

Chapter 7

Cell Characterization

STANDARD CELLS must be *characterized* so that they can be understood and used by the various tools in the CAD flow. So far if you've designed schematic and layout views of your cells the tools understand the transistor netlist and the layout of the cells, but they don't understand the function of the cells in a way that the tools can use, and other information such as the input load, speed, and power of the cell has not been extracted in a way that the tools can understand. The synthesis tools coming up in Chapter 8, for example, need to know the logic function of the cell, the load that the cell input will present to a signal connecting to it, the speed of the cell under different input slope and output loading conditions, the power that the cell will consume, and the area of the cell in order to do a good job of synthesizing a behavioral description to a collection of standard cells. *Cell characterization* is a process of simulating a standard cell with an analog simulator to extract this information in a way that the other tools can understand. This can be done through specific analog simulation (using SpectreS) whose output you look at to generate the characterization data, or by using a library characterization tool. In this case we'll use SignalStorm from Cadence.

7.1 Liberty file format

We need to encode the cell characterization data in a standard format called **liberty** format which usually uses a **.lib** file extension. **Liberty** format is an ASCII file that describes a cell's characterized data in a standard way. This file is used both by the synthesis tools described in Chapter 8 and by the place and route tools in Chapter 10. The general form of a **liberty** file is shown in Figure 7.1. At this level of detail it simply describes the overall structure of the file. Each field has lots of detail that you can add by hand, or

```
/* General Syntax of a Technology Library */
library (nameoflibrary) {
... /* Library level simple and complex attributes */
... /* Library level group statements */
... /* Default attributes */
... /* Scaling Factors for delay calculation */

/* Cell definitions */

cell (cell_name) {
    ... /* cell level simple attributes */

    /* pin groups within the cell */
    pin(pin_name) {
        ... /* pin level simple attributes */

        /* timing group within the pin level */
        timing(){
            ... /* timing level simple attributes */
            } /* end of timing */

        ... /* additional timing groups */

    } /* end of pin */

    ... /* more pin descriptions */

} /* end of cell */

... /* more cells */

} /* end of library */
```

Figure 7.1: General format of a **liberty** (.lib) file

you can have SignalStorm generate for you. In general the SignalStorm approach is much easier if everything works. For some out of the ordinary cells, however, you may have to resort to your own simulations.

Liberty file format is, as you might guess, very complex with huge numbers of special statements that can describe all sorts of parameters that would be relevant to the different CAD tools that use this format to get information about the standard cells in the library. This chapter will touch on the most important from our flow's point of view, and give a number of examples, but in this case the full file format is too large and complex to include. This information should be enough to show you how to generate working **.lib** files for our flow. The format is described in full detail in documentation from Synopsys.

An example of the header information generated from a SignalStorm

characterization run is shown in Figure 7.2. This is all data above the *cell definitions* in the **.lib** file. Later we will see how to add additional information to this technology header to generate, for example, a simple model of wire loads that estimates the delay that will be caused by the RC delays in the wiring of the circuit once it is physically placed and routed. For now we can look at the information in the header and see what's going on. The **delay_model** says that the characterized delays will be described in **table_lookup** format. This means that the delays will be described as a table of delays simulated with different input and output conditions. In our characterizations this will be the input slope on one dimension of the table and output capacitive load on the other axis. The synthesis tool can use this information, and information about the connection of the cells that it is synthesizing, to estimate delays with reasonable accuracy (assuming that the tables have enough data, and that the data is accurate of course). There are other delay models possible, but **table_lookup** is by far the most widely used.

The **in_place_swap_mode** tells the tool that uses this file that cells whose **footprints** match can be used as replacements for each other. This allows us to have, for example, inverters with the same logic function but with different output drives swapped for each other as required. To prevent this swapping you would set this mode to be **no_swapping**.

The **unit attributes** should be fairly self-explanatory. They define the default units that various electrical parameters will be described in so that simple numbers can be used in the cell descriptions with the units assumed to be these defaults. The **thresholds** define where in the waveform **slews** and **delays** are computed. In this case the **rise** and **fall** times are computed at the 30-70% points in the input or output waveform. That is, delays are defined as being measured from 30% of the total rise or fall on the input to 70% of the rise or fall of the output. The percentages are based on vdd. The default operating conditions are set which are actually only really useful if multiple operating conditions are characterized (usually typical, max, and min) but, of course, that increases characterization time. Finally the format of the **lu_table_template** is defined for both delay and power computations. These are the *look up tables* that will contain the actual timing and power data. In this case we define 5x5 tables with input slope (**input_net_transition**) on the first axis and output capacitance (**total_output_net_capacitance**) on the second axis. The actual value of the input slope (in ns) and the output loading (in pF) will be defined for each look up table defined in the cell definition section.

Delays are measured with these percentages because measuring from the 50% point can result in a negative delay. This can happen if a gate has high gain so that a small change on the input causes the output to change rapidly. By the time the input has risen to 50%, the output has already changed.

```
library(foo) {

    delay_model : table_lookup;
    in_place_swap_mode : match_footprint;

    /* unit attributes */
    time_unit : "1ns";
    voltage_unit : "1V";
    current_unit : "1uA";
    pulling_resistance_unit : "1kohm";
    leakage_power_unit : "1nW";
    capacitive_load_unit (1,pf);

    slew_upper_threshold_pct_rise : 80;
    slew_lower_threshold_pct_rise : 20;
    slew_upper_threshold_pct_fall : 80;
    slew_lower_threshold_pct_fall : 20;
    input_threshold_pct_rise : 30;
    input_threshold_pct_fall : 70;
    output_threshold_pct_rise : 70;
    output_threshold_pct_fall : 30;
    nom_process : 1;
    nom_voltage : 5;
    nom_temperature : 25;
    operating_conditions ( typical ) {
        process : 1;
        voltage : 5;
        temperature : 25;
    }
    default_operating_conditions : typical;

    lu_table_template(delay_template_5x5) {
        variable_1 : input_net_transition;
        variable_2 : total_output_net_capacitance;
        index_1 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
        index_2 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
    }
    power_lut_template(energy_template_5x5) {
        variable_1 : input_transition_time;
        variable_2 : total_output_net_capacitance;
        index_1 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
        index_2 ("1000.0, 1001.0, 1002.0, 1003.0, 1004.0");
    }
}
```

Figure 7.2: Example technology header for a **liberty** (.lib) file

Wire Load Models

An increasingly important contributor to the overall delay in a circuit is the delay in the wires. For a simple model of wire delay it's not too hard to compute the wire's contribution to delay once the wiring is actually known (i.e. after place and route). But, if you're using information in the **.lib** file to estimate performance before you have physical placement and wiring, how do you estimate the wire delays? One way is to use an educated guess at the average length of a wire for a broad class of circuits of different sizes. It turns out that, not surprisingly, average wires tend to get longer the larger the circuit is. When a program like **Synopsys** generates a circuit from a behavioral description, it has information from the **.lib** file about the size of the cells, so it has a good estimate of what the final size of the circuit might be. Using that circuit size, a *wire load model* can be used to estimate (roughly) what the wire delay will be for the final circuit.

A good wire load model might come from a company where many designs have been done using a particular cell library, and statistics have been kept on the average wire length for circuits build using those libraries. We don't have such good statistics for our library so we'll have to guess. The wire load model in Figures 7.3 and 7.3 define a wire load model derived from some rough guesses about wire length and some information about the resistance and capacitance of general wiring in our 0.6μ CMOS process. These models would be placed in the **.lib** file right after the technology information in Figure 7.2.

7.1.1 Combinational Cell Definition

The cell definition (with some details deleted so it fits on a page) that was derived from the **SignalStorm** characterization for an inverter is shown in Figure 7.5. You can see that the name of the cell is **INVX1** and the overall attributes are defined. Then each pin is described. Pin **A** is an input pin so its attributes are all related to the capacitance that this pin presents to any signal that connects to it. Remember the default units earlier in the file so these numbers are in units of pF.

The output pin **Y** is much more interesting. Delays occur at the output pins with respect to changes on the input pins and hence only output pins have timing information for combinational cells. Because it doesn't feed back into the cell the capacitance that output pin **Y** sees because of this cell is 0. The **function** of the output defines the logical behavior of the cell. In this case the function is **!A** or the inverse of the **A** input. The **timing()** section defines how the output timing relates to changes on each input pin. There's only one input pin in this case but for a gate with multiple inputs the

```
/* Because the cell area is in units of square microns, all the      *
 * distance units will be assumed to be in microns or square microns. */

/* fudge = correction factor, routing, placement, etc. */
fudge = 1.0;

/* cap = fudge * cap per micron                                     *
 * I assume cap is in capacitance units per micron *
 * (remember that our capacitance unit is 1.0pf) */
cap = fudge * 0.000030; /* .03ff/micron for avg metal */
res = fudge * 0.00008 /* 80 m-ohm/square, in kohm units */

/* length_top = the length of one side of a square die (in our case, *
 * a 4 TCU die of 2500u on a side of core area) length_10k = the *
 * length of one side of a block containing 10k gates (I'll assume *
 * this is a core of a single TCU which is 900u on a side) */
length_10k = 900;
length_top = 2500.0;

/* sqrt(5000/10000) = .71 *
 * sqrt(2000/10000) = .45 etc */
length_5k = length_10k * 0.71;
length_2k = length_10k * 0.45;
length_1k = length_10k * 0.32;
length_500 = length_10k * 0.22;
```

Figure 7.3: Define some variables in the **.lib** file for wire load calculations

timing should be characterized based on each **related_pin** (i.e. each input pin). This means that there would be multiple **timing** sections for each output pin in that case. Within each timing group the **related_pin** attribute identifies the input pin to which the timing is related.

The pin timing is defined for **cell_rise**, **rise_transition**, **cell_fall** and **fall_transition**. Recall that the look up tables were defined in the header to have input slope in ns on **index_1**, and output loading in pF on **index_2** and you can see the actual values for these parameters on each look up table. These values are chosen by you at the time you do the characterization and are based on the estimated values that you expect the cells to see in an actual circuit. The values in some of the lookup tables have been deleted so that the figure will fit on a page. The power is also calculated for each output pin and for each related input pin.

