlsim.v
```

The simulation will run as before, but will record the waveforms in the rtlsim subdirectory.

5.3.2 Viewing with SimVision

To view the waveforms, we use Cadence's SimVision [6]. From *multiply/sim*, issue the command:

```
simvision rtlsim &
```

This launches the tool, loads the rtlsim database, and returns the command prompt. The tool opens to the design-browser. Expand the signal hierarchy by highlighting the rtlsim folder, and selecting Edit - Explode. Select the tb icon. Note how the signals are displayed in the viewer. Chose 'Select - All' from the menu, and click on the waveform icon to view the selected traces (Figure 4). In the waveform viewer you can zoom-in and out, pan around, go to specific time periods, etc... As in many graphical systems, there are many ways to perform any task and it is usually easiest to learn through exploration.

If there is a particular waveform setup that you wish to record, you can save a Command script from the file menu. Note that this only saves the Setup —

[6] The previous version was called Signalscan and is still available.

such as the list of signals, cursors, zoom settings, etc... — but does NOT save the underlying signal data.

5.4 Running Gate-Level Simulations

Gate level simulations are run the same way as the RTL simulation. When running a gate-level simulation, however, you must be sure to point the simulator to the verilog models for the standard cells. Looking at *tb/gatesim.v*, this is done through a *'include* statement. Also, we typically want any gate-level waveforms to be stored in a seperate waveform database - and so the *$shm_open* uses a different filename.

The final difference in gate-level simulation includes the use of the *$sdf_annotate* system function. This function reads the design's timing data from an SDF (Standard Delay Format) file and applies it to the simulation. As the design is pulled further through the ASIC flow, the SDF file, and thus the timing in the simulation, becomes more accurate. If a specific SDF file is not yet available for the design, unreliable default settings are applied for gate-delays and the t_{cq} of a flip-flop via the digflow/vstlib/stdcells.sdf file.

6 Quick Synthesis

Cadence and Synopsys are the two primary providers of ASIC synthesis tools. Synopsys' Design Compiler (DC) has long been the standard, but Cadence's Builgates and Physical Synthesis (PKS) tools have recently emerged as a comperable, lower cost, solution.

For the purpose of this tutorial we will focus on Cadence tools, but we'll also introduce you to basic synthesis in Synopsys' DC.

The Cadence tool-set can be subdivided into 3 classes:

- Buildgates (BG) - Basic synthesis tool. Started with *bg_shell*.
- Buildgates Extreme (BGX) - Adds advanced synthesis techniques for datapath components. Started with *bgx_shell*.
- Physical Synthesis (PKS) - Adds physical awareness to BGX. Started with *pks_shell*.

All 3 flavours have the same interface, but with different capabilities. The original Buildgates is highly crippled and generates very poor results. For normal synthesis, BGX is the flavour to use, but, if the design is timing critical or floorplanning is required then PKS is the appropriate tool.

Often during initial design phases, area and timing estimates are required long before a project is ready for layout. Tables 2 and 3 list the required commands to quickly synthesize an RTL or Behavioral design using the Cadence or Synopsys tools.

Start the tools from their respectively directories (*multiply/pks* and *multiply/syn*). In the GUI version of PKS, the command prompt is available along the bottom of the screen (Figure 5). To get the command prompt in Design Analyzer (which is the GUI version of dc_shell), select *Setup - Command Window* from the Menu bar (Figure 6).

Following the commands listed in Tables 2 and 3, synthesize the signed multiplier in both tools.

By examining the generated reports, try to compare the results in terms of speed and area before we go further into the details. Exit the tools using either the GUIs, or the quit command.

6.1 Scripting Repeated Commands

Throughout the industry, GUI interfaces are rarely used. Instead, scripts are used to automate common processes. This not only reduces check-out time of licences, but ensures consistency among designs.

When either of the tools are run, they log executed commands in either *ac_shell.log* (PKS) or *command.log* (DC). To create a script, simply record the useful commands in a file and then run them using: *source filename* in PKS, or *include filename* in DC. Synthesis scripts can become quite elaborate and often make use of parameters, variables and control constructs such as *if* statements and *for* loops.

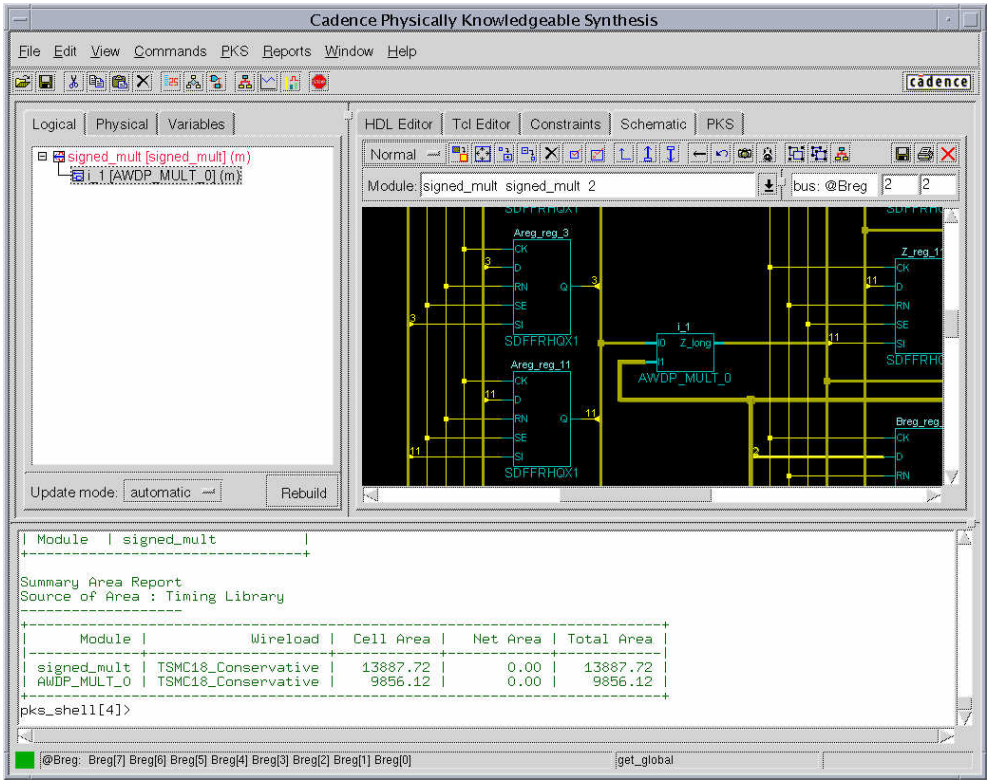


Figure 5: Screenshot of Multiplier in PKS

Start the tool:	pks_shell -gui &
Read Cell Libraries:	read_tlf ../../vstlib/cells_wc.tlf
Read Source Code:	read_verilog ../rtl/signed_mult.v
Generate Generic Hardware:	do_build_generic
Constrain the Clock:	set_clock myclk -period 10; set_clock_root -clock myclk clk
Map to Standard Cells:	do_optimize
Report the Area:	report_area
Report the Timing:	report_timing
Save Database:	write_adb adb/quicksynth.adb
Save Netlist:	write_verilog gates/quicksynth.v
Save Timing:	write_sdf -edges noedge sdf/quicksynth.sdf

Table 2: Quick Synthesis Commands In BGX/PKS

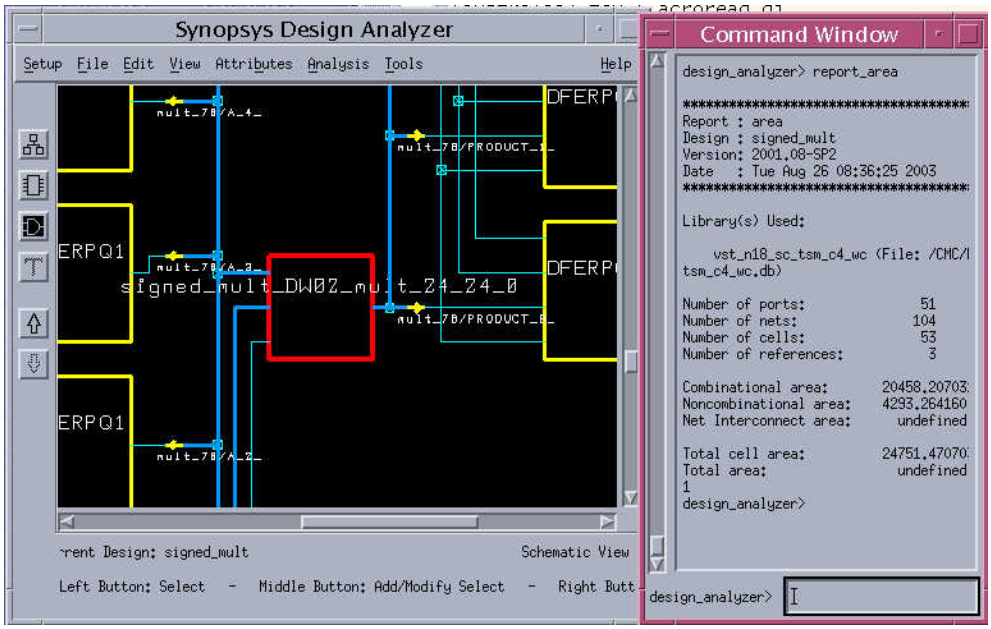


Figure 6: Screenshot of Multiplier in Design Analyzer

Start the tool:	design_analyzer &
Read Cell Libraries:	*Done automatically via the .synopsys_dc.setup startup file*
Read Source Code:	analyze -format verilog ../rtl/signed_mult.v
Generate Generic Hardware:	elaborate signed_mult
Constrain the Clock:	create_clock -name myclk -period 10 clk
Map to Standard Cells:	compile
Report the Area:	report_area
Report the Timing:	report_timing
Save Database:	write -output db/quicksynth.db
Save Netlist:	write -format verilog -output gates/quicksynth.v
Save Timing:	write_sdf -version 1.0 sdf/quicksynth.sdf

Table 3: Quick Synthesis Commands In Design Compiler

Examine the files *multiply/pks/tcl/quicksynth.tcl* and *multiply/syn/scr/quicksynth.scr*, and compare them with Tables 2 and 3. Note how values such as the clock period and root pin have been replaced with variables, allowing the script to be re-used for other designs.

From the *multiply/pks* directory, re-synthesize the multiplier automatically by issuing the command:

```
pks_shell -f tcl/quicksynth.tcl
```

This will start PKS in text mode, and immediately run the referenced script. Once synthesis is finished, it will end with the PKS command prompt. From there, you can issue further PKS commands or quit to the UNIX shell.

Remember, the GUIs are useful for learning and experimentation, but once issues are settled, scripts should be written to automatically generate your layout from RTL.

7 Getting Started with PKS

7.1 Environment Setup

In *digflow/setup.digflow.csh* the path is modified to include */CMC/tools/SOC23/tools/bin*. This is where the PKS executables reside.

7.2 The PKS Graphical User Interface (GUI)

Though the command interface is typically the best way to perform functions - this tutorial would be remiss without a few words about the PKS GUI. Notice from Figure 7 that the GUI is divided into three sections:

- The command window is used for entering tcl commands and monitoring the response.
- The Hierarchy Browser can be used to select signals or instances by name or logical relationship.
- Depending on the selection tab, the panel on the right can be used as a text editor (for HDL or tcl scripts), to setup timing constraints, or to view a schematic or physical layout.

Within the GUI, "Control-M" can be used to toggle a window section to full-size.

7.3 The PKS Command Interface (TCL)

Many of the EDA tools have been moving towards a common scripting language called TCL (pronounced "tickle"). The following are some basic points of the language:

- All variables in tcl are strings. Numeric conversion only occurs within functions, and are transparent to the programmer.

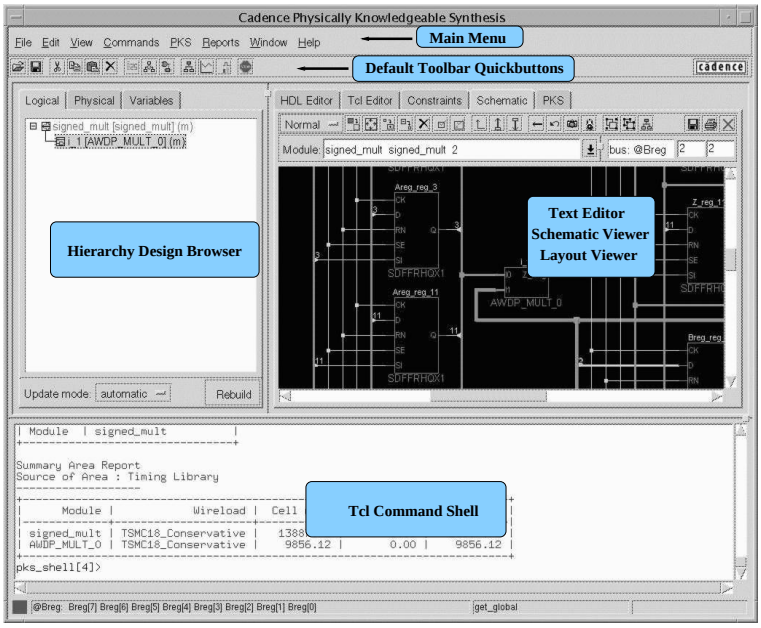


Figure 7: Layout of the PKS GUI

- Each line of a tcl statement is parsed into tokens, seperated by white space.
- The first token is the command, and all other tokens are options to that command
- Most commands work on, and return lists. Lists are arrays of words seperated by whitespace.
- To continue a command on the next line, end with the “ \ ” character.
- A good quick TCL reference can be found at: *http://panic.fluff.org/quickref/tcl.htm*

Additionally, within PKS, Cadence has defined over 200 synthesis related tcl proceduces. Keep in mind the following points:

- *help ** can be used to list all synthesis commands
- *help* <*command_name*> or <*command_name*> *-help* can be used to get information on any specific command.
- *help* <*keyword*> will list all commands related to that keyword (eg. help floorplan, help constraints, help dft).
- The TAB key can be used to complete a command name.

- Commands and switches do not need to be fully specified. (ie. *set_clock_root -clock myclk clkpin* and *set_clock_ro clkpin -cl myclk* are equivalent.)
- Most synthesis commands begin with one of:
  - get_ — to return an attribute or global variable (eg. get_fanin)
  - set_ — to set an attribute or global variable (eg. set_input_delay)
  - do_ — to perform some action (eg. do_build_generic, do_optimize)
  - report_ — report design values (eg. report_library, report_area, report_timing)
  - read_ — read an input file (eg. read_tlf, read_adb, read_sdf, read_verilog)
  - write_ — write to some output (eg. write_verilog, write_adb)

8 Digital Libraries

8.1 Logical Libraries

The first step in ASIC synthesis is to read the library data for standard cells and any macro blocks (eg. RAMS). The logical and timing data for the library may be provided in any of the following (roughly) equivalent forms[7]:

- .tlf - Cadence Timing Library Format
- .ctlf - Compiled (Binary) TLF
- .alf - Cadence Ambit Library Format
- .lib - Synopsys Library Format
- .db - Synopsys Database Format

These libraries contain:

- Design Rules
  - Maximum Slew
  - Maximum Load
  - Maximum Fanout
- Default Design Units (*typical unit*)
  - Capacitance (pF)
  - Delay (nS)
  - Area (um^2)

[7]Though tools can convert from one format to another, the process is typically buggy and frustrating.

  - Power (Dynamic - mW, Static - uW)
  - Resistance ($k\Omega$)

And then for Best, Worst, and Typical process conditions:

- Process, Temperature, Voltage Ratings
- Wireload Estimates — Average Interconnect RC vs Net Fanout
- Cell Data
  - Logical Function
  - Timing Delay Tables (Delay versus Load and Slew)
  - Pin Capacitance Estimates
  - Static and Dynamic Power Dissipation
  - Cell Area

Typically a library vendor will provide the cell data in seperate files for best, worst, and typical environments. Most circuit synthesis should be performed using the worst-case delays, however, best-case models must be considered when fixing hold-time violations. In the quick-synthesis of Section 6 we loaded only the worst case libraries, but for full synthesis we should merge the best and worst case libraries. After the merge operation, PKS will chose the fast or slow model appropriately.

To use the Artisan cells, and merge the best and worst case data into a library called "cells", issue the PKS command:

```
read_tlf -min ~/digflow/artlib/cells_bc.tlf \
         -max ~/digflow/artlib/cells_wc.tlf \
         -name cells
```

You can safely ignore the warnings "Missing 'Input( )' expression for LATCH( )".

After having read in the data, use the command *report_library -wireload -operating_cond* to view the global information listed in the library files. Using another variation of the report_library command we'll experiment with pattern matching. Issues the commands:

1. *report_library -help* to see the syntax of the command.
2. *report_library -cell NAND2** to list all variations of 2 input NAND gates.
3. *report_library -cell NAND*XL* to list all low-power (XL) NAND gates.
4. *report_library -cell NAND?X?* to list all NAND gates with un-inverted inputs.

8.2 Physical Libraries

As device sizes shrink, interconnect RC delays are becoming more significant than traditional gate delays. As such, wireload models — which assume an interconnect delay based on chip area and fan-out — are inaccurate. To decrease estimation errors, Physical Synthesis tools perform the placement and global routing of cells as part of the mapping process.

In order to perform the layout, the tool needs additional information. A .tf (technology file) or LEF[8] (Library Exchange Format) normally contains contains data regarding a process' parasitic information (ie. TSMC CMOSP18). And often a sperate LEF file contains the physical dimensions of the standard cells. In the case of the Artisan cells, all of the data has been combined in a single file and can be read using the command[9]:

```
read_lef ~/digflow/artlib/cells.lef
```

Unfortunately, there is some overlap between what is specified in the logical libraries, and what is in a LEF file. Specifically, thy both includes data regarding a cell's area and logical function. The dual-specifications can create inconsistencies. To ensure this is not the case, run the command:

```
check_library cells
```

Though all logical cells should have physical equivalents, there are rare cells — such as loading capacitors or antenna diodes — that may not have logical equivalents.

Scripts to load either the VST or Artisan cell libraries are provided as *tcl/load_vstlib.tcl* and *tcl/load_artlib.tcl*. These scripts also load additional libraries for the IO pads which are available. Once PKS starts, either of these can be run using *source tcl/<script_name.tcl>*

8.3 Section Summary

Key Points:

- Logical Library Formats can come in various forms — .tlf, .ctlf, .lib, .alf, .db.
- The units for Area, Resitance, Capacitance, Power, etc. are declared in the library file.
- Both Best AND Worst Case libraries need to be used to ensure proper operation.
- Wireload models are not accurate for high-speed, small geometry, designs

[8] Due to advanced Antenna information, there are two incompatible versions of the LEF. PKS (as well as SE) can only read the older version, whereas the router (wroute) can use the newer version.

[9] In cases where the process and cell information are in seperate files, the process information must be read first, and then the cell data is read with a *read_lef_update* command.

Read logical libraries	read_tlf -min cells_bc.tlf -max cells_wc.tlf -name cells
Report on logical libraries	report_lib
Read physical libraries	read_lef tech.lef
Additional physical libraries	read_lef_update morecells.lef
Check consistency of library	check_library cells

Table 4: Library Commands Summary

- Physical Libraries (Normally .LEFs) contain the data neccessary for layout.
- Consistency should be verified between physical and logical libraries.
- Wildcards (* and ?) can be used in TCL based pattern matching.
- The scripts *tcl/load_vstlib.tcl* and *tcl/load_artlib.tcl* are provided.

9 Reading and Constraining a Design

9.1 Reading Source Files

Once the libraries are loaded we need to read all of the project's verilog (or VHDL) source files. In large projects it is normally convenient to have one file reference all of the others with *'include* statements so that we only need to read the single file. In this case, the source file *../rtl/rtl_files.v* lists the rest of the project files.

To read the design:

```
read_verilog ../rtl/rtl_files.v
```

The design can also be read via the GUI by selecting *File — Open, Select Verilog, Select the File*. Any syntax errors are normally reported at this stage.

9.2 Generic Mapping

After having read the design into memory, we tranform it into generic hardware by running *do_build_generic* on the top level of the design. If there is an obvious top module implied by the code then no options need to be specified.

```
do_build_generic
```

During this step, it will inform you of any unsynthesizable code and outline how it is iterpreting various case statements or memory elements. You should ensure that there are no unwanted latches geneated in this stage.

If we want to build a design with non-default parameters, for example to build a 17x9 multiplier, the we specify them as a tcl list when the design is built. For example:

```
do_build_generic -param {{A_WIDTH 17} {B_WIDTH 9}}
```

9.3 Timing Constraints

In order to synthesize a design properly we need to inform the tool of all relevent boundary conditions and constraints. In large projects this is often the most complex part of the design.

In Section 6 we constrained the design merely by asserting the clock period. This assumes that our IO will not be a factor in timing analysis. If the critical path is internal to the circuit than this is okay for experimentation purposes. When the design is integrated into a larger project, however, we need to consider the boundary conditions. This involves setting:

- Input Delay - The time, after the clock edge, that it will take for the signal to reach the input port. This should be specified for both the best, and worst-case scenarios.
- External Delay - The delay a signal will experience outside of our design's boundary, before it reaches a register. Again, external delay should be specified for the best and worst-case.
- Port Capacitances - The capacitance that our design must drive, or any additional capacitance that must be driven by input drivers.
- Driving Cell/Resistance - This determines how fast the input driver can charge the port-capacitance, and is added on to the specified input delay.

When every IO of a design is registered, such as in our reference design (Figure 8), the constraints are simplified.

Unfortunately, this is not often the case and more elaborate constraints need to be applied. In Figure 9 we illustrate how to accomodate:

- Elaborate IO timing variations
- False and multi-cycle timing paths
- Clock Insertion Delay and Uncertainty

Constraints become even more complex when we need to consider data transfer across clock-domains. In this case, all clocks must be synchronously related, and the timing relationship between each domain is explicitly stated. Since constraint description can be quite involved, we will not go into such complications.

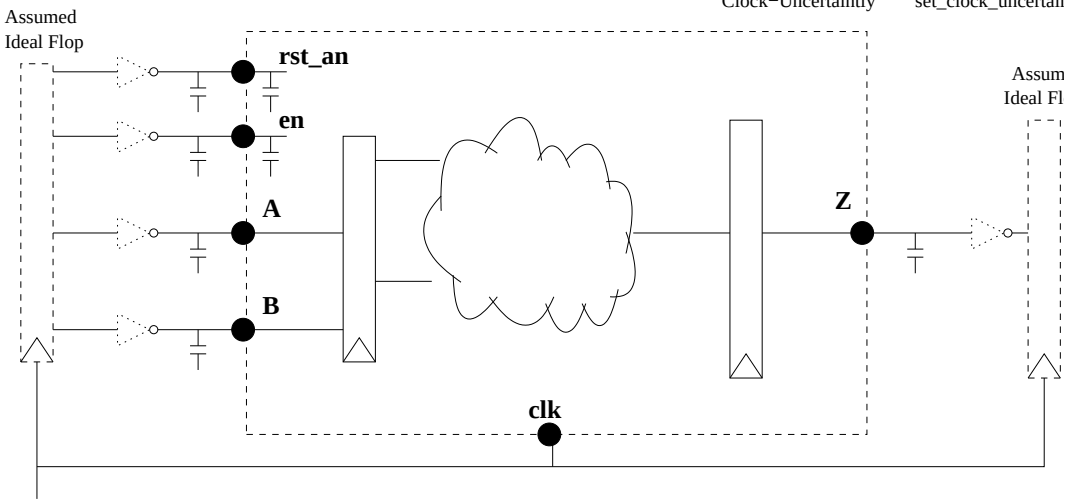
Following the commands outlined in Figure 8, finish properly constraining the tutorial design. Note the use of the command *find -inputs ** which returns a list of input object-ids. To return the names instead of object-ids, use *get_names [find -inputs *]*. Feel free to experiment with different variations of the *find* command as it can be very usefull in larger designs.

When finished, save the result in the Ambit Database Format (ADB):

```
write_adb adb/constrained.adb
```

Primary Constraints

Input		Output		Internal	
Delay	set_input_delay	Delay	set_external_delay	Clock Period	set_clock
Load	set_port_capacitance	Load	set_port_capacitance	False Path	set_false_path
Drive Cell	set_drive_cell	Resistance	set_wire_resistance	Multi−Cycle	set_cycle_addition
Resistance	set_drive_resistance	Fanout	set_fanout_load*	Clock−Insertion	set_clock_insertion
				Clock−Uncertaintly	set_clock_uncertai

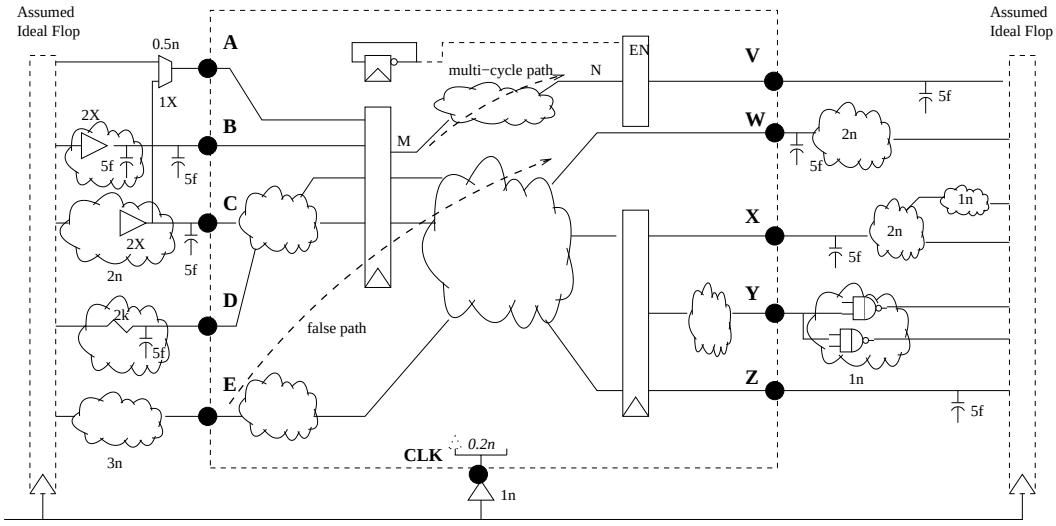


Create a symbolic clock called refclk with a 20nS period.	set_clock refclk −period 20.0
Bind clock waveform to the actual CLK pin.	set_clock_root −clock refclk clk
Assumes all inputs, other than the CLK, are driven by a 1X flop.	set_drive_cell −cell DFFX1 −pin Q [find −inputs * −noclocks]
Assumes infinite drive strength on the CLK and RST pins.	set_drive_resistance 0 clk
Set the best−case input delay to a fast tcq (200ps) Set the worst−case input delay to a slow tcq (500ps)	set_input_delay −clock refclk −early 0.2 [find −inputs * −noclock set_input_delay −clock refclk −late 0.5 [find −inputs * −noclocks]
Assume a load of 10fF (about 2 standard loads) for all ports.	set_port_capacitance 0.01 [find −ports *]
The data must arrive at least a setup time before the next edge. Ensures that we meet hold time−requirements.	set_external_delay −clock refclk −late 0.5 set_external_delay −clock refclk −early −0.1

Figure 8: Simple Constraints Specifications

Primary Constraints

Input		Output		Internal	
Delay	set_input_delay	Delay	set_external_delay	Clock Period	set_clock
Load	set_port_capacitance	Load	set_port_capacitance	False Path	set_false_path
Drive Cell	set_drive_cell	Resistance	set_wire_resistance	Multi−Cycle	set_cycle_addition
Resistance	set_drive_resistance	Fanout	set_fanout_load*	Clock−Insertion	set_clock_insertion_delay
				Clock−Uncertaintly	set_clock_uncertaintly