Cell function is described (as seen in Figure 7.5) as a **function** attribute on each output pin of the cell. The Boolean function of a combinational cell can be described using the syntax in Figure 7.6. This is a standard format called **EQN** format and is used by a number of tools, not just **liberty** files. Description of sequential cells like flip-flops is more complex.

```

wire_load("top") {
  resistance : res ;
  capacitance : cap ;
  area : 1 ; /* i.e. 1 sq micron */
  slope : length_top * .5 ;
  fanout_length(1,2500); /* length */
  fanout_length(2,3750); /* length * 1.5 */
  fanout_length(3,5000); /* length * 2 */
  fanout_length(4,5625); /* length * 2.5 */
  fanout_length(5,6250); /* length * 2.5 */
  fanout_length(6,6875); /* length * 2.75 */
  fanout_length(7,7500); /* length * 3 */
}

wire_load("10k") {
  resistance : res ;
  capacitance : cap ;
  area : 1 ;
  slope : length_10k * .5 ;
  fanout_length(1,900); /* length */
  fanout_length(2,1350); /* length * 1.5 */
  fanout_length(3,1800); /* length * 2 */
  fanout_length(4,2025); /* length * 2.5 */
  fanout_length(5,2250); /* length * 2.5 */
  fanout_length(6,2475); /* length * 2.75 */
  fanout_length(7,2700); /* length * 3 */
}

wire_load("5k") {
  resistance : res ;
  capacitance : cap ;
  area : 1 ;
  slope : length_5k * .5 ;
  fanout_length(1,639); /* length */
  fanout_length(2,959); /* length * 1.5 */
  fanout_length(3,1278); /* length * 2 */
  fanout_length(4,1439); /* length * 2.5 */
  fanout_length(5,1598); /* length * 2.5 */
  fanout_length(6,1757); /* length * 2.75 */
  fanout_length(7,1917); /* length * 3 */
}

/* define how the wire loads are selected based on total circuit area */
wire_load_selection (foo) {
  wire_load_from_area ( 0, 3000000, "5k");
  wire_load_from_area (3000000, 7000000, "10k");
}

default_wire_load_mode : enclosed ;
default_wire_load : "top" ;
default_wire_load_selection : "foo" ;
/* end of wire_load calculation */

```

Figure 7.4: Use the wire load variables to compute wire load models based on wire RC

```
/* ----- *
 * Design : INVX1 *
 * ----- */
cell (INVX1) {
  cell_footprint : inv;
  area : 129.6;
  cell_leakage_power : 0.0310651;
  pin(A) {
    direction : input;
    capacitance : 0.0159685;
    rise_capacitance : 0.0159573;
    fall_capacitance : 0.0159685; }
  pin(Y) {
    direction : output;
    capacitance : 0;
    rise_capacitance : 0;
    fall_capacitance : 0;
    max_capacitance : 0.394734;
    function : "(!A)";
    timing() {
      related_pin : "A";
      timing_sense : negative_unate;
      cell_rise(delay_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( \
          "0.147955, 0.218038, 0.359898, 0.922746, 1.76604", \
          "0.224384, 0.292903, 0.430394, 0.991288, 1.83116", \
          "0.365378, 0.448722, 0.584275, 1.13597, 1.97017", \
          "0.462096, 0.551586, 0.70164, 1.24437, 2.08131", \
          "0.756459, 0.874246, 1.05713, 1.62898, 2.44989"); }
      rise_transition(delay_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( ... ); }
      cell_fall(delay_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( ... ); }
      fall_transition(delay_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( ... ); }
    } /* end timing */
    internal_power() {
      related_pin : "A";
      rise_power(energy_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( ... ); }
      fall_power(energy_template_5x5) {
        index_1 ("0.06, 0.18, 0.42, 0.6, 1.2");
        index_2 ("0.025, 0.05, 0.1, 0.3, 0.6");
        values ( ... ); }
    } /* end internal_power */
  } /* end Pin Y */
} /* end INVX1 */
```

Figure 7.5: Example **liberty** description of an inverter (with lookup table data mostly not included)

Operator	Description
'	invert previous expression
!	invert following expression
^	logical XOR
*	logical AND
&	logical AND
space	logical AND
+	logical OR
	logical OR
1	signal tied to logic 1
0	signal tied to logic 0

Figure 7.6: Function description syntax for **Liberty** files (EQN format)

7.1.2 Sequential Cell Definition

Sequential cells in general have events driven by the clock (or gate) signals. Thus these cells are characterized by *setup* and *hold* times as well as the normal propagation delay and rise/fall times. Because of this the timing measurements required for a sequential standard cell are:

- Setup time with respect to the clock when the input data is rising (T_{slh})
- Setup time with respect to the clock when the input data is falling (T_{shl})
- Hold time with respect to the clock when the input data is rising (T_{hlh})
- Hold time with respect to the clock when the input data is falling (T_{hhl})
- Propagation delay with respect to the input when output is falling (T_{phl})
- Propagation delay with respect to the input when output is rising (T_{plh})
- Rise time (of output) per unit load (T_{rise})
- Fall time (of output) per unit load (T_{fall})

Figure 7.7 shows the measurement of setup and hold times for a positive edge triggered flip flop or for a negative level gated latch (for both of these

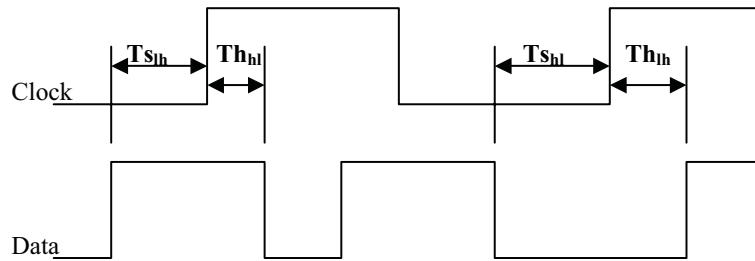


Figure 7.7: Setup and hold timing relative to the clock

sequential elements we make setup and hold time measurements with respect to the positive edge of the clock). The setup and hold times should be measured for all the input signals with respect to the clock. The propagation times and rise/fall times should be measured for all the outputs separately with respect to each of the input signals including clock.

Sequential cell functionality is identified by an **ff** group, **latch** group or **statetable** group statement inside the cell group.

ff group syntax

The **ff** group describes a flip flop (an edge triggered memory element) in a cell. General syntax of an **ff** group is

```
ff ( variable1_name , variable2_name ) {  
  clocked_on : "Boolean expression" ;  
  next_state : "Boolean expression" ;  
  clear : "Boolean expression" ;  
  preset : "Boolean expression" ;  
  clear_preset_var1 : L | H | N | T | X ;  
  clear_preset_var2 : L | H | N | T | X ;  
}
```

Variable1_name is the name of the variable whose value is the state of the non-inverting output of the flip flop. This can be considered as the 1 bit storage of the flip flop. This is the internal Q (for example). **Variable2_name** is the name of the variable whose value is the state of the inverting output of the flip flop. This is internal value of Qbar. It is these variables that are associated with the function attribute of the output pin group to describe the flip flop cell functionality. Both of the above variables should be specified even if either of the two is not connected to a primary

output pin. Valid variable names are anything except the pin names of the cell.

The **clocked_on** attribute identifies the active edge of the clock signal and is required in all **ff** groups. The **next_state** attribute is a Boolean expression that determines the value of **variable1** at the next active transition of the **clocked_on** attribute. The Boolean expression is a function of the cell inputs and the **variable1** attribute (but never of **variable2**). The next state attribute is required in all **ff** groups.

The **clear** attribute gives the active value of the clear input and is optional. The **preset** attribute gives the active value of the preset input and is also optional.

The **clear_preset_var1** attribute determines the value of **variable1** when both **clear** and **preset** are active at the same time. This attribute can be present only if, and is required if, both **clear** and **preset** attributes are present in the **ff** group.

The **clear_preset_var2** attribute determines the value of **variable2** when both **clear** and **preset** are active at the same time. Similar to **clear_preset_var1**, this attribute also can be present only if, and is required if, both **clear** and **preset** attributes are present in the **ff** group. Valid values for **clear_preset_var1** and **clear_preset_var2** are:

L for low or '0'
H for high or '1'
N for no change
T for toggle
X for unknown

The flip-flop cell is activated whenever **clear**, **preset**, or **clocked_on** changes. To sum up the above, Figure 7.8 shows the **ff** declaration for a JK flip-flop with asynchronous active low clear and preset and a D flip flop with synchronous active low clear.

latch group syntax

The **latch** group describes a latch (a level sensitive memory element) cell. General syntax of latch group is

```
latch ( variable1_name , variable2_name ) {  
  enable : "Boolean expression" ;  
  data_in : "Boolean expression" ;  
  clear : "Boolean expression" ;  
}
```

```

ff(IQ,IQN) {
clocked_on : "CLK" ;
next_state : "(J K IQ') + (J K') + (J' K' IQ)" ;
clear : "CLR'" ;
preset : "SET'" ;
clear_preset_var1 : X ;
clear_preset_var2 : X ;
}

ff (IQ, IQN) {
next_state : "D * CLR'" ;
clocked_on : "CLK" ;
}

```

Figure 7.8: **ff** descriptions for a JK and a D flip flop

```

preset : "Boolean expression" ;
clear_preset_var1 : L | H | N | T | X ;
clear_preset_var2 : L | H | N | T | X ;
}

```

The **variable1_name**, **variable2_name**, **clear**, **preset**, **clear_preset_var1**, and **clear_preset_var2** have the same meanings as that in the **ff** group discussed above.

The **enable** attribute identifies the active level of the clock signal, and the **data_in** value is the name of the data signal if it is used. The **data_in** and **enable** attributes are optional in a **latch** group, but if one of them is used, the other also must be used. The **latch** cell is activated when either of **clear**, **preset**, **enable**, or **data_in** changes. Examples of two latch cells are shown in Figure 7.9. The first is a D-latch with active-high enable and active-low clear. The second is a set-reset (SR) latch with active-low set and reset signals.

statetable group syntax

The **statetable** group describes functionality of more complex sequential cells. A state table is a sequential lookup table that specifies new values to internal nodes based on the current internal node values and inputs. The general syntax of a **statetable** group is

```

statetable( "input node names", "internal node names" ) {
table : "input node values : current internal values : next internal values,\
input node values : current internal values : next internal values,\
      :
      :
      :
}

```

```

latch(IQ, IQN) {
    enable : "CLK" ;
    data_in : "D" ;
    clear : "CLR'" ;
}

latch(IQ, IQN) {
    clear : "S'" ;
    preset : "R'" ;
    clear_preset_var1 : L ;
    clear_preset_var2 : L ;
}

```

Figure 7.9: **latch** descriptions for a D latch and SR latch

```

statetable ("J      K      CLK      CLR", "IQ" ) {
    table: "-      -      -      L      :      -      : L, \
           -      -      ~F     H      :      -      : N, \
           L      L      F      H      : L/H     : L/H, \
           H      L      F      H      : -       : H, \
           L      H      F      H      : -       : L, \
           H      H      F      H      : L/H     : H/L" ;
}

```

Figure 7.10: **statetable** description of a JK flop flop

```

input node values : current internal values : next internal values";
}

```

The valid values for inputs and internal variables are

L for low or '0'
 H for high or '1'
 F for falling or negative edge
 R for rising or positive edge
 N for no change
 T for toggle
 X for unknown

It's possible in a sequential library cell to have a **ff** or **latch** group along with a **statetable** group. But no sequential library cell can have more than one **statetable**. Also one needs to be careful not to have conflicting **ff** or **latch** groups along with a **statetable** group. An example of a **statetable** group for a JK flip flop with active-low asynchronous clear and negative edge clock is shown in Figure 7.10.

The timing attributes of a sequential cell are more complex than for a combinational cell. They are:

timing_type This attribute distinguishes a combinational and sequential cell. If this attribute is not assigned, the cell is considered combinational. The general format of this attribute is as shown below and is included in the input and/or output pin timing group of the cell. The syntax is **timing_type : value ;**

Some of the valid values for sequential timing arcs and their meaning are as follows:

rising_edge Identifies a timing arc whose output pin is sensitive to a rising signal at the input pin.

falling_edge Identifies a timing arc whose output pin is sensitive to a falling signal at the input pin.

preset A preset arc implies that you are asserting a logic 1 on the output pin when the designated **related_pin** is asserted.

clear A clear arc implies that you are asserting a logic 0 on the output pin when the designated **related_pin** is asserted.

hold_rising Designates the rising edge of the **related_pin** for the hold check.

hold_falling Designates the falling edge of the **related_pin** for the hold check.

setup_rising Designates the rising edge of the **related_pin** for the setup check on clocked elements.

setup_falling Designates the falling edge of the **related_pin** for the setup check on clocked elements.

timing_sense This attribute describes the way an input pin logically affects an output pin. Usually the **timing_sense** is included in the output pin timing group of a cell and is derived automatically from the logic function of a pin. This attribute is assigned to override the derived value or while defining a non combinational element and the general format is **timing_sense : value;**

The valid values for **timing_sense** attribute and their meanings are as follows:

positive_unate A function is said to be positive unate if a rising change on an input variable causes the output function variable to rise or not change and a falling change on an input variable causes the output function variable to fall or not change. For example, the output function of an AND gate is positive unate.

negative_unate A function is said to be negative unate if a rising change on an input variable causes the output function variable to fall or not change and a falling change on an input variable causes the output function variable to rise or not change. For example, the output function of a NAND gate is negative unate.

non_unate The **non_unate** timing sense represents a function whose output value change cannot be determined from the direction of the change in the input value. For example, the output function of an XOR gate is **non_unate**.

Also note that if a **related_pin** is an output pin, the **timing_sense** attribute for that pin must be defined.

clock This attribute is optional and indicates whether an input pin is a clock pin. The syntax is **clock : true;** or **clock : false;**

min_period This is an optional attribute on the clock pin of a flip flop or latch specifies the minimum clock period required for the input pin. The syntax is **min_period : value;**

A D-type edge triggered flip flop can be described in terms of all these attributes. The **.lib** description (with the data in the lookup tables missing so that it will fit on a page) is shown in Figure 7.11. Note that in this case the setup and hold timing are defined as scalar types, but they could just as easily (well, not really just as easily because there would be extra simulation involved) be described in a lookup table too. The first part of the flip flop description is an **ff** block describing the functionality. Then the pin descriptions follow. The **D** input pin had setup and hold timing. The **clk** pin is defined as a clock, and the **Q** pin has rise and fall timing described as lookup tables and has timing described with both **clk** and **clr** as related pins. The timing values are defined as 5 x 5 lookup tables as seen in Figure 7.5.

7.1.3 Tristate Cell Definition

The major difference of a tristate cell to other cells is that a tristate cell output can take a value of **Z** as well as **1** and **0**. The output goes to the **Z** value when the cell enable pin is de-asserted or when the tristate condition (generally a Boolean expression) is true. Electrically this **Z** value is a high impedance value on the output. It is as if the output drivers are disconnected from the output wire. It is also worthwhile to note that both combinational tristate cells as well as sequential tristate cells exist.

The type of measurements required for tristate cells are the same as any other combinational or sequential cells. However when it comes to

```

/* positive edge triggered D-flip flop with active low reset */
cell(dff) {
  area : 972;
  cell_footprint : "dff";
  ff("IQ", "IQN") {
    next_state : " D ";
    clocked_on : " G ";
    clear : " CLR' ";
  }
  pin(D) {
    direction : input;
    capacitance : 0.0225;

    timing() { /* hold time constraint for a rising transition on G */
      timing_type : hold_rising;
      rise_constraint(scalar) { values("-0.298"); }
      fall_constraint(scalar) { values("-0.298"); }
      related_pin : "G";
    }
    timing() { /* setup time constraint for a rising transition on G */
      timing_type : setup_rising;
      rise_constraint(scalar) { values("0.018"); }
      fall_constraint(scalar) { values("0.018"); }
      related_pin : "G";
    }
  } /* end of pin D */
  pin ( CLK ) {
    direction : input;
    capacitance : 0.0585;
    clock : true;
  } /* end of pin CLK */
  pin ( CLR ) {
    direction : input;
    capacitance : 0.0135;
  } /* end of pin CLR */
  pin ( Q ) {
    direction : output;
    function : "IQ";

    timing () { /* propogation delay from rising edge of CLK to Q */
      timing_type : rising_edge;
      cell_rise(lu5x5) { values( "..." );}
      rise_transition(lu5x5) { values( "..." );}
      cell_fall(lu5x5) { values( "..." );}
      fall_transition(lu5x5) { values( "..." );}
      related_pin : "CLK";
    } /* end of Q timing related to CLK */

    timing () { /* propogation delay from falling edge of clear to Q=0 */
      timing_type : clear;
      timing_sense : positive_unate;
      cell_fall(lu5x5) { values( "..." );}
      fall_transition(lu5x5) { values( "..." );}
      related_pin : "CLR";
    } /* end of Q timing related to CLR */
  } /* end of pin Q */
} /* end of cell dff */

```

Figure 7.11: Example D-type flip flop description in **.lib** format

measurement of the tristate output pin parameters with respect to the cell enable pin (or pin related to the tristate condition), one needs to remember that either the initial output value or the final output value will be **Z**. Also two sets of measurements are required with respect to the cell enable pin: one when the cell is being enabled (output going to **1** or **1**) and one when the cell is being disabled (output going to **Z**). This makes measurement of time-to-reach-tristate and time-to-start-driving a little tricky. If the outputs are not connected to capacitors the analog simulator generally simulator puts the output of a tri-stated device at around 2.5v for a 5v process, so we can measure from this point when measuring the time to or from a tristate output state (or some percentage of that point if you're using a 10% to 90% measurement, for example).

Special attributes for a tristate gate include

three_state This attribute defines a tristate output pin in a cell. The syntax is **three_state : <Boolean-expression> ;**

The Boolean expression is the condition to make the corresponding output pin to go to tristate or high impedance condition.

three_state_enable This is the value the **timing_type** attribute should take for the tristatable output pin in the “cell being enabled” timing group. The **cell_rise** and **rise_transition** will correspond to the **Z to 1** transition, and the **cell_fall** and **fall_transition** will correspond to the **Z to 0** transition.

three_state_disable This is the value then **timing_type** attribute should take for the tristatable output pin in the “cell being disabled” timing group. The **cell_rise** and **rise_transition** will correspond to the **0 to Z** transition, and the **cell_fall** and **fall_transition** will correspond to the **1 to Z** transition.

An example of a tristate inverter is shown in Figure 7.12. Note that the enable and disable timing matrix templates are different than the “regular” delay value matrix template. The indices of these templates should have been defined in the technology header of the **.lib** file (see Figure 7.2), or the actual index values can be defined before each **values** statement in the **timing** block (as they were for the inverter in Figure 7.5).