```
set_clock refclk −period 20.0
set_clock_root −clock refclk CLK

set_clock_insertion_delay −source −pin CLK 1.0
set_clock_uncertaintly −pin CLK 0.2

set_false_path −from E −to W
set_cycle_addition 1.0 −from [get_drive_pin M] −to [get_load_pin N]

set_input_delay −clock refclk −early 0.5 A
set_input_delay −clock refclk −late 2.5 A
set_input_delay −clock refclk 3.0 E
set_input_delay −clock refclk 2.0 C

set_drive_cell −cell MUX2X1 A
set_drive_cell −cell BUFX2 {B C}

set_drive_resistance 2.0 D
set_drive_resistance 0 {E CLK}

set_port_capacitance 0.010 B
set_port_capacitance 0.005 {C D V W X Z}

set_external_delay −clock refclk −early 2.0 X
set_external_delay −clock refclk −late 2.5 X
set_external_delay −clock refclk 1.0 Y
set_external_delay −clock refclk 2.0 W

set_port_capacitance [expr 2 * [get_cell_pin_load NAN2X1]] Y
```

Figure 9: More realistic Constraints

Read verilog sources	read_verilog ../rtl/rtl_files.v
Map to generic hardware	do_build_generic
Create a clock waveform	set_clock refclk -period 10
Bind the waveform to clk pin	set_clock_root -clock refclk clk
Set input drive strengths	set_drive_cell -cell DFFX1 -pin Q {A B en rst_an}
Set port loads	set_port_capacitance 0.01 {A B Z en rst_an}
Prepare infinite clock drive	set_drive_resistance 0 CLK
Set best case input delay	set_input_delay -early 0.1 {A B en rst_an}
Set worst case input delay	set_input_delay -late 0.5 {A B en rst_an}
Set best case external delay	set_external_delay -early -0.1 Z
Set worst case external delay	set_external_delay -late 0.3 Z
Save the design	write_adb adb/constrained.adb

Table 5: Constraints Command Summary

9.4 Section Summary

Once you've become familiar with the concepts of this chapter you can use or modify the PKS scripts *pks/tcl/load_rtl.tcl* and *pks/tcl/constrain_timing.tcl* or the DC scripts *syn/scr/load_rtl.scr* and *syn/scr/constrain_timing.scr* to load and constrain other designs.

Key Points:

- All Verilog (or VHDL) source files must be read into the tool.
- The top level module must be mapped to generic hardware.
- Basic timing constraints must be applied including:
  - Clock Period
  - Clock Root
  - Port Loading
  - Any Input/Output External Delays
- The *find* command can be used to return a list of object id's.
- The *get_names* command converts object-ids to names.
- An .adb file stores the complete design at any point.

10 Floorplanning

The floorplanning process takes a logical netlist and lays out the standard cells in groups of rows.

The 'chicken and the egg' phenomema is alive and well when it comes to floorplanning. We can't floorplan until we have a netlist, but we can't get an accurate netlist until we have an idea of the floorplan. The solution is to floorplan an initial netlist — but leave enough flexibility for optimization, and the addition of test-features, and clock buffers.

To get an idea of the available floorplanning options, issue the command:

```
report_floorplan_parameters
```

We will start with these, just to get an idea of how our design will eventually look. To generate the initial floorplan:

- Restart PKS, and load the appropriate libraries.
- Load (or recreate) the constrained design (*read_adb adb/constrained.adb*).
- Run the command *do_optimize* to compile and generate a default layout.
- View the generated floorplan in the GUI (Select the PKS tab in the Viewer)

From here, we see an initial estimate of the design's size and layout.

10.1 Power Planning

Though power-planning can be quite involved, we will touch on some of the more important aspects.

The essence of power planning is getting the VDD and GND rails of the standard cells connected, through low-resistance lines, to the external supply. When designing the power grid for a circuit we should keep in mind the following two rules:

1. Keep the average current density below technology limits. In the case of CMOSP18, this is 1mA/um of wire width for M1 through M5, and 1mA/um for M6.
2. Prevent IR drop from adversly affecting circuit performance. IR drop is caused when a rush of current is drawn through a high resistance line, causing a temporary supply voltage (or IR) drop. We should try to keep IR drop negligible - generally lower than 5% of the supply voltage (90mV).

In order to design our supply network, we need a reasonable estimate of the circuit's power and current consumption. Based on the clock period, and estimates of toggling activity, PKS (or DC, or SOC) can provide these figures. Keep in mind that the results we get are highly dependent on providing proper toggling densities. In PKS the toggle density is the number of transitions expected per ns.

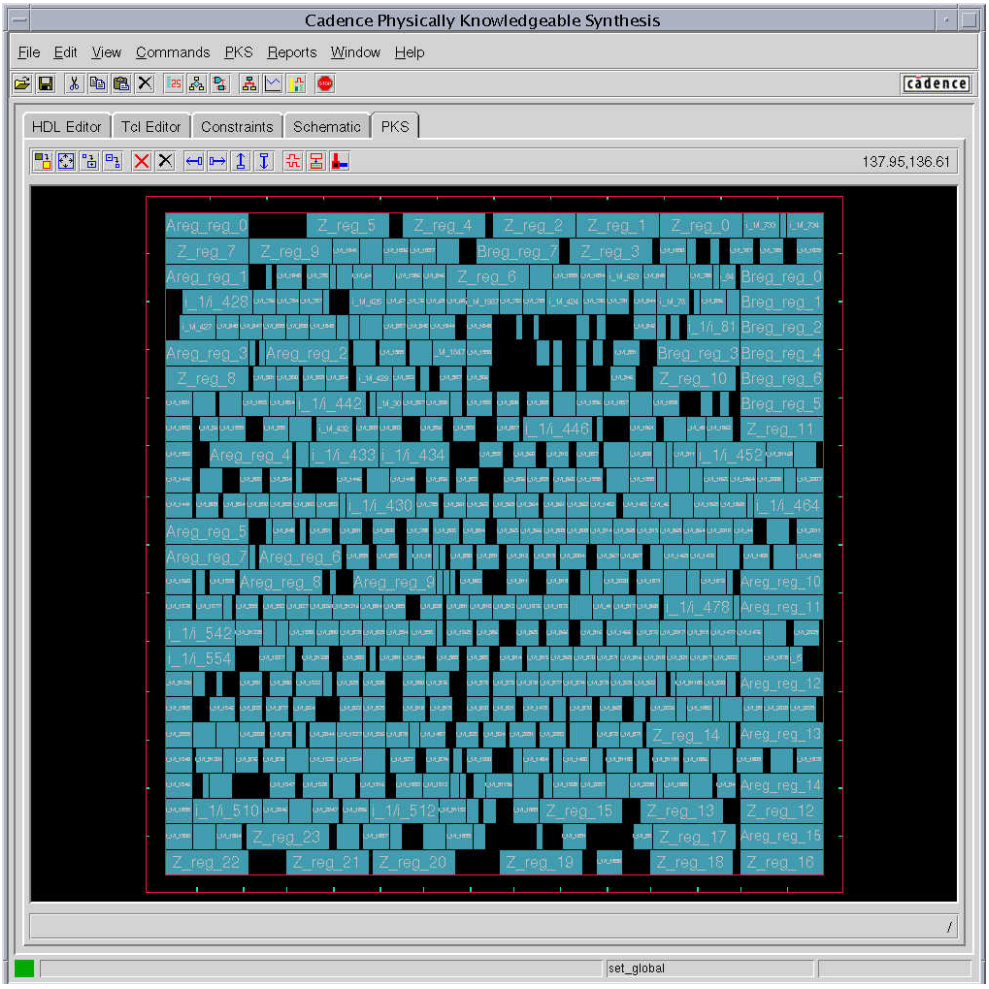


Figure 10: Result of an Initial Floorplan

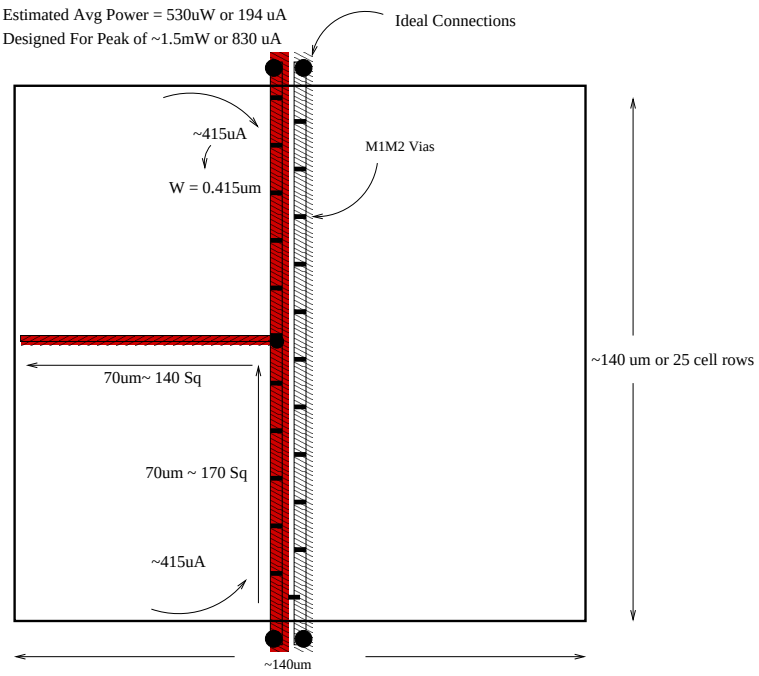


Figure 11: The powerplan as originally designed.