7.2 Cell Characterization with SignalStorm

SignalStorm is a tool from Cadence that uses Spectre to characterize cells and outputs the results in a format that can be converted into **liberty**

```

/* Enabled inverter or tristate inverter */
cell(eninv) {
    area : 324;
    cell_footprint : "eninv";
    pin(A) {
        direction : input;
        capacitance : 0.027;
    } /* end of pin A */
    pin(En) {
        direction : input;
        capacitance : 0.0135;
    } /*end of pin En */
    pin(Y) {
        direction : output;
        function : "A'";
        three_state : "En'";
        timing () {
            timing_sense : negative_unate;
            related_pin : "A";
            cell_rise(lu5x5) { values( " ... " );}
            rise_transition(lu5x5) { values( " ... " );}
            cell_fall(lu5x5) { values( " ... " );}
            fall_transition(lu5x5) { values( " ... " );}
        } /* end of enabled timing */
        timing() {
            timing_sense : positive_unate;
            timing_type : three_state_enable;
            related_pin : "En";
            cell_rise(delay_template_5x5) { values( " ... " );}
            rise_transition(delay_template_5x5) { values( " ... " );}
            cell_fall(delay_template_5x5) { values( " ... " );}
            fall_transition(delay_template_5x5) { values( " ... " );}
        } /* end of enable timing */
        timing() {
            timing_sense : negative_unate;
            timing_type : three_state_disable;
            related_pin : "En";
            cell_rise(delay_template_5x1) { values( " ... " );}
            rise_transition(delay_template_5x1) { values( " ... " );}
            cell_fall(delay_template_5x1) { values( " ... " );}
            fall_transition(delay_template_5x1) { values( " ... " );}
        } /* end of disable timing */
    } /* end of pin Y */
} /* end of eninv */

```

Figure 7.12: Example tristate inverter **.lib** description

format. It takes a bit of fiddling to get the inputs just right for this program, mostly because **SignalStorm** uses pure **Spectre** format as input whereas the NCSU tech libraries that are the basis for our cells uses the **Spice**-style input format that **SpectreS** can read. However, once the input is massaged to the right form this is a very slick program. When you consider that even for the inverter, using 5 x 5 timing matrices (25 values each), there are $6 * 25 = 150$ **Spectre** runs required to characterize the cell. Each run changes the input slope or the output loading or both and then measures the input and output waveform timing to get a number for one of the timing matrices. Cells with larger numbers of inputs and outputs, and sequential cells will require even more **Spectre** runs to characterize. Even a two-input NAND takes $12 * 25 = 300$ **Spectre** runs for full characterization using 5 x 5 matrices.

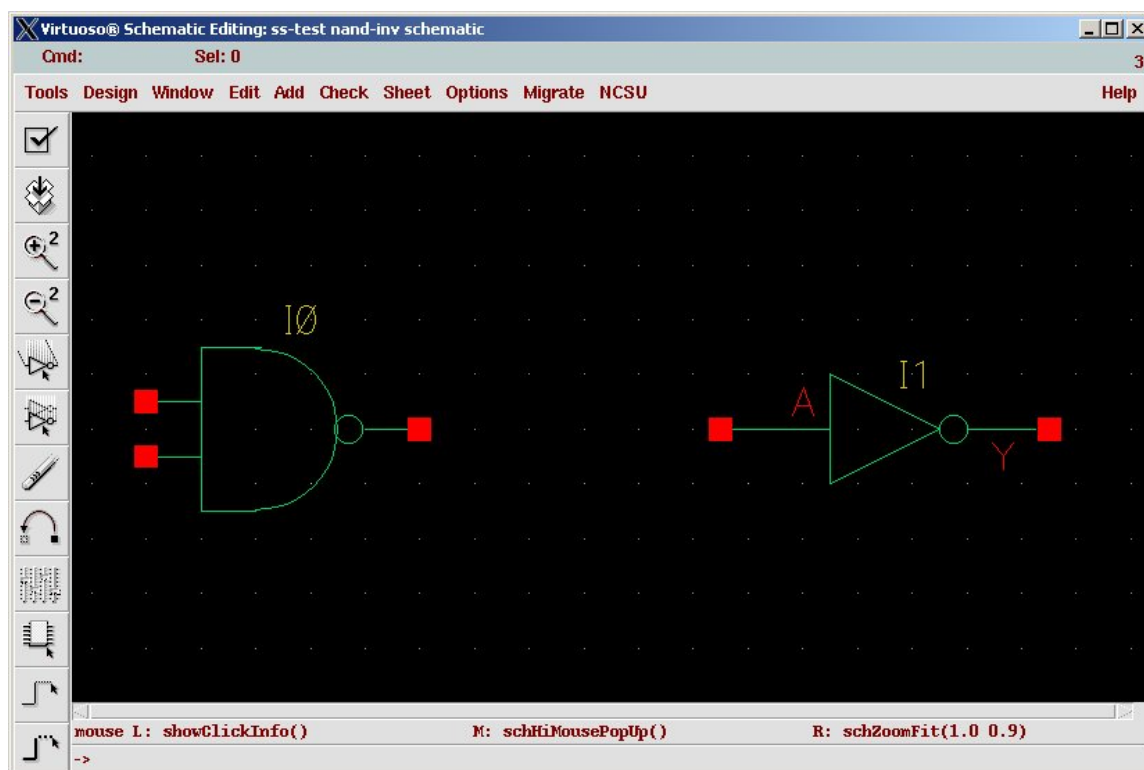
The other very nice thing about **SignalStorm** is that it even figures out for you what the functionality of your cells is. You don't have to specify in advance that a cell is, for example, a two-input and-or-invert gate. **SignalStorm** will figure this out for you based on analyzing the transistor network. Of course, you need to check to make sure that it has determined the functionality correctly, but it's quite good at this.

7.2.1 Generating the **SignalStorm** netlist

SignalStorm requires a netlist of all the cells that you would like it to characterize. This netlist should be *sub-circuit* definitions in **Spectre** format (which is very similar, but not exactly like **Spice** format). You could write these by hand, but it's much more convenient to have **Cadence** generate them for you from your schematics. The process will be very similar to the first steps of the analog simulation process from Chapter 6, but you'll have to specify the **Spectre** simulator instead of **SpectreS**, make a few more tweaks, and then stop at netlisting without actually simulating. From there (unfortunately) some hand-modification of the netlist is required.

Before you can characterize a cell you need a **cmos.sch** transistor level schematic view, and a **layout** view of each cell you want to characterize. The layout should have passed both **DRC** and **LVS**, and you should have created an **analog_extracted** view after the **LVS** was complete. It is the **analog_extracted** view that we'll use to generate the transistor netlist for **SignalStorm** characterization. This is because if you have extracted with parasitic capacitors then the **analog_extracted** view will include not only extra information on the transistor sizes, but also those parasitics.

As an example I'll start with a library that includes the inverter and nand2 cells that you've seen from Chapters 3, 4, and 5. The library includes (at least) **cmos.sch**, **layout**, and **analog_extracted** views of both of these

Figure 7.13: Schematic with one instance of **inv** and **nand2**

cells named **inv** and **nand2**.

Start by creating a schematic that contains one instance of each cell you would like to characterize. Don't connect them to anything, just place an instance in the schematic. When you save this schematic Cadence will complain that there are unconnected pins, but you can ignore that. As an example, my schematic (named **SignalStorm**) is shown in Figure 7.13.

Chapter 6 uses SpectreS because that's what is most easily supported for user-driven analog simulation by the NCSU technology files we're using.

After saving this schematic (and ignoring the warnings), open the **Analog Environment** with **Tools** → **Analog Environment**. This is just like Chapter 6. However, we need to change a couple things because **SignalStorm** requires pure **Spectre** format and not the **SpectreS** format that is used in Chapter 6.

1. Use the **Setup** → **Simulator / Directory / host** menu to open a dialog box. From there change the **Simulator** to be **Spectre** (with no "S" on the end). This will force the analog netlister to use **Spectre** format when it generates the netlist. See Figure 7.14.
2. Use the **Setup** → **Environment** menu to open a dialog box and add

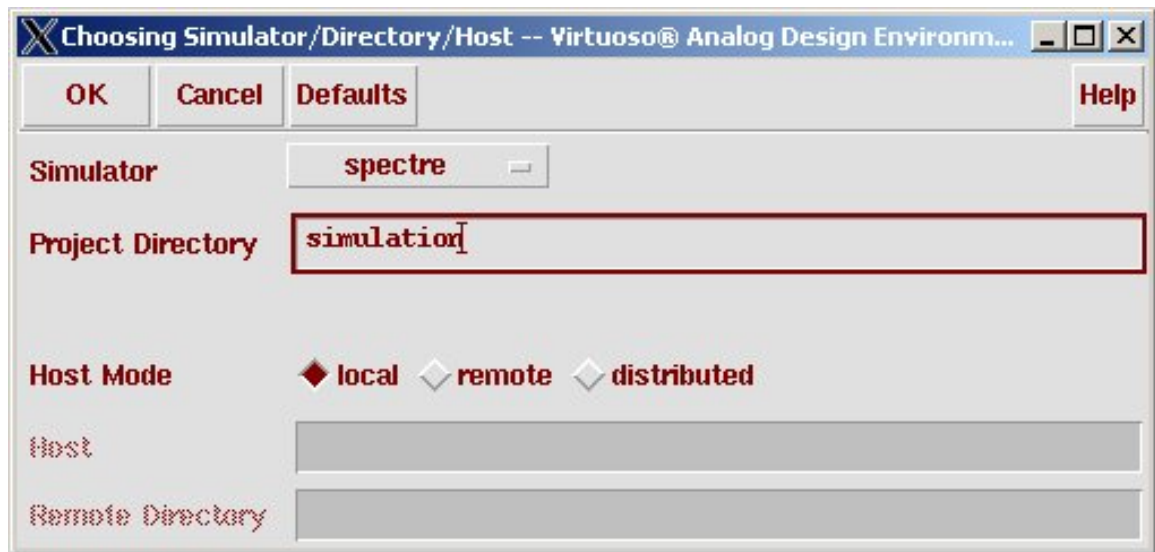


Figure 7.14: Choosing Spectre as the simulator

analog_extracted to the front of the **Switch View List** list. See Figure 7.15. This will tell the netlister to look for the **analog_extracted** view first to find the transistor netlists for each cell.

Now you can generate the netlist with **Simulation** → **Netlist** → **Create**. This will pop up a new window with the **Spectre** netlist of your schematic (the one that has instances of each cell). You should **Save As** this file to a file named **dut.scs** which will appear in the directory in which you started **cad-ncsu**. Now, unfortunately, you need to hand-edit this file to make some changes! This is because the netlister from NCSU wasn't designed to make perfect **SignalStorm** files. But, they're close. Once you've created and saved this **dut.scs** file you can close the Analog Environment and exit **Cadence**.

What **SignalStorm** wants is a file that contains nothing but **subckt** definitions. These are the descriptions of each cell as a sub-circuit. The initial **dut.scs** file is shown in Figure 7.16. You can see that it starts with some control statements, then defines each cell as a **subckt**, then includes in instance of each of those sub-circuits (the **I1** and **I0** statements are instances), and finishes with some additional control statements for the simulator. You need to take this file and modify it in the following ways:

1. First remove all the lines of text before the first **subckt** definition (leaving the comments is all right), and after the last **subckt** is finished. All **SignalStorm** wants is the **subckt** definitions, nothing else.

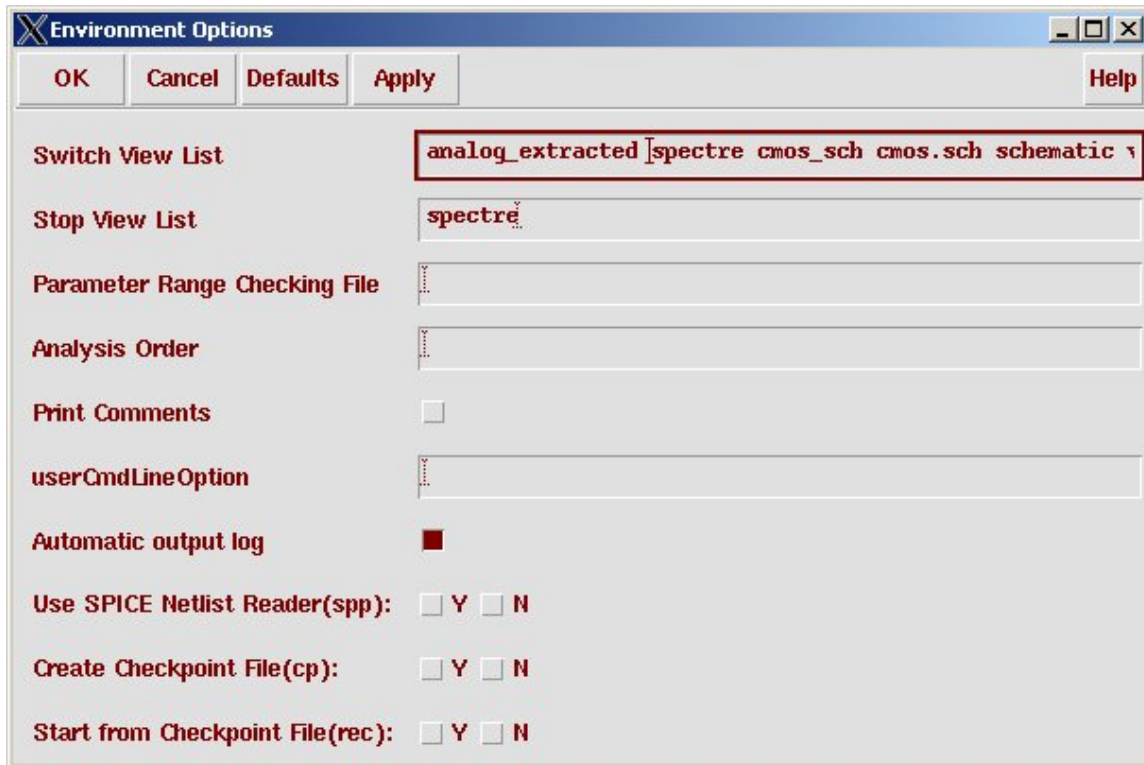


Figure 7.15: Adding **analog_extracted** to the **Switch View List**

Note that **Spectre** (at least as used by **SignalStorm** has no concept of global signals so removing the **global** statement won't cause problems.

2. **SignalStorm** doesn't like non-alpha characters in signal names. Change every instance of **vdd!** to be **vdd**.
3. **SignalStorm** doesn't like to use **0** as the default ground node. This is quite odd because this is a standard convention, but it appears to be true for the specific way that **SignalStorm** uses **Spectre**. Unfortunately, this is a trickier thing to change with search-and-replace because there are also 0's in other parameters. Make the following replacements:
 - (a) Replace "**0**" with "**gnd**" (note that there are spaces on both sides of each of those strings!).
 - (b) Replace "**(0**" with "**(gnd**".
 - (c) Replace "**0)**" with "**gnd)**".
4. Because **SignalStorm** doesn't use global signal names, you need to add **vdd gnd** to the argument list of each **subckt** definition.

A **dut.scs** file that has had all the required modifications is shown in Figure 7.17. Note that it consists only of **subckt** definitions, has **vdd gnd** added to all **subckt** argument lists, and uses **vdd** and **gnd** to define the power and ground connections.

Now that you have a modified **dut.scs** file with **subckt** definitions for the cells you want to characterize, you can start using **SignalStorm**. I recommend that you make a separate directory in which to run **SignalStorm** so that you can compartmentalize things and keep track of where the log files and result files are. I'm using a **\$HOME/IC_CAD/slc** directory for this (**slc** stands for **SignalStorm Library Characterizer**). Move your **dut.scs** file to that directory.

In your **slc** directory you need to copy some script files from the class directory, specifically the **cadence/SLC** subdirectory. The directory path is:

```
/uusoc/facility/cad.common/local/class/6710/cadence/SLC
```

From that directory, copy the **step1**, **step2** and **step3** files to your **slc** directory. These are command files you can use to drive the **SignalStorm** characterization process. These scripts are used one at a time in sequence, but they represent three important steps in the process where you may need to stop and fix errors which is why they're separate. You can use the scripts as-is, or edit them to do different things once you are more familiar with the process. To run **SignalStorm** use the `cad-slc` script in the same

```
// Generated for: Spectre
// Generated on: Aug 31 18:05:16 2006
// Design library name: tutorial
// Design cell name: signalstorm
// Design view name: schematic
simulator lang=Spectre
global 0 vdd!

// Library name: tutorial
// Cell name: inv
// View name: analog_extracted
subckt inv A Y
    \+1 (Y A vdd! vdd!) ami06P w=6e-06 l=6e-07 as=9e-12 ad=9e-12 ps=9e-06 \
        pd=9e-06 m=1 region=sat
    \+0 (Y A 0 0) ami06N w=3e-06 l=6e-07 as=4.5e-12 ad=4.5e-12 ps=6e-06 \
        pd=6e-06 m=1 region=sat
ends inv
// End of subcircuit definition.

// Library name: tutorial
// Cell name: nand2
// View name: analog_extracted
subckt nand2 A B Y
    \+3 (vdd! B Y vdd!) ami06P w=6e-06 l=6e-07 as=5.4e-12 ad=9e-12 \
        ps=1.8e-06 pd=9e-06 m=1 region=sat
    \+2 (Y A vdd! vdd!) ami06P w=6e-06 l=6e-07 as=9e-12 ad=5.4e-12 \
        ps=9e-06 pd=1.8e-06 m=1 region=sat
    \+1 (Y B _6 0) ami06N w=6e-06 l=6e-07 as=2.7e-12 ad=9e-12 ps=9e-07 \
        pd=9e-06 m=1 region=sat
    \+0 (_6 A 0 0) ami06N w=6e-06 l=6e-07 as=9e-12 ad=2.7e-12 ps=9e-06 \
        pd=9e-07 m=1 region=sat
ends nand2
// End of subcircuit definition.

// Library name: tutorial
// Cell name: signalstorm
// View name: schematic
I1 (net1 net2) inv
I0 (net5 net4 net3) nand2
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
    tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 maxwarns=5 \
    digits=5 cols=80 pivrel=1e-3 ckptclock=1800 \
    sensfile="../psf/sens.output" checklimitdest=psf
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
designParamVals info what=parameters where=rawfile
primitives info what=primitives where=rawfile
subckts info what=subckts where=rawfile
saveOptions options save=allpub
```

Figure 7.16: **dut.scs** file as generated by Analog Environment


```
// Library name: tutorial
// Cell name: inv
// View name: analog_extracted
subckt inv A Y vdd gnd
    \+1 (Y A vdd vdd) ami06P w=6e-06 l=6e-07 as=9e-12 ad=9e-12 ps=9e-06 \
        pd=9e-06 m=1 region=sat
    \+0 (Y A gnd gnd) ami06N w=3e-06 l=6e-07 as=4.5e-12 ad=4.5e-12 ps=6e-06 \
        pd=6e-06 m=1 region=sat
ends inv
// End of subcircuit definition.

// Library name: tutorial
// Cell name: nand2
// View name: analog_extracted
subckt nand2 A B Y vdd gnd
    \+3 (vdd B Y vdd) ami06P w=6e-06 l=6e-07 as=5.4e-12 ad=9e-12 \
        ps=1.8e-06 pd=9e-06 m=1 region=sat
    \+2 (Y A vdd vdd) ami06P w=6e-06 l=6e-07 as=9e-12 ad=5.4e-12 \
        ps=9e-06 pd=1.8e-06 m=1 region=sat
    \+1 (Y B _6 gnd) ami06N w=6e-06 l=6e-07 as=2.7e-12 ad=9e-12 ps=9e-07 \
        pd=9e-06 m=1 region=sat
    \+0 (_6 A gnd gnd) ami06N w=6e-06 l=6e-07 as=9e-12 ad=2.7e-12 ps=9e-06 \
        pd=9e-07 m=1 region=sat
ends nand2
// End of subcircuit definition.
```

Figure 7.17: **dut.scs** after required SignalStorm modifications

location as the `cad-ncsu` script. With these files in place you can run the characterization with the following steps. The syntax for running with the script is

```
cad-slc -S <scriptname>
```

1. `cad-slc -S step1` This script will open a database called **foo** to hold your data (with the **SignalStorm** command **db_open**). The script (shown in Figure 7.18) then sets some flags to control how **SignalStorm** works, and uses **db_install** to install a set of transistor models (typical case models for the AMI C5N 0.6 μ process that we're using) and your **dut.scs** file with the subckt descriptions into that database. The **db_gsim -force** command will evaluate the transistor networks to understand what they are, and generate test vectors for characterization. In this phase **SignalStorm** actually applies all possible input vectors to your circuit to try to figure out what it is. Brute force, but effective. The **db_gate** command will print out what gate it thinks your cell is so that you can verify that **SignalStorm** figured it out right. The **db_setup** command reads a setup file that contains lots of details about how the simulation should happen. Finally **db_close** closes the database.