```
set power_default_toggle_rate 0.05
set_switching_activity -prob 1 -td 0 -pin {en, rst_an}
set_switching_activity -prob 0.5 -td 0.05 -pin {A B Z}
set_switching_activity -prob 0.5 -td 0.2  -pin clk
report_power
```

The result indicates that the circuit will consume an average of 0.53mW, before consideration of clock buffering and test-insertion. To accomodate these additions, and respect peak power conditions, we'll design for a system that consumes 1.5mW. At 1.8V, this means that the circuit will consume up to 830uA. Viewing the initial layout, it appears that there are about 25 rows of standard cells. If we assume roughly equal power distribution, each row will draw about 35uA. If we power the design with a single stripe stripe down the middle of the design, connected at both top and bottom then:

- The stripe must handle 830uA total. Referring to Figure 11, if evenly drawing from top and bottom sources, the current bottlneck will be 415uA. To satisfy current density limits, the power and ground supply stripes must be at least 0.415um wide.
- To determine IR drop through the supply network, the furthest distance the current will travel is to the midpoint of the chip on the M2 supply,

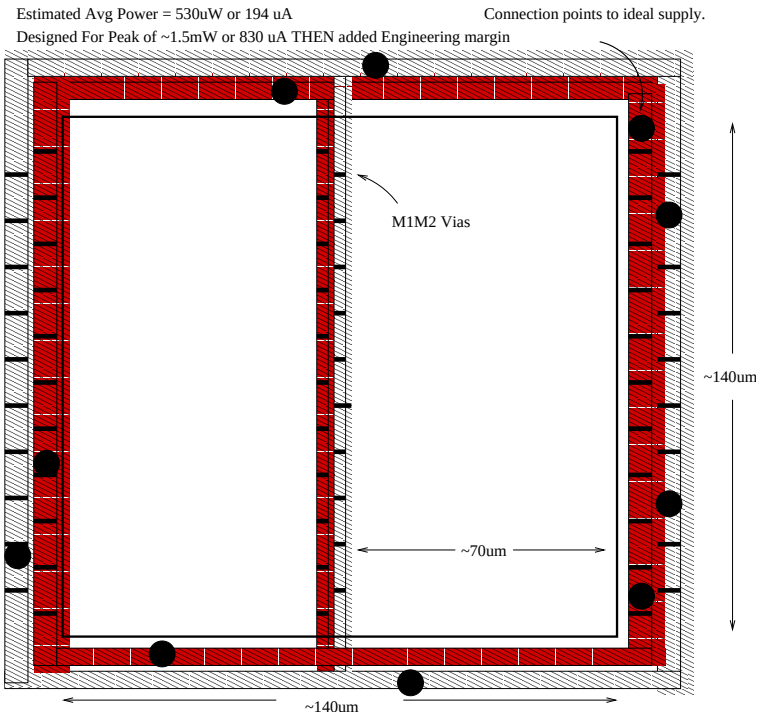


Figure 12: A powerplan with a ring and plenty of safety margin.

then across the row to an extreme edge on M1. From the initial layout this would be about 70um of 0.415um wide M2 + 70um of 0.8um wide M1. This is $170R_{sq_{M2}} + 87R_{sq_{M1}}$. Referring to the process documentation, $R_{sq_{Mx}} = 0.08\Omega$, wheras a typical VIA resistance is 6Ω. The maximum resistance this path would face would therefore be 26Ω. Even if all 800uA of the circuit's current were consumed by this single cell at the extreme edge of power, voltage drop would only be 21mV, well under the 90mV allowace.

- As the above calculations show, IR drop is not a problem, but we must ensure the supply rails are at least 0.415um wide to prevent self-heat/electromigration problems. Since the standard cell width is 0.66um, we'll use this width and spacing for our power stripes as well.

- There are tools available for more detailed analysis. Having used these relatively ballpark figures, as shown in Figure 12, we should add even more margin to the design — we'll add a power ring, with a width of 2*0.66um, and make multiple connections to the ideal supplies.

When the power grid is eventually in place, it will occupy a portion of the die which could otherwise be used for placing cells and routing signals. Though

PKS does not perform power routing, we need to inform it of these planned obstructions so that it can work around them. Issue the following command, and note that all horizontal specifications should be a multiple of the cell pitch (0.66um).

```
set_power_stripe_spec -direction vertical -layer METAL2 \
  -width 0.66 -start_from 69.96 \
  -number_stripes 1 \
  -net_spacing 0.66 \
  -net_name VDD VSS
```

To reserve space for the eventual power ring, we increase the core-to-boundary offsets. This is done through the *set_floorplan_paramaters* command as follows:

10.2 Rearranging the Layout

In older technologies with few metal layers, it was common to leave extra spacing between cells to permit routing. It is now common practice to create a dense sea of gates without spacing between rows of cells. So that we don't short VDD and GND rails, however, we must flip alternate rows. Also, because we have yet to add clock buffering, or test structures, we should allow extra space for them to be inserted later. To do this, we should reduce the target density.

To adjust the layout, issue the commands:

```
set_floorplan_param -fixed false
set_floorplan_param -flip -abut -row_utilization_initial 70
do_optimize
```

At this point, the design is assuming an ideal clock, and test related overhead has not been added. As such, timing will only get worse and so we should ensure that our requirements are met at this point. To perform a timing analysis, run:

```
report_timing
```

Try to identify the critical path in the design, and how close you are to meeting your timing goals.

11 Clock Tree Insertion

11.1 What is a clock tree?

Ideally the clock signal will arrive to all flip-flops at the same time. Due to variations in buffering, loading, and interconnect lengths, however, the clock's arrival is skewed. A clock-tree insertion tool evaluates the loading and positioning of all clock related signals and places clock buffers in the appropriate spots to minimize skew to acceptable levels. Some clock tree insertion tools, all from Cadence, include CTSGen, ctgen, and CTPKS. We will use CTPKS to create a clock tree within PKS.

11.2 Setting the Clock Tree Parameters

For the simplest clock tree insertion, we must tell PKS maximum clock skew can be tolerated. Since clock skew in normally more of a problem with respect to hold times this is typically derived from the difference between the best-case $t_{hold} - t_{cq}$ of a flop. Typically, for 0.18um designs, a skew of 50ps should be tolerable.

We also need to specify a minimum and maxiumum insertion delay for the buffer tree. Often, we will not be too concerned with these values and so we provide a wide range to give the tool more freedom.

To set-up the clock-tree generator, we use the following command:

```
set_clock_tree_constraints -pin clk \
    -min_delay 0 -max_delay 10 -max_skew 0.1
```

11.3 Building the Clock Tree

If the clock tree constraints are set before the *do_optimize* command, then a clock-tree will automatically be generated. To generate the clock-tree seperately, issue the command: *do_build_clock_tree -pin clk*.

To see the results of the clock-tree run, use the *report_clock_tree* command.