After running **cad-slc** with the **step1** script you should look carefully at the output to make sure that things are proceeding correctly. If you see messages such as **no supply0** or **no simulation** anywhere in the output of this process, you have a problem! Usually these are editing errors in **dut.scs** from the hand-edit phase of this process. Look for misspelled **vdd** and **gnd** names, or for any extra lines in the file that are not related to the **subckt** definitions. The **db_gate** command will print out what gate **SignalStorm** thinks it has extracted from your transistors. Make sure that the gate matches with what you think the gates are!

The output of the **db_gate** command for my **dut.scs** file that contains one instance of an inverter and one instance of a nand2 is shown in Figure 7.19. There is a lot more output from the **step1** phase of the process. This is just a piece of a much longer piece of output. Make sure that you are getting no errors or warnings before proceeding to **step2**!

2. `cad-slc -S step2` This script (shown in Figure 7.20) starts by opening the database named **foo** that was created in **step1**. The comment about the next three lines relates to a feature of **SignalStorm** that lets you start servers on a bunch of machines and spawn simulation jobs on those remote machines during characterization. We won't use that option, but you can easily imagine that this would be a

```

db_open foo

set_var SG_SPICE_SIMPLIFY true
set_var SG_HALF_WIDTH_HOLD_FLAG true
set_var SG_SIM_NAME "Spectre"
set_var SG_SIM_TYPE "Spectre"
set_var SG_SPICE_SUPPLY1_NAMES "vdd"
set_var SG_SPICE_SUPPLY0_NAMES "gnd"

db_install -model /uusoc/facility/.../6710/cadence/SLC/ami_c5n_typ.scs -subckt dut.scs
db_gsim -force
db_gate
db_setup -s /uusoc/facility/.../6710/cadence/SLC/setup.ss -process typical
db_close
exit

```

Figure 7.18: SignalStorm **step1** script (path names are shortened...)

```

=====
      DESIGN : INV
=====
DESIGN ( INV );
//      =====
//      PORT DEFINITION
//      =====
      INPUT A ( A );
      OUTPUT Y ( Y );
      SUPPLY0 GND ( GND );
      SUPPLY1 VDD ( VDD );
//      =====
//      INSTANCES
//      =====
      NOT ( Y, A );
END_OF_DESIGN;

=====
      DESIGN : NAND2
=====
DESIGN ( NAND2 );
//      =====
//      PORT DEFINITION
//      =====
      INPUT A ( A );
      INPUT B ( B );
      OUTPUT Y ( Y );
      SUPPLY0 GND ( GND );
      SUPPLY1 VDD ( VDD );
//      =====
//      INSTANCES
//      =====
      NAND ( Y, A, B );
END_OF_DESIGN;

```

Figure 7.19: SignalStorm **step1** output (portion)

```
db_open foo

# Remove the next 3 lines to use the ipsd/ipsc
# daemons for load balancing on multiple machines
set_var SG_SIM_USE_LSF 1
set_var SG_SIM_LSF_CMD " "
set_var SG_SIM_LSF_PARALLEL 10

set_var SG_SIM_NAME "Spectre"
set_var SG_SIM_TYPE "Spectre"
set_var SG_SPICE_SUPPLY1_NAMES "vdd"
set_var SG_SPICE_SUPPLY0_NAMES "gnd"
set_var SG_HALF_WIDTH_HOLD_FLAG true

db_spice -s spectre -p typical -keep_log
db_close
exit
```

Figure 7.20: SignalStorm **step2** script

good idea for a very large library and a very large room of computers chugging away at characterization. In our case we'll restrict ourselves to one machine, but spawn up to 10 **Spectre** jobs at the same time.

The **db.spice** command is where all the simulation action happens. It uses the **subckt** definitions you supplied in **dut.scs** and the test vectors that were derived in **step1** to simulate and extract timing and power numbers using **Spectre**. This step can take a very long time depending on the number of cells and the complexity of the cells that you're characterizing. The key here is to make sure that the results of all the simulations is **PASS**. An example of a portion of the output from this step on my example is shown in Figure 7.21. If you get **FAIL** on any of the simulations you'll have to look at the output in the **signalstorm.log** and **signalstorm.work** directories to try to figure out what's going on. Also look back at the output from **step1** to make sure you weren't starting with problems.

Once you complete **step2** correctly, the characterization data is in the **foo** database and you can move to **step3**.

3. `cad-slc -S step3` This script (shown in Figure 7.22) simply outputs to the results of **step2** into file named **foo.alf** and also outputs a Verilog file with simple behavioral descriptions of each cell in the library. The **.alf** file contains all the characterization data that was generated by **SignalStorm**. Unfortunately, **.alf** format is not quite what we want. We want the **liberty** format described in Section 7.1. Fortunately, there's an automatic way to do this.
4. Use yet another script to run the conversion program from **alf** to **lib**

```
slc> db_spice -s spectre -p typical -keep_log
```

DESIGN	PROCESS	#ID	STATUS
INV	typical	D0000	SIMULATE
INV	typical	D0001	SIMULATE
=====	=====	=====	=====
INV	typical	2	2
NAND2	typical	D0000	SIMULATE
NAND2	typical	D0001	SIMULATE
NAND2	typical	D0002	SIMULATE
NAND2	typical	D0003	SIMULATE
NAND2	typical	D0004	SIMULATE
=====	=====	=====	=====
NAND2	typical	5	5

```

--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--
                        Simulation Summary
--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--

-----+-----+-----+-----+-----+
      DESIGN | PROCESS | #ID | STAGE | STATUS
-----+-----+-----+-----+-----+
INV          | typical | D0000 | VERIFICATE | PASS
INV          | typical | D0001 | VERIFICATE | PASS
NAND2        | typical | D0000 | VERIFICATE | PASS
NAND2        | typical | D0001 | VERIFICATE | PASS
NAND2        | typical | D0002 | VERIFICATE | PASS
NAND2        | typical | D0003 | VERIFICATE | PASS
NAND2        | typical | D0004 | VERIFICATE | PASS
-----+-----+-----+-----+-----+

--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--
- Total Simulation : 7
- Total Passed      : 7(100%)
- Total Failed      : 0(0%)
--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--*_--

slc> db_close

Database : foo is closed
slc> exit

```

Figure 7.21: SignalStorm **step2** output (portion)

```
db_open foo
db_output -r foo.rep -alf foo.alf -p typical
db_verilog -r foo.v
db_close
exit
```

Figure 7.22: SignalStorm **step3** script

```
cell INV* {  
  footprint inv ;  
  area 129.6 ;  
};  
  
cell NAND2* {  
  footprint nand2 ;  
  area 194.4 ;  
};  
  
cell NOR2* {  
  footprint nor2 ;  
  area 194.4 ;  
};
```

Figure 7.23: Example **footprints.def** file

format. This script is

`cad-alf2lib <basename>` where **<basename>** is the base-name of the **alf** file. In our case this, unless you’ve edited the scripts, this will be **foo**. So, with `cad-alf2lib foo` you will generate a **foo.lib** file which contains the **liberty** version of your newly characterized library.

*It seems odd with a name like “footprint,” but cells with the same I/O and function, but with different sizes, can share the same **footprint**. For example, all inverters, regardless of physical area, should be defined to use a common **footprint** so that they can be used later for driving different loads.*

This script uses a **footprints.def** file to get some additional information about your cells. The most important missing pieces of information are the area of the cells (in square microns) and the **footprint** of the cell. The **footprint** is a way of grouping cells that have the same function but perhaps different drive strengths, for example. These cells are grouped into a common **footprint** so that later in synthesis they may be used interchangeably depending on loads being driven or speed required. An example of a **footprints.def** file is shown in Figure 7.23. This file is also in the following directory:

`/uusoc/facility/cad.common/local/class/6710/cadence/SLC`

You can copy it from there and modify as needed. This file says that all cells whose name starts with **INV** should have the same **inv** footprint, and should have area **129.6** (in square microns). Clearly you will need to edit this file to reflect the sizes and names of your cells. If you don’t have this **footprints.def** file in your directory, the **cad-alf2lib** process will still generate a **foo.lib** file, but without areas or footprints. This information is very useful to the synthesis process, though, so you should include it!

You can, of course, run through this process with small numbers of cells and then edit them into a larger **.lib** file. There’s no reason to wait until all

cells are ready before running characterization, or to re-characterize cells that haven't changed. Remember to keep the technology header of the **.lib** file intact once, and add new cell descriptions to the end.

7.2.2 Cell Naming and SignalStorm

In **step1** of the **SignalStorm** procedure a setup file named **setup.ss** is loaded from the class directory. This file sets up a number details about how the simulation should proceed. Most importantly it sets up what the indices should be for the characterization matrices. These matrices have **input_transition_time** in ns on one axis, and **total_output_net_capacitance** in pF on the other axis. The default values for these are:

```
Index DEFAULT_INDEX{
    Slew = 0.100n 0.30n 0.7n 1.0n 2.0n;
    Load = 0.025p 0.05p 0.1p 0.3p 0.6p;
};
```

However, the **setup.ss** file also defines some different loads and slews for cells that are designed to drive larger loads. Any cell whose name ends in **X1** will also have this standard load. A cell whose name ends in **X2** to indicate twice as much drive capability on the outputs will have a different set of values:

```
Index X2{
    Slew = 0.100n 0.30n 0.7n 1.0n 2.0n;
    Load = 0.050p 0.10p 0.2p 0.6p 1.2p;
};
```

Furthermore, still different loads and slews are defined for cells whose name ends in **X4** and **X8**. So, for example, if you have several different inverters in your library with different drive strengths (different output transistor sizes) named **INVX1**, **INVX2**, **INVX4** and **INVX8**, then they will each be characterized with a load and slew that best matches where they are intended to be used. Of course, you can copy the **setup.ss** file and modify things if you'd rather follow a different naming convention, or want to add or modify things. The full **setup.ss** file can be seen in Appendix C.

7.2.3 Best, Typical, and Worst Case Characterization


Commercial libraries are almost always characterized for three different versions of cell timings: best case, typical case, and worst case. These are supposed to represent three different variations on how the circuit is likely

to work. Best case timings are usually feasible, but quite optimistic. They assume transistors that operate in the fast region measured by the foundry, with a vdd that's typically 10% higher than nominal, and at a nice cool temperature. Typical case timings are supposed to model what you'll usually see in the fabricated circuit. The transistor models are taken from the middle of the measured performance distribution, vdd is nominal (5v in our 6 μ process), and the temperature is assumed to be 25C. Worst case timings are taken from the slow end of the transistor distribution, with vdd at 10% below nominal, and a nice toasty temperature of 125C. If you're trying to see what speed your circuit might run at, use the **typical** timings. If you're trying to see whether your circuit will work in more general environments, use the **worst** timings. Only a very optimistic person would ever use **best** timings!

The **step1**, **step2** and **step3** scripts are designed to measure **typical** case timings for you cells. However, if you would like to generate **best** and **worst** case libraries as well, there are versions of the scripts to do this. For example, **step[1,2,3]-worst** will measure worst case timings using worst-case transistor models from **MOSIS**, vdd assumed to be 4.5v, and a temperature of 125C. These scripts are in the class library.

7.3 Cell Characterization with Spectre

If you want to characterize cells by hand, you can do this using SpectreS by making your own test fixture circuits and doing analog simulation as described in Chapter 6. This is clearly a little more hand-work than letting SignalStorm run everything, but you have much more control over every aspect of the characterizations. My recommendation is to use SignalStorm, but here are some things to think about if you'd like to do things directly with SpectreS.

The process of characterizing cells involves running lots of simulation while varying the input slope and the output load so that you can fill out the timing tables. This argues strongly for using parametric simulation where you vary the parameters of the voltage sources and the output loads automatically. For example, you could set up a test fixture schematic as in Figure 7.24. In this test fixture the DUT is driven by a **vpulse** voltage source, and is driving load capacitor. The slope of the pulse will be controlled by a variable named **slope** and the size of the load will be controlled by a variable named **load**. To assign these component parameters a variable name, simply give the variable name in the  properties of the device instead of a numeric value.

Now you can use the same parametric simulation techniques as described in Chapter 6 Section 6.5.1. In this case you could vary the input

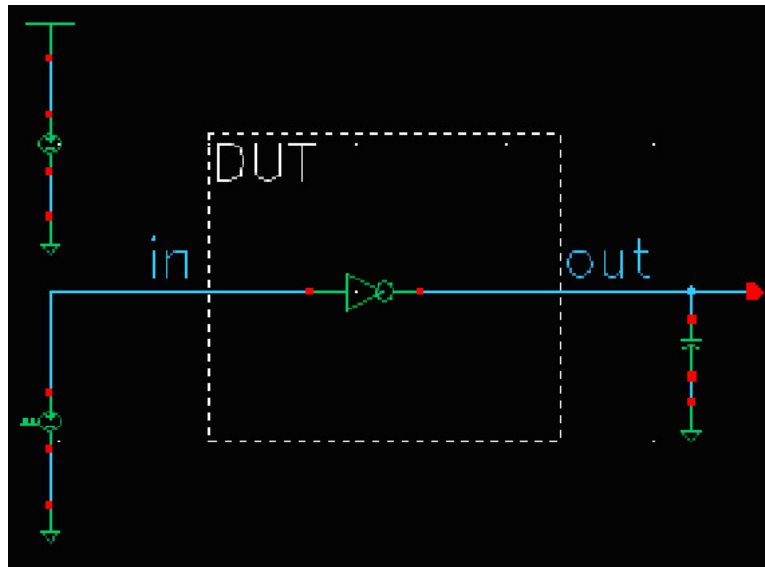


Figure 7.24: Test fixture for hand-simulation and characterization

slope to be a series of times in ns, and the output load to be a series of values in pF and generate all 25 values needed to populate the tables we’ve been using. You could, of course, also use different numbers of **slope** and **load** values to generate differently sized tables. This is easily set up in the **Analog Environment** GUI, but it can be tedious to use the menu and dialog box interfaces to set this up time after time. Another answer is to write a script to automate the process of doing the parametric simulation.

In the Cadence tool suite the separate tools under the **Design Framework** umbrella are controlled by a scripting language called *Skill*. *Skill* is a language that looks a lot like Lisp. In fact, it looks almost exactly like Lisp. But, if you aren’t as fond of Lisp syntax conventions as I am, there is a C-like syntax available for *Skill* programs. If you know the right function calls, you can write a *Skill* script to do almost anything in Cadence that you can do through the GUI. Of course, it’s sometimes a little tricky to figure out what the function calls are, and how the script gets all the information it needs, so usually *Skill* scripting is reserved for the hard-core user or for the CAD support folks. But, for some relatively simple stuff, it can be very useful. *Skill* is, of course, documented in the Cadence documentation. If you’re curious I recommend that you start with the *Skill* users manual (accessible from the Cadence help menu).

An extension to *Skill* that is specifically for writing scripts to control analog simulation is called **ocean**. An **ocean** script looks just like *Skill*, but has some additional functions that are specific to the task of interacting

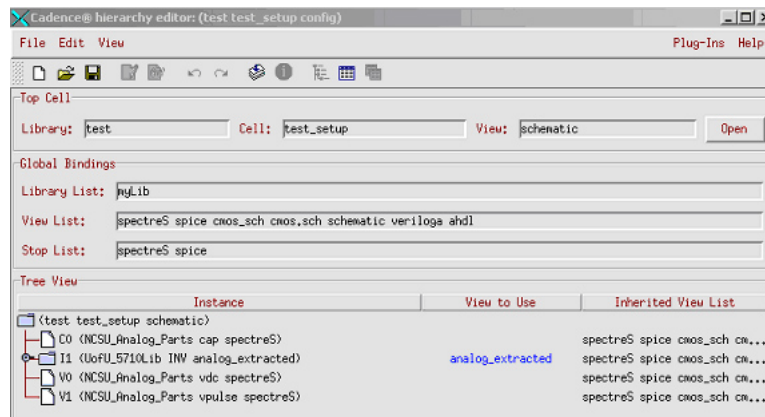
with the **Analog Environment**. For example, **ocean** scripts can set which design you're looking at, which set of waveform results you're evaluating, and which signals in those waveforms you're measuring.

I've written an ocean script that you can use or modify to perform parametric simulation of a cell and which will generate the output in a set of lookup tables formatted for the **.lib** file. It turns out that formatting the output to look like **liberty** format wants it is practically the hardest part! The other hard part is that I can't figure out is how to generate the netlist for simulation in the first place in the script, so you still need to use the **Analog Environment** GUI for part of the process, the same as for **SignalStorm**. But, after you get things set up, it's as easy as typing a function name in the CIW and you get the parametric simulation results generated for you. Of course, it's not remotely as automated as **SignalStorm** so I still recommend that you stop reading now and go back to the **SignalStorm** section!

If you're still reading, in order to use the script, you'll need a schematic that describes the DUT simulation setup. I called this schematic **test.setup** for lack of a more imaginative name. Because we're aiming to simulate over a range of output loads I put a capacitor on the output with the variable **load** as the capacitance value. Because we're also aiming to simulate over a range of input slopes, I put a **Vpulse** on the input of the DUT with the variable **slope** as the rise and fall time values. I can then use the parametric simulation setup to vary both of those values. The schematic is shown in Figure 7.24.

Notice that the DUT is being driven by the **Vpulse** and is driving into the load capacitor. If you were simulating a multi-input gate, you'd need to tie the non-driven inputs of that gate to an appropriate value that would turn the output of the gate to either an inverter or a buffer. For example, if you were simulating a 2-input NAND gate, you'd want to tie the input that you're not driving to logic 1 (vdd) so that the pulse on the other input would come through the output as an inverted signal. Remember that you need to generate timing for all outputs with respect to all inputs so in the case of a 2-input NAND you'd need timing from **A** \rightarrow **Y** and from **B** \rightarrow **Y**. Each time you would modify the schematic and generate a new netlist to simulate with the script. You also need to keep track of whether the gate is acting like an inverter (**negative_unate** timing), or as a buffer (**positive_unate** timing) for this simulation.

In order to make sure that I'm simulating the right thing, I'll need a **config** view of the **test.setup** schematic, just like you've done before. In the **config** view you will want to specify the **analog_extracted** view of whatever cell you're using as the DUT in this simulation run. In Figure 7.25 you can see the **config** view of my **test.setup** schematic with the **analog_extracted** view of my inverter selected.

Figure 7.25: **config** view for hand-simulation and characterization

To generate the analog netlist open the **test_setup** schematic and then open the **Analog Environment**. Make sure that you have the **config** view selected (use **Setup** → **Design** if it's not correct). Here's an annoying part of the process: you can't generate the netlist unless all the variables in your schematic have initial values even though you're going to override those defaults when you run the parametric simulation.

To set the variables to a starting value use the **Variables** → **Copy From Cellview** option. This will fill in the design variables pane of the **Analog Environment**. Double click on the variable names and set them to some value. In Figure 7.26 I've set slope to **500ps** and load to **500ff**. Once things look like Figure 7.26 you can use **Simulation** → **netlist** → **Create Final** to create the final netlist. All this does is create the netlist. The **ocean** script will actually run the simulation for you. You can dismiss the text window that pops up to show you the netlist that was just created.

Note that because we're going to simulate this ourselves, we'll go ahead and use SpectreS as described in Chapter 6. That is, we don't need to reset the simulator as we did for SignalStorm.

Once you have the netlist generated you need to load and run the ocean script. The script is named **test.ocn** and is in the class directory

```
/uusoc/facility/cad.common/local/class/6710/cadence
```

You should copy this script to your own directory. You might want to take a look at the file to see what it does. You might also want to look at the first section to see if there are any things you would like to change about how the script does its stuff. You might have different names for the parametric variables, for example. Or you might want to use a different set of values for the range of the parametric variables, or to use more or fewer values in those lists depending on the table templates you've defined.

Once you have things set up in the script the way you want them you need to load the script so that you can execute the new functions. You do

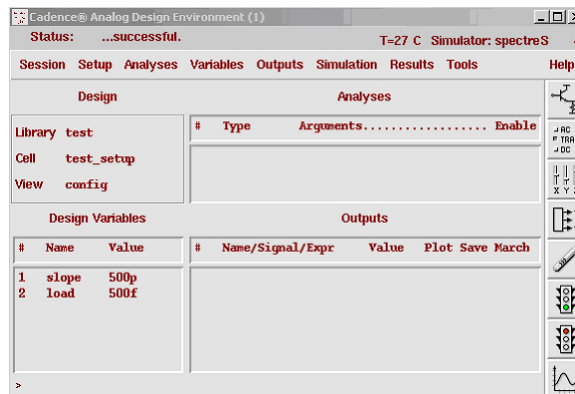


Figure 7.26: Setting variables in the **Analog Environment**

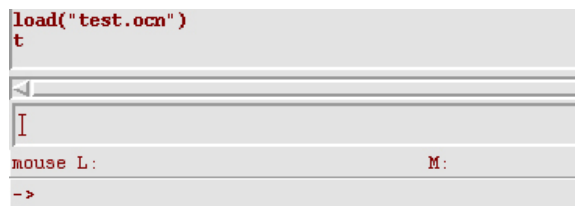


Figure 7.27: Loading the **test.ocn** script

this by typing the load command into the CIW. The command is

`load("test.ocn")` or `(load "test.ocn")`

depending on whether you like C-syntax or Lisp syntax. See Figure 7.27 for an example of what you should see in the CIW. The **t** just means that you've loaded the script successfully.

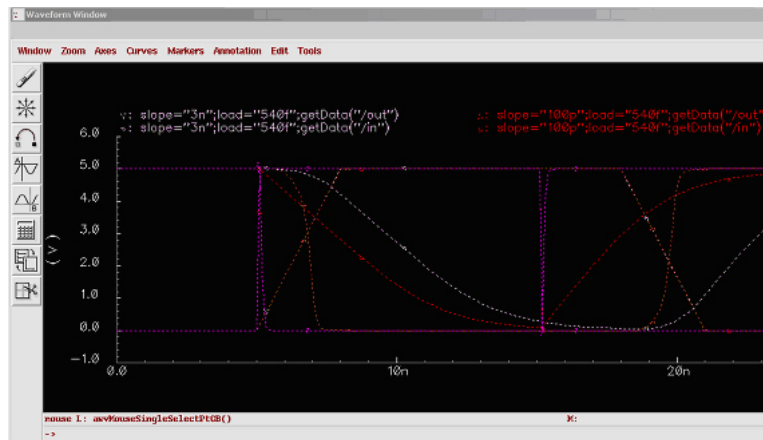
Once you've loaded the script you can type the command into the CIW to perform the parametric simulation. The command is:

`run_test()` or `(run_test)`

This function actually has an optional argument of the stop time for the transient analysis. This defaults to the value in the script, but if you want to change it you can include it in the call to **run_test**:

`run_test("80n")` or `(run_test "80n")`

What you'll see in response is a bunch of text in the CIW output pane for each of the parametric transient simulations. You might want to try this with only a couple values for load and for slope so that it doesn't take so long to run the simulations just to make sure that everything is working. You can

Figure 7.28: Plot output from a **test.ocn** simulation

easily restore the **loadlist** and **slopelist** to their full values and run it again once you get things working. If you do make changes to the **test.ocn** file, remember to load it again so that the changes will take effect!

Once the parametric simulation is complete, you'll get a plot window that shows all the parametric in and out waveforms. This is just for you to look at (and if you're annoyed by it you can comment out the **plot** line in the script.) Without the **ocean** script you could use the A and B measurement bars to measure the various timings, but the script can do this for you. An example plot is shown in Figure 7.28.

Once you've run the tests, you can print the results in **.lib** format into a file of your own choosing. The directory that this file goes into is set in the script (in this example the output file is in **IC_CAD/cadence/simulation/test_setup/<filename>**). The **fprint_results** function takes two arguments: the first is a string which is the name of the file to put the results into, and the second is either '**neg**' for a **negative_unate** timing (i.e. an inverting gate) or '**pos**' for a **positive_unate** measurement.

This outputs the data into tables that are in the right format for pasting into your **.lib** file. An example of running this script on an **X1** inverter is shown in Figure 7.29. You can run this script for each of the input/output combinations and paste the results into the timing section of the **.lib** file for each of those pins if you have more general circuits.

This is a home-grown method for deriving lookup table timings for library cells. With **SignalStorm** it may be less useful than it was, but it may still be useful for odd cells, or for understanding how to script and parametrize analog simulations in general.

```

cell_rise(lu5x5) {
  values( \
    ".145649, .461407, .821977, 1.54147, 2.95759" ,\
    ".384865, .72295, 1.07269, 1.79433, 3.21885" ,\
    ".60943, 1.02507, 1.38113, 2.08612, 3.51231" ,\
    "1.0132, 1.55007, 1.98756, 2.70889, 4.09905" ,\
    "1.65337, 2.35774, 2.89704, 3.77152, 5.1905" );}

rise_transition(lu5x5) {
  values( \
    ".148306, .610152, 1.14291, 2.2072, 4.34568" ,\
    ".305532, .700173, 1.17722, 2.21651, 4.35869" ,\
    ".469617, .894771, 1.35714, 2.32296, 4.38333" ,\
    ".743882, 1.26289, 1.70498, 2.63657, 4.52904" ,\
    "1.20801, 1.84408, 2.32966, 3.29812, 5.05465" );}

cell_fall(lu5x5) {
  values( \
    ".185292, .602238, 1.07644, 2.02688, 3.91734" ,\
    ".432254, .852127, 1.3175, 2.2643, 4.14154" ,\
    ".683672, 1.15366, 1.61688, 2.5389, 4.4227" ,\
    "1.14297, 1.72898, 2.2261, 3.14691, 4.99237" ,\
    "1.89231, 2.63401, 3.23074, 4.2367, 6.03552" );}

fall_transition(lu5x5) {
  values( \
    ".167624, .72171, 1.355, 2.62284, 5.13434" ,\
    ".309461, .78558, 1.37218, 2.62191, 5.10869" ,\
    ".474777, .961437, 1.49416, 2.65833, 5.13505" ,\
    ".780225, 1.2901, 1.82791, 2.9399, 5.23852" ,\
    "1.24947, 1.86456, 2.44295, 3.52342, 5.67453" );}

```

Figure 7.29: Results of running **test.ocn** on an inverter and printing as **.lib** tables

7.4 Converting Liberty to Synopsys Database (db) Format

Once you have generated a **liberty** formatted file that describes your cells you should be done because this is the most common industry standard format for describing cell timing and power data. But, even though this is a format that originally came from Synopsys, the Synopsys tools actually require a compiled format that is compiled from this **liberty** text description. The Synopsys **library compiler** tool can easily compile the **.lib** file into this **.db** format, but unfortunately, and for reasons I don't understand, the **.lib** file that is generated by SignalStorm actually needs a little hand-modification to be completely compatible with Synopsys! Annoying but true.

Start with the **.lib** file that you got from SignalStorm and make the following changes. If you've written your own **.lib** file then you may have other issues of course, but you also need to pay attention to these.

- There are errors in the compilation process if a pin has a **fall_capacitance_range** parameter, but not a **rise_capacitance_range**. One quick fix is just to take out all **[rise/fall]_capacitance_range** lines. This won't cause a huge difference. These lines describe the range of capacitance a pin will see during signal transitions. Alternatively, you could add the missing **[rise/fall]_capacitance_range** statement for each pin that has one but not the other. For the missing range you can use the single value for the pin capacitance for the top and bottom of the range.
- If you remove all those attributes, you may get warnings about not having them, but you can ignore them. Look at the warnings. They tell you that the compiler will assume that the range is just equal to the one measured value, which should be fine.
- If you have negative delays in any of your timing tables (I had them for **disable timings** for the **eninv** when the input rise times were slow, for example) you should "round" those negative timings to 0.
- If you want to avoid warnings about not having default values set you should include the following lines in the technology header section of your **.lib** file:

*Instructions for
compiling **.lib** into **.db**
are coming up in this
section*

```
/* Default attributes */  
  
/* Fanout (in terms of capacitive load units) */  
default_fanout_load : 0.3 ;
```

```
default_max_fanout : 10.0 ;

/* Pin Capacitance */
default_inout_pin_cap : 0.00675 ;
default_input_pin_cap : 0.00675 ;
default_output_pin_cap : 0.0 ;

/* leakage power */
default_cell_leakage_power : 0.0 ;
default_leakage_power_density : 0.0 ;
```

- If you want to include the wire load models, include the text from Figures 7.3 and 7.4 in the tech header section of your **.lib** file.

Once you have a correct (and correctly modified) **.lib** file you can compile it into a Synopsys synthesis database (**.db**) file. To do this you need to run the Synopsys **design compiler** synthesis tool, but only in a very simple way. The command to start **design compiler** or, more precisely the command-line shell version called **dc_shell** is with the command

```
syn-dc
```

This will start up **design compiler** in a mode where you can type in commands to the **design compiler** shell. What you want to do is read in the library in **.lib** format, and write out that library in **.db** format. The first command that you type to the shell is the `read_lib` command to read the **.lib** file. It looks like (`dc_shell-xg-t>` is the **design compiler** shell prompt:

```
dc_shell-xg-t> read_lib <filename>.lib
```

where, of course, `<filename>.lib` is the name of your **.lib** file. This will issue a bunch of warnings about defining new variables and perhaps about missing range attributes. You should look carefully at this stage to make sure there are no errors! If you are satisfied that there are only warnings and that the warnings are not important, you can then write out that library in **.db** format with the `write_lib` command. The name of the library is the name you defined in the **library(<name>)** command at the top of your **.lib** file.

```
dc_shell-xg-t> write_lib <libname> -o <libname>.db
```

The **-o** switch lets you specify whatever file name you like for the output file, but your life will be easier if you name it with a **.db** extension.

If this finishes without error, you can exit with the `exit` command to the **design compiler** shell. You now (hopefully) have a correctly formatted binary database for your cell library that Synopsys can use in its synthesis procedure.